IoT Toolkit

HTTP Client and JSON Parser

User Guide & Reference Manual

Document: UM15004
Software Version: 2.20
Revision: 0
Date: March 27, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

In den Weiden 11
D-40721 Hilden

Germany

Tel. +49 2103-2878-0
Fax. +49 2103-2878-28
E-mail: support@segger.com
Internet: www.segger.com

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please report it to us and we will try to assist you as soon as possible.

Contact us for further information on topics or functions that are not yet documented.

Print date: March 27, 2018

Software	Revision	Date	By	Description
2.20	0	180327	PC	Added JSON parser.
2.14	0	180214	PC	Initial release.
1.00	0	150725	PC	Internal release.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual describes all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
`User Input`	Text entered at the keyboard by a user in a session transcript.
`Secret Input`	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.
SEGGER home page	A hyperlink to an external document or web site.

Table of contents

Chapter 1

IoT toolkit

1.1 Introduction

This section presents an overview of the IoT Toolkit, its structure, and its capabilities.

1.1.1 What is the IoT Toolkit?

The SEGGER IoT Toolkit is a software library that enables you to interact with REST APIs but can equally interact with a standard HTTP web server. It combines an HTTP client with highly-efficient JSON and SAX parsers to deliver a compelling solution for embedded devices.

The SEGGER IoT Toolkit is hardware independent and integrates with other components from SEGGER such as emSSL and embOS/IP.

1.1.2 Design goals

The IoT Toolkit is designed with the following goals in mind:

- Highly modular such that unused features are never linked into the application.
- Be completely runtime configurable, adding each modular feature as needed.
- Present a simple user-level API that is easy to use without extensive setup.
- Easy to maintain both by SEGGER and anybody with access to the sources.
- Conform to all necessary standards and current best practices.
- Be efficient both in terms of resource usage and execution speed.
- Target embedded processors with limited resources as well as workstations.
- Use the the SEGGER Cryptographic Toolkit, the foundation of all SEGGER security products.

We believe all design goals are achieved by the IoT Toolkit.

1.1.3 Features

The IoT Toolkit is written in ANSI C and can be used on virtually any CPU. Here is a list of the library features:

- ISO/ANSI C source code.
- High performance.
- Small footprint.
- Runs “out-of-the-box”.
- Highly compact implementation runs effortlessly on single-chip MCUs.
- Easy-to-understand and simple-to-use API.
- Simple configuration.
- Secure communication over an open channel.
- Modular architecture links only what you need.
- Royalty-free.

1.2 Package content

The IoT Toolkit is provided in source code and contains everything required. The following table shows the content of the package:

Files	Description
`Config`	Configuration header files.
`CRYPTO`	Shared cryptographic toolkit source code.
`Doc`	IoT Toolkit documentation.
`Sample/Config`	Example IoT Toolkit user configuration.
`SEGGER`	SEGGER software component source code.
`IOT`	HTTP and JSON implementation source code.
`Application`	IoT sample applications.
`Windows/IOT`	Host-based IoT sample applications.

1.3 Sample applications

The IoT Toolkit ships with a number of sample applications that demonstrate how to integrate IoT capability into your application. Each sample application demonstrates a specific capability of the IoT Toolkit and is a small incremental step over previous examples.

The sample applications are:

Application	Description
`IOT_HTTP_GetRequest.c`	Issue a GET request to a web server.
`IOT_HTTP_RedirectRequest.c`	Capture a redirect request.
`IOT_HTTP_SecureGet.c`	Use SSL to issue a secure GET request.
`IOT_JSON_PlainTrace.c`	Display JSON events invoked in sample parse.
`IOT_JSON_PrettyTrace.c`	Show structure of JSON objects for a parse.
`IOT_JSON_MakeTree.c`	Construct a tree representation of a JSON object.
`IOT_JSON_Print.c`	Parse a JSON value read from standard input.

1.3.1 A note on the samples

Each sample that we present in this manual is written in a style that makes it easy to describe and that fits comfortably within the margins of printed paper. Therefore, it may well be that you would rewrite the sample to have a slightly different structure that fits better, but please keep in mind that these examples are written with clarity as the prime objective, and to that end we sacrifice some brevity and efficiency.

1.3.2 Where to find the sample code

All samples are included in the `Application` directory of the IoT Toolkit distribution.

Chapter 2

HTTP client

2.1 Overview

Although the HTTP protocol is mostly associated with retrieving web pages from the Internet, the SEGGER HTTP client is mainly intended to interacting with REST APIs — although it can also retrieve web content just as easily.

REST APIs usually exchange data in a structured format such as XML, JSON, or CSV. Recently, JSON has displaced XML as the preferred REST data format and JSON is the primary format, or indeed only format, supported by many services.

This section presents simple examples that can be used as a base for using the HTTP client to work with REST APIs from Twitter, Amazon, and Xively. The HTTP client also underpins the SEGGER Dropbox Client which interacts with the Dropbox REST APIs.

2.2 A simple GET request

This section will develop the code necessary to issue a request to retrieve the index page of a website. Even though this is a simple request, it requires setting up the request, executing it over the network, and processing the response.

For a complete listing of this application, see *IOT_HTTP_GetRequest complete listing* on page 22.

2.2.1 Application entry

The main application task is responsible for setting up the environment ready for issuing the request. This is simply boilerplate code that has no configuration:

```
void MainTask(void) {
  IOT_HTTP_CONTEXT   HTTP;
  CONNECTION_CONTEXT Connection;
  char               aBuf[128];
  char               aPayload[128];
  unsigned           StatusCode;
  int                Status;
  //
  SEGGER_SYS_Init();  ❶
  SEGGER_SYS_IP_Init();  ❷
```

❶ Initialize system component

The call to `SEGGER_SYS_Init()` sets up the SEGGER system abstraction layer to initialize services to the application. This abstraction layer is implemented for Windows and Linux but is not intended for production code, it is *only for demonstration*.

❷ Initialize IP component

The call to `SEGGER_SYS_IP_Init()` initializes the underlying IP stack to the point that sockets can be opened. For Windows this means starting up Winsock, and for embOS/IP it initializes and sets the IP task running.

2.2.2 Set up the HTTP request

Once the system components are initialized, it's time to set up the HTTP request.

```
IOT_HTTP_Init      (&HTTP, &aBuf[0], sizeof(aBuf));  ❶
IOT_HTTP_SetIO     (&HTTP, &_PlainAPI, &Connection);  ❷
IOT_HTTP_SetVersion(&HTTP, &IOT_HTTP_VERSION_HTTP_1v1);  ❸
IOT_HTTP_AddMethod (&HTTP, "GET");  ❹
IOT_HTTP_AddHost   (&HTTP, "www.segger.com");  ❺
IOT_HTTP_SetPort   (&HTTP, 80);
IOT_HTTP_AddPath   (&HTTP, "/");  ❻
```

❶ Initialize the client context

The call to `IOT_HTTP_Init()` initializes the HTTP context as an empty container for the request. For correct operation, a buffer must be provided that the HTTP client uses for temporary data during the request, for instance when processing request headers and status lines. The exact size of this buffer will be discussed later, and for now we size the buffer to 128 bytes which is enough to process incoming response header lines up to 127 characters in length.

> **Note**

> If header lines exceed the capacity of the buffer, HTTP processing will gracefully fail with an error.

❷ Initialize I/O interface

The call to `IOT_HTTP_SetIO()` sets up the connection and I/O API required for HTTP connections. The interface is described in the section *I/O over the network* on page 19, and we defer implementation details until later.

❸ Set the HTTP request version

The specific HTTP request version may well be defined by the REST API that you use, but in this case we ask for an HTTP/1.1 connection to the server. There are differences in requests and responses between HTTP/1.0 and HTTP/1.1 in connection with headers allowed and payload formats. For the basic connection we are establishing here, the differences between HTTP/1.0 and HTTP/1.1 requests are completely handled by the library.

It is mandatory to set an HTTP version for the request so that the HTTP version is sent to the server and so that the HTTP client can properly deal with responses.

❹ Set the HTTP method

As this is a GET request, the method is set as "`GET`" in the request. Other methods are possible, the standard ones being CONNECT, DELETE, GET, HEAD, OPTIONS, POST, PUT, and TRACE for HTTP and, importantly, REST APIs.

❺ Set the host and port

The host is set to "`www.segger.com`" using `IOT_HTTP_AddHost()` as this happens to be the host that we wish to connect to. The host name is important to many protocols and is passed through the HTTP client to the connection method in the I/O API which we describe later.

As this is plain HTTP request, the connection is made to the host using port 80, set by `IOT_HTTP_SetPort()`. This port number is passed to the connection method and is otherwise unused within the HTTP client.

❻ Set the path

We ask to retrieve the root item of the website by setting the path to "/" using `IOT_HTTP_AddPath()`.

The HTTP client is flexible when developing the path in that it allows clients to build up the paths incrementally rather than having the client specify the path in a single call. For instance, the path "`/product/iot/toolbox.htm`" could be built up by adding the path fragments "`/product/iot/`", "`toolbox`", and "`.htm`" in order as three separate calls to `IOT_HTTP_AddPath()`. The path is built by (virtually) appending each fragment onto existing fragments.

2.2.3 Connect and issue the request

With the HTTP request initialized and populated, the request is ready to be issued.

```
Status = IOT_HTTP_Connect(&HTTP);  ❶
if (Status < 0) {
  SEGGER_SYS_IO_Printf("Cannot negotiate a connection to %s:%d!\n",
                       IOT_HTTP_GetHost(&HTTP),
                       IOT_HTTP_GetPort(&HTTP));
  SEGGER_SYS_OS_Halt(100);
}
//
Status = IOT_HTTP_Exec(&HTTP);  ❷
if (Status < 0) {
  SEGGER_SYS_IO_Printf("Cannot execute %s request!\n",
                       IOT_HTTP_GetMethod(&HTTP));
  SEGGER_SYS_OS_Halt(100);
```



```
}
```

❶ Connect to the host

The call to `IOT_HTTP_Connect()` attempts to establish a connection to the host using the host name and port set in the HTTP request. If the connection cannot be made, the sample terminates with an error. Note that the request's parameters can be queried and in this case `IOT_HTTP_GetHost()` and `IOT_HTTP_GetPort()` return the host name and port set on the request, as you would expect.

❷ Issue the request

The request can be sent immediately after the connection is successfully established. `IOT_HTTP_Exec()` will issue the request, with any appropriate headers, to the host over the established connection. If a processing error occurs, the sample terminates with an error: the particular error code can come from the networking later (e.g. the connection is closed prematurely) or internally from the HTTP request (although this simple request will never raise an internal error).

2.2.4 Process the response

Once the request is successfully issued, the web server replies with a standardized response. The response is comprised of the status line, optional headers, and an optional payload. In this section we deal with the status line and the headers only.

```
Status = IOT_HTTP_ProcessStatusLine(&HTTP, &StatusCode);  ❶
if (Status < 0) {
  SEGGER_SYS_IO_Printf("Cannot process status line!\n");
  SEGGER_SYS_OS_Halt(100);
}
SEGGER_SYS_IO_Printf("Returned status code: %u\n\n\n", StatusCode);
//
Status = IOT_HTTP_ProcessHeaders(&HTTP, NULL);  ❷
if (Status < 0) {
  SEGGER_SYS_IO_Printf("Cannot process headers!\n");
  SEGGER_SYS_OS_Halt(100);
}
```

❶ Process the status line

The call to `IOT_HTTP_ProcessStatusLine()` retrieves the initial status line of the HTTP response. This must immediately follow the HTTP request being issued as the status line is sent by the host as the first part of the response.

`IOT_HTTP_ProcessStatusLine()` returns a status which indicates errors at the network level or errors in the format of the status line. If there is an error processing the status line, for instance the connection is prematurely closed or the status line is not in the standard format, the returned value is negative and the sample terminates with an error.

If `IOT_HTTP_ProcessStatusLine()` succeeds, the status code extracted from the status line is written to `StatusCode` in this instance.

> **Note**
>
> It is essential to understand that the value returned as the result of the function call is entirely independent of the value extracted from the status line: the function result indicates successful extraction of the status code, nothing more, and does not indicate whether the HTTP request "executed successfully". For instance, a HTTP 404 "not found" response does not indicate failure of the HTTP request execution.

The list of standard HTTP response codes is defined in section 6 of RFC 7231: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*.

❷ Process the response headers

Once the status code is successfully extracted, handling of the response continues by processing response headers sent by the server. It is necessary to process the headers as these generally describe things that are required to process the following payload, if any.

The call to `IOT_HTTP_ProcessHeaders()` receives and processes each header in turn, parsing some headers internally. The second parameter to the function is a callback to a user-defined function to process individual headers: in this case we pass in a null pointer which indicates that there is no special per-header processing required.

The function result, as before, indicates problems propagated from the network connection or issues internal to the HTTP component. If, for instance, a response header has invalid syntax or is too long, this would raise an error. In this sample, any error processing the response headers terminates the application.

2.2.5 Receive payload and disconnect

Following the status line and headers comes the payload which, in this case, is HTML.

```
IOT_HTTP_GetBegin(&HTTP);  ❶
do {
  Status = IOT_HTTP_GetPayload(&HTTP, &aPayload[0], sizeof(aPayload));  ❷
  SEGGER_SYS_IO_Printf("%.*s", Status, aPayload);
} while (Status > 0);
IOT_HTTP_GetEnd(&HTTP);  ❸
//
IOT_HTTP_Disconnect(&HTTP);  ❹
```

❶ Prepare to receive payload

The call to `IOT_HTTP_GetBegin()` starts the process of receiving the payload. The response headers describe the transfer encoding of the payload (identity or chunked delivery) according to the request and HTTP version and `IOT_HTT_GetBegin()` makes the preparations necessary.

❷ Receive the payload

The IoT toolkit takes care of dechunking any chunked data such that the client receives the transmitted payload cleanly and doesn't need to be concerned with transfer encodings. This is important for the chunked transfer encoding: the call to `IOT_HTTP_GetPayload()` deals with chunked and identity transfer encodings so the user doesn't need to.

`IOT_HTTP_GetPayload()` receives as much data as possible into the user's provided buffer, but may not receive all of it: the user must be prepared to receive fewer bytes than were requested, which is typical of many communication mechanisms. This is not an error, it is just the way that network and chunked communication works. All this sample does is print the received payload, so we expect to see HTML output when the sample runs.

Payload delivery is complete when `IOT_HTTP_GetPayload()` returns zero. If an error occurs during payload delivery (e.g. network or transfer formatting errors) then `IOT_HTTP_GetPayload()` returns a negative status code.

❸ Finish payload reception

Once the payload is received correctly or with an error, a call to `IOT_HTTP_GetEnd()` synchronizes the HTTP client state.

❹ Disconnect connection

As we have no need to keep the connection to the server open after receiving the payload, a call to `IOT_HTTP_Disconnect()` tears down the connection to the server.

2.2.6 I/O over the network

The preceding sections have shown how the HTTP request is set up and executed, and how the server's response is processed. The mechanics of network connection, I/O, and disconnection have been deferred until now.

The HTTP client requires a set of methods that control how a network connection is set up and torn down, and also a set of methods that implement I/O over that connection after it is established.

The API is embodied in the `IOT_IO_API` type:

```
typedef struct {
  int (*pfConnect)   (void *pConn, const char *sHost, unsigned Port);
  int (*pfDisconnect)(void *pConn);
  int (*pfSend)      (void *pConn, const void *pData, unsigned DataLen);
  int (*pfRecv)      (void *pConn,       void *pData, unsigned DataLen);
} IOT_IO_API;
```

In this example the methods are implemented by functions local to the application:

```
static const IOT_IO_API _PlainAPI = {
  _Connect,
  _Disconnect,
  _Send,
  _Recv
};
```

Here is the initialization of HTTP I/O setup again:

```
IOT_HTTP_SetIO(&HTTP, &_PlainAPI, &Connection);
```

Notice that the third argument passed into `IOT_HTTP_SetIO()` is the address of a connection context structure: this address is passed to all `IOT_IO_API` methods as their first parameter, `pConn`, and holds a single member, the socket ID:

```
typedef struct {
  unsigned Socket;
} CONNECTION_CONTEXT;
```

2.2.6.1 Connection

Connection requires that a plain socket is opened to the server.

```
static int _Connect(void *pVoid, const char *sHost, unsigned Port) {  ❶
  CONNECTION_CONTEXT * pConn;
  int                  Status;
  //
  pConn = pVoid;  ❷
  //
  Status = SEGGER_SYS_IP_Open(sHost, Port);  ❸
  if (Status >= 0) {
    pConn->Socket = Status;
  }
  return Status;
}
```

❶ Receive connection parameters

The implementation of the socket setup requires that a plain TCP socket is opened to the given host and port specified by `sHost` (a domain name that must be resolved) and `Port` which is in host byte order.

❷ Recover connection context

The first parameter, `pVoid`, is actually a pointer to the connection context (of type `CONNECTION_CONTEXT`) which holds the socket ID. This connection context was set up using `IOT_HTTP_SetIO()`.

❸ Open socket connection

The incoming parameters are used to set up a plain TCP socket to the provided host and port. Errors from the socket layer are propagated to the caller. If a socket to the server is opened without error, the socket ID is stored to the context passed into the method.

2.2.6.2 Send data

Sending data to an established connection is handled by recovering the socket ID from the context and sending the data to the socket.

```
static int _Send(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Send(pConn->Socket, pData, DataLen, 0);
}
```

2.2.6.3 Receive data

Receiving data from an established connection is handled by recovering the socket ID from the context and receiving the data from the socket.

```
static int _Recv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Recv(pConn->Socket, pData, DataLen, 0);
}
```

2.2.6.4 Disconnection

Disconnecting an established connection is handled by recovering the socket ID from the context and closing the socket. Disconnection always succeeds and returns a success code.

```
static int _Disconnect(void *pVoid) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}
```

2.2.7 Running the sample

With the sample now complete, running it produces output similar to the following:

```
C:> IOT_HTTP_GetRequest.exe

Returned status code: 301

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
```



```
<p>The document has moved <a href="https://www.segger.com/">here</a>.</p>
<hr>
<address>Apache/2.4.20 Server at www.segger.com Port 80</address>
</body></html>

C:> _
```

This has correctly executed the HTTP request and received a response. What we see is not an HTTP 200 “OK” status code but a 301 “moved permanently” redirection. The next sample shows how to capture this redirection.

2.2.8 IOT_HTTP_GetRequest complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_HTTP_GetRequest.c
Purpose     : Issue a GET request over a plain socket.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"
#include "SEGGER_SYS.h"

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

typedef struct {
  unsigned Socket;
} CONNECTION_CONTEXT;

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static int _Connect   (void *pVoid, const char *sHost, unsigned Port);
static int _Disconnect(void *pVoid);
static int _Send      (void *pVoid, const void *pData, unsigned DataLen);
static int _Recv      (void *pVoid,       void *pData, unsigned DataLen);

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_IO_API _PlainAPI = {
  _Connect,
  _Disconnect,
  _Send,
  _Recv
};

/*********************************************************************
*
*       Static code
*
**********************************************************************
```



```
*/

/*********************************************************************
*
*       _Connect()
*
*  Function description
*    Connect to host.
*
*  Parameters
*    pVoid - Pointer to connection context.
*    sHost - Name of server we wish to connect to.
*    Port  - Port number in host byte order.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Connect(void *pVoid, const char *sHost, unsigned Port) {
  CONNECTION_CONTEXT * pConn;
  int                  Status;
  //
  pConn = pVoid;
  //
  Status = SEGGER_SYS_IP_Open(sHost, Port);
  if (Status >= 0) {
    pConn->Socket = Status;
  }
  return Status;
}

/*********************************************************************
*
*       _Disconnect()
*
*  Function description
*    Disconnect from host.
*
*  Parameters
*    pVoid - Pointer to connection context.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Disconnect(void *pVoid) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}

/*********************************************************************
*
*       _Send()
*
*  Function description
*    Send data to host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to octet string to send.
*    DataLen - Octet length of the octet string to send.
*
*  Return value
*    >= 0 - Success.
```



```
*    <  0 - Processing error.
*/
static int _Send(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Send(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       _Recv()
*
*  Function description
*    Receive data from host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to object that receives the data.
*    DataLen - Octet length of receiving object.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Recv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT *pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Recv(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_HTTP_CONTEXT   HTTP;
  IOT_HTTP_PARA      aPara[20];
  CONNECTION_CONTEXT Connection;
  char               aBuf[128];
  char               aPayload[128];
  unsigned           StatusCode;
  int                Status;
  //
  SEGGER_SYS_Init();
  SEGGER_SYS_IP_Init();
  //
  IOT_HTTP_Init
  (&HTTP, &aBuf[0], sizeof(aBuf), aPara, SEGGER_COUNTOF(aPara));
  IOT_HTTP_SetIO     (&HTTP, &_PlainAPI, &Connection);
  IOT_HTTP_SetVersion(&HTTP, &IOT_HTTP_VERSION_HTTP_1v1);
  IOT_HTTP_AddMethod (&HTTP, "GET");
  IOT_HTTP_AddHost   (&HTTP, "www.segger.com");
  IOT_HTTP_SetPort   (&HTTP, 80);
  IOT_HTTP_AddPath   (&HTTP, "/");
  //
```



```
  Status = IOT_HTTP_Connect(&HTTP);
  if (Status < 0) {
    SEGGER_SYS_IO_Printf("Cannot negotiate a connection to %s:%d!\n",
                         IOT_HTTP_GetHost(&HTTP),
                         IOT_HTTP_GetPort(&HTTP));
    SEGGER_SYS_OS_Halt(100);
  }
  //
  Status = IOT_HTTP_Exec(&HTTP);
  if (Status < 0) {
    SEGGER_SYS_IO_Printf("Cannot execute GET request!\n",
                         IOT_HTTP_GetMethod(&HTTP));
    SEGGER_SYS_OS_Halt(100);
  }
  //
  Status = IOT_HTTP_ProcessStatusLine(&HTTP, &StatusCode);
  if (Status < 0) {
    SEGGER_SYS_IO_Printf("Cannot process status line!\n");
    SEGGER_SYS_OS_Halt(100);
  }
  SEGGER_SYS_IO_Printf("Returned status code: %u\n\n\n", StatusCode);
  //
  Status = IOT_HTTP_ProcessHeaders(&HTTP, NULL);
  if (Status < 0) {
    SEGGER_SYS_IO_Printf("Cannot process headers!\n");
    SEGGER_SYS_OS_Halt(100);
  }
  //
  IOT_HTTP_GetBegin(&HTTP);
  do {
    Status = IOT_HTTP_GetPayload(&HTTP, &aPayload[0], sizeof(aPayload));
    SEGGER_SYS_IO_Printf("%.*s", Status, aPayload);
  } while (Status > 0);
  IOT_HTTP_GetEnd(&HTTP);
  //
  IOT_HTTP_Disconnect(&HTTP);
  //
  SEGGER_SYS_IP_Exit();
  SEGGER_SYS_Exit();
  SEGGER_SYS_OS_Halt(Status);
}

/************************** End of file ***************************/
```


2.3 Capturing a redirection

This section extends the previous example to intercept a redirect request.

For a complete listing of this application, see *IOT_HTTP_RedirectRequest complete listing* on page 31.

2.3.1 Processing headers

Recapping the previous example's HTML output:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.segger.com/">here</a>.</p>
<hr>
<address>Apache/2.4.20 Server at www.segger.com Port 80</address>
</body></html>
```

In the HTML body there is an indication of where the redirect should end up, `https://www.segger.com/`. From RFC 7231 section 7.1.2, *Location*, the definitive redirection URL is contained in the "`Location`" header of the response. To capture that particular header requires a callback:

```
static void _HeaderCallback(      IOT_HTTP_CONTEXT * pContext,
                            const char             * sKey,
                            const char             * sValue) {
  SEGGER_SYS_IO_Printf("%s: %s\n", sKey, sValue);
}
```

This callback prints the headers encountered in the response. Rather than specifying a null function when calling `IOT_HTTP_ProcessHeaders()`, plug in the callback:

```
Status = IOT_HTTP_ProcessHeaders(&HTTP, _HeaderCallback);
```

Now when the application runs the headers are printed:

```
Date: Thu, 09 Mar 2017 00:20:08 GMT
Server: Apache/2.4.20
Location: https://www.segger.com/
Content-Length: 302
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

The only header we are interested in is `Location`, so we must preserve its content. The key and value string parameters are transitory and are only valid during execution of the callback, so it is not possible to save the value of the pointer for later, it is essential that the string is copied to an object with a longer lifetime, for instance a file-scope variable:

```
static void _HeaderCallback(      IOT_HTTP_CONTEXT * pContext,
                            const char             * sKey,
                            const char             * sValue) {
  if (IOT_STRCASECMP(sKey, "Location") == 0) {
    if (strlen(sValue)+1 < sizeof(_aRedirectURL)) {
      strcpy(_aRedirectURL, sValue);
    }
  }
}
```


2.3.2 Parsing the redirect URL

Now that the callback captures the redirect URL, the main application must identify the redirection and act upon it. But there is an issue, we have a URL which isn't directly usable by the library.

Every HTTP URL conforms to the syntax of a generic URI without user name and password, and this is:

```
scheme:[//host[:port]][/]path[?query][#fragment]
```

A URL in this form must be parsed into something that is useful, and that is an HTTP request context in this case. The following code parses a restricted subset of a full HTTP URL but still manages to be incredibly useful.

```
static void _ParseURL(IOT_HTTP_CONTEXT *pHTTP, char *sText) {
  char *pPos;
  char *sHost;
  char *sPath;
  //
  if (IOT_STRNCMP(sText, "https:", 6) == 0) {  ❶
    IOT_HTTP_AddScheme(pHTTP, "https");
    IOT_HTTP_SetPort  (pHTTP, 443);
    sText += 6;
  } else if (IOT_STRNCMP(sText, "http:", 5) == 0) {
    IOT_HTTP_AddScheme(pHTTP, "http");
    IOT_HTTP_SetPort  (pHTTP, 80);
    sText += 5;
  } else {
    IOT_HTTP_AddScheme(pHTTP, "http");
    IOT_HTTP_SetPort  (pHTTP, 80);
  }
  //
  if (IOT_STRNCMP(sText, "//", 2) == 0) {  ❷
    sText += 2;
  }
  sHost = sText;  ❸
  //
  pPos = IOT_STRCHR(sHost, '/');  ❹
  if (pPos) {
    *pPos = '\0';
    sPath = pPos + 1;
  } else {
    sPath = "";
  }
  //
  pPos = IOT_STRCHR(sHost, ':');  ❺
  if (pPos) {
    *pPos = '\0';
    IOT_HTTP_SetPort(pHTTP, (unsigned)strtoul(pPos+1, NULL, 0));
  }
  //
  IOT_HTTP_AddHost(pHTTP, sHost);  ❻
  IOT_HTTP_AddPath(pHTTP, "/");
  IOT_HTTP_AddPath(pHTTP, sPath);
}
```

❶ Determine the scheme

The scheme always appears first and this parser only recognizes the schemes `http` and `https`. If neither of these schemes occur at the front of the URL, it's reasonable to default the scheme to `http`.

Once the scheme is known, or defaulted, it's added to the HTTP request along with the scheme's default port.

❷ Be forgiving

Although the syntax of the URL would require the double-slash between the scheme and the host parts, some browsers will accept the scheme followed immediately by the host, for example "`https:www.segger.com`". It doesn't seem unreasonable, in this case, to make the "`//`" optional.

❸ Remember where the host part starts

Once the scheme and optional `//` is consumed, the host part must start. The function remembers where the host part starts so it can be isolated during the remainder of the parse.

One thing that is worth mentioning now is that the function receives a `char *` pointer rather than a `const char *` pointer because it will modify the incoming string to isolate parts of it into individual zero-terminated fragments which are passed into the HTTP request context as needed.

❹ Isolate the host

Once the start of the host is identified, the code finds its extent. The host part extends up to the first slash or, if a slash is not found, to the end of the string.

If a slash is found, the host part is isolated by writing a zero character into the incoming string which separates the host from the path. But writing this zero knocks out the leading slash of the path, so we must be careful to remember to add it back when constructing the path in the HTTP request.

❺ Parse the port

The previous step did not entirely isolate the host because everything between the double slash and the first slash (or end of string) was grabbed, including an optional port specification. If we find a colin in the host part, it happens to be a port specification that overrides the default port that was set by the scheme.

❻ Add the host and path

Now that the host and path have been found and isolated, they can be added to the HTTP request. The path is added as two fragments, a leading slash, because we overwrote it when isolating the host, and the remainder.

The path could be further parsed to isolate the query and fragment components and add them to the URL request, but this type of parsing happens to be specific to the scheme and cannot, in general, be written in a generic way to cover all schemes. Therefore, any query or fragment becomes part of the path component in the HTTP request and is "passed through" when the request is executed.

With all this done, the URL parser is ready for testing.

2.3.3 Testing the URL parser

Before putting the code to use it's always interesting to try it with a small test harness.

```
static void _test(const char *sData) {
  IOT_HTTP_CONTEXT HTTP;
  IOT_HTTP_PARA    aPara[20];
  char             aCopy[128];
  char             aURL[128];
  //
  strcpy(aCopy, sData);
  IOT_HTTP_Init(&HTTP, NULL, 0, &aPara[0], SEGGER_COUNTOF(aPara));
  _ParseURL(&HTTP, aCopy);
  IOT_HTTP_QueryURL(&HTTP, &aURL[0], sizeof(aURL));
  SEGGER_SYS_IO_Printf("%-44s - %s\n", sData, aURL);
}
```


The test harness presents valid URLs which the parser should successfully deal with:

```
_test("http://www.segger.com");
_test("https://www.segger.com");
_test("https://www.segger.com:4433");
_test("http://www.segger.com/index.htm");
_test("https://www.segger.com:4433/index.htm");
_test("http://www.segger.com:4433/product/emssl.htm");
_test("www.segger.com");
_test("www.segger.com:4433");
_test("www.segger.com/index.htm");
_test("www.segger.com:4433/index.htm");
_test("www.segger.com:4433/product/emssl.htm");
_test("https:www.segger.com");
```

Running the parser in the test harness gives us confidence that the code is doing what's expected:

```
http://www.segger.com                          - http://www.segger.com:80/
https://www.segger.com                         - https://www.segger.com:443/
https://www.segger.com:4433                    - https://www.segger.com:4433/
http://www.segger.com/index.htm                - http://www.segger.com:80/index.htm
https://www.segger.com:4433/index.htm          - https://www.segger.com:4433/index.htm
http://www.segger.com:4433/product/emssl.htm - http://www.segger.com:4433/product/emssl.htm
www.segger.com                                 - http://www.segger.com:80/
www.segger.com:4433                            - http://www.segger.com:4433/
www.segger.com/index.htm                       - http://www.segger.com:80/index.htm
www.segger.com:4433/index.htm                  - http://www.segger.com:4433/index.htm
www.segger.com:4433/product/emssl.htm          - http://www.segger.com:4433/product/emssl.htm
https:www.segger.com                           - https://www.segger.com:443/
```

2.3.4 Processing the redirect

Now that we know the redirect and have code that will parse the URL, processing the redirect is no more than a loop to follow the redirect, or a redirected redirect:

```
for (;;) {
  Status = IOT_HTTP_Connect(&HTTP);
  if (Status < 0) {
    SEGGER_SYS_IO_Printf("Cannot negotiate a connection to %s:%d!\n",
                         IOT_HTTP_GetHost(&HTTP),
                         IOT_HTTP_GetPort(&HTTP));
    SEGGER_SYS_OS_Halt(100);
  }
  //
  ...
  //
  if (StatusCode == 301 || StatusCode == 302 ||  ❶
      StatusCode == 303 || StatusCode == 307 ) {
    //
    SEGGER_SYS_IO_Printf("Redirect to %s\n\n", _aRedirectURL);
    //
    IOT_HTTP_Disconnect(&HTTP);  ❷
    IOT_HTTP_Reset(&HTTP);  ❸
    IOT_HTTP_AddMethod(&HTTP, "GET");
    _ParseURL(&HTTP, _aRedirectURL);  ❹
    //
    if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "http") != 0) {  ❺
      SEGGER_SYS_IO_Printf("Cannot handle scheme %s!\n",
                           IOT_HTTP_GetScheme(&HTTP));
      SEGGER_SYS_OS_Halt(100);
    }
  } else {
    break;
  }
```



```
}
```

❶ Detect redirect

Only specific redirects are processed; refer to RFC 7231 section 6.4 for details of these.

❷ Disconnect

We abruptly disconnect the session when we find a redirect status code and do not progress to read the payload as it's irrelevant in this instance.

❸ Reset the HTTP request

Resetting the HTTP request clears down the HTTP request but keeps the network I/O API and context and the working buffer (which is also reset). This prepares the HTTP request context to receive a new request.

Resetting the HTTP context also clears any method that has been set, so the method is added before parsing the redirect URL.

❹ Parse the redirect URL

The redirect URL is parsed into the request which makes it ready for execution.

❺ Execute the redirect URL

As this sample only supports the HTTP scheme, it's not possible to redirect to an `https` URL. In fact, anything other than the `http` scheme is rejected.

Once the redirect is parsed and accepted, the code loops to connect and retrieve the content from the redirected URL.

2.3.5 Running the sample

With the sample now complete, running it produces output similar to the following:

```
C:> IOT_HTTP_RedirectRequest.exe

Returned status code: 301

Redirect to https://www.segger.com/

Cannot handle scheme https!

C:> _
```

As the redirect is to an `https` scheme we cannot progress further at this point. The next sample shows how to add support for the `https` scheme using secure TLS connections.

2.3.6 IOT_HTTP_RedirectRequest complete listing

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                      The Embedded Experts                          *
*                         www.segger.com                             *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_HTTP_RedirectRequest.c
Purpose     : Handle HTTP redirect requests.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"
#include "SEGGER_SYS.h"

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

typedef struct {
  unsigned Socket;
} CONNECTION_CONTEXT;

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static int _Connect   (void *pVoid, const char *sHost, unsigned Port);
static int _Disconnect(void *pVoid);
static int _Send      (void *pVoid, const void *pData, unsigned DataLen);
static int _Recv      (void *pVoid,       void *pData, unsigned DataLen);

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_IO_API _PlainAPI = {
  _Connect,
  _Disconnect,
  _Send,
  _Recv
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
```



```
*/

static char _aRedirectURL[128];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _Connect()
*
*  Function description
*    Connect to host.
*
*  Parameters
*    pVoid - Pointer to connection context.
*    sHost - Name of server we wish to connect to.
*    Port  - Port number in host byte order.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Connect(void *pVoid, const char *sHost, unsigned Port) {
  CONNECTION_CONTEXT * pConn;
  int                  Status;
  //
  pConn = pVoid;
  //
  Status = SEGGER_SYS_IP_Open(sHost, Port);
  if (Status >= 0) {
    pConn->Socket = Status;
  }
  return Status;
}

/*********************************************************************
*
*       _Disconnect()
*
*  Function description
*    Disconnect from host.
*
*  Parameters
*    pVoid - Pointer to connection context.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Disconnect(void *pVoid) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}

/*********************************************************************
*
*       _Send()
*
*  Function description
```



```
*    Send data to host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to octet string to send.
*    DataLen - Octet length of the octet string to send.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Send(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Send(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       _Recv()
*
*  Function description
*    Receive data from host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to object that receives the data.
*    DataLen - Octet length of receiving object.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _Recv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Recv(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       _HeaderCallback()
*
*  Function description
*    Process response header.
*
*  Parameters
*    pContext - Pointer to HTTP request context.
*    sKey     - Pointer to key string.
*    sValue   - Pointer to value string.
*/
static void _HeaderCallback(      IOT_HTTP_CONTEXT * pContext,
                            const char             * sKey,
                            const char             * sValue) {
  if (IOT_STRCASECMP(sKey, "Location") == 0) {
    if (strlen(sValue)+1 < sizeof(_aRedirectURL)) {
      strcpy(_aRedirectURL, sValue);
    }
  }
}

/*********************************************************************
*
*       _ParseURL()
*
*  Function description
```



```
*    Parse a URL for the HTTP(S) scheme.
*
*  Parameters
*    pContext - Pointer to HTTP request context.
*    sText    - Pointer to zero-terminated URL.
*/
static void _ParseURL(IOT_HTTP_CONTEXT *pContext, char *sText) {
  char *pPos;
  char *sHost;
  char *sPath;
  //
  if (IOT_STRNCMP(sText, "https:", 6) == 0) {
    IOT_HTTP_AddScheme(pContext, "https");
    IOT_HTTP_SetPort  (pContext, 443);
    sText += 6;
  } else if (IOT_STRNCMP(sText, "http:", 5) == 0) {
    IOT_HTTP_AddScheme(pContext, "http");
    IOT_HTTP_SetPort  (pContext, 80);
    sText += 5;
  } else {
    IOT_HTTP_AddScheme(pContext, "http");
    IOT_HTTP_SetPort  (pContext, 80);
  }
  //
  if (IOT_STRNCMP(sText, "//", 2) == 0) {
    sText += 2;
  }
  sHost = sText;
  //
  pPos = IOT_STRCHR(sHost, '/');
  if (pPos) {
    *pPos = '\0';
    sPath = pPos + 1;
  } else {
    sPath = "";
  }
  //
  pPos = IOT_STRCHR(sHost, ':');
  if (pPos) {
    *pPos = '\0';
    IOT_HTTP_SetPort(pContext, (unsigned)strtoul(pPos+1, NULL, 0));
  }
  //
  IOT_HTTP_AddHost(pContext, sHost);
  IOT_HTTP_AddPath(pContext, "/");
  IOT_HTTP_AddPath(pContext, sPath);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_HTTP_CONTEXT   HTTP;
  IOT_HTTP_PARA      aPara[20];
  CONNECTION_CONTEXT Connection;
  char               aBuf[128];
```



```
  char               aPayload[128];
  unsigned           StatusCode;
  int                Status;
  //
  SEGGER_SYS_Init();
  SEGGER_SYS_IP_Init();
  //
  IOT_HTTP_Init
  (&HTTP, &aBuf[0], sizeof(aBuf), aPara, SEGGER_COUNTOF(aPara));
  IOT_HTTP_SetIO     (&HTTP, &_PlainAPI, &Connection);
  IOT_HTTP_SetVersion(&HTTP, &IOT_HTTP_VERSION_HTTP_1v1);
  IOT_HTTP_AddMethod (&HTTP, "GET");
  IOT_HTTP_AddHost   (&HTTP, "www.segger.com");
  IOT_HTTP_SetPort   (&HTTP, 80);
  IOT_HTTP_AddPath   (&HTTP, "/");
  //
  for (;;) {
    Status = IOT_HTTP_Connect(&HTTP);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot negotiate a connection to %s:%d!\n",
                           IOT_HTTP_GetHost(&HTTP),
                           IOT_HTTP_GetPort(&HTTP));
      SEGGER_SYS_OS_Halt(100);
    }
    //
    Status = IOT_HTTP_Exec(&HTTP);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot execute GET request!\n",
                           IOT_HTTP_GetMethod(&HTTP));
      SEGGER_SYS_OS_Halt(100);
    }
    //
    Status = IOT_HTTP_ProcessStatusLine(&HTTP, &StatusCode);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot process status line!\n");
      SEGGER_SYS_OS_Halt(100);
    }
    SEGGER_SYS_IO_Printf("Returned status code: %u\n\n", StatusCode);
    //
    Status = IOT_HTTP_ProcessHeaders(&HTTP, _HeaderCallback);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot process headers!\n");
      SEGGER_SYS_OS_Halt(100);
    }
    //
    if (StatusCode == 301 || StatusCode == 302 ||
        StatusCode == 303 || StatusCode == 307 ) {
      //
      SEGGER_SYS_IO_Printf("Redirect to %s\n\n", _aRedirectURL);
      //
      IOT_HTTP_Disconnect(&HTTP);
      IOT_HTTP_Reset(&HTTP);
      IOT_HTTP_AddMethod(&HTTP, "GET");
      _ParseURL(&HTTP, _aRedirectURL);
      //
      if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "http") != 0) {
        SEGGER_SYS_IO_Printf("Cannot handle scheme %s!\n",
                             IOT_HTTP_GetScheme(&HTTP));
        SEGGER_SYS_OS_Halt(100);
      }
    } else {
      break;
    }
  }
  //
  IOT_HTTP_GetBegin(&HTTP);
  do {
    Status = IOT_HTTP_GetPayload(&HTTP, &aPayload[0], sizeof(aPayload));
```



```
    SEGGER_SYS_IO_Printf("%.*s", Status, aPayload);
  } while (Status > 0);
  IOT_HTTP_GetEnd(&HTTP);
  //
  IOT_HTTP_Disconnect(&HTTP);
  //
  SEGGER_SYS_IP_Exit();
  SEGGER_SYS_Exit();
  SEGGER_SYS_OS_Halt(Status);
}

/*************************** End of file ****************************/
```


2.4 Secure communication with SSL

This section extends the previous example to support secure connections for the `https` scheme.

For a complete listing of this application, see *`IOT_HTTP_SecureGet` complete listing* on page 41.

2.4.1 Extending for SSL is easy!

Adding SSL (or to use it's proper name, TLS) is indeed straightforward. This sample will use SEGGER's emSSL because it's designed to be small, efficient, and have a clean, elegant API that simply works. However, porting this example to other SSL stacks should not be challenging, it really depends upon the complexity of the particular SSL stack, nothing more.

Previous examples have only ever needed an integer to hold the socket ID of the connection, but SSL sockets are more complex and hold more state so the connection context must be extended for that:

```
typedef struct {
  unsigned    Socket;
  SSL_SESSION Session;
} CONNECTION_CONTEXT;
```

The connection context uses the `Socket` member to hold the underlying socket ID associated with the connection and a new `Session` member to hold the SSL session context for secure sockets. The type `SSL_SESSION` is defined by the emSSL product and contains everything required to set up, manage, and tear down a secure connection over TLS.

2.4.2 Recognize the HTTPS scheme

The code that deals with the scheme is extended to recognize `https`:

```
if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "http") == 0) {
  IOT_HTTP_SetIO(&HTTP, &_PlainIO, &Connection);
} else if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "https") == 0) {
  IOT_HTTP_SetIO(&HTTP, &_SecureIO, &Connection);
} else {
  SEGGER_SYS_IO_Printf("Cannot handle scheme %s!\n",
                       IOT_HTTP_GetScheme(&HTTP));
  SEGGER_SYS_OS_Halt(100);
}
```

There are now two distinct APIs for network I/O: the existing `_PlainIO` for I/O over plain sockets and the new `_SecureIO` for I/O over secure sockets. This requires addition of a new set of I/O methods along with their dispatcher.

2.4.3 The secure I/O implementation

This is very much like the plain socket I/O implementation. First, the secure implementation of the I/O API, embodied by the `IOT_IO_API` type, is defined:

```
static const IOT_IO_API _SecureIO = {
  _SecureConnect,
  _SecureDisconnect,
  _SecureSend,
  _SecureRecv
};
```

Now the implementation. This follows the plain socket I/O API and the overall structure should be very familiar. The intention here is to show only how the connection is managed

by emSSL, not how to configure the emSSL product for correct operation — for that, please refer to the emSSL manual which contains details of how to select what's necessary.

2.4.3.1 Connection

Connection requires that a plain socket is opened to the server and that socket be upgraded to secure using an SSL handshake.

```
static int _SecureConnect(void *pVoid, const char *sHost, unsigned Port) {  ❶
  CONNECTION_CONTEXT * pConn;
  int                  Status;
  //
  pConn = pVoid;  ❷
  Status = SEGGER_SYS_IP_Open(sHost, Port);  ❸
  if (Status >= 0) {
    pConn->Socket = Status;
    SSL_SESSION_Prepare(&pConn->Session, pConn->Socket, &_IP_Transport);  ❹
    Status = SSL_SESSION_Connect(&pConn->Session, sHost);  ❺
    if (Status < 0) {
      SEGGER_SYS_IP_Close(pConn->Socket);
    }
  }
  return Status;
}
```

❶ Receive connection parameters

As with the plain socket implementation, the secure socket receives the same set of parameters: a host and port specified by `sHost` and `Port`.

❷ Recover connection context

The first parameter, `pVoid`, is still a pointer to the connection context (of type `CONNECTION_CONTEXT`) but this extended context holds both the socket ID and SSL session.

❸ Open plain socket connection

The incoming parameters are used to set up a plain TCP socket to the provided host and port. Errors from the socket layer are propagated to the caller. If a socket to the server is opened without error, the socket is ready to be upgraded to secure.

❹ Prepare the SSL session

After storing the ID in the context, the SSL session is initialized by calling `SSL_SESSION_Prepare()` with the session context, the plain socket to communicate over, and an API for doing so. The SSL transport API is described in the emSSL manual, but a synopsis is that it sends and receives data using the regular BSD socket API.

❺ Upgrade to secure

With the session prepared, a call to `SSL_SESSION_Connect()` attempts an SSL handshake to negotiate a secure connection. If this fails (there can be many reasons for this), the socket is closed.

The success or failure of the connection is propagated to the caller for it to deal with.

2.4.3.2 Send data

Sending data to an established SSL connection is no more complex than sending over a plain socket:

```
static int _SecureSend(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
```



```
  pConn = pVoid;
  return SSL_SESSION_Send(&pConn->Session, pData, DataLen);
}
```

2.4.3.3 Receive data

Receiving data from an established SSL connection is no more complex than receiving over a plain socket:

```
static int _SecureRecv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SSL_SESSION_Receive(&pConn->Session, pData, DataLen);
}
```

2.4.3.4 Disconnection

Disconnecting an established SSL connection is simple: the SSL session is disconnected and then the socket is closed.

```
static int _SecureDisconnect(void *pVoid) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  SSL_SESSION_Disconnect(&pConn->Session);
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}
```

2.4.4 Odds and ends

The final piece of housekeeping to make secure connections work is to ensure that the SSL component is initialized and finalized using `SSL_Init()` and `SSL_Exit()` as part of the main application:

```
void MainTask(void) {
  //
  SEGGER_SYS_Init();
  SEGGER_SYS_IP_Init();
  SSL_Init();
  //
  ...
  //
  SSL_Exit();
  SEGGER_SYS_IP_Exit();
  SEGGER_SYS_Exit();
  SEGGER_SYS_OS_Halt(Status);
}
```

2.4.5 Running the sample

With the sample now complete, running it produces output similar to the following:

```
C:> IOT_HTTP_SecureGet.exe

Returned status code: 301

Redirect to https://www.segger.com/

Returned status code: 200
```



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org
/TR/html4/loose.dtd"><html lang="en-US"><head><meta http-equiv="Content-Type" con
tent="text/html;"><!--<meta name="language" content="english">--><!--<meta http-e
...
<!--<script type="text/javascript" src="./js/footer.js" language="javascript"></s
cript>--><script type="text/javascript" src="./js/prettify.js"></script><script t
ype="text/javascript">prettyPrint();</script></body></html>

C:> _
```

This concludes the sample application which demonstrates how easy it is to enable the HTTP client to use SSL-secured connections.

2.4.6 IOT_HTTP_SecureGet complete listing

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_HTTP_SecureGet.c
Purpose     : REST API access using plain sockets and TLS.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"
#include "SSL.h"
#include "SEGGER_SYS.h"

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static int _PlainConnect    (void *pVoid, const char *sHost, unsigned Port);
static int _PlainDisconnect (void *pVoid);
static int _PlainSend       (void *pVoid, const void *pData, unsigned DataLen);
static int _PlainRecv       (void *pVoid,       void *pData, unsigned DataLen);
//
static int _SecureConnect   (void *pVoid, const char *sHost, unsigned Port);
static int _SecureDisconnect(void *pVoid);
static int _SecureSend      (void *pVoid, const void *pData, unsigned DataLen);
static int _SecureRecv      (void *pVoid,       void *pData, unsigned DataLen);

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

typedef struct {
  unsigned    Socket;
  SSL_SESSION Session;
} CONNECTION_CONTEXT;

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_IO_API _PlainIO = {
  _PlainConnect,
  _PlainDisconnect,
  _PlainSend,
  _PlainRecv
```



```
};

static const IOT_IO_API _SecureIO = {
  _SecureConnect,
  _SecureDisconnect,
  _SecureSend,
  _SecureRecv
};

static const SSL_TRANSPORT_API _IP_Transport = {
  SEGGER_SYS_IP_Send,
  SEGGER_SYS_IP_Recv,
  NULL
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static char _aRedirectURL[128];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _PlainConnect()
*
*  Function description
*    Connect to host using secure sockets.
*
*  Parameters
*    pVoid - Pointer to connection context.
*    sHost - Name of server we wish to connect to.
*    Port  - Port number in host byte order.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _PlainConnect(void *pVoid, const char *sHost, unsigned Port) {
  int * pSocket;
  int   Status;
  //
  Status = SEGGER_SYS_IP_Open(sHost, Port);
  if (Status >= 0) {
    pSocket  = pVoid;
    *pSocket = Status;
  }
  return Status;
}

/*********************************************************************
*
*       _PlainDisconnect()
*
*  Function description
*    Disconnect from host.
*
*  Parameters
*    pVoid - Pointer to connection context.
```



```
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _PlainDisconnect(void *pVoid) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}

/*********************************************************************
*
*       _PlainSend()
*
*  Function description
*    Send data to host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to octet string to send.
*    DataLen - Octet length of the octet string to send.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _PlainSend(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Send(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       _PlainRecv()
*
*  Function description
*    Receive data from host.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to object that receives the data.
*    DataLen - Octet length of receiving object.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _PlainRecv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SEGGER_SYS_IP_Recv(pConn->Socket, pData, DataLen, 0);
}

/*********************************************************************
*
*       _SecureConnect()
*
*  Function description
*    Connect to host, secure.
*
*  Parameters
```



```
*    pVoid - Pointer to connection context.
*    sHost - Name of server we wish to connect to.
*    Port  - Port number in host byte order.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _SecureConnect(void *pVoid, const char *sHost, unsigned Port) {
  CONNECTION_CONTEXT * pConn;
  int                  Status;
  //
  pConn = pVoid;
  Status = SEGGER_SYS_IP_Open(sHost, Port);
  if (Status >= 0) {
    pConn->Socket = Status;
    SSL_SESSION_Prepare(&pConn->Session, pConn->Socket, &_IP_Transport);
    Status = SSL_SESSION_Connect(&pConn->Session, sHost);
    if (Status < 0) {
      SEGGER_SYS_IP_Close(pConn->Socket);
    }
  }
  return Status;
}

/*********************************************************************
*
*       _SecureDisconnect()
*
*  Function description
*    Disconnect from host, secure.
*
*  Parameters
*    pVoid - Pointer to connection context.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _SecureDisconnect(void *pVoid) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  SSL_SESSION_Disconnect(&pConn->Session);
  SEGGER_SYS_IP_Close(pConn->Socket);
  //
  return 0;
}

/*********************************************************************
*
*       _SecureSend()
*
*  Function description
*    Send data to host, secure.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to octet string to send over SSL.
*    DataLen - Octet length of the octet string to send.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _SecureSend(void *pVoid, const void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
```



```
  pConn = pVoid;
  return SSL_SESSION_Send(&pConn->Session, pData, DataLen);
}

/*********************************************************************
*
*       _SecureRecv()
*
*  Function description
*    Receive data from host, secure.
*
*  Parameters
*    pVoid   - Pointer to connection context.
*    pData   - Pointer to object that receives the data over SSL.
*    DataLen - Octet length of receiving object.
*
*  Return value
*    >= 0 - Success.
*    <  0 - Processing error.
*/
static int _SecureRecv(void *pVoid, void *pData, unsigned DataLen) {
  CONNECTION_CONTEXT * pConn;
  //
  pConn = pVoid;
  return SSL_SESSION_Receive(&pConn->Session, pData, DataLen);
}

/*********************************************************************
*
*       _HeaderCallback()
*
*  Function description
*    Process response header.
*
*  Parameters
*    pContext - Pointer to HTTP request context.
*    sKey     - Pointer to key string.
*    sValue   - Pointer to value string.
*/
static void _HeaderCallback(      IOT_HTTP_CONTEXT * pContext,
                            const char             * sKey,
                            const char             * sValue) {
  if (IOT_STRCASECMP(sKey, "Location") == 0) {
    if (strlen(sValue)+1 < sizeof(_aRedirectURL)) {
      strcpy(_aRedirectURL, sValue);
    }
  }
}

/*********************************************************************
*
*       _ParseURL()
*
*  Function description
*    Parse a URL for the HTTP(S) scheme.
*
*  Parameters
*    pContext - Pointer to HTTP request context.
*    sText    - Pointer to zero-terminated URL.
*/
static void _ParseURL(IOT_HTTP_CONTEXT *pContext, char *sText) {
  char *pPos;
  char *sHost;
  char *sPath;
  //
  if (IOT_STRNCMP(sText, "https:", 6) == 0) {
    IOT_HTTP_AddScheme(pContext, "https");
    IOT_HTTP_SetPort  (pContext, 443);
```



```
    sText += 6;
  } else if (IOT_STRNCMP(sText, "http:", 5) == 0) {
    IOT_HTTP_AddScheme(pContext, "http");
    IOT_HTTP_SetPort  (pContext, 80);
    sText += 5;
  } else {
    IOT_HTTP_AddScheme(pContext, "http");
    IOT_HTTP_SetPort  (pContext, 80);
  }
  //
  if (IOT_STRNCMP(sText, "//", 2) == 0) {
    sText += 2;
  }
  sHost = sText;
  //
  pPos = IOT_STRCHR(sHost, '/');
  if (pPos) {
    *pPos = '\0';
    sPath = pPos + 1;
  } else {
    sPath = "";
  }
  //
  pPos = IOT_STRCHR(sHost, ':');
  if (pPos) {
    *pPos = '\0';
    IOT_HTTP_SetPort(pContext, (unsigned)strtoul(pPos+1, NULL, 0));
  }
  //
  IOT_HTTP_AddHost(pContext, sHost);
  IOT_HTTP_AddPath(pContext, "/");
  IOT_HTTP_AddPath(pContext, sPath);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_HTTP_CONTEXT   HTTP;
  IOT_HTTP_PARA      aPara[20];
  CONNECTION_CONTEXT Connection;
  char               aBuf[128];
  char               aPayload[128];
  unsigned           StatusCode;
  int                Status;
  //
  SEGGER_SYS_Init();
  SEGGER_SYS_IP_Init();
  SSL_Init();
  //
  IOT_HTTP_Init
  (&HTTP, &aBuf[0], sizeof(aBuf), aPara, SEGGER_COUNTOF(aPara));
  IOT_HTTP_SetIO     (&HTTP, &_PlainIO, &Connection);
  IOT_HTTP_SetVersion(&HTTP, &IOT_HTTP_VERSION_HTTP_1v1);
  IOT_HTTP_AddMethod (&HTTP, "GET");
  IOT_HTTP_AddHost   (&HTTP, "www.segger.com");
```



```
  IOT_HTTP_SetPort   (&HTTP, 80);
  IOT_HTTP_AddPath   (&HTTP, "/");
  //
  for (;;) {
    Status = IOT_HTTP_Connect(&HTTP);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot negotiate a connection to %s:%d!\n",
                           IOT_HTTP_GetHost(&HTTP),
                           IOT_HTTP_GetPort(&HTTP));
      SEGGER_SYS_OS_Halt(100);
    }
    //
    Status = IOT_HTTP_Exec(&HTTP);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot execute GET request!\n",
                           IOT_HTTP_GetMethod(&HTTP));
      SEGGER_SYS_OS_Halt(100);
    }
    //
    Status = IOT_HTTP_ProcessStatusLine(&HTTP, &StatusCode);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot process status line!\n");
      SEGGER_SYS_OS_Halt(100);
    }
    SEGGER_SYS_IO_Printf("Returned status code: %u\n\n", StatusCode);
    //
    Status = IOT_HTTP_ProcessHeaders(&HTTP, _HeaderCallback);
    if (Status < 0) {
      SEGGER_SYS_IO_Printf("Cannot process headers!\n");
      SEGGER_SYS_OS_Halt(100);
    }
    //
    if (StatusCode == 301 || StatusCode == 302 ||
        StatusCode == 303 || StatusCode == 307 ) {
      //
      SEGGER_SYS_IO_Printf("Redirect to %s\n\n", _aRedirectURL);
      //
      IOT_HTTP_Disconnect(&HTTP);
      IOT_HTTP_Reset(&HTTP);
      IOT_HTTP_AddMethod(&HTTP, "GET");
      _ParseURL(&HTTP, _aRedirectURL);
      //
      if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "http") == 0) {
        IOT_HTTP_SetIO(&HTTP, &_PlainIO, &Connection);
      } else if (IOT_STRCMP(IOT_HTTP_GetScheme(&HTTP), "https") == 0) {
        IOT_HTTP_SetIO(&HTTP, &_SecureIO, &Connection);
      } else {
        SEGGER_SYS_IO_Printf("Cannot handle scheme %s!\n",
                             IOT_HTTP_GetScheme(&HTTP));
        SEGGER_SYS_OS_Halt(100);
      }
    } else {
      break;
    }
  }
  //
  IOT_HTTP_GetBegin(&HTTP);
  do {
    Status = IOT_HTTP_GetPayload(&HTTP, &aPayload[0], sizeof(aPayload));
    SEGGER_SYS_IO_Printf("%.*s", Status, aPayload);
  } while (Status > 0);
  IOT_HTTP_GetEnd(&HTTP);
  //
  IOT_HTTP_Disconnect(&HTTP);
  //
  SSL_Exit();
  SEGGER_SYS_IP_Exit();
  SEGGER_SYS_Exit();
```



```
  SEGGER_SYS_OS_PauseBeforeHalt();
  SEGGER_SYS_OS_Halt(0);
}

/************************** End of file ***************************/
```


Chapter 3

JSON parser

3.1 Overview

The JSON parser is a simple and compact parser for JSON values. This library provides a way to parse a JSON Data Interchange Format value in a space-efficient manner: rather than parsing a complete JSON value before processing, this library uses a set of events to deliver the structure and enclosed values within the JSON value efficiently in terms of memory space and processing overhead.

The main benefits are:

- No tree structure is created, so no dynamic memory is required to hold JSON content. This is particularly effective when only some of the delivered JSON content is of interest: all other content can simply be ignored (and therefore discarded) during the parse.
- No recursive program flow, so stack space is bounded.
- Incremental parsing is ideally suited to reading data from the network for delivery to the JSON parser.

There are some limitations with this strategy:

- The JSON parser is a pure parser which detects syntax errors in its input. However, semantic errors, such as duplicate keys in the same object, are left for the client to diagnose.
- The parser fires events as it incrementally parses a JSON value and, therefore, there is no protection that the parser offers against firing events for objects with long-range syntax errors.

Standards reference

The JSON format is formalized in the following RFC:

IETF RFC 1321 — *The JavaScript Object Notation (JSON) Data Interchange Format*

3.2 Tracing events during parsing

The following is a valid JSON object:

```
{
    "name": "My TV",
    "resolutions": [
        { "width": 1280, "height": 720 },
        { "width": 1920, "height": 1080 },
        { "width": 3840, "height": 2160 }
    ]
}
```

This and subsequent examples use the above object to illustrate the operation of the JSON parser.

3.2.1 Setting up the trace

It's illuminating to see which events are fired during parsing and this is achieved using methods that dump state during a parse. What we do first is construct a table of methods that handle the events issued by the parser:

```
static const IOT_JSON_EVENTS _EventAPI = {
  _BeginObject,
  _EndObject,
  _BeginArray,
  _EndArray,
  _Key,
  _String,
  _Number,
  _Literal
};
```

These events are handled by the following methods:

```
static void _BeginObject(IOT_JSON_CONTEXT *pContext);
static void _EndObject  (IOT_JSON_CONTEXT *pContext);
static void _BeginArray (IOT_JSON_CONTEXT *pContext);
static void _EndArray   (IOT_JSON_CONTEXT *pContext);
static void _Key        (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _String     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Number     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Literal    (IOT_JSON_CONTEXT *pContext, const char *sText);
```

Each method displays the method that fired and any associated data. So, for instance, the _Key method is:

```
static void _Key(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Key = \"%s\"\n", sText);
}
```

This method ignores the incoming JSON context and shows the value of the key.

3.2.2 Using the parser

The JSON object is transcribed into a C object:

```
static const char _sJSONValue[] = {
  "{                                                  "
  "    \"name\": \"My TV\",                           "
  "    \"resolutions\": [                             "
  "        { \"width\": 1280, \"height\": 720 },      "
```



```
  "        { \"width\": 1920, \"height\": 1080 },      "
  "        { \"width\": 3840, \"height\": 2160 }       "
  "    ]                                               "
  "}                                                   "
};
```

This is parsed by a few calls to the JSON parser:

```
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;  ❶
  char             acBuf[32];  ❷
  //
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));  ❸
  if (IOT_JSON_Parse(&JSON, _sJSONValue, sizeof(_sJSONValue)-1) == 0) {  ❹
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}
```

❶ Declare a context

The JSON parser uses a single context of type `IOT_JSON_CONTEXT` that maintains the parser state.

❷ Declare a working buffer

During parsing, the JSON parser builds up *tokens* that are passed to the event handlers. The token buffer must be long enough to handle a constructed token such as a key name, a literal value, a string, and so on. The parser diagnoses inputs where the tokens are too long to be constructed in the buffer and reports this as an error.

❸ Initialize the parser

The call to `IOT_JSON_Init()` initializes the parsing context and provides the set of methods which will fire during a subsequent parse. The token buffer is also provided to the parser and this buffer must remain in scope until the parse is complete.

❸ Parse the value

The call to `IOT_JSON_Parse()` provides the JSON value to parse. In this example, the complete value is presented to the parser and we expect events to fire during parsing and a result that indicates the parse completed without error. If we receive anything other than a zero result from `IOT_JSON_Parse()` then the parse has failed.

3.2.3 Run the example

Running this outputs the trace of events that were fired by the JSON parser and shows correct operation, as expected:

```
C:> IOT_JSON_PlainTrace.exe

Begin object
Key = "name"
String = "My TV"
Key = "resolutions"
Begin array
Begin object
Key = "width"
Number = "1280"
Key = "height"
Number = "720"
End object
Begin object
```



```
Key = "width"
Number = "1920"
Key = "height"
Number = "1080"
End object
Begin object
Key = "width"
Number = "3840"
Key = "height"
Number = "2160"
End object
End array
End object

Parse OK

C:> _
```

3.2.4 IOT_JSON_PlainTrace complete listing

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_JSON_PlainTrace.c
Purpose     : Simple parse of a JSON object.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static void _BeginObject(IOT_JSON_CONTEXT *pContext);
static void _EndObject  (IOT_JSON_CONTEXT *pContext);
static void _BeginArray (IOT_JSON_CONTEXT *pContext);
static void _EndArray   (IOT_JSON_CONTEXT *pContext);
static void _Key        (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _String     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Number     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Literal    (IOT_JSON_CONTEXT *pContext, const char *sText);

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_JSON_EVENTS _EventAPI = {
```



```
  _BeginObject,
  _EndObject,
  _BeginArray,
  _EndArray,
  _Key,
  _String,
  _Number,
  _Literal
};

static const char _sJSONValue[] = {
  "{                                                  "
  "    \"name\": \"My TV\",                           "
  "    \"resolutions\": [                             "
  "        { \"width\": 1280, \"height\": 720 },      "
  "        { \"width\": 1920, \"height\": 1080 },     "
  "        { \"width\": 3840, \"height\": 2160 }      "
  "    ]                                              "
  "}                                                  "
};

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _BeginObject()
*
*  Function description
*    Handle Begin Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("Begin object\n");
}

/*********************************************************************
*
*       _EndObject()
*
*  Function description
*    Handle End Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndObject(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("End object\n");
}

/*********************************************************************
*
*       _BeginArray()
*
*  Function description
*    Handle Begin Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
```



```
static void _BeginArray(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("Begin array\n");
}

/*********************************************************************
*
*       _EndArray()
*
*  Function description
*    Handle End Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndArray(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("End array\n");
}

/*********************************************************************
*
*       _Key()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated key ID.
*/
static void _Key(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Key = \"%s\"\n", sText);
}

/*********************************************************************
*
*       _String()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated string value.
*/
static void _String(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("String = \"%s\"\n", sText);
}

/*********************************************************************
*
*       _Number()
*
*  Function description
*    Handle Number event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated numeric value.
*/
static void _Number(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Number = \"%s\"\n", sText);
}
```



```
/*********************************************************************
*
*       _Literal()
*
*  Function description
*    Handle Literal event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated literal name.
*/
static void _Literal(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Literal = \"%s\"\n", sText);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  char             acBuf[32];
  //
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  if (IOT_JSON_Parse(&JSON, _sJSONValue, sizeof(_sJSONValue)-1) == 0) {
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}

/************************** End of file ****************************/
```


3.3 Adding a user context

The previous example shows how the events are fired, but this is "context free" in that no user state is affected when the events fire. In this example we attach a user state to the parse which records how deeply nested each object, array, and value is and uses that to produce a nicely indented trace of the events that reflects the JSON object structure.

This example extends the previous example to add this state.

3.3.1 The user context

The only thing that is maintained across events is the nesting level of objects, arrays, and keys:

```
typedef struct {
  unsigned Indent;
} USER_CONTEXT;
```

The user context is attached to the parser with `IOT_JSON_SetUserContext()`:

```
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  USER_CONTEXT     User;  ❶
  char             acBuf[32];
  //
  User.Indent = 0;  ❷
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  IOT_JSON_SetUserContext(&JSON, &User);  ❸
  if (IOT_JSON_Parse(&JSON, _sJSONValue, sizeof(_sJSONValue)-1) == 0) {
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}
```

❶ Declare a user context

This is the user context that the parser will pass through to the event handlers.

❷ Initialize user context

The example starts with no indentation for trace messages.

❸ Attach user context

The user context is attached to the JSON parser context with `IOT_JSON_SetUserContext()`. The user context can be retrieved from the JSON parser context using `IOT_JSON_GetUserContext()`.

3.3.2 Using the user context

Now that the user context is attached to the JSON parser context, the methods that are invoked during parsing can retrieve that user context and maintain their own model of the parsing process. In this example, each method indents its messages to indicate the structure of the value parsed.

Rather than present all methods, a single method is used as an example:

```
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sBegin object\n", pUserContext->Indent, "");
  pUserContext->Indent += 2;
```



```
}
```

3.3.3 Run the example

Running this outputs the trace of events that were fired by the JSON parser and shows correct operation, as expected:

```
C:> IOT_JSON_PrettyTrace.exe

Begin object
  Key = "name"
  String = "My TV"
  Key = "resolutions"
  Begin array
    Begin object
      Key = "width"
      Number = "1280"
      Key = "height"
      Number = "720"
    End object
    Begin object
      Key = "width"
      Number = "1920"
      Key = "height"
      Number = "1080"
    End object
    Begin object
      Key = "width"
      Number = "3840"
      Key = "height"
      Number = "2160"
    End object
  End array
End object

Parse OK
```

3.3.4 IOT_JSON_PrettyTrace complete listing

```
/*********************************************************************
*                    (c) SEGGER Microcontroller GmbH                 *
*                         The Embedded Experts                       *
*                            www.segger.com                          *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_JSON_PrettyTrace.c
Purpose     : Simple parse of a JSON object.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"

/*********************************************************************
*
*       Local types
```



```
*
**********************************************************************
*/

typedef struct {
  unsigned Indent;
} USER_CONTEXT;

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static void _BeginObject(IOT_JSON_CONTEXT *pContext);
static void _EndObject  (IOT_JSON_CONTEXT *pContext);
static void _BeginArray (IOT_JSON_CONTEXT *pContext);
static void _EndArray   (IOT_JSON_CONTEXT *pContext);
static void _Key        (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _String     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Number     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Literal    (IOT_JSON_CONTEXT *pContext, const char *sText);

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_JSON_EVENTS _EventAPI = {
  _BeginObject,
  _EndObject,
  _BeginArray,
  _EndArray,
  _Key,
  _String,
  _Number,
  _Literal
};

static const char _sJSONValue[] = {
  "{                                                  "
  "    \"name\": \"My TV\",                           "
  "    \"resolutions\": [                             "
  "        { \"width\": 1280, \"height\": 720 },      "
  "        { \"width\": 1920, \"height\": 1080 },     "
  "        { \"width\": 3840, \"height\": 2160 }      "
  "    ]                                              "
  "}                                                  "
};

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _BeginObject()
*
*  Function description
*    Handle Begin Object event.
*
*  Parameters
```



```
*    pContext - Pointer to JSON context.
*/
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sBegin object\n", pUserContext->Indent, "");
  pUserContext->Indent += 2;
}

/*********************************************************************
*
*       _EndObject()
*
*  Function description
*    Handle End Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pUserContext->Indent -= 2;
  printf("%-*sEnd object\n", pUserContext->Indent, "");
}

/*********************************************************************
*
*       _BeginArray()
*
*  Function description
*    Handle Begin Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _BeginArray(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sBegin array\n", pUserContext->Indent, "");
  pUserContext->Indent += 2;
}

/*********************************************************************
*
*       _EndArray()
*
*  Function description
*    Handle End Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndArray(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pUserContext->Indent -= 2;
  printf("%-*sEnd array\n", pUserContext->Indent, "");
}

/*********************************************************************
*
*       _Key()
*
```



```
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated key ID.
*/
static void _Key(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sKey = \"%s\"\n", pUserContext->Indent, "", sText);
}

/*********************************************************************
*
*       _String()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated string value.
*/
static void _String(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sString = \"%s\"\n", pUserContext->Indent, "", sText);
}

/*********************************************************************
*
*       _Number()
*
*  Function description
*    Handle Number event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated numeric value.
*/
static void _Number(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sNumber = \"%s\"\n", pUserContext->Indent, "", sText);
}

/*********************************************************************
*
*       _Literal()
*
*  Function description
*    Handle Literal event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated literal name.
*/
static void _Literal(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  printf("%-*sLiteral = \"%s\"\n", pUserContext->Indent, "", sText);
}
```



```
/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  USER_CONTEXT     User;
  char             acBuf[32];
  //
  User.Indent = 0;
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  IOT_JSON_SetUserContext(&JSON, &User);
  if (IOT_JSON_Parse(&JSON, _sJSONValue, sizeof(_sJSONValue)-1) == 0) {
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}

/************************** End of file ****************************/
```


3.4 Incremental parsing

The previous examples have shown how to parse a single JSON object, in its entirety, in a single call to `IOT_JSON_Parse()`. This example demonstrates how to parse JSON values incrementally and how it is perfectly suited to parsing data originating from a socket or from the SEGGER HTTP client.

3.4.1 Setting up the parse

Many JSON values are delivered in small chunks over an HTTP connection as part of a REST API's entity body. The following code presents a JSON value, one character at a time, to the JSON parser. However, varying lengths chunks of data can be passed to the parser as they become available to your application:

```
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  char             acBuf[32];
  unsigned         i;
  int              Status;
  //
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  //
  i = 0;  ❶
  do {
    Status = IOT_JSON_Parse(&JSON, &_sJSONValue[i++], 1);  ❷
  } while (Status > 0);  ❸
  //
  if (Status == 0) {  ❹
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}
```

❶ Prepare to parse

The cursor `i` iterates over the JSON value one character at a time.

❷ Parse a little bit

The JSON parser is given one character of data and the cursor is advanced. The return value is saved so that it can be examined later.

❸ Continue parsing?

A non-zero positive value returned by `IOT_JSON_Parse()` indicates that parsing of the JSON value is not complete. In this case, the loop continues and presents the next character.

❹ Decode completion status

The completion status here is identical to the all-in-one parse of previous examples.

3.4.2 Run the example

Running this outputs the trace of events that were fired by the JSON parser and shows correct operation, as expected:

```
C:> IOT_JSON_IncrementalParse.exe

Begin object
Key = "name"
String = "My TV"
Key = "resolutions"
```



```
Begin array
Begin object
Key = "width"
Number = "1280"
Key = "height"
Number = "720"
End object
Begin object
Key = "width"
Number = "1920"
Key = "height"
Number = "1080"
End object
Begin object
Key = "width"
Number = "3840"
Key = "height"
Number = "2160"
End object
End array
End object

Parse OK

C:> _
```

3.4.3 IOT_JSON_IncrementalParse complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IOT_JSON_IncrementalParse.c
Purpose     : Incremental parse of a JSON object.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static void _BeginObject(IOT_JSON_CONTEXT *pContext);
static void _EndObject  (IOT_JSON_CONTEXT *pContext);
static void _BeginArray (IOT_JSON_CONTEXT *pContext);
static void _EndArray   (IOT_JSON_CONTEXT *pContext);
static void _Key        (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _String     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Number     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Literal    (IOT_JSON_CONTEXT *pContext, const char *sText);
```



```
/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_JSON_EVENTS _EventAPI = {
  _BeginObject,
  _EndObject,
  _BeginArray,
  _EndArray,
  _Key,
  _String,
  _Number,
  _Literal
};

static const char _sJSONValue[] = {
  "{                                                  "
  "    \"name\": \"My TV\",                           "
  "    \"resolutions\": [                             "
  "        { \"width\": 1280, \"height\": 720 },      "
  "        { \"width\": 1920, \"height\": 1080 },     "
  "        { \"width\": 3840, \"height\": 2160 }      "
  "    ]                                              "
  "}                                                  "
};

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _BeginObject()
*
*  Function description
*    Handle Begin Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("Begin object\n");
}

/*********************************************************************
*
*       _EndObject()
*
*  Function description
*    Handle End Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndObject(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("End object\n");
}

/*********************************************************************
*
```



```
*       _BeginArray()
*
*  Function description
*    Handle Begin Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _BeginArray(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("Begin array\n");
}

/*********************************************************************
*
*       _EndArray()
*
*  Function description
*    Handle End Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndArray(IOT_JSON_CONTEXT *pContext) {
  IOT_USE_PARA(pContext);
  printf("End array\n");
}

/*********************************************************************
*
*       _Key()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated key ID.
*/
static void _Key(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Key = \"%s\"\n", sText);
}

/*********************************************************************
*
*       _String()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated string value.
*/
static void _String(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("String = \"%s\"\n", sText);
}

/*********************************************************************
*
*       _Number()
*
*  Function description
*    Handle Number event.
*
*  Parameters
```



```
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated numeric value.
*/
static void _Number(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Number = \"%s\"\n", sText);
}

/*********************************************************************
*
*       _Literal()
*
*  Function description
*    Handle Literal event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated literal name.
*/
static void _Literal(IOT_JSON_CONTEXT *pContext, const char *sText) {
  IOT_USE_PARA(pContext);
  printf("Literal = \"%s\"\n", sText);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  char             acBuf[32];
  unsigned         i;
  int              Status;
  //
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  //
  i = 0;
  do {
    Status = IOT_JSON_Parse(&JSON, &_sJSONValue[i++], 1);
  } while (Status > 0);
  //
  if (Status == 0) {
    printf("\nParse OK\n");
  } else {
    printf("\nParse FAIL\n");
  }
}

/************************** End of file ****************************/
```


3.5 Building a value tree

The previous examples have shown how to parse a single JSON object and act on each event that is fired. The following example uses the sequence of events to build a tree representation of the parsed value which can then be queried and printed. This example can form the basis of a more elaborate framework that transforms the trees read and, after transformation, creates a standard external interchange representation.

3.5.1 JSON nodes

The internal representation of the JSON value is by way of a tree of nodes. Each node has a type that corresponds to a JSON value:

```
typedef enum {
  JSON_TYPE_OBJECT,
  JSON_TYPE_ARRAY,
  JSON_TYPE_PAIR,
  JSON_TYPE_STRING,
  JSON_TYPE_NUMBER,
  JSON_TYPE_TRUE,
  JSON_TYPE_FALSE,
  JSON_TYPE_NULL,
} JSON_TYPE;
```

The node is made as simple as possible:

```
typedef struct JSON_NODE JSON_NODE;
struct JSON_NODE {
  JSON_TYPE   Type;    // Type of this node
  JSON_NODE * pNext;   // Next node in list (for arrays and object keys)
  JSON_NODE * pParent; // Pointer to node's parent
  JSON_NODE * pValue;  // Pointer to value (JSON_TYPE_{KEY,ARRAY,OBJECT})
  char      * sStr;    // Only valid for JSON_TYPE_{PAIR,STRING}
  double      Number;  // Only valid for JSON_TYPE_NUMBER
};
```

The members applicable to each node are noted in the comments.

3.5.2 The user context

The context that the node builder uses is:

```
typedef struct {
  JSON_NODE *pContainer;
  JSON_NODE *pRoot;
} USER_CONTEXT;
```

The `pRoot` member points to the root node during construction and the `pContainer` member points to the most-recently-opened container (an array, object, or pair node) that accumulates new values.

3.5.3 Constructing a new node

New nodes are created by `_MakeNode()` which contains all the machinery to construct nodes and the housekeeping to maintain the relationships between nodes:

```
static JSON_NODE *_MakeNode(JSON_TYPE Type, USER_CONTEXT *pContext) {
  JSON_NODE *pNode;
  //
  pNode = malloc(sizeof(JSON_NODE));  ❶
  memset(pNode, 0, sizeof(JSON_NODE));
  pNode->Type    = Type;
```



```
  pNode->pParent = pContext->pContainer;  ❷
  //
  if (pContext->pRoot == NULL) {  ❸
    pContext->pRoot = pNode;
  } else {
    _AppendNode(&pContext->pContainer->pValue, pNode);
  }
  //
  if (Type == JSON_TYPE_ARRAY ||  ❹
      Type == JSON_TYPE_OBJECT ||
      Type == JSON_TYPE_PAIR) {
    pContext->pContainer = pNode;
  } else if (pContext->pContainer->Type == JSON_TYPE_PAIR) {  ❺
    pContext->pContainer = pContext->pContainer->pParent;
  }
  //
  return pNode;
}
```

The following points are noteworthy:

❶ Create the node

The node is allocated from the heap, set to zero, and members initialized. Using a simple mark-allocate-release style of allocator would be an excellent alternative in an embedded system.

❷ Note the parent

Each created node is initialized with a parental link. This link is used to climb back up the tree when an array or object is closed by one of the "end" methods.

❸ Housekeeping for roots and lists

The initial allocation of a node is always noted as this is the root node and the root node is never part of a list. All other nodes are appended to the end of the container that is accumulating them.

❹ Opening a new container

Newly constructed containers become the container for new nodes read by the parser, up to the point that the container is closed.

❺ Auto-closing a pair

The `JSON_TYPE_PAIR` container can only contain a single element that is the value that the key maps to. Once a value is added to the key-value pair, the key-value pair is closed and accumulation returns to the parent container.

3.5.4 Handling events

With the construction code written, handling events is little more than creating a node of the appropriate type when invoked:

```
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  _MakeNode(JSON_TYPE_OBJECT, pUserContext);
}
```

And closing down containers when required:

```
static void _EndObject(IOT_JSON_CONTEXT *pContext) {
```



```
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pUserContext->pContainer = pUserContext->pContainer->pParent;
}
```

Because the JSON parser will not fire events in an incorrect order (i.e. all paired events are invoked in the correct order), is it not necessary for the event handlers to check the validity of event sequences. That is to say, even in the presence of syntax errors, the sequence of events is well defined such that "begin *x*" is only ever paired with a corresponding "end *x*" event. This makes it straightdorward to write event handling code.

3.5.5 Back to external form

It's much simpler to convert the internal form to a human-readable external form. This function prints a JSON-style interchange representation but does not, for instance, produce any escapes in strings that would be necessary in practice:

```
static void _PrintNode(JSON_NODE *pNode, unsigned Indent) {
  for (; pNode; pNode = pNode->pNext) {
    switch (pNode->Type) {
    case JSON_TYPE_OBJECT:
      printf("\n%*s{", Indent, "");
      _PrintNode(pNode->pValue, Indent+2);
      printf("\n%*s}", Indent, "");
      break;
    case JSON_TYPE_ARRAY:
      printf("\n%*s[", Indent, "");
      _PrintNode(pNode->pValue, Indent+2);
      printf("\n%*s]", Indent, "");
      break;
    case JSON_TYPE_PAIR:
      printf("\n%*s\"%s\": ", Indent, "", pNode->sStr);
      _PrintNode(pNode->pValue, Indent+2);
      break;
    case JSON_TYPE_STRING:
      printf("\"%s\"", pNode->sStr);
      break;
    case JSON_TYPE_NUMBER:
      printf("%g", pNode->Number);
      break;
    case JSON_TYPE_TRUE:
      printf("true");
      break;
    case JSON_TYPE_FALSE:
      printf("false");
      break;
    case JSON_TYPE_NULL:
      printf("null");
      break;
    }
    if (pNode->pNext) {
      printf(", ");
    }
  }
}
```

3.5.6 Running the sample

With the sample now complete, running it produces the following output:

```
C:> IOT_JSON_MakeTree.exe

Parse OK, tree:
```



```
{
  "name": "My TV",
  "resolutions":
    [
      {
        "width": 1280,
        "height": 720
      },
      {
        "width": 1920,
        "height": 1080
      },
      {
        "width": 3840,
        "height": 2160
      }
    ]
}

C:> _
```

3.5.7 IOT_JSON_MakeTree complete listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

------------------------- END-OF-HEADER -----------------------------

File        : IOT_JSON_MakeTreeM.c
Purpose     : Parse a JSON object and create an tree structure
              that represents the object.

*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/

#include "IOT.h"
#include <stdlib.h>

/*********************************************************************
*
*       Local types
*
**********************************************************************
*/

typedef enum {
  JSON_TYPE_OBJECT,
  JSON_TYPE_ARRAY,
  JSON_TYPE_PAIR,
  JSON_TYPE_STRING,
  JSON_TYPE_NUMBER,
  JSON_TYPE_TRUE,
  JSON_TYPE_FALSE,
  JSON_TYPE_NULL,
} JSON_TYPE;
```



```
typedef struct JSON_NODE JSON_NODE;
struct JSON_NODE {
  JSON_TYPE   Type;    // Type of this node
  JSON_NODE * pNext;   // Next node in list (linking arrays and object keys)
  JSON_NODE * pParent; // Pointer to node's parent
  JSON_NODE * pValue;  // Pointer to value
 (JSON_TYPE_KEY, JSON_TYPE_ARRAY, JSON_TYPE_OBJECT)
  char      * sStr;    // Only valid for JSON_TYPE_PAIR and JSON_TYPE_STRING
  double      Number;  // Only valid for JSON_TYPE_NUMBER
};

typedef struct {
  JSON_NODE *pContainer;
  JSON_NODE *pRoot;
} USER_CONTEXT;

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static void _BeginObject(IOT_JSON_CONTEXT *pContext);
static void _EndObject  (IOT_JSON_CONTEXT *pContext);
static void _BeginArray (IOT_JSON_CONTEXT *pContext);
static void _EndArray   (IOT_JSON_CONTEXT *pContext);
static void _Key        (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _String     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Number     (IOT_JSON_CONTEXT *pContext, const char *sText);
static void _Literal    (IOT_JSON_CONTEXT *pContext, const char *sText);

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const IOT_JSON_EVENTS _EventAPI = {
  _BeginObject,
  _EndObject,
  _BeginArray,
  _EndArray,
  _Key,
  _String,
  _Number,
  _Literal
};

static const char _sJSONValue[] = {
  "{                                                  "
  "    \"name\": \"My TV\",                           "
  "    \"resolutions\": [                             "
  "        { \"width\": 1280, \"height\": 720 },      "
  "        { \"width\": 1920, \"height\": 1080 },     "
  "        { \"width\": 3840, \"height\": 2160 }      "
  "    ]                                              "
  "}                                                  "
};

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/
```



```
/*********************************************************************
*
*       _AppendNode()
*
*  Function description
*    Append node onto list.
*
*  Parameters
*    ppRoot - Pointer to list root.
*    pNode  - Pointer to node to append.
*/
static void _AppendNode(JSON_NODE **ppRoot, JSON_NODE *pNode) {
  if (*ppRoot == NULL) {
    *ppRoot = pNode;
  } else {
    while ((*ppRoot)->pNext != NULL) {
      ppRoot = &(*ppRoot)->pNext;
    }
    (*ppRoot)->pNext = pNode;
  }
}

/*********************************************************************
*
*       _MakeNode()
*
*  Function description
*    Create a new node, manage lists.
*
*  Parameters
*    Type     - Type of node to construct.
*    pContext - Pointer to use context.
*
*  Return value
*    Pointer to newly constructed node.
*/
static JSON_NODE *_MakeNode(JSON_TYPE Type, USER_CONTEXT *pContext) {
  JSON_NODE *pNode;
  //
  pNode = malloc(sizeof(JSON_NODE));
  memset(pNode, 0, sizeof(JSON_NODE));
  pNode->Type    = Type;
  pNode->pParent = pContext->pContainer;
  //
  if (pContext->pRoot == NULL) {
    pContext->pRoot = pNode;
  } else {
    _AppendNode(&pContext->pContainer->pValue, pNode);
  }
  //
  if (Type == JSON_TYPE_ARRAY || Type == JSON_TYPE_OBJECT || Type == JSON_TYPE_PAIR) {
    pContext->pContainer = pNode;
  } else if (pContext->pContainer->Type == JSON_TYPE_PAIR) {
    pContext->pContainer = pContext->pContainer->pParent;
  }
  //
  return pNode;
}

/*********************************************************************
*
*       _BeginObject()
*
*  Function description
*    Handle Begin Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
```



```
*/
static void _BeginObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  _MakeNode(JSON_TYPE_OBJECT, pUserContext);
}

/*********************************************************************
*
*       _EndObject()
*
*  Function description
*    Handle End Object event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndObject(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pUserContext->pContainer = pUserContext->pContainer->pParent;
}

/*********************************************************************
*
*       _BeginArray()
*
*  Function description
*    Handle Begin Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _BeginArray(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  _MakeNode(JSON_TYPE_ARRAY, pUserContext);
}

/*********************************************************************
*
*       _EndArray()
*
*  Function description
*    Handle End Array event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*/
static void _EndArray(IOT_JSON_CONTEXT *pContext) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pUserContext->pContainer = pUserContext->pContainer->pParent;
}

/*********************************************************************
*
*       _Key()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
```



```
*    sText    - Zero-terminated key ID.
*/
static void _Key(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  JSON_NODE    *pNode;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pNode = _MakeNode(JSON_TYPE_PAIR, pUserContext);
  pNode->sStr = malloc(strlen(sText)+1);
  strcpy(pNode->sStr, sText);
}

/*********************************************************************
*
*       _String()
*
*  Function description
*    Handle Key event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated string value.
*/
static void _String(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  JSON_NODE    *pNode;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pNode = _MakeNode(JSON_TYPE_STRING, pUserContext);
  pNode->sStr = malloc(strlen(sText)+1);
  strcpy(pNode->sStr, sText);
}

/*********************************************************************
*
*       _Number()
*
*  Function description
*    Handle Number event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated numeric value.
*/
static void _Number(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  JSON_NODE    *pNode;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
  pNode = _MakeNode(JSON_TYPE_NUMBER, pUserContext);
  sscanf(sText, "%lf", &pNode->Number);
}

/*********************************************************************
*
*       _Literal()
*
*  Function description
*    Handle Literal event.
*
*  Parameters
*    pContext - Pointer to JSON context.
*    sText    - Zero-terminated literal name.
*/
static void _Literal(IOT_JSON_CONTEXT *pContext, const char *sText) {
  USER_CONTEXT *pUserContext;
  //
  pUserContext = IOT_JSON_GetUserContext(pContext);
```



```
  if (strcmp(sText, "true") == 0) {
    _MakeNode(JSON_TYPE_TRUE, pUserContext);
  } else if (strcmp(sText, "true") == 0) {
    _MakeNode(JSON_TYPE_FALSE, pUserContext);
  } else {
    _MakeNode(JSON_TYPE_NULL, pUserContext);
  }
}

/*********************************************************************
*
*       _PrintNode()
*
*  Function description
*    Print node.
*
*  Parameters
*    pNode  - Pointer to node to print.
*    Indent - Indentation level for node.
*/
static void _PrintNode(JSON_NODE *pNode, unsigned Indent) {
  for (; pNode; pNode = pNode->pNext) {
    switch (pNode->Type) {
    case JSON_TYPE_OBJECT:
      printf("\n%*s{", Indent, "");
      _PrintNode(pNode->pValue, Indent+2);
      printf("\n%*s}", Indent, "");
      break;
    case JSON_TYPE_ARRAY:
      printf("\n%*s[", Indent, "");
      _PrintNode(pNode->pValue, Indent+2);
      printf("\n%*s]", Indent, "");
      break;
    case JSON_TYPE_PAIR:
      printf("\n%*s\"%s\": ", Indent, "", pNode->sStr);
      _PrintNode(pNode->pValue, Indent+2);
      break;
    case JSON_TYPE_STRING:
      printf("\"%s\"", pNode->sStr);
      break;
    case JSON_TYPE_NUMBER:
      printf("%g", pNode->Number);
      break;
    case JSON_TYPE_TRUE:
      printf("true");
      break;
    case JSON_TYPE_FALSE:
      printf("false");
      break;
    case JSON_TYPE_NULL:
      printf("null");
      break;
    }
    if (pNode->pNext) {
      printf(", ");
    }
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
```



```
*       MainTask()
*
*  Function description
*    Application entry point.
*/
void MainTask(void);
void MainTask(void) {
  IOT_JSON_CONTEXT JSON;
  USER_CONTEXT     User;
  char             acBuf[32];
  //
  User.pContainer = NULL;
  User.pRoot      = NULL;
  IOT_JSON_Init(&JSON, &_EventAPI, &acBuf[0], sizeof(acBuf));
  IOT_JSON_SetUserContext(&JSON, &User);
  if (IOT_JSON_Parse(&JSON, _sJSONValue, sizeof(_sJSONValue)-1) == 0) {
    printf("\nParse OK, tree:\n");
    _PrintNode(User.pRoot, 0);
    printf("\n");
  } else {
    printf("\nParse FAIL\n");
  }
}

/************************** End of file ****************************/
```


Chapter 4

API reference

4.1 HTTP client

This section describes the API of the HTTP client.

4.1.1 Preprocessor symbols

4.1.1.1 HTTP client errors

Description

Errors that the HTTP client can generate.

Definition

```
#define IOT_HTTP_ERROR_FIELD_TOO_LONG       (-700)
#define IOT_HTTP_ERROR_SOCKET_TERMINATED    (-701)
#define IOT_HTTP_ERROR_BAD_SYNTAX           (-702)
```

Symbols

Definition	Description
IOT_HTTP_ERROR_FIELD_TOO_LONG	When parsing HTTP responses
IOT_HTTP_ERROR_SOCKET_TERMINATED	Socket closed when reading response
IOT_HTTP_ERROR_BAD_SYNTAX	Value doesn’t conform to valid syntax

4.1.2 I/O types

Type	Description
IOT_IO_CONNECT_FUNC	Connect transport.
IOT_IO_DISCONNECT_FUNC	Disconnect transport.
IOT_IO_SEND_FUNC	Send data to transport.
IOT_IO_RECV_FUNC	Receive data from transport.
IOT_IO_API	I/O API for HTTP client.

4.1.2.1 IOT_IO_CONNECT_FUNC

Description

Connect transport.

Type definition

```
typedef int (IOT_IO_CONNECT_FUNC)(      void   * pSession,
                                  const char   * sHost,
                                        unsigned Port);
```

Parameters

Parameter	Description
pSession	Pointer to user-provided session context.
sHost	Zero-terminates string describing host to connect to.
Port	Port to connect to.

Return value

≥ 0 Success, connected.
< 0 Failure.

4.1.2.2 IOT_IO_DISCONNECT_FUNC

Description

Disconnect transport.

Type definition

```
typedef int (IOT_IO_DISCONNECT_FUNC)(void * pSession);
```

Parameters

Parameter	Description
pSession	Pointer to user-provided session context.

Return value

≥ 0 Success, disconnected.
< 0 Failure.

4.1.2.3 IOT_IO_SEND_FUNC

Description

Send data to transport.

Type definition

```
typedef int (IOT_IO_SEND_FUNC)(      void   * pSession,
                               const void   * pData,
                                     unsigned DataLen);
```

Parameters

Parameter	Description
pSession	Pointer to user-provided session context.
pData	Pointer to octet string to write to transport.
DataLen	Octet length of the octet string.

Return value

≥ 0 Success, written.
< 0 Failure.

4.1.2.4 IOT_IO_RECV_FUNC

Description

Receive data from transport.

Type definition

```
typedef int (IOT_IO_RECV_FUNC)(void   * pSession,
                               void   * pData,
                               unsigned DataLen);
```

Parameters

Parameter	Description
pSession	Pointer to user-provided session context.
pData	Pointer to octet string that receives data from transport.
DataLen	Octet length of the octet string.

Return value

> 0 Success, number of bytes read from transport.
= 0 Underlying transport closed gracefully.
< 0 Failure.

4.1.2.5 IOT_IO_API

Description

I/O API for HTTP client.

Type definition

```
typedef struct {
  IOT_IO_CONNECT_FUNC    * pfConnect;
  IOT_IO_DISCONNECT_FUNC * pfDisconnect;
  IOT_IO_SEND_FUNC       * pfSend;
  IOT_IO_RECV_FUNC       * pfRecv;
} IOT_IO_API;
```

Structure members

Member	Description
pfConnect	Connect transport method.
pfDisconnect	Disconnect transport method.
pfSend	Send data to transport method.
pfRecv	Receive data from transport method.

4.1.3 Request-related types

Type	Description
IOT_HTTP_PARA	Parameters for an HTTP request.
IOT_HTTP_PART_TYPE	Individual parts of an HTTP (or other) request.

4.1.3.1 IOT_HTTP_PARA

Description

Parameters for an HTTP request.

Type definition

```
typedef struct {
  const char         * sKey;
  const char         * sValue;
  IOT_HTTP_PART_TYPE   Type;
} IOT_HTTP_PARA;
```

Structure members

Member	Description
sKey	Internal: Pointer to key name or NULL
sValue	Internal: Pointer to part value or NULL
Type	Internal: Part type

Additional information

All fields in this structure are private.

4.1.3.2 IOT_HTTP_PART_TYPE

Description

Individual parts of an HTTP (or other) request.

Type definition

```
typedef enum {
  IOT_HTTP_PART_TYPE_SCHEME,
  IOT_HTTP_PART_TYPE_USER,
  IOT_HTTP_PART_TYPE_PASSWORD,
  IOT_HTTP_PART_TYPE_HOST,
  IOT_HTTP_PART_TYPE_PATH,
  IOT_HTTP_PART_TYPE_QUERY,
  IOT_HTTP_PART_TYPE_FRAGMENT,
  IOT_HTTP_PART_TYPE_METHOD,
  IOT_HTTP_PART_TYPE_HEADER,
  IOT_HTTP_PART_TYPE_SIGNED_HEADER,
  IOT_HTTP_PART_TYPE_CONTENT
} IOT_HTTP_PART_TYPE;
```

Enumeration constants

Constant	Description
IOT_HTTP_PART_TYPE_SCHEME	Scheme type (http, https…)
IOT_HTTP_PART_TYPE_USER	User part (e.g. for ftp scheme)
IOT_HTTP_PART_TYPE_PASSWORD	Password part (e.g. for ftp scheme)
IOT_HTTP_PART_TYPE_HOST	Host part
IOT_HTTP_PART_TYPE_PATH	Path fragment
IOT_HTTP_PART_TYPE_QUERY	Query parameter (introduced by "?")
IOT_HTTP_PART_TYPE_FRAGMENT	Fragment part of URI (introduced by "#")
IOT_HTTP_PART_TYPE_METHOD	Method part of request (GET, POST…)
IOT_HTTP_PART_TYPE_HEADER	Request key-value header
IOT_HTTP_PART_TYPE_SIGNED_HEADER	Request header takes part in signature
IOT_HTTP_PART_TYPE_CONTENT	Entity body content

4.1.4 Callbacks

Type	Description
IOT_HTTP_HEADER_CALLBACK_FUNC	HTTP response header callback.
IOT_HTTP_ENUMERATE_CALLBACK_FUNC	HTTP request header enumeration callback.
IOT_HTTP_AUTH_FUNC	Write authorization header.

4.1.4.1 IOT_HTTP_HEADER_CALLBACK_FUNC

Description

HTTP response header callback.

Type definition

```
typedef void ( * IOT_HTTP_HEADER_CALLBACK_FUNC(      (IOT_HTTP_CONTEXT * pContext,
                                                const char              * sKey,
                                                const char              * sValue);
```

Parameters

Parameter	Description
pConext	Pointer to HTTP context.
sKey	Zero-terminated header key.
sValue	Zero-terminated header value.

Additional information

A header callback is invoked through `IOT_HTTP_ProcessHeaders()`, with each header in the HTTP response invoking a header callback.

4.1.4.2 IOT_HTTP_ENUMERATE_CALLBACK_FUNC

Description

HTTP request header enumeration callback.

Type definition

```
typedef void ( * IOT_HTTP_ENUMERATE_CALLBACK_FUNC(
  (IOT_HTTP_CONTEXT * pContext,
                                                  const char              * sKey,
                                                  const char              * sValue,
                                                        unsigned            Index);
```

Parameters

Parameter	Description
pConext	Pointer to HTTP context.
sKey	Zero-terminated header key.
sValue	Zero-terminated header value.
Index	Zero-based index of header parameter.

Additional information

A header enumeration callback is invoked through `IOT_HTTP_EnumerateParas()`, with each parameter in the HTTP request invoking a header callback.

4.1.4.3 IOT_HTTP_AUTH_FUNC

Description

Write authorization header.

Type definition

```
typedef void (IOT_HTTP_AUTH_FUNC)(IOT_HTTP_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to HTTP context.

Additional information

This function is registered by `IOT_HTTP_SetAuth()` and is invoked when the request is executed in order to construct and send any authorization header computed over the HTTP parameters.

4.1.5 Information functions

The table below lists the functions that return HTTP Client information.

Function	Description
IOT_HTTP_GetVersionText()	Get HTTP Client version as printable string.
IOT_HTTP_GetCopyrightText()	Get HTTP Client copyright as printable string.

4.1.5.1 IOT_HTTP_GetVersionText()

Description

Get HTTP Client version as printable string.

Prototype

```
char *IOT_HTTP_GetVersionText(void);
```

Return value

Zero-terminated version string.

4.1.5.2 IOT_HTTP_GetCopyrightText()

Description

Get HTTP Client copyright as printable string.

Prototype

```
char *IOT_HTTP_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

4.1.6 Functions

Function	Description
Setup	
IOT_HTTP_Init()	Initialize request.
IOT_HTTP_Reset()	Reset request and clear parameters.
IOT_HTTP_SetIO()	Set transport I/O.
IOT_HTTP_SetAuth()	Set authorization method.
IOT_HTTP_SetVersion()	Set HTTP version for request.
Construction	
IOT_HTTP_AddScheme()	Add scheme to request.
IOT_HTTP_AddHost()	Add host name to request.
IOT_HTTP_AddUser()	Add user name to request.
IOT_HTTP_AddPassword()	Add password to request.
IOT_HTTP_AddPath()	Add path fragment to request.
IOT_HTTP_AddQuery()	Add query to request.
IOT_HTTP_AddFragment()	Add fragment to request.
IOT_HTTP_AddHeader()	Add header to request.
IOT_HTTP_AddSignedHeader()	Add signed header to request.
IOT_HTTP_AddMethod()	Add method to request.
IOT_HTTP_AddContent()	Add content request.
IOT_HTTP_SetPort()	Set port for request.
Access	
IOT_HTTP_GetScheme()	Get scheme name for request.
IOT_HTTP_GetHost()	Get host name for request.
IOT_HTTP_GetUser()	Get user name for request.
IOT_HTTP_GetPassword()	Get password for request.
IOT_HTTP_GetQuery()	Get query, by key, for request.
IOT_HTTP_GetMethod()	Get method for request.
IOT_HTTP_GetHeader()	Get header, by key, for request.
IOT_HTTP_GetPort()	Get port for request.
IOT_HTTP_EnumerateParas()	Enumerate request components.
Connection	
IOT_HTTP_Connect()	Make HTTP connection.
IOT_HTTP_Disconnect()	Close HTTP connection.
Request	
IOT_HTTP_Exec()	Execute request.
IOT_HTTP_ExecWithAuth()	Execute request with authorization.
IOT_HTTP_ExecWithBearer()	Execute request with bearer token.
IOT_HTTP_PutBegin()	Start sending request payload.
IOT_HTTP_PutPayload()	Send request payload.
IOT_HTTP_PutEnd()	Finish sending request payload.
Response	
IOT_HTTP_ProcessStatusLine()	Process HTTP response status line.
IOT_HTTP_ProcessHeaders()	Process HTTP headers.

Function	Description
IOT_HTTP_GetBegin()	Start receiving response payload.
IOT_HTTP_GetPayload()	Receive response payload.
IOT_HTTP_GetEnd()	Finish receiving response payload.
Utility	
IOT_HTTP_QueryURL()	Query encoded URL.
Low-level I/O	
IOT_HTTP_Recv()	Receive raw data from connection.
IOT_HTTP_Send()	Send raw data to connection.
IOT_HTTP_SendStr()	Send zero-terminated string to connection.

4.1.6.1 IOT_HTTP_AddContent()

Description

Add content request.

Prototype

```
void IOT_HTTP_AddContent(      IOT_HTTP_CONTEXT * pSelf,
                         const char             * sContent);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sContent	Pointer to zero-terminated content string.

4.1.6.2 IOT_HTTP_AddFragment()

Description

Add fragment to request.

Prototype

```
void IOT_HTTP_AddFragment(      IOT_HTTP_CONTEXT * pSelf,
                          const char             * sText);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sText	Pointer to fragment text.

4.1.6.3 IOT_HTTP_AddHeader()

Description

Add header to request.

Prototype

```
void IOT_HTTP_AddHeader(      IOT_HTTP_CONTEXT * pSelf,
                        const char             * sKey,
                        const char             * sValue);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sKey	Pointer to string that is the key.
sValue	Pointer to string that is the value.

4.1.6.4 IOT_HTTP_AddMethod()

Description

Add method to request.

Prototype

```
void IOT_HTTP_AddMethod(      IOT_HTTP_CONTEXT * pSelf,
                        const char             * sPath);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sPath	Pointer to method to add, e.g. “GET”.

Additional information

Only one method should be added to the HTTP context per request.

Methods defined for HTTP according to RFC 2626 are CONNECT, DELETE, GET, HEAD, OPTIONS, POST, PUT, and TRACE.

4.1.6.5 IOT_HTTP_AddHost()

Description

Add host name to request.

Prototype

```
void IOT_HTTP_AddHost(      IOT_HTTP_CONTEXT * pSelf,
                      const char             * sHost);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sHost	Pointer to host name.

4.1.6.6 IOT_HTTP_AddQuery()

Description

Add query to request.

Prototype

```
void IOT_HTTP_AddQuery(      IOT_HTTP_CONTEXT * pSelf,
                       const char             * sKey,
                       const char             * sValue);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sKey	Pointer to string that is the key.
sValue	Pointer to string that is the value.

4.1.6.7 IOT_HTTP_AddUser()

Description

Add user name to request.

Prototype

```
void IOT_HTTP_AddUser(      IOT_HTTP_CONTEXT * pSelf,
                      const char             * sUser);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sUser	Pointer to user name.

Additional information

The user name is not part of an HTTP request but is provided as a convenience for other schemes (such as FTP or Telnet) where user names and passwords are valid.

4.1.6.8 IOT_HTTP_AddPassword()

Description

Add password to request.

Prototype

```
void IOT_HTTP_AddPassword(      IOT_HTTP_CONTEXT * pSelf,
                          const char             * sPass);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sPass	Pointer to password.

Additional information

The password is not part of an HTTP request but is provided as a convenience for other schemes (such as FTP or Telnet) where user names and passwords are valid.

4.1.6.9 IOT_HTTP_AddPath()

Description

Add path fragment to request.

Prototype

```
void IOT_HTTP_AddPath(      IOT_HTTP_CONTEXT * pSelf,
                      const char             * sPath);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sPath	Pointer to path fragment to add.

4.1.6.10 IOT_HTTP_AddScheme()

Description

Add scheme to request.

Prototype

```
void IOT_HTTP_AddScheme(      IOT_HTTP_CONTEXT * pSelf,
                        const char             * sScheme);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sScheme	Pointer to scheme name.

4.1.6.11 IOT_HTTP_AddSignedHeader()

Description

Add signed header to request.

Prototype

```
void IOT_HTTP_AddSignedHeader(      IOT_HTTP_CONTEXT * pSelf,
                              const char             * sKey,
                              const char             * sValue);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sKey	Pointer to string that is the key.
sValue	Pointer to string that is the value.

4.1.6.12 IOT_HTTP_Connect()

Description

Make HTTP connection.

Prototype

```
int IOT_HTTP_Connect(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.13 IOT_HTTP_Disconnect()

Description

Close HTTP connection.

Prototype

```
void IOT_HTTP_Disconnect(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.14 IOT_HTTP_EnumerateParas()

Description

Enumerate request components.

Prototype

```
void IOT_HTTP_EnumerateParas(IOT_HTTP_CONTEXT                 * pSelf,
                             IOT_HTTP_PART_TYPE                 Type,
                             IOT_HTTP_ENUMERATE_CALLBACK_FUNC   pfCallback);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
Type	Component type to match.
pfCallback	Pointer to function for each matched component.

4.1.6.15 IOT_HTTP_Exec()

Description

Execute request.

Prototype

```
int IOT_HTTP_Exec(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.16 IOT_HTTP_ExecWithAuth()

Description

Execute request with authorization.

Prototype

```
int IOT_HTTP_ExecWithAuth(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.17 IOT_HTTP_ExecWithBearer()

Description

Execute request with bearer token.

Prototype

```
int IOT_HTTP_ExecWithBearer(      IOT_HTTP_CONTEXT * pSelf,
                            const char             * sToken);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sToken	Pointer to bearer token.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.18 IOT_HTTP_GetHeader()

Description

Get header, by key, for request.

Prototype

```
char *IOT_HTTP_GetHeader(      IOT_HTTP_CONTEXT * pSelf,
                         const char             * sKey);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sKey	Pointer to string that is the key.

Return value

= NULL Header key not found.
≠ NULL Pointer to value associated with key.

Additional information

The letter case of the key is not significant when matching, so the keys “Content-Length” and “content-length” are considered identical.

4.1.6.19 IOT_HTTP_GetMethod()

Description

Get method for request.

Prototype

```
char *IOT_HTTP_GetMethod(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

= NULL No method set.
≠ NULL Pointer to method.

4.1.6.20 IOT_HTTP_GetBegin()

Description

Start receiving response payload.

Prototype

```
void IOT_HTTP_GetBegin(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.21 IOT_HTTP_GetEnd()

Description

Finish receiving response payload.

Prototype

```
void IOT_HTTP_GetEnd(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.22 IOT_HTTP_GetHost()

Description

Get host name for request.

Prototype

```
char *IOT_HTTP_GetHost(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

= NULL No host name set.
≠ NULL Pointer to host name.

4.1.6.23 IOT_HTTP_GetPayload()

Description

Receive response payload.

Prototype

```
int IOT_HTTP_GetPayload(IOT_HTTP_CONTEXT * pSelf,
                        void             * pData,
                        unsigned           DataLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pData	Pointer to object that receives the data.
DataLen	Nonzero octet length of the object that receives the data.

Return value

> 0 Number of octets received.
= 0 Payload delivery complete.
< 0 Processing error, e.g. socket unexpectedly closed.

Additional information

The payload can be read piecemeal until no more remains, irrespective of the transfer encoding (chunked or identity).

For HTTP/1.1 connections using a chunked transfer encoding, this function manages the chunking of the data at the transport level and only delivers de-chunked data.

For HTTP/1.0 or identity transfer encodings, this function will return data read directly from the transport.

4.1.6.24 IOT_HTTP_GetPort()

Description

Get port for request.

Prototype

```
unsigned IOT_HTTP_GetPort(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

Port number in host byte order.

4.1.6.25 IOT_HTTP_GetUser()

Description

Get user name for request.

Prototype

```
char *IOT_HTTP_GetUser(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

= NULL No user name set.
≠ NULL Pointer to user name.

4.1.6.26 IOT_HTTP_GetPassword()

Description

Get password for request.

Prototype

```
char *IOT_HTTP_GetPassword(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

= NULL No password set.
≠ NULL Pointer to password.

4.1.6.27 IOT_HTTP_GetQuery()

Description

Get query, by key, for request.

Prototype

```
char *IOT_HTTP_GetQuery(      IOT_HTTP_CONTEXT * pSelf,
                        const char             * sKey);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sKey	Pointer to string that is the key.

Return value

= NULL Query key not found.
≠ NULL Pointer to value associated with key.

Additional information

The letter case of the key is not significant when matching, so the keys “Param” and “param” are considered identical.

4.1.6.28 IOT_HTTP_GetScheme()

Description

Get scheme name for request.

Prototype

```
char *IOT_HTTP_GetScheme(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

Return value

= NULL No scheme name set.
≠ NULL Pointer to scheme name.

4.1.6.29 IOT_HTTP_Init()

Description

Initialize request.

Prototype

```
void IOT_HTTP_Init(IOT_HTTP_CONTEXT * pSelf,
                   void             * pBuf,
                   unsigned           BufLen,
                   IOT_HTTP_PARA    * pPara,
                   unsigned           ParaLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pBuf	Pointer to working buffer.
BufLen	Size of working buffer.
pPara	Pointer to working HTTP parameter array.
ParaLen	Number of elements in working HTTP parameter array.

4.1.6.30 IOT_HTTP_ProcessStatusLine()

Description

Process HTTP response status line.

Prototype

```
int IOT_HTTP_ProcessStatusLine(IOT_HTTP_CONTEXT * pSelf,
                               unsigned         * pStatusCode);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pStatusCode	Pointer to object that receives the status code.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.31 IOT_HTTP_ProcessHeaders()

Description

Process HTTP headers.

Prototype

```
int IOT_HTTP_ProcessHeaders(IOT_HTTP_CONTEXT               * pSelf,
                            IOT_HTTP_HEADER_CALLBACK_FUNC   pfCallback);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pfCallback	Pointer to header callback function.

Return value

≥ 0 Success.
< 0 Processing error.

4.1.6.32 IOT_HTTP_PutBegin()

Description

Start sending request payload.

Prototype

```
void IOT_HTTP_PutBegin(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.33 IOT_HTTP_PutEnd()

Description

Finish sending request payload.

Prototype

```
void IOT_HTTP_PutEnd(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.34 IOT_HTTP_PutPayload()

Description

Send request payload.

Prototype

```
int IOT_HTTP_PutPayload(      IOT_HTTP_CONTEXT * pSelf,
                        const void             * pData,
                              unsigned           DataLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pData	Pointer to object to send.
DataLen	Nonzero octet length of object to send.

Return value

> 0 Number of octets received.
= 0 Socket closed gracefully.
< 0 Processing error, e.g. socket unexpectedly closed.

4.1.6.35 IOT_HTTP_QueryURL()

Description

Query encoded URL.

Prototype

```
int IOT_HTTP_QueryURL(IOT_HTTP_CONTEXT * pSelf,
                      char             * pURL,
                      unsigned           URLLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pURL	Pointer to object that receives the URL.
URLLen	Capacity of the object that receives the URL.

Return value

≥ 0 Success.
< 0 Processing error (buffer too small).

4.1.6.36 IOT_HTTP_Recv()

Description

Receive raw data from connection.

Prototype

```
int IOT_HTTP_Recv(IOT_HTTP_CONTEXT * pSelf,
                  void             * pData,
                  unsigned           DataLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pData	Pointer to object that receives the data.
DataLen	Nonzero octet length of the object that receives the data.

Return value

> 0	Number of octets received.
= 0	Socket closed gracefully.
< 0	Processing error, e.g. socket unexpectedly closed.

4.1.6.37 IOT_HTTP_Reset()

Description

Reset request and clear parameters.

Prototype

```
void IOT_HTTP_Reset(IOT_HTTP_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.

4.1.6.38 IOT_HTTP_Send()

Description

Send raw data to connection.

Prototype

```
int IOT_HTTP_Send(      IOT_HTTP_CONTEXT * pSelf,
                  const void             * pData,
                        unsigned           DataLen);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pData	Pointer to object that contains the data.
DataLen	Nonzero octet length of the object that contains the data.

Return value

> 0	Number of octets sent.
= 0	Socket closed gracefully.
< 0	Processing error, e.g. socket unexpectedly closed.

4.1.6.39 IOT_HTTP_SendStr()

Description

Send zero-terminated string to connection.

Prototype

```
int IOT_HTTP_SendStr(      IOT_HTTP_CONTEXT * pSelf,
                     const char             * sText);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
sText	String to send.

Return value

> 0	Number of octets sent.
= 0	Socket closed gracefully.
< 0	Processing error, e.g. socket unexpectedly closed.

4.1.6.40 IOT_HTTP_SetAuth()

Description

Set authorization method.

Prototype

```
void IOT_HTTP_SetAuth(IOT_HTTP_CONTEXT   * pSelf,
                      IOT_HTTP_AUTH_FUNC * pfWrAuth,
                      void               * pContext);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pfWrAuth	Pointer to function that writes the authorization header.
pContext	Pointer to authorization context passed to pfWrAuth.

4.1.6.41 IOT_HTTP_SetIO()

Description

Set transport I/O.

Prototype

```
void IOT_HTTP_SetIO(      IOT_HTTP_CONTEXT * pSelf,
                    const IOT_IO_API       * pAPI,
                          void             * pContext);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pAPI	Pointer to I/O API.
pContext	Pointer to context passed to I/O API.

4.1.6.42 IOT_HTTP_SetPort()

Description

Set port for request.

Prototype

```
void IOT_HTTP_SetPort(IOT_HTTP_CONTEXT * pSelf,
                      unsigned           Port);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
Port	TCP port in host byte order.

4.1.6.43 IOT_HTTP_SetVersion()

Description

Set HTTP version for request.

Prototype

```
void IOT_HTTP_SetVersion(      IOT_HTTP_CONTEXT     * pSelf,
                         const IOT_HTTP_VERSION_API * pAPI);
```

Parameters

Parameter	Description
pSelf	Pointer to HTTP context.
pAPI	Pointer to HTTP API corresponding to the HTTP version the request uses.

4.2 JSON parser

This section describes the API of the JSON parser.

4.2.1 Data types

Type	Description
IOT_JSON_VOID_EVENT_FUNC	JSON event, no parameter.
IOT_JSON_ARG_EVENT_FUNC	JSON event, string parameter.
IOT_JSON_EVENTS	JSON event handlers.

4.2.1.1 IOT_JSON_VOID_EVENT_FUNC

Description

JSON event, no parameter.

Type definition

```
typedef void (IOT_JSON_VOID_EVENT_FUNC)(IOT_JSON_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to user-supplied context.

Additional information

Function prototype for an event without a parameter.

4.2.1.2 IOT_JSON_ARG_EVENT_FUNC

Description

JSON event, string parameter.

Type definition

```
typedef void (IOT_JSON_ARG_EVENT_FUNC)(      IOT_JSON_CONTEXT * pContext,
                                       const char             * sText);
```

Parameters

Parameter	Description
pContext	Pointer to user-supplied context.
sText	Pointer to zero-terminated string argument.

Additional information

Function prototype for an event with a zero-terminated string parameter.

4.2.1.3 IOT_JSON_EVENTS

Description

JSON event handlers.

Type definition

```
typedef struct {
  IOT_JSON_VOID_EVENT_FUNC * pfBeginObject;
  IOT_JSON_VOID_EVENT_FUNC * pfEndObject;
  IOT_JSON_VOID_EVENT_FUNC * pfBeginArray;
  IOT_JSON_VOID_EVENT_FUNC * pfEndArray;
  IOT_JSON_ARG_EVENT_FUNC  * pfKey;
  IOT_JSON_ARG_EVENT_FUNC  * pfString;
  IOT_JSON_ARG_EVENT_FUNC  * pfNumber;
  IOT_JSON_ARG_EVENT_FUNC  * pfLiteral;
} IOT_JSON_EVENTS;
```

Structure members

Member	Description
pfBeginObject	Start of object event.
pfEndObject	End of object event.
pfBeginArray	Start of array event.
pfEndArray	End of array event.
pfKey	Key id event.
pfString	String value event.
pfNumber	Number value event.
pfLiteral	Literal name event.

4.2.2 Information functions

The table below lists the functions that return JSON Parser information.

Function	Description
`IOT_JSON_GetVersionText()`	Get JSON Parser version as printable string.
`IOT_JSON_GetCopyrightText()`	Get JSON Parser copyright as printable string.

4.2.2.1 IOT_JSON_GetVersionText()

Description

Get JSON Parser version as printable string.

Prototype

```
char *IOT_JSON_GetVersionText(void);
```

Return value

Zero-terminated version string.

4.2.2.2 IOT_JSON_GetCopyrightText()

Description

Get JSON Parser copyright as printable string.

Prototype

```
char *IOT_JSON_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

4.2.3 Functions

Function	Description
IOT_JSON_Init()	Initialize the JSON parser.
IOT_JSON_Parse()	Incrementally parse a JSON stream.
IOT_JSON_SetUserContext()	Set user context.
IOT_JSON_GetUserContext()	Get user context.

4.2.3.1 IOT_JSON_Init()

Description

Initialize the JSON parser.

Prototype

```
void IOT_JSON_Init(      IOT_JSON_CONTEXT * pSelf,
                   const IOT_JSON_EVENTS  * pEvents,
                         char             * pBuf,
                         unsigned           BufLen);
```

Parameters

Parameter	Description
pSelf	Pointer to JSON context.
pEvents	Pointer to handlers for JSON parser events.
pBuf	Pointer to working buffer for token construction.
BufLen	Capacity of token construction buffer.

4.2.3.2 IOT_JSON_Parse()

Description

Incrementally parse a JSON stream.

Prototype

```
int IOT_JSON_Parse(      IOT_JSON_CONTEXT * pSelf,
                   const char             * pBuf,
                         unsigned           BufLen);
```

Parameters

Parameter	Description
pSelf	Pointer to JSON context.
pBuf	Source text to parse.
BufLen	Length of source text to parse.

Return value

= 0 Parse complete without error.
< 0 Parse complete with error (syntax error or token buffer overflow).
> 0 Incomplete parse, call IOT_JSON_Parse() again with additional text.

4.2.3.3 IOT_JSON_SetUserContext()

Description

Set user context.

Prototype

```
void IOT_JSON_SetUserContext(IOT_JSON_CONTEXT * pSelf,
                             void             * pUserContext);
```

Parameters

Parameter	Description
pSelf	Pointer to JSON context.
pUserContext	Pointer to user context.

4.2.3.4 IOT_JSON_GetUserContext()

Description

Get user context.

Prototype

```
void *IOT_JSON_GetUserContext(const IOT_JSON_CONTEXT * pSelf);
```

Parameters

Parameter	Description
pSelf	Pointer to JSON context.

Return value

Pointer to user context, NULL if not set.

Chapter 5

Indexes

5.1 Type index

5.2 Function index

