HTTP Guide

Cross-Domain Development Kit XDK110
Platform for Application Development

Bosch Connected Devices and Solutions

BOSCH
Invented for life

XDK110: HTTP Guide

Document revision	2.0
Document release date	17.08.17
Workbench version	3.0.0
Document number	BCDS-XDK110-GUIDE-HTTP
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. **Subject to change without notice**

XDK HTTP Guide

Platform for Application Development

HTTP is a common protocol to transfer data and files over the network. The XDK supports HTTP natively and offers two modules to make HTTP requests. This guide will provide an introduction to both of them and will demonstrate how to use them to make GET and POST request.

Table of Contents

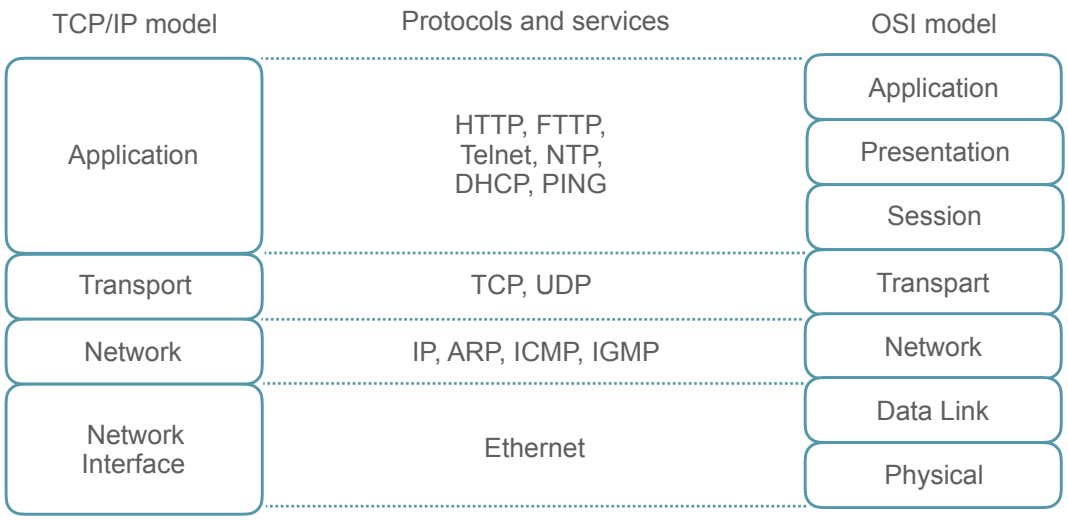
This guide postulates a basic understanding of the XDK and according Workspace. For new users we recommend going through the following guides at xdk.io/guides first:

- *Workbench Installation*
- *Workbench First Steps*
- *XDK Guide FreeRTOS*
- *XDK Guide Wi-Fi*

1. The HTTP Protocol

HTTP (HyperText Transfer Protocol) is a data transfer protocol that runs on top of the TCP/IP protocol stack (see Picture 1). It has become the standard protocol to deliver web pages, making it the foundation of data communication for the World Wide Web. Beyond that, as HTTP is not at all restricted to hypertext, it is commonly used to transfer any kind of structured data between applications. The standard port for HTTP is the port 80.

Picture 1. HTTP in the OSI and TCP/IP stack



This chapter will give a quick introduction on HTTP itself as well as the related REST paradigm. Readers who are already familiar with these topics or who are just looking for the XDK specific implementation details can safely skip to chapter 2.

Note: The XDK uses version 1.1 of the HTTP protocol ("HTTP/1.1"). The complete specification of this version is available in the form of an RFC document: https://tools.ietf.org/html/rfc2616

1.1 The HTTP Request

HTTP defines how messages between two network endpoints should be structured in order to obtain an efficient and reliable data transfer. In every interaction, one network endpoint acts as the **client** and the other as the **server**. The client sends **requests** to a server whenever it needs a certain resource or wants to perform a certain operation that the server offers. The server listens for incoming requests and tries to serve them as they arrive. For every request, the server replies with a **response**. This guide will demonstrate how to use the XDK to send requests. The XDK will therefore act as an **HTTP client**.

Note: The presented modules also offer functionality to setup an HTTP server. This will, however, not be covered in this guide.

A HTTP request consists of two parts. The first part is called the HTTP **header,** which describes the type of request and contains meta data. The second part is called the **payload**, which is the actual data that should be transferred. While HTTP doesn't specify the format or encoding of the payload (it may be HTML, JSON, XML or anything else), the header has to follow the specifications in order to be readable by the receiver. As HTTP is a text-based protocol, the header is a plain (ASCII) string. Its parts are separated by line breaks. Header and payload are separated by two line breaks. A simple request (without payload) could look like Listing 1.

Listing 1. Simple GET request

```
GET /index.html HTTP/1.1
Host: www.bosch.de
Accept: text/html,application/json
Accept-Language: en-us
Accept-Encoding: gzip, deflate
```

The only required lines in this request are the first two, which specify the protocol version, the HTTP **method** (“GET”) and the requested resource (“www.bosch.de/index.html”). Lines 3 to 5 are called “Headers” and provide additional information in the form of key-value pairs. A response to the request in Listing 1 could look like Listing 2 if the resource was found, or like Listing 3 if not.

Listing 2. Succesfull GET Response

```
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 03 Oct 2016 21:16:16 GMT
Last-Modified: Thu, 01 Sep 2016 10:15:00 GMT
Content-Length: 88
Content-Type: text/html

[HTML page]
```

Listing 3. Failed GET response

```
HTTP/1.1 404 Not Found
Server: nginx
Date: Mon, 03 Oct 2016 21:16:16 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 234

[HTML error page]
```

The number in the first line of the response (after “HTTP/1.1”) is called the HTTP **status code**. It tells the client whether the request succeeded (2XX) or failed (4XX). The numeric code is followed by its corresponding textual message. A complete list of HTTP status codes and can be found at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes or in section 6.1.1 and 10 of the RFC.

An important header line in Listing 2 and 3 is the “Content-Type” header. It specifies the expected media type[1] of which the client expects the requested resource to be. If a server can’t provide the

[1] As defined in the official media type list maintained by the Internet Assigned Numbers Authority: http://www.iana.org/assignments/media-types/media-types.xhtml

resource in this format, it responds with an error.

Listing 1-3 have been examples for a GET request, which is the most common HTTP method and is used to request (i.e. download) a resource. If a client wants to send (i.e. upload) data to the server, he can use the POST method. Listing 4 shows a POST request.

Listing 4. POST request

```
POST /create.php HTTP/1.1
Host: 192.168.1.2:80
Content-Type: text/plain
Transfer-Encoding: chunked

[encoded data to upload]
```

The “Transfer-Encoding” header defines the encoding of the attached payload. The server needs this information to be able to parse the payload correctly. HTTP supports different encoding types, most of them indicating the compression algorithm that was used on the payload. The most simple encoding is called “chunked”, which is an uncompressed encoding and is described as follows:

“The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator […]. This allows dynamically produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.” - RFC 2616

The XDK modules presented in this guide use this as their default encoding for POST requests. For a reference of the available encodings, please refer to https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding or section 3.6 of the RFC.

This guide will explain how to perform GET and POST requests in an XDK application. A complete list of all specified HTTP methods can be found at section 5.1.1. of the RFC.

1.2 The REST paradigm

Representational State Transfer (REST) is a paradigm that tries to define a uniform interface for web services to improve interoperability. REST-compliant (or “RESTful”) services usually only use the HTTP methods POST, GET, PUT and DELETE, as these methods are directly mappable to the four basic functions of persistent storage: create, read, update and delete (CRUD). They also follow a common URL scheme to access and manipulate resources (see Table 1). Response data is provided in XML, HTML, JSON or some other defined format.

Table 1. REST operations

URL	POST	GET	PUT	DELETE
http://api.example.com/collection/	Create new element	List all elements	Replace collection	Delete the collection
http://api.example.com/collection/element12	-	Retrieve element	Update element	Delete element

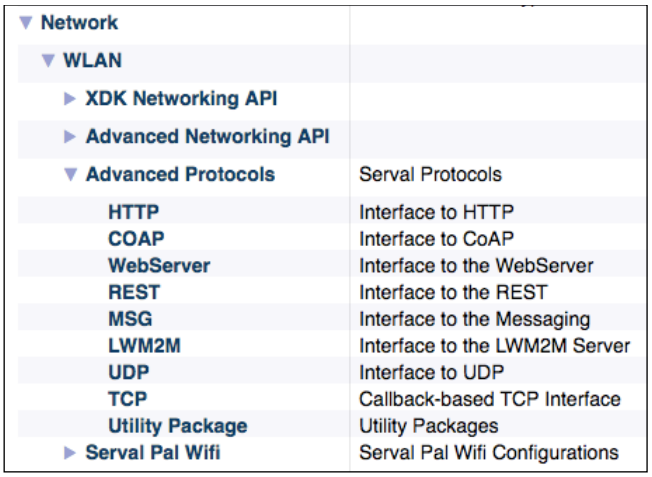
By definition, RESTful services may not store any kind of client context between requests. This means that each request has to provide all the information necessary to serve the request, and that the session state has to be handled by the client.

HTTP 1.1 was designed in compliance with the REST principles.

2. API Overview

The XDK comes with an SDK that offers a wide variety of libraries and tools to application developers. The part of the SDK that covers networking protocols is called the **Serval** stack. It contains modules for a range of application layer protocols that are used on embedded systems (see Picture 2).

Picture 2. Advanced network protocols supported by the Serval stack



Note: A complete overview of all the modules included in the SDK can be found at http://xdk.bosch-connectivity.com/xdk_docs/html/modules.html

Relevant for this guide are two modules of the Serval stack:

For basic HTTP request and response handling, the Serval stack offers the **HTTP API**.

As many HTTP based web services follow the REST guidelines, the Serval stack also offers the higher-level **REST API** to consume these services in a more straight forward way.

While the implementation with the REST API is simpler, the HTTP API offers more flexibility. It is also less memory consuming and more efficient than the REST API.

Both APIs are subdivided into three sub-modules with own header files:

Table 2. Serval submodule structure

Submodule	HTTP API Header	REST API Header
Client submodule	Serval_HttpClient.h	Serval_RestClient.h
Server submodule	Serval_HttpServer.h	Serval_RestServer.h
Shared submodule	Serval_Http.h	Serval_Rest.h

The shared submodule contains types, constants, functions, etc. that are used by both other modules. Therefore, this submodule is automatically included by the client and server module.

This guide will demonstrate how to perform GET and POST requests with both the HTTP and the REST API.

3. Network setup

To perform a network request, a client needs a working network connection to the server.
The easiest way to achieve this is to connect the XDK to the local network via its Wi-Fi interface. After setting up this link, the XDK can send requests to a locally hosted server, or, if the access point is connected to the internet, to any server on the World Wide Web. However, establishing a Wi-Fi connection requires some configuration.

For the purpose of this guide, the **HttpExampleClient** project, which is available as an example project in the XDK-Workbench, is a good starting point. It already contains code to connect to a Wi-Fi network using the WiFi and the PAL module and also showcases an HTTP request using the Serval HTTP client.

After providing the access point SSID and the password in the header file, the demo can be compiled and flashed to the XDK. When executed, the application tries to perform a GET request to a predefined server and logs the response or any error.

If the HttpExampleClient doesn't work out of the box, or a special network setup is required, please refer to the XDK Wi-Fi guide available at xdk.io/guides for further instructions.

For just the minimal setup required to establish a Wi-Fi connection, an `appInitSystem()` function similar to Code 1 can be used.

Note: The code snippets in this guide contain almost no error handling code in order to keep them short and simple. For real-world projects however, return code validation and error handling are substantial. Information on possible return and error codes can be found in the API documentation: http://xdk.bosch-connectivity.com/xdk_docs/html/modules.html.

Code 1. Minimal Wi-Fi setup

```
#include "BCDS_WlanConnect.h"
#include "BCDS_NetworkConfig.h"
#include "PAL_initialize_ih.h"
#include "PAL_socketMonitor_ih.h"
#include "PIp.h"

void appInitSystem(void * CmdProcessorHandle, uint32_t param2){
    if (CmdProcessorHandle == NULL){
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    BCDS_UNUSED(param2);
    WlanConnect_SSID_T connectSSID = (WlanConnect_SSID_T) "SSID";
    WlanConnect_PassPhrase_T connectPassPhrase = (WlanConnect_PassPhrase_T)
"PW";
    WlanConnect_Init();
    NetworkConfig_SetIpDhcp(0);
    WlanConnect_WPA(connectSSID, connectPassPhrase, NULL);

    PAL_initialize();
    PAL_socketMonitorInit();

    // ... initialize chosen HTTP module
    // ... start timers to perform network requests
}
```

4. Serval HTTP API

This chapter demonstrates how to use the Serval HTTP client module to perform simple HTTP requests.

4.1 Setup

To use the HTTP client in an XDK application, its header file needs to be included in the source file:

Code 2. Include the HTTP client module

```
#include <Serval_HttpClient.h>
```

Before using the module for the first time, it also needs to be initialized by calling the function `HttpClient_initialize()`.

4.2 Performing a GET request

After including the HTTP client module, it can be used to create and send requests. Code 3 will send a GET request for the resource “23.22.14.18:80/ip”.

Note: This IP address belongs to the website httpbin.org. The site offers various simple HTTP endpoints that can be used to test requests and response handling. The HttpExampleClient example project contains code to resolve a host name to an IP address using the PAL module.

Code 3. GET request using the HTTP API

```
// Insert this code beneath the simple WiFi setup in appInitSystem()
// assemble the request message
Ip_Address_T destAddr;
Ip_convertOctetsToAddr(23, 22, 14, 18, &destAddr);
Ip_Port_T port = Ip_convertIntToPort(80);

Msg_T* msg_ptr;
HttpClient_initRequest(&destAddr, port, &msg_ptr);
HttpMsg_setReqMethod(msg_ptr, Http_Method_Get);
HttpMsg_setReqUrl(msg_ptr, "/ip");

// send the request

static Callable_T sentCallable;
Callable_assign(&sentCallable, &onHTTPRequestSent);

// insert additional headers here, if needed (see chapter 4.4)

HttpClient_pushRequest(msg_ptr, &sentCallable, &onHTTPResponseReceived);
```

The central object here is the struct `Msg_T`. It contains all the information required to create HTTP request. The struct is created using `HttpClient_initRequest()`. Parameters are the address and port of the server that should receive the request. This call will trigger an assertion if no network connections is available.

`HttpMsg_setReqMethod()` is used to set the HTTP method of the request, in this case `Http_Method_Get`.

The last piece of information that needs to be set is the resource that we want to request (compare Listing 1). This can be done with the function `HttpMsg_setReqUrl()` that takes a simple c string as argument.

A network request is an operation that takes a previously unknown amount of time to finish. There are two ways to handle these operations. An operation that is executed **synchronously** will block its thread until the operation is finished. However, this is an undesired behaviour in a real-time environment like the XDK. The other option is to perform an operation **asynchronously**, using callback functions that are called when the operation finished. This way, execution can continue immediately after initiating the operation.

The `HttpClient_pushRequest()` function, that is used to send a the HTTP request, takes two callback functions:

The first function (second parameter) is called when the request was successfully sent, or if an error occurred that prevented the system from sending the request.

The second function (third parameter) is called when the system received a response.

Note that the the first callback function has to be wrapped into a `Callable_T` (defined in the Serval Callable module), while the second callback function is passed simply as a function pointer.

To validate the content type of the response, predefined content type string constants (like `Http_ContentType_Text_Html` or `Http_ContentType_App_Json`) can be used. These constants are defined in the shared submodule that is included by the HTTP client module (see chapter 2).

Note: Example code for GET requests can also be found in the HttpExampleClient demo. To avoid errors on response validation, the expected content type has to be adjusted to the one that is stated in the HTTP response.

Code 4 and 5 provide sample implementations of both required callback functions.

Code 4. HTTP request sent callback

```
static retcode_t onHTTPRequestSent(Callable_T *callfunc, retcode_t status)
{
	(void) (callfunc);

	if (status != RC_OK) {
		printf("Failed to send HTTP request!\r\n");
	}
	return(RC_OK);
}
```

The function from Code 4 simply validates the status code and prints an error message if something went wrong.

Code 5. HTTP response received callback

```
static retcode_t onHTTPResponseReceived(HttpSession_T *httpSession,
              Msg_T *msg_ptr, retcode_t status)
{
       (void) (httpSession);

       if (status == RC_OK && msg_ptr != NULL) {

              Http_StatusCode_T statusCode = HttpMsg_getStatusCode(msg_ptr);
              char const *contentType = HttpMsg_getContentType(msg_ptr);

              char const *content_ptr;
              unsigned int contentLength = 0;
              HttpMsg_getContent(msg_ptr, &content_ptr, &contentLength);
              char content[contentLength+1];
              strncpy(content, content_ptr, contentLength);
              content[contentLength] = 0;

              printf("HTTP RESPONSE: %d [%s]\r\n", statusCode, contentType);
              printf("%s\r\n", content);

       } else {

              printf("Failed to receive HTTP response!\r\n");
       }
       return(RC_OK);
}
```

If no error occurred and a response was received, the function from Code 5 will print the responses status code, content type and payload to the XDK Workbench console.

The `Msg_T` parameter that is passed to the response callback function is the same struct that was initially passed to the `HttpClient_pushRequest()` function, only now containing the HTTP response data as well. The individual parts of the received response can be extracted using the functions `HttpMsg_getStatusCode()`, `HttpMsg_getContentType()` and `HttpMsg_getContent()`, with a pointer to the struct as argument. Since the response payload (if existing) is stored as a plain byte buffer in memory, `HttpMsg_getContent()` cannot simply return a reference to this buffer, but has to provide its length too. This is done using additional out parameters.

To avoid formatting errors, the content of the buffer is then cast into a zero-terminated C string for printing.

4.3 Performing a POST request

Code 6 will create and send a POST request to “23.22.14.18:80/post”.

Code 6. POST request using the HTTP API

```
// Insert this code below the simple WiFi setup in appInitSystem()
// assemble the request message
Ip_Address_T destAddr;
Ip_convertOctetsToAddr(23, 22, 14, 18, &destAddr);
Ip_Port_T port = Ip_convertIntToPort(80);

Msg_T* msg_ptr;
HttpClient_initRequest(&destAddr, port, &msg_ptr);
HttpMsg_setReqMethod(msg_ptr, Http_Method_Post);
HttpMsg_setReqUrl(msg_ptr, "/post");

Msg_prependPartFactory(msg_ptr, &writeNextPartToBuffer);

// send the request

static Callable_T sentCallable;
Callable_assign(&sentCallable, &onHTTPRequestSent);

// insert additional headers here, if needed (see chapter 4.4)

HttpClient_pushRequest(msg_ptr, &sentCallable, &onHTTPResponseReceived);
```

The code for the POST request looks very similar to the code for the GET request (Code 2). There are only two differences:

1. The method is set to `Http_Method_Post`.

2. `Msg_prependPartFactory()` is used to pass a function pointer in the `Msg_T` struct.

While the first difference should be self-explanatory, the second one needs some explanation.

As mentioned in section 1.1, the HTTP API uses chunked transfer encoding for POST requests. The HTTP stack sends a new chunk whenever data is available. The pointer that is passed here to the message struct is a pointer to a functions that generates exactly these chunks. This function is called a **part factory**. It sequentially writes the payload into a handover buffer which then gets encoded and transferred to the server.

Code 7 provides a reference implementation for a part factory function that can be used for any previously known payload. On every execution, it copies a chunk of the payload data into a provided handover data structure (`OutMsgSerializationHandover_T`). This structure, being is used as an in-and-out parameter, also contains the maximum length for the next chunk as well as the current offset within the payload buffer. This state information is needed in order to calculate the next chunk.

The return value of the part factory function is used to indicate whether the end of the payload is reached. If there is still payload data left which was not serialized yet (because the available buffer was not big enough), the function uses the return code `RC_MSG_FACTORY_INCOMPLETE`. This will

cause the network stack to call the part factory again as soon as new buffer space is available. If the handed chunk is the last one for this request, the part factory function returns `RC_OK`.

Code 7. Universal part factory function

```
retcode_t writeNextPartToBuffer(OutMsgSerializationHandover_T* handover)
{
	const char* payload = "Hello";

	uint16_t payloadLength = (uint16_t) strlen(payload);
	uint16_t alreadySerialized = handover->offset;
	uint16_t remainingLength = payloadLength - alreadySerialized;
	uint16_t bytesToCopy;

	retcode_t rc;

	if ( remainingLength <= handover->bufLen ) {

		bytesToCopy = remainingLength;
		rc = RC_OK;

	} else {

		bytesToCopy = handover->bufLen;
		rc = RC_MSG_FACTORY_INCOMPLETE;
	}

	memcpy(handover->buf_ptr, payload + alreadySerialized, bytesToCopy);
	handover->offset = alreadySerialized + bytesToCopy;
	handover->len = bytesToCopy;

	return rc;
}
```

As all HTTP requests are sent using the same `HttpClient_pushRequest()` function, the sample callback functions from Code 4 and 5 can be used for the POST request as well.

4.4 Headers

When it comes to HTTP messages, the most important and flexible aspect are the headers of a message. They tell the recipient of a message how to interpret the message that has been received. Some such headers are “HOST”, “CONTENT-TYPE” or “CONTENT-LENGTH”. There are standardized headers, which are listed in the RFC 4229 document. It is important to use them in a way that aligns with how they are semantically described in that RFC document, otherwise servers may behave differently than expected.

The general format of an HTTP message is as follows:

start-line
header-fields

message-body

The start-line indicates what kind of message it is. A message is either a request or a response. In any case, the start-line is set accordingly by the XDK's HTTP API, when everything is set correctly. After the start-line, the header-fields are listed, and only one line per header-field should be used.

The header-fields are strictly separated from the message-body by an empty line. This is important, because otherwise the message-body might be interpreted as header-fields, which would most likely result in an unexpected response, or a standard "Bad Request" error.

A header-field consists of a field-name and a field-value, which are separated by a colon as follows.

field-name: field-value.

Note that whitespace is allowed after the colon, but not before the colon. For example: "CONTENT-LENGTH: 20" is allowed, whereas "CONTENT-LENGTH : 20" is not.

These are the most basic things to know about the HTTP message format and headers. As mentioned before, headers provide information regarding the interpretation of HTTP message by an application. With that in mind, the following chapter will give some insight on how to set some specific standardized headers and also how to generally create custom, non-standardized headers.

Before trying to apply this guide's code to your own application, make sure that XDK-Workbench is up-to-date, because the essential API used here is only available at Workbench version 3.0 or higher.

4.4.1 Set Host Header

The Host header is used to differentiate between multiple domains that run on a common IP. In this way, a single web server can receive requests that are addressed to different domains, and forward them to the corresponding virtual servers for processing.

The following code will set the host-header, using the Http API.

Code 8. Setting Host Header

```
HttpMsg_setHost(msg_ptr, "your.host.address");
```

This can be directly copied to your sending-implementation and has to be inserted called `HttpClient_pushRequest()` and after `HttpClient_initRequest()`.

4.4.2 Set Range Header

The Range header is used most commonly in GET requests. It indicates which parts of a file should be sent in a response. The structure of the RANGE header is most commonly used as follows.

Range: bytes=startByte-endByte
Range: bytes=startByte-
Range: bytes=start1-end1, start2-end2, start3-

The first example will return all bytes from startByte to endByte. The second example will return all bytes from startByte to the end of the addressed file. The third example shows how to use multiple ranges, combining the previous two examples.

Unfortunately, the current implementation of the function `HttpMsg_setRange()` only allows the first option, with a definite start and ending. The following code shows how to set the range header, using the Http API.

Code 9. Setting Range Header

```
HttpMsg_setRange(msg_ptr, 4, 16); // resolves to Range: 4-19
```

This can be directly copied to your sending-implementation and has to be inserted before `HttpClient_pushRequest()` and after `HttpClient_initRequest()`.

4.4.3 Set Other/Custom Headers

Custom Headers are used to provide additional information and semantics to a message, that can not be done with standardized headers. As the term “custom” indicates, they are not official, non-standardized headers. Names of custom headers can therefore be arbitrary, but in general, the names should be chosen as though they would become standardized headers. Also, it should be avoided to use names of standardized headers and use them in a different way than intended.

As such, Custom Headers should only be used when absolutely necessary, since only servers that know that particular header will be able to process the header and consequently the message.

The Serval_Http API provides the function called `HttpMsg_serializeCustomHeaders()`. While the name suggests, that only custom headers may be serialized with that function, it can also serialize other custom headers, which are not directly supported by the API. Keep in mind, that this function will not overwrite already existing headers. Consequently, serializing a header twice might lead to unexpected behaviour on the receiving endpoint.

Implementing custom headers requires two things. First, the following function has to be called.

Code 10. Setting Custom Headers

```
HttpMsg_serializeCustomHeaders(msg_ptr, serializeMyHeaders);
```

This can be directly copied to your sending-implementation and has to be called before `HttpClient_pushRequest()` and after `HttpClient_initRequest()`. This involves a new function called `serializeMyHeaders()`, which has to be implemented yet. It will include the actual custom headers, which will be serialized.

Assuming that the headers “MyHeader: Value 1” and “MyOtherHeader: Value 2” are to be serialized, the function `serializeMyHeaders()` will look as follows:

Code 11. Serializing Custom Headers in a Callback

```
retcode_t serializeMyHeaders(OutMsgSerializationHandover_T* handover)
{
	handover->len = 0;
	static const char* header1 = "MyHeader: Value 1\r\n";
	static const char* header2 = "MyOtherHeader: Value 2\r\n";
	TcpMsg_copyStaticContent(handover, header1, strlen(header1));
	handover->position = 1;
	TcpMsg_copyContentAtomic(handover, header2, strlen(header2));
	handover->position = 2;

	return RC_OK;
}
```

There are a few things to be aware of here. The strings have to end with “\r\n”, because the message format, according to the RFC, demands that every header-field ends with a carriage return and a new line. If this is not done, both headers would be written on the same line, which is not the desired effect. Additionally, adding too many “\r\n” will lead to unexpected behaviour, as well. Additionally, while error-handling is not included in the code snippet, it is important to react to the return codes, which are returned by the copying-functions `TcpMsg_copyStaticContent()` or `TcpMsg_copyContentAtomic()`. Also, the difference between these two functions is the fact that `TcpMsg_copyContentAtomic()` will only copy the content if there is enough space for the entire content, otherwise it will not. `TcpMsg_copyStaticContent()` will try to copy the content anyway.

5. Serval REST API

This chapter demonstrates how to use the Serval REST client module to perform simple HTTP requests to REST services.

5.1 Setup

To use the REST client in an XDK application, its header file needs to be included in the source file:

Code 12. Include the REST client module

```
#include <Serval_RestClient.h>
```

Before using the module for the first time, it needs to be initialized by calling the function `RestClient_initialize()`.

5.2 Performing a GET request

After including the REST client module, it can be used to create and send REST requests. Code 913will send a GET request to the REST endpoint “23.22.14.18:80/ip”.

Code 13. GET request using the REST API

```
// Insert this code beneath the simple WiFi setup in appInitSystem()
// assemble the request message
Ip_Address_T destAddr;
Ip_convertOctetsToAddr(23, 22, 14, 18, &destAddr);
Ip_Port_T port = Ip_convertIntToPort(80);

Rest_ContentFormat_T acceptedFormats[1] = { REST_CONTENT_FMT_JSON };
RestClient_ReqInfo_T getRequest;
getRequest.method = REST_GET;
getRequest.uriPath_ptr = "/ip";
getRequest.uriQuery_ptr = "";
getRequest.acceptBuffer_ptr = acceptedFormats;
getRequest.numAccept = 1;
getRequest.payloadLen = 0;
getRequest.rangeLength = 0;
getRequest.host = NULL;
Msg_T* msg_ptr;
RestClient_initReqMsg(&destAddr, port, &msg_ptr, REST_CLIENT_PROTOCOL_HTTP);
RestClient_fillReqMsg(msg_ptr, &getRequest);

// send the request
RestClient_request(msg_ptr, &onRESTRequestSent, &onRESTResponseReceived);
```

The REST API uses the `RestClient_ReqInfo_T` struct as its central data structure. It stores all information required to assemble the REST-compliant HTTP request. Table 3 lists all available field of this data structure.

Table 3. RestClient_ReqInfo_T fields

Field	Description	Req.
`method`	HTTP method	yes
`uriPath_ptr`	The path that identifies the resource (after the host name / IP)	yes
`uriQuery_ptr`	The query parameter string of the URI (or an empty string)	yes
`acceptBuffer_ptr`	Array of content types that will be accepted as response	yes
`numAccept`	Length of the `acceptBuffer_ptr` array	yes
`payload_ptr`	Buffer with the payload	no
`payloadLen`	Length of the `payload_ptr` buffer	yes
`contentFormat`	Content type of the data in the payload buffer (XML, JSON, …)	no
`msgReliabilityMode`	Ignored for `REST_CLIENT_PROTOCOL_HTTP`	no
`rangeLength`	See chapter 4.4 (Headers), set to 0 to ignore this field	yes
`rangeOffset`	See chapter 4.4 (Headers), relevant if `rangeLength` is > 0	no
`host`	Alternative Host, set to NULL to ignore this field	yes

Just like the `HttpClient_pushRequest()` function that was used to send request using the HTTP API, the REST API function to send the request, `RestClient_request()`, takes two callback parameters.

Code 14 and 15 are sample implementations for the callback functions. The provided code is exactly the same as in the callbacks provided for the HTTP API, only with different parameter lists. With the REST API, it is not needed to wrap a callback into a `Callable_T` object.

Code 14. REST request sent callback

```
static void onRESTRequestSent(Msg_T *msg_ptr, retcode_t status)
{
	(void) msg_ptr;

	if (status != RC_OK) {
		printf("Failed to send REST request!\r\n");
	}
}
```

Code 15. REST response received callback

```
static retcode_t onRESTResponseReceived(RestSession_T *restSession,
        Msg_T *msg_ptr, retcode_t status)
{
       (void) restSession;

       if (status == RC_OK && msg_ptr != NULL) {

              Http_StatusCode_T statusCode = HttpMsg_getStatusCode(msg_ptr);
              char const *contentType = HttpMsg_getContentType(msg_ptr);

              char const *content_ptr;
              unsigned int contentLength = 0;
              HttpMsg_getContent(msg_ptr, &content_ptr, &contentLength);
              char content[contentLength+1];
              strncpy(content, content_ptr, contentLength);
              content[contentLength] = 0;

              printf("HTTP RESPONSE: %d [%s]\r\n", statusCode, contentType);
              printf("%s\r\n", content);

       } else {

              printf("Failed to receive REST response!\r\n");
       }
       return(RC_OK);
}
```

5.3 Performing a POST request

Code 16 will create and send a POST request with JSON data to the REST endpoint “23.22.14.18:80/post”.

Code 16. POST request using the REST API

```
// Insert this code beneath the simple WiFi setup in appInitSystem()
// assemble the request message
Ip_Address_T destAddr;
Ip_convertOctetsToAddr(23, 22, 14, 18, &destAddr);
Ip_Port_T port = Ip_convertIntToPort(80);

Rest_ContentFormat_T acceptedFormats[1] = { REST_CONTENT_FMT_JSON };
RestClient_ReqInfo_T postRequest;
postRequest.method = REST_POST;
postRequest.uriPath_ptr = "/post";
postRequest.uriQuery_ptr = "";
postRequest.acceptBuffer_ptr = acceptedFormats;
postRequest.numAccept = 1;
postRequest.payloadLen = 34;
postRequest.payload_ptr = (uint8_t*) "{\"num\": 123, \"str\": \"Hello\"}";
postRequest.contentFormat = REST_CONTENT_FMT_JSON;
postRequest.rangeLength = 0;
postRequest.host = NULL;

Msg_T* msg_ptr;
RestClient_initReqMsg(&destAddr, port, &msg_ptr, REST_CLIENT_PROTOCOL_HTTP);
RestClient_fillReqMsg(msg_ptr, &postRequest);

// send the request
RestClient_request(msg_ptr, &onRESTRequestSent, &onRESTResponseReceived);
```

The only difference between the GET and the POST request of the REST API is the content of the `RestClient_ReqInfo_T` struct. The `method` is now `REST_POST`, the `payload_ptr` points to a buffer and `payloadLen` and `contentFormat` are set accordingly.

Note that the REST API doesn’t need a part factory. The payload is simply put into a buffer which is passed to the `RestClient_ReqInfo_T` struct by its pointer. The serialization is done by the REST client.

As all REST requests are sent using the same `RestClient_request()` function, the sample callback functions from Code 14 and 15 can be used for the POST request as well.

6. Document History and Modification

Rev. No.	Chapter	Description of modification/changes	Editor	Date
2.0		Version 2.0 initial release	AFS	2017-08-17