ST87MXX HTTP(S) Application Note

Purpose and scope

This document gives details of the AT Commands usage for embedded HTTP and HTTPS stacks supported by the ST87MXX NB-IoT module.

Document status
Official

1 General information

1.1 Acronyms and terms

Table 1. Definitions of terms

Term	Definition
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
ME	Mobile Equipment
MSC	Message Sequence Chart
TCP	Transport Control Protocol
TLS	Transport Security Layer
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
URC	Unsolicited Result Code

1.2 Reference documents

The documents listed in Table 2 provide further information.

Table 2. Document references

Reference	Document
[1]	3GPP TS 24.301 V15.8.0
[2]	3GPP TS 24.008 V15.9.0
[3]	3GPP TS 23.682 V15.10.0
[4]	3GPP TS 23.401 V15.12.0
[5]	3GPP TS 23.682 V15.10.0
[6]	3GPP TS 36.133 V15.15.0
[7]	3GPP TS 31.101 V15.3.0
[8]	3GPP TS 36.106 V12.1.0
[9]	3GPP TS 27.060 V15.0.0
[10]	3GPP TS 29.061 V15.7.0
[11]	3GPP TS 23.203 V15.4.0
[12]	3GPP TS 27.007 V15.3.0
[13]	3GPP TS 23.060 V15.4.0
[14]	3GPP TC 27.005 V10.0.0
[15]	ST87MXX UM AT commands description
[16]	ST87MXX TLS Application Note

1.3 Revision history

Table 3. Document revision history

Date	Version	Changes
2024-10-01	V1.0	Official release
2025-08-29	V2.0	Update for release

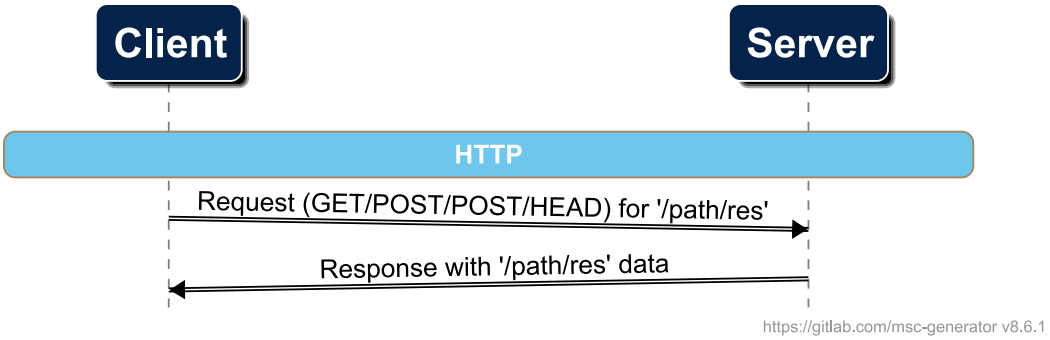
1.4 Table of contents

1.5 List of tables

2 Introduction

2.1 HTTP overview

The HTTP protocol is a protocol that allows the user to fetch or post resources over the network (such as files). It is the foundation of any data exchange on the Web, and it operates on a client-server model, which means requests are initiated by the recipient of the resource (in case of a fetch request).



HTTP: Request-response mechanism

Every request has a type, the method, and contains the path to the target resource (e.g. /path/to/file.txt).

The main HTTP methods are:

- GET, used to fetch/download resources/files
- POST, used to publish/upload resources/files or for application-specific purposes
- PUSH, used to publish/upload resources/files
- HEAD, used to get only the response header.

Each HTTP message (request or response) is composed of:

- A header, which contains information about the sender and HTTP exchange (e.g. the type of payload data, a.k.a. Content-Type). The header can contain multiple header fields (each header field represents a piece of information).
- A body, which contains the payload data.

HTTP is based on the TCP protocol; therefore, a TCP link must be established to exchange the data with the server. Usually, A new TCP connection is opened for each HTTP request and closed after the reception of the response, but it is also possible to keep the connection open (useful in case multiple exchanges must be done in a row.

2.2 AT commands

For the complete description of the AT commands and parameters supported by the ST87MXX, please refer to [15].

3 HTTP over TCP socket Use Case

3.1 Preamble

For HTTP connection and packets exchange, the ME shall configure the IP layer if the Auto-IP configuration is not set by default. See Chapter 4: of TCP/UDP/IP application note.

The following paragraphs briefly explain the AT commands necessary to make use of the HTTP protocol stack.

It is assumed that the user who wants to test this HTTP functionality uses a HTTP server running correctly on the Internet network.

3.2 Create TCP socket

To exchange data over HTTPS, a TCP socket is needed.
The ME can create a new TCP socket with the following at command:

AT#SOCKETCREATE=5,0,TCP,443,30,30,1

5 is the context ID used for IP connection: 5 it is the one used by default by the ST87MXX platform;
0 indicates the use of IPv4, **1** indicates IPv6. It should be the same chosen during IP configuration;
TCP is the type of socket;
443 is the local TCP port;
30 is the timeout set in seconds to send the request to the server;
30 is the timeout set in seconds to receive the response from the server;
1 enables the URC reception when HTTP data coming back from the network is available and can be read with **AT#HTTPREAD**.

The platform should respond with:

#SOCKETCREATE: 0
OK

This indicates that a new socket has been created with an ID equal to 0.

3.3 Connect TCP socket

After the creation of the socket, the ME can connect the TCP socket to the remote server

AT#TCPCONNECT=5,0,<server address>,<server TCP port>

5 is the context ID used for IP connection: 5 it is the one used by default by the ST87MXX platform;
0 is the Socket ID received during socket creation.

As **<server address>**, either the server IP address or domain name (e.g. example.com) can be used. If the domain name is used, the internal DNS service of the ST87MXX will resolve the IP address to use for the connection.

PS : If you do not have any HTTP server to test the TCP connections, you can use any HTTP test server on internet, for instance http://httpbin.org/anything (it accepts all HTTP requests and responds with the same data sent to it).

3.4 Disconnect TCP socket

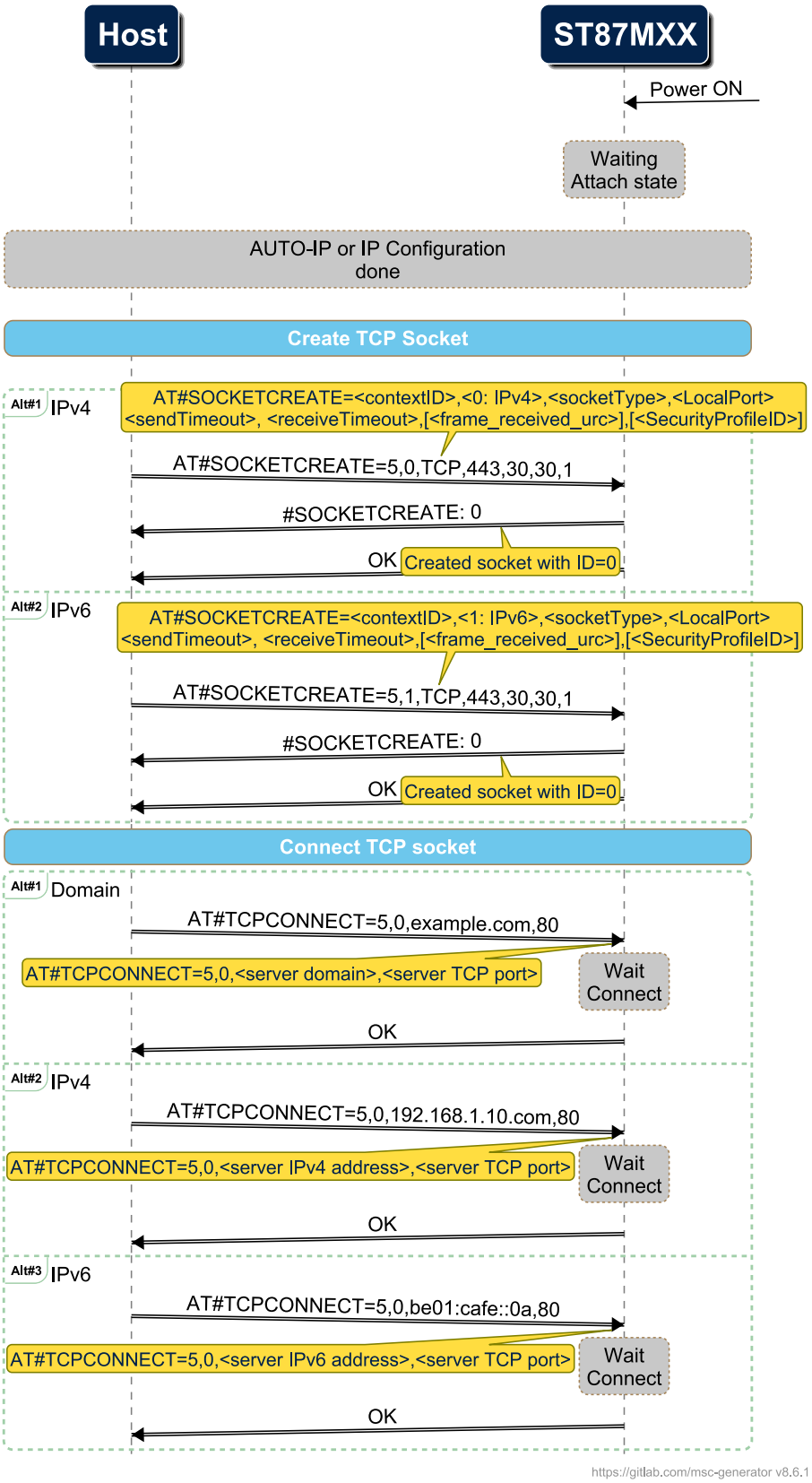
When the connection is still alive, but it needs to be closed, the ME can disconnect the TCP socket from the remote server.

AT#SOCKETCLOSE=5,0

5 is the context ID used for IP connection: 5 it is the one used by default by the ST87MXX platform.
0 is the Socket ID received during socket creation.

3.5 Example

3.5.1 MSC



3.5.2 Terminal

Below is the sequence obtained in the UART terminal when executing the scenario (in **Bold** the commands sent to the module). In the example, <server_address> is set to “example.com” and <server TCP ports> is set to 80.

```
AT#SOCKETCREATE=5,0,TCP,443,30,30,1
#SOCKETCREATE: 0

OK
AT#TCPCONNECT=5,0,example.com,80
OK
```

Alternative:

```
AT#SOCKETCREATE=5,1,TCP,443,30,30,1
#SOCKETCREATE: 0

OK
AT#TCPCONNECT=5,0,example.com,80
OK
```

4 HTTP START/STOP

4.1 HTTP START

To enable the use of the HTTP layer, the host can send the HTTP start command to start the HTTP protocol stack. This can be done before or after creating and connecting the TCP socket.

The ME can start the HTTP stack like this:

AT#HTTPSTART

OK

4.2 HTTP STOP

When the user does not want any more to use HTTP stack, he can use the HTTP stop command to release the HTTP context.

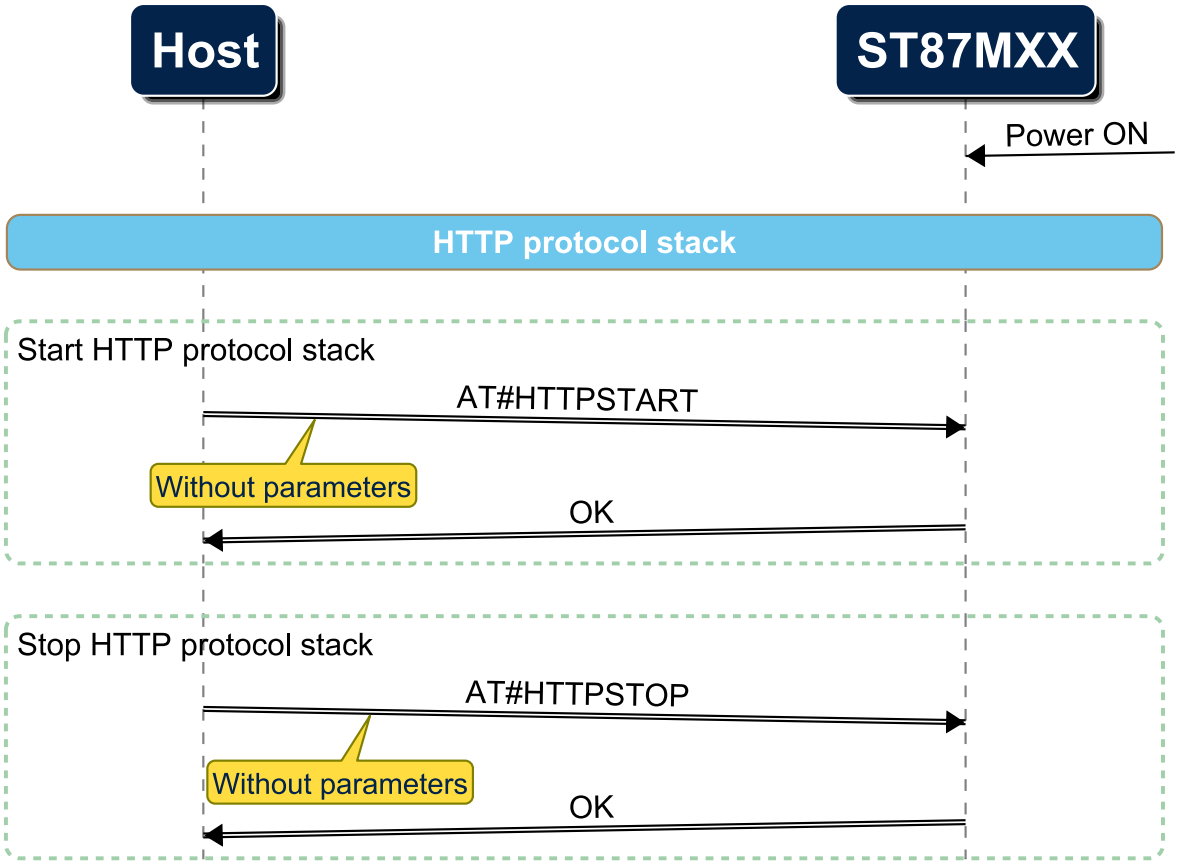
The ME can stop the HTTP stack like this:

AT#HTTPSTOP

OK

4.3 Example

4.3.1 MSC



https://gitlab.com/msc-generator v8.6.1

4.3.2 Terminal

Below is the sequence obtained in the UART terminal when executing the scenario (in **Bold** the commands sent to the module):

```
AT#HTTPSTART
OK

AT#HTTPSTOP
OK
```

5 HTTP METHODS (PUT, GET, POST, HEAD)

5.1 Preamble

When The TCP socket is created and connected, and HTTP layer has been started, the Host can now exchange data with the HTTP server.
Some methods are available to receive or send data to or from an HTTP server.
The main mechanism is almost the same for each method, and you can resume the requests with the following AT commands:

AT#HTTPMETHOD= …. Selects the HTTP method (POST, GET, …) with server address and path;
AT#HTTPHEADER= … Adds a header to the HTTP method (optional);
AT#HTTPBODY= … Adds data to the body of the HTTP method (optional, useful for long PUT/POST requests);
AT#HTTPSEND= … Sends the HTTP method to the server. Body data can be specified here optionally.

After the server receives the request, it should respond to the device.
If enabled, a URC notification is sent to the host to inform it that data can be read from the HTTP FIFO buffer.

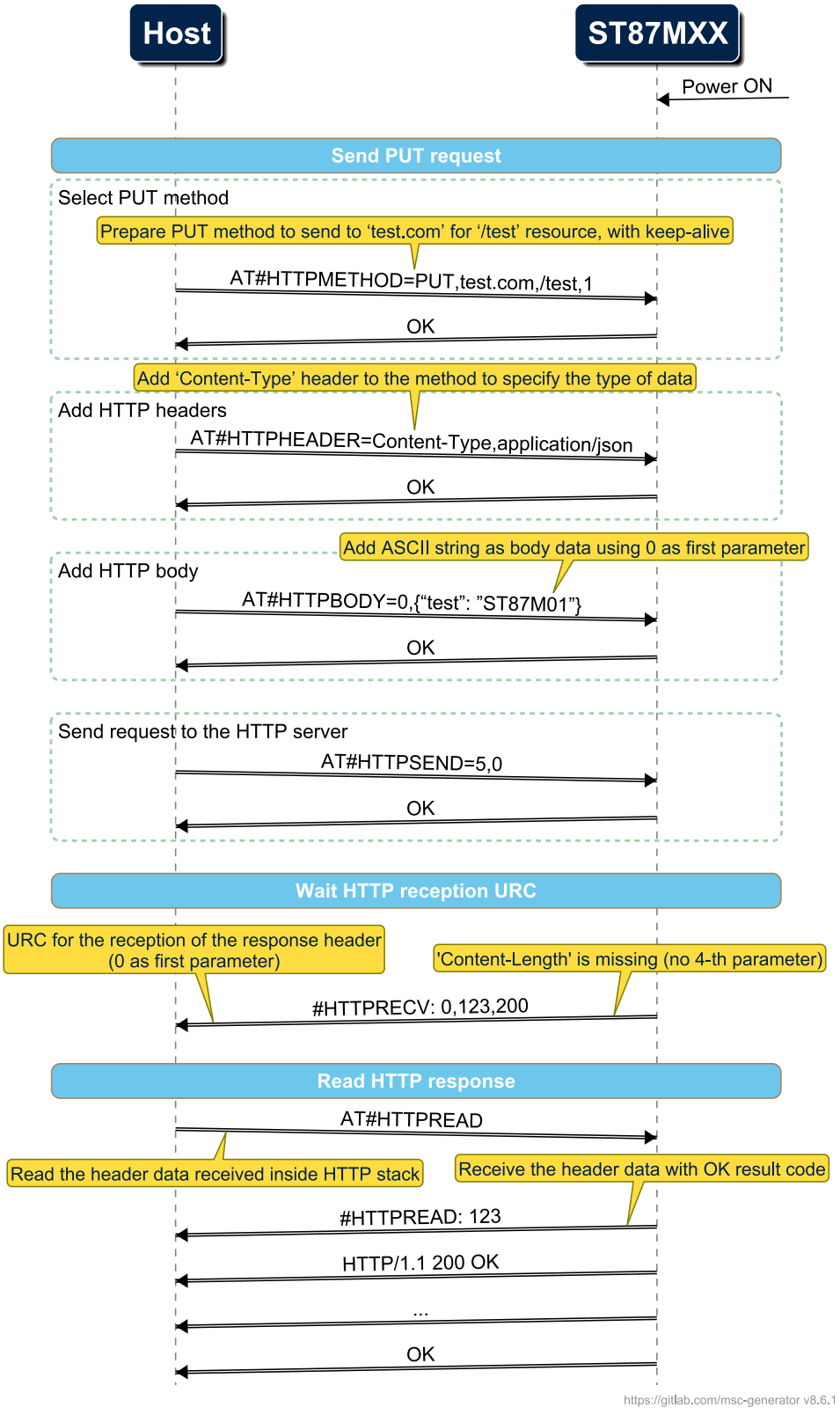
#HTTPRECV: … Asynchronous URC to inform the host that data has been received from HTTP server

AT#HTTPREAD … Reads received HTTP data (header or body data).

5.2 Usage examples

5.2.1 PUT request

5.2.1.1 MSC

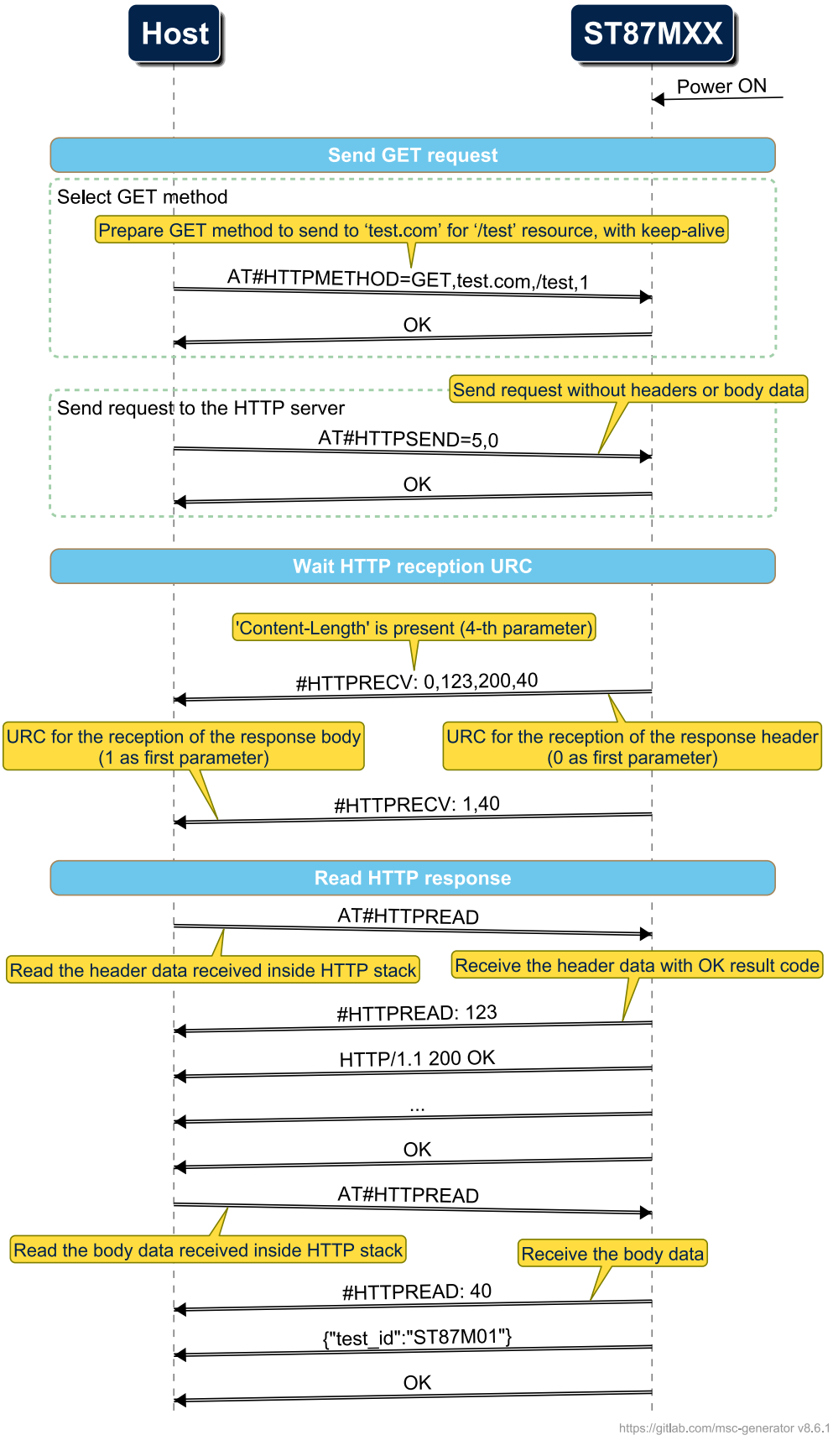


5.2.1.2 Terminal

```
AT#HTTPMETHOD=PUT,test.com,/test,1
OK

AT#HTTPHEADER=Content-Type,application/json
OK

AT#HTTPBODY=0,{“test”: ”ST87M01”}
OK

AT#HTTPSEND=5,0
OK

#HTTPRECV: 0,123,200

AT#HTTPREAD
#HTTPREAD: 123
HTTP/1.1 200 OK
Server: st.com
Date: Mon, 15 Jul 2024 11:59:50 GMT
Content-Type: text/html
Content-Length: 52
Connection: close
<html>
<head><title>200 OK Request</title></head>
</html>


OK
```

5.2.2 GET request

5.2.2.1 MSC

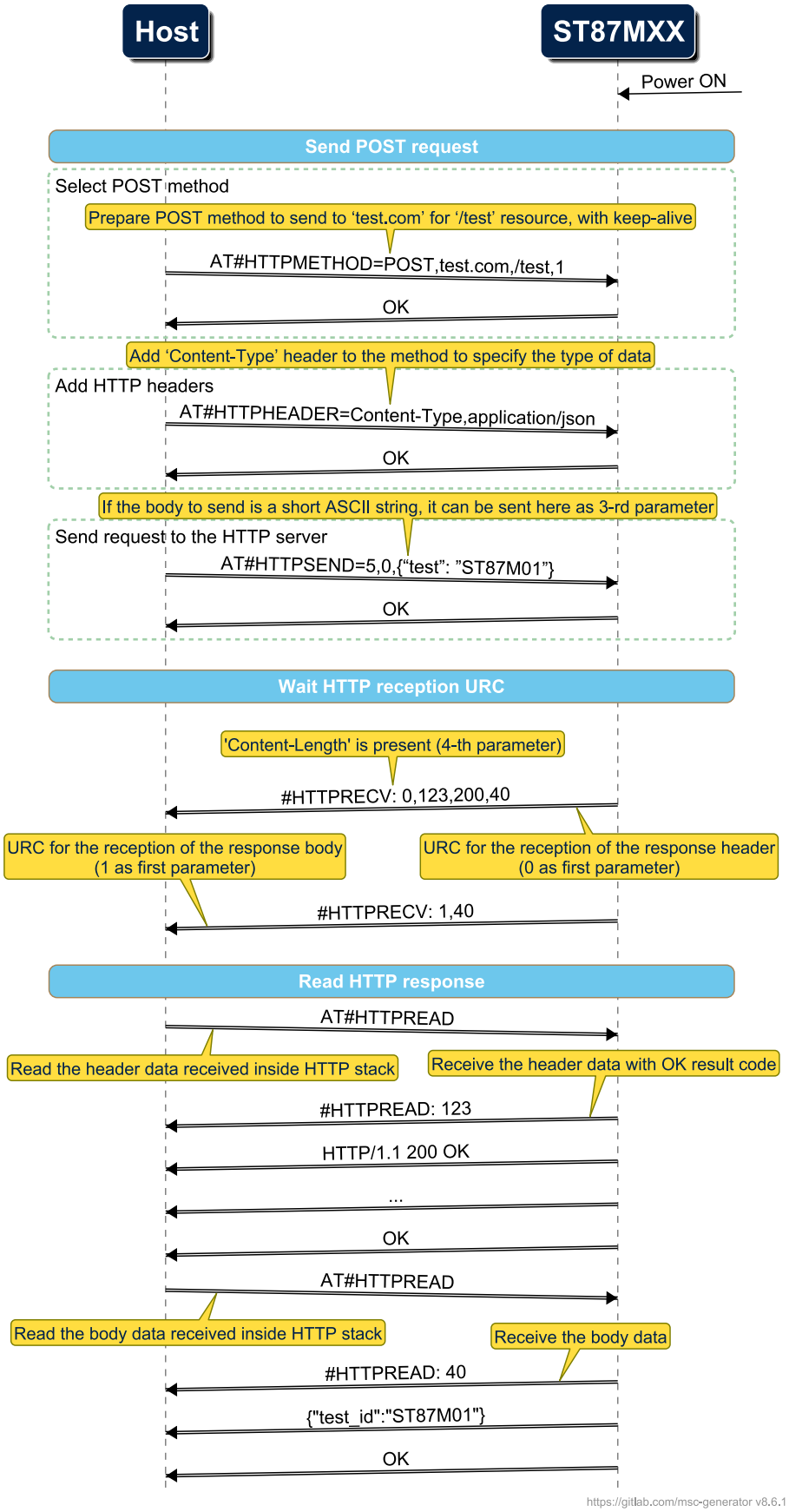


5.2.2.2 Terminal

```
AT#HTTPMETHOD=GET,test.com,/test,1
OK

AT#HTTPSEND=5,0
OK

#HTTPRECV: 0,123,200,40

#HTTPRECV: 1,40

AT#HTTPREAD
#HTTPREAD: 123
HTTP/1.1 200 OK
Server: st.com
Date: Mon, 15 Jul 2024 11:59:50 GMT
Content-Type: text/html
Content-Length: 40
Connection: close
<html>
<head><title>200 OK Request</title></head>
</html>


OK

AT#HTTPREAD
#HTTPREAD: 40
{"message":"OK","testData":"1234567890"}

OK
```

5.2.3 POST request

5.2.3.1 MSC

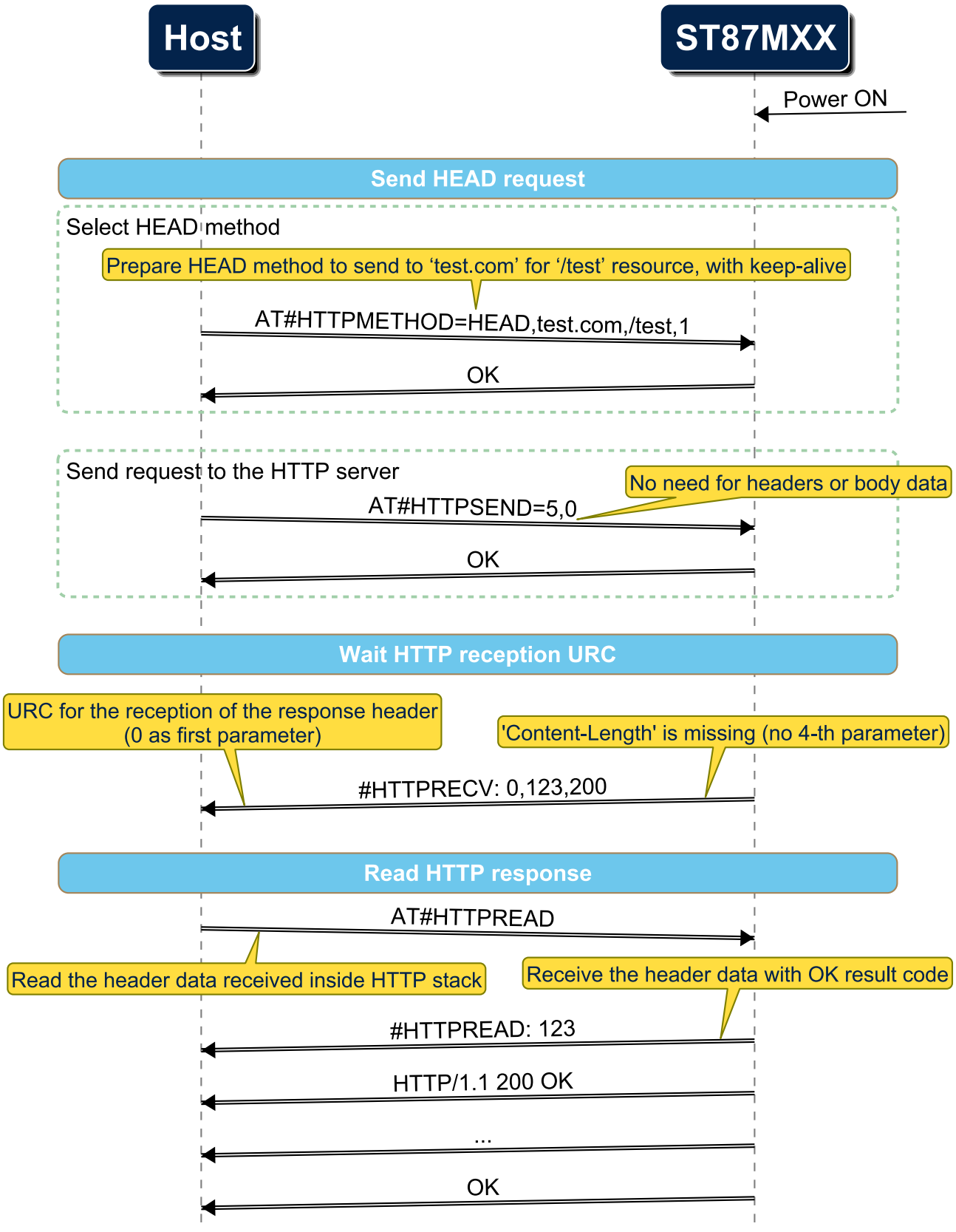


5.2.3.2 Terminal

```
AT#HTTPMETHOD=POST,test.com,/test,1
OK

AT#HTTPHEADER=Content-Type,application/json
OK

AT#HTTPSEND=5,0,{"test": "ST87M01"}
OK

#HTTPRECV: 0,123,200,40

#HTTPRECV: 1,40

AT#HTTPREAD
#HTTPREAD: 123
HTTP/1.1 200 OK
Server: st.com
Date: Mon, 15 Jul 2024 11:59:50 GMT
Content-Type: text/html
Content-Length: 40
Connection: close
<html>
<head><title>200 OK Request</title></head>
</html>


OK

AT#HTTPREAD
#HTTPREAD: 40
{"message":"OK","testData":"1234567890"}

OK
```

5.2.4 HEAD request

5.2.4.1 MSC



https://gitlab.com/msc-generator v8.6.1

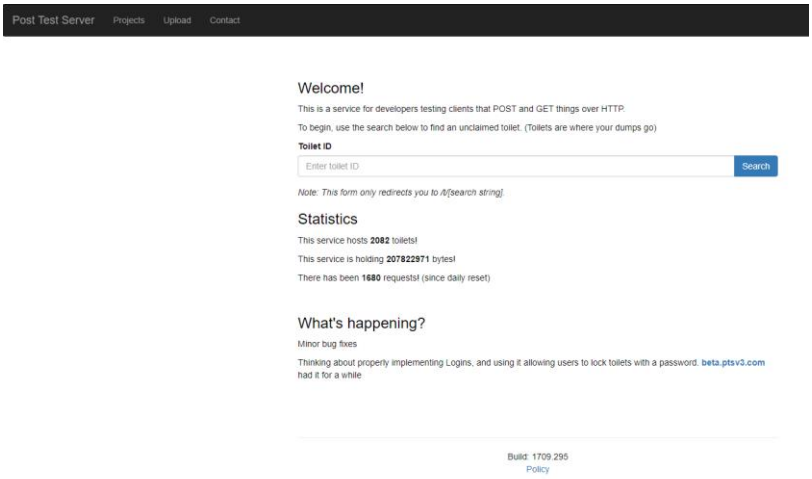
5.2.4.2 Terminal

```
AT#HTTPMETHOD=HEAD,test.com,/test,1
OK

AT#HTTPSEND=5,0
OK

#HTTPRECV: 0,123,200

AT#HTTPREAD
#HTTPREAD: 123
HTTP/1.1 200 OK
Server: st.com
Date: Mon, 15 Jul 2024 11:59:50 GMT
Content-Type: text/html
Content-Length: 52
Connection: close
<html>
<head><title>200 OK Request</title></head>
</html>


OK
```

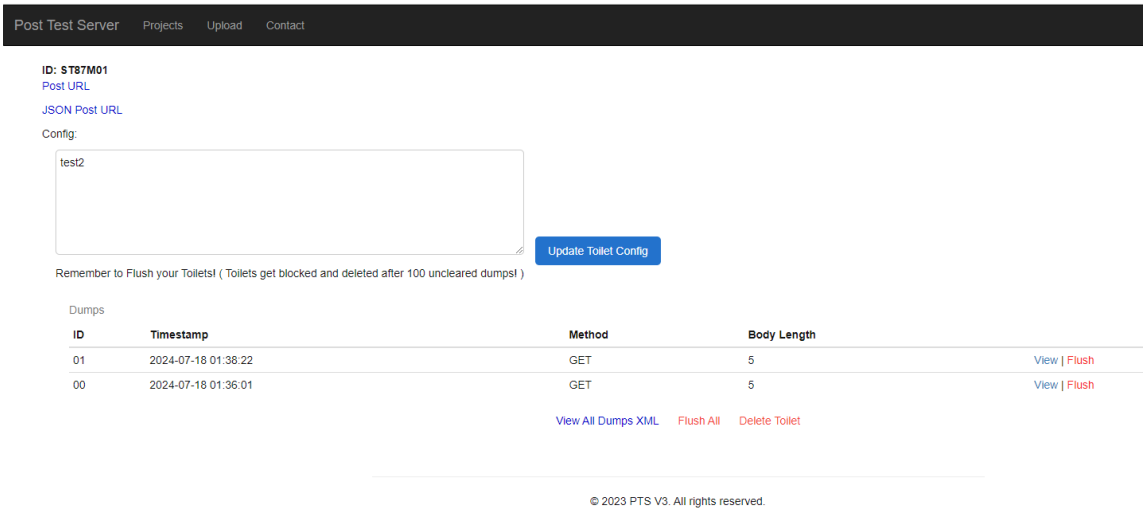
5.3 HTTP transaction with ST87MXX module

5.3.1 Example with first HTTP test server (ptsv3.com)

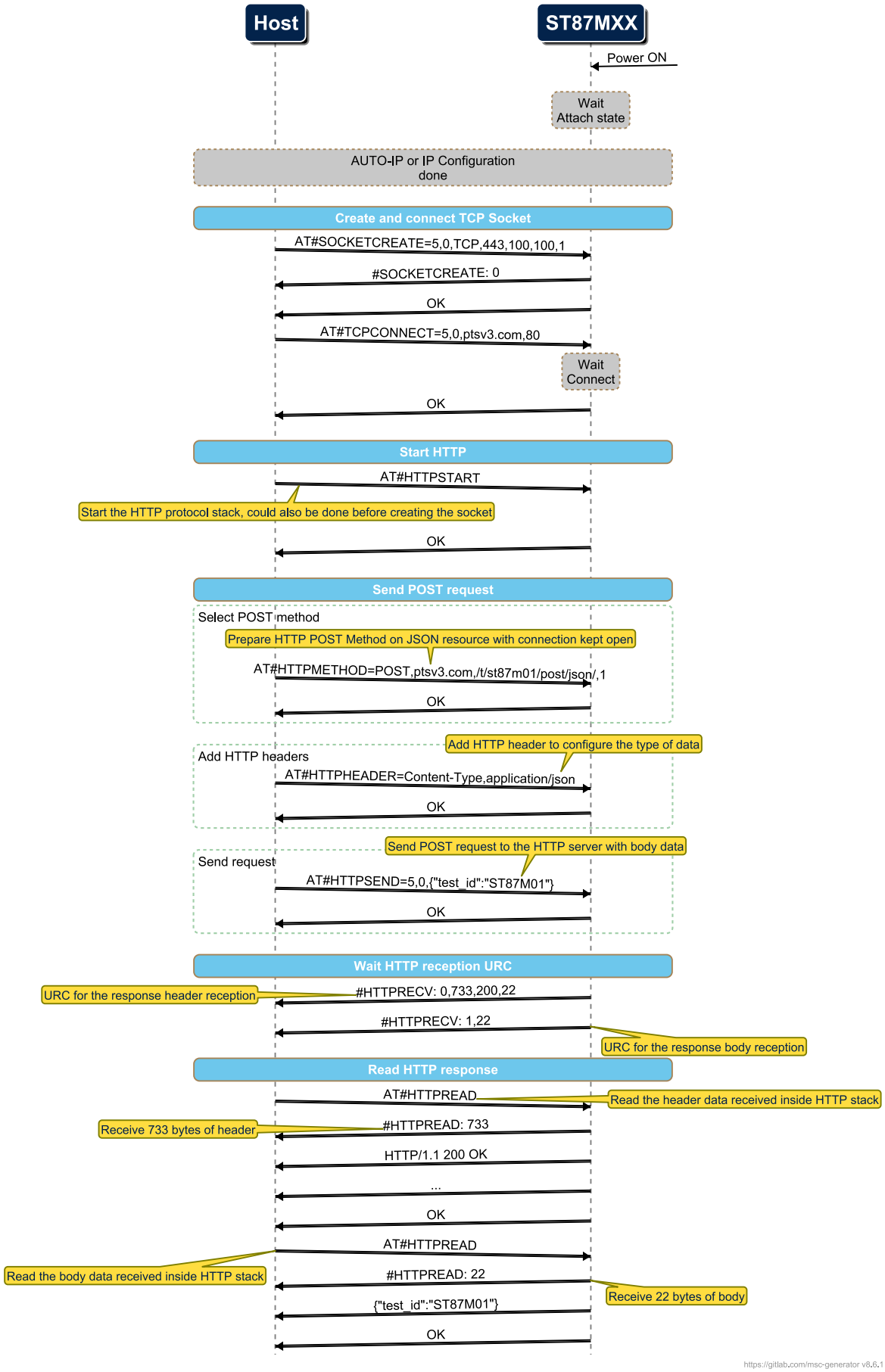
For this example, the free http test service on the web: **ptsv3.com** is used.
You can create your proper own ToiletID to have your http dedicated test folder.
For ST, we have created the ToiletID: **ST87M01**



The address used of our http Folder is: http://ptsv3.com/t/ST87M01/



5.3.1.1 MSC


Host
ST87MXX
Power ON
Wait
Attach state
AUTO-IP or IP Configuration
done
Create and connect TCP Socket
AT#SOCKETCREATE=5,0,TCP,443,100,100,1
#SOCKETCREATE: 0
OK
AT#TCPCONNECT=5,0,ptsv3.com,80
Wait
Connect
OK
Start HTTP
AT#HTTPSTART
Start the HTTP protocol stack, could also be done before creating the socket
OK
Send POST request
Select POST method
Prepare HTTP POST Method on JSON resource with connection kept open
AT#HTTPMETHOD=POST,ptsv3.com,/t/st87m01/post/json/,1
OK
Add HTTP header to configure the type of data
Add HTTP headers
AT#HTTPHEADER=Content-Type,application/json
OK
Send POST request to the HTTP server with body data
Send request
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK
Wait HTTP reception URC
URC for the response header reception
#HTTPRECV: 0,733,200,22
#HTTPRECV: 1,22
URC for the response body reception
Read HTTP response
AT#HTTPREAD
Read the header data received inside HTTP stack
Receive 733 bytes of header
#HTTPREAD: 733
HTTP/1.1 200 OK
...
OK
AT#HTTPREAD
Read the body data received inside HTTP stack
#HTTPREAD: 22
Receive 22 bytes of body
{"test_id":"ST87M01"}
OK
https://gitlab.com/msc-generator v8.6.1

5.3.1.2 Terminal

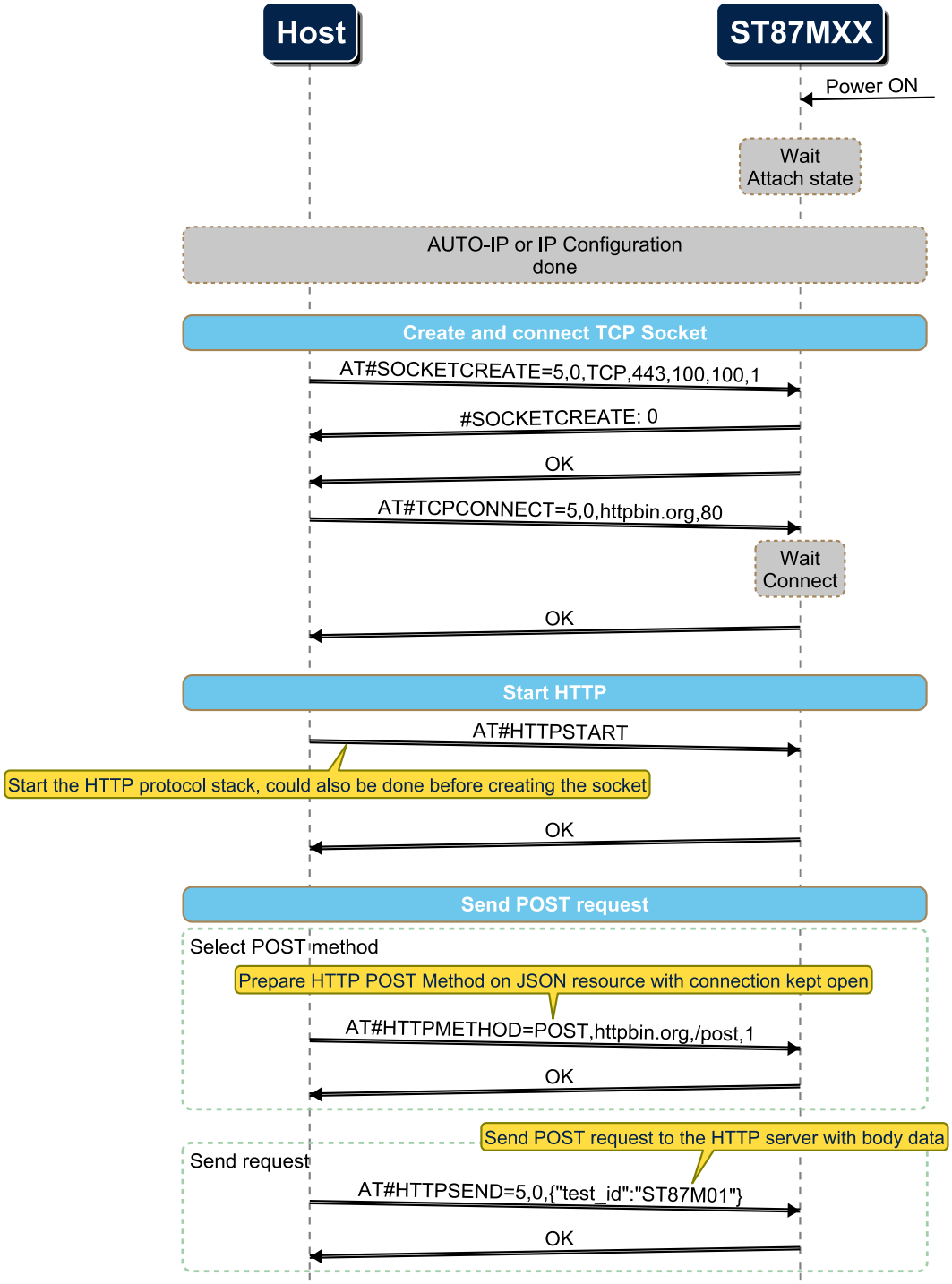
Below is the sequence obtained in the UART terminal when executing the scenario (in **Bold** the commands sent to the module):

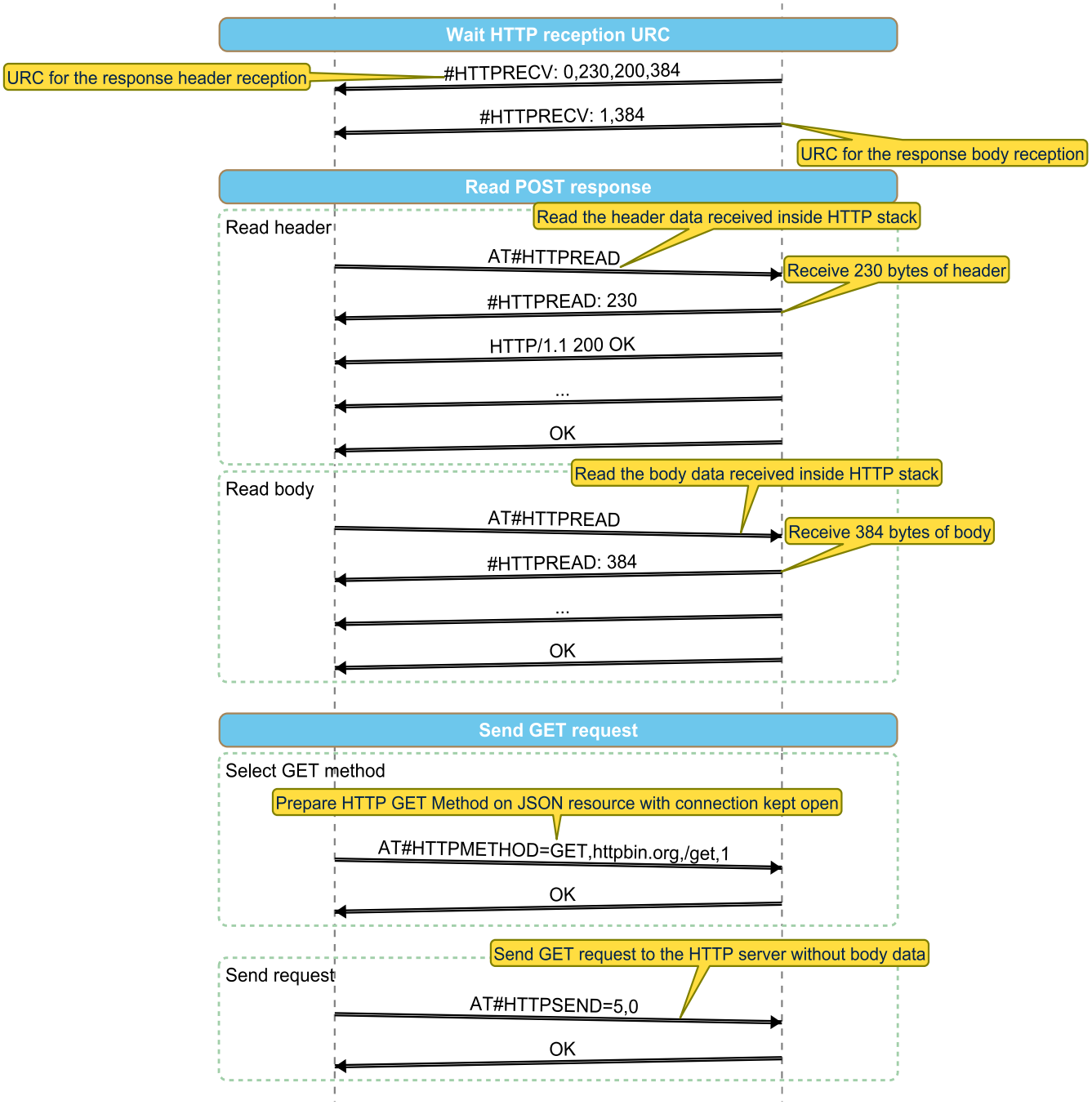
```
AT#SOCKETCREATE=5,0,TCP,443,100,100,1
#SOCKETCREATE: 0

OK
AT#TCPCONNECT=5,0,ptsv3.com,80

OK
AT#HTTPSTART
OK
AT#HTTPMETHOD=POST,ptsv3.com,/t/st87m01/post/json/,1
OK
AT#HTTPHEADER=Content-Type,application/json
OK
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK
#HTTPRECV: 0,733,200,22

#HTTPRECV: 1,22
AT#HTTPREAD
#HTTPREAD: 733
HTTP/1.1 200 OK
Date: Tue, 01 Jul 2025 13:19:28 GMT
Content-Type: application/json
Content-Length: 22
Connection: keep-alive
CF-RAY: 958631f67ca95c24-TLL
cf-cache-status: DYNAMIC
Report-To:
{"endpoints":[{"url":"https:\/\/a.nel.cloudflare.com\/report\/v4?s=DzGu2fpbniQKL
UZbv6CD%2F6uDZhmmYVL32JbYkPtLptSPJtkgX7YznTjyydUIS3OvbSTeI38%2FgW9nUE5t172yJ1rOf
9ouWEOD34K%2FkzIi0j8sbkMzqrkgSZud1oA%3D"}],"group":"cf-nel","max_age":604800}
NEL: {"success_fraction":0,"report_to":"cf-nel","max_age":604800}
Server: cloudflare
server-timing:
cfL4;desc="?proto=TCP&rtt=186243&min_rtt=186243&rtt_var=93121&sent=2&recv=4&lost
=0&retrans=0&sent_bytes=0&recv_bytes=185&delivery_rate=0&cwnd=245&unsent_bytes=0
&cid=0000000000000000&ts=0&x=0"



OK
AT#HTTPREAD
#HTTPREAD: 22
{"test_id":"ST87M01"}


OK
```

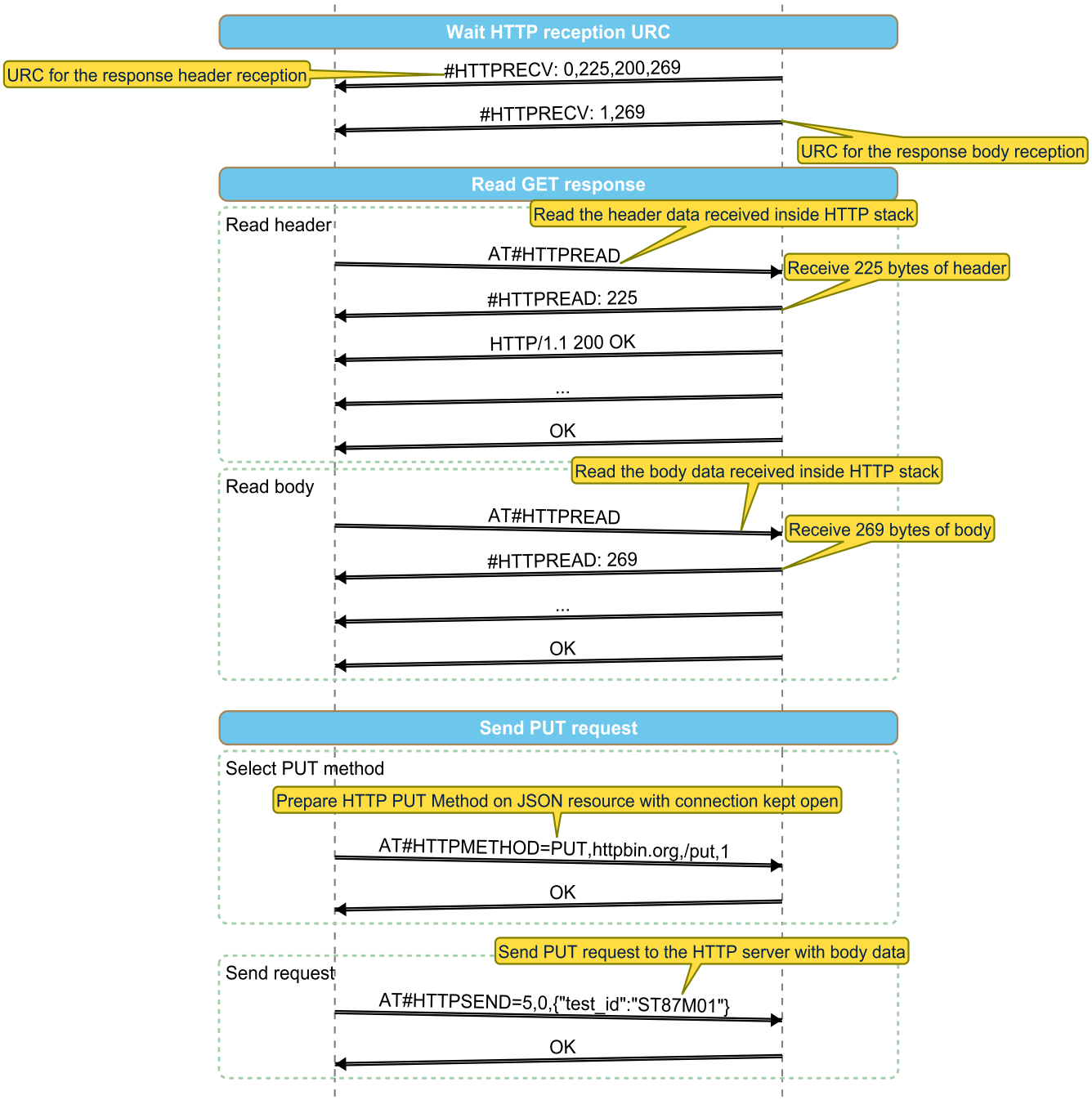
5.3.2 Example with second HTTP test server (httpbin.org)

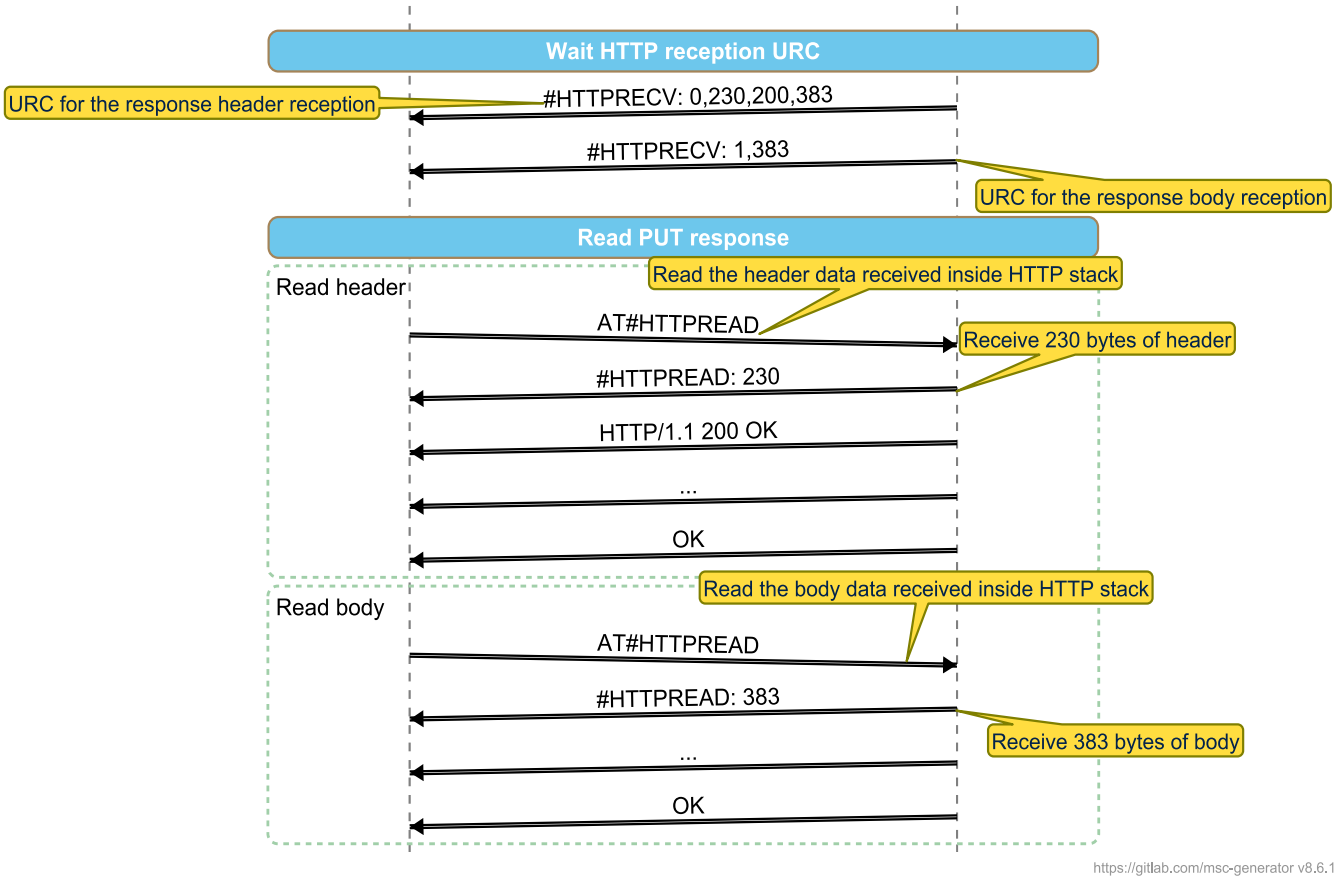
The new Tests server found inside Internet is : http://httpbin.org/#/HTTP_Methods
Below is an example of POST, GET and PUT requests.

5.3.2.1 MSC




Wait HTTP reception URC
URC for the response header reception
#HTTPRECV: 0,230,200,384
#HTTPRECV: 1,384
URC for the response body reception
Read POST response
Read header
Read the header data received inside HTTP stack
AT#HTTPREAD
Receive 230 bytes of header
#HTTPREAD: 230
HTTP/1.1 200 OK
...
OK
Read body
Read the body data received inside HTTP stack
AT#HTTPREAD
Receive 384 bytes of body
#HTTPREAD: 384
...
OK
Send GET request
Select GET method
Prepare HTTP GET Method on JSON resource with connection kept open
AT#HTTPMETHOD=GET,httpbin.org,/get,1
OK
Send request
Send GET request to the HTTP server without body data
AT#HTTPSEND=5,0
OK


Wait HTTP reception URC
URC for the response header reception
#HTTPRECV: 0,225,200,269
#HTTPRECV: 1,269
URC for the response body reception
Read GET response
Read header
Read the header data received inside HTTP stack
AT#HTTPREAD
Receive 225 bytes of header
#HTTPREAD: 225
HTTP/1.1 200 OK
...
OK
Read body
Read the body data received inside HTTP stack
AT#HTTPREAD
Receive 269 bytes of body
#HTTPREAD: 269
...
OK
Send PUT request
Select PUT method
Prepare HTTP PUT Method on JSON resource with connection kept open
AT#HTTPMETHOD=PUT,httpbin.org,/put,1
OK
Send request
Send PUT request to the HTTP server with body data
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK


Wait HTTP reception URC
URC for the response header reception
#HTTPRECV: 0,230,200,383
#HTTPRECV: 1,383
URC for the response body reception
Read PUT response
Read header
Read the header data received inside HTTP stack
AT#HTTPREAD
Receive 230 bytes of header
#HTTPREAD: 230
HTTP/1.1 200 OK
...
OK
Read body
Read the body data received inside HTTP stack
AT#HTTPREAD
#HTTPREAD: 383
Receive 383 bytes of body
...
OK
https://gitlab.com/msc-generator v8.6.1

5.3.2.2 Terminal

```
AT#SOCKETCREATE=5,0,TCP,443,100,100,1
#SOCKETCREATE: 0

OK
AT#TCPCONNECT=5,0,httpbin.org,80
OK
AT#HTTPSTART
OK
AT#HTTPMETHOD=POST,httpbin.org,/post,1
OK
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK
#HTTPRECV: 0,230,200,384

#HTTPRECV: 1,384
AT#HTTPREAD
#HTTPREAD: 230
HTTP/1.1 200 OK
Date: Tue, 01 Jul 2025 13:37:49 GMT
Content-Type: application/json
Content-Length: 384
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true



OK
AT#HTTPREAD
#HTTPREAD: 384
{
  "args": {},
  "data": "{\"test_id\":\"ST87M01\"}",
  "files": {},
  "form": {},
  "headers": {
    "Content-Length": "21",
    "Host": "httpbin.org",
    "User-Agent": "Tyrion Demo client",
    "X-Amzn-Trace-Id": "Root=1-6863e4ad-1de769bd20e670a2547c2120"
  },
  "json": {
    "test_id": "ST87M01"
  },
  "origin": "77.240.240.236",
  "url": "http://httpbin.org/post"
}



OK
AT#HTTPMETHOD=GET,httpbin.org,/get,0
OK
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK

#HTTPRECV: 0,225,200,269

#HTTPRECV: 1,269
```

```
AT#HTTPREAD
#HTTPREAD: 225
HTTP/1.1 200 OK
Date: Tue, 01 Jul 2025 13:43:29 GMT
Content-Type: application/json
Content-Length: 269
Connection: close
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true



OK
AT#HTTPREAD
#HTTPREAD: 269
{
  "args": {},
  "headers": {
    "Content-Length": "21",
    "Host": "httpbin.org",
    "User-Agent": "Tyrion Demo client",
    "X-Amzn-Trace-Id": "Root=1-6863e5ff-06e8425971a721776a629999"
  },
  "origin": "77.240.240.239",
  "url": "http://httpbin.org/get"
}


OK
AT#HTTPMETHOD=PUT,httpbin.org,/put,0
OK
AT#HTTPSEND=5,0,{"test_id":"ST87M01"}
OK

#HTTPRECV: 0,230,200,383

#HTTPRECV: 1,383
AT#HTTPREAD
#HTTPREAD: 230
HTTP/1.1 200 OK
Date: Tue, 01 Jul 2025 13:47:12 GMT
Content-Type: application/json
Content-Length: 383
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true



OK
AT#HTTPREAD

#HTTPREAD: 383
{
  "args": {},
  "data": "{\"test_id\":\"ST87M01\"}",
  "files": {},
  "form": {},
  "headers": {
    "Content-Length": "21",
```

```
    "Host": "httpbin.org",
    "User-Agent": "Tyrion Demo client",
    "X-Amzn-Trace-Id": "Root=1-6863e6dd-798a97123d05a7862fc0d278"
  },
  "json": {
    "test_id": "ST87M01"
  },
  "origin": "77.240.240.239",
  "url": "http://httpbin.org/put"
}


OK
```

6 HTTPS METHODS

6.1 Preamble

The use of Secure HTTP protocol (HTTPS) pre-requires the installation of the TLS private key and certificates inside the ST87M0 (please follow the TLS application note to create a security profile).

The HTTPS protocol can be accessed by using a secure TCP socket, that includes TLS layer, instead of a normal unsecure TCP socket. To create a secure TCP socket, an additional parameter that indicates the security profile ID must be added when creating the socket.

When the secure TCP socket is created and connected, the HTTP protocol stack uses it to exchange data, achieving HTTPS protocol communication.

6.2 Create Secured TCP socket.

To exchange data over HTTPS, a secure TCP socket is needed.
The ME can create a new secure TCP socket with the following at command:

AT#SOCKETCREATE=5,0,TCP,443,30,30,1,0

The first seven parameters are the same described in 3.2.

The eighth parameter, **0**, is an additional parameter that indicates the ID of the security profile to be used by the socket (See [16] to learn how to add TLS provisioning keys and certificates profile inside ST87MXX).

Except for the creation of the secure socket, the rest of the procedure to send or receive data from a HTTPS server is the same as the one for a HTTP server.
The same AT commands can be used to send POST or GET methods to a HTTPS server or read data from the HTTP FIFO buffer.

Also, the URC are the same for both HTTP and HTTPS, except for:

#HTTPDISC: <tls_error_code>

Where **<tls_error_code>** is the code of the error that caused the disconnection.