Build a Micro HTTP Server for Embedded System

Connect to devices more easily

Jian-Hong Pan (StarNight)

@ ELCE / OpenIoT Summit Europe 2016
Outline

- History
- HTTP Protocol
  - Header & Body
- The HTTP Server
  - Concurrency
  - CGI & FastCGI
  - Prototype with Python
  - Automation Test
  - Implemented in C
- Micro HTTP Server on RTOS
  - FreeRTOS
  - Hardware
  - Socket API
  - Select API
  - Assemble Parts
- Demo
  - If the local WiFi is accessible (XD)
Who am I

潘建宏 / Jian-Hong Pan (StarNight)

I come from Taiwan!

You can find me at ~

http://www.slideshare.net/chienhungpan/
GitHub : starnight
Facebook : Jian-Hong Pan
Email : starnight [AT] g.ncu.edu.tw
Taiwan

Formosa

Main island area: 
~35,980 km²
History

- It starts from machine controlling which controls the machine’s motion.
- It is the motor that most be used as an actuator in the machine controlling.
Motor Controlling ...
Measurement of Motor

- Parameters of a motor may change due to the environment: temperature, humidity..., etc.

- Measure the rotation of the motor:
  - With the encoder which produces square waves.
  - With the sensorless method: the waves of the phases of the motor’s voltage, current or something else.

- Also for system identification.
Send & Get of the Communication

- In traditional, a protocol over the serial port is used for communication between the computer and the controller, measuring instruments.
- The devices are distributed anywhere and the serial ports wiring with the central computer could be a problem.
- Send commands and get values through the communication over serial ports that may not as fast as we want.
Communication over Internet

- Linking the devices with the TCP/IP based internet is possible. It is faster and more convenient for management.
- Protocol over TCP/IP:
  - MQTT, CoAP ...
  - or just RESTful web API on HTTP
  - Choosing depends on case by case.

PS. Internet may not be the best solution for all of the cases, but is one of the candidate.
In General

Internet

Gateway

Device #1

Device #2

Device #n

or

Internet

Device

RS232, 485, 422
Bluetooth, Zigbee
Ethernet, WiFi
For My Condition

Full Stack / IoT is fancy!!!
I just want to have an HTTP server on the embedded system.
Limitations

- Considering the size and power restrictions, most embedded devices have limited resources. (MCU level)
  - Less processors: Usually has only one processor, single thread.
  - Less memory: On-chip RAM < 1MB.
  - Less storage: On-chip flash < 1MB.
  - Lower speed grade: Clock rate < 1GHz.
  - The on chip OS may even not provide process, thread APIs.
- The Apache, NGINX... HTTP server could not be placed in that restricted environment.

PS. The numbers mentioned above may not be the real numbers, but they are around that grade levels.
HTTP Server on OSI 7 Layers

Controlled by Application

Socket APIs

Controlled by OS

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

RFC 2616 HTTP/1.1

Hypertext Transfer Protocol -- HTTP/1.1

The HTTP protocol is a request/response protocol.

A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server.

The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

Reference: RFC 2616 1.4 Overall Operation
HTTP Request

```
GET / HTTP/1.1
Host: www.linuxfoundation.org
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-TW,zh;q=0.8,en-US;q=0.6,en;q=0.4
[truncated]Cookie: __utmz=1; __utma=103159837.2085678669.1466603403.1466603403.1466603403.1466603403
If-None-Match: W/"1466598806-0"

[Full request URI: http://www.linuxfoundation.org/]
[HTTP request 4/4]
[Prev request in frame: 2436]
[Response in frame: 2503]
```

HTTP Request
HTTP Response

```
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 22 Jun 2016 13:51:07 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Etag: W"1466598806-0"
Cache-Control: public, max-age=0
Last-Modified: Wed, 22 Jun 2016 12:33:26 +0000
Expires: Sun, 11 Mar 1984 12:00:00 GMT
Vary: Cookie
Content-Encoding: gzip

[HTTP response 4/4]
[Time since request: 0.424541585 seconds]
[Prev request in frame: 2436]
[Prev request in frame: 2443]
[Request in frame: 2457]
HTTP chunked response
Content-encoded entity body (gzip): 13812 bytes -> 57466 bytes

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>The Linux Foundation</title>

```
```
HTTP Message - Message Types

- HTTP messages consist of requests from client to server and responses from server to client.

- Request (section 5) and Response (section 6) messages use the generic message format of RFC 822 [9] for transferring entities (the payload of the message).

- Both types of message consist of a start-line, zero or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and possibly a message-body.

```
generic-message = start-line *(message-header CRLF) CRLF [ message-body ]
start-line = Request-Line | Status-Line
```

Reference: RFC 2616 4.1 Message Types
HTTP Message - Message Headers

- HTTP header fields, which include general-header (section 4.5), request-header (section 5.3), response-header (section 6.2), and entity-header (section 7.1) fields.

- Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. The field value MAY be preceded by any amount of LWS, though a single SP is preferred.

```
message-header = field-name "::" [ field-value ]
field-name = token
field-value = *( field-content | LWS )
field-content = <the OCTETs making up the field-value and consisting of either *TEXT or combinations of token, separators, and quoted-string>
```

Reference: RFC 2616 4.2 Message Headers
HTTP Message - Message Body

- The message-body (if any) of an HTTP message is used to carry the **entity-body** associated with the request or response.

\[
\text{message-body} = \text{entity-body} \mid <\text{entity-body encoded as per Transfer-Encoding}> 
\]

Reference: RFC 2616 4.3 Message Body
Request

- A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

\[
\text{Request} = \text{Request-Line} \\
\quad *(( \text{general-header} \\
\quad \quad | \text{request-header} \\
\quad \quad | \text{entity-header} ) \ CRLF) \\
\quad \text{CRLF} \\
\quad [ \text{message-body} ]
\]

Reference: RFC 2616 5 Request
The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

\[
\text{Request-Line} = \text{Method SP Request-URI SP HTTP-Version CRLF}
\]
Method

- The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
Method = "OPTIONS"  
| "GET"  
| "HEAD" 
| "POST" 
| "PUT"  
| "DELETE" 
| "TRACE" 
| "CONNECT" 
| extension-method
```

Reference: RFC 2616 5.1.1 Method
The Request-URI is a Uniform Resource Identifier (section 3.2) and identifies the resource upon which to apply the request.

\[ \text{Request-URI} = \"*\" \mid \text{absoluteURI} \mid \text{abs_path} \mid \text{authority} \]

Reference: RFC 2616 5.1.2 Request-URI
Request Header Fields

- The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

\[
\text{request-header} = \text{Accept} \\
| \text{Accept-Charset} \\
| \text{Accept-Encoding} \\
| \text{Accept-Language} \\
| \text{Authorization} \\
| \text{Expect} \\
\ldots
\]

Reference: RFC 2616 5.3 Request Header Fields
HTTP/1.1 200 OK

Server: nginx
Date: Wed, 22 Jun 2016 13:51:07 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Etag: W"1466598806-0"
Cache-Control: public, max-age=0
Last-Modified: Wed, 22 Jun 2016 12:33:26 +0000
Expires: Sun, 11 Mar 1984 12:00:00 GMT
Vary: Cookie
Content-Encoding: gzip

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>The Linux Foundation</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>

http response message header

http response message body

html document body
After receiving and interpreting a request message, a server responds with an HTTP response message.

Response = Status-Line  
*((( general-header  
| response-header  
| entity-header ) CRLF)  
CRLF  
[ message-body ]

Reference: RFC 2616 6 Response
Status-Line

- The first line of a Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Reference: RFC 2616 6.1 Status-Line
The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section 10. The Reason-Phrase is intended to give a short textual description of the Status-Code.

- **1XX:** Informational - Request received, continuing process
- **2XX:** Success - The action was successfully received, understood, and accepted
- **3XX:** Redirection - Further action must be taken in order to complete the request
- **4XX:** Client Error - The request contains bad syntax or cannot be fulfilled
- **5XX:** Server Error - The server failed to fulfill an apparently valid request

Reference: RFC 2616 [6.1.1 Status Code and Reason Phrase](https://www.rfc-editor.org/rfc/rfc2616#section-6.1.1)
Response Header Fields

- The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line.

- These header fields give information about the server and about further access to the resource identified by the Request-URI.

\[
\text{response-header} = \text{Accept-Ranges} \\
\quad | \text{Age} \\
\quad | \text{ETag} \\
\quad | \text{Location} \\
\quad \ldots
\]

Reference: RFC 2616 6.2 Response Header Fields
Entity

- Request and Response messages MAY transfer an entity if not otherwise restricted by the request method or response status code.
- An entity consists of entity-header fields and an entity-body, although some responses will only include the entity-headers.

Reference: RFC 2616 7 Entity
Entity Header Fields

- **Entity-header fields define metainformation about the entity-body or, if no body is present, about the resource identified by the request.**
- **Some of this metainformation is OPTIONAL; some might be REQUIRED by portions of this specification.**

```
entity-header = Allow |
| Content-Language |
| Content-Location |
| Content-Range |
| Expires |
| extension-header |

extension-header = message-header
```

Reference: RFC 2616 7.1 Entity Header Fields
The entity-body (if any) sent with an HTTP request or response is in a format and encoding defined by the entity-header fields.

extension-header = message-header

An entity-body is only present in a message when a message-body is present, as described in section 4.3.

The entity-body is obtained from the message-body by decoding any Transfer-Encoding that might have been applied to ensure safe and proper transfer of the message.

Reference: RFC 2616 7.2 Entity Body
After Sockets connected

Request Message:
Request-Line
*($( general-header
| request-header
| entity-header ) CRLF)
CRLF
[ message-body ]

Response Message:
Status-Line
*($( general-header
| response-header
| entity-header ) CRLF)
CRLF
[ message-body ]
The HTTP Server

Concurrency & Backend
Single Server Thread & Single Client

Client Socket

HTTP Request

HTTP Response

Server Socket

HTTP Request Message

Server Application

HTTP Response Message
Which one should be processed first?
Flow Chart of Server Socket

1. Client socket sends a request
2. Server reads a request from the client socket
3. Server process the request and build the response
4. Server writes the response to the client socket
5. Server writes finished

I/O Bound
- Client socket sends a request
- Server reads a request from the client socket
- Server writes the response to the client socket

CPU Bound
- Server process the request and build the response

I/O Bound
- Server writes finished
I/O Bound

- CPU runs faster than I/O devices. If system needs the resources of I/O devices, it will be blocked to wait for the resources.
- If there is only one client socket and request, it may not be the problem.
- If there are two or more clients and requests at the same time, the blocked I/O will hang up the server. Clients may get responses slowly or even be timeout.
Concurrency

- The server could use the process (`fork()`) or thread (`pthread library`) APIs to serve multiple clients at the same time.
  - Socket works great in blocking mode.
  - Process or thread APIs must be provided by OS. (Resources considering.)
  - Overhead of context switching.

- Use **I/O Multiplexing & Non-Blocking sockets**.
  - It could be used in the single thread situation.
  - Compared with the process and thread, it is less resources required.
  - No more processes or threads, no context switching.
I/O Multiplexing & Non-Blocking

- **select()** monitors the sockets’ \((fd_set)\) status flag and returns the status of all sockets. It exists in most OSes.
- **poll()** works like select(), but represents in different form (**pollfd**).
- **epoll()** monitors sockets’ status and trigger the related events. It returns only triggered events array. It has been implemented since Linux 2.6.
- **recv(), send()** in non-blocking mode.
- Use **fcntl()** to set the **O_NONBLOCK** (non-blocking) flag of the socket on.
RFC 3857 CGI

The Common Gateway Interface Version 1.1

Abstract

The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner. Currently, the supported information servers are HTTP servers.

Reference: RFC 3857 Abstract
Terminology

- 'script'
  
  *The software that is invoked by the server according to this interface.* It need not be a *standalone program*, but could be a *dynamically-loaded or shared library*, or even a *subroutine* in the server.

- 'meta-variable'
  
  *A named parameter which carries information from the server to the script.* It is not necessarily a variable in the *operating system's environment*, although that is the most common implementation.

Reference: RFC 3857 1.4. Terminology
Steps for CGI

1. Apache HTTP Server receives a request and parse it.
2. The server puts the request header into the environment variables, then forks to have a child process which inherits parent's environment variables.
3. The child process executes the CGI script and gets the request header fields from environment variables, the request body from STDIN.
4. The Apache HTTP Server will have the response which is produced and written from the STDOUT of the child process.
FastCGI

- It is a variation on the earlier CGI.
- Instead of creating a new process for each request, FastCGI uses persistent processes to handle a series of requests. These processes are owned by the FastCGI server, not the web server.
- To service an incoming request, the web server sends environment information and the page request itself to a FastCGI process over a socket (in the case of local FastCGI processes on the web server) or TCP connection (for remote FastCGI processes in a server farm).
- Responses are returned from the process to the web server over the same connection, and the web server subsequently delivers that response to the end-user.
- The connection may be closed at the end of a response, but both the web server and the FastCGI service processes persist.

Reference: Wiki FastCGI
NSAPI

Netscape Server Application Programming Interface

- Applications that use NSAPI are referred to as **NSAPI plug-ins**. Each plug-in implements one or more **Server Application Functions (SAFs)**.

- Unlike CGI programs, **NSAPI plug-ins run inside the server process**. Because **CGI programs run outside of the server process**, CGI programs are generally slower than NSAPI plug-ins.

- Running outside of the server process can improve server reliability by isolating potentially buggy applications from the server software and from each other.

- **NSAPI SAFs can be configured to run at different stages of request processing**.

Micro HTTP Server

- It could work on limited resources embedded system.
- It could process multiple HTTP clients concurrently.
- It parses the HTTP request message and passes the message to corresponding server application functions (SAFs) according to the Request-Line. (Like NSAPI)
- The SAFs process with the HTTP request message and build the HTTP response message.
- The server application functions can collaborate like a chain. Therefore, each server application function only does a simple job.
https://github.com/starnight/MicroHttpServer
Server Socket Flow Chart

1. Start
2. Have a socket
3. The socket listens on designated port
4. Select ready sockets
   - For each ready socket
     - Is server socket
       - Accept a new client socket
     - Not server socket
6. Is server socket
   - Not server socket
7. Is close state
   - Not close state
8. Is read state
   - Not read state
9. Read the socket
10. Build the HTTP request message
11. Process and build the HTTP response message in server application functions
12. Is write state
   - Not write state
13. Write the HTTP response message
14. Not write state
15. Close the socket
Sequential Diagram

Micro HTTP Server

Server Socket  Middleware  SAFs

Requests

I/O Multiplexing Model
select()

HTTP Request Message

NSAPI like
Dispatch

HTTP Request Message

HTTP Response Message

HTTP Response Message

Response
Prototype with Python

- The *py-version* of the repository.
- Python is so convenient to do prototypes.
- Because of that, there is a little different between Python and C version, and is more simple with I/O multiplexing and the states of ready sockets in part of 'Server Socket'.
- Both Python and C version's 'Middleware' models are the same.
- Users only have to register the routes, the server application functions (SAFs) of the routes and start the HTTP server.
Works in Python 3.2 up!

Make sure the **encoding** during **reading** and **writing sockets**.
Directory Tree in Python Version

- **lib/**:

- **static/**:
  - Static files: HTML, JS, Images ...  

- **main.py**: The entry point of Python Version Micro HTTP Server example.

- **app.py**: The web server application functions of Python Version Micro HTTP Server example.
Example of Python Version

```python
from lib.server import HTTPServer
from lib.middleware import Routes
import app

server = HTTPServer(port=8000)
routes = Routes()
routes.AddRoute("GET", "/", app.WellcomePage)
routes.AddRoute("GET", "/index.html", app.WellcomePage)
routes.AddRoute("POST", "/fib", app.Fib)

server.RunLoop(routes.Dispatch)
```

Register the routes
Run the HTTP server
The callback for new request
def WelcomePage(req, res):
    '''Default welcome page which makes response message.''
    # Build HTTP message body
    res.Body = "<html><body>Hello!<br>
    res.Body += "It's {} for now.".format(
        datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    res.Body += "</body></html>"

    # Build HTTP message header
    res.Header.append(["Status", "200 OK"])
    res.Header.append(    ["Content-Type", "text/html; charset=UTF-8;"])
Server is starting!!!
Server is started!!!
2016-07-01 21:11:34.100975 ('127.0.0.1', 37834) connected
  Parse header
  Parse body
  Send header
  Send body

Hello!
It's 2016-07-01 21:10:15 for now.
Automation Test

- The sub-directory `autotest/` of the repository
- Write a test application `client.py` which acts as an HTTP client with the Python unittest library.
- Have an HTTP client with 4 actions: `Connect`, `Request with GET method`, `Request with POST method`, `Close`.
- Have an unittest class which will execute the test scenarios.
Test Scenarios

- Only connect and close actions.
- Connect, request GET method with a specific URI and check the response and close.
- Connect, request POST method with a specific URI and check the response and close.
- Multiple clients request concurrently.
- Request different URIs to make sure the SAFs work correctly.
Continuous Integration

Use Travis CI:

https://travis-ci.org/starnight/MicroHttpServer

*Thanks to Travis CI!*
.travis.yml in the repository

- language: Python
- python version: 3.2 ~ 3.5
- before_script:
  
  Build (if needed) and execute Python and C version Micro HTTP Server in background

- script:
  
  Execute the test application to test the Python and C version Micro HTTP Server
Using worker: worker-linux-docker-1b92445e.prod.travis-ci.org:travis-linux-13

Build system information
$ export DEBIAN_FRONTEND=noninteractive
3.5 is not installed; attempting download
$ git clone --depth=50 --branch=master https://github.com/starnight/MicroHttpServer.git starnight/MicroHttpServer

This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setguid executables.

If you need to add a command to your escape context, run:
$ sudo -H -s /bin/bash -c "command"

See also:
$ htop
$ ps aux
$ python
$ pip
$ pip 7.1
$ serve
$ execute
$ cd ...

Could run:
$ make
$ ./microhttpserver
$ serve
$ execute
$ cd ...

The command "python autotest/client.py localhost:8000" exited with 0.
$ kill $SERVER_PYTHON_PID

The command "kill $SERVER_PYTHON_PID" exited with 0.
Micro HTTP Server in C

- The **c-version** of the repository.
- Also could be test with the automated test application and integrated with Travis CI.
- The C version is more efficient than the Python version. (The comparison could be found in the automated test result.)
- The C version also could be ported on embedded system.
  - The system must provides socket APIs.
  - The file system is provided for the static files.
Directory Tree in C Version

- **lib/**:

- **static/**:
  - static files: HTML, JS, Images ...

- **main.c**: The entry point of C Version Micro HTTP Server example.

- **app.c & h**: The web server application functions of C Version Micro HTTP Server example.

- **Makefile**: The makefile of this example.
Example of C Version

#include "server.h"
#include "middleware.h"
#include "app.h"

/* The HTTP server of this process. */
HTTPServer srv;

int main(void) {
    /* Register the routes. */
    AddRoute(HTTP_GET, "/index.html", HelloPage);
    AddRoute(HTTP_GET, "/", HelloPage);
    AddRoute(HTTP_POST, "/fib", Fib);
    /* Initial the HTTP server and make it listening on MHS_PORT. */
    HTTPServerInit(&srv, MHS_PORT);
    /* Run the HTTP server forever. */
    /* Run the dispatch callback if there is a new request */
    HTTPServerRunLoop(&srv, Dispatch);
    return 0;
}
```c
#include <string.h>
#include <stdlib.h>
#include "app.h"

void HelloPage(HTTPReqMessage *req, HTTPResMessage *res) {
    int n, i = 0, j;
    char *p;
    char header[] = "HTTP/1.1 200 OK\r\nConnection: close\r\n";
    "Content-Type: text/html; charset=UTF-8\r\n\n";
    char body[] = "<html><body>Hello!<br>許功蓋<br></body></html>";

    /* Build header. */
    p = (char *)res->_buf;
    n = strlen(header);
    memcpy(p, header, n);
    p += n;    i += n;
    /* Build body. */
    n = strlen(body);
    memcpy(p, body, n);
    p += n;    i += n;
    /* Set the length of the HTTP response message. */
    res->_index = i;
}
```
Micro HTTP Server C APIs

GitHub repository Wiki
https://github.com/starnight/MicroHttpServer/wiki/C-API
zack@StarNight:~$ ./microhttpserver
Listening
Accept 1 client. 127.0.0.1:57176
Parse Header
Parse body

Mozilla Firefox

http://localhost:8001/

Hello!
許功蓋
GET
/
HTTP/1.1
Host: localhost:8001
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:47.0) Gecko/20100101 Firefox/47.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Cache-Control: max-age=0
Micro HTTP Server on Embedded System

Ported on STM32F4-Discovery with FreeRTOS for Example
FreeRTOS on STM32F4-Discovery

- The Micro HTTP Server needs the socket APIs which provides by the OS. Therefore, we need an OS on the development board.
- Putting a heavy Linux OS on the limited resource board may not be a good idea. Having a light weight RTOS will be a better solution.
- Considering finding the documents and usability, FreeRTOS is chosen because of the mentioned above.
FreeRTOS is Free which means Freedom

The License could be found at http://www.freertos.org/license.txt
FreeRTOS

- Features Overview
  - http://www.freertos.org/FreeRTOS_Features.html

- FreeRTOS introduced in Wiki of CSIE, NCKU
  - http://wiki.csie.ncku.edu.tw/embedded/freertos

- RTOS objects
  - tasks, queues, semaphores, software timers, mutexes and event groups

- Pure FreeRTOS does not provide socket related APIs!!! T^T
Hardware

● STM32F4-Discovery as mainboard
  ○ STM32F407VG: Cortex-M4
  ○ USART × 2:
    ■ 1 for connecting to WiFi module
    ■ 1 for serial console
  ○ 4 LEDs for demo

● ESP01 as WiFi module
  ○ ESP8266 series
    ■ UART connecting to STM32F4-Discovery

No general internet connection (including Wifi) on board. So ...
Communication Wiring

PC

STM32F4-Discovery

PA2 USART2_TX
PA3 USART2_RX
PC6 USART6_TX
PC7 USART6_RX

UART

ESP01
RX
TX

Console

UART

UART
STM32F4-Discovery

Power & ST-LINK

USB to Serial

ESP01
HTTP Server on STM32F4-Discovery

HTTP Web API
HTML
HTTP
Socket APIs
Socket provided by OS
Serial Device Driver

STM32F4-Discovery

Application
Presentation
Session
Socket to USART
Serial Lines

ESP01 WiFi module
UART
Transport
Network
Data Link
Physical
Socket API

- **Data Types:**
  - `socket`, `sockaddr_in`

- **Constant Flags**

- **Initial socket:**
  - `socket()`
  - `bind()`

- **Server’s works:**
  - `listen()`
  - `accept()`

- **I/O:**
  - `send()`
  - `recv()`

- **Release I/O:**
  - `shutdown()`
  - `close()`

- **Manipulate I/O**
  - `setsockopt()`
  - `fcntl()`
Select API

● Data types:
  ○ fd_set
  ○ struct timeval

● I/O Multiplexing:
  ○ select()
  ○ FD_ZERO()
  ○ FD_SET()
  ○ FD_CLR()
  ○ FD_ISSET()
We also need **ESP8266 & serial drivers** which communicates with ESP01 through UART!
The protocol of the communication between the MCU and ESP01 is AT commands!
AT Commands of ESP01

https://cdn.sparkfun.com/assets/learn_tutorials/4/0/3/4A-ESP8266__AT_Instruction_Set__EN_v0.30.pdf

- AT+CWJAP: Connect to AP
- AT+CIFSR: Get local IP address
- AT+CIPMUX: Enable multiple connections
- AT+CIPSERVER: Configure as TCP server
- AT+CIPSEND: Send data
- AT+CIPCLOSE: Close TCP or UDP connection
- [num],CONNECT: A new client connected (Not listed)
- +IPD: Receive network data
Micro HTTP Server on FreeRTOS

STM32F4-Discovery connected with ESP01

FreeRTOS

ESP8266 Driver acts as NIC

Socket & Select APIs

USART Driver

Yellow blocks need to be implemented

就自幹吧！
Principles of Implementation

1. Implement the used APIs as much as possible!

2. Mocking may be used if the function is not important! → To reduce the complexity
Socket & Select APIs’ Header Files

Refer to and copy Linux header files directly.

To make it simple, merge the variety header files which are included and rewrite them into several files.

Thanks to Open Source!!!
Reference Serial Drivers of Linux

Reference: Serial Drivers [http://www.linux.it/~rubini/docs/serial/serial.html](http://www.linux.it/~rubini/docs/serial/serial.html)
Data Flow and Function Calls

- **Micro HTTP Server**
  - `send()`

- **Socket**
  - `SendSocket()`

- **ESP8266 Driver**
  - `USART_Send()`
  - `clisock.rxQueue`
  - `GetClientRequest()`
  - `xQueueReceive()`
  - `USART_ReadByte()`

- **USART**
  - `USART_SendByte()`

- **Hardware**
  - `TX/RX Lines`
  - `__rxBuf`

- **Interrupt_Handler**
  - `USART_Read()`
  - `USART_ReadByte()`

- Function Call
- Data Flow
ESP8266 Driver

- Initial the USART channel
- Makes ESP01 as a network interface
  - Translates the system calls to AT commands
- Manage socket resources
  - The file descriptors of sockets
- **USART channel mutex**
  - Both the vESP8266RTask and vESP8266TTask communicate with ESP01 through the same USART channel
- Join an access point
ESP8266 Driver Cont.

- **vESP8266RTask**
  - A persistent task parses the active requests from ESP01 (connect for accept, the requests from client’s sockets)

- **vESP8266TTTask**
  - A persistent task deals the command going to be sent to ESP01 (socket send, socket close)

- **Socket ready to read**
  - Check the socket is ready to be read for I/O multiplexing to monitor the socket’s state

- **Socket ready to write**
  - Check the socket is ready to be written for I/O multiplexing to monitor the socket’s state
Flow of vESP8266RTask

1. Start
2. Enable USART RX pipe
3. Try to take USART mutex
4. Check USART RX pipe is readable
   - More to read
     - Get ESP8266 request
     - Give USART mutex
     - Task Delay
   - No more to read
     - Task delay and block
     - Take mutex
     - Take mutex failed
     - Try to take USART mutex
Flow of vESP8266TTTask

Start

Try to take USART mutex

Take mutex

Take mutex failed

Task delay and block

ESP8266 Command

SEND

Send Socket

CLOSE

Close Socket

Give USART mutex

Task Suspend
Select System Call

```c
int select( int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
```

select() and pselect() allow a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation (e.g., read(2) without blocking, or a sufficiently small write(2)).

Reference: Linux Programmer's Manual SELECT(2)
Select System Call Cont.

- **readfds** will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file).
- **writefds** will be watched to see if space is available for write (though a large write may still block).
- **exceptfds** will be watched for exceptions.
- **nfds** is the highest-numbered file descriptor in any of the three sets, plus 1.
- **timeout** argument specifies the interval that select() should block waiting for a file descriptor to become ready.

Reference: Linux Programmer's Manual [SELECT(2)]
Select System Call Cont.

- On **success**, select() and pselect() return the **number of file descriptors contained in the three returned descriptor sets** (that is, the total number of bits that are set in readfds, writefds, exceptfds) which may be zero if the timeout expires before anything interesting happens.

- On **error**, -1 is returned, and errno is set to indicate the error; the file descriptor sets are unmodified, and timeout becomes undefined.

Reference: Linux Programmer's Manual `SELECT(2)`
**fd_set**

- **Linux/include/uapi/linux/posix_types.h**
  
  ```c
  typedef struct {
      unsigned long fds_bits[ (__FD_SETSIZE / (8 * sizeof(long))) ];
  } __kernel_fd_set;
  
  typedef __kernel_fd_set fd_set;
  ```

- **Linux/include/linux/types.h**
  
  ```c
  typedef uint64_t fd_set;
  ```

  I make it as the data type of `uint64_t`!!!

| Bits Array | fd=0 | fd=1 | fd=2 | fd=3 | fd=4 | fd=5 | fd=6 | ...
|------------|------|------|------|------|------|------|------|------|

=> Each bit of fd_set corresponds to one file descriptor in order.
Select System Call Implementation

Go through each socket whose file descriptor \( fd \) is < \( nfds \)

- The \( \_\text{readfdfs} \) is not NULL and the current \( fd \) is interested
  - Check the \( fd \) is ready to be read
    - Yes: Increase \( count \)
    - Not: Go to \( \_\text{readfdfs} \)

- The \( \_\text{writefdfs} \) is not NULL and the current \( fd \) is interested
  - Check the \( fd \) is ready to be written
    - Yes: Increase \( count \)
    - Not: Go to \( \_\text{writefdfs} \)

- The \( \_\text{exceptfdfs} \) is not NULL and the current \( fd \) is interested
  - It is a dummy function

Return \( count \)
Assemble Parts Together
Overall Flow Diagram

Booting Flow

Start

Setup LEDs and USART2 peripherals

Initial ESP8266 Driver

Create Micro HTTP Server task

FreeRTOS task scheduler

Check ESP8266 state

Add routes

Initial Micro HTTP Server

Run Micro HTTP Server

Not linked

Linked

Get interface IP
Demo
GET
/
HTTP/1.1
Host: 192.168.1.100
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:49.0) Gecko/20100101 Firefox/49.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Hello!

GREEN LED= 0
ORANGE LED= 0
RED LED= 1
BLUE LED= 1

POST
/led
HTTP/1.1
Host: 192.168.1.100
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:49.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.1.100/led
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 29
Reference

- FastCGI https://en.wikipedia.org/wiki/FastCGI
- NSAPI
  https://en.wikipedia.org/wiki/Netscape_Server_Application_Programming_Interface
- Django & Twisted by Amber Brown @ PyCon Taiwan 2016 https://www.youtube.com/watch?v=4b3rKZTW3WA
- eserv https://code.google.com/archive/p/eserv/source
- tinyhttpd http://tinyhttpd.cvs.sourceforge.net/viewvc/tinyhttpd/tinyhttpd/
- GNU Libmicrohttpd https://www.gnu.org/software/libmicrohttpd/
Thank you ~
and
Q & A