POLITECNICO DI TORINO

Collegio di Ingegneria Informatica,
del Cinema e Meccatronica

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Integration and Development of a Dynamic Web Server on Embedded Microcontrollers

Relatore:
Prof. Luciano Lavagno

Candidato:
Ramon Messinetti

Relatore Aziendale:
Giovanni Di Sirio

Luglio 2018

Integration and Development of a Dynamic Web Server on Embedded Microcontrollers
RAMON MESSINETTI
Computer Engineering
Politecnico di Torino

Abstract

Nowadays, in the IT field, is very common to hear someone pronouncing the phrase "Internet of Things". This term was coined by information technology researchers but it has widely spread into mainstream public view only more recently. Internet of Things, in the more general technical meaning, could be defined as the ability of network aware devices to collect data, through sensors, and share them across Internet. In this network aware environment is present a figure of interest, for the scope of this thesis work, called embedded web server. Basically, an embedded web server is just a little part of a full internet of things infrastructure. This kind of server could be a valid alternative to build an entire Internet of Things infrastructure, especially in case of light applications and in presence of hardware and/or code size constraints.

In parallel, the embedded market is in continuous growth and more powerful microcontrollers are available with cheaper prices. Many electronics and semiconductor manufacturers, like STMicroelectronics, provides different families of microcontrollers with various memory sizes, IP modules and features making these technologies suitable for a wide range of applications. All the considerations described above are the main reason for the great usage of embedded web servers on microcontrollers, especially in real-time and low cost environments.

Considering these background informations, Giovanni Di Sirio, the tutor during my internship period at STMicroelectronics in Arzano (NA), assigned me to the project of integrating and extending an HTTP web server on a STM32F746G-DISCO platform. In order to achieve these tasks i had to integrate Chibios, a Real Time Operating System developed by Giovanni Di Sirio, with two main software components. These components were essentially an open source TCP/IP stack called LwIP, used for the establishment of a persistent internet connection, and a generic FAT file system module known as FatFS used in order to give a file organization to a microSD card.

ChibiOS is an open-source ready to user enviroment for embedded applications. Its main qualities are performance, small size and easily portability. ChibiOS does not refer to just an RTOS scheduler but to a composition of embedded elements of an overall architecture, this is the principle behind the concept of embedded system. The Hardware Abstraction Layer, composed of high level and low level device drivers, provides the same interface for different hardware specific function implementations. Furthermore an Operating System Abstraction Layer makes the HAL completely RTOS aware.

LwIP is a light-weight implementation of the TCP/IP protocol suite that was originally written by Adam Dunkels but now is being actively developed by a team of developers distributed world-wide headed by Kieran Mansley. Its main feature is the limited RAM usage while still having a full scale TCP, this makes LwIP suitable for use in embedded systems with tens of kilobytes of free RAM.

FatFS is a FAT/exFAT file system developed for the domain of embedded systems. This module has been conceived as totally separated from the disk I/0 layer, this design choice makes FatFS portable and independent from the platform. This feature makes FatFS suitable for application based on small microcontrollers with few resources and small memory availability.

The first task of this project, following the integration phase, was to reach the goal of storing and managing the web server from an on board microSD card mounting FatFS. The basic version of this web server was initially managed by a ROM file system making it not suitable for an embedded environment because of its memory requirements.
The second main task was to extend the basic static web server into a dynamic one. In particular the general structure is approximately a three level architecture adopting the CGI mechanism, a standard protocol that provides programs running on a server with the goal of generating web pages dynamically. I have also implemented an exportable algorithm that provides a file handling mechanism that could be common for each and every cgi task written in C.

A performance analysis has been verified. From this study emerges that the performance is directly influenced by the size of the read buffer used for the open/read operation of a file stored inside the on-board microSD. Having a too much large buffer involves in an inappropriate memory burn that must be avoided in an embedded environment, but on the other hand a small buffer will increment the execution time. Furthermore the programming language adopted for the CGI process implementation is a key choice in terms of execution time. As a result of this analysis i have considered C language as the most appropriate for this scope.

The conclusion of my thesis work has produced a ready to use environment in order to build a full customizable web server. Furthermore a personal dynamic web server have been provided, with the particular feature of storing and handling files directly from an on-board microSD card. The qualities of this HTTP server are very valuable being in line with the requests of the actual market.

In my opinion, a future challenge could be to increment the applicability of this thesis work by integrating a WiFi module with the current web server. Furthermore it could be useful to extend the ready to use environment, that i have provided, by implementing an abstract interface for different models of WiFi modules. With these upgrades the connection won't be limited by a physical linkage, allowing it to be applicable in many other wireless domains.

Acknowledgements

Ringrazio la mia famiglia, i miei amici e tutte le persone che mi sono state vicine in questo percorso formativo.

Un particolare ringraziamento anche al Professor Luciano Lavagno e al relatore aziendale Giovanni Di Sirio che grazie ai loro insegnamenti e consigli hanno permesso la realizzazione di questo lavoro di tesi.

Ramon Messinetti, Napoli, Luglio 2018

Contents

List of Figures

1

Introduction

1.1 Background

Today the Internet has become ubiquitous, has touched almost every corner of the earth, and is affecting human experience in unimaginable ways. We are entering an era of the "Internet of Things". This term has been defined by different authors in many different ways. Vermesan defines the Internet of Things as simply an interaction between the physical and digital worlds, the digital world interacts with the physical world using a plethora of sensors and actuators [1]. Peña-López defines the Internet of Things as a paradigm in which computing and networking capabilities are embedded in any kind of conceivable object, we use these capabilities to query the state of the object and to change its state if possible [2]. This concept finds myriads applications in different domains but on the other hand it is deserved to remember that with the connectivity increase comes several concerns. One of the most clear issues is that Internet of things will create immense amounts of data, this behavior will weight on the Internet and forces to develop an efficient and complex infrastructure.

In this network aware environment is present a figure of interest, for the scope of this thesis work, called embedded web server. By definition an embedded web server is essentially an HTTP server, like its name lets us imagine, with the feature of being embedded inside a microcontroller. The main purpose of this entity is to be used remotely, over the Internet, to inspect and/or control the various states of a microcontroller. Upon an HTTP request, the server provides a web page showing the needed informations. The user could interact with a web server by viewing an html page through any popular browser and changing the device's settings with an HTTP request. A typical application is to access physical sensors informations remotely (temperature, humidity, light level, presence of carbon monoxide), as well as controlling functionalities remotely that make things happen in the physical world (switch on a LED, trigger an alarm, switch on an emergency exhaust fan). Considering that an embedded web server is just a little part of a full Internet of Things infrastructure, its applications are usually of limited complexity. Main advantages of an embedded server configuration are the total costs, low power context and physical size (could be just a microcontroller plus a wifi or ethernet chip). On the other hand the main constraints are the memory size and the capability of working in a real time environment.

1.2 Scope

Taking into account the informations previously mentioned, Giovanni Di Sirio, the tutor during my internship period at STMicroelectronics in Arzano (NA), assigned to me the project of integrating and extending a basic HTTP web server on a STM32F746G-DISCO. The main software components that i had to integrate together with the Real Time Operating System, developed by Giovanni himself, called ChibiOS, are an open source TCP/IP stack called LwIP and a generic FAT file system module known as FatFs. The LwIP stack was necessary in order to establish a persistent client/server connection and for its functionalities belonging to transport and internet levels. Furthermore a FAT file system module FatFS have been used for the the file/folder organization of the microSD mounted on the STM32F746G-DISCO.

The web server was initially managed by a ROM file system, this behavior could be limiting in an embedded environment because of the high usage of ROM and RAM memory. For this reason the first task to complete was the transition from a ROM based web server to a micro-SD based one. Basically, all the files beloging to the web server will be stored and managed inside the microSD mounting the file system FatFS. The correct integration of the different software components was essential in order to achieve this functionality. Once realized this scope the extension of the basic static web server into a dynamic one was requested. In particular the general structure is a three level architecture adopting the CGI mechanism. An exportable algorithm have been implemented which provides a file handling mechanism that could be applied for whichever cgi task written in C.

1.3 Outline

This thesis work is composed of three main chapters:

- **Theory:** all the theoretical concepts that are requested in order to understand better the project developed. In particular a detailed study of the software component used like Chibios, Lwip and Fatfs.

- **Methods:** it is a description of the methodologies, strategies and approach adopted in order to realize the task appointed. A brief explanation of how to setup the environment correctly is included.

- **Result:** a report that includes a description of the results achieved and a technical analysis.

2
Theory

This chapter is composed of all the theoretical concepts on which the development phase of this project have been based on. First of all it is necessary to describe the architecture and all the main features provided by the RTOS ChibiOS.
Furthermore, it is present a brief description of the adopted TCP/IP stack, called LwIP, with all its functionalities and stack organization details.
Finally the concept of file system is explained underlining the case of FatFS that is a generic FAT/exFAT filesystem module for small embedded systems.

2.1 ChibiOS

ChibiOS is a complete development environment for embedded applications including RTOS, an HAL, peripheral drivers, support files and tools, it also integrates external Open Source components in order to offer a complete solution for embedded devices [3]. It is doverous to underlying that ChibiOS does not refer to just an RTOS scheduler but to a composition of embedded elements of an overall architecture, this is the principle behind the concept of embedded system.

The application model corresponds to a single application with multiple threads.
This means:

- The runtime environment is **trusted**, the application does not need to defend from itself[3].

- The applications can run with multiple threads being an integrating part of the application and they share the same address space.

- Application and Operating System must be considered as a single memory image, in other words an unique program.

- There is no concept of "application load" except a bootloader is used that can take care of that [3].

Starting from above of the architecture in figure 2.1, ChibiOS is composed of the following blocks:

- **Application:** It is the part intended for the user code, ChibiOS provides a simple template of the main() function.

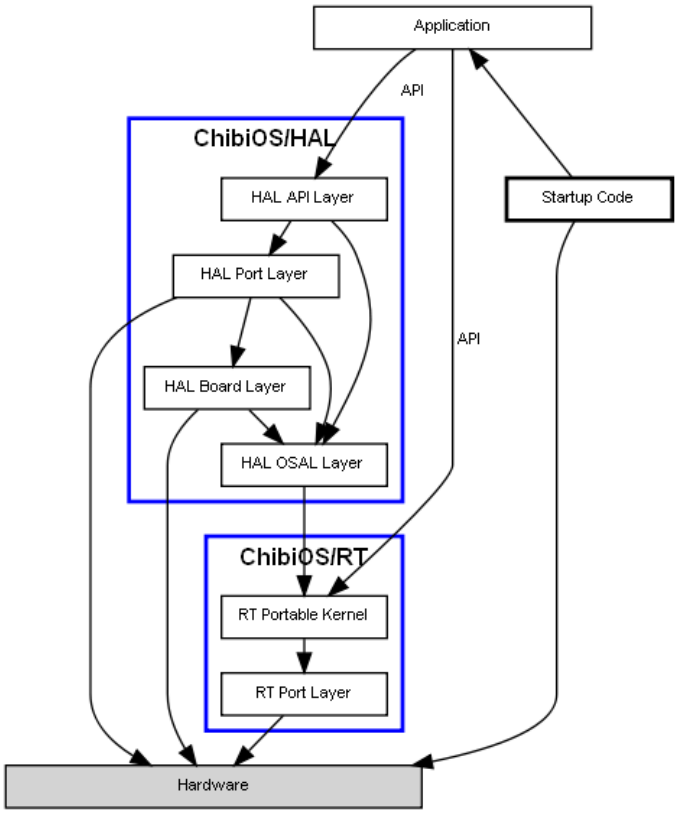

Figure 2.1: ChibiOS Organization

- **Startup Code:** This part of code is the one executed after a reset. The startup code, in short words has the responsibility of the correct initialization of core and stack. Furthermore it must invoke the main() function.

- **ChibiOS/HAL:** HAL is the acronym for Hardware Abstraction Layer, a set of device drivers for the peripherals most commonly found in micro-controllers. In the above architecture it is clear that RT does not need HAL and can be used alone if HAL is not required. On the other hand, in this architecture HAL uses RT's services through the OSAL but it could use another RTOS or even work without RTOS by implementing an OSAL over the bare metal machine.

- **ChibiOS/RT:** There are two RTOS solutions with different features:

  - **RT:** the fastest RTOS solution for embedded real time systems, extremely high performance with a very complex set of features [3].

  - **NIL:** NIL has been created with the idea to bring RTOS functionalities to very small devices and yet to be a full-fledged RTOS [3].

2.1.1 ChibiOS HAL

The HAL component is meant to be an abstraction layer between the application and the underlying micro-controller hardware. HAL offers an high level API for accessing common MCU peripheral like GPIO, ADC, SPI and so on and also take care of clocks-related and board-level initializations [4]. This abstraction layer between application and underlying hardware makes the firmware development faster and comfortable allowing a more objective-oriented approach like in C++.
In general the application doesn't know anything about hardware inner details, they are hidden but not unreachable. The main feature provided is for sure its portability on a wide range of micro-controllers.
HAL layer is itself composed of multiple layers, precisely it has four layers:

- **HAL API Layer:** This is the layer that exports the API used by the main application. This layer is perfectly portable, there are no dependencies on any specific HW architecture. In this layer there are the portable Device Drivers. Portable drivers take care of:
  - Driver API.
  - check on API parameters.
  - Driver state machine handling and checks using assertions.
  - Low level driver invocation for inner functionality.
- **HAL Port Layer:** This is the device driver implementations for a specific micro-controller or family of micro-controllers, technically called low level drivers [4]. The user application must not interact directly with the methods of this layer, indeed it must call the APIs provide by the high level device drivers. The linkage between the low level driver of the port layer and high level drivers is achieved with header inclusions inside the makefile.
- **HAL Board Layer:** This module is composed of all the specific informations of the board mounting a specific micro-controller. Basically it is composed of all the dependencies between a specific board with the HAL layer [4].
  - Name of the Board.
  - MCU type mounted on the board.
  - Clock organization, parameters and details.
  - Board initializations.
- **HAL OSAL Layer:** It is known as the Operating System Abstraction Layer. The architeture of ChibiOS requests that the HAL layer in order to implement

its functionalities has to lean on some RTOS service. The procedure of accessing to the RTOS services is implemented inside this layer in order to not lock the HAL layer to a particular real time operating system. It is clear that RT does not need HAL and can be used alone if HAL is not required but on the other hand, in this architecture HAL uses RT's services through the OSAL but it could use another RTOS or even work without RTOS by implementing an OSAL over the bare metal machine [3].

The ChibiOS/HAL was designed in C and also if the concept of objects is not present in this programming language it could be considered very object-oriented: indeed drivers are represented by a struct and each application programming interface associated to a driver requires its pointer. As this structure contains almost every information related to driver, API doesn't require a long list of parameters [5]. Let's consider the example of an SPI module. In this figure 2.2 we can see the body of a structure renamed as SPIDriver that represent an SPI driver.

```
struct SPIDriver {
  /**
   * @brief Driver state.
   */
  spistate_t                state;
  /**
   * @brief Current configuration data.
   */
  const SPIConfig           *config;

  /* End of the mandatory fields.*/
  /**
   * @brief Pointer to the SPIx registers block.
   */
  SPI_TypeDef               *spi;
  /**
   * @brief Receive DMA stream.
   */
  const stm32_dma_stream_t  *dmarx;
  /**
   * @brief Transmit DMA stream.
   */
  const stm32_dma_stream_t  *dmatx;
  /**
   * @brief RX DMA mode bit mask.
   */
  uint32_t                  rxdmamode;
  /**
   * @brief TX DMA mode bit mask.
   */
  uint32_t                  txdmamode;
};
```

Figure 2.2: SPI Driver code

Hardware-depending configurations are also grouped in a structures still designed as object called configurations [5]. One of the most advantages is that the HAL

drivers are all defined as Finite State Machines, indeed thanks to this organization we can understand in which state the driver should be and allow to check through assertions a driver state entering in driver method. The SPI driver as all the other HAL drivers used, is completed with the device drivers functions, when the Init() method is invoked during the HAL initialization phase.

2.1.2 ChibiOS RT Mailboxes

This subsection explains briefly the concept of mailbox and its implementation because of its usage during the extension of the web server functionalities. In ChibiOS/RT a Mailbox object is a circular queue of messages that can be posted and fetched from both thread and ISR contexts [6]. Mailboxes, differently from synchronous messages, are perceived for messages flowing in direction in FIFO order and in a asynchronous organization. Mailboxes thanks to their flexible nature in the communication are applicable in a lot of fields, for example:

- Thread/ISR to Thread/ISR communication, multiple senders and multiple receivers are supported [6].
- Set of pre-configured objects.

Mailboxes use the same msg_t types already seen in the synchronous messages chapter but without the limitation of the reserved 0, -1 and -2 constants [6]. Mailboxes are basically objects composed of a circular buffer, typically it is implemented as an array of msg_t. Obviously the buffer of a mailbox have maximum configurable size and it could be filled up until its full capacity. If we suppose that the circular array is full, an attempt at posting a message will cause a context switch of the thread from running to waiting until a new slot to becomes free. The mutual exclusion of the fetch and post mechanism is guaranteed by two counting semaphores.

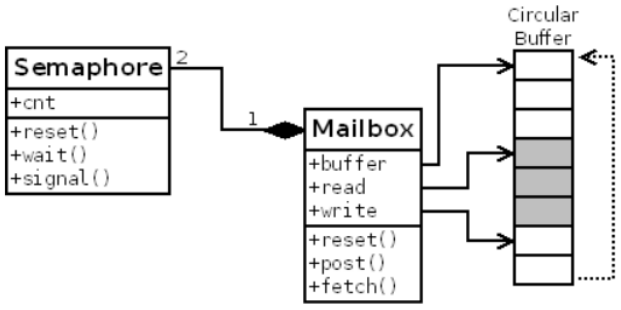

Figure 2.3: Mailbox structure

2.1.3 ChibiOS RT Events

Here following, there's a description of the working principles and implementation details of the event handling provided by ChibiOS. A possible use case of this fea-

ture is when a device drivers must perform events when I/O peripherals have to send a generic notification to upper application. ChibiOS provides this feature but while in the other operating system the mechanism of event flags is trivial, their implementation in ChibiOS is a little it more challenging. Events could be used, for example, when a task must be suspended until one or different conditions are satisfied, so in general a thread could check synchronously for events that have been notified in an asynchronous way.

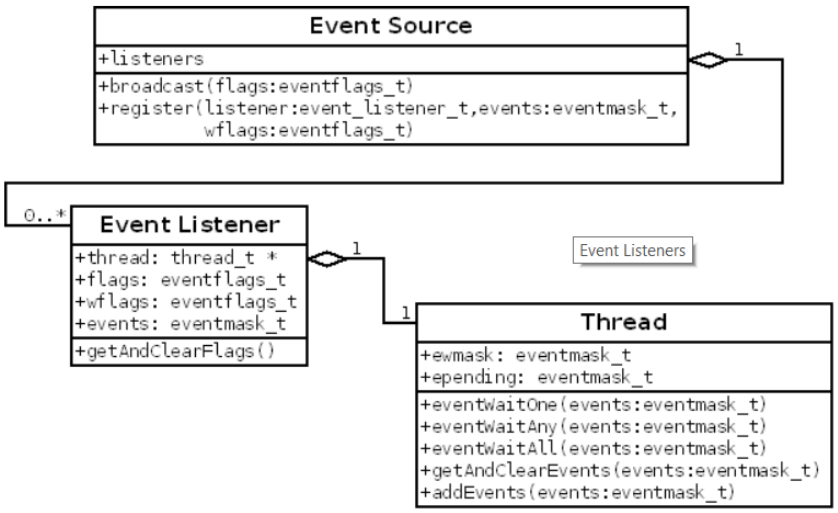

Figure 2.4: Event UML Representation

As we can see in 2.4 there are three classes of objects:

- **Event Source:** Event Sources are objects with the assignment of broadcasting a generic event to the environment. Two different methods have been provided to event sources:

  - **Register**:
    This method allows an instance of this object to listen the notification of or more events.

  - **Broadcast**:
    This function must be called whenever an event source must broadcast to all tasks registered on it that an event has occurred.

- **Event Listener:** For each and every event source one or more event listeners can be registered. Basically an event listener is connected to a specific thread. As we can see from the UML scheme in figure 2.4 there's a many-to-many relationship between threads and event sources.

- **Thread:** objects have an event listener for each event source they are listening. Its attribute ewmask is the mask of events the thread is interested in and

epending is the mask of the events waiting to be served by the thread [7].

2.2 LWIP

2.2.1 Introduction

LwIP is a light-weight implementation of the TCP/IP protocol suite that was originally written by Adam Dunkels at the Computer and Networks Architectures (CNA) lab of the Swedish Institute of Computer Science but now is being actively developed by a team of developers distributed world-wide headed by Kieran Mansley [8].
The focus of the lwIP TCP/IP implementation is to reduce the RAM usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM. Since its release, lwIP has spurred a lot of interest and is today being used in many commercial products, it has been ported to multiple platforms and operating systems and can be run either with or without an underlying OS [8].
LwIP is designed to be completely modular. The core stack is an IP implementation, on top of which the user can choose to also add TCP, UDP, DHCP, and many other protocols, including various features of each of these protocols (for example, IP fragmentation and reassembly). More features comes at the cost of increased code size and complexity, and this is fully tunable to the user's needs.
Further, since its release, LwIP has spurred a lot of interest and is today being used in many commercial products, it is designed to operate with or without an OS, and with or without support for threads. It works on 8-bit microprocessors and 32-bit microprocessors, and supports both little- and big-endian systems.

2.2.2 TCP/IP stack

Before describing the TCP/IP stack it is deserved to give a short mention about the OSI model.
The Open Systems Interconnection model (OSI model) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to its underlying internal structure and technology [9]. Its main task in to be flexible and applicable to many communication systems with a proper defined protocol. This model have been realized in order to split the communication into different abstraction layers. The first version of this stack was composed of seven layers. Although all the versions of this stack are designed so that a layer provides informations to layer above it and provide services to the layer below it. The OSI model is basically composed of these layers. Starting from the lowest one we can find the link layer, handles the communication of data that stays in a single network segment; the internet layer, providing connection between two separate networks. Furthermore there are the transport layer handling point to point communication and the application layer, which is the higher level of the stack and where are located the user applications.

For example, a layer that provides error-free communications across a network provides the path needed by applications above it, while it calls the next lower layer to send and receive packets that comprise the contents of that path [9]. Two instances at the same layer could be imagined as connected by an horizontal link in that layer.

OSI Model		
Layer		Protocol data unit (PDU)
Host layers	7. Application	Data
	6. Presentation	
	5. Session	
	4. Transport	Segment (TCP) / Datagram (UDP)
Media layers	3. Network	Packet
	2. Data link	Frame
	1. Physical	Bit

Figure 2.5: Generic OSI Stack

The **Internet protocol suite** also known as TCP/IP stack is a stack model composed of particular protocols of communication that usually finds place in the Internet fields or more generally in computer networks. It is a particular case of the OSI stack described before, clearly the main protocols adopted in this suite are the Transmission Control Protocol for the transport layer and the Internet Protocol for the network layer.
The TCP/IP stack provides a consistent host-to-host communication specifying how data should be managed, transmitted, received and routed.

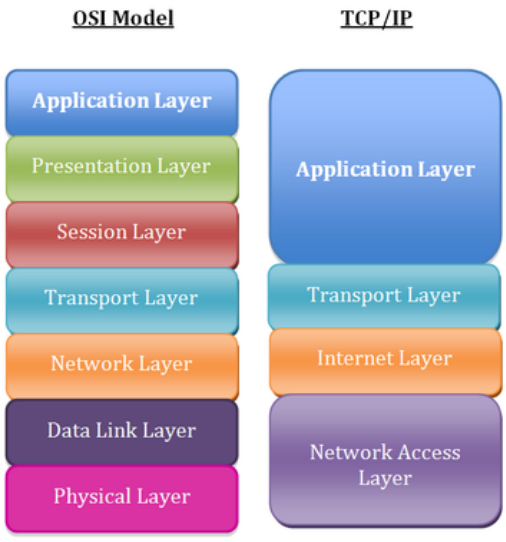

Figure 2.6: TCP/IP Stack

2.2.3 LWIP stack organization

The LwIP stack have been designed and implemented accordingly with the general TCP/IP stack layered organization. Each protocol is implemented as its own module, with a few functions acting as entry points into each protocol. Even though the protocols are implemented separately, some layer violations are made, in order to improve performance both in terms of processing speed and memory usage [10]. LwIP consists of several modules. Different modules have been designed with the purpose of implementing TCP/IP protocols (like IP, UDP and TCP) but also support modules are available. The latter provides mechanisms for the management of memory and buffer subsystems, and network interface functions. In general they could be intended as an operating system layer. The main feature that makes LwIP widely spread in embedded systems is its modular architecture and the support for many communication protocols. This modular structure could allow the user to configure the stack and avoiding to load non necessary code in flash. In figure 2.7 are represented all protocols implemented for a given layer, in particular:

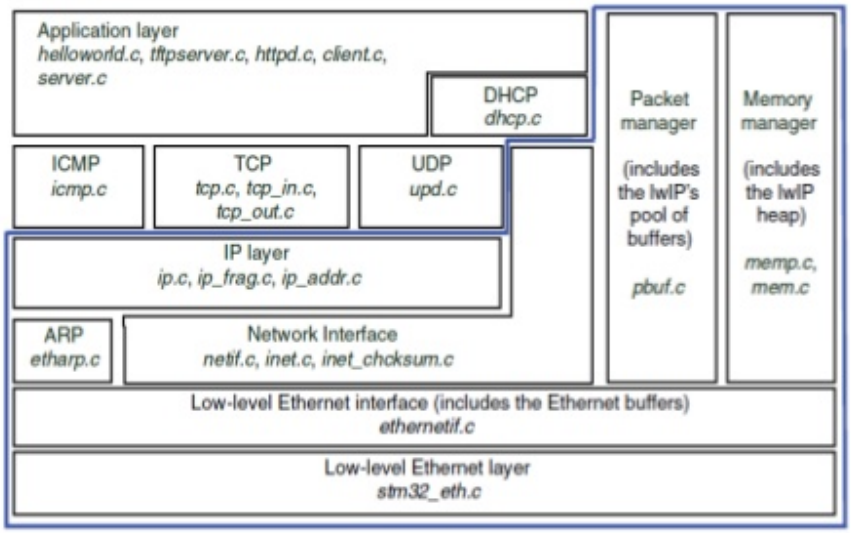

Figure 2.7: LWIP Stack

- **Link and network protocols:**
  - **ARP:** it is a protocol of the data-link layer with the purpose of translate a MAC address into an IP address.
  - **IPv4:** it is the widely spread network-layer protocol with particular application in the internet.
  - **IPv6:** the later version of IPv4, where the IP address passes to 128 bits.

  - **ICMP:** a control protocol for IP.
  - **IGMP:** a management protocol for multicast groups in IP.

- **Transport protocols:**
  - **UDP:** a connectionless protocol that differently from the tcp ip protocol doesn't provide any reliability mechanisms.
  - **TCP:** a connection-oriented protocol of the transport layer.

- **High-level protocols:**
  - **DHCP:** used when it is needed to acquire an IP address from a server.
  - **AUTOIP:** acquisition of an IP address without a server.
  - **SNMP:** used to check the network conditions.
  - **PPP:** it is known as point to point protocol that provides a direct connection between two nodes.

2.2.4 LWIP Process Model

A process model is by definition how the system has been organized into different processes for the implementation of a protocol. Different basic models are available, one of the most used is to give a personal process session to each protocol. This model is very restrictive in terms of layering and for this reason it is essential to define the communication points between the protocols. While this approach has its advantages such as protocols can be added at runtime, understanding the code and debugging is generally easier, there are also disadvantages [10]. Another serious drawback is that for each layer crossed a process must perform a context switch. Let's consider, for example, the case of a TCP segment arrive. Basically this involves to three context switches, considering from the lowest layer of the network interface, to the IP module and finally to the TCP process and application process. This involves an huge effort in terms of overhead for each operating systems, indeed a context switch is expensive in computational terms. The other spread process model is the one that "stores" communication protocols inside the kernel of the operating system. In this case the application processes will communicate with different protocols through system calls. Thanks to the peculiarity of communication protocols that are not strictly divided the previous drawback of context switches have been solved.
LwIP uses a process model in which all protocols reside in a single process and are thus separated from the operating system kernel [10]. User programs could reside in the LwIP process or in a stand-alone processes. The communication between LwIP

process and the user application coulb be achieved through function calls (in case the application program shares a task with LwIP) or thanks to higher level abstract API. Summarizing, LwIP have been implemented in the user space task than in the kernel of the OS. This choice has its advantages and disadvantages. The main advantage is the portability across different OS. Since lwIP is designed to run in small OS the delay produced by the wait overhead in case of the LwIP process have been swapped or paged out to disk is removed, indeed in a small operating system there isn't the swap mechanism and virtual memory concept. On the other hand the main problem is the waiting time for a scheduling quantum before getting a chance to service requests.

2.2.5 LWIP API Overview

The Lwip stack is implemented as a light version of a standard TCP/IP stack. BSD sockets usually copies data to be sent from the application program in buffers of the TCP/IP stack, for this reason BSD socket APIs are not a solution applicable to this stack due to the high level of abstraction provided. This kind of sockets requires this storage mechanism because the application and the TCP/IP stack are localized in different protection domains. Usually the application program is a user process and the TCP/IP stack is located in the OS kernel. This copy mechanism causes extra memory possession devoted to the allocation of the copy doubling the amount of memory required per packet. The lwIP API, although similar to the BSD APIs, operates at a more lower level and it doesn't require the copy mechanism described before, indeed the application program can handle the internal buffers directly. Furthermore a BSD compatible layer have been implemented because of the widely spread usage of the BSD socket.

2.2.5.1 Structure and implementation details

From the application's point of view, data handling in the BSD socket API is done in continuous memory regions. This is convenient for the application programmer since manipulation of data in application programs is usually done in such continuous memory chunks [10]. LwIP doesn't use this mechanism because it would be not advantageous, since it stores data in buffers where they are segmented in smaller chunks of memory. This behavior would result in a huge processing time and memory overhead due to the fact that data would have to be stored in a continuous region of memory before being passed to the upper application. For the previous considerations lwIP API allows the user program to handle data directly in specific partitioned buffers in order to avoid performance and memory loss. The APIs provided by LwIP are implemented in two parts based on the process model of the TCP/IP stack. One part have been implemented as a library linked to the application program, and the remaining part implemented inside the TCP/IP process. These two block can communicate using the interprocess communication mechanisms provided by the LwIP operating system emulation layer. These three IPC mechanisms are provided: message passing, shared memory and semaphores. In case of an operating systems that do not natively support these features, the operating system emulation layer emulates them. Three Application Program's Interfaces in order to communicate with

the TCP/IP code have been provided:

- **Callback or Raw API.**
- **Netconn API.**
- **Socket API.**

2.2.5.2 Raw API

The Raw API is based on the native API of LwIP. It is used to develop callback-based applications. When initializing the application, the user needs to register callback functions to different core events (such as TCPSent, TCPerror).
The callback functions will be called from the LwIP core layer when the corresponding event occurs. The raw API gives the best performance since it does not require thread-changes and support zero-copy both for TX and RX.

	API function	Description
TCP connection setup	tcp_new	Creates a new TCP PCB (protocol control block).
	tcp_bind	Binds a TCP PCB to a local IP address and port.
	tcp_listen	Starts the listening process on the TCP PCB.
	tcp_accept	Assigns a callback function that will be called when a new TCP connection arrives.
	tcp_accepted	Informs the LwIP stack that an incoming TCP connection has been accepted.
	tcp_connect	Connects to a remote TCP host.
Sending TCP data	tcp_write	Queues up data to be sent.
	tcp_sent	Assigns a callback function that will be called when sent data is acknowledged by the remote host.
	tcp_output	Forces queued data to be sent.
Receiving TCP data	tcp_recv	Sets the callback function that will be called when new data arrives.
	tcp_recved	Must be called when the application has processed the incoming data packet (for TCP window management).
Application polling	tcp_poll	Assigns a callback functions that will be called periodically. It can be used by the application to check if there is remaining application data that needs to be sent or if there are connections that need to be closed.
Closing and aborting connections	tcp_close	Closes a TCP connection with a remote host.
	tcp_err	Assigns a callback function for handling connections aborted by the LwIP due to errors (such as memory shortage errors).
	tcp_abort	Aborts a TCP connection.

Figure 2.8: Raw API

2.2.5.3 Netconn API

The netconn API is a sequential API designed to make the stack easier to use (compared to the event-driven raw API) while still preserving zero-copy functionality. The Netconn API is a high-level sequential API which has a model of execution based on the blocking open-read-write-close paradigm.
To function correctly, this API must run in a multi-threaded operation mode where there is a separate thread for the LwIP TCP/IP stack and one or multiple threads for the application.
All packet processing (input as well as output) in the core of the stack is done inside a dedicated thread (aka. the tcpip-thread). Application threads using the netconn API communicate with this core thread using message boxes and semaphores.

API function	Description
netconn_new	Creates a new connection.
netconn_delete	Deletes an existing connection.
netconn_bind	Binds a connection to a local IP address and port.
netconn_connect	Connects to a remote IP address and port.
netconn_send	Sends data to the currently connected remote IP/port (not applicable for TCP connections).
netconn_recv	Receives data from a netconn.
netconn_listen	Sets a TCP connection into a listening mode.
netconn_accept	Accepts an incoming connection on a listening TCP connection.
netconn_write	Sends data on a connected TCP netconn.
netconn_close	Closes a TCP connection without deleting it.

Figure 2.9: Netconn API

2.2.5.4 Socket API

The BSD compliant layer offers the possibility to use standard BSD socket APIs. These APIs are sequential and internally developed on top of the netconn layer.

2.2.6 Buffer and memory organization

The memory and buffer management system in a communication system must be prepared to accommodate buffers of very varying sizes, ranging from buffers containing full-sized TCP segments with several hundred bytes worth of data to short ICMP echo replies consisting of only a few bytes [10]. Another feature provided is the possibility to store data that is not handled by the networking subsystem in ROM.

API function	Description
socket	Creates a new socket.
bind	Binds a socket to an IP address and port.
listen	Listens for socket connections.
connect	Connects a socket to a remote host IP address and port.
accept	Accepts a new connection on a socket.
read	Reads data from a socket.
write	Writes data on a socket.
close	Closes a socket (socket is deleted).

Figure 2.10: Socket API

2.2.6.1 LwIP buffer management

A pbuf is lwIP's internal representation of a packet, and is designed for the special needs of the minimal stack [10]. The pbuf have been conceived with the feature of allocating dynamic memory to hold packet data and for letting packet data reside in static memory. Pbufs can be unified in a set forming a list called pbuf chain, this feature allows the packet to be spread over several pbufs. Pbufs are of three types, PBUF RAM, PBUF ROM, and PBUF POOL.
Pbufs general structure is represented in figure 2.11.

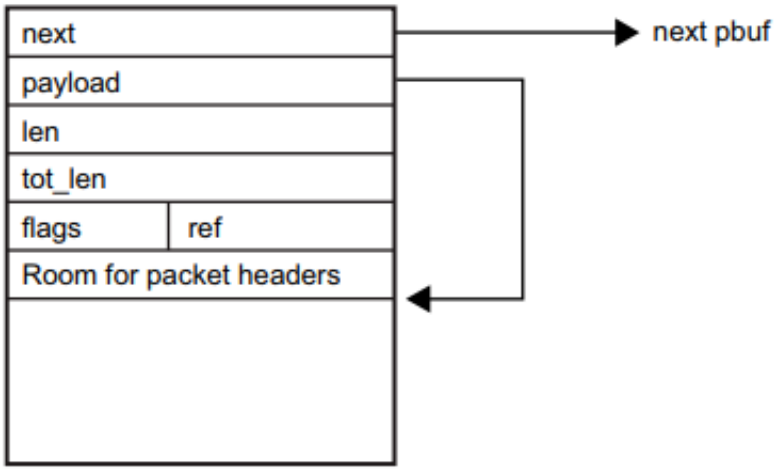

Figure 2.11: Pbuf General Structure

The three types have different uses. Pbufs of type PBUF POOL are generally used by network device drivers because the operations reserved for the allocation of a single pbuf are intrinsically fast and therefore appropriate for use in an interrupt handler. PBUF ROM pbufs are used when an application sends data that is located in memory managed by the application. This data may not be modified after the pbuf has been handed over to the TCP/IP stack and therefore this pbuf type main

use is when the data is located in ROM (hence the name PBUF ROM) [10].

2.2.6.2 LwIP memory management

The memory manager handles allocations and deallocations of contiguous regions of memory and can shrink the size of a previously allocated memory block [10]. In order to avoid that the networking system fills the total available space of the memory, has a dedicated segment of memory instead of using total memory size. Furthermore this mechanism ensures that the correct behavior of other running processes is never disturbed by the networking system in case of no more space available in memory. This organization on the other hand requires the usage of a track system of the allocated memory by using a small struct on the head of each memory block. The memory allocation takes place so that the first block with enough available space gets allocated. In case of deallocation the used flag is set to zero and in order to prevent fragmentation also the used flag of the next and previous blocks are controlled. Obviously if one of their flag is set to unused, the current block and the newest one are unified in a larger unique block.

2.3 FATFS

2.3.1 Introduction

FatFs is a FAT/exFAT file system developed for the domain of embedded systems. The FatFs module is entirely written in compliance with ANSI C that guarantees portability, furthermore it has been conceived as totally separated from the disk I/O layer. So its main advantage is to be portable and so independent of the platform. It can find place also in small microcontrollers with few resources like PIC, AVR, ARM and so on.

2.3.2 File system basics

A file system is used to have a file/folder oriented organization of a generic data. Without this component it could be impossible to distinguish two different informations, indeed they would be placed in a memory as one large chunk of data and retrieving the informations of the first or second flow would be impossible. So in general this mechanism allows to separate informations and to assign to them name and attributes for identification. Typically a file could be defined as a related collection of data. By definition a file system is the way and the rules created to manage these groups of information, different types of file systems are available. Each one has a different design, logical behavior and qualities such as security and performance. Usually each file system has its own field of application, for example ISO 9660 file system have been developed for optical discs. The concept of file system is strongly related to storage devices that uses different kind of media. Obviously the most common is the HDD, followed by flash memory, mmC, microSD card and optical discs. An user could be also use the main memory of a computer in order to create a temporary file system e. A file system is usually composed of three layers,

but the separation could be strongly explicit of more opaque. The higher layer of the fs is called logical file system and provides interaction with the user application. Basically it is made of different application program interface (API) for file handling like f_open(), _close() and f_read() and submits all to the layer below. The second layer is called Virtual File System, its purpose is to allow a customer to access uniformly different kinds of physical file systems. The last layer is called physical file system. This layer is linked to the physical operation provided by the underlying hardware, basically it handles physical blocks that must be written or read. This behavior is achieved by interacting with the device drivers of the storage device. Generally a disk could be divided in partitions, each of which has its own file system and a special sector, called MBR (Master Boot Record), used for the bootstrap. The MBR consist of the table of partitions (starting and ending address of each partition), where one of them could be labeled as "Active". On the bootstrap of the system, the BIOS access to the first sector of the MBR, loads its content in memory and finally executes it. The program loaded is composed of the following steps:

- Identifies the active partition.
- Reads in the first block (boot block) of the active partition.
- The program of the boot block loads the operating system stored in that partition.

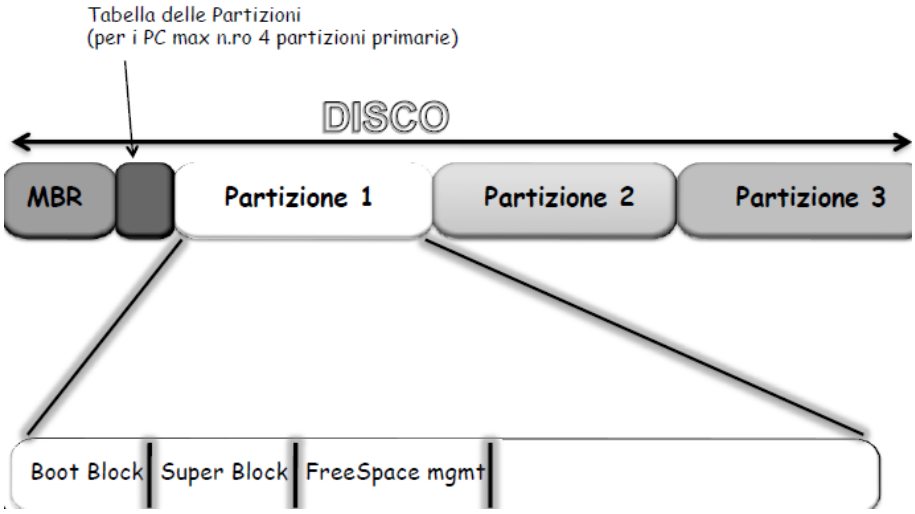

Figure 2.12: File System Layout

An Operating System uses chunks of dimension greater than the one of a generic sector of the disk. Each block consist of a series of consecutive sectors in order to increase the transfer efficiency.

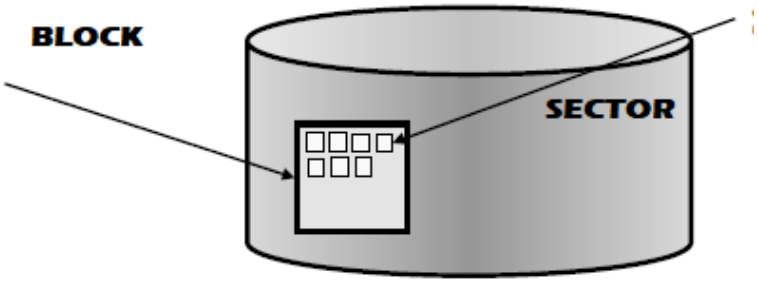

Figure 2.13: Block organization

2.3.3 FAT Basics

File Allocation Table (FAT) is a computer file system architecture and a family of industry-standard file systems utilizing it, FAT is a continuing standard which borrows source code from the original, legacy file system and proves to be simple and robust [12]. It has competitive performances even in lightweight applications, but obviously looses some performance, reliability and scalability as some modern file systems. It is used to facilitate the access and handling of files and folders providing different features like a way to time stamp when a file have been generated or modified. Furthermore there are API for file size identification but also other attributes likes whether a file is read-only or an information hiding mechanism when the file should be hidden in a folder, or file archiving at the next disk backup. An important feature is its compatibility with almost all of the most spread operating systems for personal computers, mobile devices and embedded systems. Accordingly to the disk drives evolution three major file system variants have been developed: FAT12, FAT16 and FAT32. The FAT evolutions have been always developed guaranteeing backward compatibility with existing or legacy software. Initially FAT file system was designed specifically for IBM PC mounting x86 processors. All the data structures present inside FAT file system will be stored on memory in little endian fashion. This means that if we want to mount the FAT on a platform working in big endian, every time it is needed to access FAT's data structures an endian conversion is required. Another problem consists in the fact that word data could be misaligned with respect to the word boundaries, so if a processor could not handle this case, it will need to access the data one byte at time. This means that accessing the FAT volume as simple byte array gives greater portability with respect to the case of considering word data as C structures.

2.3.3.1 FAT Master Boot Record

The Master Boot Record (MBR) occupies on one or more of the first sectors at the physical start of the device. The boot region of the MBR contains DOS boot loader code, which is written when the device is formatted (but is not otherwise used by the Dynamic C FAT file system) [13]. The boot region is followed by a partition table. It consists in four 16-byte entries which allow up to four partitions on the device. These Partition entries are composed of different important information like the starting and ending sector numbers of a specific partition and the partition type.

Furthermore there is also a field containing the total number of sectors present in a specific partition. Obviously if this number is set to zero, this means that the corresponding partition is empty and available.

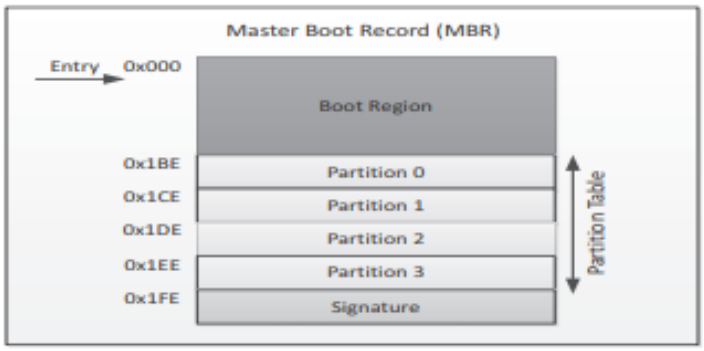

Figure 2.14: Fats MBR

2.3.3.2 FAT Partitions

The first sector of a valid FAT file system partition contains the BIOS Parameter Block (BPB), followed by the File Allocation Table (FAT), and then the Root Directory [13]. The figure below shows a device with two FAT partitions.

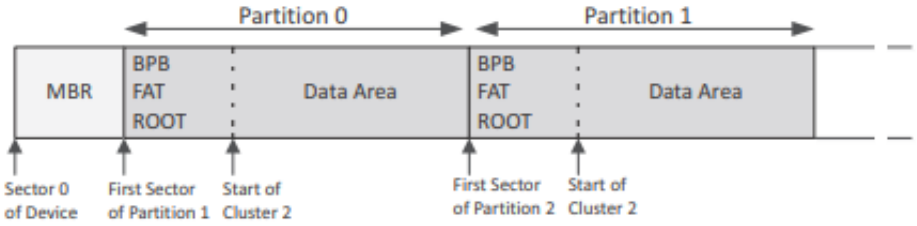

Figure 2.15: Fats Partitions

The fields of the BPB contain information describing the partition:

- The number of bytes per sector.
- The number of sectors per cluster.
- The total count of sectors on the partition.
- The number of root directory entries.

2.3.3.3 FAT Sector

Sector is the smallest unit of data block on the storage to read and write the storage device. The common sector size is 512 bytes and a larger sector size is sometimes used for some type of storage media. Each sector on the storage device is addressed

by a sector number assigned in order from top of the storage device. Since volumes are not that always placed at top of the storage, the sector number denotes the relative location origin from top of the volume and physical sector number denotes the absolute location origin from top of the storage device.
All the configuration variables of the FAT volume are stored inside a data structure called BPB (BIOS Parameter Block) This block is located inside the boot sector. The boot sector, often referred as Volume Boot Record or Private Boot Record, it is simply the first sector of the reserved area.

2.3.3.4 FAT Volume

FAT file system completes itself is called logical volume (or logical drive) [14]. The FAT logical volume is composed of four sections, each of them consisting of one or more sectors and mounted on the volume as follows:

- Reserved area (BPB).
- FAT area (allocation table for data area).
- Root directory area.
- Data area (contents of file and directory).

2.3.3.5 FAT Area

Another important element of the FAT module is the FAT area. Its main purpose is to define a linked list of the extents (cluster chain) of a file. Note that both directory and file is contained in the file and nothing different on the FAT. Indeed a folder is to all intents and purposes a file with a special field attribute that states its content is a directory table. The data area is divided into blocks composed of sectors called. Each item of FAT is associated with each cluster in the data area and the FAT value indicates the state of the corresponding cluster [14]. FAT items FAT[0] and FAT[1] are reserved so they are not associated to any cluster. Following FAT[2] corresponds to the item associated with the first cluster of data area. An important feature of FAT is its data redundancy, designed in order to avoid data looses due to an expected event like a damage. The access method to a generic FAT entry is a key concept. Basically FAT module could be seen as a generic integer array. Differently form a on-memory array, FAT is not stored on the continuous memory but is split in multiple blocks and then stored on the contiguous disk sectors. If we consider a FAT32 volume, its FAT entries will occupy 32 bits, but its 4 most significant bits are reserved, therefore only lower 28 bits are valid. On this basis, when loading a value from FAT entry of a FAT32 volume, the upper 4 bits needs to be and-masked. Image 2.16 shows the FAT usage of each FAT type.

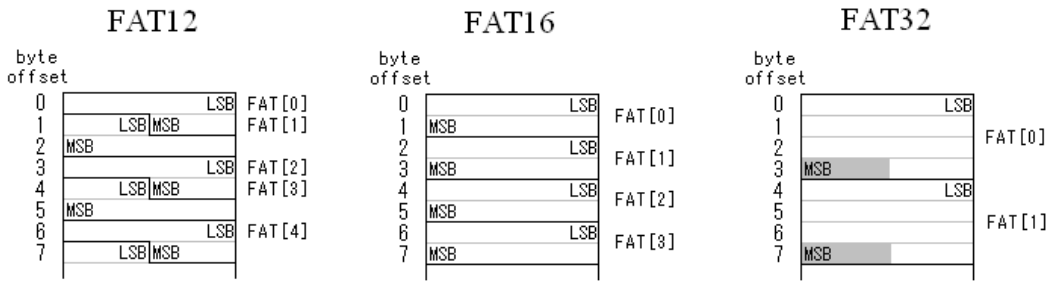

Figure 2.16: FAT types

2.3.3.6 FAT Root Directory Area

The root directory has 512 entries of 32 bytes each. An entry in the root directory is either empty or contains a file or subdirectory name, file size, date and time of last revision and the starting cluster number for the file or subdirectory.

2.3.3.7 FAT Data area

The data area takes up most of the partition. It contains file data and subdirectories. Data area of a partition must start at the second cluster.

2.3.4 FATFS Basics

FatFS is a generic FAT file system module for small embedded systems. The FatFS is written in compliance with ANSI C and completely separated from the disk I/O layer [15].
Therefore it is independent of the underlying hardware, and provides the following features:

- Windows compatible FAT file system.
- Very small footprint for code and work area.
- Many configuration options:
  - Multiple volumes availability (physical drives and partitions).
  - Multiple ANSI/OEM code pages including DBCS.
  - Long file name support in ANSI/OEM or Unicode.
  - Support for Real Time Operating Systems.
  - Multiple sector size support.

- Read-only, minimized API, I/O buffer and etc...
- FAT12, FAT16 and FAT32.
- Unlimited number of opened files concurrently considering always memory space availability.
- Up to 10 volumes.
- Configurable file size.
- Configurable volume size.
- Configurable cluster size.
- Configurable sector size.

2.3.4.1 FATFS Features

- **Duplicate file access:**
  FatFs module does not allow a duplicated file access in default. In particular it is allowed to open concurrently a file only in read mode. On the other hand duplicated open in write mode to a file is always nor permitted and open file must not be renamed or deleted, otherwise the FAT structure mounted on the volume could collapse.

- **Reentrancy:**
  File operations to the different volumes are always reentrant, this means that this method can be interrupted in any case an then safely recalled before its previous invocations completes it execution and can work. In case of file operations to the same volume the reentrancy is not ensured but there are mechanism able to be configured to thread-safe. For example when a file method is invoked while the volume is used by another task, the task calling the file function will be suspended until that task leaves file function. A timer is available in order to abort the function in case of wait time exceeded. This feature must be compliant with the underlying RTOS. The file functions fmount() and fmkfs() are an exception, indeed they are not reentrant to the same volume. For this reason when using these functions, all other threads must close the corresponding file on the volume and avoid accessing it.

- **Long File Name:**
  The FatFs module has started to support long file name (LFN) at revision 0.07. The two different file names, SFN and LFN, of a file is transparent in the file functions except for freaddir() function [15].

- **Fatfs API:**
  The FatFs module is composed of an APIs layer that implements file system APIs. Those APIs are an abstraction of the disk I/O interface, used to communicate with the appropriate physical drive. These APIs are split into four groups:
  - Group of APIs that operates with logical volume or partition.
  - Group of APIs that operates with directory.
  - Group of APIs that operates with both file and folder
  - Group of APIs that operates with file.
- **Fatfs Low Level API:**
  Since the FatFS module is completely separate from the disk I/O and RTC module, it requires some low level functions to operate the physical drive: read/write and get the current time. Because the low level disk I/O functions and RTC module are not a part of the FatFs module, they must be provided by the user. An additional interface layer residing in diskio.c has been added to add/remove dynamically physical media to the FatFs module, providing low level disk I/O functions.

2.3.4.2 FATFS Architecture

So in summary FatFs module is a middleware which provides many APIs to access the FAT volumes, such as fopen(), fclose(), fread(), fwrite() and so on. It is fundamental its portability as long as the compiler is compliant with ANSI C. Furthermore a low level disk I/O module is used to access the physical drive and an RTC module is used to get the current time. It is doverous to underline that low level disk I/O and RTC module are completely separate from the FatFs module. In order to use them the user must port this features to its platforms.

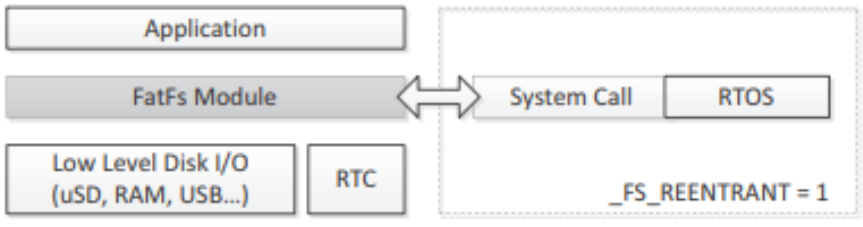

Figure 2.17: FATFS Architecture

3
Methods

This chapter describes the strategies and methodologies adopted in order to integrate and extend an HTTP web server on a STM32F746G-DISCO with the software components ChibiOS, Lwip and FatFS.

3.1 Sofware Components Integration in ChibiOS

In order to integrate and extend the functionalities of a basic http web server implemented in the LWIP stack, the first task to achieve was the correct integration and cooperation of the SW components LWIP and FATFS with the RTOS ChibiOS.

3.1.1 Used Materials

Here following are listed the SW components, the target board, programs, tools and environments that have been used for the achievement of expected results.

- **ChibiOS:**
  ChibiOS/RT is a compact and fast real-time operating system supporting multiple architectures and released under the GPL3 license. It is developed by Giovanni Di Sirio.
  Designed for embedded applications on 8, 16 and 32 bit microcontrollers, achieves its main project goals like size and execution efficiency. The kernel size can range from a minimum of 1.2 Kib up to a maximum of 5.5 KiB with all the subsystems activated on a STM32 Cortex-M3 processor.

- **ChibiStudio:**
  ChibiStudio is an open-source Eclipse development environment encapsulating all the necessary material to work with ChibiOS like Eclipse Luna 4.4.1, GCC ARM toolchain and OpenOCD 0.10.0. The adopted version of ChibiStudio is ChibiStudioPreview19.7z.

- **LwIP:**
  LwIP (lightweight IP) is a widely used open source TCP/IP stack designed for embedded systems. The focus of the LwIP TCP/IP implementation is to reduce resource usage while still having a full-scale TCP stack. This makes LwIP suitable for use in embedded systems with tens of kilobytes of free RAM

and room for around 40 kilobytes of code ROM.

- **FatFs:**
  FatFs is a generic FAT/exFAT filesystem module for small embedded systems. The FatFS module is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer. Therefore it is independent of the platform. It can be incorporated into small microcontrollers with limited resource.

- **TeraTerm:**
  Tera Term is an open-source, free, software implemented, terminal emulator (communications) program. It emulates different types of computer terminals, from DEC VT100 to DEC VT382. It supports telnet, SSH 1 and 2 and serial port connections. It also has a built-in macro scripting language (supporting Oniguruma regular expressions) and a few other useful plugins.

- **Wireshark:**
  Wireshark is a free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Originally named Ethereal, the project was renamed Wireshark in May 2006 due to trademark issues.
  Wireshark is cross-platform, using the Qt widget toolkit in current releases to implement its user interface, and using pcap to capture packets; it runs on Linux, macOS, BSD, Solaris, some other Unix-like operating systems, and Microsoft Windows.

- **STM32F746G-DISCO:**
  The 32F746GDISCOVERY Discovery board is a complete demonstration and development platform for the STMicroelectronics ARM® Cortex®-M7 core-based STM32F746NGH6 microcontroller. This microcontroller features four I2Cs, six SPIs with three multiplexed simplex I2S, SDMMC, four USARTs, four UARTs, two CANs, three 12-bit ADCs, two 12-bit DACs, two SAIs, 8- to 14-bit digital camera module interface, internal 320+16+4-Kbyte SRAM and 1-Mbyte Flash memory, USB OTG HS and FS, Ethernet MAC, FMC interface,Quad-SPI interface, SWD debugging support. This Discovery board offers everythingrequired for users to get started quickly and develop applications easily.

  - **Core:**
    The STM32F7 Series devices are built on a high-performance ARM® Cortex®-M7 32-bit RISC core operating up to 216 MHz frequency. The Cortex®-M7 core features a high performance floating point unit (FPU). The core can feature a single precision floating point unit or a double precision floating point unit (depending on the STM32F7 Series device), which support all ARM® single-precision and double precision data-processing instructions and data types.
    The Cortex®-M7 features a 6/7-stage superscalar pipeline with a branch

prediction and dual issue instructions.
The devices embed the Cortex®-M7 that features a level1 cache (L1-cache), which is split into two separated caches: the data cache (D-cache) and the instruction cache (I-cache) allowing to have a Harvard architecture bringing the best performance. These caches allow to reach a performance of 0-wait state even at high frequencies. The Cortex®-M7 has five interfaces: AXIM, ITCM, DTCM, AHBS and AHBP

- **Memories:**
  - 1 Mbytes of Flash memory (in BGA216 package).
  - 340 Kbytes of RAM (in BGA216 package).
  - 128-Mbit Quad-SPI Flash memory (external memory).
  - 128-Mbit SDRAM (64 Mbits accessible) (external memory).
  - Supports for 2-Gbyte (or more) microSD card.
- **Ethernet:**
  The 32F746GDISCOVERY Discovery board supports 10/100-Mbit Ethernet communication by a PHY LAN8742A-CZ-TR from MICROCHIP and RJ45 jack. Ethernet PHY is connected to STM32F746NGH6 via RMII interface. The 25-MHz clock for the PHY is generated by the oscillator X2.

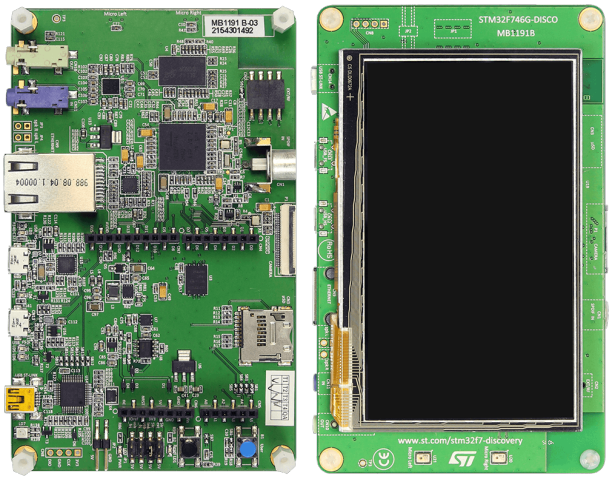

Figure 3.1: STM32F746G-DISCO board

3.1.2 Environment Set-Up

In order to compile and execute properly i had to set up the eclipse environment and the HW connection between components. In particular i have connected my

personal computer with:

- the Mini USB type B ST-LINK/V2-1 port in order to feed power and for debug purpose.
- the Ethernet RJ45 CN9 connector.

Furthermore, a SanDisk microSD card of 2GB has been inserted in the microSD connector CN3.

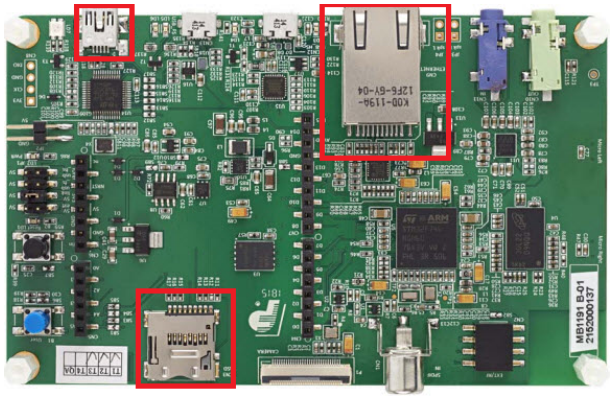

Figure 3.2: STM32F746G-DISCO connectors used

I had to change the Makefile configuration in order to include with the correct path the libraries and header files that i needed. Moreover i have modified the following compiler options for GCC/clang for debug and optimization purposes:

- -O2 this command overwrites -O0 to have performance optimized builds.
- -std=c99: in order to ensure C99 semantics which improve C type.
- -Wall: used to catch potential faults promptly.

The IP4 address of the web server is defined in a macro in lwipthread.h and is set to 192.168.1.10, the gateway ip to 192.169.1 and the netmask to 255.255.255.0. For this reason in order to establish a connection i needed to assign statically the IP4 address of the ethernet connection of my laptop with an address belonging to the same subnet like in 3.3 .

3.1.3 Integration of the basic LwIP Web Server Demo

In order to understand how to integrate this stack, my first approach was to use the ChibiOS DEMO "RT-STM32F746G-DISCO LWIP" where the basic version of an http web server have been provided by the LwIP stack itself.

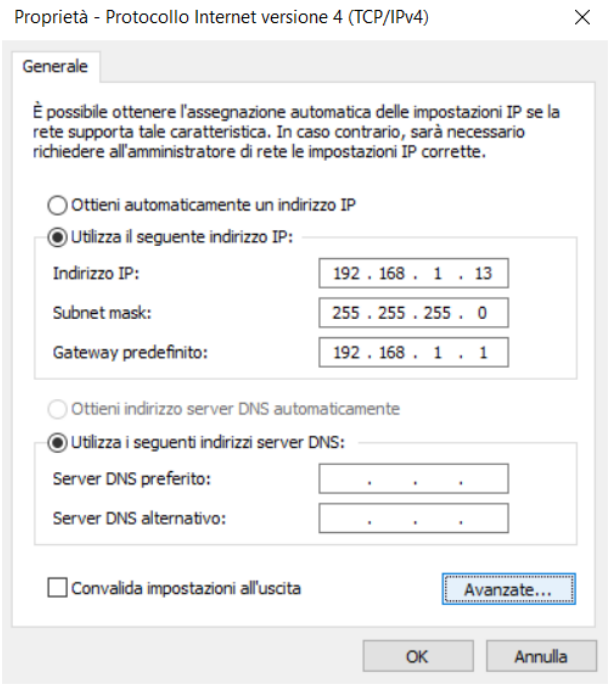

Figure 3.3: Laptop IP assignment

3.1.3.1 LwIP Software Architecture interfacing

The LwIP stack has been included in the Demo project inside the Eclipse environment and the native makefile of this stack has been written inside the inclusions of the main makefile. The application code (also known as main()) must call the function lwipInit() for the LwIP subsystem initialization using the default configuration. Furthermore also the method httpd_init() must be written in order to initialize the web server, in particular to set up a listening PCB and bind it to the defined port. Before moving on it is essential to describe the general software architecture of this stack. When the lwipInit() function is called its main purpose is to statically activate the lwip_thread() or also known as "Network Interface Thread".

- **Network Interface Thread:**
  First of all it activates the main thread of this stack tcpip_thread(), adds and set-up the network interface and in the end waits for a packet reception using the Ethernet receive interrupt service routine. Once a packet has been received it delivers them from driver buffers to the TCP/IP stack using the RTOS mailbox mechanism.

- **TCP/IP Thread:**
  This is the main thread of the LwIP stack, it has exclusive access to LwIP core functions. Other threads can communicate with this thread through the mechanism of message boxes. It also starts all the timers to make sure they are running in the right thread context. The user application communicates with the LwIP stack thanks to sequential API calls that in turn use the RTOS mailbox mechanism for inter-process communication. These sequential APIs

are blocking, indeed the application will be suspended until a response is received from the stack

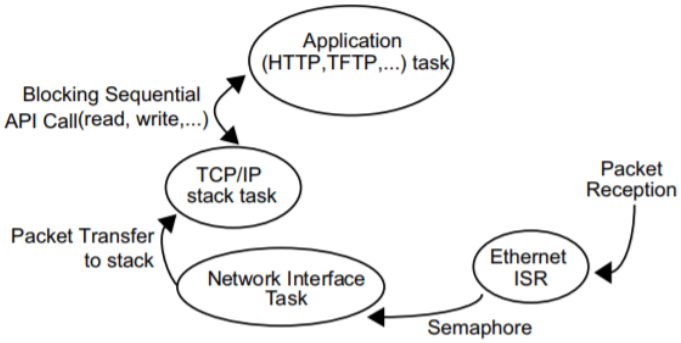

Figure 3.4: LwIP software architecture

The LwIP stack is composed of the following levels:

- **Network Interface Level:**
  This stack level in composed of all the device drivers used for physical network hardware, the latter are represented by a network interface structure. All of the netifs are saved on a global linked list thanks to the next pointer field which is linked to the next structure. In figure 3.5 there's a simpler version of the struct that i have used in order to represent a network interface.

```
struct netif {
  struct netif *next;
  char name[2];
  int num;
  struct ip_addr ip_addr;
  struct ip_addr netmask;
  struct ip_addr gw;
  void (* input)(struct pbuf *p, struct netif *inp);
  int (* output)(struct netif *netif, struct pbuf *p,
                 struct ip_addr *ipaddr);
  void *state;
};
```

Figure 3.5: Netif light structure

The three IP addresses ip address, netmask and gateway are used by the IP layer when sending and receiving packets, it is not possible to configure a network interface with more than one IP address. Rather, one network interface would have to be created for each IP address.
The input pointer points to the function that device driver should call when a packet has been received, in my case it is called low_level_input(). On the other hand a network interface is connected to a device driver through the output pointer. This pointer points to a function in the device driver that transmits a packet on the physical network and it is called low_level_output(). For these reasons i had to slightly modify these two function with appropriate device driver calls provided by the Hardware Abstraction Layer of ChibiOS

included in the header file hal_mac.h. In figure 3.5 is showed the implementation of the low_level_output(), basically, calling the specific mac device driver, waits for the availability of a transmission descriptor, writes data on it and once finished releases it and starts the transmission of the enqueued data as a single frame. Finally, the state pointer points to device driver specific state for the network interface and is set by the device driver.

```
static err_t low_level_output(struct netif *netif, struct pbuf *p) {
  struct pbuf *q;
  MACTransmitDescriptor td;

  (void)netif;
  if (macWaitTransmitDescriptor(&ETHD1, &td, TIME_MS2I(LWIP_SEND_TIMEOUT)) != MSG_OK)
    return ERR_TIMEOUT;

#if ETH_PAD_SIZE
  pbuf_header(p, -ETH_PAD_SIZE);        /* drop the padding word */
#endif

  /* Iterates through the pbuf chain. */
  for(q = p; q != NULL; q = q->next)
    macWriteTransmitDescriptor(&td, (uint8_t *)q->payload, (size_t)q->len);
  macReleaseTransmitDescriptor(&td);

  MIB2_STATS_NETIF_ADD(netif, ifoutoctets, p->tot_len);
  if (((u8_t*)p->payload)[0] & 1) {
    /* broadcast or multicast packet*/
    MIB2_STATS_NETIF_INC(netif, ifoutnucastpkts);
  }
  else {
    /* unicast packet */
    MIB2_STATS_NETIF_INC(netif, ifoutucastpkts);
  }
  /* increase ifoutdiscards or ifouterrors on error */

#if ETH_PAD_SIZE
  pbuf_header(p, ETH_PAD_SIZE);         /* reclaim the padding word */
#endif

  LINK_STATS_INC(link.xmit);

  return ERR_OK;
}
```

Figure 3.6: Low Level Output Implementation

- **Internet Protocol Level:** The internet protocol layer hasn't been modified. Briefly, i had to use two main functions ip4_input() and ip4_output(). The former is called by the network interface device driver when an IP packet is received. The function does the basic checks of the IP header such as packet size being at least larger than the header size etc, if the packet was not destined for us, the packet is forwarded. Obviously the IP checksum is always checked and finally, the packet is sent to the upper layer protocol input function. The latter function sends an IP packet on a network interface. This function constructs the IP header and calculates the IP header checksum.

- **Transport Level:** Also this layer hasn't been modified. It is composed of three main functions:
  - **tcp_input():**
    The initial input processing of TCP. It verifies the TCP header, demultiplexes the segment between the PCBs, in particular checks if the frame is destined to a connection in active, time wait or listening state. After other operations passes this frame to tcp_process(), which implements the TCP finite state machine.

  - **tcp_process():**
    This function implements the TCP state machine and takes as argument the TCP protocol control block of the incoming frame. Depending on the TCP state of the pcb passed, a different operation will be executed. In general it follow the classic three-way handshake for the establishment of a connection and the four-way handshake in order to close a connection.

  - **tcp_write():** This method is very important in order to understand some considerations that i will list in the appropriate chapter. Basically, its main purpose is to write data for sending (but does not send it immediately). Indeed it could wait in the expectation of more data being sent soon (as it can send them more efficiently by combining them together). It is possible to prompt the system to send data immediately by calling tcp_output() inside this function.

- **Application Level:**
  This is the higher layer of the stack and here reside all the user applications that needs the services provided by the lower layers. In my case the http web server, i will analyze it in the next subsection.

3.1.3.2 Web Server Native Architecture

This sample web server version is very basic and one of my tasks was to integrate it in ChibiOS, like said in the previous subsection, and extended different functionalities. Before going in the detail it is necessary to describe the general organization and its main features that had already been implemented. First of all, this web server must be initialized inside the main function by calling the appropriate method that set up a listening PCB and binds it to the defined port. Once a new incoming connection has been accepted, it allocates memory for the structure that holds the state of the connection set up the various callback functions like http_sent() and http_recv(). The standard procedure for an incoming fragment is composed of the following function calls:

- **http_recv():**
  When a new segment is received by the transport layer, the recv() callback function is activated. Essentially it informs the TCP layer that we have taken the data.
- **http_parse_request():**
  Basically it parses the method of the incoming fragment with a possible HTTP command (most used is the GET command). If there's no match the connection will be closed, otherwise the file will be searched inside the ROM file system.

- **http_send():**
  This function tries to send as much data as possible on the current PCB. Usually starting from the header and controlling that we haven't still reached

the end of the file. In this case it adds the FIN flag right into the last data segment in order to alert the receiver that a 4 way handshake is necessary in order to close the communication.

3.1.3.3 Web Server Native File Organization

The native file organization of this web server is based on the usage of a ROM file system. The latter could be generated by a c file called makefsdata.c that automatically generate the files data and links them in a tree disposition. Also a perl version of this algorithm have been developed.

```
struct fsdata_file {
  const struct fsdata_file *next;
  const unsigned char *name;
  const unsigned char *data;
  int len;
  u8_t flags;
#if HTTPD_PRECALCULATED_CHECKSUM
  u16_t chksum_count;
  const struct fsdata_chksum *chksum;
#endif /* HTTPD_PRECALCULATED_CHECKSUM */
};
```

Figure 3.7: File structure

In figure 3.7 is represented the implementation of a generic file of this web server. The first field of the struct is used to link the different files, in case of a nested inclusion, for example an http file could be composed of a text box and of an image that is another file of the web server itself. Trivially the second and third fields represent the file name and the heart of a file. The data file is represented as
As we can see this file organization has different drawbacks:

- The integration and generation of new files of the web server is not so efficient and intuitive. Indeed we need to generate a new file system ROM and it is not possible to add files during the execution time.

- Another clear problem is that the dimension of the web server is strictly subject to the memory dimension.

  - ROM memory: all the file system and data files being designed in rom fashion will be loaded entirely on the Flash memory during the loading phase od the code. It is trivial to remember that the memory usage in an embedded environment is a key concept.

  - RAM memory: there isn't a mechanism designed to reduce the RAM memory usage, also in this case the RAM management is essential due to its limited size.

```
static const unsigned char data__img_sics_gif[] = {
/* /img/sics.gif (14 chars) */
0x2f,0x69,0x6d,0x67,0x2f,0x73,0x69,0x63,0x73,0x2e,0x67,0x69,0x66,0x00,0x00,0x00,

/* HTTP header */
/* "HTTP/1.0 200 OK
" (17 bytes) */
0x48,0x54,0x54,0x50,0x2f,0x31,0x2e,0x30,0x20,0x32,0x30,0x30,0x20,0x4f,0x4b,0x0d,
0x0a,
/* "Server: lwIP/1.3.1 (http://savannah.nongnu.org/projects/lwip)
" (63 bytes) */
0x53,0x65,0x72,0x76,0x65,0x72,0x3a,0x20,0x6c,0x77,0x49,0x50,0x2f,0x31,0x2e,0x33,
0x2e,0x31,0x20,0x28,0x68,0x74,0x74,0x70,0x3a,0x2f,0x2f,0x73,0x61,0x76,0x61,0x6e,
0x6e,0x61,0x68,0x2e,0x6e,0x6f,0x6e,0x67,0x6e,0x75,0x2e,0x6f,0x72,0x67,0x2f,0x70,
0x72,0x6f,0x6a,0x65,0x63,0x74,0x73,0x2f,0x6c,0x77,0x69,0x70,0x29,0x0d,0x0a,
/* "Content-type: image/gif

" (27 bytes) */
0x43,0x6f,0x6e,0x74,0x65,0x6e,0x74,0x2d,0x74,0x79,0x70,0x65,0x3a,0x20,0x69,0x6d,
0x61,0x67,0x65,0x2f,0x67,0x69,0x66,0x0d,0x0a,0x0d,0x0a,
/* raw file data (724 bytes) */
0x47,0x49,0x46,0x38,0x39,0x61,0x46,0x00,0x22,0x00,0xa5,0x00,0x00,0xd9,0x2b,0x39,
0x6a,0x6a,0x6a,0xbf,0xbf,0xbf,0x93,0x93,0x93,0x0f,0x0f,0x0f,0xb0,0xb0,0xb0,0xa6,
0xa6,0xa6,0x80,0x80,0x80,0x76,0x76,0x76,0x1e,0x1e,0x1e,0x9d,0x9d,0x9d,0x2e,0x2e,
0x2e,0x49,0x49,0x49,0x54,0x54,0x54,0x8a,0x8a,0x8a,0x60,0x60,0x60,0xc6,0xa6,0x99,
0xbd,0xb5,0xb2,0xc2,0xab,0xa1,0xd9,0x41,0x40,0xd5,0x67,0x55,0xc0,0xb0,0xaa,0xd5,

                    ..............................

0x82,0x0c,0x36,0xe8,0xe0,0x83,0x10,0x46,0x28,0xe1,0x84,0x14,0x56,0x68,0xa1,0x10,
0x41,0x00,0x00,0x3b,};
```

Figure 3.8: Data File Representation

These limitations have been solved thank to the transfer from a ROM based file system to a microSD based. In order to achieve this task i had to integrate the FatFS file system in my project. The details of this procedure are described in the next subsection.

3.1.4 Integration of the FatFS file system

Taking in account all the theoretical overview described in section 2.3, it is deserved to give a brief explanation of the software organization of this file system. In the following figure is represented a dependency diagram that is a typical but not specific configuration of the embedded system with FatFs module.

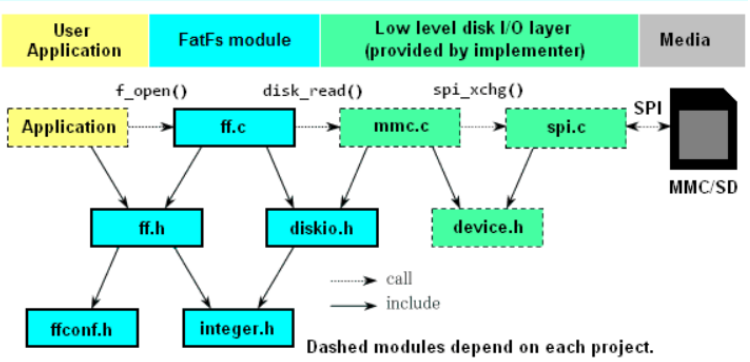

Figure 3.9: FatFS Organization

As we can see we have four fixed fields:

- **User Application:**
  This is is the user specific level where the custom application will interact with the FatFs module through the APIs provided from the latter. This layer is completely customizable.

- **FatFs module :**
  It is the core of this file system, indeed it is composed of all the application programming interfaces provided for the upper layer. These APIs are categorized in file access, directory access, file and directory management, volume management and system configuration. I had to use mainly f_mount(), f_open(), f_read(), f_write(). Going in a deeper level, but always inside the FatFs module, exist different "glue functions" that are requested by the higher level APIs in order to communicate with the HW media properly.

- **Low Level Disk I7O Layer:**
  This is the space where all the low level drivers needed to communicate with the underlying hardware. In my case all the low level drivers were provided by the HAL layer of ChibiOS.

- **Hardware Layer:**
  It is composed of the all media that could communicate with our FatFs module. In my case the microSD is the media that is going to be used.

There are two possible general scenarios:

- **Single Drive System:**
  If a working disk module with FatFS disk interface is provided, nothing else will be needed. So the customer of this file system doesn't have the effort of code implementation.

- **Multiple Drive System:**
  To attach existing disk drivers with different interface, glue functions are needed to translate the interfaces between FatFS and the drivers. This is the scenario of my project.

3.1.4.1 Glue functions implementation

In order to make the FatFS APIs available for the communication with the hardware media, in my scenario, i have to implement the body of all the glue functions residing in the lower level of the FatFS module. Those functions are mainly disk_initialize(), disk_status(), disk_read(), disk_write(), disk_ioctl().
The FatFS module provides a template for these functions implementation. Basically for each function there's a switch case statement that according to the physical drive number provides a different behavior.
In figure 3.11 is showed the implementation of one of the glue functions, in particular the disk_read().

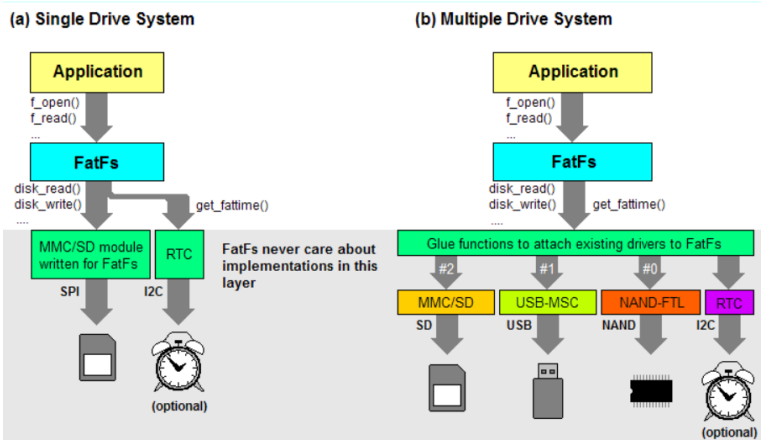

Figure 3.10: FatFS Scenarios

```
DRESULT disk_read (
    BYTE pdrv,        /* Physical drive number (0..) */
    BYTE *buff,       /* Data buffer to store read data */
    DWORD sector,     /* Sector address (LBA) */
    UINT count        /* Number of sectors to read (1..255) */
)
{
  switch (pdrv) {
#if HAL_USE_MMC_SPI
  case MMC:
    if (blkGetDriverState(&FATFS_HAL_DEVICE) != BLK_READY)
      return RES_NOTRDY;
    if (mmcStartSequentialRead(&FATFS_HAL_DEVICE, sector))
      return RES_ERROR;
    while (count > 0) {
      if (mmcSequentialRead(&FATFS_HAL_DEVICE, buff))
        return RES_ERROR;
      buff += MMCSD_BLOCK_SIZE;
      count--;
    }
    if (mmcStopSequentialRead(&FATFS_HAL_DEVICE))
        return RES_ERROR;
    return RES_OK;
#else
  case SDC:
    if (blkGetDriverState(&FATFS_HAL_DEVICE) != BLK_READY)
      return RES_NOTRDY;
    if (sdcRead(&FATFS_HAL_DEVICE, sector, buff, count))
      return RES_ERROR;
    return RES_OK;
#endif
  }
  return RES_PARERR;
}
```

Figure 3.11: Disk Read Implementation

Basically, it performs the call to different device drivers like the mmc one. These device drivers were provided by the Harware Abstraction Layer of the ChibiOS operating system. It is necessary to highlight that the mmc device driver implemented in hal_mmc_spi.c makes use of the spi hal device driver (provided by ChibiOS HAL). A function of the mmc device driver is represented in the following figure.

```
bool mmcSequentialRead(MMCDriver *mmcp, uint8_t *buffer) {
  unsigned i;

  osalDbgCheck((mmcp != NULL) && (buffer != NULL));

  if (mmcp->state != BLK_READING) {
    return HAL_FAILED;
  }

  for (i = 0; i < MMC_WAIT_DATA; i++) {
    spiReceive(mmcp->config->spip, 1, buffer);
    if (buffer[0] == 0xFEU) {
      spiReceive(mmcp->config->spip, MMCSD_BLOCK_SIZE, buffer);
      /* CRC ignored. */
      spiIgnore(mmcp->config->spip, 2);
      return HAL_SUCCESS;
    }
  }
  /* Timeout.*/
  spiUnselect(mmcp->config->spip);
  spiStop(mmcp->config->spip);
  mmcp->state = BLK_READY;
  return HAL_FAILED;
}
```

Figure 3.12: MMC read implementation

3.2 Achievement of a MicroSD Based Web Server

As mentioned in the web server native file organization section the original web server was based on the usage of a ROM file system. This file organization, if on one side doesn't require the usage of additional hardware components, on the other side involves a set of problems to be handled. In particular it is mandatory to generate a new file system ROM every time we would change the current web server. Furthermore, the ROM memory space is critical due to implicit nature of a ROM file system and it is not implemented a valid method to handle data files with a RAM memory saving strategy. For these reason is born the idea of using microSD based web server. In order to achieve this task, i had to integrate correctly the LwIP stack and the FatFS file system as explained in the two previous sections.

3.2.1 Configuration File Set-Up

In this phase i had to change different options of the configuration files opt.h and httpd_opts.h before going on with the implementation of the custom functionalities. In particular in the file opt.h :

- MEM_SIZE:
  This parameter is used to set the size of the heap memory. I have choose to use 10k memory size in order to handle properly the file fragmentation and the memory effort caused by the invocation of dynamic threads once the server will be upgraded from static to dynamic.

- TCP_MSS:
  The maximum segment size (MSS) is a parameter of the options field of the TCP header that specifies the largest amount of data, specified in bytes, that

```
#if !defined MEM_SIZE || defined __DOXYGEN__
#define MEM_SIZE                        10000
#endif
```

Figure 3.13: Heap memory tuning

a computer or communications device can receive in a single TCP segment. It does not count the TCP header or the IP header (unlike, for example, the MTU for IP datagrams). I have decided to set it to 536 bytes that is conservative value, increasing this value could speed up the transfer rate at the cost of memory occupation.

```
#if !defined TCP_MSS || defined __DOXYGEN__
#define TCP_MSS                         536
#endif
```

Figure 3.14: MSS tuning

- TCP_SND_BUFF: This field is used to set the TCP sender buffer space in term of bytes. To achieve good transfer performance, this should be at least 2 * TCP_MSS.

```
#if !defined TCP_SND_BUF || defined __DOXYGEN__
#define TCP_SND_BUF                     (2 * TCP_MSS)
#endif
```

Figure 3.15: MSS Send buffer tuning

3.2.2 HTTP Level Implementation

Following are listed some implementation details of the web server, in particular the handling of a packet in reception and sending phase.
HTTP is a protocol based on the client/server model. In my case the web browser running on the external laptop is the client and the application running on the STM32F746G is the server. The client submits an HTTP request message to the server. The latter, which provides resources such as HTML files and other content, performs other functions on behalf of the client, returns a response message to the client. The response contains completion status information about the request and may also contain requested content in its message body. A typical scenario of reception is the one represented in figure 3.26. The first step corresponds to a request of the client, let's consider, for example, the GET message request. My embedded web server will mainly run the callback function associated to the reception function. Basically, when data has been received in the correct state, a parser will be called in order to see if the message received corresponds to one of the available http requests. In negative case the connection will be dropped. The step number four in figure 3.26 corresponds to the research of the file specified by uri inside the microSD. In particular, the fs_open() primitive will be called.

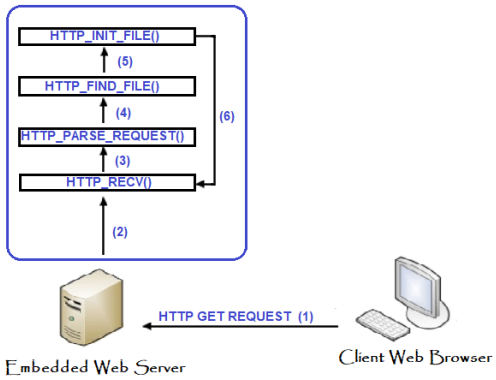

Figure 3.16: HTTP packet reception

```
int fs_open_custom(struct fs_file *file, const char *name)
{
  FRESULT fres = FR_OK;
  FFOBJID objFF;
  int fSize = 0;
  const char* prova = name;

  if(isFF_Enabled){

    FIL *ptrFileObj = (FIL *)mem_malloc(sizeof(FIL));
    if(ptrFileObj == NULL){
      mem_free(ptrFileObj);
      return(0);
    }
    fres = f_open(ptrFileObj,prova,FA_READ);
    if(fres == FR_OK){
      objFF = ptrFileObj->obj;
      fSize = objFF.objsize;
      file->data = (char*)NULL;
      file->len = fSize;
      file->index = (int)NULL;
      file->pextension = ptrFileObj;
      file->flags = 0;
      return 1;
    }
    else{
      mem_free(ptrFileObj);
      return 0;
    }
  }
  else
    return 0;
}
```

Figure 3.17: Fs_Open() Implementation

This function will interact with the microSD with an SPI communication as explained in section 3.1.4. This function will initialize the fs_file structure with some mandatory parameters like the lenght of the data and the reference of this object assigned to the file->pextension field. Once match have been found, the http_state structure, representing the session of the actual http connection, will be initialized

accordingly. Furthermore, HTTP headers to send will be determined dynamically depending on the file extension of the requested URI. Different http headers are stored ready to be used in case of the file extension match. With this operation terminates the reception phase and connection initialization.

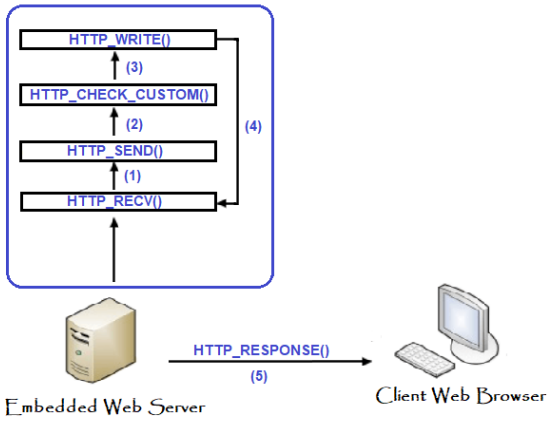

Figure 3.18: HTTP packet response

In figure 3.18 is represented the second phase of this transmission, the packet response. In the first step is invoked the send function that is responsible mainly of the header transmission, that was computed dynamically in the previous phase. Once the transmission of the header have been accomplished correctly the http_check_custom() is called. Obviously the transmission of the header, being based on the tcp protocol, could be fragmented. For this reason the http connection session is preserved for the residual part to be sent. The http_check_custom() is maybe the most important function that i have implemented. Indeed it has different purposes. First of all it checks if we have a valid file handle and if we have reached the end of the file, if one of these conditions is not true the connection will be dropped. In the actual http session, if we need to send the first fragment a RAM buffer with a tunable size parameter will be dynamically allocated.

3.2.3 Transmission Handling Issues and Details

3.2.3.1 Read Buffer Details

In order to limit the RAM usage, files contained inside the microSD are transferred in high level fragments (higher level of the tcp fragmentation). Basically, fragments of a default size are transferred inside the same http session. Once the transfer is completed, another chunk will be read and stored inside the same memory space allocated previously. This process will be iterated until the end of file have been reached.

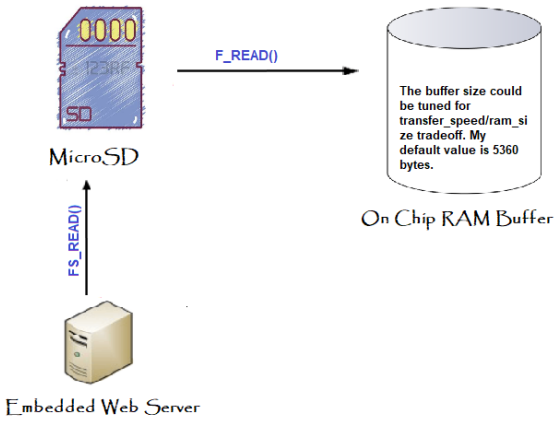

Figure 3.19: HTTP read buffer

This mechanism guarantees that the RAM usage is limited, without a trivial waste of memory. Obviously this is definitely more advantageous respect to the case of reading the file like an entire block. Indeed the RAM memory usage could be a tremendous weakness in a real time environment like the actual one. A critical parameter is the length of the RAM buffer. In case of a small parameter the transfer rate of a generic file from the server to the client could be very slow. From the tcp point of view the transfer speed is limited by the send buffer size and, if activated, by the capability of waiting in the expectation of more data being sent soon (fragment assembling). The size of the RAM buffer influences directly the total transfer rate, indeed if we have a small buffer size the total number of read operations increases. Each read operation on the microSD involves an SPI communication, for this reason having a too much small buffer size will involve in a significant memory save at the cost of a slow total transfer rate. On the other hand a big buffer will reduce the number of read operations at the cost of the memory usage.

3.2.3.2 Fragment Time Overhead

Another problem that i have found was a feature provided by the tcp level that in particular cases doesn't match exactly the requirements of a real time environment like the one actually used. In particular, the function tcp_write() in its pre-defined

state waits in the expectation of more data being sent soon and tries to combine them together when possible. In order to understand the solution adopted it is necessary to give some detail of the tcp buffer handling. The TCP header is composed of different one-bit boolean fields known as flags used to determine the flow of fragments across a TCP connection. Here are listed some of the main control flags.

- SYN:
  Start of a connection.
- ACK:
  Acknowledges received data.
- FIN: Closes a connection.
- RST: Aborts a connection in response to an error.

The figure below shows how data is store by the sender before sending, and by the receiver upon reception.

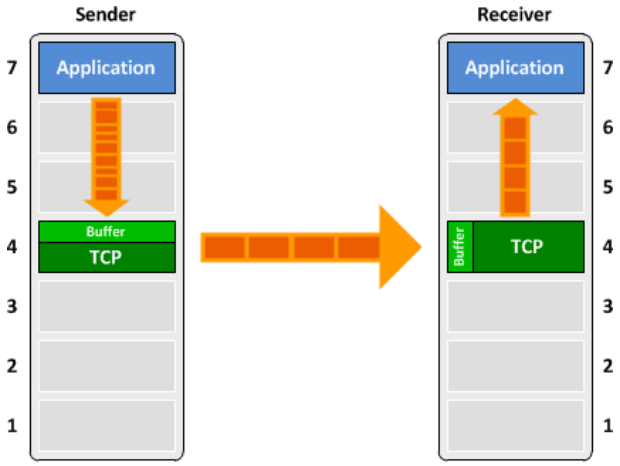

Figure 3.20: TCP buffers

These buffers allow for more efficient transfer of data when sending more than one maximum segment size worth of data. However, large buffers do more harm than good when dealing with real-time applications which require that data be transmitted as quickly as possible and small RAM size. For this reason the PSH flag comes in. The socket that TCP makes available at the session level can be written to by the application with the option of "pushing" data out immediately, rather than waiting for additional data to enter the buffer. In order to activate this behavior the PSH flag in the outgoing TCP packet must be set to 1. The other side of the connection knows to immediately forward the segment up to the application. This flag has been set in the send function represented in figure 3.21.
In particular the control instructions that will set the header of the tcp packet is the following implemented in the tcp_write() function.

```
static u8_t
http_send_data_nonssi(struct tcp_pcb *pcb, struct http_state *hs)
{
  err_t err;
  u16_t len;
  u8_t data_to_send = 0;


  /* We are not processing an SHTML file so no tag checking is necessary.
   * Just send the data as we received it from the file. */
  //len = (u16_t)LWIP_MIN(hs->left, 0xffff);
  len = (u16_t)LWIP_MIN(hs->left, 0xffff);

  /* native flag field was HTTP_IS_DATA_VOLATILE(hs)  */
  err = http_write(pcb, hs->file, &len, 0);
  if (err == ERR_OK) {
    data_to_send = 1;
    hs->file += len;
    hs->left -= len;
  }

  return data_to_send;
}
```

Figure 3.21: PSH flag set

```
/* Set the PSH flag in the last segment that we enqueued. */
if (seg != NULL && seg->tcphdr != NULL && ((apiflags & TCP_WRITE_FLAG_MORE)==0)) {
  TCPH_SET_FLAG(seg->tcphdr, TCP_PSH);
}
```

Figure 3.22: PSH control instruction

3.3 Achievement Of A Dynamic Web Server

Before going on with the implementation details of the upgrade from static to dynamic web server it is useful to give some background detail.

3.3.1 Static vs Dynamic Web Server

By definition a web server is nominated **static** when provides just static web pages. A static web page is a web page with a well defined structure that is sent to an user exactly as saved in a storage device, in contrast to dynamic web pages which are generated dynamically in different ways. Consequently, a static web page is displayed in the same way from every users and contexts. These web pages are often HTML documents stored as files in the file system and made available by the web server over HTTP. In my case the web pages are stored inside in the microSD. An in important feature of static web pages is that they never or rarely need to be updated.In order to maintain large numbers of static pages as files requests the usage of automated tools. Any personalization or interactivity has to run client-side, which is restricting.

- **Advantages**:
  - Provide improved security over dynamic websites.

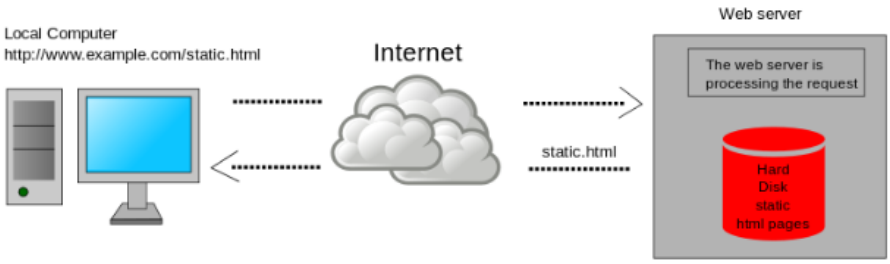

Figure 3.23: Static Web Server

  - Improved performance for end users compared to dynamic websites.
  - Fewer or no dependencies on systems such as databases or other application servers

- **Disadvantages**:
  - Dynamic functionality has to be added separately

A web server is called **dynamic** when provides dynamic and possibly static web pages. A dynamic web page can be classified in two different types: server side or client side. A server-side dynamic web page is a web page whose construction is controlled by an application server processing server-side scripts [16]. In server-side scripting, parameters determine how the assembly of every new web page proceeds, including the setting up of more client-side processing [16]. On the other hand a client-side dynamic web server processes the web page using HTML scripting running in the browser once loaded. For example JavaScript and other scripting languages determine the way the HTML in the received page is parsed into the Document Object Model.

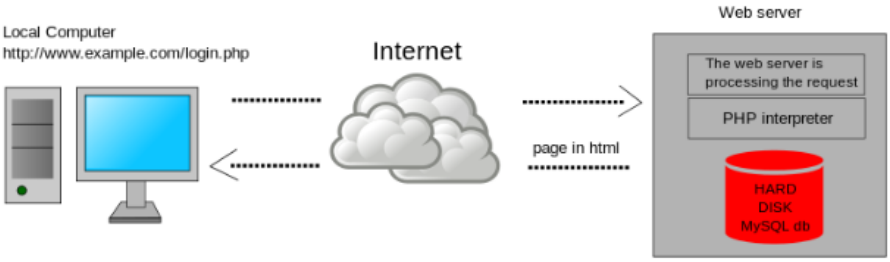

Figure 3.24: Dynamic Web Server

- **Advantages**:
  - The main advantage is intrinsic in the concept of dynamic web page. Indeed the client will see the server in a dynamic fashion.

- **Disadvantages**:
  - The main disadvantage is their cost. Indeed it involves a grater effort on the server. On each request for a dynamic web page a new task is created on the server side in order to provide the correct page. This behavior requires time, memory and slows down the system. So, it is clear that in general more powerful and expensive calculators are necessary. Other "cost" parameter are the waiting time of the client and the non trivial handling and creation of dynamic pages, that requires programming languages skills.

3.3.2 CGI Interface

Common Gateway Interface (CGI) is a standard protocol widely spread for interfacing external applications with a web servers. The programs running on a server will generate web pages dynamically. Such programs are known as CGI scripts or simply as CGIs. The specifics of how the script is executed by the server are determined by the server.

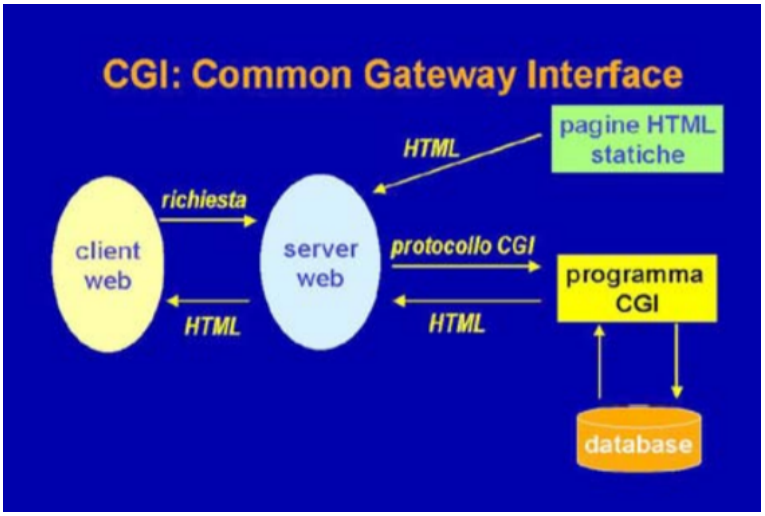

Figure 3.25: CGI Interface

A typical interaction client/server that requires a dynamic page is generally composed of the following steps:

- **Step 1**:
  The client, after having established a TCP connection with the server through the HTTP protocol, will send a GET request for a dynamic page. Optionally other input parameters will be specified, the most usual approach for parameter passing is the filling of the form module provided by the HTML language.

- **Step 2**:
  Once the request has been correctly received by the server, the latter through

the CGI interface will dynamically run the program associated to the specific request. All the parameters received in input will be passed to running thread.

- **Step 3**:
  In this step the behaviors could be different, for example it could interact with the database included on the server side. Once the active thread have terminated the execution, it will return the result of its activity to the web server as expected by the CGI interface. In this case the thread is like a gateway between the server and the database, in other words it hooks the db to the web server making it available for every client.

- **Step 4**: The server sends to the client the computed data from the CGI interface to HTTP protocol.

3.3.3 Dynamic Wed Server Implementation

3.3.3.1 Architectural Choices

As mentioned in the previous section our web server could be approximated to a 3 levels architecture. In particular the CGI interface based is the adopted one. This choices will lead to some considerations that will be examined in depth in the appropriate section. It is mandatory to describe now the CGI program, in particular the programming language chosen and the motivations. The CGI interface defines only the method with which the data are passed from server to application and vice versa, so it is just the definition of the I/O. The programmers have a wide freedom of choice on the greatest part of the application. The CGI thread could be written in every language that allows to be executed on the desired system. Examples could be C/C++, Perl, shell languages of UNIX and Fortran. Programming languages can be divided in the main categories:

- **Compiled Languages**:
  A compiled language also called imperative is a programming language whose implementations are typically compilers. Usually they are written in a development IDE that makes the creation more comfortable. This code will be checked and verified in order to find errors or warning and finally compiled. The compilation phase consists in the translation of each instruction to the correspondent machine code that will be executed by the processor. Examples of this kind of languages are C/C++, Fortran, Pascal and so on.

- **Interpreted Languages**:
  Interpreted languages follow different path, in particular they need the presence of an interpreter. The latter is a program that directly executes instructions written in scripting or programming languages without the step of compilation that translates the instructions in machine language. Usually the interpreter parses the source code and executes it directly but is is possible also that it will translate the source code in a more efficient intermediate represen-

tation ad executes this. Examples of interpreted languages are Perl, Python, Matlab, Ruby and so on.

It is important to highlight the comparison between compiled and interpreted languages in order to understand their benefits or drawback depending on the application. According to these consideration the right programming language for the CGI program have been adopted. An interpreted language involves in a lesser efficiency at run-time, indeed an interpreted program during execution requires more memory space and it is slower due to the overhead introduced by the interpreter itself. During the execution, the interpret must examine the instructions starting from the syntactic level, identify the actions to execute (eventually transforming the symbolic names of the variables in the correspondent memory addresses) and execute them. The instructions of the compiled code, already in machine language, are loaded and instantaneously executed by the processor. On the other hand, the interpretation of a program could be faster than the cycle of compilation/execution. This difference could be an advantage during the development phase, especially if it is led with fast prototyping techniques or during the debugging phase. Furthermore the greatest part of the interpreters allows the user to act on the program in execution suspending, inspecting or modifying the content of its variables and so on. For this reason it is more flexible and powerful in debug phase respect to compiled programs.
In summary:

- **Key 1**:
  The main complexity was the integration phase but from an implementative point of view the web server is not so heavy, for this reason the debug advantages introduced by an interpretative language like the Perl were not necessary.

- **Key 2**:
  From a performance point of view the interpreted language is less efficient during run-time and involves different overhead. Being in a real time environment it is preferable to avoid these situations (although in a non hard real time environment).

- **Key 3**: The STM32F7 discovery board will load in flash and partially in RAM the Chibios operating system and the TCP/IP stack included all the static and dynamic threads. For this reason it is preferable to avoid the adding of an interpreter in order to preserve the memory size at minimum.

For all the reason listed above i have chosen to write the **CGI interface in C** language. Although CGI programs are often written in interpreted languages like Perl in this application a compiled language like C is the best solution.

3.3.3.2 Implementation Details

Following are listed some implementation details of the web server, in particular the handling of a packet in reception and sending phase.

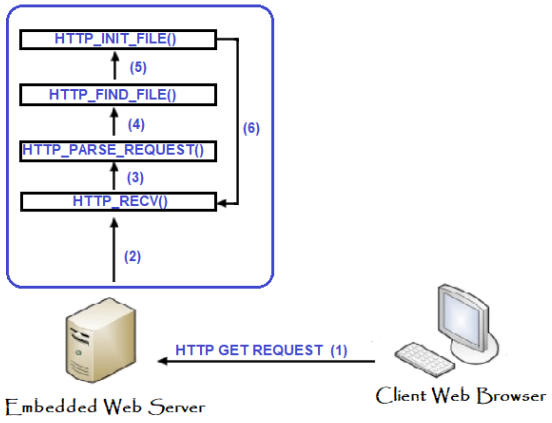

Figure 3.26: HTTP packet reception

The dynamic version of the web server described in the section "HTTP level implementation" follows almost the same architectural flow for an HTTP request and response in terms of methods and callback used. Like before the client will submit an HTTP request message to the server and the latter, which will provide both static and dynamic web pages such as HTML files and other content, returns a response message to the client. Let's consider the scenario of a message received from a client in order to explain the implementation details following a precise order. When a new connection have been established the function httpd_init() will be invoked where different parameters of the web server will be set. Furthermore a new mailbox will be allocated and there's the initialization of the available cgis addresses and handlers. The figure 3.27 describes the implemented architecture in order to provide correctly a dynamic page.

Like said before every instance of the web server will allocate its own mailbox. The mailbox mechanism allows the communication and data exchange between two different threads in a synchronous or asynchronous way. In our case the main actors that will exchange informations are LwIp thread and CGI thread. The idea behind this mechanism is the following. Basically, all begins with a request of the client, let's consider, for example, the GET message request. The embedded web server will run the callback function associated to the reception function called http_recv(). When data has been received in the correct state, a parser will be called in order to check if the message received suits one of the HTTP request considered. In negative case the connection will be dropped. Following, the step number four in figure 3.26 corresponds to the composition of the specified uri that will be used as path in order to find the correspondent file. In particular, if the URI doesn't match with one of the available CGI handlers then the control will be made with some of the files stored inside the microSD. Otherwise if a CGI handler have been found the URI parameters will be extracted by calling the function extract_uri_parameters() and stored in a persistent location at session level. Once a match has been found the

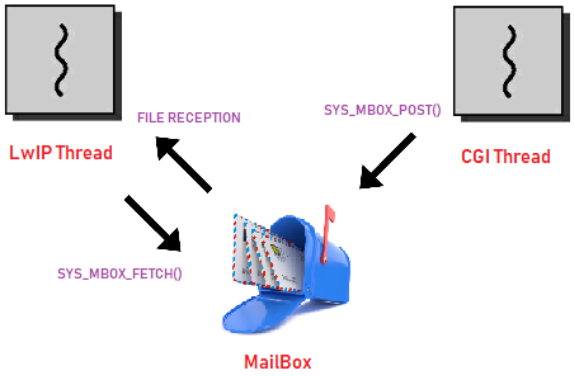

Figure 3.27: Processes interactions

correspondent handler will be invoked that basically will create a dynamic thread (cgi program) in charge of providing the correct dynamic page. The lwip thread will be suspended on a reference point until the cgi thread will reactivate it. The CGI that i have implemented will starts with some operations specific to the purpose of the current thread, so this part of code is not reusable for others cgis. But in general could exist a crossroad where it is necessary to check if some credentials are met or not. In affirmative case the flow goes on, otherwise the cgi thread will exit from execution and the normal lwip thread flow will be activated (as explained in http level implementation section). Let's remember that the main purpose of the CGI is the composition of a web serve dynamically. In order to achieve this behavior an universal algorithm, applicable and reusable for each and every cgi have been implemented.
Fist of all, the static part of a web page have been stored in fragments. This decision is advantageous for the final web page assembling, composed of the static part and the parameters coming from the clients request. In order to avoid an high ram memory usage, the mailbox queue have been initialized to 1 slot. This decision doesn't affect too much the transmission from a performance point of view, indeed as discussed in "Transmission handling issues and details" section the main element that could compromise the performance is the read buffer size. The messages that two threads are going to exchange are Memory Streams. They are a particular structure that represents a stream object composed of data and methods for reading, writing and so on. This algorithm can be listed in the following steps:

- **Step 1**:
  Check if there's enough space in the memory stream before writing, otherwise post the message.

- **Step 2a**:

```
while(i<num_fragments){
  //Check if there's enough space before writing, otherwise post the message
  if (memMess->size - memMess->eos < len_vett[i]){
    sys_mbox_post(&cgi_mbox, memMess);
    //Suspend until the message has been read by the other thread
    chSysLock();
    chThdSuspendS(&send_trp);
    chSysUnlock();
    //Now that the stream object has been read we have to re-init and clean
    msObjectInit(memMess,buffer,buf_size,0);
    memset(buffer, 0, sizeof(uint8_t)*buf_size);
  }
  else{
    //Write the fragment on the stream and increment for the next fragment
    memMess->vmt->write(memMess,(const uint8_t *)html_file[i],len_vett[i]);
    i++;
    if(i == num_fragments){
      //Last fragment POST
      sys_mbox_post(&cgi_mbox, memMess);
      chSysLock();
      chThdSuspendS(&send_trp);
      chSysUnlock();
    }
  }
```

Figure 3.28: CGI File Composition Algorithm

In case of enough space, writes the actual fragment on the stream and increment a variable for the next fragment evaluation. In case the last fragment have been processed the stream will be posted on the mailbox and the task will suspend on the reference variable until the LwIP thread reactivates it.

- **Step 2b**:
  If the stream is full the stream will be posted on the mailbox and the task will suspend until the message will be taken from the mailbox by the LwIP thread. Once the thread is reactivated it will re-init and clean the stream object. The next fragment will be processed restarting from the step 1.

The Lwip thread on the other hand will follow the standard flow described in figure 3.16 and figure 3.18. The only difference is that the function http_send() is replaced by http_send_dynamic_data(). The operations are almost the same but in this case the thread waits on the mailbox until a new message have been received. The read buffer and the considerations about the tcp level are always the same.

4

Results

In this chapter are listed all the results of my internship experience at STMicroelectronics. Starting from the realization of the different component and the realization of a dynamic web server. In the end also a performance and memory usage have been reported.

4.0.1 Software Components Integration

This was the first step of my thesis work immediately after a deep study of the software components organization. In order to achieve the integration and extension of functionalities of the basic http web server, was necessary to make possible the cooperation of the SW components LWIP and FATFS with the RTOS ChibiOS. The final result can be seen in figure 4.1

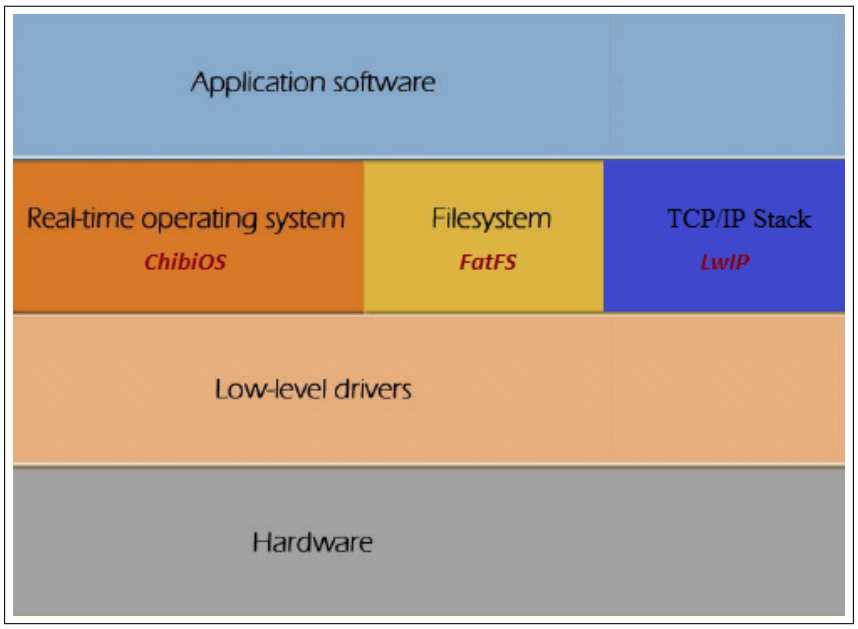

Figure 4.1: Resulting Stack Organization

The correct integration of the LwIP stack with the RT ChibiOS consisted basically in the implementation of the file sys_arch.c. This file contains different functions that allows the LwIP stack to use semaphores, mailboxes and all of their native methods. Furthermore the TCP/IP stack in order to use its functionalities required the usage of different low-level drivers that were provided by ChibiOS (MAC drivers for example). The correct integration of the FatFS in this environment required

basically the implementation of all the glue functions residing in the lower level of the FatFs module. By consulting the figure 3.9 the meaning of this operation would be more clear. The most common were disk_initialize(), disk_status(), disk_read(), disk_write(), disk_ioctl(). Also in this case, in order to let this module working properly and communicate with the microSD socket, the usage of low-level drivers were necessary (MAC and SPI drivers for example).

The conclusion of the FatFs integration results in the creation of a ready and working environment where each and every board supported by ChibiOS and including an Ethernet socket could be used with an embedded web server. Obviously the web server belongs to the application layer of the figure 4.1. So in conclusion a ready to use environment have been realized for the usage of a custom embedded web server.

4.0.2 MicroSD Based Web Server

The original and basic version of the little web server provided by the LwIP stack was developed with the feature of using only ROM file systems. This kind of structure could be useful in case of very little web servers composed of just some html static pages. But if we just consider the possibility of using images with dimension in the order of Kilo Bytes the context changes. Indeed remembering that the field of interest is real time, the memory management is a key concept due to the limited memory availability in microcontrollers. Furthermore it is mandatory to modify the file system ROM every time we would change the current web server organization. For these reason has born the idea of using the microSD card socket as container of the embedded web server. The result of this second task is represented explicitly in figure 4.2. So basically, different custom functions called by the http web server were implemented with the intent of handling data exchange with the microSD socket. Therefore a generic embedded web server residing in a microSD card have been made available for a customer.

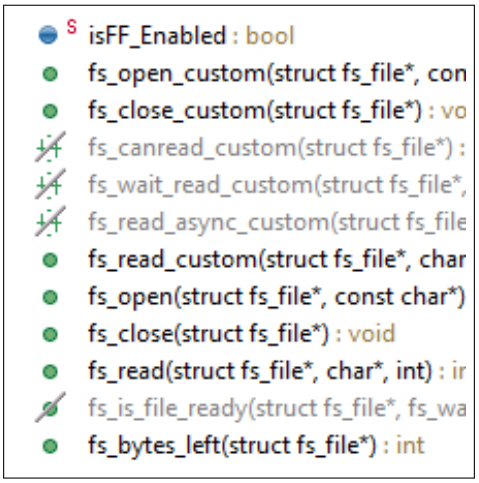

Figure 4.2: Resulting Implemented Functions

4.0.3 Dynamic microSD-Based Web Server

The final result of this thesis work is the launch of a dynamic web server that handles dynamic web pages and possibly also static web pages. This web server could be approximated to a 3 levels architecture by adopting a Common Gateway Interface (CGI) that is a standard protocol widely spread for interfacing external applications with a web servers. Basically on the server will run different thread generating web pages dynamically. Such programs are known as CGI scripts or simply as CGIs. I have decided to write their implementation in C language. So in the end a portable web serve environment have been provided and a dynamic web server using the CGI interface have been implemented with the optional feature to be stored inside a microSD socket.

4.0.3.1 Dynamic Web Server Demo

The last step of my thesis work was the development of a DEMO usable from a generic user. The latter could find this demo in the latest available version of Chibi Studio. Basically i have implemented a login panel to an hypothetical site, the mechanism can be divided in the following steps:

- **Step 1**:
  The client will send an HTTP GET to the main page of the web server. For example a client could type from a generic internet browser the url "http://192.168.1.10/Web_Server/login.html".

- **Step 2**:
  The server will respond with a static page residing inside the microSD card. This page have been written in html and CSS for the layout part. It can be seen in figure 4.3.

Figure 4.3: Web Server Login Page

- **Step 3**:
  At this point a static html page will appear to the user such that he could fill in a form with its desired parameters.

- **Step 3a**:
  The user could push the blue field, regardless of the parameters inserted there's no cgi task implemented at this moment, for this reason the server will respond with a static html page residing in microSD representing the classic 404 html error. Fore more details watch figure 4.4.

Figure 4.4: 404 error page

- **Step 3b**:
  In case of incorrect parameters there's no match with the credentials stored inside the server database. For this reason the server will respond with the same login page layout but with a credential error message, as you can see in figure 4.5.

- **Step 3c**:
  Finally, this is the case in which the user fills in the form with correct credentials. In this case the cgi URL will be invoked with the parameters inserted. Basically this will activate dynamically the thread responsible of the dynamic page generation. As explained in section 3.3.3, the file have been organized in fragments in order to insert dynamically some parameters, in my case i have used the credentials sent by the user. An universal algorithm, applicable and reusable for each and every cgi have been implemented in order to properly handle the fragmentation of the dynamic web page. As proof of the correct behavior the server will responde with the following figure 4.6.
  As it can be seen the exact parameters inserted by the user are illustrated.

Figure 4.5: Error Login Page

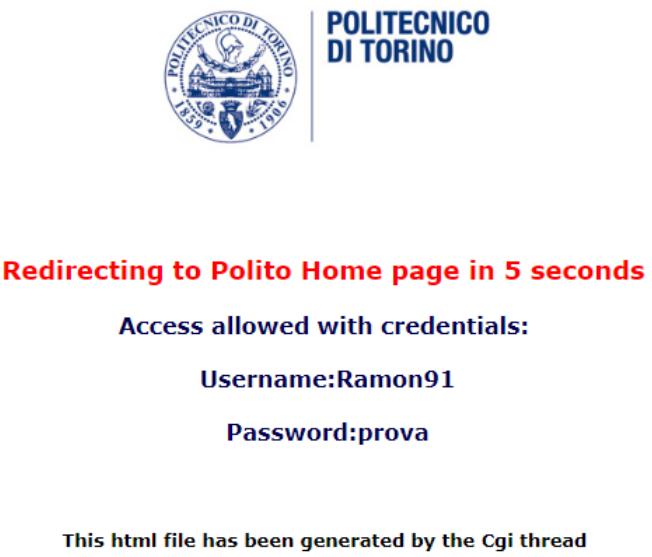

Figure 4.6: Correct Credentials Page

Finally, after 5 seconds the server automatically will send the main Web Server page.

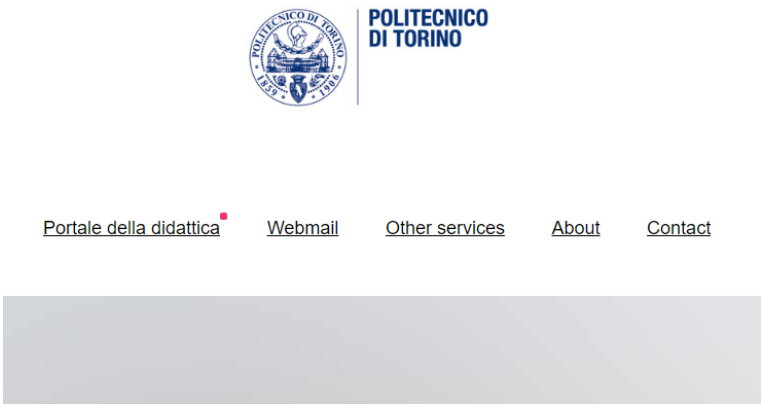

Figure 4.7: Main Web Server Page

4.0.4 Features

This thesis work has been designed with compliance to an embedded environment where memory handling and real time behavior must be guaranteed. Furthermore this project is portable on different platforms and web servers. Here following there's a brief explanation of the motivations.

4.0.4.1 Portability

The integration of the LwIP stack and FatFS file system with the RT ChibiOS results in the creation of a ready and working environment. So basically this environment is portable to all the boards supported by ChibiOS, including an Ethernet socket. Furthermore each and every custom web server can be linked to this stack with non relevant changes from the application level. So its main advantages is that a generic user could build its own web server on this structure without knowing and understanding its behavior. Obviously, this stack could be used also for platform not yet supported by ChibiOS, but in this case an user should modify or writing from scratch the low level drivers of this platform together with board and port files.

4.0.4.2 Memory Optimization

All the software components used are intended for embedded environments with low memory usage constraints. In particular the LwIP stack is the light version of a generic TCP/IP stack with a limited number of features and a small code size. The same discourse is valid for the FatFS module that is a FAT/exFAT file system developed for the domain of embedded system. The development of a dynamic microSD-based web server requested the implementation of techniques and algorithms in order to limit the RAM memory usage.

4.0.5 Performance Considerations

The two main performance/memory trade-off i came across during my thesis work were the read buffer size used when reading a file from a microSD and the program-

ming language adopted for the cgi process implementation.
Basically the size of the read buffer influences directly the total transfer rate, so if we suppose of having a small buffer size the total number of read operations increases. Let's remember that each read operation on the microSD involves an SPI communication and so having a small buffer will involve in a significant memory save at the cost of a slow total transfer rate. On the other hand a big buffer will reduce the number of read operations at the cost of the memory usage.

As a result of different considerations i decided to implement CGI processes in C language. C is a compiled language, this means that the code is analyzed and translated in compilation phase. So the execution time of this kind of programs is usually greater than the one of interpreted languages, where the analysis and execution is done at run-time. On the other hand interpreters reduce predictability, even if the execution time of a compiled program can be easily predicted basing on the clock cycles required by machine code instructions and the debugging phase is easier and faster. So the predictability considerations and memory over head due to the interpreter size lead me to use a compiled language as C.

4.0.6 Personal Considerations

Nowadays, in the IT field, Internet of Things is a very fashionable concept and considering that an embedded web server is essentially a little part of an entire IoT infrastructure, it could be very useful in light embedded applications. On the other hand this thesis work like said before provides a ready to use environment in order to build a total customizable web server. This environment has an high degree of configurability in terms of performance and memory usage. Furthermore the microSD usage is a feature very useful in case of web servers with a borderline dimension, where the size is greater then the average dimension of a standard embedded web server. Obviously the dynamic feature of the web server has a strong impact on the functionalities provided, indeed today is very rare to find servers providing just static web pages.

5
Conclusion

5.1 Achievements

The conclusion of the integration phase of ChibiOS, FatFS and LwIP results in the achievement of a ready to use environment for each board supported by ChibiOS, including an Ethernet socket on board. Therefore a ready to use environment have been provided for the usage of a custom embedded web server. Furthermore a generic embedded web server residing in a microSD card have been made available for a customer instead of using the standard architecture based on a ROM file system. A dynamic web server that handles dynamic web pages and possibly also static web pages have been provided. This web server adopts a Common Gateway Interface (CGI) that is a standard protocol widely spread for interfacing external applications with a web servers. Finally also a demonstrative demo have been implemented that verifies the correct behavior for all of the features listed above.

5.2 Limitations and Possible Improvements

As well explained in this thesis document the main limitation consist in the memory size, in particular of the flash and RAM modules. Another important limitation is that this ready to use stack environment can be applied only on boards supported by ChibiOS and mounting an Ethernet Socket. It is in every case possible to integrate others platform to ChibiOS, but in this case the user must interact with the hidden part of this stack, loosing its main feature. Several improvements can be made to further increase the efficiency of this web server:

- A first improvement could be to implement other two possible architectures for a 3 tier web architecture. One of the most common is the Server Side Include organization. In this case it is avoided the complexity of the CGI to handle both static and dynamic pages. Indeed the HTML pages includes different directives and commands that are executed by the server before providing the requested page. Another mechanism could be the Mobile Code. In this case the server will send to the client html code and another kind of code that will be executed by the client through small programs. In this manner the load will be loaded to the client through its processing capabilities.

- Another possible improvement consist in the application of another file handling mechanism, possibly with an higher performance in terms of execution

time. In particular i'm discussing about the fragmentation algorithm used inside the cgi thread. Furthermore in order to increase the performance it is contemplated to use an interpreted language for the cgi thread, considering always its drawbacks.

5.3 Future Challenge

In my opinion in order to extend the applicability of this thesis work it could be very interesting to apply all the main features discussed above on a board mounting a WiFi module. In this manner the connection is not limited to a physical linkage, allowing it to be applicable in many other domains.

Bibliography

[1] O. Vermesan, P. Friess, P. Guillemin et al., "Internet of things strategic research roadmap," in Internet of Things: Global Technological and Societal Trends, vol. 1, pp. 9–52, 2011.

[2] I. Peña-López, Itu Internet Report 2005: The Internet of Things, 2005.

[3] Giovanni Di Sirio, 2017, ChibiOS Architecture, http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:architecture.

[4] Giovanni Di Sirio, 2017, ChibiOS HAL Layer, http://www.chibios.org/dokuwiki/doku.php?id=chibios:product:hal:start.

[5] Giovanni Di Sirio, 2017, ChibiOS HAL Design, http://www.playembedded.org/blog/chibioshal-design-an-object-oriented-approach/

[6] Giovanni Di Sirio, 2017, ChibiOS MailBox, http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_mailboxes

[7] Giovanni Di Sirio, 2017, ChibiOS Events, http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_events

[8] STMicroelectronics, 2013, LwIP TCP/IP stack demonstration for STM32F4x7 microcontrollers.

[9] Wikipedia, OSI Model, https://en.wikipedia.org/wiki/OSI_model

[10] Design and Implementation of the LwIP TCP stack https://www.artila.com/download/RIO/RIO-2010PG/lwip.pdf

[11] Wikipedia, File System, https://en.wikipedia.org/wiki/File_system

[12] Wikipedia, File Allocation Table, https://en.wikipedia.org/wiki/File_Allocation_Table

[13] STMicroelectronics, 2014, Developing Applications with FatFS,
http://www.st.com/content/ccc/resource/technical/document/user_manual/61/79/2b/96/c8/

[14] ELM Chan, FAT Filesystem,
http://elm-chan.org/docs/fat_e.html

[15] ELM Chan, FatFs Generic FAT Filesystem Module,
http://elm-chan.org/fsw/ff/00index_e.html

[16] Wikipedia, Dynamic web page,
https://en.wikipedia.org/wiki/Dynamic_web_page