Balen Azez

Computerisation of Classroom Attendance Marking by Means of Real-time Wireless Network Systems

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Bachelor's Thesis
6 May 2011

Metropolia

Abstract

Author	Balen Azez
Title	Computerisation of Classroom Attendance Marking by Means of Real-time Wireless Network Systems
Number of Pages	62 pages + 8 appendices
Date	6 May 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Engineering
Instructors	Hannu Laine, Principal Lecturer Antti Piironen, Principal Lecturer

A proposal was put forward to computerise classroom attendance markings of Metropolia. The initial idea was to design and build a portable device that can detect student identification data from RFID keys and record them to be later transferred to a PC. However, based on the final preliminary design sketch, the initial proposal was extended to a wireless portable device as part of a real-time wireless system with a central server. This paper describes a possible solution for the final proposal from combining existing technologies such as the Microchip TCP/IP Stack, POSIX real-time APIs, and various embedded hardware modules. The goal of the project was to build and assess a real-time wireless system that could be used in classrooms for attendance markings by using RFID keys that would ultimately replace the present system.

Often combining existing technologies result in efficient new systems. The project design of the computerisation of classroom attendance markings utilised simple engineering methods such as adaptability, concept maps, and flowchart diagrams. Applying these methods paved the way for producing a powerful wireless unit out of several existing technologies that had been interfaced together. The resulting product and its efficiency during system testing conclude that engineers should first seek adaptation and combination of existing technologies before they resort to new solutions from scratch.

With the inclusion of ADC and analogue sensors, the product of this project could be modified and turned into a powerful, wireless data acquisition system. Such a system could be used as a solution for problem designs in other areas such as farms and field data collections for scientific researches and studies.

Keywords	attendance, classroom, TCP/IP, client-server, wireless, IPC, real-time, Microchip, PIC32MX

Contents

Abstract

Abbreviations

Abbreviations

ACK	Acknowledge
AES	Advanced Encryption Standard
API	Application Programming Interface
ARP	Address Resolution Protocol
BSD	Berkeley Socket
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CS	Chip Select
DHCP	Dynamic Host Configuration Protocol
EEPROM	Electrically Erasable Programmable Read-Only Memory
GH	Gigahertz
HD	High Definition
HTTP	Hypertext Transfer Protocol
I/O	Input/output
ICMP	Internet Control Message Protocol
ICSP	In-Circuit Serial Programming
IDE	Integrated Development Environment
IP	Internet Protocol
IPC	Inter-Process Communication
IPv4	IP version 4
IPv6	IP version 6
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light-Emitting Diodes
MAC	Media Access Control
MCLR	Memory Clear
MCU	Microcontroller Unit
MSS	Maximum Segment Size
P2P	Peer-to-Peer
PAN	Personal Area Network
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PHY	Physical Layer
POSIX	Portable Operating System Interface for Unix
PSK	Phase-Shift Keying
RFID	Radio-Frequency Identification

RR	Round-Robin
SPI	Serial Peripheral Interface
SSID	Service Set Identifier
SYN	Synchronise
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
WAN	Wide Area Network
WAP	Wireless Access Point
WPA	Wi-Fi Protected Access

1 Introduction

To computerise classroom attendance marking, a senior lecturer, here at Metropolia, proposed an idea of a portable device that could register the attendance using Radio-Frequency Identification (RFID) cards. Instead of the present system, which is a list of names on paper sheets, a teacher could carry the device into classrooms, and students, instead of manual marking, simply approach their electronic RFID keys to these devices and confirm their presence. To improve the usability of the system and eliminate the manual data transferral after the process of registration, a central server was added to the original idea to form a client-server model. Data can now be automatically collected from the portable devices without additional steps. This paper describes the steps that are required for designing and implementing such a system in a Linux-GNU environment.

The advancement of embedded technologies has made available different routes to achieving the same engineering goals. Since the problem design of the computerisation of the classroom attendance marking suggested an open-end problem, i.e. a problem with multiple solutions, the availability of different technologies rendered the solution cost-effective. With the exception of designs that might require engineering solutions from scratch, engineers mostly either adapt an existing solution or combine different solutions to form a new system. This is an engineering case where reinventing the wheel was unnecessary since combining existing systems resulted in an efficient solution.

The goal of the project was to observe and evaluate a real-time, wireless network system that was designed as a solution for the classroom attendance confirmation. While detailed explanation of some of the black boxes that are part of the system solution is outside the scope of this paper, basic explanation of the functionality and interfaces of these sub-systems will be covered. The scope of the paper is limited to explaining the higher-level design of the system and interfaces that connect its sub-systems together.

2 System Study

Registration systems are implemented in many organisations for various reasons. While the purpose of these systems is not a focus in this report, it is clear that statistical information is required by most of those organisations that have a large number of members or customers. Educational organisations, on the other hand, have their own reasons for collecting classroom attendance data. Since collecting attendance data in these institutions has not been computerised, the proposal of transforming the present system was alluring and I decided to face the challenge and design a solution for it.

In the present attendance registration system, teachers have to either ask the students to write down their names on a piece of paper, or they have to provide a list of enrolled students and ask them to mark their names. This means that excessive manual efforts go into the process of classroom attendance registrations. After the data has been collected, it takes even more time and efforts to transfer the data onto other systems for other purposes such as, perhaps, data analyses. One of the major flaws of the present system is the fact that attendance registration information is not properly used to help improving course timetables and classroom bookings. Another important piece of information which the present system lacks is the time of individual attendance records. For example, a question of whether the majority of students were present at the beginning of a class or at later times could help in improving classroom related researches.

The idea behind the new system is to design a portable, wireless device that is light, pocket-sized, and easy to use to be carried by teachers into classrooms for student attendance markings. As students have already been given RFID keys, they can now simply approach the keys to the devices and register their presence within seconds. The question of whether or not the recorded data should be kept on the device was later addressed with extending the original specification to include a central server for collecting the data after it has been stored on the device. That is, the device should store data temporarily until it transfers it to the server. Since schools have multi-classrooms with multiple courses starting at the same time, a server will have to accept multiple connections concurrently. Furthermore, data should reach the server within seconds if time was to be recorded alongside with the recorded data.

A computerised attendance registration system, therefore, would provide a number of advantages over the present system. Accurate attendance information, ease of use, ease of access of old data, the ability to generate statistical graphs out of collected data are just a few examples of what could be gained by implementing the new system. In short, general attendance behaviour of students could be predicted and designs of timetables improved based on data analyses. Figure 1 shows a basic sketch of the system collectively.

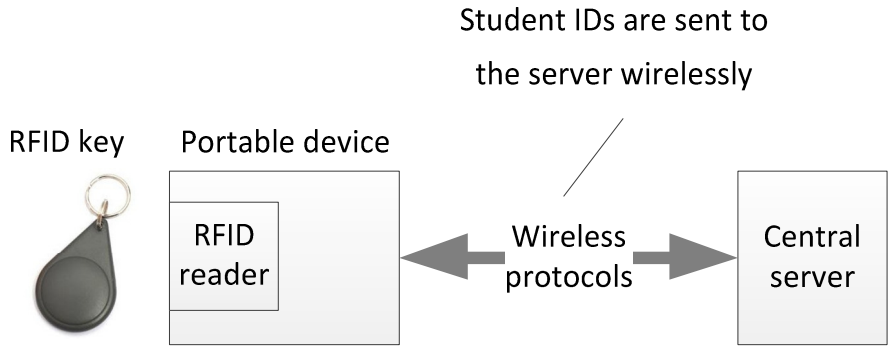


Figure 1. Basic sketch of the suggested computerised system

Figure 1 illustrates a simple diagram of the new system for the classroom attendance markings. The system has a central server, which is fixed at a defined place, and one or more portable wireless devices that can be carried into classrooms. A teacher has to first approach his or her RFID key to a portable device, then the portable device will contact the server with the ID of the teacher. The server, in turn, looks it up in the timetable, finds proper information about the classroom and the course that is being lectured, and sends an acknowledgment to the portable device in order to start recording the attendees of the course. Two methods can be used to end the attendance marking; the teachers can, once again, approach their RFID keys to end the operation, or implementing time-based registration that will end the recording after a certain time has been passed.

The data that will be exchanged between the portable device and the server, therefore, is simple texts and numbers. The users of the portable devices are teachers and students. They interact with the portable devices with their RFID keys only. The server, on the other hand, interacts with the portable devices and also requires maintenance persons. For usability reasons, the portable device can have one or more

visual or sound feedback. When an RFID key is approached to it, the device can indicate whether data have been recorded and sent to the server or, perhaps, an error prevented it to do so. A green light can be used to indicate a success and a red, an error, or simply unknown or unrecognised student.

3 Theories and System Units Analysis

3.1 General Overview

The new classroom attendance marking system can be divided into three major parts:

- a server that can handle multiple connections concurrently
- a portable device that can communicate with the server wirelessly
- an RFID reader for detecting unique identification numbers of RFID tags.

The system design requires a single, central server. The server can be built within an operating system such as Linux-GNU. The portable device, on the other hand, must be built either from scratch or from existing solutions. In order to minimise the complexity of the system, instead of using an RFID reader, simulated code can be generated within the portable device itself. This should not affect the overall system design since the RFID reader would only input alphanumerical data to these portable devices, which can be simulated with a programme written in C. Furthermore, it is more convenient to include some form of outputs in the design of the portable devices as part of a visual feedback to its users. An output can be in the form of lights, such as light-emitting diodes (LED), or by using a liquid crystal display (LCD) for showing texts and numbers.

Full portability of the portable devices can be achieved by making them wireless. This implies that the server will have to be running on a wireless-enabled computer as well. A wireless router, in this case, can be joined with a computer, on which the server is running, to form a wireless server.

3.2 A Process-based Server

A single portable device can register the attendees of a single classroom at a time. The device should detect unique identification numbers that are stored on the RFID keys. The identification numbers represent the holders, students or teachers, of the keys. The device should then transfer these data to the server wirelessly. It is clear that the server receives data from different classrooms at the same time.

Servers are composed of multiple processes that are performing tasks when handling clients. A process is an allocated memory region by the kernel of an operating system within which a set of executable codes is performing one or more tasks. They are like little computer systems only in the form of software within the system memory of a computer. Processes are independent of each other and programmatically isolated from the operating system and other processes. In this way, a crash of a process will not result in the crash of the operating system or other processes. The kernel of an operating system stores all the data that is related to a process in the process itself. Data includes the state of the current registers of the microprocessor such as programme counter, state of variables, and return addresses. The number of processes created within an operating system is only limited by the available resources. [1] Figure 2 shows a block diagram of memory regions allocated for processes.

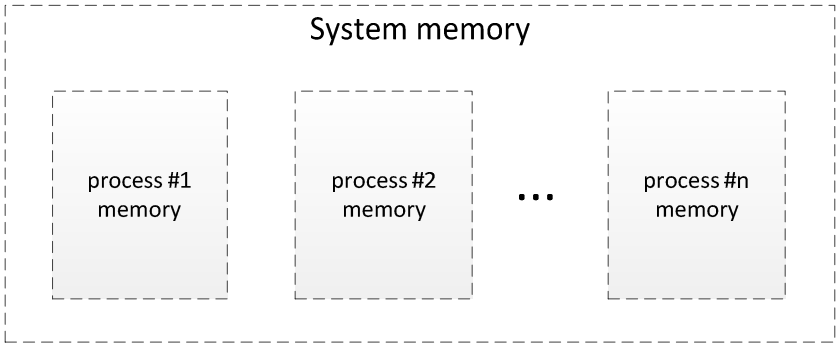


Figure 2. Memory regions within system memory allocated for processes

Figure 2 visualises memory regions that are allocated by the kernel for running processes. Complex operating systems, such as Linux-GNU and Windows, support multi-processing tasks. In such environments, two or more processes are running at the same time through sharing the time of the microprocessor. A special process,

known as scheduler, slices and allots a microprocessor time to all existing processes based on their priorities. If their priorities were the same, the time is shared by the processes equally and the scheduling is known as Round-robin (RR). If the priorities of the processes were not the same, the scheduling would be called pre-emptive, which means that a process might pre-empt another process if its priority was higher. [1]

The scheduler bases its decision for time slicing on the states of the processes. When processes are created, their current states change according to the flow of their executable codes. They might be at the running state or halted, also waiting state, by a condition or by the kernel. There are four states of a process: running, waiting, ready, and zombie states [1]. Figure 3 depicts the states of a process.

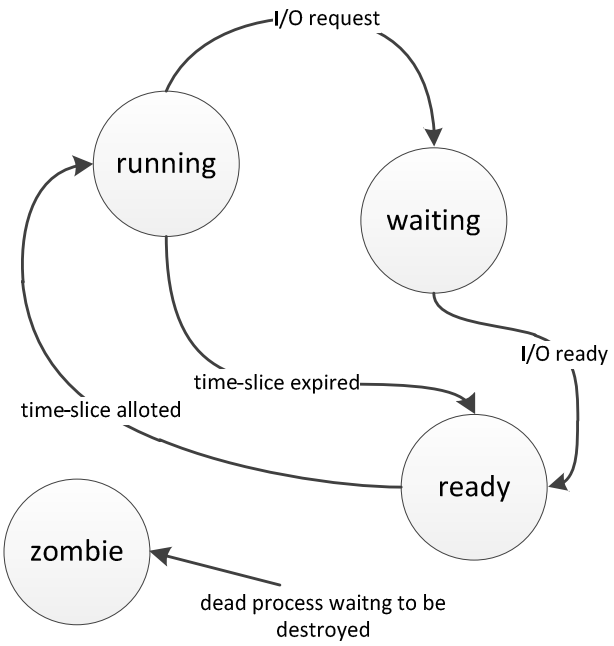


Figure 3. The states of a process

As can be seen from figure 3, when a process is in the running state, its time slice may expire. When that happens, the process will be switched to the ready state waiting for the next time slice. A process may also be switched to the waiting state when, for example, accesses one or more input/output (I/O) resources. The zombie state occurs when a process completes its tasks while the kernel has not yet fully released it. [1]

The structure of Linux operating system is such that when processes are created they are literally cloned from other processes. This cloning goes back to a special process called "init", which is an initialisation process that starts the Linux operating system. A newly cloned process is called a child process, or simply a child, and the process from which it was cloned, a parent process. When processes are created, they are assigned unique identifiers for the purpose of communications. The identifier is usually represented by an integer number. The child identifier is referred as "pid" and the parent identifier as "ppid". [2,126] Figure 4 shows a topology of children and parent processes

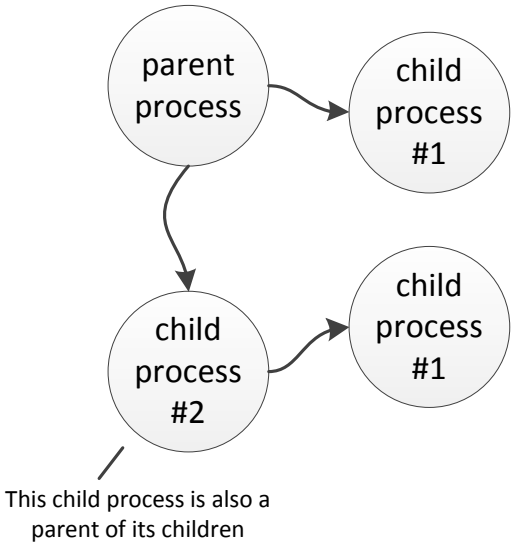


Figure 4. A parent with two children and one grand child

Figure 4 shows a logical representation of parent and child processes. As it can be seen from the figure, a child process can also have one or more children. Furthermore, the complex ownerships and permissions in Linux environment are also applied to processes. A process is said to be owned by a user which, in turn, belongs to a group. A process is also belongs to a *process group*. A process group organises relationships between processes. Children of a parent share the same process group with the parent. Process grouping is used in signal communications among parents and children and when using shared resources such as pipelines. [2,127] Pipelines are not covered in this paper.

Linux starts the init process with an identifier integer "1". Other processes are cloned from init process with special function calls known as *system calls*. Two of these system calls are fork() and clone(). When a child process is created, it is also given access to one or more resources of its parent. In this way, resources are not duplicated but shared. Resources are not released by the Linux kernel until all processes that share them have finished their tasks and exited. [1]

3.3 Inter-process Communications

A process-based server is typically composed of two or more processes. These processes, at some point, need to communicate with each other. This communication is known as inter-process communication (IPC). The IPC is the exchange of signals and data between one or more processes or between processes and the kernel. Since processes are isolated from each other, communications must be arranged through defined mechanisms that are managed by the kernel. There are three major mechanisms with which the IPC is implemented: signals, pipes, and sockets. Other mechanisms exist such as queues, semaphores, and shared memory. [1] The focus in here is on signals and sockets only

Signals are fundamental to IPC. They are asynchronous, one-way messages used between processes or between processes and the kernel. A process may also send a signal to itself. When a process receives a signal, the normal flow of its instructions is interrupted and replaced with a set of codes, defined as a function handler, and executed as a response to the signal [1]. After handling the signal, the process resumes from where it was interrupted. A signal may also have a default response or simply be ignored. Sockets, on the other hand, are used in advanced IPC. They can be used to exchange messages between processes regardless of their locations; whether they are on the same machine or on two or more different machines on a network. This means that sockets form the basis for network communications over an intranet or the Internet. [2,13,19]

3.4 Real-time Systems

In any system, if obtaining results depended on the timing of the steps that were required to attain them, this system is said to be a real-time system. This is usually the case when several tasks need to run at the same time or in parallel. In other words, time is an essential part of real-time applications in that it directly affects the intended results. In Linux environment, real-time operations are defined by one of the portable operating system interface for Unix (POSIX) standards. POSIX real-time, or POSIX.4, is a special standard that defines a set of application programming interface (API) for real-time programming. One major aim of POSIX is portability across different platforms. [3,2-9]

As two or more classrooms may use the portable devices at the same time for attendance registration, the server of the system must support some form of real-time communications. However, system designs that require real-time concepts are not all the same. There are cases in which results must be available within a defined interval of time otherwise it may cause unwanted, and often, disastrous outcomes. Real-time in such situations is called a *hard real-time*. Examples are applications that control nuclear power generators or medical equipment of which malfunctions might result in deaths. On the other hand, there are cases that require real-time and, at the same time, occasional delays do not cause system failure or deaths. Real-time in these situations is known as a *soft real-time*. The new classroom system and database applications, are examples where slight delays in communications will not result in a system breakdown. [3,2-9]

3.5 A Client-server Model

The portable devices of the new system, together with the server, form a network system that is based on a client-server model. In this case, the portable devices are said to be clients of a central server. The server may be running on a Linux or a Windows platform. The portable devices, from now onwards in this paper, will be referred to as either "clients" or "portable devices" interchangeably. The system model, collectively, forms a network of separate units, which are clients and a central server, and one or more protocols that make possible the communications between them. The

central unit that collects data from the portable devices is said to be the "server" of the system. In a Linux environment, servers are processes that provide or collect data from one or more remote, client processes. [4,3-4]

It is clear that the portable client devices should either be made out of processes within a complex operating system, just like the server, or they will have to behave like client processes should they be designed in different systems. Lastly, both clients and the server are said to be *nodes* of the network system. [4,3-4] Figure 5 shows a general block diagram of this model.

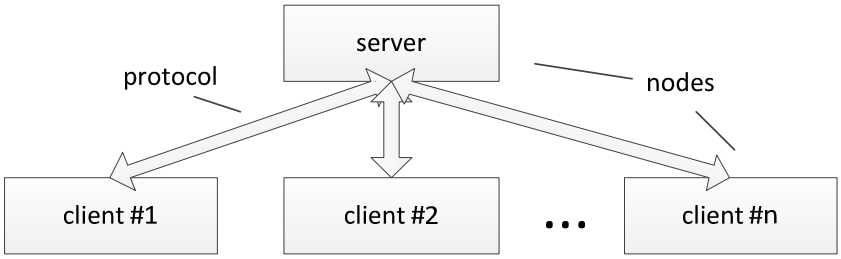


Figure 5. A server with multiple clients

As illustrated in figure 5, a server can have one or more clients to serve. The double headed arrows represent bidirectional communications between the server and the clients. Client-server model solutions demand a soft real-time design since slight delays in the timing of events will not result in total system failure; or at least such delays will not result in injuries or death even in the event of a breakdown of the system. For protocol simplicity, a server is made to listen to clients and respond to them only when they initiate requests for bidirectional communications. [3,12]

Two major disadvantages of the model are server overloading and lack of robustness. As the number of clients increases, the performance of the server decreases and eventually becomes overloaded. Without backups, a failure in the operation of the server will render the system useless. Solutions should take these problems into consideration.

3.6 TCP/IP Sockets

In order to establish network communications between the clients and the central server of the attendance registration system, one or more protocols are needed. A protocol is a set of rules for exchanging messages between nodes of a network. Protocols are required because different nodes on the network may have different hardware architecture. A protocol ensures compatibility between different hardware brands and architectures. Two of such protocols are transmission control protocol (TCP) and internet protocol (IP). [4,4,35] A general illustration of an internal structure of nodes of a network is given in figure 6.

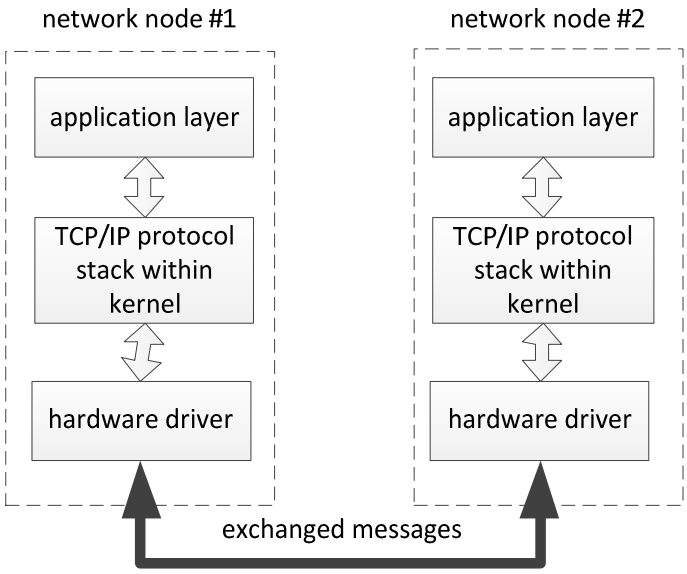


Figure 6. A diagrammatical description of two nodes in a simple network

As can be seen from figure 6, nodes of a network can be diagrammatically represented as different layers with the bottom-most layer as hardware drivers. A protocol in this diagram is represented as a conceptual layer that defines the format of the messages that are exchanged between the nodes.

The internal structure of TCP is complex and dependent on other network layers and protocols such as IP. IP is a way of addressing nodes in a network. The kernel of a computer operating system usually provides a set of API for network communications known as TCP/IP stack. The TCP/IP API covers all the required steps for the

establishment of network communications between two or more nodes. Even if a node did not support a conventional, complex operating system, such as Linux-GNU, it would still have to have an equivalent stack that supports TCP/IP. A client node requests and establishes a TCP/IP connection from and to a server node, the exchanges of messages occur between them, and the same client closes the connection after all messages have been exchanged. TCP messages require acknowledgements from the receiving sides which makes it a synchronous protocol. [4,4,35]

Sockets, on the other hand, are data structures that are created and manipulated in application layers by processes. While sockets are mainly used for IPC on different machines across networks, they can also be used for local IPC on the same machine. [4,67] Figure 7 presents a socket-based IPC between two processes.

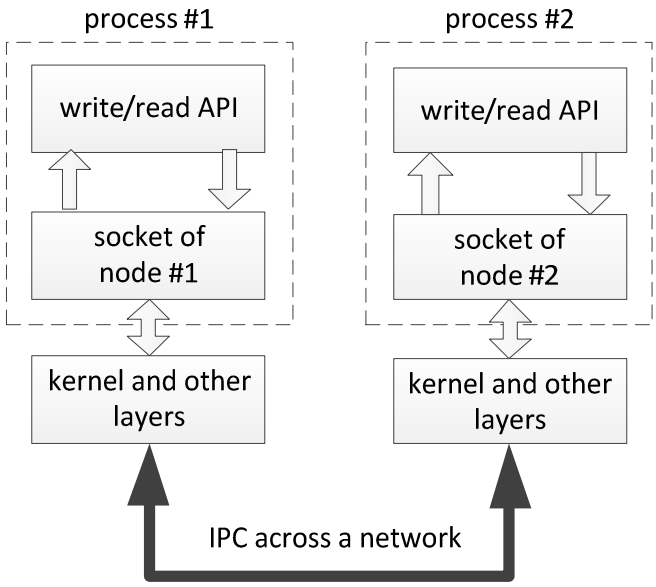


Figure 7. Socket-based IPC across a network

Figure 7 illustrates a socket-based IPC over a network. Each process opens its own writable socket. Processes pass their socket data structures to the kernel and the kernel to the processes in each message exchange. They do that because processes are isolated from operating systems and are not allowed to access hardware drivers directly. The data structure of a socket contains, among other things, a destination address and a type of communication. The destination address is either a 32-bit IP

version 4 (IPv4) address or IP version 6 (IPv6). [4,67] IPv6 is beyond the scope of this paper and hence will not be covered.

In a Linux-GNU environment, there are a number of socket data structures from which two are dedicated for internet communications. They are sockaddr_in for IPv4 and sockaddr_in6 for IPv6. The focus in this paper is on sockaddr_in, which is designed for IPv4. Figure 8 is a block diagram of an Ipv4 socket.

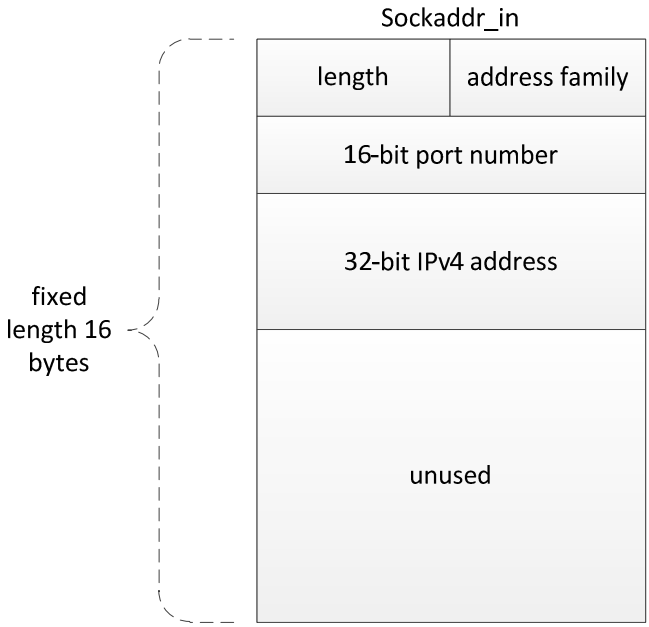


Figure 8. The IPv4 socket data structure [4,73]

Figure 8 shows the internal data structure of an IPv4 socket. The socket has a fixed length of 16 bytes from which 8 bytes are unused. The rest of the structure is divided over its members: length, address family, port number, and IPv4 address. [4,73]

An IPv4 address is arranged in four bytes. Each byte represents an integer number between 0-255. The first two bytes represent a network address and the remaining bytes, a host address. An example of the text format of this address is 192.168.0.10. Note that the numbers are separated by a dot for clarity. The first two numbers, 192.168, represent the network address and the, 0.10 is the address of the host, which is essentially an end node.

When one or more nodes form a network and communicate using TCP/IP protocols, a standard, known as *Ethernet*, is used to define the physical layer (PHY) of the network. PHY is basically the hardware implementation that performs the actual data communications between nodes of a network. A network on which two or more nodes are communicating directly through their PHY is called a local area network (LAN). However, when two or more LANs are joined together through *routers* to form a larger network, such a network is called a wide area network (WAN). [4,4-5] A simple topology of a WAN is shown in figure 9 below.

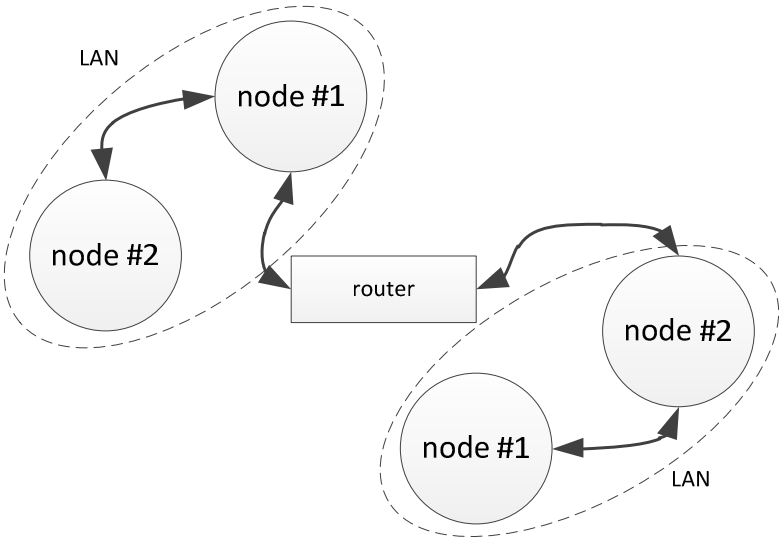


Figure 9. Basic topology of a WAN with two LANs and a router [4,5]

As can be seen from figure 9, one or more routers join two or more LANs to form a WAN. An example of a WAN is the Internet [4,5]. A node in a network structure is essentially a process performing tasks which may involve an IPC. The process can be a server listening to requests within or from outside the machine that is running on. Other processes can be clients connected to the server or asking for permission from the server for bidirectional communications. There are special networking API functions that open sockets, establish connections, and send or receive messages between processes.

3.7 TCP/IP Linux API

TCP messages are exchanged in small chunks of information called *packets*. Packets are divided into bits and bytes to represent, among other things, the destination address and data to be exchanged. To establish a TCP connection, a server must first be running and ready to accept clients for TCP network communications. In a Linux platform, the functions that are used for preparing a server are socket(), bind(), listen(), and accept(). After these functions have been respectively called with proper arguments, the server is ready to accept clients for communications. The clients, on the other hand, should also open a socket and call a special function called connect(), which will send a request to the server for communications. The hardware of the clients may not necessarily support a complex operating system such as Linux. In any cases, a TCP/IP stack must be available to the clients as well. [4,37]

The steps for a TCP connection between a server and a client are as follow:

The server:

- prepares and opens a socket with the function socket()
- calls a bind() function for defining a port number and associate it, together with the IP, with the socket
- calls listen() function to switch the socket to listening state, also known as a listening socket
- finally, calls the accept() function to receive requests from clients and establish connections. [4,37]

The client:

- prepares and opens an available socket
- calls connect() function, with an available local port number and the IP address of the server, to request for a connection with the server
- waits until the connect() function returns indicating an establishment of a connection or an error. [4,37]

To connect to and synchronise with a server, clients firstly send a packet known as a synchronise (SYN) signal, within which contains TCP/IP headers and other TCP options

such as maximum segment size (MSS). After the first call of the connect(), the client waits for an acknowledge (ACK) signal from the server. When the server receives a SYN from a client, if everything went well, it sends back an ACK packet with a SYN request of its own. Lastly, when the client receives this packet, it must send its ACK to the server. This allows the server to dedicate the present socket to this client and call a read() function for receiving its messages. This is called a TCP three-way handshake for a network connection establishment. [4,37] A set of simplified steps of a client-server communication is illustrated in figure 10.

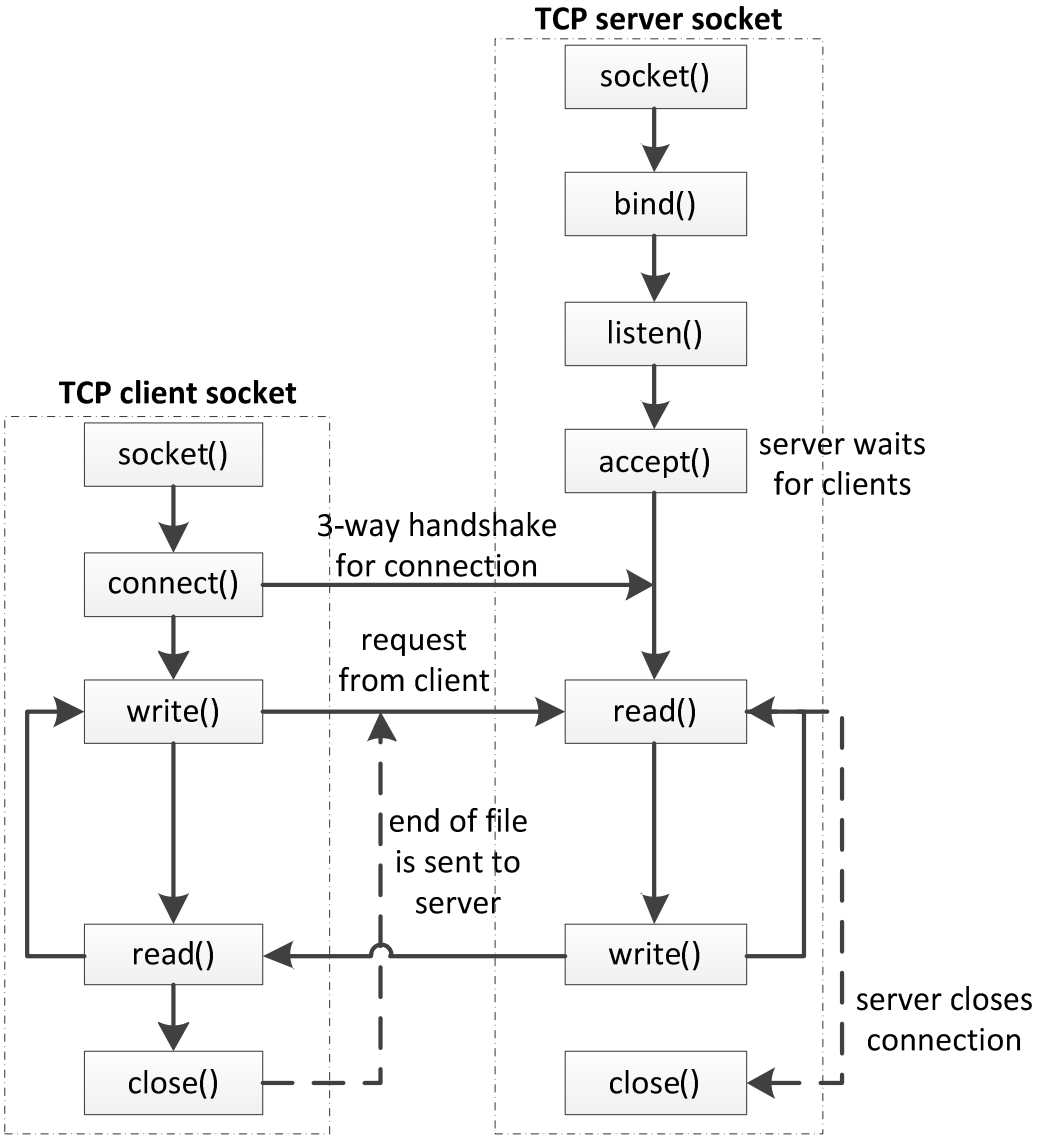


Figure 10. TCP API for network communications between a server and a client [4,95]

As depicted in figure 10, the sockets of clients and servers run through several states. The server must first open and prepare a socket for communication before a client can connect to it. It is worth noting that a server socket, after it has been fully prepared for connections, blocks and waits for SYN signals from clients. Another point to note is that, when a client closes its socket, an *end-of-file* signal is automatically sent to the server to close its client specific socket and de-allocate resources. [4,95]

3.8 Concurrent Servers

In designs where clients need to stay connected to a server for varying times, the server must be able to serve each client within an acceptable delay. Otherwise the system may spend too much time with one client while other clients may be waiting for connection ACK signals. In situations like these, a server must be able to handle multiple clients at the same time. Such a server is called a *concurrent server*. The new classroom attendance marking system requires a concurrent server since its clients request server connections independent of each other, whether concurrently or on different times. The Linux operating system provides two methods for building concurrent servers. The simpler method is to create and dedicate a child process for each client using the fork() function. The other method, which will not be further explained in this paper, is the use of threads. [4,114] Figure 11 below shows a logical diagram of a concurrent server.

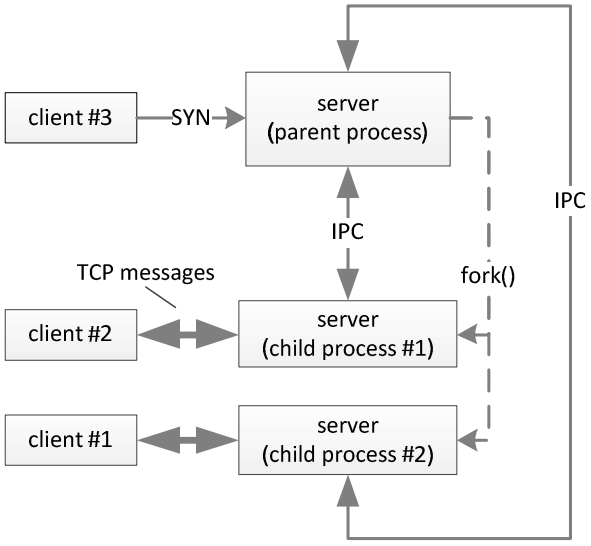


Figure 11. A concurrent server with two children already connected to two clients

Figure 11 illustrates a concurrent server with two child processes connected to two clients while another client is trying to establish a connection. The server, in this case, is in an infinite loop of waiting for clients, acknowledging connection requests, and creating child processes one for each client using the fork() function. The server process and its children can still establish IPC using, for example, pipes.

4 Existing Solutions

4.1 Wireless Solutions

In a system design where clients are required to be portable, wireless solutions can be used. For the server side, a wireless solution must handle multiple connection requests from multiple clients. This implies a single wireless system, such as a wireless access point (WAP), which can handle multiple connections at the same time. An example of a WAP is a wireless router. For the client side, a wireless transceiver system that can support duplex communications is enough since clients in a client-server model do not need to communicate with each other. This section introduces different, available solutions that can be implemented in a client-server model.

There are several wireless solutions for data communications with different specifications, each designed for a particular situation. Among wireless design specifications are range, speed, and power consumption. For example, a wireless sound system requires low data bandwidth, while a high definition (HD) video streaming requires high data bandwidth. In general, wireless solutions can be divided into two major groups: wireless modules and miniature, wireless computer systems such as Beagleboards and wireless routers.

Wireless modules are designed to be integrated into larger computer systems. They require host computer systems to control their operations and tasks. They can also be integrated into embedded systems. They can have simple interfaces, such as serial peripheral interface (SPI) bus and universal serial bus (USB), or more complex interfaces such as peripheral component interconnect (PCI). Examples of wireless modules are Wi-Fi, Bluetooth, ZigBee, and sub-gigahertz (GH) modules.

Wi-Fi is a certified radio transceiver module based on 802.11 standards. The module works in 2.4 GH frequency band which is license-free. Different generations of Wi-Fi modules are identified by a postfix letter a, b, g, or n that is added to 802.11 numbers. For example, a Wi-Fi module with 802.11g has a frequency of 2.4 GH and a maximum data rate of 54 Mbps. Wi-Fi modules implement Wi-Fi protected access (WPA) to secure hide messages during network communications. They are used in laptops, mobile phones, and other portable devices that require wireless, network communications. [5]

Bluetooth modules are designed for what is known as a wireless personal area network (PAN). It is a module that operates in a 2.4 GH free-licenced band. A wireless PAN refers to devices that are within a few meters from users and that they are connected to each other for the purpose of low bandwidth, data communications. Examples are Bluetooth-capable printers and cameras where, instead of cables, users can transfer photos to a printer wirelessly. Popular Bluetooth-based networks are of low range of up to 10 meters only. One major advantage of the Bluetooth technology is its low power consumption. While a typical Bluetooth module uses 2.5 mW of power, it has a maximum data rate of 3 Mbps only. Lastly, Bluetooth modules support a number of security technologies among which are data encryption and connection authentications. [6]

ZigBee wireless modules operate in the free-licenced band of 2.4 GH based on enhanced IEEE 802.15.4 standard. ZigBee wireless standard has been developed by a consortium called ZigBee Alliance. Apart from global 2.4 GH band, ZigBee standard also support lower, regional frequencies of 915MHz and 868MHz. ZigBee modules are used in wireless systems such as smart home networking, remote control products, and telecom services. They can be used in any type of wireless communications. [7]

Sub-GHz wireless modules work in frequencies below a GHz. These frequencies are known as regional frequencies. The range of the frequencies is 434-955 MHz. Sub-GHz modules are mainly used in automation, home and business electronics, medical applications, children toys. Microchip Technology Inc. has developed two special protocols called MiWi and MiWi peer-to-peer (P2P) for data communications

management for these devices. Among specifications of these modules are short-range, low rate, and low-power consumption. [8]

There are also miniature computer systems some of which have wireless solutions built into them. An example of these systems is Beagleboard. Beagleboard is a miniature, low-cost, complete computer system. The board is managed by an ARM-based, 32 bit microprocessor. The version 4 of the board has about 256 MB of RAM and 256 MB of NAND flash memory for storing boot-loaders and initialisation instructions. It has a powerful graphics card that can handle different resolutions up to 1280x1024 pixels. Communications with the board can be performed through various I/O interfaces among which are universal asynchronous receiver/transmitter (UART), secure digital (SD) card, and USB [9]. Together with a USB Wi-Fi dongle, Beagleboards can be turned into efficient wireless solutions.

Each of the wireless technologies that has been briefly described above has advantages and disadvantages when it comes to client-server model of networking. The focus of this paper is not on detailed descriptions of each of these technologies; rather, an aim is to outline what is available as solutions for the wireless network units of the new classroom attendance marking system. Selection of a technology over the others will have to be based on different factors, among which are software and hardware supports from manufactures and, perhaps, third-party software availability.

Based on the popularity and technologies used, Bluetooth and ZigBee are best fitted in P2P data communications. P2P networks do not have central servers or coordinators. In such networks two or more nodes of a network are connected directly. Beagleboard-like solutions are expensive and, as far as portable systems are concerned, they are not the best candidate when it comes to power consumption.

4.2 Microchip Solutions

4.2.1 Overview

Microchip Technology Inc. provides full-scale embedded solutions. Among the technologies that they produce and support are microcontroller units (MCU) and wireless modules. They also produce, among other products, flash and RAM memories, power regulators and interface technologies. Microchip also develops and maintains MPLAB integrated development environment (IDE), software solutions such as TCP/IP stack, and a full library functions for the operation of various Microchip technologies.

Based on sufficient research on different technologies and available solutions, I decided to implement and combine relevant Microchip technologies as a solution for the problem design of the classroom attendance marking system. There are several reasons for this decision which make the Microchip solutions superior to its competitors:

- Microchip has dedicated solutions for embedded engineering problems that can be configured in many different ways for particular situations.
- Microchip solutions are relatively low cost.
- apart from a powerful IDE, Microchip develops and maintains efficient software solutions, such as a full TCP/IP stack, that require minimum efforts to implement in different engineering projects.

The basic idea is to use a Wi-Fi as the wireless unit for the client-side of the system solution. A Wi-Fi requires a host MCU for control signalling and data communications. An LCD together with three LEDs will be used as visual feedback to the users of the client device. These circuits are also interfaced with the MCU and receive control signals from it. In addition, an EEPROM can save programme memory on the MCU if it was to be included in the design. An EEPROM will also receive control and data signals from the MCU. Finally, the memory resources of the MCU are limited and will not support a complex operating system. This should not be a problem since there are other ways to go round this limitation. In the next few sections of this chapter, I will introduce the Microchip technologies that are required for the implementation of this project.

4.2.2 Microchip Hardware Solutions

The project design of the classroom attendance marking can be implemented using Microchip technologies. Among these technologies are a Wi-Fi module and an MCU host. Furthermore, the project design implements some of their other products such as flash memories and programmable, power regulators. Combining these solutions to produce a new system requires a basic understanding of the interfaces of these products and the protocol they use for control and communications. The relevant technologies are introduced next.

The Microchip PIC32MX795F512L MCU

Microchip produces powerful 32 bit MCUs with different models. For example, PIC32MX795F512L model is a 100-pin MCU with a speed of up to 80 MHz, 512 KB of programme memory and 128 KB of RAM. This particular model has a number of extra embedded hardware technologies such as Ethernet and USB interfaces. Programming Microchip MCUs require a programmer device, which is produced and sold by Microchip Technologies Inc. [10,6] Some of the internal blocks of this MCU is shown in figure 12.

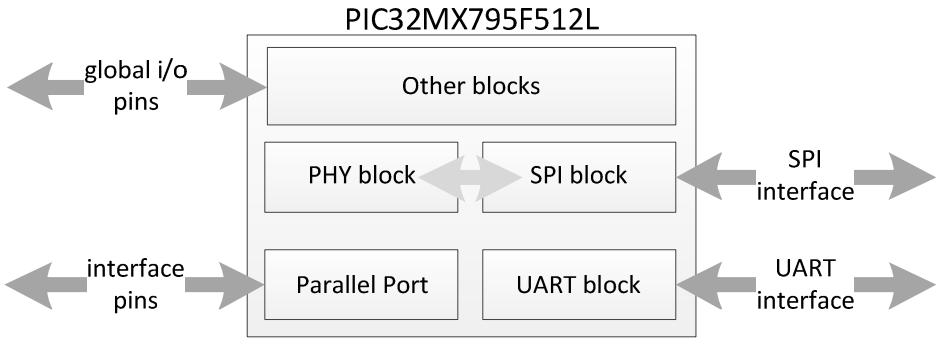


Figure 12. Built-in modules of PIC32MX and their interfaces

Figure 12 depicts a simplified block diagram of Microchip PIC32MX family MCUs. Only four relevant, communication modules are shown in the figure because the Microchip software solutions library and Microchip TCP/IP stack, which will be introduced shortly in section 4.2.3, use these modules extensively. The MCU model has built-in SPI, PHY, UART, and parallel port hardware circuitries. The PHY is internally wired and configured to work with any of the built-in SPI hardware modules. The parallel port is used for

interfacing modules that require multiple signals, usually plus eight bits, such as signals and data bits of LCDs.

Microchip MRF24WB0MA Wi-Fi Module

There are two Wi-Fi modules that Microchip produces and supports, MRF24WB0MA and MRF24WB0MB Wi-Fi radio-frequency transceiver modules. The only difference between the two, MRF24WB0MB has an external antenna connector. MRF24WB0MA Wi-Fi is a full-scale wireless solution that supports, among other things, different encryptions and security technologies. Among the security technologies that this model uses is the Wi-Fi WPA2 with phase-shift keying (PSK), which is developed by Wi-Fi alliance. The module has four control signals for the operation of the device and a 4-line SPI for data communications. [11,7] This is diagrammatically shown in figure 13.

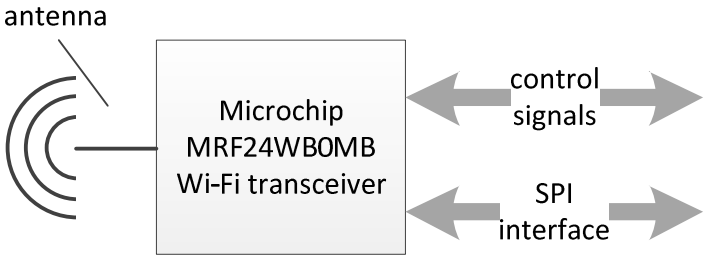


Figure 13. A simple block diagram of Microchip MRF24WB0MA Wi-Fi with SPI and control signals

As illustrated in figure 13, the interface of the Microchip MRF24WB0MA Wi-Fi is simplified to only a few control signals and SPI pins for the flow of data to and from host MCUs.

Microchip 25AA1024 EEPROM

An external, non-volatile memory module can be used for storing information such as configuration data, IPs, and other data that need to be stored and retrieved in the event of a power failure. Microchip produces a few memory models some of which are non-volatile, flash memories known as electrically erasable programmable read-only memory (EEPROM). Among these models is 25AA1024, which has a storage capacity of 1Mb (131,072 bytes) and a serial SPI interface for operation and data storage and

retrieval. This memory module, with a minor software configuration, is supported by the Microchip TCP/IP Stack. [12,7] Microchip TCP/IP Stack is explained next.

4.2.3 Microchip Software Solutions

Microchip develops and maintains three types of software: an IDE, compilers, and software libraries. MPLAB IDE is freeware software for developing applications for and programming Microchip MCUs. The programmer is attached to a PC on which MPLAB IDE is installed. Separate compiler software can be downloaded from the Microchip website and installed to work with MPLAB IDE. The focus of this section is on application solutions rather than IDEs and compilers. The main Microchip application solution is the Microchip TCP/IP stack.

The Microchip TCP/IP Stack

The Microchip TCP/IP Stack is a collection of functions and programmes that enable Microchip MCUs to provide standard TCP/IP services such as HTTP servers, sockets, and mail client. The stack is designed for PIC18, PIC24, and PIC32 of the MCU families that Microchip produces. While detailed knowledge of the implementation of the stack modules is not required in order to utilise it, conceptual knowledge of the stack structure will help understanding the interfaces of these modules. Knowledge of these interfaces is necessary in order to be able to use the stack. Implementation of the stack is based on the TCP/IP reference model. [13,1] Figure 14 depicts the traditional layers of TCP/IP reference model.

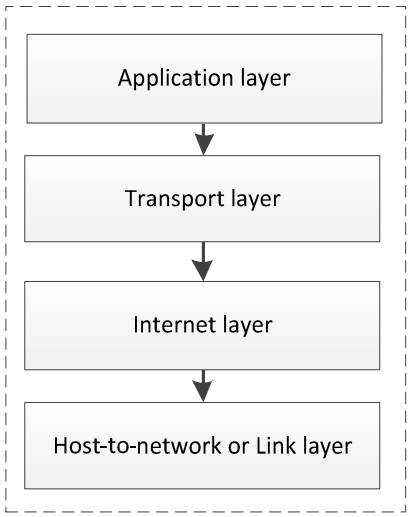


Figure 14. TCP/IP Reference Model

As shown in figure 14, the general TCP/IP reference model is composed of four abstract layers: application, transport, internet, and link layers. This general model is the blueprint for the implementation of the Microchip TCP/IP Stack. Each layer of this model request services directly from layers underneath it. The layers are said to be stacked on top of each other, hence the name TCP/IP stack. Unlike the TCP/IP reference model, the layers of Microchip TCP/IP may access resources and services of one or more layers which may not be immediately below them. This is due to resource limitations of Microchip MCUs. [13,2]

The design of the Microchip stack is such that a super task, called "StackTask", manages the overall operation of the stack. The services of the address resolution protocol (ARP), on the other hand, are also managed by a single task called "ARPTask". These two tasks must be persistently called in the background and they must be independent of the main application. This is usually accomplished within an operating system that can execute two or more tasks concurrently. However, in the absence of such operating systems, and to circumvent this challenge, Microchip follows a technique known as *cooperative multitasking*, which will be explained shortly in the next section. [13,2] Figure 15 shows the main components of the Microchip TCP/IP Stack.

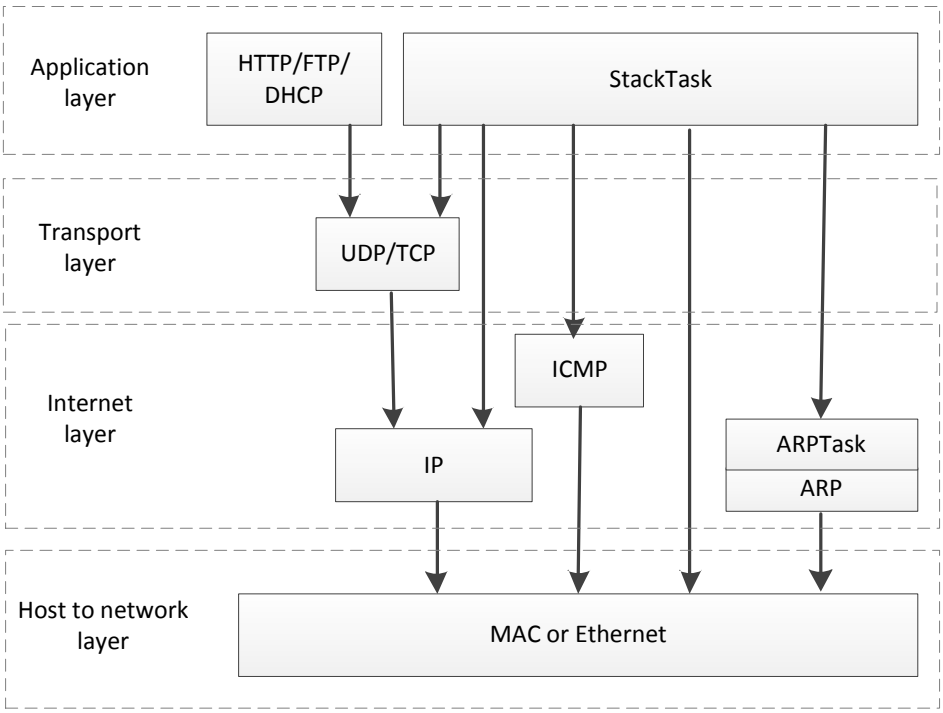


Figure 15. Components of the Microchip Stack [14]

As depicted in figure 15, the implementation of the Microchip TCP/IP Stack is based on the tradition TCP/IP reference model. The application layer is composed of the StackTask and one or more service programmes such as hypertext transfer protocol (HTTP) services and dynamic host configuration protocol (DHCP). The transport layer has two components that belong to the same package. They are TCP and user datagram protocol (UDP). The UDP is not discussed in this paper as it is not utilised by the project. The Internet layer has internet control message protocol (ICMP), which is a service that detects network request errors, an IP for directing packets to destinations, and the ARPTask. Finally, the lowest layer, which is the Ethernet layer, is the actual part that sends and receives packets. Furthermore, the application layer accesses the services of the layers that are both directly and indirectly below it. [13,2]

Cooperative Multitasking

The Microchip TCP/IP Stack is implemented in a cooperative multitasking fashion using C's "switch" statement. In cooperative multitasking, a long task is divided into individual, shorter tasks and spread over the body of a switch statement. In this context, the sub-tasks are executed one at a time in a loop. Each time a sub-task is

executed, the switch mechanism returns to the main application, so that StackTask and ARPTask are not left for long waiting to be executed. The states of the switch statement are changed with the use of C's enumeration. [13,2] Figure 16 illustrates this mechanism.

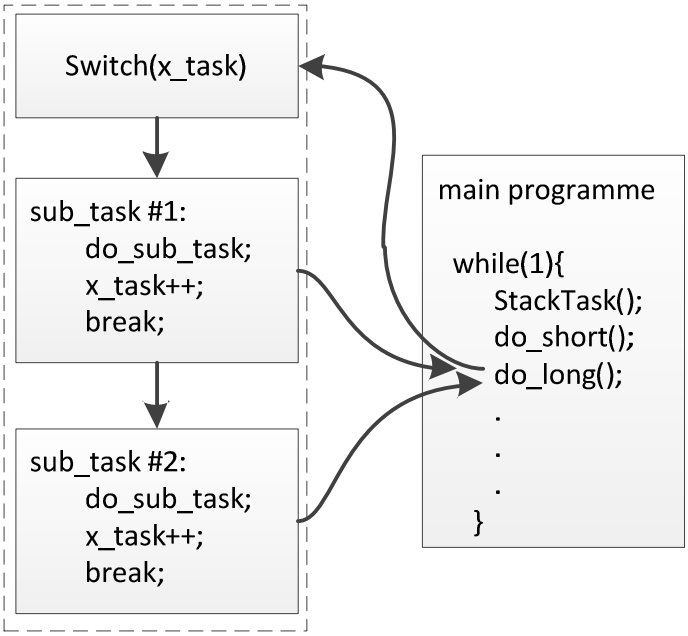


Figure 16. Cooperative multitasking using switch statement

Figure 16 shows a simplified illustration of performing a long task, do_long(), in a multistep fashion. The task is divided into smaller subtasks, sub_task #1 to sub_task #n, and the state of the task execution is preserved in x_task variable. After the execution of each subtask, the switch returns to the main application where an infinite loop calls other time-critical functions and back once again to the switch. This operation continues until all the sub-tasks of the switch are completed. [13,2]

Configuration and usage of the stack

Before the stack can be used, it has to be properly configured. The stack uses the C's pre-processor "define" for configuration. Stack services and functionalities are defined in separate header files. The main protocol service definitions are placed in TCPIPConfig.h header file from which some are commented out. To enable specific services or protocols of the stack, a "define" for that service or protocol is

uncommented. Table 1 includes a few sample configuration defines and their descriptions.

Table 1. Examples of the Microchip TCP/IP Stack configuration defines

#defines	**Description**
STACK_USE_AUTO_IP	Dynamic link-layer IP address automatic configuration protocol
STACK_USE_DHCP_CLIENT	Dynamic Host Configuration Protocol client for obtaining IP address and other parameters
STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE	HTTP Client example in GenericTCPClient.c
STACK_USE_DNS	Domain Name Service Client for resolving hostname strings to IP addresses

Table 1 shows samples of stack configuration entries. For example, to allow for dynamic Ethernet layer IP address automatic configuration, the STACK_USE_AUTO_IP is made available by uncommenting the line that contains this capitalised name. Furthermore, a set of required files is needed to be included in a project design that makes use of the stack. [14] These files are described in table 2.

Table 2. The Microchip TCP/IP Stack project files [14]

Entities	**Description**
Main file	this is the main application file
ARP.c and ARP.h	address resolution protocol file used to resolve media access control (MAC) addresses
Delay.c and Delay.h	provide required time delays of the stack components
Helpers.c and Helpers.h	provide necessary functions that are used by the stack modules
IP.c and IP.h	contains functions that define IP protocols and other internet layer functions
StackTsk.c and StackTsk.h	this a super task that manages the various calls and components of the stack to keep it running
Tick.c and Tick.h	contains timing functions used for time related operations.
HardwareProfile.h	contains hardware configuration defines
TCPIPConfig.h	provides defines and other methods for stack software configuration
MAC.h	contains data structures and macros related to MAC address layer
TCPIP.h	this is the main header file of the stack. This file should be included in the main application file

The content of table 2 is a bare minimum that must be included in any project that uses the Microchip TCP/IP Stack. In addition, the stack provides several initialisation

functions that make use of the files in table 2. Examples of initialisation functions are TickInit(), StackInit(), and hardware initialisation functions that can be given any names. Although there can be different ways to set up a Microchip TCP/IP based project, a common, general structure must be followed. For example, the stack must first be initialised before its components can be used. [14] This is shown in figure 17.

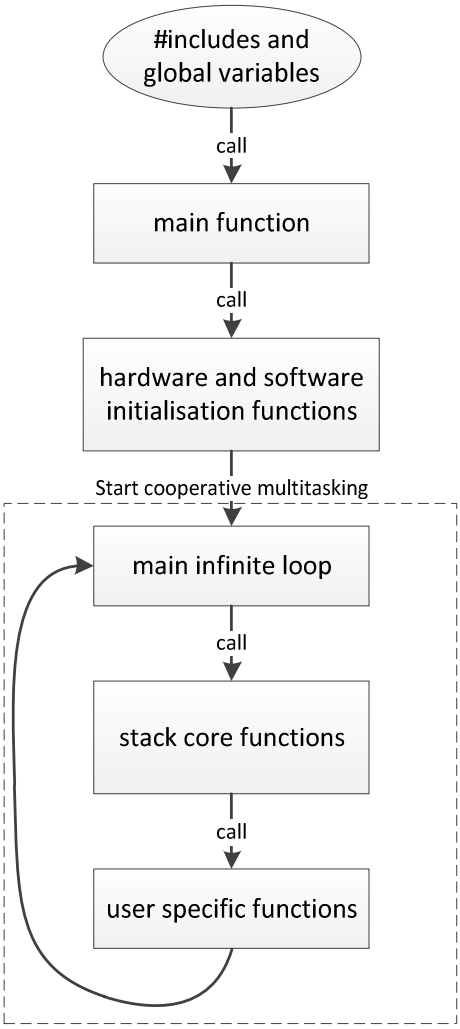


Figure 17. General structure of a Microchip TCP/IP Stack application

Figure 17 describes a general structure of a project that utilises the Microchip TCP/IP Stack. A hardware initialisation must first be performed to set up the included hardware modules such as displays, wireless modules, and hardware that needed for various communications such as UART and SPI.

5 System Design

5.1 Engineering Method

The implementation of this project followed general engineering methodologies from abstract designs to final concrete solutions. Microchip offers a wide variety of solutions to different engineering challenges. Combining two or more Microchip solutions can result in new, useful systems. The classroom challenge is one such problem that can be divided into sub-problems and tackled separately with the existing Microchip solutions. An engineering design of a system should begin with laying out relevant concepts. A conceptual design will ensure the inclusion of all the necessary components of the system as a whole and prepare the ground for the next stages of the design.

Before concepts can be turned into functional units, some feasibility evaluations are in order. After defining the system conceptually and drawing necessary relationships between the concepts, only then engineers can further define the concepts and transform them into block diagrams and the relationships into protocols. Naturally, what follows is a thorough research about hardware and software requirements. This includes different kinds of tools and development environments. Block diagrams are later transformed into functional units, both of hardware and software. Preliminary tests, then, can be conducted for each separate unit. Finally the units are combined to work as a system and extensive tests are conducted on the system as a whole.

5.2 Conceptual Design

5.2.1 Conceptual Design of the System Solution

The system is composed of two major parts, a single server and the portable devices or clients. Since the requirement is to build portable clients, the system should have no cables between the server and the clients. The server and the clients will have to be wireless. Figure 18 outlines the concepts that are involved in the designing of the system.

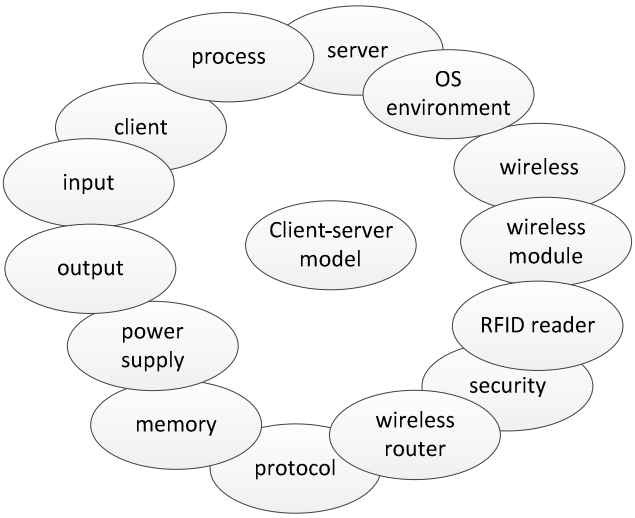


Figure 18. The initial concept map with a central concept

As can be seen from figure 18, the major concepts that surround a solution based on a client-server model are relevant, general concepts. In this context, the client-server model is the *central concept*, also known as the *domain*, from which other major concepts are driven. From these major concepts, at a later stage, sub-concepts are derived to further define the system. Figure 19 shows the domain and its sub-concepts that are linked with proper relationships. Note that the two figures of 18 and 19 are not similar in that the latter has defined relationships.

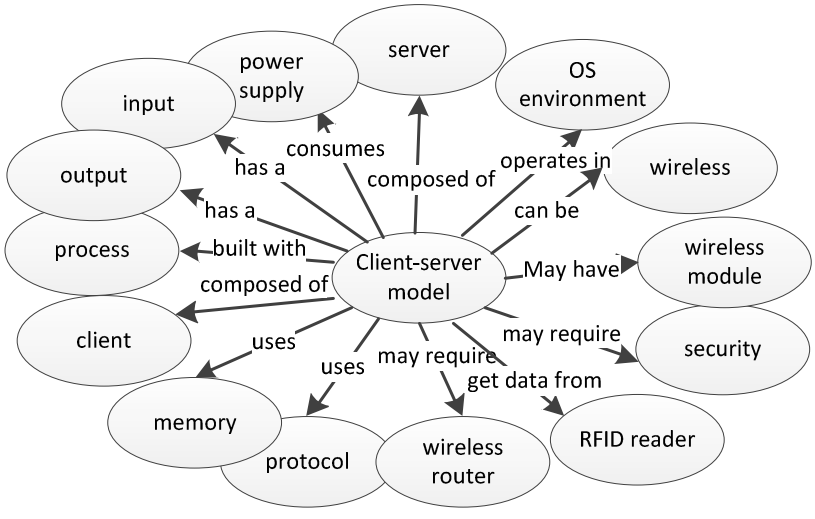


Figure 19. The concept map with relationships

Figure 19 shows the concept map of the system design with basic relationships between the domain and the derived concepts. The next step is to separate and group

related concepts together to form foundations for units. This is a necessary step in order to define the system design in terms of units. Figure 20 depicts this step.

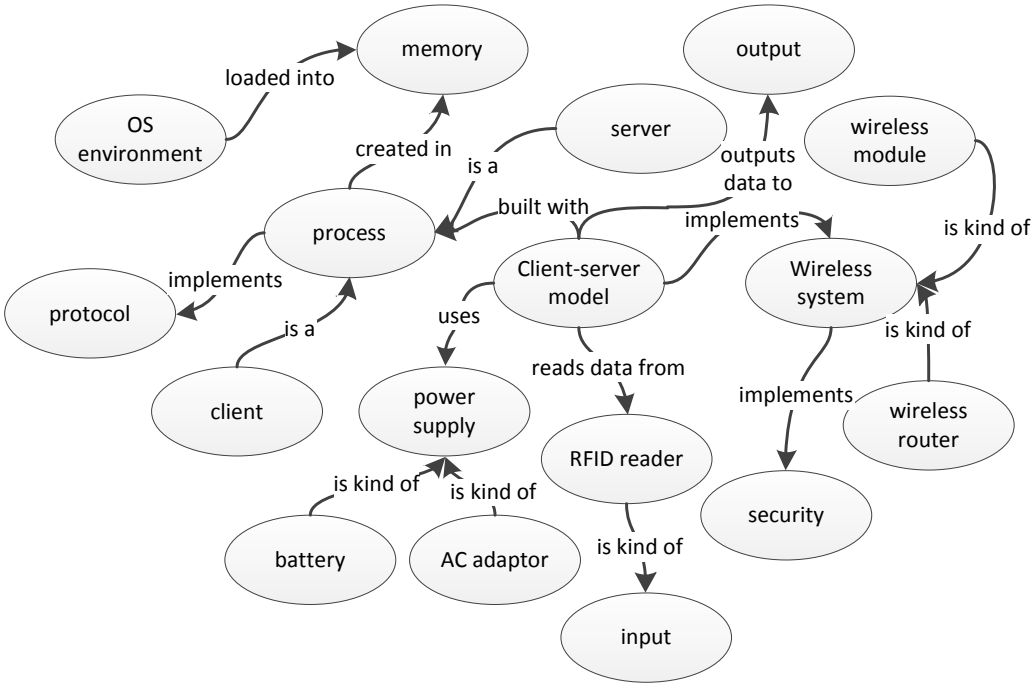


Figure 20. The concept map after further refinement

As illustrated in figure 20, the concept map became clearer when the concepts have been grouped. Grouping of the concepts will ultimately result in functional units of the system.

5.2.2 Conceptual Design of the Server Side

While the server and its clients have similar functionalities, the server differs from the clients in several ways. The main difference is the concept of concurrency. The server of a client-server model must be concurrent. Another important point is that the server is not required to be portable since it is not moved too often. However, the server is required to be wireless since the clients are portable, wireless units. A fixed server can run on a desktop or laptop PC which in turn connected to a wireless router, rather than a wireless module. A wireless module requires a host and may not support multiple connections since they are made for clients. Figure 21 shows the concepts that are involved in the design of the server.

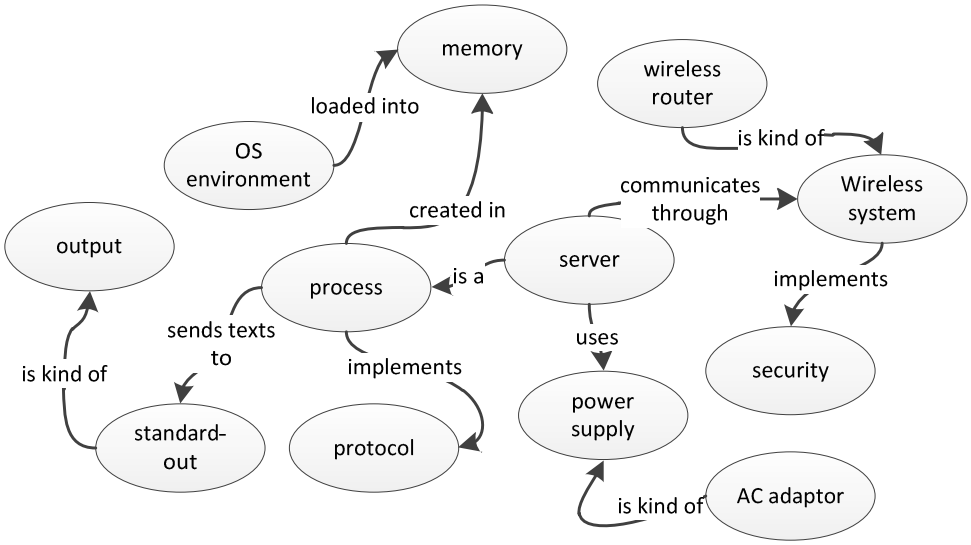


Figure 21. The concept map of the server

As can be noted from figure 21, some concepts have been eliminated of the concept map of the server. Also, the domain or the central concept now became a server.

5.2.3 Conceptual Design of the Client Side

One of major differences between a client and the server is that clients need not to support multiple connections since they do not require communicating with each other. Furthermore, one or more output feedback to the users will enhance usability of these devices. In this case, data such as IPs and the identification numbers that have been read from the RFID reader will be shown on an LCD. In addition, one or more LEDs to indicate activities when connected to the server will be added to the portable clients. A simplified concept map of a client is given in figure 22.

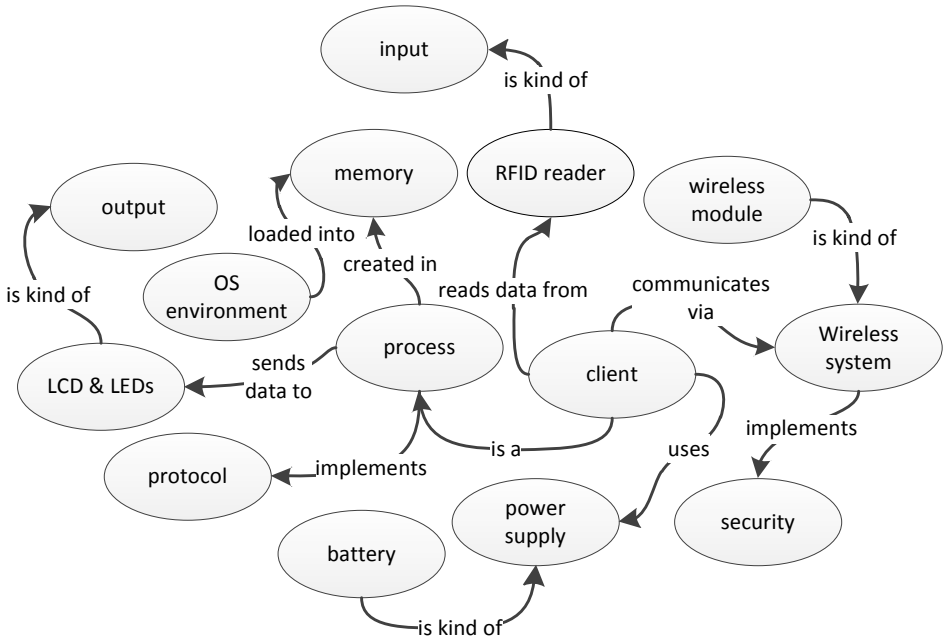


Figure 22. The concept map of a client

As illustrated in figure 22, the concept map of a client is not fundamentally different than that of the server. Protocols, for example, will have to be similar in order for a client to be able to communicate with a server.

5.3 Feasibility Evaluation

5.3.1 Wireless Systems Compatibility

As wireless systems use security technologies, a compatibility check is in order between Microchip MRF24WB0MA Wi-Fi module and wireless specifications of a standard wireless router. The wireless router is going to be part of the server system. Based on its datasheet, the MRF24WB0MA Wi-Fi module supports the following security technologies:

- Hardware Carrier sense multiple access with collision avoidance (CSMA/CA) and automatic ACK
- Automatic MAC packet retransmit
- Hardware security engine for advanced encryption standard (AES) and RC4-based ciphers

- Supports 802.1x, 802.1i security: WEP, WPA-PSK, and WPA-2-PSK [11,3].

Furthermore, the Microchip TCP/IP Stack supports 802.11x wireless standards which include AES [14]. A typical wireless router supports most security technologies including WPA2. This is sufficient information to confirm wireless security compatibilities between the systems.

5.3.2 Socket Types Compatibility

Another important consideration is the type of sockets that both the server and the clients use for IPC. The Microchip TCP/IP Stack supports various types of standard sockets. Table 3 outlines some of these sockets. [14]

Table 3. Socket types supported by the Microchip TCP/IP Stack [14]

Socket types
TCP PURPOSE GENERIC TCP CLIENT
TCP PURPOSE GENERIC TCP SERVER
TCP PURPOSE UART2 TCP BRIDGE
TCP PURPOSE HTTP SERVER
TCP PURPOSE DEFAULT
TCP PURPOSE BERKELEY SERVER
TCP PURPOSE BERKELEY CLIENT

As can be seen from table 3, the Microchip TCP/IP Stack supports all the relevant types of sockets that can be used in this project. For example, TCP Berkeley socket (BSD) is the original version of POSIX sockets. However, the generic TCP sockets are most relevant to this project since they are fully compatible with POSIX standards.

5.4 Preliminary Design

5.4.1 General Block Diagram

The system design at the level of block diagrams can now be drawn from the concept maps. Before introducing block diagrams of the functional units of the system, a general sketch of the system as a whole will reinforce the design and its general topology. The sketch is similar to that of a client-server model. Since the server is not required to be portable, a PC, a desktop or a laptop, can be used as its hardware, and a Linux operating system as its software. Then a concurrent server can be designed within the Linux operating system. However, a wireless router is still required since the topology forms a WAN, rather than a single LAN. Figure 23 depicts an overall topology of the system design which represents a wireless client-server model.

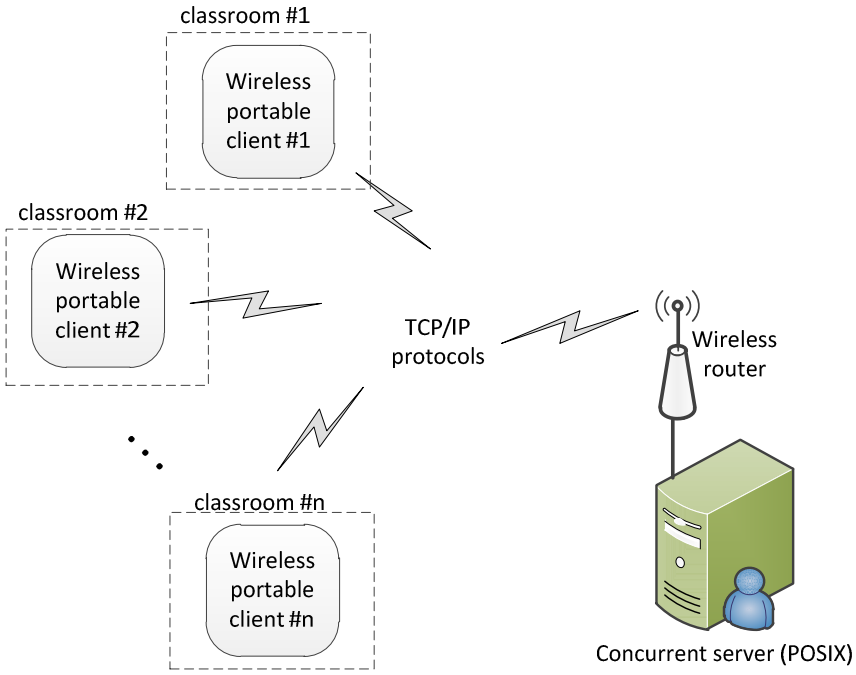


Figure 23. A general topology of the system design

As can be seen from figure 23, a PC, on which a concurrent server is running, is connected to a wireless router to which one or more clients can connect and send classroom data to the server.

5.4.2 Software Block Diagram

A software block diagram paves the way for a more detailed diagram at the later stages of the design. A software block diagram of the server side of the system is relatively easier than that of the clients since most of algorithms and software complexity is hidden in the Linux operating system. Figure 24 shows this diagram.

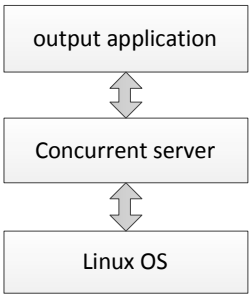


Figure 24. A general software block diagram of the system server

Figure 24 illustrates an overall software block diagram of the server of the system. While the top block is not within the scope of this paper, the basic text output will have to be shown for test purposes. Such texts, however, can be displayed using basic text editors. For further sub-blocks of the middle block, namely the concurrent server, please refer to figure 11.

Since the hardware of the client side of the system solution does not support complex operating systems, such as Linux-GNU, it has a relatively more complex software block diagram. A client will have to continually run the necessary components of the Microchip TCP/IP Stack in the background, scan input data ports to which the RFID reader is interfaced with, display IP and other feedback information on an LCD, and run other user applications, such as socket IPC and processing of data. Figure 25 shows a general software block diagram of a client.

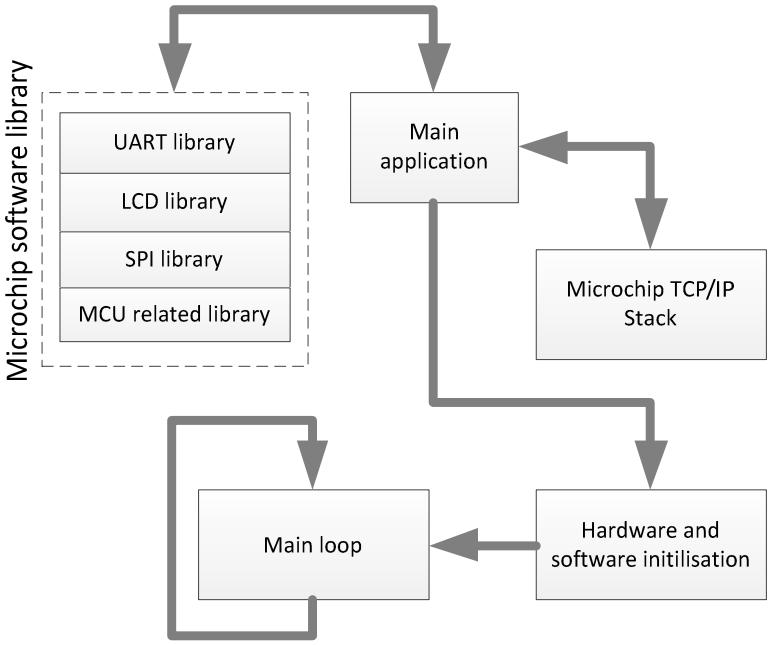


Figure 25. General software block diagram of a client

As shown in figure 25, the two main blocks of the diagram are the Microchip TCP/IP stack and the Microchip software library that is composed of different software modules and protocols for managing and programming Microchip products. The inner blocks of the main loop is shown in figure 17.

5.4.3 Hardware Block Diagram

The hardware of the server-side is basically composed of a laptop and a wireless router. However, the clients need to be designed and built from a combination of existing solutions. An LCD will be included in the hardware prototype for debugging purposes. The display can also be a permanent part of the device if tests showed that it would not consume considerable power. It is a wise design decision to include an external EEPROM, so that the extra internal memory of the MCU will be preserved for user applications. Figure 26 shows a general hardware block diagram of the client side of the system.

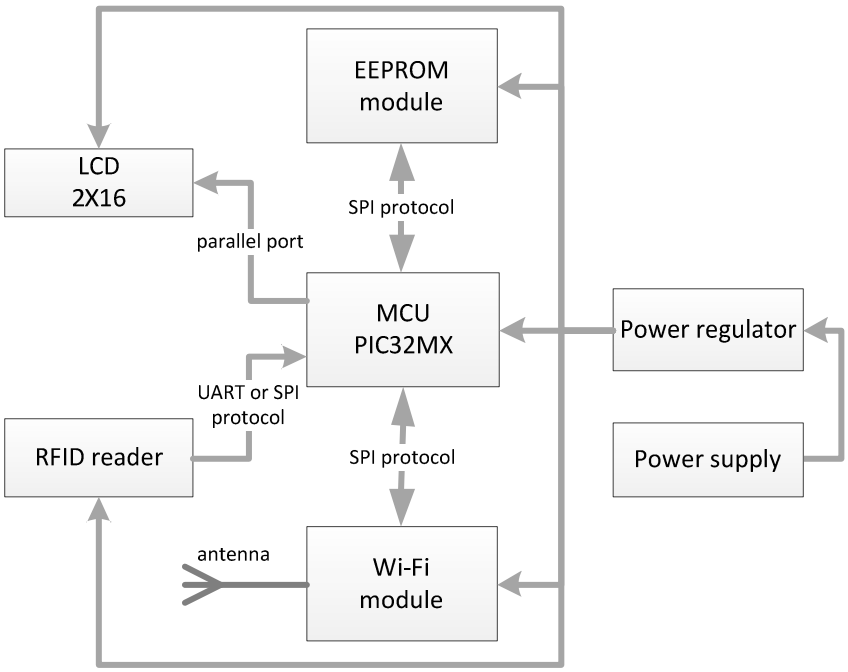


Figure 26. Hardware block diagram of the client-side of the system

As can be seen from figure 26, the hardware part of the client of the system is composed of existing solutions. Except for the LCD and RFID modules, the rest of the hardware, including the power regulator, is produced by Microchip. Also shown in the figure are various protocols for managing the communications between the modules.

5.5 Design Requirements

5.5.1 Hardware Requirements

There are two main groups of hardware that may be required in any embedded project. They are hardware functional components and hardware tools. The conceptual and block diagrams of the system define the hardware components and tools that are needed for the implementation of the system. A general list of required hardware components is shown in table 4. The list has been generated from an extensive research through various brands and models.

Table 4. A general list of hardware components for implementation stage

Hardware components	**Selected module or component**
Host MCU	Microchip PIC32MX795F512L
Wireless module	Microchip MRF24WB0MA Wi-Fi
An antenna for the wireless module	A unidirectional antenna from an old router
EEPROM module	Microchip 25AA1024 EEPROM
LCD module	A standard LCD 2X16 characters
Voltage regulator	Microchip TC1262; this is decided by the input voltages of the MCU and Wi-Fi module
Other passive and active electronic components	Various brands of decoupling capacitors and current limiting resisters, etc.
Cables and connectors	As required from old computer parts
AC power adaptors	For the server and a 5v output for a prototype client during testing stages only
Laptop with Ethernet interface	Any brand with Ethernet or wireless module
A wireless router	Any brand that supports AES technologies

Table 4 contains a list of hardware components and modules that are required for the design and implementation of the system solution. Some components in table 4 have already been fully identified as a result of an extensive research. For example, the model of the host MCU that is going to be part of the portable clients is a 100-pin, 32-bit microcontroller from Microchip. Port pins are an important factor when it comes to embedded systems. The host MCU must have enough pins to support the hardware modules in the block diagram of figure 26.

Apart from hardware components and modules, the design and implementation of the system solution will also require various hardware tools. A 100-pin MCU not only demand a special soldering system but also have to be mounted on printed circuit boards (PCB). A PCB milling machine, therefore, is also needed for the hardware part of the client devices. Fortunately, the Helsinki Metropolia University of Applied Sciences provides both of these tools.

5.5.2 Software Requirements

The design and implementation of the project require various software tools and solutions. Among the software tools are PADS of Mentor Graphics and Microchip MPLAB IDE for Windows environment. Mentor PADS is a specialised tool for designing PCBs. Microchip MPLAB, on the other hand, is for building and programming projects for Microchip's programmable products such as various models of Microchip MCUs.

Other software requirements are the Microchip TCP/IP Stack and Microchip peripheral library, which is a collection of code, written in C language, for the operation of various Microchip modules and various standard communication protocols such as SPI and UART. Finally, a Linux operating system and a GCC compiler are required for building and designing the concurrent server. Table 5 outlines these tools and solutions.

Table 5. The required software tools and solutions

Software	**Description**
MPLAB IDE v8.56	Used for building projects and programming Microchip MCUs. For 32-bit MCUs, a C32 compiler should also be installed and integrated into the IDE
PADS PCB Tools	A set of tools: PADS Logic, PADS Layout, and PADS Router. PADS logic is for designing schematics, PADS layout is for PCB design, and PADS router is used for automatic routing
Microchip TCP/IP Stack	A stack of modules for TCP/IP network communications
Microchip Peripheral Library	A collection of reusable, library code solutions, designed for Microchip products, that can be included in Microchip projects
Linux & Windows	Two operating system platform that other tools and programs can run on
GCC Compiler	For compiling the design code of the server on Linux platform

The content of table 5 describes the software tools and Microchip software solutions that are required for the design and implementation of the classroom attendance confirmation system. The three essential PADS tools of Mentor Graphics are PADS Logic, PADS Layout, and PADS Router. They are used for designing the system hardware schematics and the necessary files for milling a PCB for the client prototype boards.

5.6 Detailed Design

5.6.1 Hardware Detailed Design

The hardware part of the server is a laptop and a wireless router. No extra hardware unit of the server needs to be designed or built. However, a client prototype board with a host MCU, a wireless module, and an EEPROM module must be designed and built. There are five system units that make up the client prototype device. Each unit

has its own separate circuit that can be joined with other units through one or more interfaces. The circuits are: MCU host, Wi-Fi, EEPROM, LCD, LED, and voltage regulator circuits. A detailed design of each circuit is introduced in the next paragraph.

The selected MCU host is PIC32MX795F512L. There are several basic external components that are required for the operation of this MCU model. These components are:

- external decoupling capacitors
- external capacitors on the MCU's internal voltage regulator
- an external crystal oscillator
- in-circuit serial programming (ICSP) interface
- power and interface connectors. [10,42]

Most of the values and types of these components are directly adopted from the recommendations that are given in the datasheet of the MCU. The overall circuit diagram of the MCU host is based on the recommended minimum connection from its datasheet. The value of the decoupling capacitors is 0.1uF (100nF) of ceramic type [10,41-42]. The internal voltage regulator of the MCU requires a capacitor, which is connected to Vcap/Vcore pin, is of a typical value of 10uF [10,171]. The MCU requires an external crystal oscillator to provide clock signals to the device. Based on the MCU's datasheet, the device supports various speeds of external oscillators from which 8MHz is typically used. The oscillator circuit is connected to OSC1 and OSC2 pins of the MCU [10,121]. Figure 27 shows these components connected to the MCU.

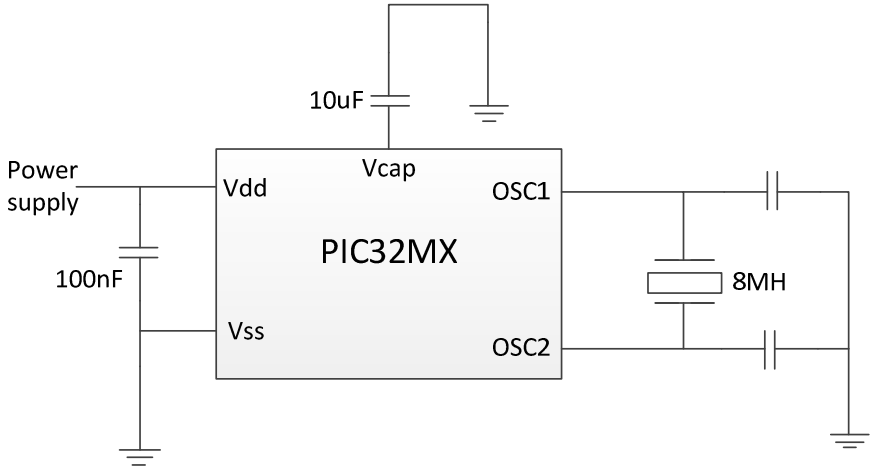


Figure 27. PIC32MX with decoupling Vcap capacitors together with an 8MHz crystal oscillator

As can be seen from figure 27, a 10uF is required for the internal voltage regulator of the MCU. An 8MHz crystal is connected to the oscillator pins and a decoupling capacitor is recommended for each Vdd pin of the MCU.

Programming the MCU requires a programmer. Microchip PICkit3 device is one such programmer that can be used for programming most of the Microchip MCU models. The programmer has six pins from which three are clock, data, and memory clear (MCLR) pins. A passive, pull-up resister of a value between (4.7-10) kΩ is recommended to be connected from the MCLR pin to the positive power supply Vdd. These values, and a general header interfacing sketch, are provided in the programmer's user guide. [15,20] Figure 28 depicts a general diagram of the pins of the programming header.

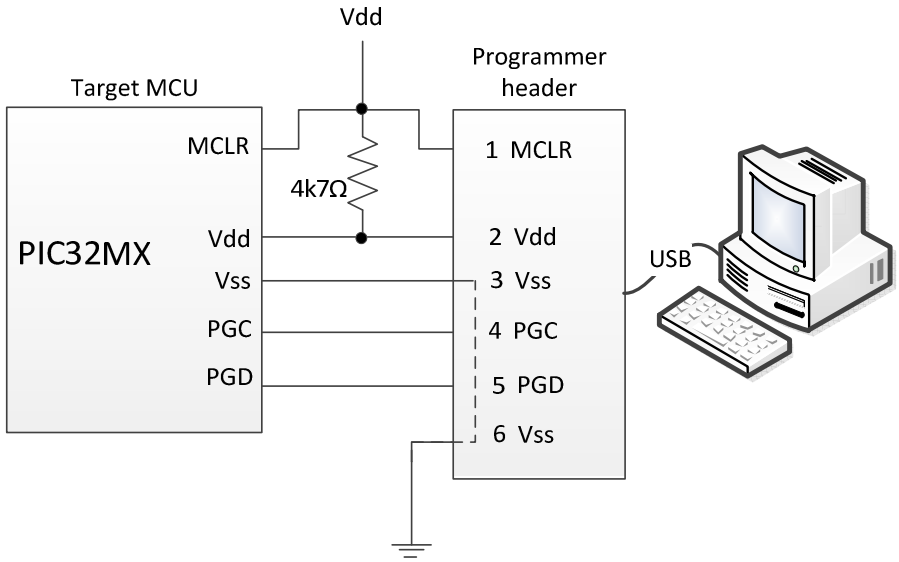


Figure 28. Programming header connected to a target MCU [15,20]

Figure 28 shows the Microchip PICkit3 programmer header connected to the ICSP pins of a Microchip MCU. The programmer has a USB interface through which a PC with a Microchip MPLAB IDE sends programming data to the target MCU.

Since the Wi-Fi module, the EEPROM, and the LCD are interfaced with the MCU, relevant pins have to be routed with proper connectors. Apart from power supply pins, there are seven control and communication pins routed out from the Wi-Fi module. Furthermore, Microchip PIC32MX family models have built-in hardware blocks for peripheral communications, such as SPI and UART. These hardware blocks are internally connected to particular pins, which are properly marked with letters and numbers. Some MCU models may have two or more units of the same technology in the same device. Table 6 shows the reserved pins for each external unit of the system design.

Table 6. Reserved pins for the external circuits

Unit circuits	**Reserved pins of the MCU**
Wi-Fi	SPI1 pins
LCD	Port E for data bits and pins RD4, RD5, RB15 for control signals E, R/W, and RS respectively
EEPROM	SPI2 pins
LED	RD8, RD1, and RD2 for LED1, LED2, and LED3 respectively
RFID reader	UART2 pins

Table 6 contains the necessary reserved ports and pins of the PIC32MX MCU for the external circuitries. Full details of the schematics are given in appendix 1.

Although the Microchip MRF24WB0MA Wi-Fi is already a complete circuit with a proper interface and control pins, some basic decoupling capacitors, together with SPI and control signal connectors, are required to be soldered to the module. The values of these components are recommended in its datasheet. The EEPROM module, on the other hand, except for a decoupling capacitor, does not require any other external components and should be mounted as close as possible to the main MCU. Other circuits of the system are the LCD and the LED circuits. The LCD circuit, apart from an 8-pin connector for data signals, requires a voltage divider for contrast tuning. The LEDs, on the other hand, require current limiting resistors only. [11,13] Full details of the schematics are given in appendix 1.

The hardware schematics are first designed with PADS Logic software tool of Mentor Graphics. The result of these schematics can be seen in Appendix 1. Then the schematics are sent to PADS Layout software tool with which PCB layouts are produced. Appendix 2 shows the PCB layouts of the top and bottom layers of the board. From the PCB layouts, then, special files are generated called CAM files. CAM files are used for final milling of the board. The board is then cleaned and prepared for mounting and soldering the modules and the components to produce the final prototype board. Appendix 3 is a photograph of the final board.

5.6.2 Software Detailed Design

The server

The server programme starts with opening a general TCP/IP socket for the server using socket() API function. This function requires three arguments in order to be able to open such socket. The arguments are the protocol family, which is also known as the domain, the type of the socket, and the type of protocol respectively. Each argument is represented with a constant. Since the design of this system is based on IPv4, the first argument is a constant named AF_INET. A SOCK_STREAM constant goes into the second argument and a zero for the last argument to represent a TCP

protocol. This function returns a file descriptor integer that will be used in later stages of the server programme.

After a socket has been opened, it is bound with a proper port using the bind() function. The bind() function requires three arguments. The server socket descriptor, an address structure of the type sockaddr_in, which is for IPv4, and finally the numerical size of the structure goes into the last argument. The address structure has to be first populated with proper values shown in appendix 4. Following bind(), the state of the socket is then changed to listening state by calling the listen() function. This function needs only two arguments, the socket descriptor and the number of clients the server should support. The number of clients in this case depends on the number of classrooms of the educational organisation that the project is designed for. Only five clients are chosen in this case.

Now the server should enter an infinite loop and in each round call the accept() function and waits for connection requests from clients. When a client requests for a connection, the accept() function returns a dedicated socket descriptor for that client. The server then calls the fork() function to initiate a dedicated child process for the client and pass the descriptor to it. Finally, the server loops back to the accept() function and waits for requests from other clients. The accept() function takes three arguments, the server socket, a pointer to the client data structure of the type sockaddr_in, this is where the source address of the client is stored for communication, and finally the size of the client data structure as the last argument. The fork does not require any arguments; it only returns a pid for the child it creates.

The fork() function clones a child process out of the parent server process. The child process is still part of the server; only, it is now dedicated to a specific client. The main content of the child server now should handle the requests from the client through the client socket file descriptor. The child server first closes the parent server socket, since it has its own socket descriptor for communications. After closing the parent server socket, the child server enters its own super loop and handles the requests from the client through send() and recv() functions. These functions require client socket descriptors, buffers for data that is sent and received, and the size of these buffers as

their third arguments. Within the loop, a conditional statement checks if a message from a client was a quit signal. Figure 29 shows a flow chart of these steps.

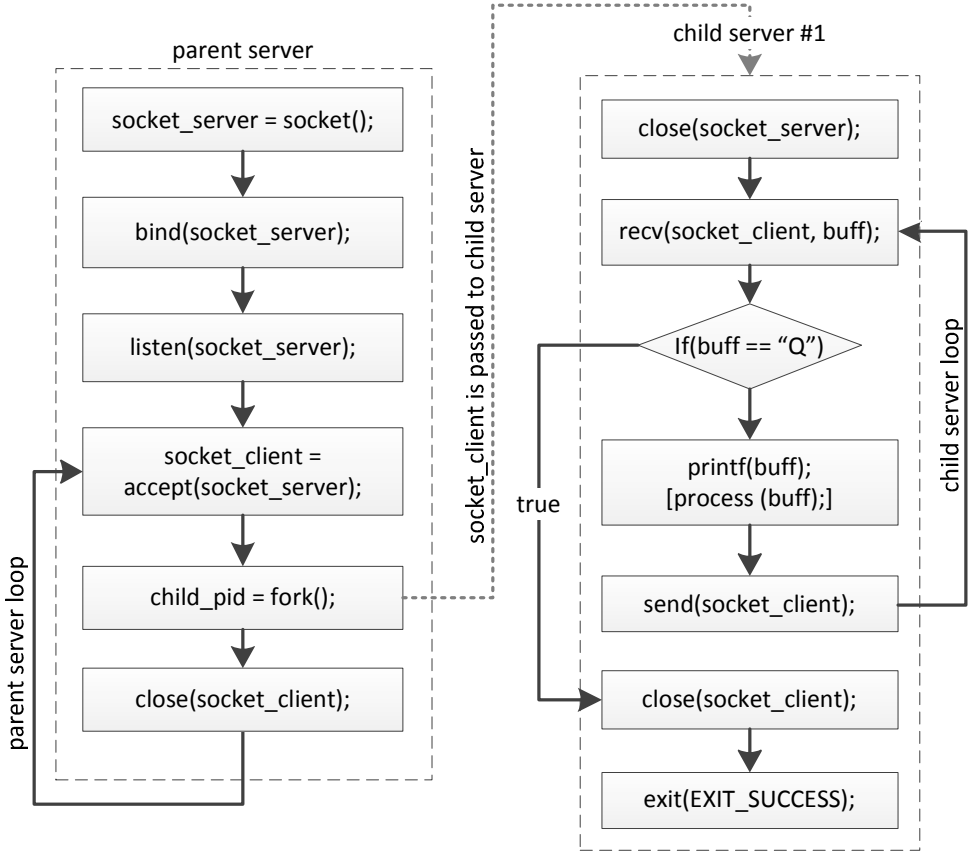


Figure 29. A flowchart of the server programme with a child

As shown in figure 29, the parent process of the server prepares a socket, binds it with a proper port number, changes its state to a listening socket, calls accept to create a socket descriptor for a requesting client, and finally forks a child server with the new socket descriptor. The parent server, then, closes this socket and starts all over again and waits for another client. In addition, a conditional statement within the loop of the child server checks for a quit message. In this case the quit signal is simply a letter "Q", however, in the presence of an RFID reader this can be the identification number of the teacher's RFID key. For simplicity, received data will not be processed but only displayed with printf() function. Processing of data is left for other projects and systems that may be designed in the future. A complete code, written in C language, is given in appendix 4.

The client

The software of the client device is rather complex. All parts of the client programme code are configured and built in the Microchip MPLAB IDE. Microchip provides several demo projects that are built for specific Microchip development boards. Reusing these demo projects for different development boards requires strenuous changes and configurations to these demos. To minimise developing time and efforts, a dedicated project needs to be set up in MPLAB IDE. Proper configurations, such as MCU model and compiler selections, then, require minimum time and efforts. Now the Microchip TCP/IP Stack can easily be included in the MPLAB IDE and properly configured for the client device. Functions and ideas can still be borrowed from the demo projects that are provided by Microchip. The steps that are required for programming a client device are outlined below:

- start a new project in MPLAB using the project wizard
- select PIC32MX795F512L MCU
- select C32 compiler, which is a 32-bit compiler for PIC32MX family models
- load the latest version of the Microchip TCP/IP Stack into the project
- borrow codes from the demo projects to be reused in the MPLAB project
- make necessary changes in the configuration files
- write the main application with tasks divided in a cooperative multitasking fashion
- programme the device using the PICKit3 programmer
- test, adjust, and re-programme until desired results have been attained.

There are two major groups of configuration files. Hardware configuration files inform the MPLAB IDE of what to enable of the MCU's built-in hardware modules, and the reserved port pins for external modules. The Microchip TCP/IP stack also requires software configurations such as what protocols should be enabled and what security should be implemented. These files are header files with an extension of "h". The hardware configuration file should, among other things, properly assign the MCU ports and pins to the mounted external hardware modules. Other information in this file that should be defined is related to system frequency and internal registers. A completely configured file named HardwareProfile.h is shown in appendix 5.

The Microchip TCP/IP Stack requires two configuration files, one for the TCP/IP stack and the other for the wireless module. They are TCPIPConfig.h and WF_Config.h. The TCPIPConfig.h contains settings related to the TCP/IP stack. The settings are, among other things, protocol types, such as TCP, settings related to software modules, such as HTTP and mail server, and default IP and MAC addresses. The best way to configure this file is by borrowing an existing file from Microchip demo projects and enable or disable the parts that are related to the client device. The main options that have to be included are STACK_USE_AUTO_IP, which enables automatic IP configuration, and STACK_USE_DHCP_CLIENT, which enables DHCP. Appendix 6 shows the content of TCPIPConfig.h configuration file.

The last software configuration file is WF_Config.h. This file contains important settings related to types and securities of wireless connections. Apart from settings that are connected to the internal registers of the Microchip Wi-Fi model, which is out of the focus of this paper, the relevant settings are selected based on general settings of a typical wireless router. Table 7 shows the main settings in this file.

Table 7. main settings in WF_Config.h

Setting	**Description**
SSID	The name of the service set identifier. This name is broadcast by a wireless router. This is set to “metropolia”
Network type	Set to “infrastructure”
Scan type	Set to active
Channels	All typical channels: 1, 6, and 11
Security type	Set to WPA auto with passphrase
PSK phrase	Pre-shared key phrase set to “metropolia”

Table 7 contains the main settings of a wireless connection in the WF_Config.h file. The security is set to WPA-auto with a passphrase. The “auto” means the security can support both versions of WPA and WPA2. Concerning the passphrase, to minimise calculations by the MCU for PSK generation from the phrase, Microchip stack also provides another setting in the same file where a generated 256-bit PSK can be placed in a hexadecimal form. To convert the phrase to a PSK, Wireshark website provides an online converter [16]. After the phrase conversion, the raw code can then be copy-pasted into the file where this setting is located. The content of WF_Config.h is shown in appendix 7.

Now that basic configuration is in place, the main application can start. The main application is simply comprised of initialisations and an infinite loop in which several tasks are continually executed. InitializeBoard() is the first initialisation function that simply sets the system bus speed, enables the programme cache, enables the modules that are connected to the prototype board by turning their chip select (CS) on, and making sure that the LEDs are off at the start. The rest of initialisation functions are LCDIinit(), TickInit(), initAppConfig(), and StackInit() respectively. These functions are part of the Microchip TCP/IP Stack and do not require any modifications. To establish a wireless connection, WF_Connect() function is called. The function scans and finds the service set identifier (SSID) from within the range of a wireless router and establishes a connection. Before this function can perform its tasks, however, a wireless router, preconfigured with the same configuration in table 7, should already be broadcasting its SSID.

The StackTask() now should constantly be called every few milliseconds. This function and the rest of the main programme must be called in a loop within which user tasks must follow cooperative multitasking method described earlier. For that reason, and since there are no operating systems running in the background, any other user tasks must be divided into small portions so that StackTask() will not be left un-called for too long. The next function call is StackApplications(), which is part of the stack that must be called after the StackTask(). This function sets, among other things, a default IP address and stores it on the EEPROM module. StackApplications() function contains most of the internal, stack related tasks of networking and security, that is unnecessary to go through them in this paper.

At this stage of the software programme, the client system is connected to the wireless router and has received a new IP. Now it is time to introduce the user application. For simplicity and ease of maintainability at later times, the tasks of the user application are defined within a separate function namely GenericTCPClient(). In fact, the template of this function is borrowed from the microchip demo projects but only reused with different contents. The details of the content of this function are explained in the next paragraph. What follows the call of this function, however, is checking for changes in

the IP address and jumping back to the beginning of the loop to call the StackTask() again.

The main tasks of GenericTCPClient() is to open and prepare a TCP socket for data communications with the server. The tasks within this function, as said earlier, are divided into sub-tasks using C's switch statement. Scanning for input data should also be performed within this function. In this case, scanning the port that the RFID reader is connected to should be constantly checked for new input data. However, since an RFID reader is unavailable to this project, data is generated instead within the function itself. The generated data is sent to the server in 10 iterations, which reflects 10 student IDs.

Generic sockets are prepared with a set of API functions that are part of the Microchip TCP/IP stack. The pieces of information that are needed to be used in these functions are: the address and port number of the remote server, which are "192.168.0.11" and 6543 in this case, and buffers for data manipulations. Table 8 contains the functions that are needed for opening and managing a generic socket.

Table 8. The Microchip TCP/IP API functions for creating and utilising generic TCP sockets

API name	Description
TCPOpen();	Opens a generic socket (client or server). It takes four arguments: the address of remote host in sting form (client), a flag telling of the type of the socket, the remote host port, and a constant indicating the purpose of the socket. This function returns a socket handler if no error.
TCPIsConnected();	Checks the state of an open socket for connection. It takes a socket handler as its only argument. It returns true if the socket was connected.
TCPIsPutReady();	Returns the number of bytes that can be written to the socket that a handler represent as the argument of this function.
TCPIsGetReady();	Same as TCPIsPutReady() only for reading.
TCPPutROMString();	Writes a string from ROM to a socket that a handler represents as the argument of this function. Data is automatically transmitted if the socket buffer was half full or a definable time has elapsed. It takes two arguments: a socket handler and a buffer.
TCPFlush();	Transmits all pending data in the socket buffer. It takes a socket handler as an argument.
TCPGetArray();	Reads and removes an array of data bytes form the received buffer of a socket. It takes 3 arguments: a socket handler, a buffer in to which data is read to, and the number of bytes to be read.
TCPDisconnect();	Disconnects the socket that its handler is given to it as its only argument.

Table 8 shows the functions that have been used in the implementation of the tasks of GenericTCPClient() function. Naturally, the first sub-task in this function is to call TCPOpen() and use its return value, which is a socket handler, in the rest of the API functions. Apart from the IP address and port number of the server, the other arguments that have to be passed to this function are TCP_OPEN_RAM_HOST, which is the type of the socket, and TCP_PURPOSE_GENERIC_TCP_CLIENT, which is its purpose. When a socket is opened, its handler, in this case "MySocket", is passed to TCPIsConnected(MySocket) to check whether or not the socket was in the state of connected. If it returned false, the operation must be aborted and the socket should be released.

Before data can be sent to the remote address, TCPIsPutReady(MySocket) is called to make sure that data can be written to the socket. Since the return of this function is

the number of bytes that can be written to the socket, the function can be used in a conditional statement. Now a buffer of the data that is to be sent to the remote server can be prepared and passed to TCPPutROMString(MySocket, buffer). Albeit data that is written to the socket will be automatically sent based on the conditions that are mentioned in table 8, TCPFlush(MySocket) is called to make sure time is not wasted. Immediately after the data was sent, TCPIsGetReady(MySocket, buffer, n) is called to receive ACK from the server, if the ACK was not right, the data will be re-sent.

The above process is repeated in a loop of ten iterations. In this case, instead of RFID reader, varying data is sent ten times before a quit message is sent to the server and finally TCPDisconnect(MySocket) is called. To indicate the flow of sent and received data, two of the on-board LEDs are lit between each operation and sent and received data is also displayed on the LCD. Figure 30 illustrates these steps in a simplified flowchart.

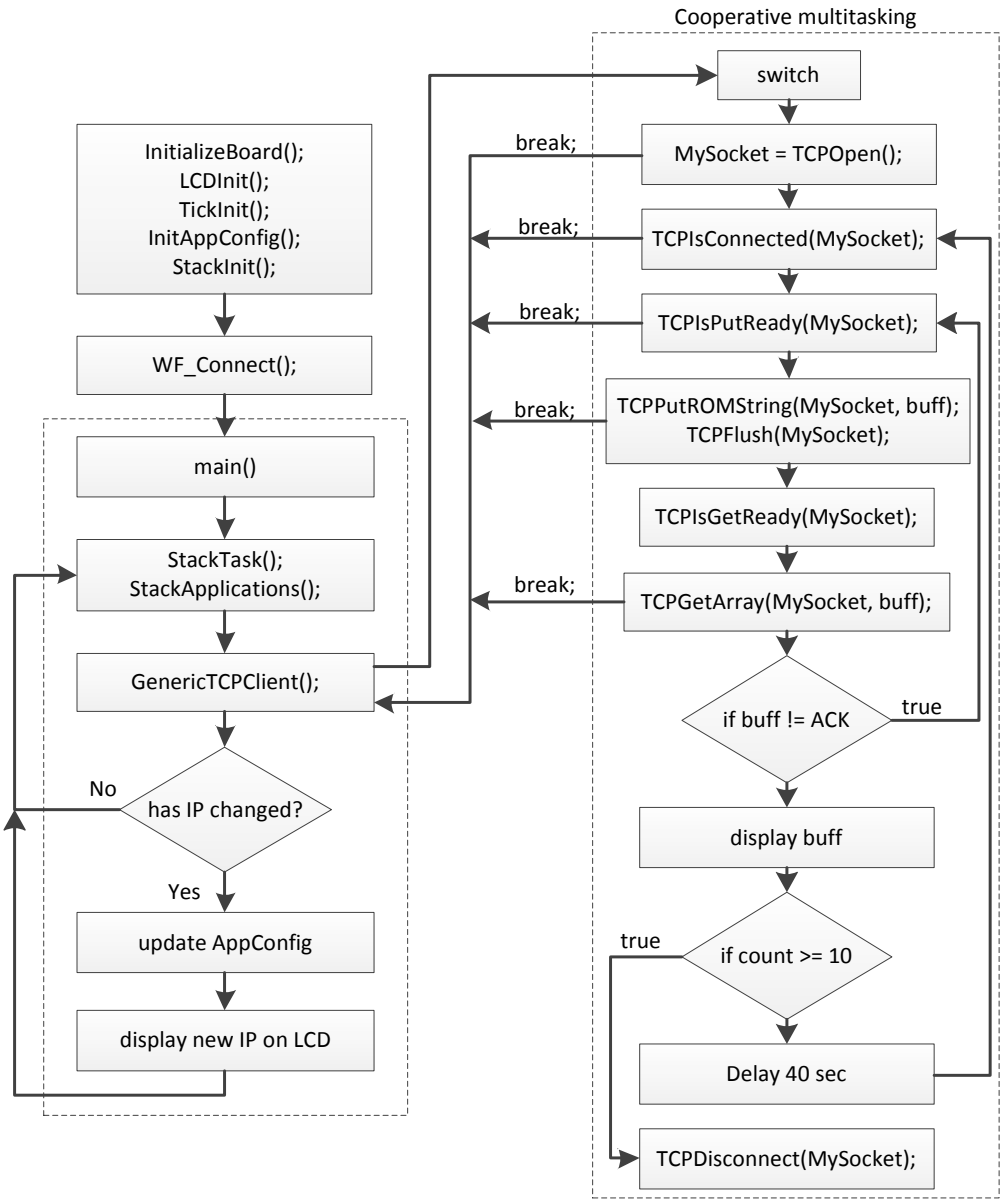


Figure 30. Client's main and user applications in a simplified flowchart

As can be seen in figure 30, a cooperative multitasking for the user application is implemented using C's switch statement. This will ensure a continual call of StackTask() function and proper operation of the stack. Partial, relevant application code is given in appendix 8.

5.7 Preliminary Tests

There are two stages of the preliminary tests: hardware and software tests. The hardware test of the client portable device starts with testing the soldered components with a multi-metre. After checking all the individual soldered points, a testing programme is in order. A testing programme can be designed to test all the ports of the MCU by simply activating them, checking them with an LED or a voltage-metre and then deactivating them. Before testing the system with the client portable device, basic software tests of the server programme can be done by writing a simple programme on a wireless-enabled laptop. The laptop can take the role of the client portable device to connect to the server and send some text messages.

6 System Test Results

To set up a testing environment that can reliably reflect a real-life situation, an additional wireless-enabled laptop was needed to take the role of a second portable client device. This was necessary to test the concurrency of the server. Figure 31 shows a diagram of the system as a whole.

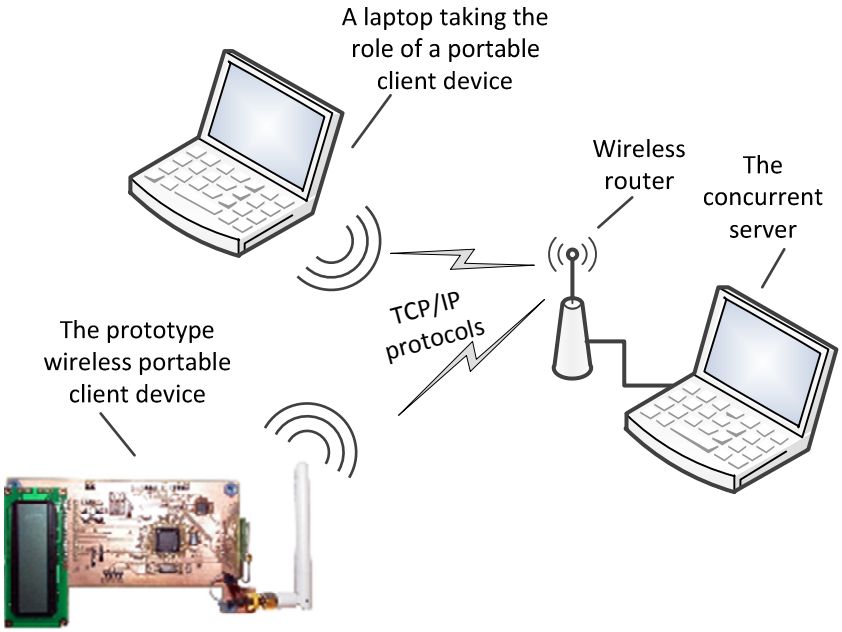


Figure 31. The system set-up for testing

As can be seen from figure 31, a WAN of four nodes has been established. The client nodes are: two portable, wireless, client devices, one is the prototype board and the other a wireless-enabled laptop. These nodes are assumed to be in two different classrooms. The other nodes of the WAN are a wireless router, and a server which is running on a laptop. The server must first be running before enabling the clients. The clients can now establish a TCP/IP connection with the server and start sending classroom attendance registration information. The exchanged messages are simple alphanumeric messages.

While the reason for a second client was to test whether or not the server was concurrent, other reasons for this node was unimportant. In fact, the program code of this node was just to simulate a second client with no particular interest in the details of the exchanged messages and the timing of these messages. The focus, however, was on testing the prototype board and its practicality and efficiency. Since there were no RFID readers, inputs to the board was simulated in the programme itself. Ten varying messages were sent to the server every four seconds and timing measurements were recorded for each message. Figure 32 is a screenshot of the outputs of the server when connected to the prototype client alone.

```
Connected to 192.168.0.10:2476

From 192.168.0.10:2476: Student 0 time:0.001770
From 192.168.0.10:2476: Student 1 time:4.788057
From 192.168.0.10:2476: Student 2 time:4.999110
From 192.168.0.10:2476: Student 3 time:4.999850
From 192.168.0.10:2476: Student 4 time:5.005706
From 192.168.0.10:2476: Student 5 time:4.995905
From 192.168.0.10:2476: Student 6 time:4.999785
From 192.168.0.10:2476: Student 7 time:5.002176
From 192.168.0.10:2476: Student 8 time:4.999178
From 192.168.0.10:2476: Student 9 time:4.998322
From 192.168.0.10:2476: Student 10 time:5.000385

A client is exiting ...
```

Figure 32. Output of the server received from the prototype board

Figure 32 shows ten outputs that were sent from the client prototype board. The first line indicates the IP address and port number of the client. This information is

displayed for each different client sending messages from different classrooms. The other pieces of information are the name of the students who approached their keys to the device, and the time it took for receiving and acknowledging the TCP/IP messages. Since the client sent a message every four seconds, the average time for both receiving and acknowledging a message was about one second. A second of time in this situation is more than sufficient for a soft real-time system. To test the concurrency of the server, the second client laptop was switched on and new results were recorded. Figure 33 shows a snapshot of the output of the server.

```
bal@bal-Inspiron-9300:~/Desktop$ ./server.o
Connected to 192.168.0.12:49453
From 192.168.0.12:49453: Hello World! time:0.000091
From 192.168.0.12:49453: Hello World! time:3.005564
From 192.168.0.12:49453: Hello World! time:3.004345
Connected to 192.168.0.10:2660
From 192.168.0.10:2660: Student 0 time:0.001419
From 192.168.0.12:49453: Hello World! time:3.004583
From 192.168.0.12:49453: Hello World! time:3.004242
From 192.168.0.10:2660: Student 1 time:4.762214
From 192.168.0.12:49453: Hello World! time:3.003802
From 192.168.0.12:49453: Hello World! time:3.004210
From 192.168.0.10:2660: Student 2 time:5.006889
From 192.168.0.12:49453: Hello World! time:3.004704
From 192.168.0.10:2660: Student 3 time:4.993195
From 192.168.0.12:49453: Hello World! time:3.004806
From 192.168.0.12:49453: Hello World! time:3.004095
From 192.168.0.10:2660: Student 4 time:5.002220
From 192.168.0.12:49453: Hello World! time:3.005586
From 192.168.0.12:49453: Hello World! time:3.004391
From 192.168.0.10:2660: Student 5 time:4.997726
From 192.168.0.12:49453: Hello World! time:3.003814
From 192.168.0.10:2660: Student 6 time:5.000317
From 192.168.0.12:49453: Hello World! time:3.003992
From 192.168.0.12:49453: Hello World! time:3.004540
From 192.168.0.10:2660: Student 7 time:5.000879
From 192.168.0.12:49453: Hello World! time:3.005444
From 192.168.0.12:49453: Hello World! time:3.004629
From 192.168.0.10:2660: Student 8 time:4.999949
From 192.168.0.12:49453: Hello World! time:3.004784
From 192.168.0.10:2660: Student 9 time:4.999145
From 192.168.0.12:49453: Hello World! time:3.004815
From 192.168.0.12:49453: Hello World! time:3.005683
From 192.168.0.10:2660: Student 10 time:4.999821

A client is exiting ...
From 192.168.0.12:49453: Hello World! time:3.004122
From 192.168.0.12:49453: Hello World! time:3.082227
```

Figure 33. Output of the server received from the prototype board and a second client

Figure 33 illustrates the results of two clients, the prototype board and a second client, outputted by the server. The second client, which was a wireless-enabled laptop, was programmed to send a "Hello World!" message every 3rd second. First the second client was switched on and only after about 6-7 seconds, the prototype board established a connection with the server. This was indicated in the 6th line from figure 33. Comparing these new results with the previous ones, it was clear that the server was indeed concurrent.

7 Discussion

While the wireless system of this project was specifically designed and implemented for classroom attendance markings, the solution can be applied in any design where a wireless client-server model is needed. For example, the system design of this project requires only slight modifications before it can be implemented in wireless data acquisition systems. This is because Microchip PIC32MX795F512L MCU has built-in analogue-to-digital converters (ADC) which can be utilised to convert data from outputs of analogue sensors to digital form and then wirelessly transfer them to the central server for processing. An example of the applications of data acquisition systems is a farm where collecting and monitoring soil-related data, temperature, and moisture may improve production. In this case, several portable clients can be installed in sensitive places on the farm and a central server can be fixed in a nearby building or anywhere on the planet with internet access, for collecting and processing the farm-related data.

Albeit the results of the system test were in line with the timing requirement for a soft real-time system, they can still be improved by eliminating unnecessary ACK signals. This can be achieved by message matching on the server, and only when a match fails, an ACK can be sent to clients for re-sending the data. In addition, there is room for improving error checking. The software design of the project does not provide full error checking. For example, when the server is unavailable a client should still operate and store student attendance information while continue checking for server availability.

A drawback of the system solution that has been given here is that it does not provide an easy mechanism for updating or upgrading the stack modules. This is due to the time constraints for the development of this project. A solution to this would be to separate the modules of the software that may require updating in the future into dedicated folders. When an update was available, it could be simply copied over the existing files in these folders and a re-build command be issued in the IDE. Another drawback is the lack of an interface to a PC. Since the MCU has several built-in hardware modules such as UARTs and USBs, an interface can easily be added to the system, which can be used to transfer data to a PC when a server was unavailable for a long enough time.

8 Conclusions

Computerising classroom attendance marking has many advantages over the old system. Data from classrooms can easily be transformed into databases for possible later analyses or usages. Accurate date can be collected over a stretch of time from which graphs can be generated and studied to predict students' attendance behaviour. This may help in improving the design of timetables and classroom reservation decisions. A major disadvantage, however, is maintaining the software programmes of these systems. The goal of this project was to test and evaluate a soft real-time, wireless system using Microchip hardware and software solutions. The project also showed that combining existing solutions can result in useful, new systems.

The results from the tests that were conducted on the system collectively show that the system design is suitable for engineering challenges that require soft real-time solutions. Furthermore, the success of these tests left no doubt that Microchip solutions are easy to implement, inexpensive in terms of cost, and effective in terms of reliability and efficiency. A minor concern, however, might be in the area of maintenance. Another concern might be that, while Microchip is committed to developing and supporting its solutions, there is no guarantee about the future existence of these solutions, especially the continuation of the Microchip TCP/IP Stack.

There are two areas of the system solution that can be further improved in the future versions of the design. Feedback signals from the portable client devices can be improved by adding sound indicators alongside the present visual feedback. The other area in which there is room for improvement, apart from attendance marking, air quality, such as oxygen level and temperature, can also be measured and sent to the central server for real-time alarms and other analysis.

References

1 Rusling D. The Linux Kernel. 1996-1999 [online]. Wokingham, UK; 1999
URL: http://tldp.org/LDP/tlk/tlk-toc.html
Accessed 11 February 2011.

2 Love R. Linux System Programming. 1st ed. USA, CA: O'Reilly Media; 2007.

3 Gallmeister B. POSIX. 4: Programming for the Real World. 1st ed. USA, CA: O'Reilly & Associates, Inc.; 1995.

4 W. Richard Stevens, Fenner B, Andrew M. Rudoff. UNIX Network Programming, The Sockets Networking. Volume 1. 3rd ed. USA, CA: Pearson Education, Inc.; 2004.

5 Discover and Learn. Wi-Fi Alliance. [online]. Austin, USA; 2011.
URL: http://www.wi-fi.org/discover_and_learn.php
Accessed 26 February 2011.

6 Developing Bluetooth Technology Solutions. Bluetooth SIG, Inc. [online]. USA 2011.
URL: http://www.bluetooth.com/ENGLISH/TECHNOLOGY/Pages/default.aspx
Accessed 26 February 2011.

7 Understanding ZigBee. ZigBee Alliance. [online]. USA 2011.
URL: http://www.zigbee.org/About/UnderstandingZigBee.aspx
Accessed 26 February 2011.

8 Wireless Design Center – Sub GHz. Microchip 2011. [online]. USA 2011.
URL: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2664&param=en542229
Accessed 27 February 2011.

9 Beagleboard Product Details. BeagleBoard.org. [online].
URL: http://beagleboard.org/hardware
Accessed 27 February 2011.

10 PIC32MX5XX/6XX/7XX Family Data Sheet. Microchip 2010 [online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/61156F.pdf
Accessed 01 March 2011.

11 MRF24WB0MA RF Transceiver Module Data Sheet. Microchip 2010 [online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/70632A.pdf
Accessed 01 March 2011.

12 25AA1024 EEPROM Module Data Sheet. Microchip 2010 [online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/21836H.pdf
Accessed 10 March 2011.

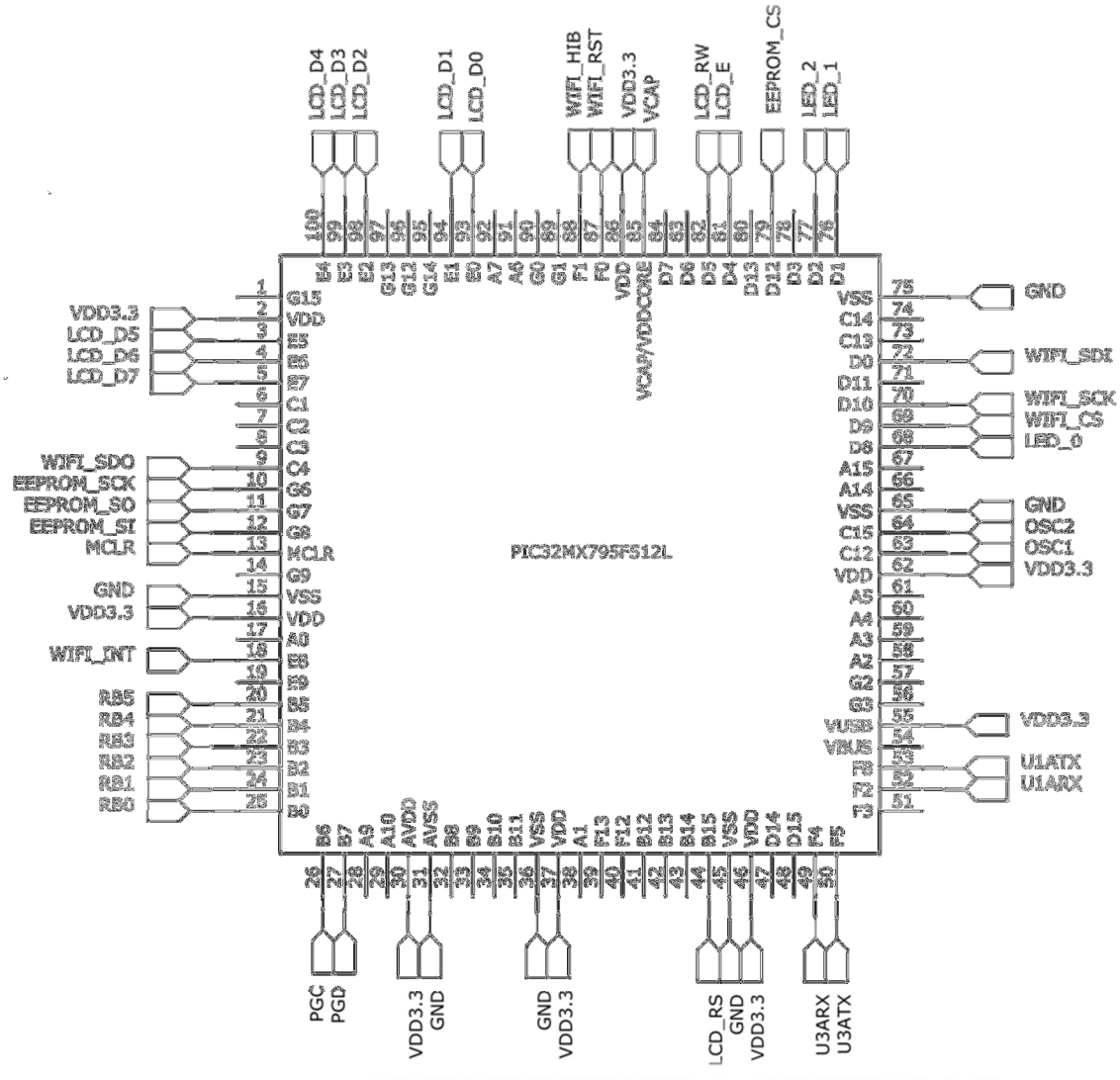
13 Rajbharti N. Application Note AN833, the Microchip TCP/IP Stack. USA:
Microchip Technology Inc.; 2008.

14 Microchip TCP/IP Stack Help 5.25v. USA: Microchip Technology Inc.; 2010.

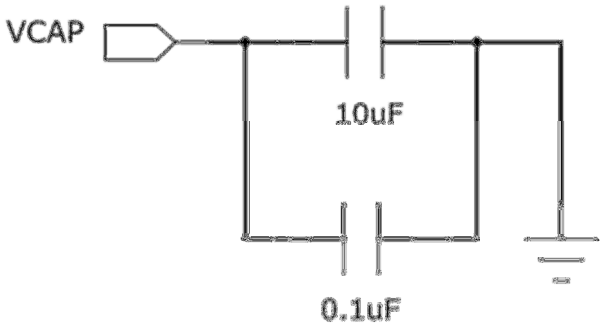
15 Microchip PICkit 3 Programmer/Debugger User's Guide. Microchip 2010 [online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf
Accessed 14 March 2011.

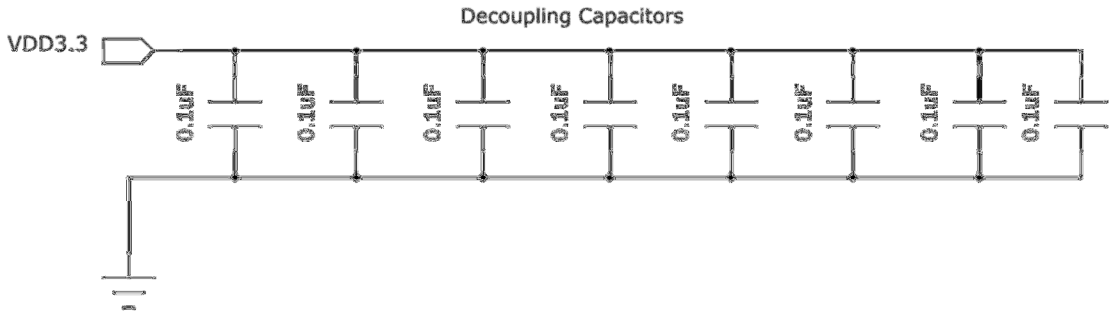
16 WPA PSK Generator. Wireshark Foundation [online].
URL: http://www.wireshark.org/tools/wpa-psk.html
Accessed 14 March 2011.

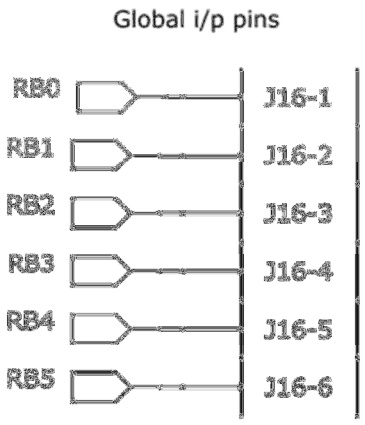
Appendix 1. Schematics of the Client Prototype Board

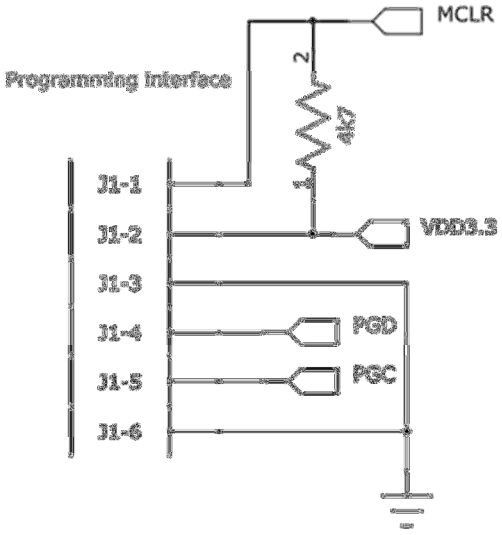


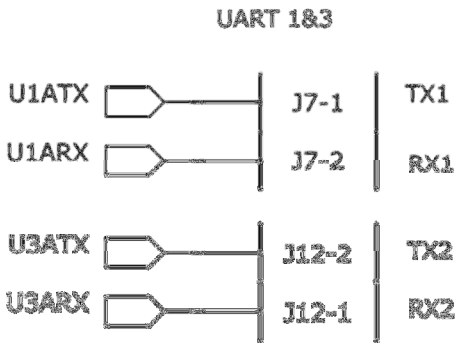
Capacitor on Internal Voltage Regulator

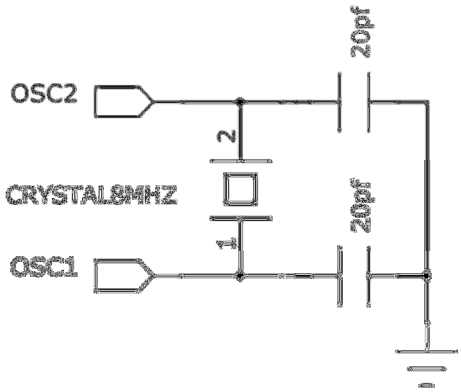


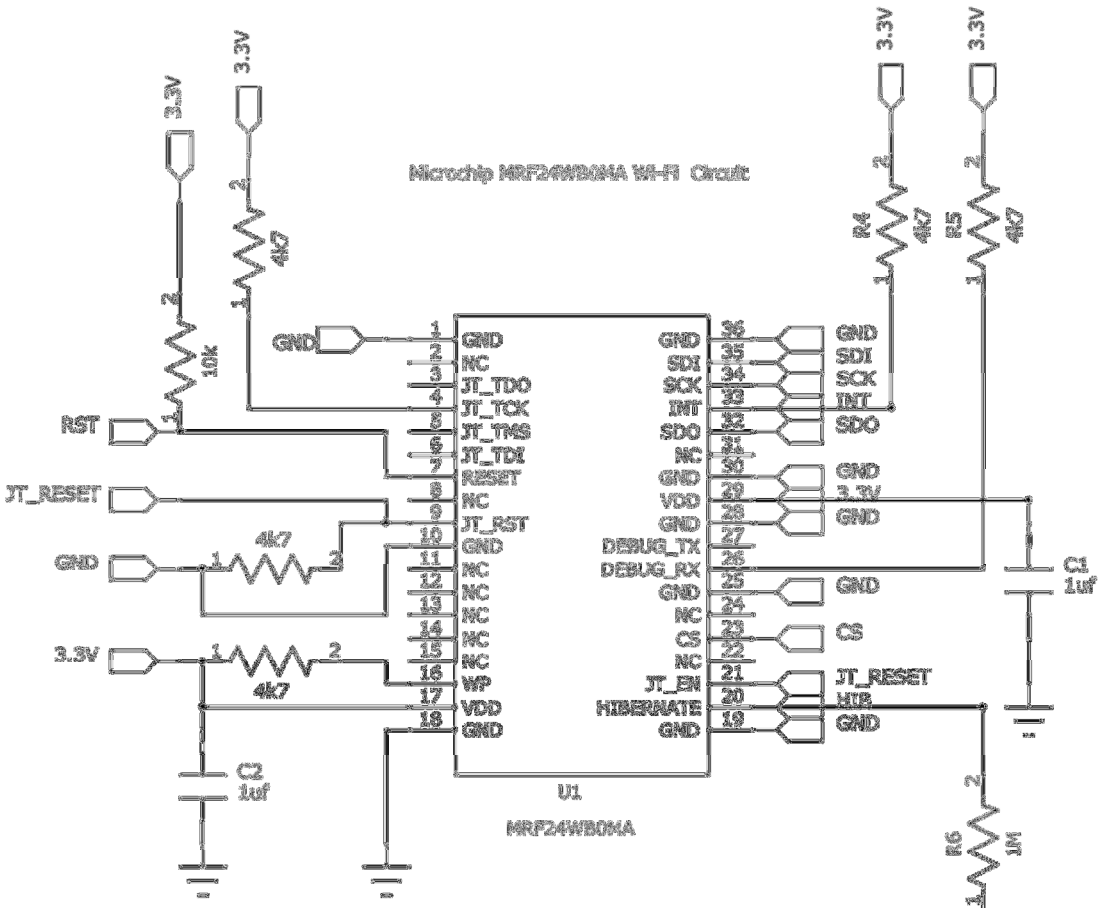

Decoupling Capacitors
VDD3.3
0.1uF
0.1uF
0.1uF
0.1uF
0.1uF
0.1uF
0.1uF
0.1uF

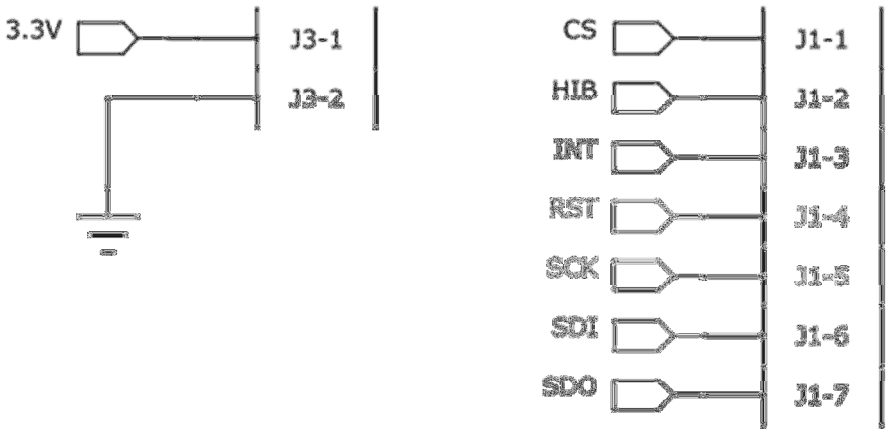

Global i/p pins
RB0
RB1
RB2
RB3
RB4
RB5
J16-1
J16-2
J16-3
J16-4
J16-5
J16-6


MCLR
Programming Interface
2
4K7
1
J1-1
J1-2
J1-3
J1-4
J1-5
J1-6
VDD3.3
PGD
PGC

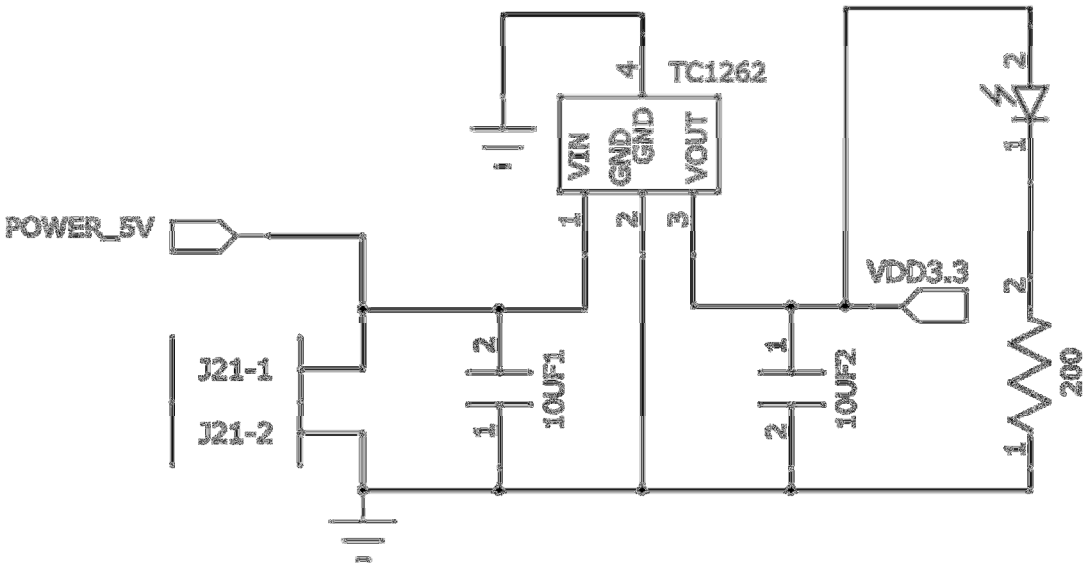

UART 1&3
U1ATX
U1ARX
J7-1
J7-2
TX1
RX1
U3ATX
U3ARX
J12-2
J12-1
TX2
RX2


20pf
OSC2
2
CRYSTAL8MHZ
1
20pf
OSC1

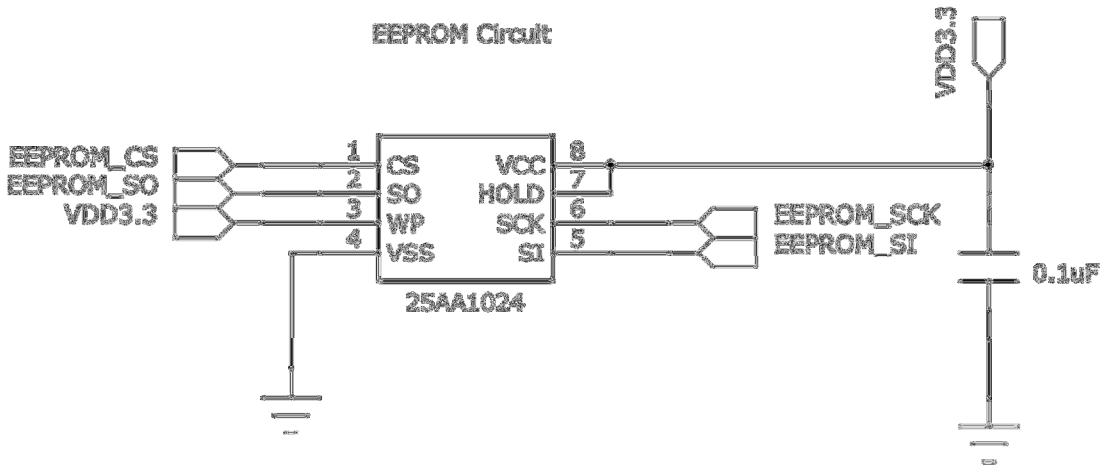

Microchip MRF24WB0MA Wi-Fi Circuit
3.3V
4k7
10k
RST
JT_RESET
GND
4k7
3.3V
4k7
C2
1uf
U1
MRF24WB0MA
GND
NC
JT_TDO
JT_TCK
JT_TMS
JT_TDI
RESET
NC
JT_RST
GND
NC
NC
NC
NC
NC
WP
VDD
GND
GND
SDI
SCK
INT
SDO
NC
GND
VDD
GND
DEBUG_TX
DEBUG_RX
GND
NC
CS
NC
JT_EN
HIBERNATE
GND
R4
4k7
R5
4k7
C1
1uf
R6
1M
CS
HIB
INT
SDO
JT_RESET


3.3V
J3-1
J3-2
CS
HIB
INT
RST
SCK
SDI
SDO
J1-1
J1-2
J1-3
J1-4
J1-5
J1-6
J1-7

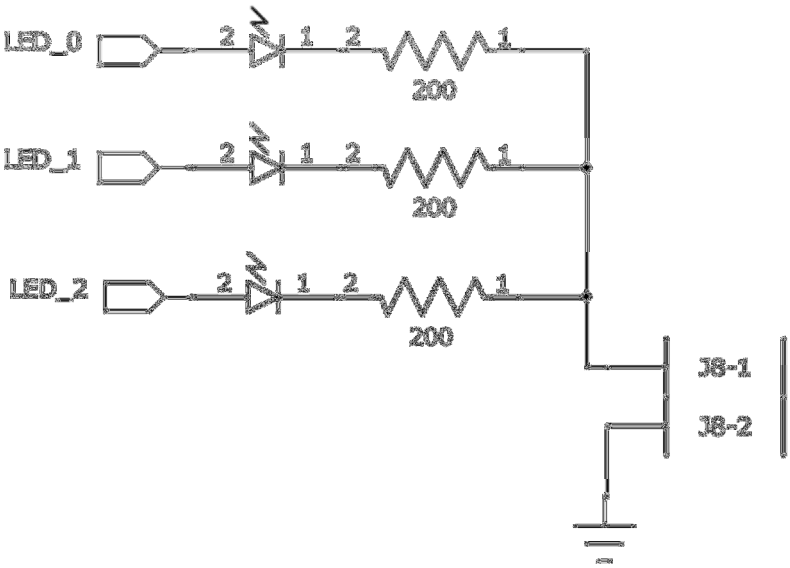
Power Circuit



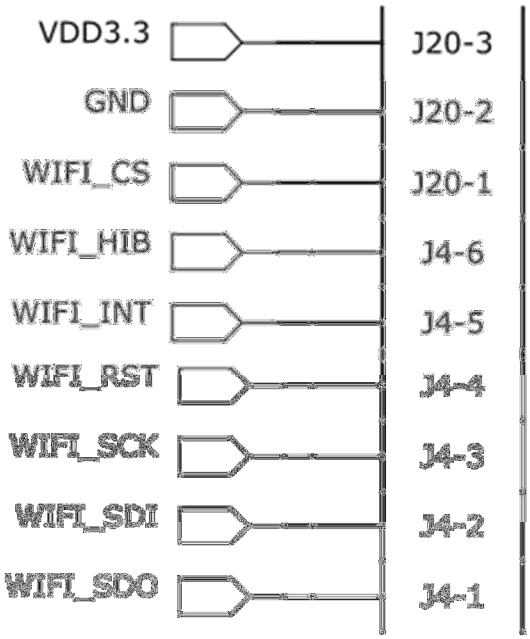
EEPROM Circuit



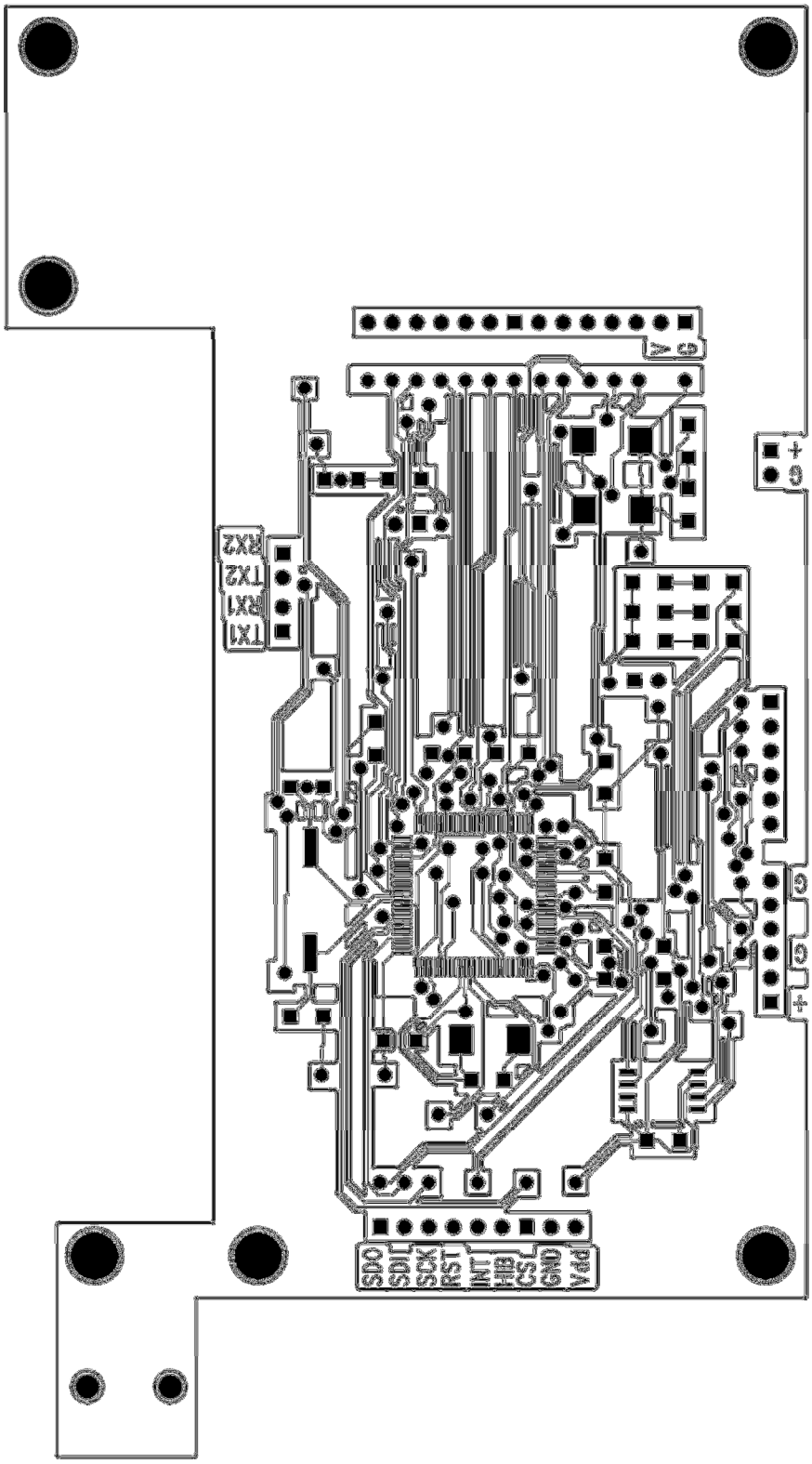
LED Circuit

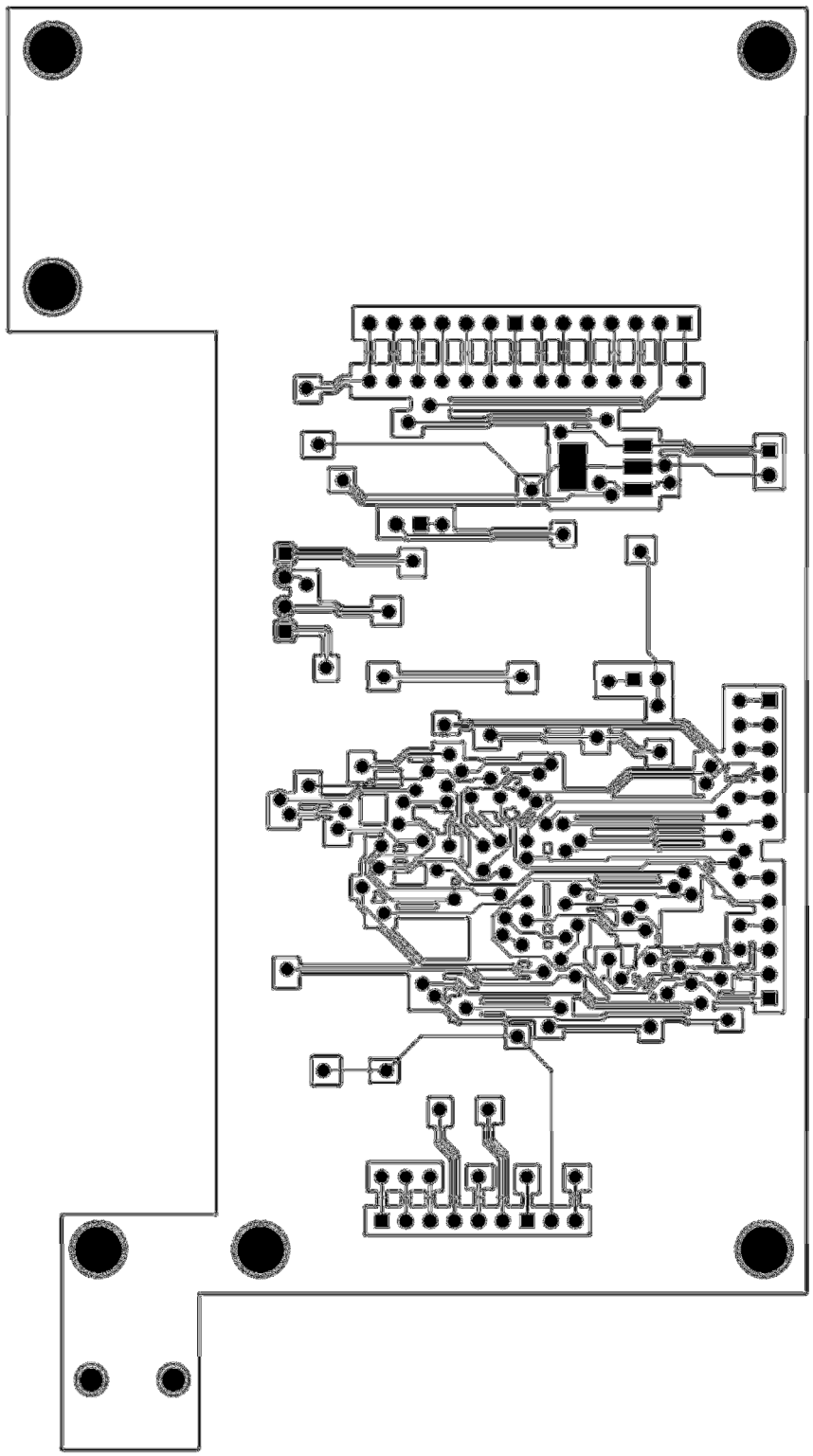


WiFi Connector

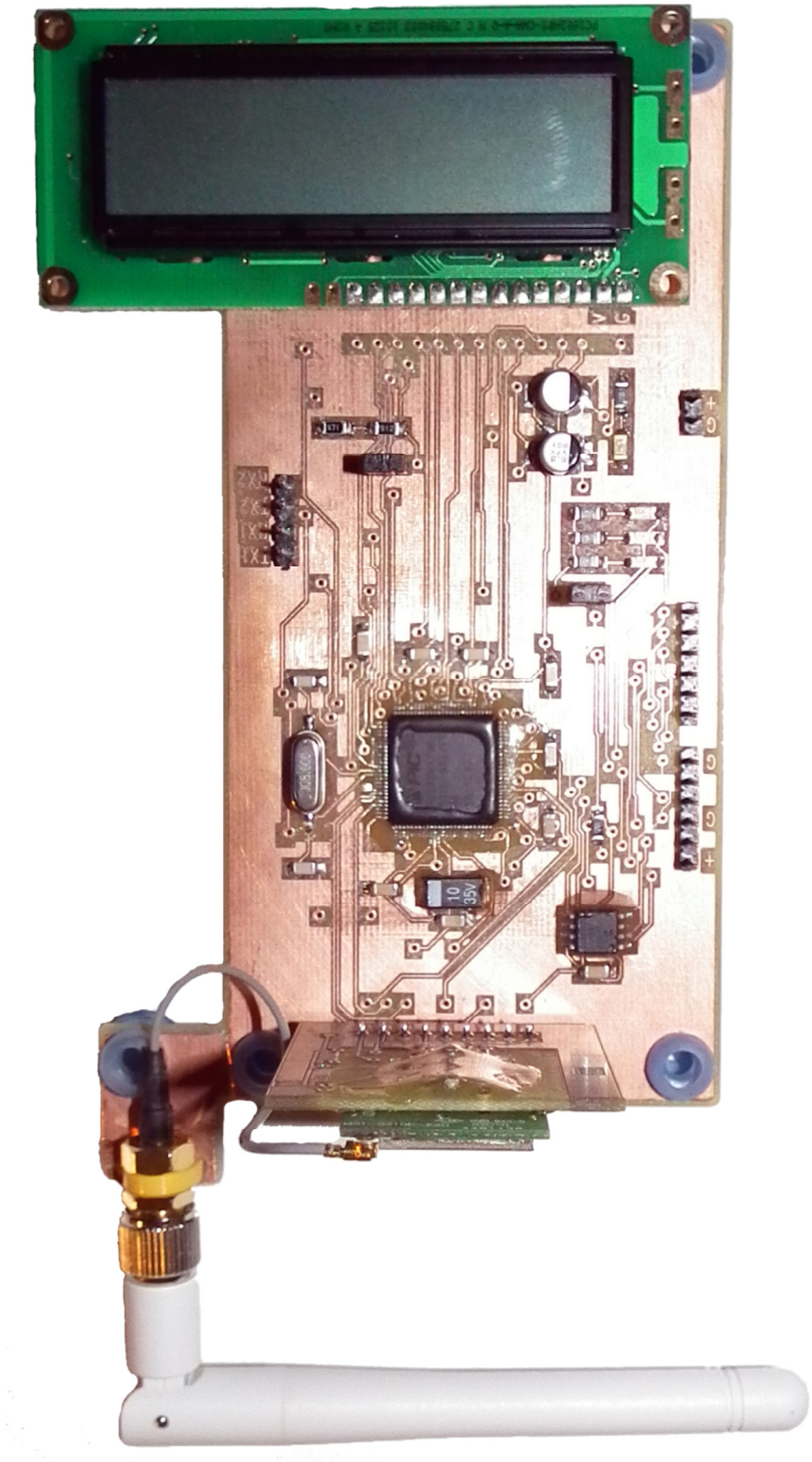


Appendix 2. PCB Layout of the Client Prototype Board





Appendix 3. A Photographic Picture of the Client Prototype Board



Appendix 4. A Simple Server Programme Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <strings.h>
#include <arpa/inet.h>

#define SERV_TCP_PORT 6543

int main (int argc, char ** argv, char ** env) {

    int socket_server, socket_client, child_pid;
    struct sockaddr_in address_client, address_server;
    socklen_t client_address_size;
    struct timespec systemTime; //for capturing time

    // open a tcp socket to listen on
    if ( (socket_server = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
        perror("server: can't open stream socket");

    bzero((void *) &address_server, (size_t)sizeof(address_server));

    address_server.sin_family = AF_INET;
    address_server.sin_addr.s_addr = htonl(INADDR_ANY);
    address_server.sin_port = htons(SERV_TCP_PORT);

        // bind to our local address and start listening to clients
    if (bind(socket_server, (struct sockaddr *) &address_server,
                sizeof(address_server)) < 0)
        perror("server: can't bind local address");

    listen(socket_server, 5);

    for ( ; ; ) {

                client_address_size = sizeof(address_client);

        /* wait for client connection and accpt it when it comes */
        if ( (socket_client = accept( socket_server, (struct sockaddr *)
                        &address_client, &client_address_size)) < 0)
            perror("server: accept error");

        if ( (child_pid = fork() ) < 0)
            perror("server: fork error");

        // the client starts here
        else if (child_pid == 0) {

                        // client process has no need of the server socket
                        close(socket_server);

                        printf( "Connected to %s:%d\n", inet_ntoa(address_client.sin_addr),
                                                ntohs(address_client.sin_port) );

                        char msg[50];
                        double tmp1, tmp2;
                        int stop = 0;
```

```
                        //read time to be used later for time calcualtions
                        clock_gettime(CLOCK_REALTIME, &systemTime);
                        tmp2 = systemTime.tv_sec + systemTime.tv_nsec/1000000000.0;
                        tmp1 = tmp2;

                        while(!stop){
                                // read message from client
                                if ( recv(socket_client, msg, sizeof(msg),0 ) <0) {
                                        perror("recv");
                                        return -1;
                                }

                                //read time right after the reading of
                                //measurement for time differences
                                clock_gettime(CLOCK_REALTIME, &systemTime);
                                tmp2 = systemTime.tv_sec + systemTime.tv_nsec/1000000000.0;

                                //has client sent quit message?
                                if( msg[0] == 'Q' || msg[1] == 'Q' || msg[2] == 'Q' ){
                                stop = 1;
                                }

                                else {
                                        //printf received message with the timing
                                        printf( "From %s:%d: %s time:%f \n",
                                                inet_ntoa(address_client.sin_addr),
                                                ntohs(address_client.sin_port),
                                                msg,tmp2-tmp1 );
                                        tmp1 = tmp2;
                                        if ( send(socket_client, "OK",
                                                sizeof("OK"),0) < 0 ) {
                                                perror("write");    return -1;
                                        }
                                }
                        }

                        close(socket_client);
                        printf("\nA client is exiting ...\n");
                                        exit(EXIT_SUCCESS);
        }

        /* the server has no need to keep the client socket open */
        close(socket_client);
    }
    return 0;
}
```

Appendix 5. Contents of HardwareProfile.h

```
#ifndef HARDWARE_PROFILE_H
#define HARDWARE_PROFILE_H

#include "Compiler.h"

// Define a macro describing this hardware set up
#define PIC32_USB_DM320003_2

// Set configuration fuses
#if defined(THIS_IS_STACK_APPLICATION)
        #pragma config FPLLODIV = DIV_1, FPLLMUL = MUL_20,
                        FPLLIDIV = DIV_2, FWDTEN = OFF, FPBDIV = DIV_2,
                        POSCMOD = HS, FNOSC = PRIPLL, CP = OFF
#endif

// Clock frequency values
// These directly influence timed events using the Tick module.
//They also are used for UART and SPI baud rate generation.
#define GetSystemClock()        (80000000ul)                    // Hz
#define GetInstructionClock()   (GetSystemClock()/1)
#define GetPeripheralClock()    (GetSystemClock()/1)

// Hardware I/O pin mappings
// LEDs
#define LED0_TRIS               (TRISDbits.TRISD8)      // Ref LED1
#define LED0_IO                 (LATDbits.LATD8)
#define LED1_TRIS               (TRISDbits.TRISD1)      // Ref LED2
#define LED1_IO                 (LATDbits.LATD1)
#define LED2_TRIS               (TRISDbits.TRISD2)      // Ref LED3
#define LED2_IO                 (LATDbits.LATD2)
#define LED_GET()               ((unsigned char)LATD & 0x07)
#define LED_PUT(a)              do{LATD = (LATD & 0xFFF8) | ((a)&0x07);}while(0)

// 25LC1024 I/O pins in SPI2
#define EEPROM_CS_TRIS          (TRISDbits.TRISD12)
#define EEPROM_CS_IO            (LATDbits.LATD12)
#define EEPROM_SCK_TRIS         (TRISGbits.TRISG6)
#define EEPROM_SDI_TRIS         (TRISGbits.TRISG7)
#define EEPROM_SDO_TRIS         (TRISGbits.TRISG8)
#define EEPROM_SPI_IF           (IFS1bits.SPI2RXIF)
#define EEPROM_SSPBUF           (SPI2BUF)
#define EEPROM_SPICON1          (SPI2CON)
#define EEPROM_SPICON1bits      (SPI2CONbits)
#define EEPROM_SPIBRG           (SPI2BRG)
#define EEPROM_SPISTAT          (SPI2STAT)
#define EEPROM_SPISTATbits      (SPI2STATbits)
```

```
// LCD Module I/O pins
#define LCD_DATA_TRIS           (*((volatile unsigned char*)&TRISE))
#define LCD_DATA_IO             (*((volatile unsigned char*)&LATE))
#define LCD_RD_WR_TRIS          (TRISDbits.TRISD5)
#define LCD_RD_WR_IO            (LATDbits.LATD5)
#define LCD_RS_TRIS             (TRISBbits.TRISB15)
#define LCD_RS_IO               (LATBbits.LATB15)
#define LCD_E_TRIS              (TRISDbits.TRISD4)
#define LCD_E_IO                (LATDbits.LATD4)

// MRF24WB0M in SPI1 slot
#define WF_CS_TRIS              (TRISDbits.TRISD9)
#define WF_CS_IO                (LATDbits.LATD9)
#define WF_SDI_TRIS             (TRISCbits.TRISC4)
#define WF_SCK_TRIS             (TRISDbits.TRISD10)
#define WF_SDO_TRIS             (TRISDbits.TRISD0)
#define WF_RESET_TRIS           (TRISFbits.TRISF0)
#define WF_RESET_IO             (LATFbits.LATF0)
#define WF_INT_TRIS             (TRISEbits.TRISE8)  // INT1
#define WF_INT_IO               (PORTEbits.RE8)
#define WF_HIBERNATE_TRIS       (TRISFbits.TRISF1)
#define WF_HIBERNATE_IO         (PORTFbits.RF1)
#define WF_INT_EDGE             (INTCONbits.INT1EP)
#define WF_INT_IE               (IEC0bits.INT1IE)
#define WF_INT_IF               (IFS0bits.INT1IF)
#define WF_INT_IE_CLEAR         IEC0CLR
#define WF_INT_IF_CLEAR         IFS0CLR
#define WF_INT_IE_SET           IEC0SET
#define WF_INT_IF_SET           IFS0SET
#define WF_INT_BIT              0x00000080
#define WF_INT_IPCSET           IPC1SET
#define WF_INT_IPCCLR           IPC1CLR
#define WF_INT_IPC_MASK         0xFF000000
#define WF_INT_IPC_VALUE        0x0C000000
#define WF_SSPBUF               (SPI1BUF)
#define WF_SPISTAT              (SPI1STAT)
#define WF_SPISTATbits          (SPI1STATbits)
#define WF_SPICON1              (SPI1CON)
#define WF_SPICON1bits          (SPI1CONbits)
#define WF_SPI_IE_CLEAR         IEC0CLR
#define WF_SPI_IF_CLEAR         IFS0CLR
#define WF_SPI_INT_BITS         0x03800000
#define WF_SPI_BRG              (SPI1BRG)
#define WF_MAX_SPI_FREQ         (10000000ul)    // Hz

#endif // #ifndef HARDWARE_PROFILE_H
```

Appendix 6. Contents of TCPIPConfig.h

```
#ifndef __TCPIPCONFIG_H
#define __TCPIPCONFIG_H

#include "GenericTypeDefs.h"
#include "Compiler.h"
#define GENERATED_BY_TCPIPCONFIG "Version 1.0.3383.23374"

//   Application Options

// Dynamic link-layer IP address automatic configuration protocol
#define STACK_USE_AUTO_IP

// Dynamic Host Configuration Protocol client for obtaining
//IP address and other parameters
#define STACK_USE_DHCP_CLIENT

//   Data Storage Options

//EEPROM Addressing Selection
#define USE_EEPROM_25LC1024
#define MPFS_RESERVE_BLOCK                              (205ul)

//Maximum number of simultaneously open MPFS2 files.
#define MAX_MPFS_HANDLES                            (7ul)


//   Network Addressing Options
// Default Network Configuration
#define MY_DEFAULT_HOST_NAME                        "balen"

#define MY_DEFAULT_MAC_BYTE1            (0x00)
#define MY_DEFAULT_MAC_BYTE2            (0x04)
#define MY_DEFAULT_MAC_BYTE3            (0xA3)
#define MY_DEFAULT_MAC_BYTE4            (0x00)
#define MY_DEFAULT_MAC_BYTE5            (0x00)
#define MY_DEFAULT_MAC_BYTE6            (0x00)

#define MY_DEFAULT_IP_ADDR_BYTE1        (169ul)
#define MY_DEFAULT_IP_ADDR_BYTE2        (254ul)
#define MY_DEFAULT_IP_ADDR_BYTE3        (1ul)
#define MY_DEFAULT_IP_ADDR_BYTE4        (1ul)

#define MY_DEFAULT_MASK_BYTE1           (255ul)
#define MY_DEFAULT_MASK_BYTE2           (255ul)
#define MY_DEFAULT_MASK_BYTE3           (0ul)
#define MY_DEFAULT_MASK_BYTE4           (0ul)

#define MY_DEFAULT_GATE_BYTE1           (169ul)
#define MY_DEFAULT_GATE_BYTE2           (254ul)
#define MY_DEFAULT_GATE_BYTE3           (1ul)
#define MY_DEFAULT_GATE_BYTE4           (1ul)

#define MY_DEFAULT_PRIMARY_DNS_BYTE1     (169ul)
#define MY_DEFAULT_PRIMARY_DNS_BYTE2     (254ul)
#define MY_DEFAULT_PRIMARY_DNS_BYTE3     (1ul)
#define MY_DEFAULT_PRIMARY_DNS_BYTE4     (1ul)

#define MY_DEFAULT_SECONDARY_DNS_BYTE1   (0ul)
#define MY_DEFAULT_SECONDARY_DNS_BYTE2   (0ul)
#define MY_DEFAULT_SECONDARY_DNS_BYTE3   (0ul)
#define MY_DEFAULT_SECONDARY_DNS_BYTE4   (0ul)
```

```
//   PIC32MX7XX/6XX MAC Layer Options
#define ETH_CFG_LINK            0
#define ETH_CFG_AUTO            1
#define ETH_CFG_10              1
#define ETH_CFG_100             1
#define ETH_CFG_HDUPLEX         1
#define ETH_CFG_FDUPLEX         1
#define ETH_CFG_AUTO_MDIX       1
#define ETH_CFG_SWAP_MDIX       1
#define EMAC_TX_DESCRIPTORS     2
#define EMAC_RX_DESCRIPTORS     8
#define EMAC_RX_BUFF_SIZE       1536

//Transport Layer Options
// Client Mode Configuration

#define STACK_CLIENT_MODE

// TCP Socket Memory Allocation
// Allocate how much total RAM (in bytes) you want to allocate
// for use by your TCP TCBs, RX FIFOs, and TX FIFOs.
        #define TCP_ETH_RAM_SIZE         (8192ul)
        #define TCP_PIC_RAM_SIZE         (0ul)
        #define TCP_SPI_RAM_SIZE         (0ul)
        #define TCP_SPI_RAM_BASE_ADDRESS (0x00)

        // Define names of socket types
        #define TCP_SOCKET_TYPES
                #define TCP_PURPOSE_GENERIC_TCP_CLIENT 0
                #define TCP_PURPOSE_GENERIC_TCP_SERVER 1
                #define TCP_PURPOSE_TELNET 2
                #define TCP_PURPOSE_FTP_COMMAND 3
                #define TCP_PURPOSE_FTP_DATA 4
                #define TCP_PURPOSE_TCP_PERFORMANCE_TX 5
                #define TCP_PURPOSE_TCP_PERFORMANCE_RX 6
                #define TCP_PURPOSE_UART_2_TCP_BRIDGE 7
                #define TCP_PURPOSE_HTTP_SERVER 8
                #define TCP_PURPOSE_DEFAULT 9
                #define TCP_PURPOSE_BERKELEY_SERVER 10
                #define TCP_PURPOSE_BERKELEY_CLIENT 11
        #define END_OF_TCP_SOCKET_TYPES

        #if defined(__TCP_C)
                #define TCP_CONFIGURATION
                ROM struct
                {
                        BYTE vSocketPurpose;
                        BYTE vMemoryMedium;
                        WORD wTXBufferSize;
                        WORD wRXBufferSize;
                } TCPSocketInitializer[] =
                {
                        {TCP_PURPOSE_GENERIC_TCP_CLIENT, TCP_ETH_RAM, 125, 100},
                };
                #define END_OF_TCP_CONFIGURATION
        #endif
#endif
```

Appendix 7. Contents of WF_Config.h

```
#ifndef __WF_CONFIG_H_
#define __WF_CONFIG_H_

#define WF_USE_SCAN_FUNCTIONS
#define WF_USE_TX_POWER_CONTROL_FUNCTIONS
#define WF_USE_POWER_SAVE_FUNCTIONS
#define WF_USE_MULTICAST_FUNCTIONS
#define WF_USE_INDIVIDUAL_SET_GETS
#define WF_USE_GROUP_SET_GETS

#define WF_DEBUG

// Default settings for Connection Management */
#define MY_DEFAULT_SSID_NAME                "metropolia"
#define MY_DEFAULT_NETWORK_TYPE             WF_INFRASTRUCTURE
#define MY_DEFAULT_SCAN_TYPE                WF_ACTIVE_SCAN
#define MY_DEFAULT_CHANNEL_LIST             {1,6,11}
#define MY_DEFAULT_LIST_RETRY_COUNT         (WF_RETRY_FOREVER)
#define MY_DEFAULT_EVENT_NOTIFICATION_LIST  (WF_NOTIFY_CONNECTION_ATTEMPT_SUCCESSFUL  | \
                                             WF_NOTIFY_CONNECTION_ATTEMPT_FAILED      | \
                                             WF_NOTIFY_CONNECTION_TEMPORARILY_LOST    | \
                                             WF_NOTIFY_CONNECTION_PERMANENTLY_LOST    | \
                                             WF_NOTIFY_CONNECTION_REESTABLISHED)

/* WF_DISABLED or WF_ENABLED */
#define MY_DEFAULT_PS_POLL                   WF_DISABLED

#define MY_DEFAULT_WIFI_SECURITY_MODE        WF_SECURITY_WPA_AUTO_WITH_PASS_PHRASE
#define MY_DEFAULT_PSK_PHRASE               "metropolia"

#define MY_DEFAULT_PSK "\
x14\x45\x64\x7d\x21\x38\x7d\xdb\
x6f\xad\x31\x36\xdf\xe4\x6f\xd2\
xf2\x4a\xc6\x2d\x5e\x09\xaa\x6b\
x8b\xa1\x53\x0a\x64\x0f\x6e\x8f"

#endif /* __WF_CONFIG_H_ */
```

Appendix 8. The Main and User Applications Code

```
// Include all headers for any enabled TCPIP Stack functions
#include "TCPIP Stack/TCPIP.h"

// Include functions specific to this stack application
#include "Main.h"

// Used for Wi-Fi assertions
#define WF_MODULE_NUMBER   WF_MODULE_MAIN_DEMO

// Declare AppConfig structure and some other supporting stack variables
APP_CONFIG AppConfig;

// Checksum of the ROM defaults for AppConfig
static unsigned short wOriginalAppConfigChecksum;

// Private helper functions.
static void InitAppConfig(void);
static void InitializeBoard(void);
static void WF_Connect(void);

// C30 and C32 Exception Handlers
void _general_exception_handler(unsigned cause, unsigned status)
{
        Nop();
        Nop();
}

int main(void)
{
        int stop = 0;
        static DWORD t = 0, connect_t = 0;
        static DWORD dwLastIP = 0;
        char connecting_str[14];
        char connecting_tmp[2];
        int connecting_counter = 0;

        // Initialize application specific hardware
        InitializeBoard();

        LCDInit();
        DelayMs(100);

        // Initialize stack-related hardware components that may be
        // required by the UART configuration routines
        TickInit();

        // Initialize Stack and application related NV variables into AppConfig.
        InitAppConfig();

        // Initialize core stack layers (MAC, ARP, TCP, UDP) and
        // application modules (HTTP, SNMP, etc.)
        StackInit();

        WF_Connect();

        while(1)
        {
                // Blink LED0 (right most one) every second.
                if(TickGet() - t >= TICK_SECOND/2ul)
                {
                        t = TickGet();
                        LED0_IO ^= 1;
                }
```

```
                // This task performs normal stack task including checking
                // for incoming packet, type of packet and calling
                // appropriate stack entity to process it.
                StackTask();

                // This tasks invokes each of the core stack application tasks
                StackApplications();

                //user application. Start cooperative multitasking
                #if defined(STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE)
                GenericTCPClient();
                #endif

                // If the local IP address has changed (ex: due to DHCP lease change)
                // write the new IP address to the LCD display
                if(dwLastIP != AppConfig.MyIPAddr.Val)
                {
                        dwLastIP = AppConfig.MyIPAddr.Val;
                        strcpypgm2ram((char*)LCDText,"Connected via:");
                        LCDUpdate();
                        DisplayIPValue(AppConfig.MyIPAddr);
                        stop++;
                }

                //a second counter shown on LCD for visual checking only
                if(stop<2)
                {
                        if(TickGet() - connect_t >= TICK_SECOND/2ul)
                        {
                                connect_t = TickGet();
                                strcpy(connecting_str, "Connecting: ");
                                sprintf(connecting_tmp, "%d", ++connecting_counter);
                                strcat(connecting_str,connecting_tmp);
                                strcat(connecting_str,"s");
                                strcpypgm2ram((char*)LCDText,connecting_str);
                                LCDUpdate();
                        }
                }

        }
}//end of main()
```

```
// Defines the server to be accessed for this application
static BYTE ServerName[] = "192.168.0.11";

// Defines the port to be accessed for this application
static WORD ServerPort = 6543;

int t1 = 0, t2=0, student_id=0;
DWORD tim = 0;
BYTE    send_str[10],recv_str[10];
BYTE    send_tmp[3];

void GenericTCPClient(void)
{

        static DWORD            Timer;
        static TCP_SOCKET       MySocket = INVALID_SOCKET;

        static enum _SocketState
        {
                start_delay = 0,
                create_socket_0,
                connect_socket_1,
                prepare_msg_2,
                send_msg_3,
                receive_msg_4,
                process_msg_5,
                time_loop_6,
                back_prepare_7,
                disconnect_socket_8,
                task_done_9
        } SocketState = 0;

        switch(SocketState)
                {

                case start_delay:

                        //wait for 40 seconds

                        if(t1<=40){
                                if(TickGet() - tim >= TICK_SECOND/2ul)
                                {
                                    tim = TickGet();
                                    t1++;
                                }
                                SocketState = start_delay;
                                break;
                        }

                        SocketState++;
                        break;

                case create_socket_0:

                        // Connect a socket to the remote TCP server
                        MySocket = TCPOpen((DWORD)&ServerName[0], TCP_OPEN_RAM_HOST,
                        ServerPort, TCP_PURPOSE_GENERIC_TCP_CLIENT);
```

```
		// Abort operation if no TCP socket of type
		// TCP_PURPOSE_GENERIC_TCP_CLIENT is available
		// If this ever happens, you need to go add one to TCPIPConfig.h
		if(MySocket == INVALID_SOCKET){
			break;
		}

		SocketState++;
		break;

	case connect_socket_1:
		Timer = TickGet();
		// Wait for the remote server to accept our connection request
		if(!TCPIsConnected(MySocket))
		{
			// Time out if too much time is spent in this state
			if(TickGet()-Timer > 5*TICK_SECOND)
			{
			// Close the socket so it can be used by other modules
				TCPDisconnect(MySocket);
				MySocket = INVALID_SOCKET;
				SocketState--;
			}
			break;
		}

		SocketState++;
		break;

	case prepare_msg_2:

		// Make certain the socket can be written to
		if(TCPIsPutReady(MySocket) < 125u)
			break;

		strcpy(send_str, "Student ");
		sprintf(send_tmp, "%d", student_id++);
		strcat(send_str,send_tmp);

		SocketState++;
		break;

	case send_msg_3:

		TCPPutROMString(MySocket, (ROM BYTE*)send_str);
		TCPFlush(MySocket);

		LED2_IO = 1;
		DelayMs(20);
		LED2_IO = 0;
		DelayMs(20);
		LED2_IO = 1;
		DelayMs(20);
		LED2_IO = 0;

		SocketState++;
		break;
```

```
        case receive_msg_4:

        // Get count of RX bytes waiting
        TCPIsGetReady(MySocket);
        // Obtian and print the server reply
        recv_str[0] = '\0';
        TCPGetArray(MySocket, recv_str, 9);

        LED1_IO = 1;
        DelayMs(20);
        LED1_IO = 0;
        DelayMs(20);
        LED1_IO = 1;
        DelayMs(20);
        LED1_IO = 0;

        SocketState++;
        break;

case process_msg_5:

        if(recv_str[0] != 'O'){
                SocketState = prepare_msg_2;
                student_id--;
                break;
        }

    if(student_id >= 11u){
                // Make certain the socket can be written to
                if(TCPIsPutReady(MySocket) < 125u)
                        break;
                strcpy(send_tmp, "QQQ");
                TCPPutROMString(MySocket, (ROM BYTE*)send_tmp);
                TCPFlush(MySocket);

                SocketState = disconnect_socket_8;
                break;
        }

        strcpy(LCDText, send_str);
        strcat(LCDText," ");
        strcat(LCDText,recv_str);
        LCDUpdate();

        SocketState++;
        break;

case time_loop_6:

        if(t2<=4){
                if(TickGet() - tim >= TICK_SECOND/2ul)
                {
                    tim = TickGet();
                    t2++;
                }
                SocketState = time_loop_6;
                break;
        }
```

```
                        SocketState++;
                        break;

                case back_prepare_7:
                        t2 = 0;
                        SocketState = prepare_msg_2;
                        break;

                case disconnect_socket_8:

                        if(t2<=4){
                                if(TickGet() - tim >= TICK_SECOND/2ul)
                                {
                                    tim = TickGet();
                                    t2++;
                                }
                                SocketState = disconnect_socket_8;
                                break;
                        }

                        strcpy(LCDText, send_str);
                        strcat(LCDText," ");
                        strcat(LCDText,recv_str);
                        LCDUpdate();

                        // Close the socket so it can be used by other modules
                        TCPDisconnect(MySocket);
                        MySocket = INVALID_SOCKET;
                        SocketState = task_done_9;
                        break;

                case task_done_9:
                        SocketState = task_done_9;
                        break;
        }
}
#endif  //#if defined(STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE)
```