

768734

**Embedded Web Server Based
Home Appliance Networks**

by

Mustafa Can Filibeli

**A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of**

**Master of Science
in
Electrical & Computer Engineering**

**Koç University
January 2005**

Koç University
Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Mustafa Can Filibeli

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

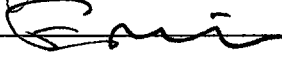
Committee Members:



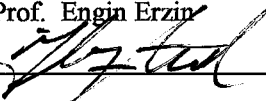
Prof. M. Reha Civanlar (Advisor)



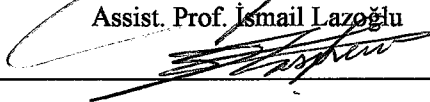
Assist. Prof. Özhur Özkasap (Advisor)



Assist. Prof. Engin Erzin



Assist. Prof. İsmail Lazoğlu



Assist. Prof. Serdar Taşiran

Date:



To Lerzan

ABSTRACT

Powerful microcontrollers are used as parts of most home and office appliances of today. Integrating web servers to these intelligent devices will aid in controlling such devices over the Internet and also creating user interfaces for them in the form of web pages. Assigning multiple functionalities to a single button help manufacturers economize user interfaces but, this makes them more complicated. Since the cost of web-based interfaces is considerably low, they can be used to provide the infrastructure for the design of simple and more user-friendly interfaces for household appliances. Also, a web page based interface is much easier to change, when needed, as compared to a hardware interface. This paper presents a novel approach to control devices over the Internet and to form device networks such that their components can make use of one another's services and functions while improving on user interfaces. The approach is used to create our prototype system, in which the main contributions are its lightweight design, automatic configuration and utilization of widely available network protocols of TCP/IP and HTTP.

ÖZETÇE

Günümüzde elektronik aletlerin çoğu güçlü mikrodenetleyiciler içermektedir. Söz konusu cihazlara web sunucuları ekleyerek, web sayfaları şeklinde kullanıcı arayüzleri oluşturmak hem düşük maliyetli olacaktır hem de İnternet üzerinden cihaz kontrolü sağlanacaktır. Üreticiler maliyetleri düşürmek için kullanıcı arayüzlerinde bir düğmeye birden fazla fonksiyon yüklemektedirler ve bu nedenle arayüzler karmaşıklaşmaktadır. Web sayfaları şeklinde oluşturulan arayüzlerin maliyetleri düşük olduğu için kolay anlaşılır arayüzler oluşturulacaktır. Aynı zamanda, donanımsal kullanıcı arayüzleriyle kıyaslandığında, web sayfası tabanlı arayüzlerde gerektiğinde değişiklik yapmak çok daha kolaydır. Bu tezde yeni bir teknik yaklaşımla akıllı cihazların İnternet üzerinden kontrol edilmesi ve cihazların bir ağ yapısı altında toplanarak birbirlerinin fonksiyonlarından faydalanmaları anlatılmaktadır.

ACKNOWLEDGEMENTS

I would like to express my profound sense of gratitude to my advisors Prof. M. Reha Civanlar and Dr. Öznur Özkasap for their persistent guidance, vigilant supervision they have provided during the entire span of my thesis work. Their patience and kindness, as well as their academic experience, have been invaluable to me. I am proud of being one of their students.

I sincerely acknowledge the environment and the opportunity given by the Faculty of Engineering and the Department of Electrical Engineering at Koç University.

I would like to thank Dr. Serdar Taşiran, Dr. Engin Erzin and Dr. İsmail Lazoğlu for serving on my advisory committee.

From my first days at Koç University, the warm support of all my friends enabled me to complete this thesis and to have a wonderful time along the way. My most sincere appreciation goes to my friends; Ali Selim Aytuna, Umut Küçükkabak and Ferit Ozan Akgül. Thanks to Can James Wetherilt, Tanır Özçelebi, Işıl Yıldırım and Alper Tolga Kocataş who made my life more pleasant at Koç University. I truly believe that all people whom I have not personally mentioned here are aware of my deep appreciation.

Finally, I am deeply indebted to my parents, my mother Eda Filibeli and my father Ali Filibeli, for their life-long love, hearty encouragement and inspiration throughout my life. Without their support and trust, it would be impossible for me to continue pursuing my dreams. After all, this thesis is tangible; hopefully it can represent some portion of the immense gratitude and love I have for them. This thesis is dedicated to them.

TABLE OF CONTENTS

List of Tables	xi
List of Figures	xii
Nomenclature	xiv
Chapter 1: Introduction	1
1.1 Motivation.....	1
1.2 Background.....	3
1.3 Thesis Goal	4
1.4 Thesis Contributions	4
1.5 Thesis Outline	4
Chapter 2: Related Work	6
2.1 Commercial Platforms	7
2.1.1 Jini	7
2.1.1.1 The Jini Infrastructure.....	10
2.1.1.2 Discussion.....	15
2.1.2 Universal Plug and Play (UPnP).....	16
2.1.2.1 The UPnP Infrastructure	17
2.1.2.2 Discussion.....	22
2.1.3 Salutation	23

2.1.3.1	Salutation Manager Tasks.....	24
2.1.3.2	Discussion.....	27
2.1.4	Home Audio/Video Interoperability.....	28
2.1.4.1	The HAVi Architecture.....	28
2.1.4.2	Discussion.....	32
2.1.5	Analysis.....	32
2.2	Relevant Research Architecture.....	35
Chapter 3:	Benefits of Embedding Web Servers to Everyday Appliances	37
3.1	Introduction.....	37
3.2	Advantages of Embedding Web Servers into Everyday Appliances.....	37
3.3	Advantages of Using Web-based Interfaces for Everyday Appliances.....	37
3.4	Summary.....	46
Chapter 4:	Ethernut Based Home Appliance Networks	47
4.1	Architecture.....	48
4.2	The Prototype Implementation	53
Chapter 5:	The Development Platform Selection	57
5.1	Hardware.....	59
5.1.1	The Ethernut Board.....	59
5.1.2	Atmel ATmega128 Microcontroller	63
5.2	Software.....	65
5.2.1	Ethernut Operating System (Nut/OS).....	65
5.2.1.1	Memory Considerations.....	66
5.2.1.1.1	Flash Rom.....	66

5.2.1.1.2	RAM	67
5.2.1.1.3	EEPROM	67
5.2.1.2	System Initialization	68
5.2.1.3	Thread Management	69
5.2.1.4	Event Management	71
5.2.1.5	Stream I/O	72
5.2.1.6	Heap Management	73
5.2.1.7	Timer Management	74
5.2.1.8	File System	74
5.2.2	Ethernet TCP Stack (Nut/Net)	74
5.2.2.1	Network Device Initialization	75
5.2.2.2	Nut/Net Protocols	76
5.2.2.3	Nut/Net Socket Application Programming Interface	78
5.2.2.4	Example: Simple TCP Server Application	80
5.2.2.5	Example: Simple Web Server Application	83
Chapter 6:	Details of the Prototype Implementation	87
6.1	Hardware Details of the Prototype Implementation	87
6.1.1	The Internet Alarm Clock	88
6.1.2	The Internet TV Remote Controller	90
6.2	Software Details of the Prototype Implementation	90
6.2.1	The HAN Server	91
6.2.2	The Address Server	99
6.2.3	The Internet Alarm Clock	100
6.2.4	The Internet TV Remote Controller	103

Chapter 7: Conclusions	106
7.1 Benefits of the HAN Architecture	107
7.2 Drawbacks and Weaknesses of the HAN Architecture	108
7.3 Analysis of the Results.....	109
7.4 Personal Reflections.....	110
7.5 Future Directions	111
 Appendix A: TCP Server Application Code	 112
 Appendix B: Web Server Application Code	 115
 Appendix C: Ethernut Home Server Source Code	 121
 Appendix D: The Internet Alarm Clock Source Code	 144
 Appendix E: The Internet Alarm Clock Web Server Source Code	 166
 Appendix F: The Internet TV Remote Controller Source Code	 198
 Appendix G: The Internet TV Remote Controller Web Server Source Code	 204
 Bibliography	 230
 Vita	 234

LIST OF TABLES

Table 2.1: Summary of major commercial service discovery protocols	33
Table 5.1: Expansion port pin configuration	60
Table 5.2: Serial Port Pin Configuration.....	61
Table 5.3: RJ-45 connector's pin configuration	62
Table 5.4: RAM Address Space Parts.....	67
Table 5.5: EEPROM Contents.....	68
Table 6.1: The LCD connections to Atmel ATMega16 microcontroller.....	89

LIST OF FIGURES

Figure 2.1: Main components of a Jini System	9
Figure 2.2: Jini System Overview	9
Figure 2.3: The Multicast Request Protocol.....	11
Figure 2.4: The Multicast Announcement Protocol	11
Figure 2.5: The Unicast Discovery Protocol.....	12
Figure 2.6: The Service Join Protocol	13
Figure 2.7: Client searching for a Service.....	14
Figure 2.8: UPnP Protocol Stack.....	16
Figure 2.9: Clients find services, and services announce their presence to Clients	18
Figure 2.10: Client performs a standard HTTP GET command to retrieve the description document.....	19
Figure 2.11: Subscription to Event Notification	21
Figure 2.12: The Salutation Architecture Overview	24
Figure 2.13: The HAVi Architecture.....	29
Figure 3.1: Sony SLV-N900 VCR.....	40
Figure 3.2: Sony SLV-N900 VCR's Main Menu listing links to basic/advanced operations	40
Figure 3.3: VCR's web page based wizard for recording programs using the timer	41
Figure 3.4: Aiwa CX-NMT70 Stereo	43
Figure 3.5: AT&T 1825 Telephone/Answering Machine	43
Figure 4.1: Ethernet PCB Version 1.3	47
Figure 4.2: Ethernet based embedded web servers architecture overview	49

Figure 4.3: Bridge.cgi Function	51
Figure 4.4: The prototype system overview	55
Figure 4.5: The Hardware of the Prototype Implementation	55
Figure 4.6: Device Coordination	56
Figure 5.1: Ethernet's expansion port	61
Figure 5.2: Ethernet's serial port.....	61
Figure 5.3: Ethernet's RJ-45 connector	62
Figure 5.4: Atmega128 Pin Configuration	64
Figure 6.1: Atmel ATmega16 Microcontroller Pin Configuration	88
Figure 6.2: Automatic network configuration	94
Figure 6.3: Device List	98
Figure 6.4: Connecting to the EHS via the Address Server	100
Figure 6.5: Setting the clock	101
Figure 6.6: Device Coordination	102
Figure 6.7: Bi-phase encoded bits	103
Figure 6.8: Sending an RC5 code word	105

NOMENCLATURE

UPnP	Universal Plug and Play
HAVi	Home Audio/Video Interoperability
JVM	Java Virtual Machine
RMI	Remote Method Invocation
XML	Extensible Markup Language
SOAP	Simple Object Access Protocol
HTTP	Hypertext Transport Protocol
HTTPU	HTTP over Unicast
HTTPMU	HTTP over Multicast
SSDP	Simple Service Discovery Protocol
URL	Uniform Resource Locator
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IP	Internet Protocol
GENA	Genral Event Notification Architecture
DHCP	Dynamic Host Configuration Protocol
SSL	Secure Socket Layer
FUDR	Functional Unit Description Record
ASN.1	Abstract Syntax Notation
SLM	Salutation Manager
TM	Transport Manager

SMP	Salutation Manager Protocol
RPC	Remote Procedure Call
WWW	World Wide Web
CGI	Common Gateway Interface
PUC	Personal Universal Controller
RTOS	Real Time Operating System
EHS	Ethernut Home Server
PSTN	Public Switched Telephone Network
GSM	Global System for Mobile Communications
GPRS	General Packet Radio Service
PPP	Point-to-Point Protocol
HAN	Home Appliance Network
UHI	Unique Home Identifier
ICMP	Internet Control Message Protocol
MAC	Media Access Control
LCD	Liquid Crystal Display
IR	Infrared
LED	Light Emitting Diode
CPU	Central Processing Unit
DNS	Domain Name Service
ARP	Address Resolution Protocol
EEPROM	Electrically Erasable Programmable Read Only Memory
ROM	Read-Only Memory
RAM	Random Access Memory
ALU	Arithmetic Logic Unit
PAP	PPP Authentication Protocol

DTR	Data Terminal Ready
Hz	Hertz



Chapter 1

INTRODUCTION

1.1 Motivation

Many home and office appliances such as televisions, VCRs, telephones, watches, ovens, toasters, dish-washers and thermostats produced over the last decade are equipped with complex features. This has become possible by the electronic industry's ability to fit more transistors into a smaller area of silicon which increases the computational capabilities of devices while decreasing the cost and the power consumption. Nowadays, most of the home and office devices contain embedded microcontrollers. One of the most important reasons for embedding computers to everyday appliances is that they make the appliance easy to use and give them new features that enhance the value of the appliance [1]. As of today, these embedded computers cost a few dollars and are as capable as a desktop computer of 1980s. Such devices can support an embedded operating system with a TCP/IP stack, interface to a 10/100 Mbps network and can be used as web servers.

As envisioned by Mark Weiser [32], embedded systems will become so ubiquitous in the future that they will be an integral part of our life. In other words, almost every object around us will contain an embedded computer system and interacting with them will be natural for everyone. Moreover, these embedded systems will communicate with each other via wired or wireless communication devices.

The Internet has been mostly used to connect personal computers so far, but shortly all kinds of appliances with embedded computers will exchange information over the Internet. The massive number of microcontrollers available in today's devices can be linked to the Internet. If these intelligent appliances could be connected to the Internet at a low cost, the way we control and manage their functions would change entirely.

As the abilities of appliances increase, the complexity of controlling these devices increases as well. However, the need for informative user interfaces is usually suppressed by the higher cost of implementation. Therefore, complex appliances come with poorly labeled buttons and overloaded functions. For instance, to set the timer of a stereo, a user should press a combination of buttons within a few seconds. Considering an environment with thousands of embedded systems, the scenario that every device has a keyboard and a display will not be an option to ease the interaction with complex appliances. We clearly should come up with a compact solution and this may be achieved through web pages that can be used as user interfaces, giving each appliance the ability to serve its user interface as a web page available over the Internet. These pages can contain not only text but also pictures, hypertexts, photographs, video and audio as in any regular web page. By the advent of the user interfaces based on web pages, all devices requiring user interaction can be controlled and managed from one device which includes a web browser such as a personal digital assistant, cell phone, etc. Also, use of web page based, button and display free designs reduce the cost of production while offering the potential of making the systems more user-friendly.

When ubiquitous embedded systems become real, it will be a challenge to provide a feasible technique for us to interact with these systems. As it was envisioned, there will be an unlimited number of different systems with an unlimited number of features.

The motivation of this thesis is the increasing demand on home networking and the vision about ubiquitous embedded systems creating a challenge to provide a feasible

technique for regular users to interact with them. As it was envisioned, there will be an unlimited number of different systems with an unlimited number of features. Given that people are increasingly carrying computerized devices with the Internet connection and web browsers, by designing low cost embedded web servers for everyday appliances, we propose a feasible technique for everyone to interact with these ubiquitous embedded systems via web pages.

1.2 Background

Remote control via the Internet using web pages is not a new feature and has already been used in home automation systems [14]. However, providing a mechanism for interaction between the devices in this environment is quite challenging. For example, an alarm clock should be able to interact with the heating system according to the alarm time and send instructions to the heating system to start warming the house before the residents wake up.

There exist some commercial platforms, which facilitate the development of applications interacting with and exploiting the abilities of networked intelligent devices, such as Universal Plug and Play (UPnP) [19], Jini [3], Salutation [2], etc. The main problems with these approaches are high processing power needs, platform independency and configuration difficulties. More details about these approaches and their problems are available in Chapter 2.

An attractive alternative to the existing techniques seems to be a web servers based system, where each appliance runs an embedded web server serving the pages needed for its own user interface. This approach is considerably simpler than the existing systems while keeping most of their functionalities as will be demonstrated in this thesis.

1.3 Thesis Goal

The goal of this thesis is to design and demonstrate a system that integrates web servers to everyday appliances to control and manage them via web pages using web browsers over the Internet with wired or wireless devices that can run web browsers. This approach provides a platform for improved user interface design. The system allows forming device networks such that their components can make use of each others' services and functions as well.

1.4 Thesis Contributions

In this thesis, we present a unique architecture for building home appliance networks by integrating embedded servers to everyday appliances. The approach is used to create our prototype system, in which the main contributions are its lightweight design, automatic configuration and utilization of widely available network protocols such as TCP/IP and http, in order to test and demonstrate the characteristics of our design.

1.5 Thesis Outline

Chapter 2 provides background information and literature review on the existing device discovery and coordination platforms. The architectural overviews of commercial platforms (UPnP, Jini, Salutation and HAVi) are given and they are compared with each other. Secondly, a few systems which are the products of academic research are briefly outlined.

Chapter 3 describes the advantages of web based approach for home networking as opposed to other approaches.

The architecture of the proposed system is described in Chapter 4. The development platform selection and the implementation details of the prototype system are given in Chapter 5 and Chapter 6 respectively.

Finally, the thesis is concluded with a short summary of the performed study, analysis of the architecture, conclusions and future directions.



Chapter 2

RED WORK

Letting the home devices communicate, interact and share functions/services with each other requires some sort of device coordination within an appliance network so that devices can register to the network, announce their services and find services offered by other devices. For enabling device coordination, a subset of the following capabilities should be provided to a device [12]:

1. declare its existence to the appliance network
2. discover other appliances in the network automatically
3. describe its services and capabilities to other devices
4. query and understand the capabilities of other devices
5. configure itself automatically
6. inter-operate with other devices wherever meaningful

In this chapter, we will first outline some of the best-known and most promising commercial platforms, which facilitate the development of distributed applications interacting with and exploiting the capabilities of miscellaneous networked devices. Secondly, results of recent academic research on this subject will be discussed.

2.1 Commercial Platforms

The purpose of this section is to provide a general idea of the state of the art of the industrial systems relevant to the work presented in this thesis.

The following platforms are reviewed:

1. Jini[®] by Sun Microsystems[®]
2. Universal Plug and Play[®] (UPnP) by UPnP Forum (3COM, Alcatel Telecom, Analog Devices, AsusTek, Cisco, Compaq, Fujitsu, Microsoft, Sun Microsystems with over 200 other participants),
3. Salutation[®] by Salutation Consortium (Canon, Consumer Electronic Association, Fuji Xerox, Konica Minolta, IBM, Ricoh with over 40 academic and individual members), and
4. Home Audio/Video Interoperability[®] (HAVi): HAVi Organization (Grundig, Hitachi, Panasonic, Sharp, Sony, Thomson, Toshiba with over 30 other participants)

2.1.1 Jini

Jini [3], developed by Sun Microsystems, is a programming model and infrastructure that aims to enable spontaneously networked devices to communicate, interact, and share each others' services. The key idea of Jini is to allow any device with a processor, memory and network connection to provide services to other entities in a network and to use such services offered by others. Jini provides mechanisms to enable devices to plug together to form a *Federation*. *Components* of the federation may be physical devices or software objects written in Java. In Jini terminology, such devices and objects are called *Services*

which can be connected to a network and announce their presence to a *Lookup Service*. Clients that wish to use these Services locate and call them to perform tasks.

The Jini system contains three main types of players which are connected to a network. There is a *Service*, such as a VCR, a printer, a digital camera, a coffee machine, etc., a *Client* that would like to use this Service and a *Lookup Service* that acts as an agent between services and Clients. Basic components of the Jini system are illustrated in Figure 2.1.

The Jini technology uses a Lookup Service with which the Services register. When a Client connects to the network, it goes through a phase, called *discovery* and *join*: the Client (possibly a device) first locates the Lookup Service (see Figure 2(a)) and then uploads a *Service Registrar Object* that it uses to access the Lookup Service (see Figure 2(b)). The Service Registrar Object will serve as a local proxy for the Lookup Service. It implements the Lookup Service's interface and the protocol for communication between the Client and the Lookup Service. If a Client wishes to use a Service, first it locates the Lookup Service. A proxy (Service Object) representing the Service is then uploaded from the Lookup Service to the requesting Client (see Figure 2(c)). The proxy implements the Service's interface and takes care of communication with the remote Service. The Client may now use the remote Service by invoking methods on the local Service Object. Once the connection is established, the Lookup Service is not involved in any of the resulting interactions between the Client and the Service (see Figure 2(d)).

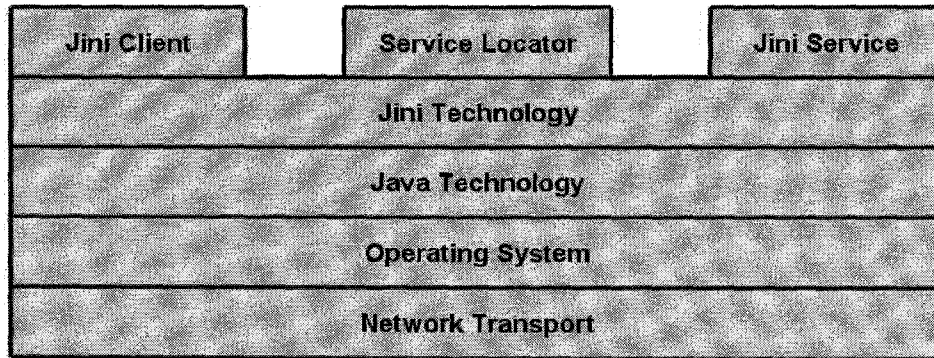


Figure 2.1: Main components of a Jini System

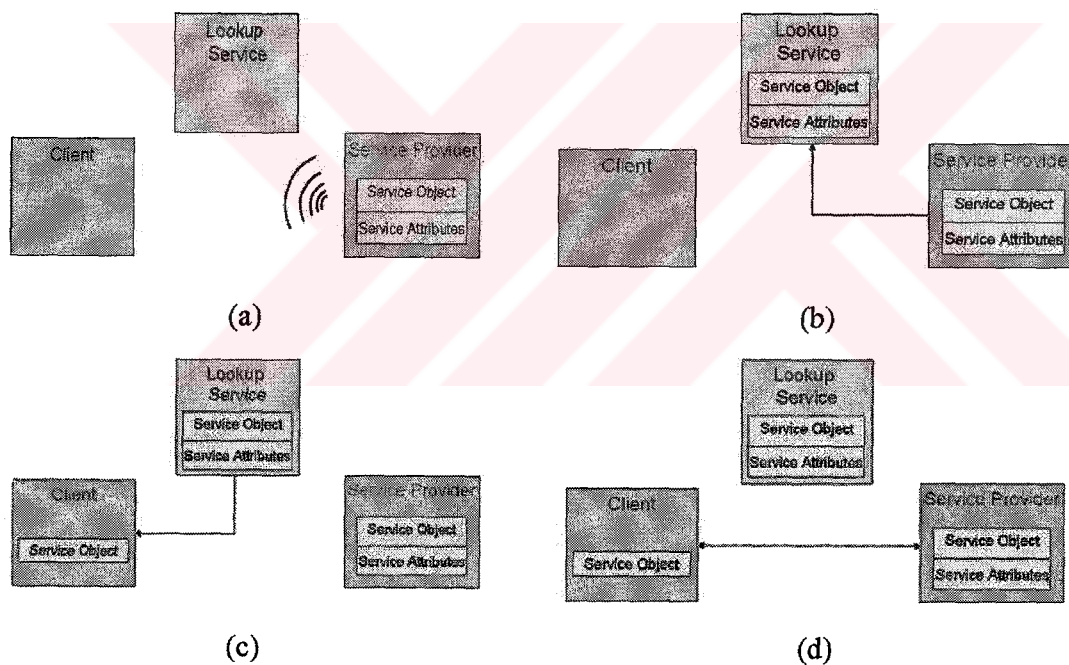


Figure 2.2: Jini System Overview – (a) Service provider seeks a Lookup Service (b) A service provider registers a Service Object and its Service Attributes with the Lookup Service (c) A Client requests a service by Service Attributes. A copy of the Service Object is moved to the Client and used by the Client to talk to the service. (d) The Client interacts directly with the Service Provider via the Service Object.

2.1.1.1 The Jini Infrastructure

The Lookup Service Protocol: A *Client* locates a service by querying a Lookup Service. The first step for the *Client* is to locate such a Service. Moreover, a Service must register itself with the Lookup Service, and, in order to do so, it must also first locate a Service.

Thus, the initial phase of both a *Client* and a *Service* is discovering a *Lookup Service*. Such a service is usually started by some independent mechanism. The search for a Lookup Service can be done either by unicast or multicast. The Lookup Service can be categorized as a specialized Jini Service that stores Services and passes them on to the Clients looking for them.

The *Multicast Request Protocol* is a Service-initiated protocol that is utilized when a Service needs to discover all Lookup Services that may exist on the local network. The process starts when an initiating Service sets itself up for sending multicast requests and receiving unicast replies. This is performed by creating sockets, one for outgoing multicast UDP messages and one for incoming TCP messages. Similarly, the Lookup Service sets up two sockets, one for incoming multicast UDP messages and another one for outgoing TCP messages (see Figure 2.3).

The *Multicast Announcement Protocol* is used by Lookup Services to announce their presence to the network. When a new Lookup Service is connected to a network, it sends out a *Multicast Presence Announcement* message and periodically repeats sending out the same message during its lifetime. The communication flow is illustrated in Figure 2.4.

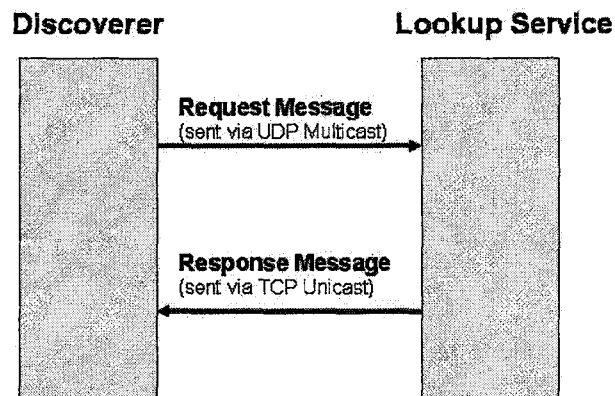


Figure 2.3: The Multicast Request Protocol

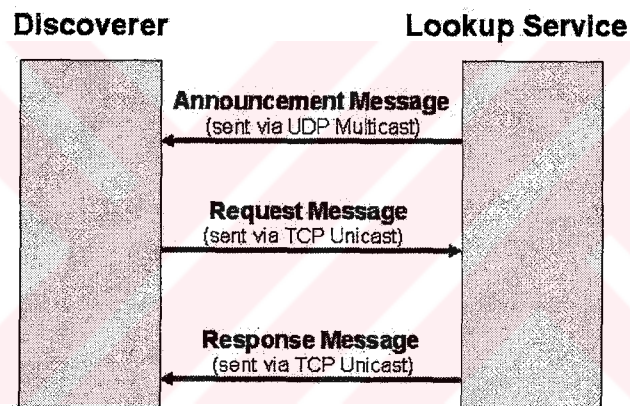


Figure 2.4: The Multicast Announcement Protocol

The *Unicast Discovery Protocol* can be used when the discoverer (a Service or a Client) already knows the location of the Lookup Service, and thus can contact it directly. It starts when the discovering entity sends a unicast request to and waits for a response from the Lookup Service. If the Lookup Service receives the request, it will respond with a Lookup Service Registrar Object.

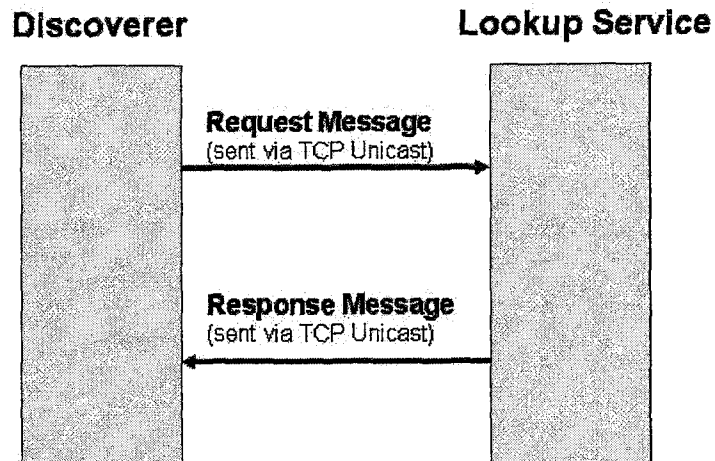


Figure 2.5: The Unicast Discovery Protocol

The discovery mechanism can be demonstrated by the following example:

1. A Jini-enabled alarm clock is plugged into the network.
2. The alarm clock sends out a multicast presence announcement.
3. The Lookup Service receives the alarm clock's announcement packet.
4. The Lookup Service contacts the alarm clock.
5. The alarm clock sends a unicast discovery request packet to the Lookup Service.
6. The Lookup Service sends a Service Registrar object.
7. The alarm clock can register itself (via join) through the Service registrar.

The Service Join Protocol: One of the most important concepts within the Jini system is a Service. A Service is an entity that can be used by a person, a program, or another Service. A Service may be a computation, a communication channel to another user, a physical device, or a user. Members of a Jini system form a federation in order to share access to Services. Services may make use of other Services, and a Client of one Service may itself

be a Service to its Clients. The Services are dynamically added to or withdrawn from a federation at any time according to demand, need, or the changing requirements of the workgroup using it.

In order for a Service to register the Service Object with a Lookup Service, the Service must first locate the Lookup Service. This can be done either by unicast or multicast as described in the Lookup Service protocol. When the Lookup Service gets a request, it sends an object back to the Server. This object, known as a Registrar, acts as a proxy to the Lookup Service, and runs in the Service's Java Virtual Machine. Any request that the Service needs to make use of the Lookup Service is made through this proxy Registrar. Any suitable protocol can be used for that purpose, but in practice the implementations of the Lookup Service will be through Java Remote Method Invocation.

The Service uses Registrar to register itself with the Lookup Service. This is done by taking a copy of the Service Object, and storing it on the Lookup Service as shown in Figure 2.6.

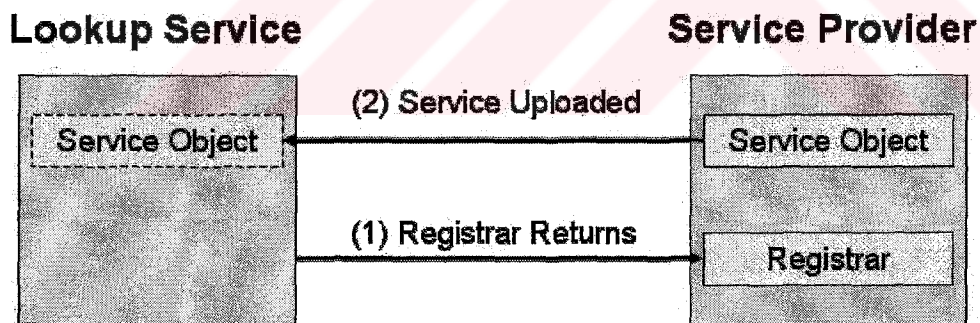


Figure 2.6: The Service Join Protocol

The Client Lookup Protocol: There are a couple of ways Clients can search for Services in a Lookup Service depending on the API used for discovery. The Lookup Service returns either a single Service *proxyObject* or a set of *ServiceItems*, each of them containing a Proxy as well as any attributes associated with the proxy (see Figure 2.7). If a Client has a proxy, it can invoke methods on the proxy to interact with the Service. At this stage, the original Service Object runs on the Service. One copy of the Service Object is stored in the Lookup Service and another copy of the Service Object runs on the Client's Java Virtual Machine (JVM). The Client can now invoke methods on the Service Object running on its own JVM.

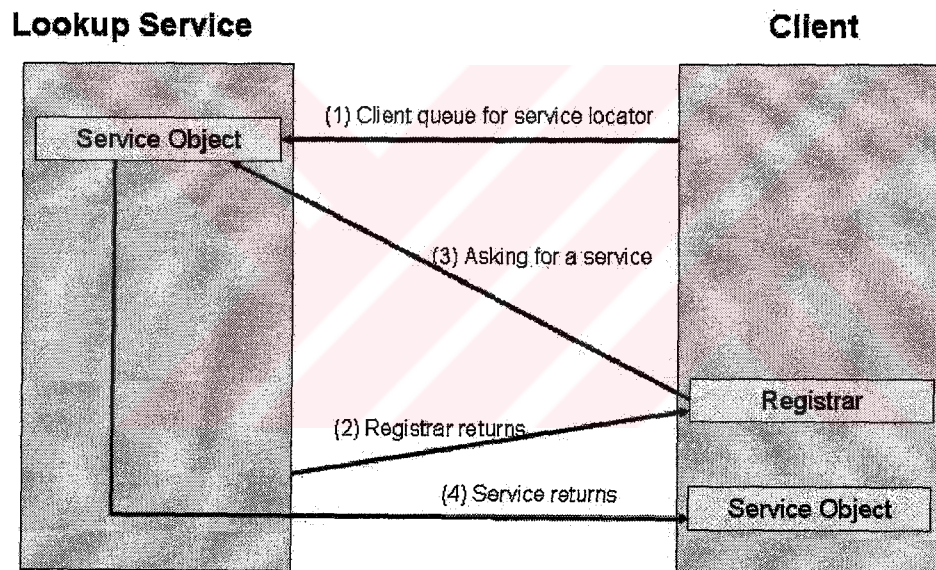


Figure 2.7: Client searching for a Service

Leasing: The Jini architecture discovers the removal of a service by the concept of Leasing. Leasing is used to maintain a connection between Lookup Services and Services/Clients. The Lookup Service grants a limited Service registration lease to other

entity. Before the expiration of the lease the other entity has to renew its registration, otherwise the Lookup Service assumes that the entity has been removed from the network.

Leasing can be used in various situations within the Jini architecture such as resource Leasing and event subscription. A Service may lease its resources to a Client so that it can operate successfully. If the Client wants to continue to use the Service, it has to renew its lease before the due time. Event subscription means that a Client may register a subscription at a Lookup Service through which the Lookup Service informs the Client when a certain Service registers or removes itself.

Moreover, if the network goes down for a short period of time (for a shorter period than the lease time), the structure of the federation is not affected. The resources allocated before the network failure will still be valid afterwards, until the lease time expires.

2.1.1.2 Discussion

Jini is based on Java technology, so it exploits several advantages of Java. It turns a heterogeneous device network into a homogeneous JVM network, which supports the Write-Once-Run-Anywhere capability, operating system independence and dynamic loading of object code, e.g. for integration of new functionalities or automatic bug fix updates [9]. Programmers do not have to worry about the communication details such as the structure of the network and protocols used. Moreover, parts of the Java code can be remotely executed through the Remote Method Invocation (RMI) [5].

The most important disadvantage of the Jini System is that running Java still costs a lot of processing power. If you want to run JVM on limited resources, such as on an 8-bit microcontroller, the only choice has been the reduced subset of Java created specially for the environment. Depending on processor, application, and component set, a typical core code needs a memory space of 40 Kbytes [49]. On the other hand, the Ethernut platform

used in the implementations, houses an Atmel Atmega128 microcontroller, which has 36 Kbytes of memory space in total. Therefore, JVM is not suitable for variety of well known, widely used microcontrollers.

2.1.2 Universal Plug and Play (UPnP)

Universal Plug and Play (UPnP) is an emerging standard endorsed by the UPnP Forum [19] to simplify the network connectivity of intelligent appliances, wireless devices and PCs in a home or business network, or even in the Internet. UPnP is an open and distributed architecture that uses existing protocols and other industry standards such as XML [21] and SOAP (Simple Object Access Protocol) [18], to enable networking as well as control and data transfer in networked devices.

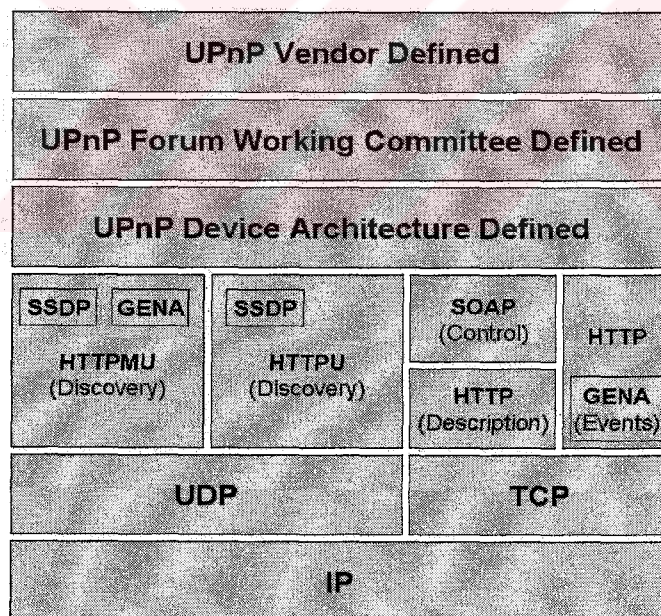


Figure 2.8: UPnP Protocol Stack

The UPnP protocol works on a much lower level than Jini. Instead of working with virtual machines, protocols such as TCP/IP and HTTP that are already implemented on nearly every device were chosen. Other protocols based on HTTP and XML are developed to enable advertisement, discovery and control of components (See Figure 2.8).

2.1.2.1 The UPnP Infrastructure

UPnP has the following five major components:

1. Device Discovery
2. Device Description
3. Presentation
4. Events and Subscriptions
5. Control

Device Discovery: The protocol enables devices to announce their presence to the network as well as discover other available devices in the network. The discovery protocol uses HTTP over unicast UDP (HTTPU) and HTTP over multicast UDP (HTTPMU) [15].

UPnP utilizes the SSDP (Simple Service Discovery Protocol) with HTTPMU that operates over multicast UDP for announcements and HTTPU UDP for replies. The joining device sends out a multicast message to advertise its presence (See Figure 2.9). This advertisement can either be directly seen by other Clients or recorded by Control Points. Control Points correspond to Lookup Services in Jini. Unlike Lookup Services, Control Points are optional components in UPnP.

When a Control Point is added to the network, it sends out a multicast search message, which is answered by unicast replies from the connected devices.

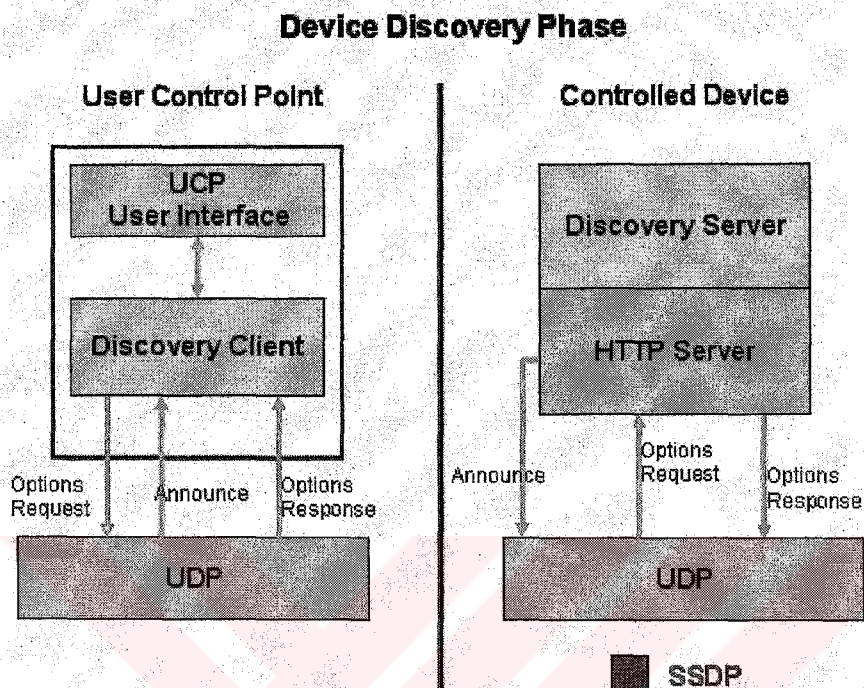


Figure 2.9: Clients find services, and services announce their presence to Clients

Moreover, a Client multicasts search messages; devices that match the criteria specified by the Client respond with a URL [16] that links to a device description document. If the Client receives one or more replies, the description phase is initiated.

If the Client does not multicast any requests, Servers can announce their device availability with a NOTIFY message. Clients who receive the NOTIFY message can then request the description document. Furthermore, Servers can also use a NOTIFY message to announce that their services are no longer available.

Device Description: A device may contain a set of services corresponding to each functional component of the device. UPnP keeps description of these services in XML files. In the device discovery phase, minimum information about the capabilities of that particular device is transferred with the discovery messages. In the description phase, control point learns more about the device by retrieving the XML file from the URL provided by the device discovery message. The XML file containing the description is requested via HTTP over TCP. The Client performs a standard HTTP GET command and the description is transferred to the Client (See Figure 2.10).

The description contains a list of actions which can be used to access the service, and variables expressing the state of the service. Any control point can subscribe to receive update messages when these variables are changed.

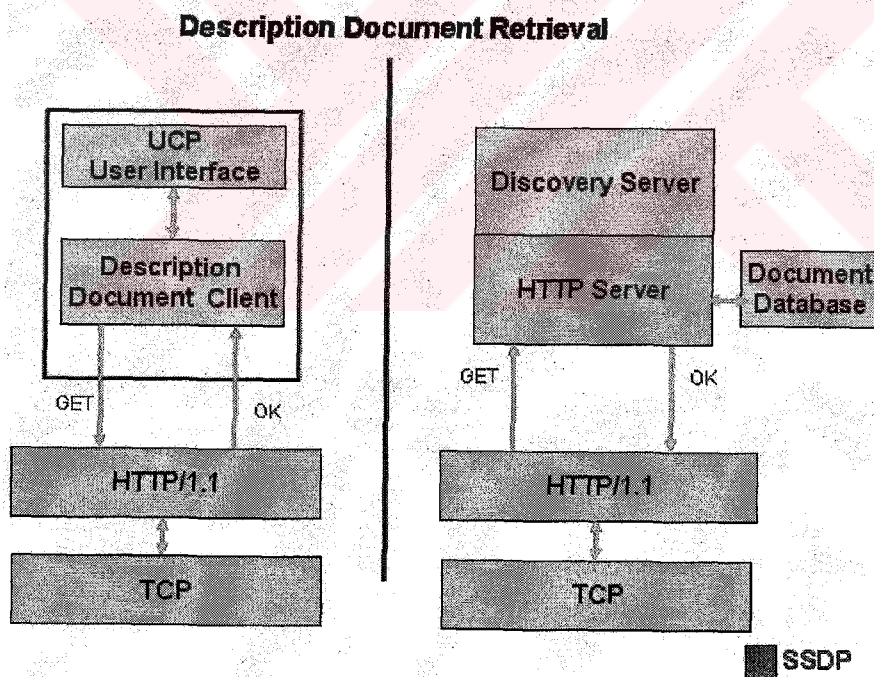


Figure 2.10: Client performs a standard HTTP GET command to retrieve the description document.

The UPnP description for a device can be divided into two logical parts: a device description which contains a description of physical and logical containers, and one or more service descriptions which describe the capabilities of the device.

The device description contains vendor-specific information, definitions of all embedded devices, URL for presentation of the device, and a listing of all services including URLs for control and eventing [17]. On the other hand, the service description defines actions, their arguments, state variables and their data type, range, and event characteristics [17].

Presentation: If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser and depending on the capabilities provided, a user can control the device and/or view its status.

The protocol for retrieving the presentation is the same as the one that retrieves the description document, HTTP over TCP. Not all devices have a presentation and not all Clients are able to display a presentation document.

Events and Subscriptions: Once a Client has discovered a device and retrieved the description and presentation, it may need to be notified of status changes in the device. The description document contains a URL for subscribing to event notification. Event notifications are sent to Client's event sink URL anytime the status of the device changes. In order to subscribe/unsubscribe to event notifications, a SUBSCRIBE/UNSUBSCRIBE request is issued to the subscription URL from the description document via HTTP over TCP. The Client specifies an event sink URL where event notifications are returned via HTTP over TCP. On the Server side, an event Server waits for subscribe/unsubscribe requests through the General Event Notification Architecture (GENA) [11] (See Figure 2.11).

In order to keep a subscription active, a Client must renew its subscription before the subscription expires by sending a renewal message. The renewal message is sent to the same URL as the subscription message, but it does not include a delivery URL for event messages. If a subscription expires, the subscription identifier becomes invalid; the notification sender stops sending messages to the Client. When a Client no longer needs notification messages from a particular device/service, the Client should cancel its subscription. Canceling a subscription reduces service, control point and network load.

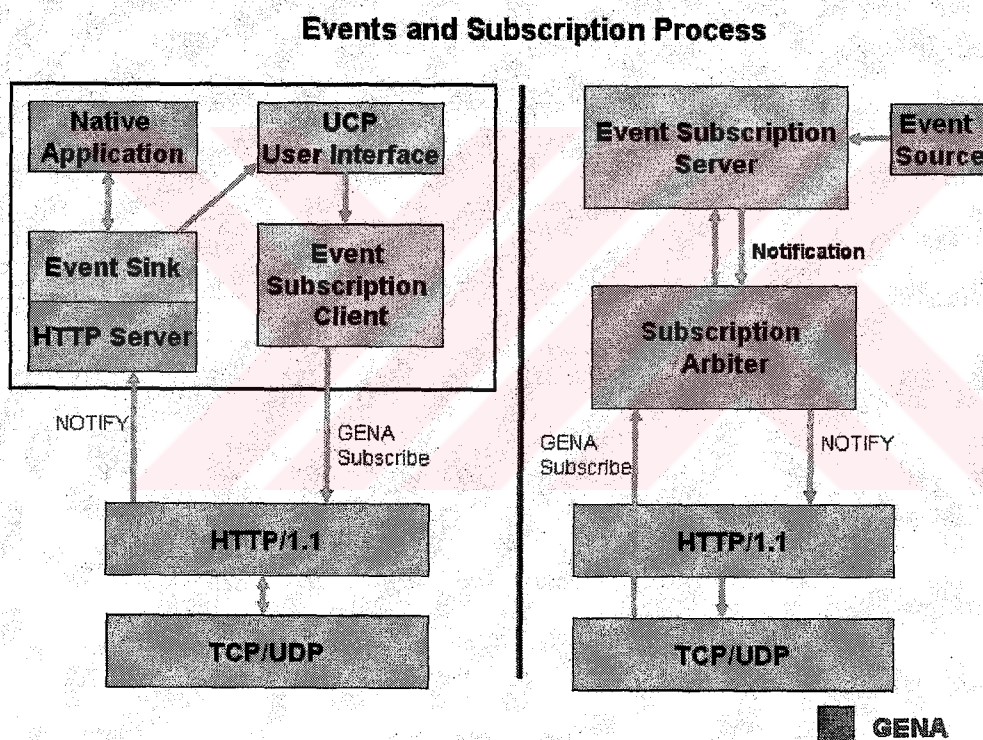


Figure 2.11: Subscription to Event Notification

Control: After a Client has discovered a useful device and retrieved its description, Client may control its services using the SOAP. To control a device, a Client invokes an action on the device's service by sending a suitable control message to the control URL for the service. In response, the service returns any results or errors back to the Client.

SOAP defines the use of XML and HTTP for remote procedure calls. UPnP uses SOAP to deliver control messages to devices and return results back to the Clients. Actions, results and errors are encapsulated in SOAP and transmitted via HTTP.

2.1.2.2 Discussion

One of the most important features of UPnP is automatic configuration of IP addresses. AutoIP [20] lets a device join the network without any explicit administration [10]. When a device connects to the network, it tries to get an IP address from a Dynamic Host Configuration Protocol (DHCP) [22] Server. In the absence of a DHCP Server, AutoIP assigns an IP address to the device from a reserved range automatically. AutoIP chooses an IP address from the reserved range and makes an ARP request to see if anyone else has already claimed that address. And, if it finds another device using the same IP address, it chooses another IP address from the reserved range and checks whether the IP address is used or not.

UPnP is a service discovery solution for small to medium size IP networks [4]. Thus, its use for home networking is ideal, especially with the AutoIP mechanism which does not require any administration.

UPnP does not specify any security mechanism for access and control of UPnP devices. Security has to be provided by the applications or services. Security standards on lower networking layers such as SSL may be used to gain security. The security features of this architecture have not been developed yet.

UPnP approaches the problem of service discovery quite well. On the other hand, it still needs time to have a complete and standardized device connection possibility [9].

2.1.3 Salutation

Salutation, developed by the Salutation Consortium, is cooperation architecture to solve the problems of service discovery and utilization among diverse appliances and equipment. The Salutation architecture provides a standard method for devices to describe and advertise their capabilities to possible Clients. It also enables devices to search other applications, services and devices for a particular capability, and to request and establish sessions with them to utilize the capability [23].

The Salutation architecture comprises services and Clients. A service is a generic term describing a function or a resource provided by a Server [24]. The service capabilities are defined by Functional Unit Description Record (FUDR) [23], which is specified by using Abstract Syntax Notation 1 (ASN.1) [26]. The FUDR's for different services are specified in the Salutation specification.

The Salutation architecture is composed of two major components: *Salutation Manager* (SLM) and *Transport Manager* (TM) (see Figure 2.12). To join a network, a service registers itself with the local or nearest SLM which keeps records about all services and Clients registered with it. The SLM, similar to the lookup service in Jini, is defined as a service broker for applications, services and devices: "A service provider registers its capability with an SLM. When a Client asks its local SLM for a service search, the search is performed by coordination among SLMs. Then, the Client can use the returned service."

The Transport Managers offer reliable communication channels, regardless of the underlying network technology. An SLM is attached to one TM for each infrastructure it is connected to (see Figure 2.12). The SLM provides a transport-independent interface to

Server and Client applications. The interface (SLM-API) includes service registration, discovery and access functions. The communication protocol independence is achieved by the interface between SLM and TM (SLM-TI).

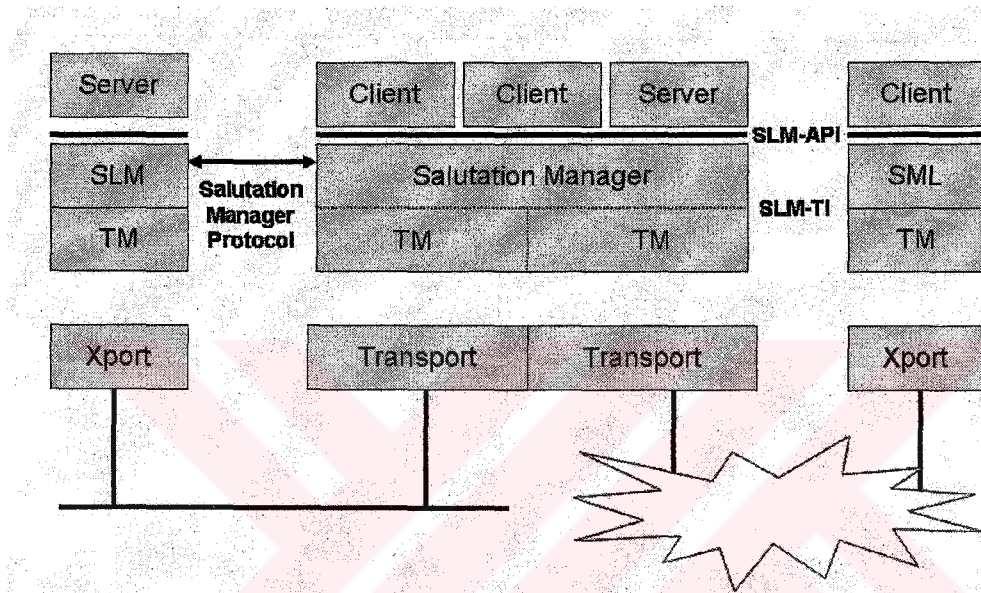


Figure 2.12: The Salutation Architecture Overview

When a Client requests a service which is not locally available, the SLM communicates with other SLMs using Salutation Manager Protocol (SMP) to perform its role (see Figure 2.12).

2.1.3.1 Salutation Manager Tasks

The Salutation Manager provides 4 basic tasks:

- Service Registry
- Service Discovery
- Service Availability
- Service Session Management

Service Registry: The Salutation Manager contains a registry to keep information about Services. Minimally, Registry only stores information about Services connected to the Salutation Manager. Optionally, the Registry may store information about Services that are registered in other Salutation Managers.

Service Discovery: The Salutation Manager can discover other Salutation Managers and find out the services registered there. Service Discovery is performed by comparing the service specifications provided by the local Salutation Manager with the service specifications available on a remote Salutation Manager. The communication protocol between Salutation Managers is called the Salutation Manager Protocol, and Remote Procedure Calls (RPC) are used to transmit the required service information. Through the information manipulation, the Salutation Manager can determine:

- The characteristics of all the Services registered at a remote Salutation Manager.
- The characteristics of a specific Service registered at a remote Salutation Manager.
- The presence of a Service matching a defined set of characteristics on a remote Salutation Manager.

Service Availability: A Client can ask its local Salutation Manager to periodically check the availability of services. The local Salutation Manager requests the appropriate

Salutation Manager to perform an Availability Check. The Availability Check is performed by exchanging RPC messages between Salutation Managers.

Service Session Management: The Service Session Management addresses the service invocation of Salutation architecture. A Service Session is started when a Client wants to use a service which is discovered through a Service Discovery. Messages between Clients and Servers have defined formats and are exchanged under a defined protocol. Such definitions of message exchange format and protocol are called *Personality Protocols*. The Salutation Manager may be instructed to operate one of three different Personality Protocols and it may or may not be included in the message exchange. In the *native mode*, the Salutation Manager is never involved in the message exchange. In the *emulated mode*, the Salutation Manager carries messages between Client and Server but it doesn't inspect the message contents. In the *salutation mode*, Salutation Manager carries messages between Clients and Servers and also it defines the message formats to be used in the session.

Networked Entities may be divided into meaningful functionality called *Functional Units*. A Functional Unit is the minimal meaningful function to form a Client or a Server. A collection of Functional Units defines a *Service Record*. For instance, a fax service can be defined by Print, Scan, and Send Fax Functional Units. Each Functional Unit contains a descriptive attribute record specified with ASN.1.

Salutation-Lite [6], which is a scaled down version of the Salutation architecture, is also worth to mention here. Salutation-Lite targets devices with small footprints. It is applicable to small information appliances such as palm-size and hand-held computers. Moreover, it is suitable for low-bandwidth networks such as IR and Bluetooth.

2.1.3.2 Discussion

“Salutation is shipping while others are still thinking!” claims the Salutation Consortium in [14] while comparing its solution to Jini and UPnP. Products equipped with Salutation are available in the market since 1997. Six member companies (Canon, Fuji, IBM Japan, Mita, Muratec and Ricoh) demonstrated some Salutation-enabled products in 1997. These products support a company intranet that automatically detects new peripherals as they get attached and makes their capabilities available for use. The new products include both hardware and software applications. Demonstrated products are digital copiers, fax machines, scanners and printers. Many of them integrate with Lotus Notes in a corporate intranet environment [2].

Salutation is nonproprietary and specifications are available for third-party development. Salutation is also a relatively lightweight platform, which is a significant benefit for current mobile phones and PDAs.

Salutation is not suitable for dynamic and large-scale networks. Currently, Salutation specification is being modified to better suit large-scale networks.

Unlike UPnP, Salutation does not address the automatic configuration because of its transport independent design. Moreover, it would not be easy to propose a solution to this problem that worked on all transport types. Thus, its use for home networking is not easy, since it needs administration for configuration. Moreover, Salutation does not specify any security mechanism for access and control of networked devices. Therefore, security mechanism has to be provided by the applications or services.

2.1.4 Home Audio/Video Interoperability (HAVi)

Major consumer electronics, semiconductor, computer and software manufacturers, namely Grundig, Hitachi, Panasonic, Sharp, Sony, Thomson and Toshiba with over 30 other participants have formed a non-profit organization called Home Audio/Video Interoperability (HAVi) for supporting the development of interoperable consumer products. The goal of the organization is to provide an open architecture standard for audio/video devices to interoperate with each other manufacturer, operating system and programming language independently.

2.1.4.1 The HAVi Architecture

The HAVi architecture can be divided into several layers (see Figure 2.13). The architecture provides a set of software elements, protocols and APIs that are needed for interoperability. The software elements of the HAVi architecture support the basic notion of network management, device abstraction, inter-device communication and user interface management. The software elements reside above a vendor specific platform.

The networking software specifies the protocols to be used by the components. These are communication mechanisms, which provide A/V streams, registries and the share of resources. In the HAVi system, addressed product range is so narrow. Therefore, it is easier to develop and implement special services and to meet the particular demands of audio and video.

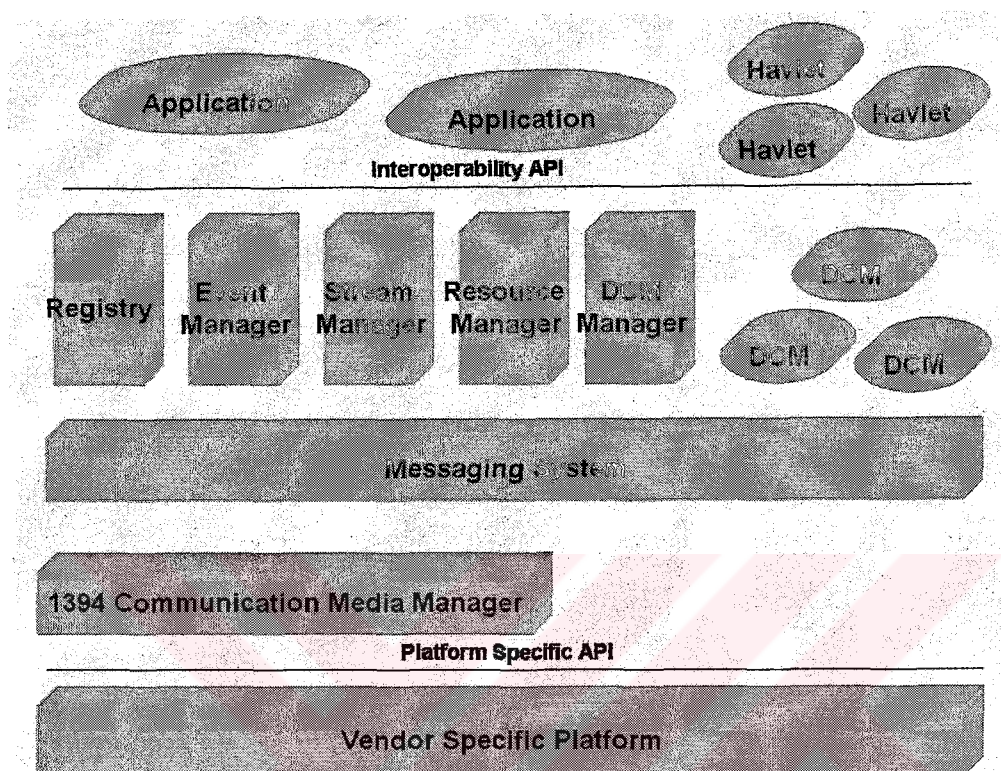


Figure 2.13: The HAVi Architecture

HAVi classifies devices into 4 categories: Full AV devices (FAV), Intermediate AV devices (IAV), Base AV devices (BAV), and Legacy AV devices (LAV). A FAV contains a complete set of software elements comprising the HAVi architecture. The primary feature of a FAV is the presence of a Java Runtime Environment for Java bytecode. An IAV does not provide a runtime environment for Java bytecode. Therefore, IAV devices are generally lower in cost than FAV devices. On the other hand, an IAV may provide support for control of devices on the home network. BAV devices do not host any of the software elements of the architecture. They only contain the base equipment. Finally, an

LAV is a device which is designed for supporting the architecture. However, it can still be integrated into a HAVi network.

Each device in the network is represented by a *Device Control Module* (DCM) as a software element. Device Control Module is composed of *Functional Component Modules* (FCM) which is similar to the functional unit of Salutation. Functional Component Modules represent controllable functions of devices. The functions of devices are accessed through the Device Control Modules.

Each device except Legacy A/V devices has Self Descriptive Device (SDD) Data keeping descriptive information about the device and the capabilities.

It is not possible to build a network with only BAV and LAV devices. The FAV and IAV contain some services which handle the communication in the HAVi network. The software elements compromising the HAVi architecture are:

Messaging System: The system has two functions: Firstly, it defines unique *Software Element Identifiers* (SEID) used to identify the elements in the network, for each object before it is registered to the network. Secondly, it performs the exchange of information between devices. Messaging System is independent of the network and transport layers. A messaging system is embedded in FAV and IAV devices.

Registry System: The purpose of the Registry System is to manage a directory of software elements available in the network. It provides an interface to register and search for software elements. Software elements register their services to the Registry System. The system keeps track of the element's SEID and its attributes. Clients use the query interface of the system to find a service matching a set of criteria.

Communication Media Manager: The Communication Media Manager (CMM) is a network dependent entity which interfaces with the underlying communication media for providing the transport mechanism to send requests to and receive indications from remote devices. Each physical medium has its own CMM. Another service provided by the CMM is to abstract the network activities and present information to the HAVi system. Moreover, it informs the Event Manager about network changes.

Event Manager: An event is the state change of a software element or the network. The Event Manager is a directory service where software elements can register to receive notifications about events.

Stream Manager: The manager provides an interface for configuring end-to-end streaming connections which may be point-to-point or utilize broadcast sources or sinks. The responsibilities of the Stream Manager can be listed as:

- Configuration of internal (within a device) and external connections (between devices).
- Managing requests for streams and the allocation/release of system resources such as bandwidth and channel numbers
- Providing global connection information
- Supports connection restoration after network failures.

Resource Manager: The Resource Manager serves to guide software elements (Clients) competing for and using the set of resources in the network. The resource management system serves to ensure that Clients that have reserved resources can rely on not being disturbed by other applications that use these resources.

There is a Resource Manager on each FAV and IAV device that hosts or can host at least one DCM.

In the HAVi system, it is possible to run applications, supported by the runtime environment. Therefore, HAVi can support user interfaces on remote display devices. The Data Driven Interaction (DDI) is used to connect the component with the user interface and the display device. If the display is Java-enabled, it may extract and run a Havlet, which is a Java application, from the DCM of the other component. Therefore, the display allows the user to control the remote component using the graphical user interface of the Havlet.

2.1.4.2 Discussion

HAVi system is programming language and operating system independent. On the other hand, the system is closely tied to the IEEE 1394 serial bus. Thus, it is not clear whether or not the system can be easily modified to operate on top of other network technologies.

2.1.5 Analysis

The Device Discovery architectures described above approach the problem of service discovery very well. Jini and UPnP are both flexible and scalable. Although Salutation is not suitable for dynamic and large-scale networks, it addresses mobile devices successfully because of its lightweight design. Moreover, it is the only platform used in commercial products in today's market.

Jini is based on dynamically downloadable Java code which is used to tie various devices together, but this structure makes Jini to employ Java technology in the development. On the other hand, UPnP is based on commonly used protocols and standards

which can be implemented easily on every device using any technology. Moreover, UPnP is the only solution that supports automatic configuration of devices using either DHCP or AutoIP.

Salutation is well defined but confined to the service discovery protocol and session management. It can operate in any network, including IP, IR and wireless. This transport independence is the strongest feature of Salutation. Salutation doesn't address features like remote event notifications.

HAVi is designed to enable interoperability only in home Audio/Video networks and supports multimedia streaming and service reservation [7].

If these frameworks are analyzed by comparing the features each one provides, Jini is the most sophisticated, providing device discovery, device lookup, leasing, service invocation, and distributed events. UPnP targets a lower level by supporting device discovery and network configuration. Salutation and HAVi target a middle level, enabling discovery and service invocation.

A number of research groups are working on the design and implementation of home networking systems using middleware technologies described above [27, 28]. However, none of the above four has wide scale deployment. UPnP's self-configuration feature is an essential mechanism that should be implemented on other platforms. Many more changes and additions seem to be needed before having a complete and standardized device connection platform that is commercially usable.

Table 2.1 summarizes the features of major service discovery protocols.

Table 2.1: Summary of major commercial service discovery protocols

	Jini	UPnP	Salutation	HAVi
Home Page	www.sun.com/jini/	www.upnp.org	www.salutation.org	www.havi.org
Platform Neutral	Partially	Yes	Yes	Partially

Dynamism	Good	Good	Moderate	Good
Target Scale	Internet Scale	Internet Scale	Small Scale	Home Networks
Main Entities	Lookup Service, Client, Service	Control Point, Services	Salutation Manager, Transport Manager, Client, Server	Registry, Stream Mgr, Resource Mgr, DCM Mgr, Server, Client, Messaging System, Comm. Media Manager
Service Repository	Lookup Service	No	Salutation Managers	Registry
Service Announcement	Discovery/Join Protocol	Advertisement	Registering with local Salutation Manager	Service Registration
Service Discovery	Query to Lookup Service	Contact control point, or listen to advertisement	Query to local Salutation Manager and cooperation among Salutation Managers	Contact registry
Access to Service	Service proxy object based on RMI	Invoking action to the service using SOAP	Service Session Management	Using Device Control Modules
Event Notification	Remote Events	Service publishes event (GENA) when state variable changes	Availability checking	Event Manager
Others	Java-centric architecture	Automatic configuration	Transport independence	Closely tied to IEEE 1394 serial bus

2.2 Relevant Research Activities

A number of research groups are working on controlling appliances from handheld devices.

The Stanford ICrafter [33] is a service framework for ubiquitous computing environments known as interactive workspaces which are spaces of interconnected computers, utility devices such as scanners and printers and Input/Output devices such as displays, microphones, keyboards, speakers, where people gather information to do collaborative activities. ICrafter allows users of interactive workspaces to interact with services which are devices such as projectors, scanners or applications such as web browsers that provides useful functions to end-users. ICrafter is a framework for distributing and composing appliance interfaces for many different controlling devices. ICrafter places intelligence in the infrastructure to select, generate, and/or adapt user interfaces which are portable across workspaces but still reflect the context of the current workspace.

ICrafter relies on an interface manager to distribute user interfaces to handheld devices. It sometimes automatically generates interfaces and other times it distributes interfaces that are already available.

The most interesting feature of ICrafter is its ability to aggregate appliances together and provide a single interface for these appliances.

The XWeb project [38] aims to separate functionality of the appliance from the device upon which it is displayed. XWeb defines an XML from which user interfaces can be created. XWeb is based on the architecture of the WWW with additional mechanism for collaboration and interaction. XWeb also supports speech interface generation.

The Personal Universal Controller (PUC) [30] is another framework for improving interfaces to complex devices by introducing an intermediary interface. A PUC engages in

two-way communication with everyday appliances, first downloading a description of the appliance's functions, and then automatically creating an interface for controlling the appliance.

The research activities mentioned above focus on automatic creation of user interfaces rather than the system and infrastructure issues. These frameworks separate the appliance from the interface. Interfaces are generated using specification languages depending on the type of the control device.



Chapter 4

ETHERNUT BASED HOME APPLIANCE NETWORKS

Device networks based on Jini, UPnP, HAVi and Salutation platforms need specialized servers to be accessed over the Internet. On the other hand, we can connect our home appliance networks together and enable access to appliances over the Internet using the Ethernut [34] which is an Open Source Hardware and Software Project for building Embedded Ethernet Devices. The hardware design of the Ethernut includes a small board (see Figure 4.1) with ATmega128 CPU and a Realtek RTL8019AS Ethernet Controller. The software part of the Ethernut project includes an Open Source Real Time Operating System (RTOS) and a TCP/IP protocol suite.

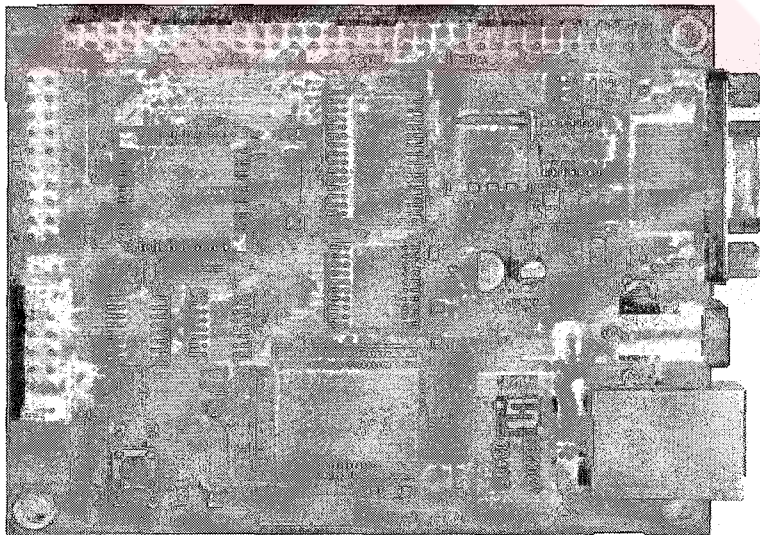


Figure 4.1: Ethernut PCB Version 1.3

In this chapter, the architecture of our implementation of the Home Appliance Network (HAN) is introduced together with the details of the prototype system.

4.1 Architecture

The Home Appliance Network connects to the Internet through the *Ethernet Home Server* (EHS) (see Figure 4.2). The Internet connection of the EHS can be established either via an analog modem over PSTN or via a GSM modem over GPRS using the Point-to-Point Protocol (PPP) [35]. Appliances within a HAN can connect to the network either by wired (10/100baseT) or wireless (IEEE 802.11) network technologies.

After connecting to the Internet, the EHS sends its IP address, *Unique Home Identifier* (UHI) and a password to an *Address Server*. The address server is a networked resource for the HAN. It keeps UHI, IP address and the password for a certain period unless it is renewed. This facilitates robust behavior in the case of Internet connection failures and IP address changes, as the Address Server will not keep the data for those HANs that are no longer available. Moreover, the Address Server prevents unauthorized access to the HANs. An Address Server with a static IP is necessary in this architecture since in every PPP connection EHS may get a different IP address which must be known by the user in order to reach the correct HAN over the Internet.

When an appliance is plugged into the network, it obtains its IP address, subnet mask and gateway address from a DHCP server if it exists in the network. If no DHCP server is available, the appliance randomly chooses an IP address from a reserved range of IP addresses and then sends out an ICMP “echo request” message [40] to detect whether the IP chosen address is used by another device. If the address is used by another device, self-configuration procedure is started over with a new IP address chosen randomly from the specified range.

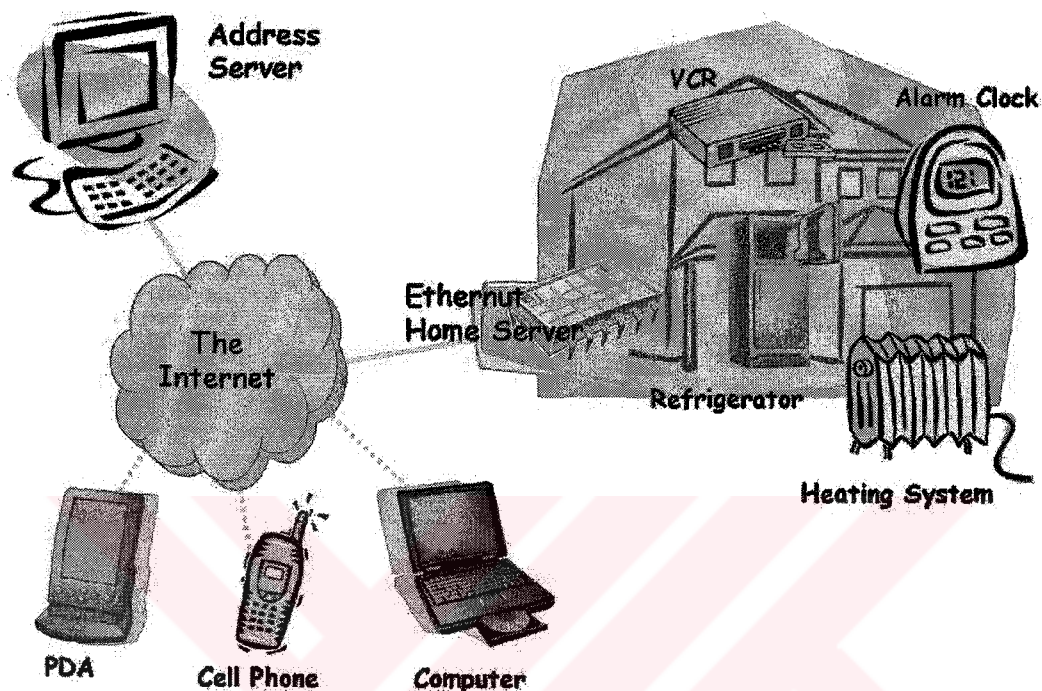


Figure 4.2: Ethernet based embedded web servers architecture overview

After self-configuration finishes, the appliance locates the EHS by broadcast unless it knows the EHS's IP address. If the EHS's address is known by the appliance, unicast can be used to send the information. The appliance sends its identifier, IP and MAC address to the EHS. The identifier contains a descriptive name of the appliance, understandable by a user, such as "The Internet Clock". If an EHS is not available in the network, appliance starts serving its web pages within the network and waits for an *EHS online announcement*. When an EHS becomes online in the network, it broadcasts an online announcement, so that appliances that are already in the network register themselves to the EHS.

Each appliance within a HAN includes a web server that keeps web pages to control and monitor its functions and status. Services of an appliance can be executed using the hyperlinks to CGI functions on this web server. A service request to a device within the HAN coming via the Internet is routed to the target device by the EHS.

After the EHS gets the information from an appliance, it uses the identifier and the IP address to construct a hyperlink on its main web page. The address contains a link to a CGI function on the EHS with parameters: an IP address, a MAC address, an Identifier and a URL which points to the root of the appliance's web server. This data embedding into the web page is needed because one EHS is used to control several appliances and the web servers are stateless. When a user clicks on one of the hyperlinks in the main page of the EHS, a CGI function running on the EHS opens a TCP socket to the target appliance and receives the HTML description of the appliance's main web page through this socket. Figure 4.3(a) demonstrates a sample web page for an appliance. The source code of the web page is given below:

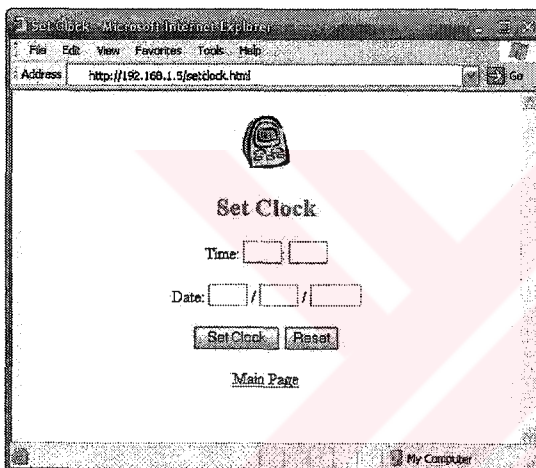
```
<html>
<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Set Clock</title>
</head>
<body>
<p align="center">
  </p>
<p align="center">
  <b><font color="#FF0000" size="5">Set Clock</font></b>
</p>

<form method="GET" action="/cgi-bin/setclock.cgi">
  <p align="center">
    Time: <input type="text" name="T1" size="2">:
         <input type="text" name="T2" size="2">
  </p>
  <p align="center">
    Date: <input type="text" name="T6" size="2"> /
         <input type="text" name="T7" size="2"> /
```

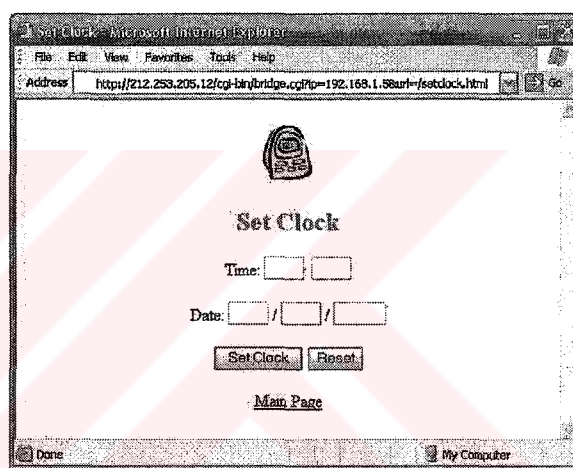
```

        <input type="text" name="T5" size="4">
    </p>
    <p align="center">
        <input type="submit" value="Set Clock" name="B1">
        <input type="reset" value="Reset" name="B2">
    </p>
</form>
<p align="center">
<a href="/index.html">Main Page</a>
</p>
</body>
</html>

```



(a)



(b)

Figure 4.3: Bridge.cgi Function - (a) Browser view of the original clock setting page. (b) Browser view of the modified clock setting page

Before sending the web page to the requester, the CGI function modifies the source of the web page such that every hyperlink, form target and image points to the same CGI function (bridge.cgi) on the EHS (the IP Address of the EHS assigned by the PPP connection is 212.253.205.12) with a different URL parameter containing the original link

on the appliance's web server. The modified code of the sample web page is given below and the modified web page is given in Figure 4.3(b):

```
<html>
<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Set Clock</title>
</head>
<body>
<p align="center">
  </p>
<p align="center">
  <b><font color="#FF0000" size="5">Set Clock</font></b>
</p>
<form method="GET" action="/cgi-bin/bridge.cgi">
  <input type="hidden" name="ip" value="192.168.1.5">
  <input type="hidden" name="url" value="setclock.cgi">
  <p align="center">
    Time: <input type="text" name="T1" size="2">:
          <input type="text" name="T2" size="2">
  </p>
  <p align="center">
    Date: <input type="text" name="T6" size="2"> /
          <input type="text" name="T7" size="2"> /
          <input type="text" name="T5" size="4">
  </p>
  <p align="center">
    <input type="submit" value="Set Clock" name="B1">
    <input type="reset" value="Reset" name="B2">
  </p>
</form>
  <p align="center">
    <a href="http://212.253.205.12/cgi-
    bin/bridge.cgi?ip=192.168.1.5&url=/index.html">Main Page</a>
  </p>
</body>
</html>
```

If the requester clicks one of these hyperlinks, the CGI function opens a TCP socket and performs actions on the target device (the IP address of the target device is 192.168.1.5)

according to the URL parameters and retrieves the resulting web page. Once more, it modifies the web page so that later requests to the appliance are directed over the EHS.

A hyperlink pointing to an appliance exists on the main web page of the EHS for a certain time unless the information is renewed. This leasing mechanism is crucial for the robust behavior of the system in case of appliance connection failures.

The architecture also supports device coordination. As I mentioned before, an alarm clock may interact with the heating system according to the alarm time and send instructions to the heating system. An appliance sends instructions to another appliance by requesting the CGI function of the pre-specified service (for instance: start warming the house) with parameters.

The architecture contains two single-points of failure: the Address Server and the EHS. In order to avoid these, a Secondary Address Server and a Secondary Ethernet Home Server can easily be integrated to the system.

4.2 The Prototype Implementation

We have created a prototype for demonstrating and testing our design. Figure 4.4 illustrates an overview of the prototype system. We implemented an Internet Alarm Clock (Figure 4.5a) and an Internet TV Remote Controller (Figure 4.5b) to be used as web embedded appliance samples controlled within our prototype.

The Internet alarm clock is implemented as a small board which houses an Atmel ATMega16 microprocessor and an LCD to display the time and date. The clock is designed without a dedicated user interface (without buttons for setting the date and time). Instead, an Ethernet board is connected to the clock. The Ethernet board contains a web server that serves web pages that constituting the user interface of the clock.

The Internet TV Remote Controller consists of another board containing an Atmel ATmega16 microcontroller and an IR LED transmitter. Unlike the conventional remote controllers, this Internet TV Remote Controller has no buttons on it! Users interact with it via web pages that are served by an Ethernet board which is connected the remote controller.

Two appliances are connected to an Ethernet via Ethernet boards. Automatic configuration property of the system was tested on the two networks, one of them housing a DHCP server. We employ an analog modem and it is connected to the EHS via RS232 (Figure 4.5c). EHS connects to the Internet using the Point-to-Point Protocol. After the connection, EHS registers itself to the Address Server with its IP address.

The alarm clock and the remote controller register themselves to the EHS as soon as they are plugged into the Ethernet.

Users can get the IP address of the EHS from the Address Server. When a user connects to the EHS using a web browser, a webpage showing the hyperlinks of the online appliances (in our prototype, the TV remote controller and the alarm clock) are displayed as shown in Figure 4.6a. Via this structure, the user can set the alarm of the clock (Figure 4.6b), and at the alarm time a selected remote controller initiates a selected command on a chosen appliance as shown in Figure 4.6c and Figure 4.6c.

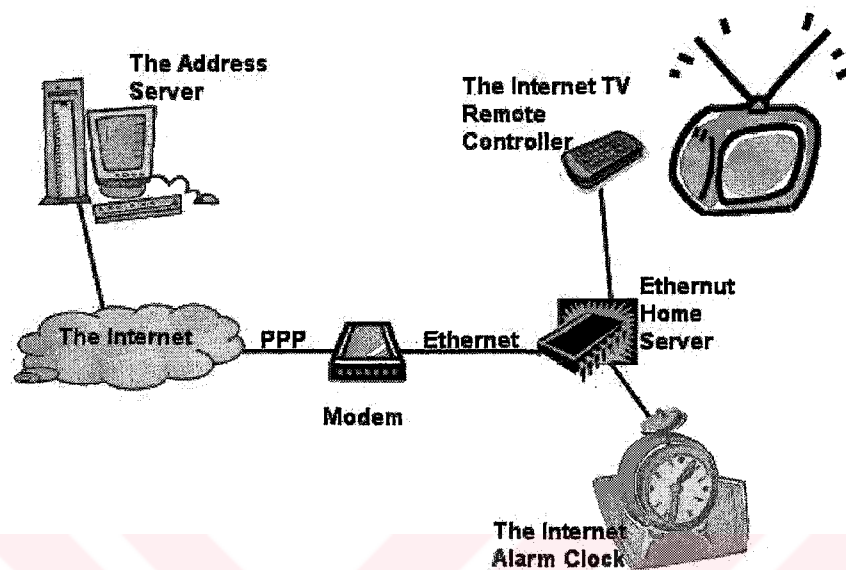


Figure 4.4: The Prototype System Overview

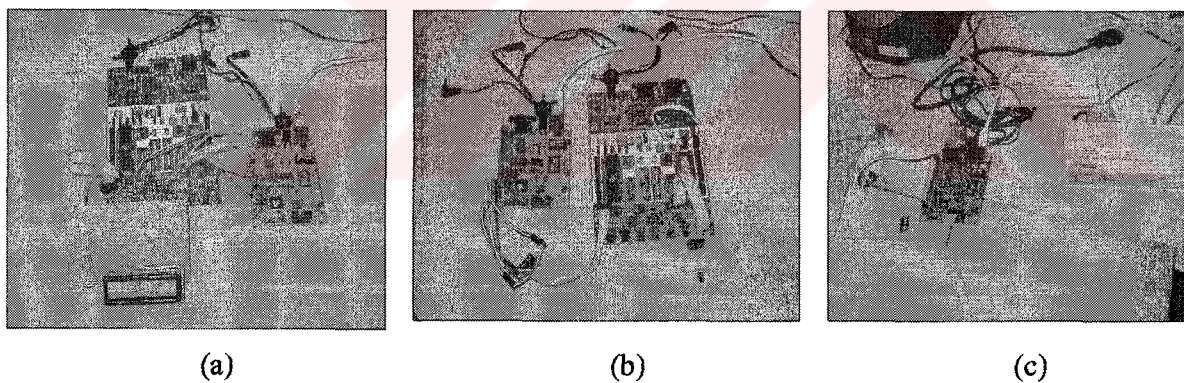


Figure 4.5: The Hardware of the Prototype Implementation - (a) Alarm Clock with embedded web server. (b) VCR remote controller with embedded web server. (c) The Ethernut Home Server

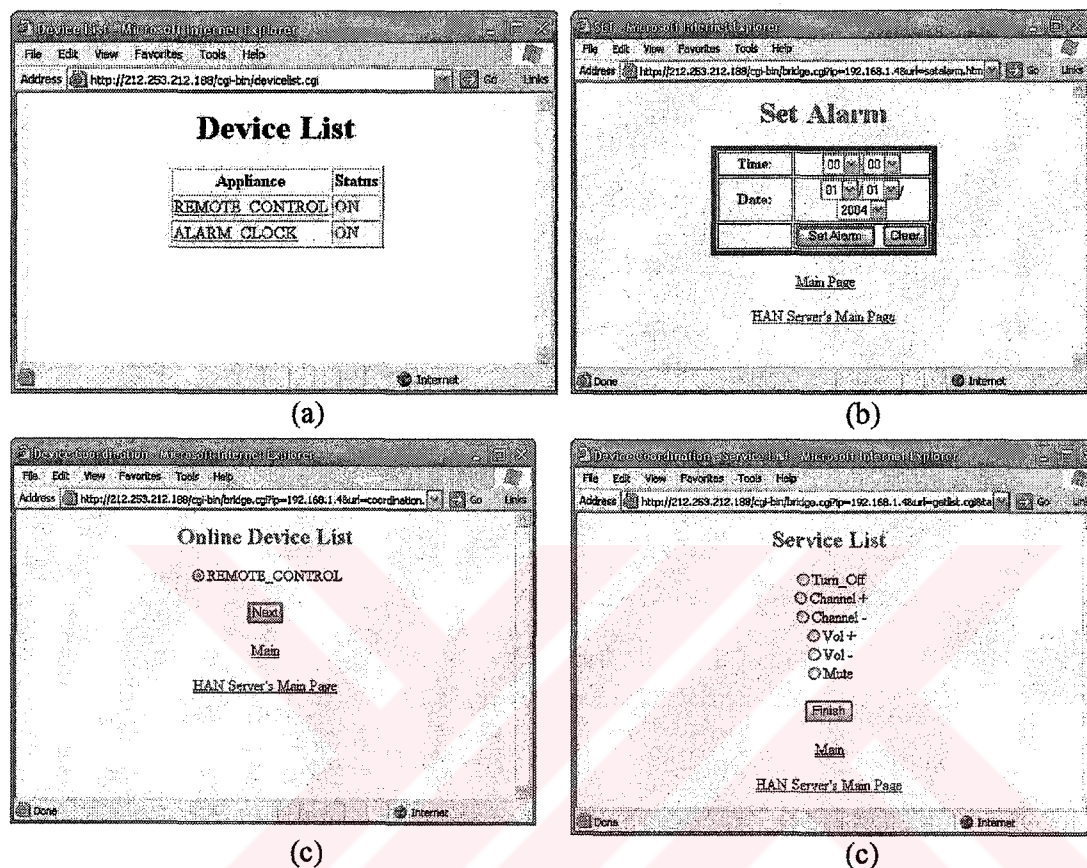


Figure 4.6: Device Coordination - (a) The web page showing the list of online devices is dynamically generated by the Ethernet Home Server. (b) The Alarm Clock is set via a form. (c) In device coordination online device list is get from EHS. (d) In device coordination related service information is get from the device itself.

In our prototype, in addition to providing device control over the Internet, the coordination between the appliances in the HAN is implemented also. Users without any information on the system can easily utilize the system since all necessary configurations are done automatically.

Chapter 3

BENEFITS OF EMBEDDING WEB SERVERS TO EVERYDAY APPLIANCES

3.1 Introduction

Recent technological developments enable us to embed low cost web servers into everyday appliances. What do we hope to gain by putting our appliances online? Basically, web servers enable users to fetch web pages and display them on their own browsers in a platform-independent manner. Information coming from the appliance's sensors can be used to generate dynamic HTML documents by Common Gateway Interface (CGI) scripts running at the server side. Therefore, any device connected this way can be monitored and controlled through CGI scripts and the results can be sent to the user's browser as a web page creating an opportunity to design user friendly appliance interfaces.

3.2 Advantages of Embedding Web Servers into Everyday Appliances

Embedding a web server is a powerful mechanism for implementing a variety of key features and functions in everyday appliances. Web enabling appliances provides an effortless method for interfacing to appliances without requiring any target side programming and such interfaces work with a universally available client program, the web browser. Any standard browser can be used to interface to web enabled appliances. Appliances can automatically interface with Windows, Linux, UNIX, and Macintosh based

systems. Client software to interface to appliances is already installed in every platform and everyone knows how to use it.

A web-enabled appliance can be accessed from anywhere in the world if it is connected to the Internet. Instead of designing special protocol for remote monitoring and control, established, powerful and universally available HTTP protocol can be used.

Another use of web enabled appliances is in remote software updates. For instance, if a dishwasher manufacturer recognizes that a product that has been on the market for sometime could be improved by a software update, sending a technician for software upgrade to every customer would be too expensive. Manufacturers can use web access to update their products' software simply by uploading the new software to devices remotely or appliances can be programmed to regularly check the manufacturer's web site for software updates and download them automatically when they are available. Via web access, products can be supported throughout their life cycle by the manufacturer and product maintenance costs can be reduced.

Furthermore, when an appliance requires maintenance, it can automatically inform the technical staff without interrupting the user with the correct and complete information about the problem.

In addition, user interfaces can be updated or extended. For example, accessibility options for helping handicapped users can be added. Furthermore, user interfaces can be featured in different languages.

Advantages of Using Web Interfaces for Everyday Appliances

Web-based interfaces are cross-platform and easier to develop. It is burdensome to implement a graphical user interface for a device. In contrast, user friendly and good looking interfaces are easy to create and maintain for web enabled appliances. Resource

demands on the target side are very low since the interface consists of HTML pages in the text format. The costs related to product documentation, training and support are lower.

Most complex appliances have many unused functions, because users find them too difficult to use. There are a couple of reasons for poor interfaces. Since manufacturers economize on buttons and displays, complex features can be used only by pressing on button combinations where there is no indication on the button labels. Moreover, the feedback given to the user is usually not enough after a performed action [30]. On the other hand, as manifested by the exponential growth of the number of web users globally, ordinary people can use web based interfaces effortlessly without any knowledge of the underlying hardware and software. For example, if you want to make timer recordings of programs using a Sony SLV- N900 VCR (see Figure 3.1), you should set the recording start time, recording stop time and the channel number. This recording operation is listed under the basic operations of the VCR. On the other hand, the operation is done in 8 steps and it is explained in the owner's manual for 4 pages.

Possible web based interfaces for the Sony SLV-N900 VCR are given in Figure 3.2, which lists the operations of the VCR and serves as a Menu, and in Figure 3.3, which is the interface for setting the timer for recording programs.

Using the web page based wizard given in Figure 3.3, users can set recording start/stop times and the channel by filling a simple web form. Moreover, information and warnings can be displayed in web pages to inform users.



Figure 3.1: Sony SLV-N900 VCR

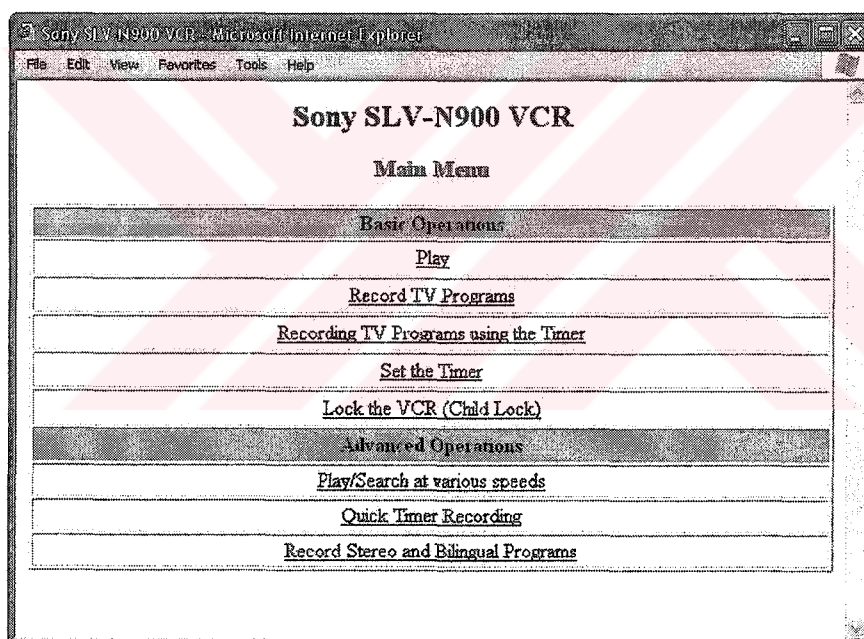
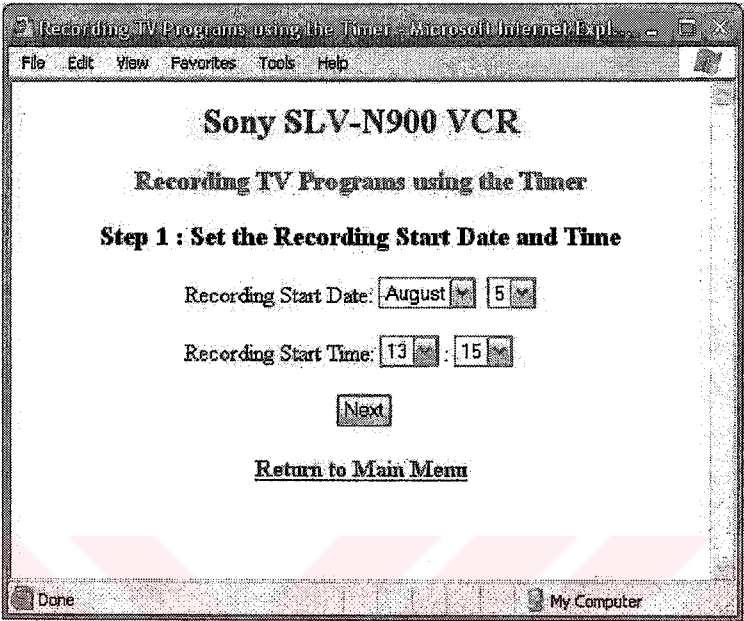
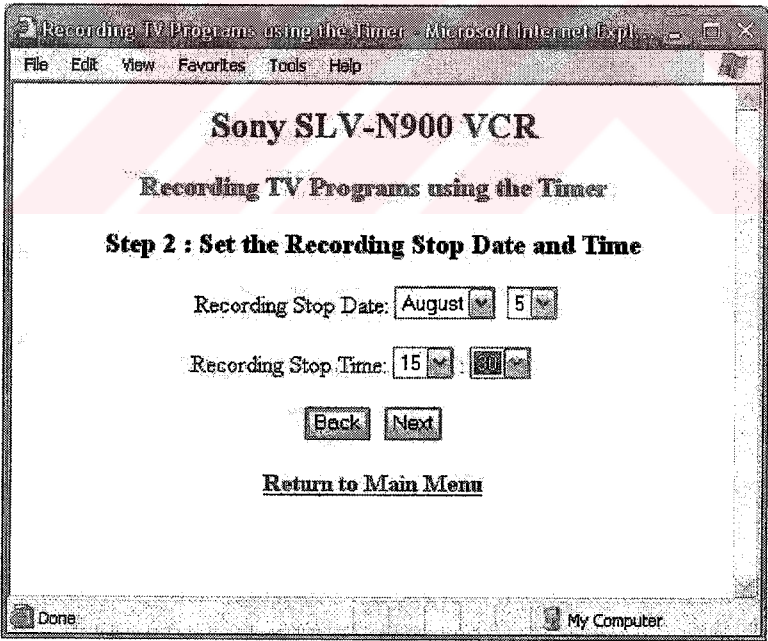


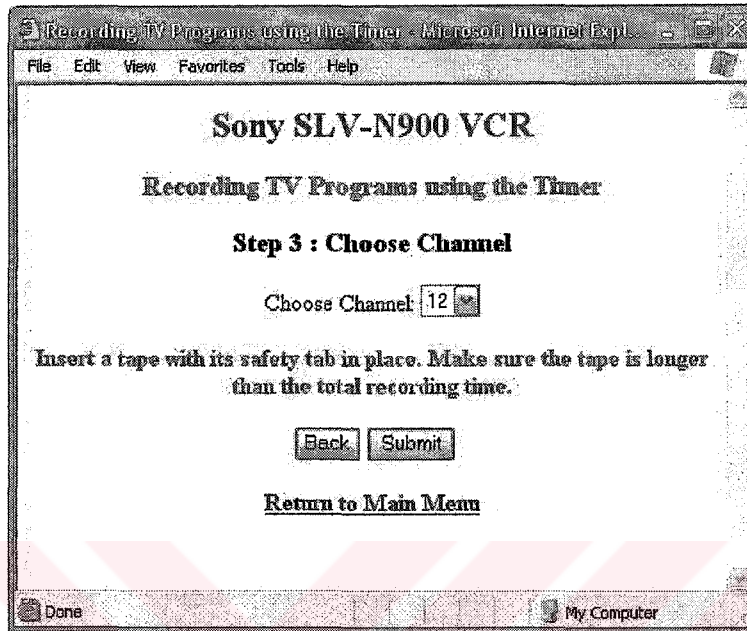
Figure 3.2: Sony SLV-N900 VCR's Main Menu listing links to basic/advanced operations



(a)



(b)



(c)

Figure 3.3: VCR's web page based wizard for recording programs using the timer.

There are many industrial and academic groups that are working on generating interfaces for complex appliances [3, 7, 29]. Human Computer Interaction Institute of Carnegie Mellon University investigates how handheld devices can be used to improve the interfaces for everyday appliances, using an approach called *Personal Universal Controller* (PUC). PUC exchanges information with the appliance. First, it downloads the description of the appliance's functions, then it creates a control panel using transferred description and finally it sends control signals to the appliance as the user operates the panel. The control panel contains figures, informative texts and extra features that ease the usage of the appliance. Actually, the research is focused on automatic generation of control panels. Two studies have been conducted in order to create high quality and user friendly control panels.

Two common appliances are used in each study: Aiwa CX-NMT70 stereo (see Figure 3.4) and AT&T 1825 telephone (see Figure 3.5).



Figure 3.4: Aiwa CX-NMT70 Stereo

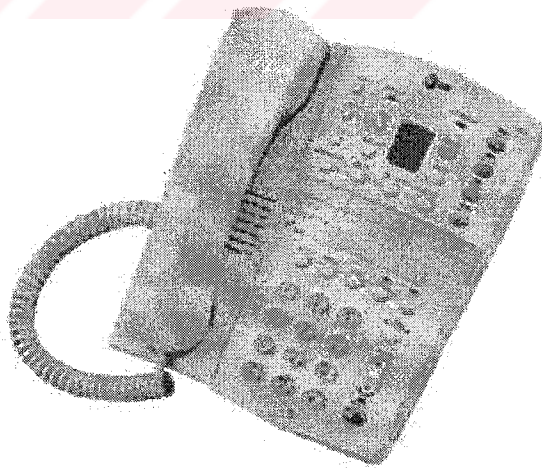


Figure 3.5: AT&T 1825 Telephone/Answering Machine

The first study is a between-subjects comparison of a handheld interface with the interface to an actual device [33]. Paper prototypes [39] were used in the first study as the handheld interface. In the second study, actual implementations of handheld interfaces were used instead of paper prototypes. The purpose of the studies was to discover how users performed using hand-designed interfaces. The subjects were given a questionnaire about their computer backgrounds. Then each subject worked with one of the four task lists: actual stereo and handheld phone; actual phone and handheld stereo; handheld stereo and actual phone; handheld phone and actual stereo. When subjects completed their tasks, another questionnaire was given that asked whether the actual appliance's interface or handheld interface was preferred. In the evaluation process three metrics were used: time to complete the tasks, number of missteps made during the tasks, and the number of times external help was requested.

Thirteen Carnegie Mellon graduate students, five female and eight male, participated the first study. All subjects had significant computer experience. Only one subject had no Palm experience. Everyone had some experience with stereo systems. Results of the first study showed that participants had difficulty using the actual appliances. On the other hand, they were able to understand and use the paper prototypes with reasonable ease.

In the second study, paper prototypes were changed with working prototypes. Interfaces for the stereo and phone that run on a PocketPC were created. Twelve students from Carnegie Mellon University participated in the second study. Unlike the first study, subjects with little or no knowledge of handheld computers were chosen. Eight men and four women with an average of five years experience using computers participated in the study. The results of both studies are very similar. External help was not requested by most of the subjects and subjects that requested help, requested only once. On the other hand, with actual stereo and actual phone, participants requested help with an average of 3.6 and

4.3 times respectively. Poor labeling, insufficient feedback and the overloaded buttons with multiple functions were the reasons for this significant difference.

Both studies showed that users were able to complete complex tasks in about one half of the time and with half as many errors as the actual device interfaces [29]. The results of experiments support the need for more informative user interfaces. In the sense of simplifying the user interface of the appliance, the handheld interface mentioned above and the web-based interface presented in this thesis both address the same requirements.

Drawbacks such as poor labeling, insufficient feedback, and the overloading of buttons with multiple functions found on actual appliances can be avoided with web page based interfaces, because there is room for labels that are more descriptive and certain multi-step functions can be put on separate web pages where navigation between steps are accomplished by hyperlinks.

Therefore, users can control complex appliances easily with well designed web-based interfaces which contain self informative buttons, hyperlinks, texts, figures, etc.

Clearly, providing user interfaces as web pages, does not guarantee having well designed user interfaces. However, a good user interface could easily be designed and maintained using web pages and associated design tools. User interfaces can be designed and debug on desktop computers before embedding them into the target appliance. This thesis focuses on the infrastructure for using web pages as user interfaces. We do not address web page design for better user interfaces.

3.4 Summary

The benefits of embedding a web server into an appliance can be summarized as follows:

- Users can interact with the appliance using a device which contains a web browser. Such devices, ranging from PCs to cell phones, are becoming universally available and using them is becoming a common knowledge.
- User-interface hardware (electromechanical) costs can be eliminated and more user-friendly interfaces can be designed with low costs.
- Controlling, monitoring and updating can be done from anywhere in the world.
- Product can be supported throughout its life cycle by the manufacturer via the upload of new software versions remotely.
- Product maintenance costs can be reduced.
- User interfaces can be updated or extended. For example, accessibility options for helping handicapped users can be added. User interface can be featured in different languages, etc.

Chapter 5

THE DEVELOPMENT PLATFORM SELECTION

When planning the implementation of the HAN system prototype, it became apparent that it would require too much time to design and construct it from elementary components. The first stage of the implementation was to investigate which components had already been developed and which components require development.

This thesis focuses on the software development of the HAN System for an embedded implementation. Therefore, finding a suitable embedded system platform, to develop the software is one of the major requirements.

An embedded web server should use the HTTP standard protocol to transmit the Web pages from the embedded system to the web browser and to transmit form data back to the embedded system attached to the appliance. The embedded system requires a network interface, such as Ethernet, a TCP/IP protocol stack, embedded web server software and static and dynamic web pages that form the user interface for that specific device.

Since embedded systems have limited CPU and memory resources and these resources are mostly used by real-time applications, end-users may have to wait up to a few seconds for an HTTP response. Multithreading should be employed in embedded systems to avoid such slow responses. Moreover, web server software must be able to respond to multiple HTTP requests simultaneously and must use non-blocking multithreading such that one HTTP transaction waiting for data will not prevent progress of another HTTP transaction.

When implementing the TCP/IP stack for microcontrollers, some functionality of it can be omitted depending on the specific application requirements. In order to serve wide range of application requirements while keeping the size compact, an embedded system should implement five basic protocols of the TCP/IP protocol suite: TCP, UDP, IP, ICMP and ARP. Moreover, application level protocols DHCP, DNS and HTTP should also be implemented.

Users can interact with appliances by submitting a form to an embedded web server. The form data is processed by a CGI script and the results are returned back to the user as an HTML page. Appliances can be monitored and controlled by forms.

Hardware of a web-enabled embedded system must have the following features: a powerful and low-cost microcontroller, compact size, flash/EEPROM for accommodating the TCP/IP stack and an Ethernet controller. Therefore, the results of the platform search focused on solutions that could provide a hardware platform with an operating system and TCP package. Moreover, the search also attempted to find products that are available at a sensible cost. Also, we considered the legal ramifications of using a platform since the long term aim of the thesis is to have something that can be manufactured in large numbers and sold.

Taking all these software and hardware requirements into consideration, we chose the Ethernut system, which is an open source hardware and software project for building embedded Ethernet devices, as the core component of our web accessible HAN system. All the schematics and PCB's of the development board are provided by the manufacturer. The design is available for free and it is legal to use it in commercial products. Ethernut provides the most attractive solution with a free OS, TCP/IP environment and hardware design.

In this chapter, we discuss hardware and software features of the Ethernut Platform.

5.1 Hardware

In this section, the hardware components of the Ethernut will be presented in detail. Next, the details of the Atmel ATmega128 microcontroller, core of the Ethernut, will be presented.

5.1.1 The Ethernut Board

The Ethernut, developed by Egnite Software GmbH [36], is a small board (80 x 100mm), which is based on Atmel's ATmega128 8-bit RISC microcontroller. For Ethernet access, it uses Realtek RTL8019AS Ethernet Controller which supports 10Mbps. The ATmega128 provides 4KB internal memory which is not sufficient for Embedded Ethernet applications. Therefore, the board is equipped with 32 KB external memory.

The Ethernut board provides the following features [41]:

- Atmel ATmega128 RISC microcontroller running at 14.7456MHz
- Full duplex IEEE 802.3 compliant Ethernut controller with on-board RJ-45 connector.
- Two RS-232 serial ports, one with on-board DB-9 connector.
- 128 Kbytes in-system programmable flash ROM
- 4 Kbytes in-system programmable EEPROM
- 32 Kbytes SRAM
- 22 programmable digital I/O lines.
- 8-channel, 10-10 bit analog/digital converter.
- Two 8-bit, two 16-bit timer/counters.
- Watchdog timer and reset controller

- LED indicators for power supply, programming mode and Ethernet activity.
- Single power supply DC 8-16V.

External devices, which may be simple I/O circuits driven by the Ethernet board, or may be equipped with their own processor, can be added to the expansion port. The expansion port contains CPU data and address bus, memory read/write signal and power supply. The expansion port details are given in Table 5.1 and illustrated in Figure 5.1.

Table 5.1: Expansion port pin configuration

Pins	Signal	Function
1 - 2	NC	Reserved for future expansion
3 - 4	VCC	5V regulated supply
5...8	GND	Signal ground
9	RESET	Low active reset signal
10	DC	Unregulated supply voltage, depends on the board's supply
11 - 12	VCC	5V regulated supply
13	RD	Low active read strobe
14	WR	Low active write strobe
15...22	DATA	8-bit data bus
23...38	ADDR	16-bit address bus
39...46	PORTE	8-bit bi-directional digital I/O Port E
47...54	PORTB	8-bit bi-directional digital I/O Port B
55...62	PORTD	8-bit bi-directional digital I/O Port D

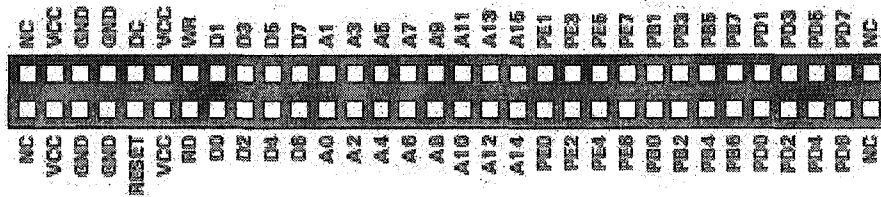


Figure 5.1: Ethernet's expansion port

Ethernut provides an on-board DB-9 RS232 serial communication port. This port is connected to the Atmel ATmega128 on-chip UART via a single chip RS-232 driver/receiver interface circuit, which converts the required voltage levels from the 5V power supply. The serial port details are given in Table 5.2 and illustrated in Figure 5.2.

Table 5.2: Serial Port Pin Configuration

Pins	Signal	Function
2	TXD	Transmit data, if pins 2 and 4 of JP1 are shortened.
3	RXD	Transmit data, if pins 1 and 3 of JP1 are shortened.
4	DSR	Data set ready, not connected.
5	GND	Signal ground.
6	DTR	Data terminal ready.
7	CTS	Clear to send.
8	RTS	Ready to send.

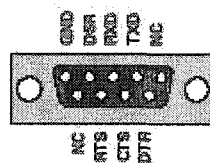


Figure 5.2: Ethernet's serial port

Ethernut provides an on-board RJ-45 connector which is connected to the Realtek 8019AS Ethernet controller via a 10Base-T transformer. The interface supports the maximum cable length of 100 meters between two Ethernet devices. The Ethernet port details are given in Table 5.3 and illustrated in Figure 5.3.

Table 5.3: RJ-45 connector's pin configuration

Pins	Signal	Function
1,2	TX	Transmit output pair.
3,6	RX	Receive input pair.
4,5	POWER	Power supply
7,8	GND	Signal ground.

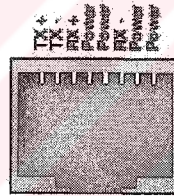


Figure 5.3: Ethernut's RJ-45 connector

The Ethernut board contains 4 LEDs. LED1 (red) is directly connected to the power supply and is lit when the power is on. LED2 (red) is lit during in-system programming. LED3 (yellow) and LED4 (green) are used to indicate Ethernet activity. If the 10BASE-T link status is OK, the yellow LED should be lit. The green LED indicates the network activity.

The system reset is controlled by a push button. The ATmega128 microcontroller provides an on-chip watchdog timer, which forces a system reset, if the application program fails to periodically update this timer as a result of software bugs or temporary hardware failures.

The ATmega128 microcontroller clock is generated by a 14.7456MHz crystal which may be replaced with a crystal of up to 16MHz. Another crystal of 32.768 kHz drives an on-chip asynchronous timer which is used for a software real-time clock. A third crystal of 20 MHz is used to drive the Ethernet Controller.

The microcontroller provides 128 Kbytes of on-chip, non-volatile flash memory space for program code and read-only data. The memory can be programmed via in-system programming. It is organized as 64K x 16 bits. The board provides 32 Kbytes SRAM, which is used as read/write data storage. In addition, the microcontroller provides 4 Kbytes of on-chip, non-volatile, electrically erasable memory. This memory is accessible under program control and via in-system programming.

5.1.2 Atmel ATmega128 Microcontroller

The Ethernut is equipped with Atmel's ATmega128 microcontroller which is based on the AVR enhanced RISC architecture. The AVR's powerful instruction set and 32 general purpose working registers which are directly connected to the Arithmetic Logic Unit (ALU) allow two independent registers to be accessed in one single instruction executed in one cycle, resulting code efficiency and throughputs up to ten times faster than CISC microcontrollers [31].

The ATmega128 provides the following features:

- 128Kbytes of In-System Programmable Flash
- 4Kbytes EEPROM
- 4Kbytes SRAM
- 53 general purpose I/O lines
- 32 general purpose working registers

- Real Time Counter (RTC)
- 4 Timer/Counters with compare modes and Pulse Width Modulation (PWM)
- 2 UARTs
- A byte oriented two-wire Serial Interface
- An 8 channel, 10-bit Analog/Digital Converter (ADC)
- Programmable Watchdog Timer with Internal Oscillator
- An SPI Serial Port
- IEEE 1149.1 compliant JTAG test interface
- Six software selectable power saving modes
- Low power consumption

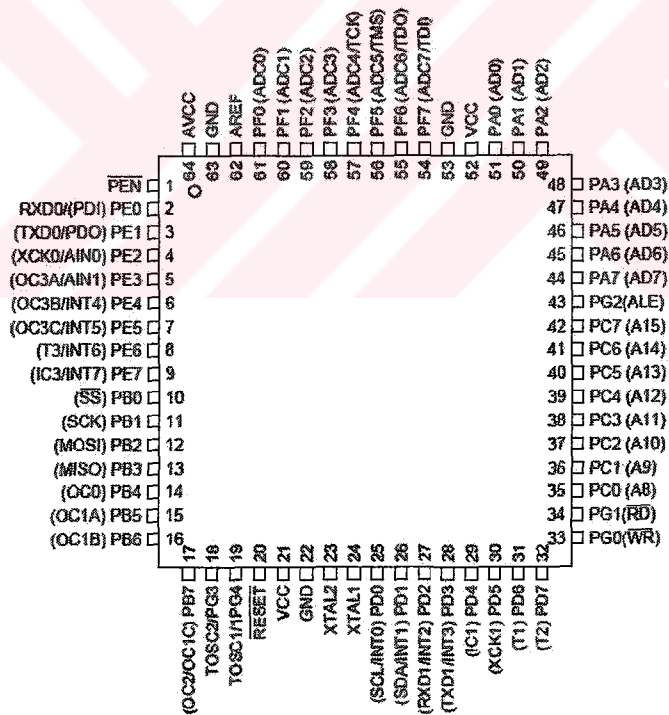


Figure 5.4: Atmega128 Pin Configuration

Atmel's ATmega128 AVR is supported with a full set of program and system development tools such as: macro assemblers, C Compilers, program debugger and simulators, in circuit emulators and evaluation kits.

Pin configuration of the microcontroller is given in Figure 5.4.

5.2 Software

In this section, the software provided on the Ethernut platform will be explained in detail. The Ethernut software is split into two functional components: The Nut/OS and the Nut/Net. The Nut/OS is a real time operating system with multithreading support, event and memory management, and a couple of device drivers. The Nut/Net is a programming interface to a TCP/IP stack that provides ARP, IP, UDP and TCP over Ethernet, automatic configuration via DHCP and TCP/UDP socket API. Next, the principles of creating an embedded web server will be described.

5.2.1 Ethernut Operating System (Nut/OS)

The Nut/OS is an Open Source implementation of a Real Time Operating System (RTOS). The simple RTOS provides the following features [42]:

- Cooperative multithreading
- Event queues
- Dynamic memory management
- Timer support
- Stream I/O functions
- Expandable device driver interface
- Flash ROM file system

5.2.1.1 Memory Considerations

Nut/OS and applications make use of three types of memory:

- Flash ROM – for program code and constant data
- RAM – for stack and variables
- EEPROM – for configuration data

Atmel's AVR microcontrollers are based on Harvard Architecture which separates the data and instruction code memory.

5.2.1.1.1 Flash ROM

Flash ROM (Read Only Memory) is a non-volatile memory which preserves its contents even if the power supply is removed. Atmel's ISP and the JTAG development systems allow erasing and writing data to the flash ROM without removing the microprocessor from the Ethernut board. All program code resides in a flash ROM. The 16 bit program counter of ATmega128 can address up to 64K words.

Constant data can also be located in the flash ROM. Placing constant data into the flash ROM saves RAM space which is scarcer than the flash ROM space.

Read access to the flash ROM is not slower than the RAM access. On the other hand, the compiler generates extra code for ROM access, which increases execution time.

5.2.1.1.2 RAM

Unlike flash ROM, random access memory (RAM) does not preserve its contents if the power supply is removed. The 64Kbyte address space is divided in several parts and memory layout is summarized in Table 5.4.

Table 5.4: RAM Address Space Parts

Address Range	Used By
0x0000 – 0x001F	32 CPU registers
0x0020 – 0x005F	64 I/O registers
0x0060 – 0x00FF	160 extended I/O registers
0x0100 – 0x10FF	On-chip internal SRAM
0x1100 – 0x7FFF	External SRAM
0x8000 – 0x82FF	Unused
0x8300 – 0x831F	RTL8019AS registers
0x8400 – 0xFFFF	Unused

5.2.1.1.3 EEPROM

Electrically erasable programmable read only memory (EEPROM) is a non-volatile memory which preserves its contents even if the power supply is removed. It is the ideal place to hold configuration information that can be used by Nut/OS and applications.

The first 128 bytes of the ATmega128 on-chip EEPROM is reserved for Nut/OS and Nut/Net. The remaining 3968 bytes are available for application usage. Table 5.5

summarizes the EEPROM contents. EEPROM read and write access is very slow. Its use should be limited to reading configuration information during the startup.

Table 5.5 EEPROM Contents

First	Last	Bytes	Description
0000	0000	1	Nut/OS configuration structure
0001	0002	2	Magic cookie
0003	0039	61	Reserved
0040	0040	1	Nut/Net configuration structure
0041	0049	9	Name of the network device
004A	004F	6	Ethernet MAC address
0050	0053	4	Previously used IP address
0054	0057	4	IP network mask
0058	005B	4	Default gateway IP address
005C	005F	4	Configured IP address
0060	0080	32	Reserved

5.2.1.2 System Initialization

Typically, C programs start executing the **main** routine. On the other hand, in Nut/OS **NutInit()** function is executed initially. It initializes the memory management and the thread system, and then starts an idle thread which initializes the timer management. Next, the application's **main()** function is called. The following code demonstrates the simplest application. It only runs in an endless loop and does nothing.

```
#include <compiler.h>

int main(void) {
    for(;;)
        ;
}
```

5.2.1.3 Thread Management

The Nut/OS supports cooperative multithreading. Applications are usually single-threaded, which sequentially execute on a single path. Multithreaded programs concurrently execute several threads. It is impossible for a single CPU to run more than a single thread simultaneously. However, Nut/OS switches the microprocessor from the context of one thread to the context of another thread. This gives an impression of simultaneously running multiple threads.

All threads execute in the same address space sharing the same hardware resources. This reduces the task switching overhead. Context switching is determined by the concept of the thread priority which determines how urgent a thread is. The priority ranges from 0 to 255 (255 – the idle thread). Lower values indicate higher priorities. The thread management principle is simple: the most urgent thread always runs if it is not waiting for an event.

The Nut/OS thread system supports cooperative multithreading which means that if threads are not waiting for specific events, they can rely on not being stopped unexpectedly (Unless there is a CPU interrupt signal). Cooperative multithreading simplifies resource sharing and results in faster and smaller code.

While switching between threads, Nut/OS saves all the register values and auto variables, which are locally declared variables within functions of the running thread and restores the previously saved register contents and auto variables of the thread being started. Therefore, each thread needs its own stack.

The memory area used for the stack is allocated when the thread is created. Determining the required stack size is difficult since the space for register storage is decided by the compiler and Nut/OS API functions called by the application may call other functions as well. Nut/OS allocates 768 bytes of stack for the main thread. Most application threads will be satisfied with 512 bytes [42].

Threads are created by calling the **NutThreadCreate**. The thread code is a C routine. To hide Ethernut specific details, applications should use the **THREAD** macro.

A simple thread application is given below, where the main thread creates a new thread before entering into an infinite loop.

```
THREAD(Thread1, arg)
{
    for(;;)
    {
        NutSleep(500);
    }
}

int main(void) {
    NutThreadCreate("t1", Thread1, 0, 192);
    for(;;)
    {
        NutSleep(125);
    }
}
```

The cooperative nature of Nut/OS should be kept in mind when building multithreading applications. If one of the threads does not call **NutSleep**, which stops execution for a specified number of milliseconds, or any other blocking function like reading from a device or waiting for an event, then other threads with higher or lower priorities would never gain CPU control.

5.2.1.4 Event Management

In a multithreaded system, threads spend most of the time waiting for an event. Nut/OS provides an event queue mechanism. Threads line up in queues when they are waiting for events. Meanwhile, other threads post events to queues to wake up threads waiting for events.

Waiting threads line up in priority ordered queues, so more than one thread may wait for the same event. The priority queues represent the central structure for thread scheduling and synchronization. A waiting queue is a simple linked list of waiting threads. The waiting queue is actually a void pointer. The queue is empty if the pointer is set to null. Threads are added to the waiting queue by setting the void pointer to the thread's info structure which contains a void pointer itself that can be used to point to another info structure. This way the OS sets up the linked lists of thread info structures. Events are posted to a waiting queue, moving the thread from sleeping state to ready-to-run state. A running thread may also broadcast an event to a specified queue, waking up all threads on that queue.

A simple event management application sample is given below: Thread1 is running an endless loop of sleeps while the main thread waits for an event posted to an event queue. HANDLE's are used for specifying the event queues. The main thread's endless loop is blocked in NutEventWait and woken by events posted by Thread1.

```
THREAD(Thread1, arg)
{
    for(;;)
    {
        NutSleep(500);
        NutPostEvent(&event_handler);
    }
}

int main(void) {
```

```
NutThreadCreate("t1", Thread1, 0, 192);  
for(;;)  
{  
    NutEventWait(&event_handler, NUT_WAIT_INFINITE);  
}
```

5.2.1.5 Stream I/O

Typical C applications make use of the standard I/O library for stream I/O. Nut/OS provides its own library called **nutcr** that overrides the functions of the compiler's runtime library. The functions available in the **nutcr** allow to port PC applications to Ethernut without too much effort. On the other hand, embedded systems do not pre-define devices for **stdin**, **stdout** and **stderr**. Therefore a typical 'Hello World' application for a PC

```
#include <stdio.h>  
  
int main(void){  
    printf("Hello World");  
}
```

requires a few additional lines for device definitions.

```
#include <stdio.h>  
#include <dev/uart.h>  
  
int main(void){  
    unsigned long baud = 115200;  
  
    /*register the I/O device*/  
    NutRegisterDevice(&devUart0, 0, 0);  
  
    /*assign the device to stdout*/  
    freopen("uart0", "w", stdout);  
}
```

```
/*set the baudrate of the serial port*/
_ioctl1(_fileno(stdout),UART_SETSPEED, &baud);

printf("Hello World");
}
```

5.2.1.6 Heap Management

Dynamic memory allocations are made from the heap which is the global resource containing all of the free memory in the system. Nut/OS does not use the heap management of the C runtime library. It employs its own heap management which handles the heap as a linked list of unused memory blocks. Nut/OS heap manager uses best-fit, address ordered algorithm to keep the heap as unfragmented as possible. This strategy ends up with few large free blocks rather than lots of small ones.

Applications use standard C calls to allocate and release memory blocks. The following example code demonstrates memory allocation and memory releasing.

```
char *buffer;
buffer = malloc(1024);           //Allocating a buffer of 1024 bytes

if(buffer){
    memset(buffer, 0xFF, 1024);   //fills buffer with 0xFF
    free(buffer);                //releases the buffer
}
else{
    puts("Out of memory!");
}
```

5.2.1.7 Timer Management

The timer management provides functions to start/stop asynchronous timers, to determine the CPU speed, to let a thread to give up the CPU for a time period and to suspend a thread temporarily.

NutSleep allows an application to delay itself for a number of system clock ticks. A clock tick occurs every 62.5 ms in the default configuration. During the sleep time, CPU control is given to another waiting thread. If there is no other thread ready to run, the CPU control is passed to the idle thread.

Nut/OS uses on-chip timer0 of the ATmega128 CPU. Other timers are available for the application when the system timer resolution is insufficient.

5.2.1.8 File System

Neither Nut/OS nor Nut/Net requires a file system. On the other hand, web servers are designed with a file system. Nut/OS provides a simple file system called **UROM**, where files are located in flash ROM. A special tool **CREATE UROM** (crurom) is used to convert directories into C source files, which are then compiled and linked to the application code.

5.2.2 Ethernet TCP Stack (Nut/Net)

TCP/IP stack implementations for desktop computers require large code and buffer space. On the other hand, embedded systems like Ethernet could not support these huge TCP/IP stacks. Nut/Net is a TCP/IP stack especially designed for small systems.

Most IP stacks rely on polling information out of the network code that buffers information until a request. In contrast, in Nut/Net, the data units are pushed up the protocol layers as soon as they are received unless one of the layers utilizes a procedure to do buffering. This strategy reduces memory requirements, eliminates queuing and threading needs [43].

Nut/Net provides the following features [42]:

- ARP, IP, UDP, ICMP and TCP protocol over Ethernet
- Automatic configuration via DHCP
- HTTP API
- CGI Function Support
- TCP and UDP socket API
- Host, net and default routing

5.2.2.1 Network Device Initialization

All applications must register the network device driver and configure the network interface before using Nut/Net functions. Typical network initialization starts with a DHCP request. If the network houses a DHCP server, it replies with a local IP address, a network mask and a default gateway address.

If no DHCP server is available in the network, previously stored configuration from the on-chip EEPROM is used.

If the EEPROM doesn't contain any configuration information and no DHCP server is available on the network, then the ARP method can be used to set the board's IP address. In this mode the Ethernet board set its address from the first ICMP packet it receives. In this case the network mask will be 255.255.255.0 and no default gateway will be configured.

The following code fulfills most requirements of network configuration:

```
if(NutDhcpIfConfig("eth0", 0, 60000)){
    u_char my_mac[] = {0x00, 0x06, 0x98, 0x01, 0x01, 0x01};
    if (NutDhcpIfConfig("eth0", my_mac, 60000)){
        u_long ip_addr = inet_addr("192.168.1.100");
        u_long ip_mask = inet_addr("255.255.255.0");
        NutNetIfConfig("eth0", my_mac, ip_addr, ip_mask);
    }
}
```

No MAC Address is passed to the *NutDhcpIfConfig* in the first call. If the EEPROM contains a MAC Address, then the DHCP server will be queried for the related IP settings and the function return 0. If the DHCP fails or the EEPROM does not contain a valid MAC address, it returns -1. In the second call, hard coded MAC address is passed to *NutDhcpIfConfig*. If it fails again, there is no valid EEPROM content and no available DHCP server. Finally, we pass hard coded IP address and IP mask to *NutNetIfConfig* to bring the network interface up.

NutNetIfConfig will store the specified address as a hard IP address into the EEPROM. When Ethernut reboots again and call *NutDhcpIfConfig*, it will use this hard IP address and never try DHCP again unless the EEPROM is erased.

5.2.2.2 Nut/Net Protocols

As mentioned before, embedded systems like Ethernut could not support full TCP/IP stacks. Nut/Net is a TCP/IP stack especially designed for small systems. In this section, protocols implemented in the Nut/Net such as TCP, UDP, IP, ICMP, ARP, HTTP, PPP, DHCP, and DNS will be briefly explained.

The Transmission Control Protocol (TCP) is a connection-oriented, reliable delivery byte-stream transport layer protocol which provides interface between a network layer

below and an application layer above. TCP implementation of Nut/Net makes sure that all data is transmitted reliably and in order. TCP requires more code and buffer space than any other part of Nut/Net.

The User Datagram Protocol (UDP) is a minimal message-oriented transport layer protocol which provides a simple interface between a network layer below and an application layer above. UDP does not guarantee message delivery and a UDP sender does not retain any state on UDP messages once sent onto the network. The advantage of the UDP is its reduced overhead. On the other hand, if datagrams get lost during transmission, the application is responsible for recovery.

Applications should use the Nut/Net socket API to utilize the TCP and UDP.

The Internet Control Message Protocol (ICMP) is part of the Internet Protocol Suite. ICMP messages are typically generated in response to errors in IP datagrams or for diagnostics or routing purposes. Each ICMP message is encapsulated within a single IP datagram and therefore does not guarantee delivery. Nut/Net automatically responds only to an ICMP echo request with an ICMP echo reply, which is useful when testing network connections. Other ICMP messages are not implemented in Nut/Net.

The Internet Protocol (IP) is a data-oriented protocol used by source and destination hosts for communicating in a packet switched internet. The Internet Protocol provides an unreliable datagram service (called as best effort) which makes no guarantees about the packet delivery. The packet may arrive damaged, it may be out of order, it may be duplicated or it may be dropped entirely. Reliability needs are served by the upper layers.

The Address Resolution Protocol (ARP) is used for mapping an Internet Protocol Address (IP Address) to a physical machine address (MAC address) that is recognized in the local network. The sender broadcasts an ARP packet with the Internet address of another host and waits for it or other host to send back its Ethernet address. Each host maintains a cache of address translations to reduce delay and loading.

The Hypertext Transfer Protocol (HTTP) is the primary method used to exchange information on the World Wide Web. The HTTP is based on TCP. Nut/Net offers a set of helper APIs to simplify writing Embedded Web Server applications.

The Point-to-Point Protocol (PPP) is used to establish a direct connection between two nodes. The Ethernut offers PPP as an alternative to the Ethernet protocol that can be used with serial ports, modems, GPRS, etc. The Nut/Net provides PPP with only client mode which means other nodes can be actively connected, but Nut/Net can't listen to incoming connection attempts.

The Dynamic Host Configuration Protocol (DHCP) provides safe, reliable and simple TCP/IP network configuration and offers dynamic configuration of Internet Protocol addresses for computers within the network. DHCP ensures that address conflicts do not occur and helps conserve the use of IP addresses.

5.2.2.3 Nut/Net Socket Application Programming Interface

Nut/Net provides an Application Programming Interface (API) based on sockets. Two items are used to establish a connection between applications to transfer data between them: the IP address and a port number. The IP address determines the host to connect to and the port number is used to determine the specific application on that host.

Typically, applications take one of the possible two roles: the Server or the Client. Servers listen for connection requests on specific port numbers. The Client port numbers are automatically selected by the Nut/Net.

Nut/Net provides a socket API both for the TCP and UDP. In order to exchange data between parties, the first step is to create a socket by calling *NutTcpCreateSocket* or *NutUdpCreateSocket*.

Server applications using the TCP should call *NutTcpAccept* with a specific port number. This function call will block until a Client application tries to connect to the Server's TCP port by calling *NutTcpConnect* with the Server's port number. After the connection, both parties exchange data by calling *NutTcpSend* and *NutTcpReceive*.

A simple TCP server looks like this:

```
TCP_SOCKET *socket = NutTcpCreateSocket();
NutTcpAccept(sock, 80);
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpSend(sock, "Connected\r\n", 11);
NutTcpCloseSocket(sock);
```

A simple TCP client looks like this:

```
TCP_SOCKET *socket = NutTcpCreateSocket();
NutTcpConnect(sock, inet_addr(10.10.10.2), 80);
NutTcpSend(sock, "Am I connected\r\n", 16);
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpCloseSocket(sock);
```

Additionally, Nut/OS allows us to attach TCP sockets to standard I/O streams enabling us to use *fprintf*, *fputs*, *fscanf*, *fgets* and other stdio functions during communication.

```
stream = _fdopen((int)((uptr_t) sock), "r+b");
```

In UDP applications, Servers create sockets by calling *NutUdpCreateSocket* with a port number. UDP clients create sockets by calling *NutUdpCreateSocket* with a port number zero. Nut/Net selects an unused port number for UDP clients during socket creation. After the connection, both parties exchange data by calling *NutUdpSendTo* and *NutUdpReceiveFrom*.

5.2.2.4 Example: Simple TCP Server Application

In this section, we will present a simple TCP Server to illustrate how network initialization is done and how the socket API is used.

As with other Ethernut applications, we need to declare a function named `main` with an infinite loop that contains a `NutSleep()` function call which enables other waiting threads to take the CPU control.

```
#include <sys/timer.h>

void main(void) {
    for(;;) {
        NutSleep(1000);
    }
}
```

To communicate via Ethernet, we have to initialize the Ethernet hardware. Firstly, we have to register the Ethernet device. *NutRegisterDevice* function, which adds hardware dependent routines and data structures to the application and informs the OS about interrupt number and I/O port address, should be called for each device that we want to use in the application. The Ethernet controller of the Ethernut board uses port address 0x8300 and interrupts 5:

```
NutRegisterDevice(&devEth0, 0x8300, 5);
```

“*devEth0*” is the device information structure of the Realtek device driver. *NutRegisterDevice* sets up some data structures and initializes the controller hardware. Ethernet network interface is configured by using *NutDhcpIfConfig* function.

```
NutDhcpIfConfig("eth0", 0, 60000);
```

The first parameter “*eth0*” is the name of the registered device and it is kept in the device information structure mentioned above.

The second parameter is the MAC address of Ethernet Controller which is an array of 6 bytes. If this parameter is zero, this routine tries to read a previously stored configuration from the EEPROM. If this retrieves a valid IP configuration, then the network interface will immediately be configured calling the *NutNetIfConfig* function with these values. If no valid configuration is present in the EEPROM, then this routine will start a DHCP Client thread and wait up to the number of milliseconds given in the third parameter of the *NutDhcpIfConfig* function for a response from a DHCP Server.

If the second parameter of the *NutDhcpIfConfig* function is a valid MAC address, EEPROM configuration will not be read. If the application specifies an IP address and a network mask then the network interface will be immediately configured with these values by calling *NutNetIfConfig* function. Otherwise the DHCP client thread will be started and wait up to the number of milliseconds given in the third parameter of the *NutDhcpIfConfig* function for a response from a DHCP Server.

NutDhcpIfConfig function returns 0, if it retrieves configuration from a DHCP Server. Otherwise, it returns -1.

```
u_char mac[6] = { 0x00, 0x06, 0x98, 0x12, 0x34, 0x56 };

if (NutDhcpIfConfig("eth0", 0, 60000)) {
    if (NutDhcpIfConfig("eth0", mac, 60000)) {
        /* no DHCP Server is available in the network*/
        u_long ip_addr = inet_addr("192.168.169.170");
        u_long ip_mask = inet_addr("255.255.255.0");

        NutNetIfConfig("eth0", mac, ip_addr, ip_mask);
        /* If not in a local network, we must also call
           NutIpRouteAdd() to configure the routing. */
    }
}
```

```
}  
}
```

In order to communicate via TCP, we need a TCP socket which contains all the information about the state of the connection between two parties.

```
TCP_SOCKET *sock;  
sock = NutTcpCreateSocket();
```

These calls allocate space from the heap memory, initialize it and return a pointer to the structure. Since the application is a TCP Server application, it must wait for connection requests on a specific port number which is done by the following code:

```
NutTcpAccept(sock, 12345);
```

This call will block until a client connects to the Server. After the connection, the Server can send and receive data by calling *NutTcpSend* and *NutTcpReceive*.

Most application protocol in the Internet environment exchange information by transmitting ASCII text lines terminated by a carriage return followed by a line feed character. This might not be a problem while sending information. On the other hand, while receiving data, arriving segments may contain either a fraction of a line or multiple lines or both.

For stream devices, operating system offers the standard I/O stream library functions. In order to use stream device functions, a TCP socket should be attached to a stream.

```
FILE *stream;  
stream = _fdopen((int)sock, "r+b");
```

This device can be used like any other Nut/OS stream device and simplifies formatted and line oriented data I/O.

```
fprintf(stream, "200 Welcome\r\n");
```

The message above occupies 12 bytes of SRAM space. Alternatively:

```
prog_char welcome[] = "200 Welcome\r\n";  
fputs_P(welcome, stream);
```

In the above example message is stored in the program space and temporarily copied to SRAM when needed.

In the next step, TCP Client sends a command to the Server. Then the Server performs desired activities and returns a response to the Client and waits for the next command. Finally, the client disconnects or sends a special character to force the Server to disconnect. The Server will then call the following lines to release the memory occupied by the stream device and terminate the socket connection respectively.

```
prog_char welcome[] = "200 Welcome\r\n";  
fputs_P(welcome, stream);
```

Next, the Server may create a new socket and wait for a new client connection. The complete TCP Server application code can be found in Appendix A.

5.2.2.5 Example: Simple Web Server Application

In the previous section, we explained how to implement a general TCP Server. A Web Server is a TCP Server which talks HTTP protocol and accepts Client calls on port 80.

HTTP defines how messages are formatted and transmitted, and what actions Web Servers and browsers (Clients) should take in response to various commands.

The initialization of the Ethernet hardware and the creation of TCP sockets are the same in every TCP Server. After a Client (browser) connects to port 80 of the Web Server, a stream device must be created from the TCP socket. Finally, a simple call to *NutHttpRequest* receives HTTP requests, parses them, checks authorization, sends requested documents, calls CGI functions and sends errors.

The code needed to build an Ethernet Web Server is:

```
#include <string.h>
#include <io.h>
#include <dev/nicrtl.h>
#include <sys/timer.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <pro/httpd.h>
#include <pro/dhcp.h>

u_char mac[6] = { 0x00, 0x06, 0x98, 0x12, 0x34, 0x56 };

int main(void){

    TCPSOCKET *sock;
    FILE *stream;

    /*Register Ethernet controller.*/
    NutRegisterDevice(&DEV_ETHER, 0x8300, 5);

    /*Configure Ethernet*/
    if (NutDhcpIfConfig("eth0", 0, 60000)) {
        if (NutDhcpIfConfig("eth0", mac, 60000)) {

            /* no DHCP Server is available in the network*/
            u_long ip_addr = inet_addr("192.168.1.100");
            u_long ip_mask = inet_addr("255.255.255.0");
            NutNetIfConfig("eth0", mac, ip_addr, ip_mask);
        }
    }

    for(;;){
```

```
sock = NutTcpCreateSocket();
NutTcpAccept(sock, 80);

/*Associate a stream with the socket to use standard I/O calls.*/
stream = _fdopen((int) sock, "r+b");
NutHttpRequest(stream);

/*Destroy the virtual stream device.*/
fclose(stream);

/*Close our socket.*/
NutTcpCloseSocket(sock);
}
```

The Web Server will respond to every request with “404 Not found” since no documents and no CGI functions are available in the Server. Moreover, the Web Server is not multithreaded. Therefore, the Server receives and responds to only one request at a time.

Common Web Servers running on general purpose computers have large storage devices and file systems. Nut/OS offers similar functionality with a very simple file system based on data structures that reside in the flash memory. A Nut/OS utility is used to convert web pages to an array of hexadecimal numbers after adding the proper headers. Resulting hexadecimal numbers and headers are saved in a C source file, which is normally compiled and linked to the applications.

The Common Gateway Interface (CGI) is a standardized way of writing server-side scripts that the Server will run when a request for the relevant URL is received. In other words, CGI is a way for interfacing applications with web servers.

If you register a CGI function by calling:

```
NutRegisterCgi("setclock.cgi", "SetTimeDate");
```

The web server will call a function named *SetTimeDate* if the browser requests the URL *cgi-bin/setclock.cgi*. The Nut/Net will not send any response to the client if the *SetTimeDate* function returns zero. On the other hand, if the function returns -1, Nut/Net will send an internal error page to the browser.

The access to subdirectories can be restricted by calling the built-in *NutRegisterAuth* function.

```
NutRegisterAuth("directoryname", "username:password")
```

If a client requests any file from this directory, it will be prompted to enter a username and a password.

A complete Web Server application code can be found in Appendix B.

Chapter 6

DETAILS OF THE PROTOTYPE IMPLEMENTATION

6.1 Hardware Details of the Prototype Implementation

In order to demonstrate the characteristics of our design, we implemented two appliances: an Internet alarm clock and an Internet TV remote controller. Atmel's ATMega16 microcontrollers are used in the implementations. The ATMega16 is a low-power 8-bit microcontroller based on the RISC architecture. It achieves 1 MIPS per MHz by its powerful instructions [44]. The ATMega16 provides the following features:

- 16 Kbytes of In-System Programmable Flash Program Memory
- 512 bytes of EEROPM
- 1 Kbytes of SRAM
- 32 general purpose I/O lines
- 32 general purpose working registers
- 3 Timer/Counters
- Internal and external interrupts
- A serial programmable USART
- An A/D Converter

Pin configuration of the microcontroller is given in Figure 6.1.

(XCK/T0) PB0	1	40	PA0 (ADC0)
(T1) PB1	2	39	PA1 (ADC1)
INT2/AIN0) PB2	3	38	PA2 (ADC2)
OC0/AIN1) PB3	4	37	PA3 (ADC3)
(SS) PB4	5	36	PA4 (ADC4)
(MOSI) PB5	6	35	PA5 (ADC5)
(MISO) PB6	7	34	PA6 (ADC6)
(SCK) PB7	8	33	PA7 (ADC7)
RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7 (TOSC2)
XTAL1	13	28	PC6 (TOSC1)
(RXD) PD0	14	27	PC5 (TDI)
(TXD) PD1	15	26	PC4 (TDO)
(INT0) PD2	16	25	PC3 (TMS)
(INT1) PD3	17	24	PC2 (TCK)
(OC1B) PD4	18	23	PC1 (SDA)
(OC1A) PD5	19	22	PC0 (SCL)
(ICP1) PD6	20	21	PD7 (OC2)

Figure 6.1: Atmel ATmega16 Microcontroller Pin Configuration

The hardware details of the two appliances are given below.

6.1.1 The Internet Alarm Clock

The hardware we used for the Internet Alarm Clock is as follows:

- An Atmel ATmega16 microcontroller
- An LCD Display (2 x 16 characters)
- A Speaker

To display the time and date, an LCD module with two rows of sixteen characters is used. The LCD module is connected to the Atmel ATmega16 microcontroller's PORT A as shown in the Table 6.1.

Table 6.1: The LCD connections to Atmel ATmega16 microcontroller.

LCD Connections	ATmega16 Connections
Data Lines 4-7	PORTA PINS 4-7
Write/Read	GND
Enable	PORT A PIN 0
Register Select	PORT A PIN 1
VCC	VCC
GND	GND

The LCD module has three control lines, Write/Read, enable and register select. It has an on board processor with two registers that are accessed via a 4-bit bus. There are two registers for writing on the LCD. The first one is the control register and the second one is the data register. These two registers use the same bus but they are selected by the register select control line. The Write/Read control is always kept low to indicate that the registers are continuously being written to. The control register is used to move the cursor around the display. To write 8-bits of data into the control register, the register select is set to 0 to select the command register and the data lines are then enabled. The high 4 bits of the command are written to the data lines and a short delay allows the data lines to settle. Toggling the enable line initiates a new write cycle and the low 4 bits of the command are written to the data lines. Another short delay allows the data to settle and enable is set back to 0 and the select line is set to 1 in order to select the data register.

To display a character, the character needs to be written to the data register. This is done exactly the same as writing the control register except that at the beginning Register Select is set to 1 instead of 0. The LCD needs to be initialized, before data can be displayed on it.

The speaker is connected to the PORTD PIN5 of the microcontroller. PORTB and PORT C are connected to the Ethernut Board.

6.1.2 The Internet TV Remote Controller

The hardware we used in the Internet TV Remote Controller is listed below:

- An Atmel ATMega16 microcontroller
- An IR emitting diode
- A 220ohm resistor

The Internet TV Remote Controller is designed to control RC5 [25] compatible devices. Manchester encoded [37] infrared pulses are generated and sent when a command is read from PORTD which is connected to the Ethernut Board of ATMega16 microcontroller.

6.2 Software Details of the Prototype Implementation

Although controlling the appliances within the network they are connected to is straightforward, it is difficult to interface with the appliances over the Internet. This is because assigning a regular IP number to every appliance is not realistic with IPv4. The Nut/Net does not support Network Address Translation (NAT) protocol which allows hosts on an internal network to communicate with hosts on an external network using a single IP address. Therefore, we developed a mechanism for transferring web pages located in the appliances within the home network to clients that are requesting web pages over the Internet. Moreover, we had to find a solution to learn the IP address of the HAN Server which was changing in every PPP connection. We decided to assign a unique identifier to

every HAN Server and to keep unique identifier - IP address couples, which are updated by the HAN Servers repetitively, in another server (the Address Server) with a static IP address. Furthermore, we had to develop easily configurable appliances that could run in every environment. When an IP device is plugged to a network, it has to be configured with an IP address, subnet mask and optionally a gateway and DNS Server. If the network houses a DHCP Server, self configuration can easily be managed. On the other hand, DHCP servers are not widely available in home networks. In order to support self configuration, we have developed a mechanism by which appliances are equipped with the necessary configuration.

In this section, the software components of the prototype implementation will be explained in detail.

6.2.1 The HAN Server

The *Home Appliance Network* (HAN) connects to the Internet through the *Ethernut Home Server* (EHS). In the prototype system, the Internet connection is established via an analog modem over PSTN using the Point-to-Point Protocol (PPP) [35]. The system also supports the Internet connection over GPRS via a GSM modem. The EHS connects to the network by wired (10/100baseT) network technology.

When the EHS becomes online, it registers the network device driver by calling *NutRegisterDevice* function and configures the network interface automatically.

The EHS tries to locate a DHCP Server by calling the *NutDhcpIfConfig* function without a MAC Address. However before that, we copy the EEPROM configuration and erase it since when Ethernut reboots again and calls *NutDhcpIfConfig*, it will use the IP configuration stored in the EEPROM unless it is empty. This avoids wrong configuration when we change network settings or migrate to another network. If a MAC address is

found in EEPROM, then the DHCP server will be queried for the related IP settings and the function will return 0. If the function returns -1, we do not know whether the EEPROM does not contain a valid MAC address or DHCP is not found in the network. Therefore, the *NutDhcpIfConfig* function is called again with a MAC address. If the function returns -1 again, then it is obvious that there is no valid EEPROM content and no DHCP Server is available in the network. If there is no DHCP Server in the network, then an IP address is chosen randomly from a pre-specified range. After that, an ICMP “echo request” message [40] is sent to the chosen IP address. If a reply is not received within a few milliseconds, the configuration is saved to the EEPROM and used. If a reply is received then it means that another host is using the chosen IP configuration. In this case, another IP address is chosen randomly and verification procedure is reiterated. The flowchart of the automatic network configuration is given in Figure 6.2.

After network configuration finishes, the EHS uses Point to Point Protocol to establish the Internet connection. The PPP connection initiator is called by the PPP Client. PPP Client connects to a PPP Server. The Ethernut platform supports only the client mode therefore it is not able to answer the incoming PPP connection attempts. After the connection is established either party may run TCP/UDP Servers or Clients.

In order to establish connection using the PPP protocol two devices have to be initialized: an UART device and the PPP device which is implemented by Nut/OS.

```
NutRegisterDevice(&devUart0, 0, 0);  
NutRegisterDevice(&devPPP, 0, 0);
```

After device registration, we open the PPP device. The name of the device is *ppp*.

```
pppcom = _open("ppp:uart0/username/password", _O_RDWR | _O_BINARY);
```

The first parameter is the name of the UART device, which PPP should use. The name of the device `devUart0` is `uart0`. The next part contains the login information.

If the attempt to open the PPP device succeeds, it returns a handle for the device. Baud rate can be set using this handle.

```
u_long lctl = 115200;  
_ioctl(pppcom, UART_SETSPEED, &lctl);
```

PPP can be used with different serial hardware. We use the US Robotics Sportster 28800 external modem connected to a telephone line. We can communicate with the modem by sending Hayes commands and receiving responses via the UART device.

Nut/OS provides the *NutChat* function, which executes commands on the modem and returns the results. These commands are usually called as chat scripts. A usual chat script is made up of one or more pair of strings. The first part of the pair is the string to be sent to the modem and the second part is the expected result. A simple chat script is given below:

```
"AT OK ATD 93990000 CONNECT"
```

The first pair is the string `AT` and the string `OK`. When executed, *NutChat* function sends out `AT` followed by a carriage return and expects `OK`. If the modem is Hayes compatible, it will respond to the `AT` command with a string `OK`. As soon as the string `OK` is received by the *NutChat* function, it sends `ATD93990000` which is the Hayes command for dialing the number 93990000. As soon as the modem establishes a connection, it sends something like `CONNECT 33600`. After the *NutChat* function detects the string `CONNECT`, it will return 0 meaning that the modem has been successfully connected to the remote host.

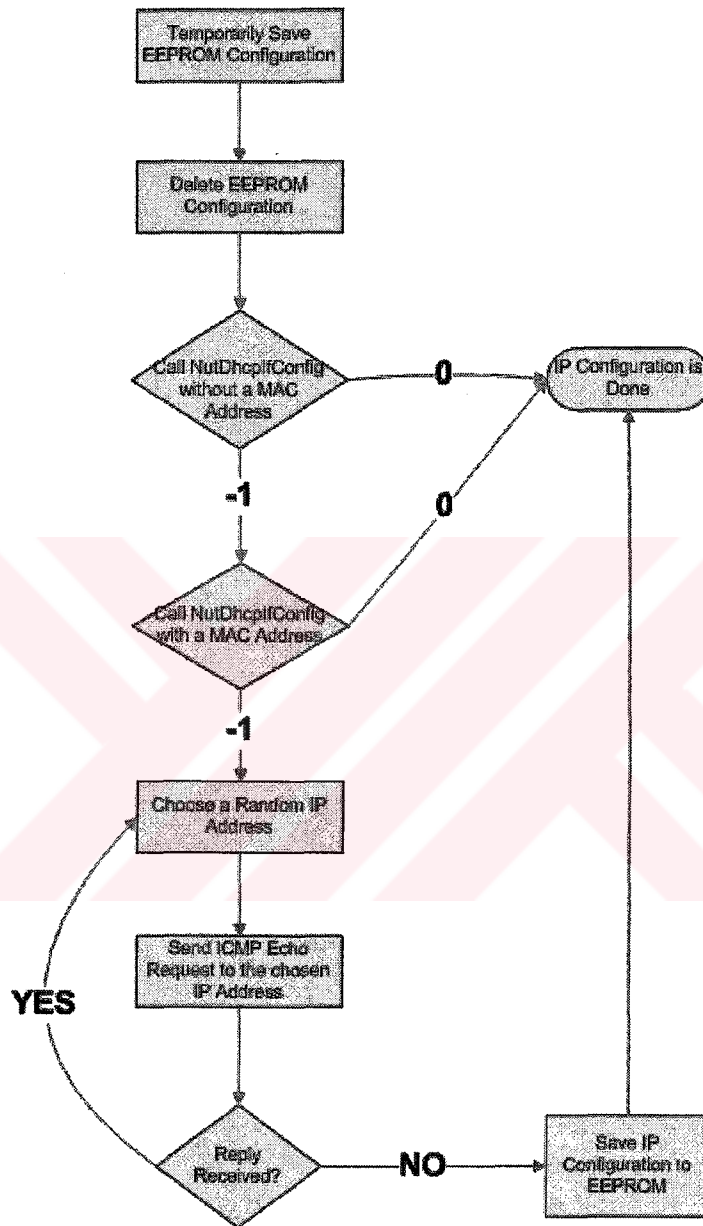


Figure 6.2: Automatic network configuration

As soon as the connection is established, the EHS needs to configure the PPP network device.

```
rc = NutNetIfConfig("ppp", 0, 0, 0);
```

Usually, parameters are set to zero except the first one. This is because the PPP server on the remote side automatically provides these parameters.

Finally, the authentication protocol is initiated. Ethernut supports PPP Authentication Protocol (PAP) [45] only. The drawback of the PAP is that it sends usernames and passwords to the remote host without encryption. If authentication succeeds, Ethernut will accept any local IP address for its PPP network interface.

The problem that remains with the PPP connection is the hanging up. Since the modems simply transport binary data from one end to another, they are not PPP aware. When a party terminates a PPP connection, the modem connection is not terminated. One solution would be the use of escape sequence detection which is supported by most Hayes compatible modems. If a Hayes compatible modem receives three consecutive + characters, they will change their mode to command. So sending ATH command will be enough for hang up. On the other hand, the detection of “+++” works quite rarely.

We use the idle timeout for hanging up. If no traffic is generated within a specified time, the modem automatically hangs up. The drawback of this method is that it causes the connection to stay alive longer than necessary.

The most reliable solution is, using a hardware handshake line. DTR (Data Terminal Ready) is used for this purpose. Unfortunately, Ethernut board does not provide DTR functionality.

After connecting to the Internet, EHS creates a thread which requests a URL with its IP address, *Unique Home Identifier* (UHI) and a password from the *Address Server*. Address Server keeps UHI, IP address and the password for a certain period unless it is renewed.

Users should enter the specified password to get the IP address of the HAN. A sample registration request URL is given below:

`http://www.hanaddressserver.com/register.asp?id=902162738383&ip=212.32.44.122&pass=password`

The number of the phone used by the HAN Server can be chosen as the Unique Home Identifier as used in the previous sample. The details of Address Server can be found in the next section.

The EHS creates another thread for announcing its presence to the appliances that are in the network and for replying Server Discovery Requests. When this thread is created, it broadcasts an online announcement, so that appliances that are already in the network register themselves to the EHS. After the announcement, a UDP socket, which listens for Server Discovery Requests from appliances, is created. When a Server Discovery Request is received from an appliance, EHS sends its IP address to the appliance. This thread repetitively listens for Server Discovery Requests. Moreover, this thread is also responsible for requesting the registration page from the Address Server repeatedly with its UHI, IP address and password. In addition, this thread sends the list of registered appliances to requesting devices.

EHS creates 3 more threads which are dedicated to web requests. Any request coming to port 80 is processed by these threads.

There are 3 important CGI functions in the EHS implementation:

- **RegisterClient:** Processes client registration requests.
- **DeviceList:** Returns a web page displaying the registered appliances.

- **Bridge:** Modifies the links within the web pages of appliances to make these links accessible over the Internet. Bridge function processes appliances' web pages only when they are requested from the Internet.

The EHS keeps an array of structures which contain MAC address, appliance definition, IP address and the registration time. When an appliance is plugged into the network, it registers itself with the EHS by calling *RegisterClient* CGI function. It sends its MAC Address, identifier string and IP address. The identifier contains a descriptive name of the appliance such as "The Internet Clock". The EHS keeps track of the registration time. The EHS sets an appliance's status to offline after a certain time unless the registration is renewed. This leasing mechanism is crucial for the robust behavior of the system in case of appliance connection failures. A sample browser view of *DeviceList* CGI output is given in Figure 6.3.

If a HAN user requests the list of appliances that are registered to the EHS, he/she will call the *DeviceList* CGI function. *DeviceList* CGI function constructs a web page containing links to appliances' main web pages by using the identifiers and IP addresses of appliances. The status of every appliance is checked and updated during processing. If an appliance has not renewed its registration for a certain amount of time, the hyperlink to that appliance will be disabled and the appliance is reported as offline.

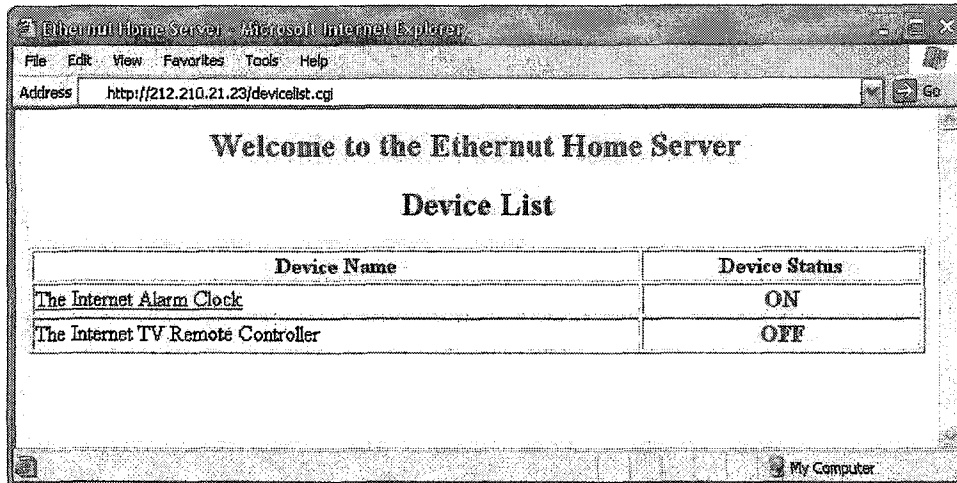


Figure 6.3: Device List - Only one of two devices is accessible. The Internet TV Remote Controller has not renewed its registration. Therefore it is offline.

If a user clicks on one of the hyperlinks in the main page of the EHS, the *Bridge* CGI function will open a TCP socket to the target appliance and receive the appliance's main web page through this socket. Before sending the web page to the requester, the *Bridge* CGI function modifies the source of the web page such that every hyperlink and form target points to the *Bridge* CGI function on the EHS with a different URL parameter containing the original link on the appliance's web server. If the web page contains images, the *Bridge* CGI function cannot transfer these images because of the memory limitations. Therefore, images are removed by the *Bridge* CGI function. If the requester clicks one of these hyperlinks, the *Bridge* CGI function opens a TCP socket and performs actions on the target device according to the URL parameters and retrieves the resulting web page. Once more, it modifies the web page so that later requests to the appliance are directed over the *Bridge* CGI function. A sample modified web page source code is given in Section 4.1.

The complete code of the Ethernet Home Server can be found in Appendix C.

6.2.2 The Address Server

A Server with a static IP is necessary in the Home Appliance Network architecture since in every PPP connection, EHS may get a different IP address which must be known by the HAN user to reach the HAN over the Internet.

The Address Server contains two web pages built with ASP.NET (Active Server Pages). The first page is for Ethernut Home Servers. They register their Home Appliance Networks by requesting the *register.aspx* residing in the Address Server with their Unique Home Identifiers, IP addresses and passwords. The registration information is kept in an SQL database.

The Registration thread running on the EHS should repeatedly request the registration page from the Address Server since the Address Server keeps EHS information for a certain period unless it is renewed.

The other web page residing in the Address Server is the query page (*login.aspx*) (See Figure 6.4a). A HAN user should enter his/her UHI and access password to query the IP address of his/her HAN. If the HAN is found in the database, the Address Server checks the password. If the password entered by the user and the password in the database match and the status of the HAN is online, the user will automatically be directed to his/her EHS. (See Figure 6.4b)

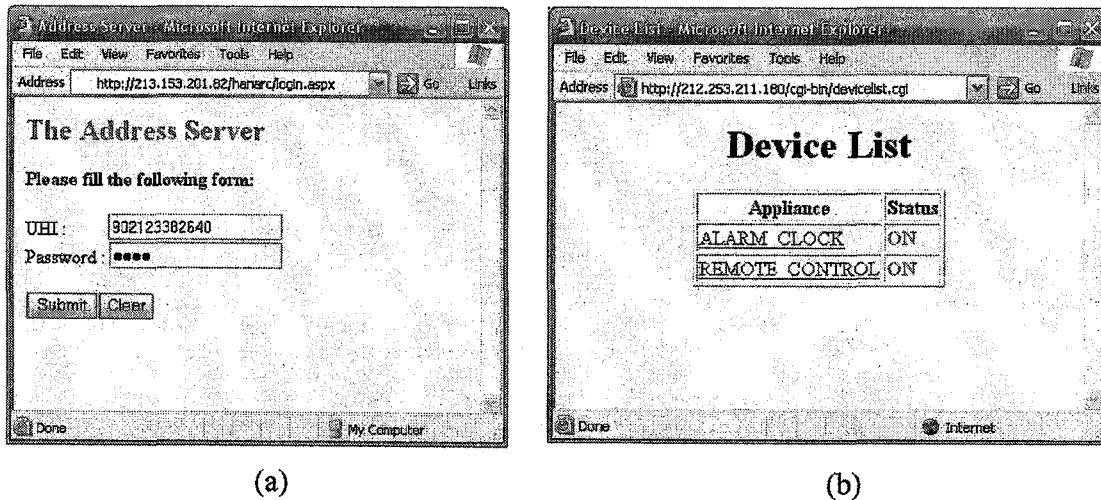


Figure 6.4: Connecting to the EHS via the Address Server - (a) The User queries if his/her HAN is connected to the Internet by entering UHI and password. (b) The HAN is connected to the Internet. Therefore, the user is redirected to his/her EHS's home page.

6.2.3 The Internet Alarm Clock

The Internet Alarm Clock is programmed with assembly. It displays the current time and date on an LCD. It alarms when the set alarm time matches with the current time. The alarm clock has no buttons to set it. It is set via the web interface and can work independently after it is set. The assembly code can be found in Appendix D.

The web interface of the Internet Alarm Clock contains four web pages. The first page is used to set the time via *setclock.cgi* (See Figure 6.5a). The second web page is used to set the alarm and configure the action that is performed on the alarm time via *setalarm.cgi* (See Figure 6.5b). Third and forth pages are for device coordination. They are used to configure device coordination in two steps. In the first step, user selects an appliance from a list of online devices which is obtained from the EHS (See Figure 6.6a). In the second

step, services of the appliance which is selected in the first step are displayed as a list (See Figure 6.6b). This list is obtained from the appliance itself. User selects a service which will be invoked at the alarm time by the alarm clock by calling the related CGI function.

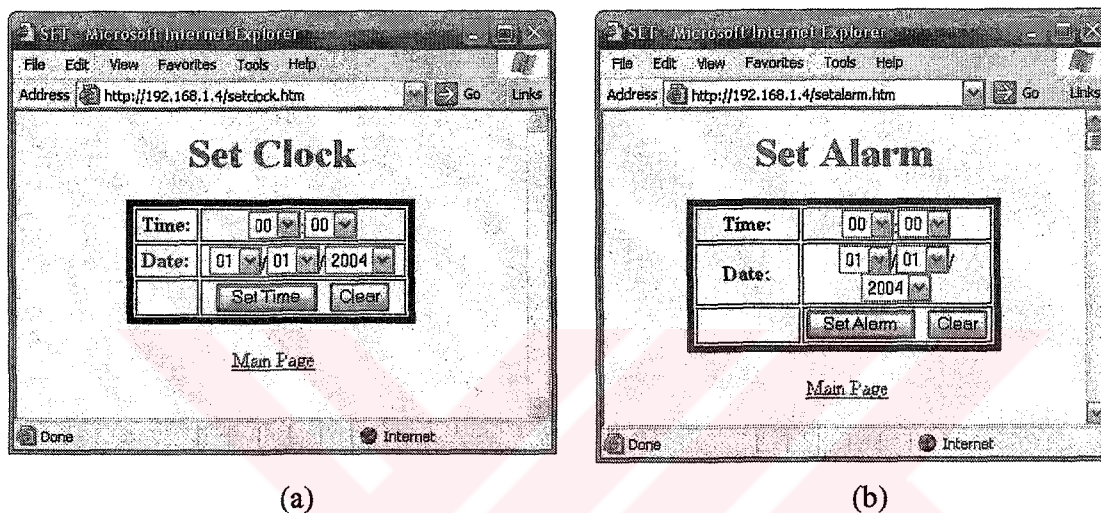


Figure 6.5: Setting the clock - (a) The user enters the time and date information. After submitting the form, *setclock.cgi* function sends the information to the clock via PORT B. (b) The user enters the alarm time and date information. After submitting the form, *setalarm.cgi* function sends the information to the clock via PORT B.

The *setclock.cgi* function sends the time information read from the user to the PORT B of alarm clock. The *setalarm.cgi* function sends the alarm time information to the PORT B of the alarm clock. The *getApplianceList* function obtains the list of online appliances from the EHS. The *getServiceList* function takes target appliance's IP address as a parameter and obtains the list of services offered by the appliance.

The Internet Alarm clock creates 4 threads which are dedicated to web requests. Any request coming to port 80 is processed by these threads.

In addition, a *Server Query* thread is created which is responsible for locating the EHS and registering itself with the EHS repeatedly. Moreover, a *Device Coordination* thread is created which is responsible for calling the appropriate service of the target device at the alarm time. It checks alarm status and if the alarm is on, it calls the pre-selected service (makes an HTTP request to a CGI page) of the target appliance. Finally, a *Service List* thread which is responsible for sending its list of services to requester appliances is created.

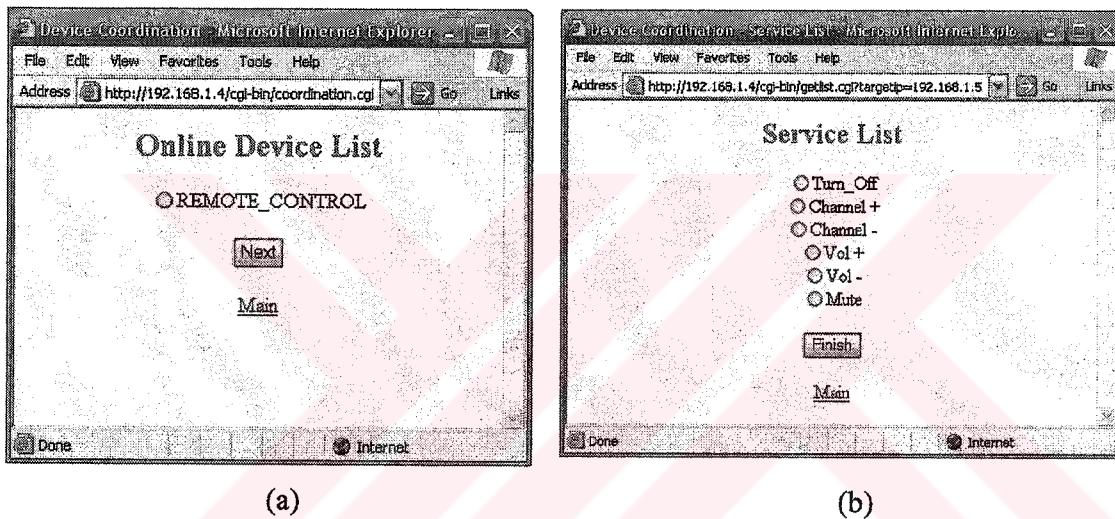


Figure 6.6: Device Coordination - (a) The user selects an appliance from the online device list which is obtained from the EHS (b) The user selects a service from the service list which is obtained from the target device.

The complete source code of the Internet Alarm Clock Web Server can be found in Appendix E.

6.2.4 The Internet TV Remote Controller

The Internet TV Remote Controller sends Manchester encoded [37] infrared pulses to control RC5 [25] compliant devices. The assembly code consists of 2 parts. The first part gets the command to send from the Ethernet board. In the second part, modulation of the output is generated according to the command read in the first part.

To ensure immunity to interference from other IR sources such as the sun and lamps, bi-phase encoding (Manchester) is used for RC5 code words. As shown in Figure 6.7, each bi-phase encoded bit is a symbol comprising two logic levels with a transition in the middle.

In RC5, bi-phase code words modulate a 36kHz carrier, before being transmitted via IR LED. Since the repetition period of 36kHz carrier is 27.778us and the duty factor is 25%, the carrier pulse duration is 6.994us. The high part of each bit in RC5 code word contains 32 carrier pulses. Therefore, 1 bit period is $64 \times 27.78\mu\text{s} = 1.778\text{ms}$.

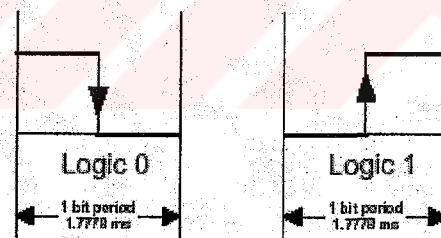


Figure 6.7: Bi-phase encoded bits

A complete RC5 code word contains 14 bits, so it takes 24.889ms to transmit. Each 14 bit RC5 code word consists of:

- 1 START bit (always logic 1)
- 1 FIELD bit (denotes command codes between 0 to 63 and 64 to 127)

- 1 CONTROL bit (toggles after each key release and initiates a new transmission)
- 5 SYSTEM ADDRESS bits (selects one of 32 possible systems)
- 6 COMMAND bits (selects one of 64 possible RC5 commands)

Figure 6.8 contains the flow diagram of the routine used for sending out an RC5 code word. The program starts with sending a bit counter to 14 and sending the start bit. Startmod routine is used to send 36 kHz modulated bits. After transmitting the start bit, the next 13 bits of the RC5 code word are transmitted, using the Manchester encoding. According to the RC5 specification, a delay of 889 microseconds is programmed using Timer0. Then, the next two bits of the RC5 code word are checked to decide if a short (modulated) pulse, a long pulse or another extra delay should follow. When the bit counter reaches zero, the program returns back to the main loop and the chip enters the power down mode again.

The complete assembly source code can be found in Appendix F.

The web server of the Internet TV Remote Controller contains a simple web page for selecting the command to send and CGI functions sending the command to the remote controller via PORT D. The *getApplianceList* function obtains the list of online appliances from the EHS. The *getServiceList* function takes target appliance's IP address as a parameter and obtains the list of services offered by the appliance.

The web server creates 4 threads which are dedicated to web requests. Any request coming to port 80 is processed by these threads.

In addition, a *Server Query* thread is created which is responsible for locating the EHS and registering itself with the EHS repeatedly. Finally, a *Service List* thread which is responsible for sending its list of services to requester appliances is created.

The complete source code of the Internet TV Remote Controller Web Server can be found in Appendix G.

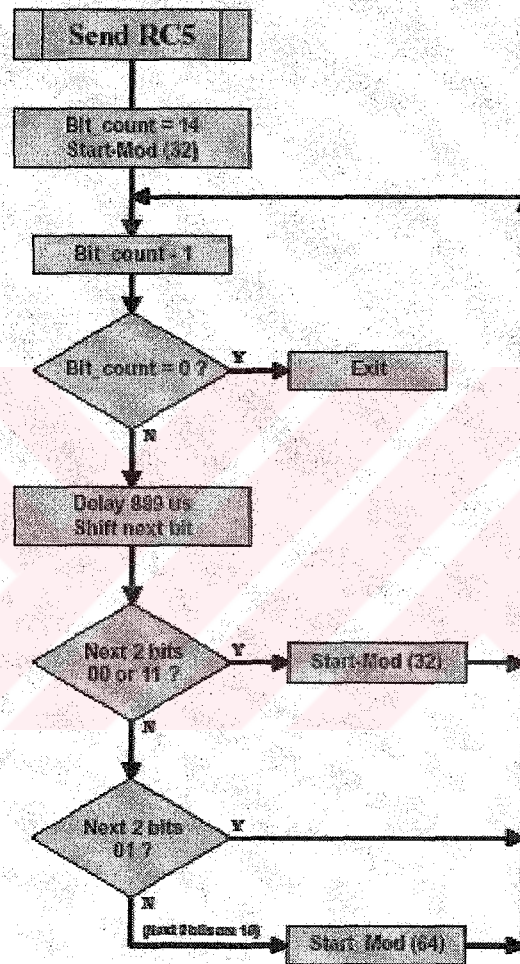


Figure 6.8: Sending an RC5 code word

Chapter 7

CONCLUSIONS

Many home and office appliances produced over the last decade are equipped with embedded microcontrollers. Shortly, embedded systems will become so ubiquitous that they will be an integral part of our life. In other words, almost every object around us will contain an embedded computer system and interacting with them will be unavoidable for everyone.

The Internet has been mostly used to connect general purpose computers so far, but shortly all kinds of appliances with embedded computers may start exchanging information over the Internet.

When ubiquitous embedded systems become real, it will be a challenge to provide a feasible interface for us to interact with these systems that have numerous features.

We designed application architecture and built and demonstrated a prototype system that can integrate web servers to everyday appliances with embedded microcontrollers to control and manage them via web pages using regular web browsers over the Internet. The system allows forming device networks such that their components can easily make use of each others' services and functions.

The unique architecture presented in this thesis is used to create the prototype system which, in turn, used to test and demonstrate the characteristics of our design. The main contributions of this architecture can be summarized as:

- its lightweight design

- automatic configuration and
- utilization of widely available network protocols of TCP/IP and HTTP

7.1 Benefits of the HAN Architecture

Embedded Web Servers approach used in this thesis provides several interesting benefits, which were demonstrated in the prototype system.

Embedded Web Server based HANs provide a general purpose technique, which allows users to employ a single control device with the Internet connection to interact with diverse appliances with embedded systems that he or she may encounter in his or her daily life. Moreover, web enabling appliances provides a method for interfacing to appliances without any client side programming. In addition, it provides platform independence. We based our architecture on powerful and universally available TCP, IP and HTTP protocols instead of designing special protocols for communication and interaction.

Our approach is also flexible, dynamic and adaptive. For instance, manufacturers can easily incorporate different features and functionality in their products or improve user interfaces by designing new web pages and CGI functions. Moreover, manufacturers can perform software updates remotely. They can extend or update appliance interfaces by simply uploading new web pages to appliances' web servers. Accessibility options for helping handicapped users or multi-language features can be added to appliance interfaces.

Our approach places minimal resource requirements on appliances as they must only implement a web server on the existing embedded systems. The approach allows device manufacturers to implement complex functionality, such as value-added services, even for the simplest appliances with microcontrollers.

Designing HAN compatible appliances is simple since the developers of embedded systems can use traditional programming techniques and conventions. Moreover, there is

no need to implement low level components such as TCP/UDP sockets, CGI function handlers, thread manager, DHCP Client or HTTP file system. The APIs are presented completely in the prototype system.

The costs related to user interface design, product documentation, training and support can be reduced considerably with our approach.

Retrofitting existing devices with digital interfacing to a HAN can be achieved easily by application level programming. Devices must have enough digital I/O pins with the same reference voltage so all data could be transmitted in parallel. Moreover, if the microcontroller of a device has enough resources to run a web Server, retrofitting can be done by only re-programming the microcontroller.

The HAN server is probably the most complex component in our architecture. It connects the home appliance network to the Internet enabling HAN components to be accessible over the Internet.

For consumers, configuration of HAN compatible appliances is completely transparent. When an appliance is plugged into the network, it obtains its network configuration automatically. Moreover, appliance interfaces as web pages are easier to interact with since they can contain labels that are more descriptive and certain multi-step tasks can be put on separate web pages where navigation between steps are accomplished by hyperlinks.

7.2 Drawbacks and Weaknesses of the HAN Architecture

One drawback in the HAN architecture is the communication overhead related to the appliance registration. An appliance should register itself with the HAN Server in certain periods of time. In the Client side, a thread, which is dedicated especially for registration, sleeps for a certain period of time. When it wakes up, it opens a TCP socket and makes an HTTP request to the HAN Server. Therefore, there is no need to use an additional thread on

the Server side. Appliance registration process makes use of existing HTTP threads in the HAN Server. Since appliance registration takes place in the local network and amount of information exchanged during registration process is about 1 KB, communication overhead becomes a less serious problem.

The current version of the HAN Server has some inherent hardware weaknesses. Firstly, the number of appliances that can be registered is not scalable enough to be used in environments that contain a very large number of appliances. Currently, the HAN Server supports 10 appliances connected concurrently. The reason for this limitation is the available RAM capacity. (The Ethernut 1.3 comes with 4 KB of Internal RAM and 32KB of external RAM). However, this is clearly only a problem with the current hardware implementation and, by no means, a deficiency in the architecture.

Another implementation problem is with displaying pictures. When a request to an appliance's web page comes from the Internet, a bridge function modifies the links within the web page to make these links accessible over the Internet. If the web page contains images, the bridge function cannot transfer these pictures because of the memory limitations.

The Ethernut needs at least 8 KB of free RAM space to process an HTTP request. When an image is requested via the bridge function, the Ethernut enters low memory condition and all HTTP requests are dropped.

The latest version of the Ethernut PCB, which comes with 512 KB of external RAM, is 2.1. If the HAN Server is imported to the latest version, the memory problems can easily be solved. The HAN Server source code is written compatible with the version 2.1.

7.3 Analysis of the Results

As stated in Section 1.3, the primary goal of this thesis was to design a system that

integrates web servers to everyday appliances to control and manage them via web pages using web browsers. This was described in detail in Chapter 4. The secondary goal of the thesis was to demonstrate a prototype system implementation. This was explained in Chapter 6.

The benefits discussed in Section 7.1 strongly suggest that the ideas presented in this thesis should be evaluated in detail when systems facilitating interaction with ubiquitous embedded systems are designed. Similarly, some of the ideas presented in this thesis may also be useful when dealing with ubiquitous computing in general. Especially, the flexibility, dynamism and adaptability provided by the web pages as interfaces should not be overlooked, as they may be the essential requirements in future ubiquitous computing.

7.4 Personal Reflections

The most difficult task for this thesis was to make decisions about the design and implementation that could greatly effect whether the task can be completed in the time available.

One of the greatest challenges of the project was to design and implement within the limited resources of the Ethernut. The main difficulty came with the lack of RAM and overhead caused by switching between threads.

Another challenge was the unsupported Point-to-Point Protocol. Two years ago, when this project started, the PPP was not supported by the Nut/OS. In the second quarter of 2003, Nut/OS provided the Point-to-Point protocol. On the other hand, the PPP stack was full of bugs and problems. We have contributed operating system's PPP stack by supplying debug feedback. The first PPP connection is established after a tiresome work that continued for one year, in the second quarter of 2004.

The big advantage of the Ethernut is that it allows very cheap production of the end product, fulfilling one of the main objectives of the thesis.

7.5 Future Directions

As discussed earlier, appliances within the HAN are connected to the network by wired (10/100baseT) network technology in the prototype system. Thus, the next step in the future work should be to find ways to provide wireless connections for appliances. Actually, Michael Fisher succeeded in implementing Wifi on an Ethernut Board by adding a PCMCIA interface, which is based on a Xilinx XC95108 and a WLAN Card. Wireless LAN driver and schematics can be found on the Ethernut's web site.

As stated in Chapter 5, Web Servers are designed with a file system in mind. On the other hand, Nut/OS uses a virtual file system where files are located in code ROM. Thus the size of web pages served by the Ethernut Web Server is limited. Future research could also address the integration of a read-only file system which requires additional hardware like an IDE or Compact Flash Interface. The first prototype which combines an IDE and Compact Flash interface on a single add-on board is developed by Egnite. File system is still under development and it is not released for application use.

As argued in Chapter 6, HAN Server connects to the Internet via an external modem which is connected to a serial port, over PSTN. The prototype system may provide the Internet connection via a GPRS modem over GPRS as a future work.

Finally, as mentioned earlier, this thesis focuses on the infrastructure for using web pages as user interfaces. We do not address web page design for better user interfaces. Therefore, the methods of generating user friendly interfaces as web pages may be investigated in the future.

Appendix A

TCP SERVER APPLICATION CODE

```
#include <string.h>
#include <stdio.h>
#include <io.h>

#ifdef ETHERNUT2
#include <dev/lanc111.h>
#else
#include <dev/nicrtl.h>
#endif
#include <dev/debug.h>

#include <sys/version.h>
#include <sys/heap.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <pro/dhcp.h>
#include <sys/confnet.h>

static char buff[128];

/*
 * Constant data is located in the flash ROM.
 * Placing constant data into the flash ROM saves RAM
 * which is scarcer than the flash ROM space.
 */
prog_char banner_P[] = "Welcome to TCP Server. Type help to get help.\r\n";
prog_char help_P[] = "List of commands follows\r\n"
    "m[emory]\t\tQueries number of RAM bytes free.\r\n"
    "q[uit]\t\tTerminates connection.\r\n" ".\r\n";
prog_char mem_fmt_P[] = "%u bytes RAM free\r\n";

/*
 * Process client requests.
 */
void ProcessRequests(FILE * stream)
{
    int got;
    char *cp;
```

```

/*
 * Send a welcome banner.
 */
fputs_P(banner_P, stream);
for (;;) {

    /*
     * Flush output and read a line.
     */
    fflush(stream);
    if (fgets(buff, sizeof(buff), stream) == 0)
        break;

    /*
     * Delete the end of line.
     */
    if ((cp = strchr(buff, '\r')) != 0)
        *cp = 0;
    if ((cp = strchr(buff, '\n')) != 0)
        *cp = 0;

    /*
     * Memory info.
     */
    if (strcmp(buff, "memory", got) == 0) {
        fprintf_P(stream, mem_fmt_P, NutHeapAvailable());
        continue;
    }

    /*
     * Quit connection.
     */
    if (strcmp("quit", buff, got) == 0) {
        break;
    }

    /*
     * Display help text on any unknown command.
     */
    fputs_P(help_P, stream);
}
}

/*
 * Main
 */
int main(void)
{
    TCPSOCKET *sock;
    FILE *stream;
    u_long baud = 115200;
    u_char mac[6] = { 0x00, 0x06, 0x98, 0xB1, 0x01, 0x55 };

    NutRegisterDevice(&DEV_ETHER, 0x8300, 5);

    /*
     * Assign stdout to the UART device.
     */
    freopen("uart0", "w", stdout);

```

```

NutNetLoadConfig("eth0");
memcpy(confnet.cdn_mac, mac, 6);
NutNetSaveConfig();
/*
 * Setup the ethernet device. Try DHCP first. If this is
 * the first time, boot with empty EEPROM and no DHCP server
 * was found, use hardcoded values.
 */
printf("Configure eth0...");
if (NutDhcpIfConfig("eth0", 0, 60000)) {
    printf("initial boot...");
    if (NutDhcpIfConfig("eth0", mac, 60000)) {
        u_long ip_addr = inet_addr("192.168.1.100");
        u_long ip_mask = inet_addr("255.255.255.0");

        printf("no DHCP...");
        NutNetIfConfig("eth0", mac, ip_addr, ip_mask);
    }
}
puts("OK");
printf("IP: %s\n", inet_ntoa(confnet.cdn_ip_addr));
/*
 * Infinite loop for connections.
 */
for (;;) {
    /*
     * Create a socket.
     */
    if ((sock = NutTcpCreateSocket()) != 0) {
        /*
         * Listen on port 23.
         */
        printf("Waiting for a telnet client...");
        if (NutTcpAccept(sock, 23) == 0) {
            puts("connected");
            /*
             * Open a stream and associate it with the socket, so
             * we can use standard I/O
             */
            if ((stream = _fdopen((int) sock, "r+b")) != 0) {
                /*
                 * Process client requests.
                 */
                ProcessRequests(stream);
                puts("Disconnected");
                /*
                 * Close the stream.
                 */
                fclose(stream);
            } else
                puts("Assigning a stream failed");
        } else
            puts("failed");
        /*
         * Close our socket.
         */
        NutTcpCloseSocket(sock);
    }
}

```

Appendix B

WEB SERVER APPLICATION CODE

```
#define MYMAC    0x00, 0x06, 0x98, 0x12, 0x34, 0x56
#define MYIP     "192.168.192.100"
#define MYMASK   "255.255.255.0"

#include <string.h>
#include <io.h>

#ifdef ETHERNUT2
#include <dev/lanc111.h>
#else
#include <dev/nicrtl.h>
#endif
#include <ctype.h>
#include <string.h>
#include <dev/debug.h>
#include <dev/urom.h>
#include <stdlib.h>
#include <sys/version.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/heap.h>
#include <sys/confnet.h>
#include <sys/socket.h>
#include <strtok_r.h>
#include <arpa/inet.h>

#include <pro/httpd.h>
#include <pro/dhcp.h>

#ifdef NUTDEBUG
#include <sys/osdebug.h>
#include <net/netdebug.h>
#endif

static char *html_mt = "text/html";

static int SetDate(FILE * stream, REQUEST * req) {
    char *c[3];
    char *p[3];
    char *day,*month,*year;
    int dayint,monthint,yearint,dayinttemp=0,monthinttemp=0,yearinttemp=0;
```

```

/*
 * Constant data is located in the flash ROM.
 * Placing constant data into the flash ROM saves RAM
 * which is scarcer than the flash ROM space.
 */
static prog_char html_head[] = "<HTML><BODY><BR><H3>The Date has been set</H3>";
static prog_char html_body[] = "<p><a href=\"/index.html\">return to
main</a></BODY></HTML></p>";

/* Send HTML header. */
fputs_P(html_head, stream);

if (req->req_query) {
    u_char i;

    /* Extract 3 parameters. */
    for (i = 0; i < 3; i++) {
        if(i == 0) {
#ifdef __IMAGECRAFT__
            /*
             * Extract the parameters.
             */
            c[0] = strtok(req->req_query, "=");
#else
            /*
             * There's no strtok in the GCC library.
             */
            c[0] = strtok_r(req->req_query, "=", &qp1);
#endif
        }
        else {
#ifdef __IMAGECRAFT__
            c[i] = strtok(0, "=");
#else
            c[i] = strtok_r(0, "=", &qp1);
#endif
        }
    }
#ifdef __IMAGECRAFT__
    p[i] = strtok(0, "&");
#else
    p[i] = strtok_r(0, "&", &qp1);
#endif
    }

    dayint = atoi(p[0]);
    monthint = atoi(p[1]);
    yearint = atoi(p[2]);

    //Set directions of I/O Registers
    outp(0xFC, DDRE);
    outp(0xFF, DDRB);
    outp(0x03, DDRD);

    //set day
    dayinttemp = dayint & 3;
    outp(dayint+128, PORTE);
    outp(dayinttemp, PORTD);

```

```

    outp(0x1D, PORTB);
    delay(900);
    cbi(PORTB,4);
    cbi(PORTB,5);
    cbi(PORTB,3);
    delay(100);
    outp(0x00,PORTB);
    outp(0x00,ORTE);

    //set month
    monthinttemp = monthint & 3;
    outp(monthint+128,ORTE);
    outp(monthinttemp,PORTD);
    outp(0x1E, PORTB);
    delay(900);
    cbi(PORTB,4);
    cbi(PORTB,5);
    cbi(PORTB,3);
    delay(100);
    outp(0x00,PORTB);
    outp(0x00,ORTE);

    //set year
    yearinttemp = yearint & 3;
    outp(yearint+128,ORTE);
    outp(yearinttemp,PORTD);
    outp(0x13, PORTB);
    delay(900);
    cbi(PORTB,4);
    cbi(PORTB,5);
    cbi(PORTB,3);
    delay(100);
    outp(0x00,PORTB);
    outp(0x00,ORTE);

    outp(0xFF,PORTB);
    outp(0xFF,ORTE);
    outp(0x00, DDRE);
    outp(0x00, DDRD);
    outp(0x00, DDRB);

    //Send the HTML section of the CGI after processing the request
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);
    fputs_P(html_head, stream);

    fprintf(stream, "<h4><b>New Settings:</b></h4>");
    fprintf(stream, "<b>Date:</b> %.2d/%.2d/20%.2d<br>", dayint, monthint, yearint);
    fputs_P(html_body, stream);
    fflush(stream);
    return 0;
}

```

```
/*
 * The endless loop in this thread waits for a client connect,
 * processes the HTTP request and disconnects.
 */
THREAD(Service, arg)
{
    TCPSOCKET *sock;
    FILE *stream;
    u_char id = (u_char) ((u_short) arg);

    /*
     * Now loop endless for connections.
     */
    for (;;) {

        /*
         * Create a socket.
         */
        if ((sock = NutTcpCreateSocket()) == 0) {
            printf("[%u] Creating socket failed\n", id);
            NutSleep(5000);
            continue;
        }

        /*
         * Listen on port 80. This call will block until a connection
         * from a client is received.
         */
        NutTcpAccept(sock, 80);
        printf("[%u] Connected, %u bytes free\n", id, NutHeapAvailable());

        /*
         * Wait until at least 8 kByte of free RAM is available. This will
         * keep the client connected in low memory situations.
         */
        while (NutHeapAvailable() < 8192) {
            printf("[%u] Low mem\n", id);
            NutSleep(1000);
        }

        /*
         * Associate a stream with the socket so
         * we can use standard I/O calls.
         */
        if ((stream = _fdopen((int) sock, "r+b")) == 0) {
            printf("[%u] Creating stream device failed\n", id);
        } else {
            /*
             * This API call parses the client's HTTP request,
             * send any requested file from the
             * registered file system or handle CGI requests by calling
             * our registered CGI routine.
             */
            NutHttpProcessRequest(stream);

            /*
             * Destroy the virtual stream device.
             */
            fclose(stream);
        }
    }
}
```

```
        /*
         * Close our socket.
         */
        NutTcpCloseSocket(sock);
        printf("[%u] Disconnected\n", id);
    }
}

/*
 * Main
 */
int main(void)
{
    u_long baud = 115200;
    u_char i;

    /*
     * Initialize the uart device.
     */
    NutRegisterDevice(&devDebug0, 0, 0);
    freopen("uart0", "w", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    NutSleep(200);

    /*
     * Register Ethernet controller.
     */
    if (NutRegisterDevice(&DEV_ETHER, 0x8300, 5))
        puts("Registering device failed");

    /*
     * LAN configuration using EEPROM values or DHCP/ARP method.
     * If it fails, fixed values are used.
     */
    if (NutDhcpIfConfig("eth0", 0, 60000)) {
        u_char mac[] = { MYMAC };
        u_long ip_addr = inet_addr(MYIP);
        u_long ip_mask = inet_addr(MYMASK);

        puts("EEPROM/DHCP/ARP config failed");
        NutNetIfConfig("eth0", mac, ip_addr, ip_mask);
    }
    printf("%s ready\n", inet_ntoa(confnet.cdn_ip_addr));

    /*
     * Register our device for the file system.
     */
    NutRegisterDevice(&devUrom, 0, 0);

    /*
     * Register the CGI sample, which display interesting
     * system informations.
     */
    NutRegisterCgi("setdate.cgi", SetDate);

    /*
     * Protect the cgi-bin directory with
     * username and password.
     */
}
```

```
NutRegisterAuth("cgi-bin", "username:password");

/*
 * Start four server threads.
 */
for (i = 1; i <= 4; i++) {
    char *thname = "httpd0";

    thname[5] = '0' + i;
    NutThreadCreate(thname, Service, (void *) (u_short) i, 640);
}

NutThreadSetPriority(254);
for (;;)
    NutSleep(60000);
}
```

Appendix C

ETHERNUT HOME SERVER SOURCE CODE

```
/*  
***** Written By M. Can Filibeli *****  
*****  
*/  
  
/*  
* HanID and Password Declarations  
* HAN Server registers to Address  
* Server with these values  
*/  
#define HANID "902123382640"  
#define PASS "1826"  
#define ADDRESSSERVERIP "213.153.201.82"  
/*  
* Network device settings.  
*/  
#define MACADDRESS {0x00, 0x06, 0x98, 0x11, 0x02, 0x33}  
#define BASENETWORKADDRESS "10.10.10.0"  
#define BASENETWORKMASK "255.255.255.0"  
#define RANDOMIZATIONCONSTANT 10  
#define SERVERUDPPORT 10005  
#define CLIENTUDPPORT 10006  
#define SERVERTCPPORT 10015  
#define CLIENTTCPPORT 10016  
#define NUMBEROFAPPLIANCES 10  
  
/*  
* PPP Device Settings  
*/  
/* Username for ISP Authentication */  
#define PPPUSER "filipa"  
/* Password for ISP Authentication */  
#define PPPPASS "lerzancan"  
/* Modem Chat String - Contains AT Commands to dial an ISP */  
/* 08222630000 is the ISP's phone number */  
#define PPPCHAT "' AT OK ATEOV1 OK ATE0 OK ATD93990000 CONNECT"  
/* PPP Device Driver */  
#define PPPDEV devUart0  
/* PPP Physical device to send and receive chat script/data over the modem */  
#define PPPCOM "uart0"  
/* UART device's Baudrate value */  
#define PPPSPEED 115200  
/* UART device's character receive timeout value */  
#define PPPRXTO 1000
```

```

/*
 * Debug output device settings.
 */
/* Debug Device Driver */
#define DBGDEV      devDebug1
/* Physical device to send debug information to a terminal */
#define DBGCOM      "uart1"
/* UART device's Baudrate value */
#define DBGSPPEED   115200

#define REGISTRATIONEXPIRESINSECONDS 120
#include <io.h>
#include <string.h>
#include <netdb.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <netdb.h>
#include <fcntl.h>
#include <dev/debug.h>
#include <dev/urom.h>
#include <dev/hd44780.h>
#include <dev/ahdlcavr.h>
#include <dev/uartavr.h>
#include <dev/ppp.h>
#include <dev/chat.h>
#include <sys/nutconfig.h>
#include <sys/version.h>
#include <sys/thread.h>
#include <sys/event.h>
#include <sys/timer.h>
#include <sys/heap.h>
#include <sys/confnet.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <net/if_var.h>
#include <net/route.h>
#include <pro/httpd.h>
#include <pro/dhcp.h>
#include <netinet/icmp.h>

#ifdef ETHERNUT2
#include <dev/lanc111.h>
#else
#include <dev/nicrtl.h>
#endif

#ifdef NUTDEBUG
#include <sys/osdebug.h>
#include <net/netdebug.h>
#endif

static char *html_mt = "text/html";

/*
 * Structure for Appliance information:
 *
 * Appliance information contains
 * its mac address, definition, ip

```

```

    * address and the last time it
    * registers itself with the HAN Server.
    */
struct Appliance{
    char mac[12];
    char definition[15];
    u_long ip;
    long last_on;
};

/* Holds the network address of the device */
u_long BaseNetworkIP;
/* Holds the network mask of the device */
u_long BaseNetworkMASK;
/* Holds the Network IP address of the device */
u_long NetworkIP;
/* Holds the PPP IP address of the device */
u_long pppLocalIP = 0;

struct Appliance *aptable;
int LastRegistered = -1;

/*
 * Function prototypes
 */
void replace(char* buff, char* seekStr, int seekLen, char* replaceStr, int replaceLen);
int find(char* buff, char *seekStr, int seekLen);
void writestream(char* buff, FILE* stream);

/*
 * When a user requests a web page residing on an
 * appliance, this function opens a TCP socket to
 * the target appliance and receives the HTML
 * description of the appliance's main web page
 * through this socket. Before sending the web
 * page to the requester, the CGI function
 * modifies the source of the web page such that
 * every hyperlink, form target, and image
 * address points to the this function (bridge.cgi)
 * on the EHS with a different URL parameter containing
 * the original link on the appliance's web server.
 */
static int Bridge(FILE * stream, REQUEST * req) {
    char *c[10];
    char *p[10];
    char temppl[75];
    static u_char bufflocal[768];
    char tagaddress[100] = "href=\"http://";
    char tagaction[100] = "action=\"http://";
    char hidden[200] = "input type=\"hidden\" name=\"ip\" value=\"";
    char newaddress[125];
    u_char numParams;
    int bite,k=0;
    size_t len;
    size_t rc;
    u_long rip;
    TCPSOCKET *socknew;

    /*
     *The special type 'prog_char' forces the

```

```

* string literals to be placed in flash ROM. This saves us a lot of
* precious RAM.
*/
static prog_char head[] = "<html><head><meta http-equiv=\"Content-Type\"
content=\"text/html; charset=windows-1252\"><title>Parameter Error!</title></head><body>";
static prog_char bodyparam[] = "<p><span style=\"background-color: #FFFFFF\"><font
size=\"6\" color=\"#FF0000\"><b>Error!</b></font></span></p><p><span style=\"background-
color: #FFFFFF\"><b>Some or all parameters for bridging is corrupt.
</b></span></p></body></html>";
static prog_char networkunreachable[] = "<p><span style=\"background-color:
#FFFFFF\"><font size=\"6\" color=\"#FF0000\"><b>Error!</b></font></span></p><p><span
style=\"background-color: #FFFFFF\"><b>Bridging cannot be done. The requested network is
unreachable.</b></span></p></body></html>";
static prog_char hostunreachable[] = "<p><span style=\"background-color:
#FFFFFF\"><font size=\"6\" color=\"#FF0000\"><b>Error!</b></font></span></p><p><span
style=\"background-color: #FFFFFF\"><b>Requested host is
unreachable.</b></span></p></body></html>";

/*
* Extract up to 10 request parameters
* The first request parameter is the ip address of
* the ethernut server which is in the ethernet. The
* second parameter is the url of the web page that
* we want to get. Other parameters are optional and
* they usually exist for web pages that contain
* cgi scripts
*/
if (req->req_query) {
#ifdef __IMAGECRAFT__
    char *qp1;
#endif
    /* Extract parameters. */
    for (numParams = 0; numParams < 10; numParams++){
        if(numParams == 0) {
#ifdef __IMAGECRAFT__
            /*
            * Extract the parameters. Note, that it is potentially
            dangerous
            problem
            switch
            * to use strtok in multithreaded applications. There's no
            * here, because we can be sure that there will by no thread
            * between the first and the last call to strtok. Otherwise we
            * must use strtok_r().
            */
            c[0] = strtok(req->req_query, "=");

#else
            /*
            * There's no strtok in the GCC library. So we take the chance
            to
            * demonstrate strtok_r().
            */
            c[0] = strtok_r(req->req_query, "=", &qp1);
#endif
        }
        else {
#ifdef __IMAGECRAFT__
            c[numParams] = strtok(0, "=");
#else
            c[numParams] = strtok_r(0, "=", &qp1);
#endif
        }
    }
}
}

```

```

#endif
        }
        /* If the parameter is empty then exit from
         * parameter extraction process. Else, continue
         * with the next parameter.
         */
        if(!c[numParams]){
            break;
        }
#ifdef __IMAGECRAFT__
        p[numParams] = strtok(0, "&");
#else
        p[numParams] = strtok_r(0, "&", &qp1);
#endif
    }

    }

    /* Check if the first two parameter name are ip and url respectively */
    if((strcmp(c[0], "ip") != 0) || (strcmp(c[1], "url") != 0)){
        /* These useful API calls create a HTTP response for us. */
        NutHttpSendHeaderTop(stream, req, 200, "Ok");
        NutHttpSendHeaderBot(stream, html_mt, -1);
        fputs_P(head, stream);
        fputs_P(bodyparam, stream);
        fflush(stream);
        return 0;
    }

    /* Check if the ip parameter returned have the same network address with
     * the HAN
     */
    if((inet_addr(p[0]) & BaseNetworkMASK) != BaseNetworkIP){
        /* These useful API calls create a HTTP response for us. */
        NutHttpSendHeaderTop(stream, req, 200, "Ok");
        NutHttpSendHeaderBot(stream, html_mt, -1);
        fputs_P(head, stream);
        fputs_P(networkunreachable, stream);
        fflush(stream);
        return 0;
    }

    rip = inet_addr(p[0]);
    strcpy(temppl, p[1]);
    if(numParams > 2){
        strcat(temppl, "?");
        strcat(temppl, c[2]);
        strcat(temppl, "=");
        strcat(temppl, p[2]);
        if (numParams > 3)
        {
            int i;
            for(i=3; i<numParams; i++)
            {
                strcat(temppl, "&");
                strcat(temppl, c[i]);
                strcat(temppl, "=");
                strcat(temppl, p[i]);
            }
        }
    }

```

```

    }
    strcat(hidden, p[0]);
    strcat(hidden, "\\");
    strcat(hidden, "><input type=\"hidden\" name=\"url\" value=\"");

    if ((socknew = NutTcpCreateSocket()) != 0) {
        if (NutTcpConnect(socknew, rip, 80) == 0) {
            int flag=0;

            /* Send HTTP request to the server. NutTcpSend() doesn't
             * guarantee to send out all bytes, thus the loop.
             */
            strcpy(bufflocal, "GET /" );
            if(strstr(temppl, ".cgi"))
            {
                strcat(bufflocal, "cgi-bin/");
                strcat(bufflocal, temppl);
            }
            else{
                strcat(bufflocal, temppl);
            }
            strcat(bufflocal, " HTTP/1.0\r\nAccept: text/plain, text/html,
image/gif, image/jpeg, */*\r\n\r\n");
            len = (int) strlen(bufflocal);
            for (rc = 0; rc < len; rc += bite){
                if ((bite = NutTcpSend(socknew, bufflocal + rc, len - rc)) <=
0){
                    break;
                }
            }

            /*
             * Read server response and send it to the UART.
             */
            strcpy(newaddress, inet_ntoa(pppLocalIP));
            strcat(newaddress, "/cgi-bin/bridge.cgi");
            strcat(tagaction, newaddress);
            strcat(tagaction, "\\");
            strcat(newaddress, "?ip=");
            strcat(newaddress, p[0]);
            strcat(newaddress, "&url=");
            strcat(tagaddress, newaddress);

            /*
             * While getting the response from the remote appliance,
             * we modify the html such that hyperlinks are bridged over
             * the bridge.cgi. Moreover, it replaces <!--h--> tag with hidden
             * form elements that contain the original appliance's IP
             * address and the original action parameter. This version
             * removes images instead of bridging and displaying them because
             * of the memory limitations.
             */
            while ((bite = NutTcpReceive(socknew, bufflocal,
sizeof(bufflocal)/4)) > 0) {
                int actLocal = -1;
                char *findEnd;
                char mainpage[] = "<p align=\"center\"><a href=\""/>HAN
Server's Main Page</a></p></bo";
                bufflocal[bite] = '\0';

```

```

        strlen("href=\"/");
        strlen(tagaddress));

        replace(bufflocal, "href=\"/cgi-bin/", 15, "href=\"/");
        replace(bufflocal, "href=\"/", 7, tagaddress,

        /* Remove Images */
        replace(bufflocal, "<img ", 5, "<", 1);
        /* Modify form target */
        actLocal = find(bufflocal, "action=\"", 8);
        if (actLocal >= 0) {
            char *temp;
            char temp2[sizeof(bufflocal)/4];
            char temp3[25];
            char temp4[20] = " ";
            strncpy(temp2, bufflocal, sizeof(bufflocal)/4);
            actLocal += 8;
            temp = strtok(temp2 + actLocal, "\"");
            strcpy(temp4, temp);
            strcat(hidden, &temp4[9]);
            strcat(hidden, "\"");
            strcpy(temp3, "action=\"");
            strcat(temp3, temp);
            strcat(temp3, "\"");
            replace(bufflocal, temp3, strlen(temp3), tagaction,
strlen(tagaction));
        }
        /* Add IP address and URL as hidden form fields */
        replace(bufflocal, "!-h--", 6, hidden, strlen(hidden));
        /* Add a link to EHS's Home Page */
        replace(bufflocal, "</bo", 4, mainpage, strlen(mainpage));
        findEnd = strstr(bufflocal, "</html>");
        if (findEnd) {
            flag = 1;
        }
        findEnd += 7;
        *findEnd = '\0';
        fprintf(stream, bufflocal);
        fflush(stream);
        if (flag == 1) {
            break;
        }
        NutSleep(100);
    }
} else {
    printf("\n Connection refused by the remote");
    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);
    fputs_P(head, stream);
    fputs_P(hostunreachable, stream);
    fflush(stream);
    return 0;
}
printf("Disconnecting %s:%u\n TCP CODE: %d", inet_ntoa(rip), 80,
NutTcpError(socknew));
NutTcpCloseSocket(socknew);
}
NutSleep(100);
return 0;
}

```

```

/*
 * This cgi function displays the list of registered appliances.
 * Appliances should re-register in every 2 minute. If an appliance
 * fails to re-register in 2 minutes, this cgi function do not
 * display a link pointing the appliances. List contains hyperlinks
 * for appliances that have valid registrations. Moreover, if a request
 * this cgi comes over the Internet, all links contain a reference to
 * the bridge.cgi. On the other hand, if a request originates within
 * the network, links do not contain references to the bridge function.
 * The EHS keeps track of the registration time. The EHS sets an
 * appliance status to offline after a certain time unless the
 * registration is renewed.
 */
static int DeviceList(FILE * stream, REQUEST * req) {
    int i = 0;
    long current_time;
    extern TCPSOCKET *tcpSocketList;
    TCPSOCKET *ts;
    u_long remoteIP;
    int request_in_network = 0;
    static prog_char head[] = "<HTML><HEAD><TITLE>Device List</TITLE></HEAD>"
        "<BODY><CENTER><H1>Device List</H1>\r\n"
        "<TABLE BORDER><TR><TH>Appliance</TH><TH>Status</TH></TR>\r\n";
    static prog_char tfmt1[] = "<TR><TD>%s</TD><TD>%s";
    static char *device_states[] = { "<FONT COLOR=#CC0000><B>OFF</B></FONT>", "<FONT
COLOR=#339966><B>ON</B></FONT>" };
    static prog_char foot[] = "</TABLE></BODY></HTML>";

    if (LastRegistered == -1) {
        static prog_char head_no_device_registered[] = "<HTML><HEAD><TITLE>Device
List</TITLE></HEAD>"
            "<BODY><CENTER><H1>Device List</H1>\r\n"
            "<BODY><CENTER><H3><FONT COLOR=#CC0000><B>Currently No Device is
registered to the Home Appliance Network</B></FONT></H3>\r\n"
            "</BODY></HTML>";
        NutHttpSendHeaderTop(stream, req, 200, "Ok");
        NutHttpSendHeaderBot(stream, html_mt, -1);
        fputs_P(head_no_device_registered, stream);
        fflush(stream);
        return 0;
    }

    /*
     * Check if the request is originating within the network
     * or not.
     */
    for (ts = tcpSocketList; ts; ts = ts->so_next) {
        if (ts->so_state == TCPS_ESTABLISHED) {
            if ((ts->so_remote_addr & BaseNetworkMASK) == BaseNetworkIP) {
                request_in_network = 1;
            }
            else {
                request_in_network = 0;
                remoteIP = pppLocalIP;
            }
            break;
        }
    }
}

```

```

NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBot(stream, html_mt, -1);
fputs_P(head, stream);

for(i=0; i<=LastRegistered ; i++){
    char link[150] = "<a href=\"http://\"";
    time(&current_time);
    if(((current_time - aptable[i].last_on) < REGISTRATIONEXPIRESINSECONDS)){
        if(request_in_network == 0){
            strcat(link, inet_ntoa(remoteIP));
            strcat(link, "/cgi-bin/bridge.cgi?");
            strcat(link, "ip=");
            strcat(link, inet_ntoa(aptable[i].ip));
            strcat(link, "&url=/">");
        }
        else{
            strcat(link, inet_ntoa(aptable[i].ip));
            strcat(link, ">");
        }
        strcat(link, aptable[i].definition);
        fprintf_P(stream, tfmt1, link, device_states[1]);
        fputs("</a></TD></TR>\r\n", stream);
        fflush(stream);
    }
    else{
        fprintf_P(stream, tfmt1, aptable[i].definition, device_states[0]);
        fputs("</TD></TR>\r\n", stream);
        fflush(stream);
    }
}
fputs_P(foot, stream);
fflush(stream);
return 0;
}

/*
 * The EHS keeps an array of structures which contain
 * MAC address, appliance definition, IP address and
 * the registration time. When an appliance is plugged
 * into the network, it registers itself with the EHS
 * by calling this CGI function. It sends its MAC Address,
 * identifier string and IP address. The identifier contains
 * a descriptive name of the appliance such as "The
 * Internet Clock".
 */
static int RegisterClient(FILE * stream, REQUEST * req) {
    int i = 0;
    int available = -1;
    char *c[3];
    char *p[3];
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    if (req->req_query) {
#ifdef __IMAGECRAFT__
        char *qp1;
#endif
        u_char i;
        /* Extract 3 parameters. */

```

```

        for (i = 0; i < 3; i++) {
            if(i == 0) {
#ifdef __IMAGECRAFT__
                /*
                * Extract the parameters. Note, that it is potentially
                dangerous
                * to use strtok in multithreaded applications. There's no
                problem
                * here, because we can be sure that there will by no thread
                switch
                * between the first and the last call to strtok. Otherwise we
                * must use strtok_r().
                */
                c[0] = strtok(req->req_query, "=");
#else
                /*
                * There's no strtok in the GCC library. So we take the chance
                to
                * demonstrate strtok_r().
                */
                c[0] = strtok_r(req->req_query, "=", &qpl);
#endif
            }
            else {
#ifdef __IMAGECRAFT__
                c[i] = strtok(0, "=");
            #else
                c[i] = strtok_r(0, "=", &qpl);
            #endif
        }
    }

#ifdef __IMAGECRAFT__
    p[i] = strtok(0, "&");
#else
    p[i] = strtok_r(0, "&", &qpl);
#endif
    }

    if (LastRegistered == -1) {
        LastRegistered = 0;
        strncpy(aptable[LastRegistered].mac,
            p[0], sizeof(aptable[LastRegistered].mac) - 1);
        strncpy(aptable[LastRegistered].definition,
            p[1], sizeof(aptable[LastRegistered].definition) - 1);
        aptable[LastRegistered].ip = inet_addr(p[2]);
        time(&aptable[LastRegistered].last_on);
    }
    else {
        int k = 0;
        for (k = 0; k <= LastRegistered; k++) {
            if (strcmp(aptable[k].mac, p[0], sizeof(aptable[k].mac) - 1) == 0) {
                strncpy(aptable[k].definition,
                    p[1], sizeof(aptable[k].definition) - 1);
                aptable[k].ip = inet_addr(p[2]);
                time(&aptable[k].last_on);
                available = 1;
                break;
            }
        }
        else {

```

```

        available = 0;
        continue;
    }
}

if(available == 0){
    if(LastRegistered == NUMBEROFAPPLIANCES - 1){
        printf("\n\nMAXIMUM NUMBER OF APPLIANCES ALLOWED IS
EXCEEDED...");
        return -1;
    }
    else{
        LastRegistered++;
        strncpy(aptable[LastRegistered].mac,
p[0], sizeof(aptable[LastRegistered].mac) - 1);
        strncpy(aptable[LastRegistered].definition,
p[1], sizeof(aptable[LastRegistered].definition) - 1);
        aptable[LastRegistered].ip = inet_addr(p[2]);
        time(&aptable[LastRegistered].last_on);
    }
}

// Send HTML header.
for(i = 0; i<=LastRegistered; i++){
    //fprintf(stream, "mac: %s definition: %s ip: %s last_on: %ld <BR>",
    aptable[i].mac, aptable[i].definition, inet_ntoa(aptable[i].ip), aptable[i].last_on);
}
fflush(stream);
return 0;
}

/*
 * Mainly, this thread send replies to the server
 * discovery messages coming from appliances. When this
 * thread is created (when the server becomes online)
 * it broadcasts its IP address for the appliances that
 * are in the network. After the online announcement,
 * this thread waits for Server Discovery Messages. In
 * addition, it sends EHS registration information
 * to the Address Server. Moreover, it sends information
 * about registered appliances for device coordination.
 */
THREAD(ServerQueryReplier, arg) {
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buff[35];
    int count = 0;
    sock = NutUdpCreateSocket(SERVERUDPPORT);

    /*
     * Send the IP of the server for appliances which are already in the network
     */
    {
        int k = 0;
        while(k<4){

            if(NutUdpSendTo(sock, inet_addr("255.255.255.255"), CLIENTUDPPORT, "IPA", 3) == 0)
                k++;
            else continue;
        }
    }
}

```



```

else
    continue;
}
sprintf(msg, "?OK=%d", j);
strcat(msg, "&def=");

strcat(msg, aptable[j].definition);

strcat(msg, inet_ntoa(aptable[j].ip));

if(NutUdpSendTo(sock,
ip_addr_from, port_from, msg, sizeof(msg))==0){
    if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buff, 35, 1000) > 0){
        char
        int dev =
        msgrcv[5];
        -1;
        strncpy(msgrcv, buff, 2);
        msgrcv[2]
        = '\0';
        if(strncmp(msgrcv, "OK", 2)==0){
            dev = atoi(&buff[3]);
            = dev + 1;
            if(dev>LastRegistered)
                break;
            else
                continue;
        }
        else{
            failed++;
            if
            (failed==3){
                NutUdpSendTo(sock, ip_addr_from, port_from, "FAILED", sizeof("FAILED"));
                break;
            }
            else
                continue;
        }
    }
    else continue;
    break;
}
else continue;
}
}
else continue;
}
}

```

```

        }
        else{
            printf("\nServer couldn't send a reply to
ALQ");
            continue;
        }
    }
}
/* Register to the Address Server */
else
{
    TCP_SOCKET *socknew;
    u_char buff[120];
    int i,len,rc,bite,total_bytes = 0;
    count = 1;
    socknew = NutTcpCreateSocket();
    if (NutTcpConnect(socknew, inet_addr(ADDRESSSERVERIP),80) == 0)
    {
        printf("\nConnected to The Address Server");
        strcpy(buff, "GET ");
        strcat(buff, "/hanarc/register.aspx?hanid=");
        strcat(buff, HANID);
        strcat(buff, "&pass=");
        strcat(buff, PASS);
        strcat(buff, "&ip=");
        strcat(buff, inet_ntoa(pppLocalIP));
        strcat(buff, " HTTP/1.0\r\nUser-Agent: Ethernut\r\n\r\n");
        len = (int) strlen(buff);
        for (rc = 0; rc < len; rc += bite){
            if ((bite = NutTcpSend(socknew, buff + rc, len - rc))
<= 0){
                break;
            }
        }
        while ((bite = NutTcpReceive(socknew, buff, sizeof(buff) -
1)) > 0) {
            total_bytes += bite;
            buff[bite] = 0;
        }
        printf("...Registered!\n");
    }
    else
    {
        printf("\nAddress Server is death");
        NutTcpDestroySocket(socknew);
        continue;
    }
    NutUdpDestroySocket(sock);
}

/*
 * HTTP service thread.
 * The endless loop in this thread waits for a client connect,

```

```

* processes the HTTP request and disconnects. If one client
* has established a connection, further connect attempts will
* be rejected. Typically browsers open more than one connection
* in order to load images concurrently. So we run this routine by
* several threads.
*
*/
THREAD(Service, arg)
{
    TCPSOCKET *sock;
    FILE *stream;
    u_char id = (u_char) ((uptr_t) arg);

    /*
     * Now loop endless for connections.
     */
    for (;;) {

        /*
         * Create a socket.
         */
        if ((sock = NutTcpCreateSocket()) == 0) {
            printf("[%u] Creating socket failed\n", id);
            NutSleep(5000);
            continue;
        }

        /*
         * Listen on port 80. This call will block until we get a connection
         * from a client.
         */
        NutTcpAccept(sock, 80);
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega103__)
        printf("[%u] Connected, %u bytes free\n", id, NutHeapAvailable());
#else
        printf("[%u] Connected, %lu bytes free\n", id, NutHeapAvailable());
#endif

        /*
         * Wait until at least 4 kByte of free RAM is available. This will
         * keep the client connected in low memory situations.
         */
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega103__)
        while (NutHeapAvailable() < 4096) {
#else
        while (NutHeapAvailable() < 4096) {
#endif
            printf("[%u] Low mem\n", id);
            NutSleep(1000);
        }

        /*
         * Associate a stream with the socket so we can use standard I/O calls.
         */
        if ((stream = _fdopen((int) ((uptr_t) sock), "r+b")) == 0) {
            printf("[%u] Creating stream device failed\n", id);
        } else {
            /*
             * This API call saves us a lot of work. It will parse the
             * client's HTTP request, send any requested file from the

```

```

        * registered file system or handle CGI requests by calling
        * our registered CGI routine.
        */
        NutHttpProcessRequest(stream);

        /*
        *
        */
        fclose(stream);
    }
    /*
    * Close our socket.
    */
    NutTcpCloseSocket(sock);
    printf("[%u] Disconnected\n", id);
}

/*
 * Randomly chooses an IP address based on the previously
 * defined BASENETWORKADDRESS.
 */
void Randomize(u_char *mac)
{
    for(;;)
    {
        u_long cpu_crystal = NutGetTickCount();
        u_long ip;
        int ip_last;

        NETBUF *nb = NutNetBufAlloc(0, NBUF_APPLICATION, 50);

        /*
        * An IP address is chosen randomly using the tick count of
        * the cpu crystal.
        */
        ip_last = (int) ((cpu_crystal + RANDOMIZATIONCONSTANT) % 253) + 2;
        ip = inet_addr(BASENETWORKADDRESS) + 16777216 * ip_last;

        NutNetIfConfig("eth0", mac, ip, inet_addr(BASENETWORKMASK));

        /*
        * ICMP echo request message is sent.
        */
        if(NutIcmpOutput(8, ip, nb) == -1)
        {
            NutNetBufFree(nb);
            break;
        }
        else
        {
            NutNetBufFree(nb);
            continue;
        }
    }
}

/*
 * automatically determines the network settings
 * for our network.

```

```

*/
void AutoIp(u_long old_ip, u_long old_mask, u_char *mac)
{
    /*
     * we check whether the previously stored IP address is valid for
     * the network we are in. If the previously stored IP address
     * is not valid for our network, we randomly choose
     * an IP address based on the predefined BASENETWORKIP. If the
     * previously stored IP address is valid for our network, first, we
     * check if the IP address is in use in our network. If yes, we
     * randomly choose another IP address. If no, we used the IP address
     * to configure our Ethernet Home Server.
     */
    if((old_ip & inet_addr(BASENETWORKMASK)) != inet_addr(BASENETWORKADDRESS))
    {
        Randomize(mac);
    }
    else
    {
        NETBUF *nb = NutNetBufAlloc(0, NBUF_APPLICATION, 50);
        Nutsleep(1000);
        NutNetIfConfig("eth0", mac, old_ip, old_mask);

        /*
         * ICMP echo request message is sent.
         */
        if(NutIcmpOutput(8, old_ip, nb) == 0)
        {
            NutNetBufFree(nb);
            Randomize(mac);
        }
        else
        {
            NutNetBufFree(nb);
        }
    }
    BaseNetworkIP = (confnet.cdn_ip_addr & confnet.cdn_ip_mask);
    BaseNetworkMASK = confnet.cdn_ip_mask;
    NetworkIP = confnet.cdn_ip_addr;
}

/*
 * This function is used to edit original HTML source
 * received by the bridge function.
 * bufflocalreplace: string which contains the seekStr
 * seekStr: string to be replaced
 * seekLen: size of seekStr
 * replaceStr: string which will be inserted instead of seekStr
 * replaceLen: size of replaceStr
 */
void replace(char* bufflocalreplace, char* seekStr, int seekLen, char* replaceStr, int
replaceLen) {
    char c;
    int len;
    int curPos = 0;
    int pos;
    int newPos = 0;

    char newBuff[768];
    char* buffTemp = bufflocalreplace;

```

```

do{
    pos = find(buffTemp, seekStr, seekLen);

    if(pos == -1)
        break;

    /* add the intermediate jurgan */
    strncpy(&newBuff[newPos], buffTemp, pos);

    newPos += pos;

    /* add the replacement string */
    strncpy(&newBuff[newPos], replaceStr, replaceLen);

    newPos += replaceLen;

    buffTemp = &buffTemp[pos + seekLen];

}while(pos != -1);

/* attach rest of the string to the end of our new buffer */
strcpy(&newBuff[newPos], buffTemp);
sprintf(bufflocalreplace, "%s", &newBuff);
}

/*
 * Used to send HTML modified by the replace function
 * over the specified stream object.
 */
void writestream(char* buff, FILE* stream)
{
    fputs(buff, stream);
    fflush(stream);
}

/*
 * Returns the position of seekStr in the buff string
 * buff: String to be searched for the position of seekStr
 * seekStr: Search string
 * seekLen: length of seekStr
 */
int find(char* buff, char *seekStr, int seekLen) {
    char c;
    int len;
    int pos = 0;
    int globalPos = 0;
    char pass = 1;

    while(buff[globalPos] != '\0'){
        c = buff[globalPos];

        /* if first character fits then check the remaining characters */
        if(c == seekStr[0]){
            pos = 1;
            pass = 1;
            /* if buffer size is not exceeded */
            while(buff[globalPos] != '\0' && pass ){
                if(pos < seekLen){

```

```

        c = buff[globalPos + pos];
        if(seekStr[pos] != c)
            pass = 0;
        pos++;
    }
    else
        break;
}

if(pass){
    return globalPos;
}

globalPos++;
}
return -1;
}
/*
 * Main application routine.
 *
 * Nut/OS automatically calls this entry after initialization.
 */
int main(void)
{
    u_long baud = 115200;
    u_char i;
    int pppcom;
    PPPDCB *dcb;
    u_long lctl;
    int rc;

    /* Allocate space for list that holds the appliance information*/
    aptable = (struct Appliance *)malloc(NUMBEROFAPPLIANCES * sizeof(struct Appliance));
    memset(aptable, 0, NUMBEROFAPPLIANCES * sizeof(struct Appliance));

    BaseNetworkIP = inet_addr(BASENETWORKADDRESS);
    BaseNetworkMASK = inet_addr(BASENETWORKMASK);
    /*
     * Initialize the uart device.
     */
    NutRegisterDevice(&DBGDEV, 0, 0);
    freopen(DBGCOM, "w", stdout);

    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    NutSleep(200);
    printf("\n\nEthernut HAN SERVER", NutVersionString());

#ifdef NUTDEBUG
    NutTraceTcp(stdout, 0);
    NutTraceOs(stdout, 0);
    NutTraceHeap(stdout, 0);
    NutTracePPP(stdout, 0);
#endif

    /*
     * Register Ethernet controller.
     */
#ifdef __AVR_ATmega128__ || defined(__AVR_ATmega103__)
    if (NutRegisterDevice(&DEV_ETHER, 0x8300, 5))

```

```

#else
    if (NutRegisterDevice(&devEth0, NIC_IO_BASE, 0))
#endif
        puts("Registering device failed");

/*
 * The Ethernet Home Server tries to retrieve the local IP address,
 * network mask and default gateway from a DHCP server. If no DHCP
 * server responds within a specified number of milliseconds, the
 * Ethernet randomly chooses an IP address based on the predefined
 * BASENETWORKIP.
 */
{
    u_long old_ip;
    u_long old_mask;

    /*
     * Save the last used IP address and IP mask stored
     * in the EEPROM.
     */
    NutNetLoadConfig("eth0");
    old_ip = confnet.cdn_ip_addr;
    old_mask = confnet.cdn_ip_mask;

    /*
     * Clear the last used IP address and IP mask stored
     * in the EEPROM. This is used to guarantee the automatic
     * configuration since the IP address stored in the EEPROM is
     * used when there is no DHCP Server. For example: if we do
     * not clear the EEPROM values and migrate from a network
     * housing a DHCP server to a network without a DHCP server,
     * previously used IP configuration will be used in the
     * new network.
     */
    confnet.cdn_ip_addr = inet_addr("0.0.0.0");
    confnet.cdn_ip_mask = inet_addr("0.0.0.0");
    NutNetSaveConfig();

    /*
     * In the first call to NutDhcpIfConfig, no MAC address is
     * passed. If one is found in EEPROM, then the DHCP server
     * will be queried for the related IP settings and the
     * function returns 0. Otherwise it returns -1, but we don't
     * know, whether the EEPROM does't contain a valid MAC address
     * or DHCP failed. Thus NutDhcpIfConfig is called once again
     * but this time with a hardcoded MAC address. If it fails again
     * then we are sure, that there is no available DHCP Server.
     * Last thing we can do is to pass IP address and IP mask to
     * NutNetIfConfig to get the network interface up and running.
     * Therefore, we dynamically determine IP address values by
     * using AutoIP function.
     */
    if (NutDhcpIfConfig("eth0", 0, 20000)) {
        u_char mac[] = MACADDRESS;
        if (NutDhcpIfConfig("eth0", mac, 20000)) {
            AutoIp(old_ip, old_mask, mac);
        }
    }
}

```

```

        printf("%s ready\n", inet_ntoa(confnet.cdn_ip_addr));
        /*
        * we save the IP configuration to EEPROM.
        */
        NutNetSaveConfig();
    }
    BaseNetworkIP = (confnet.cdn_ip_addr & confnet.cdn_ip_mask);
    BaseNetworkMASK = confnet.cdn_ip_mask;
    NetworkIP = confnet.cdn_ip_addr;
    NutRegisterDevice(&PPPDEV, 0, 0);
    NutRegisterDevice(&devPpp, 0, 0);

    /*
    * Open PPP device. Specify physical device, user and password.
    */
    printf("Open uart...");
    if ((pppcom = _open("ppp:" PPPCOM "/" PPPUSER "/" PPPPASS, _O_RDWR | _O_BINARY)) ==
-1) {
        printf("Failed to open " PPPCOM "\n");
        for (;;)
        }
        puts("done");

        /*
        * Set PPP line speed.
        */
        lctl = PPPSPEED;
        _ioctl(pppcom, UART_SETSPEED, &lctl);

        /*
        * The PPP driver doesn't set any receive timeout, but
        * may require it.
        */
        lctl = PPPRXTO;
        _ioctl(pppcom, UART_SETREADTIMEOUT, &lctl);

#ifdef NUTDEBUG
    /*
    * Optionally enable PPP trace.
    */
    NutTracePPP(stdout, 0);
#endif

    /*
    * This delay may be removed. It is quite helpful during development.
    */
    NutSleep(2000);

    /*
    * PPP connection loop.
    */
    for (;;) {
        /*
        * Connect using a chat script. We may also set any
        * required hardware handshake line at this stage.
        */
        printf("Connecting...");
        if ((rc = NutChat(pppcom, PPPCHAT)) != 0) {
            printf("no connect, reason = %d\n", rc);
            continue;
        }
    }

```

```

    }
    puts("done");

    /*
    * We are connected, configure our PPP network interface.
    * This will initiate the PPP configuration negotiation
    * and authentication with the server.
    */
    printf("Configure PPP...");
    rc = NutNetIfConfig("ppp", 0, 0, 0);
    if (rc != 0) {
        printf("failed: %d",rc);
        /*
        * Optionally toggle DTR to hang up the modem.
        */
        continue;
    }
    puts("done");

    /*
    * Set name server and default route. Actually the PPP interface
    * should do this, but the current release doesn't.
    */
    dcb = devPpp.dev_dcb;
    NutDnsConfig2(0, 0, dcb->dcb_ip_dns1, dcb->dcb_ip_dns2);
    NutIpRouteAdd(0, 0, dcb->dcb_remote_ip, &devPpp);
    pppLocalIP = dcb->dcb_local_ip;
    /*
    * Display our IP settings.
    */
    printf("    Local IP: %s\n", inet_ntoa(dcb->dcb_local_ip));
    printf("    Remote IP: %s\n", inet_ntoa(dcb->dcb_remote_ip));
    printf("    Primary DNS: %s\n", inet_ntoa(dcb->dcb_ip_dns1));
    printf("    Secondary DNS: %s\n", inet_ntoa(dcb->dcb_ip_dns2));

    /*
    * Register our device for the file system.
    */
    NutRegisterDevice(&devUrom, 0, 0);

    /*
    * Protect the cgi-bin directory with
    * user and password.
    */
    //NutRegisterAuth("cgi-bin", "root:root");

    /*
    * Register our CGI functions. This will be called
    * by http://host/cgi-bin/funcname.cgi?anyparams
    */
    NutRegisterCgi("register.cgi", RegisterClient);
    NutRegisterCgi("devicelist.cgi", DeviceList);
    NutRegisterCgi("bridge.cgi", Bridge);

    /*
    * Start three server threads.
    */
    for (i = 1; i <= 3; i++) {
        char *thname = "httpd0";

```

```
        thname[5] = '0' + i;
        NutThreadCreate(thname, Service, (void *) (uptr_t) i, 2048);
    }

    /*
     * Start ServerQueryReplier thread
     */
    NutThreadCreate("QReply", ServerQueryReplier, (void *) (u_short) i, 768);

    /*
     * Set the priority of the main thread to 254.
     * Setting a thread's priority to 254 makes it
     * the idle thread.
     */
    NutThreadSetPriority(254);
    for (;;)
        NutSleep(60000);
```



Appendix D

THE INTERNET ALARM CLOCK SOURCE CODE

```

;*****ONLINE ALARM CLOCK*****
;written by Eda Baykan, M. Can Filibeli
;KOC university
;computer engineering
;*****

; - PORTA is connected to lcd
; - PORTA, Bits 4..7=Data, Bit 1=RS, Bit0=E*

; Subroutines:
; - Lcd4Init: Initialise the LCD
; - Lcd4Chr : Display the character in temp_var on the LCD

;*****
; - PORTB is connected to the Ethernet HAN Client
; - PORTD is (pin5) connected to speaker

;*****
; - Timer0 is used as a chronometer
; - Timer1 and Timer2 are used to play songs

;*****
; - THERE ARE FOUR SONGS
;   - 1->SIMPSON
;   - 2->MISSION
;   - 3->BUMTEST
;   - 0->MOUNTAINKING
; - which_song register determines the song to play
;*****

.include "m16def.inc"

;*****

.org    $000
        rjmp    RESET
.org OC2addr
        rjmp    OC2int
.org OC0addr
        rjmp    OC0int

;*****
;Register definitions

```

```

.def second = r1 ; current second
.def minute = r2 ; current minute
.def hour = r3 ; current hour
.def day = r4 ; current day
.def month = r5 ; current month
.def year = r9 ; current year

.def alarm_day = r6 ; alarm day
.def alarm_minute = r7 ; alarm minute
.def alarm_hour = r8 ; result
.def remainder = r10 ; remainder

.def say = r11
.def duration1 = r12

.def duration_select = r13 ;duration of song is selected
.def which_song = r14

.def time_temp = r26
.def temp = r27
.def counter = r16
.def temp_var = r18

.def tonel = r20
.def toneh = r21
.def dura = r22
.def tone = r23
.def notep1 = r24
.def noteph = r17

.def ten_count = r15
.def hold_pinc = r25
.def tempe = r28
.def can = r29

;stop = 1 -> Alarm is offalarm
;stop = 0 -> Alarm is on
.def stop=r19

;*****

.equ TEMPO =185

; **LCD constants
.equ cLcdWrite=0b11111111 ; Data direction write the LCD
.equ cLcdDummy=0b00111000 ; Dummy-Function-Word
.equ mLcdRs=0b00000010 ; RS-Bit Mask
.equ bLcdEn=0 ; Enable Bit

;*****
;note constants
.equ c = $00
.equ cs = $01
.equ d = $02
.equ ds = $03
.equ e = $04
.equ f = $05
.equ fs = $06

```

```

.equ    g      = $07
.equ    gs     = $08
.equ    a      = $09
.equ    as     = $0a
.equ    b      = $0b
.equ    c2     = $20
.equ    cs2    = $21
.equ    d2     = $22
.equ    ds2    = $23
.equ    e2     = $24
.equ    f2     = $25
.equ    fs2    = $26
.equ    g2     = $27
.equ    gs2    = $28
.equ    a2     = $29
.equ    ash2   = $2a
.equ    b2     = $2b
.equ    p      = $FF
.equ    O0     = $00
.equ    O1     = $10
.equ    O2     = $20
.equ    O3     = $30
.equ    O4     = $40

;*****

.MACRO enactive
    nop
    nop
.ENDMACRO

;*****

RESET:

    ldi    temp,high(RAMEND)
    out    SPH,temp
    ldi    temp,low(RAMEND)
    out    SPL,temp

; - PORT Initializations

;   - PORTB is used to receive day,month,year,hour,minute
;   - alarm day, alarm month, alarm hour and alarm minute
;   - setting via the web server
    ldi temp,0b00000000
    out DDRB,temp

;   - PORTC is used to receive the control selection which
;   - is used to determine which setting will be modified
;   - with the value in PORTB
;   - Moreover, it receives Stop_Alarm signal from the web
;   - server and sends Alarm_is_On signal to the web server
    ldi temp,0b10000000
    out DDRC,temp

;   - PORTD is used to play songs
;   - PIN5 is defined as output.. (OC1A output pin)
;   - PIN5 is connected to the speaker

```

```

        ldi temp,0b11100000
        out DDRD,temp
;        - PORTA is connected to the LCD
        ldi temp,0b11111111
        out DDRA,temp

;*****

;        - Initial Clock settings
        clr temp
        ldi temp,0
        mov hour,temp
        ldi temp,0
        mov minute,temp
        ldi temp,0
        mov second,temp
        ldi temp,4
        mov year,temp
        ldi temp,1
        mov month,temp
        ldi temp,1
        mov day,temp

;        - Register initializations
        clr counter
        clr hold_pinc
        clr tempe
        clr say
        clr remainder
        clr duration1
        clr temp
        clr duration_select
        clr which_song
        clr time_temp
        clr temp_var
        clr alarm_minute
        clr alarm_hour
        clr alarm_day

        ldi temp,0b00000000
        mov remainder,temp
;        - Initial Alarm Settings
        ldi stop,1
        clr ten_count
        ldi temp,0
        mov alarm_hour,temp
        ldi temp,1
        mov alarm_minute,temp
        ldi temp,1
        mov alarm_day,temp

;        - Timer0 Settingd

;        clock 16/8 MHZ
        ldi temp,0b00000010
        out TCCR0,temp
        ldi temp,0
        out TCNT0,temp
;        enable timer 0 output compare match enable,and timer2

```

```

        ldi temp,10000010
        out TIMSK,temp
;      load 200 to compare register
        ldi temp,0b11001000
        out OCR0,temp

;      - Timer1 && Timer2 Settings
;      Count T/C1 at 4 MHz, clear on OC1A
        ldi temp,(1<<COM1A0)
        out TCCR1A,temp

;      Count T/C2 at 0.25 ms / tick
        ldi temp,(7<<CS20)+(1<<CTC2)
        out TCCR2,temp
        ldi temp,TEMPO
        out OCR2,temp

        sei

        ldi toneh,0x40
        ldi tone1,0x00
        ldi dura,1
        ldi tone,0

;      -Initial Alarm Song and duration settings
        ldi temp,0
        mov which_song,temp
        ldi temp,3
        mov duration_select,temp
        mov duration1,duration_select

;1->SIMPSON
;2->MISSION
;3->BUMTEST
;0->MOUNTAINKING

;*****

;      - duration_select takes any values
;      - For example if the user enters 3 , the song will be played for 3
;      - times with 10 seconds interval when the alarm time matches
;      - the current time

        ldi temp,1
        cp which_song,temp
        breq simpson
        ldi temp,2
        cp which_song,temp
        breq mission
        ldi temp,3
        cp which_song,temp
        breq bumtest
        ldi temp,0
        cp which_song,temp
        breq king

;      if no song is chosen bumtest is played as default..
        rjmp bumtest

simpson:

```

```

        ldi     noteph,high(NOTESIMPSON*2)
        ldi     notepl,low(NOTESIMPSON*2)
        rjmp    lcd

mission:
        ldi     noteph,high(NOTEMISSION*2)
        ldi     notepl,low(NOTEMISSION*2)
        rjmp    lcd

bumtest:
        ldi     noteph,high(NOTEBUMTEST*2)
        ldi     notepl,low(NOTEBUMTEST*2)
        rjmp    lcd

king:
        ldi     noteph,high(NOTEKING*2)
        ldi     notepl,low(NOTEKING*2)

;      - LCD Initializations
lcd:
        rcall   Lcd4Init
        rcall   Lcd4Clear
        rcall   Lcd4Home

        ;ldi temp,0
        ;mov alarm_hour,temp
        ;ldi temp,1
        ;mov alarm_minute,temp
        ;ldi temp,1
        ;mov alarm_day,temp

mloop:
        rjmp    mloop
        reti

;*****
OC10int:
        inc counter
        cpi counter,8 ;if 1 second has passed
        breq rbaykan
        reti
rbaykan:
        rcall baykan
        reti

;*****

baykan:
        inc say
        clr counter
        ldi temp_var,1
        cp temp_var,say
        breq change_second
        ret

change_second:

;      - If the alarm is off (stop = 1), alarm_off
;      - signal is sent to the web server via PORTC

```

```

;      - by clearing PIN7
;      - If the alarm is on (stop = 0), alarm_on
;      - signal is sent to the web server via PORTC
;      - by setting PIN7
      cpi stop,1
      breq send_alarm_off_signal
      rjmp send_alarm_on_signal

send_alarm_off_signal:
      cbi portc, 7
      rjmp cont_change_second

;      - If the alarm is on, we check PORTC's PINs 5
;      - and 6. If PIN5 is set and PIN6 is cleared
;      - then alarm is stopped.
send_alarm_on_signal:
      sbi portc,7
      in tempe, PINC
      andi tempe, 0b01100000
      cpi tempe, 32
      breq end_alarming
      rjmp cont_change_second

end_alarming:
      ldi stop,1
      clr duration1
      rcall ends

cont_change_second:
      clr can
      ldi tempe,128
      in can,PINC
      cp can,tempe
      brge check
      rjmp cont

;      - Check the control PORT (PINC) and branch
;      - to the proper setting section
check:
      sub can,tempe
      in hold_pinc,PINC
      andi hold_pinc, 0b00011111

      cpi hold_pinc,0b00010011
      breq choose_year

      cpi hold_pinc,0b00011110
      breq choose_month

      cpi hold_pinc,0b00011101
      breq choose_day

      cpi hold_pinc,0b00011100
      breq choose_hour

      cpi hold_pinc,0b00011011
      breq choose_minute

set_alarm:

```

```
    cpi hold_pinc,0b00011010
    breq choose_alarm_day_jump

    cpi hold_pinc,0b00011001
    breq choose_alarm_hour_jump

    cpi hold_pinc,0b00011000
    breq choose_alarm_minute_jump

    cpi hold_pinc,0b00010111
    breq choose_song_jump

    cpi hold_pinc,0b00010110
    breq choose_duration_jump
    rjmp cont

choose_alarm_day_jump:
    rjmp choose_alarm_day
choose_alarm_hour_jump:
    rjmp choose_alarm_hour
choose_alarm_minute_jump:
    rjmp choose_alarm_minute
choose_song_jump:
    rjmp choose_song
choose_duration_jump:
    rjmp choose_duration

wait_select:
    rjmp cont;
choose_year:
    mov tempe, remainder
    cpi tempe, 0b00011110
    breq wait_select
    cpi tempe, 0b00010011
    breq ycorrect
    ldi tempe, 0b00000000
    rjmp cont
ycorrect:
    ldi tempe, 0b00011100
    mov remainder, tempe
    andi can, 0b00001111
    mov year, can
    rjmp cont

choose_month:
    mov tempe, remainder
    cpi tempe, 0b00011101
    breq wait_select
    cpi tempe, 0b00011110
    breq mcorrect
    ldi tempe, 0b00000000
    rjmp cont
mcorrect:
    ldi tempe, 0b00010011
    mov remainder, tempe
    andi can, 0b00001111
    mov month, can
    rjmp cont
```

```
choose_day:
    mov tempe, remainder
    cpi tempe, 0b00000000
    breq dcorrect
    ldi tempe, 0b00000000
    rjmp cont
dcorrect:
    ldi tempe, 0b00011110
    mov remainder, tempe
    andi can, 0b00011111
    mov day, can
    rjmp cont

choose_hour:
    mov tempe, remainder
    cpi tempe, 0b00010011
    breq wait_select
    cpi tempe, 0b00011100
    breq hcorrect
    ;cpi tempe, 0b00011100
    ;breq hcorrect
    ldi tempe, 0b00000000
    rjmp cont
hcorrect:
    ldi tempe, 0b00011011
    mov remainder, tempe
    andi can, 0b00011111
    mov hour, can
    rjmp cont

choose_minute:
    mov tempe, remainder
    cpi tempe, 0b00011100
    breq wait_select
    cpi tempe, 0b00011011
    breq mincorrect
    ldi tempe, 0b00000000
    rjmp cont
mincorrect:
    andi can, 0b00111111
    mov minute, can
    clr second
    ldi tempe, 0b00000000
    mov remainder, tempe
    rjmp cont

choose_alarm_day:
    mov tempe, remainder
    cpi tempe, 0b00000000
    breq adcorrect
    ldi tempe, 0b00000000
    rjmp cont
adcorrect:
    ldi tempe, 0b00011001
    mov remainder, tempe
    andi can, 0b00011111
```

```
    mov alarm_day,can
    rjmp cont

choose_duration:
    mov tempe, remainder
    cpi tempe, 0b00010111
    breq wait_select2
    cpi tempe, 0b00010110
    breq adurcorrect
    ldi tempe, 0b00000000
    rjmp cont
adurcorrect:
    ldi tempe, 0b00000000
    mov remainder, tempe
    andi can, 0b00000011
    mov duration_select,can
    rjmp cont

choose_song:
    mov tempe, remainder
    cpi tempe, 0b00011000
    breq wait_select2
    cpi tempe, 0b00010111
    breq asongcorrect
    ldi tempe, 0b00000000
    rjmp cont
asongcorrect:
    ldi temp, 0b00010110
    mov remainder, temp
    andi can, 0b00000011
    mov which_song,can
    rjmp cont

choose_alarm_hour:
    mov tempe, remainder
    cpi tempe, 0b00011010
    breq wait_select2
    cpi tempe, 0b00011001
    breq ahcorrect
    ldi tempe, 0b00000000
    rjmp cont
ahcorrect:
    ldi tempe, 0b00011000
    mov remainder, tempe
    andi can, 0b00011111
    mov alarm_hour,can
    rjmp cont

choose_alarm_minute:
    mov tempe, remainder
    cpi tempe, 0b00011001
    breq wait_select2
    cpi tempe, 0b00011000
    breq amincorrect
    ldi tempe, 0b00000000
    rjmp cont
amincorrect:
    ldi tempe, 0b00010111
    mov remainder, tempe
    andi can, 0b00011111
```

```

        mov alarm_minute,can
        rjmp cont

wait_select2:
        rjmp cont;

cont:
        clr counter

refresh_lcd:
;***DAY/MONTH/YEAR
;****HOUR:MINUTE:SECOND
        rcall Lcd4Clear
        clr ten_count
        mov time_temp,day
        cpi time_temp,10
        brlo lsbld
ten_katd:
        subi time_temp,10
        inc ten_count
        cpi time_temp,10
        brge ten_katd
msbd:
;
        DAY First Digit
        ldi temp_var,131
        rcall Lcd4Ctrl
        ldi temp,48
        add ten_count,temp
        mov temp_var,ten_count
        rcall Lcd4Chr
lsbd:
        ldi temp,0
        cp ten_count,temp
        brne go6
;
        DAY First Digit
        ldi temp_var,131
        rcall Lcd4Ctrl
        ldi temp_var,48
        rcall Lcd4Chr
go6:
;
        DAY Second Digit
        ldi temp_var,132
        rcall Lcd4Ctrl
        subi time_temp,-48
        mov temp_var,time_temp
        rcall Lcd4Chr

        ldi temp_var,133
        rcall Lcd4Ctrl
;
        DAY/MONTH
        ldi temp_var, '/'
        rcall Lcd4Chr

        clr ten_count
        mov time_temp,month
        cpi time_temp,10
        brlo lsbmo

```

```

ten_katmo:
    subi time_temp,10
    inc ten_count
    cpi time_temp,10
    brge ten_katmo
msbmo:
;    MONTH First Digit
    ldi temp_var,134
    rcall Lcd4Ctrl
    ldi temp,48
    add ten_count,temp
    mov temp_var,ten_count
    rcall Lcd4Chr
lsbmo:
    ldi temp,0
    cp ten_count,temp
    brne go5
;    MONTH Second Digit
    ldi temp_var,134
    rcall Lcd4Ctrl
    ldi temp_var,48
    rcall Lcd4Chr
go5:
;    MONTH Second Digit
    ldi temp_var,135
    rcall Lcd4Ctrl
    ldi temp,48
    add time_temp,temp
    mov temp_var,time_temp
    rcall Lcd4Chr

    ldi temp_var,136
    rcall Lcd4Ctrl
;    DAY/YEAR
    ldi temp_var, '/'
    rcall Lcd4Chr

;    YEAR BEGINS WITH 20
;    YEAR First Digit
    ldi temp_var,137
    rcall Lcd4Ctrl
    ldi temp_var,50
    rcall Lcd4Chr

;    YEAR Second Digit
    ldi temp_var,138
    rcall Lcd4Ctrl
    ldi temp_var,48
    rcall Lcd4Chr

    clr ten_count
    mov time_temp,year
    cpi time_temp,10
    brlo lsby
ten_katy:
    subi time_temp,10
    inc ten_count
    cpi time_temp,10

```

```

        brge ten_katy
msby:
;       YEAR Third Digit
        ldi temp_var,139
        rcall Lcd4Ctrl
        ldi temp,48
        add ten_count,temp
        mov temp_var,ten_count
        rcall Lcd4Chr

lsby:
        ldi temp,0
        cp ten_count,temp
        brne go4
;       YEAR Third Digit
        ldi temp_var,139
        rcall Lcd4Ctrl
        ldi temp_var,48
        rcall Lcd4Chr
go4:
        subi time_temp,-48
;       YEAR Forth Digit
        ldi temp_var,140
        rcall Lcd4Ctrl
        mov temp_var,time_temp
        rcall Lcd4Chr

        clr ten_count
        mov time_temp,hour
        cpi time_temp,10
        brlo lsbh
ten_kath:
        subi time_temp,10
        inc ten_count
        cpi time_temp,10
        brge ten_kath
msbh:
;       HOUR First Digit
        ldi temp_var,196
        rcall Lcd4Ctrl
        ldi temp,48
        add ten_count,temp
        mov temp_var,ten_count
        rcall Lcd4Chr
lsbh:
        ldi temp,0
        cp ten_count,temp
        brne go3
;       HOUR First Digit(0)
        ldi temp_var,196
        rcall Lcd4Ctrl
        ldi temp_var,48
        rcall Lcd4Chr
go3:
;       HOUR Second Digit
        ldi temp_var,197
        rcall Lcd4Ctrl
        subi time_temp,-48
        mov temp_var,time_temp

```

```

        rcall Lcd4Chr

;       HOUR:MINUTE
        ldi temp_var,198
        rcall Lcd4Ctrl
        ldi temp_var,':'
        rcall Lcd4Chr

        clr ten_count
        mov time_temp,minute
        cpi time_temp,10
        brlo lsbm
ten_katm:
        subi time_temp,10
        inc ten_count
        cpi time_temp,10
        brge ten_katm
msbm:

        ldi temp_var,199
        rcall Lcd4Ctrl
        ldi temp,48
        add ten_count,temp
        mov temp_var,ten_count
        rcall Lcd4Chr
lsbm:
        ldi temp,0
        cp ten_count,temp
        brne go2
;       MINUTE First Digit(0)
        ldi temp_var,199
        rcall Lcd4Ctrl
        ldi temp_var,48
        rcall Lcd4Chr
go2:
;       MINUTE Second Digit
        ldi temp_var,200
        rcall Lcd4Ctrl
        subi time_temp,-48
        mov temp_var,time_temp
        rcall Lcd4Chr

        ldi temp_var,201
        rcall Lcd4Ctrl
;       MINUTE:SECOND
        ldi temp_var,':'
        rcall Lcd4Chr

        clr ten_count
        mov time_temp,second
        cpi time_temp,10
        brlo lsb
ten_kat:
        subi time_temp,10
        inc ten_count
        cpi time_temp,10
        brge ten_kat
msb:
;       SECOND First Digit
        ldi temp_var,202

```

```

        rcall Lcd4Ctrl
        ldi temp,48
        add ten_count,temp
        mov temp_var,ten_count
        rcall Lcd4Chr
lsb:
        ldi temp,0
        cp ten_count,temp
        brne gol
;        SECOND First Digit(0)
        ldi temp_var,202
        rcall Lcd4Ctrl
        ldi temp_var,48
        rcall Lcd4Chr
gol:
;        SECOND Second Digit
        ldi temp_var,203
        rcall Lcd4Ctrl
        subi time_temp,-48
        mov temp_var,time_temp
        rcall Lcd4Chr

        inc second
        ldi temp_var,60
        cp second,temp_var
        breq change_minute
        ret

change_minute:

        clr second
        inc minute

        ldi temp,1
        cp stop,temp
        brne ff

        cp alarm_minute,minute
        brne ff

        cp alarm_hour,hour
        brne ff

        cp alarm_day,day
        brne ff

;        - If alarm time matches with the currnet time
;        - make alarm on (stop = 0)
;        - begin to play song
        ldi stop,0
;        Count T/C1 at 4 MHz, clear on OC1A
        ldi temp,(1<<COM1A0)
        out TCCR1A,temp
        ret

ff:
        ldi stop,1
        ldi temp_var,60
        cp minute,temp_var
        breq change_hour

```

```
        ret

change_hour:

    clr minute
    inc hour
    ldi temp_var, 24
    cp hour, temp_var
    breq change_day
    ret

change_day:

    clr hour
    inc day
    ldi temp_var, 2
    cp month, temp_var
    breq february_check
    ldi temp_var, 31
    cp day, temp_var
    breq day_30_months
    ldi temp_var, 32
    cp day, temp_var
    breq change_month
    ret

february_check:

    ldi temp_var, 30
    cp day, temp_var
    breq change_month
    ret

day_30_months:

    ldi temp_var, 4
    cp month, temp_var
    breq change_month
    ldi temp_var, 6
    cp month, temp_var
    breq change_month
    ldi temp_var, 9
    cp month, temp_var
    breq change_month
    ldi temp_var, 11
    cp month, temp_var
    breq change_month
    ret

change_month:

    inc month
    clr day
    inc day

check_month:

    ldi temp_var, 13
    cp month, temp_var
    breq change_year
```

```

        ret

change_year:

        clr month
        inc month
        inc year
        ret

;*****

OC2int:
;       if stop = 1 -> no sound
        cpi stop,1
        breq ends
        dec dura
        breq newnote
        reti

ends:
        ldi temp,0
        out TCCR1A,temp
        rati

newnote:

        mov ZL,notepl
        mov ZH,noteph
        lpm
        mov tone,r0
        adiw ZL,1
        lpm
        mov dura,r0
        adiw notepl,2

        cpi tone,$00
        brne noendsong
;       if end of the song, stop the counter
        ldi temp,(0<<CS10)+(1<<CTC1)
        out TCCR1B,temp
        dec duration1

;       if the song has been played for duration_select times
;       stop playing...
        breq finish_playing
        rjmp gon

finish_playing:

        ldi stop,1
        clr duration1

;1->SIMPSON
;2->MISSION
;3->BUMTEST
;0->MOUNTAINKING

gon:
        ldi temp,1
        cp which_song,temp

```

```

    breq simpson1
    ldi temp,2
    cp which_song,temp
    breq mission1
    ldi temp,3
    cp which_song,temp
    breq bumtest1
    ldi temp,0
    cp which_song,temp
    breq king1
;    if no song is chosen bumtest is played as default..
    rjmp bumtest1;

simpson1:
    ldi    notepl,high(NOTESIMPSON*2)
    ldi    notepl,low(NOTESIMPSON*2)
    rjmp   endnote
mission1:
    ldi    notepl,high(NOTEMISSION*2)
    ldi    notepl,low(NOTEMISSION*2)
    rjmp   endnote
bumtest1:
    ldi    notepl,high(NOTEBUMTEST*2)
    ldi    notepl,low(NOTEBUMTEST*2)
    rjmp   endnote
king1:
    ldi    notepl,high(NOTEKING*2)
    ldi    notepl,low(NOTEKING*2)
    rjmp   endnote

noendsong:
;    If $FF then pause
    cpi    tone,$FF
    brne   playtone
    ldi    temp,(0<<CS10)+(1<<CTC1)
    out    TCCR1B,temp
    rjmp   endnote

playtone:
;    Read frequency for halftone
    ldi    ZH,high(HALFTONETBL*2)
    ldi    ZL,low(HALFTONETBL*2)
    mov    temp,tone
    andi   temp,0x0f
    lsl    temp
    add    ZL,temp
    ldi    temp_var,0
    adc    ZH,temp_var

;    Read OCR value for halftone from table
    lpm
    mov    tone1,r0
    ori    ZL,1
    lpm
    mov    toneh,r0

;    Shift to correct octave
    mov    temp,tone
;    andi   temp,0xf0

```

```

octloop:
    subi    temp,0x10
    brcs    goon4
    lsr     toneh
    ror     tonel
    rjmp    octloop
goon4:

;   Set up new frequency and start counter (in case it was stopped)
    out     OCRAH,toneh
    out     OCRAL,tonel
    ldi     temp,(1<<CS10)+(1<<CTC1)
    out     TCCR1B,temp
endnote:
    reti

;*****

Lcd4Init:
;   Wait a second for the LCD
    rcall   LcdDelay1s
;   Data direction to output
    ldi     temp_var,cLcdWrite
    out     DDRA,temp_var
;   Dummy to catch LCD
    ldi     temp_var,cLcdDummy
;   Send three times with 5 ms delay
    rcall   Lcd4Set ;
    rcall   LcdDelay5ms
    ldi     temp_var,cLcdDummy
    rcall   Lcd4Set
    rcall   LcdDelay5ms
    ldi     temp_var,cLcdDummy
    rcall   Lcd4Set
    rcall   LcdDelay5ms
;   Function Set to 4 Bit
    ldi     temp_var,0b00101000
;   Output on the Control Port LCD
    rcall   Lcd4Ctrl
;   Cursor display shift
    ldi     temp_var,0b00010100
    rcall   Lcd4Ctrl
;   LCD on
    ldi     temp_var,0b00001100
    rcall   Lcd4Ctrl
;   Entry mode
    ldi     temp_var,0b00000110
;   Shift Characters to right
    rcall   Lcd4Ctrl
    ret
;*****

Lcd4Clear:
;   Set Lcd Clear
    ldi     temp_var,0b00000001
    rcall   Lcd4Ctrl
    ret
;*****

Lcd4Home:
;   Set LCD Home Position
    ldi     temp_var,0b00000010

```

```

        rcall Lcd4Ctrl
        ret
;*****
;
; Output of temp_var on the Control-Port of the LCD
;
Lcd4Ctrl:
    push temp_var ; save byte
    andi temp_var,0xF0 ; clear lower nibble
    rcall Lcd4Set ; output upper nibble
    pop temp_var ; restore byte
    swap temp_var ; swap lower and upper nibble
    andi temp_var,0xF0 ; clear lower nibble
    rcall Lcd4Set ; output lower nibble
    rjmp LcdDelay5ms ; done.
;*****

Lcd4Chr:
    push temp_var ; save char on stack
    andi temp_var,0xF0 ; clear lower nibble
    sbr temp_var,mLcdRs ; Set RS-Bit
    rcall Lcd4Set ; output nibble
    pop temp_var ; get char from stack
    swap temp_var ; swap nibbles
    andi temp_var,0xF0 ; clear lower nibble
    sbr temp_var,mLcdRs ; Set RS-Bit
    rcall Lcd4Set ; output nibble
    rjmp LcdDelay50us ; ready
;
; Send nibble in temp_var to LCD
;*****
Lcd4Set:
    out PORTA,temp_var ; Byte to output port
    nop
    sbi PORTA,bLcdEn ; Set Enable-Bit
    enactive ; Delay macro
    cbi PORTA,bLcdEn ; Clear Enable Bit
    nop
    ret
;*****

; Delay by 1 second on start-up
;
LcdDelay1s:
    ldi temp_var,255
LcdDelay1s1:
    rcall LcdDelay5ms
    dec temp_var
    brne LcdDelay1s1
    ret
;*****
; Delay by 5 ms following each Control Word
;
LcdDelay5ms:
    ldi temp_var,20
ldi:

```

```

    rcall delay
    dec    temp_var
    brne ldi
    ret

;*****
;
; Delay by 50 Microseconds after each Char
;
LcdDelay50us:
    rcall delay
    ret

;*****

delay:
    ldi temp,$FF
d1:
    dec temp
    brne d1
    ret

;*****
HALFTONETBL:
.dw    61156,57724,54484,51426,48540,45815
.dw    43244,40817,38526,36364,34323,32396
;*****
NOTESIMPSON:
.db    48,2,52,4,54,4,57,8,55,2,52,4,48,4
.db    0,0
;*****

NOTEBUMTEST:
.db    0x44,2, 0x43,2, 0x42,2, 0x41,2, 0x42,2, 0x41,2, 0x40,2, 0x3b,2, 0x40,2
.db    0x3b,2, 0x3a,2, 0x39,2, 0x38,2, 0x37,2, 0x36,2, 0x35,2, 0x34,2, 0x33,2
.db    0x32,2, 0x31,2, 0x32,2, 0x31,2, 0x30,2, 0x2b,2, 0x30,2, 0x2b,2, 0x2a,2
.db    0x29,2, 0x28,2, 0x27,2, 0x26,2, 0x25,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2
.db    0x22,2, 0x21,2, 0x20,2, 0x1b,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2, 0x22,2
.db    0x21,2, 0x20,2, 0x1b,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2, 0x22,2, 0x21,2
.db    0x20,2, 0x1b,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2, 0x22,2, 0x21,2, 0x20,2
.db    0x1b,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2, 0x20,2, 0x25,2, 0x24,2, 0x23,2
.db    0x24,2, 0x23,2, 0x22,2, 0x21,2, 0x20,2, 0x21,2, 0x22,2, 0x23,2, 0x24,2
.db    0x23,2, 0x22,2, 0x21,2, 0x20,2, 0x25,2, 0x24,2, 0x23,2, 0x24,2, 0x23,2
.db    0x22,2, 0x21,2, 0x20,2, 0x21,2, 0x22,2, 0x23,2, 0x24,1, 0x20,1, 0x23,2
.db    0x22,2, 0x21,2, 0x22,2, 0x21,2, 0x20,2, 0x1b,2, 0x20,2, 0x21,2, 0x22,2
.db    0x23,2, 0x24,2, 0x25,2, 0x24,2, 0x23,2, 0x24,2, 0x23,2, 0x22,2, 0x21,2
.db    0x22,2, 0x21,2, 0x20,2, 0x1b,2, 0x20,2, 0x21,2, 0x22,2, 0x23,2, 0x24,2
.db    0x26,2, 0x27,2, 0x28,2, 0x29,2, 0x28,2, 0x27,2, 0x26,2, 0x25,2, 0x2a,2
.db    0x29,2, 0x28,2, 0x29,2, 0x28,2, 0x27,2, 0x26,2, 0x25,2, 0x26,2, 0x27,2
.db    0x28,2, 0x29,2, 0x28,2, 0x27,2, 0x26,2, 0x25,2, 0x2a,2, 0x29,2, 0x28,2
.db    0x29,2, 0x28,2, 0x27,2, 0x26,2, 0x25,2, 0x26,2, 0x27,2, 0x28,2, 0x29,2
.db    0x28,2, 0x27,2, 0x26,2, 0x27,2, 0x26,2, 0x25,2, 0x24,2, 0x25,2, 0x26,2
.db    0x27,2, 0x28,2, 0x29,2, 0x2a,2, 0x29,2, 0x28,2, 0x27,2, 0x28,2, 0x27,2
.db    0x26,2, 0x27,2, 0x26,2, 0x25,2, 0x24,2, 0x25,2, 0x26,2, 0x27,2, 0x28,2
.db    0x29,2, 0x2a,2, 0x29,2, 0x28,2, 0x29,2, 0x19,2, 0x18,2, 0x19,2, 0x18,2
.db    0x19,2, 0x18,2, 0x19,2, 0x1a,2, 0x19,2, 0x1a,2, 0x19,2, 0x1a,2, 0x19,2
.db    0x1a,2, 0x19,2, 0x29,2, 0x28,2, 0x29,2, 0x28,2, 0x29,2, 0x28,2, 0x29,2
.db    0x29,2, 0x2a,2, 0x29,2, 0x2a,2, 0x29,2, 0x2a,2, 0x29,2, 0x2a,2, 0x29,2
.db    0x29,2, 0x2a,2, 0x29,2, 0x28,2, 0x29,2, 0x2a,2, 0x29,2, 0x28,2, 0x29,2
.db    0x2a,2, 0x29,2, 0x28,2, 0x29,2, 0x2a,2, 0x29,2, 0x28,2, 0x39,8, 0xFF,1

```

```
.db 0x22,16,0,0
;*****
NOTEEMISSION:

.db 38,8,39,8,38,8,39,8,38,8,39,8,38,8,39,8,38,8,39,8
.db 40,8,41,8,42,8,43,8,43,8,38,8,39,8,38,8,39,8,38,8,39,8,38,8,39,8,38,8,39,8
.db 0,0

;*****
NOTEKING:

.db c2,8, d2,8, e2,8, f2,8, g2,8, e2,8, g2,16, a2,8, f2,8, a2,16,g2,8, e2,8, g2,16
.db c2,8, d2,8, e2,8, f2,8, g2,8, e2,8, g2,16, a2,8, f2,8, a2,16,g2,32
.db 0,0
;*****
```



Appendix E

THE INTERNET ALARM CLOCK WEB SERVER SOURCE CODE

```

/*****
/*****   Written By M. Can Filibeli   *****/
/*****
/*****

/*
 * MAC Address of the CLOCK
 */
#define MACADDRESS 0x00, 0x06, 0x98, 0x12, 0xAB, 0x01
#define MAC "00069812AB01"
/*
 * Definition of the Appliance
 */
#define DEFINITION "ALARM_CLOCK"
/*
 * BASENETWORKADDRESS and BASENETWORKMASK are used to
 * obtain IP address using the AutoIP function
 * if there is no DHCP Server available in the network.
 * For example, if there is no DHCP Server in the Appliance
 * network, the IP address of the appliance will be 10.10.10.x.
 * RANDOMIZATIONCONSTANT is used to prevent acquiring the same
 * IP address with other appliances when many appliances are
 * connected to the appliance network at the same time.
 */
#define BASENETWORKADDRESS "10.10.10.0"
#define BASENETWORKMASK "255.255.255.0"
#define RANDOMIZATIONCONSTANT 35
/*
 * SERVERUDPPORT is the HAN Server's UDP Port for message exchange.
 * CLIENTUDPPORT is the Appliance's UDP Port which is used for exchanging
 * messages with the HAN Server.
 */
#define SERVERUDPPORT 10005
#define CLIENTUDPPORT 10006
/*
 * LISTSERVERUDPPORT and LISTCLIENTPORT is used the exchange Appliance
 * service lists between appliances.
 */
#define LISTSERVERUDPPORT 10007
#define LISTCLIENTUDPPORT 10008
/*
 * Maximum number of appliances allowed in HAN. For the prototype
 * implementation, maximum number is 10 because of RAM limitations.
 */
#define NUMBEROFAPPLIANCES 3

```

```
/*
 * Maximum number of services allowed per appliance. For the prototype
 * implementation, maximum number is 10 because of RAM limitations.
 */
#define NUMBEROFSERVICES 10
/*
 * Number of Services available for this appliance for device coordination
 */
#define AVAILABLESERVICE 2
/*
 * Include files
 */
#include <string.h>
#include <io.h>
#include <stdlib.h>
/*
 * If Ethernut PCB version 2.x is used, we include network device
 * driver for it is included.
 */
#ifdef ETHERNUT2
#include <dev/lanc111.h>
#else
#include <dev/nicrtl.h>
#endif
#include <dev/debug.h>
#include <dev/urom.h>
#include <dev/hd44780.h>
#include <dev/uartavr.h>
#include <dev/ppp.h>
#include <dev/chat.h>
#include <arpa/inet.h>
#include <netinet\icmp.h>
#include <sys/version.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/event.h>
#include <sys/heap.h>
#include <sys/confnet.h>
#include <sys/socket.h>
#include <pro/httpd.h>
#include <pro/dhcp.h>
#include <net/if_var.h>
#include <net/route.h>
#include <fcntl.h>
#include <netdb.h>
/*
 * If we define NUTDEBUG, debug drivers for
 * debugging are included
 */
#ifdef NUTDEBUG
#include <sys/osdebug.h>
#include <net/netdebug.h>
#endif

/* Function Prototypes*/
int alarmstatus(void);
int stopalarming(void);

static char *html_mt = "text/html";
static u_char buffquery[512];
```

```

static u_char buffer[100];
/*
 * Holds the network address of the appliance
 */
u_long BaseNetworkIP;
/*
 * Holds the network mask of the appliance
 */
u_long BaseNetworkMASK;
/*
 * Holds the IP address of the appliance
 */
u_long NetworkIP;
/*
 * Holds the IP address of the HAN Server if available
 */
u_long serverIP;
/*
 * Holds if the HAN Server is online or not.
 * While 0 means the HAN Server is offline,
 * 1 means the HAN Server is online.
 */
u_long serverOnline=0;
/*
 * Holds the IP address of the appliance which will be connected
 * during device coordination.
 */
char coordinationIP[20];
/*
 * Holds the target url which will be called during device coordination.
 */
char coordinationLink[50];

/*
 * Struct for Appliance Information
 */
struct Appliance{
    char definition[15];
    char ip[20];
};
/*
 * Struct for Appliance's Service Information
 */
struct Service{
    char definition[15];
    char url[15];
};

struct Appliance *aplist;
struct Service *servList;
int numberRegistered = -1;

int serviceNumber = AVAILABLESERVICE;
int serviceNumberRemote = 0;

/*
 * Used to get the list of online appliances from the HAN Server.
 * The appliance information is saved to aplist structure.
 * Returns 0, if the HAN Server cannot be located. Else, returns
 * the number of online appliances. Returns -1 if an error occurs

```

```

* during message exchange.
*/
int getApplianceList()
{
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buff[70];
    int i;

    if(serverOnline == 0){
        printf("\nServer is offline");
        return 0;
    }
    else{
        printf("\nServer is online");
        /* Create a UDP Socket */
        sock = NutUdpCreateSocket(CLIENTUDPPORT);
        /* Send an ALQ (Appliance List Query) message to HAN Server */
        if(NutUdpSendTo(sock, serverIP, SERVERUDPPORT, "ALQ", 3) == 0){
            /* Receive the number of online appliances*/
            if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buff, 10, 1000)
> 0){
                if(strncmp(buff, "NR:", 3) == 0){
                    buff[4] = '\0';
                    numberRegistered = atoi(buff + 3);
                    /* Send the SENDLIST request to the HAN Server*/
                    if(NutUdpSendTo(sock, serverIP, SERVERUDPPORT,
"SENDLIST", 8) == 0){
                        int failed=0;
                        /*
is
information
                        * Continue if the number of online appliances
                        * bigger than the number of appliance
                        * received.
                        */
                        for(i=0; i<numberRegistered;){
                            int num = 0;
                            char def[20];
                            char ip[20];
                            /* Receive appliance information*/
                            if(NutUdpReceiveFrom(sock,
&ip_addr_from, &port_from, buff, 50, 1000) > 0){
                                /* Extract appliance's sequence
number, definition and IP address*/
                                if(strncmp(buff, "?OK=", 4) ==
0){
                                    char *c[3];
                                    char *p[3];
                                    int extract;

                                    for(extract=0;extract<3;extract++){
                                        if(extract == 0)
                                            c[0] =
                                            strtok(buff, "=");
                                        }
                                        else
                                        {

```

```

        c[extract] = strtok(0, "=");

        strtok(0, "&");

        information to the aplist structure if
        expected and the number received are equal

        strcpy(aplist[i].ip, ip);
        strcpy(aplist[i].definition, def);
        sprintf(buff, "OK:%d", i);

        acknowledge message with the last received sequence number*/
        NutUdpSendTo(sock, serverIP, SERVERUDPPORT, buff, sizeof(buff));

        if(i >= numberRegistered)

        message with the last sequence
        received if the sequence number
        number received are not equal.

        sprintf(buff, "OK:%d", i);
        NutUdpSendTo(sock, serverIP, SERVERUDPPORT, buff, sizeof(buff));

        NutUdpDestroySocket(sock);

        }
        p[extract] =
    }
    num = atoi(p[0]);
    strcpy(def, p[1]);
    strcpy(ip, p[2]);
    /*
     * Save the appliance
     * the sequence number
     */
    if(num == i){
        buff[4] = '\0';
        /* Send
        i++;

        break;
        else continue;
    }
    /*
     * Send an acknowledge
     * number successfully
     * expected and the
     */
    else{
        failed++;
        if(failed < 3)
            continue;
    }
    else{
        failed++;
        if(failed < 3){
            continue;
        }
        else{
            return -1;

```

```

    }
    else{
        NutUdpDestroySocket(sock);
        return -1;
    }
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
else{
    printf("Failed...");
    NutUdpDestroySocket(sock);
    return -1;
}
}
NutUdpDestroySocket(sock);
return i;
}

/*
 * Used to get the list of services of an appliance from
 * the appliance itself. The service information is saved
 * to servList structure. Returns -1, if the appliance
 * cannot be located or an error occurs
 * during message exchange. Else, returns
 * the number of online appliances.
 */
int getServiceList(u_long deviceIP)
{
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buff[70];
    int i;

    /* Create a UDP Socket */
    sock = NutUdpCreateSocket(LISTCLIENTUDPPORT);
    /* Send SLQ(Service List Query) message to the target appliance */
    if(NutUdpSendTo(sock, deviceIP, LISTSERVERUDPPORT, "SLQ", 3) == 0){
        /* Receive the number of available services*/
        if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buff, 10, 10000) > 0){
            if(strncmp(buff, "SNR:", 4) == 0){
                buff[5] = '\0';
                serviceNameRemote = atoi(buff + 4);
                /* Send the SERVLIST request to the appliance*/
                if(NutUdpSendTo(sock, deviceIP, LISTSERVERUDPPORT,
                    "SERVLIST", 8) == 0){
                    int failed=0;

```

```

/*
 * Continue if the number of services available is
 * bigger than the number of service information
 * received.
 */
for(i=0; i<serviceNumberRemote;){
    int num = 0;
    char def[15];
    char url[15];
    /* Receive service information*/
    if(NutUdpReceiveFrom(sock, &ip_addr_from,
        &port_from, buff, 50, 1000) > 0){
        /* Extract service's sequence number.
        definition and IP address*/
        if(strncmp(buff, "?OK=", 4) == 0){
            char *c[3];
            char *p[3];
            int extract;

            for(extract=0;extract<3;extract++){
                if(extract == 0) {
                    c[0] =
                }
                else
                {
                    c[extract] =
                }
                p[extract] = strtok(0,
            }
            num = atoi(p[0]);
            strcpy(def,p[1]);
            strcpy(url,p[2]);
            /*
             * Save the service information
             * the sequence number expected
            */
            if(num==i){
                strcpy(servList[i].url,url);
                strcpy(servList[i].definition, def);
            i);
            message with the last received sequence number*/
            deviceIP, LISTSERVERUDPPORT, buff, sizeof(buff));

            if(i>=serviceNumberRemote){
                break;
            }
            else continue;
        }
    }
}

```

```

with the last sequence
if the sequence number
received are not equal.

i);
deviceIP, LISTSERVERUDPPORT, buff, sizeof(buff));

/*
 * Send an acknowledge message
 * number successfully received
 * expected and the number
 */
else{
    sprintf(buff, "OK:%d",
            NutUdpSendTo(sock,
                failed++;
                if(failed<3)
                    continue;
            }
        }
    else{
        failed++;
        if(failed<3){
            continue;
        }
        else{
            return -1;
        }
    }
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    printf("Failed...");
    NutUdpDestroySocket(sock);
    return -1;
}
}
NutUdpDestroySocket(sock);
return i;
}

/*
 * In order to send signals to clock's microprocessor,
 * these signals should be output for some time. NutSleep()
 * function is not used here since declaring a PORT as

```

```

* output slows down the thread switching
* actions. Therefore, the period of sleeping cannot be
* determined healthily.
*/
void delay(long k){
    long i; int j;
    //spend some time
    for(i=0;i<=k;i++){
        //spend some time
        for(j=0;j<=2;j++){
            ;
        }
    }
    return;
}
/*
* CGI Function - Displays the list of online appliances as a web page
*
* This routine must have been registered by NutRegisterCgi() and is
* automatically called by NutHttpRequest() when the client
* request the URL 'cgi-bin/coordination.cgi'.
*/
static int Coordination(FILE * stream, REQUEST * req)
{
    /*
    * The special type 'prog_char' forces the
    * string literals to be placed in flash ROM.
    * This saves us a lot of RAM.
    */
    static prog_char html_head[] = "<html><head><title>Device
Coordination</title></head><body><p align=\"center\"><font color=#CCFFFF
size=\"16\"></font>";
    static prog_char html_error[] = "<b><font size=\"5\" color=\"#FF0000\">No Suitable
Appliance for Coordination</font></b><br></p><a
href=\"/index.html\">Main</a></center></body></html>";
    static prog_char html_error2[] = "<b><font size=\"5\" color=\"#FF0000\">HAN Server
is Offline!</font></b><br></p><a href=\"/index.html\">Main</a></center></body></html>";
    static prog_char html_form1[] = "<form action=\"/cgi-bin/getlist.cgi\"
method=\"get\">";
    static prog_char html_form2[] = "<!--h--><b><font size=\"5\"
color=\"#FF0000\">Online Device List</font></b><br><br>";
    static prog_char html_option[] = "<input type=\"radio\" name=\"targetip\"
value=\"\"";
    static prog_char html_end[] = "<br><input type=\"submit\"
value=\"Next\"></form></p><center><a href=\"/index.html\">Main</a></center></body></html>";
    /* Get the list of online appliances from the HAN Server*/
    int onlineApp = getApplianceList();
    int i;
    int listedCount=0;

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);
    fputs_P(html_head, stream);
    fputs_P(html_form1, stream);
    fputs_P(html_form2, stream);

    if(serverOnline==0){
        fputs_P(html_error2, stream);
        fflush(stream);
    }
}

```

```

        return 0;
    }

    for(i=0;i<onlineApp;i++){
        if(inet_addr(aplist[i].ip) == NetworkIP){
            continue;
        }
        listedCount++;
        fputs_P(html_option, stream);
        fprintf(stream, "%s">%s<br>", aplist[i].ip, aplist[i].definition);
    }
    if(onlineApp == 0){
        fputs_P(html_error, stream);
        fflush(stream);
        return 0;
    }
    fputs_P(html_end, stream);
    fflush(stream);
    return 0;
}

/*
 * CGI Function - Displays the list of services as a web page
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/getlist.cgi'.
 */
static int ServiceList(FILE * stream, REQUEST * req)
{
    char *c[5];
    char *p[5];
    int numParams;
    /*
     * The special type 'prog_char' forces the
     * string literals to be placed in flash ROM.
     * This saves us a lot of RAM.
     */
    static prog_char html_head[] = "<html><head><title>Device Coordination - Service
List</title></head><body><p align=\"center\"><font color=#CCFFFF size=\"16\"></font>";
    static prog_char head_error[] = "<font size=\"5\" color=\"#FF0000\">Target Appliance
is not online...</font></b></p><a href=\"/index.html\">Main</a></center></body></html>";
    static prog_char html_form1[] = "<form action=\"/cgi-bin/finish.cgi\"
method=\"get\">";
    static prog_char html_form2[] = "<!--h--><b><font size=\"5\"
color=\"#FF0000\">Service List</font></b><br><br>";
    static prog_char html_option[] = "<input type=\"radio\" name=\"targeturl\"
value=\"\"";
    static prog_char html_end[] = "<br><input type=\"submit\"
value=\"Finish\"></form></p><center><a
href=\"/index.html\">Main</a></center></body></html>";
    int numberOfServices = 0;
    int i;
    /*
     * Extract up to 10 request parameters
     */
    if (req->req_query) {
#ifdef __IMAGECRAFT__
        char *qp1;
#endif

```

```

        /* Extract parameters. */
        for (numParams = 0; numParams < 10; numParams++){
            if(numParams == 0) {
#ifdef __IMAGECRAFT__
                /*
                * Extract the parameters. Note, that it is potentially
                * dangerous
                * to use strtok in multithreaded applications. There's no
                * problem
                * here, because we can be sure that there will be no thread
                * switch
                * between the first and the last call to strtok. Otherwise we
                * must use strtok_r().
                */
                c[0] = strtok(req->req_query, "=");
            #else
                /*
                * There's no strtok in the GCC library. So we take the chance
                * to
                * demonstrate strtok_r().
                */
                c[0] = strtok_r(req->req_query, "=", &qp1);
            #endif
        }
        else {
#ifdef __IMAGECRAFT__
            c[numParams] = strtok(0, "=");
        #else
            c[numParams] = strtok_r(0, "=", &qp1);
        #endif
        }
        /* If the parameter is equal to targetip then exit from
        * parameter extraction process. Else, continue
        * with the next parameter.
        */
        if(strcmp (c[numParams], "targetip") == 0){
#ifdef __IMAGECRAFT__
            p[numParams] = strtok(0, "&");
        #else
            p[numParams] = strtok_r(0, "&", &qp1);
        #endif
            strcpy(coordinationIP, p[numParams]);
            break;
        }
        else continue;
    }
}
/*
 * Get the list of services from the appliance whose IP
 * address is equal to the coordinationIP
 */
numberOfServices = getServiceList(inet_addr(coordinationIP));
/* These useful API calls create a HTTP response for us. */
NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBot(stream, html_mt, -1);

fputs_P(html_head, stream);
/*
 * If the number of services is less than 1,
 * then display an error.

```

```

    */
    if(numberOfServices < 1){
        fputs_P(head_error, stream);
        fflush(stream);
        return 0;
    }
    fputs_P(html_form1, stream);
    fputs_P(html_form2, stream);
    /*
     * Display the service list
     */
    for(i=0;i<numberOfServices;i++){
        fputs_P(html_option, stream);
        fprintf(stream, "%s\">%s<br>", servList[i].url, servList[i].definition);
    }
    fputs_P(html_end, stream);
    fflush(stream);
    return 0;
}
/*
 * CGI Function - Finalizes the device coordination process
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/finish.cgi'.
 */
static int FinishCoordination(FILE * stream, REQUEST * req) {
    char *c[3];
    char *p[3];
    int numParams;
    /*
     * The special type 'prog_char' forces the
     * string literals to be placed in flash ROM.
     * This saves us a lot of RAM.
     */
    static prog_char html_head[] = "<html><head><title>Device Coordination -
Done</title></head><body><p align=\"center\">";
    static prog_char html_form[] = "<b><font size=\"5\" color=\"\#FF0000\">Coordination
Settings are saved...</font></b><br><br>";
    static prog_char html_end[] = "</p><center><a
href=\"\$/index.html\">Main</a></center></body></html>";
    int numberOfServices = 0;
    int i;
    /*
     * Extract up to 10 request parameters
     */
    if (req->req_query) {
#ifdef __IMAGECRAFT__
        char *qp1;
#endif
        /* Extract parameters. */
        for (numParams = 0; numParams < 10; numParams++){
            if(numParams == 0) {
#ifdef __IMAGECRAFT__
                /*
                 * Extract the parameters. Note, that it is potentially
                 dangerous
                 * to use strtok in multithreaded applications. There's no
                 problem

```

```

switch
    * here, because we can be sure that there will by no thread
    * between the first and the last call to strtok. Otherwise we
    * must use strtok_r().
    */
    c[0] = strtok(req->req_query, "=");
#else
    /*
    * There's no strtok in the GCC library. So we take the chance
    to
    * demonstrate strtok_r().
    */
    c[0] = strtok_r(req->req_query, "=", &qp1);
#endif
    }
    else {
#ifdef __IMAGECRAFT__
        c[numParams] = strtok(0, "=");
    #else
        c[numParams] = strtok_r(0, "=", &qp1);
    #endif
    }
    /* If the parameter is equal to targeturl then exit from
    * parameter extraction process. Else, continue
    * with the next parameter.
    */
    if(strcmp (c[numParams], "targeturl") == 0){
#ifdef __IMAGECRAFT__
        p[numParams] = strtok(0, "&");
    #else
        p[numParams] = strtok_r(0, "&", &qp1);
    #endif
        strcpy(coordinationLink, "cgi-bin/");
        strcat(coordinationLink, p[numParams]);
        break;
    }
    else continue;
}
}
/* These useful API calls create a HTTP response for us. */
NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBot(stream, html_mt, -1);

fputs_P(html_head, stream);
fputs_P(html_form, stream);
fputs_P(html_end, stream);
fflush(stream);
return 0;
}

/*
* CGI Function - Sets the clock
*
* This routine must have been registered by NutRegisterCgi() and is
* automatically called by NutHttpProcessRequest() when the client
* request the URL 'cgi-bin/setclock.cgi'.
*/
static int SetClock(FILE * stream, REQUEST * req) {
    char *c[5];
    char *p[5];

```

```

char *hour,*minute,*day,*month,*year;
int hourint,minuteint,dayint,monthint,yearint;
int dayinttemp=0,monthinttemp=0,yearinttemp=0,hourinttemp=0,minuteinttemp=0;
/*
 * The special type 'prog_char' forces the
 * string literals to be placed in flash ROM.
 * This saves us a lot of RAM.
 */
static prog_char html_head[] = "<html><title>Done</title><body><BR><H3
align=\"center\"><font color=\"#FF0000\">The Clock is Set</font></H3>";
static prog_char html_body[] = "<center><h4><u><font color=\"#0000FF\"><b>New
Settings:</b></font></u></h4>";
static prog_char html_body1[] = "<font color=\"#0000FF\"><b>Time: </b></font>";
static prog_char html_body2[] = "<font color=\"#0000FF\"><b>Date: </b></font>";
static prog_char html_body3[] = "<font></font><a
href=\"/index.html\">Main</a></center></body></html>";

if (req->req_query) {
    u_char i;

    /* Extract 5 parameters. */
    for (i = 0; i < 5; i++) {
        if(i == 0) {
#ifdef __IMAGECRAFT__
            /*
             * Extract the parameters. Note, that it is potentially
             * dangerous
             * to use strtok in multithreaded applications. There's no
             * problem
             * here, because we can be sure that there will by no thread
             * switch
             * between the first and the last call to strtok. Otherwise we
             * must use strtok_r().
             */
            c[0] = strtok(req->req_query, "=");
#else
            /*
             * There's no strtok in the GCC library. So we take the chance
             * to
             * demonstrate strtok_r().
             */
            c[0] = strtok_r(req->req_query, "=", &qp1);
#endif
        }
        else {
#ifdef __IMAGECRAFT__
            c[i] = strtok(0, "=");
#else
            c[i] = strtok_r(0, "=", &qp1);
#endif
        }
#ifdef __IMAGECRAFT__
        p[i] = strtok(0, "&");
#else
        p[i] = strtok_r(0, "&", &qp1);
#endif
    }

    hourint = atoi(p[0]);

```

```
minuteint = atoi(p[1]);
dayint = atoi(p[2]);
monthint = atoi(p[3]);
yearint = atoi(p[4]);

/* Configure data direction settings for PORTE, PORTD and PORTB
 * PORTB pins 0 to 7 are set to be output
 * PORTD pins 0 and 1 are set to be output
 * PORTE pins 2 to 7 are set to be output
 */
sbi(DDRB,0);
sbi(DDRB,1);
sbi(DDRB,2);
sbi(DDRB,3);
sbi(DDRB,4);
sbi(DDRB,5);
sbi(DDRB,6);
sbi(DDRB,7);
sbi(DDRD,0);
sbi(DDRD,1);
sbi(DDRE,2);
sbi(DDRE,3);
sbi(DDRE,4);
sbi(DDRE,5);
sbi(DDRE,6);
sbi(DDRE,7);

/* sends day setting to clock*/
dayinttemp = dayint & 3;
outp(dayint+128,PORTE);
outp(dayinttemp,PORTD);
outp(0x1D, PORTB);
delay(600);

/* sends month setting to clock*/
monthinttemp = monthint & 3;
outp(monthint+128,PORTE);
outp(monthinttemp,PORTD);
outp(0x1E, PORTB);
delay(600);

/* sends year setting to clock*/
yearinttemp = yearint & 3;
outp(yearint+128,PORTE);
outp(yearinttemp,PORTD);
outp(0x13, PORTB);
delay(600);

/* sends hour setting to clock*/
hourinttemp = hourint & 3;
outp(hourint+128,PORTE);
outp(hourinttemp,PORTD);
outp(0x1C, PORTB);
delay(600);

/* sends minute setting to clock*/
minuteinttemp = minuteint & 3;
outp(minuteint+128,PORTE);
outp(minuteinttemp,PORTD);
outp(0x1B, PORTB);
```

```

delay(700);

/* Re-configure data direction settings for PORTE, PORTD and PORTE
 * PORTE pins 0 to 7 are set to be input
 * PORTD pins 0 and 1 are set to be input
 * PORTE pins 2 to 7 are set to be input
 */
cbi(DDRB,0);
cbi(DDRB,1);
cbi(DDRB,2);
cbi(DDRB,3);
cbi(DDRB,4);
cbi(DDRB,5);
cbi(DDRB,6);
cbi(DDRB,7);
cbi(DDRD,0);
cbi(DDRD,1);
cbi(DDRE,2);
cbi(DDRE,3);
cbi(DDRE,4);
cbi(DDRE,5);
cbi(DDRE,6);
cbi(DDRE,7);
outp(0xff,PORTE);
outp(0xff,PORTD);
outp(0xff, PORTE);
/* These useful API calls create a HTTP response for us. */
NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBot(stream, html_mt, -1);

fputs_P(html_head, stream);
fputs_P(html_body, stream);
fputs_P(html_body1, stream);
fprintf(stream, "%.2d:%.2d<br>", hourint, minuteint);
fputs_P(html_body2, stream);
fprintf(stream, "%.2d/%.2d/20%.2d<br><br>", dayint, monthint, yearint);
fputs_P(html_body3, stream);
fflush(stream);
return 0;
}

/*
 * CGI Function - Sets the alarm
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/alarm.cgi'.
 */
int SetAlarm(FILE * stream, REQUEST * req) {
    char *hour,*minute,*day,*month,*year,*snooze,*melody;
    int hourint,minuteint,dayint,monthint,yearint,snoozeint,melodyint;
    int snoozeinttemp=0,melodyinttemp=0,dayinttemp=0,monthinttemp=0;
    int yearinttemp=0,hourinttemp=0,minuteinttemp=0;
    char *c[7];
    char *p[7];
    /*
     * The special type 'prog_char' forces the
     * string literals to be placed in flash ROM.
     * This saves us a lot of RAM.
     */

```

```

    static prog_char html_head[] = "<html><title>Done</title><body><BR><H3
align=\"center\"><font color=\"#FF0000\">Alarm Settings are Saved</font></H3>";
    static prog_char html_body[] = "<center><h4><u><font color=\"#0000FF\"><b>New
Settings:</b></font></u></h4>";
    static prog_char html_body1[] = "<font color=\"#0000FF\"><b>Time: </b></font>";
    static prog_char html_body2[] = "<font color=\"#0000FF\"><b>Date: </b></font>";
    static prog_char html_body3[] = "<a href=\"/index.html\">return to
main</a></center></body></html>";

    /* Send HTML header. */
    fputs_P(html_head, stream);

    if (req->req_query) {
        u_char i;

        /* Extract 7 parameters. */
        for (i = 0; i < 7; i++) {
            if (i == 0) {
#ifdef __IMAGECRAFT__
                /*
                * Extract the parameters. Note, that it is potentially
                * dangerous
                * to use strtok in multithreaded applications. There's no
                * problem
                * here, because we can be sure that there will by no thread
                * switch
                * between the first and the last call to strtok. Otherwise we
                * must use strtok_r().
                */
                c[0] = strtok(req->req_query, "=");
            #else
                /*
                * There's no strtok in the GCC library. So we take the chance
                * to
                * demonstrate strtok_r().
                */
                c[0] = strtok_r(req->req_query, "=", &qpl);
            #endif
            }
            else {
#ifdef __IMAGECRAFT__
                c[i] = strtok(0, "=");
            #else
                c[i] = strtok_r(0, "=", &qpl);
            #endif
            }
        }
        p[i] = strtok(0, "&");
    #else
        p[i] = strtok_r(0, "&", &qpl);
    #endif
    }

    hourint = atoi(p[0]);
    minuteint = atoi(p[1]);
    dayint = atoi(p[2]);
    monthint = atoi(p[3]);
    yearint = atoi(p[4]);
    melodyint = 0;

```

```
snoozeint = 3;

/* Configure data direction settings for PORTB, PORTD and PORTE
 * PORTE pins 0 to 7 are set to be output
 * PORTD pins 0 and 1 are set to be output
 * PORTE pins 2 to 7 are set to be output
 */
sbi(DDRB,0);
sbi(DDRB,1);
sbi(DDRB,2);
sbi(DDRB,3);
sbi(DDRB,4);
sbi(DDRB,5);
sbi(DDRB,6);
sbi(DDRB,7);
sbi(DDRD,0);
sbi(DDRD,1);
sbi(DDRE,2);
sbi(DDRE,3);
sbi(DDRE,4);
sbi(DDRE,5);
sbi(DDRE,6);
sbi(DDRE,7);

/* sends alarm day setting to clock*/
dayinttemp = dayint & 3;
outp(dayint+128,PORTE);
outp(dayinttemp,PORTD);
outp(0x1A, PORTB);
delay(600);

/* sends alarm hour setting to clock*/
hourinttemp = hourint & 3;
outp(hourint+128,PORTE);
outp(hourinttemp,PORTD);
outp(0x19, PORTB);
delay(600);

/* sends alarm minute setting to clock*/
minuteinttemp = minuteint & 3;
outp(minuteint+128,PORTE);
outp(minuteinttemp,PORTD);
outp(0x18, PORTB);
delay(600);

/* sends alarm melody setting to clock*/
melodyinttemp = melodyint & 3;
outp(melodyint+128,PORTE);
outp(melodyinttemp,PORTD);
outp(0x17, PORTB);
delay(600);

/* sends alarm duration setting to clock*/
snoozeinttemp = snoozeint & 3;
outp(snoozeint+128,PORTE);
outp(snoozeinttemp,PORTD);
outp(0x16, PORTB);
delay(700);

/* Re-configure data direction settings for PORTB, PORTD and PORTE
```

```

    * PORTE pins 0 to 7 are set to be input
    * PORTD pins 0 and 1 are set to be input
    * PORTE pins 2 to 7 are set to be input
    */
    cbi(DDRB,0);
    cbi(DDRB,1);
    cbi(DDRB,2);
    cbi(DDRB,3);
    cbi(DDRB,4);
    cbi(DDRB,5);
    cbi(DDRB,6);
    cbi(DDRB,7);
    cbi(DDRD,0);
    cbi(DDRD,1);
    cbi(DDRE,2);
    cbi(DDRE,3);
    cbi(DDRE,4);
    cbi(DDRE,5);
    cbi(DDRE,6);
    cbi(DDRE,7);
    outp(0xff,PORTE);
    outp(0xff,PORTD);
    outp(0xff,PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    fputs_P(html_body1, stream);
    fprintf(stream, "%.2d:%.2d<br>", hourint, minuteint);
    fputs_P(html_body2, stream);
    fprintf(stream, "%.2d/%.2d/20%.2d<br><br>", dayint, monthint, yearint);
    fputs_P(html_body3, stream);
    return 0;
}

/*
 * CGI Function - For testing
 * Waits until the alarm is on. When the alarm is on, it stops the alarm
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/test.cgi'.
 */
int test(FILE * stream, REQUEST * req)
{
    /*
     * The special type 'prog_char' forces the
     * string literals to be placed in flash ROM.
     * This saves us a lot of RAM.
     */
    static prog_char html_head[] = "<html><title>Done</title><body><BR><H3
align=\"center\"><font color=\"#FF0000\">Alarm Test</font></H3>";
    static prog_char html_body[] = "<a href=\"/index.html\">return to
main</a></center></body></html>";

    while(1)
    {

```

```

        if(alarmstatus()==0){
            continue;}
        else
        {
            stopalarming();
            break;
        }
    }
    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 1;
}

/*
 * CGI Function - Stops alarm
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/stop.cgi'.
 */
int stopalarm(FILE * stream, REQUEST * req) {
    /*
     * The special type 'prog_char' forces the
     * string literals to be placed in flash ROM.
     * This saves us a lot of RAM.
     */
    static prog_char html_head[] = "<html><title>Done</title><body><BR><H3
align=\"center\"><font color=\"#FF0000\">The Alarm is Stopped</font></H3>";
    static prog_char html_body[] = "<center><a href=\"/index.html\">return to
main</a></center></body></html>";

    /*Send the stop signal*/
    stopalarming();

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);
    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    fflush(stream);
    return 1;
}

/*
 * Reads the alarm status from the clock
 * and returns 1 if it is on or 0 otherwise
 */
int alarmstatus()
{
    int status,i;
    for(i=0;i<=50;i++){
        status = inb(PINB);
        status &= 0x80;
        if(status == 128){

```

```

        return 1;
    }
    else{
        continue;
    }
}
return 0;
}
/*
 * Sends signal to the clock to stop alarming
 */
int stopalarming()
{
    long i;
    sbi(DDRB,5);
    sbi(DDRB,6);
    outp(0x20,PORTB);
    NutSleep(1500);
    cbi(DDRB,5);
    cbi(DDRB,6);
    outp(0xff,PORTB);
    return 1;
}

/*
 * HTTP service thread.
 *
 * The endless loop in this thread waits for a client connect,
 * processes the HTTP request and disconnects. Nut/Net doesn't
 * support a server backlog. If one client has established a
 * connection, further connect attempts will be rejected.
 * Typically browsers open more than one connection in order
 * to load images concurrently. So we run this routine by
 * several threads.
 */
THREAD(Service, arg)
{
    TCPSOCKET *sock;
    FILE *stream;
    u_char id = (u_char) ((u_short) arg);

    /*
     * loop endless for connections.
     */
    for (;;) {

        /*
         * Create a socket.
         */
        if ((sock = NutTcpCreateSocket()) == 0) {
            printf("[%u] Creating socket failed\n", id);
            NutSleep(5000);
            continue;
        }

        /*
         * Listen on port 80. This call will
         * block until we get a connection
         * from a client.

```

```

    */
    NutTcpAccept(sock, 80);
    printf("[%u] Connected, %u bytes free\n", id, NutHeapAvailable());

    /*
     * Wait until at least 8 kByte of free RAM is
     * available. This will keep the client
     * connected in low memory situations.
     */
    while (NutHeapAvailable() < 8192) {
        printf("[%u] Low mem\n", id);
        NutSleep(1000);
    }

    /*
     * Associate a stream with the socket
     * so we can use standard I/O calls.
     */
    if ((stream = _fdopen((int) sock, "r+b")) == 0) {
        printf("[%u] Creating stream device failed\n", id);
    } else {
        /*
         * This API call parses the client's HTTP request,
         * send any requested file from the
         * registered file system or handle CGI requests by calling
         * our registered CGI routine.
         */
        NutHttpRequestProcess(stream);

        /*
         * Close the virtual stream device.
         */
        fclose(stream);
    }

    /*
     * Close our socket.
     */
    NutTcpCloseSocket(sock);
    printf("[%u] Disconnected\n", id);
}

/*
 * ServiceList thread.
 *
 * The endless loop in this thread waits for an appliance connect,
 * processes its request and sends information about its
 * available services.
 */
THREAD(ServiceListThread, arg){
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buffquery[70];

    /* Create a UDP Socket */
    sock = NutUdpCreateSocket(LISTSERVERUDPPORT);
    for(;;){

```

[illegible]

```

the sequence number of the                                * and
                                                           *
acknowledge message is equal,                             * send
                                                           * and
the next service information                               */
                                                           j
the sequence number if available

    if(strncmp(msgrcv,"OK",2)==0){
        dev = atoi(&buffquery[3]);
= dev + 1;
        if(j>=serviceNumber)
            break;
        else
            continue;
    } else{
failed++;
if
(failed==3){
    NutUdpSendTo(sock, ip_addr_from, port_from, "FAILED", sizeof("FAILED"));
break;
} else
continue;
}
else continue;
break;
}
else continue;
}
else continue;
}
else{
printf("\nServer couldn't send a reply to
continue;
}
}
}
}
}
SLQ");
/*
```

```

* DeviceCoordination thread.
*
* The endless loop in this thread checks if the alarm is on or not.
* If the alarm is on and coordination request has already been
* received, it stops alarming and requests the URL which is held by
* coordinationLink from the appliance whose IP address is held by
* the coordinationIP.
*/
THREAD(DeviceCoordination, arg)
{
    size_t len;
    size_t rc;
    int bite;
    u_long rip;
    TCPSOCKET *socknew;

    for(;;){
        /* Checks alarm status*/
        if(alarmstatus() == 0){
            /* If alarm is off, thread sleeps for 1 second*/
            NutSleep(1000);
        }
        /* if the alarm is on, it checks if there is a coordination request*/
        else if((strcmp(coordinationIP, " ") != 0) && (strcmp(coordinationLink, " ")
!= 0)){
            /* stops alarming*/
            stopalarming();
            if ((socknew = NutTcpCreateSocket()) != 0) {
                /* connects to the target appliance*/
                if (NutTcpConnect(socknew, inet_addr(coordinationIP), 80) ==
0) {
                    /* requests the URL which is located in the
coordinationLink */
                    strcpy(buffer, "GET /" );
                    strcat(buffer, coordinationLink);
                    strcat(buffer, " HTTP/1.0\r\nAccept: text/plain,
text/html, image/gif, image/jpeg, */*\r\n\r\n");
                    len = (int) strlen(buffer);
                    for (rc = 0; rc < len; rc += bite){
                        if ((bite = NutTcpSend(socknew, buffer + rc,
len - rc)) <= 0){
                            break;
                        }
                    }
                    while ((bite = NutTcpReceive(socknew, buffer,
sizeof(buffer))) > 0) {
                        }
                    }
                }
                else {
                    printf("\n Connection refused by the remote");
                }
                strcpy(coordinationLink, " ");
                strcpy(coordinationIP, " ");
            }
        }
    }
}

```

```

/*
 * ServerQuery thread.
 *
 * The endless loop in this thread broadcasts a Server Query
 * when it is created. If the HAN server is not located
 * by this broadcast, thread waits for a HAN Server
 * Online announcement. After locating the HAN Server
 * either by a Server query or a HAN Server Online
 * announcement, appliance registers itself with the
 * HAN Server, by requesting register.cgi with its IP and
 * MAC addresses and its definition. Registration takes
 * place in every 40 seconds. If it fails to locate the
 * HAN Server during registration, it broadcasts a Server
 * Query and wait for a HAN Server response.
 */
THREAD(ServerQuery, arg) {

    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    TCPSOCKET *socknew;
    int i,len,rc,bite,total_bytes = 0;

    /*
     * serverFlag = 1 shows if the HAN Server has already
     * been working in the Appliance Network when the
     * client become online. serverFlag = 0 shows that the client
     * appliance become available before the HAN Server.
     */
    int serverFlag = -1;

    /*Opens a UDP PORT.*/
    sock = NutUdpCreateSocket(CLIENTUDPPORT);
    /* ServerQuery thread works as long as the client works. */
    for(;;){
        buffquery[0] = 'c';
        /*
         * Client tries to locate the HAN Server by sending Broadcast UDP
         * messages to the HAN. Client tries to locate the server 3 times.
         * If it doesn't locate the HAN Server, it starts to wait the Server's
         * online announcement.
         */
        for(i = 1; i < 4; i++){
            buffquery[0] = 'c';
            /*Client asks SIP (Server IP) by broadcast.*/
            if(NutUdpSendTo(sock, inet_addr("255.255.255.255"), SERVERUDPPORT,
"SIP", 3) == 0){
                /*
                 * If the Server is Online, it answers with IPA (IP Address)
                 * by unicast.
                 */
                buffquery[0] = 'c';
                if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from,
buffquery, 512, 5000) > 0){
                    if(strncmp(buffquery, "IPA", 3) == 0){
                        printf("Server is Located: %s",
inet_ntoa(ip_addr_from));

                        serverIP = ip_addr_from;
                        /*Server is located.*/
                        serverFlag = 1;

```

```

serverOnline = 1;
break;
    }
    }
    else{
        printf("Failed...");
    }
}
/*
 * Server cannot be located.
 */
serverFlag = 0;
serverOnline = 0;
printf("\nNo Server is Located");
}
/*
 * If the server is not located in the query, Client waits for
 * HAN Server Announcement Message.
 */
if(serverFlag == 0){
    for(;;){
        buffquery[0] = 'c';
        if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from,
buffquery, 4, NUT_WAIT_INFINITE) > 0){
            if(strncmp(buffquery, "IPA", 3) == 0){
                printf("\nNew HAN Server is Located: %s",
inet_ntoa(ip_addr_from));

                serverOnline = 1;
                serverIP = ip_addr_from;
                break;
            }
        }
        else continue;
    }
}

/* Register appliance with the HAN Server every 60 seconds*/
for(;;){
    buffquery[0] = 'c';
    /*Create a TCP PORT.*/
    socknew = NutTcpCreateSocket();
    printf("\nConnecting %s.", inet_ntoa(serverIP));
    if(NutTcpConnect(socknew, serverIP, 80) == 0) {
        /* Server is located */
        printf(" Connected.");
        strcpy(buffquery, "GET ");
        strcat(buffquery, "/cgi-bin/register.cgi?mac=");
        strcat(buffquery, MAC);
        strcat(buffquery, "&def=");
        strcat(buffquery, DEFINITION);
        strcat(buffquery, "&ip=");
        strcat(buffquery, inet_ntoa(confnet.cdn_ip_addr));
        strcat(buffquery, " HTTP/1.0\r\nUser-Agent: Ethernut [en]

(NutOS)\r\n\r\n");

        len = (int) strlen(buffquery);
        /* Send the registration request */
        for (rc = 0; rc < len; rc += bite){
            if ((bite = NutTcpSend(socknew, buffquery + rc, len -
rc)) <= 0){
                break;
            }
        }
    }
}

```

```

    }
    while ((bite = NutTcpReceive(socknew, buffquery,
sizeof(buffquery) - 1)) > 0) {
        total_bytes += bite;
        buffquery[bite] = 0;
    }
    NutTcpDestroySocket(socknew);
    NutSleep(60000);
}
else {
    /* Server cannot be located */
    serverOnline = 0;
    NutSleep(40000);
    break;
}
}
}
NutUdpDestroySocket(sock);
}

/*
 * Randomly chooses an IP address based on the previously
 * defined BASENETWORKADDRESS.
 */
void Randomize(u_char *mac)
{
    for(;;)
    {
        u_long cpu_crystal = NutGetTickCount();
        u_long ip;
        int ip_last;

        NETBUF *nb = NutNetBufAlloc(0, NBAF_APPLICATION, 50);

        /*
         * An IP address is chosen randomly using the tick count of
         * the cpu crystal and the RANDOMIZATIONCONSTANT.
         */
        ip_last = (int) (((cpu_crystal + RANDOMIZATIONCONSTANT) % 253) + 2);
        ip = inet_addr(BASENETWORKADDRESS) + 16777216 * ip_last;

        NutNetIfConfig("eth0", mac, ip, inet_addr(BASENETWORKMASK));
        /*
         * ICMP echo request message is sent to check if the IP
         * address is used by another device.
         */
        if(NutIcmpOutput(8, ip, nb) == -1)
        {
            NutNetBufFree(nb);
            break;
        }
        /*
         * If the IP Address is used by another device, randomly choose
         * another IP address.
         */
        else
        {
            NutNetBufFree(nb);
            continue;
        }
    }
}

```

```

    }
}

/*
 * automatically determines the network settings
 * for our network.
 */
void AutoIp(u_long old_ip, u_long old_mask, u_char *mac)
{
    /*
     * we check whether the previously stored IP address is valid for
     * the network we are in. If the previously stored IP address
     * is not valid for our network, we randomly choose
     * an IP address based on the predefined BASENETWORKIP. If the
     * previously stored IP address is valid for our network, first, we
     * check if the IP address is in use in our network. If yes, we
     * randomly choose another IP address. If no, we used the IP address
     * to configure our Ethernet Home Server.
     */
    if((old_ip & inet_addr(BASENETWORKMASK)) != inet_addr(BASENETWORKADDRESS))
    {
        Randomize(mac);
    }
    else
    {
        NETBUF *nb = NutNetBufAlloc(0, NBUF_APPLICATION, 50);
        Nutsleep(1000);
        NutNetIfConfig("eth0", mac, old_ip, old_mask);

        /*
         * ICMP echo request message is sent.
         */
        if(NutIcmpOutput(8, old_ip, nb) == 0)
        {
            NutNetBufFree(nb);
            Randomize(mac);
        }
        else
        {
            NutNetBufFree(nb);
        }
    }
    BaseNetworkIP = (confnet.cdn_ip_addr & confnet.cdn_ip_mask);
    BaseNetworkMASK = confnet.cdn_ip_mask;
    NetworkIP = confnet.cdn_ip_addr;
}

/*
 * Fills servList struct with the service
 * information available for device coordination
 */
void fillservList()
{
    strcpy(servList[0].definition, "Set_Clock");
    strcpy(servList[0].url, "setclock.cgi");
    strcpy(servList[1].definition, "Set_Alarm");
    strcpy(servList[1].url, "setclock.cgi");
}

```

```

/*!
 * Main application routine.
 *
 * Nut/OS automatically calls this entry after initialization.
 */
int main(void)
{
    u_long baud = 115200;
    u_char i;
    BaseNetworkIP = inet_addr(BASENETWORKADDRESS);
    BaseNetworkMASK = inet_addr(BASENETWORKMASK);

    /* Allocate space for list that holds the appliance information*/
    aplist = (struct Appliance *)malloc(NUMBEROFAPPLIANCES * sizeof(struct Appliance));
    memset(aplist, 0, NUMBEROFAPPLIANCES * sizeof(struct Appliance));
    /* Allocate space for list that holds the service information*/
    servList = (struct Service *)malloc(NUMBEROFSERVICES * sizeof(struct Service));
    memset(servList, 0, NUMBEROFSERVICES * sizeof(struct Service));

    fillservList();
    outp(0xFC, PORTE);
    outp(0xFF, PORTB);
    outp(0x03, PORTD);
    strcpy(coordinationIP, " ");
    strcpy(coordinationLink, " ");
    /*
     * Initialize the uart device.
     */
    NutRegisterDevice(&devDebug0, 0, 0);
    freopen("uart0", "w", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    NutSleep(200);
    printf("\n\nOnline Alarm Clock\n");

#ifdef NUTDEBUG
    NutTraceTcp(stdout, 0);
    NutTraceOs(stdout, 0);
    NutTraceHeap(stdout, 0);
    NutTracePPP(stdout, 0);
#endif

    /*
     * Register Ethernet controller.
     */
    #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega103__)
        if (NutRegisterDevice(&DEV_ETHER, 0x8300, 5))
    #else
        if (NutRegisterDevice(&devEth0, NIC_IO_BASE, 0))
    #endif

        puts("Registering device failed");

    /*
     * The Appliance tries to retrieve the local IP address,
     * network mask and default gateway from a DHCP server. If no DHCP
     * server responds within a specified number of milliseconds, the
     * Ethernut randomly chooses an IP address based on the predefined
     * BASENETWORKIP.
     */
    {

```

```

    u_long old_ip;
    u_long old_mask;

    /*
     * Save the last used IP address and IP mask stored
     * in the EEPROM.
     */
    NutNetLoadConfig("eth0");
    old_ip = confnet.cdn_ip_addr;
    old_mask = confnet.cdn_ip_mask;

    /*
     * Clear the last used IP address and IP mask stored
     * in the EEPROM. This is used to guarantee the automatic
     * configuration since the IP address stored in the EEPROM is
     * used when there is no DHCP Server. For example: if we do
     * not clear the EEPROM values and migrate from a network
     * housing a DHCP server to a network without a DHCP server,
     * previously used IP configuration will be used in the
     * new network.
     */
    confnet.cdn_ip_addr = inet_addr("0.0.0.0");
    confnet.cdn_ip_mask = inet_addr("0.0.0.0");
    NutNetSaveConfig();

    /*
     * In the first call to NutDhcpIfConfig no MAC address is
     * passed. If one is found in EEPROM, then the DHCP server
     * will be queried for the related IP settings and the
     * function returns 0. Otherwise it returns -1, but we don't
     * know, whether the EEPROM does't contain a valid MAC address
     * or DHCP failed. Thus NutDhcpIfConfig is called once again
     * but this time with a hardcoded MAC address. If it fails again
     * then we are sure, that there is no available DHCP Server.
     * Last thing we can do is to pass IP address and IP mask to
     * NutNetIfConfig to get the network interface up and running.
     * Therefore, we dynamically determine IP address values by
     * using AutoIP function.
     */
    if (NutDhcpIfConfig("eth0", 0, 20000)) {
        u_char mac[] = { MACADDRESS };
        if(NutDhcpIfConfig("eth0", mac, 20000)){
            AutoIp(old_ip, old_mask, mac);
        }
    }
    NetworkIP = confnet.cdn_ip_addr;
    printf("%s ready\n", inet_ntoa(confnet.cdn_ip_addr));

    /*
     * we save the IP configuration to EEPROM.
     */
    NutNetSaveConfig();
}

/*
 * Register our device for the file system.
 */
NutRegisterDevice(&devUrom, 0, 0);

/*
 * Register CGI Functions
 */

```

```
NutRegisterCgi("setalarm.cgi", SetAlarm);
NutRegisterCgi("setclock.cgi", SetClock);
NutRegisterCgi("stop.cgi", stopalarm);
NutRegisterCgi("test.cgi", test);
NutRegisterCgi("coordination.cgi", Coordination);
NutRegisterCgi("getlist.cgi", ServiceList);
NutRegisterCgi("finish.cgi", FinishCoordination);

/*
 * Protect the cgi-bin directory with
 * user and password.
 */
//NutRegisterAuth("cgi-bin", "root:root");

/*
 * Start four server threads.
 */
for (i = 1; i <= 3; i++) {
    char *thname = "httpd0";

    thname[5] = '0' + i;
    NutThreadCreate(thname, Service, (void *) (u_short) i, 1280);
}

/*
 * Start other threads.
 */
NutThreadCreate("QReply", ServerQuery, (void *) (u_short) 5, 768);
NutThreadCreate("DCoordination", DeviceCoordination, (void *) (u_short) 6, 512);
NutThreadCreate("ServiceListThread", ServiceListThread, (void *) (u_short) 7, 512);

/*
 * Set the priority of the main thread to 253.
 * Setting a thread's priority to 253 makes it
 * the idle thread.
 */
NutThreadSetPriority(253);
for (;;) {
    NutSleep(60000);
}
}
```

Appendix F

THE INTERNET TV REMOTE CONTROLLER SOURCE CODE

```
; RC5 Encoder for Philips TV
; ATMEGA16 version
; Written By Erdiñ Çiftçi- Semih Yumupak
; PORTD receives input from the Ethernet Web Server
; PORTB PIN5 outputs IR Signals

.nolist
.include "m16def.inc"
.list
;counter in order to modulate the signal
.def counter = r18
;number of bits that we have to send to the device. In RC5 standart it is 14.
.def bit_count = r19
.def ptr = r20 ;pointer to show the address of the command
.def zero = r2
.def code1perm = r3
;this includes the bits beginning from the second (toggle) till the 9th.
.def code1 = r21
;this includes the bits from 9th to 14th and the least significant 3 bits are stated as
zero
.def code2 = r22
.def temp1 = r23
.def temp2 = r24
;to handle the toggling bit
.def toggle = r25

.org 0x000
rjmp reset
.org 0x026
rjmp startmod

reset:
    ldi r17, 0b00000000
    out TIMSK, r17
    out TCCR0, r17
    ldi r17, 0b00000000
    out TCNT0, r17

    ldi code1, 0b01000000
    ; set up the stack pointer
    ldi r16, high(RAMEND)
    out SPH, r16
```

```
ldi R16, low(RAMEND)
out SPL, r16

ldi r16, 0xFF
out DDRB, r16 ; output
ldi r16, 0
out DDRD, r16 ; input
sei
;is the compare value at 4 MHz for 36 Khz frequency.
ldi r17, 0b00011011
out OCR0, r17
ldi r17, 0b10000000
mov codeiperm,r17

main:
ldi toggle,0
; rcall delay3
in temp1,PIND
;Loops until data come from port D. We shall not use 0 as a datum value.
cpi temp1, 0
breq main
; for taking data from the switches
; com temp1
; rcall delay3
;-----
; compare the value read from PORTD
cpi temp1,1
breq one
cpi temp1,2
breq two
cpi temp1,3
breq three
cpi temp1,4
breq four
cpi temp1,5
breq five
cpi temp1,6
breq six
cpi temp1,7
breq seven
cpi temp1,8
breq eight
cpi temp1,9
breq nine
cpi temp1,10
breq zero
cpi temp1,11
breq on_off
cpi temp1,12
breq mute
cpi temp1,13
breq volump
cpi temp1,14
breq volumm
cpi temp1,15
breq contrastp
cpi temp1,16
breq contrastm
cpi temp1,17
```

```

    breq channelp
    cpi temp1,18
    breq channelm
    cpi temp1,19
    breq coklu_kanal
    cpi temp1,20
    breq time
    rjmp main
;-----
    finish:

;    ldi ptr,0
;    pointer's value is determined by the input
;    add ptr, temp1
;    ldi ZL, low(TABLE*2)
;    ldi ZH, high(TABLE*2)
;    add ZL, ptr
;    adc ZH, zero
;    take the value from the table
;    lpm
;    mov code2, r0
;    out PORTA, code2

;    clr r0
;    ldi code2,0b00001100
;    code1+code2 is the whole 13 bit code
;    lsl code2
;    lsl code2

    rol code2
    rol code1
    rjmp RC5
;-----
one:    ldi code2,0b00000110
        rjmp finish
two:    ldi code2,0b00001010
        rjmp finish
three:  ldi code2,0b00001110
        rjmp finish
four:   ldi code2,0b00010010
        rjmp finish
five:   ldi code2,0b00010110
        rjmp finish
six:    ldi code2,0b00011010
        rjmp finish
seven:  ldi code2,0b00011110
        rjmp finish
eight:  ldi code2,0b00100010
        rjmp finish
nine:   ldi code2,0b00100110

```

```

        rjmp finish
zero:    ldi code2,0b00000010
        rjmp finish
on_off:  ldi code2,0b00110010
        rjmp finish
mute:    ldi code2,0b00110110
        rjmp finish
volump:  ldi code2,0b01000010
        rjmp finish
volumm:  ldi code2,0b01000110
        rjmp finish
contrastp: ldi code2,0b01110010
        rjmp finish
contrastm: ldi code2,0b01110110
        rjmp finish
channelp: ldi code2,0b10000010
        rjmp finish
channelm: ldi code2,0b10000110
        rjmp finish
coklu_kanal: ldi code2,0b00101010
        rjmp finish
time:    ldi code2,0b10011010
        rjmp finish
maint:   rjmp main

; PHILIPS TV REMOTE CONTROL ENCODING
;
; VALUE TO P1  REMOTE KEY  COMMAND CODE FIELD (DEC)
; -----
;TABLE:
;          .DB 0b00000000, 0b00000100 ; 0, 1
;          .DB 0b00001000, 0b00001100 ; 2, 3
;          .DB 0b00010000, 0b00010100 ; 4, 5
;          .DB 0b00011000, 0b00011100 ; 6, 7
;          .DB 0b00100000, 0b00100100 ; 8, 9
;          .DB 0b00110000, 0b00110100 ; ON/OFF, Mute
;          .DB 0b01000000, 0b01000100 ; Volume+, Volume-
;          .DB 0b01001000, 0b01001100 ; Bright+,Bright- ; 12
;          .DB 0b01110000, 0b01110100 ; Contrast+,Contrast-
;          .DB 0b10000000, 0b10000100 ; Program+,Program-

RC5: ;sends RC5 code of the Philips device
; # of bits of the RC5 code
ldi bit_count, 14
ldi counter, 32
; sends out 36 KHz modulated bits
rjmp startmod

loop:

```

```

        ldi r17, 0b00000000
        out TIMSK, r17
        out TCCR0, r17
        ldi r17, 0b00000000
        out TCNT0, r17
        dec bit_count
        cpi bit_count, 0
        brne jump

; this part changes the toggle bit at the end of the transmission of the code
        lsl code1perm
        lsl code1perm
        rol toggle
        com toggle
        ror toggle
        ror code1perm
        sec
        ror code1perm
        sub toggle,toggle
        ror code1perm
        mov code1,code1perm
        clc
        rol code1perm
;-----
jump:
        cpi bit_count, 0
        breq maint
        cpi bit_count,0
        breq maint
        rcall delay
        sub temp1,temp1
        sub temp2,temp2
        rol code1
        adc temp1,zero
        rol code1
        adc temp2, zero
        mov r17, temp2
        ror temp2
        ror code1
        mov temp2,r17
        mov r17, temp1
        ror temp1
        ror code1
        mov temp1,r17

        clc
; shifts the whole code left
        rol code2
        rol code1
        clc

        sub temp1,temp2
; if the first two bits are equal, it sends out 36 KHz modulated bits for 889us
        cpi temp1, 0
        breq otuziki

        cpi temp1, -1
        breq loop
; if the first two bits are equal, it sends out 36 KHz modulated bits for 889*2us
        cpi temp1, 1

```

```
        breq altmisdort

otuziki:
        ldi counter, 32
        rjmp startmod

altmisdort:
        ldi counter, 64
        rjmp startmod

startmod:
dec counter
cpi counter, 0
breq loop
ldi r17, 0b11111111
out PORTB, r17
; enables the interrupt
ldi r17, 0b00001001
out TCCR0, r17

ldi r17, 0b00000010
out TIMSK, r17

sei
ldi r17, 0b00000000
out PORTB, r17

wait:
rjmp wait

;-----
; approximately 900 micro second delay
delay:
        ldi temp2, $11
d2:
        nop
        ldi temp1, $0f
d1:
                dec temp1
                brne d1
        dec temp2
        brne d2
        ret

;-----
; long delay
delay3:
        ldi temp2, $ff
d21:
        ldi r16, $ff
d12:
                dec r16
                brne d12
        dec temp2
        brne d21
        ret
```

Appendix G

THE INTERNET TV REMOTE CONTROLLER WEB SERVER SOURCE CODE

```
/*
 * MAC Address of the TV_REMOTE
 */
#define MACADDRESS 0x00, 0x06, 0x98, 0x13, 0xAB, 0x01
#define MAC "00069813AB01"
/*
 * Definition of the Appliance
 */
#define DEFINITION "REMOTE_CONTROLLER"
/*
 * BASENETWORKADDRESS and BASENETWORKMASK are used to
 * obtain IP address using the AutoIP function
 * if there is no DHCP Server available in the network.
 * For example, if there is no DHCP Server in the Appliance
 * network, the IP address of the appliance will be 10.10.10.x.
 * RANDOMIZATIONCONSTANT is used to prevent acquiring the same
 * IP address with other appliances when many appliances are
 * connected to the appliance network at the same time.
 */
#define BASENETWORKADDRESS "10.10.10.0"
#define BASENETWORKMASK "255.255.255.0"
#define RANDOMIZATIONCONSTANT 70
/*
 * SERVERUDPPORT is the HAN Server's UDP Port for message exchange.
 * CLIENTUDPPORT is the Appliance's UDP Port which is used for exchanging
 * messages with the HAN Server.
 */
#define SERVERUDPPORT 10005
#define CLIENTUDPPORT 10006
/*
 * LISTSERVERUDPPORT and LISTCLIENTPORT is used the exchange Appliance
 * service lists between appliances.
 */
#define LISTSERVERUDPPORT 10007
#define LISTCLIENTUDPPORT 10008
/*
 * Maximum number of appliances allowed in HAN. For the prototype
 * implementation, maximum number is 10 because of RAM limitations.
 */
#define NUMBEROFAPPLIANCES 10
/*
 * Maximum number of services allowed per appliance. For the prototype
```

```

    * implementation, maximum number is 10 because of RAM limitations.
    */
#define NUMBEROFSERVICES 10
/*
    * Number of Services available for this appliance for device coordination
    */
#define AVAILABLESERVICE 6
/*
    * Include files
    */
#include <string.h>
#include <io.h>
#include <stdlib.h>
/*
    * If Ethernut PCB version 2.x is used, we include network device
    * driver for it is included.
    */
#ifdef ETHERNUT2
#include <dev/lanc111.h>
#else
#include <dev/nicrtl.h>
#endif
#include <dev/debug.h>
#include <dev/urom.h>
#include <dev/hd44780.h>
#include <dev/uartavr.h>
#include <dev/ppp.h>
#include <dev/chat.h>
#include <arpa/inet.h>
#include <netinet/icmp.h>
#include <sys/version.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/event.h>
#include <sys/heap.h>
#include <sys/confnet.h>
#include <sys/socket.h>
#include <pro/httpd.h>
#include <pro/dhcp.h>
#include <net/if_var.h>
#include <net/route.h>
#include <fcntl.h>
#include <netdb.h>
/*
    * If we define NUTDEBUG, debug drivers for
    * debugging are included
    */
#ifdef NUTDEBUG
#include <sys/osdebug.h>
#include <net/netdebug.h>
#endif

static char *html_mt = "text/html";
static u_char buffquery[512];
static u_char buffer[100];
/*
    * Holds the network address of the appliance
    */
u_long BaseNetworkIP;
/*

```

```

    * Holds the network mask of the appliance
    */
    u_long BaseNetworkMASK;
    /*
    * Holds the IP address of the appliance
    */
    u_long NetworkIP;
    /*
    * Holds the IP address of the HAN Server if available
    */
    u_long serverIP;
    /*
    * Holds if the HAN Server is online or not.
    * While 0 means the HAN Server is offline,
    * 1 means the HAN Server is online.
    */
    u_long serverOnline=0;
    /*
    * Holds the IP address of the appliance which will be connected
    * during device coordination.
    */
    char coordinationIP[20];
    /*
    * Holds the target url which will be called during device coordination.
    */
    char coordinationLink[50];

    /*
    * Struct for Appliance Information
    */
    struct Appliance{
        char definition[15];
        char ip[20];
    };
    /*
    * Struct for Appliance's Service Information
    */
    struct Service{
        char definition[15];
        char url[15];
    };

    struct Appliance *aplist;
    struct Service *servList;
    int numberRegistered = -1;

    int serviceNumber = AVAILABLESERVICE;
    int serviceNumberRemote = 0;

    /*
    * The special type 'prog_char' forces the
    * string literals to be placed in flash ROM.
    * This saves us a lot of RAM.
    */
    static prog_char html_head[] = "<html><title>Done</title><body><BR><H3
align=\"center\"><font color=\"\#FF0000\">Signal sent</font></H3>";
    static prog_char html_body[] = "<center><a href=\"/index.html\">return to
main</a></center></body></html>";

    /*

```



```

c[0] =
}
else
{
}
p[extract] =
}
num = atoi(p[0]);
strcpy(def,p[1]);
strcpy(ip,p[2]);
/*
 * Save the appliance
 * the sequence number
 */
if(num==i){
    strcpy(aplist[i].ip,ip);
    strcpy(aplist[i].definition, def);
    sprintf(buff,"OK:%d", i);
    buff[4]='\0';
    /* Send
acknowledge message with the last received sequence number*/
    NutUdpSendTo(sock, serverIP, SERVERUDPPORT, buff, sizeof(buff));
    i++;
    break;
    else continue;
}
/*
 * Send an acknowledge
 * number successfully
 * expected and the
 */
else{
    sprintf(buff,"OK:%d", i);
    NutUdpSendTo(sock, serverIP, SERVERUDPPORT, buff, sizeof(buff));
    failed++;
    if(failed<3)
        continue;
}
}
else{
    failed++;
    if(failed<3){
        continue;
    }
}

```

```

    }
    else{
        NutUdpDestroySocket(sock);
        return -1;
    }
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
else{
    printf("Failed...");
    NutUdpDestroySocket(sock);
    return -1;
}
}
NutUdpDestroySocket(sock);
return i;
}

/*
 * In order to send signals to remote controller's microprocessor,
 * these signals should be output for some time. NutSleep()
 * function is not used here since declaring a PORT as
 * output slows down the thread switching
 * actions. Therefore, the period of sleeping cannot be
 * determined healthily.
 */
void delay(long k){
    long i; int j;
    //spend some time
    for(i=0;i<=k;i++){
        //spend some time
        for(j=0;j<=2;j++){
            ;
        }
    }
    return;
}

/*
 * Used to get the list of services of an appliance from
 * the appliance itself. The service information is saved
 * to servList structure. Returns -1, if the appliance
 * cannot be located or an error occurs

```

```

* during message exchange. Else, returns
* the number of online appliances.
*/
int getServiceList(u_long deviceIP)
{
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buff[70];
    int i;

    /* Create a UDP Socket */
    sock = NutUdpCreateSocket(LISTCLIENTUDPPORT);
    /* Send SLQ(Service List Query) message to the target appliance */
    if(NutUdpSendTo(sock, deviceIP, LISTSERVERUDPPORT, "SLQ", 3) == 0){
        /* Receive the number of available services*/
        if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buff, 10, 10000) > 0){
            if(strncmp(buff, "SNR:", 4) == 0){
                buff[5] = '\0';
                serviceNumberRemote = atoi(buff + 4);
                /* Send the SERVLIST request to the appliance*/
                if(NutUdpSendTo(sock, deviceIP, LISTSERVERUDPPORT,
"SERVLIST", 8) == 0){
                    int failed=0;
                    /*
                    * Continue if the number of services available is
                    * bigger than the number of service information
                    * received.
                    */
                    for(i=0; i<serviceNumberRemote;){
                        int num = 0;
                        char def[15];
                        char url[15];
                        /* Receive service information*/
                        if(NutUdpReceiveFrom(sock, &ip_addr_from,
&port_from, buff, 50, 1000) > 0){
                            /* Extract service's sequence number.
                            definition and IP address*/
                            if(strncmp(buff, "?OK=", 4) == 0){
                                char *c[3];
                                char *p[3];
                                int extract;

                                for(extract=0;extract<3;extract++){
                                    if(extract == 0) {
                                        c[0] =
                                        strtok(buff, "=");

                                        }
                                        else
                                        {
                                            c[extract] =
                                            strtok(0, "=");

                                        }
                                        p[extract] = strtok(0,
"&");

                                    }
                                    num = atoi(p[0]);
                                    strcpy(def,p[1]);
                                    strcpy(url,p[2]);
                                    /*

```

```

to the servlist list structure if
and the number received are equal

        strcpy(servList[i].url,url);
        strcpy(servList[i].definition, def);
i);

message with the last received sequence number*/
deviceIP, LISTSERVERUDPPORT, buff, sizeof(buff));

        if(i>=serviceNumberRemote){

with the last sequence
if the sequence number
received are not equal.

i);
deviceIP, LISTSERVERUDPPORT, buff, sizeof(buff));

        NutUdpDestroySocket(sock);

        }
    }
}

        * Save the service information
        * the sequence number expected
        */
        if(num==i){

                sprintf(buff,"OK:%d",
buff[4]='\0';
/* Send acknowledge
NutUdpSendTo(sock,
i++;

                break;
        }
        else continue;
    }
    /*
    * Send an acknowledge message
    * number successfully received
    * expected and the number
    */
    else{
        sprintf(buff,"OK:%d",
NutUdpSendTo(sock,
failed++;
if(failed<3)
        continue;
    }
}
else{
    failed++;
    if(failed<3){
        continue;
    }
    else{
        return -1;
    }
}
}
else{
    NutUdpDestroySocket(sock);
    return -1;
}
}
}

```

```

        else{
            NutUdpDestroySocket(sock);
            return -1;
        }
    }
    else{
        NutUdpDestroySocket(sock);
        return -1;
    }
}
else{
    printf("Failed...");
    NutUdpDestroySocket(sock);
    return -1;
}
}
NutUdpDestroySocket(sock);
return i;
}

/*
 * CGI Function - TurnOFF
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpProcessRequest() when the client
 * request the URL 'cgi-bin/close.cgi'.
 */
int Close(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(11, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Channel Minus
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpProcessRequest() when the client
 * request the URL 'cgi-bin/channelm.cgi'.
 */
int ChannelM(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(18, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

```

```

        fputs_P(html_head, stream);
        fputs_P(html_body, stream);
        return 0;
    }

/*
 * CGI Function - Channel Plus
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/channelp.cgi'.
 */
int Channelp(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(17, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Volume Down
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/volm.cgi'.
 */
int VolM(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(14, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Volume Up
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/volp.cgi'.
 */
int VolP(FILE * stream, REQUEST * req)
{

```

```
    outp(0xFF, DDRB);
    outp(13, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Mute
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpProcessRequest() when the client
 * request the URL 'cgi-bin/mute.cgi'.
 */
int Mute(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(12, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 1
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpProcessRequest() when the client
 * request the URL 'cgi-bin/one.cgi'.
 */
int One(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(1, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}
```

```
/*
 * CGI Function - Opens Channel 2
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/two.cgi'.
 */
int Two(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(2, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 3
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/three.cgi'.
 */
int Three(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(3, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 4
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/four.cgi'.
 */
int Four(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(4, PORTB);
    NutSleep(50);
    outp(0, PORTB);
}
```

```
/* These useful API calls create a HTTP response for us. */
NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBot(stream, html_mt, -1);

fputs_P(html_head, stream);
fputs_P(html_body, stream);
return 0;
}

/*
 * CGI Function - Opens Channel 5
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/five.cgi'.
 */
int Five(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(5, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 6
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/six.cgi'.
 */
int Six(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(6, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 7
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
```

```
* request the URL 'cgi-bin/seven.cgi'.
*/
int Seven(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(7, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 8
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/eight.cgi'.
 */
int Eight(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(8, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * CGI Function - Opens Channel 9
 *
 * This routine must have been registered by NutRegisterCgi() and is
 * automatically called by NutHttpRequest() when the client
 * request the URL 'cgi-bin/nine.cgi'.
 */
int Nine(FILE * stream, REQUEST * req)
{
    outp(0xFF, DDRB);
    outp(9, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
```

```
        fputs_P(html_body, stream);
        return 0;
    }

    /*
    * CGI Function - Opens Channel 0
    *
    * This routine must have been registered by NutRegisterCgi() and is
    * automatically called by NutHttpRequest() when the client
    * request the URL 'cgi-bin/zero.cgi'.
    */
    int Zero(FILE * stream, REQUEST * req)
    {
        outp(0xFF, DDRB);
        outp(10, PORTB);
        NutSleep(50);
        outp(0, PORTB);

        /* These useful API calls create a HTTP response for us. */
        NutHttpSendHeaderTop(stream, req, 200, "Ok");
        NutHttpSendHeaderBot(stream, html_mt, -1);

        fputs_P(html_head, stream);
        fputs_P(html_body, stream);
        return 0;
    }

    /*
    * CGI Function - Double
    *
    * This routine must have been registered by NutRegisterCgi() and is
    * automatically called by NutHttpRequest() when the client
    * request the URL 'cgi-bin/double.cgi'.
    */
    int Double(FILE * stream, REQUEST * req)
    {
        outp(0xFF, DDRB);
        outp(19, PORTB);
        NutSleep(50);
        outp(0, PORTB);

        /* These useful API calls create a HTTP response for us. */
        NutHttpSendHeaderTop(stream, req, 200, "Ok");
        NutHttpSendHeaderBot(stream, html_mt, -1);

        fputs_P(html_head, stream);
        fputs_P(html_body, stream);
        return 0;
    }

    /*
    * CGI Function - Sets Timer
    *
    * This routine must have been registered by NutRegisterCgi() and is
    * automatically called by NutHttpRequest() when the client
    * request the URL 'cgi-bin/timer.cgi'.
    */
    int Timer(FILE * stream, REQUEST * req)
    {
        outp(0xFF, DDRB);
```

```

    outp(20, PORTB);
    NutSleep(50);
    outp(0, PORTB);

    /* These useful API calls create a HTTP response for us. */
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBot(stream, html_mt, -1);

    fputs_P(html_head, stream);
    fputs_P(html_body, stream);
    return 0;
}

/*
 * HTTP service thread.
 *
 * The endless loop in this thread waits for a client connect,
 * processes the HTTP request and disconnects. Nut/Net doesn't
 * support a server backlog. If one client has established a
 * connection, further connect attempts will be rejected.
 * Typically browsers open more than one connection in order
 * to load images concurrently. So we run this routine by
 * several threads.
 */
THREAD(Service, arg)
{
    TCPSOCKET *sock;
    FILE *stream;
    u_char id = (u_char) ((u_short) arg);

    /*
     * loop endless for connections.
     */
    for (;;) {

        /*
         * Create a socket.
         */
        if ((sock = NutTcpCreateSocket()) == 0) {
            printf("[%u] Creating socket failed\n", id);
            NutSleep(5000);
            continue;
        }

        /*
         * Listen on port 80. This call will
         * block until we get a connection
         * from a client.
         */
        NutTcpAccept(sock, 80);
        printf("[%u] Connected, %u bytes free\n", id, NutHeapAvailable());

        /*
         * Wait until at least 8 kByte of free RAM is
         * available. This will keep the client
         * connected in low memory situations.
         */
        while (NutHeapAvailable() < 8192) {
            printf("[%u] Low mem\n", id);

```

```

        NutSleep(1000);
    }

    /*
    * Associate a stream with the socket
    * so we can use standard I/O calls.
    */
    if ((stream = _fdopen((int) sock, "r+b")) == 0) {
        printf("[%u] Creating stream device failed\n", id);
    } else {
        /*
        * This API call parses the client's HTTP request,
        * send any requested file from the
        * registered file system or handle CGI requests by calling
        * our registered CGI routine.
        */
        NutHttpProcessRequest(stream);

        /*
        * Close the virtual stream device.
        */
        fclose(stream);
    }

    /*
    * Close our socket.
    */
    NutTcpCloseSocket(sock);
    printf("[%u] Disconnected\n", id);
}

/*
* ServiceList thread.
* The endless loop in this thread waits for an appliance connect,
* processes its request and sends information about its
* available services.
*/
THREAD(ServiceListThread, arg){
    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    u_char buffquery[70];

    /* Create a UDP Socket */
    sock = NutUdpCreateSocket(LISTSERVERUDPPORT);
    for(;;){
        /* Wait for an SLQ(Service List Query) message from an appliance*/
        if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buffquery, 30,
NUT_WAIT_INFINITE) > 0){
            if(strncmp(buffquery, "SLQ", 3) == 0){
                int i;
                for(i=0; i<3; i++){
                    int j=0, kj=0;
                    char numberRegistered[7];
                    sprintf(numberRegistered, "SNR:%d", serviceNumber);
                    /* Send the number of available services*/

```

```

        if(NutUdpSendTo(sock, ip_addr_from, port_from,
numberRegistered, sizeof(numberRegistered))==0){
        /* Wait for the SERVLIST request from the
        appliance*/
        if(NutUdpReceiveFrom(sock, &ip_addr_from,
&port_from, buffquery, 35, 10000) > 0){
            if(strncmp(buffquery, "SERVLIST", 8) ==
0){
                int failed=0;
                /*
                * Continue if the number of services
                * smaller than the last sequence number
                * with the acknowledge message
                */
                for(j=0;j<serviceNumber;){
                    char msg[100];
                    sprintf(msg,"?OK=%d",j);
                    strcat(msg,"&def=");

                    strcat(msg,servList[j].definition);

                    strcat(msg,servList[j].url);

                    /* Send service
                    information*/
                    if(NutUdpSendTo(sock,
                    ip_addr_from, port_from, msg, sizeof(msg))==0){
                        /*
                        * Receive the
                        * last service
                        * target
                        */
                        if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from, buffquery, 35, 1000) > 0){
                            char
                            msgrcv[5];
                            int dev =
                            -1;

                            strncpy(msgrcv, buffquery,2);

                            = '\0';

                            last sent sequence number
                            the sequence number of the
                            acknowledge message is equal,
                            the next service information
                            the sequence number if available
                        }
                    }
                }
            }
        }
    }

```

```
if(strncmp(msgrcv,"OK",2)==0){  
    dev = atoi(&buffquery[3]);  
  
    = dev + 1;  
  
    if(j>=serviceNumber)  
        break;  
    else  
        continue;  
  
}  
else{  
    failed++;  
    if  
(failed==3){  
        NutUdpSendTo(sock, ip_addr_from, port_from, "FAILED", sizeof("FAILED"));  
        break;  
    }  
    else  
        continue;  
  
} else continue;  
break;  
}  
else continue;  
}  
else continue;  
}  
else{  
    printf("\nServer couldn't send a reply to  
continue;  
}  
}  
}  
}  
}  
}  
/*  
 * ServerQuery thread.  
 */  
 * The endless loop in this thread broadcasts a Server Query  
 * when it is created. If the HAN server is not located  
 * by this broadcast, thread waits for a HAN Server  
 * Online announcement. After locating the HAN Server  
 * either by a Server query or a HAN Server Online  
 * announcement, appliance registers itself with the  
 * HAN Server, by requesting register.cgi with its IP and
```

```

* MAC addresses and its definition. Registration takes
* place in every 40 seconds. If it fails to locate the
* HAN Server during registration, it broadcasts a Server
* Query and wait for a HAN Server response.
*/
THREAD(ServerQuery, arg) {

    UDPSOCKET *sock;
    u_long ip_addr_from;
    u_short port_from;
    TCPSOCKET *socknew;
    int i, len, rc, bite, total_bytes = 0;

    /*
    * serverFlag = 1 shows if the HAN Server has already
    * been working in the Appliance Network when the
    * client become online. serverFlag = 0 shows that the client
    * appliance become available before the HAN Server.
    */
    int serverFlag = -1;

    /*Create a UDP PORT.*/
    sock = NutUdpCreateSocket(CLIENTUDPPORT);

    /* ServerQuery thread works as long as the client works. */
    for(;;){
        buffquery[0] = 'c';
        /*
        * Client tries to locate the HAN Server by sending Broadcast UDP
        * messages to the HAN. Client tries to locate the server 3 times.
        * If it doesn't locate the HAN Server, it starts to wait the Server's
        * online announcement.
        */
        for(i = 1; i < 4; i++){
            buffquery[0] = 'c';
            /*Client asks SIP (Server IP) by broadcast.*/
            if(NutUdpSendTo(sock, inet_addr("255.255.255.255"), SERVERUDPPORT,
"SIP", 3) == 0){
                /*
                * If the Server is Online, it answers with IPA (IP Address)
                * by unicast.
                */
                buffquery[0] = 'c';
                if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from,
buffquery, 512, 5000) > 0){
                    if(strncmp(buffquery, "IPA", 3) == 0){
                        printf("Server is Located: %s",
inet_ntoa(ip_addr_from));

                        serverIP = ip_addr_from;
                        /*Server is located.*/
                        serverFlag = 1;
                        serverOnline = 1;
                        break;
                    }
                }
            }
            else{
                printf("Failed...");
            }
        }
    }
    /*

```

```

        * Server cannot be located.
        */
        serverFlag = 0;
        serverOnline = 0;
        printf("\nNo Server is Located");
    }
    /*
    * If the server is not located in the query, Client waits for
    * HAN Server Announcement Message.
    */
    if(serverFlag == 0){
        for(;;){
            buffquery[0] = 'c';
            if(NutUdpReceiveFrom(sock, &ip_addr_from, &port_from,
buffquery, 4, NUT_WAIT_INFINITE) > 0){
                if(strncmp(buffquery, "IPA", 3) == 0){
                    printf("\nNew HAN Server is Located: %s",
inet_ntoa(ip_addr_from));

                    serverOnline = 1;
                    serverIP = ip_addr_from;
                    break;
                }
            }
            else continue;
        }
    }

    /* Register appliance with the HAN Server every 60 seconds*/
    for(;;){
        buffquery[0] = 'c';
        printf("\nConnecting %s.", inet_ntoa(serverIP));
        /*Create a TCP PORT.*/
        socknew = NutTcpCreateSocket();
        if(NutTcpConnect(socknew, serverIP, 80) == 0) {
            /* Server is located */
            printf(" Connected.");
            strcpy(buffquery, "GET ");
            strcat(buffquery, "/cgi-bin/register.cgi?mac=");
            strcat(buffquery, MAC);
            strcat(buffquery, "&def=");
            strcat(buffquery, DEFINITION);
            strcat(buffquery, "&ip=");
            strcat(buffquery, inet_ntoa(confnet.cdn_ip_addr));
            strcat(buffquery, " HTTP/1.0\r\nUser-Agent: Ethernut [en]

(NutOS)\r\n\r\n");

            len = (int) strlen(buffquery);
            /* Send the registration request */
            for (rc = 0; rc < len; rc += bite){
                if ((bite = NutTcpSend(socknew, buffquery + rc, len -
rc)) <= 0){
                    break;
                }
            }
            while ((bite = NutTcpReceive(socknew, buffquery,
sizeof(buffquery) - 1)) > 0) {
                total_bytes += bite;
                buffquery[bite] = 0;
            }
            NutTcpDestroySocket(socknew);
        }
    }

```

```

        NutSleep(60000);
    }
    else {
        /* Server cannot be located */
        serverOnline = 0;
        NutSleep(40000);
        break;
    }
}

NutUdpDestroySocket(sock);
}

/*
 * Randomly chooses an IP address based on the previously
 * defined BASENETWORKADDRESS.
 */
void Randomize(u_char *mac)
{
    for(;;)
    {
        u_long cpu_crystal = NutGetTickCount();
        u_long ip;
        int ip_last;

        NETBUF *nb = NutNetBufAlloc(0, NBAF_APPLICATION, 50);

        /*
         * An IP address is chosen randomly using the tick count of
         * the cpu crystal and the RANDOMIZATIONCONSTANT.
         */
        ip_last = (int) (((cpu_crystal + RANDOMIZATIONCONSTANT) % 253) + 2);
        ip = inet_addr(BASENETWORKADDRESS) + 16777216 * ip_last;

        NutNetIfConfig("eth0", mac, ip, inet_addr(BASENETWORKMASK));
        /*
         * ICMP echo request message is sent to check if the IP
         * address is used by another device.
         */
        if(NutIcmpOutput(8, ip, nb) == -1)
        {
            NutNetBufFree(nb);
            break;
        }
        /*
         * If the IP Address is used by another device, randomly choose
         * another IP address.
         */
        else
        {
            NutNetBufFree(nb);
            continue;
        }
    }
}

/*
 * automatically determines the network settings
 * for our network.

```

```

*/
void AutoIp(u_long old_ip, u_long old_mask, u_char *mac)
{
    /*
     * we check whether the previously stored IP address is valid for
     * the network we are in. If the previously stored IP address
     * is not valid for our network, we randomly choose
     * an IP address based on the predefined BASENETWORKIP. If the
     * previously stored IP address is valid for our network, first, we
     * check if the IP address is in use in our network. If yes, we
     * randomly choose another IP address. If no, we used the IP address
     * to configure our Ethernet Home Server.
     */
    if((old_ip & inet_addr(BASENETWORKMASK)) != inet_addr(BASENETWORKADDRESS))
    {
        Randomize(mac);
    }
    else
    {
        NETBUF *nb = NutNetBufAlloc(0, NBUF_APPLICATION, 50);
        NutSleep(1000);
        NutNetIfConfig("eth0", mac, old_ip, old_mask);

        /*
         * ICMP echo request message is sent.
         */
        if(NutIcmpOutput(8, old_ip, nb) == 0)
        {
            NutNetBufFree(nb);
            Randomize(mac);
        }
        else
        {
            NutNetBufFree(nb);
        }
    }
    BaseNetworkIP = (confnet.cdn_ip_addr & confnet.cdn_ip_mask);
    BaseNetworkMASK = confnet.cdn_ip_mask;
    NetworkIP = confnet.cdn_ip_addr;
}

/*
 * Fills servList struct with the service
 * information available for device coordination
 */
void fillservList()
{
    strcpy(servList[0].definition, "Turn_Off");
    strcpy(servList[0].url, "close.cgi");
    strcpy(servList[1].definition, "Channel +");
    strcpy(servList[1].url, "chp.cgi");
    strcpy(servList[2].definition, "Channel -");
    strcpy(servList[2].url, "chm.cgi");
    strcpy(servList[3].definition, "Vol +");
    strcpy(servList[3].url, "volp.cgi");
    strcpy(servList[4].definition, "Vol -");
    strcpy(servList[4].url, "volm.cgi");
    strcpy(servList[5].definition, "Mute");
    strcpy(servList[5].url, "mute.cgi");
}

```

```

/*!
 * Main application routine.
 *
 * Nut/OS automatically calls this entry after initialization.
 */
int main(void)
{
    u_long baud = 115200;
    u_char i;
    BaseNetworkIP = inet_addr(BASENETWORKADDRESS);
    BaseNetworkMASK = inet_addr(BASENETWORKMASK);

    /* Allocate space for list that holds the appliance information*/
    aplist = (struct Appliance *)malloc(NUMBEROFAPPLIANCES * sizeof(struct Appliance));
    memset(aplist, 0, NUMBEROFAPPLIANCES * sizeof(struct Appliance));
    /* Allocate space for list that holds the service information*/
    servList = (struct Service *)malloc(NUMBEROFSERVICES * sizeof(struct Service));
    memset(servList, 0, NUMBEROFSERVICES * sizeof(struct Service));

    fillservList();

    strcpy(coordinationIP, " ");
    strcpy(coordinationLink, " ");
    /*
     * Initialize the uart device.
     */
    NutRegisterDevice(&devDebug0, 0, 0);
    freopen("uart0", "w", stdout);
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
    NutSleep(200);
    printf("\n\nOnline TV Remote Controller\n");

#ifdef NUTDEBUG
    NutTraceTcp(stdout, 0);
    NutTraceOs(stdout, 0);
    NutTraceHeap(stdout, 0);
    NutTracePPP(stdout, 0);
#endif

    /*
     * Register Ethernet controller.
     */
    #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega103__)
        if (NutRegisterDevice(&DEV_ETHER, 0x8300, 5))
    #else
        if (NutRegisterDevice(&devEth0, NIC_IO_BASE, 0))
    #endif
        puts("Registering device failed");

    /*
     * The Appliance tries to retrieve the local IP address,
     * network mask and default gateway from a DHCP server. If no DHCP
     * server responds within a specified number of milliseconds, the
     * Ethernut randomly chooses an IP address based on the predefined
     * BASENETWORKIP.
     */
    {

```

```

    u_long old_ip;
    u_long old_mask;

    /*
     * Save the last used IP address and IP mask stored
     * in the EEPROM.
     */
    NutNetLoadConfig("eth0");
    old_ip = confnet.cdn_ip_addr;
    old_mask = confnet.cdn_ip_mask;

    /*
     * Clear the last used IP address and IP mask stored
     * in the EEPROM. This is used to guarantee the automatic
     * configuration since the IP address stored in the EEPROM is
     * used when there is no DHCP Server. For example: if we do
     * not clear the EEPROM values and migrate from a network
     * housing a DHCP server to a network without a DHCP server,
     * previously used IP configuration will be used in the
     * new network.
     */
    confnet.cdn_ip_addr = inet_addr("0.0.0.0");
    confnet.cdn_ip_mask = inet_addr("0.0.0.0");
    NutNetSaveConfig();

    /*
     * In the first call to NutDhcpIfConfig no MAC address is
     * passed. If one is found in EEPROM, then the DHCP server
     * will be queried for the related IP settings and the
     * function returns 0. Otherwise it returns -1, but we don't
     * know, whether the EEPROM does't contain a valid MAC address
     * or DHCP failed. Thus NutDhcpIfConfig is called once again
     * but this time with a hardcoded MAC address. If it fails again
     * then we are sure, that there is no available DHCP Server.
     * Last thing we can do is to pass IP address and IP mask to
     * NutNetIfConfig to get the network interface up and running.
     * Therefore, we dynamically determine IP address values by
     * using AutoIP function.
     */
    if (NutDhcpIfConfig("eth0", 0, 20000)) {
        u_char mac[] = { MACADDRESS };
        if (NutDhcpIfConfig("eth0", mac, 20000)) {
            AutoIp(old_ip, old_mask, mac);
        }
    }
    NetworkIP = confnet.cdn_ip_addr;
    printf("%s ready\n", inet_ntoa(confnet.cdn_ip_addr));

    /*
     * we save the IP configuration to EEPROM.
     */
    NutNetSaveConfig();
}

/*
 * Register our device for the file system.
 */
NutRegisterDevice(&devUrom, 0, 0);

/*
 * Register CGI Functions

```

```

*/
NutRegisterCgi("close.cgi", Close);
NutRegisterCgi("chm.cgi", ChannelM);
NutRegisterCgi("chp.cgi", ChannelP);
NutRegisterCgi("volm.cgi", VolM);
NutRegisterCgi("volp.cgi", VolP);
NutRegisterCgi("mute.cgi", Mute);
NutRegisterCgi("zero.cgi", Zero);
NutRegisterCgi("one.cgi", One);
NutRegisterCgi("two.cgi", Two);
NutRegisterCgi("three.cgi", Three);
NutRegisterCgi("four.cgi", Four);
NutRegisterCgi("five.cgi", Five);
NutRegisterCgi("six.cgi", Six);
NutRegisterCgi("seven.cgi", Seven);
NutRegisterCgi("eight.cgi", Eight);
NutRegisterCgi("nine.cgi", Nine);
NutRegisterCgi("double.cgi", Double);
NutRegisterCgi("timer.cgi", Timer);

/*
 * Protect the cgi-bin directory with
 * user and password.
 */
//NutRegisterAuth("cgi-bin", "root:root");

/*
 * Start four server threads.
 */
for (i = 1; i <= 3; i++) {
    char *thname = "httpd0";

    thname[5] = '0' + i;
    NutThreadCreate(thname, Service, (void *) (u_short) i, 1280);
}

/*
 * Start other threads.
 */
NutThreadCreate("QReply", ServerQuery, (void *) (u_short) 5, 768);
NutThreadCreate("ServiceListThread", ServiceListThread, (void *) (u_short) 7, 512);

/*
 * Set the priority of the main thread to 253.
 * Setting a thread's priority to 253 makes it
 * the idle thread.
 */
NutThreadSetPriority(253);
for (;;) {
    NutSleep(60000);
}
}

```

BIBLIOGRAPHY

- [1] G. Borriello, R. Want, "Embedded Computation Meets the World Wide Web", *Communications of the ACM*, 2000, pp. 59–66
- [2] Available Online: <http://www.salutation.org>
- [3] Available Online: <http://www.jini.org>
- [4] Adrian Friday, Nigel Davies, Elaine Catterall, "Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments", Second ACM International Workshop on Data Engineering for Wireless and Mobile Access, May 2001
- [5] Java RMI Specifications, Available Online:
<http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html>
- [6] Bob Pascoe, "Salutation-Lite: Find-and-Bind Technologies for Mobile Devices", Salutation Consortium, June 6, 1999
- [7] R. Lee, S. Gibbs, A. Dara-Abrams, E. Eytchison, "Networking Home Entertainment Devices with HAVi", *Computer*, Vol. 33 Issue 7, 2000, pp. 171-178
- [8] Sumi Helal, "Standards for Service Discovery and Delivery", *IEEE Pervasive Computing*, July/Sept. 2002
- [9] Björn H. Gerth, "Service Discovery in Home Environments", Braunschweig, 2003
- [10] Andrew Liu, "Universal Plug and Play Connects the Home", *Intel Developer Update Magazine*, July 2000
- [11] J. Cohen, S. Aggarwal, Y. Y. Goland, "General Event Notification Architecture Base: Client to Arbiter", IETF draft, Available Online: <http://www.upnp.org/download/draft-cohen-gena-client-01.txt>
- [12] John Rekish, "UPnP, Jini and Salutation - A Look at Some Popular Coordination Frameworks for Future Networked Devices", California Software Laboratories, June 1999
- [13] UPnP Specifications, Available Online: www.upnp.org/resources/specifications.asp

-
- [14] Bob Pascoe, "Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun", Salutation Consortium, June 1999
- [15] Yaron Y. Goland, "Multicast and Unicast UDP HTTP Messages", IETF draft, Available Online: <http://www.watersprings.org/pub/id/draft-goland-http-udp-00.txt>
- [16] T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators (URL)", Available Online: <http://www.ietf.org/rfc/rfc1738.txt>
- [17] Atinav UPnP SDKs Concepts Guide, Available Online: http://www.atinav.com/download/UPnP_Concepts.pdf
- [18] World Wide Web Consortium, SOAP Specifications, Available Online: <http://www.w3.org/TR/soap/>
- [19] UPnP Forum Web Site: <http://www.upnp.org>
- [20] Stuart Cheshire, "Dynamic Configuration of IPv4 link-local addresses", IETF draft, Available Online: <http://files.zeroconf.org/draft-ietf-zeroconf-ipv4-linklocal.txt>
- [21] World Wide Web Consortium, Extensible Markup Language (XML), Available Online: <http://www.w3.org/XML>
- [22] R. Droms, "Dynamic Host Configuration Protocol", RFC2131, March 1997
- [23] Salutation Consortium, "Salutation Architecture Specification Version 2.0c Part 1", the Salutation Consortium, June 1, 1999.
- [24] Salutation Consortium, "Salutation Architecture Specification Version 2.0c Part 2", the Salutation Consortium, June 1, 1999.
- [25] RC-5 Decoder, Available Online: <http://www.xs4all.nl/~sbp/projects/ircontrol/picir/rc5.htm>
- [26] ASN.1 Information Web Site: <http://asn1.elibel.tm.fr/en/>
- [27] Tatsuo Nakajima, "Experiences with Building Middleware for Audio and Visual Networked Home Appliances on Commodity Software", ACM Multimedia 2002

- [28] Dong-Sung Kim, Jae-Min Lee, Wook Hyun Kwon, In Kwan Yuh, "Design and Implementation of Home Network Systems Using UPnP Middleware for Networked Appliances", IEEE Transactions on Consumer Electronics, pp.963-972,2002
- [29] J. Nichols, B.A. Myers, M. Higgins, J. Hughes, T.K. Harris, R. Rosenfeld, M. Pignol "Generating Remote Control Interfaces for Complex Appliances", Symposium on User Interface Software and Technology of the ACM, pp.161-170, 2002
- [30] J. Nichols, B.A. Myers, "Studying the Use of Handhelds to Control Smart Appliances", ICDCS'03, pp. 274-279, 2003
- [31] Atmel ATmega128 8-bit Microcontroller Datasheet, Available Online:
http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
- [32] M. Weiser, "The Computer for the 21st Century", Scientific American, pp. 66-75, September 1991
- [33] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, T. Winograd, "ICrafter: A Service Framework for Ubiquitous Computing Environments", in UBICOMP, Atlanta, pp. 56-75, 2001
- [34] Ethernut Web Site: <http://www.ethernut.de>
- [35] W. Simpson, "The Point-to-Point Protocol (PPP)", RFC1661, July 1994
- [36] Egnite Software GmbH Web Site: <http://www.egnite.de>
- [37] Information on Manchester Encoding, Available Online:
<http://www.erg.abdn.ac.uk/users/gorry/course/phy-pages/man.html>
- [38] D. R. Olsen Jr., S. Jefferies, T. Nielsen, W. Moyes, P. Fredrickson, "Cross Model Interaction using XWeb", in Proceedings UIST'00, ACM SIGGRAPH Symposium on User Interface Software and Technology, San Diego, CA, pp.191-200, 2000
- [39] M. Rettig, "Prototyping for Tiny Fingers", Communications of the ACM, 21-27, 1994
- [40] J. Postel, "Internet Control Message Protocol", RFC792, September 1981
- [41] Ethernut 1.3 Hardware Manual, Available Online:
<http://www.ethernut.de/pdf/enhwm13e.pdf>

[42] Ethernut 1.3 Software Manual, Available Online:

<http://www.ethernut.de/pdf/enswm20e.pdf>

[43] Nut/Net API, Online Available: http://www.ethernut.de/api/group__xgNutNet.html

[44] Atmel ATmega16 8-bit Microcontroller Datasheet, Available Online:

http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf

[45] B. Lloyd, W. Simpson, "PPP Authentication Protocols", RFC1334, October 1992



VITA

Mustafa Can Filibeli was born in Istanbul on January 7, 1980. He graduated from Ayazağa Işık Lisesi in 1998. He received his B.S. degree in Computer Engineering from Istanbul University in 2002. In September 2002, he joined the M.S. Program in Electrical and Computer Engineering at Koç University, as a research and teaching assistant.

