

Mission 5: Real-Time project on a "naked" computer

Tommaso Marinelli

Matteo Di Pirro

January 15, 2018

1 User manual

Figure 1 depicts the router configuration. Three components are required to use the board as a DHCP (Dynamic Host Configuration Protocol) relay:

- DHCP server, assigns IP addresses to clients;
- DHCP client, requests an IP address;
- DHCP relay, bridges between the client and the server.

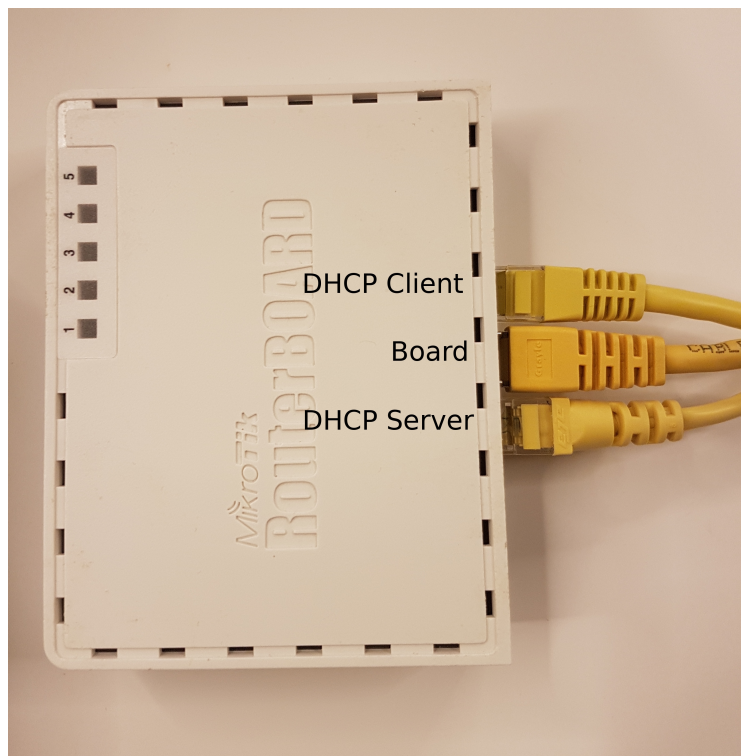


Figure 1: Router wiring

Server and clients are supposed to belong to two different networks (otherwise a relay would not be necessary, as they could communicate “directly”). The server must be connected to Port 1. The relay and the client(s) must be connected to ports from 2 to 5. A user can use these ports interchangeably. In Figure 1, relay and client are connected to Port 2 and 3, respectively.

You should set your Ethernet interface to require an IP address assigned dynamically (in other words, you should not specify a static IP address).

The board (henceforth called *relay*) displays a number of messages to let the client know what is going on. For instance, *Send to client* means it is dispatching a packet (coming from the server) to you. General purpose messages, such as the previous one, are displayed on the first line of the LCD screen. Assigned IP addresses, on the other hand, are displayed on the second line. Seeing such an address means that you are about to receive it.

When started up, the LCD screen reads *Waiting client* on the first line and the relay's IP address on the second. Such a configuration means you can connect your computer and wait for an address.

The right-most LED blinks to signal that the relay is up and running.

2 Documentation for system engineers

2.1 Compilation and download

The program comes with a `Makefile` that can be used to compile the program. To this end, the right command to use is `make`. It will generate some files in the current directory (`.`) and in `./Objects/`. In order to download the program into the naked computer, the user should follow the following steps (supposing the router is correctly configured):

- Run `tftp 192.168.97.60` in the same directory as `DHCPRelay.c`. The `tftp` environment will start;
- `binary`, to send the program as binary;
- `trace`, to see what happens;
- `verbose`, to see more information;
- `put DHCPRelay.hex`.

The last command has to be run only when the board is ready to receive the program. This happens during the three seconds after its reboot.

The board comes with a RouterBoard. It is configured with two different networks in order to test the relay. A LAN (Local Area Network) comprises Ports 2, 3, 4, and 5; a WAN (Wide Area Network) is set at Port 1. The DHCP server should be connected to the WAN, while both relay and clients should be on Ports from 2 to 5. The router built-in DHCP server should be disabled in order to let the relay work properly. Once disabled, it is not possible to communicate neither with the router itself nor with the board. Every device should be assigned a static IP address for communication purposes (meaning configuring the router and sending the program to the board). Examples are as follows:

- 192.168.97.15 to send to program to the board;
- 192.168.88.10 to configure the router.

2.2 DHCP server configuration

A DHCP server must be added to the network and configured to assign a free IP address within a certain range to the clients in the same subnet of the relay. On UNIX-based systems, the *Internet Systems Consortium DHCP Server* (`dhcpcd`) can be used as a daemon which provides this service; a sample configuration using the addresses specified in the project is shown in the following code box.

```
1 default-lease-time 600;
2 max-lease-time 7200;
3
4 subnet 192.168.10.0 netmask 255.255.255.0 {
5     # No IP address provided
6 }
```

```

7
8 subnet 192.168.97.0 netmask 255.255.255.0 {
9     option routers 192.168.97.1;
10    option subnet-mask 255.255.255.0;
11    option broadcast-address 192.168.97.255;
12
13    range 192.168.97.150 192.168.97.250;
14 }

```

Listing 1: Sample dhcpd.conf

2.3 Debug mode

If something is not working properly a special mode can be activated which allows to view information messages during critical phases of the program execution. The **UART interface** (Universal Asynchronous Receiver-Transmitter) is used to transmit the debug messages to an external terminal (typically a PC); the receiver must be connected to the RS232 port of the Olimex board and it must be set with these parameters:

- Baud rate: 9600
- Data bits: 8
- Stop bits: 1
- Parity: Odd

The debug mode is **disabled by default**; it can be activated adding the option `-DUART_DEBUG_ON` to the CFLAGS in the Makefile and recompiling the code. Some predefined macros are used to print single characters (`DEBUGCHAR`), blocks of characters of any length (`DEBUGBLOCK`) and strings (`DEBUGMSG`) throughout the code; these macros are only effective in debug mode.

Please note that this debug mode only works in a “blocking” mode, and may therefore result in additional delays while executing the software.

3 Documentation for programmers

3.1 Specification

The program implements a DHCP relay. Figure 2 depicts how it works.

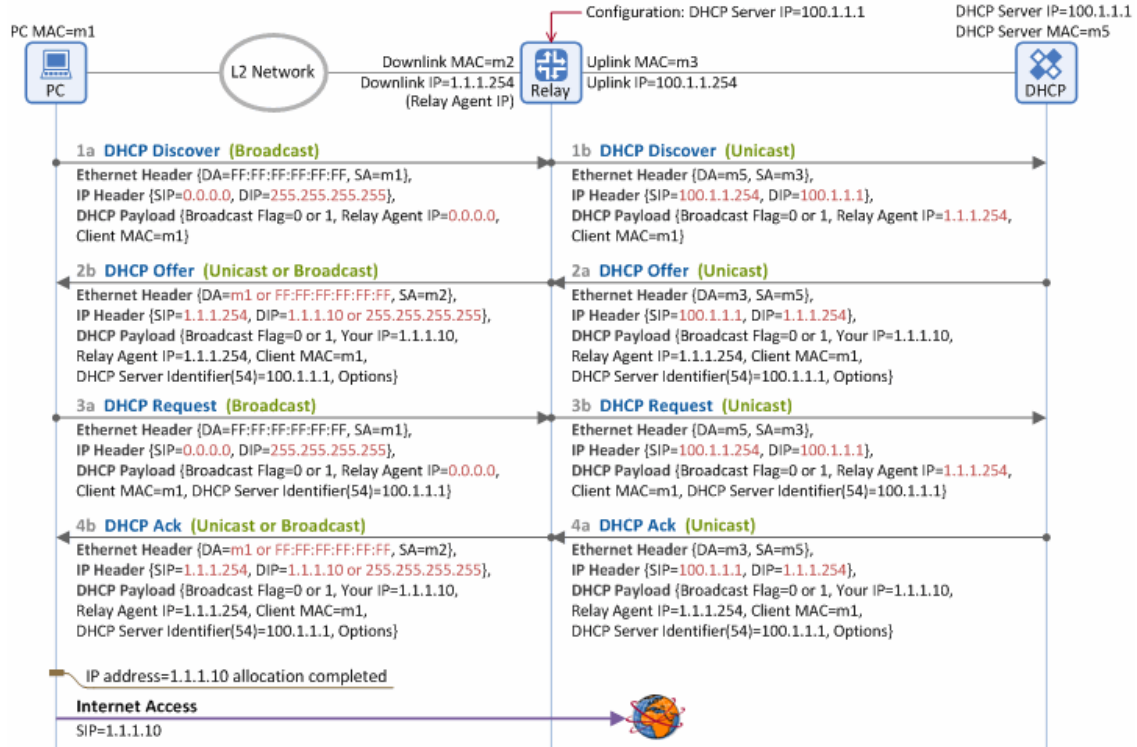


Figure 2: Example of DHCP protocol communication¹

Basically, a DHCP relay is a “man in the middle” between the client and the DHCP server. It helps the client to contact the server in order to obtain an IP address. It does so receiving the broadcast packets sent by clients and forwarding them in an unicast connection to one (or more) DHCP servers, and vice versa, forwarding the server responses in broadcast to the clients.

A relay slightly modify each packet. The actual changed fields depend on where the packet is coming from, as follows:

- **Server → Client:**
 - **Ethernet Payload**
 - * Destination MAC Address: DHCP Relay Uplink MAC → Broadcast
 - * Source MAC Address: DHCP Server MAC → DHCP Relay MAC Address
 - **IP Payload**
 - * Source IP Address: DHCP Server IP Address → DHCP Relay Downlink IP
 - * Destination IP Address: DHCP Relay Downlink IP → Broadcast
- **Client → Server:**
 - **Ethernet Payload**
 - * Destination MAC Address: Broadcast → DHCP Server MAC
 - * Source MAC Address: PC MAC Address → DHCP Relay Uplink MAC

¹<https://www.netmanias.com/en/?m=view&id=techdocs&no=6000>

– **IP Payload**

- * Source IP Address: 0.0.0.0 (no IP address) → DHCP Relay Uplink IP
- * Destination IP Address: Broadcast → DHCP Server IP

– **DHCP Payload**

- * Gateway IP Address (GIADDR): 0.0.0.0 → DHCP Relay Downlink IP

3.2 Design

Our ASG (**A**synchronous **S**tate **G**raphs) diagram is depicted in Figure 3.

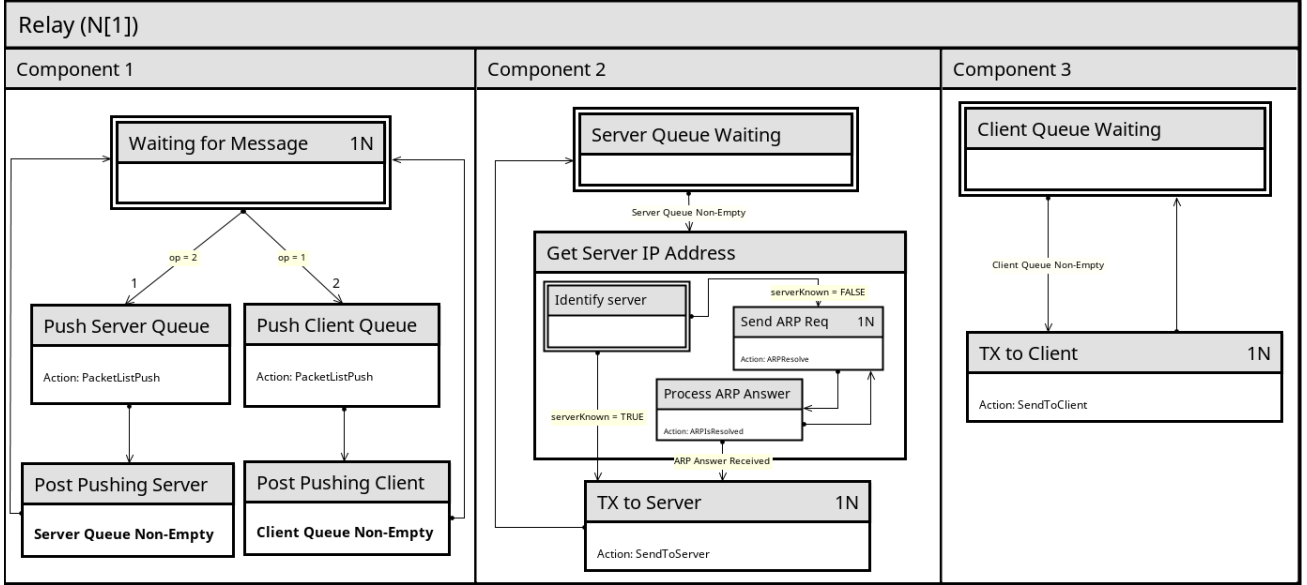


Figure 3: ASG diagram modeling a DHCP relay

The DHCP relay consists of three parallel components following a producer-consumer pattern. Component 1 waits for a packet, either from a client (broadcast) or a server (unicast). Components 2 and 3 then push the modified (note the modification is intended to be a consumer’s responsibility, but this is not mandatory) packet in a queue (two different queues are used, respectively for the server and clients). The corresponding consumer polls on its queue in order to detect a new packet. The polling has not to be confused with active waiting, since components are supposed to be parallel: if the queue is empty the corresponding producer simply releases the processor, allowing other components to be dispatched.

Component1’s WAITING FOR PACKETS state has got two outgoing transitions. If both of them may be crossed, the “server” one goes first. This means in our design the server has a higher priority than the clients. We consider the former to be more important than the latter, since it plays a significant role in the DHCP handshake. The protocol cannot proceed unless at least one server offers an IP address, and clients may retransmit their requests if no answer is received. We assign 2 as a priority to the server transition and 1 to the client one.

Component 2 and 3 are similar in their behavior. The only difference is the ARP request made by Component2 in order to get to know the server MAC address. This is necessary only the first time a client requires an IP address. Afterwards, a boolean flag, `serverKnown` in the diagram, becomes TRUE, meaning that no ARP requests are needed anymore. Even if this approach only works with one server, it can easily be further extended to work with as many servers as needed, just by using a list or an array.

A different approach might be issuing the ARP request(s) at initialization time. In this way, transmitting a packet to the server would not require an “on the fly” request. This is a reasonable

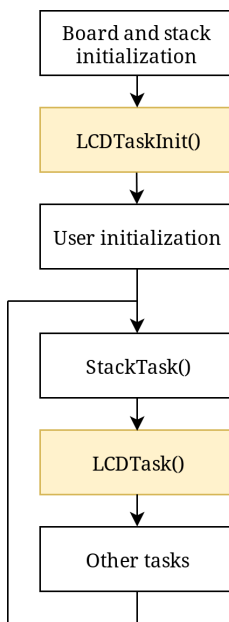
solution as soon as the DHCP server(s) is(/are) static. It is our belief that DHCP servers might change dynamically during the lifetime of the relay. Issuing ARP request during the initialization would require a new initialization if a DHCP server is dynamically added. Our solution, on the other hand, does not require any changes or restarts. Supposing to have a procedure to add a server, it would be enough for that procedure to set to `FALSE` the corresponding `serverKnown` flag for that server. A correct ARP request will be therefore issued for that server only.

One last point worthy to be explained is the resource `N` (going for *Network*). The board may only transmit in a half-duplex manner, and it is therefore necessary to allow only one substate at a time to use the network. This is the meaning of `N`. Each substate accessing the network must acquire `N` before being allowed to receive or transmit. Every time this happens, `N` is set to `TRUE` and the other substates cannot acquire it (thus it is a mutual exclusive resource). If a component C_1 needs the network, but the latter is being used by another component C_2 , C_1 has to wait until C_2 sets `N` to `FALSE` again. There is no risk of deadlock, because there cannot be a circular wait (we only use one resource). In a “usual” priority-based scheduling there could be the risk of starvation, meaning that C_1 never has the possibility to use the network because a higher priority component acquires it beforehand. Nonetheless, in our design and with our scheduler this cannot happen. For an informal proof of such a claim see 3.4.4.

3.3 LCD non-blocking module

The key concept in multitasking real-time systems is that tasks must be “small enough”, in order not to prevent the processor for executing other operations. In this project it is required to use the LCD display to print information messages, but the C functions provided by Microchip are not optimized for multitasking and take a non negligible time to be processed; the `LCDBlocking` module has been thus converted in `LCDNonBlocking` and integrated into the TCP/IP stack to provide a better display handling in an environment with different tasks.

The most time-demanding functions in the LCD library (`LCDInit`, `LCDErase`, `LCDUpdate`) have been **split in smaller states**. When called, the LCD task can only execute a part of a function: the following ones are executed at successive task calls; furthermore, each state can be accessed only if a sufficient amount of time has passed from the previous one, in line with what happened in the original file where some delays were placed between instructions.



The main functions of the new library are `LCDTaskInit` and `LCDTask`. The first one is an initialization function which is used to assign default values to control variables and configure the resources needed for the correct operation handling. The latter is the proper task, which runs inside the cooperative multitasking loop.

These functions are called in the main entity as a design choice, but they could have also been integrated into the `StackTsk` file of the TCP/IP stack.

The big difference between the new library and the old one is that the operations are not executed immediately but are “appended” somewhere, so that the task can pick and perform them in successive steps without losing information about other operation requests happening in the meanwhile.

In order to do this, a **circular list** (actually, a circular array of structures) has been implemented: each time an *Init*, *Erase* or *Update* operation is required, the list is filled with a code representing that operation and the text to write, if needed; this guarantees that the display always reflects the correct history of operations regardless of the state of the shadow copy.

The circular list static allocation required the availability of more than 256 bytes in memory, that is the maximum dimension allowed by default due to internal division of databanks in the PIC18. To overcome this limitation, the linker script was modified to create a single databank of twice the size and a `#pragma` directive was introduced to memorize the list in that exact memory location; the solution was fully tested and does not introduce any kind of issue.

The delay handling is entrusted to **Timer 1** (external 32.768 kHz oscillator) because it is not used by any other part of the system; the timer is in a bounded configuration and the initial value of the register is calculated according to the necessary minimum delay for each stage. If such a waiting condition is active, the LDC task checks for an overflow of the timer register before proceeding to the execution.

The instruction flow of the LCD task is represented in the next page; it is a simple diagram which is not meant to explain the detailed content of each block but the general behavior of the module. For in-depth analysis it is possible to read the code.

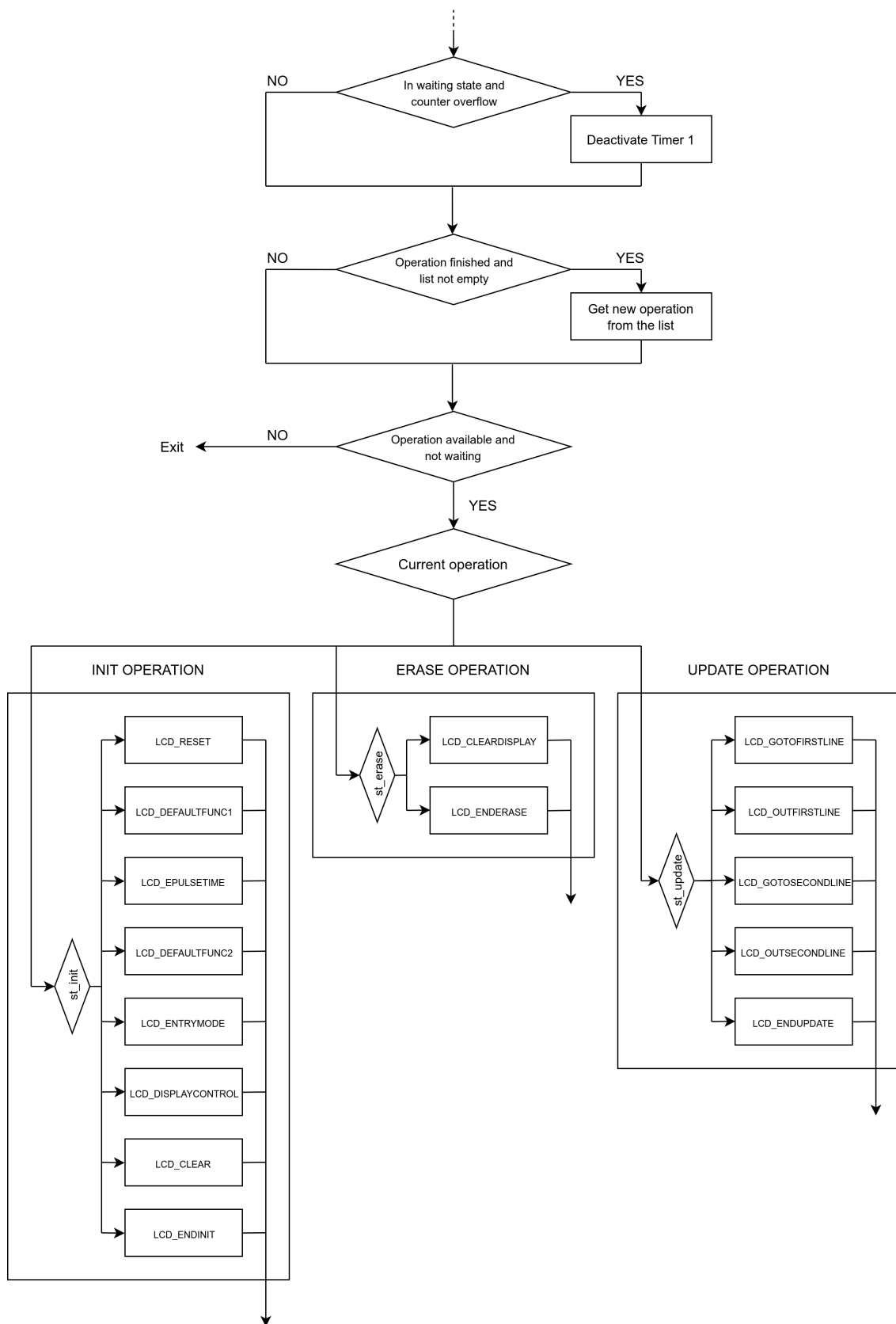


Figure 4: LCD task instruction flow chart

3.4 Relay

The main entry point for the relay is the file `DHCPRelay.c`, containing the implementation. Function signatures and enum definitions are in its header file, `DHCPRelay.h`. The definitions here contained implement the cooperative scheduling. Relay is refined in three main parallel components, representing, respectively, the wait for a message (both from clients and server) and the transmissions. Each enum type corresponds to a component in the ASG diagram. Thus, `Component1()`, `Component2()`, and `Component3()` define the proper actions to be taken when the corresponding component is dispatched. It is not the aim of this document to explain how an ASG diagram should be translated into code, but, in brief, the three function aforementioned should contain a switch checking for the current state and managing actions and transitions accordingly. The C file contains the actual implementations.

Our implementation makes use of two queue in order to store packets incoming both from the server and clients (`ClientMessages` and `ServerMessages`). We implement them using a circular list. Seen the limitations in allocating more than 256 bytes and seen the size of the information we store, we decided to set the queues size to 5 (the maximum allowed is 7). An implementation with queues makes the entire managing slightly more difficult. We not only store the DHCP header, but some information used to compose the forwarded message, too. This additional data is the `MessageType` (i.e. the content of the `MessageType` option), and the accepted IP address, if any. A boolean flag (`IPAddressNotNull`) is used to make the distinction. If `TRUE`, the field `RequiredAddress` is meaningful. Otherwise, its content is just random bytes.

With fixed-sized queues we may get in trouble when the two ratios (sending are receiving) are not balanced. We decided to manage them with the following policy: if a queue Q is full when a packet is to be pushed (meaning it has already been received) we discard the oldest packet in Q . We do not expect this policy to lead to packets loss, since clients usually retransmit their packets if they receive no answer. On the other hand, we do not expect the server queue to be overflowed frequently, since the relay interfaces with only one server.

We take advantage of `PacketCircularList`'s `PacketListIsEmpty` method to simplify the translation from ASG to C. We remove `POST PUSHING SERVER` and `POST PUSHING CLIENT` in favor of a direct check on the queue size. This simplifies our program's structure and reduces the overhead (even if small) caused by the "canonical" translation of those two states. Hence, the rendez-vous `Server Queue Non-Empty` and `Client Queue Non-Empty` are to be translated, respectively, with `PacketListIsEmpty(&ServerMessages)` and `PacketListIsEmpty(&ClientMessages)`.

Please note that the Mikrotik TCP IP stack allows us to "directly" modify only the DHCP header. The other modifications the relay should do occur at transmission time.

3.4.1 Waiting component

The waiting component is basically a polling on the two open sockets (in the basic implementation we assume there is only one server and client). If enough bytes are written (and ready to be read) in the socket we start the reading procedure, actually implemented by `GetPacket()`. This function takes two parameters: a pointer to the variable to store the read packet in and the socket to read from. It basically checks if something is waiting in the socket buffer, and, if so, reads the packet (performing a basic check on the hardware type and length). It then reads the options, taking into account only the `MessageType` and the `RequestedIPAddress`. If everything works, it returns 0. Otherwise, an error code is given, as follows:

- -1 if no packet is available on the selected socket, meaning there are less than 241 bytes in its buffer;
- -2, wrong hardware type
- -3, wrong hardware length

- -4, parameters are invalid

The waiting state checks the client and the server socket. It might happen that both of them contain packets ready to be read. In such a scenario, the server as a higher priority and is served first. A flag, `prevFromClient`, is set to `TRUE` and checked when the component comes again in the waiting state. If it is `TRUE`, a client is served; otherwise, both the sockets are checked again. This rules client starvation out.

The final step is pushing the packet into the corresponding queue.

3.4.2 Transmission to the server

Transmitting a packet to the DHCP server has two prerequisites: the queue must be not empty and the server MAC address should be known. The former condition becomes `TRUE` if and only if a packet has not only been received, but pushed into the queue, too. Once a packet is ready, if the server MAC address is not known (meaning `serverKnown == FALSE`) an ARP request is issued to get to know the address. When a response is received, `serverKnown` becomes `TRUE` and no more ARP requests will be issued. Afterwards, the packet is sent to the server via the function `SendToServer()`.

`SendToServer` first checks if the server socket contains enough free space. Socket's remote IP address is set to the server one, and so is the MAC. It pops a packet from its queue and modifies only the `GIADDR` field, using its own IP address. It sets to 0 every unused field and the magic cookie to the value reported in RFC 1533. The other fields are either taken from the popped packet or generated on the fly. An example of the former is every data contained in the DHCP header, as well as the accepted client IP address, if any. An example of the latter is the subnet mask. The minimum size of such a packet is 300 bytes, in order to ensure compatibility with old DHCP relays that discard packets smaller than 300 octets.

3.4.3 Transmission to the client

Transmitting a packet to a client is simpler than transmitting to a server. It is indeed not necessary to know any MAC addresses because the packet is sent in broadcast to the client network. Thus, the only prerequisite for this action is a non empty queue. When this is the case, `SendToClient()` is invoked.

This function operates more or less as `SendToServer` does, with a few slight differences. The first is the socket remote IP address, which is here set to broadcast (remember the client has not got an IP address unless the very end of the DHCP "handshake"). The MAC address is correctly set to be the one of the client. The latter value is taken from `CHADDR`. The second is the magic cookie, set to `0x63538263`.

3.4.4 Starvation

Suppose `Component1` has acquired the network in order to receive a packet, and suppose `Component 2` and `3` are waiting for the network in their "idle" state. In such a scenario, their queues are not empty. `Component1` receives the packet, since it is allowed to use both the network and the processor. In this sense, the receiving operation may be considered as atomic: no one can do anything else since `GetPacket` never releases the processor and the scheduler is a non-preemptive one. It then executes to completion, and releases both the network (meaning it sets `N` to `FALSE`) and the processor (i.e. it lets another component execute). At this point, `Component2` has the possibility to acquire the network, and it does so. This means neither `Component1` nor `Component3` may go ahead with their network-based operations (note that `Component1` is now one of its "push" substates, which do not require the network). `Component2` also does not release neither the network nor the processor during the transmission. It afterwards releases both of them, and `Component3` has its chance to proceed.

If either `Component2` or `Component3` is not ready to transmit (its queue is empty), it does not compete for `N`.

This reasoning also applies considering Component2 or Component3 as a starting point. The scheduler always schedule these components in a “circular” manner (indeed, it is a round robin one), and there is no chance that a component never executes.

3.4.5 Enabling the relay mode

In order to enable the DHCP Relay mode, one should configure the router as said and do small modifications to the definitions provided in the Mikrotik TCP IP stack. In particular, `STACK_USE_DHCP_CLIENT` and `STACK_USE_DHCP_SERVER` should be disabled, while `STACK_USE_DHCP_RELAY` should be enabled. These definitions can be found in `TCP_IP.h`.

4 Program listings

This section contains the code of the most relevant files produced for this project. Rest of the code can be read in the `src` folder.

4.1 LCD non-blocking

This section lists the non-blocking LCD code. A blocking, functionally-equivalent version was provided by Mikrotik.

```
1 #define __LCDNONBLOCKING_C
2
3 #include "../Include/TCPIP_Stack/Delay.h"
4
5 #define __18F97J60
6 #define __SDCC__
7 #include "../Include/HardwareProfile.h"
8 #include "../Include/TCPIP_Stack/TCPIP.h" //ML
9
10 #ifdef USE_LCD
11
12 // #define FOUR_BIT_MODE
13 #define SAMSUNG_S6A0032 // This LCD driver chip has a different means of entering 4-
    bit mode.
14 #define ROWCHARS 16
15
16 // LCDText is a 32 byte shadow of the LCD text. Write to it and
17 // then call LCDUpdate() to copy the string into the LCD module.
18 BYTE LCDText[ROWCHARS*2+1];
19
20 static BYTE LCDi, LCDj;
21
22 // States of the initialization function
23 static enum LCDInit_states
24 {
25     LCD_RESET,
26     LCD_DEFAULTFUNC1,
27     LCD_EPULSETIME,
28     LCD_DEFAULTFUNC2,
29     LCD_ENTRYMODE,
30     LCD_DISPLAYCONTROL,
31     LCD_CLEAR,
32     LCD_ENDINIT
33 } st_init;
34
35 // States of the erase function
36 static enum LCDErase_states
37 {
38     LCD_CLEARDISPLAY,
39     LCD_CLEARLOCAL
40 } st_update;
41
42 // States of the update function
43 static enum LCDUpdate_states
44 {
45     LCD_GOTOFIRSTLINE,
46     LCD_OUTFIRSTLINE,
47     LCD_GOTOSECONDLIN,
48     LCD_OUTSECONDLIN,
49     LCD_ENDUPDATE
50 } st_erase;
51
52
53 // Control flags
54 static char LCDWaiting;
55 static char LCDOpInProgress;
56 // Current operation info
57 static char LCDCurrentOrder;
```

```

58 static char LCDCurrentText[ROWCHARS*2+1];
59
60 static char tmpBuf[33];
61
62 /*****
63 * Function:          static void LCDWrite(BYTE RS, BYTE Data)
64 *
65 * Input:             RS - Register Select - 1:RAM, 0:Config registers
66 *                   Data - 8 bits of data to write
67 *
68 * Output:            None
69 *
70 * Overview:          Controls the Port I/O pins to cause an LCD write
71 * *****/
72 static void LCDWrite(BYTE RS, BYTE Data)
73 {
74     #if defined(LCD_DATA_TRIS)
75         LCD_DATA_TRIS = 0x00;
76     #else
77         LCD_DATA0_TRIS = 0;
78         LCD_DATA1_TRIS = 0;
79         LCD_DATA2_TRIS = 0;
80         LCD_DATA3_TRIS = 0;
81         #if !defined(FOUR_BIT_MODE)
82             LCD_DATA4_TRIS = 0;
83             LCD_DATA5_TRIS = 0;
84             LCD_DATA6_TRIS = 0;
85             LCD_DATA7_TRIS = 0;
86         #endif
87     #endif
88     LCD_RS_TRIS = 0;
89     LCD_RD_WR_TRIS = 0;
90     LCD_RD_WR_IO = 0;
91     LCD_RS_IO = RS;
92
93     #if defined(FOUR_BIT_MODE)
94         #if defined(LCD_DATA_IO)
95             LCD_DATA_IO = Data>>4;
96         #else
97             LCD_DATA0_IO = Data & 0x10;
98             LCD_DATA1_IO = Data & 0x20;
99             LCD_DATA2_IO = Data & 0x40;
100            LCD_DATA3_IO = Data & 0x80;
101        #endif
102        Nop();           // Wait Data setup time (min 40ns)
103        Nop();
104        LCD_E_IO = 1;
105        Nop();           // Wait E Pulse width time (min 230ns)
106        Nop();
107        Nop();
108        Nop();
109        Nop();
110        Nop();
111        Nop();
112        Nop();
113        Nop();
114        LCD_E_IO = 0;
115    #endif
116
117    #if defined(LCD_DATA_IO)
118        LCD_DATA_IO = Data;
119    #else
120        LCD_DATA0_IO = ((Data & 0x01) == 0x01);
121        LCD_DATA1_IO = ((Data & 0x02) == 0x02);
122        LCD_DATA2_IO = ((Data & 0x04) == 0x04);
123        LCD_DATA3_IO = ((Data & 0x08) == 0x08);
124        #if !defined(FOUR_BIT_MODE)
125            LCD_DATA4_IO = ((Data & 0x10) == 0x10);
126            LCD_DATA5_IO = ((Data & 0x20) == 0x20);
127            LCD_DATA6_IO = ((Data & 0x40) == 0x40);

```

```

128     LCD_DATA7_IO = ((Data & 0x80) == 0x80);
129     #endif
130 #endif
131     Nop();           // Wait Data setup time (min 40ns)
132     Nop();
133     LCD_E_IO = 1;
134     Nop();           // Wait E Pulse width time (min 230ns)
135     Nop();
136     Nop();
137     Nop();
138     Nop();
139     Nop();
140     Nop();
141     Nop();
142     Nop();
143     LCD_E_IO = 0;
144
145     // // Uncomment if you want the data bus to go High-Z when idle
146     // // Note that this may make analog functions work poorly when using
147     // // Explorer 16 revision 5 boards with a 5V LCD on it. The 5V LCDs have
148     // // internal weak pull ups to 5V on each of the I/O pins, which will
149     // // backfeed 5V weekly onto non-5V tolerant PIC I/O pins.
150     // #if defined(LCD_DATA_TRIS)
151     //     LCD_DATA_TRIS = 0xFF;
152     // #else
153     //     LCD_DATA0_TRIS = 1;
154     //     LCD_DATA1_TRIS = 1;
155     //     LCD_DATA2_TRIS = 1;
156     //     LCD_DATA3_TRIS = 1;
157     //     #if !defined(FOUR_BIT_MODE)
158     //         LCD_DATA4_TRIS = 1;
159     //         LCD_DATA5_TRIS = 1;
160     //         LCD_DATA6_TRIS = 1;
161     //         LCD_DATA7_TRIS = 1;
162     //     #endif
163     // #endif
164     // LCD_RS_TRIS = 1;
165     // LCD_RD_WR_TRIS = 1;
166 }
167
168 /*****
169  * Function:          static void LCDInitExec()
170  *
171  * Input:              None
172  *
173  * Output:             None
174  *
175  * Overview:          LCD INIT operation execution function
176  * *****/
177 static void LCDInitExec(void)
178 {
179     switch (st_init)
180     {
181     case LCD_RESET:
182         // ----- Part moved to LCDTaskInit -----
183         //memset(LCDText, ' ', sizeof(LCDText)-1);
184         //LCDText[sizeof(LCDText)-1] = 0;
185         // Setup the I/O pins
186         LCD_E_IO = 0;
187         LCD_RD_WR_IO = 0;
188         #if defined(LCD_DATA_TRIS)
189             LCD_DATA_TRIS = 0x00;
190         #else
191             LCD_DATA0_TRIS = 0;
192             LCD_DATA1_TRIS = 0;
193             LCD_DATA2_TRIS = 0;
194             LCD_DATA3_TRIS = 0;
195             #if !defined(FOUR_BIT_MODE)
196                 LCD_DATA4_TRIS = 0;
197                 LCD_DATA5_TRIS = 0;

```

```

198     LCD_DATA6_TRIS = 0;
199     LCD_DATA7_TRIS = 0;
200     #endif
201 #endif
202 LCD_RD_WR_TRIS = 0;
203 LCD_RS_TRIS = 0;
204 LCD_E_TRIS = 0;
205 // Waiting time: 40 ms
206 LCDWaiting = 1;
207 TMR1L = (0x10000 - 1311) & 0xff;
208 TMR1H = (0x10000 - 1311) >> 8;
209 T1CONbits.TMR1ON = 1;
210 // Next state
211 st_init = LCD_DEFAULTFUNC1;
212 break;
213
214 case LCD_DEFAULTFUNC1:
215     // Set the default function
216     // Go to 8-bit mode first to reset the instruction state machine
217     // This is done in a loop 3 times to absolutely ensure that we get
218     // to 8-bit mode in case if the device was previously booted into
219     // 4-bit mode and our PIC got reset in the middle of the LCD
220     // receiving half (4-bits) of an 8-bit instruction
221     LCD_RS_IO = 0;
222     #if defined(LCD_DATA_IO)
223         LCD_DATA_IO = 0x03;
224     #else
225         LCD_DATA0_IO = 1;
226         LCD_DATA1_IO = 1;
227         LCD_DATA2_IO = 0;
228         LCD_DATA3_IO = 0;
229         #if !defined(FOUR_BIT_MODE)
230             LCD_DATA4_IO = 0;
231             LCD_DATA5_IO = 0;
232             LCD_DATA6_IO = 0;
233             LCD_DATA7_IO = 0;
234         #endif
235     #endif
236     //Nop();           // Wait Data setup time (min 40ns)
237     //Nop();
238     // Next state
239     st_init = LCD_EPULSETIME;
240     LCDi = 0;
241 break;
242
243 case LCD_EPULSETIME:
244     LCD_E_IO = 1;
245     Delay10us(1);     // Wait E Pulse width time (min 230ns)
246     LCD_E_IO = 0;
247     // Cyclic access
248     LCDi++;
249     if (LCDi >= 3u)
250         st_init = LCD_DEFAULTFUNC2;
251     // Waiting time: 2 ms
252     LCDWaiting = 1;
253     TMR1L = (0x10000 - 66) & 0xff;
254     TMR1H = (0x10000 - 66) >> 8;
255     T1CONbits.TMR1ON = 1;
256 break;
257
258 case LCD_DEFAULTFUNC2:
259     #if defined(FOUR_BIT_MODE)
260         #if defined(SAMSUNG_S6A0032)
261             // Enter 4-bit mode (requires only 4-bits on the S6A0032)
262             #if defined(LCD_DATA_IO)
263                 LCD_DATA_IO = 0x02;
264             #else
265                 LCD_DATA0_IO = 0;
266                 LCD_DATA1_IO = 1;
267                 LCD_DATA2_IO = 0;

```

```

268         LCD_DATA3_IO = 0;
269     #endif
270     Nop();           // Wait Data setup time (min 40ns)
271     Nop();
272     LCD_E_IO = 1;
273     Delay10us(1);    // Wait E Pulse width time (min 230ns)
274     LCD_E_IO = 0;
275     #else
276         // Enter 4-bit mode with two lines (requires 8-bits on most LCD controllers)
277         LCDWrite(0, 0x28);
278     #endif
279     #else
280         // Use 8-bit mode with two lines
281         LCDWrite(0, 0x38);
282     #endif
283     // Waiting time: 61 us
284     LCDWaiting = 1;
285     TMR1L      = (0x10000 - 2) & 0xff;
286     TMR1H      = (0x10000 - 2) >> 8;
287     T1CONbits.TMR1ON = 1;
288     // Next state
289     st_init = LCD_ENTRYMODE;
290     break;
291
292     case LCD_ENTRYMODE:
293         LCDWrite(0, 0x06); // Increment after each write, do not shift
294         // Waiting time: 61 us
295         LCDWaiting = 1;
296         TMR1L      = (0x10000 - 2) & 0xff;
297         TMR1H      = (0x10000 - 2) >> 8;
298         T1CONbits.TMR1ON = 1;
299         // Next state
300         st_init = LCD_DISPLAYCONTROL;
301         break;
302
303     case LCD_DISPLAYCONTROL:
304         LCDWrite(0, 0x0C); // Turn display on, no cursor, no cursor blink
305         // Waiting time: 61 us
306         LCDWaiting = 1;
307         TMR1L      = (0x10000 - 2) & 0xff;
308         TMR1H      = (0x10000 - 2) >> 8;
309         T1CONbits.TMR1ON = 1;
310         // Next state
311         st_init = LCD_CLEAR;
312         break;
313
314     case LCD_CLEAR:
315         LCDWrite(0, 0x01);
316         // Waiting time: 2 ms
317         LCDWaiting = 1;
318         TMR1L      = (0x10000 - 66) & 0xff;
319         TMR1H      = (0x10000 - 66) >> 8;
320         T1CONbits.TMR1ON = 1;
321         // Next state
322         st_init = LCD_ENDINIT;
323         break;
324
325     case LCD_ENDINIT:
326         // End of operation
327         LCDOpInProgress = 0;
328         // Next state
329         st_init = LCD_RESET;
330         break;
331
332     default:
333         // Do nothing
334         break;
335 }
336 }
337

```



```

338 /*****
339 * Function:          static void LCDEraseExec()
340 *
341 * Input:             None
342 *
343 * Output:            None
344 *
345 * Overview:          LCD ERASE operation execution function
346 * *****/
347 static void LCDEraseExec(void)
348 {
349     switch(st_erase)
350     {
351         case LCD_CLEARDISPLAY:
352             // Clear display
353             LCDWrite(0, 0x01);
354             // Waiting time: 2 ms
355             LCDWaiting = 1;
356             TMR1L      = (0x10000 - 66) & 0xff;
357             TMR1H      = (0x10000 - 66) >> 8;
358             T1CONbits.TMR1ON = 1;
359             // Next state
360             st_erase = LCD_CLEARLOCAL;
361             break;
362
363         case LCD_CLEARLOCAL:
364             // ----- Do not execute: done at invoke time -----
365             // Clear local copy
366             // memset(LCDText, ' ', 32);
367             // End of operation
368             LCDOpInProgress = 0;
369             // Next state
370             st_erase = LCD_CLEARDISPLAY;
371             break;
372
373         default:
374             // Do nothing
375             break;
376     }
377 }
378
379 /*****
380 * Function:          static void LCDUpdateExec()
381 *
382 * Input:             None
383 *
384 * Output:            None
385 *
386 * Overview:          LCD UPDATE operation execution function
387 * *****/
388 static void LCDUpdateExec(void)
389 {
390     switch(st_update)
391     {
392         case LCD_GOTOFIRSTLINE:
393             DEBUGMSG("-> SENT TEXT: ");
394             DEBUGBLOCK(LCDCurrentText, 33, 1);
395             DEBUGMSG("\r\n");
396             // Go home
397             LCDWrite(0, 0x02);
398             // Waiting time: 2 ms
399             LCDWaiting = 1;
400             TMR1L      = (0x10000 - 66) & 0xff;
401             TMR1H      = (0x10000 - 66) >> 8;
402             T1CONbits.TMR1ON = 1;
403             st_update = LCD_OUTFIRSTLINE;
404             LCDi = 0;
405             break;
406
407         case LCD_OUTFIRSTLINE:

```

```

408 // Erase the rest of the line if a null char is
409 // encountered (good for printing strings directly)
410 if(LCDCCurrentText[LCDi] == 0u)
411 {
412     LCDWrite(1, ' ');
413     for(LCDj=LCDi; LCDj < 16u; LCDj++)
414     {
415         LCDCurrentText[LCDj] = ' ';
416     }
417 } else
418     LCDWrite(1, LCDCurrentText[LCDi]);
419 // Cyclic access
420 LCDi++;
421 if (LCDi >= 16u)
422     st_update = LCD_GOTOSECONDLIN;
423 // Waiting time: 61 us
424 LCDWaiting = 1;
425 TMR1L = (0x10000 - 2) & 0xff;
426 TMR1H = (0x10000 - 2) >> 8;
427 T1CONbits.TMR1ON = 1;
428 break;
429
430 case LCD_GOTOSECONDLIN:
431 // Set the address to the second line
432 LCDWrite(0, 0xC0);
433 // Waiting time: 61 us
434 LCDWaiting = 1;
435 TMR1L = (0x10000 - 2) & 0xff;
436 TMR1H = (0x10000 - 2) >> 8;
437 T1CONbits.TMR1ON = 1;
438 st_update = LCD_OUTSECONDLIN;
439 LCDi = 16;
440 break;
441
442 case LCD_OUTSECONDLIN:
443 // Erase the rest of the line if a null char is
444 // encountered (good for printing strings directly)
445 if(LCDCCurrentText[LCDi] == 0u)
446 {
447     LCDWrite(1, ' ');
448     for(LCDj=LCDi; LCDj < 32u; LCDj++)
449     {
450         LCDCurrentText[LCDj] = ' ';
451     }
452 } else
453     LCDWrite(1, LCDCurrentText[LCDi]);
454 // Cyclic access
455 LCDi++;
456 if (LCDi >= 32u)
457     st_update = LCD_ENDUPDATE;
458 // Waiting time: 61 us
459 LCDWaiting = 1;
460 TMR1L = (0x10000 - 2) & 0xff;
461 TMR1H = (0x10000 - 2) >> 8;
462 T1CONbits.TMR1ON = 1;
463 break;
464
465 case LCD_ENDUPDATE:
466 DEBUGMSG("-> PARSED TEXT: ");
467 DEBUGBLOCK(LCDCCurrentText, 33, 1);
468 DEBUGMSG("\r\n");
469 LCDOpInProgress = 0;
470 st_update = LCD_GOTOFIRSTLIN;
471 break;
472
473 default:
474 // Do nothing
475 break;
476 }
477 }

```

```

478
479
480 /*****
481 * Function:          void LCDTaskInit(void)
482 *
483 * Input:             None
484 *
485 * Output:            None
486 *
487 * Overview:          Initialization of the entities used in the LCD task
488 *****/
489 void LCDTaskInit(void)
490 {
491     // Initial states
492     st_init   = LCD_RESET;
493     st_update = LCD_GOTOFIRSTLINE;
494     st_erase  = LCD_CLEARDISPLAY;
495     // Initial values
496     LCDWaiting = 0;
497     LCDOpInProgress = 0;
498     LCDi = 0;
499     LCDj = 0;
500     // Circular list initialization
501     LCDListInit();
502     // Timer1 configuration
503     T1CONbits.TMR1ON = 0;    // disable timer1
504     T1CONbits.RD16   = 1;    // use timer1 16-bit counter
505     T1CONbits.T1CKPS0 = 0;    // prescaler set to 1:1
506     T1CONbits.T1CKPS1 = 0;
507     T1CONbits.T1OSCEN = 1;    // timer1 oscillator enable
508     T1CONbits.TMR1CS  = 1;    // external clock selected
509     PIR1bits.TMR1IF   = 0;    // clear timer1 overflow bit
510     // Clear LCDText
511     memset(LCDText, ' ', sizeof(LCDText)-1);
512     LCDText[sizeof(LCDText)-1] = 0;
513 }
514
515
516 void LCDTask(void)
517 {
518     if (PIR1bits.TMR1IF && LCDWaiting)    // Time expired
519     {
520         LCDWaiting = 0;
521         T1CONbits.TMR1ON = 0;
522         PIR1bits.TMR1IF = 0;
523     }
524     if (!LCDOpInProgress)    // No operations being executed
525     {
526         if (!LCDListIsEmpty())    // The list contains at least one operation
527         {
528             LCDListPop(&LCDCurrentOrder, LCDCurrentText);    // Retrieve the operation to
execute
529             LCDOpInProgress = 1;    // Set the execution flag
530             DEBUGMSG("POPPED: ");
531             ultoa(LCDCurrentOrder, tmpBuf, 10);
532             DEBUGMSG(tmpBuf);
533             DEBUGMSG("\r\n");
534         }
535     }
536     if (!LCDWaiting && LCDOpInProgress)    // Not waiting for timers
537     {
538         switch (LCDCurrentOrder)
539         {
540             case 1:    // Init operation
541                 LCDInitExec();
542                 break;
543             case 2:    // Erase operation
544                 LCDEraseExec();
545                 break;

```

```

547
548     case 3:                // Update operation
549         LCDUpdateExec();
550         break;
551
552     default:
553         // Do nothing
554         break;
555 }
556 }
557 }
558
559
560 /*****
561  * Function:          void LCDInit(void)
562  *
563  * Input:             None
564  *
565  * Output:            None
566  *
567  * Overview:          Add an INIT operation to the LCD circular list
568  *****/
569 void LCDInit(void)
570 {
571     LCDListPush(1, "");
572     DEBUGMSG("PUSHED: 1\r\n");
573 }
574
575 /*****
576  * Function:          void LCDErase(void)
577  *
578  * PreCondition:      LCDInit() must have been called once
579  *
580  * Input:             None
581  *
582  * Output:            None
583  *
584  * Overview:          Add an ERASE operation to the LCD circular list
585  *                    and clean the LCDText shadow copy
586  *****/
587 void LCDErase(void)
588 {
589     LCDListPush(2, "");
590     DEBUGMSG("PUSHED: 2\r\n");
591     memset(LCDText, ' ', 32);
592 }
593
594 /*****
595  * Function:          void LCDUpdate(void)
596  *
597  * PreCondition:      LCDInit() must have been called once
598  *
599  * Input:             None
600  *
601  * Output:            None
602  *
603  * Overview:          Add an UPDATE operation to the LCD circular list
604  *****/
605 void LCDUpdate(void)
606 {
607     LCDListPush(3, LCDText);
608     DEBUGMSG("PUSHED: 3\r\n");
609 }
610
611 #endif // #ifdef USE_LCD

```

Listing 2: LCDNonBlocking.c

4.2 DHCP Relay

This section lists the code implementing the relay functionality.

4.2.1 Header

Listing 3 lists the relay header file.

```
1  /*****
2  * FileName:      DHCPRelay.h
3  * Dependencies:  Compiler.h
4  * Processor:     PIC18, PIC24F, PIC24H, dsPIC30F, dsPIC33F, PIC32
5  * Compiler:      Microchip C32 v1.05 or higher
6  *               Microchip C30 v3.12 or higher
7  *               Microchip C18 v3.30 or higher
8  *               HI-TECH PICC-18 PRO 9.63PL2 or higher
9  *****/
10 #ifndef _DHCPRELAY_H
11 #define _DHCPRELAY_H
12 #define STACK_USE_DHCP_RELAY
13 #include "GenericTypeDefs.h"
14 #include "TCPIP_Stack/TCPIP.h"
15
16 // #define BAUD_RATE      (19200)    // bps
17
18 #if !defined(THIS_IS_STACK_APPLICATION)
19     extern BYTE AN0String[8];
20 #endif
21
22 // MLvoid DoUARTConfig(void);
23
24 // ML#if defined(EEPROM_CS_TRIS) || defined(SPIFLASH_CS_TRIS)
25 // ML void SaveAppConfig(void);
26 // ML#else
27     #define SaveAppConfig()
28 // ML#endif
29
30 // MLvoid SMTPDemo(void);
31 void PingDemo(void);
32 // MLvoid SNMPTrapDemo(void);
33 // MLvoid GenericTCPClient(void);
34 // MLvoid GenericTCPServer(void);
35 // void BerkeleyTCPClientDemo(void);
36 // void BerkeleyTCPServerDemo(void);
37 // void BerkeleyUDPClientDemo(void);
38
39 #ifdef STACK_USE_DHCP_RELAY
40     // enum representing the current relay component on the processor
41     typedef enum {
42         INIT,        // init the DHCP relay parameters
43         COMP1,       // listening for packets
44         COMP2,       // sending to server
45         COMP3        // sending to client
46     } CURRENT_COMPONENT;
47
48     typedef enum {
49         WAITING_FOR_MESSAGE,    // Polling for packets
50         /*SERVER_MESSAGE_T,
51         CLIENT_MESSAGE_T,
52         FROM_SERVER,
53         FROM_SERVER_T,*/
54         PUSH_SERVER_QUEUE,      // push in the server queue
55         PUSH_SERVER_QUEUE_T,
56         /*FROM_CLIENT,
57         FROM_CLIENT_T,*/
58         PUSH_CLIENT_QUEUE,      // push in the client queue
59         PUSH_CLIENT_QUEUE_T,
60     } COMPONENT1;
61
```

```

62     typedef enum {
63         SERVER_QUEUE_WAITING,    // wait for a packet to be sent
64         SERVER_QUEUE_WAITING_T,
65         GET_SERVER_IP_ADDRESS,   // issue an ARP request
66         GET_SERVER_IP_ADDRESS_T,
67         IDENTIFY_SERVER_TO_TX,
68         TX_TO_SERVER,            // actually transmit the packet
69         TX_TO_SERVER_T
70     } COMPONENT2;
71
72     typedef enum {
73         CLIENT_QUEUE_WAITING,    //wait for a packet to be sent
74         CLIENT_QUEUE_WAITING_T,
75         TX_TO_CLIENT,            // transmit the packet
76         TX_TO_CLIENT_T
77     } COMPONENT3;
78
79     typedef enum {
80         IDENTIFY_SERVER,          // check if a DHCP server is known
81         IDENTIFY_SERVER_TO_ARP,
82         SEND_ARP_REQUEST,        // issue the ARP request
83         SEND_ARP_REQUEST_T,
84         PROCESS_ARP_ANSWER,      // get the answer or reissue the request
85         PROCESS_ARP_ANSWER_T
86     } GET_SERVER_IP_ADDRESS_COMP;
87
88     static int DHCPRelayInit();
89     static void DHCPRelayTask();
90     static int GetServerPacket();
91     static int GetClientPacket();
92     static void SendToServer();
93     static void SendToClient();
94     static void Component1();
95     static void Component2();
96     static void Component3();
97 #endif
98
99 // An actual function defined in DHCPRelay.c for displaying the current IP
100 // address on the LCD.
101 #if defined(__SDCC__)
102     void DisplayIPValue(DWORD IPVal);
103     void DisplayString(BYTE pos, char* text);
104     void DisplayWORD(BYTE pos, WORD w);
105 #else
106     void DisplayIPValue(IP_ADDR IPVal);
107 #endif
108
109 #endif // _DHCPRELAY_H

```

Listing 3: DHCPRelay.h

4.2.2 Implementation

Listing 4 lists the actual relay implementation.

```
1  /*****
2  *
3  *   Main Application Entry Point for the DHCPRelay.
4  *
5  *****/
6
7  /*
8  *   This symbol uniquely defines this file as the main entry point.
9  *   There should only be one such definition in the entire project,
10 *   and this file must define the AppConfig variable as described below.
11 *   The processor configuration will be included in HardwareProfile.h
12 *   if this symbol is defined.
13 */
14 #define THIS_INCLUDES_THE_MAIN_FUNCTION
15 #define THIS_IS_STACK_APPLICATION
16
17 // define the processor we use
18 #define __18F97J60
19 // define the compiler we use
20 #define __SDCC__
21
22 // include all hardware and compiler dependent definitions
23 #include "Include/HardwareProfile.h"
24 // Include all headers for any enabled TCPIP Stack functions
25 #include "Include/TCPIP_Stack/TCPIP.h"
26
27 // Include functions specific to this stack application
28 #include "Include/DHCPRelay.h"
29
30 #if !defined(STACK_CLIENT_MODE)
31     #define STACK_CLIENT_MODE
32 #endif
33
34 #define BROADCAST                0xFFFFFFFF // broadcast address
35 // server's IP address
36 #define SERVER_IP_ADDR_BYTE1    (192ul)
37 #define SERVER_IP_ADDR_BYTE2    (168ul)
38 #define SERVER_IP_ADDR_BYTE3    (10ul)
39 #define SERVER_IP_ADDR_BYTE4    (1ul)
40
41 // Declare AppConfig structure and some other supporting stack variables
42 APP_CONFIG AppConfig;
43 BYTE AN0String[8];
44
45 // sockets
46 UDP_SOCKET serverToClient;
47 UDP_SOCKET clientToServer;
48
49 // components needed for the cooperative scheduling
50 CURRENT_COMPONENT currentComponent;
51 COMPONENT1 comp1;
52 COMPONENT2 comp2;
53 COMPONENT3 comp3;
54 GET_SERVER_IP_ADDRESS_COMP comp2_2;
55
56 // queues to store packets coming from server and clients
57 PacketList ServerMessages;
58 PacketList ClientMessages;
59
60 // temporary variables used to store the packets before pushing
61 // them in the corresponding queue
62 PACKET_DATA serverPacket;
63 PACKET_DATA clientPacket;
64
65 IP_ADDR RequiredAddress; // IP address accepted by te server
66
```

```

67 BOOL serverTurn; // used to alternate server and client listening
68 //used to store whether the packet contained an accepted IP address
69 BOOL IPAddressNotNull;
70 BOOL N; // network resource
71 BOOL serverKnown; // TRUE once the server's MAC address has been resolved
72 BOOL prevFromServer;
73 BOOL prevFromClient;
74
75 NODE_INFO ServerInfo; // server's IP and MAC addresses
76
77 // Private helper functions.
78 // These may or may not be present in all applications.
79 static void InitAppConfig(void);
80 static void InitializeBoard(void);
81 void DisplayWORD(BYTE pos, WORD w); //write WORDs on LCD for debugging
82
83 //
84 // PIC18 Interrupt Service Routines
85 //
86 // NOTE: Several PICs, including the PIC18F4620 revision A3 have a RETFIE
87 // FAST/MOVFF bug
88 // The interruptlow keyword is used to work around the bug when using C18
89
90 //LowISR
91 #if defined(__18CXX)
92     #if defined(HI_TECH_C)
93         void interrupt low_priority LowISR(void)
94     #elif defined(__SDCC__)
95         void LowISR(void) __interrupt (2) //ML for sdcc
96     #else
97         #pragma interruptlow LowISR
98         void LowISR(void)
99     #endif
100     {
101     TickUpdate();
102     }
103     #if !defined(__SDCC__) && !defined(HI_TECH_C)
104         //automatic with these compilers
105         #pragma code lowVector=0x18
106     void LowVector(void){_asm goto LowISR _endasm}
107     #pragma code // Return to default code section
108     #endif
109
110
111 //HighISR
112     #if defined(HI_TECH_C)
113         void interrupt HighISR(void)
114     #elif defined(__SDCC__)
115         void HighISR(void) __interrupt (1) //ML for sdcc
116     #else
117         #pragma interruptlow HighISR
118         void HighISR(void)
119     #endif
120     {
121         //insert here code for high level interrupt, if any
122     }
123     #if !defined(__SDCC__) && !defined(HI_TECH_C)
124         //automatic with these compilers
125         #pragma code highVector=0x8
126     void HighVector(void){_asm goto HighISR _endasm}
127     #pragma code // Return to default code section
128     #endif
129
130 #endif
131
132 const char* message; //pointer to message to display on LCD
133
134 /**
135  * Init the relay. This function opens the sockets to the server and the client and
136  * initializes the components used for the cooperative scheduling.

```



```

137 * @return 0 if everything succeeded, a negative number otherwise:
138 *         -) -1, if the server socket could not be open
139 *         -) -2, if the client socket could not be open
140 */
141 int DHCPRelayInit() {
142     // init the components
143     currentComponent = COMP1;
144
145     comp1 = WAITING_FOR_MESSAGE;
146     comp2 = SERVER_QUEUE_WAITING;
147     comp3 = CLIENT_QUEUE_WAITING;
148     comp2_2 = SEND_ARP_REQUEST;
149
150     // open the sockets
151     clientToServer = UDPOpen(DHCP_SERVER_PORT, NULL, DHCP_CLIENT_PORT);
152     serverToClient = UDPOpen(DHCP_CLIENT_PORT, NULL, DHCP_SERVER_PORT);
153
154     if (serverToClient == INVALID_UDP_SOCKET) {
155         DisplayString(0, "Invalid Server ");
156         return -1;
157     }
158     if (clientToServer == INVALID_UDP_SOCKET) {
159         DisplayString(16, "Invalid Client ");
160         return -2;
161     }
162
163     // init the queues
164     PacketListInit(&ServerMessages);
165     PacketListInit(&ClientMessages);
166
167     // init some boolean flags needed to organize the work
168     serverTurn = FALSE;
169
170     IPAddressNotNull = FALSE;
171
172     N = FALSE;
173
174     serverKnown = FALSE;
175
176     // set the server's IP address
177     ServerInfo.IPAddr.Val =
178         SERVER_IP_ADDR_BYTE1 |
179         SERVER_IP_ADDR_BYTE2<<8ul |
180         SERVER_IP_ADDR_BYTE3<<16ul |
181         SERVER_IP_ADDR_BYTE4<<24ul;
182
183     return 0;
184 }
185
186 /**
187 * Read a DHCP packet (if any) from a socket, and store it in the 'pkt'
188 * parameter. The function reads the DHCP header and store it for future
189 * use. It performs some basic checks on the hardware type (which must be
190 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
191 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
192 * 2u (BOOT_REPLY)) and the magic cookie.
193 * @param pkt Pointer to a packet storing the packet read from the network
194 * @param socket Socket used to read the packet (if any)
195 * @return 0 If everything succeeded, a negative number otherwise:
196 *         -) -1 if no packet is available on the selected socket, meaning
197 *             there are less than 241 bytes in its buffer;
198 *         -) -2, wrong hardware type
199 *         -) -3, wrong hardware length
200 *         -) -4, pkt is null or the socket is invalid
201 */
202 static int GetPacket(PACKET_DATA* pkt, UDP_SOCKET socket) {
203     // does the current socket have enough bytes ready to be read?
204     if (UDPIsGetReady(socket) < 241u) {
205         return -1;
206     }

```

```

207 // parameters validation check
208 if (pkt != NULL && socket != INVALID_UDP_SOCKET) {
209     BYTE        toBeDiscarded; // used to throw away unused fields
210     DWORD       magicCookie;
211     BOOTP_HEADER Header; // packet header
212     BYTE        Type = 0u; // MessageType
213     BYTE        Option; // used to iterate over the DHCP options
214     BYTE        Len; // length of the current option
215     BYTE        i; // used to add 0 paddings
216
217     UDPGetArray((BYTE*)&Header, sizeof(Header)); // get the header
218
219     // validate hardware interface and message type
220     if (Header.HardwareType != 1u) {
221         return -2;
222     }
223
224     if(Header.HardwareLen != 6u) {
225         return -3;
226     }
227
228     /*
229     * read and discard the following unused fields:
230     * - client hardware address
231     * - server host name
232     * - boot filename
233     */
234     for(i = 0; i < 64+128+(16-sizeof(MAC_ADDR)); i++) {
235         UDPGet(&toBeDiscarded);
236     }
237
238     // obtain magic cookie
239     UDPGetArray((BYTE*)&magicCookie, sizeof(DWORD));
240     // process options
241     while (UDPGet(&Option) && Option != DHCP_END_OPTION) {
242         UDPGet(&Len); // get the length
243         switch (Option) {
244             case DHCP_MESSAGE_TYPE:
245                 UDPGet(&Type); // get the message type
246                 memcpy(&(pkt -> MessageType), &Type, sizeof(BYTE)); // copy Type
247                 switch (Type) {
248                     case DHCP_DISCOVER_MESSAGE:
249                         DisplayString(16, "DHCP Discover  ");
250                         break;
251                     case DHCP_REQUEST_MESSAGE:
252                         DisplayString(16, "DHCP Request  ");
253                         break;
254                     case DHCP_OFFER_MESSAGE:
255                         DisplayString(16, "DHCP Offer    ");
256                         break;
257                     case DHCP_ACK_MESSAGE:
258                         DisplayString(16, "DHCP ACK      ");
259                         break;
260                 }
261                 break;
262             // get the accepted IP address
263             case DHCP_PARAM_REQUEST_IP_ADDRESS:
264                 if (Len == 4u) {
265                     UDPGetArray((BYTE*)&RequiredAddress, 4);
266                     IPAddressNotNull = TRUE;
267                     DisplayIPValue(RequiredAddress.Val);
268                 }
269                 break;
270         }
271         // remove any unprocessed bytes
272         while(Len) {
273             UDPGet(&i);
274             Len--;
275         }
276     }

```

```

277     // discard the rest of the buffer (it contains the 0 padding)
278     if (Option == DHCP_END_OPTION) {
279         UDPPDiscard();
280     }
281
282     // prepare the packet to be pushed
283     memcpy(&(pkt -> Header), &Header, sizeof(BOOTP_HEADER));
284     if (IPAddressNotNull == TRUE) {
285         memcpy(&(pkt -> RequiredAddress), &RequiredAddress, sizeof(IP_ADDR));
286         memcpy(&(pkt -> IPAddressNotNull), &IPAddressNotNull, sizeof(BOOL));
287     }
288     IPAddressNotNull = FALSE; // turn off flag
289     DisplayIPValue(pkt -> Header.ClientIP.Val);
290     return 0;
291 } else {
292     return -4;
293 }
294 }
295
296 /**
297  * Read a packet from the server socket, if any, and store it in
298  * 'serverPacket'. The function reads the DHCP header and store it for future
299  * use. It performs some basic checks on the hardware type (which must be
300  * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
301  * not validate the message type (which may be both 1u (BOOT_REQUEST) and
302  * 2u (BOOT_REPLY)) and the magic cookie.
303  * @return 0 If everything succeeded, a negative number otherwise:
304  *      -) -1 if no packet is available on the selected socket, meaning
305  *          there are less then 241 bytes in its buffer;
306  *      -) -2, wrong hardware type
307  *      -) -3, wrong hardware length
308  *      -) -4, pkt is null or the socket is invalid
309  */
310 static int GetServerPacket() {
311     int res = GetPacket(&serverPacket, serverToClient);
312     if (res == 0) {
313         compl = PUSH_CLIENT_QUEUE;
314         prevFromServer = TRUE;
315     }
316     return res;
317 }
318
319 /**
320  * Read a packet from the client socket, if any, and store it in
321  * 'clientPacket'. The function reads the DHCP header and store it for future
322  * use. It performs some basic checks on the hardware type (which must be
323  * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
324  * not validate the message type (which may be both 1u (BOOT_REQUEST) and
325  * 2u (BOOT_REPLY)) and the magic cookie.
326  * @return 0 If everything succeeded, a negative number otherwise:
327  *      -) -1 if no packet is available on the selected socket, meaning
328  *          there are less then 241 bytes in its buffer;
329  *      -) -2, wrong hardware type
330  *      -) -3, wrong hardware length
331  *      -) -4, pkt is null or the socket is invalid
332  */
333 static int GetClientPacket() {
334     int res = GetPacket(&clientPacket, clientToServer);
335     if (res == 0) {
336         compl = PUSH_SERVER_QUEUE;
337     }
338     return res;
339     //return GetPacket(&clientPacket, clientToServer);
340 }
341
342 /**
343  * Send a packet to the server, taking it from ServerMessages. This function assumes
344  * that queue to be not empty. The function copies the message in the socket's buffer
345  * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
346  * @precondition ServerMessages is not empty

```

```

347 * @precondition ServerInfo contains both the server's IP and MAC address. If the latter
348 * is not known, an ARP request should be made in order to get it.
349 */
350 static void SendToServer() {
351     // check if the buffer has enough space
352     if (UDPIsPutReady(clientToServer) >= 300u) {
353         BYTE i; // used to add the 0 padding
354         UDP_SOCKET_INFO *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
        socket
355         // pop the packet
356         PACKET_DATA pkt;
357         PacketListPop(&pkt, &ServerMessages);
358         DisplayString(0, "Send to Server ");
359
360         // set socket info
361         socket -> remoteNode.IPAddr.Val = ServerInfo.IPAddr.Val;
362         for(i = 0; i < 6; i++) {
363             socket -> remoteNode.MACAddr.v[i] = ServerInfo.MACAddr.v[i];
364         }
365
366         // copy header DHCP
367         UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
368         UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
369         UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
370         UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));
371         UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID)
    );
372         UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.
        SecondsElapsed));
373         UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
374         UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
375         UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
376         UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
377         UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
378         UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));
379
380         // the other fields are set to zero
381         for (i = 0; i < 202u; i++) {
382             UDPPut(0);
383         }
384
385         // put magic cookie as per RFC 1533.
386         UDPPut(99);
387         UDPPut(130);
388         UDPPut(83);
389         UDPPut(99);
390
391         // put message type
392         UDPPut(DHCP_MESSAGE_TYPE);
393         UDPPut(DHCP_MESSAGE_TYPE_LEN);
394         UDPPut(pkt.MessageType);
395
396         // Option: Subnet Mask
397         UDPPut(DHCP_SUBNET_MASK);
398         UDPPut(sizeof(IP_ADDR));
399         UDPPutArray((BYTE*)&AppConfig.MyMask, sizeof(IP_ADDR));
400
401         // Option: Server identifier
402         UDPPut(DHCP_SERVER_IDENTIFIER);
403         UDPPut(sizeof(IP_ADDR));
404         UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));
405
406         // Option: Router/Gateway address
407         UDPPut(DHCP_ROUTER);
408         UDPPut(sizeof(IP_ADDR));
409         UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));
410
411         // if there is an IP address, add it
412         if (pkt.IPAddressNotNull == TRUE) {
413             UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS);

```

```

414         UDPPut (DHCP_PARAM_REQUEST_IP_ADDRESS_LEN);
415         UDPPutArray((BYTE*)&pkt.RequiredAddress, sizeof(IP_ADDR));
416         IPAddressNotNull = FALSE; // reset the global variable used as a flag
417     }
418
419     UDPPut (DHCP_END_OPTION); // end the packet
420
421     // add zero padding to ensure compatibility with old BOOTP relays that discard
422     // packets smaller than 300 octets
423     while(UDPTxCount < 300u) {
424         UDPPut (0);
425     }
426
427     UDPFlush(); // transmit
428 }
429 }
430
431 /**
432  * Send a packet to the client, taking it from ClientMessages. This function assumes
433  * that queue to be not empty. The function copies the message in the socket's buffer
434  * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
435  * @precondition ClientMessages is not empty
436  */
437 static void SendToClient() {
438     // check if the buffer has enough space
439     if (UDPIsPutReady(serverToClient) >= 300u) {
440         BYTE i; // used to add te 0 padding
441         UDP_SOCKET_INFO *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
         socket
442         // pop the packet from the queue
443         PACKET_DATA pkt;
444         PacketListPop(&pkt, &ClientMessages);
445         DisplayString(0, "Send to Client ");
446
447         // set socket info
448         socket -> remoteNode.IPAddr.Val = BROADCAST;
449         socket -> remotePort = DHCP_CLIENT_PORT;
450         for(i = 0; i < 6u; i++){ // copy client's MAC address (and take it from CHADDR)
451             socket -> remoteNode.MACAddr.v[i] = pkt.Header.ClientMAC.v[i];
452         }
453
454         // copy header DHCP
455         UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
456         UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
457         UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
458         UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));
459         UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID)
);
460         UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.
SecondsElapsed));
461         UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
462         UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
463         UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
464         UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
465         UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
466         UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));
467
468         // the other fields are set to zero
469         for (i = 0; i < 202u; i++) {
470             UDPPut (0);
471         }
472
473         // put magic cookie 0x63538263, little endian
474         UDPPut (0x63);
475         UDPPut (0x82);
476         UDPPut (0x53);
477         UDPPut (0x63);
478
479         // put message type
480         UDPPut (DHCP_MESSAGE_TYPE);

```

```

481     UDPPut (DHCP_MESSAGE_TYPE_LEN);
482     UDPPut (pkt.MessageType);
483
484     UDPPut (DHCP_END_OPTION); // end packet
485
486     // add zero padding to ensure compatibility with old BOOTP relays that discard
487     // packets smaller than 300 octets
488     while (UDPTxCount < 300u) {
489         UDPPut (0);
490     }
491
492     UDPFlush(); // transmit
493 }
494 }
495
496 /** Schedule the first parallel component.
497  * This component is responsible for getting the packets from the network and pushing
498  * them
499  * in the right queue (depending if a packet comes from the server or from the client).
500  */
501 static void Component1() {
502     switch (comp1) {
503         // root
504         case WAITING_FOR_MESSAGE:
505             if (prevFromClient == TRUE) {
506                 comp1 = PUSH_SERVER_QUEUE;
507             } else {
508                 GetServerPacket();
509                 GetClientPacket();
510                 // serve the server, but serve the client at next round
511                 if (prevFromServer == TRUE && comp1 == PUSH_SERVER_QUEUE) {
512                     prevFromServer = FALSE;
513                     prevFromClient = TRUE;
514                     comp1 = PUSH_CLIENT_QUEUE;
515                 }
516             }
517             break;
518         case PUSH_CLIENT_QUEUE:
519             if (PacketListPush(&ClientMessages, &serverPacket) == 0) {
520                 // cross the transaction iff the push succeeded
521                 comp1 = PUSH_CLIENT_QUEUE_T;
522             } else {
523                 break;
524             }
525         case PUSH_CLIENT_QUEUE_T:
526             comp1 = WAITING_FOR_MESSAGE;
527             break;
528         // client branch
529         case PUSH_SERVER_QUEUE:
530             if (PacketListPush(&ServerMessages, &clientPacket) == 0) {
531                 // cross the transaction iff the push succeeded
532                 comp1 = PUSH_SERVER_QUEUE_T;
533             } else {
534                 break;
535             }
536         case PUSH_SERVER_QUEUE_T:
537             comp1 = WAITING_FOR_MESSAGE;
538             break;
539     }
540 }
541
542 /**
543  * Schedule the second parallel component.
544  * This component is responsible for popping a packet coming from the client,
545  * if any, and sending it to the server. It also makes an ARP request the very
546  * first time a transmission to the server is required. After its MAC address
547  * has been resolved, it sets the 'serverKnown' flag to TRUE and stops sending
548  * ARP requests.
549  */
550 static void Component2() {

```

```

550     switch (comp2) {
551         case SERVER_QUEUE_WAITING:
552             if (!PacketListIsEmpty(&ServerMessages)) {
553                 // cross iff the queue is not empty
554                 comp2 = SERVER_QUEUE_WAITING_T;
555             } else {
556                 break;
557             }
558         case SERVER_QUEUE_WAITING_T:
559             comp2 = GET_SERVER_IP_ADDRESS;
560             break;
561         case GET_SERVER_IP_ADDRESS:
562             switch (comp2_2) {
563                 case IDENTIFY_SERVER:
564                     if (serverKnown == TRUE) {
565                         comp2 = IDENTIFY_SERVER_TO_TX;
566                         break; // immediately go to transmission if the server is known
567                     } else {
568                         comp2_2 = IDENTIFY_SERVER_TO_ARP;
569                     }
570                 case IDENTIFY_SERVER_TO_ARP:
571                     if (N == FALSE) {
572                         comp2_2 = SEND_ARP_REQUEST;
573                         N = TRUE;
574                     }
575                     break;
576                 case SEND_ARP_REQUEST:
577                     ARPResolve(&ServerInfo.IPAddr);
578                     DisplayString(0, "Send ARP Request");
579                     comp2_2 = SEND_ARP_REQUEST_T;
580                 case SEND_ARP_REQUEST_T:
581                     comp2_2 = PROCESS_ARP_ANSWER;
582                     N = FALSE;
583                     break;
584                 case PROCESS_ARP_ANSWER:
585                     if (ARPIsResolved(&ServerInfo.IPAddr, &ServerInfo.MACAddr) == TRUE)
586             {
587                 DisplayString(0, "MACAddr Resolved");
588                 serverKnown = TRUE;
589                 comp2_2 = PROCESS_ARP_ANSWER_T;
590             } else {
591                 if (N == FALSE) {
592                     comp2_2 = SEND_ARP_REQUEST;
593                     N = TRUE;
594                 }
595                 break;
596             }
597                 case PROCESS_ARP_ANSWER_T:
598                     if (N == FALSE) {
599                         comp2_2 = SEND_ARP_REQUEST;
600                         comp2 = TX_TO_SERVER;
601                         N = TRUE;
602                     }
603                     break;
604             }
605             if (comp2 != IDENTIFY_SERVER_TO_TX) {
606                 // go through the transition if the server was known
607                 break;
608             }
609         case IDENTIFY_SERVER_TO_TX:
610             if (N == FALSE) {
611                 N = TRUE;
612                 comp2 = TX_TO_SERVER;
613             }
614             break;
615         case TX_TO_SERVER:
616             SendToServer();
617             comp2 = TX_TO_SERVER_T;
618         case TX_TO_SERVER_T:
619             N = FALSE;

```

```

619         comp2 = SERVER_QUEUE_WAITING;
620         break;
621     }
622 }
623
624 /**
625  * Schedule the third parallel component.
626  * This component is responsible for popping a packet coming from the server,
627  * if any, and sending it to the client.
628  */
629 static void Component3() {
630     switch (comp3) {
631         case CLIENT_QUEUE_WAITING:
632             if (!PacketListIsEmpty(&ClientMessages)) {
633                 comp3 = CLIENT_QUEUE_WAITING_T;
634             } else {
635                 break;
636             }
637         case CLIENT_QUEUE_WAITING_T:
638             if (N == FALSE) {
639                 comp3 = TX_TO_CLIENT;
640                 N = TRUE;
641             }
642             break;
643         case TX_TO_CLIENT:
644             SendToClient();
645             comp3 = TX_TO_CLIENT_T;
646         case TX_TO_CLIENT_T:
647             N = FALSE;
648             comp3 = CLIENT_QUEUE_WAITING;
649             break;
650     }
651 }
652
653 /**
654  * Handle the scheduling of the DHCP relay, including the transitions
655  * among the different components. The scheduling is done iff the DHCP
656  * is actually enabled, meaning the flag 'AppConfig.Flags.bIsDHCPEnabled'
657  * is not zero.
658  */
659 static void DHCPRelayTask() {
660     if (AppConfig.Flags.bIsDHCPEnabled) {
661         switch(currentComponent) {
662             case INIT:
663                 if (DHCPRelayInit() == -1) {
664                     UDPClose(serverToClient);
665                     UDPClose(clientToServer);
666                     currentComponent = INIT;
667                 }
668                 break;
669             case COMP1:
670                 Component1();
671                 currentComponent = COMP2;
672                 break;
673             case COMP2:
674                 Component2();
675                 currentComponent = COMP3;
676                 break;
677             case COMP3:
678                 Component3();
679                 currentComponent = COMP1;
680                 break;
681         }
682     } else {
683         DisplayString(0, "DHCP Not Enabled");
684     }
685 }
686
687 //
688 // Main application entry point.

```



```

689 //
690
691
692 #if defined(__18CXX) || defined(__SDCC__)
693 void main(void)
694 #else
695 int main(void)
696 #endif
697 {
698     static TICK t = 0;
699     TICK nt = 0; //TICK is DWORD, thus 32 bits
700     BYTE loopctr = 0; //ML Debugging
701     WORD lloopctr = 14; //ML Debugging
702
703     static DWORD dwLastIP = 0;
704
705     // Initialize interrupts and application specific hardware
706     InitializeBoard();
707
708     // Initialize Timer0, and low priority interrupts, used as clock.
709     TickInit();
710
711     // Initialize Stack and application related variables in AppConfig.
712     InitAppConfig();
713
714     // Initialize core stack layers (MAC, ARP, TCP, UDP) and
715     // application modules (HTTP, SNMP, etc.)
716     StackInit();
717
718     #ifdef UART_DEBUG_ON
719         UARTConfig();
720     #endif
721
722     #ifdef USE_LCD
723         LCDTaskInit();
724     #endif
725
726     // Initialize and display message on the LCD
727     LCDInit();
728     DelayMs(100);
729     DisplayString (0,"Waiting client "); //first arg is start position on 32 pos LCD
730
731     currentComponent = INIT;
732
733     // Now that all items are initialized, begin the co-operative
734     // multitasking loop. This infinite loop will continuously
735     // execute all stack-related tasks, as well as your own
736     // application's functions. Custom functions should be added
737     // at the end of this loop.
738
739     // Note that this is a "co-operative multi-tasking" mechanism
740     // where every task performs its tasks (whether all in one shot
741     // or part of it) and returns so that other tasks can do their
742     // job.
743     // If a task needs very long time to do its job, it must be broken
744     // down into smaller pieces so that other tasks can have CPU time.
745
746
747     while(1)
748     {
749
750         // Blink LED0 (right most one) every second.
751         nt = TickGetDiv256();
752         if((nt - t) >= (DWORD)(TICK_SECOND/1024ul))
753         {
754             t = nt;
755             LED0_IO ^= 1;
756             ClrWdt(); //Clear the watchdog
757         }
758

```

```

759 // This task performs normal stack task including checking
760 // for incoming packet, type of packet and calling
761 // appropriate stack entity to process it.
762 StackTask();
763
764 // This tasks invokes each of the core stack application tasks
765 // StackApplications(); //all except dhcp, ping and arp
766
767 // LCD task
768 #ifdef USE_LCD
769 LCDTask();
770 #endif
771
772 // Process application specific tasks here.
773 #ifdef STACK_USE_DHCP_RELAY
774 DHCPRelayTask();
775 #endif
776
777 // If the local IP address has changed (ex: due to DHCP lease change)
778 // write the new IP address to the LCD display, UART, and Announce
779 // service
780 if(dwLastIP != AppConfig.MyIPAddr.Val)
781 {
782     dwLastIP = AppConfig.MyIPAddr.Val;
783     #if defined(__SDCC__)
784         DisplayIPValue(dwLastIP); // must be a WORD: sdcc does not
785                                     // pass aggregates
786     #else
787         DisplayIPValue(AppConfig.MyIPAddr);
788     #endif
789 }
790 } //end of while(1)
791 } //end of main()
792
793 /*****
794 Function DisplayWORD:
795 writes a WORD in hexa on the position indicated by
796 pos.
797 - pos=0 -> 1st line of the LCD
798 - pos=16 -> 2nd line of the LCD
799
800 __SDCC__ only: for debugging
801 *****/
802 #if defined(__SDCC__)
803 void DisplayWORD(BYTE pos, WORD w) //WORD is a 16 bits unsigned
804 {
805     BYTE WDigit[6]; //enough for a number < 65636: 5 digits + \0
806     BYTE j;
807     BYTE LCDPos=0; //write on first line of LCD
808     unsigned radix=10; //type expected by sdcc's ultoa()
809
810     LCDPos=pos;
811     ultoa(w, WDigit, radix);
812     for(j = 0; j < strlen((char*)WDigit); j++)
813     {
814         LCDText[LCDPos++] = WDigit[j];
815     }
816     if(LCDPos < 32u)
817         LCDText[LCDPos] = 0;
818     LCDUpdate();
819 }
820 /*****
821 Function DisplayString:
822 Writes an IP address to string to the LCD display
823 starting at pos
824 *****/
825 void DisplayString(BYTE pos, char* text)
826 {
827     BYTE l= strlen(text)+1;
828     BYTE max= 32-pos;

```

```

829     strcpy((char*)&LCDText[pos], text, (l<max)?l:max );
830     LCDUpdate();
831 }
832 #endif
833
834 /*****
835  Function DisplayIPValue:
836  Writes an IP address to the LCD display
837  *****/
838
839 #if defined(__SDCC__)
840 void DisplayIPValue(DWORD IPdw) // 32 bits
841 #else
842 void DisplayIPValue(IP_ADDR IPVal)
843 #endif
844 {
845     BYTE IPDigit[4]; //enough for a number <256: 3 digits + \0
846     BYTE i;
847     BYTE j;
848     BYTE LCDPos=16; //write on second line of LCD
849 #if defined(__SDCC__)
850     unsigned int IP_field, radix=10; //type expected by sdcc's uitoa()
851 #endif
852
853     for(i = 0; i < sizeof(IP_ADDR); i++) //sizeof(IP_ADDR) is 4
854     {
855 #if defined(__SDCC__)
856         IP_field = (WORD) (IPdw>>(i*8))&0xff; //ML
857         uitoa(IP_field, IPDigit, radix); //ML
858 #else
859         uitoa((WORD) IPVal.v[i], IPDigit);
860 #endif
861
862         for(j = 0; j < strlen((char*)IPDigit); j++)
863         {
864             LCDText[LCDPos++] = IPDigit[j];
865         }
866         if(i == sizeof(IP_ADDR)-1)
867             break;
868         LCDText[LCDPos++] = '.';
869     }
870     if(LCDPos < 32u)
871         LCDText[LCDPos] = 0;
872     LCDUpdate();
873 }
874
875
876
877 /*****
878  Function:
879  static void InitializeBoard(void)
880
881  Description:
882  This routine initializes the hardware. It is a generic initialization
883  routine for many of the Microchip development boards, using definitions
884  in HardwareProfile.h to determine specific initialization.
885
886  Precondition:
887  None
888
889  Parameters:
890  None - None
891
892  Returns:
893  None
894
895  Remarks:
896  None
897  *****/
898 static void InitializeBoard(void)

```

```

899 {
900     // LEDs
901     LED0_TRIS = 0; //LED0
902     LED1_TRIS = 0; //LED1
903     LED2_TRIS = 0; //LED2
904     LED3_TRIS = 0; //LED_LCD1
905     LED4_TRIS = 0; //LED_LCD2
906     LED5_TRIS = 0; //LED5=RELAY1
907     LED6_TRIS = 0; //LED7=RELAY2
908     #if (!defined(EXPLORER_16) &&!defined(OLIMEX_MAXI)) // Pin multiplexed with
909         // a button on EXPLORER_16 and not used on OLIMEX_MAXI
910         LED7_TRIS = 0;
911     #endif
912     LED_PUT(0x00); //turn off LED0 - LED2
913     RELAY_PUT(0x00); //turn relays off to save power
914
915     // Set clock to 25 MHz
916     // The primary oscillator runs at the speed of the 25MHz external quartz
917     OSTUNE = 0x00;
918
919     // Switch to primary oscillator mode,
920     // regardless of if the config fuses tell us to start operating using
921     // the the internal RC
922     // The external clock must be running and must be 25MHz for the
923     // Ethernet module and thus this Ethernet bootloader to operate.
924     if (OSCCONbits.IDLEN) //IDLEN = 0x80; 0x02 selects the primary clock
925         OSCCON = 0x82;
926     else
927         OSCCON = 0x02;
928
929     // Enable Interrupts
930     RCONbits.IPEN = 1; // Enable interrupt priorities
931     INTCONbits.GIEH = 1;
932     INTCONbits.GIEL = 1;
933 }
934 }
935
936 /*****
937 * Function:          void InitAppConfig(void)
938 *
939 * PreCondition:      MPFSInit() is already called.
940 *
941 * Input:             None
942 *
943 * Output:            Write/Read non-volatile config variables.
944 *
945 * Side Effects:      None
946 *
947 * Overview:          None
948 *
949 * Note:              None
950 *****/
951
952 static void InitAppConfig(void)
953 {
954     AppConfig.Flags.bIsDHCPEnabled = TRUE;
955     AppConfig.Flags.bInConfigMode = TRUE;
956
957     //ML using sdcc (MPLAB has a trick to generate serial numbers)
958     // first 3 bytes indicate manufacturer; last 3 bytes are serial number
959     AppConfig.MyMACAddr.v[0] = 0;
960     AppConfig.MyMACAddr.v[1] = 0x04;
961     AppConfig.MyMACAddr.v[2] = 0xA3;
962     AppConfig.MyMACAddr.v[3] = 0x01;
963     AppConfig.MyMACAddr.v[4] = 0x02;
964     AppConfig.MyMACAddr.v[5] = 0x03;
965
966     //ML if you want to change, see TCPIPConfig.h
967     AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 |
968         MY_DEFAULT_IP_ADDR_BYTE2<<8ul | MY_DEFAULT_IP_ADDR_BYTE3<<16ul |

```

```

969         MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
970     AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
971     AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 |
972         MY_DEFAULT_MASK_BYTE2<<8ul | MY_DEFAULT_MASK_BYTE3<<16ul |
973         MY_DEFAULT_MASK_BYTE4<<24ul;
974     AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
975     AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 |
976         MY_DEFAULT_GATE_BYTE2<<8ul | MY_DEFAULT_GATE_BYTE3<<16ul |
977         MY_DEFAULT_GATE_BYTE4<<24ul;
978     AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 |
979         MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul |
980         MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul |
981         MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
982     AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 |
983         MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul |
984         MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul |
985         MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;
986 }

```

Listing 4: DHCPRelay.c