

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

Tommaso Marinelli

Matteo Di Pirro

January 7, 2018

1 User manual

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 comprises the ports 2, 3, 4, and 5; a WAN is set at the port 1. The DHCP server should be connected to the WAN, while both relay and clients should be on the 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 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** 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.

2.3 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 `TCPIP.h`.

3 Documentation for programmers

3.1 Specification

The program implements a DHCP relay. Figure 1 depicts how it works.

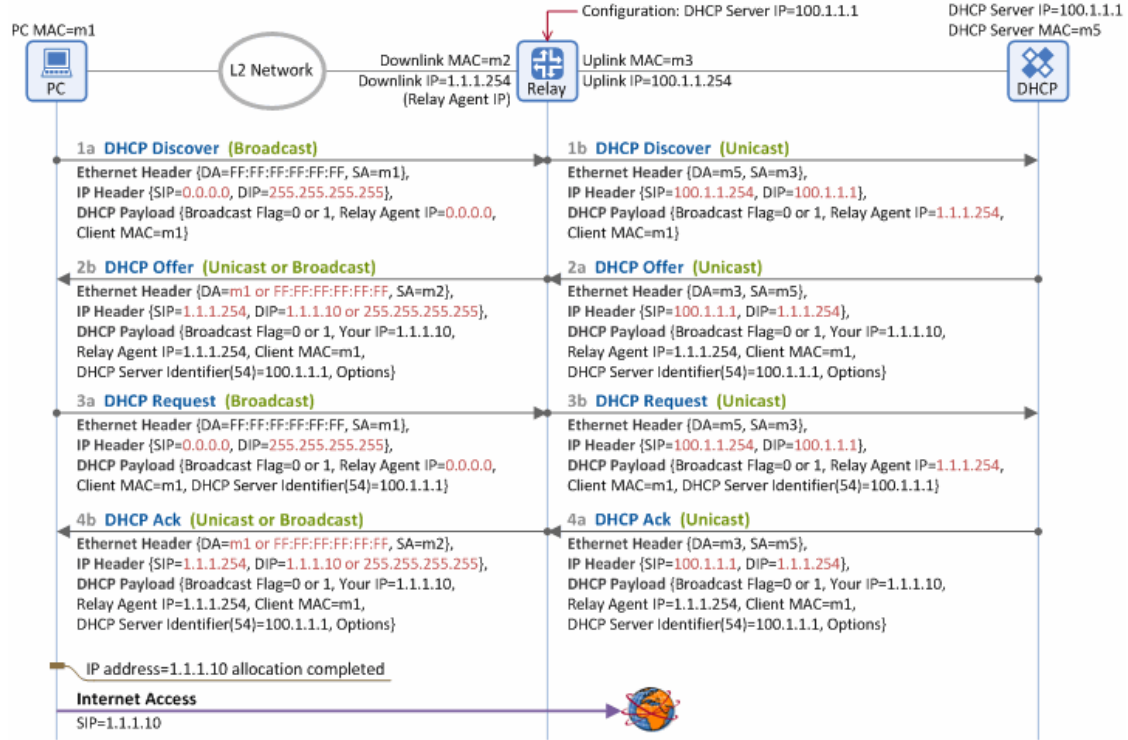


Figure 1: Example of DHCP protocol communication¹

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
 - **IP Payload**

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

- * 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

Our ASG diagram is depicted in Figure 2.

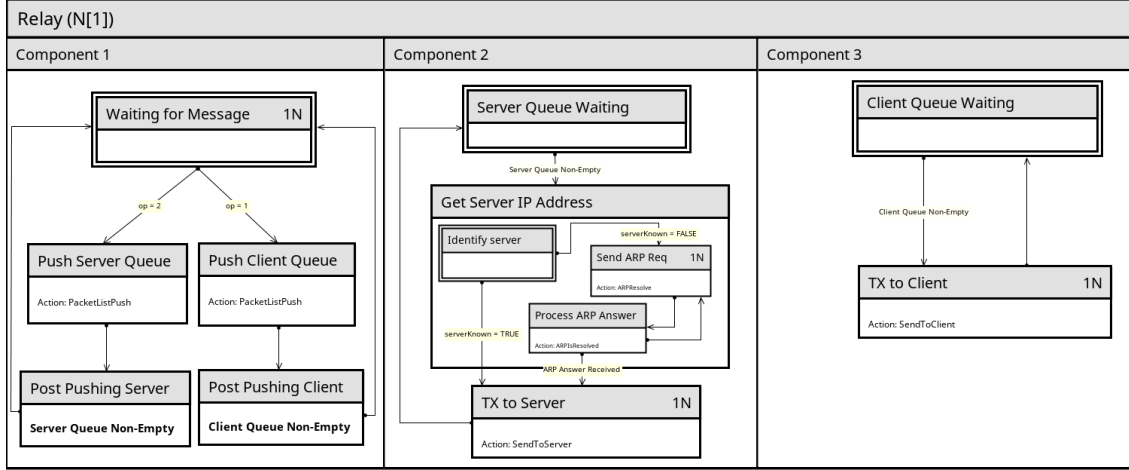


Figure 2: ASG diagram modeling a DHCP relay

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.

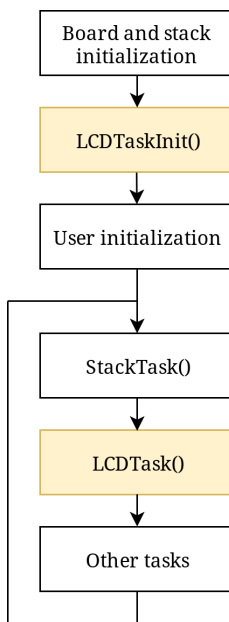
One last point worthy to be explained is the resource N (going for *Network*). The board may only transmit in a half-duplex manner, and it is therefore necessary to allow only one substate at a time to use the network. This is the meaning of N. Each substate accessing the network must

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 3.3.4.

3.2 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.

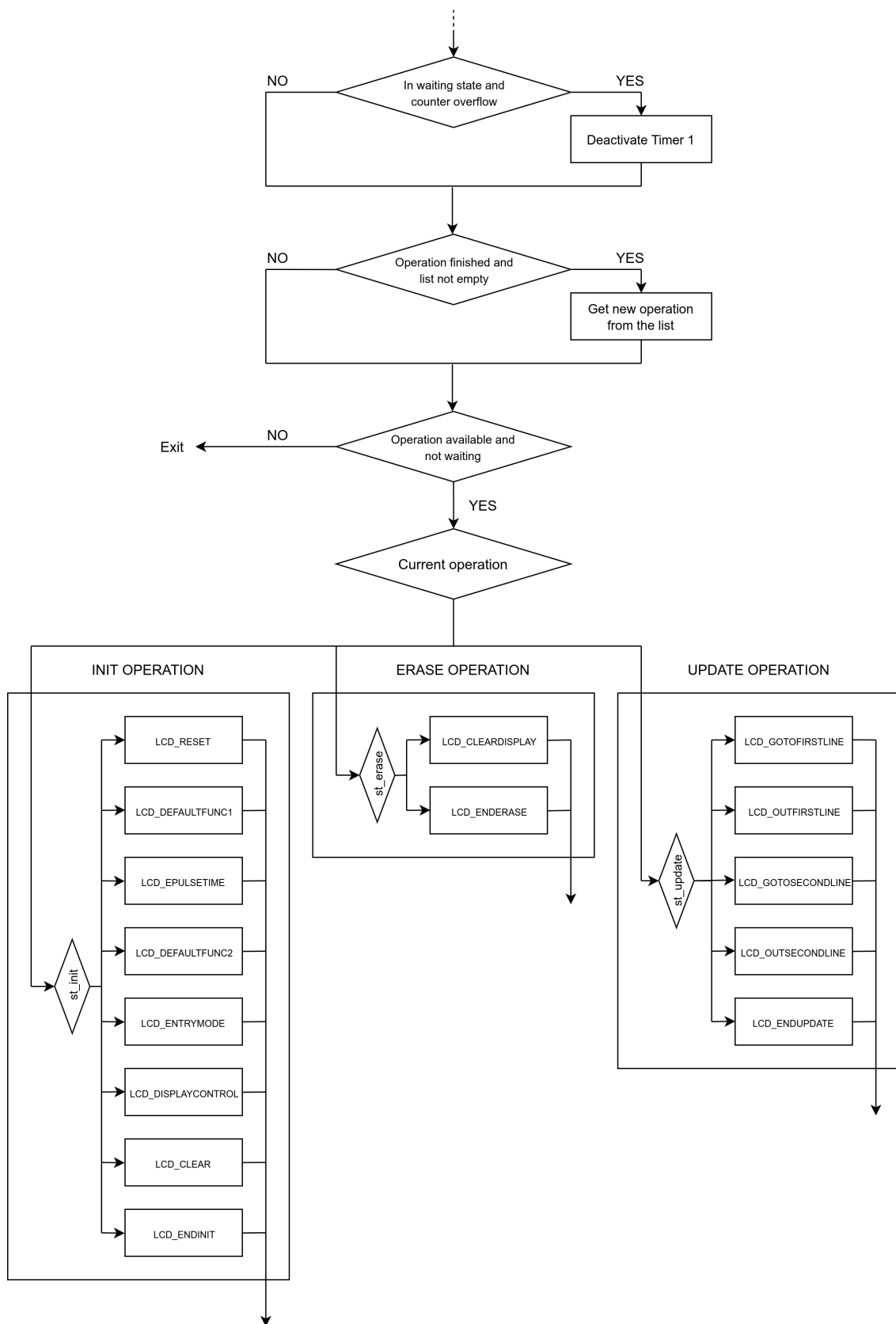


Figure 3: LCD task instruction flow chart

3.3 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 functions 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 queues 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 and 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.

3.3.1 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 has 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.

3.3.2 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.

3.3.3 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`.

3.3.4 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. 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, and there is no chance that a component never executes.

4 Program listings

4.1 DHCP Relay

Listing 4.1 lists the relay header file, while Listing 4.1 lists the actual implementation.

```
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 ANOString[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

```

```

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    (97ul) // (10ul)
39 #define SERVER_IP_ADDR_BYTE4    (10ul)
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 the 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         DEBUGMSG("NOT READY\r\n");
206         return -1;
207     }
208     // parameters validation check
209     if (pkt != NULL && socket != INVALID_UDP_SOCKET) {
210         BYTE toBeDiscarded; // used to throw away unused fields
211         DWORD magicCookie;
212         BOOTP_HEADER Header; // packet header
213         BYTE Type = 0u; // MessageType
214         BYTE Option; // used to iterate over the DHCP options
215         BYTE Len; // length of the current option
216         BYTE i; // used to add 0 paddings
217
218         UDPGetArray((BYTE*)&Header, sizeof(Header)); // get the header
219
220         // validate hardware interface and message type
221         if (Header.HardwareType != 1u) {
222             return -2;
223         }
224
225         if(Header.HardwareLen != 6u) {
226             return -3;
227         }
228
229         /*
230         * read and discard the following unused fields:
231         * - client hardware address
232         * - server host name
233         * - boot filename
234         */

```

```

235     for(i = 0; i < 64+128+(16-sizeof(MAC_ADDR)); i++) {
236         UDPGet(&toBeDiscarded);
237     }
238
239     // obtain magic cookie
240     UDPGetArray((BYTE*)&magicCookie, sizeof(DWORD));
241     // process options
242     while (UDPGet(&Option) && Option != DHCP_END_OPTION) {
243         UDPGet(&Len); // get the length
244         switch (Option) {
245             case DHCP_MESSAGE_TYPE:
246                 UDPGet(&Type); // get the message type
247                 memcpy(&(pkt -> MessageType), &Type, sizeof(BYTE)); // copy Type
248                 switch (Type) {
249                     case DHCP_DISCOVER_MESSAGE:
250                         DisplayString(16, "DHCP Discover  ");
251                         break;
252                     case DHCP_REQUEST_MESSAGE:
253                         DisplayString(16, "DHCP Request  ");
254                         break;
255                     case DHCP_OFFER_MESSAGE:
256                         DisplayString(16, "DHCP Offer    ");
257                         break;
258                     case DHCP_ACK_MESSAGE:
259                         DisplayString(16, "DHCP ACK      ");
260                         break;
261                 }
262                 break;
263                 // get the accepted IP address
264             case DHCP_PARAM_REQUEST_IP_ADDRESS:
265                 if (Len == 4u) {
266                     UDPGetArray((BYTE*)&RequiredAddress, 4);
267                     IPAddressNotNull = TRUE;
268                     DisplayIPValue(RequiredAddress.Val);
269                 }
270                 break;
271         }
272         // remove any unprocessed bytes
273         while(Len) {
274             UDPGet(&i);
275             Len--;
276         }
277     }
278     // discard the rest of the buffer (it contains the 0 padding)
279     if (Option == DHCP_END_OPTION) {
280         UDPDiscard();
281     }
282
283     // prepare the packet to be pushed
284     memcpy(&(pkt -> Header), &Header, sizeof(BOOTP_HEADER));
285     if (IPAddressNotNull == TRUE) {
286         memcpy(&(pkt -> RequiredAddress), &RequiredAddress, sizeof(IP_ADDR));
287         memcpy(&(pkt -> IPAddressNotNull), &IPAddressNotNull, sizeof(BOOL));
288     }
289     IPAddressNotNull = FALSE; // turn off flag
290     DisplayIPValue(pkt -> Header.ClientIP.Val);
291     return 0;
292 } else {
293     return -4;
294 }
295 }
296
297 /**
298 * Read a packet from the server socket, if any, and store it in
299 * 'serverPacket'. The function reads the DHCP header and store it for future
300 * use. It performs some basic checks on the hardware type (which must be
301 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
302 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
303 * 2u (BOOT_REPLY)) and the magic cookie.
304 * @return 0 If everything succeeded, a negative number otherwise:

```

```

305 *      -) -1 if no packet is available on the selected socket, meaning
306 *          there are less then 241 bytes in its buffer;
307 *      -) -2, wrong hardware type
308 *      -) -3, wrong hardware length
309 *      -) -4, pkt is null or the socket is invalid
310 */
311 static int GetServerPacket() {
312     int res = GetPacket(&serverPacket, serverToClient);
313     if (res == 0) {
314         DisplayString(0, "To Push Client");
315         compl = PUSH_CLIENT_QUEUE;
316         prevFromServer = TRUE;
317     }
318     return res;
319 }
320
321 /**
322 * Read a packet from the client socket, if any, and store it in
323 * 'clientrPacket'. The function reads the DHCP header and store it for future
324 * use. It performs some basic checks on the hardware type (which must be
325 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
326 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
327 * 2u (BOOT_REPLY)) and the magic cookie.
328 * @return 0 If everything succeeded, a negative number otherwise:
329 *      -) -1 if no packet is available on the selected socket, meaning
330 *          there are less then 241 bytes in its buffer;
331 *      -) -2, wrong hardware type
332 *      -) -3, wrong hardware length
333 *      -) -4, pkt is null or the socket is invalid
334 */
335 static int GetClientPacket() {
336     int res = GetPacket(&clientPacket, clientToServer);
337     if (res == 0) {
338         DisplayString(0, "To Push Server");
339         compl = PUSH_SERVER_QUEUE;
340     }
341     return res;
342     //return GetPacket(&clientPacket, clientToServer);
343 }
344
345 /**
346 * Send a packet to the server, taking it from ServerMessages. This function assumes
347 * that queue to be not empty. The function copies the message in the socket's buffer
348 * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
349 * @precondition ServerMessages is not empty
350 * @precondition ServerInfo contains both the server's IP and MAC address. If the latter
351 * is not known, an ARP request should be made in order to get it.
352 */
353 static void SendToServer() {
354     // check if the buffer has enough space
355     if (UDPisPutReady(clientToServer) >= 300u) {
356         BYTE i; // used to add the 0 padding
357         UDP_SOCKET_INFO *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
socket
358         // pop the packet
359         PACKET_DATA pkt;
360         PacketListPop(&pkt, &ServerMessages);
361         DisplayString(0, "Send to Server");
362
363         // set socket info
364         socket -> remoteNode.IPAddr.Val = ServerInfo.IPAddr.Val;
365         for(i = 0; i < 6; i++) {
366             socket -> remoteNode.MACAddr.v[i] = ServerInfo.MACAddr.v[i];
367         }
368
369         // copy header DHCP
370         UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
371         UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
372         UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
373         UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));

```



```

374     UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID)
);
375     UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.
SecondsElapsed));
376     UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
377     UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
378     UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
379     UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
380     UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
381     UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));
382
383     // the other fields are set to zero
384     for (i = 0; i < 202u; i++) {
385         UDPPut(0);
386     }
387
388     // put magic cookie as per RFC 1533.
389     UDPPut(99);
390     UDPPut(130);
391     UDPPut(83);
392     UDPPut(99);
393
394     // put message type
395     UDPPut(DHCP_MESSAGE_TYPE);
396     UDPPut(DHCP_MESSAGE_TYPE_LEN);
397     UDPPut(pkt.MessageType);
398
399     // Option: Subnet Mask
400     UDPPut(DHCP_SUBNET_MASK);
401     UDPPut(sizeof(IP_ADDR));
402     UDPPutArray((BYTE*)&AppConfig.MyMask, sizeof(IP_ADDR));
403
404     // Option: Server identifier
405     UDPPut(DHCP_SERVER_IDENTIFIER);
406     UDPPut(sizeof(IP_ADDR));
407     UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));
408
409     // Option: Router/Gateway address
410     UDPPut(DHCP_ROUTER);
411     UDPPut(sizeof(IP_ADDR));
412     UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));
413
414     // if there is an IP address, add it
415     if (pkt.IPAddressNotNull == TRUE) {
416         UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS);
417         UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS_LEN);
418         UDPPutArray((BYTE*)&pkt.RequiredAddress, sizeof(IP_ADDR));
419         IPAddressNotNull = FALSE; // reset the global variable used as a flag
420     }
421
422     UDPPut(DHCP_END_OPTION); // end the packet
423
424     // add zero padding to ensure compatibility with old BOOTP relays that discard
425     // packets smaller than 300 octets
426     while(UDPTxCount < 300u) {
427         UDPPut(0);
428     }
429
430     UDPFlush(); // transmit
431     //DisplayString(0, "CL to SERV");
432 }
433 }
434
435 /**
436  * Send a packet to the client, taking it from ClientMessages. This function assumes
437  * that queue to be not empty. The function copies the message in the socket's buffer
438  * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
439  * @precondition ClientMessages is not empty
440  */
441 static void SendToClient() {

```

```

442 // check if the buffer has enough space
443 if (UDPIsPutReady(serverToClient) >= 300u) {
444     BYTE i; // used to add te 0 padding
445     UDP_SOCKET_INFO *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
socket
446 // pop the packet from the queue
447 PACKET_DATA pkt;
448 PacketListPop(&pkt, &ClientMessages);
449 DisplayString(0, "Send to Client");
450
451 // set socket info
452 socket -> remoteNode.IPAddr.Val = BROADCAST;
453 socket -> remotePort = DHCP_CLIENT_PORT;
454 for(i = 0; i < 6u; i++){ // copy client's MAC address (and take it from CHADDR)
455     socket -> remoteNode.MACAddr.v[i] = pkt.Header.ClientMAC.v[i];
456 }
457
458 // copy header DHCP
459 UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
460 UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
461 UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
462 UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));
463 UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID)
);
464 UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.
SecondsElapsed));
465 UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
466 UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
467 UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
468 UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
469 UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
470 UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));
471
472 // the other fields are set to zero
473 for (i = 0; i < 202u; i++) {
474     UDPPut(0);
475 }
476
477 // put magic cookie 0x63538263, little endian
478 UDPPut(0x63);
479 UDPPut(0x82);
480 UDPPut(0x53);
481 UDPPut(0x63);
482
483 // put message type
484 UDPPut(DHCP_MESSAGE_TYPE);
485 UDPPut(DHCP_MESSAGE_TYPE_LEN);
486 UDPPut(pkt.MessageType);
487
488 UDPPut(DHCP_END_OPTION); // end packet
489
490 // add zero padding to ensure compatibility with old BOOTP relays that discard
491 // packets smaller than 300 octets
492 while (UDPTxCount < 300u) {
493     UDPPut(0);
494 }
495
496 UDPPflush(); // transmit
497 }
498 }
499
500 /** Schedule the first paralel component.
501 * This compoennt is responsible for getting the packets from the network and pushing
them
502 * in the right queue (depending if a packet comes from the server or from the client).
503 */
504 static void Component1() {
505     switch(comp1) {
506         // root
507         case WAITING_FOR_MESSAGE:

```

```

508     DEBUGMSG("WAITING\r\n");
509     if (prevFromClient == TRUE) {
510         compl = PUSH_SERVER_QUEUE;
511     } else {
512         GetServerPacket();
513         GetClientPacket();
514         // serve the server, but serve the client at next round
515         if (prevFromServer == TRUE && compl == PUSH_SERVER_QUEUE) {
516             prevFromServer = FALSE;
517             prevFromClient = TRUE;
518             compl = PUSH_CLIENT_QUEUE;
519         }
520     }
521     /*if (serverTurn == TRUE && GetServerPacket() == 0) {
522         DEBUGMSG("WAITING SERVER\r\n");
523         compl = PUSH_CLIENT_QUEUE;
524         //compl = SERVER_MESSAGE_T;
525     } else {
526         if (serverTurn == FALSE && GetClientPacket() == 0) {
527             DEBUGMSG("WAITING CLIENT\r\n");
528             compl = PUSH_SERVER_QUEUE;
529             //compl = CLIENT_MESSAGE_T;
530         }
531     }
532     serverTurn = (serverTurn == TRUE) ? FALSE : TRUE;*/
533     break;
534     /*case SERVER_MESSAGE_T:
535         if (N == FALSE && UDPIsGetReady(serverToClient) > 240u) {
536             compl = FROM_SERVER;
537             N = TRUE;
538         }
539         break;
540     case CLIENT_MESSAGE_T:
541         if (N == FALSE && UDPIsGetReady(clientToServer) > 240u) {
542             compl = FROM_CLIENT;
543             N = TRUE;
544         }
545         break;
546     // server branch
547     case FROM_SERVER:
548         if (GetPacket(&serverPacket, serverToClient) == 0) {
549             compl = FROM_SERVER_T;
550         } else {
551             compl = WAITING_FOR_MESSAGE;
552             break;
553         }
554     case FROM_SERVER_T:
555         N = FALSE;
556         compl = PUSH_CLIENT_QUEUE;
557         break;*/
558     case PUSH_CLIENT_QUEUE:
559         DEBUGMSG("PUSH CLIENT\r\n");
560         if (PacketListPush(&ClientMessages, &serverPacket) == 0) {
561             // cross the transition iff the push succeeded
562             compl = PUSH_CLIENT_QUEUE_T;
563         } else {
564             break;
565         }
566     case PUSH_CLIENT_QUEUE_T:
567         DEBUGMSG("PUSH CLIENT T\r\n");
568         compl = WAITING_FOR_MESSAGE;
569         break;
570     // client branch
571     /*case FROM_CLIENT:
572         if (GetPacket(&clientPacket, clientToServer) == 0) {
573             compl = FROM_CLIENT_T;
574         } else {
575             compl = WAITING_FOR_MESSAGE;
576             DisplayString(16, "ERROR");
577             break;

```

```

578     }
579     case FROM_CLIENT_T:
580         N = FALSE;
581         compl = PUSH_SERVER_QUEUE;
582         break;*/
583     case PUSH_SERVER_QUEUE:
584         DEBUGMSG("PUSH SERVER\r\n");
585         if (PacketListPush(&ServerMessages, &clientPacket) == 0) {
586             // cross the transaction iff the push succeeded
587             compl = PUSH_SERVER_QUEUE_T;
588         } else {
589             break;
590         }
591     case PUSH_SERVER_QUEUE_T:
592         DEBUGMSG("PUSH SERVER T\r\n");
593         compl = WAITING_FOR_MESSAGE;
594         break;
595 }
596 }
597
598 /**
599  * Schedule the second parallel component.
600  * This component is responsible for popping a packet coming from the client,
601  * if any, and sending it to the server. It also makes an ARP request the very
602  * first time a transmission to the server is required. After its MAC address
603  * has been resolved, it sets the 'serverKnown' flag to TRUE and stops sending
604  * ARP requests.
605  */
606 static void Component2() {
607     switch (comp2) {
608         case SERVER_QUEUE_WAITING:
609             DEBUGMSG("SERVER_QUEUE_WAITING\r\n");
610             if (!PacketListIsEmpty(&ServerMessages)) {
611                 // cross iff the queue is not empty
612                 comp2 = SERVER_QUEUE_WAITING_T;
613             } else {
614                 break;
615             }
616         case SERVER_QUEUE_WAITING_T:
617             DEBUGMSG("SERVER_QUEUE_WAITING_T\r\n");
618             comp2 = GET_SERVER_IP_ADDRESS;
619             break;
620         case GET_SERVER_IP_ADDRESS:
621             switch (comp2_2) {
622                 case IDENTIFY_SERVER:
623                     if (serverKnown == TRUE) {
624                         comp2 = IDENTIFY_SERVER_TO_TX;
625                         break; // immediately go to transmission if the server is known
626                     } else {
627                         comp2_2 = IDENTIFY_SERVER_TO_ARP;
628                     }
629                 case IDENTIFY_SERVER_TO_ARP:
630                     if (N == FALSE) {
631                         comp2_2 = SEND_ARP_REQUEST;
632                         N = TRUE;
633                     }
634                     break;
635                 case SEND_ARP_REQUEST:
636                     ARPResolve(&ServerInfo.IPAddr);
637                     DisplayString(0, "Send ARP Request");
638                     comp2_2 = SEND_ARP_REQUEST_T;
639                 case SEND_ARP_REQUEST_T:
640                     comp2_2 = PROCESS_ARP_ANSWER;
641                     N = FALSE;
642                     break;
643                 case PROCESS_ARP_ANSWER:
644                     if (ARPIsResolved(&ServerInfo.IPAddr, &ServerInfo.MACAddr) == TRUE)
645             {
646                 DisplayString(0, "MACAddr Resolved");
647                 serverKnown = TRUE;

```

```

647         comp2_2 = PROCESS_ARP_ANSWER_T;
648     } else {
649         if (N == FALSE) {
650             comp2_2 = SEND_ARP_REQUEST;
651             N = TRUE;
652         }
653         break;
654     }
655     case PROCESS_ARP_ANSWER_T:
656         if (N == FALSE) {
657             comp2_2 = SEND_ARP_REQUEST;
658             comp2 = TX_TO_SERVER;
659             N = TRUE;
660         }
661         break;
662     }
663     if (comp2 != IDENTIFY_SERVER_TO_TX) {
664         // go through the transition if the server was known
665         break;
666     }
667     case IDENTIFY_SERVER_TO_TX:
668         if (N == FALSE) {
669             N = TRUE;
670             comp2 = TX_TO_SERVER;
671         }
672         break;
673     case TX_TO_SERVER:
674         DEBUGMSG("TX SERVER\r\n");
675         SendToServer();
676         comp2 = TX_TO_SERVER_T;
677     case TX_TO_SERVER_T:
678         DEBUGMSG("TX SERTVER T\r\n");
679         N = FALSE;
680         comp2 = SERVER_QUEUE_WAITING;
681         break;
682 }
683 }
684
685 /**
686  * Schedule the third parallel component.
687  * This component is responsible for popping a packet coming from the server,
688  * if any, and sending it to the client.
689  */
690 static void Component3() {
691     switch (comp3) {
692         case CLIENT_QUEUE_WAITING:
693             DEBUGMSG("CLIENT_QUEUE_WAITING\r\n");
694             if (!PacketListIsEmpty(&ClientMessages)) {
695                 comp3 = CLIENT_QUEUE_WAITING_T;
696             } else {
697                 break;
698             }
699         case CLIENT_QUEUE_WAITING_T:
700             DEBUGMSG("CLIENT_QUEUE_WAITING_T\r\n");
701             if (N == FALSE) {
702                 comp3 = TX_TO_CLIENT;
703                 N = TRUE;
704             }
705             break;
706         case TX_TO_CLIENT:
707             DEBUGMSG("TX CLIENT\r\n");
708             SendToClient();
709             comp3 = TX_TO_CLIENT_T;
710         case TX_TO_CLIENT_T:
711             DEBUGMSG("TX CLIENT T\r\n");
712             N = FALSE;
713             comp3 = CLIENT_QUEUE_WAITING;
714             break;
715     }
716 }

```

```

717
718 /**
719  * Handle the scheduling of the DHCP relay, including the transitions
720  * among the different components. The scheduling is done iff the DHCP
721  * is actually enabled, meaning the flag 'AppConfig.Flags.bIsDHCPEnabled'
722  * is not zero.
723  */
724 static void DHCPRelayTask() {
725     /*ARPResolve(&ServerInfo.IPAddr);
726     if (ARPIsResolved(&ServerInfo.IPAddr,&ServerInfo.MACAddr) == TRUE) {
727         DisplayString(0, "Nothing");
728     } else {
729         DisplayString(0, "Done");
730     }
731 */
732     if (AppConfig.Flags.bIsDHCPEnabled) {
733         switch(currentComponent) {
734             case INIT:
735                 if (DHCPRelayInit() != -1) {
736                     UDPClose(serverToClient);
737                     UDPClose(clientToServer);
738                     currentComponent = INIT;
739                 }
740                 break;
741             case COMP1:
742                 Component1();
743                 currentComponent = COMP2;
744                 break;
745             case COMP2:
746                 Component2();
747                 currentComponent = COMP3;
748                 break;
749             case COMP3:
750                 Component3();
751                 currentComponent = COMP1;
752                 break;
753         }
754     } else {
755         DisplayString(0, "DHCP Not Enabled");
756     }
757 }
758
759 //
760 // Main application entry point.
761 //
762
763
764 #if defined(__I8CXX) || defined(__SDCC__)
765 void main(void)
766 #else
767 int main(void)
768 #endif
769 {
770     static TICK t = 0;
771     TICK nt = 0; //TICK is DWORD, thus 32 bits
772     BYTE loopctr = 0; //ML Debugging
773     WORD lloopctr = 14; //ML Debugging
774
775     static DWORD dwLastIP = 0;
776
777     // Initialize interrupts and application specific hardware
778     InitializeBoard();
779
780     // Initialize Timer0, and low priority interrupts, used as clock.
781     TickInit();
782
783     // Initialize Stack and application related variables in AppConfig.
784     InitAppConfig();
785
786     // Initialize core stack layers (MAC, ARP, TCP, UDP) and

```

```

787 // application modules (HTTP, SNMP, etc.)
788 StackInit();
789
790 #ifdef UART_DEBUG_ON
791     UARTConfig();
792 #endif
793
794 #ifdef USE_LCD
795     LCDTaskInit();
796 #endif
797
798 // Initialize and display message on the LCD
799 LCDInit();
800 DelayMs(100);
801 DisplayString (0,"OlimexA"); //first arg is start position on 32 pos LCD
802
803 /*#ifdef STACK_USE_DHCP_RELAY
804     DHCPRelayInit();
805 #endif*/
806 currentComponent = INIT;
807
808 // Now that all items are initialized, begin the co-operative
809 // multitasking loop. This infinite loop will continuously
810 // execute all stack-related tasks, as well as your own
811 // application's functions. Custom functions should be added
812 // at the end of this loop.
813
814 // Note that this is a "co-operative multi-tasking" mechanism
815 // where every task performs its tasks (whether all in one shot
816 // or part of it) and returns so that other tasks can do their
817 // job.
818 // If a task needs very long time to do its job, it must be broken
819 // down into smaller pieces so that other tasks can have CPU time.
820
821
822 while(1)
823 {
824
825     // Blink LED0 (right most one) every second.
826     nt = TickGetDiv256();
827     if((nt - t) >= (DWORD)(TICK_SECOND/1024ul))
828     {
829         t = nt;
830         LED0_IO ^= 1;
831         ClrWdt(); //Clear the watchdog
832     }
833
834     // This task performs normal stack task including checking
835     // for incoming packet, type of packet and calling
836     // appropriate stack entity to process it.
837     StackTask();
838
839     // This tasks invokes each of the core stack application tasks
840     // StackApplications(); //all except dhcp, ping and arp
841
842     // LCD task
843     #ifdef USE_LCD
844         LCDTask();
845     #endif
846
847     // Process application specific tasks here.
848     #ifdef STACK_USE_DHCP_RELAY
849         DHCPRelayTask();
850     #endif
851
852     // If the local IP address has changed (ex: due to DHCP lease change)
853     // write the new IP address to the LCD display, UART, and Announce
854     // service
855     if(dwLastIP != AppConfig.MyIPAddr.Val)
856     {

```

```

857         dwLastIP = AppConfig.MyIPAddr.Val;
858         #if defined(__SDCC__)
859             DisplayIPValue(dwLastIP); // must be a WORD: sdcc does not
860                                     // pass aggregates
861         #else
862             DisplayIPValue(AppConfig.MyIPAddr);
863         #endif
864     }
865 } //end of while(1)
866 } //end of main()
867
868 /*****
869 Function DisplayWORD:
870 writes a WORD in hexa on the position indicated by
871 pos.
872 - pos=0 -> 1st line of the LCD
873 - pos=16 -> 2nd line of the LCD
874
875 __SDCC__ only: for debugging
876 *****/
877 #if defined(__SDCC__)
878 void DisplayWORD(BYTE pos, WORD w) //WORD is a 16 bits unsigned
879 {
880     BYTE WDigit[6]; //enough for a number < 65636: 5 digits + \0
881     BYTE j;
882     BYTE LCDPos=0; //write on first line of LCD
883     unsigned radix=10; //type expected by sdcc's ultoa()
884
885     LCDPos=pos;
886     ultoa(w, WDigit, radix);
887     for(j = 0; j < strlen((char*)WDigit); j++)
888     {
889         LCDText[LCDPos++] = WDigit[j];
890     }
891     if(LCDPos < 32u)
892         LCDText[LCDPos] = 0;
893     LCDUpdate();
894 }
895 /*****
896 Function DisplayString:
897 Writes an IP address to string to the LCD display
898 starting at pos
899 *****/
900 void DisplayString(BYTE pos, char* text)
901 {
902     BYTE l= strlen(text)+1;
903     BYTE max= 32-pos;
904     strncpy((char*)&LCDText[pos], text, (l<max)?l:max );
905     LCDUpdate();
906 }
907 #endif
908
909 /*****
910 Function DisplayIPValue:
911 Writes an IP address to the LCD display
912 *****/
913
914 #if defined(__SDCC__)
915 void DisplayIPValue(DWORD IPdw) // 32 bits
916 #else
917 void DisplayIPValue(IP_ADDR IPVal)
918 #endif
919 {
920     BYTE IPDigit[4]; //enough for a number <256: 3 digits + \0
921     BYTE i;
922     BYTE j;
923     BYTE LCDPos=16; //write on second line of LCD
924     #if defined(__SDCC__)
925         unsigned int IP_field, radix=10; //type expected by sdcc's ultoa()
926     #endif

```



```

927
928     for(i = 0; i < sizeof(IP_ADDR); i++) //sizeof(IP_ADDR) is 4
929     {
930 #if defined(__SDCC__)
931         IP_field = (WORD) (IPdw>>(i*8))&0xff; //ML
932         uitoa(IP_field, IPDigit, radix); //ML
933 #else
934         uitoa((WORD)IPVal.v[i], IPDigit);
935 #endif
936
937         for(j = 0; j < strlen((char*)IPDigit); j++)
938         {
939             LCDText[LCDPos++] = IPDigit[j];
940         }
941         if(i == sizeof(IP_ADDR)-1)
942             break;
943         LCDText[LCDPos++] = '.';
944     }
945     if(LCDPos < 32u)
946         LCDText[LCDPos] = 0;
947     LCDUpdate();
948 }
949
950
951
952 /*****
953 Function:
954     static void InitializeBoard(void)
955
956 Description:
957     This routine initializes the hardware. It is a generic initialization
958     routine for many of the Microchip development boards, using definitions
959     in HardwareProfile.h to determine specific initialization.
960
961 Precondition:
962     None
963
964 Parameters:
965     None - None
966
967 Returns:
968     None
969
970 Remarks:
971     None
972 *****/
973 static void InitializeBoard(void)
974 {
975     // LEDs
976     LED0_TRIS = 0; //LED0
977     LED1_TRIS = 0; //LED1
978     LED2_TRIS = 0; //LED2
979     LED3_TRIS = 0; //LED_LCD1
980     LED4_TRIS = 0; //LED_LCD2
981     LED5_TRIS = 0; //LED5=RELAY1
982     LED6_TRIS = 0; //LED7=RELAY2
983 #if (!defined(EXPLORER_16) &&!defined(OLIMEX_MAXI)) // Pin multiplexed with
984     // a button on EXPLORER_16 and not used on OLIMEX_MAXI
985     LED7_TRIS = 0;
986 #endif
987     LED_PUT(0x00); //turn off LED0 - LED2
988     RELAY_PUT(0x00); //turn relays off to save power
989
990     // Set clock to 25 MHz
991     // The primary oscillator runs at the speed of the 25MHz external quartz
992     OSTUNE = 0x00;
993
994     // Switch to primary oscillator mode,
995     // regardless of if the config fuses tell us to start operating using
996     // the the internal RC

```

```

997 // The external clock must be running and must be 25MHz for the
998 // Ethernet module and thus this Ethernet bootloader to operate.
999 if(OSCCONbits.IDLEN) //IDLEN = 0x80; 0x02 selects the primary clock
1000 OSCCON = 0x82;
1001 else
1002 OSCCON = 0x02;
1003
1004 // Enable Interrupts
1005 RCONbits.IPEN = 1; // Enable interrupt priorities
1006 INTCONbits.GIEH = 1;
1007 INTCONbits.GIEL = 1;
1008
1009 }
1010
1011 /*****
1012 * Function: void InitAppConfig(void)
1013 *
1014 * PreCondition: MPFSInit() is already called.
1015 *
1016 * Input: None
1017 *
1018 * Output: Write/Read non-volatile config variables.
1019 *
1020 * Side Effects: None
1021 *
1022 * Overview: None
1023 *
1024 * Note: None
1025 *****/
1026
1027 static void InitAppConfig(void)
1028 {
1029 AppConfig.Flags.bIsDHCPEnabled = TRUE;
1030 AppConfig.Flags.bInConfigMode = TRUE;
1031
1032 //ML using sdcc (MPLAB has a trick to generate serial numbers)
1033 // first 3 bytes indicate manufacturer; last 3 bytes are serial number
1034 AppConfig.MyMACAddr.v[0] = 0;
1035 AppConfig.MyMACAddr.v[1] = 0x04;
1036 AppConfig.MyMACAddr.v[2] = 0xA3;
1037 AppConfig.MyMACAddr.v[3] = 0x01;
1038 AppConfig.MyMACAddr.v[4] = 0x02;
1039 AppConfig.MyMACAddr.v[5] = 0x03;
1040
1041 //ML if you want to change, see TCIPConfig.h
1042 AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 |
1043 MY_DEFAULT_IP_ADDR_BYTE2<<8ul | MY_DEFAULT_IP_ADDR_BYTE3<<16ul |
1044 MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
1045 AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
1046 AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 |
1047 MY_DEFAULT_MASK_BYTE2<<8ul | MY_DEFAULT_MASK_BYTE3<<16ul |
1048 MY_DEFAULT_MASK_BYTE4<<24ul;
1049 AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
1050 AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 |
1051 MY_DEFAULT_GATE_BYTE2<<8ul | MY_DEFAULT_GATE_BYTE3<<16ul |
1052 MY_DEFAULT_GATE_BYTE4<<24ul;
1053 AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 |
1054 MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul |
1055 MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul |
1056 MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
1057 AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 |
1058 MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul |
1059 MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul |
1060 MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;
1061 }

```