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

Tommaso Marinelli   Matteo Di Pirro

January 15, 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 lenght (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.

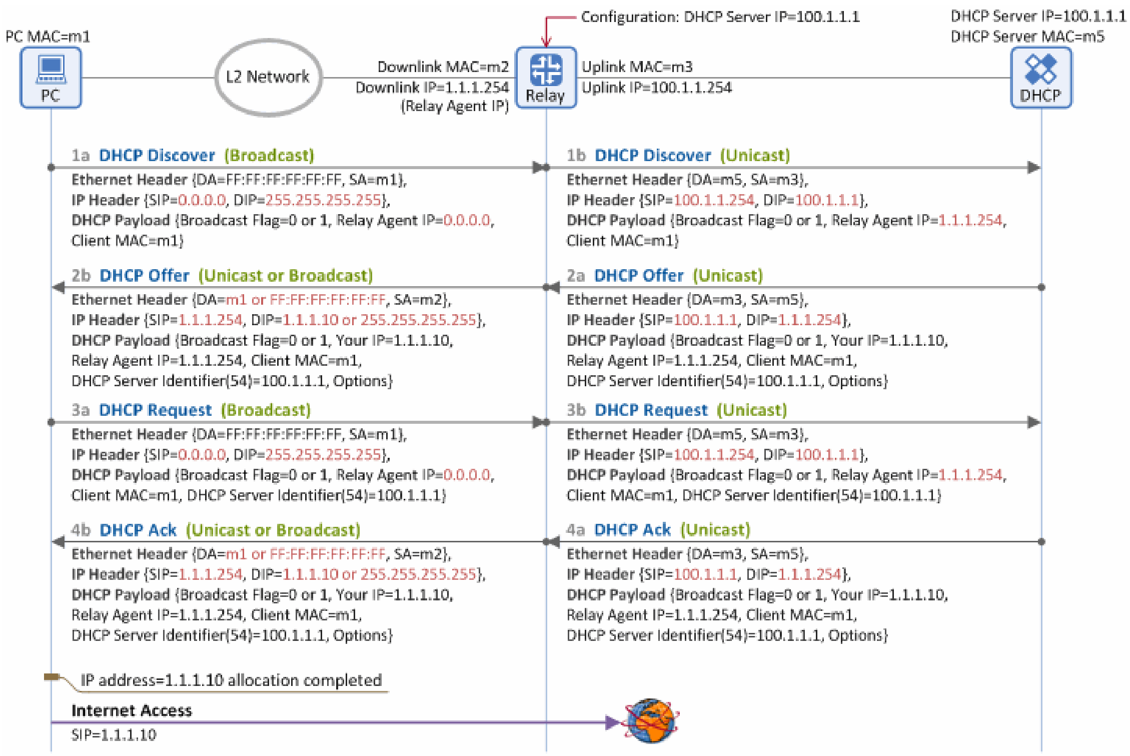


Figure 1: Example of DHCP protocol communication[1]

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

Our ASG diagram is depicted in Figure 2.

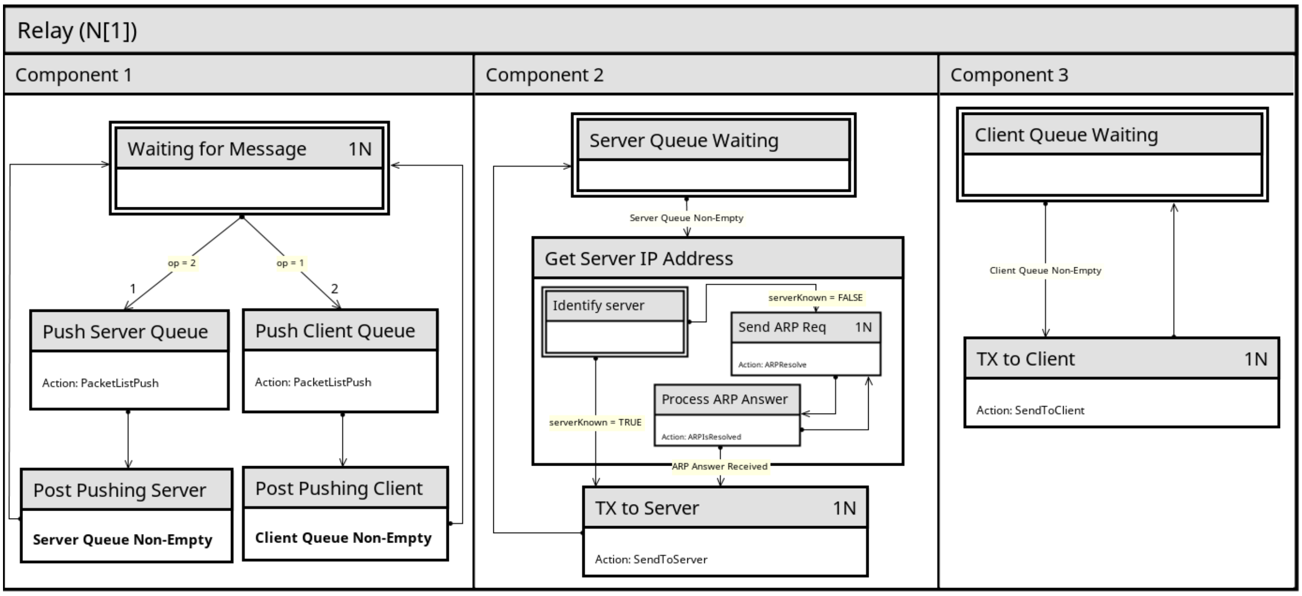


Figure 2: ASG diagram modeling a DHCP relay

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.

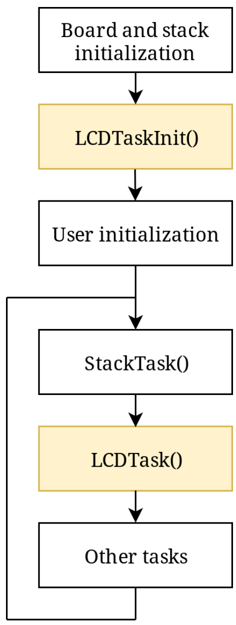
A different approach might be issuing the ARP request(s) at initialization time. In this way, transmitting a packet to the server would not require an "on the fly" request. This is a reasonable solution as soon as the DHCP server(s) is(/are) static. It is our belief that DHCP servers might change dynamically 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.

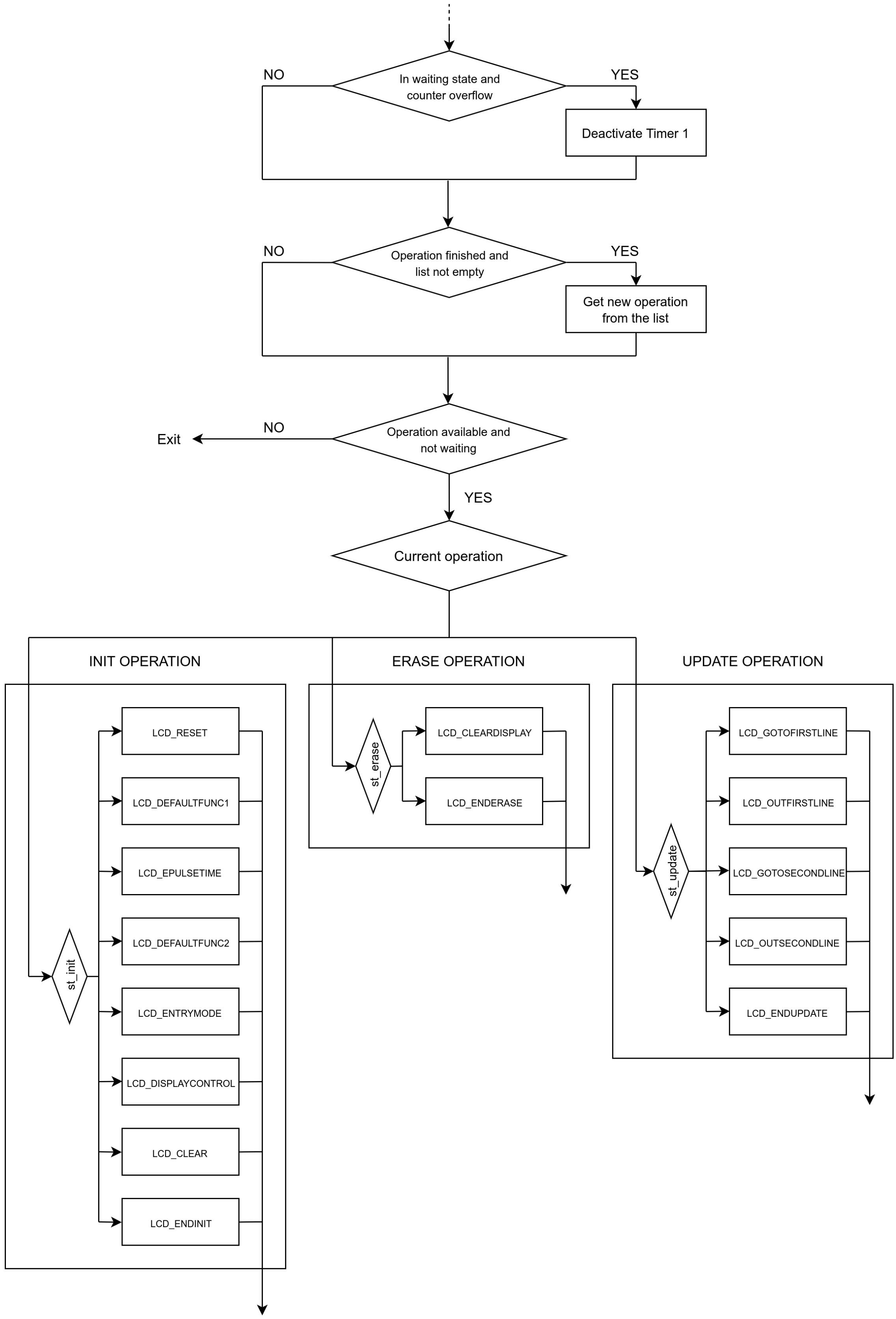


Figure 3: LCD task instruction flow chart

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 form 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 that 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.

3.3.1 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 then 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.

3.3.2 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.

```
/*********************************************************************
 * FileName:        DHCPRelay.h
 * Dependencies:    Compiler.h
 * Processor:       PIC18, PIC24F, PIC24H, dsPIC30F, dsPIC33F, PIC32
 * Compiler:        Microchip C32 v1.05 or higher
 *          Microchip C30 v3.12 or higher
 *          Microchip C18 v3.30 or higher
 *          HI-TECH PICC-18 PRO 9.63PL2 or higher
 ********************************************************************/
#ifndef _DHCPRELAY_H
#define _DHCPRELAY_H
#define STACK_USE_DHCP_RELAY
#include "GenericTypeDefs.h"
#include "TCPIP_Stack/TCPIP.h"

//#define BAUD_RATE       (19200)    // bps

#if !defined(THIS_IS_STACK_APPLICATION)
  extern BYTE AN0String[8];
#endif

//MLvoid DoUARTConfig(void);

//ML#if defined(EEPROM_CS_TRIS) || defined(SPIFLASH_CS_TRIS)
//ML  void SaveAppConfig(void);
//ML#else
  #define SaveAppConfig()
//ML#endif

//MLvoid SMTPDemo(void);
void PingDemo(void);
//MLvoid SNMPTrapDemo(void);
//MLvoid GenericTCPClient(void);
//MLvoid GenericTCPServer(void);
//void BerkeleyTCPClientDemo(void);
//void BerkeleyTCPServerDemo(void);
//void BerkeleyUDPClientDemo(void);

#ifdef STACK_USE_DHCP_RELAY
    // enum representing the current relay component on the processor
    typedef enum {
        INIT,   // init the DHCP relay parameters
        COMP1,  // listening for packets
        COMP2,  // sending to server
        COMP3   // sending to client
    } CURRENT_COMPONENT;

    typedef enum {
        WAITING_FOR_MESSAGE,     // Polling for packets
        /*SERVER_MESSAGE_T,
        CLIENT_MESSAGE_T,
        FROM_SERVER,
        FROM_SERVER_T,*/
        PUSH_SERVER_QUEUE,       // push in the server queue
        PUSH_SERVER_QUEUE_T,
        /*FROM_CLIENT,
        FROM_CLIENT_T,*/
        PUSH_CLIENT_QUEUE,       // push in the client queue
        PUSH_CLIENT_QUEUE_T,
    } COMPONENT1;

    typedef enum {
        SERVER_QUEUE_WAITING,    // wait for a packet to be sent
        SERVER_QUEUE_WAITING_T,
```

```
        GET_SERVER_IP_ADDRESS,  // issue an ARP request
        GET_SERVER_IP_ADDRESS_T,
        IDENTIFY_SERVER_TO_TX,
        TX_TO_SERVER,           // actually transmit the packet
        TX_TO_SERVER_T
    } COMPONENT2;

    typedef enum {
        CLIENT_QUEUE_WAITING,   //wait for a packet to be sent
        CLIENT_QUEUE_WAITING_T,
        TX_TO_CLIENT,           // transmit the packet
        TX_TO_CLIENT_T
    } COMPONENT3;

    typedef enum {
        IDENTIFY_SERVER,        // check if a DHCP server is known
        IDENTIFY_SERVER_TO_ARP,
        SEND_ARP_REQUEST,       // issue the ARP request
        SEND_ARP_REQUEST_T,
        PROCESS_ARP_ANSWER,     // get the answer or reissue the request
        PROCESS_ARP_ANSWER_T
    } GET_SERVER_IP_ADDRESS_COMP;

    static int DHCPRelayInit();
    static void DHCPRelayTask();
    static int GetServerPacket();
    static int GetClientPacket();
    static void SendToServer();
    static void SendToClient();
    static void Component1();
    static void Component2();
    static void Component3();
#endif

// An actual function defined in DHCPRelay.c for displaying the current IP
// address on the LCD.
#if defined(__SDCC__)
    void DisplayIPValue(DWORD IPVal);
    void DisplayString(BYTE pos, char* text);
    void DisplayWORD(BYTE pos, WORD w);
#else
    void DisplayIPValue(IP_ADDR IPVal);
#endif

#endif // _DHCPRELAY_H
```

```
/*********************************************************************
 *
 *  Main Application Entry Point for the DHCPRelay.
 *
 *********************************************************************/

/*
 * This symbol uniquely defines this file as the main entry point.
 * There should only be one such definition in the entire project,
 * and this file must define the AppConfig variable as described below.
 * The processor configuration will be included in HardwareProfile.h
 * if this symbol is defined.
 */
#define THIS_INCLUDES_THE_MAIN_FUNCTION
#define THIS_IS_STACK_APPLICATION

// define the processor we use
#define __18F97J60
// define the compiler we use
#define __SDCC__

// inlude all hardware and compiler dependent definitions
#include "Include/HardwareProfile.h"
// Include all headers for any enabled TCPIP Stack functions
```

```
#include "Include/TCPIP_Stack/TCPIP.h"

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

#if !defined(STACK_CLIENT_MODE)
    #define STACK_CLIENT_MODE
#endif

#define BROADCAST                0xFFFFFFFF // broadcast address
// server's IP address
#define SERVER_IP_ADDR_BYTE1     (192ul)
#define SERVER_IP_ADDR_BYTE2     (168ul)
#define SERVER_IP_ADDR_BYTE3     (97ul)//(10ul)
#define SERVER_IP_ADDR_BYTE4     (10ul)

// Declare AppConfig structure and some other supporting stack variables
APP_CONFIG AppConfig;
BYTE AN0String[8];

// sockets
UDP_SOCKET serverToClient;
UDP_SOCKET clientToServer;

// components needed for the cooperative scheduling
CURRENT_COMPONENT currentComponent;
COMPONENT1 comp1;
COMPONENT2 comp2;
COMPONENT3 comp3;
GET_SERVER_IP_ADDRESS_COMP comp2_2;

// queues to store packets coming from server and clients
PacketList ServerMessages;
PacketList ClientMessages;

// temporary variables used to store the packets before pushing
// them in the corresponding queue
PACKET_DATA serverPacket;
PACKET_DATA clientPacket;

IP_ADDR RequiredAddress; // IP address accepted by te server

BOOL serverTurn; // used to alternate server and client listening
//used to store whether the packet contained an accepted IP address
BOOL IPAddressNotNull;
BOOL N; // network resource
BOOL serverKnown; // TRUE once the server's MAC address has been resolved
BOOL prevFromServer;
BOOL prevFromClient;

NODE_INFO ServerInfo; // server's IP and MAC addresses

// Private helper functions.
// These may or may not be present in all applications.
static void InitAppConfig(void);
static void InitializeBoard(void);
void DisplayWORD(BYTE pos, WORD w); //write WORDs on LCD for debugging

//
// PIC18 Interrupt Service Routines
//
// NOTE: Several PICs, including the PIC18F4620 revision A3 have a RETFIE
// FAST/MOVFF bug
// The interruptlow keyword is used to work around the bug when using C18

//LowISR
#if defined(__18CXX)
    #if defined(HI_TECH_C)
        void interrupt low_priority LowISR(void)
    #elif defined(__SDCC__)
```

```
        void LowISR(void) __interrupt (2) //ML for sdcc
    #else
        #pragma interruptlow LowISR
        void LowISR(void)
    #endif
    {
  TickUpdate();
    }
    #if !defined(__SDCC__) && !defined(HI_TECH_C)
           //automatic with these compilers
        #pragma code lowVector=0x18
  void LowVector(void){_asm goto LowISR _endasm}
  #pragma code // Return to default code section
    #endif


//HighISR
    #if defined(HI_TECH_C)
        void interrupt HighISR(void)
    #elif defined(__SDCC__)
        void HighISR(void) __interrupt(1) //ML for sdcc
    #else
        #pragma interruptlow HighISR
        void HighISR(void)
    #endif
    {
      //insert here code for high level interrupt, if any
    }
    #if !defined(__SDCC__) && !defined(HI_TECH_C)
           //automatic with these compilers
  #pragma code highVector=0x8
  void HighVector(void){_asm goto HighISR _endasm}
  #pragma code // Return to default code section
    #endif

#endif

const char* message;  //pointer to message to display on LCD

/**
 * Init the relay. This function opens the sockets to the server and the client and
 * initializes the components used for the cooperative scheduling.
 * @return 0 if everything succeeded, a negative number otherwise:
 *         -) -1, if the server socket could not be open
 *         -) -2, if the client socket could not be open
 */
int DHCPRelayInit() {
    // init the components
    currentComponent    = COMP1;

    comp1                    = WAITING_FOR_MESSAGE;
    comp2                    = SERVER_QUEUE_WAITING;
    comp3                    = CLIENT_QUEUE_WAITING;
    comp2_2                  = SEND_ARP_REQUEST;

    // open the sockets
    clientToServer           = UDPOpen(DHCP_SERVER_PORT, NULL, DHCP_CLIENT_PORT);
    serverToClient           = UDPOpen(DHCP_CLIENT_PORT, NULL, DHCP_SERVER_PORT);

    if (serverToClient == INVALID_UDP_SOCKET) {
        DisplayString(0, "Invalid Server");
        return -1;
    }
    if (clientToServer == INVALID_UDP_SOCKET) {
        DisplayString(16, "Invalid Client");
        return -2;
    }

    // init the queues
    PacketListInit(&ServerMessages);
```

```
    PacketListInit(&ClientMessages);

    // init some boolean flags needed to organize the work
    serverTurn              = FALSE;

    IPAddressNotNull        = FALSE;

    N                       = FALSE;

    serverKnown             = FALSE;

    // set the server's IP address
    ServerInfo.IPAddr.Val    =
            SERVER_IP_ADDR_BYTE1 |
            SERVER_IP_ADDR_BYTE2<<8ul |
            SERVER_IP_ADDR_BYTE3<<16ul |
            SERVER_IP_ADDR_BYTE4<<24ul;

    return 0;
}

/**
 * Read a DHCP packet (if any) from a socket, and store it in the 'pkt'
 * parameter. The function reads the DHCP header and store it for future
 * use. It performs some basic checks on the hardware type (which must be
 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
 * 2u (BOOT_REPLY)) and the magic cookie.
 * @param pkt Pointer to a packet storing the packet readfrom the network
 * @param socket Socket used to read the packet (if any)
 * @return 0 If everything succeeded, a negative number otherwise:
 *         -) -1 if no packet is available on the selected socket, meaning
 *            there are less then 241 bytes in its buffer;
 *         -) -2, wrong hardware type
 *         -) -3, wrong hardware length
 *         -) -4, pkt is null or the socket is invalid
 */
static int GetPacket(PACKET_DATA* pkt, UDP_SOCKET socket) {
    // does the current socket have enough bytes ready to be read?
    if(UDPIsGetReady(socket) < 241u) {
        DEBUGMSG("NOT READY\r\n");
        return -1;
    }
    // parameters validation check
    if (pkt != NULL && socket != INVALID_UDP_SOCKET) {
        BYTE            toBeDiscarded; // used to throw away unused fields
        DWORD           magicCookie;
        BOOTP_HEADER    Header; // packet header
        BYTE            Type = 0u; // MessageType
        BYTE            Option; // used to iterate over the DHCP options
        BYTE            Len; // length of the current option
        BYTE            i; // used to add 0 paddings

        UDPGetArray((BYTE*)&Header, sizeof(Header)); // get the header

        // validate hardware interface and message type
        if (Header.HardwareType != 1u) {
            return -2;
        }

        if(Header.HardwareLen != 6u) {
            return -3;
        }

        /*
        * read and discard the following unused fields:
        * - client hardware address
        * - server host name
        * - boot filename
        */
```

```
        for(i = 0; i < 64+128+(16-sizeof(MAC_ADDR)); i++) {
            UDPGet(&toBeDiscarded);
        }

        // obtain magic cookie
        UDPGetArray((BYTE*)&magicCookie, sizeof(DWORD));
        // process options
        while (UDPGet(&Option) && Option != DHCP_END_OPTION) {
            UDPGet(&Len); // get the length
            switch (Option) {
                case DHCP_MESSAGE_TYPE:
                    UDPGet(&Type); // get the message type
                    memcpy(&(pkt -> MessageType), &Type, sizeof(BYTE)); // copy Type
                    switch (Type) {
                        case DHCP_DISCOVER_MESSAGE:
                            DisplayString(16, "DHCP Discover   ");
                            break;
                        case DHCP_REQUEST_MESSAGE:
                            DisplayString(16, "DHCP Request    ");
                            break;
                        case DHCP_OFFER_MESSAGE:
                            DisplayString(16, "DHCP Offer      ");
                            break;
                        case DHCP_ACK_MESSAGE:
                            DisplayString(16, "DHCP ACK        ");
                            break;
                    }
                    break;
                // get the accepted IP address
                case DHCP_PARAM_REQUEST_IP_ADDRESS:
                    if (Len == 4u) {
                        UDPGetArray((BYTE*)&RequiredAddress, 4);
                        IPAddressNotNull = TRUE;
                        DisplayIPValue(RequiredAddress.Val);
                    }
                    break;
            }
            // remove any unprocessed bytes
            while(Len) {
                UDPGet(&i);
                Len--;
            }
        }
        // discard the rest of the buffer (it contains the 0 padding)
        if (Option == DHCP_END_OPTION) {
            UDPDiscard();
        }

        // prepare the packet to be pushed
        memcpy(&(pkt -> Header), &Header, sizeof(BOOTP_HEADER));
        if (IPAddressNotNull == TRUE) {
            memcpy(&(pkt -> RequiredAddress), &RequiredAddress, sizeof(IP_ADDR));
            memcpy(&(pkt -> IPAddressNotNull), &IPAddressNotNull, sizeof(BOOL));
        }
        IPAddressNotNull = FALSE; // turn off flag
        DisplayIPValue(pkt -> Header.ClientIP.Val);
        return 0;
    } else {
        return -4;
    }
}

/**
 * Read a packet from the server socket, if any, and store it in
 * `serverPacket`. The function reads the DHCP header and store it for future
 * use. It performs some basic checks on the hardware type (which must be
 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
 * 2u (BOOT_REPLY)) and the magic cookie.
 * @return 0 If everything succeeded, a negative number otherwise:
```

```
 *         -) -1 if no packet is available on the selected socket, meaning
 *            there are less then 241 bytes in its buffer;
 *         -) -2, wrong hardware type
 *         -) -3, wrong hardware length
 *         -) -4, pkt is null or the socket is invalid
 */
static int GetServerPacket() {
    int res = GetPacket(&serverPacket, serverToClient);
    if (res == 0) {
        DisplayString(0, "To Push Client");
        comp1 = PUSH_CLIENT_QUEUE;
        prevFromServer = TRUE;
    }
    return res;
}

/**
 * Read a packet from the client socket, if any, and store it in
 * `clientrPacket`. The function reads the DHCP header and store it for future
 * use. It performs some basic checks on the hardware type (which must be
 * ETHERNET (== 1u)) and the hardware length (which must be == 6u). It does
 * not validate the message type (which may be both 1u (BOOT_REQUEST) and
 * 2u (BOOT_REPLY)) and the magic cookie.
 * @return 0 If everything succeeded, a negative number otherwise:
 *         -) -1 if no packet is available on the selected socket, meaning
 *            there are less then 241 bytes in its buffer;
 *         -) -2, wrong hardware type
 *         -) -3, wrong hardware length
 *         -) -4, pkt is null or the socket is invalid
 */
static int GetClientPacket() {
    int res = GetPacket(&clientPacket, clientToServer);
    if (res == 0) {
        DisplayString(0, "To Push Server");
        comp1 = PUSH_SERVER_QUEUE;
    }
    return res;
    //return GetPacket(&clientPacket, clientToServer);
}

/**
 * Send a packet to the server, taking it from ServerMessages. This function assumes
 * that queue to be not empty. The function copies the message in the socket's buffer
 * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
 * @precondition ServerMessages is not empty
 * @precondition ServerInfo contains both the server's IP and MAC address. If the latter
 * is not known, an ARP request should be made in order to get it.
 */
static void SendToServer() {
    // check if the buffer has enough space
    if (UDPIsPutReady(clientToServer) >= 300u) {
        BYTE                 i; // used to add the 0 padding
        UDP_SOCKET_INFO      *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
     socket
        // pop the packet
        PACKET_DATA          pkt;
        PacketListPop(&pkt, &ServerMessages);
        DisplayString(0, "Send to Server");

        // set socket info
        socket -> remoteNode.IPAddr.Val = ServerInfo.IPAddr.Val;
        for(i = 0; i < 6; i++) {
            socket -> remoteNode.MACAddr.v[i] = ServerInfo.MACAddr.v[i];
        }

        // copy header DHCP
        UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
        UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
        UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
        UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));
```

```
        UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID));
        UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.SecondsElapsed));
        UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
        UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
        UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
        UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
        UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));

        // the other fields are set to zero
        for (i = 0; i < 202u; i++) {
            UDPPut(0);
        }

        // put magic cookie as per RFC 1533.
        UDPPut(99);
        UDPPut(130);
        UDPPut(83);
        UDPPut(99);

        // put message type
        UDPPut(DHCP_MESSAGE_TYPE);
      UDPPut(DHCP_MESSAGE_TYPE_LEN);
      UDPPut(pkt.MessageType);

        // Option: Subnet Mask
      UDPPut(DHCP_SUBNET_MASK);
      UDPPut(sizeof(IP_ADDR));
      UDPPutArray((BYTE*)&AppConfig.MyMask, sizeof(IP_ADDR));

        // Option: Server identifier
        UDPPut(DHCP_SERVER_IDENTIFIER);
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));

        // Option: Router/Gateway address
        UDPPut(DHCP_ROUTER);
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&ServerInfo.IPAddr.Val, sizeof(IP_ADDR));

        // if there is an IP address, add it
        if (pkt.IPAddressNotNull == TRUE) {
            UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS);
            UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS_LEN);
            UDPPutArray((BYTE*)&pkt.RequiredAddress, sizeof(IP_ADDR));
            IPAddressNotNull = FALSE; // reset the global variable used as a flag
        }

        UDPPut(DHCP_END_OPTION); // end the packet

        // add zero padding to ensure compatibility with old BOOTP relays that discard
        // packets smaller that 300 octets
        while(UDPTxCount < 300u) {
            UDPPut(0);
        }

        UDPFlush(); // transmit
        //DisplayString(0, "CL to SERV");
    }
}

/**
 * Send a packet to the client, taking it from ClientMessages. This function assumes
 * that queue to be not empty. The function copies the message in the socket's buffer
 * iff there are at least 300bytes free. 300 bytes is the minimum size of a sent packet.
 * @precondition ClientMessages is not empty
 */
static void SendToClient() {
```

```
    // check if the buffer has enough space
    if (UDPIsPutReady(serverToClient) >= 300u) {
        BYTE                i; // used to add te 0 padding
        UDP_SOCKET_INFO     *socket = &UDPSocketInfo[activeUDPSocket]; //get the current
     socket
        // pop the packet from the queue
        PACKET_DATA         pkt;
        PacketListPop(&pkt, &ClientMessages);
        DisplayString(0, "Send to Client");

        // set socket info
        socket -> remoteNode.IPAddr.Val = BROADCAST;
        socket -> remotePort = DHCP_CLIENT_PORT;
        for(i = 0; i < 6u; i++){ // copy client's MAC address (and take it from CHADDR)
            socket -> remoteNode.MACAddr.v[i] = pkt.Header.ClientMAC.v[i];
        }

        // copy header DHCP
        UDPPutArray((BYTE*)&(pkt.Header.MessageType), sizeof(pkt.Header.MessageType));
        UDPPutArray((BYTE*)&(pkt.Header.HardwareType), sizeof(pkt.Header.HardwareType));
        UDPPutArray((BYTE*)&(pkt.Header.HardwareLen), sizeof(pkt.Header.HardwareLen));
        UDPPutArray((BYTE*)&(pkt.Header.Hops), sizeof(pkt.Header.Hops));
        UDPPutArray((BYTE*)&(pkt.Header.TransactionID), sizeof(pkt.Header.TransactionID)
    );
        UDPPutArray((BYTE*)&(pkt.Header.SecondsElapsed), sizeof(pkt.Header.
    SecondsElapsed));
        UDPPutArray((BYTE*)&(pkt.Header.BootpFlags), sizeof(pkt.Header.BootpFlags));
        UDPPutArray((BYTE*)&(pkt.Header.ClientIP), sizeof(pkt.Header.ClientIP));
        UDPPutArray((BYTE*)&(pkt.Header.YourIP), sizeof(pkt.Header.YourIP));
        UDPPutArray((BYTE*)&(pkt.Header.NextServerIP), sizeof(pkt.Header.NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr)); // giaddr
        UDPPutArray((BYTE*)&(pkt.Header.ClientMAC), sizeof(pkt.Header.ClientMAC));

        // the other fields are set to zero
        for (i = 0; i < 202u; i++) {
            UDPPut(0);
        }

        // put magic cookie 0x63538263, little endian
        UDPPut(0x63);
      UDPPut(0x82);
      UDPPut(0x53);
      UDPPut(0x63);

        // put message type
        UDPPut(DHCP_MESSAGE_TYPE);
      UDPPut(DHCP_MESSAGE_TYPE_LEN);
      UDPPut(pkt.MessageType);

        UDPPut(DHCP_END_OPTION); // end packet

        // add zero padding to ensure compatibility with old BOOTP relays that discard
        // packets smaller that 300 octets
        while (UDPTxCount < 300u) {
            UDPPut(0);
        }

        UDPFlush(); // transmit
    }
}

/** Schedule the first paralel component.
 * This compoennt is responsible for getting the packets from the network and pushing
    them
 * in the right queue (depending if a packet comes from the server or from the client).
 */
static void Component1() {
    switch(comp1) {
        // root
        case WAITING_FOR_MESSAGE:
```

```
            DEBUGMSG("WAITING\r\n");
            if (prevFromClient == TRUE) {
                comp1 = PUSH_SERVER_QUEUE;
            } else {
                GetServerPacket();
                GetClientPacket();
                // serve the server, but serve the client at next round
                if (prevFromServer == TRUE && comp1 == PUSH_SERVER_QUEUE) {
                    prevFromServer = FALSE;
                    prevFromClient = TRUE;
                    comp1 = PUSH_CLIENT_QUEUE;
                }
            }
            /*if (serverTurn == TRUE && GetServerPacket() == 0) {
                DEBUGMSG("WAITING SERVER\r\n");
                comp1 = PUSH_CLIENT_QUEUE;
                //comp1 = SERVER_MESSAGE_T;
            } else {
                if (serverTurn == FALSE && GetClientPacket() == 0) {
                    DEBUGMSG("WAITING CLIENT\r\n");
                    comp1 = PUSH_SERVER_QUEUE;
                    //comp1 = CLIENT_MESSAGE_T;
                }
            }
            serverTurn = (serverTurn == TRUE) ? FALSE : TRUE;*/
            break;
        /*case SERVER_MESSAGE_T:
            if (N == FALSE && UDPIsGetReady(serverToClient) > 240u) {
                comp1 = FROM_SERVER;
                N = TRUE;
            }
            break;
        case CLIENT_MESSAGE_T:
            if (N == FALSE && UDPIsGetReady(clientToServer) > 240u) {
                comp1 = FROM_CLIENT;
                N = TRUE;
            }
            break;
        // server branch
        case FROM_SERVER:
            if (GetPacket(&serverPacket, serverToClient) == 0) {
                comp1 = FROM_SERVER_T;
            } else {
                comp1 = WAITING_FOR_MESSAGE;
                break;
            }
        case FROM_SERVER_T:
            N = FALSE;
            comp1 = PUSH_CLIENT_QUEUE;
            break;*/
        case PUSH_CLIENT_QUEUE:
            DEBUGMSG("PUSH CLIENT\r\n");
            if (PacketListPush(&ClientMessages, &serverPacket) == 0) {
                // cross the transiction iff the push succeeded
                comp1 = PUSH_CLIENT_QUEUE_T;
            } else {
                break;
            }
        case PUSH_CLIENT_QUEUE_T:
            DEBUGMSG("PUSH CLIENT T\r\n");
            comp1 = WAITING_FOR_MESSAGE;
            break;
        // client branch
        /*case FROM_CLIENT:
            if (GetPacket(&clientPacket, clientToServer) == 0) {
                comp1 = FROM_CLIENT_T;
            } else {
                comp1 = WAITING_FOR_MESSAGE;
                DisplayString(16, "ERROR");
                break;
```

```
            }
        case FROM_CLIENT_T:
            N = FALSE;
            comp1 = PUSH_SERVER_QUEUE;
            break;*/
        case PUSH_SERVER_QUEUE:
            DEBUGMSG("PUSH SERVER\r\n");
            if (PacketListPush(&ServerMessages, &clientPacket) == 0) {
                // cross the transiction iff the push succeeded
                comp1 = PUSH_SERVER_QUEUE_T;
            } else {
                break;
            }
        case PUSH_SERVER_QUEUE_T:
            DEBUGMSG("PUSH SERVER T\r\n");
            comp1 = WAITING_FOR_MESSAGE;
            break;
    }
}

/**
 * Schedule the second parallel component.
 * This component is responsible for popping a packet coming from the client,
 * if any, and sending it to the server. It also makes an ARP request the very
 * first time a transmission to the server is required. After its MAC address
 * has been resolved, it sets the `serverKnown` flag to TRUE and stops sending
 * ARP requests.
 */
static void Component2() {
    switch (comp2) {
        case SERVER_QUEUE_WAITING:
            DEBUGMSG("SERVER_QUEUE_WAITING\r\n");
            if (!PacketListIsEmpty(&ServerMessages)) {
                // cross iff the queue is not empty
                comp2 = SERVER_QUEUE_WAITING_T;
            } else {
                break;
            }
        case SERVER_QUEUE_WAITING_T:
            DEBUGMSG("SERVER_QUEUE_WAITING_T\r\n");
            comp2 = GET_SERVER_IP_ADDRESS;
            break;
        case GET_SERVER_IP_ADDRESS:
            switch (comp2_2) {
                case IDENTIFY_SERVER:
                    if (serverKnown == TRUE) {
                        comp2 = IDENTIFY_SERVER_TO_TX;
                        break; // immediately go to transmission if the server is known
                    } else {
                        comp2_2 = IDENTIFY_SERVER_TO_ARP;
                    }
                case IDENTIFY_SERVER_TO_ARP:
                    if (N == FALSE) {
                        comp2_2 = SEND_ARP_REQUEST;
                        N = TRUE;
                    }
                    break;
                case SEND_ARP_REQUEST:
                    ARPResolve(&ServerInfo.IPAddr);
                    DisplayString(0, "Send ARP Request");
                    comp2_2 = SEND_ARP_REQUEST_T;
                case SEND_ARP_REQUEST_T:
                    comp2_2 = PROCESS_ARP_ANSWER;
                    N = FALSE;
                    break;
                case PROCESS_ARP_ANSWER:
                    if (ARPIsResolved(&ServerInfo.IPAddr, &ServerInfo.MACAddr) == TRUE)
    {
                        DisplayString(0, "MACAddr Resolved");
                        serverKnown = TRUE;
```

```
                        comp2_2 = PROCESS_ARP_ANSWER_T;
                    } else {
                        if (N == FALSE) {
                            comp2_2 = SEND_ARP_REQUEST;
                            N = TRUE;
                        }
                        break;
                    }
                case PROCESS_ARP_ANSWER_T:
                    if (N == FALSE) {
                        comp2_2 = SEND_ARP_REQUEST;
                        comp2 = TX_TO_SERVER;
                        N = TRUE;
                    }
                    break;
            }
            if (comp2 != IDENTIFY_SERVER_TO_TX) {
                // go through the transition if the server was known
                break;
            }
        case IDENTIFY_SERVER_TO_TX:
            if (N == FALSE) {
                N = TRUE;
                comp2 = TX_TO_SERVER;
            }
            break;
        case TX_TO_SERVER:
            DEBUGMSG("TX SERVER\r\n");
            SendToServer();
            comp2 = TX_TO_SERVER_T;
        case TX_TO_SERVER_T:
            DEBUGMSG("TX SERTVER T\r\n");
            N = FALSE;
            comp2 = SERVER_QUEUE_WAITING;
            break;
    }
}

/**
 * Schedule the third parallel component.
 * This component is responsible for popping a packet coming from the server,
 * if any, and sending it to the client.
 */
static void Component3() {
    switch (comp3) {
        case CLIENT_QUEUE_WAITING:
            DEBUGMSG("CLIENT_QUEUE_WAITING\r\n");
            if (!PacketListIsEmpty(&ClientMessages)) {
                comp3 = CLIENT_QUEUE_WAITING_T;
            } else {
                break;
            }
        case CLIENT_QUEUE_WAITING_T:
            DEBUGMSG("CLIENT_QUEUE_WAITING_T\r\n");
            if (N == FALSE) {
                comp3 = TX_TO_CLIENT;
                N = TRUE;
            }
            break;
        case TX_TO_CLIENT:
            DEBUGMSG("TX CLIENT\r\n");
            SendToClient();
            comp3 = TX_TO_CLIENT_T;
        case TX_TO_CLIENT_T:
            DEBUGMSG("TX CLIENT T\r\n");
            N = FALSE;
            comp3 = CLIENT_QUEUE_WAITING;
            break;
    }
}
```

```

/**
 * Handle the scheduling of the DHCP relay, including the transitions
 * among te different components. The scheduling is done iff the DHCP
 * is actually enabled, meaning the flag `AppConfig.Flags.bIsDHCPEnabled`
 * is not zero.
 */
static void DHCPRelayTask() {
  /*ARPResolve(&ServerInfo.IPAddr);
  if (ARPIsResolved(&ServerInfo.IPAddr,&ServerInfo.MACAddr) == TRUE) {
        DisplayString(0, "Nothing");
    } else {
        DisplayString(0, "Done");
    }
}*/
    if (AppConfig.Flags.bIsDHCPEnabled) {
        switch(currentComponent) {
            case INIT:
                if (DHCPRelayInit() == -1) {
                    UDPClose(serverToClient);
                    UDPClose(clientToServer);
                    currentComponent = INIT;
                }
                break;
            case COMP1:
                Component1();
                currentComponent = COMP2;
                break;
            case COMP2:
                Component2();
                currentComponent = COMP3;
                break;
            case COMP3:
                Component3();
                currentComponent = COMP1;
                break;
        }
    } else {
        DisplayString(0, "DHCP Not Enabled");
    }
}

//
// Main application entry point.
//


#if defined(__18CXX) || defined(__SDCC__)
void main(void)
#else
int main(void)
#endif
{
static TICK t = 0;
TICK nt = 0;  //TICK is DWORD, thus 32 bits
BYTE loopctr = 0;  //ML Debugging
WORD lloopctr = 14; //ML Debugging

static DWORD dwLastIP = 0;

    // Initialize interrupts and application specific hardware
    InitializeBoard();

    // Initialize Timer0, and low priority interrupts, used as clock.
    TickInit();

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

    // Initialize core stack layers (MAC, ARP, TCP, UDP) and
```

```
    // application modules (HTTP, SNMP, etc.)
    StackInit();

    #ifdef UART_DEBUG_ON
        UARTConfig();
    #endif

    #ifdef USE_LCD
        LCDTaskInit();
    #endif

    // Initialize and display message on the LCD
    LCDInit();
    DelayMs(100);
    DisplayString (0,"OlimexA"); //first arg is start position on 32 pos LCD

    /*#ifdef STACK_USE_DHCP_RELAY
        DHCPRelayInit();
    #endif*/
    currentComponent = INIT;

    // Now that all items are initialized, begin the co-operative
    // multitasking loop.  This infinite loop will continuously
    // execute all stack-related tasks, as well as your own
    // application's functions.  Custom functions should be added
    // at the end of this loop.

    // Note that this is a "co-operative multi-tasking" mechanism
    // where every task performs its tasks (whether all in one shot
    // or part of it) and returns so that other tasks can do their
    // job.
    // If a task needs very long time to do its job, it must be broken
    // down into smaller pieces so that other tasks can have CPU time.


    while(1)
    {

         // Blink LED0 (right most one) every second.
        nt =  TickGetDiv256();
        if((nt - t) >= (DWORD)(TICK_SECOND/1024ul))
        {
            t = nt;
            LED0_IO ^= 1;
            ClrWdt();  //Clear the watchdog
        }

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

        // This tasks invokes each of the core stack application tasks
        //        StackApplications(); //all except dhcp, ping and arp

        // LCD task
        #ifdef USE_LCD
            LCDTask();
        #endif

        // Process application specific tasks here.
        #ifdef STACK_USE_DHCP_RELAY
            DHCPRelayTask();
        #endif

        // If the local IP address has changed (ex: due to DHCP lease change)
        // write the new IP address to the LCD display, UART, and Announce
        // service
        if(dwLastIP != AppConfig.MyIPAddr.Val)
        {
```

```
            dwLastIP = AppConfig.MyIPAddr.Val;
                #if defined(__SDCC__)
                    DisplayIPValue(dwLastIP); // must be a WORD: sdcc does not
                                                // pass aggregates
                #else
                    DisplayIPValue(AppConfig.MyIPAddr);
                #endif
        }
    }//end of while(1)
}//end of main()

/**************************************************
 Function DisplayWORD:
 writes a WORD in hexa on the position indicated by
 pos.
 - pos=0 -> 1st line of the LCD
 - pos=16 -> 2nd line of the LCD

 __SDCC__ only: for debugging
**************************************************/
#if defined(__SDCC__)
void DisplayWORD(BYTE pos, WORD w) //WORD is a 16 bits unsigned
{
    BYTE WDigit[6]; //enough for a  number < 65636: 5 digits + \0
    BYTE j;
    BYTE LCDPos=0;  //write on first line of LCD
    unsigned radix=10; //type expected by sdcc's ultoa()

    LCDPos=pos;
    ultoa(w, WDigit, radix);
    for(j = 0; j < strlen((char*)WDigit); j++)
    {
       LCDText[LCDPos++] = WDigit[j];
    }
    if(LCDPos < 32u)
       LCDText[LCDPos] = 0;
    LCDUpdate();
}
/**************************************************
 Function DisplayString:
 Writes an IP address to string to the LCD display
 starting at pos
**************************************************/
void DisplayString(BYTE pos, char* text)
{
   BYTE l= strlen(text)+1;
   BYTE max= 32-pos;
   strlcpy((char*)&LCDText[pos], text,(l<max)?l:max );
   LCDUpdate();
}
#endif

/**************************************************
 Function DisplayIPValue:
 Writes an IP address to the LCD display
**************************************************/

#if defined(__SDCC__)
void DisplayIPValue(DWORD IPdw) // 32 bits
#else
void DisplayIPValue(IP_ADDR IPVal)
#endif
{
    BYTE IPDigit[4]; //enough for a number <256: 3 digits + \0
    BYTE i;
    BYTE j;
    BYTE LCDPos=16;  //write on second line of LCD
#if defined(__SDCC__)
    unsigned int IP_field, radix=10; //type expected by sdcc's uitoa()
#endif
```

```

    for(i = 0; i < sizeof(IP_ADDR); i++) //sizeof(IP_ADDR) is 4
    {
#if defined(__SDCC__)
       IP_field =(WORD)(IPdw>>(i*8))&0xff;      //ML
       uitoa(IP_field, IPDigit, radix);      //ML
#else
       uitoa((WORD)IPVal.v[i], IPDigit);
#endif

       for(j = 0; j < strlen((char*)IPDigit); j++)
       {
     LCDText[LCDPos++] = IPDigit[j];
       }
       if(i == sizeof(IP_ADDR)-1)
      break;
       LCDText[LCDPos++] = '.';

    }
    if(LCDPos < 32u)
       LCDText[LCDPos] = 0;
    LCDUpdate();
}


/*****************************************************************************
  Function:
    static void InitializeBoard(void)

  Description:
    This routine initializes the hardware.  It is a generic initialization
    routine for many of the Microchip development boards, using definitions
    in HardwareProfile.h to determine specific initialization.

  Precondition:
    None

  Parameters:
    None - None

  Returns:
    None

  Remarks:
    None
  ***************************************************************************/
static void InitializeBoard(void)
{
  // LEDs
    LED0_TRIS = 0;  //LED0
  LED1_TRIS = 0;  //LED1
  LED2_TRIS = 0;  //LED2
  LED3_TRIS = 0;  //LED_LCD1
  LED4_TRIS = 0;  //LED_LCD2
  LED5_TRIS = 0;  //LED5=RELAY1
  LED6_TRIS = 0;  //LED7=RELAY2
#if (!defined(EXPLORER_16) &&!defined(OLIMEX_MAXI))    // Pin multiplexed with
  // a button on EXPLORER_16 and not used on OLIMEX_MAXI
  LED7_TRIS = 0;
#endif
        LED_PUT(0x00);  //turn off LED0 - LED2
  RELAY_PUT(0x00); //turn relays off to save power

  // Set clock to 25 MHz
  // The primary oscillator runs at the speed of the 25MHz external quartz
    OSCTUNE = 0x00;

  // Switch to primary oscillator mode,
        // regardless of if the config fuses tell us to start operating using
        // the the internal RC
```

```
  // The external clock must be running and must be 25MHz for the
  // Ethernet module and thus this Ethernet bootloader to operate.
    if(OSCCONbits.IDLEN) //IDLEN = 0x80; 0x02 selects the primary clock
    OSCCON = 0x82;
  else
    OSCCON = 0x02;

  // Enable Interrupts
  RCONbits.IPEN = 1;    // Enable interrupt priorities
        INTCONbits.GIEH = 1;
        INTCONbits.GIEL = 1;

}

/*********************************************************************
 * Function:        void InitAppConfig(void)
 *
 * PreCondition:    MPFSInit() is already called.
 *
 * Input:           None
 *
 * Output:          Write/Read non-volatile config variables.
 *
 * Side Effects:    None
 *
 * Overview:        None
 *
 * Note:            None
 ********************************************************************/

static void InitAppConfig(void)
{
  AppConfig.Flags.bIsDHCPEnabled = TRUE;
  AppConfig.Flags.bInConfigMode = TRUE;

//ML using sdcc (MPLAB has a trick to generate serial numbers)
// first 3 bytes indicate manufacturer; last 3 bytes are serial number
  AppConfig.MyMACAddr.v[0] = 0;
  AppConfig.MyMACAddr.v[1] = 0x04;
  AppConfig.MyMACAddr.v[2] = 0xA3;
  AppConfig.MyMACAddr.v[3] = 0x01;
  AppConfig.MyMACAddr.v[4] = 0x02;
  AppConfig.MyMACAddr.v[5] = 0x03;

//ML if you want to change, see TCPIPConfig.h
  AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 |
            MY_DEFAULT_IP_ADDR_BYTE2<<8ul | MY_DEFAULT_IP_ADDR_BYTE3<<16ul |
            MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
  AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
  AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 |
            MY_DEFAULT_MASK_BYTE2<<8ul | MY_DEFAULT_MASK_BYTE3<<16ul |
            MY_DEFAULT_MASK_BYTE4<<24ul;
  AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
  AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 |
            MY_DEFAULT_GATE_BYTE2<<8ul | MY_DEFAULT_GATE_BYTE3<<16ul |
            MY_DEFAULT_GATE_BYTE4<<24ul;
  AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 |
            MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul  |
            MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul  |
            MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
  AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 |
            MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul  |
            MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul  |
            MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;
}
```