Chapter 7: Ethernet Interface Software

This chapter presents all the detailed information about the TCP/IP Stack used to manage the network activity of the microcontroller module over the Ethernet as well as the PCSCP (Process Control Studio Communication Protocol); our main application that enables clients to remotely connect to control loops via portable devices which makes use of the TCP/IP Stack implemented.

Here, the TCP/IP protocols are not discussed in details, readers who are interested in the protocols details are encouraged to read the related RFC documents.

7.1 TCP/IP Stack

A TCP/IP stack defines a set of protocols that allows network devices to connect to a specific device and exchange data on a network. These protocols, defined by RFC (request for comments), enable an embedded device to send email, serve web pages, transfer files, and provide other basic connectivity functions. Figure 3 is a simplified illustration of a user application working through the TCP stack model and illustrates how a TCP/IP stack and the MC9S7NE64 Ethernet controller fit into the system.

The TCP/IP Stack used in our project is the Freescale OpenTCP Stack developed by Viloa Systems. OpenTCP is a highly robust and portable implementation of the TCP/IP and Internet application-layer protocols intended for the implementation of TCP/IP functionality in a constrained environments. This project contains ports, optimizations, and extensions of OpenTCP for Freescale microcontrollers including the 16-bit MC9S7NE-Family.

7.1.1 OpenTCP TCP/IP Stack Structure

The OpenTCP Stack is composed of a suite of programs (C-files) that provide services to custom TCP/IP applications as the Connection Server implemented in our project as well as other standard applications that are not implemented in our project as HTTP Servers, FTP Servers, Mail Clients, … etc. It is entirely written in C which allows easy porting to any existing microcontroller platform for which a C compiler is available.

The TCP/IP Stack is implemented in a modular fashion, with all of its services creating highly abstract layers. Potential users don’t have to know the details of all the intricacies of lower TCP/IP layers specifications in order to effectively use it.

In addition to protocol modules source and header files; such as “tcp.c”, “udp.c”, “ip.c”, “tcp_ip.h”, ”ip.h”, ”ethernet.h”, some global header files reside in the stack; they are essential for the proper execution of the application as they set the necessary global variables and definitions so, these header files must be included in the main application.

Common Header files include:

- “system.h”

  This file holds #defines of Network Transmit Buffer Size, a pointer to that buffer, the master clock (Interrupt driven free-running clock) and the structure that holds all of the network-related information about the network interface.

  It also defines macros to:

  - Read/Write data (either byte or word) from the Ethernet Controller
  - Read data to a buffer in memory
  - Write data from a buffer to Ethernet Controller
  - Check if there is new data in the Ethernet controller (this macro must be invoked periodically)
  - Initialize reading the received Ethernet frame or sending an Ethernet packet from a given address in the Ethernet Controller
  - Discard Ethernet packet when not used anymore

- “datatypes.h”

  This file holds #defines of data types used in the OpenTCP sources so that recompiling for another MCU is easier even when the other MCU is using different size default values; it defines constants for BYTE, WORD, …

- “globalvariables.h”

  It holds declarations of global variables that are commonly used in other OpenTCP modules as well as OpenTCP applications in general. It consists basically of a group of externs.

- “mottypes.h”

  It holds common definitions for core registers block. MCU register definition is done in separate files, describing each peripheral register block as a datastructure.

- “debug.h”

  This file contains debug settings for OpenTCP and its modules. Debugging in this case only assumes a function that sends a null-terminated string over a serial port.

- "mc9s7ne64.h"

  This implements an IO devices mapping.

- "timers.h"

  OpenTCP timers interface file which includes timers function declarations, constants, etc.

In addition the file "address.c" contains definitions for hardware and IP addresses (MAC address, IP address, Subnet Mask, …) of the microcontroller that can be modified as needed.

7.1.2 How the TCP/IP Stack works

The OpenTCP TCP/IP Stack divides the TCP/IP Stack into multiple layers. As mentioned before, data flows upwards and downwards within stack layers sequentially. For our TCP/IP stack software, the C-code of a protocol of a specific layer resides in a separate C-source file, higher levels protocols include C-header files containing functions and variables of these lower levels in order to make use of the lower level protocol. As an example, services and API's (Application Programming Interfaces) call lower levels protocols; HTTP application calls TCP protocol which in turn calls IP protocol and so on.

However, unlike the TCP/IP reference model the OpenTCP TCP/IP stack can directly access one or more layers which are not directly below it, of course that doesn't necessary work for all applications, some services and applications data need intelligent processing in the previous adjacent layer before it can be passed to the next one.

The OpenTCP TCP/IP Stack achieves *multitasking*; it is live in the sense that the different layers not only act when a service is requested but also when events like new packet arrival or time-out occur so, layers are able to perform some timed operations asynchronously and handle different events simultaneously. As a result, user's main application can perform its main task without having to manage the stack as well.

7.1.3 Using the TCP/IP Stack

The OpenTCP Stack consists of source and header files in C which support many protocols and services like ARP, BOOTP, DHCP Client, DNS Client, HTTP Web Server, POP3 Client, SMTP Client, TCP, IP, TFTP Client, UDP, and ICMP(for Ping). The Stack doesn't implement all modules and applications normally present in the TCP/IP stack. However, they can always be implemented as a separate task or module if required, provided that proper C-header files are included.

Our Project application (The Connection Server), however, doesn't implement all protocols and services supported by the OpenTCP TCP/IP Stack. HTTP, FTP, DNS and Mail Clients (POP3, SMTP) for example are not used in our project but they can be implemented in other applications by including the corresponding header files as mentioned before. Including the appropriate header files in the project is critical for correct compilation and execution, since each of the modules comprising the stack resides in its own file.

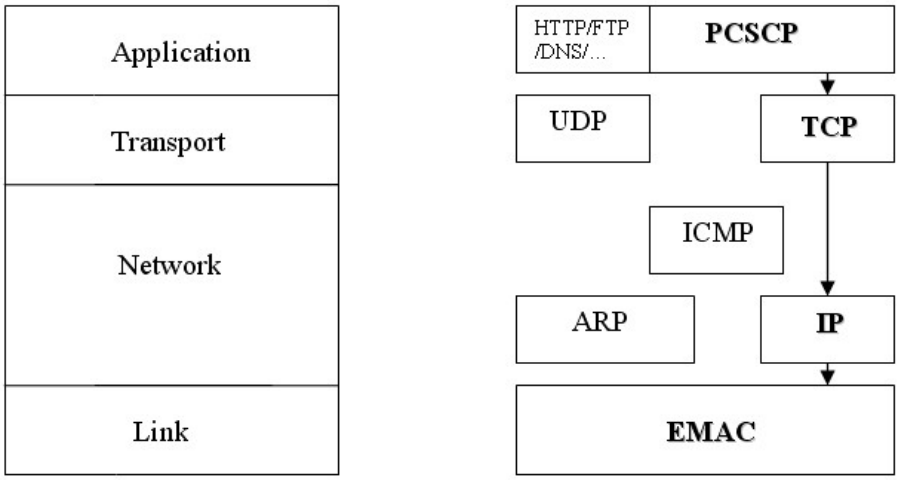

Figure 7.1 Project's utilization of the stacks.

The project executes its main application module ("main.c" source file) which contains the appropriate header files for the used modules in addition to the function calls to the relevant functions in other C-files

The Main module carries out the most important job for the proper execution of the project. In the main module, processor-dependent stuff (I/O ports, timers, …) are initialized, enabling/disabling interrupts (either globally or individually) is decided, the clock source is selected, setup of individual ports is done, bus clock, baud rate, oscillator frequency are set, relevant global variables and definitions are initialized and set.

In our project, we created a source file for our application "pcscp.c" (Process Control Studio Connection Protocol), there; we included the header files for the used modules and the common header files necessary for the proper operation of the whole project. In the main file and at the start of the main function, we put a function call after system initialization that calls a function in the pcscp.c source file; this function is responsible for initialization in addition to other tasks. These details will be thoroughly discussed in the next section.

7.1.4 Protocols Used

In our Connection server application, we used the TCP protocol of the Transport layer, that was due to the fact that our experimental tests showed that TCP transmission through wireless network is far more reliable than UDP despite all its drawbacks including overhead. UDP showed very poor performance and reliability during our tests for wireless connections.

Next, we shall briefly discuss the modules of common protocols implemented in our application, namely, modules of TCP protocol and IP protocol along with frame formats of these protocols as well as the Ethernet Frame Format.

- **TCP:**

  As mentioned before, the TCP module resides in a separate source file which contains the API functions to be called from higher-layers applications such as our main application of the project.

  Functions are available to:

  - Initialize the TCP module; initialize all sockets and corresponding TCP Control Blocks to known state.
  - Create a socket (Obtain a free socket from TCP socket pool)
  - Release a TCP socket once the application does not need the TCP socket anymore.
  - Put TCP socket to listen on a given port (provided that the socket is a server socket)
  - Initialize connection establishment towards remote IP& Port Number (provided that the socket is a client socket)
  - Send user data over TCP using given TCP socket.
  - Close connection over a given socket.
  - Get current state of the socket; whether it is free, connected, listening, closed , ...
  - Check and process the received TCP frame.
  - Create and send TCP packet, based on data supplied as function parameters and data stored in the socket's TCP Control Block.

  The corresponding header file contains function prototypes as well as constants, declarations, ... , it also includes the structure that holds header fields from the received TCP packet *"tcp_frame"* and the structure that holds various fields used to keep track of TCP socket states, settings and event listener function *"tcb"* (TCP Control Block). Instances of these two structures are created and used within functions in the source file.

TCP Frame Format

Apart from data that exist in the network buffer, TCP Frame Header fields are the members of the *"tcp_frame"* structure, they are:

sport: Source port
dport: Destination port
seqno: Sequence number
ackno: Acknowledgement number
hlen_flags : Header length and flags
window : Size of window
checksum : TCP packet checksum
urgent : Urgent pointer
buf_index : Offset from the start of network buffer

- **IP:**

  Functions in the IP source file are available to:

  - Send an IP Frame, it performs all of the necessary preparation in order to send out an IP packet.
  - Process the received Ethernet Frame by checking necessary header information.
  - Construct checksum of the IP header.
  - Check IP frame's checksum; checksum of an IP packet is calculated and compared with the received checksum.

  The corresponding header file contains function prototypes as well as constants, declarations, reserved addresses … , it also includes the structure holding information about various fields of the IPv4 header *"ip_frame"*, an nstance of this structure is created and used within functions in the IP source file.

  IP Packet Format

  IP Packet Header fields are the members of the *"ip_frame"* structure, they are:

  Vihl : Version & Header Length field
  Tos : Type Of Service
  tlen : Total Length
  id : IP Identification number
  frags : Flags & Fragment offset
  ttl : Time to live
  protocol : Protocol over IP
  checksum : Header Checksum
  sip : Source IP address
  dip : Destination IP address
  buf_index : Next offset from the start of network buffer

- **Ethernet:**

  The *"ethernet.h"* header file contains definitions for constants and buffer addresses as well as the structure that holds information about the Ethernet frames. Function to write and read from the Ethernet controller are defined in another file *"system.c"*

  Ethernet Frame Format

  Ethernet Frame Header fields are the members of the *"ethernet_frame"* structure, they are:

  destination[ETH_ADDRESS_LEN] : destination hardware address as read from the received Ethernet frame
  source[ETH_ADDRESS_LEN] : Source hardware address as read from the received Ethernet frame
  frame_size : Size of the received Ethernet frame
  protocol : Protocol field of the Ethernet header, the stack works with two protocols; IP and ARP
  buf_index : Address in the Ethernet controller's buffer where data can be read from.

7.2 PCSCP (Process Control Studio Communication Protocol)

Our main application performs the function of a Connection Server that is responsible for managing requests or commands from remote users; the microcontroller (Ethernet Controller) acts as a server which accepts connections from PCS software via Laptops or PDA's, it then decides what to do depending on the type of received packets; whether it should forward the received packet to the control loop if the packet contains values or set points to be changed in the control loop, or returns acknowledgement for another type of packets representing new connection or channels to be monitored.

These details shall be discussed thoroughly later in this section.

7.2.1 The Main Application Features

- Connection between a computer terminal (Laptop or PDA) and the micro-controller (Ethernet module) is established via a predefined handshaking scenario to handle any errors during connection stage or any illegal requests or commands.
- The operator at the computer terminal can monitor more than one process simultaneously depending on the number of free sockets available at the micro-controller.
- The microcontroller can handle multiple requests at the same time depending on free sockets available, it can also control more than one process depending on free channels available as mentioned in the control part in this documentation.

- After connection establishment, the operator at the computer terminal can just monitor the remote process or control it; by changing set-points or any other control parameters.
- The operator can terminate the connection at any time
- An idle connection is automatically terminated after a predefined period of time.
- A status message is sent to the remote computer terminal to show whether the request or command has been accepted or not, the error messages show clearly where the error lied.
- The application parameters like available sockets, IP address, … can be easily modified and updated as we need depending on the implied conditions.

7.2.2 PCSCP Steps

The protocol steps proceed as follows:

1- The Server (microcontroller) allocates and opens a server socket called *"welcome socket"*. This is the common socket used to receive connection requests from all clients (computer terminals). All clients should first send their requests to this socket before another dedicated one is opened for them.

2- If connection is approved, the server opens a dedicated connection socket for this client from the pool of available sockets, and then it sends a message to the client to inform him that connection is approved and tell him the port number of the connection socket. (Then proceed to step(4))

3- If an error had occurred (probably because no free sockets are available), an error message is reported to the client showing the error type.

4- The client then sends to the connection socket a message with the channels and controllers he wants to monitor/control.

5- If the request is approved, the server sends a message to the client to inform him with the approval. (Then proceed to step(7))

6- If an error had occurred (probably because the requested channels are already in use), an error message is reported to the client showing the error type.

7- At this stage, the client can monitor or control the channels he requested, the connection protocol forwards any incoming messages to the control phase to take the suitable control action, and forwards the reply from control phase to the client.

8- If the client issues a terminate requests, the sever closes the dedicated connection socket and acknowledges the request.

The next diagram shows a simple flowchart of the connection requests and commands sequence.

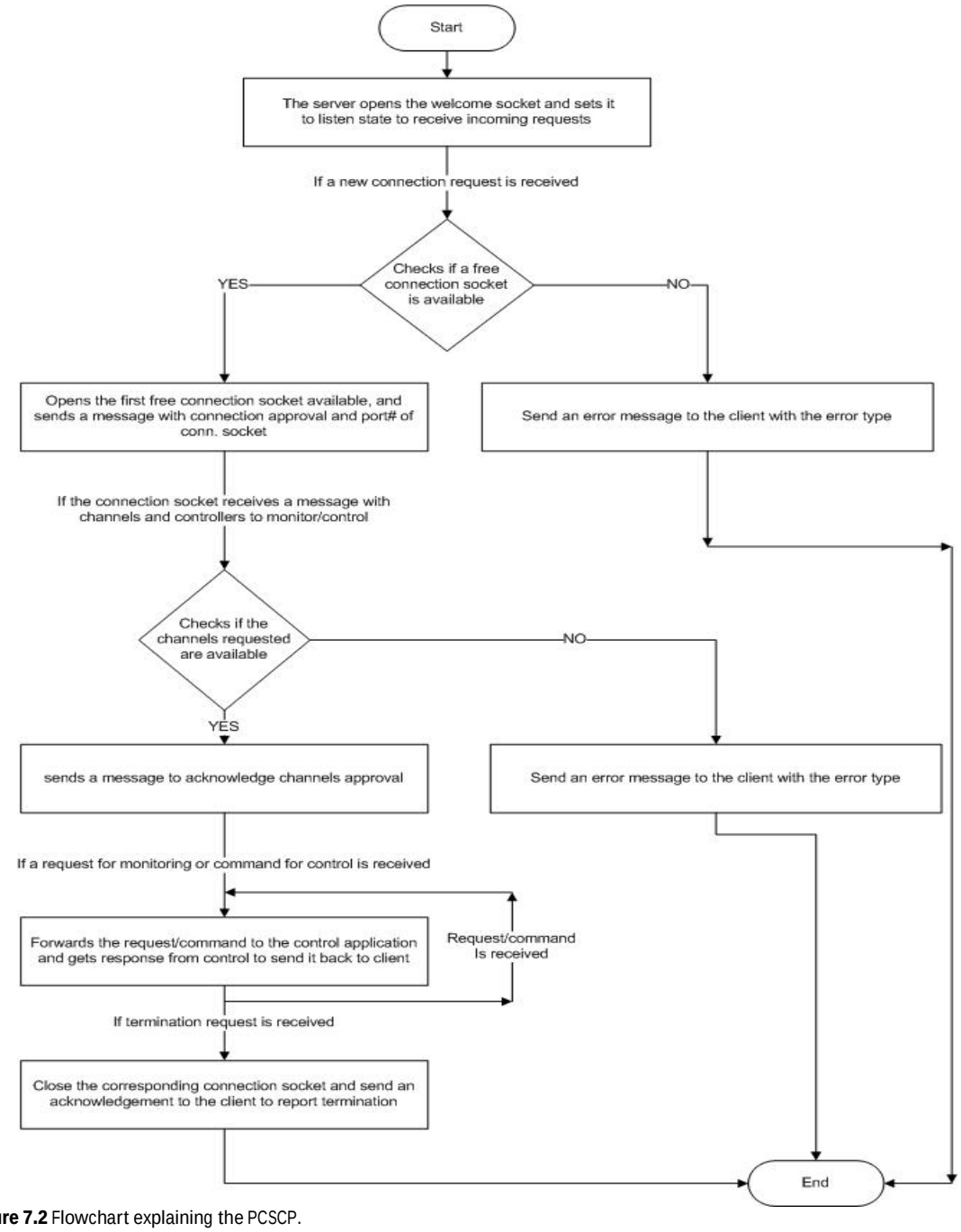

Figure 7.2 Flowchart explaining the PCSCP.

NOTE:

- Any data sent to/from the server or client is accompanied with a *"type byte"* at the start. The purpose of the type byte is to show the meaning of the data after it. The server checks that byte to know whether the data indicates new connection or channels to be monitored or anything else.
- An idle connection socket will be automatically closed after a predefined period of time.

7.2.3 PCSCP Functions

In our *"pcscp.c"* file which is our main application; after includes and definitions for type bytes, welcome socket port number and other stuff, the protocol functions are implemented.
A brief description of the functions implemented in our PCSCP application will be presented in this section.

- ✓ **TCP Application Initialize (tcp_appl_init)**
  Initializes resources needed for the TCP socket application. Here, the welcome socket is created, opened and set to *listen* state in order to receive requests from clients. It also sets the all the flags of connection sockets to zero (not reserved yet)
- ✓ **Find Free Socket (find_free_socket)**
  This function is invoked to find the first free socket from the socket pool; it searched for the first socket whose flag equals zero which indicates that it's free.
- ✓ **TCP Application Event Listener (tcp_appl_eventlistener)**
  It represents the function that will be called when the welcome socket receives a connection. It handles the request depending on the event type on the welcome socket. After regular TCP handshaking operation, packets containing data are stored in an array of bytes and sent to another function (take_action) to start/continue the handshaking scenario between the server and the client at the computer terminal.
- ✓ **TCP Application Event Listener1 (tcp_appl_eventlistener1)**
  It is similar to the previous function but it's called when the dedicated connection sockets receive requests or commands, in case of data event after TCP regular handshaking, it stores the data in an array of bytes then it calls the function (take_action1) to deal with the command or request from the client with the suitable reply.
- ✓ **TCP Take Action (take_action)**
  It consists mainly of a switch-case structure; it checks the type byte of the received data to interpret what this data means and take appropriate decisions. For example, in case of a new connection request, it tries to find a free socket using (find_free_socket) function and if it succeeds it sends back a packet of data indicating connection approval and the port number of the connection socket, else it sends a packet of data indicating an error and the error-type. And so on as explained in the previous section.
- ✓ **TCP Take Action1 (take_action1)**
  Its structure is similar to the previous function but it's for taking decisions for data from connection sockets, not the welcome socket.
- ✓ **TCP Application Send (tcp_appl_send)**
  It is invoked within other functions to send TCP message to some predefined host; it places the message to be sent in the network buffer after an offset for the TCP header, and then invokes another function in the TCP/IP stack that sends the packet through the Ethernet interface.

This concludes the description of the PCSCP protocol, our main application of the Ethernet Interface implemented in our project.

7.3 Other Software Tools Used

During different project phases, we used some software tools for compiling, debugging and testing. These software tools include:

- **Freescale CodeWarrior IDE :**

We used CodeWarrior for compilation of our C code, for building the project and for programming it to the flash memory of the microcontroller. It also has very useful utilities including the debugger that can execute the program step by step to show where the error has occurred (if there is any). All these functions are available through a very user-friendly interface.

- **Ethereal :**

Ethereal is a perfect packet sniffer, network and protocol analyzer. We used Ethereal extensively in troubleshooting during the testing phase as it captures the out-coming and in-going packets through a network interface card and analyzes the packets contents including header fields of all protocols involved, this was very useful for testing.

This concludes the Software part of the Ethernet Interface implemented in our project.