Performance Evaluation of Small TCP/IP Stack on Low Power Processor

Master Thesis’s Project

Huisheng Zhou
2012-4-19
Acknowledgements

First, I want to thank Mohammad Badawi for giving me the opportunity to have such an exciting thesis work and for the helpful advices, recommendations and support. I also want to thank my friend Nan Li for giving me a lot of help and fruitful advices in the whole thesis work. A special thank for everybody who supports and helps me finish this great thesis work.
Abstract

The uIP is an open source TCP/IP stack capable of being used with tiny 8- and 16-bit microcontrollers. Leon3 is a low power, high performance 32-bit processor. In this thesis, a port of uIP to Leon3 has implemented in order to see the performance of a minimal TCP/IP stack on a low power, high performance processor. An improved checksum calculation for uIP is implemented in order to utilize the 32-bit architecture resources. The purposes of making this analysis is to see how much the performance improvement can be achieved by using more advanced processor and more improved checksum calculation instead of the original 8- and 16-bit processors and generic 8-bit checksum calculation. A detailed performance test has performed. The test results show a detailed analysis of performance improvement in processing and energy consumption.
# Contents

List of Figures ......................................................................................................................... 6
List of Tables .............................................................................................................................. 7

Chapter 1 Introduction ............................................................................................................... 8

Chapter 2 Preliminaries ........................................................................................................... 9
  2.1 OSI model ......................................................................................................................... 9
  2.2 TCP/IP ................................................................................................................................ 10
  2.3 The uIP TCP/IP stack ........................................................................................................ 12
    2.3.1 Introduction .................................................................................................................. 12
    2.3.2 Architecture Specific Functions ................................................................................. 14
    2.3.3 Main control loop ....................................................................................................... 14
    2.3.4 Memory Management ................................................................................................. 14
    2.3.5 Protothreads .............................................................................................................. 15
    2.3.6 Protosocket library ..................................................................................................... 16
  2.4 Platform Architecture ...................................................................................................... 17
    2.4.1 LEON3 SPARC V8 Processor ...................................................................................... 18
    2.4.2 GRLIB IP Library ....................................................................................................... 19
    2.4.3 GRETH 10/100 Mbit Ethernet MAC .......................................................................... 19

Chapter 3 Device Driver and Improved Checksum Implementation ........................................... 21
  3.1 Transmitter driver design ................................................................................................. 21
    3.1.1 Transmitter DMA interface ....................................................................................... 21
    3.1.2 Driver Implementation ............................................................................................... 23
  3.2 Receiver driver design ..................................................................................................... 24
    3.2.1 Receiver DMA interface ............................................................................................ 24
    3.2.2 Driver Implementation ............................................................................................... 26
  3.3 Timer function ................................................................................................................ 27
    3.3.1 Timer unit overview ................................................................................................. 27
    3.3.2 Timer function implementation ............................................................................... 28
  3.4 Initialization .................................................................................................................... 28
    3.4.1 Initialization for uIP ................................................................................................... 29
    3.4.2 Initialization for app .................................................................................................. 29
    3.4.3 Initialization for device driver ................................................................................... 29
  3.5 Main control loop ........................................................................................................... 31
  3.7 32-bit checksum calculation ............................................................................................ 32

Chapter 4 Performance Test ................................................................................................... 35
  4.1 Test environment ............................................................................................................. 35
    4.1 ICMP performance test ................................................................................................. 35
    4.2 UDP performance test .................................................................................................... 37
      4.2.1 Packet transmission time test .................................................................................. 37
      4.2.2 Reception time test ................................................................................................ 39
      4.2.3 Reception rate test .................................................................................................. 43
    4.3 TCP performance test .................................................................................................. 45
    4.4 Energy consumption test ............................................................................................... 47
List of Figures

Figure 1 Description of OSI model .................................................................9
Figure 2 TCP Header Format ........................................................................11
Figure 3 TCP “Three-Way Handshake” Connection Establishment Procedure ...12
Figure 4 The concept of uIP........................................................................13
Figure 5 Spartan3-1500 FPGA architecture ...................................................18
Figure 6 LEON3 processor core block diagram [2] .........................................18
Figure 7 GRETH block diagram .....................................................................19
Figure 8 Block diagram of the internal structure of the GRETH.......................20
Figure 9 Transmitter descriptor register [3] ....................................................22
Figure 10 GRETH transmitter descriptor register [3] ......................................22
Figure 11 Transmitter driver .........................................................................24
Figure 12 GRETH receiver descriptor register [3] ..........................................25
Figure 13 GRETH receiver descriptor pointer register [3] ..............................25
Figure 14 Receiver driver .............................................................................27
Figure 15 Timer Control Register [3] .............................................................28
Figure 16 MAC Address MSB [3] .................................................................30
Figure 17 MAC address LSB [3] ...................................................................30
Figure 18 Memory allocation .......................................................................30
Figure 19 Main control loop .........................................................................32
Figure 20 8-bit checksum calculation flowchart ............................................33
Figure 21 32-bit checksum calculation flowchart ..........................................34
Figure 22 RTT test using ping command ......................................................36
Figure 23 Round-trip time as a function of packet size ....................................37
Figure 24 Transmission time as a function of packet size ...............................38
Figure 25 packETH packet builder ..............................................................39
Figure 26 packETH packet generator ..........................................................40
Figure 27 Reception time with 8-bit checksum calculation as a function of packet size ...41
Figure 28 Reception time with 32-bit checksum calculation as a function of packet size ...42
Figure 29 Reception rate as a function of inter packet delay with 8-bit checksum calculation ...........................................................................................................44
Figure 30 Reception rate as a function of inter packet delay with 32-bit checksum calculation ...........................................................................................................44
Figure 31 TCP throughput and RTT as a function of advertise window size ....46
Figure 32 TCP throughput vs. Simultaneous connections .............................47
Figure 33 Energy consumption for receiving 64-byte UDP packet ................48
Figure 34 Energy consumption for receiving 1024-byte UDP packet ..........48
List of Tables

Table 1 TCP/IP feature and their availability in the uIP .............................................................. 13
Table 2 Code size and RAM usage in bytes for uIP on the Atmel AVR platform [14] .............. 15
Table 3 GRETH registers ........................................................................................................... 29
Table 4 Result for RRT test using ping command ................................................................. 36
Table 5 Time delay for udp transmission ................................................................................. 38
Table 6 Reception time with 8-bit checksum calculation ......................................................... 40
Table 7 Reception time with 32-bit checksum calculation ....................................................... 41
Table 8 Performance improvement for udp reception time .................................................... 42
Table 9 Packet reception with 8-bit checksum calculation ..................................................... 43
Table 10 Packet reception with 32-bit checksum calculation .................................................. 44
Table 11 Test results for TCP throughput with different window sizes ............................... 46
Table 12 Test results for TCP throughput with simultaneous connections ......................... 47
Chapter 1 Introduction

The computer network, also referred to as just a network, consists of two or more computers, and typically other devices as well (such as printers, external hard drives, modems and routers), that are linked together so that they can communicate with each other and thereby exchange commands and share data, hardware and other resources [1]. More simply, a computer network is two or more computers linked together for the purpose of sharing information, resources and other things. Communications via computer network is down by using plenty of communication protocols. Transmission Control Protocol suite and Internet Protocol suite are two of the most well-known technologies.

With the success of Internet, now the TCP/IP protocol suite has become a global standard for communication. For embedded system, being able to run native TCP/IP makes it possible to connect the system directly to the intranet or even the global Internet. Embedded devices with full TCP/IP support will become common nodes in modern network in the future. Therefore, the resource usage and energy consumption of TCP/IP stack on embedded device will become an important issue.

In this thesis, a detailed performance analysis of minimal TCP/IP stack on low power, high performance processor has been performed. The uIP [5] TCP/IP stack which was developed by Adam Dunkels of the “Networked Embedded System” group at the Swedish Institute of Computer Science and Leon3 [2] low power, high performance processor which was developed by Aeroflex Gaisler Research are chosen. Since uIP is a minimal stack designed for 8-bit microcontroller and Leon3 is a 32-bit processor which is distributed as GRLIB IP library [3], a network device driver has been implemented and an improved 32-bit checksum calculation function has been implemented to utilize the resources.
Chapter 2 Preliminaries

2.1 OSI model

The Operating Systems Interconnection model (OSI model) is an effort to standardize networking that was started in 1977 [4] by the International Organization for Standardization (ISO), along with International Telecommunication Union Telecommunication Standardization Sector (ITU-T). It characterized and standardized the function of a communication system in terms of abstraction layers. An instance of a layer provides services to its upper layer instances while receiving services from the layer below. Figure 1 shows the general information of OSI model.

<table>
<thead>
<tr>
<th>OSI Model</th>
<th>Data unit</th>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host layers</td>
<td>Data</td>
<td>7. Application</td>
<td>Network process to application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6. Presentation</td>
<td>Data representation, encryption and decryption, convert machine dependent data to machine independent data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. Session</td>
<td>Inter host communication, managing sessions between applications</td>
</tr>
<tr>
<td></td>
<td>Segments</td>
<td>4. Transport</td>
<td>End-to-end connections, reliability and flow control</td>
</tr>
<tr>
<td>Media layers</td>
<td>Packet/Datagram</td>
<td>3. Network</td>
<td>Path determination and logical addressing</td>
</tr>
<tr>
<td></td>
<td>Frame</td>
<td>2. Data link</td>
<td>Physical addressing</td>
</tr>
<tr>
<td></td>
<td>Bit</td>
<td>1. Physical</td>
<td>Media, signal and binary transmission</td>
</tr>
</tbody>
</table>

As we can see from the figure above, OSI model has 7 abstraction layers. Each layer has its own functionality related with its upper and lower layer. Details for each layer are presented below:
Layer 1 (physical layer): This layer defines electrical and physical specification for device. It defines a relationship between a device and a transmission medium. The major functions and services are:
- Establishment and termination of a connection to a communication medium.
- Determine how to effectively share communication resources among multiple users.
- Performing modulation between representation of digital data in user equipment and corresponding signal transmission over communication channel.

Layer 2 (data link layer): This layer performs data transmission between network entities and error detection that may occur in the physical layer.

Layer 3 (network layer): This layer performs transferring variable length data sequence from one network host to another network destination. It also performs network routing and may perform fragmentation and reassembly. The network layer may divided into three sublayers:
- Subnetwork access, which considers protocols that deals with interface to network.
- Subnetwork-dependent convergence, when it is necessary to bring the level of transit network up to level of network on either side.
- Subnetwork-independent convergence, which handles transfer across multiple networks.

Layer 4 (transport layer): This layer provides transparent transfer of data between end systems, or hosts, and is responsible for error detection and recovery and flow control. It ensures complete data transfer.

Layer 5 (session layer): This layer establishes, manages and terminates connections between applications. The session layer sets up, coordinates and terminates conversations between applications at each end. It deals with sessions and connection coordination.

Layer 6 (presentation layer): This layer provides data representation (e.g. encryption and decryption) by translating from data application to network format and vice versa. The presentation layers transform data into the form that the application layers can accept.

Layer 7 (application layer): This layer mainly deals with applications and end-user processes. Everything in this layer is application-specific, e.g. communication partner, quality of service, authentication and privacy. This layer provides application services for file transfers, e-mail, and other network software services.

2.2 TCP/IP

The TCP/IP protocol is short for TCP/IP protocol suite, named for two key protocols: Transmission Control Protocol and Internet Protocol. Transmission Control Protocol accepts data from data stream, segments it into chunks, and adds a TCP header
creating a TCP segment. The TCP segment is then encapsulated into an Internet Protocol (IP) datagram. A TCP segment consists of a segment header and a data section. The TCP header contains 10 mandatory fields and 1 optional filed. Figure 2 shows a TCP header structure.

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td>Acknowledgement Number</td>
</tr>
<tr>
<td>HLEN</td>
<td>Reserved</td>
</tr>
<tr>
<td>Checksum</td>
<td>Window</td>
</tr>
<tr>
<td>Options (if any)</td>
<td>Padding</td>
</tr>
<tr>
<td>Data</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 2 TCP Header Format

- Source Port (16 bits) – identify the sending port.
- Destination Port (16 bits) – identify the receiving port.
- Sequence number (32 bits) – sequence number has a duel role:
  - If the SYN flag is set to 1, then this is an initial sequence number.
  - If the SYN flag is set to 0, then this is an accumulated sequence number of the first data byte of this packet for the current session.
- Acknowledge number (32 bits) – if the ACK flag is set then this number is the next sequence number that the receiver is expecting.
- HLEN – specify the size of the TCP header in 32-bit words. The minimum number is 5 and maximum number is 15 words.
- Reserved (3 bits) – for future use and should always set to zero.
- Flags (9 bits) – contains 9 1-bit flags.
- Window size (16 bits) – the size of the receive window, which specifies the number of bytes that receiver is willing to receive.
- Checksum (16 bits) – used for error-checking of the header and data.
- Urgent pointer (16 bits) – if URG is set, then this field indicates the last urgent data byte.
- Options – The length of this field is determined by the data offset.
- Padding – padding is used to ensure that TCP header ends and data begins on a 32 bit boundary. The padding is composed of zero.

TCP/IP processes the connection establishment by using a method called: The “Three-Way Handshake”. Figure 3 shows a normal “Three-Way Handshake” procedure.
From the figure 3, we can see that the normal process of establishing a connection between a TCP client and server involves three steps: the client sends a SYN message to server; the server receives the SYN message and sends a message that combines an ACK for the client’s SYN and contains the server’s SYN; and then the client sends an ACK for the server’s SYN when client receives the SYN+ACK from server. After this “Three-Way Handshake” a connection is established between client and server.

2.3 The uIP TCP/IP stack

In this section, a detailed introduction of uIP TCP/IP stack which is mainly summarized from [5] is introduced.

2.3.1 Introduction

The uIP is an open source TCP/IP stack capable of being used with tiny 8- and 16-bit microcontrollers [5]. It was developed by Adam Dunkels of the “Networked Embedded System” group at the Swedish Institute of Computer Science. uIP is now widely used in many embedded systems industry and has been ported to many platforms.

Traditional TCP/IP implementations have required too much resource both in terms of code size and memory usage to be useful in the 8- and 16- bit systems [5]. It is impossible to fit the full TCP/IP stack into the system with only a few numbers of
kilobytes of RAM and room for less than 100 kilobytes of code size. Thus, the uIP is designed and implemented.

The uIP removes some certain mechanisms in the interface between the application and stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections so that it has very small code size and RAM usage. Table 1 shows the TCP/IP feature and their availability in uIP.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Y/N</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP: Options</td>
<td>N</td>
<td>Dropped when received</td>
</tr>
<tr>
<td>IP: Fragment Reassembly</td>
<td>Y</td>
<td>No simultaneous reassembly</td>
</tr>
<tr>
<td>IP: Broadcast and Multicast</td>
<td>Y</td>
<td>For UDP only</td>
</tr>
<tr>
<td>ICMP</td>
<td>Y</td>
<td>Only ICMP echo messages</td>
</tr>
<tr>
<td>TCP: listening Connections</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TCP: RTT Estimation</td>
<td>Y</td>
<td>Uses periodic timer</td>
</tr>
<tr>
<td>TCP: Retransmissions</td>
<td>Y</td>
<td>Requires Application support</td>
</tr>
<tr>
<td>TCP: Flow Control</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TCP: Sliding Window</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>TCP: Urgent Data</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>TCP: Congestion Control</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>TCP: Out-of-Sequence Data</td>
<td>N</td>
<td>Left for user</td>
</tr>
<tr>
<td>Security</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 TCP/IP feature and their availability in the uIP

The uIP is mostly concerned with TCP and IP protocols and upper layer protocols will be referred to as “the application”. Lower layer protocols are often implemented in hardware or firmware and will be referred to as “the network device” that are controlled by the network device driver. Figure 4 shows the concept of uIP.
2.3.2 Architecture Specific Functions

The uIP requires some functions specifically for the architecture on which uIP is intended to run. The uIP has implemented some architecture specific functions.

- **Checksum Calculation**: uIP implemented a generic checksum function, it also leaves it open for an architecture specific implementation of the two functions: `uip_ipchksum()` and `uip_tcpchksum()`.
- **32-bit Arithmetic**: The TCP protocol uses 32-bit sequence numbers, and a TCP implementation will have to do a lot of 32-bit additions as part of the normal protocol processing. Since 32-bit arithmetic is not natively supported by many platforms for which uIP is intended to, the uIP implemented a generic 32-bit addition function called `uip_add32()`.

2.3.3 Main control loop

The uIP stack can be run either as a task in a multitasking system, or as a main program in a singletasking system. In both cases, the uIP should be run in main control loop which do two things repeatedly:

- Check if a packet has received from network.
- Check if a periodic timeout has occurred.

If the network has received a packet from network, the input handler function, `uip_input()`, should be invoked by the main control loop. If the function returns, one or more reply packets should be produced and sent out by the device driver.

Periodic timeouts are used to driver TCP mechanisms that depend on timers, such as delayed acknowledgements, retransmissions and round-time estimations. If the periodic timeout has occurred, the timer handler function, `uip_periodic()`, should be invoked by the main control loop. There is one thing we should know that the TCP/IP stack may perform retransmission when periodic timeout has happened, the network device driver should be called to send out packets that may have been produced.

2.3.4 Memory Management

From the implementation of uIP, we can see that RAM is the scarcest resources. With only a few kilobytes of RAM available, mechanism used in traditional TCP/IP stack could not applied by the uIP.

The uIP did not use explicit dynamic memory allocation. Instead, it defines a global buffer to hold packets and has a fixed table for holding connection state [5].
The buffer is large enough to hold one packet for the maximum size. When a packet arrives from network, the device driver will put the packet into the global buffer and call the main control loop. If the packet contains data, the input handler function will notify the application to handle the data. The important thing is that the data in the buffer will be overwritten by the next incoming packet, the application has to either act immediately on the data or copy the data to a secondary buffer for later processing. The data in the buffer will not be overwritten by the new incoming packet before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by network device or by the device driver.

The global buffer is not only used for incoming packets, but also for the packets that contain outgoing data. To send the data, the application passes the pointer to the data together with the data length to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and application data out on the network.

The total memory usage for uIP depends heavily on the applications. The memory configuration determines both the amount of traffic the system should be handle and the maximum amount of simultaneous connections. An application which handles email sending and lots of multiple simultaneous clients requires more RAM than the application which just run a single Telnet server. Table 2 shows the code size and RAM usage for uIP. The code has been compiled for the 8-bit Atmel AVR architecture using gcc version 3.3 and code optimization option turn on.

<table>
<thead>
<tr>
<th>Module</th>
<th>Code size</th>
<th>Static RAM</th>
<th>Dynamic RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet buffer</td>
<td>0</td>
<td>100 - 1500</td>
<td>0</td>
</tr>
<tr>
<td>IP/ICMP/TCP</td>
<td>3304</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>TCP outbound connection</td>
<td>646</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>IP fragment reassembly</td>
<td>764</td>
<td>98 – 1498</td>
<td>0</td>
</tr>
<tr>
<td>UDP</td>
<td>720</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Web server</td>
<td>994</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Checksums</td>
<td>636</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ARP</td>
<td>1324</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2 Code size and RAM usage in bytes for uIP on the Atmel AVR platform [14]

The total RAM usage depends on how large the packet buffer is, how many TCP connection slots it has and how many ARP table entries that are allocated.

2.3.5 Protothreads

Protothreads are a type of lightweight stackless threads designed for severely memory constrained systems. It provides a blocking context on top of an even-driven system, without overhead of per-thread stacks. The purpose of protothreads is to implement
sequential flow of control without complex state machines or full multi-threading. Protothread provides conditional blocking inside C functions [5].

There are two advantages of protothreads, one is that protothreads provides a sequential code structure that allows for blocking functions, the other one is protothreads do not require a separate stack.

The main feature of protothreads are:
- No machine specific code – the protothreads library is pure C.
- Does not use error-prone functions such as longjump().
- Very small RAM overhead – only two bytes per protothread.
- Can be used with or without an OS.

The protothreads API provides 4 basic operations:
- Initialization: PT_INIT(
- Execution: PT_BEGIN(
- Conditional blocking: PT_WAIT_UNTIL()
- Exit: PT_END()

On top of these, two convenience functions are built:
- Reversed condition blocking: PT_WAIT_WHILE()
- Protothread blocking: PT_WAIT_THREAD()

### 2.3.6 Protosocket library

The protosocket library is implemented to provide an interface to uIP stack that is similar with traditional BSD socket interface.

Protosocket library uses protothreads to provide sequential control flow. This makes the protosockets lightweight in terms of memory, but also makes the protosockets inherit the functional limitations of protothreads. Each protosocket lives only within a single function. Protosockets only work with TCP connections.

Programs written with protosocket library are executed in a sequential fashion and do not have to be implemented as explicit state machines.

Protosocket library defines a large number of functions, but in this thesis work I only use part of these functions which is listed below:

1. **#define PSOCK_INIT(psock, buffer, buffersize)**
   This macro initializes a protosocket and must be called before the protosocket is used. This macro also specifies the input buffer and size for the protosocket.

2. **#define PSOCK_BEGIN(psock)**
   This macro starts the protothread associated with the protosocket and must come before other protosocket call in this function.
3. **#define PSOCK_READTO(psock, c)**  
This macro will block waiting for the data and read the data into the input buffer specified in the `PSOCK_INIT()`. Data is only read until the specific character appears in the data stream.

4. **#define PSOCK_SEND_STR(psock, str)**  
This macro will sends a null-terminated string over protosocket.

5. **#define PSOCK_EXIT(psock)**  
This macro terminates the protothread of the protosocket and should always be conjunction with `PSOCK_CLOSE()`.

6. **#define PSOCK_CLOSE(psock)**  
This macro closes a protosocket and can only be called from within the protothread in which the protosocket lives.

### 2.4 Platform Architecture

In this thesis, the GR-XC3S-1500 development board [6] has been chosen for which uIP is intended. The GR-XC3S board is a compact, low-cost development board which has been developed in cooperation with Gaisler Research to enable the evaluation of LEON2 and LEON3/GRLIB processor systems.

This board incorporates a 1.5 million gate XC3S 1500 FPGA device from Xilinx Spartan3 family. The GR-XC3S-1500 board has the following features:
- LEON3 SPARC V8 processor
- 8KB Instruction cache and 4KB Data cache
- Clock generator multiply or divide 50MHz board clock
- 8 MB flash prom (8Mx8) and 64 MB SDRAM (16Mx32)
- Two RS-232 interfaces
- USB-2.0 PHY
- 10/100 Mbit/s Ethernet PHY
- Two PS/2 interfaces
- VGA video DAC and 15-pin connector
- JTAG interface for programming and debug
- 4x20 pin expansion connectors

Figure 5 shows the architecture of the platform used in this thesis. The blue marked unit is used in this thesis.
2.4.1 LEON3 SPARC V8 Processor

The LEON VHDL model, developed by Gaisler Research, implements a 32-bit processor conforming to the IEEE-1754 (SPARC V8) architecture and instruction set. It is designed for embedded applications with the following features on-chip: separate instruction and data caches, hardware multiplier and divider, interrupt controller, debug support unit with trace buffer, two 24-bit timers, two UART’s, power down function, watchdog, 16-bit I/O port, flexible memory controller and PCI interface [6]. The LEON3 is an updated design with an advanced 7-stage pipeline and multi-processor support for better performance. The processor core can be extensively configured through configuration program. Figure 6 shows an block diagram of LEON3 processor core.

![Figure 6 LEON3 processor core block diagram](image-url)


2.4.2 GRLIB IP Library

The GRLIB IP Library is an integrated set of reusable IP cores, designed for system-on-chip (SOC) development [3]. The IP cores are centered around the common on-chip bus, and use a coherent method for simulation and synthesis. The library is vendor independent, with support for different CAD tools and target technologies. The overall concept of GRLIB is to provide a standardized and vendor-independent infrastructure to deliver reusable IP cores.

2.4.3 GRETH 10/100 Mbit Ethernet MAC

Gaisler Research’s Ethernet Media Access Controller (GRETH) implements 10/100 Mbit/s Ethernet Media Access Controller (MAC) with AMBA host interface. The core implements the 802.3-2002 Ethernet standard and supports both MII and RMII PHY interfaces. Receive and transmit data is autonomously transferred between the Ethernet 802.3 Codec and AMBA AHB bus using DMA transfers. The transmitter and receiver use descriptors to make sure that multiple Ethernet packets can be received and transmitted without CPU involvement. The GRETH control register are accessed through an APB interface. Figure 7 shows the block diagram of GRETH.

![Figure 7 GRETH block diagram](image)

Figure 7 gives a general block diagram for GRETH. Figure 8 shows the detailed block diagram of internal structure of the GRETH.
The GRETH consists of 3 functional units: The DMA channels, MDIO interface and the optional Ethernet Debug Communication Link (EDCL). In this thesis, the main functionality used is DMA channels.

The main functionality of DMA channels is to transfer data between an AHB bus and an Ethernet network. There is one transmitter DMA channel and one Receiver DMA channel. They are connected with AHB bus to perform data transfer from network to host memory and vise versa.

The functionality of register unit is to control and configure the status of GRETH which is manipulated by device driver. Detailed information is introduced in the next chapter.
Chapter 3  Device Driver and Improved Checksum Implementation

The uIP TCP/IP stack is a well-implemented, portable stack with lower memory usage and smaller code size. In order to analyze the performance of uIP on Leon3, the first thing to do is to port the uIP to Leon3. Thus, a device driver for network device which is GR-XC3S-1500 development board should be implemented. Then an improved 32-bit checksum calculation method is implemented to utilize the resource usage of uIP.

The design and implementation of network device driver are mainly focused on 5 parts: Transmitter design, Receiver design, Timer design, Initialization and Main control loop. The uIP is written by pure C language, for consistency consideration, the driver and checksum method should be written in C language.

The main idea of network device driver is to use the register unit to control and configure the network device. Therefore the register unit is important. The detailed information of the register unit is introduced in the following.

3.1 Transmitter driver design

3.1.1 Transmitter DMA interface

The transmitter DMA interface is used for transmitting data on the Ethernet network. The transmission is done using descriptors located in memory. Figure 9 shows the structure of transmitter descriptor.
The descriptor is shown in the Figure 9. As can be seen, the number of bytes to be sent should be set in the length field and the address field should be pointed to the buffer area from where the packet data will be loaded. The address must be word-aligned. If the interrupt enable (IE) bit is set, an interrupt will be generated when the packet has been sent. It also requires that the transmitter interrupt bit in the control register is also set. The Wrap (WR) bit should also be set to one to make the descriptor pointer wrap to zero after this descriptor has been used. If this bit is not set, the pointer will increment by 8. After all this has been set, the descriptor enable (EN) bit should be set to enable the descriptor. The descriptor should not be touched until the enable bit has been cleared by the GRETH.

Enabling a descriptor is not enough to start the transmission. A pointer to the memory area holding the descriptors (a descriptor table) must first be set in the GRETH. This is done in the transmitter descriptor pointer register. The address must be aligned to a 1 KB boundary. Figure 10 shows the structure of transmitter descriptor register.

As shown in Figure 10, bits 31 to 10 hold the base address of descriptor table. Bits 9 to 3 form a pointer to an individual descriptor. The first descriptor address should be located at the base address and when it has been used by GRETH, the pointer field is incremented by 8 to point at the next descriptor. The pointer will automatically wrap back to zero when the next 1 KB boundary has been reached. The pointer field should never be touched when a transmission is active.
The final step to activate the transmission is to set the transmission enable bit in the control register. This tells the GRETH that there are more active descriptors in the descriptor table. This bit should always be set when new descriptors are enabled. The descriptors must always be enabled before the transmit enable bit is set.

The descriptor enable bit can be used as the indicator. When this bit has been cleared by the GRETH, it means this descriptor can be used now.

### 3.1.2 Driver Implementation

Now comes to the implementation of the transmitter driver. The first thing is to define a user-defined structure called `my_private` which contains every data type that is needed.

```c
struct my_private {
    unsigned char *tx_buf[GRETH_TXBD_NUM];
    unsigned char *rx_buf[GRETH_RXBD_NUM];
    unsigned char *tx_data;
    unsigned char *rx_data;

    u16 tx_next;
    u16 tx_last;
    u16 tx_free;
    u16 rx_current;

    struct greth_reg *regs;
    struct greth_bd *tx_bd_base;
    struct greth_bd *rx_bd_base;
    struct greth_timer *timer;
    struct greth_scalar *scalar;

    u32 tx_bd_base_addr, rx_bd_base_addr;
};
```

Then the implementation of transmitter diver - `my_tx()` should be started. The first thing is to set the transmitter descriptor pointer to point to the correct memory address where the descriptor is located. This is done by the transmitter descriptor pointer – `struct greth_bd *bdp`. The pointer will point to the descriptor table base address plus offset. After that, all the values of descriptors should be set according to the rules. Then WR bit should be checked to see whether it should wrapped. After all has been done, the descriptor enable bit and transmitter enable bit should be set. Figure 11 shows the dataflow for the transmitter driver.
3.2 Receiver driver design

3.2.1 Receiver DMA interface

The receiver DMA interface is used for receiving data from Ethernet network. The reception is done using descriptors located in the memory. Figure 12 shows a single descriptor for receiver descriptor register.
The receiver descriptor is shown in Figure 12. As can be seen from Figure 12, the address field should point to a word-aligned buffer area where the received data should be stored. The Wrap (WR) bit is also a control bit that should be set before the descriptor is enabled. It will be explained later in this section.

Enabling a descriptor is not enough to start reception. A pointer to the memory area holding the descriptors must first be set in the GRETH. This is done in the receiver descriptor pointer register. The address must be aligned to a 1 KB boundary. Figure 13 shows the structure of the receiver descriptor pointer register.

As shown in Figure 13, bits 31 to 10 hold the base address of descriptor table. Bits 9 to 3 form a pointer to an individual descriptor. The first descriptor address should be located at the base address and when it has been used by GRETH, the pointer field is incremented by 8 to point at the next descriptor. The pointer will automatically wrap back to zero when the next 1 KB boundary has been reached. The WR bit in the descriptor can be set to make the pointer wrap back before 1 KB boundary has been reached. The descriptor pointer field should never be touched when the reception is active. When everything is ready, the final step to activate reception is to set the receiver enable bit in the control register. This will make GRETH read the first descriptor and wait for incoming packet.

The GRETH will clear the descriptor enable bit to indicate a completed reception. The LENGTH field indicates the number of bytes received to this descriptor. The OE,
CE, FT, AE bits are status bits that indicate the errors for the packet received by this descriptor. If there is any errors occurred during reception, GRETH will set corresponding bits according to the conditions. All of these 4 status bits are zero means a reception without errors. They are described in the Figure 12.

Packets arriving that are smaller than the minimum Ethernet size of 64 bytes are not considered as a reception and will be discarded. The current receiver descriptor will be left untouched until the first packet arriving with an accepted size.

### 3.2.2 Driver Implementation

The driver function for receiver called `my_read()`. The driver function has a return value indicating the length of received packet. The first thing to do is to locate the descriptor in the descriptor table, this is done using descriptor pointer. Then we need to read the register value. As was described in the previous section, the GRETH will clean the descriptor enable bit to indicate a completed reception. Thus, we need to check the descriptor enable bit to see whether this is a completed reception. If yes, then we continue processing. If not, then we need to leave this descriptor untouched until the reception is finished and enable another reception. If this is a completed reception, we need to check if there is any error during reception. We check the 4 status bits: OE, CE, FT and AE to see if there is any error. If yes, we print out some debug information and enable another reception. If not, we set return value which is the length of received packet and copy packet to the global buffer. After a successful reception, descriptor enable bit must be set to enable the next descriptor for receiving packet. If this is the last descriptor in the descriptor table, then WR bit must be set to wrap the descriptor pointer back to zero. After all descriptor’s bits are set correctly, the receiver enable bit in the control register must be set to activate the reception. Descriptor table pointer then will point to the next descriptor, preparing for next incoming packet. The receiver will return the packet length. Figure 14 shows the flowchart of the receiver driver.
3.3 Timer function

3.3.1 Timer unit overview

The development board has its own timer unit, so we do not need to implement the timer unit. This timer unit mainly consists of one prescaler and four decrementing timers. The timer unit registers are accessed through the APB bus (See Figure 5).
The operation of the timer is controlled through its control register. A timer is enabled by setting its enable bit (EN) in the control register. The timer value is then decremented on each prescaler tick which is controlled by timer oscillator. The frequency of timer oscillator is 25MHZ. When a timer underflows, it will automatically be reloaded with the value of corresponding timer reloading register, if the restart bit (RS) is set, otherwise it will stop at -1 and reset the enable bit.

Each timer can be reloaded with the value in its reloading register at any time by writing a ‘1’ to the load bit (LD) in the control register.

In this thesis, only one timer is used. Figure 15 shows the timer control register.

<table>
<thead>
<tr>
<th>31</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DH</td>
<td>CH</td>
<td>IP</td>
<td>IE</td>
<td>LD</td>
<td>RS</td>
<td>EN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

31 – 7: Reserved.
6: DH – Debug Halt. State of timer when DF=0. Read only. 0 = Active, 1 = Frozen.
5: CH – Chain with preceding timer.
4: IP – Interrupt pending. 0 = Interrupt not pending, 1 = Interrupt pending.
3: IE – Interrupt Enable. 0 = Interrupt disabled, 1 = Interrupt enabled.
2: LD – Load Timer. Set to 1 to reset the timer value.
1: RS – Restart. Set to 1 to restart the timer.
0: EN – Timer Enable. 0 = Disable, 1 = Enable.

Figure 15 Timer Control Register [3]

### 3.3.2 Timer function implementation

The implementation of timer function is simple. Since the timer is actually a decrementing timer, the only thing to do is to set the enable bit (EN) and get the return value of the timer as the time stamp. The function `rt_timval()` is implemented to do that. The timer also needs to be reset at some circumstances, a reset function was also implemented called `reset_timer()`.

### 3.4 Initialization

With the transmitter and receiver driver has been implemented, the next thing to be done is to implement the initialization part. For this porting, there are three initializations need to implement: initialization for uIP stacks, initialization for Apps and initialization for device driver.
3.4.1 Initialization for uIP

The initialization for uIP – `uip_init()` is actually implemented by the uIP stack itself. The main idea of this initialization is to clear all the listening ports and connections. This function should be called before you want to use uIP stack.

3.4.2 Initialization for app

The initializations for apps are implemented by users. In this thesis, two simple apps have been implemented – one is for TCP processing and the other is for UDP processing. Therefore, two initializations must be constructed to make these two apps work properly. The main concepts of these two initializations are as follows:

- For TCP app, the proto-socket is used to implement to app. The app will watch the flags that uIP uses to establish or close the connections. Therefore the initialization is simple. Since this app is running on the board as a TCP client server, listening to the specific connection port will be the initialization job. The `uip_tcp_app_init()` function is implemented to listen to some specific connection ports using uIP self-defined function `uip_listen()`.

- For UDP app, the protothreads is used here since the proto-socket is only used for TCP. The app can be run as a server or a client. The initialization is not the same with TCP. Instead of listening port, the port binding is used here. This initialization will bind server port to client port so that it fixes the destination and source port numbers in UDP header. The `uip_udp_app_init()` is implemented to bind the user-defined server port and client port together.

3.4.3 Initialization for device driver

There are mainly two things for the device driver initialization: allocate the memory for transmitter descriptors and receiver descriptors and set the initial bits for GRETH registers. Table 3 shows the GRETH registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>APB Address offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control register</td>
<td>0x0</td>
</tr>
<tr>
<td>Status/Interrupt-source register</td>
<td>0x4</td>
</tr>
<tr>
<td>MAC Address MSB</td>
<td>0x8</td>
</tr>
<tr>
<td>MAC Address LSB</td>
<td>0xC</td>
</tr>
<tr>
<td>MDIO Control/Status</td>
<td>0x10</td>
</tr>
<tr>
<td>Transmit descriptor pointer</td>
<td>0x14</td>
</tr>
<tr>
<td>Receiver descriptor pointer</td>
<td>0x18</td>
</tr>
<tr>
<td>EDCL IP</td>
<td>0x1C</td>
</tr>
</tbody>
</table>

Table 3 GRETH registers
In this thesis, MDIO and EDCL registers are not used. Control register is used to enable transmission and reception, status register is used to detect error. They will not be discussed here. The transmit descriptor pointer, receiver descriptor pointer, MAC address MSB and MAC address LSB register are set in the initialization. The transmit descriptor pointer and receiver descriptor pointer are introduced in previous sections (see chapter 3.1.1 and 3.2.1), they will not be discussed here. Figure 16 and Figure 17 shows MAC address MSB and MAC address LSB registers.

31 16 15 0
RESERVED Bit 47 downto 32 of the MAC Address

Figure 16 MAC Address MSB [3]

31 0
Bit 31 downto 0 of the MAC Address

Figure 17 MAC address LSB [3]

The MAC address must be set according to the register format before using the device. Also the memory allocation must be done by using malloc() function. Figure 18 shows the memory allocation.

As can see from transmitter and receiver descriptor pointer (see chapter 3.1.1 and 3.2.1), 128 descriptors for both transmitter and receiver are allocated. Since each descriptor has a pointer point to a buffer area from where the packet will be loaded, the buffer area must be assigned to some memory space. The buffer area is assigned to 1520 byte according to the maximum packet size for Ethernet.
### 3.5 Main control loop

The main control loop for uIP mainly does two things repeatedly:

- Check if a packet has received from network.
- Check if a periodic timeout has occurred.

uIP introduce a global buffer to put received packets or . If the network has received an IP packet from network, the input handler function, `uip_input()`, should be invoked by the main control loop. After the function returns, the global buffer `uip_buf[]` should be checked to see if there are packets to be sent. If yes, device driver for transmitter should be invoked to send out the packets. If the network has received an arp packet, the arp packet handler function, `uip_arp_in()`, should be invoked by the main control loop. After the function returns, the device driver should be call to send out the packets that may have produced.

Periodic timeouts are used to driver TCP mechanisms that depend on timers, such as delayed acknowledgements, retransmissions and round-time estimations. If the periodic timeout has occurred, the timer handler function, `uip_periodic()`, should be invoked by the main control loop. Because the uIP TCP/IP stack may perform retransmissions when dealing with a timer event, the network device driver should call to send out the packets that may have produced. Figure 19 shows the main control loop.
3.7 32-bit checksum calculation

In a small embedded system, the processing overhead is dominated by the copying of packet data from network device to host memory, and checksum calculation [8]. Apart from the checksum calculation and copying data, the processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application [8]. Since the delay for copying data is almost the same for 8-bit architecture and 32-bit architecture, the checksum calculation is the most important factor of the performance.

Since the performance of uIP is mainly dominated by checksum calculation, the efficiency of checksum calculation is important. The uIP has implemented a generic 8-bit checksum calculation method. Since Leon3 is a 32-bit processor, 8-bit checksum calculation is not efficient for 32-bit architecture. Therefore, a 32-bit checksum calculation method is implemented in order to utilize the 32-bit architecture.

In order to have a clear idea on how to implement the 32-bit checksum, the mechanism of 8-bit checksum must be clear. Figure 20 shows how the generic 8-bit

---

Figure 19 Main control loop

---
checksum works.

As can be seen from Figure 20, the main idea for 8-bit checksum is to add the packet byte by byte, store result in a 16-bit variable and check if there is a carry in every addition. If yes, the carry must be added to the result. This method is adjusted to 8-bit architectures since they do not have enough bit width to handle more than 8-bit data.

However, for 32-bit architecture, the bit width is much larger than 8-bit architecture’s, therefore more data can be handled in each addition. The main concept for 32-bit architecture is to define a 32-bit variable that hold the addition results and do addition 16 bits by 16 bits. Since the result is hold in a 32-bit variable, the carry for each addition is not checked until the addition is finished. Then the higher 16 bits and lower 16 bits must be added to get the final checksum. Figure 21 shows the flowchart of 32-bit checksum calculation.
Compared with 8-bit checksum calculation, 32-bit checksum calculation has some advantage:

- It adds 16 bits instead of 8 bits (byte).
- It uses 32-bit variables to store the result for each addition, so it has no need to check carry for each addition. It just needs to do the carry check for the last addition.
- Utilize the resource of the 32-bit architecture.
Chapter 4 Performance Test

This chapter introduces different tests for calculating the performance of uIP. Since uIP support UDP, TCP and very simple ICMP protocol processing, there are mainly 4 tests for uIP are considered: ICMP performance test, UDP performance test, TCP performance test and power consumption test.

4.1 Test environment

For performance test, an Intel(R) Core(TM)2 Duo CPU computer with Linux ubuntu OS, a 10/100M switch and the gr-xc3s-1500 development board are connected to set the whole test environment. The address for the computer is set to 192.168.0.55 and 192.168.0.53 for the development board.

4.1 ICMP performance test

The ICMP implementation in uIP is very simple as it is restricted to only implement ICMP echo message [5]. Therefore, the Round-Trip Time (RTT) for ICMP is tested. The Round-Trip Time (RTT) is the length of time it takes for one data packet to be sent plus the length of time it takes for an acknowledgement of that packet to be received [7]. The RTT is a very important factor for network performance. In the context of computer networking, the RTT is also known as ping time. Therefore, in this test, ping command is used to test the RTT for ICMP.

There are 3 options for ping command used:

- **-s packet size**: Specifies the number of data bytes to be sent
- **-c count**: Stop after sending count ECHO_REQUEST packets.
- **-i interval**: Wait interval seconds between sending each packet.
Figure 22 RTT test using ping command

Figure 22 shows an example of RTT test with ping command. There are 20 packets with 100 byte for each packet and inter packet delay is 1ms. The statistics shows the results for this ping test.

In order to make this test more accurate and convincing, more test patterns have been used and tested. The results for RTT with different packet size and different inter packet delay are shown in the Table 4.

<table>
<thead>
<tr>
<th>packet size</th>
<th>500 us</th>
<th>10 ms</th>
<th>100 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 byte</td>
<td>0.097</td>
<td>0.102</td>
<td>0.125</td>
</tr>
<tr>
<td>100 byte</td>
<td>0.121</td>
<td>0.124</td>
<td>0.145</td>
</tr>
<tr>
<td>128 byte</td>
<td>0.132</td>
<td>0.137</td>
<td>0.154</td>
</tr>
<tr>
<td>200 byte</td>
<td>0.167</td>
<td>0.172</td>
<td>0.188</td>
</tr>
<tr>
<td>256 byte</td>
<td>0.193</td>
<td>0.197</td>
<td>0.214</td>
</tr>
<tr>
<td>300 byte</td>
<td>0.215</td>
<td>0.218</td>
<td>0.235</td>
</tr>
<tr>
<td>400 byte</td>
<td>0.260</td>
<td>0.263</td>
<td>0.282</td>
</tr>
<tr>
<td>500 byte</td>
<td>0.304</td>
<td>0.312</td>
<td>0.330</td>
</tr>
<tr>
<td>600 byte</td>
<td>0.351</td>
<td>0.361</td>
<td>0.376</td>
</tr>
<tr>
<td>700 byte</td>
<td>0.400</td>
<td>0.410</td>
<td>0.426</td>
</tr>
<tr>
<td>800 byte</td>
<td>0.446</td>
<td>0.454</td>
<td>0.472</td>
</tr>
<tr>
<td>900 byte</td>
<td>0.495</td>
<td>0.504</td>
<td>0.520</td>
</tr>
<tr>
<td>1024 byte</td>
<td>0.555</td>
<td>0.566</td>
<td>0.580</td>
</tr>
</tbody>
</table>

Table 4 Result for RTT test using ping command
Figure 23 Round-trip time as a function of packet size

Table 4 shows the test result for RTT with different packet size and different inter packet delay. The results are in terms of millisecond. Figure 23 shows the RTT as a function of packet size based on the results shown in the Table 4. We can see from the figure that the RTT is linearly with packet size and is very high performance. The reason for high performance is that uIP only implemented a very simple echo process. If uIP receive an echo request, it just simply reset some header bits and sends echo response back. Therefore the RTT looks like this.

### 4.2 UDP performance test

The uIP supports UDP protocol processing, therefore the performance for UDP processing is important. In this test, the udp packet transmission time, udp reception rate and udp reception time are tested.

#### 4.2.1 Packet transmission time test.

In this test, the transmission time for one udp packet is tested to see how much performance the uIP can be achieved after porting to Leon3. The transmission program is implemented and downloaded to the board as a transmitter.

For this test and the all the tests after, the performance for application itself are not considered. Therefore the application should keep as simple as possible. In this test, uIP do not invoke the application, so the application part is not implemented. The transmission can be divided into 2 parts: driver setting and uIP process. As have mentioned before, the uIP have a global that hold not only the incoming packets but
also the packets that are to be transmitted. Therefore, the packets should be put into the global buffer by using `memcpy`. The time delay for each part is tested and shown in Table 5.

<table>
<thead>
<tr>
<th>Packet Size</th>
<th>64B</th>
<th>128B</th>
<th>256B</th>
<th>512B</th>
<th>1KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>uIP process time (us)</td>
<td>11.44</td>
<td>16.8</td>
<td>29.08</td>
<td>52.64</td>
<td>111.4</td>
</tr>
<tr>
<td>Global buffer memcpy (us)</td>
<td>1.36</td>
<td>2.4</td>
<td>3.8</td>
<td>7.68</td>
<td>13.12</td>
</tr>
<tr>
<td>Setting driver (us)</td>
<td>0.92</td>
<td>0.84</td>
<td>0.8</td>
<td>0.92</td>
<td>1.08</td>
</tr>
<tr>
<td>Total delay (us)</td>
<td>13.72</td>
<td>20.04</td>
<td>33.68</td>
<td>61.24</td>
<td>125.6</td>
</tr>
</tbody>
</table>

Table 5 Time delay for udp transmission

Figure 24 Transmission time as a function of packet size

Table 5 shows the time delay for each part of the whole transmission process. The results are in terms of microseconds. Figure 24 gives a clear image of the transmission time as a function of packet size. As can be seen from the figure, time delays for driver setting are almost the same because some register bits are just need to be set. The time delay for `memcpy` and uIP process is linearly with packet size. The uIP process takes most of the time in a transmission because most of the processes are done within this part especially for the checksum calculation.

For this test, only generic checksum is used and tested because the behavior of uIP process for transmission is similar to the reception. Therefore, the difference of the performance for the two checksum can be presented either in transmission or reception. There is no need to test both transmission and reception to see the difference. Therefore, only one checksum is used here and the difference of two checksum usage is tested in reception part.
4.2.2 Reception time test

In this test, the reception time for one udp packet with generic 8-bit checksum and improved 32-bit checksum is tested to see how much performance the uIP can be achieved after porting to Leon3.

Since uIP is a TCP/IP stack for 8-bits architecture and Leon3 is a 32-bit microcontroller, there should be a performance improvement due to improved checksum. As have mentioned above, the performance can be dominated by checksum calculation and copying, the time delays for these two parts are important. The uIP have implemented a generic checksum calculation based on 8-bit architectures which will be used in this test. Also the improved 32-bit checksum calculation mechanism based on 32-bit architecture which will also be used to see how much difference it will have compared with 8-bit generic checksum calculation. The copying data from network device to host memory will make no difference for 8-bit checksum and 32-bit checksum.

In this test, the packETH 1.7 [9] will be used as a transmitter and uIP will be run as a receiver on the board. Figure 25 and 26 shows how packETH works.

![Figure 25 packETH packet builder](image-url)
The udp packet can be build in packETH packet builder. In packETH generator, the number of packets and inter packet delay can be set. The uIP itself do not handle the data in the udp packet but just pass the data to application. As have described in the beginning of the performance test, the performance of application is not considered. Therefore the application should be as simple as possible. In this test, the application only copy the data from the global buffer to a secondary buffer which is the solution described in section 2.3.4.

Firstly, the reception time with 8-bit checksum calculation is tested. Table 6 and Figure 27 give the detailed results about the reception time.

<table>
<thead>
<tr>
<th></th>
<th>64B</th>
<th>128B</th>
<th>256Byte</th>
<th>512Byte</th>
<th>1KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global buffer memcpy(us)</td>
<td>3.04</td>
<td>5</td>
<td>8.8</td>
<td>16.56</td>
<td>32.04</td>
</tr>
<tr>
<td>Driver setting(us)</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>uIP process memcpy(us)</td>
<td>1.6</td>
<td>2.56</td>
<td>3.6</td>
<td>7.68</td>
<td>12.8</td>
</tr>
<tr>
<td>ip checksum delay(us)</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td>udp checksum delay(us)</td>
<td>3.48</td>
<td>8.16</td>
<td>18</td>
<td>38.56</td>
<td>78.56</td>
</tr>
<tr>
<td>uIP process delay(us)</td>
<td>4.16</td>
<td>4.08</td>
<td>4.2</td>
<td>4.08</td>
<td>4.12</td>
</tr>
<tr>
<td>Total reception delay(us)</td>
<td>14.96</td>
<td>22.48</td>
<td>37.28</td>
<td>69.56</td>
<td>130.2</td>
</tr>
</tbody>
</table>

Table 6 Reception time with 8-bit checksum calculation
As can be seen from above, the whole reception process is divided into 5 parts: driver setting, global buffer memcpy delay, checksum calculation, uIP process delay and the memcpy delay in the application. The checksum calculation and global buffer memcpy takes most of the overhead. The reception time is linearly with packet size.

Secondly, the reception time for udp packet with 32-bit checksum calculation is tested. Table 7 and Figure 28 show the details about the result.

<table>
<thead>
<tr>
<th></th>
<th>64B</th>
<th>128B</th>
<th>256Byte</th>
<th>512Byte</th>
<th>1KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global buffer memcpy(us)</td>
<td>3.04</td>
<td>5</td>
<td>8.8</td>
<td>16.56</td>
<td>32.04</td>
</tr>
<tr>
<td>Driver setting(us)</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>uIP process memcpy(us)</td>
<td>1.6</td>
<td>2.56</td>
<td>3.6</td>
<td>7.68</td>
<td>12.8</td>
</tr>
<tr>
<td>ip checksum delay(us)</td>
<td>1.44</td>
<td>1.44</td>
<td>1.44</td>
<td>1.44</td>
<td>1.44</td>
</tr>
<tr>
<td>udp checksum delay(us)</td>
<td>2.68</td>
<td>5.24</td>
<td>12.2</td>
<td>23.2</td>
<td>47.8</td>
</tr>
<tr>
<td>uIP process delay(us)</td>
<td>3.76</td>
<td>3.96</td>
<td>4.04</td>
<td>3.92</td>
<td>4.04</td>
</tr>
<tr>
<td>Total reception delay(us)</td>
<td>13.32</td>
<td>19</td>
<td>30.88</td>
<td>53.6</td>
<td>98.92</td>
</tr>
</tbody>
</table>

Table 7 Reception time with 32-bit checksum calculation
As can be seen from above, the driver setting and memcpy makes no difference but the checksum overhead is decreased. The larger the packet size, the more decreased overhead. Table 8 shows how much performance is improved for total udp reception time using 32-bit checksum calculation.

<table>
<thead>
<tr>
<th>Packet Size (Bytes)</th>
<th>64B</th>
<th>128B</th>
<th>256B</th>
<th>512B</th>
<th>1KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum Improvement (%)</td>
<td>23</td>
<td>31</td>
<td>33</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>Total Improvement (%)</td>
<td>11</td>
<td>15.5</td>
<td>17.2</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 8: Performance improvement for udp reception time

From Table 8, the performance improvement is increased when packet size is increased.

Let’s assume the checksum delay for 8-bit architecture and 32-bit architecture is C8 and C32, the performance improvement is P1, the improvement can be calculated as the formula \( P1 = \frac{C8 - C32}{C8} \times 100\% \). As the packet size increased, C8 and C32 is increased linearly, C8-C32 is not linearly with the packet size. However, when packet size grows larger, the difference between C8-C32 for each packet size is becoming smaller, so the improvement will be stable at some value which in our case is around 39%.

Let’s assume the total delay for 8-bit architecture and 32-bit architecture is T8 and T32, the performance improvement is P2, the improvement can be calculated as the formula \( P2 = \frac{T8 - T32}{T8} \times 100\% \). As the packet size increased, T8 and T32 is
increased linearly, T8-T32 is not linearly with the packet size. However, when packet size grows larger, the difference between T8-T32 for each packet size is becoming smaller, so the improvement will be stable at some value which in our case is around 24%.

### 4.2.3 Reception rate test

In this test, the reception rate for uIP running on Leon3 is tested to see how much performance it can be achieved.

The reception rate is a value that can be used to see how much performance uIP can be achieved under high traffic load for reception. In this test, packETH 1.7 [9] is used to generate and send different sizes of packets. For each packet size, different inter packet delays are used and tested to indicate different traffic load. For this test, 2 different checksum calculations are applied as well as the reception time test to see whether there is improvement between the two calculations.

Firstly, the reception rate with 8-bit checksum calculation is tested. In this test, 1MB data is transmitted with different packet size. Table 9 shows how many packets are received in the receiver end. The numbers in the parentheses in the first row are indicating how many packets are transmitted for different packet sizes from the transmitter end.

<table>
<thead>
<tr>
<th>Delay\size</th>
<th>64 B(16000)</th>
<th>128B(8000)</th>
<th>256B(4000)</th>
<th>512B(2000)</th>
<th>1KB(1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 us</td>
<td>9511</td>
<td>3015</td>
<td>1000</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>25 us</td>
<td>11747</td>
<td>3630</td>
<td>1100</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>30 us</td>
<td>13951</td>
<td>4340</td>
<td>1280</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>35 us</td>
<td>16000</td>
<td>4960</td>
<td>1453</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>40 us</td>
<td>16000</td>
<td>5584</td>
<td>1650</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>45 us</td>
<td>16000</td>
<td>6201</td>
<td>1844</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>50 us</td>
<td>16000</td>
<td>6997</td>
<td>2142</td>
<td>702</td>
<td>341</td>
</tr>
<tr>
<td>100 us</td>
<td>16000</td>
<td>8000</td>
<td>3880</td>
<td>1189</td>
<td>408</td>
</tr>
<tr>
<td>150 us</td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>1645</td>
<td>528</td>
</tr>
<tr>
<td>200 us</td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>641</td>
</tr>
<tr>
<td>250 us</td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>795</td>
</tr>
<tr>
<td>300 us</td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>892</td>
</tr>
<tr>
<td>350 us</td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 9 Packet reception with 8-bit checksum calculation

From Table 9, the packets received for 512B and 1KB under 45us inter packet delay are not recorded because there is packet loss from the transmitter end. If the inter packet delay is the same, the larger the packet size, the higher the traffic load. Figure 29 shows the reception rate as a function of inter packet delay.
From Figure 29, we can see that for the same inter packet delay, the larger the packet size, the lower the reception rate. The reception rate is related to the packet process time and inter packet delay. For small packet, the process time is short and traffic load is low under the same inter traffic delay compared with the large packet. Therefore, the reception rate is higher than the large packet.

Secondly, the reception rate with 32-bit checksum calculation is tested under the same test environment. Table 10 and Figure 30 show the test result.

<table>
<thead>
<tr>
<th></th>
<th>64 B(16000)</th>
<th>128B(8000)</th>
<th>256B(4000)</th>
<th>512B(2000)</th>
<th>1KB(1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>20 us</strong></td>
<td>10230</td>
<td>3569</td>
<td>1218</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>25 us</strong></td>
<td>12650</td>
<td>4284</td>
<td>1334</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>30 us</strong></td>
<td>15057</td>
<td>5150</td>
<td>1650</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>35 us</strong></td>
<td>16000</td>
<td>5890</td>
<td>1828</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>40 us</strong></td>
<td>16000</td>
<td>6693</td>
<td>2071</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>45 <strong>us</strong></td>
<td>16000</td>
<td>7439</td>
<td>2341</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>50 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>2560</td>
<td>810</td>
<td>411</td>
</tr>
<tr>
<td><strong>100 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>1446</td>
<td>476</td>
</tr>
<tr>
<td><strong>150 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>655</td>
</tr>
<tr>
<td><strong>200 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>815</td>
</tr>
<tr>
<td><strong>250 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>987</td>
</tr>
<tr>
<td><strong>300 us</strong></td>
<td>16000</td>
<td>8000</td>
<td>4000</td>
<td>2000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 10 Packet reception with 32-bit checksum calculation
In this test, 32-bit checksum calculation is used instead of the 8-bit checksum calculation. Compared with the 8-bit's reception rate, 32-bit reception rate has an apparent improvement due to the efficiency checksum calculation that reduces the process time.

4.3 TCP performance test

The previous tests are for UDP protocol. Since UDP is not a reliable protocol, in this test, TCP performance is tested as TCP is a reliable protocol.

In this test, two experiments for throughput are designed and tested. The first experiment is designed for testing TCP throughput with different advertised window size. The second one is designed for testing how multi-connections can affect the throughput. In both experiments, the uIP is acted as a TCP client server using 32-bit checksum calculation. A TCP client program has been implemented to send data from computer side.

For the first experiment, the uIP is configured to adjust different advertised window sizes. The timer unit is used to record the time delay for the whole process and the RTT for TCP is recorded from wireshark [10]. Table 11 shows the test results.

<table>
<thead>
<tr>
<th>win size(B)</th>
<th>100</th>
<th>300</th>
<th>500</th>
<th>700</th>
<th>900</th>
<th>1100</th>
<th>1300</th>
<th>1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>pkt size(B)</td>
<td>100</td>
<td>300</td>
<td>500</td>
<td>350</td>
<td>450</td>
<td>550</td>
<td>650</td>
<td>750</td>
</tr>
<tr>
<td>delay(ms)</td>
<td>4.84</td>
<td>2.63</td>
<td>2.14</td>
<td>2.45</td>
<td>2.24</td>
<td>2.13</td>
<td>2.05</td>
<td>2.00</td>
</tr>
</tbody>
</table>

Figure 30 Reception rate as a function of inter packet delay with 32-bit checksum calculation
Table 11 Test results for TCP throughput with different window sizes

<table>
<thead>
<tr>
<th>RTT (us)</th>
<th>161</th>
<th>242</th>
<th>310</th>
<th>345</th>
<th>402</th>
<th>462</th>
<th>520</th>
<th>583</th>
</tr>
</thead>
</table>

The TCP client sends the same size of data with 15KB for each of the window sizes. Therefore, the throughput is calculated as data size divided by delay. As can be seen from Table 10, the transmitted packet size is adjusted with window size. The packet sizes are equal to window sizes up to 500B. However, when window size is larger than 500B, the transmitted packet size is equal to half of the window size based on the window scaling introduced in the [11]. Figure 31 shows the throughput and RTT as a function of different advertised window sizes.

![Figure 31 TCP throughput and RTT as a function of advertise window size](image)

As the first experiment shows relation between throughput and window sizes, now we want to know the relation between throughput and simultaneous connections. Therefore, a fixed window size is chosen (500B) which should not cause the window scaling. Then up to 10 simultaneous connections are introduced to see the difference of throughput. Table 12 and Figure 32 give the results of the test.

<table>
<thead>
<tr>
<th>connections</th>
<th>delay(ms)</th>
<th>RTT(us)</th>
<th>Throughput(Mbyte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.14</td>
<td>310</td>
<td>6.9</td>
</tr>
<tr>
<td>2</td>
<td>4.72</td>
<td>445</td>
<td>6.36</td>
</tr>
<tr>
<td>3</td>
<td>7.15</td>
<td>663</td>
<td>6.30</td>
</tr>
<tr>
<td>4</td>
<td>9.49</td>
<td>890</td>
<td>6.32</td>
</tr>
<tr>
<td>5</td>
<td>11.97</td>
<td>1130</td>
<td>6.26</td>
</tr>
<tr>
<td>6</td>
<td>14.44</td>
<td>1354</td>
<td>6.23</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>17.07</td>
<td>1611</td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>19.26</td>
<td>1810</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>21.70</td>
<td>1575</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>24.17</td>
<td>1805</td>
</tr>
</tbody>
</table>

Table 12 Test results for TCP throughput with simultaneous connections

From Table 12 and Figure 32, we can see that the throughput is decreased but not too much when there are simultaneous connections established.

From the experiments above, we can see the trends of the throughput in TCP for different window sizes and simultaneous connections.

4.4 Energy consumption test

In this test, the energy consumption will be tested with generic checksum calculation and 32-bit checksum calculation and to see how much energy can be saved using the 32-bit checksum. Energy consumption is estimated in real-case where Leon3 is running as ASIC chip. The calculation is based on the model in [12]. In this model, instruction energy is characterized for Leon3 synthesized using TSMC90 nm technology and 400 Mhz frequency.

For this test, we are more interested in how much energy can be saved using the improved checksum calculation. Therefore, the UDP reception energy consumption is tested because it contains the checksum calculation and is simple and easy to test compared with TCP. The GDB [13] debugger tool is used to collect the instructions for receiving one UDP packet. Two packet sizes are used, one is 64 byte and one is
1K byte. Then the energy consumption is calculated based on model in [12]. The Figure 33 and Figure 34 show the energy consumption for receiving 64 byte packet and 1KB packet.

For this test, the results are presented as function of cache miss rate. The
calculation method from [12] is presenting energy in case of cache miss or cache hit, but the cache miss or cache hit for development board could not be tested, the numbers of the cache miss rate shown in Figure 33 and Figure 34 are just reference numbers in representing the results.

From the figures above, we can see that the energy is saved around 10% for receiving 64-byte UDP packet and 30% for receiving 1024-byte UDP packet. As have described in section 4.2.2, the performance improvement will be higher when packet size is larger. Therefore, the larger the packet size, the more energy will be saved.
Chapter 5  Conclusion and Future Works

5.1  Conclusion

In this thesis, there are mainly four things are performed:

- A deep study for the uIP TCP/IP stack.
- Successfully ported the uIP TCP/IP stack to a low power, high performance processor – Leon3.
- An improved 32-bit checksum calculation method has been implemented to utilize the 32-bit architecture.
- A detailed performance analysis of small TCP/IP stack on low power processor has been performed.

The following are the results of the performance analysis:

- The performance for ICMP is very high due to the simple implementation of ICMP protocol in uIP.
- The performance of uIP is mainly dominated by the checksum calculation.
- The checksum delay is improved from 23% up to around 39% using 32-bit checksum calculation compared with the generic checksum calculation.
- The reception time is improved from 10% up to around 25% using 32-bit checksum calculation compared with the generic checksum calculation.
- The TCP throughput is ranged from 3Mbyte/s up to 7.5Mbyte/s for only one connection. The throughput will be slightly decreased when simultaneous connections are applied.
- The energy consumption for receiving packets will be saved from 10% up to 30% after using the 32-bit checksum calculation.

The performance for uIP running on 32-bit architecture has an apparent improvement compared with running on 8-bit architecture.

5.2  Future Works

As an extension for this thesis, there may have some further work to do:

- More advanced and complex application can be implemented such as video codec.
- Performance for other “network device” can be tested such as wireless.
- Security mechanism can be implemented.
Bibliography


http://www.sics.se/~adam/old-uip/size.html