Small TCP/IP stacks for micro controllers

By: Lucas van der Ploeg

Supervisors:

ir. Wout Klaren (3T)

Dr. ir. Pieter-Tjerk de Boer (Universiteit Twente)

1. Preface

My name is Lucas van der Ploeg and I am a student at the University of Twente. I am studying Telematics and as part of this study, I had to do a 14-week Bachelor assignment. I decided to look for an assignment outside the University and found a company called 3T BV about 100m from the main entrance of the campus and another one called WMC a little further away. 3T had the most interesting assignment for me so I decided go to 3T.

Because the University of Twente had recently switched to quartiles instead of trimesters, I had to do a 14-week assignment in an 11-week period. Fortunately the holidays started right after that period so I could continue my assignment during the first weeks of the holiday. As I was working at a company, I had to make at least 40 hours a week, which is something many students are not used to. This did mean the assignment would not take up much more than the 14 weeks required.

While I worked on my assignment, I spent a whole lot of time on finding out "simple" things like, how to work with a micro controller and how to program in C/C++. When I finally could make the micro controller say "Hello world" to me, getting it to open a TCP connection seemed like a piece of cake. I did learn a lot from this and I think the things I learned are very useful. However, it would have been nice if I could have spent more time on the actual assignment and less time on learning the basics of embedded systems programming.

I want to thank my supervisors Wout Klaren and Pieter-Tjerk de Boer for all their good advice and the time they spend helping me. I also want to thank my other roommates at 3T for making it a lot of fun being there.

2. Abstract

There are many small TCP/IP implementations available for micro controllers, both commercial and open source. I compared these implementations by doing some research on the internet and by testing some of them on a Motorola ColdFire processor.

It appeared LwIP was the most used open source implementation and most other open source implementations were a bit limited or outdated. The many commercial implementations all promised roughly the same, for about the same prise. However, most did not give very specific information online and did not offer an option to test their implementation before buying it. Quadros Quadnet and ARC RTCS TCP/IP did offer free demo versions but due to the limited time available, I could only intensively test Quadros.

To use the LwIP stack I needed to configure it to work without an operating system and I needed to write a driver for the Ethernet controller. I created some test and debug applications to find out the best way of using the LwIP stack. I found out that when you use the LwIP stack correctly it is stable and reliable.

I designed a few tests to compare and test the LwIP stack and the Quadros stack. I found out that getting started with the Quadros stack was a lot less time consuming then getting started with the LwIP stack. However, when running the tests the Quadros stack was not very stable and reliable. When you buy Quadros, Quadnet could probably help you fix these problems.

When you have the knowledge and the time to configure an open source TCP/IP implementation, there is no need to buy a commercial implementation, as I could not find any important limitations in LwIP. When you need a working implementation fast you could use a commercial implementation like Quadros, however you still need to spend some time getting acquainted with the implementation before you can actually use it. Another possible advantage of a commercial stack like ARC RTCS TCP/IP is the tools you can use to configure the stack. When you often create applications using a TCP/IP stack these tools can speed up implementation.

3. Table of Contents

4. Introduction

4.1 The assignment

Small TCP/IP stacks for micro controllers.

There are micro controllers available with internal Flash ROM and RAM. When using a small tcp/ip stack without an operating system or with a very limited operating system, it is possible to have internet capabilities in embedded systems without the need for extra RAM and ROM chips. This reduces the hardware costs.

There are multiple open source and commercial implementations on the market. The assignment is to select some of these implementations and use and test them in a simple application on a mcf5282 coldfire processor using only the internal ROM and RAM. The goal is to find out the differences in performance and capabilities of these implementations.

4.2 3T

3T BV is a research and development company that specialises in microelectronics and embedded systems. 3T has about 35 employees and its main office is located in Enschede, a second office is located in Best. The company originated from an organisation called CME (Centrum for Micro Electronics). CME was founded in 1982 by the government to stimulate knowledge gathered by the three technical universities (Delft, Eindhoven and Enschede) to get from the universities to small companies. In 1988, the CME division in Enschede founded a company called Twente Technology Transfer BV. In 1994, it was reborn as 3T BV.

4.3 Approach

To get a first idea of the available TCP/IP stacks I did a lot of searching on the internet. I made an overview of the most important available implementations. I selected an open source and a commercial implementation to investigate further.

I wanted to compare the implementations at four different aspects.

4.3.1 Ease of use

How easy it is to use an implementation for the first time and create your application on it. Also how easy it is to maintain you application and to make changes when you have already implemented your application with it. To learn more about this I wrote an Ethernet driver for the LwIP stack to run on the ColdFire and I wrote some test applications.

4.3.2 Stability

A very important factor is the stability of an implementation. You should be able to rely on a TCP/IP implementation to run for years without needing a reset or any maintenance. No fatal errors should occur or it should at least recover from those errors. As we don't have time to wait for a few years and look for error we will have to stress test the stacks a little. I devised some tests for this purpose and ran them on the stacks.

4.3.3 Performance

A factor that could be important in some applications is the performance of a TCP/IP stack. How much traffic can a stack handle and how much CPU time does the stack need with a certain amount of load. To learn more about this I devised two tests.

4.3.4 Cost

The total cost of an implementation is mainly dependet on three factors. The purchase cost of the product. The cost of the person-hours and possibly training needed to implement the application using the product. And the cost of maintenance after the application is installed.

An open source is free to purchase but might still be more expensive than a commercial implementation when getting your application to work is much more time consuming. Some commercial TCP/IP implementations allow you to upload new applications using the TCP/IP stack. This makes is possible to update your product dynamically after distribution. This could be a huge advantage.

5. Overview of available implementations

5.1 Open Source implementations

5.1.1 uIP

uIP is an implementation of the TCP/IP protocol stack intended for small 8-bit and 16-bit microcontrollers. It is completely RFC1122 compliant but has some limitations. For instance, a retransmit is managed by the stack, but the data that needs to be retransmitted is requested from the user application.

uIP can be used together with Contiki, a very small OS which supports dynamic application download and a gui using VNC. The uIP stack uses less then 10kB ROM and 2kB RAM and Contiki can easily fit in 100kB ROM and 10kB RAM. You can use it any way you want as long as you leave a copy of the copyright notice in the source and/or documentation.

http://www.sics.se/~adam/uip/

5.1.2 LwIP

LwIP is a TCP/IP implementation designed for small code size and efficient memory usage. It is still widely used, and implemented. And is designed to use with or without an operating system. lwIP uses around 40kB of RAM and 30kB ROM and you can use it any way you want as long as you leave a copy of the copyright notice in the source and/or documentation.

http://www.sics.se/~adam/lwip/

5.1.3 uC/IP

uC/IP is a TCP/IP stack developed for microcontrollers and embedded systems but is not often used. It based on the BSD TCP/IP implementations and is still a bit large compared to other implementations. uC/IP carries the BSD license so you can freely use it as long as you leave a copy of the copyright notice in the source and/or documentation.

http://ucip.sourceforge.net/

5.1.4 tinytcp, wattcp and others

There are a lot of (semi) Open Source TCP/IP stacks available for DOS, they are often very old and no longer in use. They are not intended for use in embedded systems and sometimes have a paid licence for commercial use.

http://www.unusualresearch.com/tinytcp/tinytcp.htm

http://www.wattcp.com

5.1.5 BSD 4.4

A lot of TCP/IP stacks are based on the BSD implementation. Because of its size it is not very useful for embedded systems; however it might be useful as a reference.

5.2 Commercial implementations

All commercial TCP/IP implementations listed below, promise to be very efficient and robust. They all use a zero copy mechanism to make efficient use of the resources.

5.2.1 CMX-tcp/ip (or CMX-MicroNet for 8/16 bits)

CMX-tcp/ip runs on CMX-RTX RTOS or without an RTOS. It supports many processors including the ColdFire. A configuration tool is available and the stack uses about 20kB of ROM. CMX TCP/IP pricing starts at $9,000 and is provided with full source code, no royalties on shipped products, and free technical support and software updates. There is no demo or tryout version available for the ColdFire.

http://www.cmx.com/tcpip.htm

5.2.2 NetX

NetX is the TCP/IP stack of the ThreadX RTOS; it uses about 5 to 20 kB of code depending on configuration. It is delivered with configuration tools and there are training courses available. A licence costs around $5000 to use it for multiple applications but on only one processor type. The ColdFire is supported but there is no demo version for the ColdFire on the website.

http://www.rtos.com/procNX-coldfire.asp

5.2.3 NicheStack

NicheStack and NicheLite are 2 TCP/IP implementations. NicheStack requires about 50kB ROM and RAM and NicheLite only 12kB. Both come with a configuration and debug tool. You get the source code royalty free, and 12 months support. No price information is given, but you can download a 1-hour demo. But I could not create my own application on it so I could not test on it. The demo only shows a webpage.

http://www.iniche.com/nichestack.php

5.2.4 ARC RTCS TCP/IP

ARC has a TCP/IP stack and RTOS with an evaluation package, but they give no price and licence information. They say it's small but not how small. It comes with a configuration and performance analysis tools.

http://www.mqxembedded.com/products/rtcs/

5.2.5 RTXC Quadnet TCP/IP

The Quadnet TCP/IP stack runs on the Quadros TROS, it requires about 256kB ROM and 32kB RAM. There are three versions available, a free special edition containing a preconfigured binary version with no restrictions, the standard edition with a configurable binary, and a fully configurable professional edition including all sources. The standard edition costs $17.500 per project, and the professional edition costs $31.500 per project. The free edition does not seem to work properly.

http://www.quadros.com/products/communication-stacks/rtxc-quadnet/

5.2.6 TargetTCP

TargetTCP is the TCP/IP stack from TargetOS. It can also run without an RTOS and requires about 30kB of ROM and 32kB of RAM. For $9800 you get a licence to use the source at a specified location for multiple projects. There is no demo version on the website.

http://www.blunkmicro.com/tcp.htm

5.2.7 uC/TCP-IP

uC/TCP-IP runs on uC/OS-II, it uses about 100kB of ROM and 60kB of RAM, The tcp/ip stack is not complete (ICMP incomplete, no IP fragmentation, no IP routing/forwarding) and you have to buy a licence for every end product. There is no demo version available.

http://www.ucos-ii.com/products/tcp-ip/tcp-ip.html

5.3 Selection

LwIP is specially designed for micro controllers and not adapted from an implementations used for workstations. It appears to be a complete TCP/IP stack without shortcuts and with all functionality of a large stack. LwIP is also the only one with an active user community. Because of these three reasons I decided to use LwIP as an open source TCP/IP protocol stack.

The decision for which commercial stack to test was a bit more difficult, there ware many implementations available and they all promised roughly the same. Quadros Quadnet and ARC RTCS TCP/IP both offered a free demo version that would run on the ColdFire evaluation board I could use so I wanted to test them both. Unfortunately, I did not have enough time to try both so I only tested Quadros.

	Pro	Con
uIP	Free, very small	Much left to user application
LwIP	Free, complete, frequently used	
uC/IP	Free	Large, not often used
Tinytcp, wattcp and others	Free	Not very usable
BSD 4.4	Free, stable	Too big
CMX-tcp/ip	ColdFire support, updates included	No demo
NetX	ColdFire support, tools & training available	No demo
NicheStack	ColdFire support, tools available	Limited demo
ARC RTCS TCP/IP	ColdFire support, tools available	
RTXC Quadnet TCP/IP	ColdFire support	Unstable demo
TargetTCP	ColdFire support	No demo
uC/TCP-IP	ColdFire support	Incomplete, no demo

6. LwIP

There are multiple reasons why I decided to test and evaluate LwIP instead of other available TCP/IP protocol stacks. LwIP is specially designed for micro controllers; other small TCP/IP implementations are developed for DOS or derived from the BSD implementation and are less efficient on microcontrollers. It also seemed that LwIP is the most referenced small TCP/IP stack, and the only one still being improved with an active user forum.

I decided to test LwIP without an operating system. By running LwIP without an operating system, I expected to get the highest performance from LwIP with the smallest system requirements. The performance would not be influenced by operating system characteristics.

6.1 About LwIP

LwIP is short for Lightweight Internet Protocol, a small TCP/IP implementation designed for microcontrollers with limited memory resources and processing power. Adam Dunkels originally developed LwIP at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS). Presently it is maintained by a group of 19 volunteers at Savannah, a website for distribution and maintenance of Free Software that runs on free operating systems.

LwIP is a full-scale TCP/IP implementation with optimised code size and memory usage. It includes the following protocols:

- Internet Protocol (IP), versions four and six, for worldwide addressing. It includes fragmentation and reassembly, and forwarding over multiple interfaces.
- Internet Control Message Protocol (ICMP), versions four and six, for network state related messages.
- User Datagram Protocol (UDP), for simple data frame transmission.
- Transmission Control Protocol (TCP), including congestion control, RTT estimation and fast recovery/fast retransmit for byte stream connections.
- Dynamic Host Configuration Protocol (DHCP), for automatic address assignment.
- Point to Point Protocol (PPP), for communication over serial lines.
- Serial Line Internet Protocol (SLIP), for communication over serial lines
- Address Resolution Protocol (ARP), for mapping between Ethernet and IP addresses.

All these protocols are optional and you can replace them by your own version or add your own protocols. You can choose between to API's to use the protocols. Using the raw call-back API you directly call the functions from the TCP/IP stack, this ensures optimal performance. You can also use the optional Berkeley-alike socket API. This API offers you some easy to use functions and handles the communication with the TCP/IP stack for you. This is less efficient then the raw API but easier to use.

The LwIP stack also includes some memory management functionality and optionally some statistics are kept for debugging and performance analysis to help with tuning.

There are multiple example applications, ports and network interface drivers available you can use directly or as an example for your own driver, port and application.

6.2 Usage

To use LwIP you need to do some configuration, a network interface driver, and of course a working environment. Your application needs to initialize the stack and regularly call some timer routines. In the following paragraphs, I describe how to configure the stack, and what initialisation and timer functions your application should call. I also describe how to use the raw TCP API and how to create a network interface driver.

To get a working environment you need a compiler, some linker and hardware initialisation scripts and some basic function like printf() for debugging. LwIP uses its own memory management so you don't need malloc() and free() routines, but you might want an operating system for some thread control. Getting a working environment can be very tricky even when you already have some examples from your hardware supplier.

6.2.1 Configuration

To use the LwIP stack first you need to define some settings in four header files and optionally create a sys_arch.c for the OS emulation layer. The main configuration file is called lwipopts.h and changes the default settings from opt.h. The other three header files are called cc.h, sys_arch.h and perf.h and contain OS and environment depended options. For all files, there are multiple examples available in the CVS tree.

6.2.1.1 lwipopts.h

In this file you can enable or disable parts of the stack, you can set the buffer sizes, and you can enable debugging. You can see a complete list of all options and their default settings in opt.h.

I tested the LwIP stack on the Motorola ColdFire, without an operating system, with only three extra parts of the stack enabled. I defined NO_SYS so all the semaphore and mailbox functions have null definitions. This can only be used when all LwIP functions are called in the same priority level so they do not interrupt each other. I also enabled LWIP_DHCP for automatic IP configuration and LWIP_STATS_DISPLAY for displaying a list of statistics on LwIP.

The size and number of all buffers I have chosen in this file are explained later on.

6.2.1.2 cc.h

This header file contains compiler and architecture dependent options like definitions integer sizes.

6.2.1.3 perf.h

In perf.h two functions are defined for performance measurement.

6.2.1.4 sys_arch.c and sys_arch.h

These two files define functions for the OS emulation layer. When you want to integrate LwIP with an operating system there are a few functions you have to create. These functions are used by LwIP for communicating with the operating system. Which functions you have to define is described in a document found in the CVS directory called sys_arch.txt. In addition, multiple working examples are available.

6.2.2 Initialisation

Before you use functions from the LwIP stack you have to initialise all the parts in a specified order. And when you use DHCP you have to wait for DHCP to resolve some IP settings.

The first function you have to call is stats_init() to zero all statistics. These statistics are very useful for debugging and performance tuning but you could disable them in a production release.

If you use an operating system you should call sys_init() to initialise the OS emulation layer. This OS emulation layer maps functions needed by LwIP to OS specific functions.

Next you have to initialise all memory buffers by calling mem_init() for the heap memory, memp_init() for a predefined number of different structures and pbuf_init() for a pool of pbuf's.

When this is done you can initialise the protocols by calling netif_init(), ip_init() and optionally udp_init() and tcp_init().

Now the LwIP stack is completely initialised, but before you can start using the stack, you need to start calling some functions at regular intervals as described below and you need to register and enable a network device. This is done by calling netif_add() and netif_set_default(). When you have specified the IP address of the interface, the net mask and the gateway IP address you can call, netif_set_up(). When you want DHCP to

configure the IP settings you call dhcp_start(). After enabling the interrupts you have to wait for netif_is_up() to return true before you use the network device.

When all parts of LwIP are initialised, you can start to register TCP listeners and other services that use the LwIP functionality. An example of the initialisation is found in main.c.

6.2.3 Timer

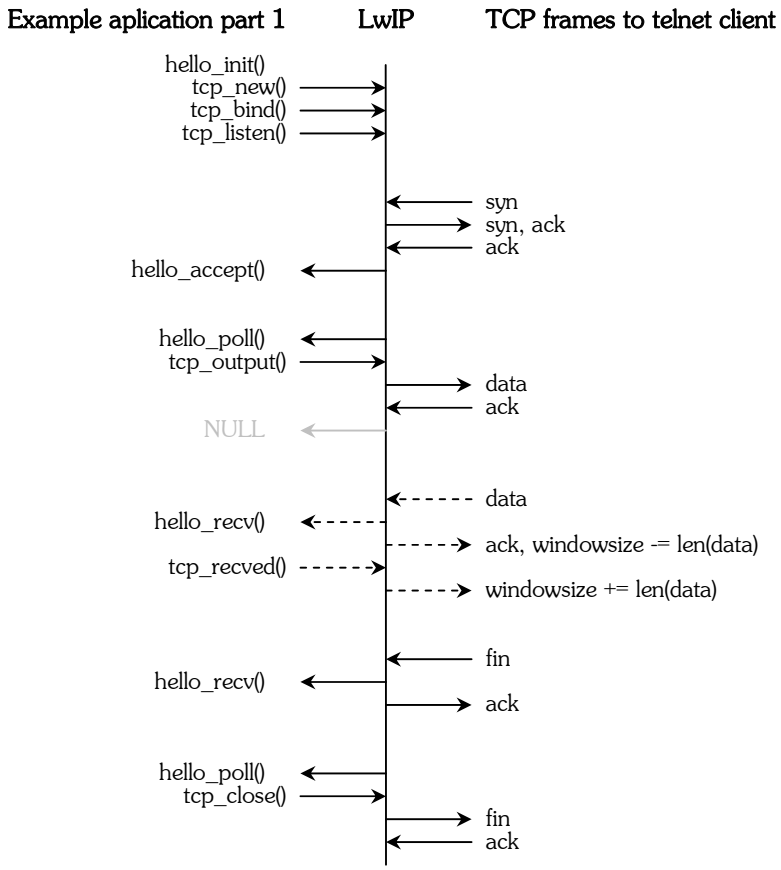
There are a number of functions in the LwIP stack that have to be called at certain intervals. All the functions and there intervals are listed below and an example interrupt routine can be found in main.c in the appendices. The intervals can are given in the header corresponding header files and can be tuned.

tcp_fasttmr()	250ms
tcp_slowtmr()	500ms
ip_reass_tmr()	500ms
dhcp_fine_tmr()	500ms
dhcp_coarse_tmr()	60000ms
etharp_tmr()	5000ms

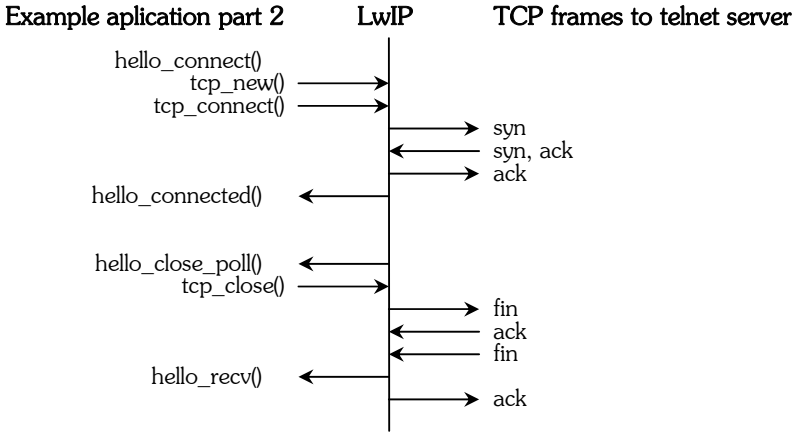
6.2.4 TCP

In the CVS tree of LwIP rawapi.txt is found, this document describes how to use the raw callback functions of LwIP. Because the information in the document is incomplete, outdated and does not give a clear example I will explain how you can use a TCP connection with LwIP by giving an 'as short as possible' example and commenting on it. As you can see, you need a large amount of code to use the raw callback API. The example consists of 2 parts that can be started by calling the functions hello_init() and hello_connect() at a priority level higher or equal to that of the Ethernet controller.

In the first part, I open a listening TCP connection that accepts a connection. After accepting the connection, it receives and confirms all incoming data, while trying to send a "hello world" message. The implementation waits for the other side to close the connection and responds by also closing the connection. The implementation keeps trying to send "hello world" and send a close from a poll function until it succeeds.



In the second part of the example, I open a connection to a telnet server and start receiving and confirming all incoming data with the same function as in the first part. Only this time I use a poll function that always tries and keeps trying to close the connection until it succeeds.



6.2.4.1 Listen for incoming TCP connections

1. To open a listening TCP connection you first need to create a tcp_pcb (protocol control block) structure using tcp_new(). In this structure, LwIP stores all information about a connection. If tcp_new() returns NULL no room is available for a new TCP connection and you can't open a new listening TCP connection.
2. When you succeeded in creating a new PCB you can try to bind it to a port and IP address using tcp_bind(). When you want to bind the listening connection to all local IP addresses or you only have one local IP address, you can use IP_ADDR_ANY as IP address. If the port is already in use tcp_bind() will

return ERR_USE and you can't open a listening connection at that port. Do not forget to cleanup the pcb when this happens.

3. The next step is to tell LwIP to start listening. For the actual listening connection, LwIP uses a different (smaller) pcb structure. This structure is allocated when you call tcp_listen(). If no space is available for a new listening pcb tcp_listen() returns NULL, if the allocation succeeds LwIP clears the original pcb and starts listening. When NULL is returned, you should clear the original pcb yourself.

4. The last step is to set some options in the pcb. You can give LwIP an argument, which is returned to you each time LwIP calls one of your callback functions. Usually this is a pointer to a block of status information you can use, but in our example, no status information is needed yet so we set it to NULL. The second option you should set is the priority of incoming connections. Each connection has a priority level, when all connections are in use, the connection that has a priority level equal to or lower than the priority level of the incoming connection, and has been idle the longest, will be removed.
The last thing you need to do is to specify your accept function using tcp_accept().

```
err_t hello_init(void)
{
   struct tcp_pcb * pcb;
   struct tcp_pcb * lpcb;

1.    if ((pcb = tcp_new()) == NULL)
         return ERR_MEM;

2.    if (tcp_bind(pcb, IP_ADDR_ANY, 22) != ERR_OK)
      {
         tcp_abort(pcb);
         return ERR_USE;
      }

3.    if ((lpcb = tcp_listen(pcb)) == NULL)
      {
         tcp_abort(pcb);
         return ERR_MEM;
      }

4.    tcp_arg(lpcb, NULL);
      tcp_setprio(lpcb, TCP_PRIO_NORMAL);
      tcp_accept(lpcb, hello_accept);

      return ERR_OK;
}
```

6.2.4.2 Accept an incoming TCP connection (passive open)

1. When someone tries to connect to our listening TCP connection and room for a new pcb can be allocated our previously specified accept function is called. In almost every case you need to allocate some memory for status information and set the location as the argument LwIP gives you when calling one of your functions. If you cannot, you can abort the connection by returning an ERR_MEM. In our case, we reserve only one byte.

2. When you have decided to accept the connection, you should declare your callback functions. You should at least declare an error and receive function. The error function is called when something goes wrong and is used to inform you the connection is no longer available and you should free the memory you were using for the connection. The receive function passes you the received data, or a NULL pointer when a close is received.

3. Optionally you can specify a poll function that is called periodically and a sent function that informs you when data you have sent has been confirmed. For this example, we are not interested in when data has been confirmed so we do not specify a sent function. We specify a poll function to be called every two TCP gross timer periods of half a second. If you want the connection not to be lost when to many other connections occur you should set the priority to a higher level using tcp_setprio()

4. To finish the accept function we return ERR_OK.

```
err_t hello_accept(void *arg, struct tcp_pcb *pcb, err_t err)
{
   u8_t *state;
```

```
1.    if ((state = mem_malloc(1)) == NULL)
         return ERR_MEM;
      *state = 0;
      tcp_arg(pcb, state);

2.    tcp_err(pcb, hello_err);
      tcp_recv(pcb, hello_recv);

3.    tcp_sent(pcb, NULL);
      tcp_poll(pcb, hello_poll, 2);

4.    return ERR_OK;
   }
```

6.2.4.3 Opening an outgoing tcp connection (active open)

1. To open an outgoing connection the first thing you usually want to do is reserve some memory, in our example just 1 byte for status information.
2. When this succeeds we will try to reserve a new tcp_pcb for the connection using tcp_new(). If this fails you should free the memory previously reserved and give up. You can also close another connection or retry using tcp_alloc() with a higher priority.
3. Now you can set the options in the same way as explained above. Instead of the poll function from our previous example, we let LwIP call a close function every five seconds.
4. Afterwards the pcb of the connection is ready and you can call tcp_connect(). If no room is available to create a TCP syn segment (a segment to inform the other side you want to open a connection), tcp_connect() returns ERR_MEM. In this case you can give up and clear the allocated memory and pcb or you can keep trying.
5. If you specified a connected function when calling tcp_connect() your connected function is called when the connection is established. If the connection fails, your error function is called. Currently, no error is given to your connected function. In the connected function, you can for example send some data to the other host. To keep this example simple, we try this only once.

```
   err_t hello_connect(void)
   {
      u8_t *state;
      err_t err;
      struct tcp_pcb *pcb;
      struct ip_addr ipaddr;
      IP4_ADDR(&ipaddr, 192,168,0,112);

1.    if ((state = mem_malloc(1)) == NULL)
         return ERR_MEM;
      *state = 1;

2.    if ((pcb = tcp_new()) == NULL)
      {
         mem_free(state);
         return ERR_MEM;
      }

3.    tcp_arg(pcb, state);
      tcp_err(pcb, hello_err);
      tcp_recv(pcb, hello_recv);
      tcp_sent(pcb, NULL);
      tcp_poll(pcb, hello_poll_close, 10);

4.    err = tcp_connect(pcb, &ipaddr, 22, hello_connected);
      if (err != ERR_OK)
      {
         mem_free(state);
         tcp_abort(pcb);
      }
      return err;
   }

5. err_t hello_connected(void *arg, struct tcp_pcb *pcb, err_t err)
   {
      tcp_write(pcb, helloworld, 12, 0)
      return ERR_OK;
   }
```

6.2.4.4 Receiving data

1. When data or a FIN-flag (passive close) has arrived your previously defined receive function is called. If the pbuf pointer is NULL, a FIN-flag is received. To cleanly close a connection, both sides have to successfully send a FIN-flag. Therefore, if you have already sent a fin flag you can clean up. If you haven't sent a FIN-flag yet you have to send it by calling tcp_close(). It is possible there is no room for a new tcp segment containing the FIN-flag so you have to keep trying to call tcp_close() until LwIP is able to store the FIN-flag. If you try closing only once the connection might stay in the close-wait state (see appendix 13.1 TCP state diagram)
2. If the pbuf pointer is set, the pbuf contains the received data. When you are done handling the data you should clear the pbuf and afterwards tell LwIP how many bytes you have handled using tcp_recved(). This enables LwIP to increase the receive window so new data can be send to us.
3. When you have successfully handled the received data you should return ERR_OK.

```
err_t hello_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err)
{
   u8_t *state = (u8_t *)arg;
   u16_t len;

1. if (p == NULL)
      if (*state == 255)                  /* close send */
         hello_end(pcb, state);
      else                                /* close not yet send */
         *state |= 2;
2. else
   {
      len = p->tot_len;
      pbuf_free(p);
      tcp_recved(pcb, len);
   }

3. return ERR_OK;
}
```

6.2.4.5 Sending data

As the tcp_write() and tcp_close() functions might fail you have to keep trying until you succeed. You can use the poll and sent functions for this purpose but you could also use a different thread or the background loop for this purpose. You can use the status variables from the argument to remember what you wanted to send.

The last argument of tcp_write() can be set to 0 or 1, when set to 1 the data you wanted to sent is copied and you can immediately reuse or clear the memory after the tcp_write() function returns successfully. When set to 0, you can clear or reuse the memory when the data has been acknowledged indicated by sent call-back.

LwIP can combine multiple small pieces of data queued by tcp_write() into one tcp packet. This is done by waiting a while after a tcp_write() before actually sending. If you do want to send data immediately you can call tcp_output() after the tcp_write().

```
const char *helloworld = "hello world\n";

err_t hello_poll(void *arg, struct tcp_pcb *pcb)
{
   u8_t *state = (u8_t *)arg;

   if ((*state & 1) == 0)          /* hello world not yet send */
      if (tcp_write(pcb, helloworld, 12, 0) == ERR_OK)
         *state |= 1;

   if (*state == 3)                /* close received and hello world send */
      if (tcp_close(pcb) == ERR_OK)
         hello_end(pcb, state);

   return ERR_OK;
}
```

6.2.4.6 Closing a tcp connection

There are three ways for a connection to close. You requested the close yourself, the other side requested the close, or an error has occurred and the connection is aborted.

1. When a connection is aborted, LwIP clears the pcb and afterwards calls your error function. In your error function, you should clear the memory you used and prevent your threads or background loop from using the deleted pcb.
2. When the other side has sent you a close, you receive an empty data segment. You can still send some final data and afterwards (in the example indicated by state=3) you have to call a tcp_close(). When you succeeded in sending a close, you should cleanup. (see also the paragraph about receiving)
3. When you are the one to send the fin flag first, in our example done by defining a poll function to be called after 5 seconds that calls a tcp_close(), you have to wait for the other side to send a fin flag back. Meanwhile you can still receive data. When the fin flag arrives, you can cleanup.
4. To cleanup, you should free the memory you used for status information. The TCP connection could still be in the CLOSING, of LAST_ACK state (see TCP state diagram) waiting for a last ack, This means LwIP could still try to use one of the call-back functions, although our status memory has been cleared. To prevent this you should set all call-back functions to NULL.

```
   void hello_err(void *arg, err_t err)
   {
1.    mem_free(arg);
   }

   err_t hello_poll_close(void *arg, struct tcp_pcb *pcb)
   {
      u8_t *state = (u8_t *)arg;

      if (tcp_close(pcb) == ERR_OK)
      {
         if ((*state & 2) == 2)       /* close received */
2.          hello_end(pcb, state);
         else                         /* close not yet received */
3.          *state = 255;
      }

      return ERR_OK;
   }

   void hello_end(struct tcp_pcb *pcb, u8_t *state)
   {
4.    tcp_err(pcb, NULL);
      tcp_recv(pcb, NULL);
      tcp_sent(pcb, NULL);
      tcp_poll(pcb, NULL, 0);
      mem_free(state);
   }
```

6.3 Network interface driver

To use lwip, you need one or more network interface drivers. There are a few examples available including a PPP and SLIP for serial connections and a few Ethernet drivers. When you want to use an Ethernet controller, there is an ARP implementation available you can use for your driver. For the PPP and SLIP drivers you just need to define some in and output routines in “sio.h”.

In this chapter, I describe the Ethernet driver I made for the Motorola ColdFire 5282 (MCF5282) Ethernet controller and what could be done to improve it. As the documentation on the usage of LwIP is very limited, I will also explain how you can implement a network interface driver for other devices.

6.3.1 MCF5282 Ethernet driver

For the Motorola ColdFire 5282 microprocessor, I have written a driver for the internal Ethernet controller. The driver initializes the Ethernet controller and copies data between the lwip buffers and the Ethernet

controller's buffers. It would be more efficient to merge the two different buffer types so the data does not have to be copied. There was no time for me to implement this, but I will explain how this could be done.

6.3.1.1 Zero copy

LwIP uses buffers called pbuf and each frame is spread over a chain of one or more pbuf's. The MCF5282 Ethernet controller uses a ring of buffer descriptors; each buffer descriptor points to a block of data and a frame can be spread over multiple data blocks.

To send data without first copying it you just have to copy the data and length fields from each pbuf in the pbuf chain to a consecutive buffer descriptor. You have to increment reference count of the pbuf with pbuf_ref(), so the pbuf won't be deleted until the Ethernet controller is done with it. When the Ethernet controller informs you a frame has been sent you have to check how many buffer descriptors are cleared and clear the corresponding pbufs (decrease the reference count) using pbuf_free().

To pass received data to the LwIP stack without copying it you can use a special pbuf type called PBUF_REF. When a frame is received you can allocate a pbuf for each block of data described in a buffer descriptor using pbuf_alloc() and chain the pbufs using pbuf_cat(). To know when the lwip stack and the user are done with the frame you have to increase the reference count from the pbuf chain with pbuf_ref(). And regularly check the reference count of the pbuf's to see if the reference count has been decreased back to one. When this is the case you are the only one using it and you can clear both the buffer descriptors and the pbuf chain. You have to clear the first non-empty buffer descriptor to be used by the Ethernet controller and make it point to the freed block of memory. This could be an other descriptor than originally used because the first packet received is not always the first packet freed.

Another way to pass the received data to the LwIP stack without copying it is by using the pbuf's from the PBUF_POOL. The LwIP stack supports a pool with a predefined number of pbuf's that have a predefined length. You can use these pbuf's by making all receive buffer descriptors point to a location within the data segment of a pbuf from the pbuf pool. Because the data segments from the pbuf pool are not aligned to sixteen bytes you have to align each buffer descriptor pointer within the data segment of the pbuf. This is not very memory efficient as each buffer could lose fifteen bytes. You have to regularly check for freed pbuf's in the pool so you can reuse them.

6.3.1.2 DMA errors

From experience of 3T, I learned there seems to be a small problem in the ColdFire 5282 with DMA. When data is written to a memory block from the processor and a DMA device simultaniously, data might get corrupted. This results in a TCP checksum error on frames with no checksum error on the Ethernet frame. This means you cannot ignore the TCP and IP checksums even if the lower layer can ensure only correct frames will be delivered. A possible solution might be to use separate memory banks for DMA and CPU writes.

6.3.1.3 Pointers misaligned

Another problem I encountered once is that the send buffer descriptor pointer of the driver and the Ethernet controller somehow got misaligned. The result was the driver kept waiting for buffer x to be cleared, while the Ethernet controller was waiting for buffer y to be filled. This could be easily remedied by checking if the other buffers are also full, when the driver encounters a full buffer descriptor. If this is the case and the buffer descriptor that should be emptied first is still in use, you should reset the driver and Ethernet controller.

It is not unthinkable other "impossible" events occur so it could build in some checks that result in a reset when something goes terribly wrong.

6.3.2 Initialisation

In the network interface driver you have to implement an initialisation function that will be called from netif_add(). In this function, you have to define a two-letter name describing the interface, the function that should be called to send an IP packet, the MTU and some flags. You also should initialise your hardware.

If the device is an Ethernet controller you should also call etharp_init() and define a hardware (MAC) address and the function to be called by the ARP protocol to send an Ethernet frame. You should also make sure a etharp_tmr() is called every 4 seconds.

6.3.3 Sending

In the function you defined to sent IP packets, you should simply make sure the IP packet is sent to the hardware. In case of an Ethernet device you should pass the packet to etharp_output(). This function will create an Ethernet header in front of the IP packet and send the Ethernet frame to the output function you defined for sending Ethernet frames. In this second output function, you send the frame to the hardware.

6.3.4 Receiving

When you have received an IP packet from your hardware, you should send it to the input function defined in the netif structure. In case of an Ethernet device, you should first check the type field. When the Ethernet frame contains an ARP packet you should send it to etharp_arp_input(), when the frame contains an IP packet you should first call etharp_ip_input() and remove the Ethernet header using pbuf_header() before you send it to the input function. It would also be wise but not strictly necessary to check the Ethernet checksum and the Ethernet destination field and drop the broken or unwanted packets. In case of the MCF5282 Ethernet controller, this is done by the hardware.

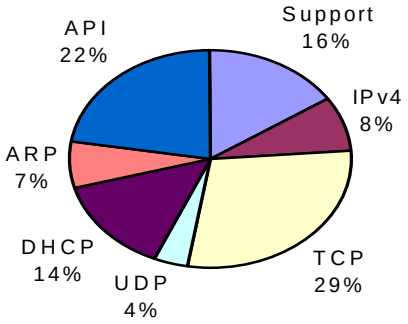
6.4 Configuration an Tuning

There are a lot of options, you can configure in LwIP. There are a number of parts you might not need and can disable and there are many buffer sizes you can change. For more information about the exact memory usage of all parts of the LwIP stack on the ColdFire I have included an overview of all the memory used by each part in appendix 13.5.

6.4.1 Disable parts

- The Berkeley-alike socket API is a very large part of the LwIP code (22%) and you need an operating system to use it. The API also uses some extra memory for buffers and messages. Using the socket API is a bit more common, easier and it might save some implementation time. However, when you have a very limited amount of memory available you can do exactly the same without the socket API.
- When you are not using DHCP or any other UDP protocol you could leave out UDP, however this won't save a lot of code size (1.55Kb on the ColdFire)
- DHCP is disabled by default. DHCP enables the board to automatically connect to most networks. You could also put the IP configuration in the implementation itself or ask the user to setup the IP configuration manually every boot every boot.
- LwIP can keep some information about the number of packets sent, the number of errors and memory usage. Disabling the stats will save 276 bytes of ram and 2.59Kb of code; it also saves the processor the small trouble of counting.
- Using a zero copy network interface driver saves a lot of RAM, as you do not need extra buffers for the network interface. It is also much more efficient because copying data is relatively CPU intensive.
- The checksum calculation is also cpu intensive and you could disable the checksum checking for incoming packets. You have to make sure that broken packets are discarded by the lower layers so no broken packets will arrive at your implementation.

module	size (kB)	percentage
Support	6.37	15.79%
IPv4	3.32	8.23%
TCP	11.47	28.43%
UDP	1.55	3.84%
DHCP	5.76	14.28%
ARP	2.84	7.04%
API	9.03	22.38%
total	40.34	100.00%



6.4.2 Buffer sizes

LwIP has three ways of allocating RAM. It is important to use values that do not contradict each other, for example a maximum segment size larger then the output buffer size is not very useful and could even lead to errors. The easiest way to know which values are most suited for your application is by testing it in the way you think it would typically be used and looking at the memory statistics to see how much memory and structures were used. You can set the values to a safe margin and if still memory errors occur you have to make choices between what is more important.

6.4.2.1 mem

LwIP has its own memory heap, controlled by mem_alloc() and mem_free(), this heap is mainly used for storing outgoing data and by the user application.

You could make sure the heap will never overflow by setting a high value for MEM_SIZE and low values for the number of TCP connections (MEMP_NUM_TCP_PCB), the size of the TCP send buffer (TCP_SND_BUF) and the number of TCP segments (TCP_SND_QUEUELEN). This will not make sure you always have room to send data because the send buffer or maximum number of segments might still be reached. This just prevents connections from not being able to send data, when other connections have used all the available memory.

In most cases you can just set the memory size high enough to allow a few connections to have their send buffers full while you could have some other connections open that are idle. A full memory heap is not a big problem because LwIP will just return an error to your application and your application can try again later.

I have set the send buffer for the connections to 2048 bytes, and the memory heap to 16000 bytes. This means the memory heap will take up 25% of the memory and about 7 connections can have their send buffers full before the heap will overflow. Increasing the buffer size will speed up sending because more data can be on its way at the same time, but it decreases the number of connections that can send at the same time without the heap getting full.

6.4.2.2 memp

For structures like PCB's, segments and ROM pbuf descriptors, LwIP uses a predefined amount of structures controlled by memp_alloc() and memp_Free().

The number of TCP listeners and UDP connections used is often known at compile time so you could set the corresponding memp values to exactly the number you need. The number of TCP connections that will be open at the same time is harder to predict. To help prevent connection failure when too many connections are established or in the time_wait state (a closed connections that is not sure the other side has successfully closed too), LwIP will automatically overwrite the oldest connection in time_wait state or another connection with a priority lower or equal to the new one.

There are also a number of pbuf structures in the memp memory. They are used to point to read only blocks of memory, so if you often send data directly from ROM you should allocate a lot of these pbuf structures, when you will never send data from ROM you can set this number to zero.

6.4.2.3 pbuf

For incoming frames LwIP uses a pool of pbuf's with a predefined length. They can be allocated and freed with pbuf_alloc(,PBUF_POOL) and pbuf_free(). The number and size of these pbufs should accommodate a number of receive windows. When these pbufs get full, data is lost and has to be retransmitted. You depend on congestion algorithms to slow down the amount of traffic sent to you. So a small receive window (TCP_WND) is advised, but a too small window will slow down the receiving speed.

In some cases, when you have your own buffer pool, or when you know the size of an incoming frame beforehand, another type of pbuf's is more suited for your network interface driver. In this case, you can just disable the pbuf pool by setting PBUF_POOL_SIZE to 0.

6.5 IO routines

I found out that even a very simple program, like an echo server, using the raw API of LwIP, was very large and complicated. This means creating your own applications using the raw API would take long and the chance of making mistakes would be rather large.

The example applications were running completely on interrupt sources which means they are running in a high priority level. It would be saver to run your application on a lower interrupt level so LwIP keeps running when an application hangs or asks a lot of processing time.

It would also be nice to be able to use the same IO function, like printf(), used for the serial console, for TCP an UDP connections.

Because of these three reasons, I decided to expand the IO routines from the serial console with routines to handle TCP connections. And to implement a background loop where you can register you applications to run in a low priority level. You can select a TCP or serial connection for each background application so the IO routines are automatically mapped to the connection of your choice.

It would have been nice to use locking functions to send and receive; however, this would mean the complete background loop locks. The only way to prevent this is by using different stacks for each application. This would come very close to writing your own operating system and that is not what we wanted. Instead, I use polling functions to check if data has arrived, or can be sent.

6.5.1 Usage

I have created a structure to describe a "connection", the same structure is used for a serial connection and a TCP connection, and it could be used for an UDP connection. You can use a pointer to this structure to select the connection you want to use (using io_select()). There are default pointers present for uart0, uart1 and uart2.

6.5.1.1 Main loop

For using the main loop you only need two functions, main_loop_add() and main_loop_del(). In main_loop_add() you register your main loop function with a connection and an argument. It returns a pointer to the structure describing the main loop function. The main loop functions will be called one after another, to keep the loops cycling fast they have to do only a short piece of code at a single call of the function. The main loop function can be removed with main_loop_del().

6.5.1.2 TCP

To open a listening TCP connection using the IO routines you call io_tcp_listen(), with the port number, acknowledge and close functions, and optionally an argument that is given to the acknowledge function when a connection is successfully opened. In your Acknowledge function, you need to return the argument for the close function.

The example below does exactly the same as in the raw API example. It opens a listener to port 22, when a client connects hello_ack() is called and “hello world” is send to the client. Also the hello_ack() function registers a main loop to receive the incoming data and returns the main loop function pointer so the close function knows what main loop function to remove. If no main loop could be registered the connection is closed. In the main loop function, we check if data has arrived. If so, we receive and acknowledged the data.

```
void main()
{
    io_tcp_listen(22, NULL, hello_ack, NULL, hello_close);
}

void * hello_ack(void * arg, struct con *conn)
{
    struct ml *ml;
    printf("hello world\n");
    ml = main_loop_add(NULL, conn, hello_main);
    if (ml == NULL)
        io_tcp_close(conn);
    return ml;
}

void hello_main(void * arg)
{
    if (char_present())
    {
        in_buf();
        in_buf_ok();
    }
}

void hello_close(void * arg)
{
    main_loop_del(arg);
}
```

The other example used was to open a connection to a server, send a “hello world” and close the connection. With the IO routines you can use io_tcp_add() to open a connection. This function will lock until it succeeds or fails. This may take a few seconds depending on the number of retries you have configured. You have to give an IP address and port number, but in our case, no argument or close function is needed as we close the connection ourselves. With io_select() we switch to the just created connection and we send the “hello world” and close the connection.

```
void main()
{
    struct con *old;
    struct con *new;
    struct ip_addr ip;

    IP4_ADDR(&ip, 192,168,0,112);
    new = io_tcp_add(&ip, 22, NULL, NULL, NULL);
    if (new != NULL)
    {
        old = io_select(new);
        printf("hello world\n");
        close();
        io_select(old);
    }
}
```

6.5.2 Implementation

6.5.2.1 Sending

LwIP can combine multiple small data segments into one TCP frame to decrease the number of packets that need to be sent. For every call to tcp_output() a new segment is made and only a few segments can be combined. Therefore, it would not be very efficient to send every byte to LwIP separately. This is why I implemented a small output buffer in the IO routines that combine the bytes send to tcp_out_char() and send them to LwIP when the buffer is full or when io_tcp_flush() is called. When you try to send a char, and we fail to empty the buffer to the LwIP stack, we have a problem because we cannot send back a failure. This is why I decided to wait for empty buffer space when we are in low priority and to throw the char away when we are in high priority. More about this dilemma in the last paragraph of this chapter.

6.5.2.2 Receiving

When a data pbuf is received, the pointer is stored in the connections structure, if a pbuf pointer is already stored in the structure the pbuf's are concatenated. When the application calls in_char() a byte is removed from the pbuf, acknowledged and returned. When in_buf() is called, the first pbuf is removed from the pbuf chain and returned, but not yet acknowledged. You have to call in_buf_ok() re acknowledge and remove the pbuf (a window update is send).

6.5.2.3 Closing

As explained in the raw API description, there are three ways to close a connection. By request of the application, by request of the other side or after an error occurs. Because a remote close or an error can come in while a main loop function is using the connection and we cannot remove a main loop function while it is running we have to actually delete the main loop functions at the main loop.

When an error occurs the connection state is set to closed so from that moment no more data will be send. A main loop function is registered to actually delete the connection and to call the connection's close function to free the memory (and main loop functions) used by the connection. When this is done, the main loop function deletes itself and possibly some main loop closing functions that were interrupted by an error before they could successfully close.

In the IO routines, no difference is made between the application and remote sides closing the connection. This means no new data can be sent after the other side sends a close but the data that is already in the buffers will be sent. A main loop function is registered to wait for the buffers to clear and to keep trying to send a close until it succeeds, when the close is sent to LwIP the main loop function calls the close function registered in the connection to free the memory (and main loop functions) used by the connection. When this is done, the main loop function deletes itself.

6.5.2.4 Locking

When you are using a background loop instead of an operating system you should never use locking functions because all background loop functions will have to wait for your lock. The io_tcp_add() function I implemented will lock for only a few seconds (so the implementation will recover) but still it would be better to use a call-back function for this purpose. A bigger problem is the out_char() function used by printf(). Printf() doesn't return a failure or success flag so you can only choose to lock, or to lose the data, when no more room is available. Both these solutions are not what you want so the only solution left is not to call printf() when not enough room is available. You could check this in the IO routines before a main loop function is called or before each printf() itself. If you make sure enough memory is available before you call the printf(), and you don't fill the same memory from interrupt functions, you can safely use printf.

6.5.3 Conclusion

The IO routines I wrote make it a lot easier to handle more than one connection and more than one application at a time. However, there are many additions you could make to improve the routines. I do not think it would be wise to enhance the IO routines a whole lot because when you want a lot more than these routines now do, it would be simpler and faster to use an existing operating system with the socket API.

In some cases, it would be more efficient to use the raw API without any enhancements. It could be your application never uses more then one connection simultaneously. Also in some applications, the TCP event can be handled very quickly. In these cases, there is no need for the extra code as there is no need to divide the processing time.

6.6 Testing applications

To intensively test LwIP, I wrote multiple applications and tried some of the existing applications for LwIP. I discovered that there are many things you can do wrong, and that is why I wrote an extensive explanation on how to use LwIP. For instance, you have to know how exactly to close a connection and when to stop using it. I also learned you have to use the right configuration and know not to send your data byte for byte.

I started testing LwIP using an http daemon and echo server that worked with the raw socket API. The amount of code needed for these examples was large and the code was not very well commented. This is why I decided to create my own IO routines and test it with some example applications.

6.6.1 Echo server

While creating the IO routines I used a very simple echo server as a test. It is just a routine that waits for incoming connections, accepts the connections, and sends and receives data.

6.6.2 Shell server

While working with the LwIP stack I felt the need for some debug information on demand. This is why I wrote a very simple shell to print all kinds of status information like connection state's LwIP counters, Ethernet controller counters, main loop functions, IP configuration and more. These functions can be used to debug later applications.

6.6.3 Proxy server

As an easy way to test the stack with multiple connections and a lot of traffic, I created a proxy like application. When you connect to this application, it opens a connection to the local proxy server and routes all the data between the two connections. This meant I could setup my web browser to use the ColdFire as a proxy server. While opening a website multiple connections are opened and closed and data is transferred in both directions. With a lot of surfing preferably at pages with a lot of pictures I had a very simple way to stress test the stack with all kind of events. After some debugging all data arrived correctly, no errors occurred and after the connections were closed all memory was freed. Even when a lot of packets got lost, the transfer rate slowed down but no errors occurred and eventually all data arrived correct and in the right order.

Only one problem I did not fix and should be fixed when you really want to use this application. When both the input and output buffers are full no new acknowledge can be received to clear the output buffers and because no data can be sent, the input buffers are not cleared by the proxy application. This is why your application should always acknowledge received data, or abort the connection, within a small time-span. If not the memory can not be cleared and the stack has to wait for a few timeouts and (failing) retransmissions before it wil abort the connection by itself. During this time no traffic is possible!

6.6.4 Test applications

I also wrote some test application to do some measurements. These applications are described in a later chapter.

7. Quadros

Quadros offers a TCP/IP stack called “TRXC Quadnet”. It includes an operating system and socket API. They have a free version available that consists of a Binary image of over 200kB. The Binary image contains a default application that can download user applications so you can upload your new applications over the Ethernet connection. A small example application is available.

It soon became clear the implementation was not very stable and would stop working regularly without any apparent reason. Even when running the default application Quadros would not run stable. There was nothing in the documentation to explain this, but I do not think it is supposed to be this unstable. The binary version I used was specially compiled for the hardware platform I was using and is included by the hardware supplier. As the LwIP stack did not show any problems with the hardware except for the MII, I do not think it could be a hardware problem. When you would actually buy a version of the Quadros TCP/IP stack support is included and they would probably fix this problem. However, it does not vote well for them to supply a faulty demo version.

7.1 Test application

Still I decided to test the stack more intensively and write my own test application. The example application just started some binary services so no example was available on how to write your own application. With some searching I found out I could use the Berkeley-alike socket API to implement my own application, unfortunately some small parts I needed to use a select were left out so I had to implement them myself. I discovered the implementation was not exactly the same as the Berkeley-alike socket API, but it would probably work.

My implementation did work however often the connections did not close as expected and I often had to reboot the system. I implemented one application that could switch between incoming, outgoing and bidirectional traffic and could print the load to the serial terminal. To calculate the load I ran the same piece of code as I used in the LwIP stack, in the null task of Quadros. And I initiated my own timer to time the seconds.

8. Test setup

To get a better idea of the difference between the different TCP/IP stacks I have devised some tests. The first thing I wanted to know is how stable a stack is. I also wanted to know how much data an implementation could handle and how much processing time it would take. The creation of the test applications themselves also tells a lot about the stacks.

8.1 Stability

8.1.1 Too many connections

The first and simplest test was just opening more connections than a stack could handle and closing them. The stack should not give any errors and it should cleanup the connections correctly.

8.1.2 Too much incoming traffic

To check what would happen if the input buffers get full we have to send a lot of data to the stack. Because a connection should have a window size smaller than the input buffers, I used multiple connections to fill up the input buffers. I displayed the transfer speed for each connection during half a second for twenty-five seconds. It would be nice if the connection speed would be divided evenly over the connections. The connections should not fail and you should be able to close all connections afterwards, even when some buffers are still full. This test also shows how an implementation handles packet loss because received packets that cannot be stored will be discarded.

8.1.3 Too much outgoing traffic

I repeated the same test as described above with outgoing traffic. As the outgoing queue of a connection also has a size smaller than the output buffers, (and probably smaller than the receive window of the other side) again multiple connections are needed. No errors should occur and it would be preferred all connections create an even amount of traffic.

8.1.4 To much bidirectional traffic

The last test to test the stability of a TCP/IP stack is to test a lot of traffic to an echo server. This tells us if the stack can handle a lot of data without errors

8.2 Performance

8.2.1 Maximum speed

To get an indication of how much data a TCP/IP stack can handle I decided to test the maximum speed for, incoming, outgoing and bidirectional traffic. I tested multiple times with an increasing amount of connections to see what effect the number of connections has on the total amount of traffic a stack can handle.

8.2.2 Load

Something else that interested me was how much resources an implementation needs to handle a certain amount of traffic. To do this I measured the number of times a piece of code could cycle, during one second, on the ColdFire, with all interrupts disabled. I compared this value to the number of times the same piece of code could cycle, during one second, in a background loop, while the stack is handling a certain amount of traffic. I repeated the test with different amounts of traffic and with traffic in both directions separately and simultaneously.

8.3 Implementation

8.3.1 Test Client

To test the stacks I needed a test application to run on a workstation to open multiple connections with a specified amount of traffic. I could not find a suited test application so I decided to write one of my own. My only experience in programming on a Windows workstation was in Java but in my experience, Java is not very fast and not very stable. As the workstation I was working on was also a bit limited I was afraid, a Java implementation would influence the test results. This is why I decided to write my test application on a Linux workstation using C++.

I created a class to open a test connection with a specified amount of traffic. To limit the transfer speed I took the sum of the sent and the received data, and divided this by the amount of time the test was running. I waited for the actual speed to be lower than the wanted speed before sending or receiving another data fragment. The TCP/IP stacks on the ColdFire would not immediately slow down sending when the test class slows down the receiving, because the buffers have to be filled first. This is why the test had to wait a while before taking measurements.

I made an option to enable or disable the outgoing traffic from the test connection class. By enabling the outgoing traffic and connecting to a dummy server on the ColdFire I could test incoming traffic on the ColdFire. By enabling outgoing traffic to an echo server on the ColdFire I could test bidirectional traffic. And by disabling outgoing traffic and connecting to a traffic generating server on the ColdFire I could test with outgoing traffic from the ColdFire.

I created a number of functions that make use of this test connection class to open multiple connections to the ColdFire. One of these functions opens multiple connections and measures the total speed. Another opened one connection and generated different amounts of traffic while requesting the load from a special server on the ColdFire that returns the load when it receives a message. Another test function I created just opened a lot of connections and tried to transfer a lot of data over all the connections. It showed the amount of traffic it could generate for each connection during half a second.

8.3.2 Test servers

8.3.2.1 LwIP raw API

To run tests on the ColdFire I needed an echo server that sends all received data back to the client, a dummy server that receives and ignores all incoming data and a traffic generating server that keeps trying to send as much data as possible. When a number of Clients were connected to the traffic-generating server, I could not connect another one because the already connected clients were using all output buffers. To remedy this I toggled the sending of the data on and off when a byte was received so the test client could decide when the test servers should start and stop sending.

Another test server I needed was a server that started the load test function in a background loop and returns the last measured load to the client when a byte is received. This enabled the test client request the load from the Coldfire.

8.3.2.2 LwIP low priority IO functions

I created the same three traffic-handling servers using my IO functions to test the difference between working in low and high priority levels. The load test would not be very useful as it would run in the same priority level as the IO functions and influence the results.

8.3.2.3 Quadros

As the free Quadros version is limited to only four TCP connections including the listening connections I created only one test server that I could switch between sending, receiving and echoing mode with the serial terminal. The load was also requested using the serial terminal.

9. Test results

Unfortunately, I could test the implementations only at 10Mbit half duplex due to problems with the MII bus on the MCF5282 evaluation board I used. Running on 100Mbit full duplex would give very different results as no frames would be lost on transfer and the Ethernet controller would be able to send Ethernet pause frames to prevent buffer overruns. The higher number of errors and buffer overruns did however give more interesting test results.

Because of the limited time available and the limitations and instability of Quadros I could not test Quadros as intensively as LwIP.

9.1 Stability

9.1.1 Too many connections

LwIP can handle as many connections as you have defined. When LwIP has too many connections open to open a new one, it can automatically abort an old connection. LwIP does this by first looking for a connection in time-wait state (see appendix 13.1 TCP state diagram), if no such connection is available it will abort the connection that has been idle the longest and that has a priority lower or equal to that of the new connection. This makes sure you can always connect to a TCP listener with a high priority. I tested this by opening to many connections with different priorities and LwIP reacted exactly as it was supposed to do.

In the free Quadros TCP/IP stack I could only test 3 connections simultaneously. When you tried to open a fourth it just failed.

9.1.2 Too much incoming traffic

When the incoming buffers of a TCP/IP stack are full and a new segment arrives, the stack discards the new data segment (congestion). The sending side does not know this so it waits for a timeout before it retries to send this data. Meanwhile the data segments of other connections that did fit in the buffers will be acknowledged and these connections will keep transferring data.

In the test results (see appendix 13.2.1.1), you can see that using the raw API of LwIP, about four or five connections can keep transferring but the other connections are idle. Sometimes an idle connection gets lucky and is able to start transferring after a timeout. As a result, one of the other connections will soon fail. It would be preferred that when congestion occurs all receive windows are reduced so all connections will slow down. However, the receive windows are already very small and have to be reduced to a value below the maximum segment size. This would not be very efficient.

In appendix 13.2.1.2 you can see working with a background loop will give roughly the same result as the raw API. The only difference is that the resulting speeds are a bit more random. This is due to the fact that the window updates from the active connections are send from the background loop instead of immediately after arrival as in the raw API version of the test program.

Unfortunately, I could test Quadros only with three connections (see appendix 13.2.1.3). This meant no congestion occurred and nothing noticeable happened. The transfer rate was divided evenly.

9.1.3 Too much outgoing traffic

When the memory of LwIP is full or all TCP segment descriptors are in use, no new data can be send. When testing LwIP using the raw API this resulted in only a few connections being able to send. When an acknowledgement arrives a segment is cleared and the corresponding connection is notified so it can send new data. This means connections that are already sending have a much higher chance at sending new data than connections that have to wait for a polling event (see appendix 13.2.2.1). When using the low priority IO functions the transfer rates are divided more evenly but the total transfer rate is lower (see appendix 13.2.2.2).

When running the same test on Quadros the transfer rates were extremely low (but stable). I have not found any explanation for this (see appendix 13.2.2.3).

9.1.4 Too much bidirectional traffic

When the send buffers get full the raw API echo server I wrote for LwIP was not very efficient. Both the receive buffers and the memory was full so no acknolages could arrive and no memory could be cleared. This meant most of the time the connections were waiting for timeouts (see appendix 13.2.3.1).

In this case, the low priority IO functions (see appendix 13.2.3.2) were much more efficient.

Again the Quadros implementation (see appendix 13.2.3.3) was very slow. In addition, although I could only open 3 connections the transfer rates became unstable.

9.2 Performance

9.2.1 Maximum transfer speed

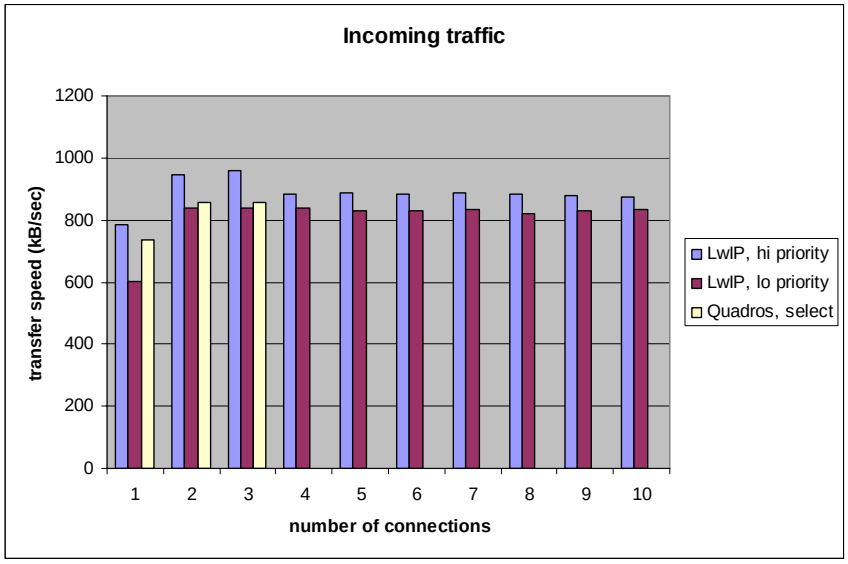
To measure the maximum transfer speed I opened one to ten connections and measured the maximum incoming, outgoing and bidirectional transfer speed. I compared the result from high priority raw API servers, low priority IO function servers and Quadros server using the socket API. The complete result can be found in the appendix and an overview is shown in the charts below.

9.2.1.1 Incoming trafic

When sending traffic to only one connection the transfer rate is clearly smaller than when you use multiple connections. Because I configured the receive window to only 2048 bytes the transfer speed of one connection using the LwIP stack was not optimal. The difference between one or more connections on the Quadros stack was smaller, this could indicate the Quadros stack uses a bigger receive window.

Although the Quadros stack, claims to be zero copy, while my LwIP stack did copy the data from the Ethernet controller to the stack, the LwIP stack appears to be faster using the raw API. Apparently, the handling of the packets has more influence on the transfer speed than being zero copy or not.

In appendix 13.3.1 you can also see that when only using 4 or less connections the transfer rates remained stable because the total size of the receive windows was smaller then the receive buffer. When sending traffic over more connections data gets lost and the transfer speeds become a bit less stable. (Testing on full duplex would probably have given a different result as the Ethernet controller could have sent a pause-frame and the frames would be delayed instead of lost.)

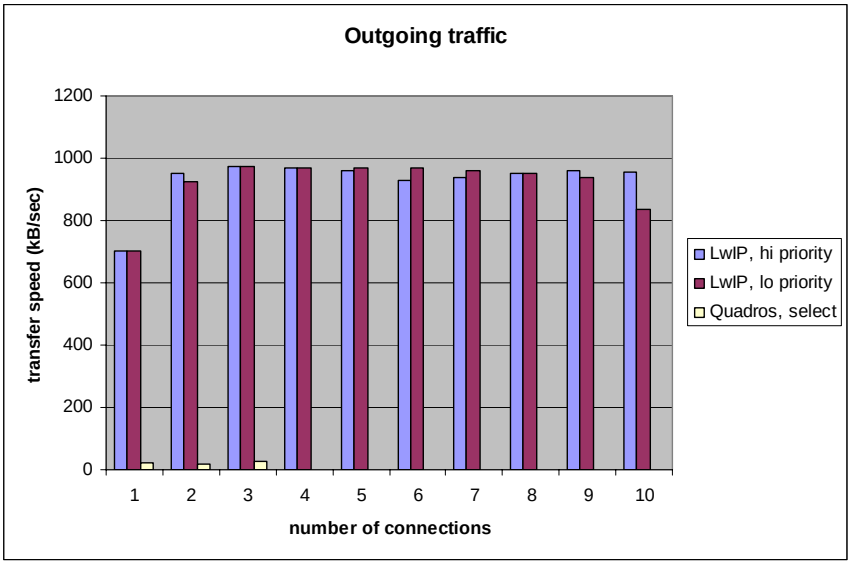


9.2.1.2 Outgoing traffic

When using only one connection for outgoing traffic again the maximum transfer speed was not optimal, the LwIP stack had an output buffer of 2048 bytes so it could only have 2048 bytes of data in the output buffers waiting to be acknowledged. This meant the stack had to wait for acknowledgements before sending more data.

This time the difference between using the high and low priority servers was very small. The Quadros stack hardly transferred any data. I do not know why the Quadros stack could not transfer the outgoing data any faster. It could be an undocumented restriction of the demo version or a mistake in the Quadros stack. I expect Quadnet could repair this problem rather quickly so the performance would be roughly the same as that of LwIP.

In appendix 13.3.2 you can see a little difference between the raw API version test applications and the low interrupt-priority main loop functions. The main loop divided the transfer speeds more evenly across the different connections.

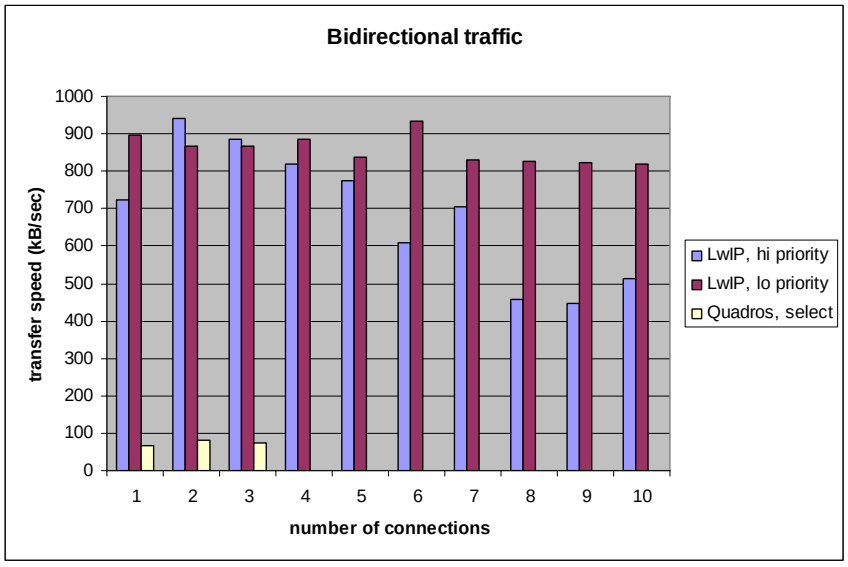


9.2.1.3 Bidirectional traffic

When testing with bidirectional traffic the results became very irregular, especially using the high priority raw API on LwIP. Even when I repeated the test many times, the results stayed unpredictable so the only thing you can tell from the actual values is that many packets got lost and a lot of timeouts occurred. (see appendix 13.3.3.1)

The low priority IO functions were a bit more efficient as they continuously tried to echo the received data and not only on timeout, sent, and receive events. When one connection gets a sent event it cannot always immediately send new data, as it may not have received data pending.

Again Quadros performance was very poor, hopefully Quadnet will fix this problem when you decide to buy Quadros.



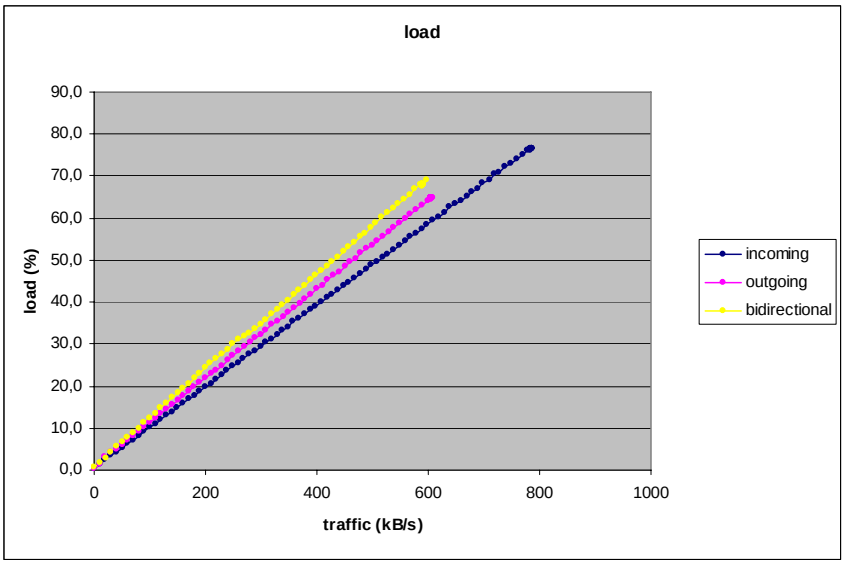
9.2.2 Load with certain amounts of traffic.

The load is measured by counting the number of times the background loop cycles, divided by the number of times the background cycles with LwIP disabled. An increasing amount of traffic is created and the actual traffic rate is compared to the load. The measurement is done 5 times and the average is used. Some values are ignored because they did not reach the target speed due to loss-timeouts. The actual values can be found in appendix 13.4 and a chart is made from the averages and shown below.

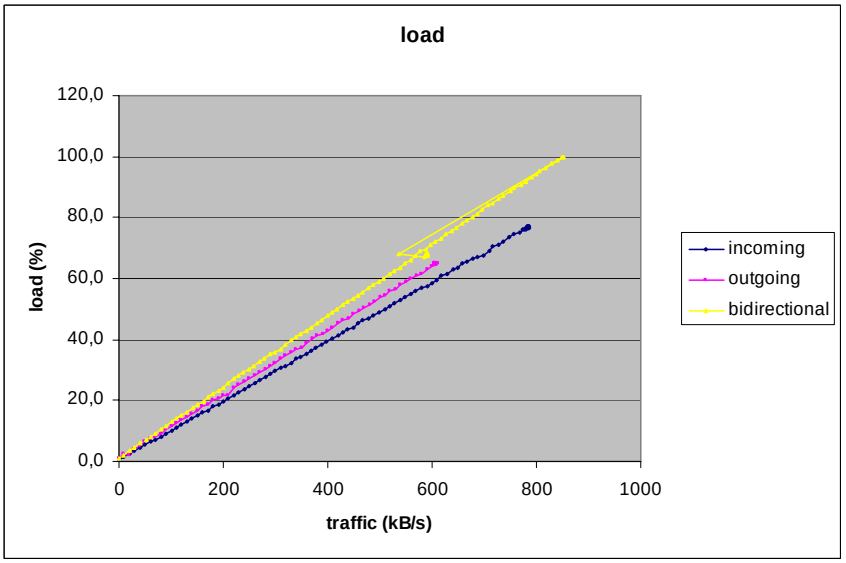
I have tested the load only on LwIP using the raw API. The low priority IO functions would run in the same priority level as the load counter so no meaningful measurement could be taken. I was not able to run an accurate load test on the Quadros stack because it only worked well at very low speeds or full speed incoming traffic.

As you can see, there is a linear correlation between the transfer speed and the load. When the transfer rate reached a certain maximum speed, it stayed at that level. The output buffers were full and could not be emptied until an acknowledgement arrives or the receive window was fully used and the test application could not send any more data until a window update was sent.

- The load for incoming traffic will increase with 0.97% for every 10 kB/sec.
- The load for outgoing traffic will increase with 1.06% for every 10 kB/sec.
- The load for combined bidirectional traffic will increase with 1.15% for every 10kB/sec.



One test ran I got a different result for the bidirectional traffic. Somehow, the buffers did not get full and the transfer rate and load kept climbing until the load reached 100%. When this happened, the buffers did get full and the transfer rate fell back to the same value the other tests would not cross. This proves that a small difference in timing can have a huge effect; it might be the difference between packets colliding regularly or never.



10. TCP/IP with or without an operating system

For most of the small TCP/IP stacks I found, you could choose between running with or without an operating system. To help with the choice of using an operating system I will list some advantages and disadvantages.

10.1 Advantages of using an operating system

- When you run a TCP/IP stack without an operating system the stack would run completely in one priority level. When the stack is overwhelmed with more data than it can handle, it will use all the CPU time it can get. This means applications in the background loop will be overtaken completely. When using an operating system you can easily prevent this.
- When using an operating system programming gets a little easier as you can use locking functions to communicate with the TCP/IP stack. For instance, you could use a Berkeley-alike socket API. You can also make use of the operating system to separate each connection in a different thread so you do not have to worry about handling multiple connections simultaneously.
- Using an operating system makes your application easier to port to different systems. Most systems use operating systems so most applications are written to be used with operating systems.
- When something goes wrong in the TCP/IP stack the operating system will keep running and might be able to take action to recover.

10.2 Disadvantages of using an operating system

- You might not have enough ROM and RAM to run an operating system alongside your application, or it will decrease the efficiency of your program by using memory that would otherwise be used by the application.
- The operating system uses some CPU time, which reduces the amount of CPU time left for the application.
- An operating system increases the complexity of a system which means more things can go wrong and it is harder to find out where something went wrong.
- When using an operating system you have less control over the system, usually the operating system can do exactly what you want but in some cases it can not. When you are not using an operating system you can schedule the order of your functions any way you like, although it is a lot more work.
- When you want to integrate a TCP/IP stack with an existing application that does not use an operating system it could be easier to keep working without an operating system.

11. Conclusion

There are many TCP/IP stacks you can choose from and for each stack, you can make many configuration choices. I started with this document with four factors that play a role in choosing a TCP/IP stack. I will explain what I have learned about the TCP/IP stacks for each of these factors.

11.1 Ease of use

When you have little experience in working with microcontrollers, a commercial TCP/IP stack would be advisable. You could pay a company some money and in some cases, they will create a, ready to run, system for your microcontroller. All you need to do is create your application in a way that is much like creating an application for a workstation. For some commercial stacks, you will need to port it to the microcontroller of your choice yourself.

When you do have a lot of experience with microcontrollers, it would not be very hard to start using an open source implementation like LwIP. It will however take up some time to port the stack to your system, but you do not have to wait for someone else to create your port.

11.2 Stability

From commercial TCP/IP stacks you would expect stability would not be a problem. As we have seen with Quadros this is not always the case. You do get some support with commercial stacks so hopefully they will fix these kinds of problems.

When you want to use an open source implementation, you should look for a stack that is often used and intensively tested. LwIP is a good example of this. When errors do occur, you have to debug it yourself, or wait for someone else to solve the problem.

11.3 Performance

In paragraphs 9.2.1.2 and 9.2.1.3 we saw a big difference in performance. But while testing the Quadros stack, it did some times reach a speed close to that of LwIP, only never for long and not every time I tried. It appears there is some kind of problem in the Quadros version I tested, this could probably be fixed rather easily when you have access to the source code. When this problem is repaired, I expect little difference between the Quadros and the LwIP stack.

In addition, as all commercial stacks promise to be zero copy and very fast, I do not expect a huge difference between any of them. Tuning and configuration have more influence on the performance. Some commercial stacks like ARC RTCS supply a performance tool to help you optimising your configuration. In LwIP you can use a number of counters to optimise your configuration.

11.4 Cost

When you have little experience in embedded systems it could be cheaper to buy a commercial TCP/IP stack. For some commercial stacks, you can buy one licence for multiple projects. You may decide to pay for the support so you do not have to learn how to configure a TCP/IP stack yourself. Some commercial stacks come with tools to help you configure the stack and speed up the process of creating new applications; this can be useful when you create many simple applications.

When you do have experience with embedded systems or want to get some experience in embedded systems, it is cheaper to use an open source TCP/IP stack like LwIP. There are no indications that LwIP has any vulnerabilities or bugs, but you do have to be very careful not to make any mistakes in the way you use the stack, you will have to have a deeper understanding of what happens within the stack to be able to create stable system.

11.5 Final note

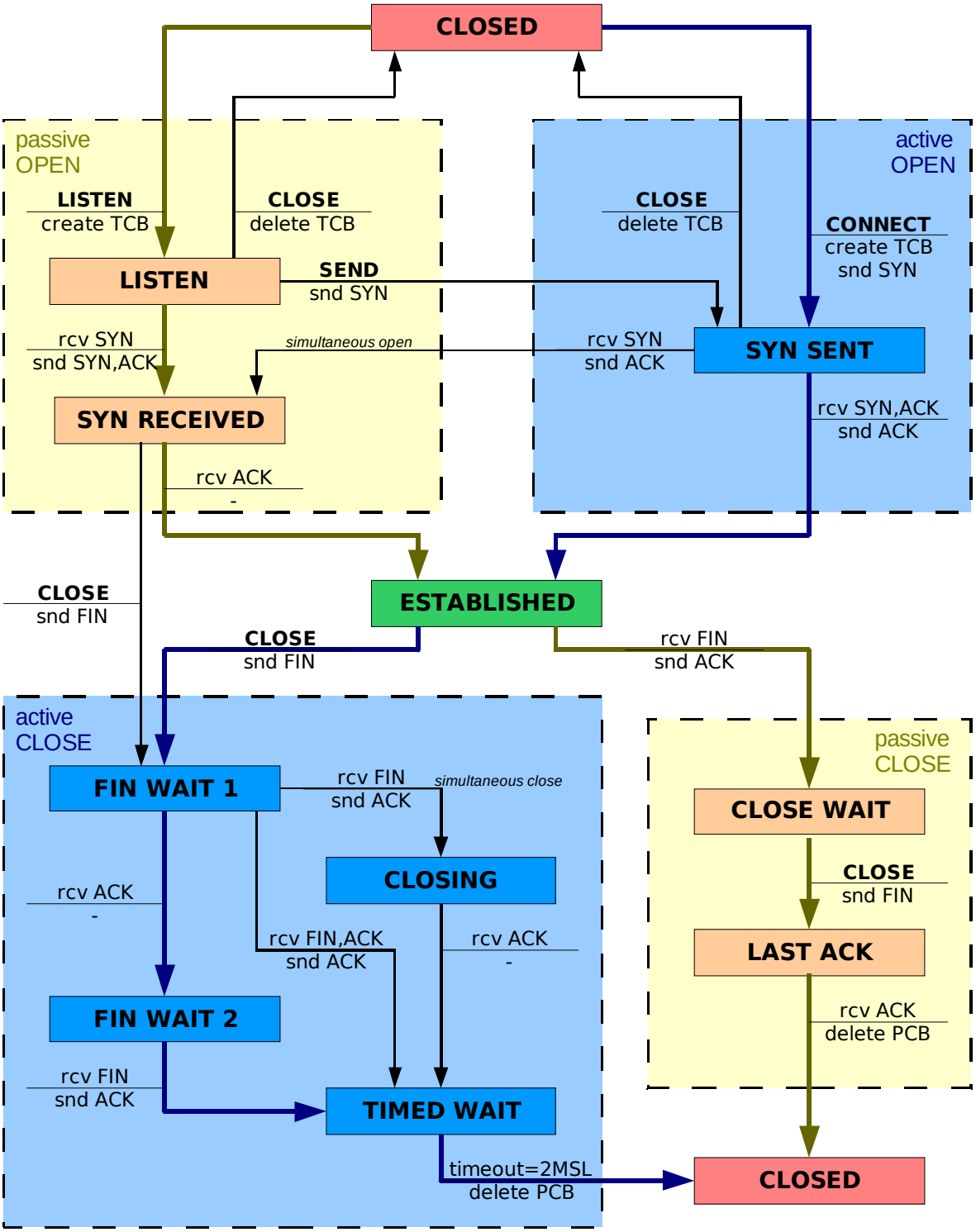
The choice for using an open source or a commercial implementation is very dependent on your experience and the application. You have to choose between handling it yourself and letting others worry about the details. The cheap way and the easy way. The risky solution and the less risky solution.

12. Abbreviations

ARP	Address resolution protocol, a protocol used by Ethernet to map IP addresses on Ethernet addresses.
DMA	Direct Memory Access, used by devices to communicate with memory without needing the processor
PPP	Point to Point Protocol, used for internet over serial lines.
SLIP	Serial Link Internet Protocol, used for internet over serial lines.
LwIP	Lightweight Internet Protocol, a TCP/IP stack I tested.
TCP	Transmission Control Protocol, for connection oriented byte stream connections over IP.
IP	Internet Protocol, for worldwide addressing.
CPU	Central Processing Unit, the main processor.
MAC	Media Access control.
API	Application Programming Interface, the interface between two pieces of code
PCB	Protocol Control Block, a block of information about a connection.
RAM	Random Access Memory, Memory that can read and written at any time.
ROM	Read Only Memory, Memory that can only be read, and sometimes written with some limitations.
MII	Media Independent Interface, interface between the physical interface and an Ethernet chip.

13. Appendix

13.1 TCP state diagram



13.2 Loss test

To test how the implementations handle buffer overflows I opened a few connections and generated more traffic than the implementation could handle. After starting the traffic, I waited a while for the transfer speeds to stabilise and took 50 measurements of the transfer speeds for each connections and the total transfer speed (all in kB/sec).

The transfer speeds for each connection is given in the first columns and the last column illustrates the total transfer speed. The last two rows give the average transfer speeds from all measurements and the percentage of measurements that indicated zero traffic.

13.2.1 Incoming traffic

13.2.1.1 LwIP high priority raw API

```
open 10 connections, check transfer rate every 500 ms 50 times.

  1  :    0  188  196  160  194  111    0    0    0    0  ->  851
  2  :    0  129  207  121  209  211    0    0    0    0  ->  880
  3  :    0  194  198  139  129  200    0    0    0    0  ->  860
  4  :    0  223  223  225    0  223    0    0    0    0  ->  896
  5  :    0  223  223  223    0  223    0    0    0    0  ->  894
  6  :    0  207  149  202   82  207    0    0    0    0  ->  849
  7  :    0  115  182  200  200  202    0    0    0    0  ->  900
  8  :    0  229  227  231    0  231    0    0    0    0  ->  919
  9  :    0  217  217  215    0  215    0    0    0    0  ->  866
 10  :    0  219  221  219    0  223    0    0    0    0  ->  884
 11  :    0  223  223  225    0  223    0    0    0    0  ->  896
 12  :    0  223  223  221    0  219    0    0    0    0  ->  888
 13  :    0  207  207  207    0  211    0    0    0    0  ->  835
 14  :    0  184  203  115  176  203    0    0    0    0  ->  884
 15  :    0    0  217  215  215  215    0    0    0    0  ->  864
 16  :    0  186   49  219  219  219    0    0    0    0  ->  894
 17  :    0  221   29  221  223  223    0    0    0    0  ->  919
 18  :    0    1  202  198  194  200    0    0    0   62  ->  859
 19  :    0    0  215  219  215  219    0    0    0    3  ->  874
 20  :    0   15   66  198  203  202    0    0    0  156  ->  843
 21  :    0  221    0  223  219    0    0    0    0  219  ->  884
 22  :    0  217    0  215  219    0    0    0    0  219  ->  872
 23  :    0  209    0  211  207   27    0    0    0  205  ->  862
 24  :    0  203    0  205  125  207    0    0    0  113  ->  857
 25  :    0  223    0  221  188  219    0    0    0   41  ->  894
 26  :    0  192   35  203    0  207    0    0    0  205  ->  845
 27  :    0    0  215  219    0  215    0    0    0  215  ->  866
 28  :    0  172  202  111    0  207    0    0    0  205  ->  900
 29  :    0   19  223  219    0  219    0    0    0  219  ->  902
 30  :    0   52  200  202    0  202    0    0    0  203  ->  861
 31  :    0   21  211  215    0  215    0    0    0  215  ->  880
 32  :    0  139  209  131    3  211    0    0    0  176  ->  872
 33  :    0  200  119  203    0  203    0    0    0  145  ->  872
 34  :    0  223    0  219    0  219    0    0    0  223  ->  886
 35  :    0  223    0  223    0  223    0    0    0  223  ->  894
 36  :    0  202   72  156    0  207    0    0    0  205  ->  845
 37  :    0  121  200  174    0  202    0    0    0  202  ->  900
 38  :    0  213  211  211    3  213    0    0    0    3  ->  859
 39  :    0  223  227  225    0  225    0    0    0    0  ->  902
 40  :    0  215  213  217    0  219    0    0    0   19  ->  886
 41  :    0    3  221  219    0  219    0    0    0  223  ->  888
 42  :    0    0  215  215    0  215    0    0    0  215  ->  863
 43  :    0   31  217  215    0  219    0    0    0  217  ->  902
 44  :    0  207  205   50    0  203    0    0    0  209  ->  878
 45  :    0  215  213    3    0  213    0    0    0  211  ->  859
 46  :    0  211  213    0    0  213    0    0    0  215  ->  855
 47  :    0  223  221    0    0  219    0    0    0  219  ->  884
 48  :    0  227  227    0    0  229    0    0    0  227  ->  912
 49  :    0  209  211    0    0  209    0    0    0  211  ->  843
 50  :    0  221  219    0    0  221    0    0    0  219  ->  882

total:    0  161  164  174   64  200    0    0    0  113  ->  877
idle : 100%   8%  14%  10%  62%   4% 100% 100% 100%  36%  ->  53%
```

13.2.1.2 LwIP low priority IO functions

```
open 10 connections, check transferrate every 500 ms 50 times.

  1  :  280  256    7   19    3    0    0    0    0    0  ->  568
  2  :  160  313   23  309   25    0    0    0    0    0  ->  833
  3  :   60  305  129  302   13    0    0    0    0    0  ->  812
  4  :  249  158  266   96   70    0    0    0    0    0  ->  841
  5  :  290  229  166   25  109    0    0    0    0    0  ->  821
  6  :  176  276  182  168    1    0    0    0    0    0  ->  806
  7  :    3   37  392  213  196    0    0    0    0    0  ->  843
  8  :   52  217  147  156  256    0    0    0    0    0  ->  831
  9  :  139  319    0  166   84    0    0    0    0    0  ->  710
 10  :  205  221  178  180    0    0    0    0    0    0  ->  786
 11  :  309  215   80  223    0    0    0    0    0    0  ->  829
 12  :  205  311   68  109  111    0    0    0    0    0  ->  808
 13  :  123  296  143  133  141    0    0    0    0    0  ->  837
 14  :  309   15  282   82  145    0    0    0    0    0  ->  835
 15  :  176   88  254  233   76    0    0    0    0    0  ->  829
 16  :  156  178  160  170  168    0    0    0    0    0  ->  835
 17  :  123  156  105  188  254    0    0    0    0    0  ->  829
 18  :  151  258  170  125   31    5   90    0    0    0  ->  833
 19  :  196  194  103   31    5  137  186    0    0    0  ->  855
 20  :  333  149   23  158    3   29  131    0    0    0  ->  829
 21  :  200   47  135  180    9   25  243    0    0    0  ->  841
 22  :  264  133   90    0  268   41   50    0    0    0  ->  849
 23  :   49  366  307   47   13   17   11    0    0    0  ->  813
 24  :   50  292   13   23  123   92  156    0    0    0  ->  753
 25  :   35  152    1  149  129  225  121    0    0    0  ->  815
 26  :  100  313    0  253   37  125    0    0    0    0  ->  829
 27  :   60   52    0  358  233    0  135    0    0    0  ->  841
 28  :   49  202  170    7   47  296   50    0    0    0  ->  823
 29  :  317   27  198    3    3  145  123    0    0    0  ->  819
 30  :  241  119  137    7   62   94   54    0    0    0  ->  717
 31  :    0  172    0    3  225   74  239    0    0    0  ->  715
 32  :  133  109   45  109    0  254  176    0    0    0  ->  829
 33  :   35  292  113    5   23  221  121    0    0    0  ->  813
 34  :  239  149   35    7    0  166  239    0    0    0  ->  837
 35  :   49  300    0  156    0  278   25    0    0    0  ->  810
 36  :    0  190  160  249    0  227    0    0    0    0  ->  827
 37  :    0   50    0  147  103  237  270    0    0    0  ->  810
 38  :   74  105  127    7  117  315   68    0    0    0  ->  817
 39  :  241   76  121   50   92  147   82    0    0    0  ->  811
 40  :  307  180    5    7    5  158   39    0    0    0  ->  706
 41  :  156  417  102    3   84   52    5    0    0    0  ->  823
 42  :   19   52  133  145  176    3  176    0    0  101  ->  809
 43  :  205    7  162  109  245    0   35    0    0   37  ->  804
 44  :  256   78  123   88   45  121   90    0    0    9  ->  813
 45  :    3  200  107   15    9  280    5    0    0  184  ->  808
 46  :    0  182  200  100    0  133  154    0    0   39  ->  810
 47  :    0  221   62    0    0   84  329    0    0   47  ->  745
 48  :    3  219    7  164    9   49    9    0    0  343  ->  808
 49  :    0  211    1   13  215   56  176    0    0   19  ->  696
 50  :    0  135    0    0   50  152  111    0    0  372  ->  823

total:  136  185  109  110   80   85   74    0    0   23  ->  804
idle :  14%   0%  14%   6%  16%  38%  38% 100% 100%  82%  ->  40%
```

13.2.1.3 Quadros, socket API

```
open 3 connections, check transferrate every 500 ms 50 times.

  1  :  284  284  282  ->  851
  2  :  282  282  282  ->  847
  3  :  172  172  172  ->  517
  4  :  282  282  284  ->  849
  5  :  284  284  282  ->  851
  6  :  282  282  282  ->  847
  7  :  286  284  284  ->  855
  8  :  284  284  286  ->  855
  9  :  284  284  284  ->  853
 10  :  280  282  282  ->  845
 11  :  288  284  284  ->  857
 12  :  284  286  286  ->  857
 13  :  280  282  282  ->  845
 14  :  284  284  284  ->  853
 15  :  286  282  282  ->  851
 16  :  282  286  282  ->  851
 17  :  284  282  286  ->  853
 18  :  282  284  282  ->  849
 19  :  284  282  284  ->  851
 20  :  284  282  282  ->  849
 21  :  282  286  284  ->  853
 22  :  284  284  284  ->  853
 23  :  286  284  284  ->  855
 24  :  282  284  286  ->  853
 25  :  284  284  284  ->  853
 26  :  286  284  284  ->  855
 27  :  284  282  282  ->  849
 28  :  282  286  284  ->  853
 29  :  282  282  282  ->  847
 30  :  284  284  284  ->  853
 31  :  280  278  278  ->  837
 32  :  284  286  286  ->  857
 33  :  288  286  288  ->  863
 34  :  282  282  282  ->  847
 35  :  282  282  282  ->  847
 36  :  286  284  284  ->  855
 37  :  282  286  284  ->  853
 38  :  282  282  284  ->  849
 39  :  288  286  284  ->  859
 40  :  286  286  288  ->  861
 41  :  280  282  278  ->  841
 42  :  282  282  286  ->  851
 43  :  284  284  282  ->  851
 44  :  282  282  282  ->  847
 45  :  282  282  284  ->  849
 46  :  286  284  282  ->  853
 47  :  284  284  286  ->  855
 48  :  282  284  282  ->  849
 49  :  286  282  284  ->  853
 50  :  280  282  282  ->  845

total:  281  281  281  ->  844
idle :   0%   0%   0%  ->   0%
```

13.2.2 Outgoing traffic

13.2.2.1 LwIP high priority raw API

```
open 10 connections, check transferrate every 500 ms 50 times.

  1  :    0    0    0  327    0  327  325    0    0    0  ->  980
  2  :    0    0    0  325    0  325  325    0    0    0  ->  976
  3  :    0    7    0  398    0  398  158    0    0    0  ->  963
  4  :    0    0    0  472    0  474    0    0    0    0  ->  947
  5  :    0    0    0  468   13  466    0    0    0    0  ->  949
  6  :    0  264    0  211    0  209  266    7    0    0  ->  961
  7  :    0  478    0    0    0    0  476    0    0    0  ->  955
  8  :    0  460    0    0   31    0  462    0    0    0  ->  955
  9  :    0  184    7  419    0    5  186   11    0    0  ->  815
 10  :    0    0    0  708    0    0    0    0    0    0  ->  708
 11  :    0    0    0  553  323    0    0    0    0    0  ->  876
 12  :    0   27    3  151  149  198  203  203    0    0  ->  937
 13  :    0    0    0    0    0  325  327  329    0    0  ->  982
 14  :    0    0    0    0    0    5  474  472    0    0  ->  953
 15  :    0    9  329    0   27    0  125  455    0    0  ->  947
 16  :    0    0  388    0    0  272    0  302    0    0  ->  963
 17  :    0    0  468   13    0  470    0    0    0    0  ->  953
 18  :    0    9  468    0    5  468    0    0    0    0  ->  953
 19  :    0    0  466    0    0  464    0   15    0    0  ->  947
 20  :    0    0   88   35    0  462  378    0    0    0  ->  965
 21  :    0  276    0    0  272  345   74    0    0    0  ->  968
 22  :    0  388    0    0  390   66    0  115    0    0  ->  961
 23  :    0  453   31    9  453    0    0    0    0    0  ->  947
 24  :   23  464    0    0  466    0    0    0    0    0  ->  955
 25  :    0   39    0    0  451  417   27   11    0    0  ->  947
 26  :    0    0    3    5  474  474    0    0    0    0  ->  959
 27  :  315    0    0    0  333  329    0    0    0    0  ->  978
 28  :    3    0    3    3  458  462   19    0    0    0  ->  953
 29  :    0    0    3    3  470  468    0    0    0    0  ->  947
 30  :    0    0    0    0  470  468   13    0    0    0  ->  953
 31  :   13    0   15   15  455  445    5    0    0    0  ->  951
 32  :   33    0  435  433   37    0    0    0    0    0  ->  939
 33  :    0    7  437  435    0    0    7    7    0    0  ->  896
 34  :   52    0    0    0   56    0    0    0    0    0  ->  109
 35  :  470    0    0    0  470   11    0    0    0    0  ->  953
 36  :  406    5   11    0  406    0   56   62    0    0  ->  949
 37  :    0    0    0   15    0    0  470  466    0    0  ->  953
 38  :    0    0    0    0    0   60  443  447    0    0  ->  951
 39  :   88    3   84    0    0  272  264  264    0    0  ->  978
 40  :  370    0  374  103  100    0    0    0    0    0  ->  949
 41  :    0    0    0  478  480    0    0    0    0    0  ->  959
 42  :    0    3    0  458  353   11    5  113    0    0  ->  947
 43  :  119    0  121  354    0    0    0  351    0    0  ->  947
 44  :  480    0  480    0    0    0    0    0    0    0  ->  961
 45  :  341  178  339    0   19   19    3    0    0    0  ->  902
 46  :    0  608    0   33    0    0    0  147    0    0  ->  788
 47  :    0  478    0    0    0    0    0  478    0    0  ->  957
 48  :    7  300  158    0   27  158    3  298    0    0  ->  955
 49  :    0    0  415  131    0  415    0    0    0    0  ->  963
 50  :    0    0  327  325    0  325    0    0    0    0  ->  978

total:   54   93  109  137  144  192  102   91    0    0  ->  925
idle :  72%  58%  52%  46%  48%  36%  50%  60% 100% 100%  ->  62%
```

13.2.2.2 LwIP low priority IO functions.

```
open 10 connections, check transferrate every 500 ms 50 times.

  1  :    0    0   25    0    0    0   25    0   43   41  ->  135
  2  :    0    0   23    0    0    0   23    0   23   23  ->   94
  3  :    7    0   94    0   96    0   86   98   17  107  ->  508
  4  :    0    0  154    0  154    0  158  217    0  182  ->  868
  5  :    0    1  149    1  160    3  178  211    0  192  ->  900
  6  :   90    0  162    0  156    0  170   96   11  186  ->  874
  7  :  166    0  196    0  182    0  164    0    0  145  ->  855
  8  :  152    0  186    0   80  119  182    0    0  149  ->  870
  9  :   76    7   66   49    0  113   72   45    5  117  ->  555
 10  :    0    0    0   25    0   25    0   25    0   25  ->  101
 11  :    0    0    0   45   47   54    0   64    0   45  ->  256
 12  :    5   11    0   35   45   64    5   17   47    0  ->  233
 13  :    0    0   47   43   74   78    0    0    0    0  ->  243
 14  :    0    0  105  133  158  164    0    0    0  113  ->  676
 15  :    9    5  115   21  184  113  117  145    5  147  ->  866
 16  :    0    0  141    0  156    0  174  221    0  168  ->  862
 17  :    0    0  131    0  147    0  168  217    0  188  ->  853
 18  :    9   43   43    0   43   13   37   56   11   88  ->  347
 19  :    0   43    0   51    0    0    0   49    0   47  ->  190
 20  :    0   25    0   25    0    0    0   25    0   25  ->  101
 21  :    5  121    0  107  194  203    0    7    5  129  ->  776
 22  :    0  170    3  176   23  176  176    0    0  170  ->  898
 23  :    0  182    0  154    0  174  207    0    0  170  ->  890
 24  :   58   74    0   43    0   60   25    0   13   58  ->  335
 25  :   23   21   66    0    0   62    0   66    0   66  ->  307
 26  :    0    0   25    0    0   25    0   25    0   25  ->  101
 27  :    0    0   49   56    5   49   23    3   60    3  ->  252
 28  :  164   15  160  209    0  182    0    0  154    0  ->  888
 29  :  172    0  170  217    5  176    5    0  137    0  ->  886
 30  :  135    5  141    3  149  205  215    5    0    5  ->  868
 31  :   98   98    1    0  258  264    1    5    0   76  ->  806
 32  :  147  166    0    0  213  205    0    0   17  131  ->  882
 33  :   96  111    0   13  190  205    5   17    1   70  ->  713
 34  :  127  164   19  198    5  182    0  166    0    0  ->  864
 35  :    0   90  131  152    0  225    0  227   29    0  ->  857
 36  :    0   86  233  253    9    0   23   94  100   21  ->  821
 37  :   33  147    0  154    0    0  125  168   86  180  ->  896
 38  :  113  258    0  262    0    0    0   98    0  107  ->  841
 39  :  105  235   35  219   33    7    0   94    5  100  ->  837
 40  :  172    7  200    0  184    0    0  154    0  158  ->  878
 41  :  200    0  192    0  170    0   35  152    0  154  ->  906
 42  :  186   45  156   45    0    3  113  152   13  133  ->  851
 43  :    0  166    0  160    0    0  164  186    0  186  ->  864
 44  :    0  176    0  131   45    0  147  196    0  186  ->  882
 45  :    0  166    0    0  158   54  125  192    9  180  ->  888
 46  :    3  162   33    0  174  168    0  182    0  174  ->  900
 47  :    0  168    0   54  160  158    0  172    0  190  ->  906
 48  :    0  176    0   88   41  149    5  158   60  196  ->  876
 49  :    7  170   84    0    0  107    0  164  105  229  ->  870
 50  :    0  166  178    0   68  186    0  158    0  158  ->  917

total:   47   69   70   62   75   79   59   86   19  105  ->  677
idle :  48%  34%  34%  40%  34%  34%  40%  24%  54%  12%  ->  35%
```

13.2.2.3 Quadros, socket API

```
open 3 connections, check transferrate every 500 ms 50 times.

  1  :   17   17   17  ->   52
  2  :    7    9    7  ->   25
  3  :    7    7    9  ->   25
  4  :    9    7    7  ->   25
  5  :    7    7    7  ->   23
  6  :    7    9    7  ->   25
  7  :    7    7    9  ->   25
  8  :    9    7    7  ->   25
  9  :    7    7    7  ->   23
 10  :    7    9    7  ->   25
 11  :    7    7    9  ->   25
 12  :    9    7    7  ->   25
 13  :    7    7    7  ->   23
 14  :    7    9    7  ->   25
 15  :    7    7    9  ->   25
 16  :    9    7    7  ->   25
 17  :    7    7    7  ->   23
 18  :    7    9    7  ->   25
 19  :    7    7    9  ->   25
 20  :    9    7    7  ->   25
 21  :    7    7    7  ->   23
 22  :    7    9    7  ->   25
 23  :    7    7    9  ->   25
 24  :    9    7    7  ->   25
 25  :    7    7    7  ->   23
 26  :    7    9    7  ->   25
 27  :    7    7    9  ->   25
 28  :    9    7    7  ->   25
 29  :    7    7    7  ->   23
 30  :    7    9    7  ->   25
 31  :    7    7    9  ->   25
 32  :    9    7    7  ->   25
 33  :    7    7    7  ->   23
 34  :    7    9    7  ->   25
 35  :    7    7    9  ->   25
 36  :    9    7    7  ->   25
 37  :    7    7    7  ->   23
 38  :    7    9    7  ->   25
 39  :    7    7    9  ->   25
 40  :    9    7    7  ->   25
 41  :    7    7    7  ->   23
 42  :    7    9    7  ->   25
 43  :    7    7    9  ->   25
 44  :    9    7    7  ->   25
 45  :    7    7    7  ->   23
 46  :    7    9    7  ->   25
 47  :    7    7    9  ->   25
 48  :    9    7    7  ->   25
 49  :    7    7    7  ->   23
 50  :    7    9    7  ->   25

total:    8    8    8  ->   25
idle :   0%   0%   0%  ->   0%
```

13.2.3 Bidirectional traffic

13.2.3.1 LwIP high priority raw API

```
open 10 connections, check transferrate every 500 ms 50 times.

  1  :   11    0    0    0    0    0    0    3    0    0  ->   15
  2  :    0    0    0    0    0    0   24    0    0    0  ->   24
  3  :    0    0    0    0    0    0   50    0    0    0  ->   50
  4  :    0    0    0    0    0    0   50    0    0    0  ->   50
  5  :    0    0   11    7    0    0   69    0    0    0  ->   89
  6  :    0    3    0    0  125  129    0    0    0    0  ->  258
  7  :    0    0    0    0    0    0    0    0    3    3  ->    7
  8  :    0    0    0    0    0    0    0    0    0    0  ->    0
  9  :    0    0    0    0    0    0    0    0    0    0  ->    0
 10  :    0    0    0    0    0    0    0    0    0    0  ->    0
 11  :    0    0    0    0    0    0    0    0    0    0  ->    0
 12  :    0    0    0    0    0    0    0    0    0    0  ->    0
 13  :    0    0    0    0    0    0    0    0    0    0  ->    0
 14  :    0    0    0    0    0    0    0    0    0    0  ->    0
 15  :    0    3    0    0    0    0    0    0    0    0  ->    3
 16  :    0    0    0    0    0    0    0    0    0    0  ->    0
 17  :    0    0    0    0    0    0    0    0    0    0  ->    0
 18  :    0  183    0    0    0    0    0    0   14   21  ->  219
 19  :    0    0    0    0    0    0    0    0    0    1  ->    1
 20  :    0    0    0    0    0    0    0    0    0    0  ->    0
 21  :    0    1    0    0    0    0    0    0    0    0  ->    1
 22  :    0    0    0    0    0    0    0    0    5    0  ->    5
 23  :    0    0    0    0    0    0    0  658    0    0  ->  658
 24  :    0    0    0    0    0    0    0  733    0    0  ->  733
 25  :    0   64    0    0    0    0    0  670   20   16  ->  771
 26  :    0    0    0    0    0    0    0  710   40    0  ->  752
 27  :    3    0    0    0    0    1    0  298   45  324  ->  673
 28  :    0    0    0    0    0   35    0   32    2    0  ->   69
 29  :    0  128    0    0    0    0    0  163   11    0  ->  302
 30  :    0   15    0    0    3    0    0  118    1    0  ->  140
 31  :    0  290    0    0  419    1    0    0    0    0  ->  711
 32  :    0  271    0    0  275   11    0    0    2    0  ->  561
 33  :    0    5    0    0    0    0    0    3    0    0  ->    9
 34  :   47    0    0    0    0    0    0    0    0    0  ->   47
 35  :   50    0    0    0    0    0    1    0   43    0  ->   96
 36  :   23  301    0    0    0  391    0    3   31    0  ->  751
 37  :    0    1    0    0    0  496    0    0  499    0  ->  997
 38  :    0    4    0    0    0  376    0    3  381    0  ->  767
 39  :    6  114    0    0    0    3    0    3   13    0  ->  142
 40  :    0    0    0    0    0    0    0    0    0    0  ->    0
 41  :    0    0    0    0    0    0    0    0    0    0  ->    0
 42  :    0    0    0    0    0    3    0    0    0    0  ->    3
 43  :    0    0    0    0    0    0    0    0    0    0  ->    0
 44  :    0    0    0    0    0    0    0    0    0    0  ->    0
 45  :    0   36    0    0    0    0    0    0    0    0  ->   36
 46  :  262   57    1    0    0    0    0    0   16    0  ->  338
 47  :    0    0    1    0    0    0    0    0    0    0  ->    1
 48  :    0    4    0    0    0    0    0    0    0    0  ->    4
 49  :  131   11  159    0    0    0    0    0    1    0  ->  305
 50  :  486   13  485    0    0    0    0    0    1    0  ->  987

total:   20   30   13    0   16   29    3   68   22    7  ->  211
idle :  80%  60%  90%  98%  92%  80%  90%  74%  64%  90%  ->  81%
```

13.2.3.2 LwIP low priority IO functions

```
open 10 connections, check transferrate every 500 ms 50 times.

  1  :  125   19  110  108   11   54   62  198   43    7  ->  742
  2  :   78  117  213   80   17    0   27  104   61   90  ->  791
  3  :  233   33   41    9   88  121  107   34   29  106  ->  808
  4  :   65  196   96  129   15   88    5  117   66   72  ->  853
  5  :   29    9  174  202   33    9    7   56   29  174  ->  727
  6  :   15   25   78   11   56    0   17  247  100  253  ->  806
  7  :   62  123   24   94  140    0    5  119   39  182  ->  792
  8  :   31   33  113    0  274   33    0   27  131  158  ->  804
  9  :  243  121    9    0    1  256    0   23  111   41  ->  810
 10  :   29    0  118   41    0  112   92  102  115  224  ->  835
 11  :   21   37   82  243    0   23  103  115   43  151  ->  821
 12  :   54   25    0   64  115   96  168   27   92  176  ->  821
 13  :  123    0  131    7  113  151  111    5  113   68  ->  827
 14  :  154  194    0   31   70   29  113   94   25  125  ->  839
 15  :   33   66  117  113    0   17  231  178    5    7  ->  772
 16  :  152   23  127  160    0   11  268   58   29    3  ->  837
 17  :   80   25  178   62    0    0   64  149  160  105  ->  827
 18  :   31   66  115   41    0    0  290   39  147   43  ->  774
 19  :   81  192   18  172    0   39  100   62   96   83  ->  846
 20  :   64  213   70    0    0   39   76    0  152  213  ->  831
 21  :  215  125   11  207   43   47   47    0   50   47  ->  796
 22  :  103  233    5   94   49   64    0   98  188    0  ->  837
 23  :  152  115  137  168   29  113    0  131    0    0  ->  849
 24  :   68  152   50  113   52   70   49   56  125   92  ->  833
 25  :  233   35   15   25  213   29  203   27    0   35  ->  819
 26  :  131   37    0   11  229    0  188    0  217    0  ->  815
 27  :  149  100    0   39   49  333    1    0   58    3  ->  735
 28  :  200  109    0  117    5  119    0   17  243    0  ->  813
 29  :  117  202    0  100   90  101    0   62  154    0  ->  829
 30  :  202   56    0   74   70  200    3   52  162    5  ->  829
 31  :  268   29    0  249    0   52    0  105  103    0  ->  810
 32  :  115   37    0   88    0  176    0  282   11  101  ->  813
 33  :  119  182    0  103    7  107    0  253   52    1  ->  829
 34  :  198  207    0    1    0    0    0  207  203    0  ->  819
 35  :  243  154    0    7    0    0    0  130  263   43  ->  843
 36  :  215  209    0    0    0   31    0  209   88   76  ->  831
 37  :  153  121    0   19  144  196    0   86   88   13  ->  823
 38  :  181  149    0    0  291    0    0   39    0  162  ->  823
 39  :   62  127    0    0   70   29    0   86  273   80  ->  730
 40  :   27  141    0    0  199   86    0  115   47  190  ->  807
 41  :    7  282    0    0   17   39    0  127   43  282  ->  799
 42  :   29  188    0    0   46   80   78   43  192  194  ->  853
 43  :   90  200    0    0   62  123   25  180   70   66  ->  819
 44  :  249  145    0   50  305    0   15    0    0   28  ->  794
 45  :   13   72    0  286  105    0   98    0   50  176  ->  804
 46  :   98   51    0   17  110    0  266    0  184   78  ->  807
 47  :  241   82    0   68   86    0  176    0  154    0  ->  810
 48  :   43  121    0  221  268    0   52    0   96    0  ->  804
 49  :    7   54    0  239  130    0  125    0  254    0  ->  814
 50  :  164  209    0  141    0    0   84    0  227    0  ->  827

total:  116  109   40   80   72   61   65   81  104   79  ->  812
idle :   0%   4%  54%  20%  28%  32%  34%  22%   8%  20%  ->  22%
```

13.2.3.3 Quadros, socket API

```
open 3 connections, check transferrate every 500 ms 50 times.

  1  :   40   42   18  ->  100
  2  :   19   19    7  ->   45
  3  :   28   32    6  ->   67
  4  :   22   19    2  ->   44
  5  :   15   15    7  ->   38
  6  :   26   30    2  ->   60
  7  :   22   17   13  ->   53
  8  :   19   19   10  ->   48
  9  :   13   13   10  ->   36
 10  :   11   17    5  ->   34
 11  :   11   11   11  ->   34
 12  :   11    9   11  ->   32
 13  :   28   26    4  ->   60
 14  :   11   11    5  ->   28
 15  :   26   24    2  ->   54
 16  :   11   15    7  ->   33
 17  :   28   28    6  ->   63
 18  :   15   17    8  ->   40
 19  :   19   17    9  ->   45
 20  :    5    7    5  ->   19
 21  :   28   26   10  ->   66
 22  :    9    7    7  ->   24
 23  :   22   26    0  ->   50
 24  :   15   11    8  ->   34
 25  :   30   28    5  ->   64
 26  :   20   22   11  ->   55
 27  :   22   20   10  ->   54
 28  :    5   11    9  ->   26
 29  :   22   22    6  ->   52
 30  :   26   20   11  ->   59
 31  :   15   19   13  ->   47
 32  :   19   15   11  ->   45
 33  :    3    7    5  ->   17
 34  :   20   22    4  ->   48
 35  :   19   13    7  ->   39
 36  :   26   28    0  ->   56
 37  :   20   24   13  ->   59
 38  :   15   15   17  ->   47
 39  :   11    9    3  ->   24
 40  :   19   17    4  ->   41
 41  :   19   19    9  ->   47
 42  :   13   19   10  ->   43
 43  :   22   17   18  ->   58
 44  :    5    7    6  ->   19
 45  :   17   15    2  ->   35
 46  :   19   21   11  ->   51
 47  :    9   11    5  ->   26
 48  :   15   15    2  ->   33
 49  :   24   23   13  ->   61
 50  :   19   22    8  ->   50

total:   18   18    8  ->   45
idle :   0%   0%   0%  ->   0%
```

13.3 Speed test results

Open 1 to 10 connections and check transfer rate during 5 times 5 seconds.

1. The first column gives the number of connections.
2. The second column gives the average total speed.
3. The third column gives the maximum total speed during 5 seconds.
4. The fourth column gives the percentage of times no traffic occurred on a connection during 5 seconds.
5. The last columns give the average speed per connection.

All speeds are again in kB/sec

13.3.1 Incoming traffic

13.3.1.1 LwIP on high priority test applications

```
 1  :  785  786   0%  785
 2  :  945  947   0%  472  472
 3  :  956  959   0%  318  318  318
 4  :  881  885   0%  220  220  220  220
 5  :  882  887   8%  200  204   87  211  178
 6  :  881  885  13%  180  197  206  150  126   19
 7  :  881  886   5%   77  184   44  156  141  186   90
 8  :  874  883   0%   77  145  136  164   83   70   77  119
 9  :  873  878  11%   86   59  148  134  133    5   55  127  122
10  :  871  874  20%  102  112  136  135  118  133   58   38   35    0
```

13.3.1.2 LwIP on low priority test applications

```
 1  :  566  604   0%  566
 2  :  833  839   0%  424  409
 3  :  832  838   0%  255  282  294
 4  :  835  837   0%  227  150  225  233
 5  :  827  829   0%  180  182  130  164  169
 6  :  826  829  20%  234  230   20  146    0  194
 7  :  827  833   0%  103  136  158  111  102   58  157
 8  :  811  823   7%  134  163  127  119   58   70   94   43
 9  :  816  829   0%   96  112   88   86   67  112  108   79   64
10  :  817  832  22%  104  141   74   93   81   49    0   65   94  112
```

13.3.1.3 Quadros with test applications using select

```
 incoming traffic
1  :  706   737  0%  706
2  :  854   858  0%  427  427
3  :  343   855 20%  114  114  114
```

13.3.2 Outgoing traffic

13.3.2.1 LwIP on high priority test applications

```
 1  :  703  703   0%  703
 2  :  951  952   0%  475  475
 3  :  947  972   0%  345  243  358
 4  :  943  970   0%  354  181  304  102
 5  :  907  961   0%  243  131  166  175  191
 6  :  815  929   0%  114  189  185  132  117   75
 7  :  919  937   0%  129  106  115  171  142   90  163
 8  :  932  950   0%  150  165  119  139  112   86   84   73
 9  :  876  962  11%  121   61  101  170  124   62  106  126    0
10  :  904  955  26%  110  107   30  110  115   94  195  121    0   18
```

13.3.2.2 LwIP on low priority test applications

```
 1  :  703  703   0%  703
 2  :  917  926   0%  472  444
 3  :  931  972   0%  296  303  331
 4  :  947  969   0%  211  231  306  198
 5  :  953  969   0%  105  237  263  131  214
 6  :  926  968   0%  151  143  252  140   81  156
 7  :  893  961   2%  111  175   62  115  192  118  117
 8  :  936  953   0%  122  135   82  171  139   70  113  101
 9  :  866  936   0%   60   97   79   53  109   88  114   97  166
10  :  657  834   0%   92   54   83   55   48   55   55   75   60   75
```

13.3.2.3 Quadros with test applications using select

```
1  :   24   24   0%   24
2  :   15   16   0%    7    7
3  :   24   25   0%    8    8    8
```

13.3.3 Bidirectional traffic

13.3.3.1 LwIP on high priority test applications

```
 1  :  724  725   0%  724
 2  :  911  941   0%  480  430
 3  :  760  886   0%  263  199  298
 4  :  774  820   0%  265   91  239  178
 5  :  714  775  12%   81  297   42  116  176
 6  :  257  610  60%   36  101   22   49   30   16
 7  :  503  706  17%  125  122    0    0   69   60  125
 8  :   97  459  75%    0    0    0    5    4    3   83    0
 9  :  180  448  60%   55    0    0    0    0   69   53    0    0
10  :  108  514  84%    5    0    0    0    0    0  103    0    0    0
```

13.3.3.2 LwIP on low priority test applications

```
 1  :  886  896   0%  886
 2  :  848  867  40%  812   36
 3  :  843  867  53%   21   19  802
 4  :  861  885  55%    6   42   55  756
 5  :  811  836  32%   59  365  174   85  125
 6  :  848  933  43%  141    4   68   47  510   75
 7  :  817  829   0%  133  123  123  117  116   90  114
 8  :  806  825   0%  105  113   98   74  110  101   82  120
 9  :  817  823   8%   80   14   80  104   67  130  141   81  114
10  :  804  820  12%    5  133   85   73  105    3  100  137   78   81
```

13.3.3.3 Quadros with test applications using select

```
1  :   59   65   0%   59
2  :   60   81   0%   30   30
3  :   52   73   0%   19   21   10
```

13.4 Load test results

Open 1 connection, generate an increasing amount of traffic and measure the load (percentage of processing time spend on the TCP/IP stack). To get an accurate result for the target speed the measurement was repeated 5 times and the average was taken over the test results that came close to the target speed.

13.4.1.1 Incoming traffic

Target Speed	Test 1		Test 2		Test 3		Test 4		Test 5		Average	
	Load	Speed	Load	Speed	Load	Speed	Load	Speed	Load	Speed	Load	Speed
kB/s	‰	kB/s	‰	kB/s	‰	kB/s	‰	kB/s	‰	kB/s	%	kB/s
0	5	0	5	0	6	0	5	0	5	0	0,5	0
10	13	10	15	10	13	10	15	10	15	10	1,4	10
20	25	20	25	20	23	20	25	20	25	20	2,5	20
30	34	30	34	30	34	30	35	30	34	30	3,4	30
40	44	40	44	40	44	40	44	40	44	40	4,4	40
50	52	50	54	50	54	50	54	50	56	50	5,4	50
60	63	60	63	60	63	60	63	60	63	60	6,3	60
70	73	70	73	70	71	70	73	70	73	70	7,3	70
80	83	80	81	80	81	80	81	80	83	80	8,2	80
90	91	90	90	90	91	90	90	90	95	90	9,1	90
100	100	100	100	100	100	100	102	100	103	100	10,1	100
110	112	110	112	110	110	110	108	110	112	110	11,1	110
120	120	120	120	120	122	120	120	120	120	119	12,0	120
130	129	130	130	129	130	130	130	129	130	129	13,0	129
140	139	140	141	139	141	140	139	140	139	140	14,0	140
150	149	150	149	150	149	150	147	150	147	149	14,8	150
160	158	160	158	159	159	160	158	160	159	160	15,8	160
170	170	170	168	170	170	169	170	169	166	169	16,9	169
180	178	180	178	180	178	180	180	179	178	180	17,8	180
190	190	190	187	190	188	190	187	190	187	190	18,8	190
200	197	200	197	200	198	200	198	199	198	200	19,8	200
210	205	209	207	209	207	210	207	210	207	210	20,7	210
220	215	219	217	219	217	219	217	219	215	219	21,6	219
230	226	229	227	228	227	229	226	229	226	229	22,6	229
240	236	239	238	239	236	239	238	239	236	239	23,7	239
250	248	249	246	249	248	249	246	249	244	249	24,6	249
260	256	259	256	259	256	259	256	259	256	259	25,6	259
270	266	269	266	269	266	268	265	269	265	269	26,6	269
280	277	278	275	279	275	279	275	279	275	279	27,5	279
290	283	289	283	289	285	289	285	289	285	289	28,4	289
300	295	299	294	299	295	299	294	298	295	299	29,5	299
310	302	308	304	309	306	308	304	309	304	308	30,4	308
320	312	318	312	318	314	318	312	318	312	318	31,2	318
330	326	328	323	329	321	328	321	329	324	328	32,3	328
340	333	338	333	338	333	338	331	338	334	338	33,3	338
350	341	348	343	348	343	348	341	348	341	348	34,2	348
360	353	358	353	358	353	358	353	358	353	358	35,3	358
370	362	368	363	368	362	368	360	368	362	368	36,2	368
380	372	378	372	378	372	378	370	378	373	378	37,2	378
390	380	388	382	388	380	388	382	388	382	388	38,1	388
400	390	398	389	398	390	398	389	398	392	398	39,0	398
410	401	408	399	408	401	408	402	408	401	408	40,1	408
420	409	418	411	418	413	418	413	418	409	418	41,1	418
430	421	428	419	428	418	428	419	427	421	428	42,0	428
440	430	438	431	438	430	438	428	438	430	438	43,0	438
450	438	448	438	448	440	448	440	448	440	448	43,9	448
460	448	458	448	458	448	458	448	458	447	457	44,8	458
470	458	468	458	468	458	467	457	468	458	468	45,8	468
480	467	478	470	478	469	479	469	479	469	477	46,9	478
490	479	488	479	489	479	488	477	488	477	489	47,8	488
500	486	499	489	497	489	500	482	498	491	498	48,7	498
510	492	509	499	509	498	506	494	508	496	498	49,5	509
520	504	517	511	519	504	518	509	520	508	518	50,7	518
530	515	527	516	528	515	529	515	528	515	527	51,5	528
540	521	537	523	539	525	537	528	537	526	537	52,5	537
550	535	549	533	550	537	548	538	549	535	548	53,6	549
560	547	557	545	560	543	559	545	559	545	559	54,5	559
570	554	567	555	569	552	568	559	568	554	567	55,5	568
580	562	579	564	580	564	580	567	579	565	580	56,4	580
590	576	589	572	589	576	590	571	589	572	590	57,3	589
600	584	600	582	599	586	598	584	597	586	599	58,4	599
610	594	607	596	608	593	609	596	609	594	607	59,5	608
620	601	620	605	619	608	618	603	620	603	618	60,4	619
630	615	627	615	629	615	629	613	630	615	629	61,5	629
640	627	639	623	639	628	640	628	637	623	639	62,6	639
650	637	649	635	649	633	646	633	650	633	649	63,4	649
660	644	656	642	660	639	660	645	656	644	659	64,3	658
670	654	670	649	668	650	670	656	669	650	669	65,2	669
680	667	678	666	676	661	679	662	676	664	679	66,4	678
690	669	689	669	689	673	689	673	688	673	689	67,1	689
700	684	699	683	699	683	699	686	697	679	696	68,3	698
710	691	710	691	709	691	710	691	709	690	710	69,1	710
720	703	719	707	718	705	718	701	718	703	719	70,4	718
730	707	727	703	728	708	726	707	730	710	730	70,7	728
740	720	738	724	740	730	738	720	739	718	738	72,2	739
750	739	747	730	750	730	748	732	749	724	745	73,1	748
760	742	759	741	759	741	757	739	759	741	759	74,1	759
770	744	769	751	769	751	770	751	768	754	769	75,0	769
780	761	779	764	778	758	777	759	776	759	779	76,0	778
790	766	786	766	785	766	784	764	784	764	784	76,6	786
800	766	784	764	785	766	782	766	783	764	783	76,5	783
810	766	786	766	784	764	786	766	779	763	708	76,6	784
820	764	785	764	784	766	784	766	783	766	783	76,5	784
830	764	784	764	784	764	785	766	785	766	786	76,5	785
840	764	786	764	786	766	783	764	786	766	786	76,5	785
850	764	786	764	786	766	786	766	786	766	783	76,5	785
860	766	786	766	786	768	784	766	786	764	785	76,6	785
870	766	786	766	786	766	786	764	786	764	785	76,5	786
880	766	786	766	785	763	783	763	784	763	782	76,4	784
890	759	780	766	786	761	784	764	783	766	785	76,3	784
900	764	785	764	785	766	784	766	785	764	785	76,5	785
910	764	782	761	780	761	782	764	784	764	784	76,3	782
920	761	781	763	784	764	784	763	784	764	785	76,3	784
930	764	785	764	784	766	785	766	786	766	786	76,5	785
940	766	786	766	786	768	786	766	786	766	786	76,6	786
950	763	785	764	786	764	786	764	786	766	786	76,4	786
960	764	786	766	786	764	786	764	786	764	786	76,4	786
970	766	786	764	785	764	786	766	786	766	786	76,5	786
980	766	785	766	786	766	786	766	786	764	786	76,6	786
990	764	786	766	786	766	786	766	786	766	786	76,6	786
1000	764	786	766	786	766	786	766	786	764	786	76,5	786

Outgoing traffic

Target Speed	Test 1		Test 2		Test 3		Test 4		Test 5		Average	
	Load	Speed	Load	Speed	Load	Speed	Load	Speed	Load	Speed	Load	Speed
kB/s	‰	kB/s	‰	kB/s	‰	kB/s	‰	kB/s	‰	kB/s	%	kB/s
0	5	0	5	0	5	0	6	0	5	0	0,5	0
10	13	10	12	10	12	10	17	10	18	10	1,4	10
20	32	20	30	20	32	20	30	20	30	20	3,1	20
30	42	30	42	30	40	30	40	30	40	30	4,1	30
40	51	40	51	40	51	40	49	40	52	40	5,1	40
50	63	50	61	50	61	50	63	50	61	50	6,2	50
60	73	60	71	60	69	60	71	60	73	60	7,1	60
70	81	70	83	70	83	70	81	70	81	70	8,2	70
80	91	80	93	80	91	80	93	80	93	80	9,2	80
90	100	90	102	90	105	90	105	90	103	90	10,3	90
100	115	100	113	100	113	100	113	100	115	100	11,4	100
110	125	110	122	110	124	110	124	110	125	110	12,4	110
120	136	120	136	120	134	120	136	120	136	120	13,6	120
130	146	130	147	130	147	130	147	130	147	130	14,7	130
140	156	140	158	140	156	140	156	140	154	140	15,6	140
150	166	150	168	150	166	150	166	150	168	150	16,7	150
160	178	160	178	160	178	160	176	160	178	160	17,8	160
170	187	170	187	170	188	170	188	170	188	170	18,8	170
180	198	180	198	180	198	179	198	180	197	179	19,8	180
190	209	190	209	190	209	190	207	190	209	190	20,9	190
200	219	200	219	200	219	200	219	200	219	200	21,9	200
210	229	209	227	209	231	210	229	210	227	210	22,9	210
220	239	220	239	220	238	220	239	220	238	220	23,9	220
230	249	229	249	230	249	229	251	229	251	229	25,0	229
240	261	240	260	240	261	240	261	240	260	240	26,1	240
250	270	249	273	250	272	250	273	250	273	250	27,2	250
260	283	260	282	260	283	260	283	260	283	260	28,3	260
270	294	270	294	270	295	270	294	270	292	270	29,4	270
280	302	280	304	280	302	279	304	280	304	280	30,3	280
290	314	289	316	290	314	290	314	290	314	289	31,4	290
300	321	299	324	300	328	299	324	299	324	299	32,4	299
310	333	310	334	309	334	309	333	310	336	309	33,4	309
320	345	320	346	319	348	320	345	320	345	320	34,6	320
330	358	329	356	330	355	330	355	329	356	330	35,6	330
340	365	339	368	340	367	339	365	340	367	339	36,6	339
350	377	350	377	349	377	349	377	349	377	349	37,7	349
360	389	359	387	360	390	359	385	360	387	359	38,8	359
370	397	369	397	369	397	369	399	369	399	370	39,8	369
380	409	380	409	379	407	379	409	379	409	317	40,9	379
390	219	364	421	390	419	390	419	390	419	390	42,0	390
400	430	400	430	399	430	399	431	400	433	400	43,1	400
410	441	410	441	409	441	409	441	410	441	410	44,1	410
420	453	419	450	419	453	420	452	419	452	419	45,2	419
430	462	429	462	429	464	429	464	430	462	429	46,3	429
440	475	439	472	439	472	440	472	440	472	439	47,3	439
450	484	450	484	450	484	450	484	450	484	450	48,4	450
460	496	460	494	459	494	459	494	460	494	460	49,4	460
470	504	469	504	469	504	469	506	469	504	469	50,4	469
480	515	479	516	479	518	479	515	479	516	480	51,6	479
490	525	489	528	490	528	489	528	489	526	488	52,7	489
500	537	499	537	499	535	499	537	499	535	499	53,6	499
510	547	510	549	510	547	509	550	509	543	509	54,7	509
520	555	519	555	520	559	519	557	519	557	519	55,7	519
530	571	528	567	529	567	529	567	529	567	529	56,8	529
540	576	539	579	538	577	538	579	539	577	539	57,8	539
550	588	548	591	550	588	549	591	549	589	549	58,9	549
560	601	559	598	559	599	559	599	560	601	560	60,0	559
570	608	569	610	569	611	568	610	568	610	568	61,0	568
580	622	578	618	579	620	578	622	579	622	579	62,1	579
590	630	588	630	588	628	589	630	589	632	589	63,0	589
600	640	598	642	599	640	599	640	599	644	599	64,1	599
610	647	609	647	608	650	608	645	606	650	606	64,8	608
620	647	606	649	607	647	606	649	606	647	606	64,8	606
630	649	606	650	606	649	606	647	606	645	606	64,8	606
640	645	606	647	606	649	606	647	606	645	606	64,7	606
650	649	606	647	606	647	607	647	606	647	607	64,7	606
660	645	606	647	606	647	606	645	607	649	606	64,7	606
670	645	606	649	606	645	606	649	606	647	606	64,7	606
680	649	606	649	606	647	606	645	606	39	198	64,8	606
690	647	606	647	606	649	606	649	607	647	606	64,8	606
700	647	606	647	606	647	606	647	607	649	606	64,7	606
710	647	607	649	606	650	606	647	606	649	606	64,8	606
720	649	606	649	606	645	606	649	606	649	606	64,8	606
730	647	606	649	606	649	606	649	606	516	499	64,9	606
740	647	611	647	606	644	606	649	606	647	606	64,7	607
750	647	606	649	606	649	606	647	606	647	606	64,8	606
760	649	606	649	606	647	606	647	606	647	606	64,8	606
770	647	606	649	606	647	606	649	606	645	606	64,7	606
780	645	606	645	606	647	607	649	606	649	606	64,7	606
790	645	606	645	606	649	606	650	606	649	606	64,8	606
800	649	606	647	606	645	606	649	606	647	606	64,7	606
810	647	606	649	606	647	607	644	606	647	606	64,7	606
820	649	606	650	606	647	606	647	606	645	606	64,8	606
830	649	606	647	606	647	606	650	606	647	606	64,8	606
840	647	606	649	606	650	606	645	606	647	606	64,8	606
850	647	606	647	606	647	606	647	607	649	606	64,7	606
860	647	606	647	606	649	606	649	606	649	606	64,8	606
870	645	606	649	606	649	606	649	606	649	606	64,8	606
880	650	606	649	606	647	606	649	607	649	606	64,9	606
890	647	606	649	606	645	606	649	607	647	606	64,7	606
900	645	607	647	606	647	606	649	607	647	606	64,7	606
910	649	606	647	606	645	606	649	606	647	606	64,7	606
920	645	606	647	606	649	606	649	606	647	607	64,7	606
930	649	606	649	606	649	606	647	606	647	606	64,8	606
940	650	606	647	606	649	606	650	606	650	606	64,9	606
950	649	606	647	606	647	606	647	606	649	606	64,8	606
960	650	606	647	607	649	606	647	606	649	606	64,8	606
970	647	606	649	606	649	606	647	606	645	606	64,7	606
980	645	606	649	606	649	606	644	605	650	606	64,7	606
990	649	606	647	606	649	607	645	606	647	607	64,7	606
1000	647	606	645	606	647	606	647	607	647	606	64,7	606

13.4.1.2 Bidirectional speed

Target Speed kB/s	Test 1 Load ‰	Test 1 Speed kB/s	Test 2 Load ‰	Test 2 Speed kB/s	Test 3 Load ‰	Test 3 Speed kB/s	Test 4 Load ‰	Test 4 Speed kB/s	Test 5 Load ‰	Test 5 Speed kB/s	Average Load %	Average Speed kB/s
0	6	0	5	0	6	0	5	0	5	0	0,5	0
10	18	10	18	11	17	11	17	11	17	11	1,7	11
20	30	21	30	21	30	21	30	21	30	21	3,0	21
30	44	31	42	31	44	31	44	31	44	31	4,4	31
40	56	41	56	41	56	41	56	41	56	41	5,6	41
50	68	51	68	51	66	51	68	51	68	51	6,8	51
60	78	60	78	60	80	60	78	60	80	60	7,9	60
70	88	70	90	70	90	70	90	70	90	70	9,0	70
80	100	80	102	80	100	80	102	80	100	80	10,1	80
90	112	90	112	90	113	90	113	90	113	90	11,3	90
100	125	100	124	100	125	100	125	100	125	100	12,5	100
110	137	110	136	110	136	110	136	110	137	110	13,6	110
120	147	120	149	120	149	120	147	120	149	120	14,8	120
130	161	130	159	130	158	130	159	130	159	130	15,9	130
140	173	140	171	140	173	140	173	140	171	140	17,2	140
150	185	150	183	150	185	150	185	150	183	150	18,4	150
160	192	160	195	160	193	160	193	160	195	160	19,4	160
170	207	170	207	170	207	170	207	170	205	170	20,7	170
180	219	180	219	180	217	180	219	180	217	179	21,8	180
190	229	189	227	189	229	189	232	189	231	189	23,0	189
200	243	199	244	199	243	199	244	199	243	199	24,3	199
210	255	209	253	209	255	209	255	209	253	209	25,4	209
220	263	219	266	219	265	219	266	219	266	219	26,5	219
230	278	229	277	229	277	229	277	230	278	229	27,7	229
240	289	240	287	240	289	240	289	239	289	239	28,9	240
250	300	249	302	249	299	249	300	249	299	249	30,0	249
260	311	260	311	259	311	260	311	259	312	259	31,1	259
270	324	269	323	269	323	269	314	269	314	269	32,0	269
280	326	279	326	279	324	279	326	279	326	279	32,6	279
290	338	289	334	289	336	289	334	289	336	289	33,6	289
300	346	299	348	299	346	299	348	299	346	299	34,7	299
310	360	309	360	309	360	309	358	309	358	309	35,9	309
320	372	319	372	320	372	319	372	319	370	319	37,2	319
330	382	329	382	329	379	329	382	329	384	329	38,2	329
340	394	339	394	339	394	339	392	339	394	339	39,4	339
350	406	348	406	348	406	349	402	348	404	348	40,5	348
360	416	358	419	358	416	359	419	359	418	358	41,8	358
370	430	368	428	368	428	369	428	368	426	368	42,8	368
380	440	378	440	378	441	378	441	378	440	378	44,0	378
390	450	388	452	388	452	388	452	388	452	388	45,2	388
400	462	398	462	398	464	398	462	398	464	398	46,3	398
410	475	408	474	408	475	408	474	408	474	408	47,4	408
420	487	418	486	418	486	418	487	418	484	418	48,6	418
430	496	428	496	428	498	428	498	428	498	428	49,7	428
440	508	438	508	438	509	437	508	438	508	438	50,8	438
450	520	447	518	447	520	448	521	448	520	447	52,0	447
460	532	457	532	458	532	458	530	457	530	457	53,1	457
470	538	467	542	467	542	467	542	467	543	468	54,1	467
480	555	477	554	477	555	477	555	478	554	477	55,5	477
490	565	487	565	488	564	487	565	488	564	487	56,5	487
500	579	497	577	498	577	497	576	497	577	497	57,7	497
510	589	507	588	506	589	506	589	508	589	506	58,9	507
520	601	516	601	517	601	517	599	517	601	518	60,1	517
530	611	526	613	527	611	527	613	526	613	526	61,2	526
540	625	538	623	537	627	537	623	537	623	536	62,4	537
550	632	546	635	548	633	547	633	547	633	547	63,3	547
560	645	556	647	557	645	557	647	557	645	557	64,6	557
570	657	567	657	567	657	567	657	567	657	568	65,7	567
580	669	577	669	576	669	577	673	576	671	577	67,0	577
590	683	586	681	587	681	586	681	587	681	587	68,1	587
600	693	597	693	596	693	596	691	596	693	597	69,3	596
610	681	595	681	590	683	591	681	592	676	590	68,2	592
620	679	590	683	590	679	589	679	590	681	590	68,0	590
630	683	590	676	587	676	589	681	591	679	588	67,9	589
640	681	589	674	589	681	590	678	588	676	589	67,8	589
650	679	590	678	589	683	591	679	589	681	591	68,0	590
660	681	591	678	589	679	591	678	589	683	591	68,0	590
670	679	591	678	589	676	589	679	591	676	587	67,8	589
680	679	590	679	591	681	592	678	590	679	591	67,9	591
690	681	591	679	590	674	587	679	591	679	589	67,8	590
700	686	591	679	588	679	591	678	589	681	592	68,1	590
710	683	591	678	590	679	589	678	589	679	591	67,9	590
720	678	588	679	589	678	591	676	590	679	589	67,8	589
730	674	586	676	588	678	590	684	594	679	589	67,8	589
740	674	590	681	591	676	589	678	589	681	591	67,8	590
750	681	589	679	591	679	592	678	589	674	590	67,8	590
760	681	592	686	591	679	591	678	589	676	588	68,0	590
770	678	589	679	591	679	589	681	588	681	592	68,0	590
780	681	587	683	592	673	588	679	591	679	591	67,9	590
790	678	589	678	589	679	586	676	588	683	592	67,9	589
800	678	588	681	590	674	589	681	589	681	590	67,9	589
810	676	589	683	593	679	588	679	592	678	589	67,9	590
820	678	591	686	592	684	592	681	590	679	588	68,2	591
830	681	591	674	587	681	591	683	591	684	591	68,1	590
840	678	590	678	588	679	589	679	590	679	591	67,9	590
850	679	589	679	589	679	589	681	592	679	588	67,9	589
860	679	588	681	590	679	592	681	586	678	589	68,0	589
870	683	592	676	588	678	590	679	590	679	590	67,9	590
880	678	589	678	591	678	588	679	590	679	588	67,8	589
890	678	590	678	588	679	590	676	587	679	590	67,8	589
900	679	591	679	590	681	590	678	590	681	590	68,0	590
910	678	588	681	591	679	591	681	589	681	590	68,0	590
920	681	591	674	589	684	591	676	590	679	590	67,9	590
930	681	590	681	588	679	592	678	590	679	590	68,0	590
940	679	591	678	590	679	589	683	593	679	591	68,0	591
950	683	590	676	588	683	592	681	590	678	588	68,0	590
960	681	590	678	590	678	589	674	590	676	589	67,7	590
970	678	590	679	590	681	589	679	591	679	589	67,9	590
980	678	589	681	592	678	588	676	588	681	590	67,9	589
990	676	592	678	589	678	589	678	589	679	589	67,8	590
1000	678	591	679	590	678	588	678	590	683	590	67,9	590

13.5 LwIP memory usage

Code size and RAM and ROM usage when compiled for the ColdFire. All values are in bytes.

- text is the as ambler code
- rodata are the variables that resides in ROM and are read only
- data are the variables that can be written and have an initial value witch means they reside in both ROM and RAM and are copied from ROM to RAM at startup.
- bss are the variables without an initial value, they reside only in RAM
- common are global variables used by multiple source files.

	text	rodata	data	bss	common	total ROM	Total RAM
src/core/ipv4/icmp.o	632	0	0	0	0	632	0
src/core/ipv4/ip_addr.o	110	8	0	0	0	118	0
src/core/ipv4/ip_frag.o	1656	8	0	7376	0	1664	7376
src/core/ipv4/ip.o	1002	0	0	0	0	1002	0
src/core/inet.o	1280	0	0	16	0	1280	16
src/core/mem.o	966	0	0	16017	0	966	16017
src/core/memp.o	458	44	0	5204	0	502	5204
src/core/netif.o	622	0	0	10	0	622	10
src/core/stats.o	818	407	0	0	1	1225	1
src/core/pbuf.o	1768	0	0	13063	0	1768	13063
src/core/raw.o	606	0	0	4	0	606	4
src/core/tcp_in.o	5536	0	0	44	4	5536	48
src/core/tcp_out.o	2540	0	0	0	0	2540	0
src/core/tcp.o	3668	13	6	1	20	3687	27
src/core/udp.o	1588	0	0	8	0	1588	8
src/core/dhcp.o	5902	0	4	0	0	5906	4
src/netif/etharp.o	2906	6	0	200	0	2912	200
src/api/api_msg.o	2674	0	44	0	0	2718	44
src/api/sockets.o	3806	0	44	6	0	3850	50
src/api/tcpip.o	206	0	0	9	0	206	9
src/api/api_lib.o	2560	0	0	0	0	2560	0
	41304	486	98	41958	25	41888	42081
src/support/io.o	4318	373	116	1	0	4807	117
src/support/printf.o	1854	0	0	0	0	1854	0
src/netif/ethernetif.o	2078	231	0	0	12244	2309	12244