IP-stack adaptation
Specialized for a microcontroller

Lukas Land
Abstract

Adapting of Ip-stack for a microcontroller

Lukas Land

An adjustment of the IP stack to a microcontroller has been attempted. The Ip stack manages IP traffic, which occurs through incoming outgoing data packets, and these packets are attempted to be stored with call buffers instead of current implementation. Programming was done in C, and a lot of time was spent looking through currently used code in order to better understand how it functioned. Data packets were then sent from a computer to the microcontroller, and by printing out variable locations and sizes, an understanding of how the stack worked and where data were stored, could be reached.

The project led to the insight that the project was more difficult than first estimated, the reason being that much of the software was based on the handling of data packets. The project resulted in an increased understanding of how the adaptation can be implemented, and how the current stack actually works.

Handledare: Greger Geldern, Sven-Åke Eriksson
Ämnesgranskare: Maria Wall
Examinator: Martin Sjödin
ISSN: 1401-5757, UPTEC F** ***/
Preface

My name is Lukas Land and I am a student at Uppsala University. I am studying a five year master program in Engineering Physics and as part of this study, I had the opportunity to do a bachelor assignment as well. I decided to take chance to learn something different and also get a look into a possible future work environment. I looked for an assignment outside the University and settled on a company in my home town. They proposed many different tasks but we eventually settled on one where the goal was to improve the IP-stack used in one of their microcontroller. More specifically the goal was to improve the handling of incoming data ”packets” so that the risk of losing packets was minimized. As I was working at a company, I worked on site for about 40 hours a week. In the beginning a spend A LOT of time learning about the simpler things concerning the assignment, such as the syntax involved with C-programming, network basics, and the basics of microcontrollers in general. After that the majority of the time was spent understanding the previously implemented stack, and figuring out how I could improve it. With the assignment finished I can say that I’ve learned a lot, for sure, both about computer networking, embedded systems and programming. I do however believe that the project might have been better suited as a master thesis project instead. 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 Greger and Sven-Åke for all their good advice and the time they spend helping me. I would also like to thank the other employees at Research Electronics for the good times spent in the coffee room.

Lukas
## Contents

1 Background ........................................... 4  
   1.1 Research Electronics ............................ 4  
   1.2 Objective ........................................... 4  
   1.3 Purpose and questions ............................... 4  

2 Theory ............................................. 4  
   2.1 Microcontrollers ...................................... 5  
   2.2 Drivers and porting .................................... 5  
   2.3 Processor operations and set-up .................. 5  
   2.4 Hardware interrupts ................................... 6  
   2.5 Pointers, memory and ring buffers ............... 7  
   2.6 IP stack .............................................. 9  

3 Method ........................................... 9  
   3.1 Set-up and tools ..................................... 10  
      3.1.1 Hardware ......................................... 10  
      3.1.2 Software ......................................... 10  
      3.1.3 Debugging ......................................... 10  
   3.2 Study ............................................. 11  
   3.3 Testing ............................................ 11  

4 Results .......................................... 11  

5 Discussion and conclusion ........................... 14  

A Appendix ........................................... 17
Populärvetenskaplig sammanfattning

En anpassning av IP-stacken till en microcontroller har försöks genomföras. IP-stacken hanterar IP-trafik, som sker genom inkommande utgående datapaket, och dessa paket försöktes lagras med ringbuffer istället för nuvarande implementation. Programmeringen gjordes i C. Projetet ledd fram till en insikt att projektets varit svårare än först beräknat, dr anledningen var att mycket av mjukvaran baserades på hantering av datapaket. Projektet resulterade i en ökad frstelse fr hur anpassningen kan genomföras, samt hur nuvarande stack egentligen fungerar.
1 Background

1.1 Research Electronics

The company is located in Siljansns, Dalarna, a small village around lake Siljan. They work a lot with microcontrollers deliver electronics that are tailored to customer requirements. They have a lot of experience with both hardware and software. Their electronic solutions can be found in elevators, ships, microphones and many other places, all around the world.

1.2 Objective

The objective of this project was to sort out and improve the implemented IP stack for one of the company's circuit boards. More specifically the way that incoming data packets were stored and handled was to be altered to more closely resemble the way the software handled other types of input. The packets were to immediately on arrival be moved to what’s known as a ring buffer(See Section 2.5). The ring buffer mechanism would have to be set up, and the packets would in the Ethernet interrupt service routine (See section 2.4), be moved from the short buffer that the hardware has access, to the larger ring buffer.

1.3 Purpose and questions

The reason that the packet queuing mechanism needs improving is so that it isn’t so time sensitive. If enough data packets would income in a short span of time, the previous mechanism could not handle all the packets and would start writing in already used memory which would mean that some packets would get overwritten.

2 Theory

This section covers the basics of the involved concepts. It is quite lengthy since a large portion of the project time was spent on learning about them.
2.1 Microcontrollers

A microcontroller is actually the microprocessor itself, which in its turn is a programmable calculator capable of reading scripts and performing various tasks.[1] The microprocessors are usually soldered on a circuit board together with other electronic components, which serve as input or output for the processor.

For example a circuit board might consist of a microprocessor, a diode, and a light sensitive sensor. Now say you intend to make the diode blink. This could relatively easily be done with an analogue circuit, which cuts the voltage to the diode as soon as the light sensor picks up the light, and then puts the voltage back on once the light is turned off. The processor is a very versatile way of doing these kind of tasks without having to construct a circuit of your own for each individual task you intend to execute. The construction part is then moved from actual hardware to software programming. This makes the microprocessor a very convenient and, through years of development, an extremely powerful tool.

2.2 Drivers and porting

Some tasks are not as simple as making a diode blink, and therefore require so called drivers to let the microprocessor know how to handle certain input. Examples of this is USB, WiFi, Ethernet and countless others. Drivers are universal and work on many different devices, but often needs to be ported to the different devices so that the intended process work as expected in the specific hardware environment. This is the job of a so called porting engineer which basically makes sure that after soldering a lets say USB port on a circuit board, the device handles the input as to be expected, or desired. While constructing a driver for a type of port or input, one must have great knowledge of the subject, but less so about the specific device or microprocessor that the driver will then work on, but the porting engineer on the other hand needs a lot of understanding about the specific device used, its constraints and its specifications.

2.3 Processor operations and set-up

As previously mentioned, the microcontroller can be regarded as a powerful calculator capable of executing scripts to perform various tasks. The way the processor
does this is by reading lines of code and taking the corresponding action. Actually, the code is first translated into bits, ones and zeros, and packed into packets that the processor can read. The size of these packets depend on the type of processor. For example, a 64 bit computer processor operating at 4GHz clock speed performs a whooping 4 million operations each second, and each operation takes 64 bits (or 8 bytes) as input, and produces the same amount of bits as output. \[1\]

The processor can be set to execute lines infinitely by using a "while loop". The processor will read the code inside this loop forever, or until the power is turned off. This method is called a "super loop" and is frequently used. Another way to set up the processor is to use what’s known as an operating system, which also contain a super loop, but is able to prioritize and select which operation should be processed at any given time. This helps with larger systems bit might cause the system to slow down a bit since the operating system itself needs to continuously be processed. Using an operating systems might also complicate the software and give less control to the programming for what happens when etc.

2.4 Hardware interrupts

The processor can be forced to hop to certain code blocks instantly with what’s known as an interrupt. \[2\] There are interrupts for many different purposes but they ensure that the processor will execute desired code instantly once they occur. A simple example to demonstrate the usefulness of interrupts is presented below.

**Demonstrative example of the usefulness of interrupts**

A processor is set up to run a super loop and will execute the code inside the loop infinitely, or until the power is turned off. Inside the loop the processor checks to see if an external button connected to the processor has been pressed. This is done by reading a buffer attached to the button with the capacity to store one bit. The bit is one if the button is pressed, and zero if the button is not pressed, and the processor resets the buffer back to zero each time it checks its value. If the processor reads a one, it then increments a local variable located in the on-chip memory in order to keep track of how many times the button has been pressed.

A problem will arise if the button is pressed twice or more before the value
of its buffer is checked and reseted. The processor will then read a one from the button buffer value, but interpret it as only one button press. Without hardware interrupts, the only other way of fixing this problem would be to shorten the time it takes for the super-loop to revolve, and this is not always possible since the super-loop might be responsible for many different tasks.

With hardware interrupts this is done with relative ease. The button press would then be set to trigger an interrupt, causing the processor to execute desired code. This code is often referred to as an ”interrupt service routine”. Inside this routine the local variable could be incremented and the button buffer value could be reseted to zero. Since the interrupt mechanism works instantly, rapidly pressing the button would no longer be problem.

The interrupts needs to be set up in the processor start up files, and also associated with the desired code block in order to work.

2.5 Pointers, memory and ring buffers

One area in which programming for a microprocessor really differs from, for example programming an application for a modern PC, is memory management. The available memory space on a PC is often huge in comparison, while a microprocessor might only have access to a few kilobytes or less, for a given mechanism. This means that the programmer has to be more aware of where, and how much, memory is allocated so that the processor doesn’t run out of memory and malfunctions.

Pointers

A very useful function in C programming is the ”pointer”; a 4 byte variable type that can be set to point to another variables memory address, or an arbitrary memory address.[3] This is very useful when setting up memory spaces and buffers, and also when extracting data from these. The increment command (++) can be applied to these pointers in order to tick the pointer up in memory, and the distance the pointer moves is dependent on the size of the type that the pointer was declared as. A pointer of the type ”char” will still be 4 bytes in size, but when incremented will point only 1 byte ahead in memory. An int pointer would
instead point 4 bytes ahead if incremented, and if using a struct pointer, which is a pointer of a user defined type which can contain other data types, the pointer could move an arbitrary amount of bytes ahead.

**Ring buffers**

The ring buffer is a buffer structure which works very well combined with interrupt based data writing. It called a ring buffer since it can easily be visualized as a circular buffer where the last available memory slot is followed by the first one (see Figure 2). The handling of writing and reading data from a ring buffer is done by using two of the previously discussed pointers. The pointers are commonly referred to as fill and empty pointers. The fill pointer decides where data should be placed, and is incremented to point to the next slot each time data is written. The empty pointer points to the oldest unprocessed data, and is incremented every time data is read from the buffer. The final thing which differentiates the ring buffer from any normal buffer, is a wrap around check. This is done each time a pointer is incremented to see if it still points in the points inside of the buffer memory. If the pointer has been incremented to far, and point outside of the buffer memory, it is simply set to point to the start of the buffer instead.[4]

So, a ring buffer consists of a fill- and an empty pointer, and two variables to hold the address of the start and end of the allocated buffer memory.
2.6 IP stack

The IP stack could be described as the process of receiving and transmitting IP datagram.[5] IP datagrams are data packets, containing a file or message, which also contain an "IP-header", meaning additional information regarding the sender and the intended receiver. Computer networks are often separated into different "layers", where each one is responsible for different parts of the network processes. This is presented in Figure 2, where the IP protocol is located at the third layer, meaning that the IP stack is not concerned with higher level layers and without them is instead used in local networks for path determination and addressing. The layers underneath are also of interest since they handle the communication between the processor and the Ethernet port, and the memory management. The physical layer is one of the most difficult and most important layers, as it translates bit streams (i.e., alternating voltage square waves) to information which can then be read by the upper layer; the "Data Link". The physical layer on a circuit board is usually implemented by a so-called PHY chip, which is an off-chip component connected to both the processor and the Ethernet port. The upper layers on the other hand, are concerned with global data transport and addressing and finally application which handles the underlying layers seamlessly.

3 Method

Since previous knowledge in programming was limited to higher level programming languages such as C# and Java, a lot of the time was spent learning about C programming. This process was intertwined with sorting out the existing code and relating it similar mechanisms which resembled the desired structure. Examples of these were the present implementation of CAN and UART, which were interrupt driven and stored data in a ring buffer structure.

Figure 2: The layers of the network model called the OSI model.
3.1 Set-up and tools

This section describes the different tools used in order to conduct the research and test code etc. To see how these were used during the assignment, see Section 4.

3.1.1 Hardware

The following hardware were used in the project:

- Two computers, both with Ethernet ports
- Two microcontroller circuit boards
- Two USB to COM port dongles
- An oscilloscope

3.1.2 Software

The software, and their respective purposes, that were used in the project:

- IAR embedded workbench, Programming and compiling
- Visual Studio Code, Programming
- Monitor software developed by the company, Debugging
- Flash Magic, Flashing

3.1.3 Debugging

Debugging was usually done by calling a print method in the code which printed out variable values and text to the UART output port on the circuit board which in its turn was connected to a monitoring software on the PC.

However, this method had its limits. The print methods didn’t work very well if they were called in ISR’s and could sometimes cause the processor to freeze up. Therefore prints to the monitor were almost always avoided. An alternative method was instead used, where a pin on the circuit board would in the code be set to either high or low voltage in the start of the program. The voltage of the pin was

1Interrupt service routines
could then be changed in the code and measured with an oscilloscope to ensure that certain code had been executed. Of course this could not present variable values and other important information but turned out to be incredibly useful during the assignment.

3.2 Study

A large portion of the project was spent on studying the previously implemented code in order understand how packets were received and stored. This was done continually though the project.

3.3 Testing

In order to understand where the data was stored etc. a lot of testing and debugged had to be done. This, combined with a manual of the processor and it’s memory map, turned out to be a great way of understanding the mechanism. In the initialization of the memory buffers, the memory addresses of many variables were printed out. Data packets was then sent from a computer using a software called Packet Sender, and the places were different data was placed could be seen.

4 Results

The result from this project was an understanding of how the current buffer structure works, the possibility and difficulty of restructuring it into the desired structure, and a small improvement of the current buffer that increased its capacity. Thus this section will present the discoveries made though studying the code.

Current buffer structure

The current buffer structure was found to work in the following way. During the initialization of the processor, meaning when the microcontroller is turned on, a method called pk_init is called. This method sets up ”raw packets”, a struct of 44 bytes, inside of a for-loop. This sets up as many ”raw packets” as we wants and manually allocates this memory meaning that they will always be there in that
memory space. This seemed very strange since the expected data packets were expected to be a lot larger than 44 bytes.

These “raw packets” were revealed to NOT be the actual data itself, and in order to avoid confusing moving forward we will refer to them as “packet controllers” instead, since that’s essentially what they turned out to be. The packet controllers did contain a pointer to the actual packet buffer located in the memory space recommended for Ethernet traffic, which were of the correct size. A cleaver mechanism was discovered and understood in this process, which might speak for this way of setting the buffer up; the distinction of big and small packets buffers.

This meant that some packets could be stored as full sized in a big packet buffer, while some other could be stored as in smaller packet buffers if the fitted, while the packet controllers would all be of the same size; 44 bytes.

So, packet controllers are initially set up in memory. These are not added to an array for later accessing, as one might expect, but instead added to a linked list referred to as a ”queue”. Two queues were created initially; littlefreeq, and bigfreeq, and they should be thought of as the packet controllers with an associated packet buffer, either big or little, that’s free. This was incredibly confusing as first, partly due to the fact that the struct for the packet controller was actually called ”rawpacket”.

These packet controllers, and thereby also the packet buffer containing all the actual data could now be allocated by calling a function called pk_alloc(queue q) and returned to the free queues by calling pk_free(queue q). Note that terminology of using ”pk” in the function names serves to further confuse the reader since an actual packet could never be allocated, they simply arrive and are stored.

What’s actually allocated and freed is a packet controller associated with either a big or small packet buffer. Allocated and free simply refers to whether the packet controller is part of a ”free queue”(either lilfreeq och bigfreeq). So for example a packet controller would be allocated in order to let its packet buffer be used, and freed by letting the packet controller be returned to the queue.

The reason for overstating this so much is because understanding this mechanisms was one the most difficult parts of the assignment.
The Ethernet software interface was set up initially as well. This interface was the type struct eth_params (See Appendix) and contained a lot of relevant data members. The most relevant ones to us were the RX_DESC and RX_PEND data members, and although we will not go into much detail of how this works here, its this mechanism that ties the packet controllers to the actual data being written to memory by the PHY. To put it simple, the Ethernet interface has three packet controllers allocated, meaning they’re not part of any free queue, but are associated with a packet buffer. These packet controllers are always of the type associated with a big packet buffer. When data is written into the packet buffer an interrupt is triggered.

In the ISR, the function eth_recv is then called, which returns one of the three packet controllers allocated by the Ethernet interface, the one associated with the just filled packet buffer. The ISR then places this packet controller into another queue called the ”rcvdq”, the receive queue. The packet controllers part of this queue is bound to not be part of any free queue, and is also associated with the packet buffer containing the unprocessed, newly arrived data. The term ”associated” in this context, really refers to the packet controller struct having a data member pointing to the packet buffer, as discussed earlier.

Data processing was done by polling the length of the receive queue in order to see if it contained any packet controllers associated with packet buffers containing unprocessed data packets. The data packet could then be located and operated on by using a function called getq(queue q), which returned the relevant packet controller, and then accessing the packet controllers packet buffer. The packet buffer in structured in a certain way where information about the the packet and its payload size etc. is stored in the first part of the buffer. These information parts are called ”headers” (See Figure 3) and is added by the transmitter in order for the packet to be transmitted correctly and then

Figure 3: The encapsulation of UDP packets.
read by the receiver in order to interpret the package correctly.

Action could then be taken depending on the payload and finally the packet controller was freed using the pk_free function so that it’s packet buffer could once again be written to.

5 Discussion and conclusion

The desired structure for handling incoming data packets were a ring buffer of sufficient size, meaning enough packets could be fitted in to it without having to worry about packets being overwritten or running out of memory. The research revealed that the already implemented stack was a lot more complicated, and perhaps cleverer, than this.

This might have been because the implemented stack is intended to be used in a variety of different programming environments and conditions. In this application where small packets are expected and the company likes to work with high control over the machine and as little overhead code as possible this obviously is not the best solution for them (hence the reason for the project).

The incoming packets were placed into memory which was then linked to the control packets located in another memory and the control packets was then flagged as allocated. The control packets were in their turned parts of different queues which was essentially a linked list. The queues were sorted into three different categories depending on the type of raw buffer they were connected to; littlefreeq, bigfreeq and rcvdq, meaning the packet controllers tied to a little raw buffer, big buffer, and some control packets allocated to have their raw buffer directly written to by the PHY on incoming data.

During the final stages of the process, where an understanding of underlying mechanism had been made, the conclusion that the current code was unnecessarily complicated could be drawn. This relates back to the initial questioning, if a ring buffer mechanism could be made and also how difficult that process would be. The conclusion here is that it could definitely be made, by scrapping the control packet mechanism entirely and instead accessing the raw buffer memory directly.

This could simplify the entire process, although perhaps at the cost of versatility,
and would definitely work since all the relevant information is already stored in the protocol headers, which are also in the raw buffer. The task of doing this however would almost certainly require a complete rewrite of all the documents involved with the stack. The reason for this is because the control packets are used almost everywhere throughout the code, as input and output of different functions.
References

How stuff work: How microprocessors work [updated 2008; read 2018-06-08]
Available via:
https://computer.howstuffworks.com/microprocessor.htm

Atari Magazine: Interrupts made easy (computer hardware interrupts) [updated Feb 1993; read 2018-06-08]
Available via:
https://www.atarimagazines.com/compute/issue149/60_Interrupts_made_easy.php

Online Computer Teacher: Pointers in C programming language A tutorial for beginners [updated 2017-11-17; read 2018-06-08]
Available via:
http://www.onlinecomputerteacher.net/pointers-in-c.html

Embed Journal: Implementing Circular/Ring Buffer in Embedded C [updated May 2014; read 2018-06-08]
Available via:

Tech Target: OSI model (Open Systems Interconnection) [updated 2018-05-18; read 2018-06-08]
Available via:
https://searchNetworking.techtarget.com/definition/OSI

16
A Appendix

Rawpacket

The rawpacket struct, referred to in the text as "packet controller". The data member beg_raw_buff points to the beginning of the actual packet buffer, where the data packet is located.

```
struct rawpacket {
    struct rawpacket *next; // queue link
    char  *beg_raw_buff;    // beginning of raw buffer
    unsigned int raw_buff_length; // length of raw buffer
    char  *beg_protocol;    // beginning of protocol/data
    unsigned int protocol_length; // length of protocol/data
    char *m_data;           // pointer to TCP data in nb_buff
    unsigned int m_len;     // length of m_data
    long   tstamp;          // packet timestamp
    ip_addr f_host;         // IP address associated with packet
    unsigned short int type; // IP==0800 filled in by receiver(rx)
    or net layer.(tx)
    unsigned inuse;         // use count, for cloning buffer
};
typedef struct rawpacket *PACKET; // struct netbuf in netbuf.h
```
Image credits

Figure 1
G. Rivero
(https://commons.wikimedia.org/wiki/File:Osi-model-jb.svg),
..OSI RM Model, License:
https://www.gnu.org/licenses/fdl-1.2.html

Figure 2A
Muhammad Ajjan
(https://commons.wikimedia.org/wiki/File:Circular_Buffer_Animation.gif),
..Circular buffer animation, License
https://creativecommons.org/licenses/by-sa/4.0/#
This image has been altered form it’s original form. It can be used freely by anyone
without further restrictions apart from the original CC license, linked above.

Figure 2B
I, Cburnett
(https://commons.wikimedia.org/wiki/File:Circular_buffer_-_empty.svg),
..Circular buffer - empty, License:
https://creativecommons.org/licenses/by-sa/3.0/legalcode

Figure 3
(Wikipedia users) en:User:Cburnett original work, colorization by en:User:Kbrose
(https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg),
..UDP encapsulation, License:
https://creativecommons.org/licenses/by-sa/3.0/legalcode