# Embedded Software Development for the Internet Of Things


Klaus Elk

# Embedded Software Development for the Internet Of Things

# Embedded Software Development for the Internet Of Things

Klaus Elk

August 30, 2016



*See also: facebook.com/klauselkbooks and www.klauselk.dk*

# Chapter 1
# Introduction

## 1.1 Preface

This book is not a programming textbook. It is not a Software Engineering book either. Software Engineering typically only deals with the processes surrounding software development - e.g. Agile or Waterfall.

This book is about embedded software development. That means that you will see some programming - almost exclusively in “C” - as well as something about processes and tools. Nevertheless, the main scope is to cover the major domains an Embedded Software developer will typically meet. Operating Systems and CPU’s are the most basic and an overview of these is what Part I is all about.

The target is development for IOT - the Internet Of Things. The main difference between embedded and IOT is that IOT is always using the Internet Protocol stack[1] . For this reason Part II has two textbook chapters on networking - wired and wireless (with focus on WiFi). Wireshark is heavily used here to visualize the theory. Since IOT is very often about interpreting data from sensors, basic Digital Signal Processing is relevant. Many modern CPU’s have built-in DSP’s for this reason and Part II therefore also has a textbook chapter on digital filters.

Finally Part III is about Best Practices in software architecture as well as tools and processes. Throughout the book the author draws on good and bad real-life experiences, which makes the book informal and relevant.

## 1.2 Who is this book for?

Details can be found on the web - when you know what to look for. The goal of this book is to provide a qualified overview and help the reader to find more. It may be read by senior students, but it is more likely to be read by programmers and developers who have done some programming of PC’s in Windows or Linux and are now faced with developing devices for the Internet Of Things. Basic programming skills are thus assumed whereas no prior knowledge on TCP/IP and digital signal processing is needed. Based on 30+ years’ experience as a developer, university-teacher and manager the author’s personal views on best practices in the embedded world are offered.

- Part I introduces the main features of Operating Systems and the current state on CPU’s and thus mainly builds the foundation for part II.
- Part II - on Internet Protocols and Digital Filters - is the main “textbook” and measured in pages is half the book. It gives a lot of facts, backed up by examples.
- Part III is the author’s experience with best practices when working with mainly embedded networking. This is more than general processes - many examples of tools and how to use them are given.

The three parts may be read individually with references between them serving as

appetizers. Each chapter ends with a section containing suggestions for further reading.

## 1.3 About The Author

Klaus Elk graduated as a Master of Science in Electronics, from the Danish Technical University in Copenhagen in 1984 with a thesis on Digital Signal Processing. Since then he has worked in the private industry within the domains of telecommunication, medical electronics, and sound and vibration. He also holds a degree as a Bachelor in Marketing. In a period of 10 years Klaus Elk was - besides his R & D job - teaching at the Danish Technical University. The subjects were Object Oriented Programming (C++ and Java), and the Internet Protocol Stack. Today he is R & D Manager in Instrumentation at Brüel & Kjær Sound & Vibration.

# Part I
# The Basic System

# Chapter 2
# How to select an OS?

In “the old days” you would pick a CPU first and then discuss the Operating System (OS) - after discussing whether such was really necessary at all. Today it is more common to choose the OS first and then a CPU - or a CPU-family. In reality it is often an iterative process. You may e.g. decide to go for Linux, but this requires an MMU (Memory Management Unit), and this may drive you towards too big and expensive CPU’s. In that case you may have to redo you original choice. The following table is a “kick-start” to this chapter - showing an overall trend from simple to advanced:

| OS/Kernel/Language | Type |
|---|---|
| Simple Main | Strictly Polling |
| Ruby | Co-Routines |
| Modula-2 | Co-Routines |
| Windows 3 | Non-preemptive scheduler |
| ARM *mbed* simple | Interrupts + Main with FSM |
| OS-9 | Preemptive Real-Time kernel |
| Enea OSE | Preemptive Real-Time kernel |
| Windows-CE | Preemptive Real-Time kernel |
| QNX Neutrino | Preemptive Real-Time kernel |
| SMX | Preemptive Real-Time kernel |
| Windows-NT | Preemptive OS |
| ARM *mbed* advanced | Preemptive scheduler |
| Linux | Preemptive OS |
| RT-Linux | Preemptive Real-Time OS |
| VX-Works | Preemptive Real-Time OS |

We will dive into the various types and degrees of operating systems and their pros and cons. Along the way important parameters are introduced. The solutions are ordered in the most reader-friendly way towards a full **preemptive RTOS** - “Real Time Operating System”.

## 2.1 No OS and strictly polling

The simplest embedded system has no operating system, which leaves some low-level details to the programmer. If you are using “C” there is a “*main()*” function that your “official” program starts from at power-up. Since there is no OS, this must be assured by configuring compiler, linker and locater [1] - and also by calling a small assembly program first that copies the program to RAM, disables interrupts, clears the data-area and prepares the stack and stack-pointer.

*I once used a compiler-package that did all the above. Unfortunately, the vendor had*

*forgotten to create the code that executes the code with all the global C-variable-initializations, something that you normally take for granted. So after realizing this, I had the choice between completing the tool myself or remembering to initialize all global variables explicitly in a special "init" function in main(). This is a typical example of the difference between programming in the embedded world and on a PC where the tools are more "polished" than those for the smaller systems.*

"*main()*" in an OS-less system has an infinite loop that could look like this:

```
01 int main(int argc, char *argv[])
02 {
03    for(;;)
04    {
05       JobA();
06       JobB();
07       JobA();
08       JobC();
09    }
10 }
```

**Listing 2.1:** Round-Robin Scheduling

This is a "**Round-Robin**" scheme with the slight enhancement that JobA() has gotten more "attention" (not really priority) by letting it get access to the CPU with shorter intervals than the other processes. Inside each job we read the relevant inputs from our code when we have the time. This is known as "**polling**" We might even make a loop where we test an input again and again until it goes from one state to another. This is called "**busy-waiting**" as the CPU does nothing else than loop. Introducing such a loop in say JobB, will however be a disaster for JobA and JobC - they will not be executed until this state-change occurs. And what if the state-change we are waiting for in this loop is actually depending on JobA or JobC starting something? In such a scenario we would have a **deadlock**. Another problem with a busy-wait loop is that you waste a lot of energy, as the CPU is not allowed to go into any form of power-saving. So busy-waiting in a loop may at times be OK - but not in a system as simple as this.

Another concept that is still without any operating system whatsoever, but a lot more clever, is to introduce FSM's - Finite-State-Machines, where you read all inputs, decide what has changed and take action as shown in Listing 2.2.

```
01 int main(int argc, char *argv[])
02 {
03    for(;;)
04    {
05       ReadSensors();        // Read all inputs
06       ExtractEvents();      // Is temp above limit?,
07       StateEventHandling();    // Take action
08    }
09 }
```

**Listing 2.2:** Main with Finite-State-Machine

Listing 2.3 is one of three Finite State Machines that together control a TOE - TCP Offload Engine. The TOE implements the actual transmissions of TCP in hardware while the rest is handled in the embedded software. Later we will look into sockets and TCP, and it will be clear that the listing very directly represents a good part of Figure 4.9 which is a graphic representation of the TCP connection states. For now it is however more relevant to understand the concept of an FSM.

Each column is a state on "this" side of a TCP-socket that at a given time is the "current state". Each row represents an event that occurs while in this state - e.g. an ACK

has been received. Each element in the table contains the action to take, and the next state. In order to fit the table into this book it has been split in two. In the real C-code, the part after “Table continuing here” is placed to the right of the lines above, so that we have a table with 7 rows and 7 columns. Clearly FSM’s are not just practical in a simple OS-less system but can be used anywhere. The FSM shown in Listing 2.3 was used in a Linux system. FSM’s are very popular among hardware designers, but not used by many software designers which is a pity. Nevertheless, many modern frameworks contain FSM’s inside and offers “event-driven” models.

```
01 struct action connected_map[EV_MINOR(EV_ENDING_COUNT)][ST_MINOR(ST_ENDING_COUNT)] =
02 {
03 //st_normal        st_close_wait    st_last_ack      st_fin_wait_1   //EVENT
04 //NORM             CL_W             LACK             FW_1            // ev_end_
05 {{error,    NORM},{error,    CL_W},{error,    LACK},{error,    FW_1},// <error>
06 {{send_fin, FW_1},{send_fin, LACK},{no_act,   LACK},{no_act,   FW_1},// close
07 {{error,    NORM},{error,    CL_W},{req_own,  OWN },{fw1_2,    FW_2},// RxData_ACK
08 {{ack_fin,  CL_W},{error,    CL_W},{error,    LACK},{ack_fin,  CL_G},// RxData_FIN
09 {{error,    NORM},{error,    CL_W},{error,    LACK},{ack_fin,  TM_W},// RxData_FIN_ACK
10 {{error,    NORM},{error,    CL_W},{fin_to,   CL  },{fin_to,   CL  },// TiTimeOut
11 {{abort,    GHO },{abort,    GHO },{abort,    GHO },{abort,    GHO },// RxDataExc_RST
12 };
13 // Table continuing here
14  st_fin_wait_2    st_closing       st_time_wait                    //EVENT
15  FW_2             CL_G             TM_W                            // ev_end_
16 {error,     FW_2},{error,    CL_G},{error,   TM_W}},               // <error>
17 {no_act,    FW_2},{no_act,   CL_G},{no_act,  TM_W}},               // close
18 {error,     FW_2},{cl_ack,   TM_W},{error,   TM_W}},               // RxData_ACK
19 {ack_fin,   TM_W},{error,    CL_G},{error,   TM_W}},               // RxData_FIN
20 {error,     FW_2},{error,    CL_G},{error,   TM_W}},               // RxData_FIN_ACK
21 {req_own,   OWN },{req_own,  OWN },{req_own, OWN }},               // TiTimeOut
22 {abort,     GHO },{abort,    GHO },{abort,   GHO }},               // RxDataExc_RST
```

**Listing 2.3:** One of three Finite-State-Machines for a TOE

One really nice thing about FSM’s is that they are both implementation and documentation, and they typically can fit into a page on screen. Compared to a number of *if-else* or *switch* clauses it is much more “clean” and therefore much easier to create and keep error-free. With the simple overview it is easy to spot a missing combination of a state and an event. It is also a compact solution; in the example the same FSM-code works for all sockets, we only need to keep the current state per socket, and the statemachine is called with two parameters only - the socket handle and the incoming event. Incidentally - when not coding in C++ but in C it is a common pattern that the first parameter is the “object you do not have”. Thus, if the C++ version is “*socket->open(a, b)*”, this becomes “*open(socket, a, b)*” in C.

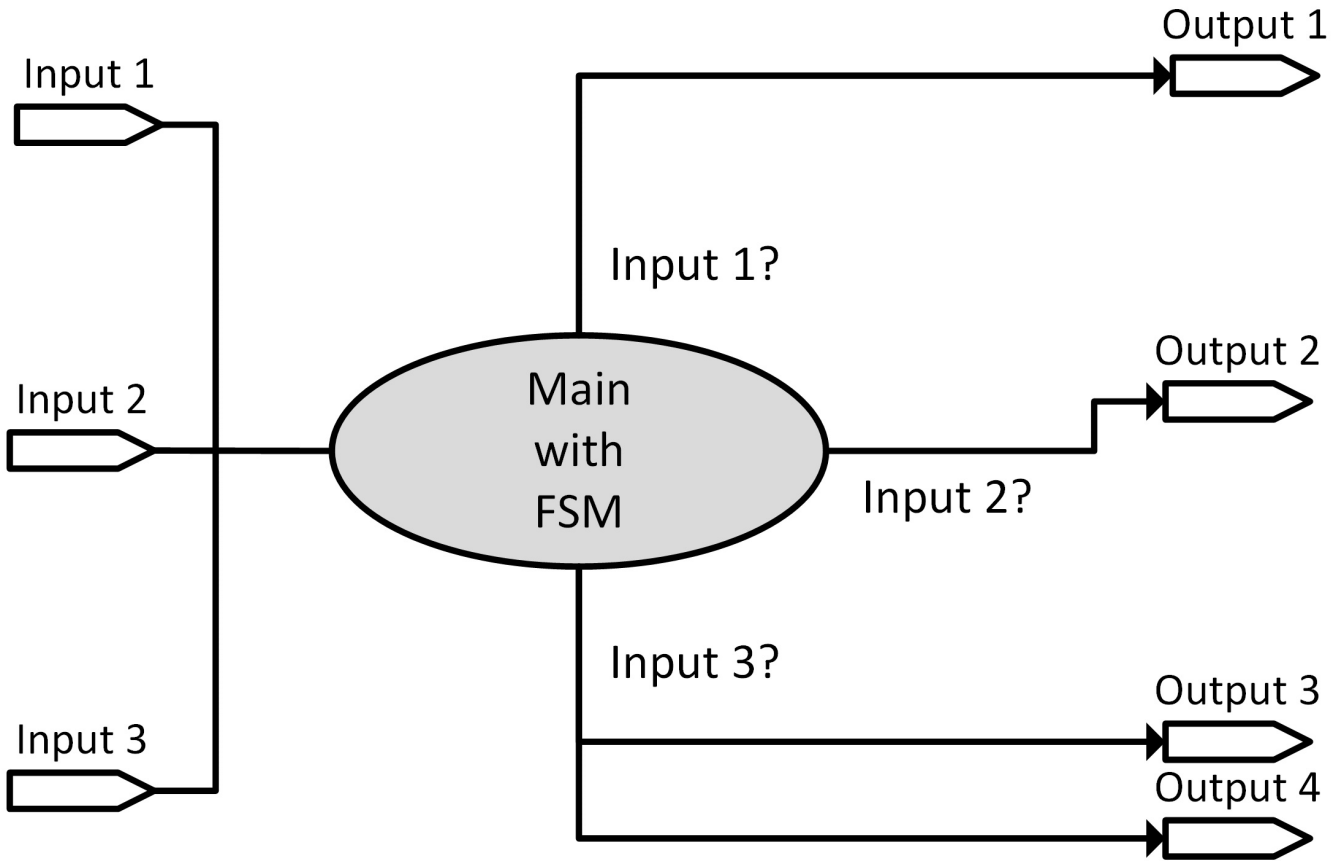



**Figure 2.1:** Finite State Machine in Main

The main advantage of the OS-less system in Figure 2.1 is simplicity. There is no third-party OS that you need to understand and get updates of. This can be very relevant if the system is to last many years. A part of this simplicity is that the application may read inputs and write to outputs directly. There is no "driver" concept. This can be really nice in a small system with a single developer, but it is also the downside as a simple error can lead to disasters. Figure 2.1 introduces a small setup that we will see a couple of times more:

- Input 1 - which leads to some processing and eventually to a change in Output 1.
- Input 2 - which leads to some processing and eventually to a change in Output 2.
- Input 3 - which leads to some processing and eventually to a change in Outputs 3 and 4.

## 2.2 Co-routines

Co-routines are not like the tasks (which we will get back to) of an OS - but they do have similar traits:

1. There can be many instances of the same co-routine - typically one per resource - e.g. an actor in a game or a cell in a biological simulation.
2. Each instance can pause at some point while the CPU executes something else. It keeps its state and can continue on from the given point.
3. This pause must be invoked by the co-routine itself by "yielding" to another co-routine. There is however no caller and callee. This is supported by some languages -

not “C” but e.g. by Ruby and Modula-2.

Co-routines are mostly of academic interest in today’s embedded world. They might come in fashion again - you never know.

## 2.3 Interrupts

Instead of polling the various inputs, **interrupts** are generated when inputs change. One or more interrupt routines read the inputs and take action. An interrupt is what happens when an external event in hardware asynchronously triggers a change in the execution flow. Typically, a given pin on the CPU is mapped to an **interrupt-number**. In a fixed place in the memory-layout you find the **interrupt-vector** which is an array with a fixed number of bytes per interrupt - containing mainly the address that the CPU must jump to. This is the address of the **Interrupt Service Routine** (ISR). When entering the ISR all interrupts are disabled[2] . It is up to the ISR whether or not to re-enable interrupts.

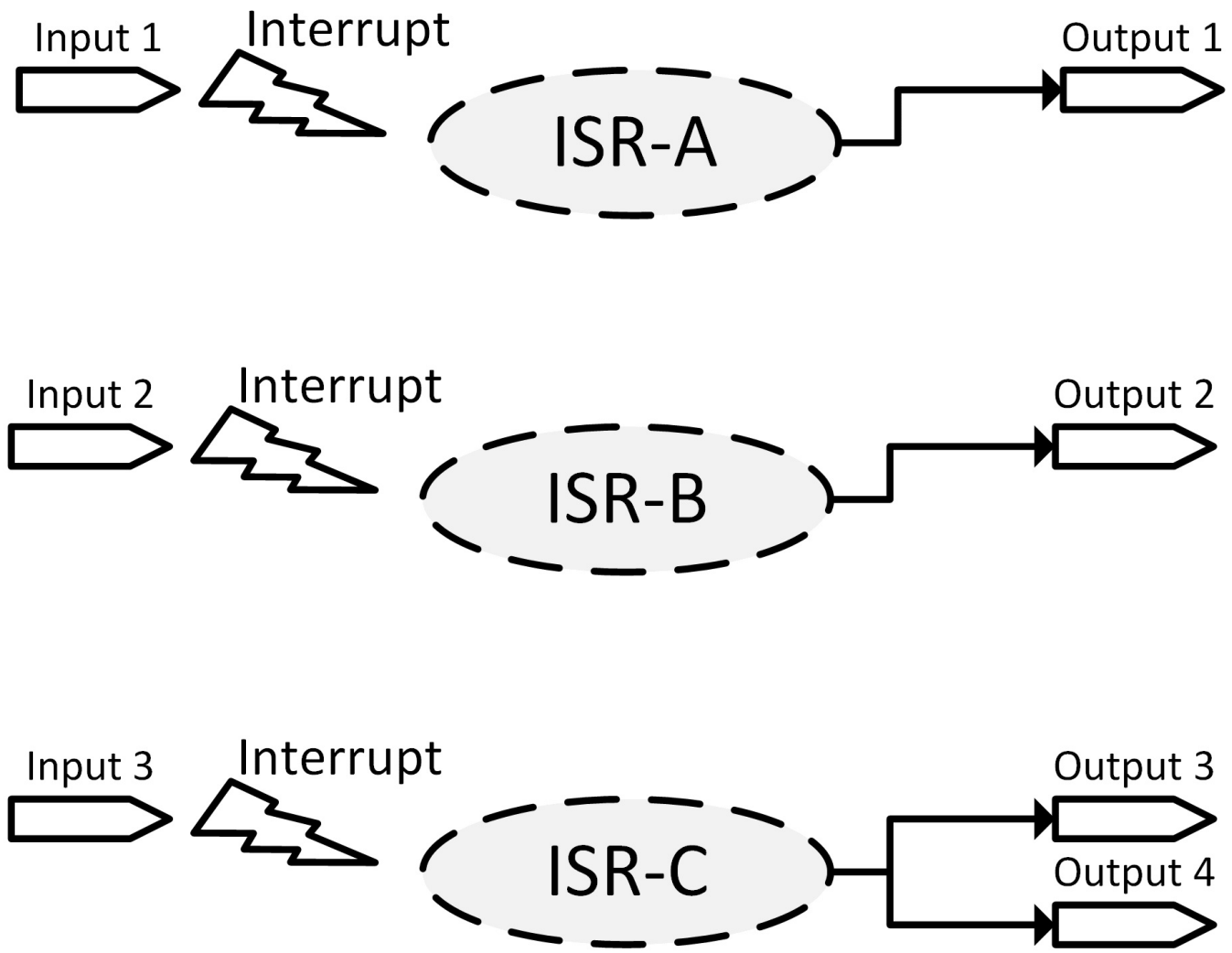



**Figure 2.2:** Purely Interrupt Controlled System

In such a “purely interrupt-controlled system” the Interrupt Service Routine may do everything that is related to a given event. See Figure 2.2 There are variants of such a system:

1. Each input has its own Interrupt Service Routine (ISR), and its own interrupt priority. Thus one interrupt may interrupt the main-loop (stacking the registers it plans to use), and then this may become interrupted by the next, higher level interrupt etc. This is known as **nested** interrupts. Nested interrupts is very normal in the bigger systems, but if all actions to inputs are done inside the interrupt routines, it becomes very important that the nested interrupt "understands" the exact state of the system, and of course this depends on how far we got in the interrupted interrupt. Furthermore, many systems do not have interrupt levels enough to match all the inputs.
2. As above, but nested interrupts are not allowed, so that all other interrupts must wait until the first is done. This is however really bad for the "interrupt latency" - the worst-case reaction time - on the highest prioritized interrupts.
3. Many inputs trigger the same interrupt, and the first thing the ISR must do is to find out which input actually changed state. This is **daisy-chaining** interrupts. The order in which you test for the various events becomes a "sub-priority" so to speak.

*A particularly nasty problem with interrupts, that I have experienced, is related to the difference between "edge triggered" and "level triggered" interrupts. If an interrupt is level-triggered, you will keep getting the interrupt until the level is changed - either by the hardware itself or from code - typically in your ISR. An edge-triggered interrupt, on the other hand, only happens at the up- or down-going edge of the pulse. If your interrupts are disabled in that short instant, you never get an interrupt - unless the edge is latched in CPU-hardware, which is not done in all CPU's.*

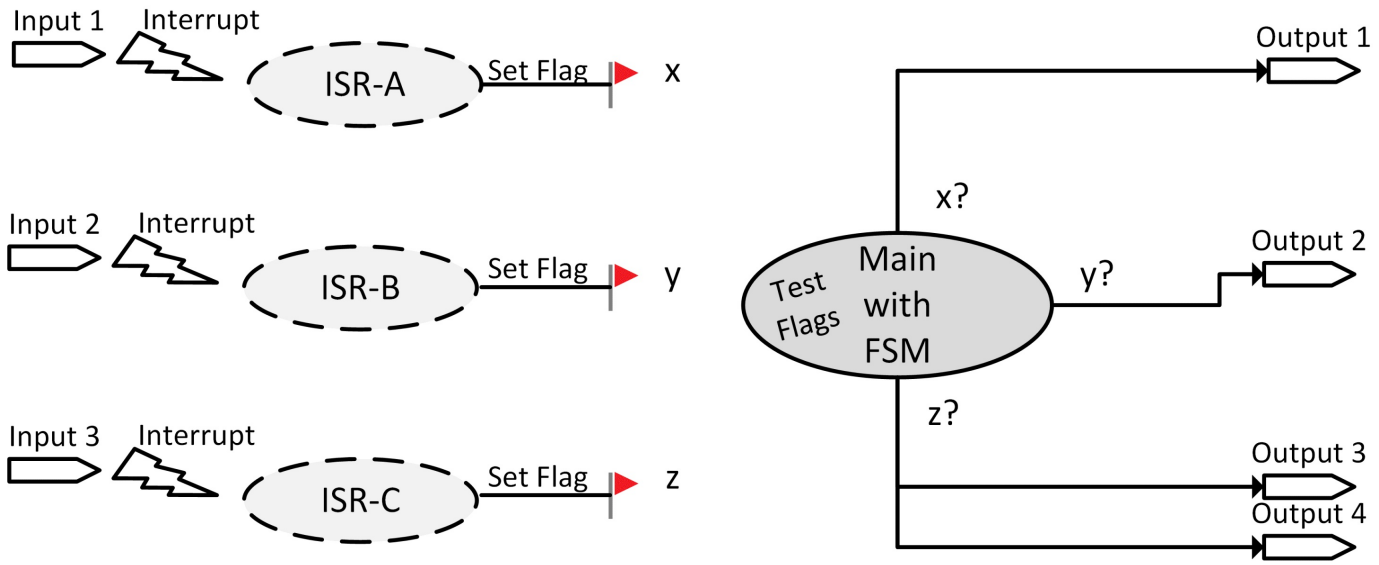



**Figure 2.3:** Interrupt System with Finite State Machine in Main

The general rule in any good system with interrupts is like guerrilla warfare: "move fast in, do the job and move fast out". This is to achieve the best interrupt latency for the other interrupts. This means that the actual ISR only does the minimal stuff needed. This

could e.g. be to set a flag or read a sample from an A/D-converter before it is overwritten by the next sample. In the latter case the ISR will save the sample in a RAM-based buffer, which later will be read from a standard process or task. In such a system the interrupt-latency must be less than $1/f_s$ - where $f_s$ is the sample-frequency. A system like this can thus detect external events very fast, but it still offers no help to the developer in terms of multi-tasking (a concept we will see shortly).

If, however the main-loop is broken up into very small pieces of code that may even be organized with the help of Finite State Machines, then it is possible to react to the flags set in the ISR as soon as one of the small pieces of code is done, and then decide (via the FSM) what the next piece of code is, see Figure 2.3.

This is exactly what ARM has done in their basic version of the free *"mbed"* OS. Here the flags from the ISR's are called events. ARM *mbed* prioritizes the interrupts in the usual way and they also offer priority on the "pseudo threads" - the small pieces of code. This simply means that if "pseudo threads" A and B both are waiting for an event from the same interrupt, then the one with the highest priority is started first. Since all these "pseudo threads" are started on a specific point and run to end, there is no *preemption* - one task in application code never takes over the CPU from another - it is only interrupted by ISR's and these can use the single CPU stack for the specific registers they use. This saves a lot of RAM-space, and is very practical in a small system.

Thus *mbed* is tailored for e.g. the small 32-bit Cortex M0 CPU's that have scarce resources (including no MMU). What makes *mbed* interesting is that it has a lot of the extras that are normally seen on larger OS'es, TCP/IP stack, Bluetooth LE stack etc. It also boasts a HAL (Hardware Abstraction Layer) that makes the code the same for other CPU's in the ARM family. In this way *mbed* is well positioned and does look very interesting.

Note that the ARM *mbed* alternatively can be configured to use a preemptive scheduler as described in the next section. This takes up more space, but also makes *mbed* a member of a more serious club.

## 2.4 A small real-time kernel

Typically the aforementioned concepts are only used in very small and simple systems. It is really practical to separate the various tasks. A real-time kernel offers exactly that - a *task* concept. You can also say that a kernel is all about managing resources. The basic theory is that you set aside a task for each independent resource in the system. This could be a printer, a keyboard, a hard-drive or a production-line "station" (or parts of this). It is not uncommon though to have more tasks, if this makes your code more maintainable. It is however not a good idea simply to assign a task to each developer as this will require more coordination among tasks. The less coordination you need between tasks the better. In fact - almost all the things that can make the use of a kernel complex is related to interaction between tasks. Tasks may be in one of the following states (See Figure 2.4):

- **Dormant**
  The task is not yet brought to life. This is to be done explicitly by the application.

- **Ready**
  The task can run - it's just waiting for the current "running" task to "get off the CPU".
- **Running**
  Actually executing code. There can only be one running task per CPU core.
- **Blocked**
  The task is waiting for something to happen. This could e.g. be a socket in a *recv()* call - waiting for in-data. When data comes, the tasks becomes "Ready". Incidentally - a socket will also block if you write to it with *send()* and the assigned OS transmit buffer is full.

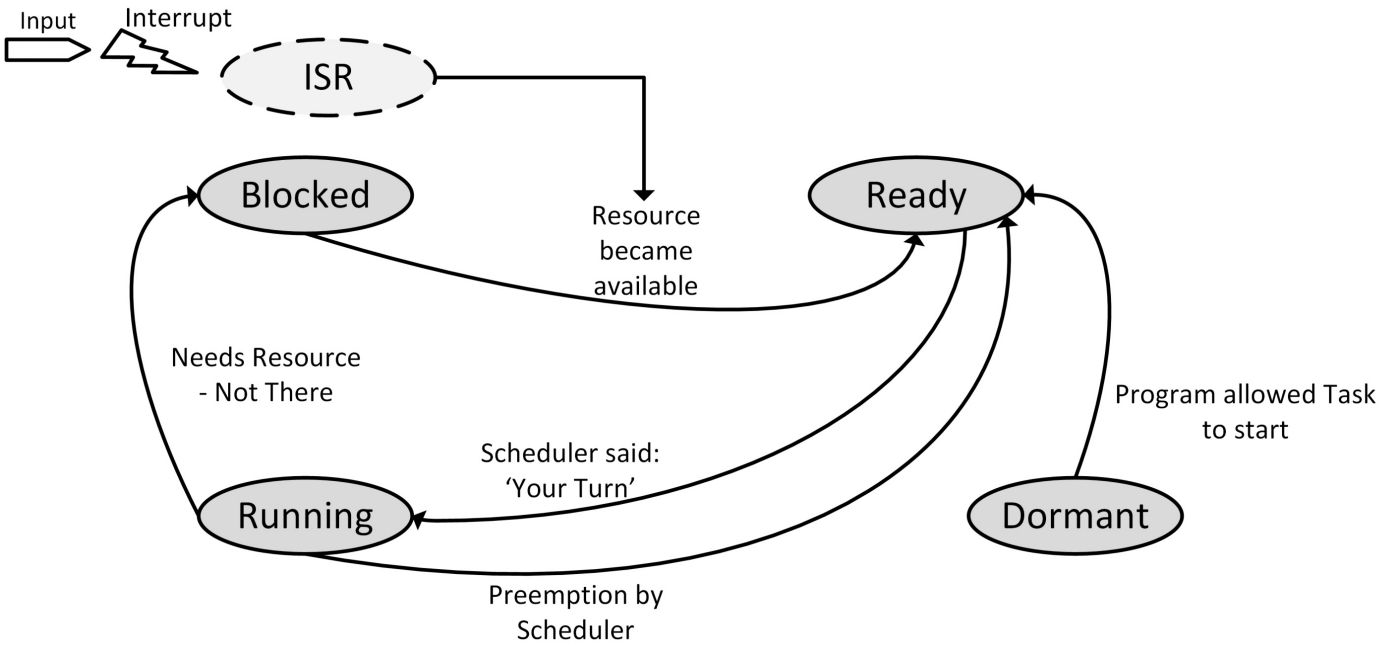



**Figure 2.4:** OS with Preemptive Scheduling

Most kernels today support **preemption**. This means that application code in tasks is not just interrupted by ISR's; when a task has run for an allowed time - the so-called timeslice - the **scheduler** may stop it "in mid-air" and instead start another task. As no-one knows which registers the current or the next task needs, all registers must be saved on a stack per task. This is the **"context switch"**. This is different from an interrupt-routine where you only need to save the registers used by the routine itself. A context switch can even occur before the time-slice is used. If somehow a higher prioritized task has become "ready", the scheduler may move the currently "running" low priority task to "ready" (thus not executing anymore but still willing to do so). More advanced kernels support **Priority Inversion** . This is when a high priority task is currently "blocked" - waiting for a low priority task to do something that will unblock it. In this case the low priority tasks "inherits" the priority of the waiting task until this is unblocked.

Figure 2.5 shows our little system again - now full-blown with interrupts and tasks. In some ways the tasks are now simpler as each task is blocked until awakened by the OS due to something that happened in the ISR. The figure shows how the three ISR's respectively use an x, y or z data-structure, and that the three tasks each wait on one of these. There is not necessarily a 1:1 correspondence - all three tasks might e.g. have

waited on “y”. The nature of x,y and z is different from OS to OS. In Linux the awaiting tasks calls *wait_event_interruptible()* while the ISR calls *wake_up_interruptible()*. Linux uses “wait queues” so that several tasks may be awoken by the same event. The term “interruptible” used in the call does not refer to the external interrupt, but to the fact that the call may also unblock on a “signal” such as e.g. CTRL-C from the keyboard. If the call returns non-zero this is what happened.

Tasks can also communicate to each other with these low-level mechanisms as well as semaphores, but often a better way is to use messages. C-*structs* or similar can be mapped to such messages that are sent to a queue (sometimes referred to as a mailbox). A design where all tasks are waiting at their own specific queue, and are awakened when there is “mail” is highly recommended. A task may have other queues that it waits on in specific cases, or it may block waiting for data in or out, but it usually returns to the main queue for the next “job”. One specific advantage in using messages is that some kernels seamlessly extend this to work between CPU’s - a local network so to speak. Another advantage is that it allows you to debug on a higher level which we will see later. “Zero Message Queue” is a platform independent implementation that supports this.

A word of caution: Do not try to repair deadlocks by changing priorities on tasks. The simple rule on mutexes, semaphores, etc is that they must always be acquired in the same order by all tasks. If one task “takes A then B” and another “takes B then A” then one day the two tasks will have taken respectively A and B, both will block and never unblock. Even though the rule is simple it is not easy to abide to.

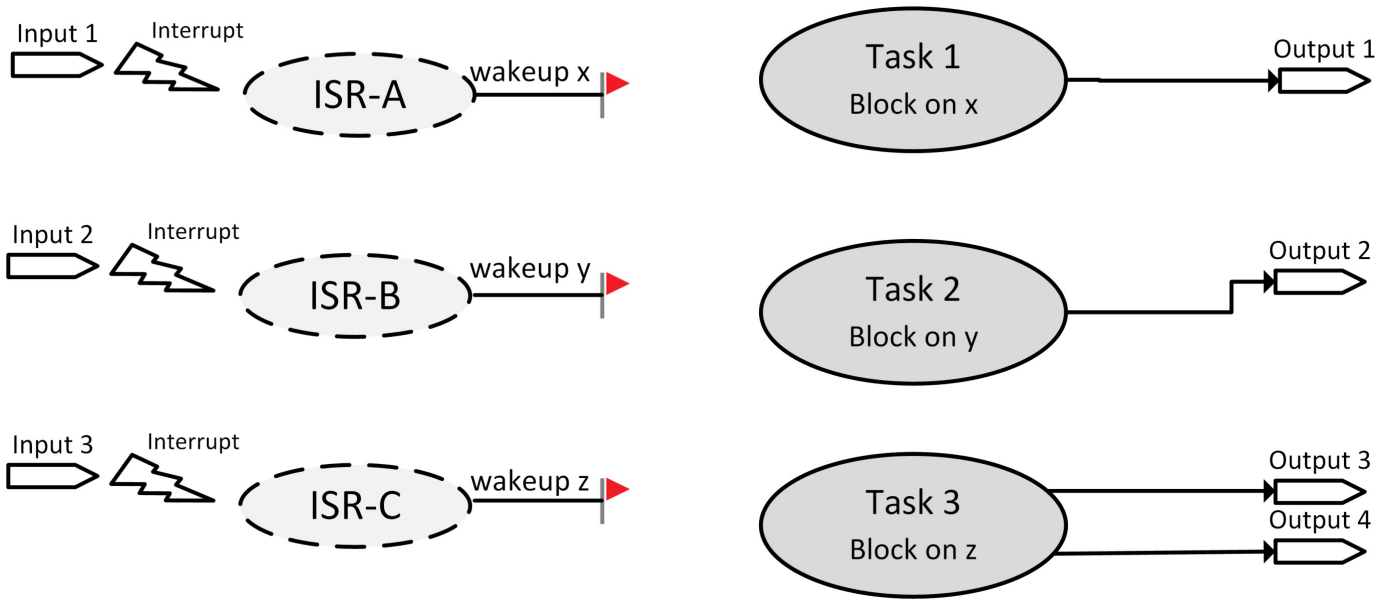



**Figure 2.5:** Tasks and interrupts

## 2.5 A non-preemptive Operating System

We typically talk about an “Operating System” when there is a user interface as well as a scheduler. So even though the OS in this way is “bigger” than a kernel it may have a less advanced scheduler. A very good example here is Windows 3. This was a huge and very complex operating system with a lot of new and exciting GUI-stuff and tools - and with Windows 3.1 we got “True-Types” which was a breakthrough in printing for most people.

However, from an RTOS (Real-Time-Operating-System) point-of-view Windows 3 was not so advanced. Windows 3 did support interrupts and it also had a task concept, but contrary to a small RTOS kernel it did not support preemption. So Windows 3 was pretty much as Figure 2.4 without the preemption action. Another thing that Windows 3 was missing - and which all good OS'es have today - is the support for the MMU. This was supported in the intel 80386 CPU which was the standard at the time, but the software hadn't caught up with the hardware. Anyway, an input could generate an interrupt as seen before, but even if this meant that a long-awaited resource was now ready, the scheduler could not move a low-prioritized task away from the CPU to make way for the now "Ready" high-priority task. Only when one task "yielded" could another task get to the CPU. This is not the same kind of "yield" as with co-routines. The Windows version is easy to implement in C. The way Windows 3 yields is that it performs specific OS calls - typically "*sleep()*". Sleep takes as input the minimum number of seconds or microseconds the process wants to be taken of the CPU - thus making time for other tasks. In Windows 3 code you would often see *sleep* called with "0" as parameter - meaning that the task *could* continue, but on the other hand was also prepared to leave the CPU at this point. Windows 3 also introduced a variant of Berkeley Sockets call WinSock . As we will see later - if you try to read data that hasn't arrived yet from a socket, your task will block in a preemptive system. In the days of Windows 3 this was standard in Unix, but Windows couldn't handle it - yet. So Microsoft invented WinSock where a socket could tell you that it "WOULDBLOCK" if only it could, so could you please write a kind of loop around it with a *sleep*, so that you do not continue before there is data or the socket is closed. It would not be surprising if this was the kind of behavior that made Linus Thorvalds start writing Linux. The lack of preemption support was also one of the chief reasons why Microsoft developed Windows NT - which in newer versions is known as Windows XP, Vista, 7, 8 or 10 - Windows to all "normal" people. All this is not just an anecdote - you may still see kernels for very small systems that are non-preemptive - like *mbed* in its simple version.

It is important that not only the OS or kernel supports preemption, but also that the C-library code is supporting this well. There are two overlapping terms that we need to consider here:

- **Re-entrant**
  If a function is re-entrant it means that it can be used recursively. In other words, by the same thread. In order for this to happen it must not use static data, and is therefore using the stack. A classic example of a non-reentrant C-function is "*strtok()*" which is very fast and efficient but keeps and modifies the original string.
- **Thread-Safe**
  If a function is thread-safe it means that it may be used from different threads of execution in parallel. This is accomplished with the use of locks or critical sections.

The two terms are sometimes mixed up. What you need as an embedded programmer is typically "the full Monty" - you need library functions to be thread-safe and re-entrant. Many kernels and operating systems supply two versions of their library - one for multi-tasking (thread-safe), and one not. The reason for having the latter at all is that is is smaller and executes faster. It makes sense in non-preemptive systems, or systems with no

OS at all, as we saw at the beginning of this chapter.

It should be noted that there are now modern non-blocking sockets. In order to create servers with tens of thousands of connections there are various solutions known as Asynchronous I/O, IO-Port-Completion and similar terms that create an FSM within the OS, and thus feature an event-driven programming model. The basic IOT-device will however not serve many clients directly. Typically there will only be one or two clients in the cloud - servicing the many clients. For this reason and because classic sockets are universally implemented we are focusing on the classic socket paradigm. In any case the underlying TCP is the same.

## 2.6 Full OS

The small kernel gives you all you need in order to multi-task and handle interrupts. Gradually kernels have added file-systems and TCP/IP-stacks to their repertoire. When a kernel comes with drivers for many different types of hardware as well as tools that the user may run on the prompt, and typically a Graphical User Interface (GUI) - then we have a full OS. Today Linux is the most well-known and used full OS in the embedded world. Windows also comes in versions that are targeting the embedded world, but somehow Microsoft never really has had their heart in it. Windows CE is dying out - there really are very few hardware vendors supporting it and the development environment - “Platform Builder” - can be very disappointing if you are used to Visual Studio. Microsoft has marketed first Windows XP, then Windows 8 in a “fragmented” version for the embedded world, and is now marketing Windows 10. However, the embedded world typically demands operating systems that are maintained longer than it takes for Microsoft to declare something a “legacy” - and Windows is difficult to shrink down to small systems. If you can use a standard industrial PC with standard Windows in an application, then by all means do it. You can take advantage of Visual Studio with C# and all its bells & whistles. This is a fantastic and very productive environment.

Neither Linux, nor Windows are what can be called real-time systems (except Windows CE). There are many definitions of the term “real-time”, but the most commonly used is that there must be a deterministic interrupt latency. In other words, you need to know the worst-case time it takes from something in the hardware changes state until the relevant ISR is executing its first instruction. Both Linux and Windows are designed for high throughput - not for deterministic interrupt latency. An example of a true Real-Time OS (RTOS) is VxWorks from WindRiver.

So if it’s not real-time - how come Linux is so popular in the embedded world? First and foremost, it is popular for its availability of drivers and libraries for almost anything. Your productivity as an embedded developer is extremely much higher when you can draw on this massive availability. Secondly there’s the community. Should you get stuck, there are many places to ask for help - in most cases you only have to browse a little to find a similar question with a good answer. Finally, we have the Open Source which we will discuss separately. The fact is that generally high throughput is actually very nice, and in reality there are typically not many hard real-time demands in a system. Reading the A/D-converter sample before the next sample overwrites it is one such example. There are several solutions to this problem:

- Apply a real-time patch to Linux. In this way Linux becomes a real-time system - but there is no such thing as a free lunch. In this case the cost is that some standard drivers are not working any more. As this is one of the main reasons for choosing Linux it can be a high price.
- Add external hardware to handle the few hard real-time cases. This could e.g. be an FPGA that collects 100 samples from the A/D. Theoretically Linux still might not make it - but in reality it's not a problem with the right CPU.
- Add internal hardware. Today we see ARM CPU's that contain two CPU's - one with a lot of horsepower, perfect for Linux, and another one that is small and well suited for handling interrupts. As it does nothing else it can work without an OS - or with a very simple kernel. This CPU shares some memory space with the bigger CPU, and can thus place data in buffers - ready for the bigger brother. In the simple example with an A/D-converter many CPU's are however capable of buffering data directly from an $I^2S$ bus or similar.

Another problem that Linux has, is that it demands a Memory Management Unit (MMU). This is in fact a very nice component in the larger CPU's that cooperates with the OS, in such a way that it guarantees that one task cannot in any way mess up another task - or even read its data. Actually tasks in such a system are often called *Processes*. A process is protected from other processes by the MMU, but this also means that there is no simple sharing of memory. When this is relevant a process may *spawn threads*. Threads in the same process-space can share memory and thus are very much like tasks in a smaller system without MMU. This is very relevant on a PC, and nice to have in an embedded system, but if you want a really small system it won't have an MMU. It is possible to compile Linux to work without the MMU - but again this inhibits a lot of drivers.

There is a similar lesson to learn from Ethernet. Ethernet is not perfect and it cannot guarantee a deterministic delay like Firewire can. Still Firewire is losing and Ethernet has survived since 1983 (at various speeds that is). The cheap "good-enough" solution wins over the expensive perfect solution **if** it can solve problems outside a small community.

## 2.7 Open Source, GNU licensing and Linux

When it comes to kernels and OS'es they come in two major flavors - Open Source or not. If you come from the Windows world you might wonder why so many embedded developers want open source. Surely a lot of people believe in the cause - the concept of not monopolizing knowledge. However - to many developers "open" literally means that you can open the lid and look inside.

Here are a couple of reasons why this is so important:

1. If you have a problem and you are debugging, you will eventually end in the kernel/OS. If you have the source, you may be able to find out what is wrong. Oftentimes you may be able to make a work-around in your own code. This would be pure guesswork in a "closed" environment.
2. As above - but there is no possible work-around, you need to change the OS. With open-source you can actually do this. You should definitely try to get your change

into the actual code-base, so that the next update contains your fix. There is nothing more frustrating than finding a bug - and then realizing that you have found it before. Also, the GNU Public License requires you to make the improvement public, so getting back into the official kernel makes life easier - which is the whole point.
3. As stated earlier, a lot of embedded code lives for many years. It's a pain if the OS doesn't, and if it is open source you can actually maintain it yourself.

If you come from the "small kernel" embedded world you are probably used to compiling and linking one big "bin" or "exe" or similar. This keeps you in total control of what the user has on his/her device. You may have noticed that embedded Linux systems look much like a PC in the way that there are tons of files in a similar-looking file-system. This is a consequence of the open source licensing concept. If you are a commercial vendor you charge for your system that includes a lot of open source besides your application. This is OK as long as the parts that originate from open source are redistributed "as is". This makes configuration control much more difficult, and you may want to create your own distribution - or "distro". Take a look at "Yocto".
GNU means "GNU is Not Unix". It was created in the US university environment as a reaction to some lawsuits on the use of Unix. The purpose is to spread the code without prohibiting commercial use, but also very focused on not being "fenced in" by commercial interests. The basic GNU license allows you to use all the open-source programs, but you cannot merge them into your source-code, and not even link to them without being affected by the "copy-left" rule that means that your program source must then also be public.

There are many sites claiming that you are allowed to dynamically link without being affected by the copy-left clause, however an FAQ on gnu.org raises and answers the question: *Does the GPL have different requirements for statically vs dynamically linked modules with a covered work? No. Linking a GPL covered work statically or dynamically with other modules is making a combined work based on the GPL covered work. Thus, the terms and conditions of the GNU General Public License cover the whole combination.*

This means that your code must call all this GPL code as executables. This is not a "workaround" but the intended way. This fact is probably responsible for keeping one of the really nice features from Unix: You can call programs on the command-line and you can call them from your program or script - it works the same way. Based on the Linux philosophy that states that programs should be "lean and mean" or in other words; do only one thing, but do it good, and the fact that most programs use files, or rather *stdin* and *stdout*, you really **can** benefit from the GPL programs this way. This is very different from Windows where command-line programs are rarely used from within applications - see Section 7.4.

But if you are not allowed to link to anything, then what about libraries? It will be impossible to create anything proprietary working with open-source. This is where the "Lesser GNU Public License" comes into the picture. The founders of GNU realized that it would inhibit the spread of the Open Source concept if this was not possible. All the system libraries are under this license that allows linking in any form, but states that if you **statically** link, then you must distribute your object file (not the source), so that other people are able to update when newer libraries become available. This makes dynamic

linking the preferred choice.

The GNU org are, however, very keen on not having too much code slip into the LGPL. There is even a concept called “sanitized headers”. This is typically headers for LGPL libraries that are shaved down and pre-approved by GNU for use in proprietary code. In order to use a library, you need the header-files, and the fact that someone even thinks that sanitizing these is needed shows how serious the GPL is. The main rule is to keep things completely separated - never start a proprietary code module based on open source. There are alternatives to the GPL such as FreeBSD license, that aim to make it easier to make a living on products based on their code. Such libraries may also be used from proprietary code.

Still, Linux adheres to GNU. There is a much debated case on LKM’s - “Loadable Kernel Modules”. As stated by the name, these are program-parts that are dynamically loaded into the kernel, and one vendor has made a proprietary LKM. I am not a lawyer but I fail to see how this cannot violate the GPL. The way I understand it this has been more ignored than really accepted by the GNU community.

## 2.8 OS constructs

The following is a short list and explanation of some of the OS constructs discussed.

| Concept | Basic Usage |
|---|---|
| atomic | A Linux macro that assures atomicity on variables that are not normally atomic - e.g. a variable in external memory. |
| critical section | Generally code that must only be accessed by one thread at a time. Typically protected by a mutex. Specifically, on Windows a critical section is a special, effective mutex for threads in the same process |
| event | Overloaded term. Kernel-wise Windows uses events that other threads/processes may wait on - blocking or not in e.g. “*WaitForMultipleObjects*” |
| semaphore | Can handle access to n instances of a resource at the same time. The semaphore is initialized to “n”. When a process or thread wishes to access the protected data, the semaphore is decremented, if it becomes 0 the next requesting process/thread is blocked. When the data is released, the semaphore is incremented |
| lock | A mutex can be said to implement a lock |
| mutex | Like a semaphore initialized to 1. However only the owner of the “lock” can “unlock”. The ownership facilitates priority inversion. |
| signal | A Unix/Linux asynch event like CTRL-C or “kill n”. A process can block |

| | |
|---|---|
| | until a signal is received, but it may also be “interrupted” in the current flow to run a “signal-handler”. Like interrupts, signals can be masked. |
| spinlock | A low-level mutex in Linux that does not sleep, and thus can be used inside the kernel. It is a busy-wait and thus effective for short waits. Used in Multiprocessor systems to avoid concurrent access. |
| queue | High-level construct for message passing. |

## 2.9 Further Reading

- Andrew S. Tanenbaum: Modern Operating Systems
  This is a kind of classic and very general in its description of Operating Systems. The latest edition is the 4’th.
- Jonathan Corbet and Alessandro Rubini: Linux Device Drivers
  This is a central book on Linux drivers, and if you understand this you understand everything about critical sections, mutexes etc. The latest edition is the 3’rd.
- lxr.linux.no.
  This is a good place to start browsing the Linux source code. There are also GIT archives at www.kernel.org, but lxr.linux.no is very easy to simply jump around in. It is good when you just want to learn the workings of linux, but is also good to have in a separate window when you are debugging.
- Mark Russinovich et al: Windows Internals Part 1 & 2
  To avoid it all being Linux these books by the fabulous developers of “Sysinternals” are included. This was originally a website with some fantastic tools that was - and still are - extremely helpful for a Windows developer. These guys knew more about Windows than Microsoft did - until they merged.
- Simon: An Embedded Software Primer
  This book includes a small kernel called uC, and is using this for samples on how to setup and call tasks, ISR’s etc. Includes description of some specific low-level HW circuits. The book is using Hungarian notation which can be practical in samples, but not recommended in daily use.
- C. Hallinan: Embedded Linux Primer
  This is a very thorough book on the Linux OS.

# Chapter 3
# Which CPU to use?

## 3.1 Overview

As stated in Chapter 2 the choice of CPU is very related to the choice of Operating System. As an embedded developer you may not always get to choose which CPU you will be working with - it may have been decided long ago in a previous project, or by someone else in the organization. Still, understanding the various parameters will help you get the best out of what you have, and also make it easier for you to drive the choice of the next CPU. It will also enhance your dialogue with the digital designers.

Historically the CPU was the basic chip that executed your code and everything else was outside this as other chips. Then came the microcontroller. This was specifically made for smaller embedded systems, containing a few peripherals such as timers, a little on-board RAM, interrupt-controller and sometimes also EPROM to store a program. As integration increased we ended with the modern SOC - System-On-a-Chip. In such chips what used to be called the CPU is now known as the "core" (but is typically much more powerful in itself than older CPU's) and the term "CPU" can mean anything from the core to the full SOC chip.

An example of such an SOC chip is the Texas Instruments AM335x - where the "x" refers to variations in clock cycles and on-board peripherals. This is also nicknamed "Sitara" and is shown in Figure 3.1. It is particularly interesting as it is used in the BeagleBone Black Board. This is a hobbyist/prototyping board that resembles the Raspberry Pi. The Beaglebone Black is referred to in this book in several contexts. Inside the AM335x is an ARM Cortex-A8 - and a **lot** of integrated peripherals. The core runs the actual program and dictates the instruction set, while the other components decides the overall functionality and interfaces. This chapter is dedicated to the full integrated chip concept which is referred to as the CPU and the term "core" is used for the part that executes the code.

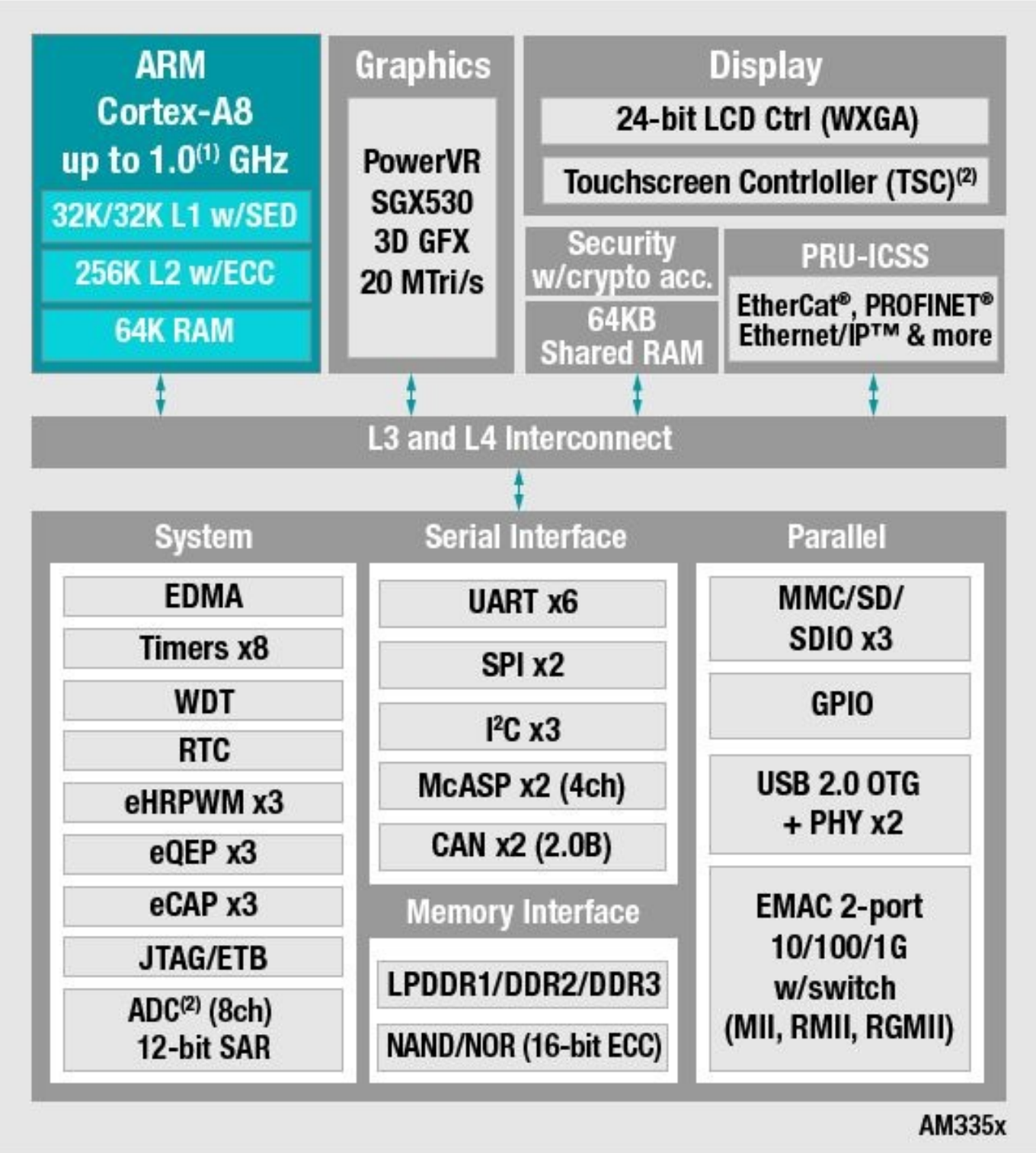
Modern computerized devices are thus created like "Russian Dolls" or "Chinese Boxes". A company creates a product that may be based on a computer-board from another vendor. The vendor of this board has used an SOC like the AM335x from TI, and TI has bought the IP (Intellectual Property) for the core from ARM. ARM delivers cores to many modern designs from many vendors - but there are also alternatives.

| Concept | Purpose (short) |
|---|---|
| CPU Core | Basic programmable unit |
| MMU-Support | Needed for High-End OS'es |
| DSP-Support | Signal Analysis |
| Power Consumption | Battery-powered, Heat-Generation |
| Peripheral Units | A/D, UART, MAC, USB, WiFi… |
| Basic architecture | For specific solutions |

| | |
|---|---|
| Built-in RAM | For speed and simplicity |
| Built-in cache | For speed |
| Built-in EEPROM or Flash | Field-Upgradeable |
| Temperature range | Environmental requirements |
| JTAG Debugger Support | HW Debug without ICE |
| OS'es supporting | See Chapter 2 |
| Upgrade Path | If you grow out of current needs |
| Second sources | If the vendor stops or raises price |
| Price in quantity | Basic but also complex |
| Evaluation Boards | Benchmark and/or Prototype |
| Tool-Chain | Compilers, Debuggers etc. |
| Vendor Support | When problems occur |

In the following we will look a little closer at each of the above subjects.

---




NOTES:
(1) >800MHz available on 15x15 package,13x13 supports up to 600MHz
(2) Use of TSC will limit available ADC channels
SED: Single error detection/parity

**Figure 3.1:** AM335c - Courtesy Texas Instruments

## 3.2 CPU Core

Not everything is about ARM, but they are very popular in modern embedded designs. ARM has created three different “Cortex” (for “Core Technology”) “profiles”:

- **A** - for Application
  This is the biggest, highest performing and most integrated designs with “bells and whistles”. These CPU’s can do hardware control, but are better suited for number-crunching as they include DSP and NEON media processing engines - see Figure 3.1. They integrate a lot of the peripherals that we will go into in this chapter. They also are very well suited for running Linux or other high-end OS’es. The Cortex-50 series actually supports 64-bit and is used in some high-end mobile phones.
- **R** - for Realtime
  These favor engine-control, robotics etc. where it is important that latency is low and security is high. It is also a good choice for network routers, media-players and similar devices that do not require the performance of the Cortex-A’s, but do need data here and now.
- **M** - for Microcontroller
  These are classic micro-controllers for handling external hardware. They are offered as soft cores for FPGA, but are also sold in numerous versions integrated with memory and peripherals. With no MMU included they cannot run standard Linux.

It’s probably not a coincidence that the three types spell “ARM” - but nevertheless they do reflect some major segments within the embedded world, that other microcontrollers and SOC’s fit into as well. It is however not given which is most relevant in an IOT design, as IOT is a very broad term covering anything from intelligent light-bulbs (“M” profile), over ATM’s (“R” profile) to advanced image-processing (“A” profile).

As we have discussed before; Linux is not really well suited for realtime but is a great help to get the most out of an advanced “Application Profile” CPU. In many ways it is even a waste to use such a device to control simple I/O. For these reasons some vendors are integrating e.g. both A- and M-profile CPU’s in the same chip. This may allow you to get “the best of both worlds”. The A-CPU can do advanced number crunching on Linux, and the M-CPU can focus on HW-interaction - and maybe do completely without an OS as discussed in sections 2.1 and 2.3. The integration thus continues - now towards multiple cores and or DSP’s.

## 3.3 CPU Architecture

The best known CPU-architecture is “von Neumann”. Here, program and data are accessed by the same data-bus and addressed via the same address-bus. A major advantage here is that the CPU has few pins and is easy to “design in”. Another advantage is that it is (or rather may be) possible to update the program in the field. If, however, you are very security-aware it may be preferable NOT to be able to change the instructions that the device executes. The alternative to von Neumann is “Harvard” - where data and program has separate buses. This is particularly popular with DSP’s. A Digital Signal Processor executes a lot of “Multiply-Accumulate” instructions, and typically (as we will see in Chapter 6) the Multiply operands are one of each:

- A constant - from the program.
- A data-value - e.g. from an A/D-converter or previous calculations.

Thus the Harvard architecture allows the DSP and/or Core to fetch constants and data at the same time. In the ARM series the v6 family is Von Neumann - used in e.g. Cortex M0. The v7 family is Harvard and is used in the "Application Profile" CPU's that mostly also has DSP-extensions. The ARM Cortex-A8 was the first to utilize the v7 family (yes - the names and numbers are not that easy to follow).
Another parameter on micro-processor architecture is whether it is CISC or RISC based. CISC - Complex Instruction Set Computing is "the old way". In this case a single assembly instruction may perform a complex operation - but it takes a lot of cycles. In a Reduced Instruction Set Computer each assembly instruction is simpler and executes much faster - but the same complex operation will demand more instructions, making assembly coding harder. However, little code today is written in assembly language. C and higher-level languages are totally dominating, so the important thing is "how fast does my C program run". And typically RISC wins here. The main reason we still have a lot of CISC-processors around is the fact that intel 80x86 CPU's are used in almost all modern desktop computers - no-matter whether they run Windows, Mac OS-X or Linux.

An important architectural parameter for a CPU is the degree of "pipelining"'[1] . The execution of an instruction has several stages - e.g. fetch the instruction, decode it, fetch the operands, execute the instruction and store result. To utilize the buses efficiently the CPU may have several pipeline stages that handles the instructions in an overlapping fashion. This has inherent "hazards" as one instruction may change an operand that the next instruction has already fetched - with the old value. This is yet a reason for using a compiler.

CPU's also come as either big-endian or little-endian. If the ASCII-string "Hello" starts at address 0x100 this is where the 'H' is (assuming C-style) and at address 0x101 we have 'e' then 'l' etc. Similarly, a byte array is also laid out sequentially. However, the bytes in the 32-bit dataword 0x12345678 can be laid out in two major ways (see Figure 3.2):

1. **Big-Endian**
   0x12 is placed at address 0x100, then 0x34 at address 0x101 etc. An argument for this is that when you see the data in a debugger as bytes they are placed in the same order as when you tell the debugger that you are looking at 32-bit words. Motorola was on this side in the "endian-wars".
2. **Little-Endian**
   0x78 is placed at address 0x100, then 0x56 at address 0x101 etc. An argument for this is that most significant byte is at the highest address. Intel was on this side with most of their CPU's - including 80x86 - although 8051 actually is big-endian.

Big-endian is also defined as "Network Byte Order" - meaning that this should be used for network data as this is often interchanged between platforms. This we will see when setting up port-numbers (16-bit) and IPv4-addresses (32-bit), but it should also be used for the actual application data, though this rule is not followed by all protocols. If you are sending application data between two PC's - which are Little Endian - there is quite a performance overhead in changing to network order and back again. At least the "endianness" must be documented.

Many modern CPU's can be configured to one or the other.

## 3.4 Word-Size

The word-size of a CPU is defined as the width of its internal databus. The address bus may be the same size - or wider. There are also examples of variants with different internal and external bus sizes. The most well-known is the 16-bit 8086 that had an external 16-bit databus, but later came in an 8-bit external databus variant called 8088 - made immortal by the IBM PC XT.

In the embedded world 64-bit CPU's has not really taken off. The question is whether to use 8, 16 or 32 bits. You may be surprised that 8 bits are still relevant. The site "embedded.com" has had some good discussions on this. Bernard Cole in January 06, 2014 takes up the thread from an earlier article by Jack Ganssle and argues that the Intel 8051 in the world of IOT can be a big threat to modern ARM CPU's.

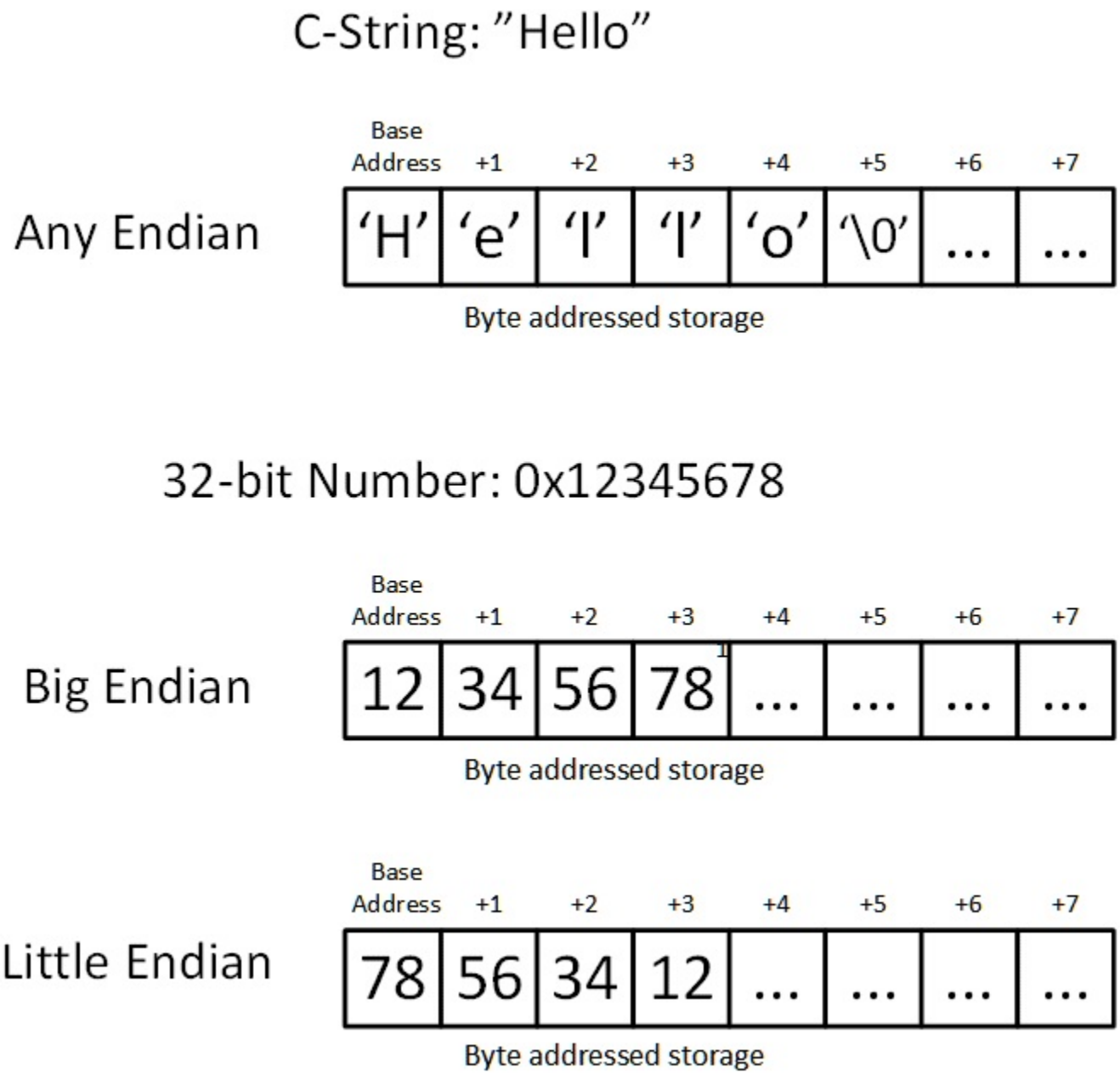



**Figure 3.2:** Endian-ness

The basic advantage of 8-bit controllers is that they are very cheap. The underlying patents have run out, and vendors can now make derivatives of the 8051 without paying license fees to Intel - and there are definitely still many 8051-based designs on the market. Universities and others have ported minimal TCP/IP stacks - even with IPv6 support - to 8051. HW-vendors are still upgrading these designs with more built-in peripherals as well as constantly speeding them up. This happens by executing instructions on fewer cycles as well as running at higher clock-speeds. Finally, there is a large base of developers that are very confident on and efficient with this architecture.

So if your design is extremely price-sensitive the 8-bit CPU's may be interesting. If, on the other hand, you want a highly integrated and high-performing modern System-On-Chip (SOC) solution the 32-bit CPU's are definitely the place to look. It is a fact however, that only recently have 32-bit CPU's overtaken the 8-bit CPU's as the ones most sold.

That leaves the 16-bit CPU's in a kind of limbo and if you look at internet discussions it certainly does seem like 16-bit designs are not really worth discussing.

*I am not sure that 16-bit CPU's can be written off. Some Vendors such as Texas Instruments, Frescale/NXP and Microchip Technology supply to many designs, whereas you don't hear much about designs from Renesas. Nevertheless, this is one of the major vendors and their latest family of micro-controllers is 16-bit based. The explanation may be that Renesas is targeting the automotive industry where the number of units is high, but the number of integrators is relatively low - causing less hype on the internet.*

## 3.5 MMU - Memory Managed Unit

As described in Chapter 2 the MMU is a hardware-entity in the CPU that isolates the OS-processes completely from each other. With this it is impossible for one process to overwrite data in another process - it cannot even read it. This also has some overhead. If e.g. one process asks for a socket-buffer that it will put data into, this buffer must be cleared before it is handed to the process, as it might have been used by another process earlier on - and the new process may not see these data. Apart from these IT-security concerns an MMU is your friend in catching bugs. A common and sometimes very difficult problem is "Memory-Overwrite". This comes in many flavors - some simply can write anywhere in your memory - typically due to a "wild" or non-initialized C pointer. If this happens in a system with an MMU, it will typically get caught. You will get an *exception* from the MMU. So make sure to catch this - at least on the top-level of your code. If you can "dump" the stack in the exception-handler you may be able to see the culprit.

## 3.6 RAM

No real program can run without RAM. Naturally, if you can fit it inside the CPU, you don't need anything outside the CPU. Also, accessing internal RAM is faster than accessing external RAM. A desktop computer may have Gigabytes of Dynamic RAM (DRAM). This cannot possibly fit into the CPU. The advantage of Dynamic RAM is that it takes less space than Static RAM (SRAM). It does however need a "DRAM-controller" that constantly "visits" RAM-cells to keep contents fresh. This is often built into the CPU.

SRAM is typically faster and more expensive than DRAM and does not need refreshing. Modern DRAM is known as SDRAM which is very confusing as we now have both the "S" and the "D" so which is it? It turns out that the "S" here is for "Synchronous". It is still Dynamic and the latest ones are called DDR with a generation number - DDR, DDR2 and DDR3. DDR means "Double Data Rate" which specifies that data is fetched on both the up- and down-going flank of the clock.

Micro-controllers like Intel 8051 and the newer Atmel Atmega128A has internal SRAM. In some scenarios this may be all that is needed. In other scenarios you may - in the Atmel case - utilize the fact that various part of the chip may be put into sleep-mode while the internal SRAM still works. You keep the system-stack and interrupt-vectors here. It may even be possible to turn external RAM completely off in sleep-mode, but this requires that your design is able to start almost from scratch.

## 3.7 Cache

Over the last many years, CPU-performance has been improving more than memory-performance. As described earlier it is therefore common to have the hardware insert "wait-states" when fetching data from external memory. To handle this growing gap "**cache**" was invented. This is an intermediate memory - inside the CPU - that is a "shadow" of the latest blocks of data fetched from the external memory. In a Von Neumann system (see Section 3.3) you need a single cache - in a Harvard system a "split-cache" must be used if both program and data is to be cached. If - in a system without cache - your code is executing a fast "inner-loop", it is spending a lot of clock cycles waiting for the same parts of the program being fetched from external memory - with wait-states - again and again. If instead you have a cache and this is big enough to hold all the code in the inner-loop, then you are executing code much faster. Cache may come in several layers, and can be complex to set up. A common problem is that most modern CPU's are using "memory-mapped I/O". When you read from this you may read new data at every read - unless it is cached, in which case the cache will keep supplying the same first value. This is the reason why memory-mapped I/O must be declared non-cache-able on a low level. You might think that the C keyword "*volatile*" will help you - but it wont. Many compilers give you the option to prioritize either speed or size - in the sense that the compiler can either maximize the speed of the program or minimize the number of instructions. In a system with cache it is recommended to prioritize size. This is due to the fact that the fewer bytes your program takes up, the bigger is the chance that important loops fit in the cache - and this will boost speed much more than what any optimizing compiler can do. Not all modern CPU's have cache. This is a feature that is typically only included in the larger CPU's. E.g. the ARM Cortex M0, M1, M2, M3 and M4 do not have internal cache.

## 3.8 EEPROM and Flash

If the CPU has built-in Flash it can store its own program here. Unlike built-in RAM you normally don't see CPU's that have both internal **and** external flash. It's either one or the other. EEPROM (Electrical Erasable Programmable Memory) could also be used for

storing the program, but typically there is not room for a program here. Instead EEPROM is mostly used for setup-data. This could be user-changeable data as an IP-address, production data like a MAC-address, or even something like “Number of hours in service”. The latter can be a challenge if the EEPROM only allows a limited amount of writes.

## 3.9 FPU - Floating Point Unit

There are many degrees of math assistance. E.g. many newer derivatives of the old Intel 8051 have added special multipliers to assist the small 8-bit core with 16-bit integer multiplications. Some has even added true IEEE 754 Floating-Point units with single precision (8 bit exponent and 24 bit mantissa) or double precision (11 bit exponent and 53 bit mantissa), in C respectively known as *float* and *double*. When comparing FPU’s you should also take note of how many cycles they use to do the various multiplications and divisions. To compare accurately is very difficult as the various CPU structures don’t fetch their variables in the same way and at the same speed. This is where the number of MFLOPS - Million Floating Point Operations Per Second - becomes interesting.

## 3.10 DSP

If you plan on doing Digital Signal Processing you should check the number of MAC’s - Multiply and Accumulate - operations you can perform in the relevant accuracy (Integer, Single or Double). As discussed in Chapter 6, a Multiply-Accumulate with an output-shifter is really important for filters implemented with integer arithmetic. If an FFT is to be implemented, so-called “Bit Reversed Addressing” will also save a lot of cycles. Some of the high-performance CPU’s include SIMD extensions. SIMD is Single-Instruction-Multiple-Data. This means that exactly the same instruction can be performed on many elements of an array at the same time. The ARM NEON e.g. has 32 64-bit wide registers that can be seen as an array. This is particularly relevant in image-processing etc. Note however that compilers do not support this directly.

## 3.11 Upgrade Path

Most CPU-cores are part of a family that has many levels of performance, on-board peripherals and temperature variants. ARM is a special case as the core CPU blueprint is leased to many chip-vendors that build on top of this. In this way the number of ARM derivatives is huge compared to any other embedded CPU. On the other hand, there may be big differences between the various vendors versions. An upgrade path is really important as systems tend to grow and grow. If the bigger & better device is even plug-in compatible with the old, then your day is made. Different manufacturers tend to grow in different directions. Whereas one keeps pushing performance upwards, another is pushing down the energy used per instruction (and introduces better sleep modes).

## 3.12 Second Sources

It is really nice to know that if your manufacturer of CPU’s decides to stop production or

raises prices there is at least one other manufacturer. This is however not always possible. At least set up a process so that you receive “Last-Time-Buy”. This is similar to the pub calling “last round”. You need to be there fast and stock what you need until you have a work-around. The more special your CPU is the smaller is the chance of second sourcing.

## 3.13 Price

This sounds simple, but in reality is a complex area. Most components such as capacitors, resistances and op-amps are more or less clones of each other and you can buy whichever is cheapest after your design is complete. However, when it comes to the larger semiconductor chips as Micro-controllers, Micro-processors, FPGA’s, A/D-converters and voltage-regulators there is a lot of bargaining.

It is not a good idea to start a design by choosing components, and then when you are ready to go into production start negotiating prices. The vendor already knows that you have based a product on this chip, and your basis for negotiation is thus lousy. It is more optimal to negotiate the prices up-front. This is a very annoying way to work for many designers, as they typically want to pick & choose themselves during the project. It is also difficult for professional purchasers as they typically don’t have the full overview that the designers have. It also means a limitation on how much you really can discuss with your dealer before you make your choice. Obviously all this is only relevant if you are producing in relatively large quanta. If you only create small series, the development costs are so much higher than production costs, that the latter is really not that interesting.

## 3.14 Export Control

There is a number of complicated rules about what non-US companies are allowed to export to which countries - and how to register this. Some rules apply to the value of US goods in your product, stating e.g. that max 25% (10% for selected countries like North Korea and Syria) may be of US origin. This “De Minimis” is calculated as your total costs for US products compared to your sales price.

Other rules are more technical - e.g. stating that if a system is capable of sampling more than 100 Msamples/second for a given system (adding all channels) the end-user must be registered. There are also rules about encryption levels.

Finally there are rules about “Dual-Use” - which is technology that can be used in both military and non-military scenarios - versus technology developed specifically for military purposes. I am no lawyer and as the penalties for violations are quite severe - we are talking imprisonment here - you should not take my word but enlist some competent people on these matters.

Anyway - if you are a non-US company and you are buying expensive chips, such as some CPU’s and FPGA’s, it may be advantageous to buy some non-US parts to stay on the safe side in De Minimis calculations. This is also valid for licensed software.

## 3.15 Evaluation Boards

Also known as EVM's - Evaluation Modules. Especially when trying out a new CPU these are extremely valuable. MIPS - Million Instructions Per Second - may have been calculated under circumstances different from your application. The best chance to get a good guess on the actual performance is to implement it - or at least the "fast path". The major vendors typically sell very cheap EVM's so that it is indeed doable to check some essential code on 2-3 platforms - if you know from start what is "essential". However, EVM's may be used further on in the development process. It is sometimes possible to create a lot of embedded software for the final target - using EVM's. Cheap hobby devices such as Arduino, Raspberry Pi and Beaglebone Black may serve the same purpose. See also Chapter 8. The functionality and form-factor of the Beaglebone Black and the Raspberry Pi is almost identical - see Figure 3.3. The Beaglebone Black is slightly more practical than the Raspberry for prototyping as it has two rows of connectors on which it is easy to fixate a "cape". You can buy all sorts of capes equipped with e.g. a display, relays, batteries, motor-control and much more.

**Figure 3.3:** Beaglebone Black (left) and Raspberry Pi

## 3.16 Tool-chain

It doesn't help to have the best CPU in the world if the tools are bad. Compiler, Linker, Debuggers and Editors can be more or less productive, and preferably should suit the development teams work-flow. If e.g. the team is used to working with Integrated Development Environments (IDE's) and is forced to use the "vi" editor with a command-line debugger, you should expect some very unhappy faces. Typically, these tools are related to the OS chosen, as well as the CPU. The OS may demand specific file-formats and specific setup of MMU etc, while the CPU may dictate the location of interrupt-vectors, Special-Function-Registers, handling of internal versus external RAM etc.

## 3.17 Benchmarking

It is clear that CPU's are implemented in many different ways. How do we compare their performance in e.g. number-crunching or network ability? The use of Evaluation Boards is suggested, but this is only practical when you have zoomed in on a few options. Before you get to this point it is relevant to look at benchmarks. These are numbers supplied by vendors as well as independent sources. The most basic is MIPS - Million Instructions per Second. It does not say much as there are great differences in what an instruction may accomplish. However, sometimes a given function in a software library may be specified by the vendor to require a specific amount of MIPS on a specific CPU type, and in this case the MIPS are interesting.

A little more relevant is MFLOPS - Million Floating Point Operations Per Second - which must be qualified with the precision used. In scientific applications the "Linpack" benchmark is often used for this.

The site eembc.org performs many benchmarks test on full systems - but also on CPUs. It is interesting because a lot of these data are available without membership. The general score on CPU's is called "CoreMark", while a benchmark on networking is called "Netmark".

Another often used benchmark is DMIPS - Dhrystone MIPS - which is a test that runs a specific program that performs a standard mix of classic operations - not including floating point. These tests are sometimes updated so as to "fool" the clever compilers that may optimize main parts away. DMIPS are practical when you want to compare different CPU-architectures.

Many Benchmarks are given per MHz as the CPU's often come in many speed-variants.

## 3.18 Power-Consumption

In some applications power is plenty and is not a problem. In other situations, the device must work on batteries for a long time and/or must not emit much heat. Comparing CPU's from data sheets can be extremely difficult. A modern CPU can tune its voltage as well as its clock-frequency and it can put various peripherals into more or less advanced sleep-modes.

The best way to compare, is to buy Evaluation Modules of the most likely candidates and run benchmark tests that resemble your application. It is easy to measure the power-consumption of an EVM-board by inserting a Watt-Meter in the supply-line. Alternatively insert a small serial resistance and measure the voltage drop over this. From this you can use Ohms law to calculate the current (assuming that voltage and current are in phase), and by multiplying this with the supply voltage you have a good estimate of the Watts consumed. The EVM may have a lot of hardware that you do not need, hence you don't want this to be a part of your calculations. You should document your measurements in e.g. excel and calculate the deltas between running with and without this or that feature or algorithm. As stated, modern CPU's can speed up and down dynamically. Sometimes this is fine - at other times you will want to use a "policy" that fixates the CPU speed and voltage in your tests.

## 3.19 Vendor Support

Most problems are solved in the development team - many by searching the web. Nevertheless, you may get really stuck and need support from a vendor. This may be local sales-support or specialists from the head quarter. If you are used to getting good support from your vendor you may take this for granted - but it isn't. Having a good relation with a vendor is a great help - but on the other hand it may also inhibit your price negotiations - see Section 3.16.

## 3.20 JTAG Debugger

Many Integrated Circuits have a JTAG-interface for testing the IC mounted in the PCB during production. This interface can also be used for simple debugging as single-stepping and inspecting and changing variables. This is built into almost all modern CPU's. Some CPU's also has trace facilities.

## 3.21 Peripherals

Surely, if you need peripherals that are not there, you will need to add them externally. This will affect price, power-consumption, board-space and development time. On the other hand - the more special peripherals the manufacturer has built into the CPU - the more you are locked-in with this manufacturer, and the less tools will fit. So when you look at these highly integrated devices you should take an extra look at the compiler and libraries: Are there language-extensions that support your needs? As discussed earlier you can find modern designs with a 8051 nucleus on steroids - among these a floating point processor. But does the C-compiler help you take advantage of this automatically every time you perform a floating point operation? - or do you have to remember to use a special macro? This would require discipline that few teams have, and you would not be able to leverage old code without a lot of "global substitutes" (followed by a lot of testing). Some peripherals even require the support of the OS.

Building blocks like DSP and Floating Point Unit are listed in other sections. You could very well argue that they are peripherals. On the other hand, Cache and MMU are not peripherals, as they are tightly integrated with the functionality of the core CPU. In between these "camps" are e.g. RAM. Often this has very specific usages in the system and cannot be regarded as a peripheral. At other times it performs completely like external RAM - it is simply convenient to have inside the chip, and in that case it might be listed as a peripheral.

Anyway - here is a list of "peripherals not mentioned until now":

- **Interrupt Controller**
  In the very first computer systems this was an external chip. This is the device that maps and prioritizes incoming interrupts and is programmed to e.g. allow nesting of interrupts with higher priority.
- **DMA**
  Direct Memory Access is a way to help the CPU move large sets of data more

effectively. It may e.g. be able to move data from external memory to a disc in *burst-mode*, where it completely takes over the bus. Other modes are *cycle-stealing* in which case the DMA and the CPU must negotiate who has the bus, and *transparent* where the DMA only uses the bus when the CPU does not.

- **MAC - Media Access Control**
  This block implements layer 2 of the internet protocol stack - aka the Data-Link Layer - for Ethernet. It typically comes as Fast Ethernet (100 Mbit + 10 Mbps) or Giga (1000 Mbps + 100 Mbps +10 Mbps). Externally you still need the PHY that implements Layer 1 - the physical layer - the magnetics and the connector. Note that if you can live with only 10 Mbps and the accompanying higher latency, there is power to save.
- **Switch**
  Old-style coax-Ethernet allowed devices to be connected as pearls on a string. Modern network devices are however connected like the spokes on a wagon-wheel, with a switch at the center. This has numerous advantages like robustness and full bandwidth in both directions on the “spoke”. The main drawback is the extra cabling. There are times where network-devices are laid out on a line and bandwidth requirements are small and the old-style coax-solution would be very nice. In such scenarios you could fix the problem with a small 3-port switch at each device. One port is connected to the device, one is upstream and the last is downstream. This 3-port switch is built into many modern SOC’s. Don’t forget that the traffic load grows downstream and that you should specify a maximum number of devices that you can guarantee in your application.
- **A/D-converters**
  The best Analog/Digital converters are not built into microcontrollers, but sometimes you don’t need the best. If you can live with 10-12 bits resolution and a relatively low sample-rate and possibly extra jitter, this may be the perfect solution. You may also find a D/A - Digital to Analog - converter.
- **UART - Universal Asynchronous Receive/Transmit**
  This used to be a very important part, as it is needed for an RS-232 connection. RS-232 was the preferred physical connection for many smaller devices for many years. It was sometimes a disappointment for customers as they thought that as long as there is an RS-232, then “we can connect”. However, it is only the physical layer, and without a protocol, it is worthless. For simple monitoring purposes you may get lucky with a simple cable that connects Rx on the device to Tx on a PC, as well as the opposite and GND (typically pins 2,3 and 7). On the PC you might try “9600,N,8,1” - which means 9600 Baud, No handshake, 8 bits and 1 stopbit. If this does not work, try 115 kBaud which in many years was the fastest possible on PC’s. Today UART’s are still used in development. There may be no external RS-232 connector on “the box”, but many PCB’s (Printed Circuit Board) have a small connector for a serial connection used for logging etc. during debugging. A good example is the BeagleBone Black, where you can order the cable as an accessory. Another area for UARTS are IrDa (infrared) connections and similar.
- **USB-controller**
  USB took over after RS-232 as the preferred desktop connection to PC’s. With the experience from RS-232, the inventors of USB did not stop with the physical layer.

They also introduced a number of standard *Device Classes*, so that simple usage is "plug'n play". It is possible to "tunnel" TCP/IP through USB - something that many smaller devices support. In the USB-world there is normally a master (e.g. a PC) and a slave. But advanced devices may be slave to a PC one day, and then master to a set of headphones the next day. This is solved with USB On-The-Go - OTG - which allows for both roles (not at the same time).

The new USB-C standard is very promising. It may replace a lot of existing short cables.

- **CAN - Controller Area Network**
  This is a small, cheap and very robust bus that was designed by Bosch to be used in cars. Today it is also used on the factory-floor, for sensors and a lot of other devices. See Section 7.5.
- **WiFi**
  Some chips have the wireless equivalent of the MAC built in, but the antenna is obviously outside. Sometimes the antenna may be laid out directly on the PCB. This book has a full chapter on Wireless Networks - see Chapter 5.
- **Bluetooth or Bluetooth Low Energy (BLE)**
  The original Bluetooth is giving way in IOT for BLE. This does not allow for high bit rates, but is extremely flexible and can be used for pairing WiFi-devices and transfer of limited information at low rates - see Section 5.9. Version 5 of Bluetooth is announced at the end of 2016, and it would not be a surprise if we see something with the datarates of Bluetooth Classic and the connectivity of Bluetooth Low Energy (Smart). See Section 5.9.
- **Buses**
  Standard buses for interfacing to external devices. Typically *SPI* and $I^2C$. Many CPU's have intelligent buffering of data on these buses - saving the core from many interrupts. Even more basic is GPIO (General Purpose I/O) which is single pins that may be controlled from software.
- **PRU - Programmable Realtime Unit**
  Texas Instruments are using PRU's - Programmable Realtime Units - in some of their ARM derivatives - including the AM335x family. PRU's have a very limited instruction set and can in no way compete with a DSP, but they can move, add and subtract data with short notice without disturbing the main core and its OS, and they have good access to interrupts, GPIO-pins and memory. This is very convenient as the main cores and the bigger OS'es like Linux are not good at handling low-latency realtime-requirements. Programming the PRU requires a "PRU-assembler" - thus an additional build-step.
- **McAsp - Multi-Channel Audio Serial Port**
  This is another Texas Instruments specialty, that can be used to chain chips together.
- **RTC - Real-Time-Clock**
  This is normally a chip with a small power-source - typically a coin-cell - that is able to keep calendar time when power is off. In the IOT-case your device may be connected more or less always and can use NTP (Network Timing Protocol) to maintain calendar time, but knowing correct time from boot will help you to get better logs. There may also be scenarios where internet is not available all the time,

and if data needs to be correctly time-stamped an RTC is hard to live without. Many license-schemes also depend on knowing not just the time-of-day, but also day and year.

- **Timers**
  A HW-timer is hard to do without and most microcontrollers have several. One timer is used by the operating system, and one is typically a dedicated watchdog timer. A watchdog must be reset (“kicked”) by the software before it has counted down to zero. If this is not done it will reset the CPU - assuming that something “bad” has happened - e.g. an infinite loop or a deadlock. Fast timers are used for measuring the length of pulses as well as for generating them.
- **Memory Controller**
  As previously stated - Dynamic RAM needs a DRAM-controller. This is often built into the chip. Similarly, Flash controllers are built into many newer CPU’s.
- **Chryptographic Unit**
  There are many clever ways to encrypt and decrypt data today. The algorithms for this is based on large polynomials and primes. If this math is implemented in normal software, it will be way to slow for most purposes. A Cryptographic unit may be implemented in hardware or some very low-level code stored within the device running in a special DSP.
- **Display Controller**
  LCD, Touchscreen and other technologies typically require special control.
- **HDMI Controller**
  HDMI is a protocol used by modern TV’s high quality video and audio. Video is an important part of most of ARM’s A-profile CPU’s.
- **Graphics Engine**
  This is an accelerator for OpenGL graphics.
- **LCD Controller**
  This is a more low-level controller for graphics and text on dedicated LCD hardware.
- **GNSS**
  GNSS means Global Navigation Satellite System. In daily life we talk about GPS - Global Positioning System - as this was the first system and has been the only one for years. Today there is also the Russian Glonass, EU has Galileo coming and the Chinese are on their way with Beidou. In the world of IOT it can be very interesting to know your position even though it is fixed, as this makes it easier to verify that you are indeed in contact with the right device. Naturally it gets much more interesting when things move. From position changes over time you can calculate speed and acceleration, but be aware that professional systems that provide these parameters often combine the GPS/GNSS data with accelerometer data, to get better results. Most people do not realize that the accurate positions are the result of accurate timing. You can get the time with a better precision than 50 ns. In external devices this typically comes in the form of a string on a serial bus, that is matched with a timing signal - typically 1 pulse-per-second. Incidentally - GPS needs to take both Einsteins theories of relativity into account to work correctly.

When looking at the built-in peripherals in a given CPU, it is important to understand that the full set is never available at the same time. The limiting factor is the number of

pins on the IC, and normally the internal functionality is multiplexed to these pins. This means that you will have to choose between A and B in a lot of sets. It is a good idea to ask the vendor for an overview of which features inhibits what.

## 3.22 Make versus Buy

The previous sections are all assuming that you are creating and building your own electronics design - possibly indirectly with the help of a design company. If you have very special needs or are selling in large quanta this is the only way to go. If - on the other hand - your primary business is software for e.g. mobile apps and cloud, then you may want a shorter route to the hardware that controls whatever sensors or actuators you are using.

It is easy to get started with a Raspberry Pi, a Beaglebone or an Arduino board. These are all fantastic - they are low cost, easy to buy, quickly delivered and there are tons of supporting hardware as well as software tools. Most of these are however **not** industrial strength in terms of general robustness and temperature range. If e.g. power to a Beaglebone is lost in a “bad moment” the device is “bricked”. That is simply not good enough for real life products.

Here are some industrial alternatives:

- **Libelium**
  Libelium is a modern Spanish company that sells hardware to system integrators. The embedded IOT-devices are called “WaspMote”. They connect to sensors and are wireless - using ZigBee - connected to “Meshlium” routers that connect to the internet and the cloud. Examples are numerous “smart” projects - parking, traffic, irrigation, radiation etc.
- **Arduino Industrial 101**
  This is not just an industrial temperature range and general robust version of an Arduino. The basic Arduino is great for I/O but does not have much processing power. This board has an Atheros AR9331 MIPS processor running a Linux variant called Linino - developed via the OpenWRT which is well known for its many WiFi router implementations.
- **Hilscher netIOT**
  Hilscher is an old German company, and is an interesting example of how IOT in many ways is uniting well-proven existing technologies with modern cloud technology. Hilscher has their core competence in creating ASIC’s for fieldbuses. This is e.g. EtherCAT and CANOpen (see Section 7.5.2) which is used for factory floor automation. Many solutions programmed “from the bottom” today could be solved faster and cheaper with the help of e.g. CANOpen.

  Hilschers product series “netIOT” allows integrators to connect their factory automation to the cloud via e.g. the MQTT protocol which is suitable for smaller systems. This means that you can design and implement systems without making any hardware. An example could be a car factory in Brazil - monitored from Germany.

## 3.23 Further Reading

The main source of information is the various vendors' websites.

- Derek Molloy: Exploring Beaglebone
  This is a great book on getting the most out of the BeagleBone Black running debian Linux. Whether you learn most Linux or CPU-Hardware depends on where you start from. Derek Molloy also has a website with information newer than the book.
- embedded.com
  This is an interesting site with articles from many of the "gurus" in the embedded world.
- eembc.org
  This site performs many benchmark test on systems - but also on CPUs. It is interesting because a lot of these data are available without membership. The general score on CPU's is called "CoreMark", while a benchmark on networking is called "Netmark".
- wikipedia.org
  As you probably know already; this is a fantastic site. University scholars are typically not allowed to use information from this site as it can be difficult to trace the origins of information. For practical purposes this is not a problem.
- www.bis.doc.gov/index.php/forms-documents/doc_view/1382-de-minimis-guidance
  This is a good starting point on export control.

# Part II
# IOT Technologies

# Chapter 4
# Networks

## 4.1 Introduction

*The good thing about standards is that there are so many to choose from*

This is funny because most people agree that using standards is great, but nevertheless new standards keep popping up all the time. This is especially true when it comes to protocols. However, most of these protocols are “Application Layer” protocols. The “Internet Protocol Stack” has proven victorious - and it is the core of IOT. Instead of going through a myriad of Application Protocols we are focusing on the Internet Protocol Stack - and especially TCP. Look at any IOT application protocol and you will find TCP just beneath it (on few occasions UDP). If TCP does not work seamlessly, then the application protocol won’t work either. Unfortunately a stupid designed application protocol **can** slow a system down - see Section 4.20. This chapter also introduces REST in Section 4.11, which is an important concept used in many new protocols.

## 4.2 Cerf & Kahn - internet as net of nets

Vinton Cerf and Robert Kahn are honored as the inventors of the internet - which goes back to the 1960’ies - long before the World-Wide-Web. The simple beauty of the concept is that they realized that instead of fighting over which local network was best, we should embrace them all and build a “virtual” net on top of these local networks - hence the term “inter-net”. The Internet Protocol Stack is shown in Figure 4.1. The term “Protocol Stack” means that a number of protocols are running on top of each other - the layering pattern (see Section 7.3). They are normally numbered 1-5 from the bottom. Thanks to Cerf & Kahn we have the “virtual” IP-addresses on layer 3. The application is layer 5, and devices that have applications talking together are called “hosts” - no matter whether they act as a client or server at the application layer. Typically, we have Ethernet at the lowest two layers with the physical 48-bit MAC-Address at the Data-Link Layer and the copper or fiber cable at the physical layer. The MAC-address is typically fixed for a given physical **interface** (not device) and is written as six bytes in hexadecimal, separated by “:”. This address is like a Social-Security Number and does not change even though the device - e.g. a laptop - is moved from home to work. Inside a “subnet” - effectively behind a router - the devices are communicating with each other using local-net addresses on Layer 2. In most cases this is the Ethernet MAC-addresses. Devices communicating on layer 2 are called “nodes”. A PC and a smart-phone are both nodes and hosts. A router is only a node as long as we don’t communicate directly to an embedded Webserver inside it or similar.

## 4.3 Life of a packet

In this section we will follow a packet from a web-browser to a web-server. The routing described would be no different if the communicating parties were a cloud server and an embedded device in the field. The term “packet” is not precise, but very common. As

shown in Figure 4.1 the various layers each have their own name for the packet, which grows on its way down through the stack as each layer adds a header (and in the Ethernet Layer also a tail). The correct general term is “PDU” - Protocol Data Unit.

When a host on a local-network needs to contact a host outside the network it will first send the packet to its “gateway” router on the interface that is in “this” subnet. In the case of Figure 4.2 this interface has IPv4-address 192.168.0.1 and MAC-address 00:18:e7:8b:ee:b6.

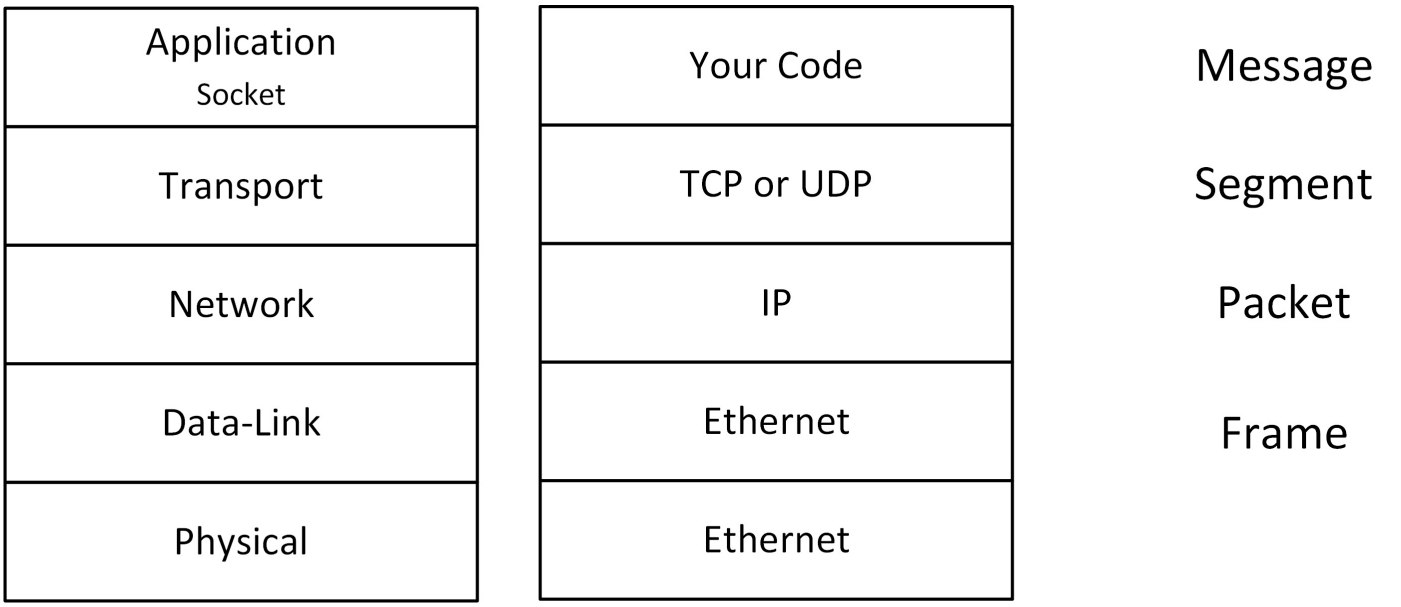



**Figure 4.1:** The generic layers, typical variant and the data unit names

Figure 4.2 shows the web-browsers request from a web-client to a web-server (from right to left). These are both termed hosts, as they “terminate” the path on the application level as well as on the transport level (TCP). This is shown as the dashed lines between the two applications and between the two transport layers. The addresses of the devices are noted at the top. In reality the packet follows the solid line and is “stored and forwarded” by the switch as well as by the router. The router “has its hands” on several things in the packet from the client:

- Client’s IP-address as Network-Layer Source
  This remains unchanged
- Web-Server’s IP-Address as Network-Layer Destination
  This remains unchanged.
- Client’s MAC-address as Link-Layer Source
  This is replaced by the Router with its own MAC-Address.
- Routers MAC-address as Link-Layer Destination
  This is replaced by the Router with the Web-Server’s MAC-Address (or the next Router’s MAC-address - whichever comes first).
- If IPv4 is used the so-called “hop-count” is decremented and when it reaches zero the packet is thrown away. This concept is in order to stop packets from circling forever.
- Changing the hop-count introduces the need to change the IPv4 checksum.

The bottom of Figure 4.2 shows the source and destination addresses in the packet as it

moves from left to right. The data is from a simple GET request. IP4 addresses are used for simplicity, and the 192.168.x.y address is a NAT-address. See Section 4.8.

The switch is completely transparent, changing nothing in the packet, and the only way you may recognize its presence is by the “store-and-forward” delay it introduces when it first waits for the whole packet to be “clocked in” - and then likewise transmits it out on another port. This is actually a little confusing. You might think that since the router - a layer 3 device - changes the address on layer 2 - the link-layer, so does the switch which **is** a layer 2 device - but it doesn’t. This transparency is what makes a switch a fantastic plug’n play device - contrary to a router which requires a lot of “setting up”.

When the Web-server responds to the request from the client it simply exchanges sources with destinations on both levels and the whole thing is repeated - now going from left to right. It actually switches one more source/destination pair: the TCP ports. These are not mentioned above, as the router don’t care about them. They are on the layer higher than what a router understands. An official web-server is always port 80, while the client’s TCP-port is carefully randomly chosen. This we will get back to.

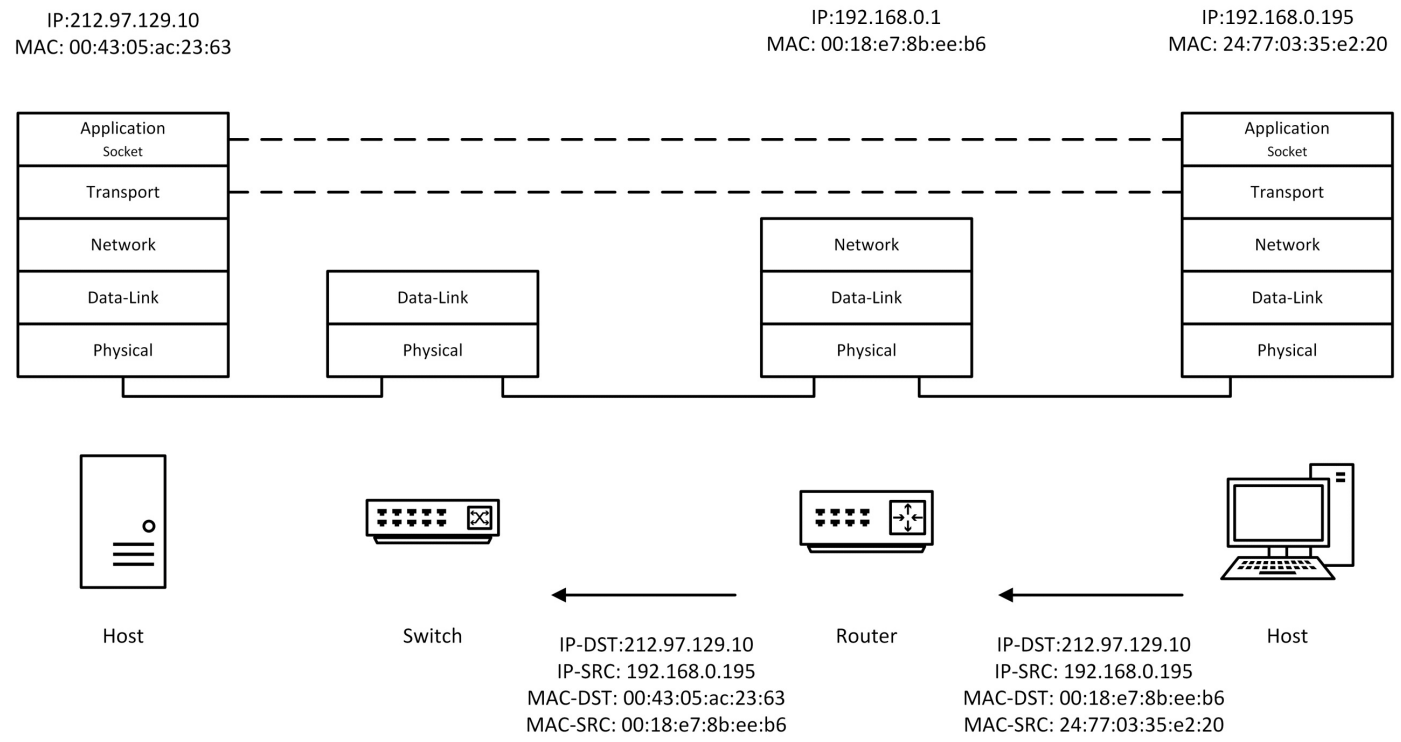



**Figure 4.2:** Internet with some network elements

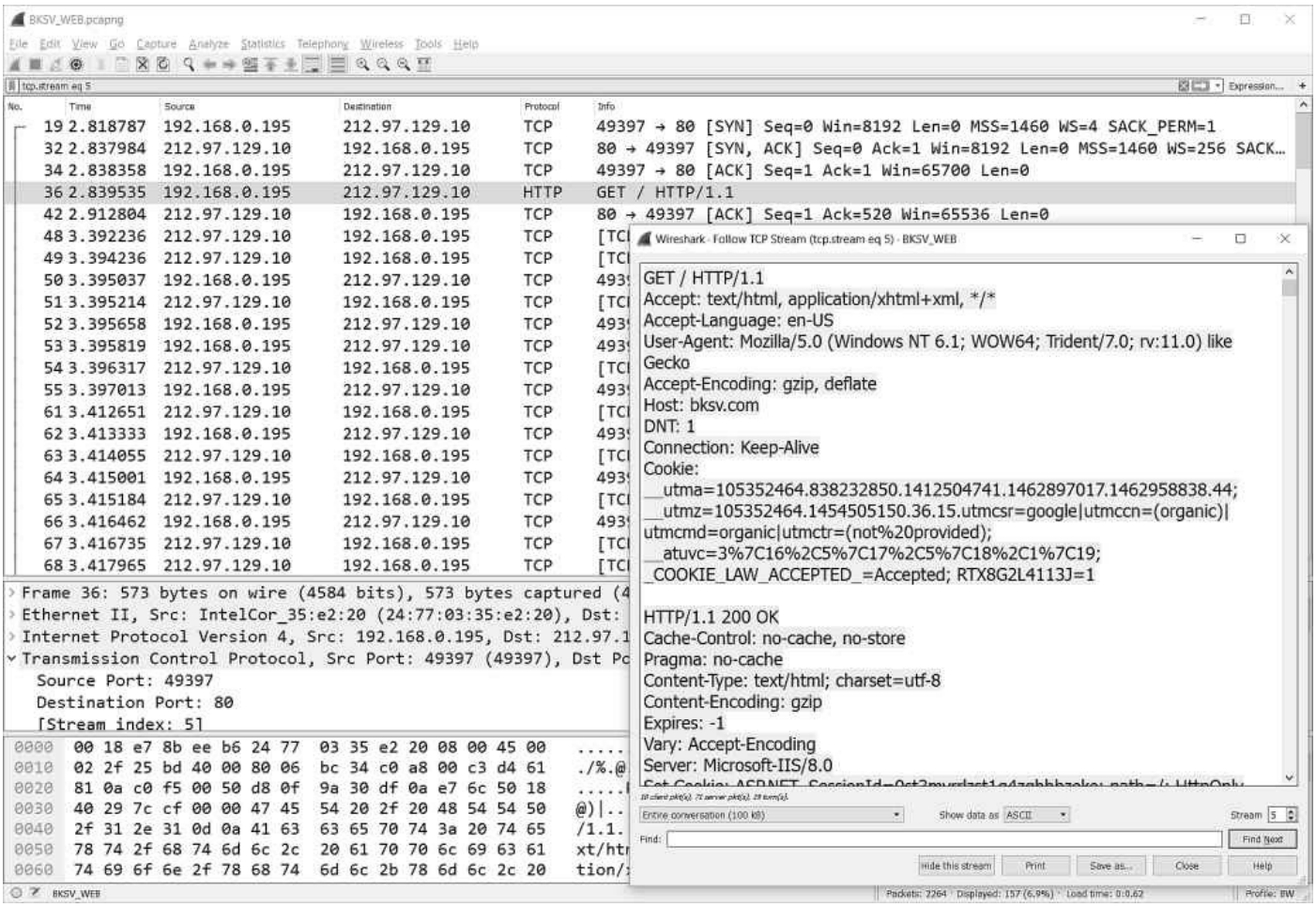



**Figure 4.3:** Transmission with HTTP in clear text

Figure 4.3 shows the the scenario from Figure 4.2, now captured on Wireshark. Each line in the top window in Wireshark is a “frame”, which is the transmitted unit on the Ethernet layer. The correct term for the transmitted unit in TCP is a “segment”, but we will generally keep referring to “frames”, since this is what Wireshark enumerates. The relation between frames (in the Ethernet Layer), packets (in the IP layer), segments (in the Transport Layer) and messages (in the Application Layer) can be tricky. Several small messages may be added into one segment, while large messages may be split over several segments - with TCP the boundaries between messages from the application layer disappears due to the “stream” nature of TCP. A TCP segment may be up to 64 kBytes, but TCP has a “Maximum Segment Size” parameter that is normally set so that a single segment can fit into an Ethernet frame. When this is the case - as it is in most samples in this book - there is a 1:1 correspondence between a segment in TCP and a frame in the Ethernet. We only see the relevant conversation, which is why many frame numbers are missing in the left column. This filter was easily created by right-clicking on the frame with the HTTP-request (no 36), and selecting “Follow TCP-Stream” in the context sensitive menu. This again was easy as Wireshark by default fills the “info” field with relevant data from the “highest” protocol it knows, in this case HTTP, so finding the “GET” request was simple. The other result of the “Follow TCP-stream” is the window on-top at the right, showing the HTTP-communication in ASCII by default. It even colors the client part red and the server-part blue. This is very nice, but can be a little confusing as this is the full conversation, not just the selected frame, and thus includes the following frames that in the info-field contain the text “[TCP segment of reassembled PDU]”.

The middle window shows the selected frame (no 36). The bottom window shows headers and data in hexadecimal. A nice feature is that if you select something in the middle window the corresponding binary data is selected in the bottom window. Notice that Wireshark shows the Internet Protocol Stack “Bottom Up” with the Ethernet on top and HTTP at the bottom. Each of these can be expanded as we shall see later. Each of the non-expanded lines in the middle window still show the most important information - TCP Ports, IP Addresses and MAC-addresses. As the latter are handed out in bundles, Wireshark often recognizes the leftmost 3 bytes and inserts a vendor-name in one of the two versions of the same MAC-address. This makes it easier to guess which device you are actually looking at. In this case we see that the client is a PC, with an Intel motherboard.

HTTP doesn’t just fly out of the PC in frame no 36. Notice frames 19,32 and 34. Together they form the “three-way-handshake” of TCP that initiates a “socket”. You can always start a Wireshark capture and start collecting frames, but it’s better to catch these first, as they contain some initial information, which you may need as we shall soon see. It is important to get some terms right: the **TCP-client** is the host that sent the first frame (19) with only the SYN-flag set and the **TCP-server** is the one that responds with both SYN and ACK set (frame 32). The terms client & server are related to TCP, but they correspond to the web-browser and the web-server. In theory a communication could open on another TCP-socket in the opposite direction, but this is not the case here.

The terms “sender” (or transmitter) and “receiver” are **not** the same as server and client - and they are more dynamic. You may argue that most information is going from the server to the browser, but certainly not all. In a TCP-socket data may - and typically will - flow in both directions, and it is up to the application layer to manage this. However, for now we are digging into what we may learn from the initial handshake via the “info” field on frame 19 in Wireshark in Figure 4.3:

- **SYN**
  This flag (bit) is set by the TCP-client only in the initial opening request for a TCP. The answer from the TCP-server on this particular packet, also contains the SYN flag.
- **ACK** [not in frame 19]
  Contained in all frames except the client’s opening SYN. This allows you to tell the client from the server.
- **Seq=0**
  The 32-bit “sequence number” of the first byte in the segment sent. Many protocols number their packets, but TCP numbers the bytes - allowing for smarter retransmissions. The SYN flag and the closing FIN-flag (outside the screen), are counted as bytes in this sense. Because we got the transmission from start, Wireshark is nice and is giving us relative sequence numbers (starting from 0). If you open the hex-view on frames you will see that the sequence number does not start from 0. In fact this is an important part of the security. Sequence numbers are unsigned and simply wraps.
- **Ack** [not in frame 19]
  The sequence number that the client or server expects to see from the other side. As

either side will sometimes send ACK's with no data there is nothing wrong in finding several frames with the same Seq or Ack no. Wireshark will help you and tell you if either side, or Wireshark itself, has missed a frame.

- **Win=8192**
  Window-Size. This 16-bit number used by both sides tells the other side how much space it currently has in its receive buffer. This is used by the sender to know when to stop transmitting, and wait for the receiver to pass the data up to the application above it. As the receiver guarantees its application that data is delivered only once, without gaps, and in order, this buffer may easily be full if an early packet was lost. Not until the gap is filled by a retransmission will the data go to the application. This is the reason why you sometimes will see packets that are data-less, but are marked by Wireshark as a "Window Update". This means that the receiver (either client or server) wishes to tell the other side: "I finally got rid of some data to my application, and I now have room for more".
- **WS=4**
  Window Scale. TCP is old, and originally a 16-bit number was thought to be enough for the Window-Size. But today's "long-fat-pipes" are only utilized decently if we allow for a lot of data on-the-fly. The backward-compatible fix was to introduce a "window-scale" which is an option in the opening handshake. If the client uses this option, the WS contains the number of bits the future Window-Size numbers could be left-shifted from the client. If the server responds by also using this option, this is an acknowledgement that the client's suggestion is accepted, and the WS sent from the server is the scale the server will use in its Windows-Size. Thus the two WS do not have to be the same. All this explains why Wireshark often reports a Window-Size larger than 65535 even though the Window-Size is a 16-bit number. This is the main reason why you should always try to get the handshake included in the capture.
- **SACK_PERM=1**
  Selective Acknowledge Permitted. The original implementation of TCP is rather simple. The ACK-number can be understood as: "This is how far I have received data from you without gaps". So the transmitter may realize that it has sent something that the receiver has lost, and it will retransmit bytes from this number. However, with the "big-fat-pipes", the sender may need to resend a lot. With SACK_PERM the host says that it is capable of understanding a later addendum - the selective acknowledge. This allows the receiver to be more specific about what is has received and what not. The ACK no still numbers the next expected byte and **all** bytes before this are safely received but with the help of SACK the receiver can tell the transmitter about some "well-received" newer data blocks after a gap.
- **MSS=1460**
  Maximum-Segment-Size. This is the maximum-size in bytes of the payload in the TCP-segment (the data from the application). This is related to the MTU - see Section 4.16.

  If segments longer than MSS are sent, they may get there, but the overhead will affect performance. This is where packets on the IP-layer becomes "fragmented" into more frames in the Ethernet layer - see Section 4.13.

- **49397->80**

These are the port-numbers in play. Browsers expect web-servers to use port 80 by default as we see here. 49397 is the “ephemeral” port-number selected randomly by the operating system for the web-browser.

Amazing how much there is to learn from studying the “info” on a single packet. The term “socket” was used loosely along the way. A TCP-socket is completely defined by the “tupple” consisting of 5 elements: The two IP-addresses, the two Port-numbers, and the protocol (TCP). If the web-browser wishes to open more sockets towards the same web-server there is really only one of these that can be changed; the web-browser’s (client’s) own TCP port. This is exactly what happens when the browsers OS picks a new ephemeral (semi-random) number.

## 4.4 Life before the packet

When e.g. a PC on a local Ethernet is told to talk to another PC on the same local-net, the PC is given an IP-address to talk to. However, the two nodes [1] needs to communicate via their MAC-addresses. Typically the PC knows the MAC-address of the Gateway router from the original DHCP, but what about host-to-host? This is where ARP comes in - “Address-Resolution-Protocol”. A host keeps a list of corresponding IP and MAC addresses, known as an ARP-cache, and if the IP-address of the destination is not in this, the host will issue an ARP-request - effectively asking “Who has IP address xx?” When another node replies, the information is stored in the ARP-cache. The replying host may also put the requesting node in its ARP-table as there apparently is going to be a communication.

Listing 4.1 shows a PC’s ARP-cache. All the entries marked as “dynamic” are generated as just described, and they all belong to the subnet with IP-addresses 192.168.0.x. The ones that are marked “static” are all (except one) in the IP-range 224.x.y.z and 239.x.y.z. This is a special range reserved for “multicasting”, and this is emphasized by their MAC-addresses that all start with 01:00:5e and ends with a byte that is the same number as the last byte in the IP-address.[2] The last entry is the “broadcast” where all bits in both MAC and IP-address are 1. Such a broadcast will never get past a router. Note that in Listing 4.1 there is no entry for 192.168.0.198.

Listing 4.2 shows an ICMP “ping” request - a low level “are you there?” to the IP-address 192.168.0.198, and a following display of the ARP-cache - now with an entry for this address.

```
01 C:\Users\kelk>arp -a
02
03 Interface: 192.168.0.195 – 0xe
04   Internet Address      Physical Address      Type
05   192.168.0.1           00-18-e7-8b-ee-b6     dynamic
06   192.168.0.193         00-0e-58-a6-6c-8a     dynamic
07   192.168.0.194         00-0e-58-dd-bf-36     dynamic
08   192.168.0.196         00-0e-58-f1-f3-f0     dynamic
09   192.168.0.255         ff-ff-ff-ff-ff-ff     static
10   224.0.0.2             01-00-5e-00-00-02     static
11   224.0.0.22            01-00-5e-00-00-16     static
12   224.0.0.251           01-00-5e-00-00-fb     static
13   224.0.0.252           01-00-5e-00-00-fc     static
14   239.255.0.1           01-00-5e-7f-00-01     static
15   239.255.255.250       01-00-5e-7f-ff-fa     static
16   255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

**Listing 4.1:** ARP-Cache with router at top

```
01 C:\Users\kelk>ping 192.168.0.198
02
03 Pinging 192.168.0.198 with 32 bytes of data:
04 Reply from 192.168.0.198: bytes=32 time=177ms TTL=128
05 Reply from 192.168.0.198: bytes=32 time=4ms TTL=128
06 Reply from 192.168.0.198: bytes=32 time=3ms TTL=128
07 Reply from 192.168.0.198: bytes=32 time=5ms TTL=128
08
09 Ping statistics for 192.168.0.198:
10     Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
11 Approximate round trip times in milli-seconds:
12     Minimum = 3ms, Maximum = 177ms, Average = 47ms
13
14 C:\Users\kelk>arp -a
15
16 Interface: 192.168.0.195 – 0xe
17   Internet Address      Physical Address      Type
18   192.168.0.1           00-18-e7-8b-ee-b6     dynamic
19   192.168.0.193         00-0e-58-a6-6c-8a     dynamic
20   192.168.0.194         00-0e-58-dd-bf-36     dynamic
21   192.168.0.196         00-0e-58-f1-f3-f0     dynamic
22   192.168.0.198         10-7b-ef-cd-08-13     dynamic
23   192.168.0.255         ff-ff-ff-ff-ff-ff     static
24   224.0.0.2             01-00-5e-00-00-02     static
25   224.0.0.22            01-00-5e-00-00-16     static
26   224.0.0.251           01-00-5e-00-00-fb     static
27   224.0.0.252           01-00-5e-00-00-fc     static
28   239.255.0.1           01-00-5e-7f-00-01     static
29   239.255.255.250       01-00-5e-7f-ff-fa     static
30   255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

**Listing 4.2:** ARP-Cache with new entry

Figure 4.4 shows a Wireshark capture of the scenario. The first of three ping requests is sent in frame 42, with a reply in frame 45. Before this however we see the ARP request from the pinging PC in frame 40, and the response in frame 41. Notice that the source and destination addresses here are not IP-addresses, but Ethernet-addresses (partly filled with the vendor-names of the MAC's). Also note that the request naturally is a broadcast, while the reply is a "unicast". The response could also have been a broadcast, but a unicast disturbs less.

We also here see an explanation for the extended time for the first ping. The responding Windows 7 PC, decides to do its own ARP, 100 ms after having answered the original ARP. This is a defense against "ARP poisoning". If more than one device answers this second ARP it is a sign of a security problem. As a side note, the display filter used in Wireshark is somewhat nontraditional. To avoid a swarm of unrelated traffic in the figure, it was easy to filter based on frame numbers.

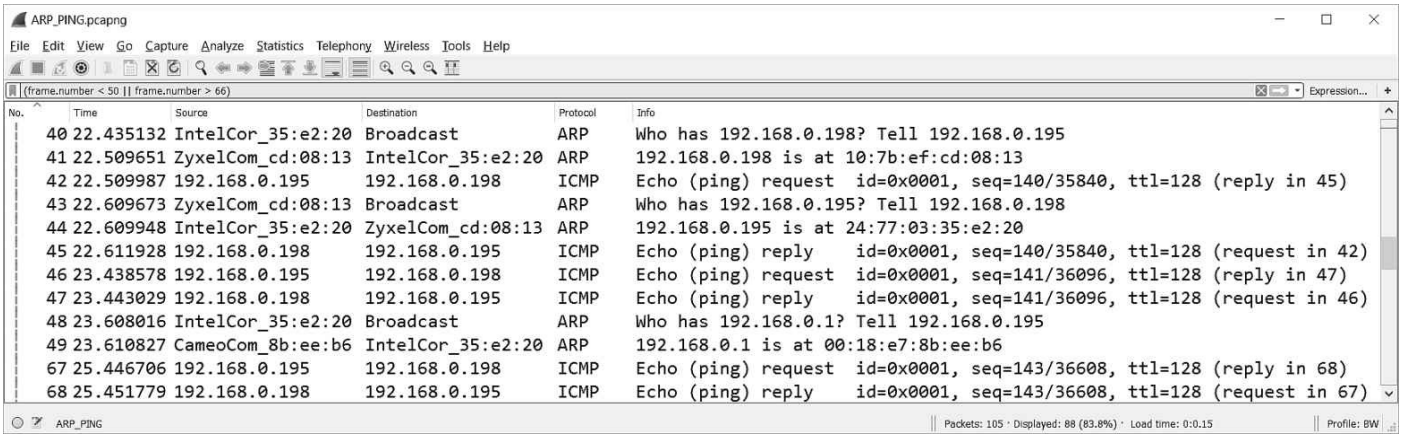



**Figure 4.4:** A ping provoking ARP

## 4.5 Getting an IP address

Network interfaces are born with a unique 48-bit MAC-address. This is like a persons social security number - it follows the interface for life. Not so with IP-addresses. IP-addresses are hierarchical to facilitate routing. This is the same as letters that are addressed to country - ZIP - street and number - floor. This means that IP-addresses may change and with mobile equipment they change a lot. There are a number of ways to get an IP-address:

- **Statically Configured**
  This can be good in a - surprise - static system as the address is then ready from power-up and does not change. Many systems are using static addresses in one of the private IP-ranges - see Section 4.8 - in “islands” that are not connected to the internet but to a dedicated Network Interface Card on a PC. At B & K we provide large systems with Racks and Modules that goes into Slots in the Racks - each IP-addressable. We introduced an addressing scheme for the IP-address of a module: 192.168.(rackno).(slotno) - and network mask 255.255.0.0. With this it is easy to find a module from its address.
- **DHCP**
  Dynamic Host Configuration Protocol. This is practical with laptops and phones that move from home to work to school etc. When the device joins the network it gets a relevant IP address - and more.
- **Reserved DHCP**
  Via their web-page most SOHO (Small Office/Home Office) routers allow you to fixate the DHCP address handed out to your various devices - linking it to the MAC-address of the device. This is very practical as your PC, phone or whatever can remain a DHCP-client wherever you go and still you can be sure to always have the same IP-address in your home. This is nice for the more advanced stuff.
- **Link Local**
  When a device is setup to be a DHCP-client, but cannot “see” a DHCP-server, it waits some time and finally takes an address in the range 169.254.x.y. First it tests that relevant candidate address are free - using ARP - and then it announces the claim with “Gratuitous ARP” - see Section 4.4. This is a nice fallback solution that will allow two hosts to communicate, if you can get past all the security. Microsoft calls link-local for “Auto-IP”.

  Note that should the DHCP-server become visible after a link-local address is picked, the device will change IP-address to the one given from the server.

- **Link Local IP6**
  IPv6 includes another version of Link-Local that makes the address unique - based on the MAC-address on the interface.

## 4.6 DHCP

Listing 4.3 shows how the DHCP process is easily provoked on a windows PC.

```
01 C:\Users\kelk>ipconfig /release
02 …skipped…
03 C:\Users\kelk>ipconfig /renew
```

**Listing 4.3:** DHCP release & renew

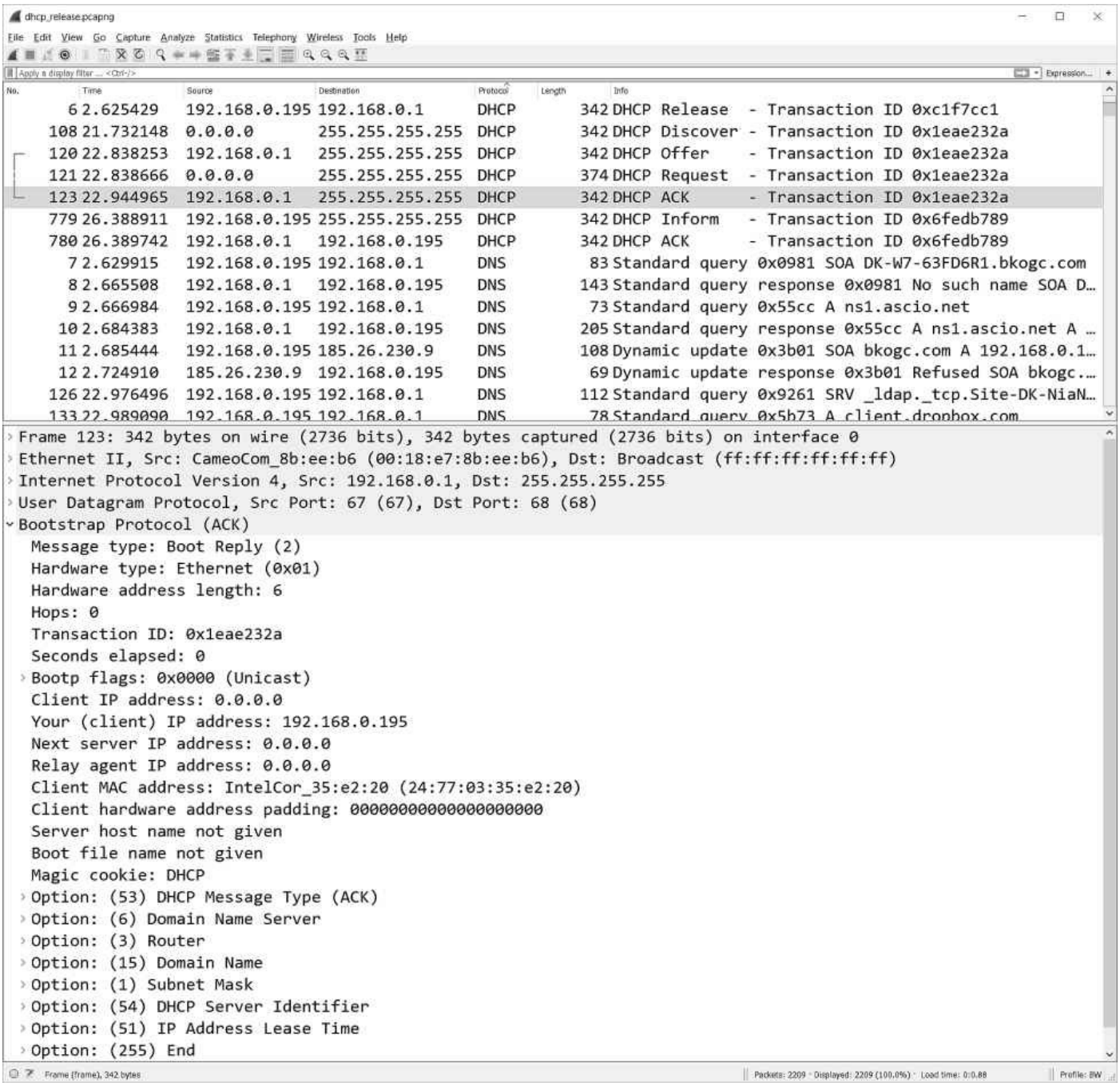



**Figure 4.5:** DHCP commands and options

Figure 4.5 shows the corresponding Wireshark capture. This time no filter was applied. Instead Wireshark was told to sort on the "protocol" by clicking on this column. This organizes the frames alphabetically by the name of the protocol, and then by frame number. Frame by frame we see:

- 6: "DHCP Release" from the PC caused by line 1 in 4.3. Here the PC has the same address as in the previous sections: 192.168.0.195. After this it is dropped.
- 108: "DHCP Discover" from the PC. This is a broadcast on the IP-level which can be seen by the address: 255.255.255.255. It is also a broadcast on the Ethernet level with the address: ff:ff:ff:ff:ff:ff. This is a request for an IP-address from any DHCP-server. Note the "Transaction ID": 0x1eae232a - not the same as in the Release.
- 120: "DHCP Offer" from the server - sent as broadcast since the PC has restarted the

process and therefore has no address yet. However, this offer contains the MAC-address of the PC as well as the original Transaction ID and cannot be received by mistake by another client. It contains the future IP-address and mask, as well as the IP-address of the DNS-server (see Section 4.9) and Gateway Router. It also includes the "Lease Time" which is how long the PC can wait before it asks for a renewal.

- 121: "DHCP Request" from the PC. Why is that? - we just had the whole thing handed on a silver-plate! The explanation is that another DHCP-server might also have offered us an address etc. So this packet contains more or less the same as the offer - but now from the PC to the server. There is one extra thing: the "Client Fully Qualified Domain Name" which is the "Full Computer Name" also found in the "System" group in the Control Panel on Windows PC's. This actually explains why a home-router may identify a work-PC with your company's domain.
- 123: "DHCP ACK" from the server. Finally, the deal is sealed.
- 779: "DHCP Inform" from the PC. This is a new message in relation to the standards. According to IETF, it was introduced to enable hosts with a static IP-address to get some of the other information (DNS etc). Also according to IETF this has been seen to be kind of misused. This appears to be one such case.
- 780: "DHCP ACK" from the server. Answer to the above.

When a DHCP release is about to run out, or "ipconfig /renew" is used without the release, only the DHCP Request from the PC and the DHCP ACK from the server is seen - and in this case they are unicast. In other words, the PC does not "forget" its address but uses it while it is renewed.

Figure 4.6 shows the "Link Local" scenario where the DHCP-client cannot see any server. Frames 294 and 296 are the last attempts on DHCP. With frame 295 the PC starts ARP'ing for 197.254.138.207. This is what the protocol stack inside the PC is planning on using, but only if its not already in use. After the third unanswered ARP we see a "Gratuitous ARP". This is the PC using the ARP-protocol - not to ask for anyone else with the link-local address, but actually informing its surroundings that it is now taken. Frame 299 is the "selected" and in the window below we see that "Sender Address" is 169.254.138.207. In other words, the PC is asking for this address, and saying that it has it. Hence the term "Gratuitous".

---

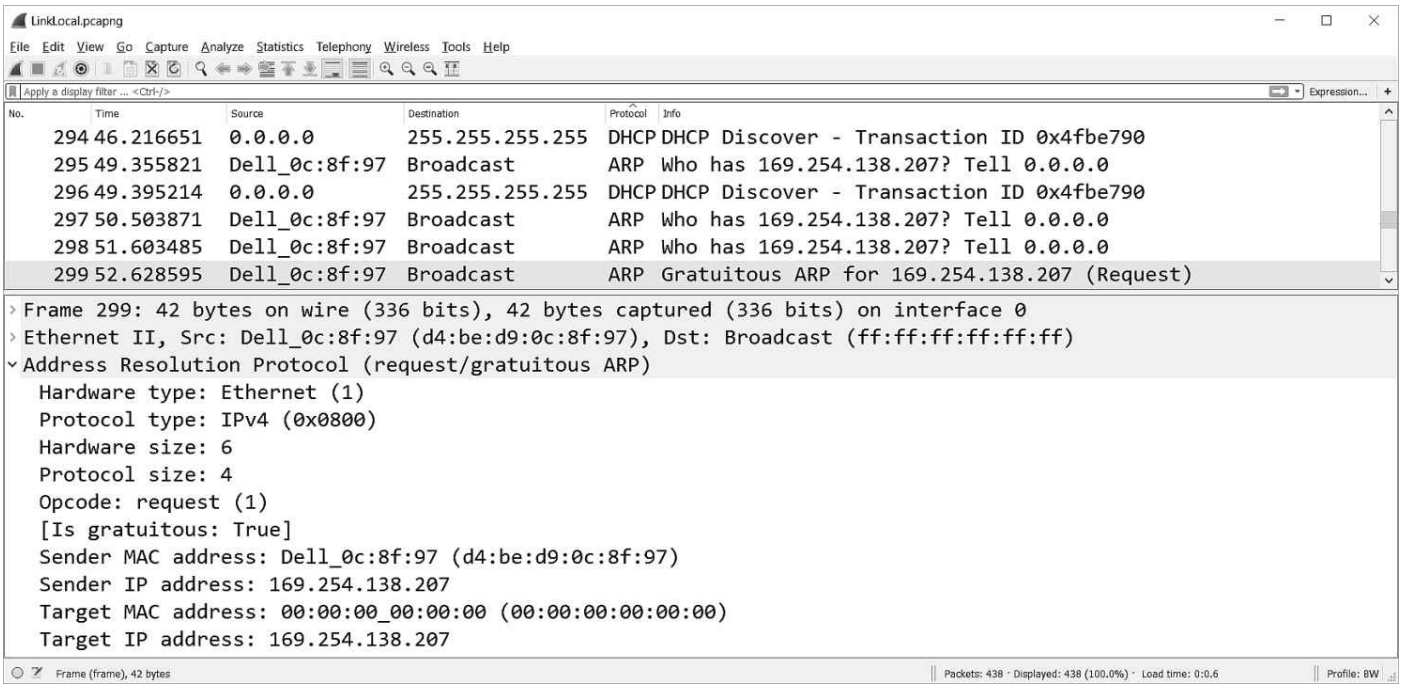



**Figure 4.6:** Link-Local with Gratuitous ARP

## 4.7 Network Masks, CIDR and Special Ranges

We have touched briefly on Network Masks and many people working with computers have a basic understanding of these, but when it comes to the term “subnet” it gets more fluffy. It might help to do a backwards definition and say that a subnet is the network island you have behind a Router. As discussed in Section 4.3, frames inside such an island are sent by their Ethernet address - using ARP-cache and ARP-protocol. Inside the island all IP-addresses share the same first “n” bits - defined by the mask. A typical mask in a SOHO (Small Office/Home Office) installation is 255.255.255.0. This basically means that the first 24 bits are common and defines the Network ID, and this is the subnet, while the last 8 bits are what separates the hosts from each other - their Host-ID. Since we do not use addresses that ends with “0”, and the Host-ID address with all bits set is for broadcasts, this leaves 254 possible Host-ID’s in the above example. Network masks used to be either 255.0.0.0 or 255.255.0.0 or 255.255.255.0 - and the networks were respectively named class A, B and C. These terms are still used way too much. In many companies class C was too little, while class B was too much - and who needs class A? A lot of address space was wasted this way as big corporations were given a class A to use at will. While everybody was waiting for IPv6 to help us out, CIDR and NAT was born. CIDR is “Classless Inter-Domain Routing”. It allows subnet definitions to “cross the byte border”. To help, a new notation was invented where “/N” means that the first N-bit of the given address is the subnet, and thus defines the subnet mask as the leftmost N-bits, leaving 32-N for the Host ID. The IP-address 192.168.0.56/24 is the same as a classic class C mask, and thus the Host-ID is 56. However, we can also have 192.168.0.66/26, meaning that the mask is 255.255.255.192, the Network ID is 192.168.0.64 and the Host ID is 2. Relatively easy numbers are used in these examples, but without a program it quickly becomes complicated - which is probably why the old masks are die-hards. There are however many apps that help with this.

## 4.8 Reserved IP-Ranges

We have seen addresses like 192.168.0.x a lot of times. Now, let us take a look at what is known as “Reserved IP ranges”. The “Private Ranges” subset will not get past a Router. This makes them perfect for the home or company network, as they are not blocking existing global addresses and can be reused again and again in various homes and institutions. Another advantage of these addresses not being routed is that they cannot be addressed directly from the outside of the home or office - thus improving security. The “Multicast Range” is also known as “Class D”. There are more ranges, but these are the most relevant:

| CIDR | Range | Usage |
|---|---|---|
| 10.0.0.0/8 | 10.0.0.0-10.255.255.255 | Private |
| 169.254.0.0/16 | 169.254.0.0-169.254.255.255 | Link-Local |
| 172.16.0.0/12 | 172.16.00-172.31.255.255 | Private |
| 192.168.0.0/16 | 192.168.0.0-192.168.255.255 | Private |
| 224.0.0.0/4 | 224.0.0.0-239.255.255.255 | Multicast |
| 240.0.0.0/4 | 240.0.0.0-255.255.255.254 | Future use |
| 255.255.255.255 | Broadcast | Only on subnet |

## NAT

NAT - Network Address Translation - has been extremely successful - at least measured by how it has saved IP-addresses. The reason is that most households needs to be able to act as TCP-clients against a lot of different servers, but very few households have a server that others need to be client against. They are in fact mostly happy (or ignorant) about not being addressable directly from the internet. A NAT assures that we on the inside can live with our private IP-addresses, while on the outside we have a single IP-address. This external address may even change once-in-awhile without causing problems - again because we have no servers. So what does it do?

Say that the external address of a company or home is 203.14.15.23 and on the inside we have a number of PC’s, tablets and phones in the 192.168.0.0/16 subnet. Now the phone with address 192.168.12.13 uses a web-browser to contact an external web-server (port 80). The NAT opens the packet and swaps the internal IP address with the external. It also swaps the source port number with something it chooses. The packet moves to the web-server where the request is answered and source and destinations of ports and IP-addresses are swapped, so that the packet is routed back to the external address where the NAT switches back. If the NAT originally also stored the external address (rightmost column in table) it can even verify that this in-going packet indeed is an answer to something we sent. The NAT is said to be stateful.

The table below is using the standard notation IP:Port. To make it easy to see what is

happening the NAT ports are consecutive, which they are not in real life. All rows, except the last, show packets generated internally, going out to web-servers or mail-servers and being replied. The last row shows how an external bandit attempts to fake a packet as a response to a web-request. As there is no match it is dropped.

| Intranet | NAT | Internet |
|---|---|---|
| 192.168.12.13:4603 | 203.14.15.23:9000 | 148.76.24.7:80 |
| 192.168.12.13:4604 | 203.14.15.23:9001 | 148.76.24.7:80 |
| 192.168.12.10:3210 | 203.14.15.23:9002 | 101.23.11.4:25 |
| 192.168.12.10:3211 | 203.14.15.23:9003 | 101.23.11.4:25 |
| 192.168.12.28:7654 | 203.14.15.23:9004 | 145.87.22.6:80 |
| Nothing | Nothing | 205.97.64.6:80 |

The basic NAT-functionality only requires the NAT to store the two left columns (thus being stateless). By looking at the incoming TCP-segments from the internet it can throw away any segment that hasn't got the "ACK" bit set, as this can only be a client opening "SYN" which is normally not allowed, see Section 4.12.

By storing the third column as well security is improved. The NAT concept can also be used on server-parks for load-balancing when the protocol is stateless, see Section 4.11. The mappings shown in the table are *dynamic*, as they are generated from traffic. It is possible to create static mappings. If e.g. a PC on the intranet is setup as a company web-server, the static mapping will send all packets towards port 80 to this PC. This way we are actually able to have a server inside our network, but now we will prefer that the external IP address is fixated, as this is what we tell DNS-servers is our address.

Not everyone is happy about NAT. It conflicts with the concept of layering, see Section 7.3. If an application embeds information about ports and IP-addresses in the application protocol, this is not translated, which will cause problems.

## 4.9 DNS

The Domain Name System was invented because people are not really good at remembering long numbers. You may be able to remember an IPv4 address, but probably not an IPv6 address. With the DNS we can remember a simple name like "google.com" instead of long numbers. This is not the only feature. DNS also gives us a first-hand scale ability assistance. Listing 4.4 was created on a Windows 10 PC. When looking up google.com we get 16 IPv4 answers (and one answer with an IPv6 address). A web-server e.g. could use this to open 16 different sockets targeting different servers. Most applications are lazy and are simply using the first answer and for this reason the next call will show that the answer is "cycled". Google does not need this kind of help, they are masters in server parks, but for many smaller companies this means that they can have a scalable system with a few web-servers simply by registering them to the same name.

```
01 C:\Users\kelk>nslookup google.com
02 Server:  UnKnown
03 Address:  192.168.0.1
04
05 Non-authoritative answer:
```

```
06 Name:    google.com
07 Addresses:  2a00:1450:4005:80a::200e
08           195.249.145.118
09           195.249.145.114
10           195.249.145.98
11           195.249.145.88
12           195.249.145.99
13           195.249.145.103
14           195.249.145.113
15           195.249.145.109
16           195.249.145.119
17           195.249.145.84
18           195.249.145.123
19           195.249.145.93
20           195.249.145.94
21           195.249.145.89
22           195.249.145.108
23           195.249.145.104
```

**Listing 4.4:** Simple nslookup

In order for a URL to be handed out by the DNS is needs to be registered. This can be done via many organizations as a simple search will show. Once registered the name and the belonging IP-addresses are programmed into the 13 root DNS-servers. When your program asks its local DNS-resolver about a URL the request will be sent upwards through the hierarchy of DNS servers until it is answered. Answers are cached for some time in the servers they pass. This time is the TTL - Time-To-Live - which is set by the administrator.

The actual DNS request is using UDP. This is the fastest way, with no “social throttling” and these requests are so small that they can easily fit into a single UDP-segment - no matter which protocols are beneath it. DNS requests has two forms - recursive or not. In the recursive case the DNS-server takes over the job and queries further up the hierarchy, and eventually comes back with a result. In the non-recursive case the resolver must ask the next DNS-server in the hierarchy itself - using an IP-address for this that it got in the first answer etc. Figure 4.7 shows a Wireshark caption of an older “googling”, with some completely different answers. The selected frame is no 2 - containing the answer. We see that the request asked for - and got - a recursive query. We also see that in this case we got four answers and in frame 3 we see that the web-browser is using the first (64.233.183.103) for its HTTP-request. The request is of type “A” which is when we ask for an IPv4 address for a given URL. Had it been “AAAA” we would have received IPv6 addresses. There are many more types, including “PTR” used for reverse lookups.

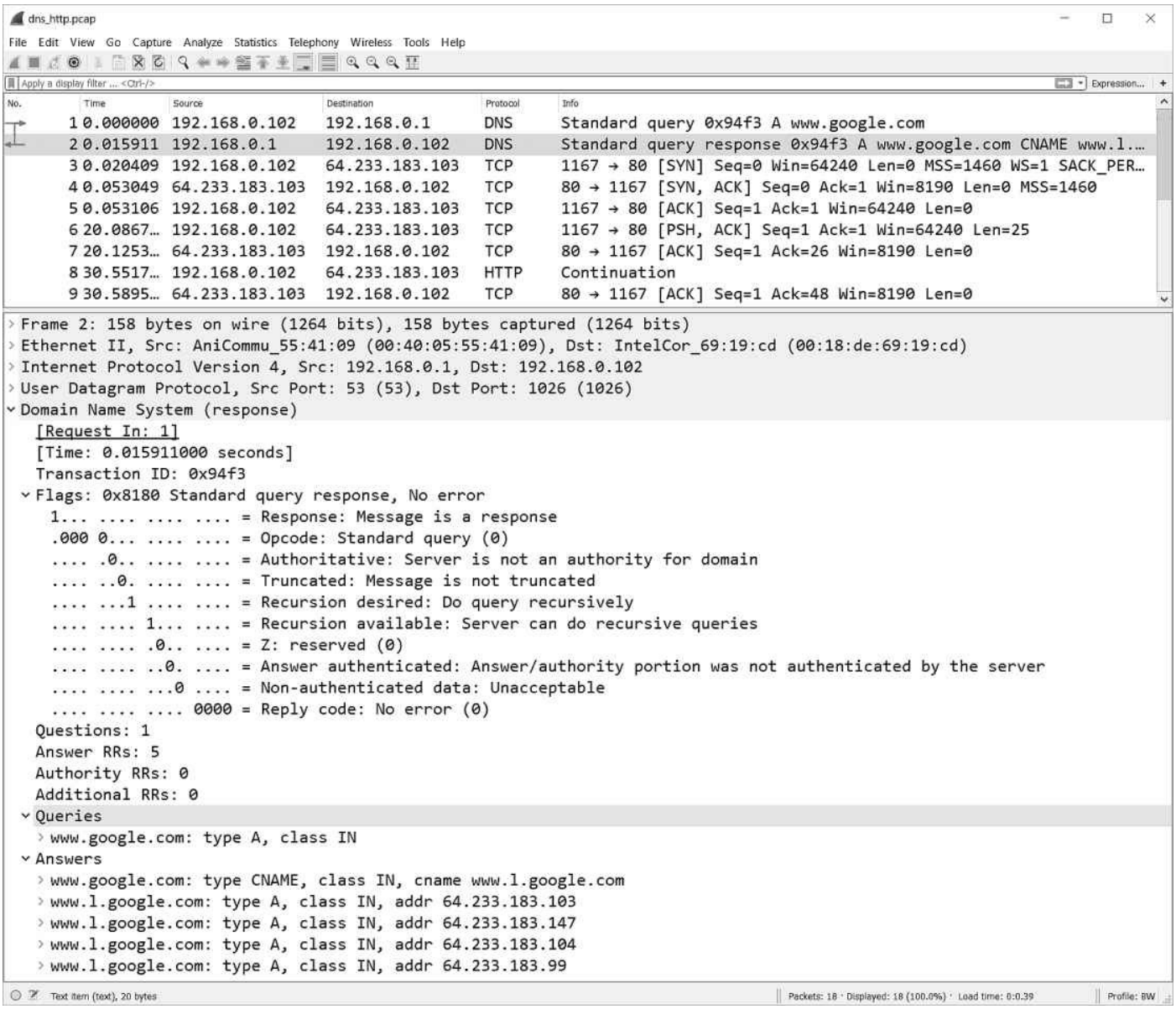



**Figure 4.7:** A DNS request for google.com

Many smaller embedded systems operate directly on IP-addresses and the whole DNS concept is then irrelevant. A typical IOT system will have its cloud-server registered in the DNS but normally not the devices.

## 4.10 Introducing HTTP

Telnet is not used very much anymore, as there is absolutely no security in it. Nevertheless, it is great for showing a few basics on HTTP. Listing 4.5 shows a telnet session from a Linux PC (the windows version is not good, and not part of the basic install anyway).We are basically doing exactly the same thing as was done before with a browser; an HTTP-GET request in the root at www.bksv.com. We specifically need telnet to abandon its default port (23) and use port 80 instead. Once connected to www.bksv.com we can send all the same lines that we can see in the Wireshark communication in clear-text in Figure 4.3. However, this time only the two that are mandatory was sent - see Listing 4.5, lines 5 and 6. Line 5 is no big surprise; with “www.bksv.com” written

originally in the browser, and nothing more, we go to the root - “/” - and HTTP/1.1 is the standard supported by the browser.

But how come we need to send “Host: www.bksv.com?” After all it was just written already in line 1: “telnet www.bksv.com 80”. The reason is that the URL in line 1 is immediately substituted with an IP address before leaving the PC at all - thanks to the DNS. A modern web-server-installation can actually host many “sites”. We need line 6 to tell the server which site we want to talk to.

That was the mandatory headers. According to the HTTP-standard we now send two line terminations (CR-LF), and then any “body”. Since this is a GET there is no body, and we therefore see the response starting in line 8. The response does have some body, which we see starting in line 21, after the two times CR-LF. This body is HTML - no surprise there. Also note line 18 that states “Content-Length” - the number of bytes in the body, pretty much the same as a file-length.

```
01 kelk@debianBK:~$ telnet www.bksv.com 80
02 Trying 212.97.129.10…
03 Connected to www.bksv.com.
04 Escape character is '^]'.
05 GET / HTTP/1.1
06 Host: www.bksv.com
07
08 HTTP/1.1 200 OK
09 Cache-Control: no-cache, no-store
10 Pragma: no-cache
11 Content-Type: text/html; charset=utf-8
12 Expires: -1
13 Server: Microsoft-IIS/8.0
14 Set-Cookie: ASP.NET_SessionId=nvyxdt5svi5x0fovm5ibxykq; path=/; HttpOnly
15 X-AspNet-Version: 4.0.30319
16 X-Powered-By: ASP.NET
17 Date: Sun, 15 May 2016 11:46:29 GMT
18 Content-Length: 95603
19
20
21 <!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN"
22         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
23 <html id="htmlStart" xmlns="http://www.w3.org/1999/xhtml"
24           lang="en" xml:lang="en">
25 <head><META http-equiv="Content-Type" content="text/html;␣charset=utf-8">
26 <title>Home - Bruel &amp;amp; Kjaer</title>
27 <meta http-equiv="X-UA-Compatible" content="IE=edge">
28 <meta name="DC.Title" content="…….”
```

**Listing 4.5:** Telnet as webbrowser

The main HTTP commands, and what they do:

| HTTP | Usage | SQL |
|---|---|---|
| POST | **C**reates Information | INSERT |
| GET | **R**etrieves information | SELECT |
| PUT | **U**pdates information | UPDATE |
| DELETE | **D**eletes information | DELETE |

Note how SQL-database statements suddenly popped up. In the database world we operate with the acronym CRUD, for “Create”, “Retrieve”, “Update” and “Delete”. This is basically enough to manage, expose and utilize information, and isn’t that exactly what IOT is about?

## 4.11 REST

While many of us were struggling to comprehend SOAP, the architectural concept from Microsoft and others, a dissertation for a doctorate in Computer Science was handed in by Roy Fielding in the year 2000. In this he introduced an architectural concept called REST - Representational State Transfer. In short this was a concept for managing a device - or resources - be it small as a sensor or huge as Amazon. He gave the following ground-rules for REST:

| Rule | Explanation |
|---|---|
| Addressable | Everything is seen as a resource that can be addressed - e.g. with an URI/URL |
| Stateless | Client and Server cannot get out of synch |
| Safe | Information may be retrieved without side-effects |
| IdemPotent | The same action may be performed more than once without side-effects |
| Uniform | Use simple and well-known idioms |

Surely Roy Fielding already had his eyes on HTTP, although the concept does not have to be implemented using HTTP. There are many reasons why HTTP has been victorious. Simplicity is one - fitting the “Uniform Criteria”. However, the most important fact about HTTP is that it basically **is** stateless. If you take FTP, client and server needs to agree on “where are we now?” This is very bad for scalability - you cannot have one FTP-server answer one request, and then let another in the server-park handle the next. This is exactly what HTTP allows. You can fetch a page from a web-server, and when opening it up, it is full of links to embedded figures etc. When the web-browser next requests these, it can be other servers in the park that delivers them - independent of each other, because every resource has a unique URL (Addressable Rule).

HTTP also lives up to the “Safe” criteria; there are no side-effects of forcing re-fetching of a page many times (apart from burning CPU-cycles). So out of the box HTTP delivers. An obvious trap, which must be avoided, is to have a command such as “Increment Number”. This suddenly adds state on the application level. So if a value is “8” and you want it to be “9”, then ask for “9”. This allows an impatient user (or application) to resend the command without side-effects. Probably REST is actually inspired by some hard learned experiences in the database-world, where there are “client-side cursors” and “server-side cursors”. A database system with a server-side cursor, forces the database to hold state for every client - just like FTP. With a client-side cursor you put the load on state-keeping on the client. Clients are often smaller machines - but they are many, so this scales much better.

There are situations where a common idea of state cannot be avoided. If you are remote-controlling a car, you need to “start the engine” before you can drive, but even

there you may be able to derive, that if the user wants to drive and the engine is not running, we better start it.

It is pretty clear what GET and DELETE does, but what are the differences between PUT and POST? It turns out that PUT is very symmetric to GET. A GET on a specific URL gives you the resource for this URL, and a PUT updates it. POST handles the rest. Typically, you cannot create an object by doing something on the not-yet-existing URL. Instead you ask the "parent" in the containment tree - using POST - to create a child, and it typically gives you the exact URL of the new child. This all fits with the table spelling CRUD. The final thing that you use the POST for is performing actions. These are a little bit on the "edge" of REST, but hard to do without.

```
01 http://10.116.1.45/os/time?showas=sincepowerup
```

**Listing 4.6:** Sample REST GET

Listing 4.6 demonstrates another quality in REST: It is human readable. In fact, it is not hard to create small test-scripts in your favorite language. Likewise, REST is easier to debug in Wireshark than most application protocols. Once you have decided to use REST on HTTP there are still a few decisions to take:

- The data-format in the body. Typical choices are XML or JSON. XML may be the one best known in your organization, while JSON is somewhat simpler.
- How to organize the resources. This is basically the "object-model" - see Section 7.5. If you can create an object model that the whole development team understands and relates to, you are very far. The fact that you can piece together the URL by following the object model from its root is extremely simple - and very powerful. Figure 4.8 is an example of such an object-model, seen via its test-GUI - courtesy Jeppe Kronborg.
- It is possible to address every "node" in the object tree specifically and read/write data. It might also be possible to go to a specific level in the object tree and from there handle the containment hierarchy in JSON or XML. Which concept to choose? This will typically depend on how you want to safeguard the system against unwanted changes, or just the actions you need to do on a change. If you plan on having different user-roles that each has a specific set of what they can change and what not, then it's a good idea to address the nodes individually, so that it is the plugin in the web-browser that uniformly manages user-access, instead of the individual objects. As discussed in Section 7.1 there is a huge performance boost in setting a large tree in one swoop. You might end up enabling the addressing of sub-nodes as well as delivering/fetching whole sub-trees as JSON or XML.

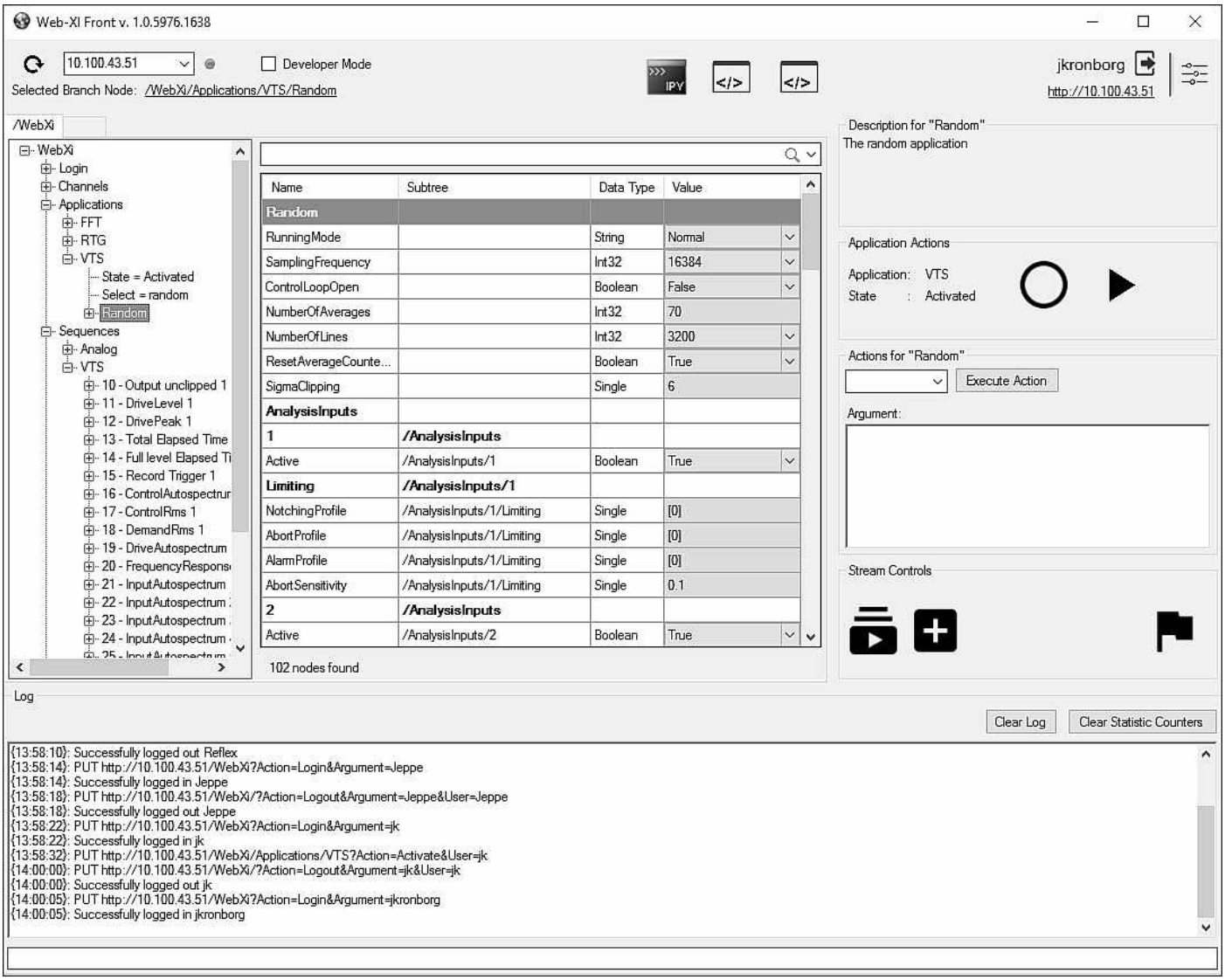



**Figure 4.8:** Test-program showing an Object Model on REST

## 4.12 TCP sockets on IPv4 under Windows

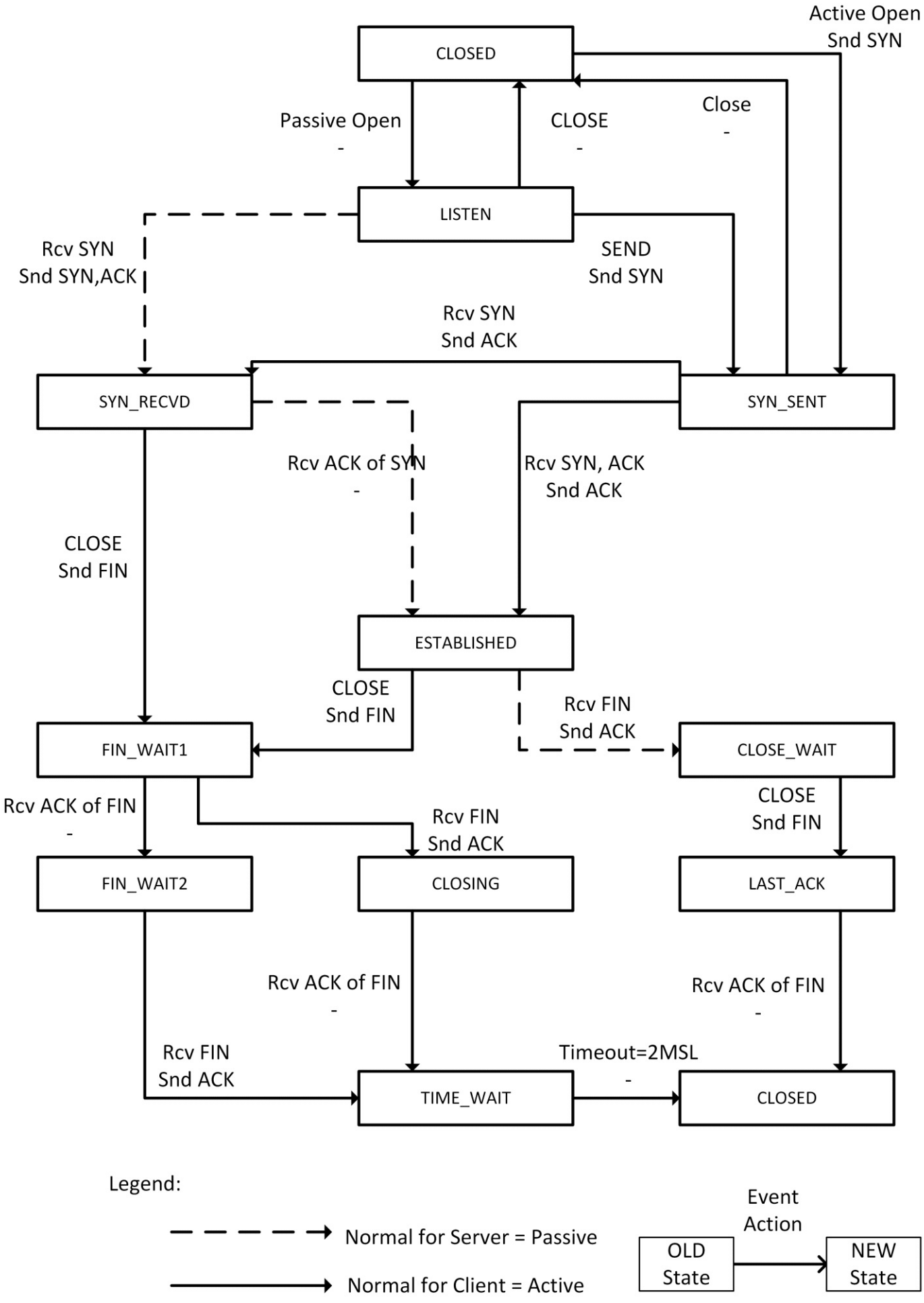
In Section 4.3 we went through the “opening ceremony” for TCP - the three-way handshake. Now is the time to go more in detail with TCP - Transmission Control Protocol - RFC793. The standard has a rough state-diagram, redrawn in Figure 4.9. The state where we want to be is “Established”, the rest is just building up or tearing down the socket. Similar to the opening three-way handshake the closing is normally done using two two-way handshakes. This is because the socket can send data in both directions, and both transmitters can independently say “I have no more to say” which the other side can ACK. Older Windows stacks are lazy - they typically skip the two closing handshakes and instead send an RST - Reset. This is bad practice. The client is called the “active” part as it sends the first SYN and almost always also the first FIN. The server is “passive” and responds to the first SYN with SYN, ACK (meaning that both flags are set). Likewise it will respond to the FIN with an ACK. Receiving the FIN will unblock any *recv()* calls that the application layer has, and if there is no outstanding reply, the server will send its FIN, which the client then answers with an ACK. Now comes some waiting, to assure that all retransmissions are gone, and any wild packets will have reached their 0 hop-count, and

finally the socket is closed. This long sequence is to assure that when the socket is reused there will be no interference from packets belonging to the old socket.

It is easy to see all the sockets running on your PC by using the "netstat" command which works in Windows as well as Linux. With the "-a" option it shows "all" - meaning that listening servers are included, while the "-b" option will show the programs or processes to which the socket belongs. Listning 4.7 shows an example - with a lot of lines cut out to save space. Note the many sockets in "TIME_WAIT".

```
01 C:\Users\kelk>netstat -a -b -p TCP
02
03 Active Connections
04
05   Proto  Local Address          Foreign Address        State
06   TCP    127.0.0.1:843          DK-W7-63FD6R1:0        LISTENING
07  [Dropbox.exe]
08   TCP    127.0.0.1:2559         DK-W7-63FD6R1:0        LISTENING
09  [daemonu.exe]
10   TCP    127.0.0.1:4370         DK-W7-63FD6R1:0        LISTENING
11  [SpotifyWebHelper.exe]
12   TCP    127.0.0.1:4664         DK-W7-63FD6R1:0        LISTENING
13  [GoogleDesktop.exe]
14   TCP    127.0.0.1:5354         DK-W7-63FD6R1:0        LISTENING
15  [mDNSResponder.exe]
16   TCP    127.0.0.1:5354         DK-W7-63FD6R1:49156    ESTABLISHED
17  [mDNSResponder.exe]
18   TCP    127.0.0.1:17600        DK-W7-63FD6R1:0        LISTENING
19  [Dropbox.exe]
20   TCP    127.0.0.1:49386        DK-W7-63FD6R1:49387    ESTABLISHED
21  [Dropbox.exe]
22   TCP    127.0.0.1:49387        DK-W7-63FD6R1:49386    ESTABLISHED
23  [Dropbox.exe]
24   TCP    192.168.0.195:2869     192.168.0.1:49704      TIME_WAIT
25   TCP    192.168.0.195:2869     192.168.0.1:49705      TIME_WAIT
26   TCP    192.168.0.195:2869     192.168.0.1:49706      TIME_WAIT
27   TCP    192.168.0.195:52628    192.168.0.193:1400     TIME_WAIT
28   TCP    192.168.0.195:52632    192.168.0.193:1400     TIME_WAIT
29   TCP    192.168.0.195:52640    192.168.0.194:1400     TIME_WAIT
```

**Listing 4.7:** Extracts from running Netstat




CLOSE and SEND are from application, Timeout is from implementation
All other events are received on socket from peer.

**Figure 4.9:** TCP State-Event Diagram

Socket handling in C can seem a little cumbersome, especially the server side and on Windows it requires a few lines more than on Linux. After a few repetitions it gets easier, and the client side is a lot simpler. Listings 4.8 and 4.9 together form the skeleton of a web-server on a windows PC. If you compile it and run it, you can fire up a browser and write: *http://localhost:4567* - this will give a response. "localhost" is the same as IP-address 127.0.0.1, which is the local PC. This code is not waiting at the usual port 80 but at port 4567, which we tell the web-browser by adding ":4567". Note that if you refresh, your portnumber will change as the operating system provides new ephemeral ports.

The following is a walk-through of the most interesting lines in Listing 4.8:

- 1-8. Includes, the socket-lib inclusion and global static variables.
- 12. Unlike Linux, Windows needs the WSAStartup to use sockets at all.
- 19. The basic *socket* for the server is created. AF_INET means "Address Family Internet" - there are also pipes and other kinds of sockets. "SOCK_STREAM" is TCP. UDP is called "SOCK_DGRAM" (datagram). "IPPROTO_TCP" is redundant, and if you write 0 it still works. The socket call returns an integer that is used as a handle in all future calls on the socket.
- 26-30. Here we define which IP-address and port the server will wait on. Typically IP is set to 0 - the "wildcard address" - partly because it will work even if you change IP address, partly because it will listen to all your NIC's. The port-number must be specific. The macro *"htons"* means host-to-network-short, and is used to convert 16-bit 'shorts' - like a port-number - from the hosts "endian-ness" to the networks (see Section 3.3). The *inet_addr()* function outputs data in network order, so we do not need "*htonl*" here. Note that the macros changes nothing if the CPU is big-endian.
- 32. We now handover the result of the above struggle to the socket in a *bind* call.
- 38. We tell the server to actually *listen* to this address and port. The parameter - here 5 - is the number of sockets we ask the OS to have ready on stock for incoming connections. Our listening server is not a complete 5-tupple as there is no remote IP-address or port. "netstat -a" would show the status as "LISTENING".
- 47. The *accept()* call is the real beauty of the server. This call blocks until a client actually does a SYN (connect). When this happens the *accept()* unblocks. The original listening server keeps listening for new customers, and the new socket that is returned from the *accept()* call is a full-blown 5-tupple. We have used one of the 5 "stock" sockets, so the OS will create a new in the background. "netstat" would show the status of this new socket as "ESTABLISHED".
- 58. A thread is spawned to handle the established socket, while the original thread loops back and blocks - waiting for the next customer.
- 65. If we break out of the loop, we need to clean-up.

Continuing with Listing 4.9:

- 9. We use *getpeername()* so that we can write out IP-address and port-no of the remote client.

- 13-17. The *recv()* call. If there is no data this blocks, otherwise it returns the number of bytes received so far. As this is a stream, there is absolutely no guarantee that we read chunks of data in the same size as they were sent - we need to accumulate “manually”. This is a side-effect in TCP that confuses many. If *recv* returns 0 it means that the client at the other end has sent it’s FIN, and our socket has ACK’ed it. It can also return a negative number, which will be an error-code that a serious program should handle.
- 19. We need to loop the *recv()* until the full request is received. As this “web-server” only supports HTTP-GET there is no body, so once we see two pairs of CR-LF we are good.
- 23. We create the body of our HTML first. This is practical as it gives us the length of the body.
- 29. Part of the above is to convert an IP-address from a 32-bit number to the well-known x.y.z.v format using *inet_ntoa()*.
- 33. The header is generated - including the length of the body. Note that it ends with two CR-LF pairs.
- 36. We send the header.
- 39. Finally, we send the body. This again demonstrates the stream-nature of TCP. The other side does not see that there were two sends. It **may** need to *recv()* 1, 2 or more times.
- 41. We do a *shutdown()* of our sending side. This generates a FIN to the remote.
- 42. We do a *closesocket()*.
- 45-46. As a small finesse we close the parent socket if we have received “quit” - e.g. “http://localhost:4567/quit”.

---

```
01 #include ”stdafx.h”
02 #include <winsock2.h>
03 #include <process.h>
04 #pragma comment(lib, ”Ws2_32.lib”)
05
06 static int threadCount = 0;
07 static SOCKET helloSock;
08 void myThread(void* thePtr);  // Forward declaration of thread method
09
10 int main(int argc, char* argv[])
11 {
12    WSADATA wsaData; int err;
13    if ((err = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
14    {
15       printf(”Error␣in␣Winsock”);
16       return err;
17    }
18
19    helloSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
20    if (helloSock == INVALID_SOCKET)
21    {
22       printf(”Invalid␣Socket␣-␣error:␣%d”, WSAGetLastError());
23       return 0;
24    }
25
26    sockaddr_in hello_in;
27    hello_in.sin_family = AF_INET;
28    hello_in.sin_addr.s_addr = inet_addr(”0.0.0.0”);  // wildcard IP
29    hello_in.sin_port = htons(4567);
30    memset(hello_in.sin_zero, 0, sizeof(hello_in.sin_zero));
31
32    if ((err = bind(helloSock, (SOCKADDR*)&hello_in, sizeof (hello_in))) != 0)
33    {
34       printf(”Error␣in␣bind”);
35       return err;
36    }
37
38    if ((err = listen(helloSock, 5)) != 0)
39    {
40       printf(”Error␣in␣listen”);
41       return err;
42    }
43    sockaddr_inremote;int    remote_len = sizeof(remote);
44
```

```
45    while (true)
46    {
47       SOCKET sock = accept(helloSock, (SOCKADDR*)&remote, &remote_len);
48       if (sock == INVALID_SOCKET)
49       {
50          printf("Invalid␣Socket␣-␣error:␣%d\n", WSAGetLastError() );
51          break;
52       }
53
54       printf("Connected␣to␣IP:␣%s,␣port:␣%d\n",
55              inet_ntoa(remote.sin_addr), remote.sin_port);
56
57       threadCount++;
58       _beginthread(myThread, 0, (void *)sock);
59    }
60
61    while (threadCount)
62       Sleep(1000);
63
64    printf("End␣of␣the␣line\n");
65    WSACleanup();
66    return 0;
67 }
```

**Listing 4.8:** Windows server-socket LISTEN

```
01 void myThread(void* theSock)
02 {
03    SOCKET sock = (SOCKET)theSock;
04    char rcv[1000]; rcv[0] = '\0'; // Allocate buffer
05    int offset = 0; int got;
06
07    sockaddr_inremote;
08    intremote_len = sizeof(remote);
09    getpeername(sock, (SOCKADDR*)&remote, &remote_len);
10
11    do // Build whole message if split in stream
12    {  // 0: Untimely FIN received, <0: Error
13       if ((got = recv(sock, &rcv[offset],
14                       sizeof(rcv)-1-offset, 0)) <= 0)
15          break;
16       offset += got;
17       rcv[offset] = '\0'; // Terminate the string
18       printf("Total␣String:␣%s\n", rcv);
19    } while (!strstr(rcv, "\r\n\r\n")); // No body in GET
20    // Create a HTML Message
21    char msg[10000];
22
23    int msglen = _snprintf_s(msg, sizeof(msg)-1, sizeof(msg),
24       "<html><title>ElkHome</title><body>"
25       "<h1>Welcome␣to␣Klaus␣Elk's␣Server</h1>"
26       "<h2>You␣are:␣IP:␣%s,␣port:␣%d␣-␣%d'th␣thread,␣and␣you␣sent:"
27       "<p>%s␣</p></h2>"
28       "</body></html>",
29       inet_ntoa(remote.sin_addr), remote.sin_port, threadCount, rcv);
30
31    // Create a new header and send it before the message
32    char header[1000];
33    int headerlen = _snprintf_s(header, sizeof(header)-1, sizeof(header),
34       "HTTP/1.1␣200␣OK\r\nContent-Length:␣%d\r\nContent-Type:␣"
35       "text/html\r\n\r\n", msglen);
36    send(sock, header, headerlen, 0);
37
38    // Now send the message
39    send(sock, msg, msglen, 0);
40
41    shutdown(sock, SD_SEND);
42    closesocket(sock);
43    threadCount—;
44
45    if (strstr(rcv, "quit"))
46       closesocket(helloSock);
47 }
```

**Listing 4.9:** Windows server-socket ESTABLISHED

Figure 4.10 shows the output from the "webserver". Note the very last header-line written: "Connection: Keep-Alive". This is the web-browsers part of a negotiation, and as the web-server reply does not contain this header it does not happen. If both parties agree on "Keep-Alive" we have a so-called "pipelining" where multiple sequential HTTP-requests and responses can happen on the same open TCP socket. The default behavior is to close the socket after every Request-Response pair, but this creates a lot of overhead. Much faster to stay on the same socket. This gives the programmer some work in finding the "borders" between Requests in one direction and Responses in the other.

A browser can also open parallel sockets. When you access a homepage your browser retrieves the “basic” HTML-page. Embedded in this may be a lot of other elements, pictures, logos, banners etc. The first page contains the URL to all these. This means that the basic page is retrieved alone in the first round-trip, but in the next the browser will retrieve 5 figures if it has 5 parallel sockets and there are at least 5 figures. Since the URL’s are unique the embedded pages may be delivered by different servers in a server-park. This is the beauty of HTTP; no state - each element can be retrieved “as is”.

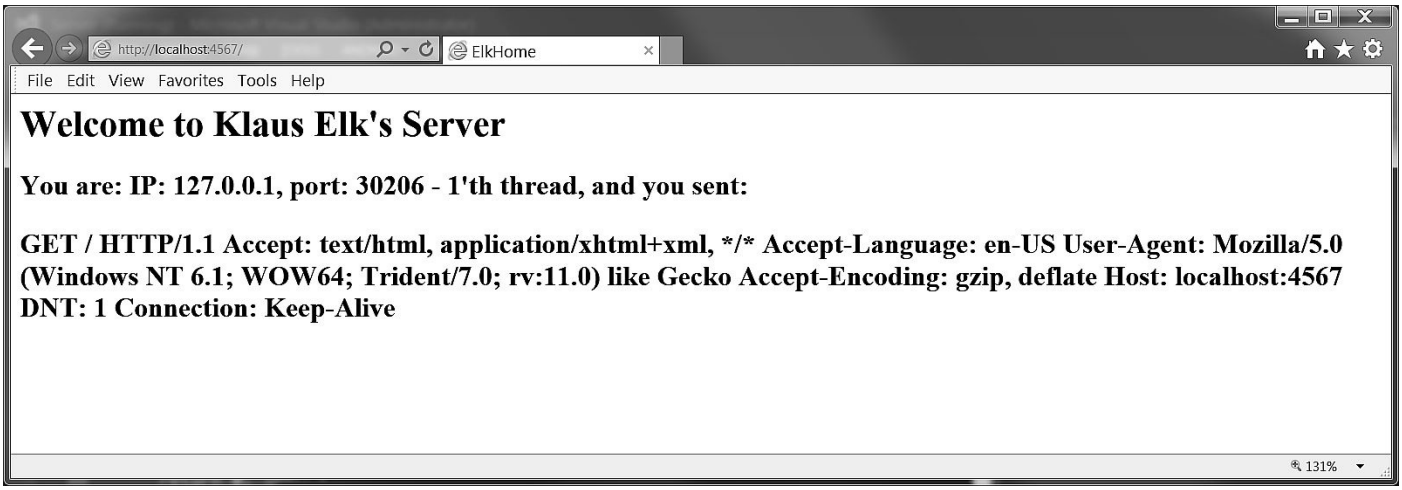



**Figure 4.10:** Webbrowser with response from simple server

## 4.13 IP Fragmentation

In version 4 of the IP layer it is responsible for “adapting” the size of the segments that come from “above” to what the underlying layer can handle. This is a little strange; we have already seen how TCP breaks up huge messages from the application layer to segments that will fit into Ethernet frames further down, so how come there is this adapter function?

Normally the TCP MSS - Maximum Segment Size - is created so that even after adding the necessary headers it can still fit within an Ethernet frame. But the packet is routed - sometimes over many different links, and one of these may not allow frames so big[3] . The router that “looks into” such a link is therefore responsible for - in its IPv4 layer - “fragmenting” the packet. The IP-packet is thus broken into smaller packets. They will stay fragmented for the rest of the journey until they get to the IPv4 layer of the receiving host. Here they are collected, and only when all fragments are together is the packet delivered to the layer above.

This is easily demonstrated with the help of the “ping” command. We tend to think of UDP and TCP as the only clients to the IP-layer - but there is also ICMP. Listing 4.10 shows a normal ping command with the “-l” parameter that decides the length of the data sent.

```
01 C:\Users\kelk>ping -l 5000 192.168.0.1
02
03 Pinging 192.168.0.1 with 5000 bytes of data:
04 Reply from 192.168.0.1: bytes=5000 time=12ms TTL=64
05 Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64
06 Reply from 192.168.0.1: bytes=5000 time=7ms TTL=64
07 Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64
```

```
08
09  Ping statistics for 192.168.0.1:
10      Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
11  Approximate round trip times in milli-seconds:
12      Minimum = 6ms, Maximum = 12ms, Average = 7ms
```

**Listing 4.10:** ICMP ping with 5000 bytes

Figure 4.11 shows a Wireshark capture of this:

- The column called “Length” is the payload in the highest level protocol dissected (shown in the “Protocol” column). The 5000 bytes are sent in totally four IPv4 packets. The first three are shown as such in frames 13,14 and 15 each containing 1480 bytes, but with frame 16 the protocol-stack has the full ICMP packet and we see the data-length of it in the first Length field.
- The “Total Length” column is in all cases the payload in the IP layer. If we add the numbers, we get 3*1500+588 = 5088. This is the 5000 databytes plus 4 IP-headers of 20 bytes each plus one ICMP header of 8 bytes. The last line in the figure shows the four payload lengths for the IP-layer - adding up to 5008 - the 5000 databytes and the ICMP header.
- The “Fragment Offset” is, as the name says, the offset of the first byte in the original non-fragmented packet. Wireshark is helping us here - the offset in the binary header is shifted three bits to the right to save bits but Wireshark shifts them back for us in the dissection. In order to make this work fragmentation occurs at 8-byte boundaries.
- The info field shows us an ID generated by the transmitting IPv4 layer that is the same for all fragments from the same package and incremented for the next.

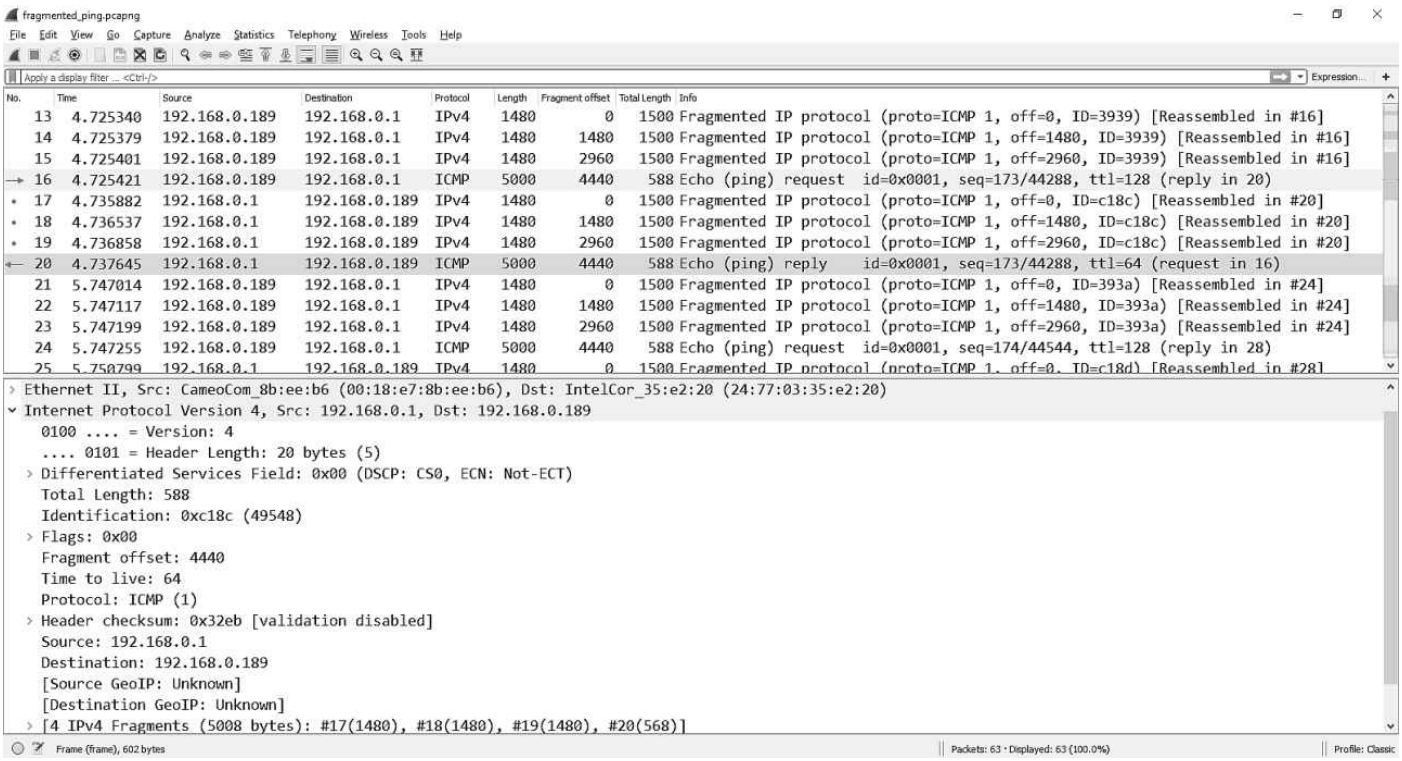



**Figure 4.11:** Fragmented ICMP ping with 5000 bytes of data

In IPv6 fragmentation has been removed. To simplify routers and improve performance, the router that “looks into” a path with a smaller maximum frame-size than incoming frames now sends an error back to the originator so that the problem can be

dealt with at the origin.

The reason space has been devoted to fragmentation is that if you see this happening, you should ask why and probably make sure it does not happen in the future.

## 4.14 Introducing IPv6 addresses

The main reason for introducing IPv6 was the fact that the IPv4 address space was exhausted - partly because of waste in the old class system. Moving from 32 bits to 128 bits certainly cures that. While the good people on the standard committees were in the machine room, they also fixed a number of minor irritation points in IPv4. Quite opposite the old telephone system, the internet is supposed to be based on intelligent equipment at either end of conversations (the hosts), and “stupid” - and therefore stable - equipment inside the actual net. The fact that IPv4 has dynamic header-size, means that routers need to be more complex than otherwise necessary. Similarly, the IPv4 checksum needs to be recalculated in every router as the TTL (hopcount) is decremented. This is not optimal. IPv6 still has the hopcount - now even called so - but the checksum is gone, and the header-size is static. There is still a much better checksum in the Ethernet, and the 1’s complement checksum is still found in UDP and TCP.

The following table describes how IPv4 and IPv6 Notation differs.

| Concept | IPv4 | IPv6 |
|---|---|---|
| Bit-width | 32 | 128 |
| Groupsize | 1 Byte | 2 Bytes |
| Notation | Decimal | Hexadecimal |
| Separator | . | : |
| Skip front zero | Yes | Yes |
| Skip zero groups | No | Yes |
| Mixed Notation | No | Yes |

As the above table says, IPv6 addresses are grouped in 16-bit chunks, written as 4-digit hexadecimal numbers. We are allowed to skip zeroes in front of each group and we can also skip **one** consecutive streak of zeros. Here are some examples:

`fc00:a903:0890:89ab:0789:0076:1891:0123` can be written as:
`fc00:a903:890:89ab:789:76:1891:123`

`fc00:0000:0000:0000:0000:98ab:9812:0b02` can be written as:
`fc00::98ab:9812:b02`

`::102.56.23.67` is an IPv4 address used in an IPv6 program.

The next table shows the functional differences between the two IP versions.

| Function | IPv4 | IPv6 |
|---|---|---|
| Checksum | Yes | No |
| Variable Header Size | Yes | No |

| | | |
|---|---|---|
| Fragmenting | Yes | No |
| Hopcount Name | TTL | Hopcount |
| Flow Label | No | Yes |
| Scope & Zone ID | No | Yes |

IPv6 has introduced two new fields:

1. **Flow Label**
   RFC 6437 says “From the viewpoint of the network layer, a flow is a sequence of packets sent from a particular source to a particular unicast, anycast, or multicast destination that a node desires to label as a flow”. This is e.g. used in Link Aggregation[4] . The std says that all packets in a given 5-tupple (TCP, Source- and Dest IP-address and port-numbers) SHOULD be assigned the same 20-bit unique number - preferably a hash of the 5-tuple. Traditionally the 5-tuple has been used directly by network devices, but as some of these fields may be encrypted the Flow Label is a help. If NOT used it MUST be set to 0 - which is what is done in this book.
2. **Scope ID**
   Defined in RFC 4007 and RFC 6874. RFC 4007 allows for a scope ID for link-local addresses that can be the device, the subnet or global. RFC 6874 defines a special case where a “zone ID” is used to help the stack use the right interface. Apparently this special case is the only one used yet. It is not a field in the IPv6-header but a part of the address-string in link-local scenarios written after a “%”, that helps the stack use the right interface. On Windows this is the interface number (seen with “netstat -nr”), on Linux it could be e.g. “eth0”.

The following table gives some examples on addresses in the two systems:

| Address | IPv4 | IPv6 |
|---|---|---|
| Localhost | 127.0.0.1 | ::1 |
| Link-local | 169.254.0.0/16 | fe80::/64 |
| Unspecified | 0.0.0.0 | ::0 or :: |
| Private | 192.168.0.0/16 etc | fc00::/7 |

## 4.15 TCP sockets on IPv6 under Linux

Now that we have looked at Windows, let’s see something similar on Linux. In the following we will examine a small test program that can run as server or client on different machines. In this case it is run on a single Linux PC. This time we will try IPv6 - it might as well have been the other way around.

```
01 ….main…
02     int sock_comm;
03     int sock1 = do_socket();
04
05     if (is_client)
06     {
07        do_connect(sock1, ip, port);
08        sock_comm = sock1;
09     }
10     else // server
```

```
11    {
12       do_bind(sock1, 0, port);
13       do_listen(sock1);
14       sock_comm = do_accept(sock1);
15    }
16
17    TestNames(sock_comm);
18    gettimeofday(&start, NULL);
```

**Listing 4.11:** Linux Test Main

The basic work flow is seen in Listing 4.11. The *main()* shows the overall flow in a client versus a server on a single page. In both cases we start with a *socket()* call. On a client we then need a *connect()*, and are then in business for *recv()* and *send()*. On a server we still need the *bind()*, *listen()* and *accept()* before we can transfer data. This is exactly like the windows server, and the client scenarios are also equal. The functions called in Listing 4.11 are shown in 4.12 and 4.13. The error-handling and other printout-code is removed as focus is on the IPv6 socket handling. Even though this is Linux and IPv6, it looks a lot like the Windows version using IPv4 we saw with the web-server. The most notable difference between the OS'es is that on Linux we do not need WSAStartup() or anything similar. This is nice, but not really a big deal.

Much more interesting in the Linux version is very subtle and only shows itself in the *close()* call on the socket. On Windows this is called *closesocket()*. Add to this that on Linux you don't need to use *send()* and *recv()*. It is perfectly all right to instead use *write()* and *read()*. That means that you can open a socket and pass it on as a file handle to **any** process that uses files (e.g. with *fprintf()*). That certainly can make life a lot easier for an application programmer.

```
01 int do_connect(int socket, const char *ip, const bool port)
02 {
03 …
04    struct sockaddr_in6 dest_addr;       // will hold the destination addr
05    dest_addr.sin6_family = AF_INET6;    // host byte order
06    dest_addr.sin6_port = htons(port);   // short, network byte order
07
08    int err = inet_pton(AF_INET6, ip, &dest_addr.sin6_addr);
09    …
10    int retval = connect(socket, (struct sockaddr *)&dest_addr,
11                         sizeof(dest_addr));
12    …
13 }
14
15 int do_bind(int socket, const char *ip, int port)
16 {
17    struct sockaddr_in6 src_addr;        // will hold the source addr
18    src_addr.sin6_family = AF_INET6;     // host byte order
19    src_addr.sin6_port = htons(port);    // short, network byte order
20    if (ip)
21    {
22       int err = inet_pton(AF_INET6, ip, &src_addr.sin6_addr);
23       {
24          printf("Illegal␣address.\n");
25          exit(err);
26       }
27    }
28    else
29       src_addr.sin6_addr = in6addr_any;
30
31    int retval = bind(socket, (struct sockaddr *)&src_addr,
32                      sizeof(src_addr));
33    …
34 }
35
36 int do_listen(int socket)
37 {
38    int retval = listen(socket, 5); // Backlog of 5 sockets
39     …
40 }
41
42 int do_socket()
43 {
44    int sock = socket(AF_INET6, SOCK_STREAM, 0);
45    …
46 }
47
48 int do_close(int socket)
```

```
49 {
50    …
51    int retval = close(socket);
52    …
53 }
```

**Listing 4.12:** Linux Sockets part I

Code-wise there is a number of differences between IPv4 and IPv6 - and almost all relates to the functions that are not part of the actual socket-handling:

- Instead of “AF_INET” we use “AF_INET6”.
- The address structures used end with “_in6” instead of “_in”.
- *inet_ntoa()* and *inet_aton()* are replaced by respectively *inet_ntop()* and *inet_pton()*. The new functions work with both IPv4 and IPv6 addresses.
- We still use the *ntohs()* and *htons()* macros, as they are used on the 16-bit ports in TCP which is unchanged. However we do not use *ntohl()* and *htonl()* on IP-addresses as they are to big for 32-bit words.
- There are new macros as *in6addr_any*.

*It can be a pain to get the address structures right in the start. The most tricky parameter is the one inside a sizeof() - with the size of the structure that is normally the last parameter in several calls - e.g. connect() and bind(). The compiler will not complain as long as you write something that evaluates to an integer. But it certainly makes a difference what is written. I find that I make less mistakes if I use the variable-name in the sizeof() instead of the type, since I have typically just populated the variable and the compiler will complain here if I try to put e.g. in6addr_any inside a 32-bit IP-address.*

```
01 int do_accept(int socketLocal)
02 {
03    …
04    struct sockaddr_in6 remote;
05    socklen_t adr_len = sizeof(remote);
06
07    int retval = accept(socketLocal,(struct sockaddr *)&remote,&adr_len);
08    …
09    char ipstr[100];
10    inet_ntop(AF_INET6, (void*) &remote.sin6_addr, ipstr, sizeof(ipstr));
11    printf(”Got␣accept␣on␣socket␣%d␣with:␣%s␣port␣%d␣-␣new␣socket␣%d\n”,
12           socketLocal, ipstr, ntohs(remote.sin6_port), retval);
13 }
14
15 int do_send(int socket, int bytes)
16 {
17    …
18    if ((err = send(socket, numbers, bytes, 0)) < 0)
19    …
20    return 0;
21 }
22
23 int do_recv(int socket, int bytes)
24 {
25    int received = recv(socket, nzbuf,bytes-total, 0);
26    …
27 }
28
29 void TestNames(int socketLocal)
30 {
31    struct sockaddr_in6 sock_addr;
32    socklen_t adr_len = sizeof(sock_addr);
33    char ipstr[100];
34
35    getpeername(socketLocal,(struct sockaddr *) &sock_addr, &adr_len);
36
37    inet_ntop(AF_INET6, (void*) &sock_addr.sin6_addr, ipstr,
38              sizeof(ipstr));
39
40    printf(”getpeername:␣IP=␣%s,␣port=␣%d,␣adr_len=␣%d␣\n”, ipstr
41          , ntohs(sock_addr.sin6_port), adr_len);
42
43 }
```

**Listing 4.13:** Linux Sockets part II

Listing 4.14 shows the execution. First the server is created, this is the default. It is told to receive 10000 bytes. It creates the socket, binds it to the default port (12000) and blocks in the *accept()* call. The "&" makes it run in the background. Now the client is started with the "-c" option and asked to send 10000 bytes. It gets an ephemeral port number from Linux - 43171. This unblocks the waiting server and we are done. The speed given is affected by writing it all to the screen.

Figure 4.12 shows the same scenario from a Wireshark point-of-view. We recognize the port-numbers as well as the "::1" localhost IPv6 address. You may ask how the marked frame 4 can send 10000+ bytes as we normally have a maximum of 1460 bytes (making room for TCP and IP headers). The explanation is that the normal Maximum Segment Size stems from the Ethernet which we are skipping here. Interestingly Wireshark still shows Ethernet in the stack - but with zero-addresses in both ends.

Note that it is easy to capture a localhost communication on Linux, as the "lo" interface is readily available. Not so on Windows. To do a capture on the localhost on Windows you need to change your route table to send all frames to the gateway and back again. Now you see them twice, and timing is quite obscured. And don't forget to reset the route table. An alternative is to install the Windows Loopback Adapter.

```
01 kelk@debianBK:~/workspace/C/sockserver6$ ./socktest -r –bytes=10000 &
02 [1] 1555
03 kelk@debianBK:~/workspace/C/sockserver6$ Will now open a socket
04 Binding to ip: localhost, port: 12000 on socket: 3
05 Listen to socket 3
06 Accept on socket 3
07
08 kelk@debianBK:~/workspace/C/sockserver6$ ./socktest -c -s –bytes=10000
09 Will now open a socket
10 Connecting to ip: ::1, port: 12000 on socket: 3
11 Got accept on socket 3 with: ::1 port 43171 - new socket 4
12 getpeername: IP= ::1, port= 43171, adr_len= 28
13 Loop     0
14 Plan to receive 10000 bytes on socket 4
15 getpeername: IP= ::1, port= 12000, adr_len= 28
16 Loop     0
17 Plan to send 10000 bytes on socket 3
18 Received 10000 bytes at address 0x7ffcb3b34e30:
19    0    1    2    3    4    5    6    7    8    9
20 ……
21   7c8  7c9  7ca  7cb  7cc  7cd  7ce  7cf
22 Totally transmitted: 10000 in 1 loops in 0.001910 seconds =    41.9 Mb/s
23 ….
24 [1]+  Done                    ./socktest -r –bytes=10000
25 kelk@debianBK:~/workspace/C/sockserver6$
```

**Listing 4.14:** Running Linux Test

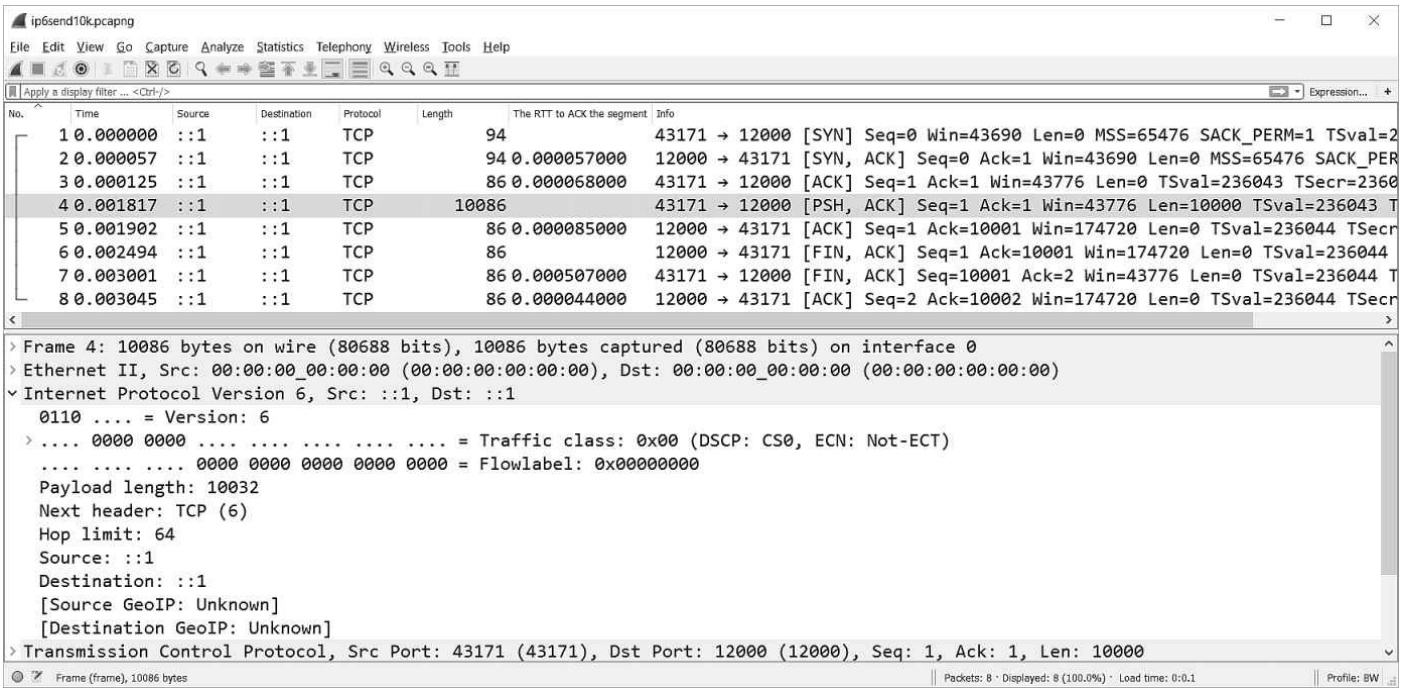



**Figure 4.12:** TCP on Linux localhost with IPv6

## 4.16 Data Transmission

Until now our focus on the TCP-related stuff has been about getting the transmission going. Now is the time to look at the actual transmissions. When looking at TCP it makes little difference whether we have IPv4 or IPv6 beneath it - there is no TCPv6. Figure 4.13 shows a Wireshark capture of a network device (10.116.120.155) transmitting data at high speed (for an embedded device) to a PC (10.116.121.100). Note that all frames with data have length 1460. This fits nicely with an Ethernet MTU - Maximum Transmission Unit - or Payload - of 1500 bytes minus 20 bytes per each of the TCP and IP headers. The total Ethernet packages here is 1514 according to Wireshark. The 14 bytes difference is the Source and Destination MAC addresses, each 6 bytes, in the Ethernet frame plus the “Type” Field that says “0x0800” (not visible on the figure) for IPv4.

This is a pattern that is seen again and again. An incoming frame is received at the lowest layer, and this layer needs to look at a given field to know where to deliver it. Here Ethernet delivers it to IP, which delivers to TCP, which delivers to the application process with the help of the port number.

Interestingly the PC’s Sequence Number is constantly “1” while the device is racing towards higher numbers. This means that the PC hasn’t sent anything since its SYN-flag. This is a one-sided conversation. Outside the figure at the bottom line in the “dissection” of the selected frame there is a line that says “Validation Disabled” for the checksum. If checksum validation is enabled Wireshark will, mistakenly, flag all outgoing frames as having a bad checksum, because modern systems no longer calculate the checksum in the stack. They leave it to the Network Interface Card, and as Wireshark is between these, it will see wrong checksums. It makes no sense to check in-going frames either, as the NIC only lets those frames through that have correct checksum.

If we select one of the frames with data (as in the figure), then select “Statistics” in the

top-menu, then “TCP Stream Graphs” and finally “Time Sequence (Stevens)” we get a great view of how data is sent over time, see Figure 4.14.

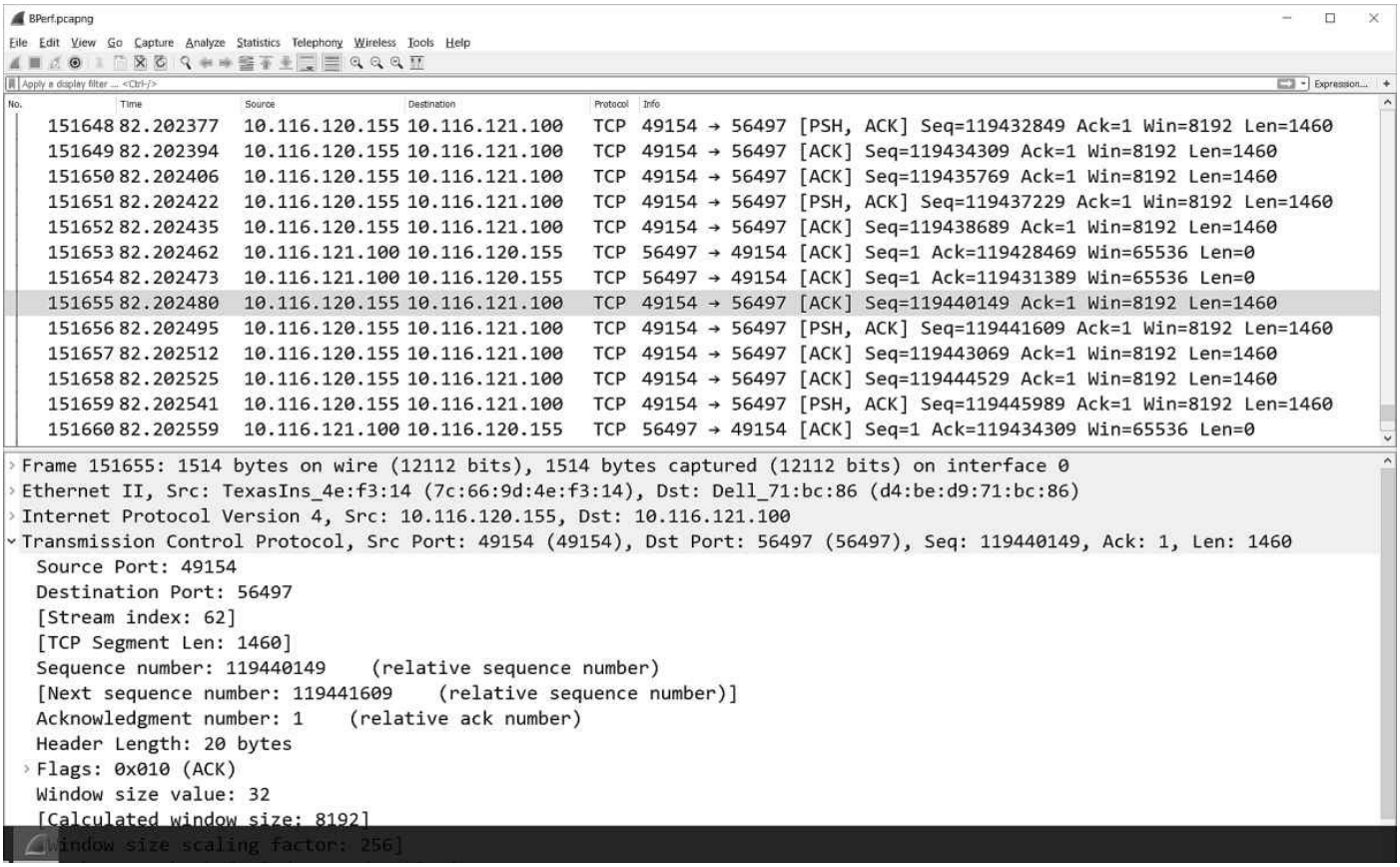



**Figure 4.13:** Transmitting many bytes

The Stevens graph, named after the author of the famous network books, shows the Sequence Numbers over time. This is the same as the accumulated number of bytes sent. Seldom do you see a line this straight. By reading the point in e.g. X = 20s and finding that this corresponds to 80 MBytes, we easily calculate the data rate to be 80 MByte/20s = 4 MByte/s or 32 Mbps. The biggest challenge is counting the number of zeros. Inside the same window, you can also select e.g. Roundtrip-Time. Another menu-selection can give an I/O-graph, but in this case they are as boring as Figure 4.14.

Figure 4.15 shows a similar measurement from some years ago. If you click with a mouse on the flat part of the graph, Wireshark goes to the frames in question. Here you should e.g. look for retransmissions, as these will occupy bandwidth without progressing sequence numbers. You can also try “Analyze” in the main menu, and then “Expert Information”. This is shown in Figure 4.16. Here the PC sends up to 20 Duplicate ACK’s for the same frame, effectively repeating the phrase: “The last Sequence Number I was expecting to see from you was xx and you are now much further ahead, please go back and start from xx”. If you click in this window you will be at the guilty frame in the main window. Since Wireshark hasn’t lost any packages, and reports **suspected** retransmissions, it hasn’t seen these packages either. Apparently the device is sending packages that nobody sees. This was exactly what was happening. We had a case of “crosstalk” between an oscillator and the PHY in this prototype. We will get back to this in Section 4.17.

Note that when you are capturing transmissions like this, and not just random frames,

it is a good idea to turn off the live update of the screen. Wireshark is very good at capturing, but the screen cannot always keep up. Very often you do not need to capture the actual data - just the headers - so you can ask Wireshark to only capture e.g. 60 bytes per frame. This will allow the PC to “punch above its weight”.

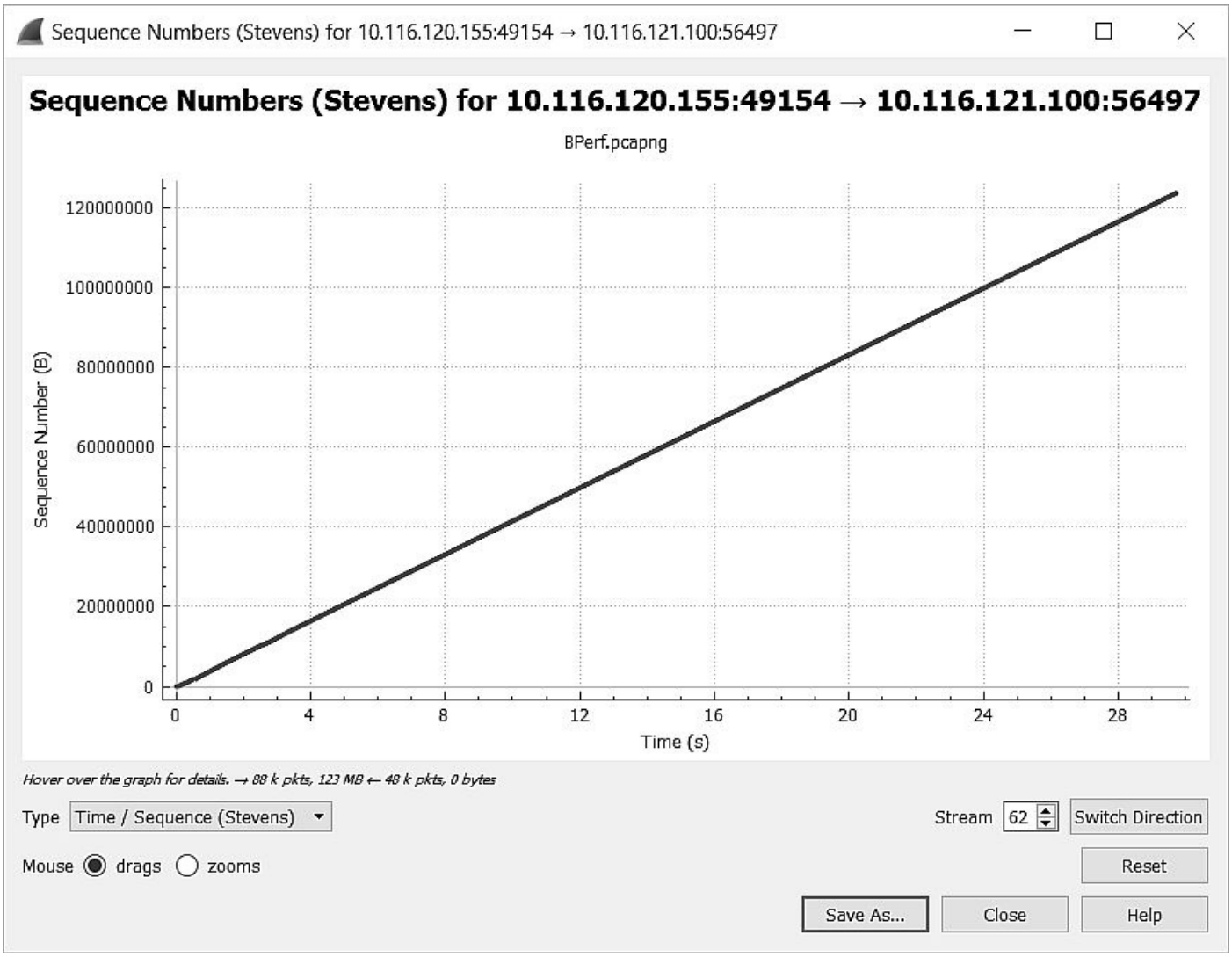



**Figure 4.14:** Nice Stevens Graph of Sequence Number versus time

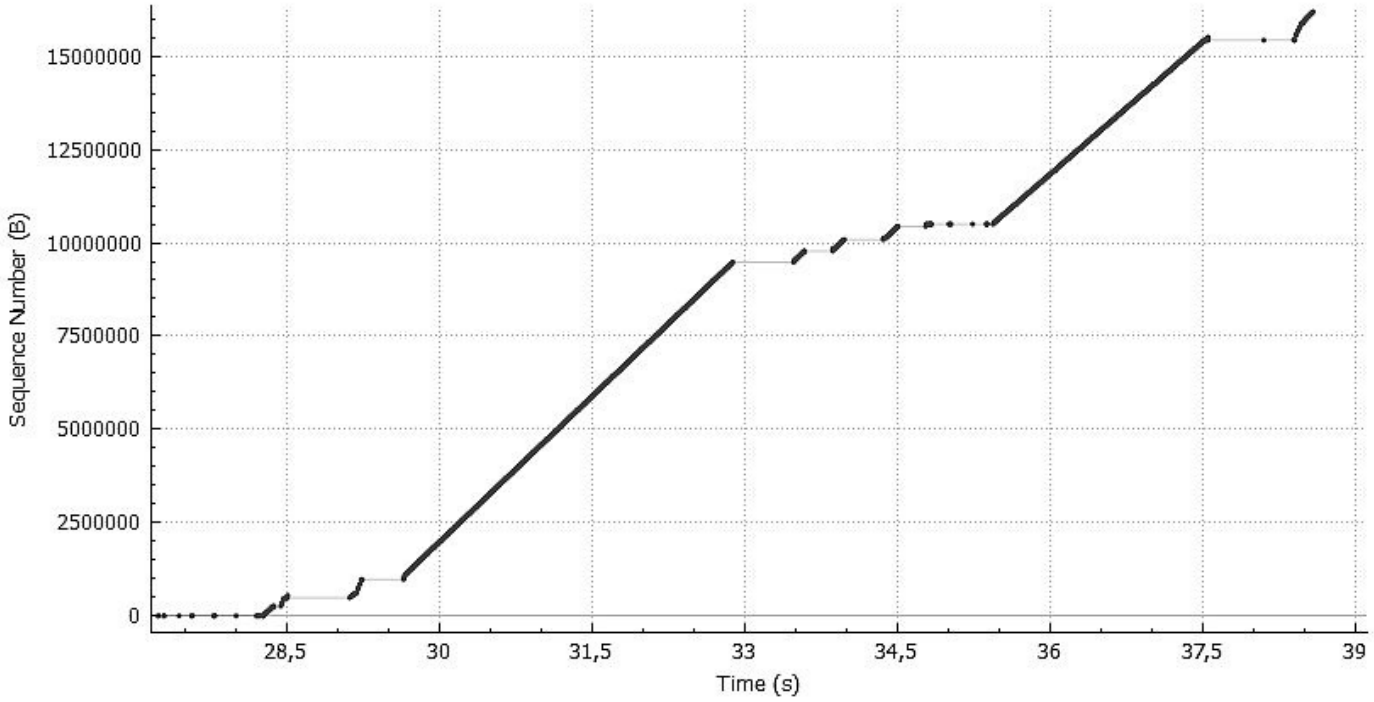



**Figure 4.15:** Stevens Graph with many retransmissions

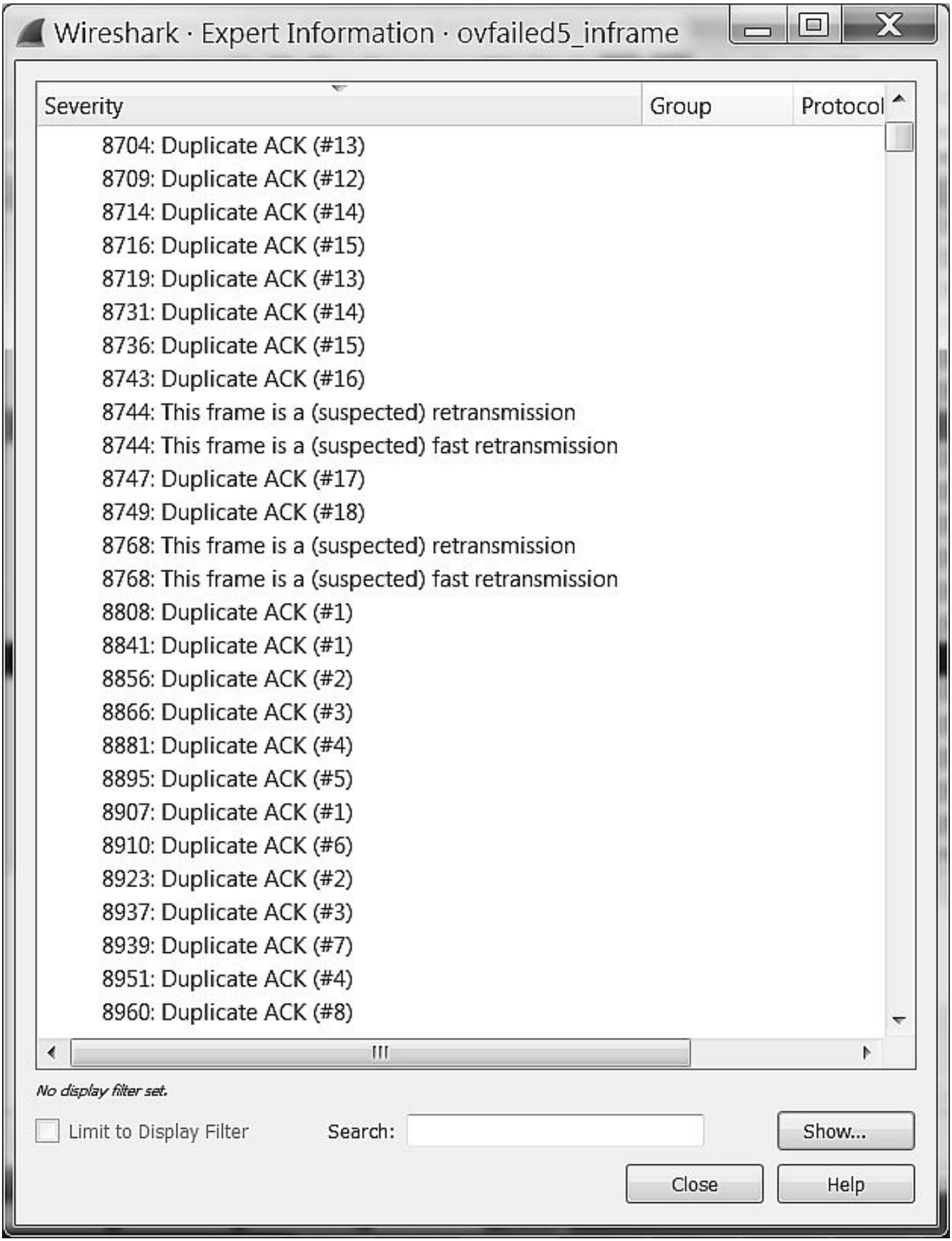



**Figure 4.16:** Expert Info with lost data

## 4.17 Physical Layer Problems

In Section 4.16 we saw that a problem on the physical layer showed symptoms in a Wireshark capture, but wasn't really nailed down. Hardware Engineers apply something called an Eye-Diagram in these cases. A good sample is shown in Figure 4.17. Basically a test program is run that sends all "symbols" in many sequences. Various parameters are measured, but in fact the diagram is directly usable as is.

There is however a huge threshold for an embedded programmer to fire up a test like this - even if the equipment is available, and very often it is not.

*My colleagues and I saw the problem with the physical layer in a prototype that we could split open, but what if you want to measure on a closed box? And what if a customer far away has a problem? I have had cases where I could ask a customer to do a Wireshark capture and send it to me, but I have yet to meet the guy who can convince his customer that he should do an Eye-Diagram. There should be a better way to diagnose physical layer problems. Once diagnosed it is a good idea to roll out the cannons to understand and fix the problem.*

The main reason you don't see these things in Wireshark is that a corrupt Ethernet frame also has a bad checksum. As stated earlier, modern Network Interface Cards - NIC's - have built-in checksum calculations. But that was the IP and TCP checksums - even the oldest NIC's have hardware-check of the Ethernet checksum (which BTW is a CRC which is much more advanced than a checksum). On the Ethernet level there are no retransmissions. If a "bad" frame is received it is simply thrown away. This normally happens so rarely that a TCP retransmit is a fine and simple solution. If you are using UDP it's just bad luck, and therefore up to your application to handle.

Even though there are no actions taken in the protocol-stack, there might be an error-counter, and normally there is. In reality your faithful NIC on the PC, and very often also in an embedded system is counting CRC-errors. All you need to do is to read them, but how?

This is where SNMP - Simple Network Management Protocol - comes in. Most Operating Systems actually supports this. So all you need to do is to download an SNMP-client, fire it up, key in your PC or your device IP-address and you are in business. See Section 8.13. After our experiences with the faulty prototype we took this one step further and built in the relevant SNMP requests, in our diagnostics tool. That allows us to ask a customer with a problem to run the test and mail the results.

Inexperienced software developers often jump to the conclusion that the bug is not in their software - it must be a hardware problem. More senior developers have experienced so many software bugs that they always look for the bug in software first - and then they check and check again. But one place where you actually should be prepared for hardware problems is the PHY - the device that sits between the MAC and the magnetics (and the connector) and is responsible for converting the more or less [5] analog waveform on the wire to a bit pattern. Many types of transients can occur at the wire. They will go through the connector and the magnetics to the PHY, where they may kill the signal - and in some scenarios even kill the PHY. So make sure that the correct protection circuitry is in place.

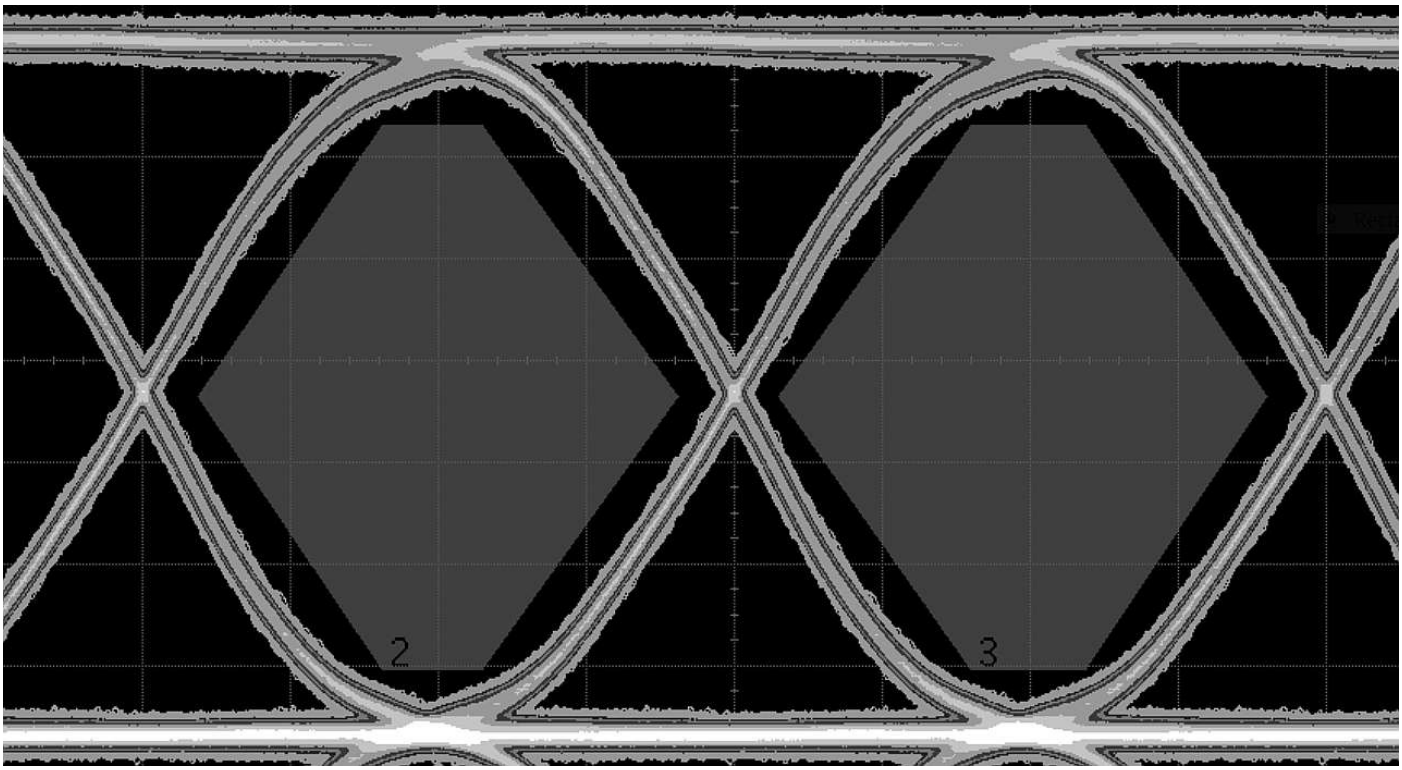



**Figure 4.17:** A typical good Eye Diagram

## 4.18 UDP Sockets

We have been using the terms broadcast - aka anycast - and multicast without really going into what the meaning is:

| -cast | Meaning | Protocols |
|---|---|---|
| Unicast | 1:1 | TCP, UDP |
| Multicast | 1:Members | UDP |
| Broadcast | 1:All | UDP |

A unicast is sent from one process on one host to another process on another (or the same) host. A broadcast is sent from one process on one host to all other hosts - on the same subnet. A broadcast will never get through a router. A multicast looks pretty much like a broadcast, but as stated it is only for members. What does that mean? A router can typically be configured to simply “flood” multicasts coming in on one port to all other ports or it can support group-membership and spanning-trees and all sorts of advanced stuff that leads to routers only forwarding multicasts to ports that somewhere further out has a **subscriber** for the given address. A multicast-address is something special. In IPv4 it is all addresses in the “Class D” range 224.0.0.0-239.255.255.255. Take for instance PTP - Precision Time Protocol. It uses the IP-addresses 224.0.1.129, 224.0.1.130, 224.0.1.131 and 224.0.1.132 as well as 224.0.0.107. Anyone who wants these messages must listen to one or more of these IP-addresses, as well as their own. But what about ARP if several hosts have the same IP-address? ARP is not used here. Instead the MAC must listen to some programmed, specific addresses on top of the one it is “hardcoded” for. The MAC addresses are simply 01:00:5e:00:00:00 OR’ed with the lower 23-bits of the

Multicast IP-address. This means that MAC hardware has extra programmable “filter-addresses” apart from the static one (which might be programmable in e.g. EEPROM).

It seems UDP can do everything - why use TCP at all?

| Feature | TCP | UDP |
|---|---|---|
| Flow | Stream | Datagram |
| Retransmit | Yes | No |
| Guaranteed Order | Yes | No |
| Social Throttle | Yes | No |
| Maximum Size | No (stream) | (536) bytes |
| RTT 1’st Byte | 1.5 | 0.5 |

We have seen the stream nature of TCP in action. It is a two-edged sword that gives programmers headaches trying to re-find the “borders” in what they sent, but it is also an extremely powerful concept, as retransmissions can be much more advanced than simply retransmitting frames. The retransmit, the guarantee that every byte makes it all the way (if in any way possible), that it comes only once, and in the right order compared to the other bytes are all important features in serious applications as web or file-transfer. “Social Throttle” relates to TCP’s “digestion” handling, where all sockets will lower their speed in case of network problems. UDP has none of these advanced concepts. It is merely a thin skin on top of IP-packets. DNS is a good example on using UDP (see Section 4.9). These packets are rather small and can be resend by the application. If there is digestion-trouble due to a DNS-problem, it makes sense that they are faster - non-throttled. When it comes to the maximum size, an UDP-packet can be 64k - but it is recommended not to allow it to “IP-fragment” - see Section 4.13. A size less than 536 bytes will guarantee this. Finally - how long does it take before the first byte of a message is received? With TCP we need the three-way handshake. Data may be embedded in the last of the three “ways”. This corresponds to 1.5 Roundtrips, as a full roundtrip is “there and back again”. UDP only takes a single one-way datagram, corresponding to 0.5 RTT. TCP actually has to do the closing dance afterwards. This is overhead but not latency. Note that if the TCP-socket is kept alive, the next message only has a latency of 0.5 RTT - as good as UDP.

## 4.19 A case of UDP on IPv6

```
01 void sendit()
02 {
03    int sock = socket(AF_INET6, SOCK_DGRAM, 0);
04    if (sock <= 0)
05    {
06       printf(”Opening␣socket␣gave␣error:␣%d\n”,sock);
07       exit(sock);
08    }
09
10    int err;
11    char *mystring = ”The␣center␣of␣the␣storm\n”; // Message to send
12
13    struct sockaddr_in6 dst_addr;
14    memset(&dst_addr, 0, sizeof(dst_addr)); // Clears scope and flow
15    dst_addr.sin6_family = AF_INET6;
16    dst_addr.sin6_port = htons(2000);       // TCP is unchanged in ip6
17
18    if ((err=inet_pton(AF_INET6, ”::1”, &dst_addr.sin6_addr)) == 0)
19    {
20       printf(”Illegal␣address.\n”);
21       exit(err);
```

```
22    }
23
24    if ((err = sendto(sock, mystring, strlen(mystring)+1, 0,
25    (struct sockaddr *)&dst_addr, sizeof(dst_addr))) < 0)
26    {
27       printf("Could not send. Error: %d\n", err);
28       exit(err);
29    }
30 }
```

**Listing 4.15:** IPv6 UDP Send on Linux

Listing 4.15 shows a UDP transmission using IPv6. As we saw with TCP on IPv6: Instead of "AF_INET" "AF_INET6" is used. Some sources use "PF_INET6", it makes no difference as one is defined as the other. Since we are using UDP, the socket is opened with "SOCK_DGRAM" as the second parameter. Also here the IPv4 "sockaddr_in" structure has been replaced by its cousin with "in" replaced by "in6" and the same rule applies to the members of the *struct*. We still use *htons* on the port number, as this is UDP and the Transport Layer is unchanged. In line 18 we see the IPv6 address "::1". This is the "localhost" address, where in IP4 we normally use "127.0.0.1". We also see the function *inet_pton()* which is the recommended function to use as it supports IPv4 as well as IPv6. Finally in line 24 we see the UDP "*sendto()*" which is unchanged from IPv4, but called with "in6" *structs*. In TCP we use the command "*send()*" without a "to", because in TCP the destination is given already in the *connect()* call. Here there is no *connect()* because there is no connection [6]. Our next message may be *sendto()* someone else. Just as we could call *bind()* before *connect()* in TCP we could also do it in UDP before *sendto()* but we normally don't in either case. One reason is that its more work, another reason is that if we try to bind to an already used port we run into problems. Therefore, we normally go with the ephemeral port number that the OS hands out, since we normally do not care about the client/active socket's port number.

Listing 4.16 shows the code for a receiver that waits for the data sent in the previous listing. Here we set up the listener for "in6addr_any", the same as "::0". The probably unexpected code in lines 17-23 asks the operating system to create this socket even though there may already be a socket from a previous run hanging in "TIME-WAIT". This is not needed if the code is just run once, but somehow it never is. Now we need a bind for the listening-socket. Naturally it must be listening at the right port, and the "any" IP-address is practical; if it makes it this far it must be to one of our interfaces, so why not wait at them all and do away with the need to find out which IP-addresses we have where… Then we declare a "holder" for the remote address, which we could print out (but don't), and finally we call *recv()*. This blocks until a message is received, and when that happens it is written to the terminal.

```
01 void recvit()
02 {
03    int sock = socket(AF_INET6, SOCK_DGRAM, 0);
04    if (sock <= 0)
05    {
06       printf("Opening socket gave error: %d\n",sock);
07       exit(sock);
08    }
09
10    // The inet address we will listen at
11    struct sockaddr_in6 listen_addr;
12    memset(&listen_addr, 0, sizeof(listen_addr));
13    listen_addr.sin6_family = AF_INET6 ;
14    listen_addr.sin6_port = htons(2000);
15    listen_addr.sin6_addr = in6addr_any;
16
17    int on=1;
18    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
19                        (char *)&on,sizeof(on)) < 0)
```

```
20    {
21       printf("setsockopt(SO_REUSEADDR) failed");
22       exit(0);
23    }
24
25    int err;
26    err = bind(sock, (struct sockaddr *) &listen_addr,
27                sizeof(struct sockaddr_in6));
28    if (err)
29    {
30       printf("Binding gone wrong - error %d\n", err);
31       exit(err);
32    }
33
34    char mystring[100];
35
36    // Space for remote socket data
37    struct sockaddr_in6 remote_addr;
38    socklen_t addr_len = (socklen_t) sizeof(remote_addr);
39
40    err = recvfrom(sock, mystring, sizeof(mystring),  0,
41         (struct sockaddr *) &remote_addr, &addr_len);
42    if (err < 0)
43    {
44       printf("Could not recv. Error: %d\n", err);
45       exit(err);
46    }
47    else
48       printf("Received: %s", mystring);
49 }
```

**Listing 4.16:** IPv6 UDP Receive

Figure 4.17 shows the test of the two programs. First *recv()* is run in the background. We then use *netstat*[7] to show that we have a udp6 listener on port 2000. Note that it also gives us the process-number and the name of the program. When the sender-program is run the receiver unblocks and terminates.

```
01 kelk@debianBK:~/workspace/C/sockserver6$ ./udprecv &
02 [1] 2734
03 kelk@debianBK:~/workspace/C/sockserver6$ netstat -au -pn -6
04 Active Internet connections (servers and established)
05 Proto Recv-Q Send-Q Local Address Foreign Address PID/Program name
06 udp6       0      0 :::48725      :::*            -
07 udp6       0      0 :::34455      :::*            -
08 udp6       0      0 :::10154      :::*            -
09 udp6       0      0 :::961        :::*            -
10 udp6       0      0 :::2000       :::*            2734/udprecv
11 udp6       0      0 :::111        :::*            -
12 udp6       0      0 :::5353       :::*            -
13 kelk@debianBK:~/workspace/C/sockserver6$ ./udpsend
14 Received: The center of the storm
15 [1]+  Done                    ./udprecv
```

**Listing 4.17:** Running on localhost

Figure 4.18 shows a Wireshark capture of the scenario. At first only the "udpsend" program is run in frame 1. Interestingly this generates an ICMPv6 "Port Unreachable" error in frame 2 as there is no receiver. This also happens on communication outside the localhost and enables us to write code that actually **can** handle retransmissions etc. of UDP-packets.

22 seconds later the full scenario is run as in 4.17, and this time there is no ICMP-error as the waiting socket accepts it.

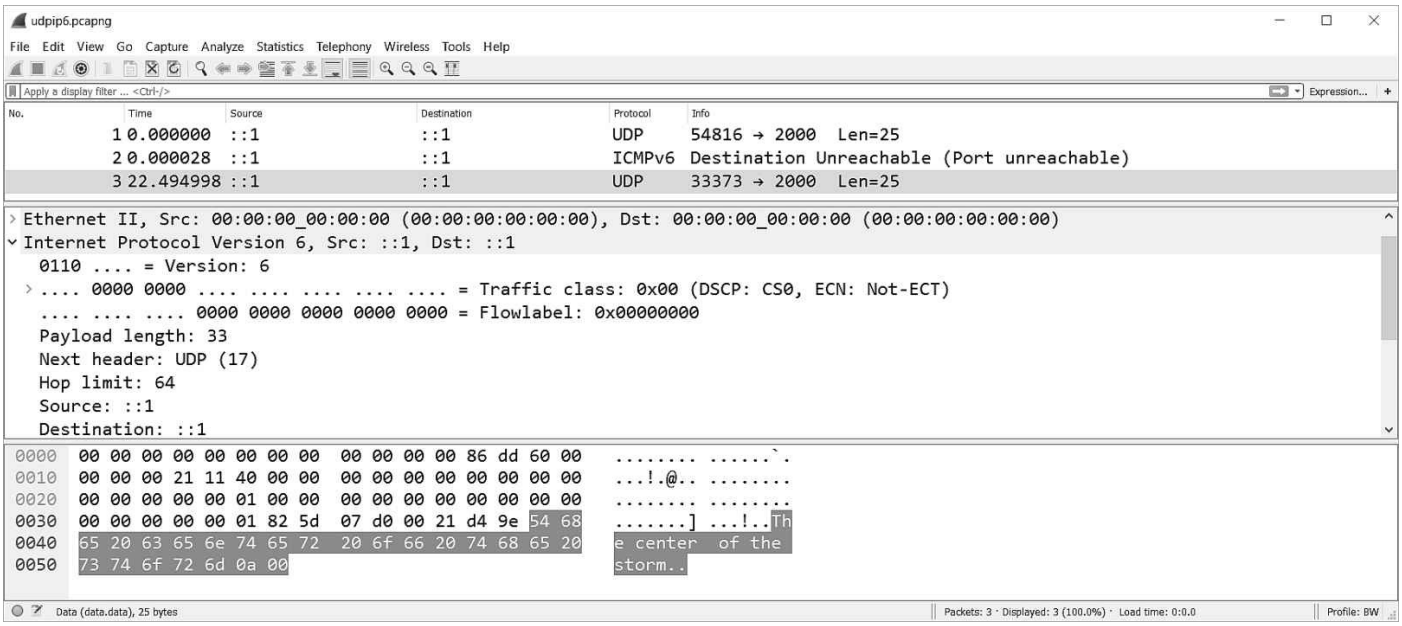



**Figure 4.18:** UDP on IP6 with ICMPv6 error

## 4.20 Application Layer Protocols

As stated in the introduction to this chapter, there are myriads of application layer protocols. It is impossible to go through them all, so instead we will look at the criteria for choosing one or the other.

**Important Parameters for Protocols**

| | |
|---|---|
| Standard | Domain or Company |
| Documentation | Good or Bad |
| Flow | Pipelining versus Stop & Go |
| State | Stateless or Stateful |
| Low-level | Binary or Textual |
| Flexibility | Forgiving like XML - or not |
| Compatibility | Versionized |
| Dependencies | Requiring Special OS, Language etc |
| Power | Designed for power savings |

- **Standard**
  Is the protocol already a standard in your company or in the application domain - or both? If this is the case it takes very good arguments to pick another protocol. After all - protocols tie equipment together. Do you really want to be the person responsible for the fact that the new product doesn't work in a system with the old product - or that the software developers are delayed another half year to support your new fancy protocol?
- **Documentation**

If different companies in a common application domain are using the same protocol it will almost certainly be well documented. This is not a given truth when a single company has many products using the same protocol. A change in technology may force you to re-implement existing protocols - or start all over. In this case it is important how well documented the old protocols are.

- **Flow**
  You may have a very fast connection and yet not be able to utilize it. If there is a long distance between the communicating parties, this means a delay. We typically talk about the roundtrip-time which is the time it takes to send a packet from A to B plus an answer back to A. If A and B are close geographically, the roundtrip-time is dominated by the time it takes to "clock out" the packet and the time it takes to go up through the stack on A and B. If A and B are at each side of the globe the roundtrip-time is dominated by the time it takes a packet to travel through the wires and intermediate routers (or the satellite link). The term "long fat pipes" means that we have fast (fat) but also long wires. A "Stop & Go" protocol completely kills any speed in this as a "Stop & Go" protocol requires a response from the receiver to every packet sent - before the next packet can be sent. The opposite of this is "pipelining" where you can have many data in the pipe. TCP has gone through a lot of changes to support "long fat pipes" - mainly the "Receive Window" can be very big - allowing a **lot** of data to be in *transit*. You might argue that HTTP actually does require a response to every request and thus is a bad protocol. However, as explained in Section 4.10 HTTP does not require a response to request "n" before sending request "n+1". When you load a new page in your browser, the first request for this page has to be answered, but when you have this answer you may simultaneously request all referenced pictures, banners etc. See Section 4.10.
- **State**
  As explained in Section 4.11 on REST there is a lot to be gained by using stateless protocols. The main benefit in an IOT device is probably the possibility of pipelining as described above. One of the other main benefits with REST is the ability to use server-parks, but when you see the IOT-device as a server - as you should - it is rarely part of a "park". There may be many almost identical IOT-devices, but each is typically connected to unique sensors, or in other ways providing unique data.
- **Low-Level**
  Many embedded developers prefer binary protocols as they are compressed compared to handling e.g. the same numbers as hex-ascii. A 16-bit number does not need more than 16-bits in a binary protocol, but in hex-ascii it takes a full byte for each nibble - thus 32-bits for the same 16-bit number. If data is represented in e.g. JSON or XML it takes a lot more space as e.g. parameter-names includes overhead and in the case of single variables typically take more space than the data. The PC-programmer at the other end often prefers XML as this is a well-known technology and he/she has a lot of tools available for parsing these data. Between the two parties we have the wire. Watching a binary protocol in Wireshark can be a pain, whereas the same data in JSON or XML is close to self-explaining.

  The choice becomes a trade-off between CPU-time, wire-speed and which developers you listen to. It is true that connections are getting faster and that yesterday's PC-technology often becomes tomorrow's embedded technology, so there is a trend

towards textual representations in the general embedded world. However, many IOT-applications will weigh low energy consumption higher than any of the previously mentioned parameters. This will move the balance somewhat back towards the binary protocols as each bit sent has a cost in Joules.

- **Flexibility**
  Many protocols are rigid and cannot handle missing data or several versions. This is one of the reasons that XML is so popular. There is no requirement to send all defined parameters. This however, introduces the need for a more or less advanced default-handling.
- **Compatibility**
  It is easy to forget a version number in a protocol. This results in some very clumsy code once changes are made, so never create or use a protocol without a version number. A lot can be learned from the TCP/IP stack. It is amazing how many ways you can implement TCP within the standard - and how it can be extended in a backwards-compatible manner.
- **Dependencies**
  Not so long ago DCOM was a popular remote protocol. One of the major problems with DCOM was that it was completely tied to the Windows OS. Before this CORBA was very popular in the Unix world. It required expensive tools that you could also get for PC's but it never really caught on here.

  A major reason for the success of REST on top of HTTP - see Section 4.11 - is that it is easy to understand, runs without complex tools and on any OS. Another problem with DCOM and CORBA was that they were based on remote procedure calls, and did not perform very well if one end was suddenly not available. In the IOT-world we need a much more loosely coupled concept with clients and servers instead of masters and slaves. There are "cross-overs" like JSON which is a "lean and mean" replacement for XML. It was made for Java, but was easily moved to other languages.

- **Power**
  This subject was already touched upon in the discussion on binary versus textual. Section 7.1 goes into more details on protocol performance which is closely linked to power consumption.

## 4.21 Alternatives to the socket API

The socket concept we are using in this book is generally known as Berkeley sockets. The basic API is the same on all major platforms but there are many variations in the implementation of the options and the underlying TCP/IP-stack. By now it is clear that programming directly on the socket-interface can be a little frustrating sometimes. Especially the fact that TCP is streamed and therefore does not keep the boundaries from the transmitting hosts "*send()*" calls to the receiving hosts "*recv()*" calls can be annoying.

"Raw sockets" is **not** the same as Berkeley sockets - although many programmers believe so. With a real "raw socket" things are even more primitive as you are also responsible for all the headers. This is only relevant in test scenarios where you want to

test error-handling. We will get back to this in Section 8.8.

In order to tweak the various socket options you often end up using the Berkeley sockets directly. Nevertheless, there are situations where you can get very far with a high-level library - and these are getting better all the time. The far most used is “libcurl”. It is open source licensed under the MIT/X license and states that you may freely use it in any program. It supports a very long list of application protocols - among these HTTP and HTTPS (which basically is HTTP on top of secure sockets) - on almost any platform known to mankind. In any case the libcurl site is a very good starting point as it has a page with a long list of its competitors and their license-types.

If scripting is acceptable, Python with and without ScaPy is a great tool - see Section 8.8. PHP on top of libcurl is also very effective. You may argue that Python and PHP are for web-servers, but the typical IOT-device will act as a server and if you are using REST you are probably implementing a web-server. Another library is the “Libmicrohttpd” library which is based on the “GLPL” - license - see Section 2.7.

You may prefer a full web-server. If the Apache web-server is too big for your device, the Go-Ahead web-server may be interesting. Not so many years ago it was very small and primitive, but it seems to have come a long way since then. Go-Ahead comes with a GPL-license as well as a commercial royalty-free license.

An alternative to the various more or less independent libraries is to use C# and mono if your platform is Linux-based. C# has a large number of great libraries - including some for handling HTTP. With this solution you get a lot of functionality that you may be spoiled with if you are used to working on the Windows platform. This is a drastic solution to a minor problem. One of the main advantages of the Linux platform is the huge user society available that may help in many cases. If you put C# and mono between your application and the OS it may not be so simple to find someone in the same situation. Naturally, if you are on the Windows platform then C# is mainstream.

## 4.22 Further Reading

- Kurose & Ross: Computer Networking: A Top-Down Approach
  I used to teach at the Danish Technical University based on this book. The top-down approach is good when the whole concept of networks is new. As teaching material for 101 on Computer Networks it cannot - and should not - dive deep, but it gives a great overview. It also includes a very good chapter on Security - ciphers, symmetric versus public & private keys etc.
- Laura Chappel: Wireshark Network Analysis
  An extremely detailed guide to Wireshark. This is a book filled with information and tricks.
- Stevens: TCP/IP Illustrated Volume 1
  This is **the** book on the internet protocol stack. It is old now and sadly will not be updated by Stevens. Still it is extremely well-written and if you are interested in networking volume 1 is a must. Volume 2 is about implementing the stack, and volume 3 is about application layer protocols as HTTP and newer sources are recommended on these subjects.

- Richardson & Ruby: RESTFul Web Services
  This book explains REST very well. It is not really targeted for the embedded world but all the basic principles are the same as on big web-servers.
- tcpipguide.com
  A very informative website with many good figures.

# Chapter 5
# Wireless Networks

## 5.1 WiFi Basics

There are many wireless technologies. Of these some are WLAN, and of these most are IEEE 802.11 based. The WiFi Alliance has registered “WiFi” as a trademark for IEEE 802.11-based products. This chapter is mostly about WiFi.

Many of the Wireshark captures in Chapter 4 were actually done on a PC connected to a WiFi SOHO Router. This is transparent when using Wireshark alone. However, if you buy the device “AirPcap” from Riverbed, you can get a lot more insight.

A normal setup is running in “Infrastructure Mode”, where a Wireless Router is known as the “Access Point” - AP - and all iPad’s, PC’s and IOT-devices on WiFi, are known as “Stations” - STA’s. They all “talk” on the same channel which means that only one AP or STA at a time can talk. The whole inter-working setup is a “Basic Service Set” and similar systems nearby that can interfere are known as “Overlapping Basic Service Sets”.

The “Basic Service Set ID” - BSSID - identifies the AP. It is the MAC address of the radio. The “Service Set ID” - SSID - identifies the WLAN and may be present on several AP’s. When an AP handles several SSID’s each has a BSID, thus an AP is assigned as many MAC-addresses as the number of SSID’s it can serve.

| Term | Explanation |
|---|---|
| Channel | Specific Frequency |
| SSID | Service Set ID |
| STA | Station. A device talking to an AP |
| AP | Access Point.<br>The correct term for the Router in the middle |
| Infrastructure Mode | The Concept of an AP in the middle |
| BSS | Basic Service Set. 1 AP+n*STA |
| BSSID | ID of a given BSS |
| OBSS | Overlapping BSS.<br>Neighbour on same channel |
| HT | High Throughput defined in 802.11n |
| VHT | Very High Throughput. Defined in 802.11ac |

In the 2.4 GHz band there are 11 channels (more in some countries), but they overlap. If interference is to be avoided you can have separate Basic Service Sets on channel 1, 6 and 11 at the same time. WiFi is defined by standards 802.11, with the following letters being very important. The first really used standard was 802.11b, then for some years

802.11g, and currently 802.11n and 802.11ac are where the action is - both including the 5 GHz band. If you have 802.11b equipment you should get rid of it as is monopolizes the airtime as we shall see.

## 5.2 The Access Point as a Repeater

The most common use of an Access Point is access to the internet. In this scenario the frames are simply sent over the air once. You can buy small routers for use in e.g. a hotel, that connects by wire to the hotels wired internet, and then allows you to hookup several devices wireless to the AP. The typical user of an iPad, phone or PC rarely needs to interconnect these devices, they just need the internet, and if they are teenagers, it's probably your fault if they can't get this connection. These routers/Access Points sometimes can be set to be "IP Isolated" or not, which means "should the individual Station be isolated from the others - or allowed to see them?" In a home you probably need better connectivity - e.g. a NAS-disk with movies, music or just backup.

The isolated scenario is the same as the first part of the scenario we are going to look into now, where one STA sends data to other stations on the same network, see Figure 5.1. Here a multicast frame is sent from one STA to the others. On a wired network it would be done in one transmission, but not here. When one STA is sending something it is by definition (in infrastructure mode) to the AP, and the other STA's therefore go into sleep-mode to save power. To help in this there is generally information in all frames about the expected duration - see Section 5.8. Frame 2419 is from the STA to the AP, while frame 2421 is the same data from the AP to all the other STA's. When setting up the AirPcap, Per-Packet-Information - PPI - was chosen which we see in the dissections of the two frames in Figure 5.2:

1. Frame 2419 is sent at 130 Mbps which is an 802.11**n** speed, but 2421 is sent at 6 Mbps - an 802.11**g** speed - the lowest common denominator. It would be even worse if 802.11**b** devices were present. Often the AP can be setup to "legacy mode", where it complies with the b standard - even though there are no detected b devices, or it can be in "mixed" mode where it respects b-devices if they are there. The last mode is "greenfield" where only n-devices are respected. This adds very little value compared to mixed mode with no b-devices. So using mixed mode is playing it safe.[1]
2. When right-clicking a line in Wiresharks overview you get a popup-menu. Here frame 2419 was selected as a time-reference. The following time-stamps are now relative to this. Note that this can be done several times in the same capture so be careful when reading a time-difference, there could be an intermediate REF you have missed. Here it shows us that the time from frame 2419 is sent and until 2421 is sent is 57 µs. Sending the 352 bits at 6 Mbps theoretically takes 58.7 µs.
3. The frames are identical on the UDP level as well as the IP-level - but not on 802.11.
4. Both frames note the MAC source as IntelCor_35:db:68 and MAC destination as the multicast address 01:00:5e:00:00:de.
5. There's a BSS ID which in both frames is ubiquiti_54:ad:6f - the MAC-address of the Access Point.
6. Frame 2419 has a "Transmitter" which is the same as its MAC source, IntelCor_35:db:68, and a "Receiver" which is the Access Point (ubiquiti_54:ad:6f)

7. Frame 2421 has a “Transmitter” which is the same as the AP’s MAC, ubiquiti_54:ad:6f and a “Receiver” - the multicast: 01:00:5e:00:00:de

Whereas a wired Ethernet frame only has a Source and Destination MAC address, the Wireless Ethernet has room for four addresses and “To DS” and “From DS” bits. Luckily, Wireshark interprets this for us as:

| Term | Explanation |
|---|---|
| Source | MAC of original source |
| Destination | MAC of original destination |
| Receiver | MAC of Receiving Radio now |
| Transmitter | MAC of Transmitting Radio now |

In other words Wireshark helps us keep the original source and destination terms, and is using “Receiver” and “Transmitter” for the intermediate STA or AP on the specific frame.

| No. | Time | Source | Destination | Protocol | Data rate | Info |
|---|---|---|---|---|---|---|
| 2419 | *REF* | 192.168.1.12 | 239.0.0.222 | UDP | 130 | 57341 → 15000 Len=256 |
| 2420 | 0.000002 | | IntelCor_35:db:68 … | 802.11 | 24 | Acknowledgement, Flags=. |
| 2421 | 0.000057 | 192.168.1.12 | 239.0.0.222 | UDP | 6 | 57341 → 15000 Len=256 |

**Figure 5.1:** Multicast on WiFi is two-fold

```
Frame 2419: 406 bytes on wire (3248 bits), 406 bytes captured
PPI version 0, 84 bytes
802.11 radio information
IEEE 802.11 QoS Data, Flags: .......T.
  Type/Subtype: QoS Data (0x0028)
> Frame Control Field: 0x8801
  .000 0000 0011 0000 = Duration: 48 microseconds
  Receiver address: Ubiquiti_54:ad:6f (dc:9f:db:54:ad:6f)
  Destination address: IPv4mcast_de (01:00:5e:00:00:de)
  Transmitter address: IntelCor_35:db:68 (24:77:03:35:db:68)
  Source address: IntelCor_35:db:68 (24:77:03:35:db:68)
  BSS Id: Ubiquiti_54:ad:6f (dc:9f:db:54:ad:6f)
  STA address: IntelCor_35:db:68 (24:77:03:35:db:68)
```

(a) Frame 2419 from STA to AP’s

```
Frame 2421: 352 bytes on wire (2816 bits), 352 bytes captured
PPI version 0, 32 bytes
802.11 radio information
IEEE 802.11 Data, Flags: ......F.C
  Type/Subtype: Data (0x0020)
> Frame Control Field: 0x0802
  .000 0000 0000 0000 = Duration: 0 microseconds
  Receiver address: IPv4mcast_de (01:00:5e:00:00:de)
  Destination address: IPv4mcast_de (01:00:5e:00:00:de)
  Transmitter address: Ubiquiti_54:ad:6f (dc:9f:db:54:ad:6f)
  Source address: IntelCor_35:db:68 (24:77:03:35:db:68)
  BSS Id: Ubiquiti_54:ad:6f (dc:9f:db:54:ad:6f)
  STA address: IPv4mcast_de (01:00:5e:00:00:de)
```

(b) Frame 2421 from AP to STA's

**Figure 5.2:** Multicast in WiFi

## 5.3 How is speed calculated?

In Section 5.2 we saw different speeds listed by Wireshark. But how are these obtained? This is a very advanced subject, but we will focus on the headlines that are found in Wireshark:

- **MCS index**
  This index is a combination of spatial streams, modulation type and coding rate. A spatial stream is an independent data flow in "space". The modulation type is the number of states each transmitted symbol can have. A bit only has two states but e.g. 64-QAM has 64-states per symbol corresponding to 6 bits. The coding rate is the "true rate" of information when error-correction is accounted for.
- **Bandwidth**
  When we say that there are 11 channels in the 2.4 GHz range, each is 20 MHz. It is possible to combine two into a 40 MHz channel, but this will cost 2/3 of the whole 2.4 GHz band when used (we are still time-sharing the channel).
- **Guard Interval**
  This is the space between symbols. The normal guard is 800 ns, a short guard is 400 ns. Note that at the time of this writing AirPcap only supports the normal guard.
- **Subcarriers**
  As discussed we may have a number of channels in e.g. the 2.4 GHz range. They all lie close to 2.4 GHz but in reality they are 5 MHz spaced. Channel 1 is at 2412 MHz, channel 2 at 2417 MHz etc. With this information it is no wonder that a 20 MHz wide channel will invade the neighbors - giving us only channel 1,6 and 11 undisturbed. Using Orthogonal Frequency Division Multiplexing - OFDM - each channel is split into a number of subchannels - or rather subcarriers. Each of these is generated and deciphered using Fast Fourier Transform - FFT - technology.

If we have MCS index of 7, long Guard Interval and a 20 MHz channel the data rate can be calculated:

| Factor | Value |
|---|---|
| Spatial streams | 1 Flow |
| 64-QAM = 64-states/symbol | 6 bits/symbol |
| Time per symbol 4 µs incl long guard | 250 k/s |
| Subcarriers in 20 MHz | 52 |
| Left After Forward Correction | 5/6 |

Totally this is: 1 * 6 bit * 250k/s * 52 * 5/6 = **65 Mb/s**

A short guard will cut-off 0.4 µs of the 4 µs, increasing the speed with 11%. You can find tables of MCS indexes at Wikipedia and other places, but as we shall see this is also built into Wireshark.

This speed is the maximum theoretical speed without retransmissions and without time-sharing the channel with other devices.

## 5.4 Case: WiFi Data-transmission

Figure 5.3 shows a scenario with a data-producing device that is wired to a small Asus router and via this is transmitting data to a “pre-Air” iPad. The recording is made with AirPcap and using the PPI-header from the 802.11 layer. A fantastic feature in Wireshark is that when you “open up” the dissector you can right-click on an interesting value and select “Show as Column”. This has been done with all the columns shown between “protocol” and “info”. Let’s walk through the case:

- There is a little communication going on with a Sonos music system that is running on another AP - seen in frame 835709 and colored black. It does not seem to cause problems.
- All TCP-communication happens within aggregated frames. Aggregated frames are a very important part of the newer wireless standards. As we will see later (section 5.7) there is a lot of overhead before a transmission can be started, so it is of great value to “keep going”. Disregarding the two first lines, we first have a block of aggregated TCP-data in 9 frames with a total of 13890 (8*1622+914) bytes from the Device via the Asus AP. This is followed by a reply from the iPad STA, with a total of 6 TCP frames, which based on earlier wired analysis of the same scenario, we know are all ACK’s. This is confirmed by Wireshark in the “RTT” column which is Wiresharks calculation of the Round-Trip-Time. This can only be calculated with the detection of an ACK on a previous message. Interestingly, the RTT is measured to be 14-20 ms, which is about the same as the duration of 5-10 aggregated frames.
- Data is sent with MCS index 7 which as we know is 20 MHz band and long guard (SGI - Short Guard Interval - false). The TxRate shows 65 Mbps which is consistent with the calculations in Section 5.3.
- RSSI - Received Signal Strength Indication - shows 255 for all the aggregated TCP

frames, except the last in each aggregation, which is 66 from the Asus and 59 from the iPad. This number is an indication of the received power - the 255 values should be ignored. RSSI can only be used when comparing measurements with the same equipment as it is not rooted in mW or similar, but quite arbitrary. It is however interesting that the Asus and the iPad are in the same “area” of values.

- Before each datatransmission is a “Clear-To-send” - CTS - with a value in us in the “Duration” field. This is not the duration of the CTS, but an estimate on the duration of the following data-transmission. By marking each CTS as REFerence, it is easy to check the actual duration. This is shown by the arrows from the CTS to the first frame after the aggregated transmission. Section 5.8 goes into more details on this. It is clear that the estimate is best for the longer duration.
- The CTS is an answer to an RTS - Request-To-Send. When the iPad is about to send its ACK’s we see an RTS from the iPad to the Asus in frames 835710 and 835719, both immediately followed by the CTS from the Asus AP. As the CTS and RTS are sent from different devices they may reach a wider audience (“hidden terminal problem”).
- We also see RTS-CTS pairs before the Asus STA is about to send. Why is that? This is an infrastructure system; as long as the AP is transmitting the STA’s will shut up. And there is no transmitter on the CTS, only the Asus as destination. The explanation is that the Asus AP is asking and giving itself permission. This is partly to warn Neighboring BSS’es that this channel is going to be taken, partly to tell other STA’s on our network that they might as well take a nap for the duration of the transmission and thus save energy.

Figure 5.4 shows another tool applied - Chanalyzer with Wi-Spy from MetaGeek. We are measuring on channel 6, on the SSID called BK3660A-100039. The information about the WiFi SSID’s in the bottom table comes from the PC’s wireless NIC. Based on this info the tool draws a “profile” of a given network - dotted line - over the actual measurement.

Apart from looking very fancy, this is actually a very nice tool. We can see a lot of stuff on Wireshark but it’s all above the physical layer. You may have problems with connections that only show up in Wireshark as retransmissions. Noise on the physical level from e.g. a micro-wave oven, a wireless mouse or an infrared movement-detector may interfere. This can be seen with a tool such as Wi-Spy.

---

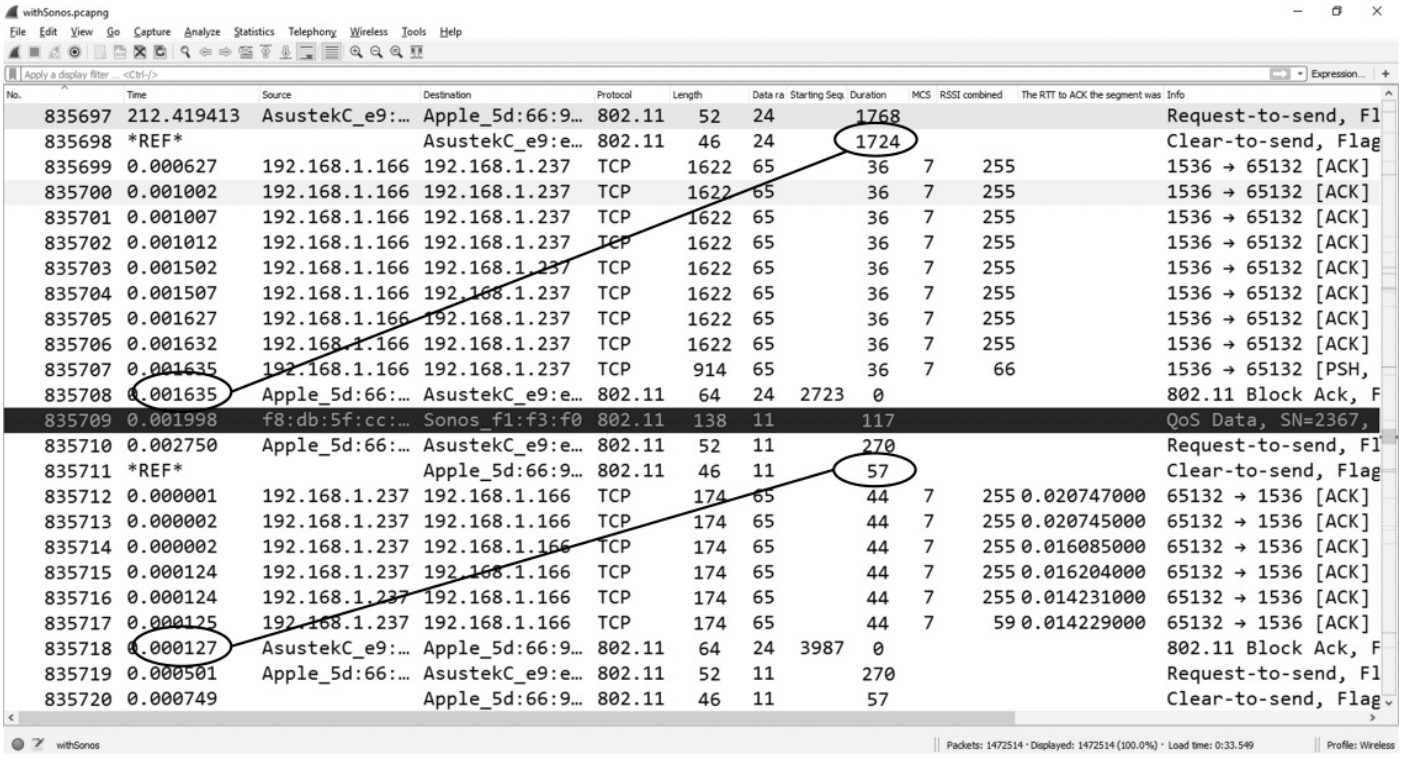



**Figure 5.3:** Full speed ahead on WiFi

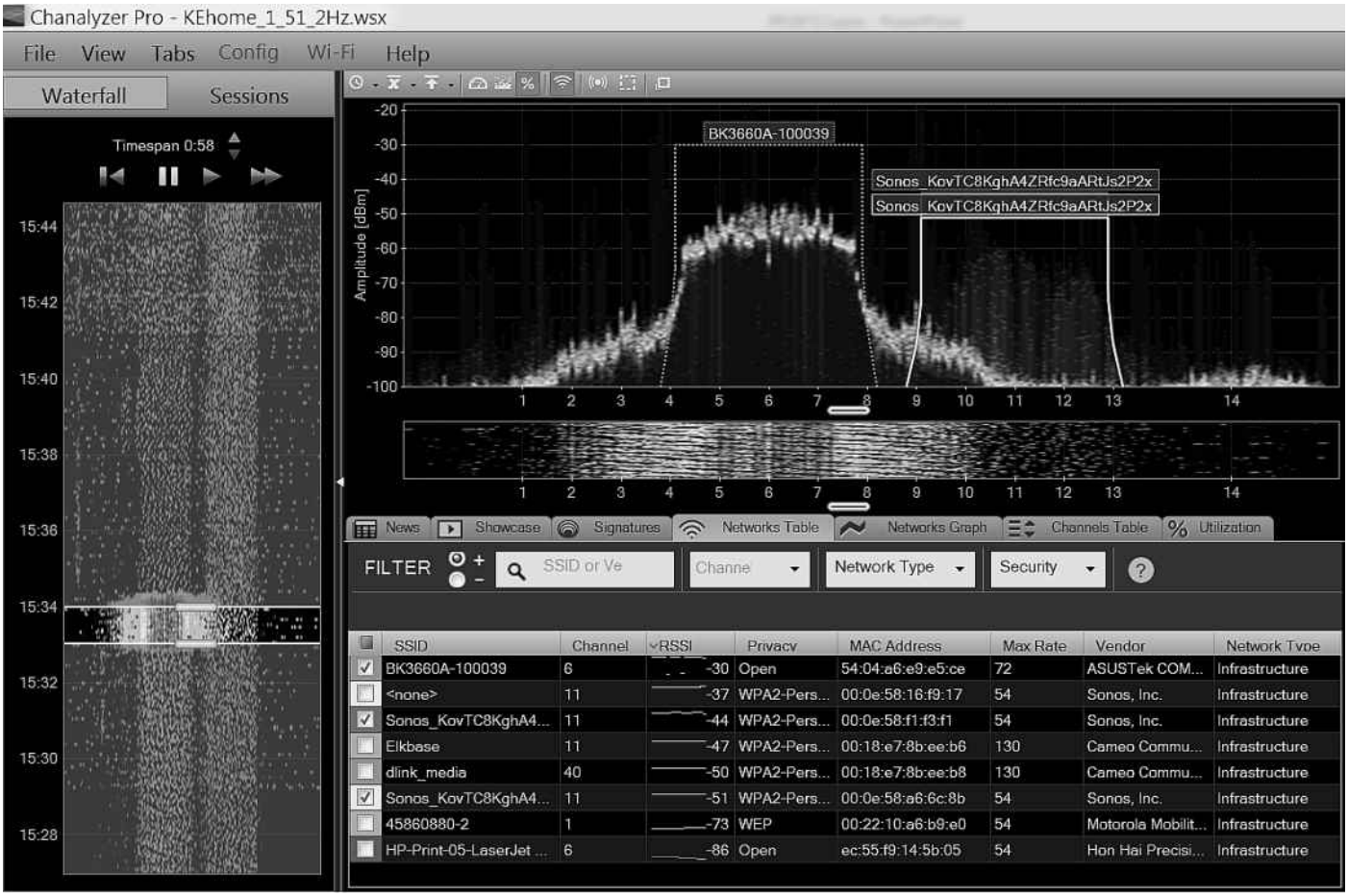



**Figure 5.4:** WiFi spectrum of our transmission

## 5.5 Case: Beacons

We are not completely done with the transmission in the previous section. It is interesting to know why we ended up with the MCS index 7, and why only 20 MHz Bandwidth, as the Asus AP was setup to support 40 MHz. And in 20 MHz mode we should get 72 Mbps according to Figure 5.4. It turns out that there is a handshake, just as in the TCP SYN and many other scenarios. The handshake from an Access Point is its “Beacon” which is typically transmitted each 100 ms, see Figure 5.5. The handshake from the STA is its “Probe Request”, see Figure 5.6. The boxes show the most interesting stuff. Clearly the Asus AP supports the 40 MHz wide band as well as the Short Guard Interval, but the iPad supports neither. AirPcap does not support the short guard interval, so if both devices had supported this, we would not be able to follow the transmissions. Both ends support Aggregated frames up to 64kBytes - much longer than the 13890 bytes we have seen in the transmissions. A STA or an AP does not have to make aggregate frames max-size. The timing of incoming frames is weighed against latency when deciding how many frames to aggregate into one.

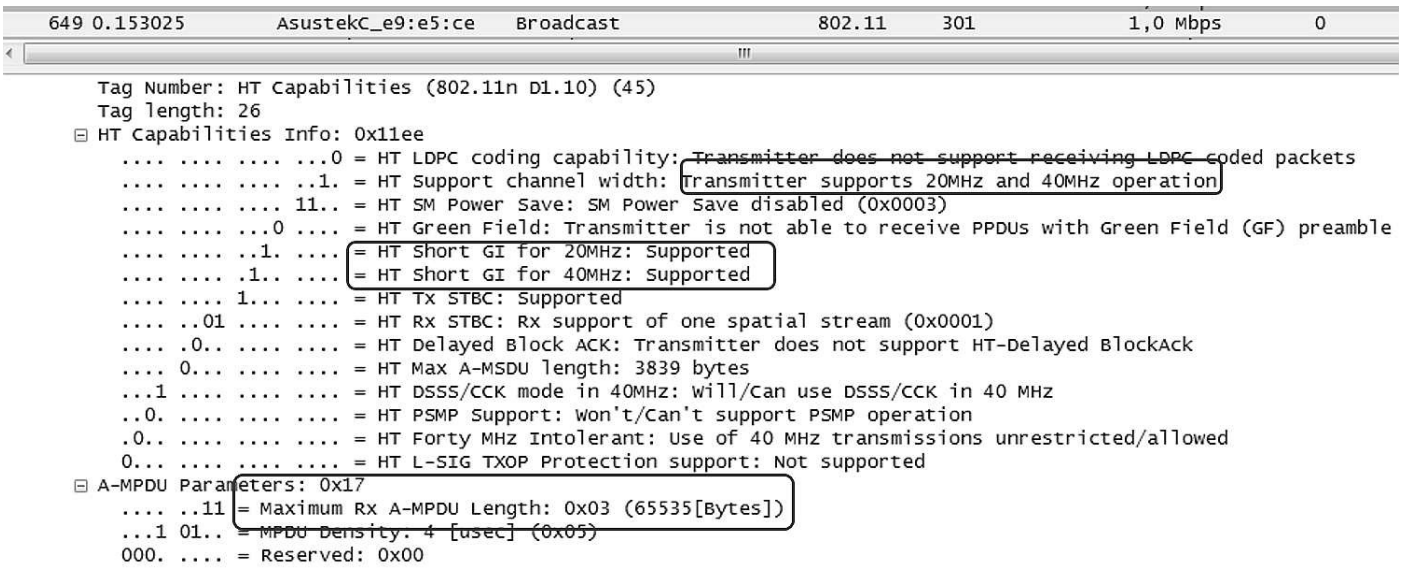



**Figure 5.5:** Beacon from the Asus AP

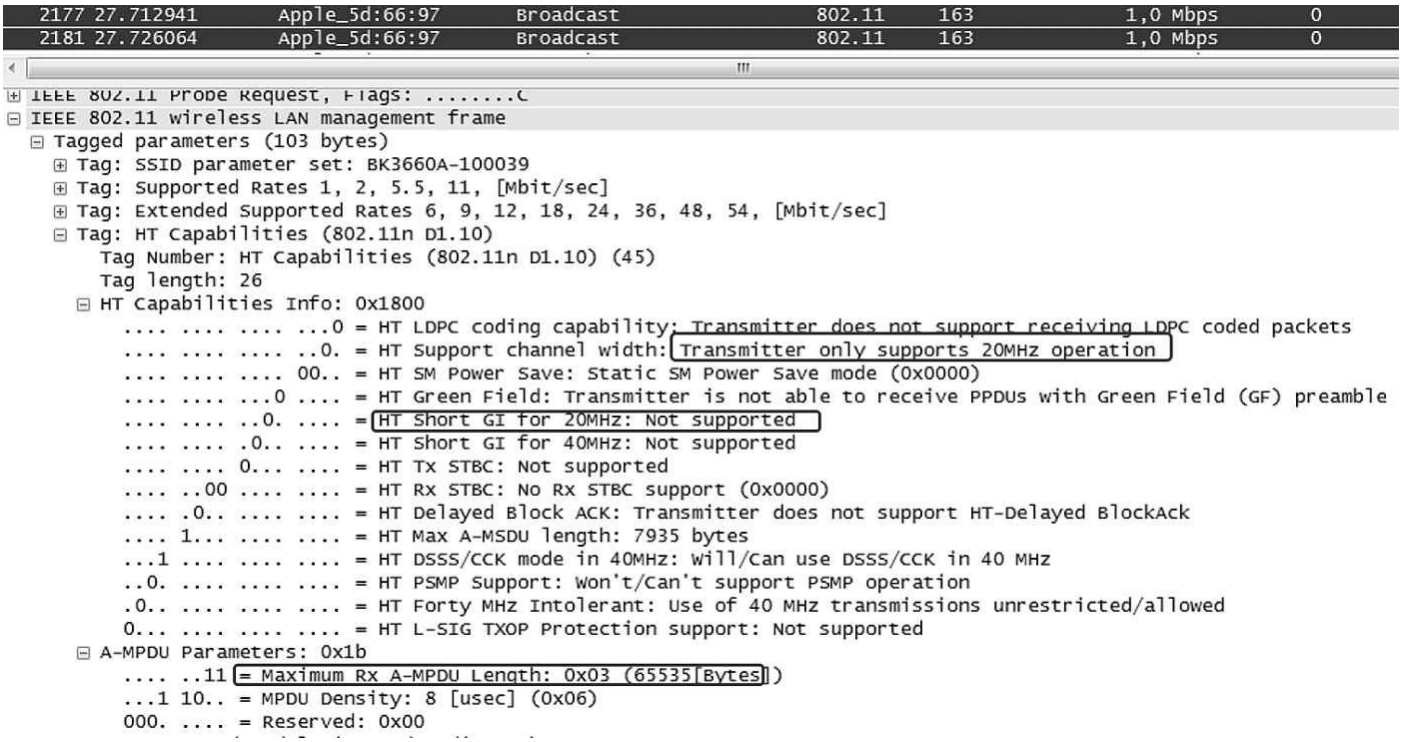



**Figure 5.6:** Probe from the iPad STA

## 5.6 Why so hesitant?

The setup that we have studied for some time, was working fine. On the GUI there were “moving curves” and no loss of data. However, once in a while there was a short “hesitation”. Timing it, it appeared to be every 5 minutes. Picking “Statistics” in Wireshark, then “I/O-graph” on a longer capture, Figure 5.7 was created. The top line at app. 20M bit/s (y-axis is on the right for this) is data in the direction where data actually flows. It is clear that something happens after 5 minutes. The area of the “dive” fits with the area in the opposite direction, explaining that no data is lost, but verifies the “hesitation” experienced. Making the similar graph on RTT - Round-Trip Time - in Figure 5.8 verified the symptom [2]

Browsing the standards it became clear that according to 802.11n all STA’s are required to scan all WiFi channels every 300 seconds and report their findings back to the AP. The AP collects this information and spreads it to all STA’s so they can update their “Navigation Vector”. Knowing what to look for now, Figure 5.9 was produced. The figure shows a Wireshark display filter on the “power-management” bit (showing up as “P” in the info-field) and the relevant BSSID (the MAC address of the AP). Every 300 seconds the iPad takes 13 small “Power Naps”. Technically the 2.4 GHz band has 14 channels (not all legal everywhere), and when the iPad is listening to one of these it needs to scan the 13 others every 300 s. So far no workaround has been found for this.

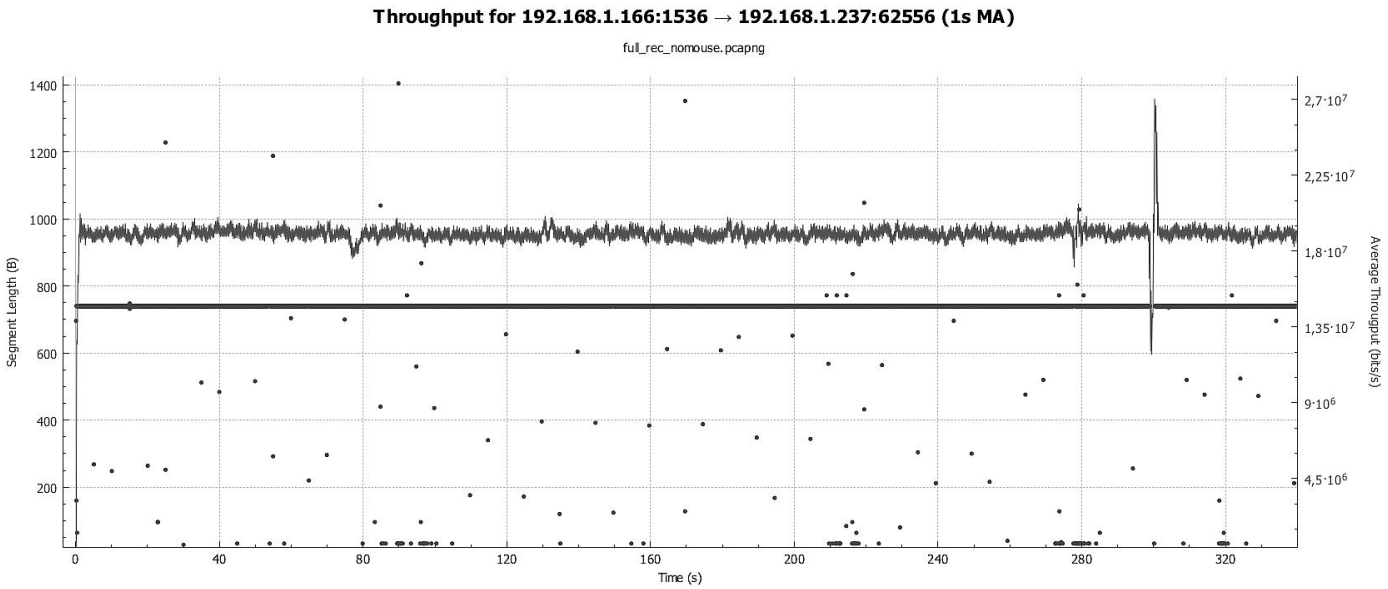



**Figure 5.7:** Wireshark Throughput graph

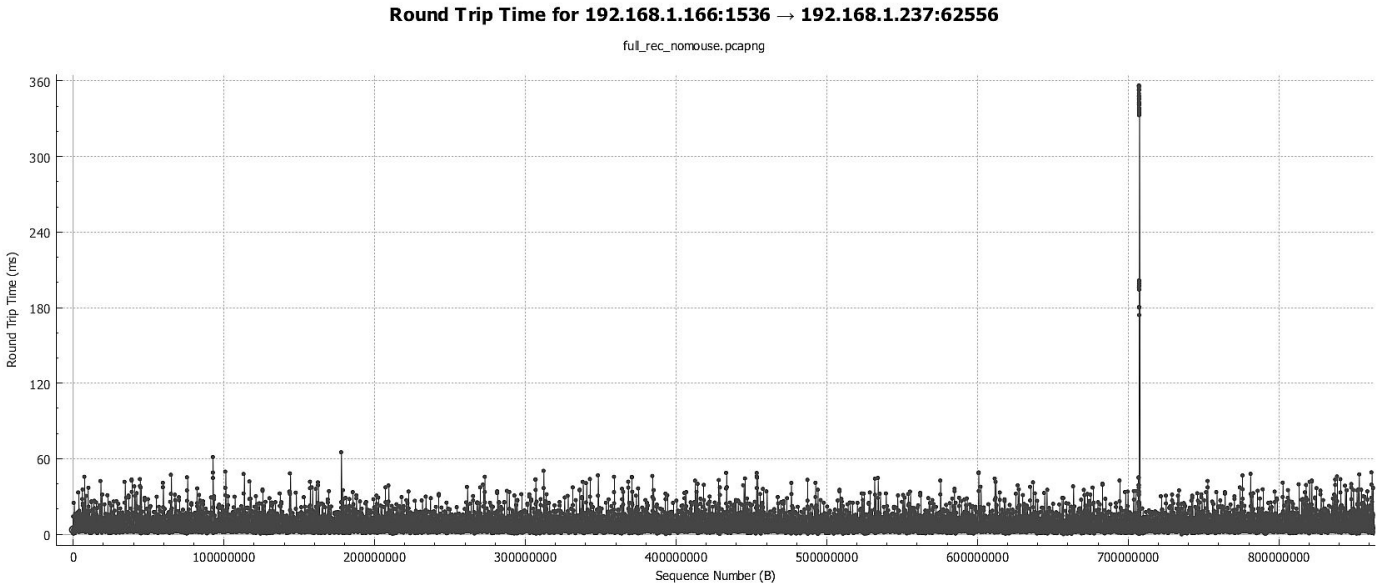



**Figure 5.8:** Wireshark RTT

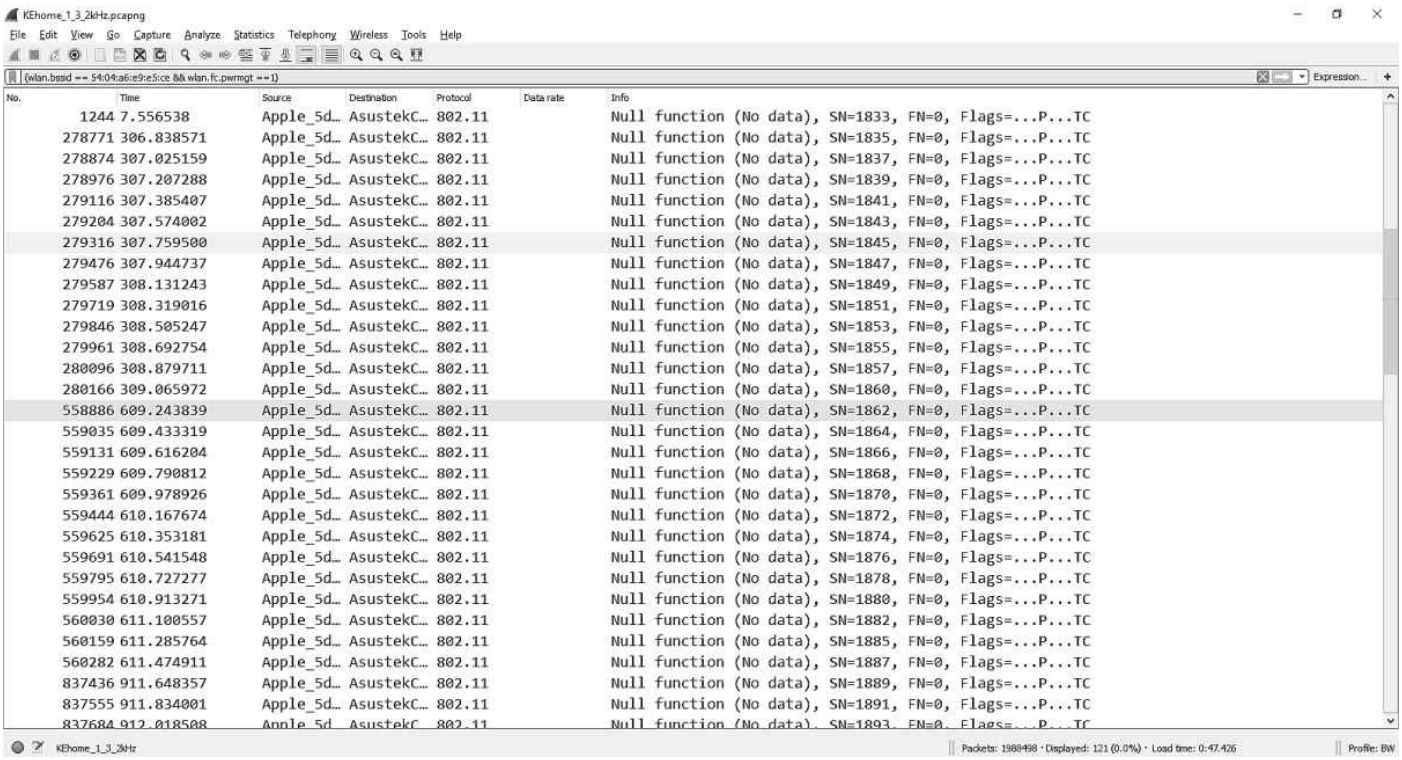



**Figure 5.9:** iPad Power Nap. 13 breaks every 300s

## 5.7 Aggregated Frames

As smarter types of coding was developed it was clear that the outstanding problem was the overhead involved with each frame. The usual solution when you have a lot of overhead per something is to bundle and share the overhead, and this is exactly what has been done. This is called Aggregation and in Section 5.4 we saw “aggregated frames”.

The thing that is aggregated is PDU’s - Protocol Data Units. A PDU is often loosely stated the “packet” at any level. As stated earlier the correct term in Ethernet is a “Frame”, while in IP it is a “Packet” and in TCP it is a “Segment”. At the application level it is a “Message” or simply “data”. An MPDU is a MAC PDU, in other words the packet as seen going in and out of the MAC. Confusingly there are two different types;

1. A-MPDU. Aggregate MAC Protocol Data Unit.
2. A-MSDU. Aggregate MAC Service Data Units.

We will focus here on the first which we saw used in Section 5.4. In a later test with similar hardware as before, but this time using OpenWrt as firmware in the same type of Asus router a capture was done, which is the case for this and the next section.

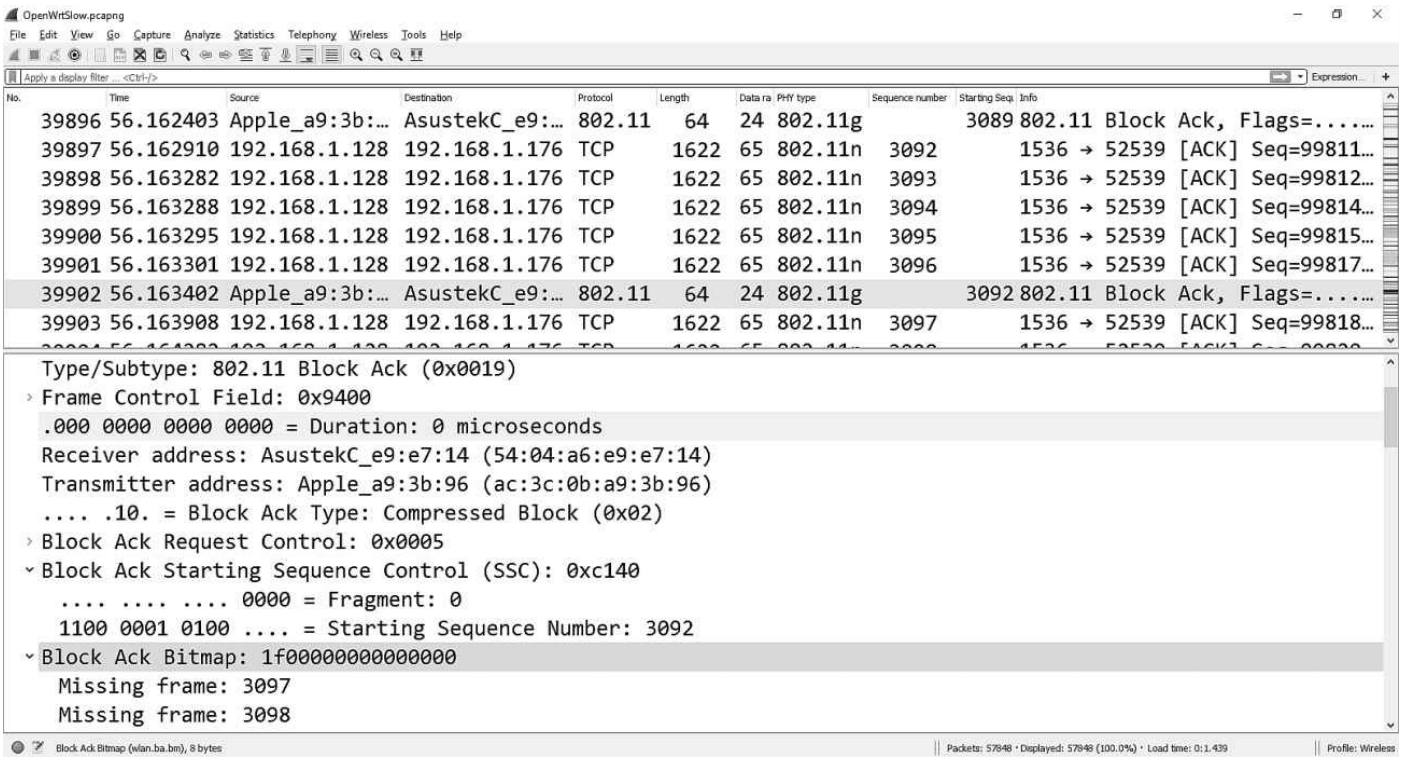



**Figure 5.10:** A-MPDU's and their contents

Figure 5.10 shows the following:

- Data is basically sent from 192.168.1.128 which is wired to the Access Point - the Asus - which is wireless connected to an iPad.
- Wireshark is setup to show the A-MPDU sequence numbers by looking in the dissection of one frame and clicking "Apply as Column". Similarly, "PHY type", "Data Rate" and the "Starting Sequence number" in the Block Acknowledge are chosen as columns.
- Frames 39897-39901 are in reality an 802.11 A-MPDU with sequence numbers 3092-3096, in other words 5 MPDU's.
- Frame 39902 is an 802.11 Block Acknowledge and as it is the selected frame it is dissected in the middle window. We see the starting sequence number is 3092. The Bitmap "1f000…" has five "1" bits and the rest of the 64-bits are "0". This corresponds to the sequence numbers in Frames 39897-39901. So we have a quick answer - a short RTT (Round Trip Time).
- At the bottom we see "Missing Frame 3097…3098…". This can be somewhat misleading. It is Wireshark interpreting the zeroes in the bitmask. As this is always 64-bits it is not a statement about lost frames, only frames not seen yet.
- The data is sent at 65 Mbps with PHY type 802.11n because the Asus and the iPad have agreed that this is fine, while the Block ACK's are sent at 24 Mbps PHY type 802.11g. This is because the setup of the Access Point (the Asus) allows for 802.11g. A "g" STAtion on the network uses this information, as it uses RTS and CTS. It is not a huge problem to send these relatively short frames at a third of the possible speed.

## 5.8 Channel Assessment

We have seen the use of Request-To-Send and Clear-To-Send several times. These terms

are ancient, but they have been updated for modern WiFi. 802.11n does not need these if it is alone in the world, but the concept has proven very robust - see Figure 5.11. The RTS is sent by the STA Transmitter-in-spe to the AP after there has been silence in the air for a time called DIFS - Distributed Interframe Space. Should this collide with a similar transmission from another STA or the AP, it will abort and retry a semi-random delay later. The RTS contains the expected transmission time, and the address of the receiver. This allows 3'rd parties to adjust their NAV - Network Allocation Vector - and sleep until the expected end of the final ACK. The CTS serves a similar purpose, but as it is sent from the AP it may reach other 3'rd parties (Hidden Terminal problem). Once we have a designated transmitter-receiver pair the "deadtime" between frames can be shorter. This is the SIFS - Short Interframe Space.

A STA or AP may use a "CTS-to-self" to protect against neighbors that do not understand the 802.11n protocol.

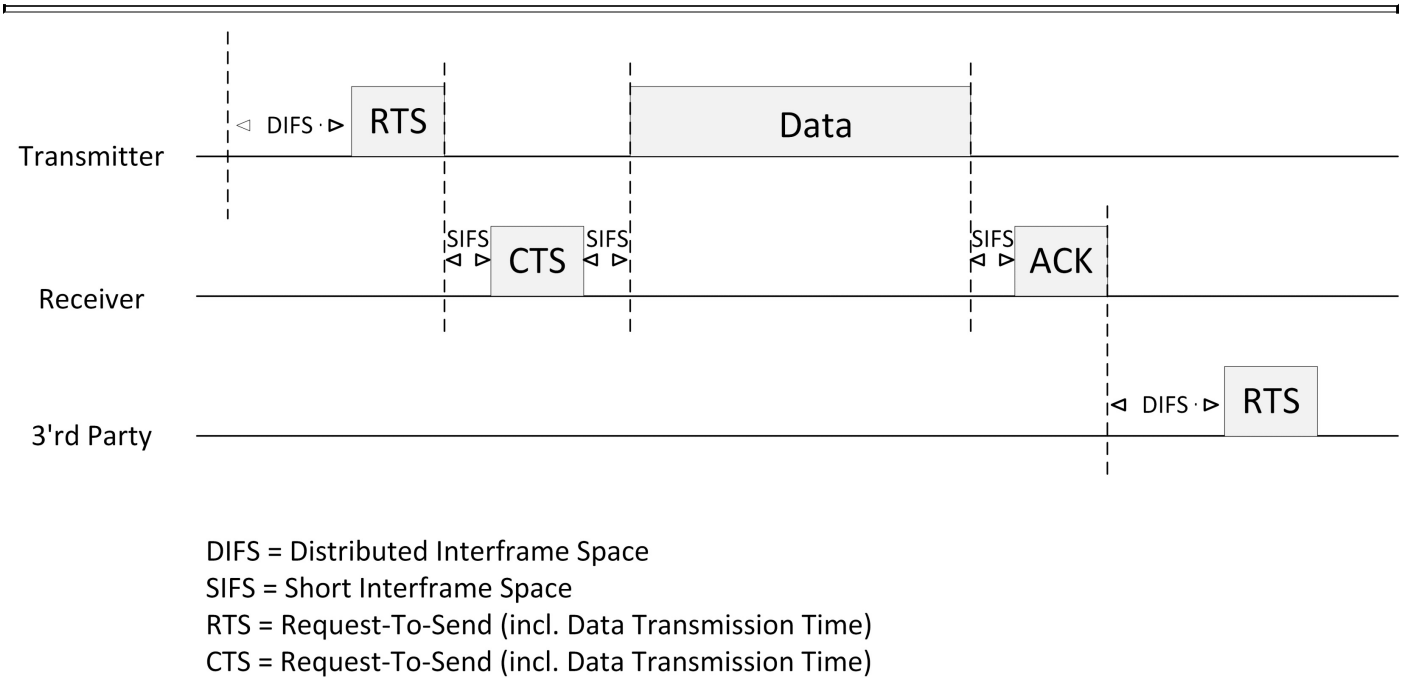



**Figure 5.11:** Channel Assessment with RTS and CTS

## 5.9 Bluetooth Smart

Some years ago there was a war between WiFi and Bluetooth about being the preferred Go-To-Internet network. One of the main reasons why WiFi won the war is that 802.11 (WiFi) is built on top of the existing Ethernet protocols as we have seen. This gives the connectivity needed. Since then WiFi has been all about performance, getting more and more bytes across. Naturally, power-savings has also been interesting as WiFi is heavily used on laptops, but when e.g. using a device like ChromeCast there is typically power at the AP as well as at the ChromeCast. And here you want your high-resolution Netflix without glitches. Meanwhile Bluetooth has found an increasing niche in the "small-device-to-phone" market. This is where you don't really need to route to the internet, you just want the headphones or whatever to connect to you phone. In the "small device" end of the market Bluetooth is thus well known and the main focus has been on battery

savings. Now everybody is gearing up for IOT, and the war is sort of restarting. WiFi is weaker than Bluetooth on the power-side, while Bluetooth Classic still really hasn't got the connectivity, and is way behind on performance.

The following table is a comparison of Bluetooth Classic and BLE - Bluetooth Low Energy a.k.a. Bluetooth Smart.

| Feature | Classic | Smart |
|---|---|---|
| Band | 2.4 GHz | 2.4 GHz |
| Distance | 30m | 50m |
| Datarate | 2100 kByte/s | 260 (650) kByte/s |
| Tx Power Max | 100 dBm | 10 dBm |
| Peak Current Max | 30 mA | 15 mA |
| Sleep Current Max | - | 1uA |
| Broadcast/Beacon Concept | No | Yes |
| Connect Up+Down | 300 ms | 3 ms |

The data-rates in Bluetooth Classic are governed by which mode is chosen. The first version of BLE had very short frames - 39 Bytes - but in the extended version in the 4.2 standard frames can be 257 bytes. This explains why the theoretical max rate is given as two numbers. The broadcast concept is new with BLE. Apple has an iBeacon concept that only broadcasts, and never does anything else. This is to be used in e.g. a department store, or a museum, where you walk around with your iPhone and catch different offers or news.

The war between WiFi and Bluetooth may be on again, but the rules have changed. The wireless IOT-devices we hear about at the moment are supposed to run very long on batteries, only transmitting or receiving a small blurb once a minute or even less frequently. A good guess is that the Bluetooth guys have decided to avoid a direct war, and instead bet on their strongest virtue - the lower power consumption. This is why they invented Bluetooth Low Energy - BLE. BLE is enhancing what they are already good at. But what about the connectivity? IOT devices does not just want to connect to your personal phone, they want to connect to the cloud. For this reason BLE in Bluetooth 4.2 has the following connectivity addenda:

| Name | What it is | Usage |
|---|---|---|
| GATT | REST API | Server |
| HTTP | Proxy service | Client |
| 6LoWPAN | IPv6 tunnel | Client & Server |

None of these can connect the device directly to the cloud, they all need gateway functionality, which may be in a phone or a more static, dedicated device. The GATT solution is for BLE devices that are mounted or located in a home or working place, where they are within reach of a "BLE gateway", with a built-in HTTP server, statically connected - e.g. via WiFi - to the internet gateway. This allows the owner/employee to

pick up his phone anywhere in the world and call up the HTTP-server in the Gateway, and via this the device, and check temperature, turn on heating, call up his bath-weight measurements from the last month etc. This is an existing solution.

The second solution is where a client inside the device needs to call something in the cloud. To do this it requires a “proxy”, this time with an HTTP-client, which could be the same gateway as before or a mobile phone. Surely the latter makes it more intermittent.

6LoWPAN is like a generally connected IPv6 device. Here the device is again near a gateway, and via this it connects more transparently to the cloud where it offloads data and gets new stored instructions. The gateway could be an application on a phone, or a static gateway. This solution is not tied up to HTTP - it could be anything that uses IP. If the IP-address is that of the phone, then the connection is local, like Bluetooth Classic. The connectivity, once configured, is thus better than the other solutions, 6LoWPAN needs the GATT part for the original discovery and setup phase. At the time of this writing iPhone does not support 6LoWPAN.

It is probably due to the new connectivity addendums that BLE is now being marketed as Bluetooth SMART. It sort-of turns Bluetooth Classic into Bluetooth DUMB, but it is already out there. There is still plenty of room for Bluetooth Classic, as it delivers a much higher bandwidth than the SMART does. This may change with Bluetooth 5.0 which may bridge the gap between SMART’s connectivity and Classic’s speed.

## 5.10 Further Reading

- Perahia & Stacey: Next Generation Wireless LANs
  A little dry but very informative

# Chapter 6
# Digital Filters

## 6.1 Why Digital?

In regards to embedded software we tend to see the world as full of analog signals that we want to measure or generate - digitally. Even if you have a sensor that outputs a digital value, it's because another embedded programmer has digitized it for you. A lot can be done without leaving the analog world, but if you are an embedded programmer you probably don't need much convincing - of course it must be done digitally. Still, let's recap the advantages of "going digital":

*As a kid I opened my tape-recorder (hopefully you have at least heard about these things). Inside it, was a lot of small things that could be turned with a screwdriver. I turned them all but it didn't really improve. Later I learned that these "potentiometers" were necessary to compensate for all the components that were not 100% the specified value. So somewhere at a production-site, someone had actually adjusted each of these. A LOT of work. Unfortunately - due to aging of the components this calibration ought to be done at regular intervals. And indeed professional measurement institutions are carefully marking their equipment with "next calibration date".*

- With digital signal processing there is little need for calibration. Imagine getting rid of (almost) all of these calibrations and instead have clone-able, unchanging specifications. That is what the digital world brings us.
- Digital Signal Processing is so much more flexible than analog. You can fit a lot of filters into a DSP (Digital Signal Processor) or plain CPU. In the analog case you would need a lot of components - either completely separate, or switch-able. Unfortunately, not even a solid-state analog switch is ideal. It would be nice if it was either an open or a closed circuit, but nothing is perfect.
- Field-upgradeable. Digital filters can be changed with a firmware-update. This is mostly a good thing.

Nevertheless, there are situations where you need analog circuits:

- Before A/D (Analog To Digital) conversion. According to the "sample-theorem" your sample-frequency must be at least twice that of any frequency contained in your signal. Typically, the only way to assure this is to insert a low-pass filter before the A/D-converter. Such a filter is called an *anti-aliasing filter*.
- Likewise after D/A-conversion you need a *reconstruction filter* that cuts off all frequencies generated by the conversion above half the sample-frequency.
- If you want to measure on e.g. a small 8 kHz signal that is "hiding" in noise with a large amplitude - e.g. 50 Hz or 60 Hz power-line noise, you need a filter that removes say everything below 1 kHz. Without this filter your A/D might saturate, or it might simply not have the dynamic range to show you the small signal.
- It is not always so that the thing you want to measure gives you a nice "stiff" voltage that matches the range of your A/D converter input. Very often you need to *condition*

your signal first. This may require amplification, current-to-voltage-conversion or at least buffering. Similarly, your output often needs to be, at least, amplified.

So yes, we want to be digital, but there are a few - very important - components that needs to be analog. No chain is stronger than the weakest link. This includes the measurement chain.

Even though you do not include or write any Digital Signal Processing code there is a good chance that your device relies heavily on DSP. If you have WiFi, there is some very advanced stuff going on - but it will be in dedicated hardware - it simply has to be, in order to have decent performance and not use too much energy. Implementing e.g. IEEE 802.11n is a perfect task for dedicated hardware. It does not need the flexibility praised earlier. It just needs to do the same thing over and over. Also there is some fast number-crunching in your Ethernet MAC that calculates the Ethernet 32-bit CRC (Cyclic-Redundancy-Check). This is not advanced signal processing, but your CPU would be severely disabled if you had to do it in software.

## 6.2 Why Filters?

There are reasons that we have embedded programmers and we have DSP-programmers. The DSP-domain is huge and heavily loaded with mathematics. On a PC it is very simple today to call an FFT (Fast-Fourier-Transform) function in an *intel* library. But how to get from this to a “Power-Density Spectrum”? - how many lines do we need? how many averages? and which window function? This book is for embedded programmers doing IOT, and we will stick to filters in the time-domain. You often need to do some filtering, simply to compress the amount of data. These data are typically passed on to the “cloud” where the advanced Digital Signal Processing may take place - depending on your application. To enable you to implement filters we will also look into number-representations. This is the basics for any Digital Signal Processing.

## 6.3 Basic Signal Analysis

This is a practical guide, so we are skipping most of the theoretical explanations and move directly to the facts and the derived rules. Clarity is prioritized over absolute detailed correctness.

### 6.3.1 About the Sample Frequency

In general we use the term $f_s$ for the sample-frequency: $f_s = (1/\Delta T)$
- where $\Delta T$ is the time between samples from the A/D-Converter. In POTS (Plain-Old-Telephone-System) $f_s = 8$ kHz. This means that $\Delta T = 125$ µs.

Sometimes you will see the symbol ωused. This is the “*angle-velocity* of the rotating unity-vector”, which is used all the time in the formulas. What we need to know is simply: $\omega = 2*\pi*f$. Another mathematical trick is to show a frequency axis with positive and negative frequencies. This is just a trick - but it very nicely explains phenomenona such as aliasing. Aliasing is what happens if you violate the “sample-theorem”: Signal content

from the original analog signal is “mirrored” around the $f_s$, and spoils your data. In other words, the sample theorem says: All signals above $f_s/2$ must be removed before sampling (or in reality be below a certain level). This is also called the “Nyquist-Criteria”, and $f_s/2$ is known as the “Nyquist-Frequency”,

DSP Algorithms - including filters - are completely relative to the sample-frequency. This means that if you have created a lowpass filter with a cut-off frequency at 3 kHz with $f_s$=20 kHz, and you raise $f_s$ to 40 kHz, the same filter will now cut-off at 6 kHz. For this reason you will often see *normalized* filter-characteristics where DC is 0 and $f_s$ is 1.

### 6.3.2 Time and Frequency Domains and their Duality

There is a one-to-one correspondence between a signal in the *Time-Domain* and the *Frequency-Domain*. This means that it is possible to go from one to the other - and back again. The time-domain is what you see on an oscilloscope with magnitude as function of time - the frequency domain is with frequency on the x-axis, and the y-axis is typically magnitude and/or phase. Sometimes it actually makes sense to convert the signal from the Time-Domain to the Frequency-Domain, perform the algorithm here and go back to the Time-Domain.

A single vertical line in the Frequency Domain corresponds to an everlasting sine in the Time-Domain - see Figure 6.2. If this single line is located at f=0, then we have a completely horizontal line in the Time-Domain - a DC. Likewise a single vertical line in the Time-Domain corresponds to a flat spectrum - the same level on all frequencies. In fact it is a good rule-of-thumb that something that is very “edgy” or “spiky” in one domain is very wide in the other.

Signals in the time domain are real - as opposed to complex[1] . However, in the Frequency Domain they can easily be complex.

Any signal can be decomposed to a number of “sinusoids” - or sines in daily speech. Any signal that is repetitive, has a “discrete” spectrum - a number of lines, so-called *harmonics*, each *n*f*, where “n” is a natural number (1,2,3..) and *f* is the repetition frequency. This is also known as the *base* frequency or the first harmonic.

### 6.3.3 Analog and Digital

In the analog world we deal with *continuous* signals, whereas in the digital world they are *time-discrete* - another word for sampled. In the analog/continuous world the time signal is noted like x(t), the transfer-function of a system is noted H(f), and a Frequency Spectrum is noted as X(f). In the digital/discrete world the time-signal is noted as x[n], the transfer function is known as H(z) and the resulting Frequency Spectrum is also here noted as X(f). Note that the spectrum of a sampled signal is generally continuous.

In both worlds the transfer function can be represented as a fraction with a polynomial in the nominator and another in the denominator. The roots to the first is known as *zeros*, while the roots of the second is known as *poles*. Knowing the positions of these in a Cartesian coordinate system in the analog case, or in a polar-plot in the digital case can tell

you everything about the transfer function. Very briefly on the analog domain: Imagine that you are the magnitude function being drawn, following the frequency axis from 0 to infinity. Any pole or zero ahead of you is insignificant. Once you pass a zero you start rising with 20 dB/decade (or 6 dB/octave) and when you pass a pole you descent in the same way.

The same way a given signal has one representation in the Time Domain and another in the Frequency domain, the *impulse response* of a system, h(t), in the Time-Domain, maps to the *transfer function* of a system, H(f). This is the same in the analog and the digital world - although the impulse response is called h[n] in the digital case.

The impulse-response is defined as the output of a system in the time-domain, when subjected to a Dirac-impulse - an infinitely narrow pulse with an area equal to 1.

H(f) is normally defined as Y(f)/X(f) where Y(f) is the frequency spectrum of the output and X(f) is the frequency spectrum of the input. Thus the name “Transfer Function” is well chosen, as it describes how the spectrum of an input signal is transferred by going through the system. We can now deduct that a system with a long impulse-response has a narrow frequency response.

It is probably not a surprise that h(t) is real, while the transfer function, H(f), of a system is complex. One way to look at this complex function is to show Magnitude as function of Frequency and Phase as function of frequency. When we show both it is often known as a Bode-plot, see Figure 6.1.

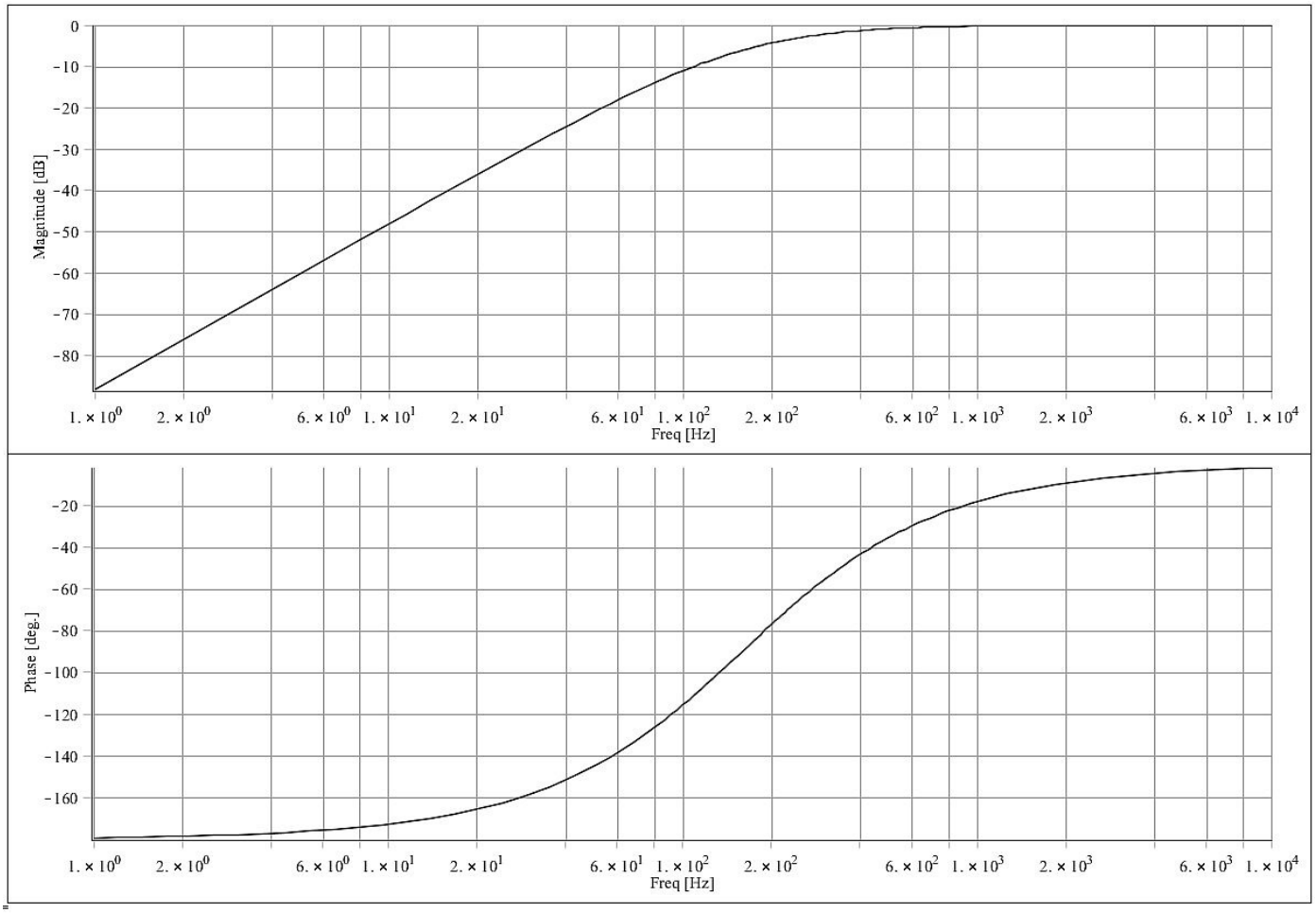



**Figure 6.1:** Bode plot - Magnitude and Phase - of an IIR-filter

### 6.3.4 More Duality

There are many forms of duality in Signal Analysis. We have dealt with the duality between Time and Frequency Domains as well as the duality between Analog/Continuous and Discrete/Sampled systems. The third duality is between the frequency spectrum of a single pulse, and the frequency spectrum of the same pulse, repeated indefinitely. The spectrum of a single pulse is continuous, while the spectrum of the same repeated pulse is discrete. But apart from that they are the same. This is the fact that the success of the Fast-Fourier-Transform is based on. A true discrete Fourier transform has a number of computations that relates to $n^2$ where "n" is the number of samples. The FFT only relates to n*log(n). To apply an FFT, however, requires some tricks, that can degrade your measurement - but that is out of the scope of this book.

A single square-pulse in the Time-Domain becomes a sin(x)/x function in the Frequency-Domain. See Figure 6.3. The wider the pulse is in time, the narrower the sin(x)/x becomes. The wide pulse also becomes higher in the frequency domain. As the height of the sin(x)/x at the center is the DC-value this is no surprise. The slimmest pulse in Figure 6.4 is 1 high from -1 to 1, and we calculate over a span from -10 to 10. This gives us a DC of 0.1. The other pulse is double as wide - giving a DC of 0.2.

A true everlasting square-signal with frequency *f* can be created from an infinite number of harmonics of the *Base-Frequency*. If the square signal has a duty-cycle of 50% there are only odd harmonics. This is shown in Figure 6.5. It is also shown progressively in figures 6.6 and 6.7 where we are going the other way and actually are building the square-wave from more and more sines. Note that these figures are all in the Time-Domain. In Figure 6.6 a) we see the ideal square-wave together with the first harmonic. They clearly have the same frequency but otherwise do not look much alike. In b) we again see the ideal square wave, now together with the first 3 harmonics added as one wave and the first 5 harmonics added as another - slightly better - approximation. Figure 6.7 a) shows the first 21 harmonics added, and b) shows the first 111 harmonics added - both also together with the ideal square-wave. It is clear that the more harmonics we include, the closer we get to the square-wave. It can be proven that any signal can thus be "built" from sines.

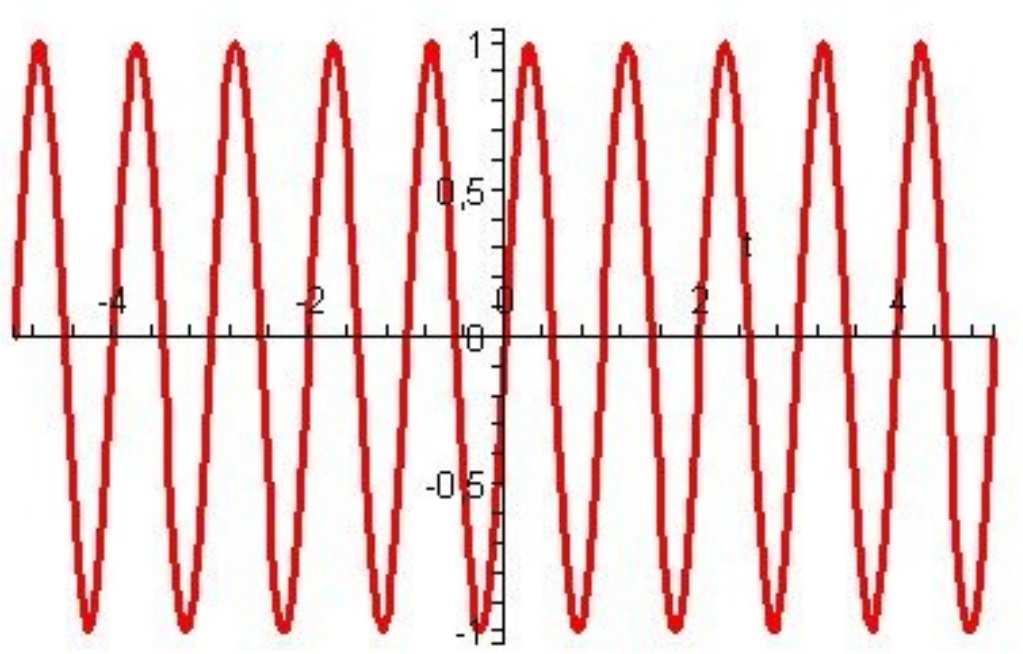
### 6.3.5 A Well-behaving System

A "well-behaving" system is linear. This means that if we add two signals in the time-domain, the frequency spectrum of the resulting signal will be the same as adding the spectra of the two original signals. If a signal is amplified by a factor A, the lines in the Magnitude-Spectrum are also multiplied by A, while the phase is unchanged.

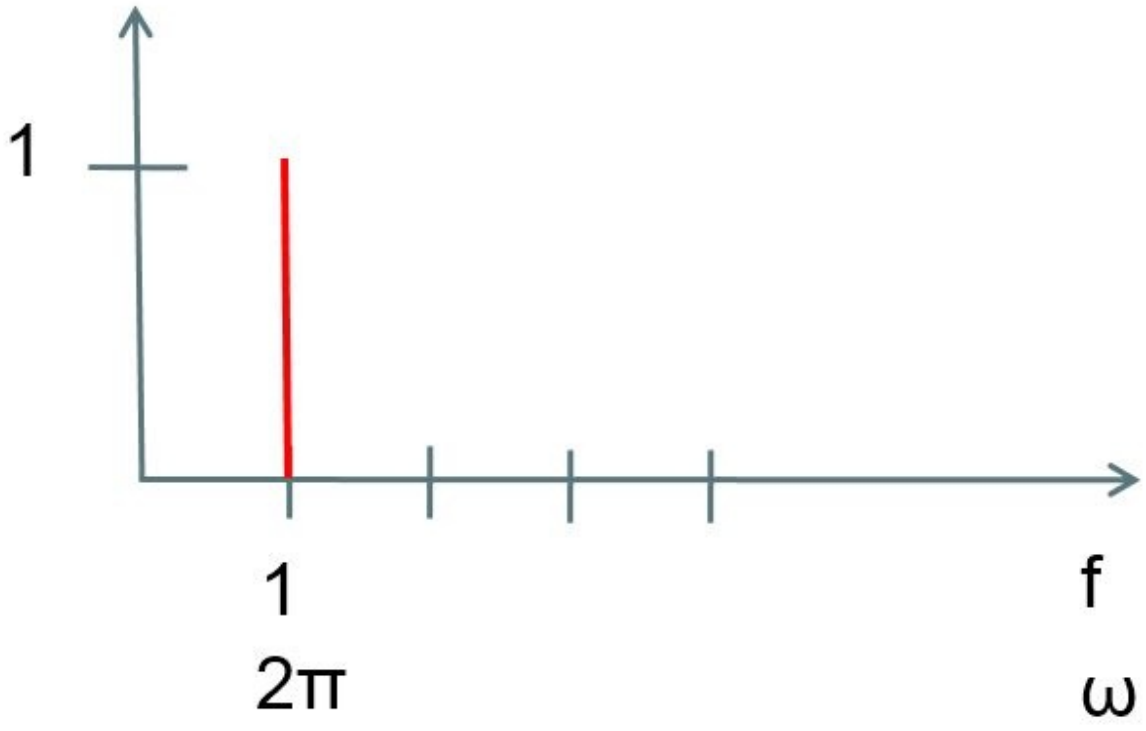
A "well-behaving system" is Time-Invariant. This basically means that given the exact same input at 5 o'clock as you gave it at 4, you will get exactly the same output.

The typical thing that can make a system not well-behaving is very often "saturation". If you try to add two analog sines with an amplitude of 7V each, and your power "rails" are +/- 10V, then you will run into problems. The same thing can happen in the digital

domain.

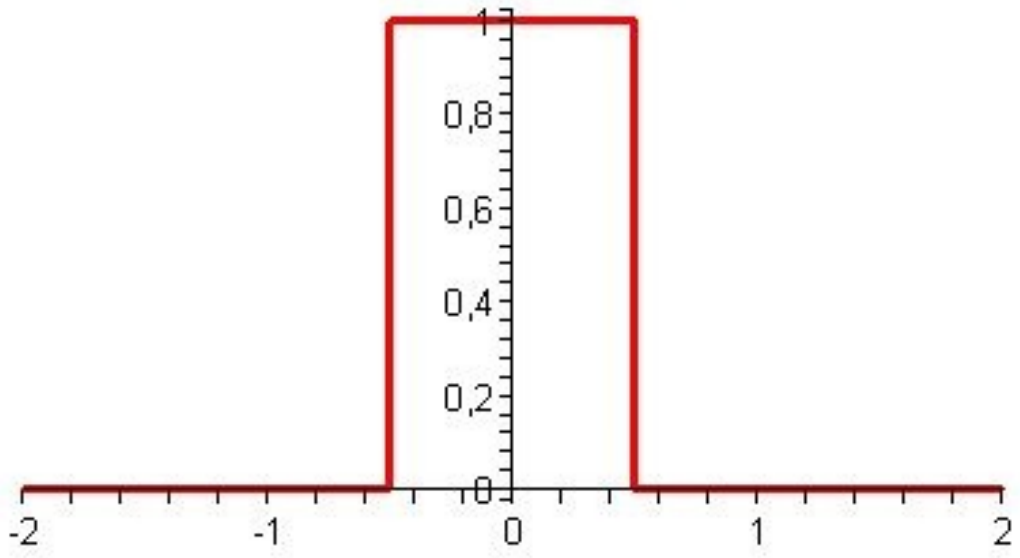



(a) Time Domain

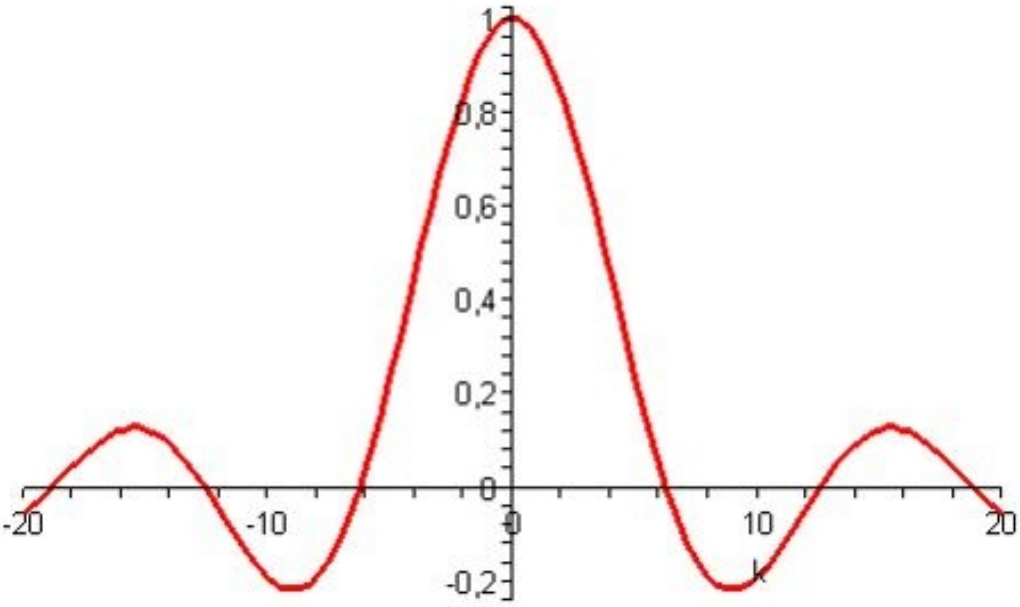



(b) Frequency Domain

**Figure 6.2:** An everlasting sine of 1 Hz

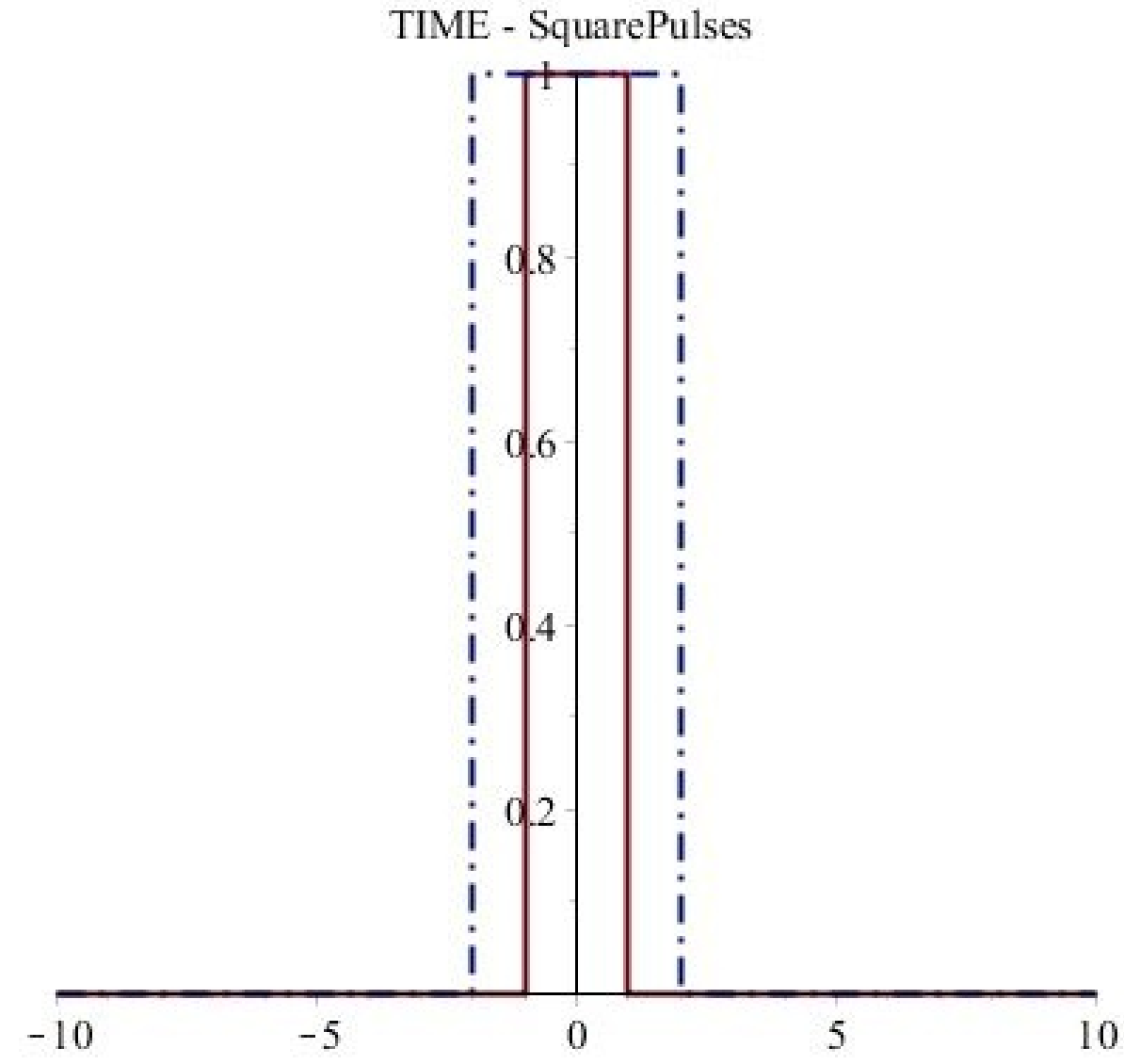



(a) Time Domain

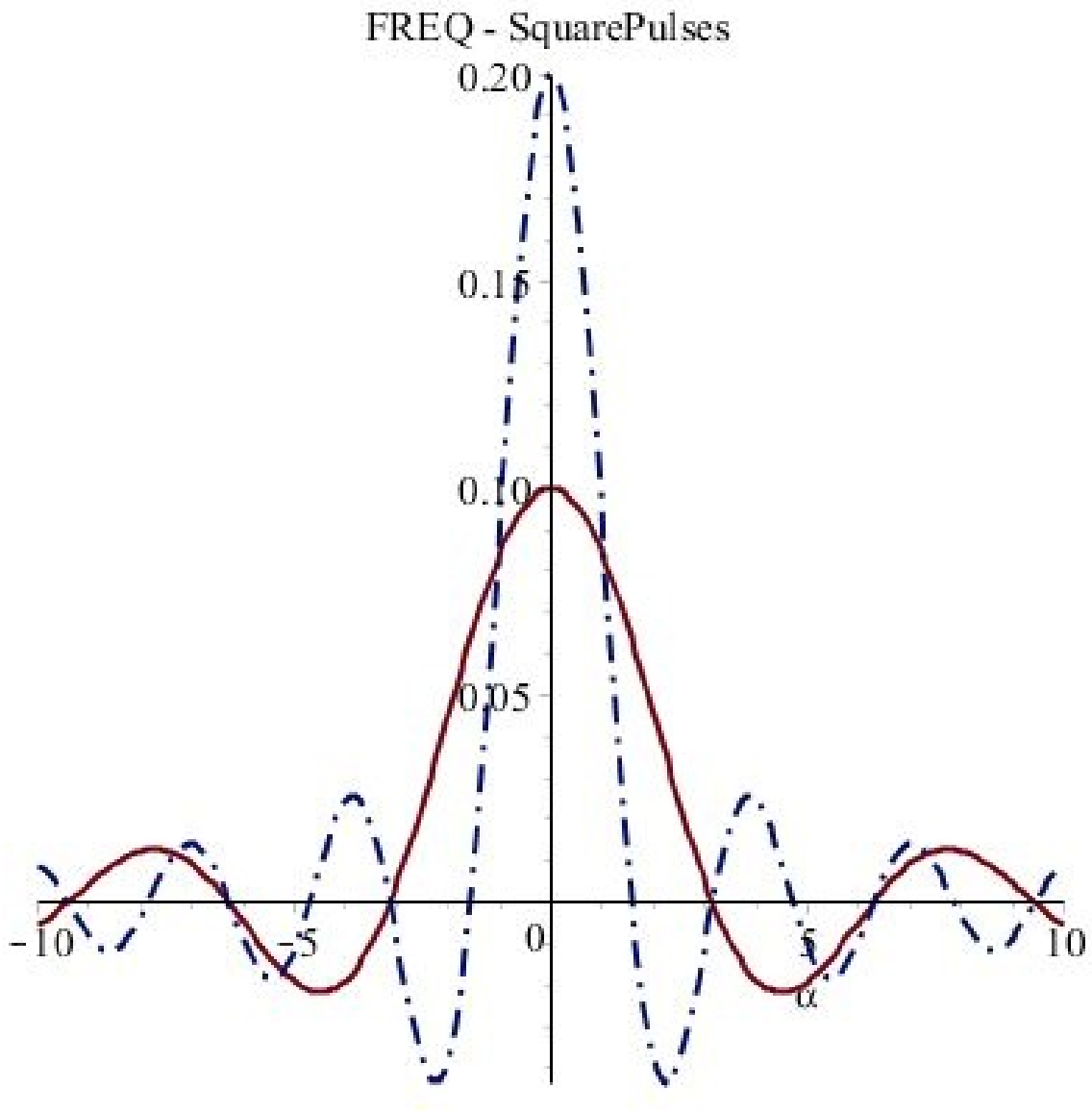



(b) Frequency Domain

**Figure 6.3:** A single square becomes sin(x)/x in the Frequency Domain



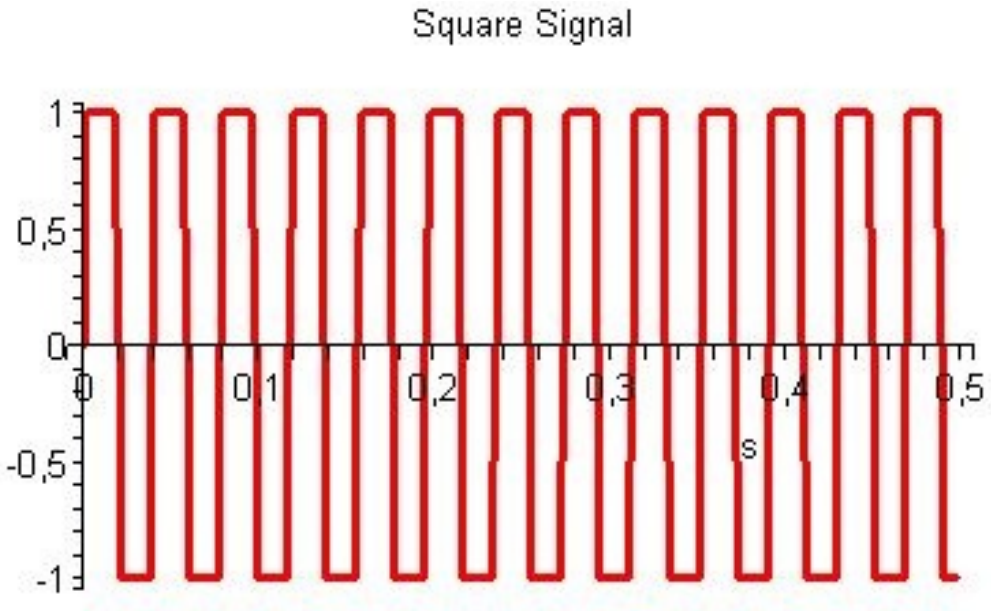

(a) Time Domain




(b) Frequency

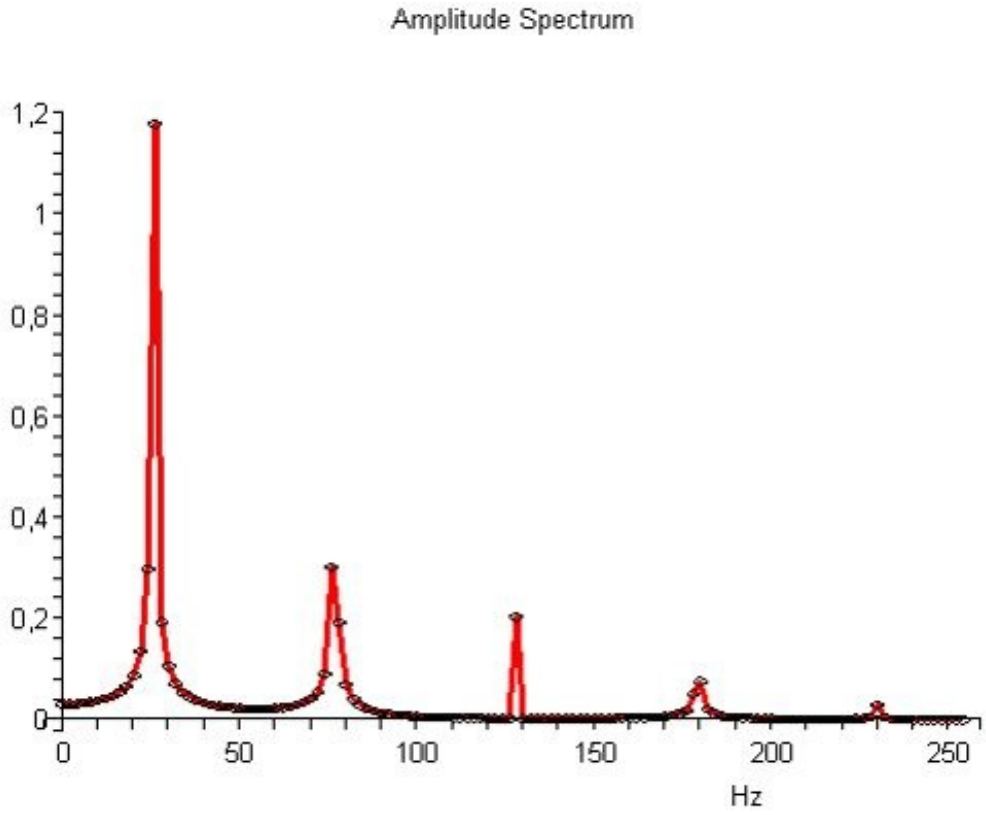
Domain

**Figure 6.4:** Widest pulse - slimmest & highest sin(x)/x



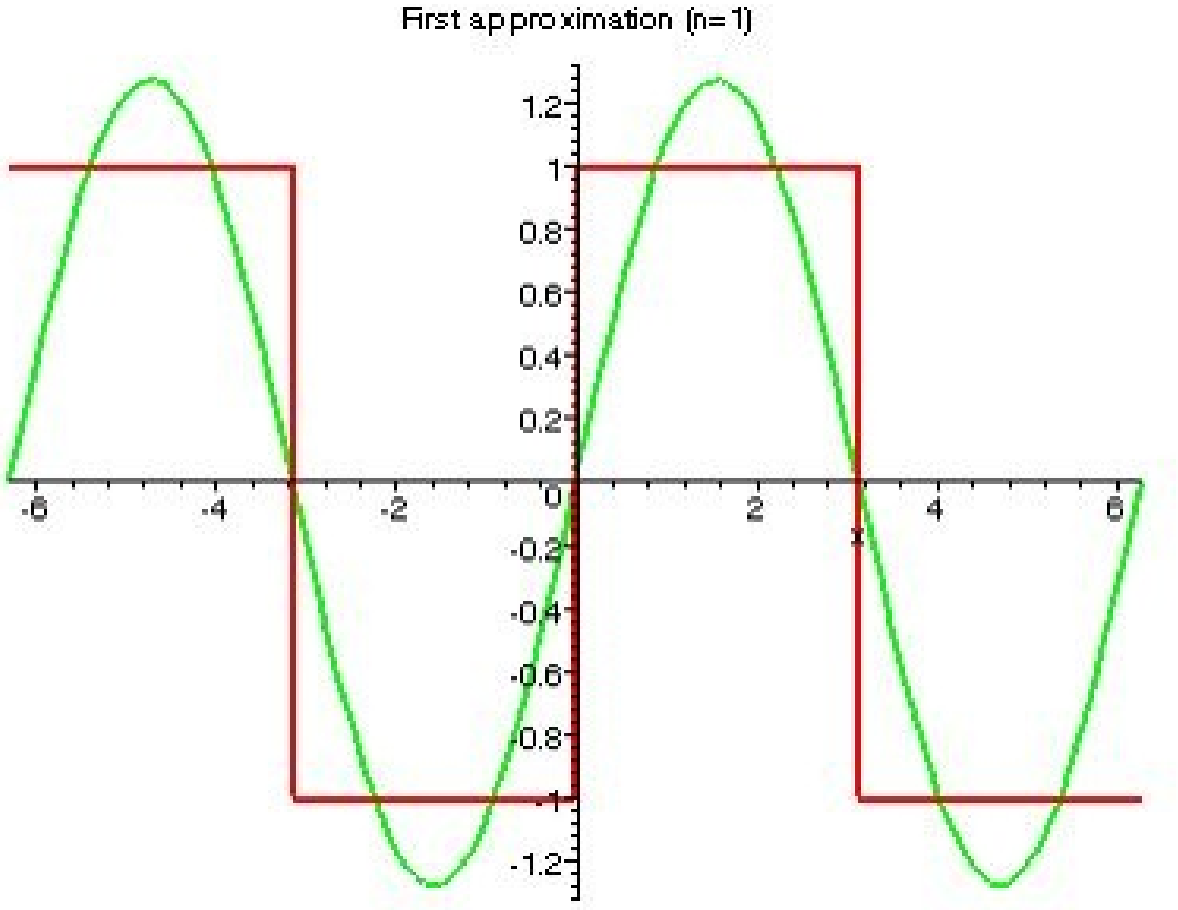

(a) Time

Domain

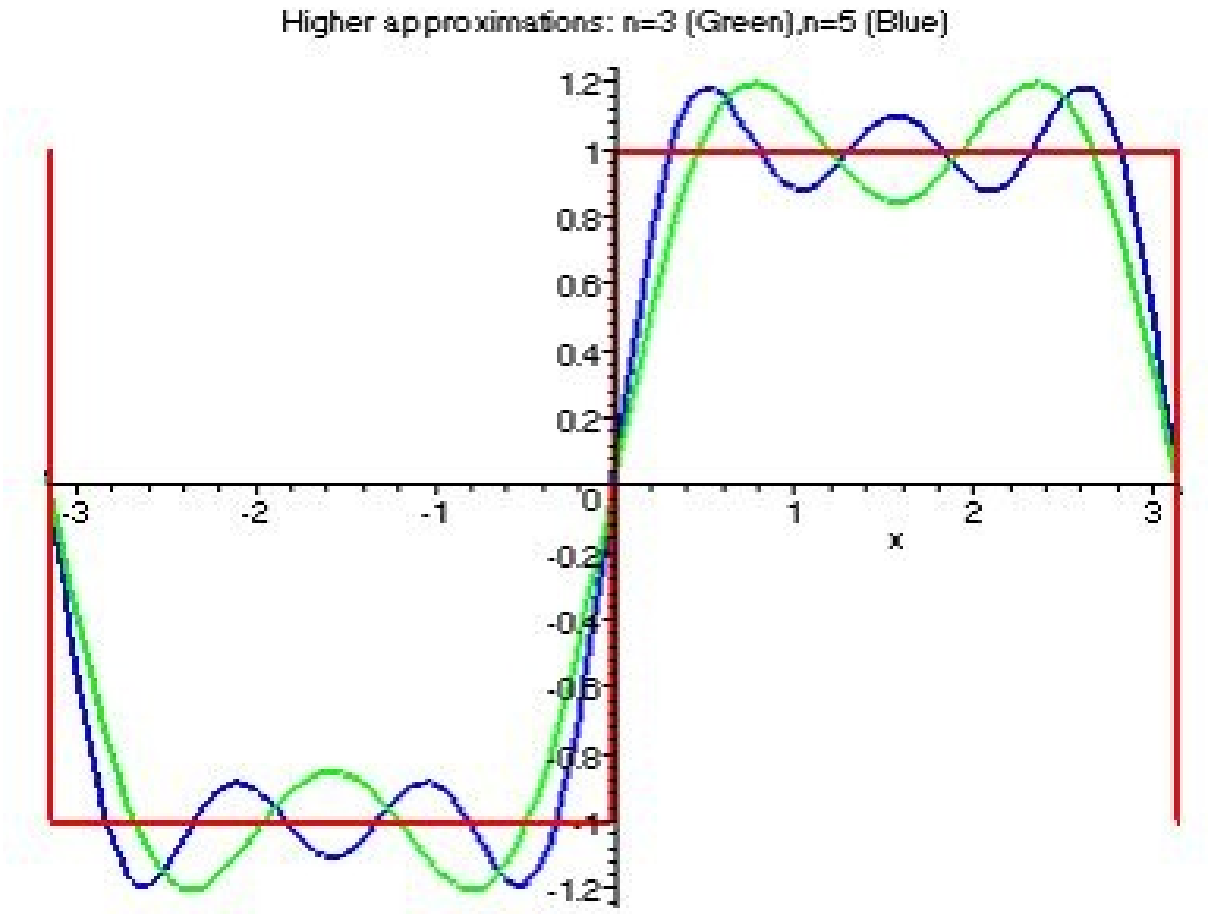



(b) Frequency Domain

**Figure 6.5:** A square-wave duty-cycle 50%

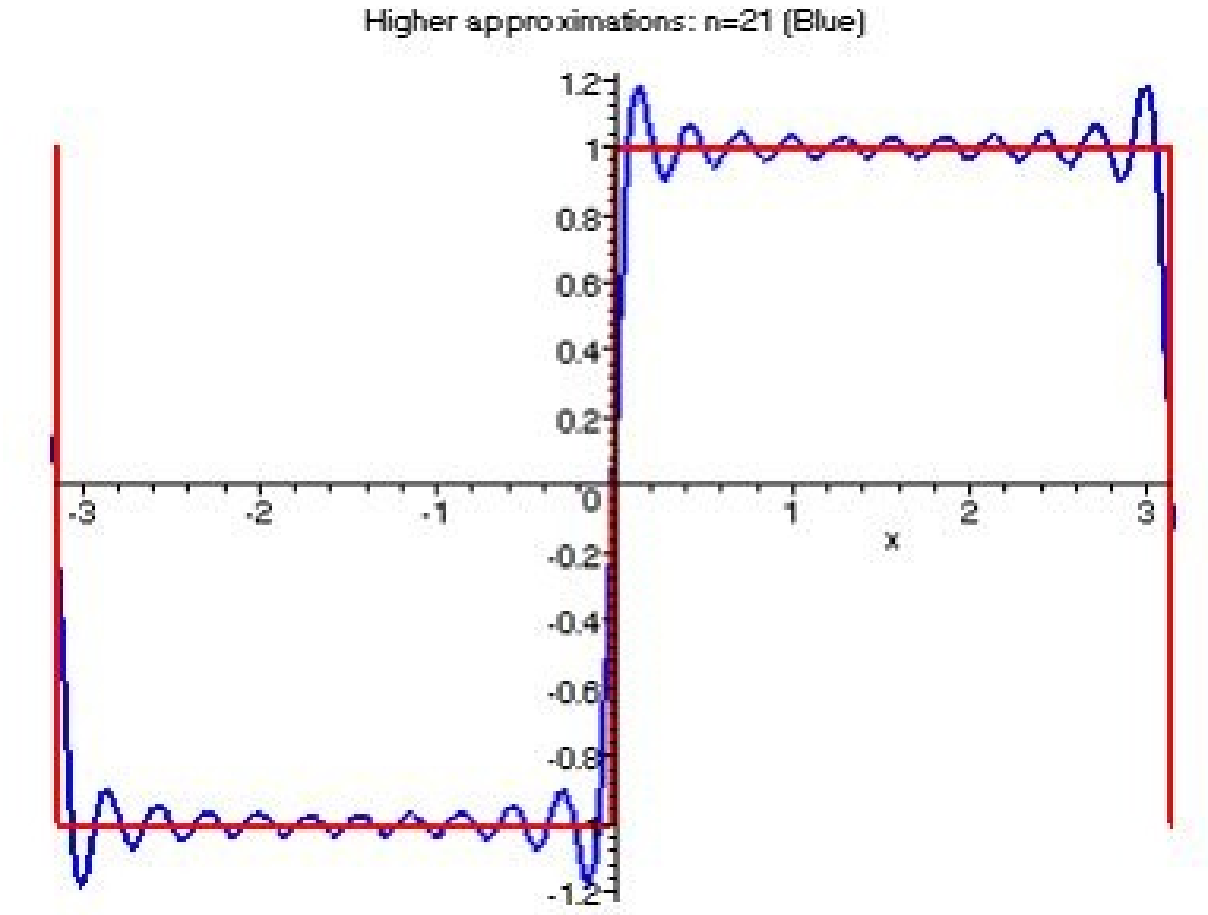



(a) 1'st Harmonic

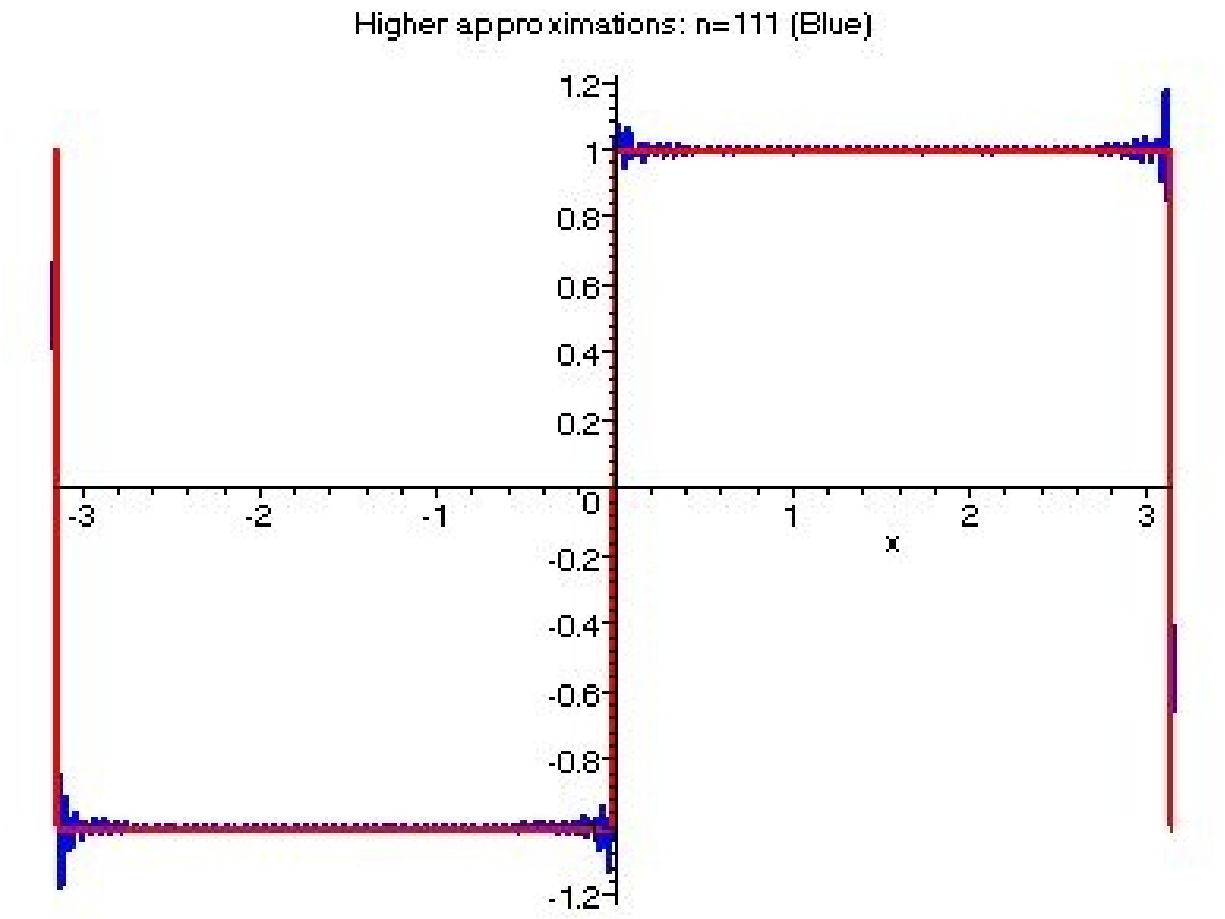



(b) First 3 and first 5 Harmonics

**Figure 6.6:** Building up a square-wave - all Time Domain




(a) First 21 Harmonics




(b) First 111 Harmonics

**Figure 6.7:** Building up a square-wave - all Time Domain II

## 6.4 IIR filter Basics

In the previous section we briefly touched on the “Impulse Response” of a system. The term IIR actually means “Infinite Impulse Response”. It sounds advanced, but in reality all analog filters are of the IIR type. As you may know, a simple RC-circuit where a Capacitor is loaded via a Resistor has a “time-constant” $\tau=RC$. This gives us the time to load the Capacitor to approximately 63% of the ultimate value. We are closing in on this ultimate value in an asymptotic way, as you probably have heard many times. But that means that we will never get there. Of course this is theory, it doesn’t bother an analog or a power engineer. But there you have it - that’s what IIR means. IIR-filters are very popular in small DSP systems for the following reasons:

- It is possible to convert a standard analog filter to a digital IIR filter. In the youth of Digital Signal Processing this was very important, because filter-design programs were very rudimentary - close to non-existing. It was very practical that you could reach for your filter-table-book on your shelf, find a filter with a fitting cut-off-frequency, pass-band ripple, stop-band damping and roll-off, and convert it to digital. This conversion is actually a little complex - known as the bi-linear transformation, but if you only have to do it once, it’s doable. Don’t forget that even though the youth of DSP was after the invention of internet it was before the World-Wide-Web. That explains why there even was a filter-table-book and it was used…

- Many digital systems replaced existing analog systems, and were expected to perform as these old systems. In many cases there even was a standard prescribing a specific analog filter, that you could convert to a digital IIR-filter. This is still the case in some domains.
- IIR filters typically require fewer calculations than the alternative. Likewise, it also uses less memory - program as well as data.
- There is a tendency to focus on magnitude, not phase. Analog filters are of many interesting types - Butterworth, Chebyshev, Elliptical and Bessel are the most common. They all present some elegant mathematical solutions to filter problems, such as having the minimum pass-band-ripple or the steepest roll-off. Some of them has decent phase characteristic, some are terrible, but none are perfect.

Apart from the above more or less historical benefits - here are a few more facts on IIR:

- IIR filters can go into oscillation if designed poorly. This is typically due to quantification noise (round-off errors) when fitting the filter coefficients into a limited number of bits. Oscillations can be really hefty as a real oscillator, or, more commonly, so-called “limit-cycles”. This is when e.g. filtering an impulse and the filter should “die out”, but instead keeps cycling around the steady-state value with a small but annoying magnitude.
- The reason why the RC-circuit in the analogy starts charging fast, but then charges slower, is that the current through the Resistor is dropping as the voltages on either side of the resistor are getting closer to each other. In other words, there is a feedback from the output. You could also say that the near past is influencing the near future. The same can be said about IIR-filters.

## 6.5 Implementing IIR

The general formula for an IIR filter is:

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right)$$

Typically, the filter is scaled so that $a_0$ is 1. y[n] is the current sample in the output, whereas x[n] is the newest sample in the input to the filter. y[n-1] is the previous output and x[n-1] is the previous input sample. So the newest P input samples are each multiplied with a coefficient that is specific to its “newness” and likewise the newest Q outputs are multiplied with other coefficients specific to their “newness”. And then it is all added together. You might not want all the intermediate products, instead you would clear an “accumulator” first and then add the products to this one by one. Hence the focus on the time it takes to do a MAC - Multiply-Accumulate.

A typical way to visualize this is seen in Figure 6.8. $Z^{-1}$ is the mathematical way to show a delay of one sample. Thus the IIR-filter shown needs to keep the latest three input samples and the previous two output samples to do a calculation of the next output sample. It needs to perform 5 MAC’s per output sample. This is not much, and it is quite representative - known as a *Biquad* filter. Looking at this figure you may understand why IIR-filters are popular. Once you have the filter-coefficients, the implementation is pretty

simple. IIR filters of higher order than two, could be implemented by simply extending the concept shown in Figure 6.8. However, as stated earlier, IIR-filters can become unstable, oscillating, and it has proven more stable to cascade Bi-Quads as the one shown instead. It is recommended to start with the poles closest to the x-axis - paired with similar zeros. Sample code for a BiQuad is shown in Listing 6.1. There are many filter-design programs, but before you go out and buy an expensive solution, that may even require a lot of programming and math understanding, scan some of the available free programs. Figure 6.9 is from a free program by IowaHills. The example demonstrates a lot of the standard filter design-criteria:

| **Parameter** | **Value** | **Meaning** |
|---|---|---|
| Basic Type | Elliptic | Equiripple. Steep flank |
| Omega C | 3 dB Cut-off | Break-Frequency |
| Gain | 0 dB | Overall amplification |
| Sample Frequency | 1 | Normalized |
| Ripple | 0.02 dB | Pass-band ripple |
| Stop Band | 60 dB | Stop-Band Damping |
| Poles | 4 | 2 BiQuad’s |

In Figure 6.9 the “Coefficients” is checked which leads to the text at the right side. This contains:

- Coefficients for Biquad-Sections in the optimal order.
- The N coefficients from the basic IIR-formula. Could be used directly.
- The 4 Zero’s. See Figure 6.10.
- The 4 Poles. See Figure 6.10.

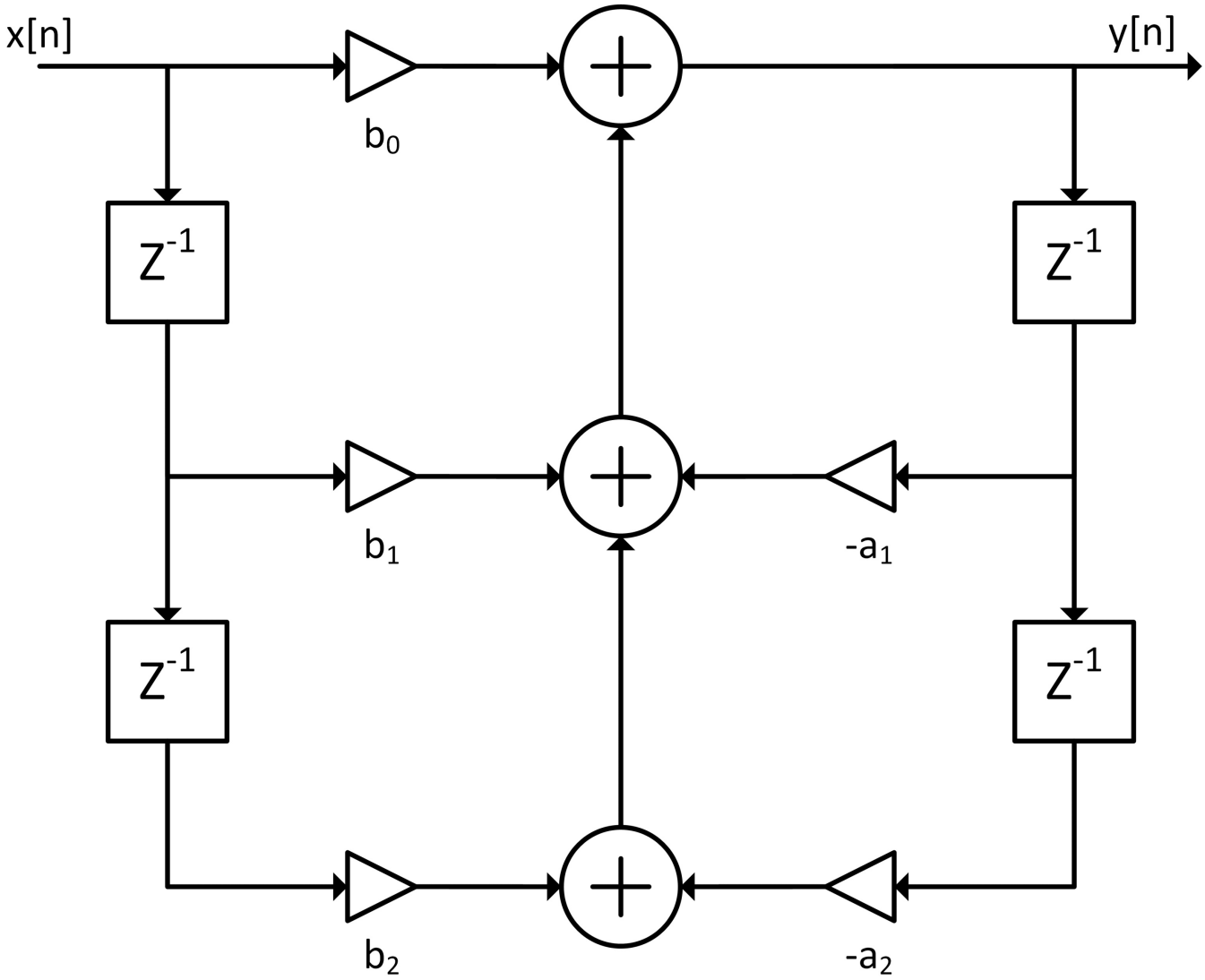



**Figure 6.8:** BiQuad IIR Filter Implementation

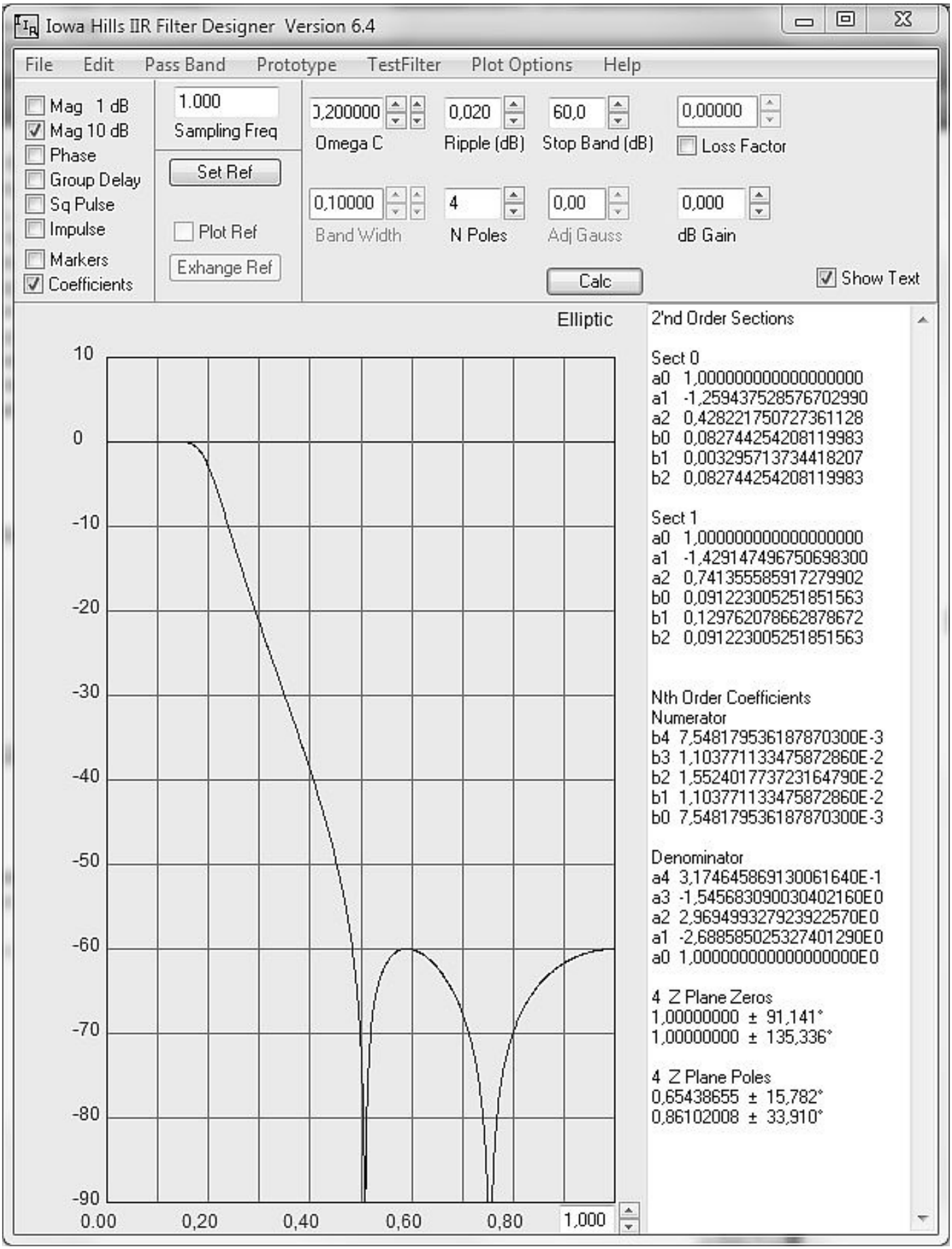



**Figure 6.9:** IIR-Filter Design by IowaHills

If the filter is to be stable the poles must be inside the unit circle. Poles as well as

Zeros are always either real or complex conjugate pairs.

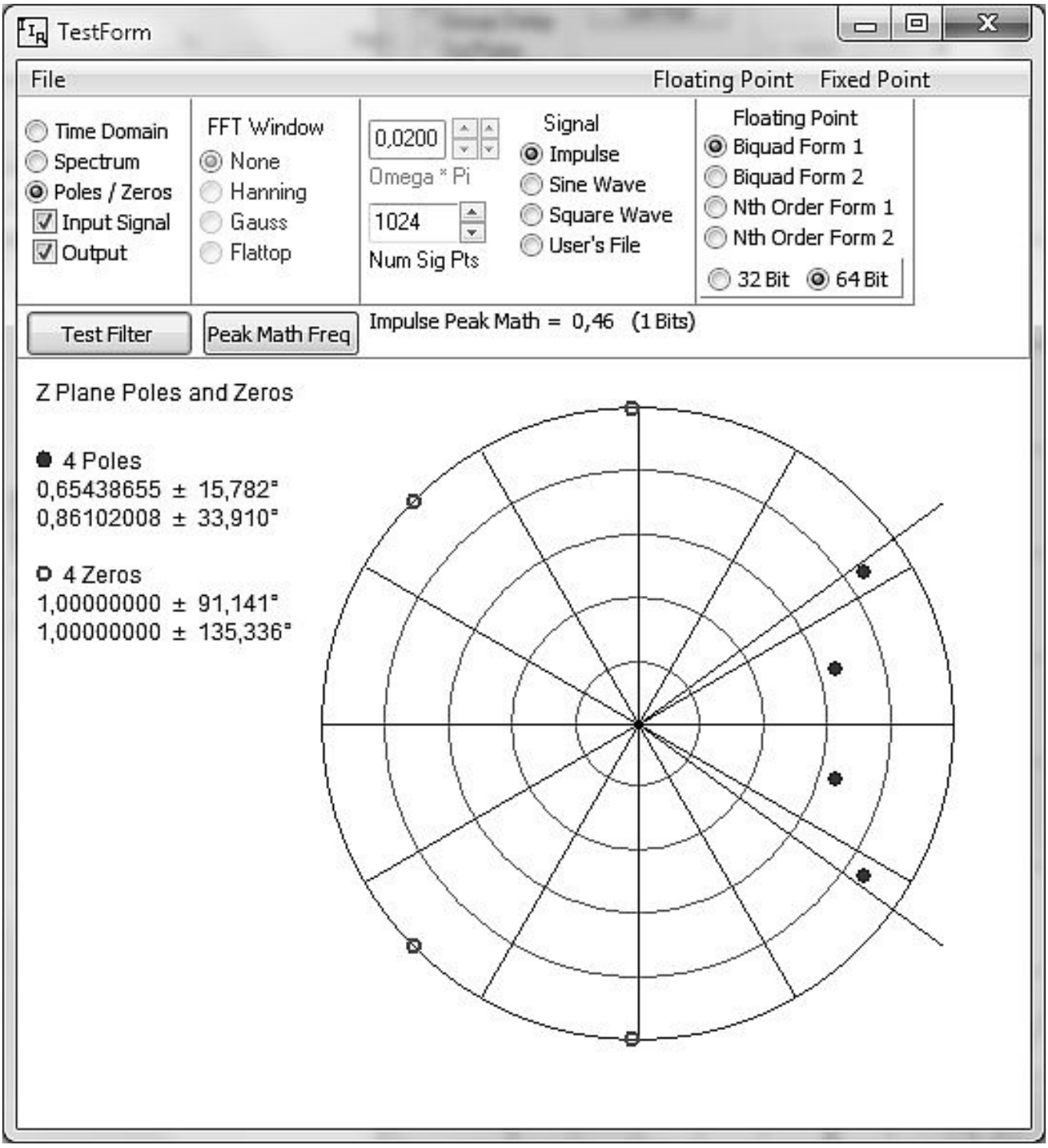



**Figure 6.10:** Pole-Zero Plot of the Elliptic Filter

BiQuads are so popular that you can find small DSP's or Mixed-Signal chips that has them as ready-made building-blocks - just waiting for your coefficients.

```
01 …
02 typedef struct
03 {
04         float a0, a1, a2, a3, a4;
05         float x1, x2, y1, y2;
06 }
07 biquad;
08 …
09 float BiQuad(float sample, biquad * b)
10 {
```

```
11      float result;
12
13      /* compute result */
14      result = b->a0 * sample + b->a1 * b->x1 + b->a2 * b->x2 -
15               b->a3 * b->y1 - b->a4 * b->y2;
16
17      /* shift x1 to x2, sample to x1 */
18      b->x2 = b->x1;
19      b->x1 = sample;
20
21      /* shift y1 to y2, result to y1 */
22      b->y2 = b->y1;
23      b->y1 = result;
24
25      return result;
26  }
```

**Listing 6.1:** BiQuad IIR filter code. Created by Tom St. Denis

## 6.6 FIR Filter basics

In contrast to IIR-filters, FIR-filters have no analog counterpart. They simply can only be realized digitally. This means that we cannot use an analog-filter as a starting point. FIR filters can basically also do lowpass, bandpass and highpass - and combinations hereof. The acronym means “Finite Impulse Response”. This has no feedback from the output. Due to the missing feedback, FIR filters cannot go into oscillation or “limit-cycles” like IIR-filters. The general formula for an FIR filter is:

$$y[n] = \frac{1}{a_0}(\sum_{i=0}^{P} b_i x[n-i])$$

The formula is exactly the IIR formula without the last sum with all the previous outputs fed back. Correspondingly the block-diagram is also the same as in the IIR-case, see Figure 6.11, only there is no right side. In this case it is **not** a representative filter. Without the feedback the filter has to contain a lot more elements. However, implementing a FIR filter is done by directly extending Figure 6.11 - no cascading is needed.

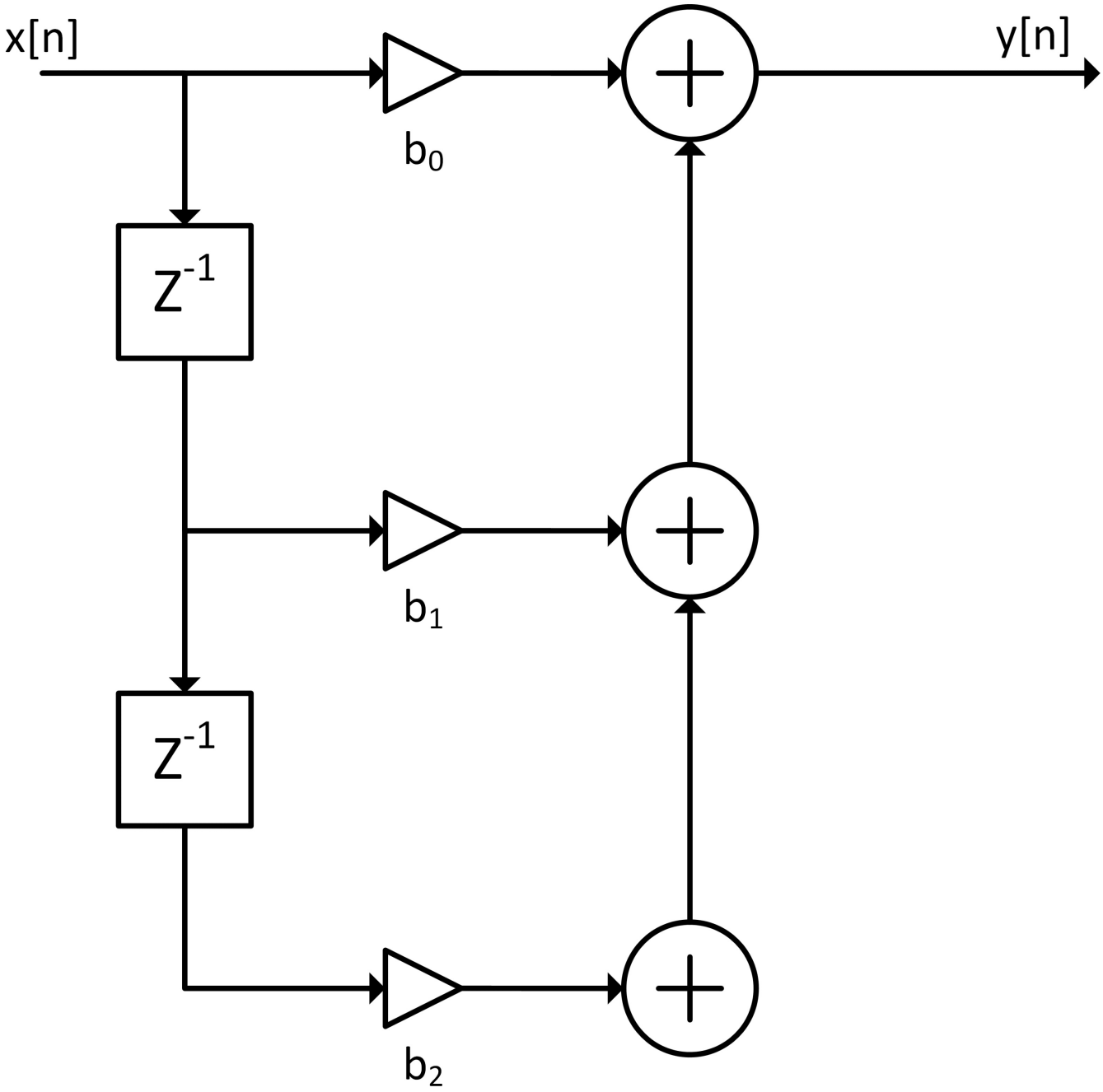



**Figure 6.11:** 2. Order FIR Filter

FIR filters thus typically require more resources than IIR-filters. The absolutely most interesting fact on FIR filters is however, that they easily can be created to have a linear phase. This means that if you plot the phase of the filter's transfer function - H(f) - it will be a straight (but steep) line. This simply means that the filter delays the whole signal with a time that is equal to the slope of this line. This is typically not a problem. In most cases it just means that the sound hits the loudspeaker, say 12 µs later - not something anyone can hear. If, however another signal to a loudspeaker nearby is **not** correspondingly delayed, then everyone can hear it. Thus we need to delay the other signal the same amount of time - therefore we need this delay to be an integer number of samples.

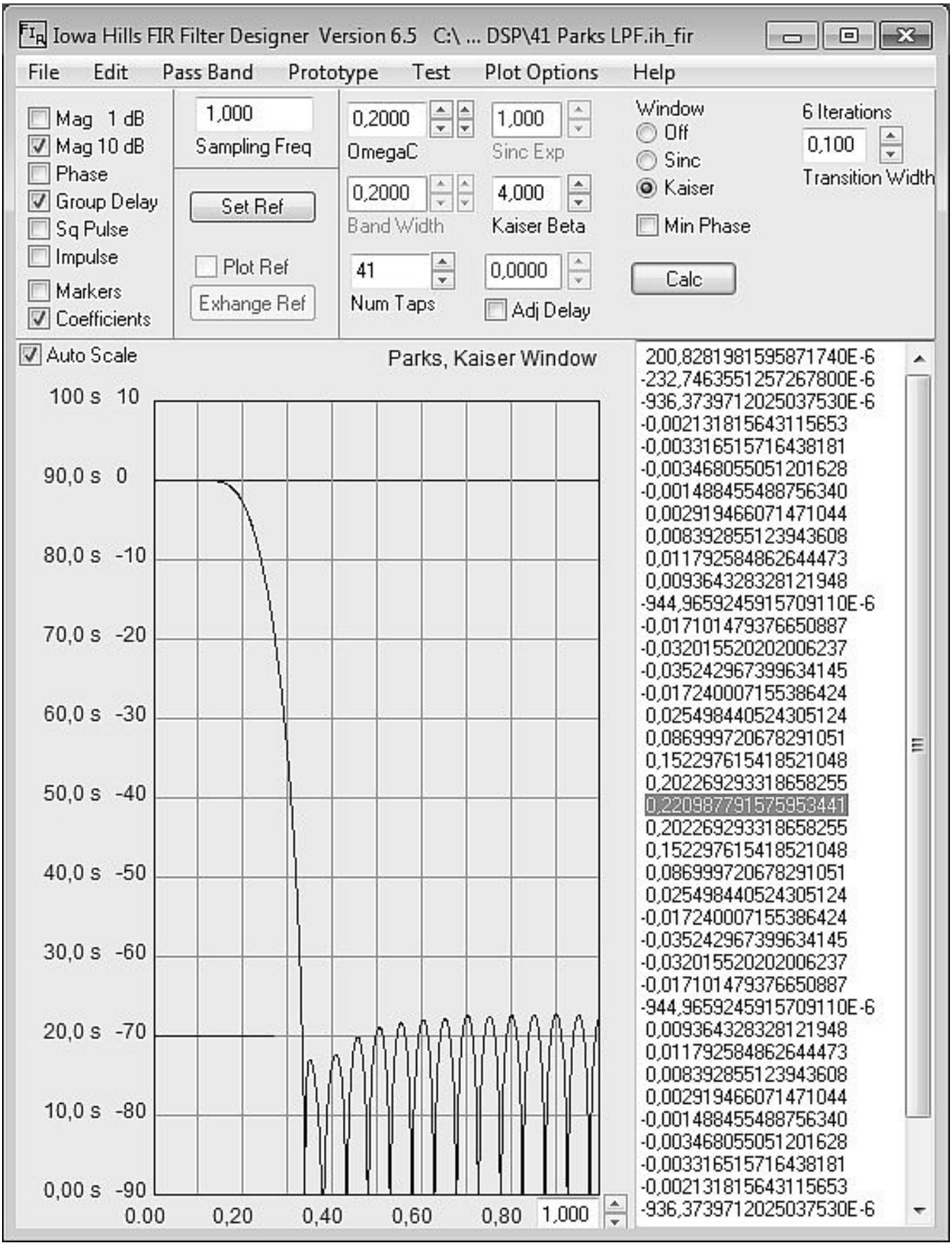



**Figure 6.12:** IowaHills FIR Designer

As stated earlier, in IIR filters there is a “feedback” from the output. A major

downside of this is that you need to calculate all your output samples, because they are needed when calculating the next sample. Skipping any calculations is not an option. Now why would anyone want to do that anyway? One of the most common processes in Signal Processing is “decimation”. Here’s an example: Let’s say we have a signal sampled at $f_s$=96 kHz. This is a common sample-rate in professional audio. For some reason we only care about the content below 7.6 kHz, and we have a lot of processing to do with this signal. It makes sense to go down to a fourth of the original sample rate - to 24 kHz. We cannot go down to 12 kHz, since this would violate the sample-theorem (7.6 kHz being greater than 12/2 kHz). We might consider something that is not the original sample frequency divided by $2^n$, but it typically is more complicated and invites other problems.

So basically we want to throw away 3 out of every 4 samples. This we cannot just do. If there is **any** content between the “future” $f_s/2$ (12 kHz), and the current $f_s/2$ (48 kHz), we need to remove this before we throw away samples. To do this we need to apply a lowpass filter. If we use an IIR filter that requires 10 Multiply-Accumulates per sample for two BiQuad-Sections at the high sample rate, we are effectively spending 4*10 = 40 MAC’s per resulting output sample when we have thrown away 3/4 of the samples. With an FIR-filter we only need to calculate the actual output samples that we need. That means that the FIR filter is not competing with 10 MAC’s per sample but with 40 MAC’s per sample. So if we can do the FIR in 39 MAC’s it would actually be faster. Add to this that the FIR-filter is truly “vectorized” - once you have set it up correctly many DSP’s are very fast. If the DSP has HW built-in BiQuads for IIR, this is also very fast, but if it only has the fast Multiply-Accumulate there is a lot of shuffling around in-between samples.

An ideal filter would typically be a square box in the frequency domain. If it’s a low-pass filter it will be centered around DC (remember the Frequency axis with a negative side). If its a high-pass filter it will be centered around $f_s/2$, and a band-pass filter is somewhere in between. So one way to implement an FIR filter would be to do an FFT (Fast Fourier Transform), transforming the signal to the frequency domain, multiply it sample-by-sample with the square, and then perform an iFFT (inverse Fast Fourier Transform) to get back into the time domain. This can actually be done on short time “snippets”, but it is not simple[2] on a data-stream. Instead we transfer the square from the frequency domain to time-domain. What was a multiplication in the frequency domain becomes a *convolution* in the time domain (we will get to this shortly).

In the introductory section on signal analysis we saw a square in the time-domain, that became a sin(x)/x function in the frequency domain. If we are taking a square from the frequency domain located around DC - in other words a Low-Pass-filter - to the time domain, it actually becomes a sin(x)/x. Look at figures 6.3 and 6.4, swap frequency- and time-domain, and you see the Low-Pass FIR-filter as a sin(x)/x in the time-domain. As we are working with discrete signals the sin(x)/x is sampled. In DSP we call each of these samples in the FIR filter a *tap*. Figure 6.12 shows the FIR-designer from IowaHills. This time there are no cascaded sections or Poles & Zeros - just a lot of coefficients. The Center coefficient is marked with a cursor. Note that the other coefficients are symmetric around this. This gives us the linear phase.

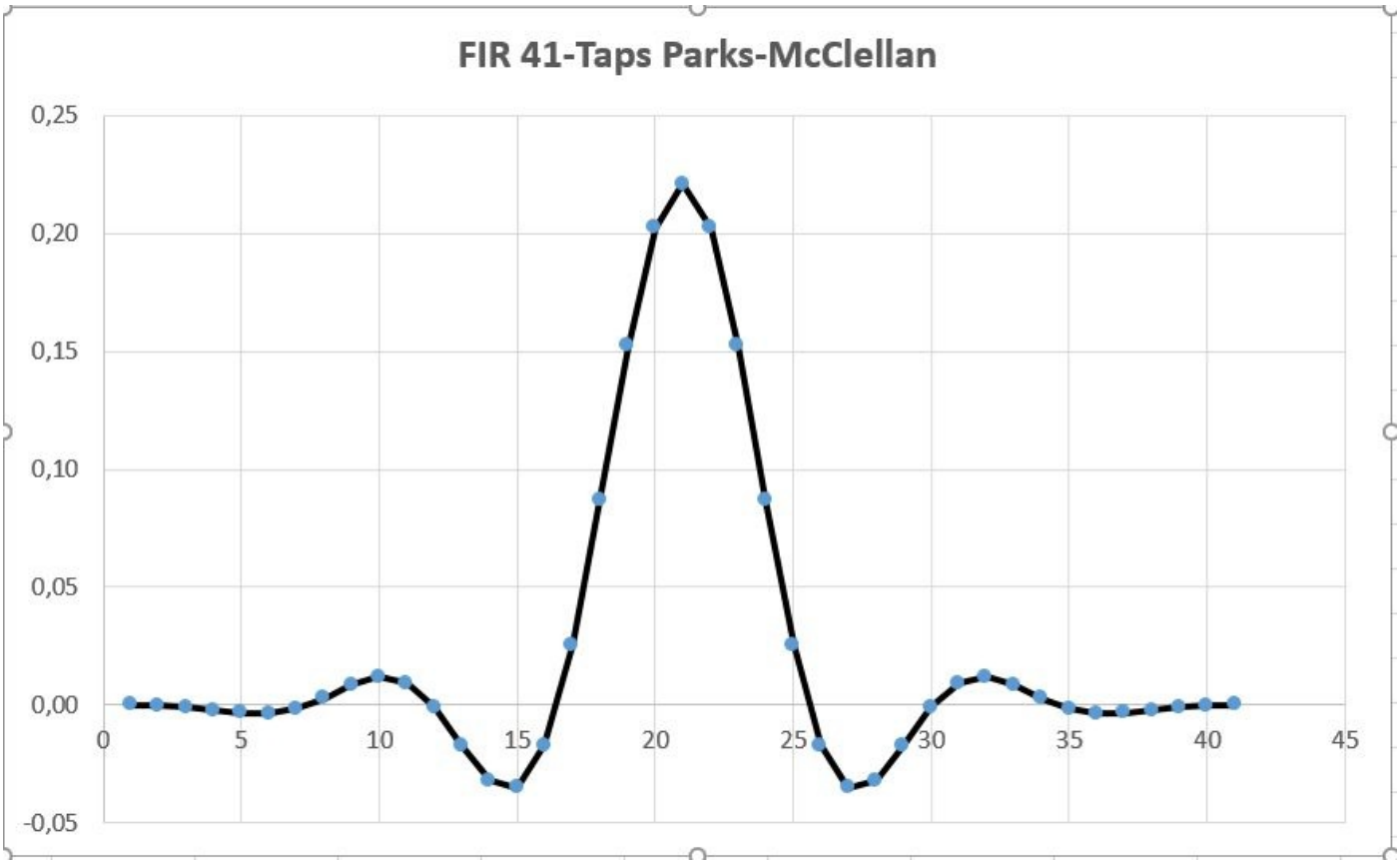



**Figure 6.13:** Coefficients from FIR design program

## 6.7 Implementing FIR

As previously stated, convolution is the process we apply to our data and the taps of the FIR filter. This is the process Figure 6.11 shows. We "park" the coefficients next to the samples, multiply each sample with a filter-coefficient, accumulate the products and then we have an output sample next to the center of the filter. There are two things that you may have been thinking:

- If we need to park the filter-taps with the center next to the current sample, we will not have samples and taps correctly lined up if there is an even number of taps. No, and that's why we should always generate and use FIR-filters with an odd number of taps.
- How can we park next to the "current" sample? It's fine that half of our filter-constants are pointing back to old samples, but what about the part that points ahead of the current sample - so to speak into the future? That is impossible. Yes, that's what the mathematicians call a non-causal filter. We need to work on a delayed signal so that we have all input-samples ready. This is another way to explain the delay. It is not difficult to realize that we need to delay the signal ((N-1)/2)*ΔT, where N is the (odd) number of taps in the filter.

Designing FIR-filters is not quite as easy as it may sound. Simply calculating a sin(x)/x in e.g. Excel, doesn't take you all the way. The ideal sin(x)/x is infinite, and if N is infinite we cannot delay the signal even N/2, and we don't have the computing power for N MAC's either. So we need to "chop off the skirts" of the filter somewhere - hence

the name “Finite Impulse Response”. To do this without disturbance requires some additional math. Normally a window-function is applied to the filter-function - calculated from an “equiripple” - or least-squares - approach named Parks-McClellan after its inventors. So, like with the IIR-filter the use of an existing filter or a filter-design program - e.g. like Figure 6.12 is recommended. Excel is however not completely useless. Figure 6.13 shows the filter-coefficients from the filter-design program in an Excel graph.

A simple way to calculate a high-pass filter is to first calculate a low-pass filter with the requested cut-off frequency and then invert the sign of every second tap (starting from the center, leaving this as is).

Figure 6.14 shows a buffer that implements a circular storage with the same number of samples as there are filter-coefficients. The coefficients are placed with h[0] next to the oldest sample, h[1] next to the one a little newer until the we have h[NoOfTaps-1] next to the newest. Just like the FIR-formula each coefficient is multiplied with the sample next to it and all the products are summed - again the Multiply-Accumulate.

When the next sample is inserted, we move the FIR filter-coefficients one sample so that its center is now parked alongside the next, newer, input-sample. The process is repeated indefinitely.

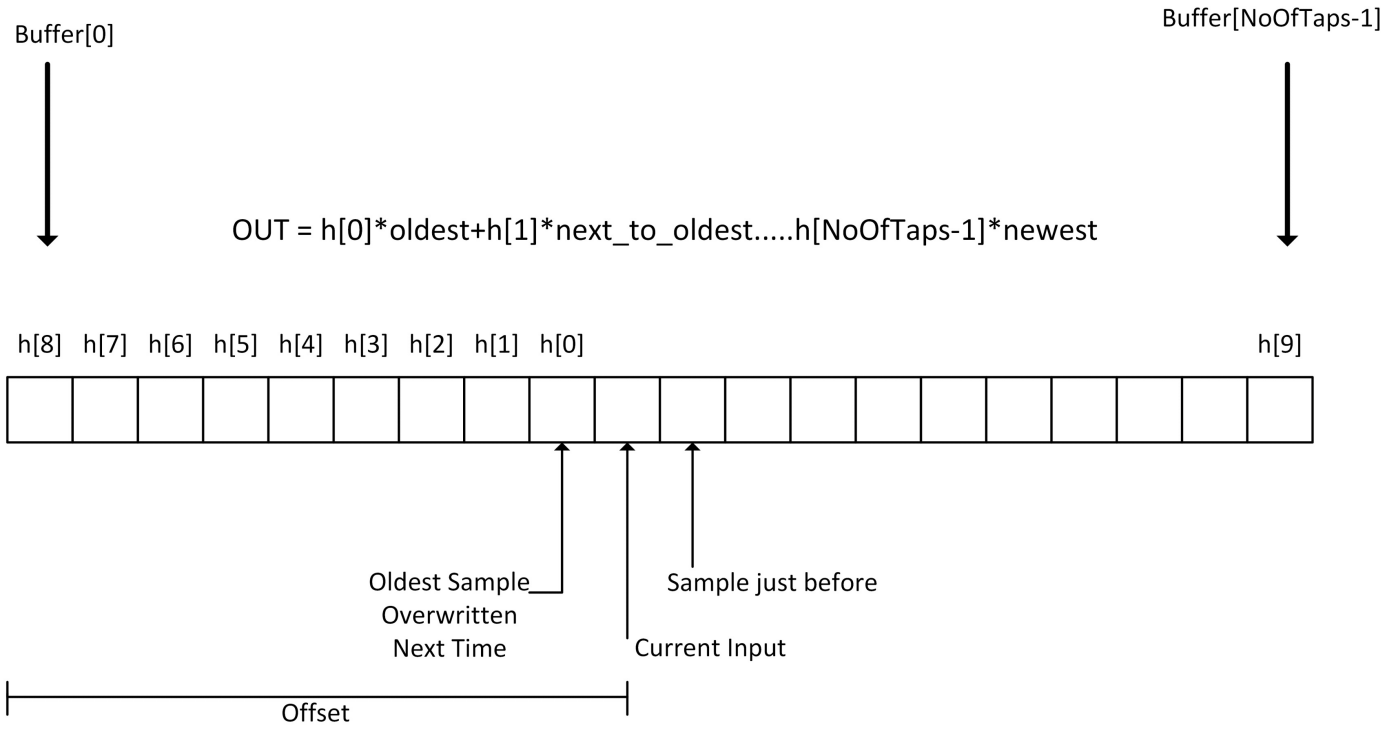



**Figure 6.14:** Buffer-usage for MIT’s implementation

Figure 6.14 is created to demonstrate the very efficient piece of C-code made at MIT that implements a FIR-filter (my comments inserted).

```
01 double Fir1::_Filter(double input)
02 {
03     double *coeff     = coefficients;      // Pointer to coefficients[0]
04     double *coeff_end = coefficients+taps; // Point to after array
05
06     double *buf_val   = buffer +offset;    // buffer is a class variable
07
08     *buf_val = input;                      // buf_val is 'current' pointer
09     double output = 0;                     // ready to accumulate
```

```
10
11    while(buf_val >= buffer)              // Until left edge
12       output += *buf_val– * *coeff++;    // MultAcc and move pointers
13
14    if (++offset >= taps)                 // At right edge with samples
15       offset = 0;
16
17    return output;
18 }
```

**Listing 6.2:** MIT FIR filter code

For the MIT-listing we have following free-usage license:

License: MIT License (http://www.opensource.org/licenses/mit-license.php). Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

## 6.8 Integers

FPU’s - Floating Point Unit’s - are gaining ground and they make coding a lot easier. With such a device you can write your embedded code in plain C using *float* and *double* as usually on bigger CPU’s. However, the smaller systems often do not have this and you are forced to work with integers, as a Floating Point Library with a software implementation is usually too slow for serious signal processing.

A “feinschmecker” may actually prefer the integer arithmetic. Say that you are working on a 32-bit CPU and your numbers are in 32-bit. With an IEEE-754 binary32 (previously known as “single precision”), aka “float” in C, you have a 24-bit mantissa (including the sign) and an 8-bit exponent. This gives you a much larger dynamic range than a 32-bit integer can, but the **precision** is lower - there are simply 8 bits more in the integer. So if you understand the theory, your data and your design really well, then you may be able to use the integer range well without getting overflows. This again may enable you to pick a smaller CPU and use less power (and/or money).

This, and the following sections, is all about working with integers. We will start with a “back to basics”. The positive number in any number-system can be written as a function of the “base” as:

$$Value = \sum_{n=0}^{p} a_n * base^n$$

Remember that any number raised to the power of 0 is 1. Here are some examples with 3 digits in various bases:

- The generic case: $a_2*base^2+a_1*base^1+a_0*base^0$
- Base 10, “256”: $2*10^2+5*10^1+6*10^0 = 2*100+5*10+6 = 256$
- Base 2, “101”: $1*2^2+0*2^1+1*2^0 = 2*2+0+1 = 5$
- Base 16, “a2c”: $10*16^2+2*16^1+12*16^0 = 10*256+2*16+12 = 2604$
- Base 8, “452”: $4*8^2+5*8^1+2*8^0 = 256+40+2 = 298$

The number-systems are known as “Decimal”, “Binary”, “Hexadecimal” [3] and “Octal”.

*Most C-programmers know that hexadecimal numbers are written with "0x" in front. The hexadecimal number above is thus written "0xa2c" or "0xA2C". However, considerably fewer realize that the number "0452" is in fact our octal number from the list above, and is thus 298 in the decimal system. I once helped a guy chase a bug down. He had written a lot of numbers in a table and some of them he had written a "0" in front to align the columns. They alone were read as octal. A nasty bug.*

In standard C we have a choice between "int", "unsigned int" and "signed int" as well as the "short" and the "long" (and newer "long long") variants. If you don't write "unsigned" you get the signed version by default. However, in the small 8-bit CPU's it is common to use "char" for small numbers. There is also "signed" and "unsigned" here for exactly this purpose. But for some reason there is no default for signed or unsigned.

Make it a rule to always use the "signed" version in DSP - e.g. by a *typedef*. If you just write "char" it's up to the compiler default whether you get the signed or unsigned version. The program may be working nicely, but after a compiler upgrade or change, or even only when the new guy is compiling at his PC, mysterious things happen, and it turns out that one variable was declared "char", and it went from being signed to unsigned by a changing default. See c99 for some great addenda for these cases. Also note that the C standard only guarantees the following:

```
01 sizeof(short) <= sizeof(int)    // This is guaranteed - but no more
02 sizeof(int)   <= sizeof(long)   // This is guaranteed - but no more
```

**Listing 6.3:** Standard C data-sizes

This means that if the normal *int* is 16-bit, there's probably a good chance that a *short int* is 8 bit, but the *long int* could easily be 16-bit also, if the CPU is a 16-bit one.

The way negative integers are represented, is in reality decided by the fact that we want to be able to use the same adding circuit as we use for unsigned integers. We want to be able to add -x to x and get 0. So if using hexadecimal numbers and a single byte, what do we add to 0x01 to get the result 0x00? The answer is **0xff**. So for a 16-bit word we have some values:

| Hexadecimal | Decimal |
|---|---|
| 0x7fff | 32767 |
| 0x4000 | 16384 |
| 0x0001 | 1 |
| 0x0000 | 0 |
| 0xffff | -1 |
| 0xc000 | -16384 |
| 0x8000 | -32768 |

## 6.9 Fixed Point Arithmetic

It is custom to work with a *virtual binary point* when you are doing integer arithmetic. This is typically placed just right to the sign-bit. An example with a positive 8-bit number, with the virtual point written:

$0.1011010B = 1*2^{-1}+0*2^{-2}+1*2^{-3}+ 1*2^{-4}+0*2^{-5}+1*2^{-6}+0*2^{-7}$

Hexadecimally this is written without the point - as 0x5a. The easy way to get from the hexadecimal value is to convert to decimal and then divide by $2^{N-1}$ where N is the number of bits. In our case we have:

0x5a = 5*16+10 = 90. *and* $90/2^7$=0.703125

If you have a decimal fraction between 0 and 1, you go the other way by multiplying with $2^{N-1}$. This number is rounded[4] to a full integer and then converted to hexadecimal (or binary if you prefer). Traditionally there has been a number of different ways to have a sign bit as MSB - Most Significant Bit.

- MSB as sign. The rest as magnitude.
- Negative Numbers as “One’s Complement” of the absolute value - which is simply inverting all bits.
- Negative Numbers as “Two’s Complement” of the absolute value.

In modern hardware and CPU’s there is no doubt: “signed” means two’s complement - **except** for one important place: The checksum on IP- and TCP-headers, where one’s complement is used. But this chapter is on DSP, so we can safely assume two’s complement. Let’s examine the facts:

- It is named “Two’s Complement” because the bit pattern for x >= 0 is x as defined above, while the bit pattern for x <0 is 2-|x|.
- You can get from any number x to -x with the following operation:
  *Invert all bits, then add 1.* Take e.g. the positive number 6 in a 4-bit system. This is 0110 binary. First Invert: 1001. Then add 1: 1010. Test by adding 6: 1010 + 0110 = 0. There are other clever algorithms, but it’s better to know one by heart, than two halfheartedly.
- With two’s complement you can add signed numbers in the same way that you can add unsigned numbers. If, however, after an addition, or subtraction, the *carry* is different from the sign-bit (MSB), we have an *overflow*. If there is an *overflow flag* in the CPU, then this is set. It is typically known as the “V” flag, while the carry is the “C” flag and the “Z” flag means that the last instruction (that affects the flags) ended with a zero result. Finally, the “N” or “S” flag is the sign - following the MSB. An overflow basically means that two positive numbers added together gave a negative result, or that a negative number added to another negative number gave a positive result etc. In such cases there is a *wrap* . The overflow flag can be checked in assembler but not in C, however in some systems you can set this to generate an exception (aka trap). In many DSP’s you can choose *saturation* mode , which means that the DSP chooses the max positive or negative number instead of “wrapping”. This is not as good as it sounds, and you should be able to disable this function because:
- If you are working with two’s complement integer arithmetic, you may have designed e.g. a filter in a way that there is no overflow in the final sum of many products, but there may temporarily be along the way. Due to the way two’s complement wraps you are saved - **unless** saturation mode was on. This is one of the reasons integer DSP calculations are interesting; getting the maximum out of the dynamic range is an

art form. But time-consuming.

The Two's Complement are designed so that adding numbers works fine even though the CPU "thinks" it's working on integers. Modern CPU's also do subtraction, which also works on Two's Complement numbers. Multiplication is a little more complicated.

## 6.10 Q-Notation and Multiplication

If we multiply two "normal" signed 16-bit integers we end up with a 32-bit integer. There is twice the amount of bits. There used to be 1 sign bit and 15 integer bits in each, so in the result there are 2 sign bits and 30 integer bits. This is not a problem, the CPU will automatically copy the bit next to the MSB, into the MSB. This is called *Sign Extension*. It doesn't change the "weight" of the bits behind it.

Not so with fractional numbers like Two's Complement. If we end up with two sign bits, our result is precisely half of what it should be. And it becomes more complicated; the virtual point does not have to be right after the sign - you may want to set it so that you can have e.g. filter-coefficients larger than 1. In order to manage this, the "Q" notation was introduced. A 16-bit word using normal Two's complement is noted Q15 - or Q0.15 - specifying that the Quotient part is taking up 15 bits. The same 16-bit word might also be used for Q13 aka Q2.13, meaning that there are 13-bits in the fraction and 2 bits in the integer part. We will stick to "classic" Two's Complement and call it Q7 in bytes, Q15 in 16-bit words, and Q31 in 32-bit words.

As we started out with Q15 we have Q30 after a multiplication. In other words, the "stupid" CPU has given us two sign bits. We need to shift one bit left and then keep the upper 16 bits as the result. A real DSP typically has the now often mentioned Multiply-Accumulate. If it's a 16-point DSP, this register will be 32-bit. As we have seen, filters are based on the sum of many products, so instead of rounding each multiplication to 16-bits, the DSP accumulates the numbers in the 32-bit accumulator, saving us the many round-off errors. Finally, when the result is read as a 16-bit number, the fixed-point DSP will typically silently shift the result one bit left for you - assuming that you are doing Q15 arithmetic. There are variants where the so-called "barrel shifter" can be programmed to do a specific number of shifts - sometimes there is just the selection of e.g. 0,1 or 4. If you are using a standard - non-DSP - CPU with Q15 or Q31 arithmetic, it is up to you to remember the last left shift.

### 6.10.1 Division

Floating Point units also supports Division. But it will typically take longer than multiplication. This is also the case for Integer Division. It is true that division by $2^n$ can be performed by right-shifting n-times with sign-extension, but this makes the code harder to read. It may be necessary in the famous tight inner-loop - but check the compiler-listing for a division with such a number first. Most compilers know this trick. Similarly, a floating point division with a constant might be faster by multiplying with the reciprocal. If there is something to gain here, and the listing shows that the compiler actually doesn't know the trick, it's a good idea not to calculate the reciprocal by hand and mysteriously

multiply with it in the code. If you want to divide by pi, you can write: “y=x*(1.0/pi)” The compiler will divide the constants for you at compile-time.

### 6.10.2 BCD

IBM invented BCD - Binary Coded Decimal. The reason was that the Fixed Point arithmetic may be great for Signal Analysis, but IBM was in it for the money - so to speak. If you are working with Dollars and Cents (or the equivalent in another currency) you really don’t need to represent a number such as 1/2+1/4+1/8+1/16… but you would terribly much like to express e.g. 15.30. So IBM sacrificed some bits; instead of letting each 4-bit nibble count from 0 to 15, they let it count from 0 to 9 - and the next step was an *auxiliary carry* - yet a CPU-flag - which incremented the next nibble, or was used by the programmer for this purpose. Even the famous Digital VAX actually had support for BCD. This is however history now, and you should not ever expect to do signal analysis in BCD.

## 6.11 Further Reading

- Oppenheim & Schafer: Discrete-Time Signal Processing
  This is the author’s own replacement for: “Digital Signal Processing” which is a fantastic classic.
- Rabiner & Gold: Theory and Application of Digital Signal Processing
  Another classic.
- www.iowahills.com Filter Design
  The filter design programs from this site has been used in this book with permission from the author. If you want to dig deeper into implementation of IIR and FIR filters this is a good place to start.
- musicdsp.org
  A good source on filters and other audio-related DSP. The biquad-filter by Tom St. Denis is used in this chapter is from this site.
- web.mit.edu/2.14/www/Handouts/PoleZero.pdf
  A very good description of poles and zeros and their relation to the Bode-Plot.
- Erik Hüche: Digital Signal Behandling (Danish)
  A down-to-earth textbook. The chapters on hardware are outdated, but the theory is well explained.

# Part III
# Best Practices

# Chapter 7
# Software Architecture

## 7.1 Design for Performance

*Working with software, PC as well as embedded, for more than 30 years, has led me to realize that a system designed for performance is often also a simple system, easy to explain, easy to understand, and robust. On the other hand, code that is performance-optimal is mostly difficult to create, even more difficult to read a year later, and often prone to bugs. Add to this that the performance gain in a good design, over a bad design, may be a factor of 10, whereas performance improvements in code optimization is measured in percentages and 25 percent is a "boost".*

There are typically very few "inner-loops" such as the ones in digital filters discussed in Chapter 6. This all leads to the following mantra:

*Design for Performance - Code for Maintainability*

An example: In the last many years, Object-Oriented Programming (OOP) has prevailed in PC-programming. It has been a fantastic tool for especially GUI-programming. The idea that you take the point-of-view, of one of many, more or less identical, shapes on the screen is great. If you deal with the intricacies of "Garbage Collection" and hidden instantiations of heap-based objects, you can also write embedded code in OOP. But there is one part of the OOP that doesn't go well with a distributed system such as IOT.

It is very typical, in OOP, that first you instantiate an object, then you set a lot of "properties". The properties are "private attributes" with "access-methods". This ensures encapsulation. Consider what happens if this concept is copied to an IOT system design. Imagine that we have 1000 IOT-devices, each with 100 objects, each with 5 properties that we want to setup from our PC. This means 1000*100*5 = 0.5 million roundtrips. If you do a "tracert" (more on this in Chapter 4) you may see that a typical time for a packet to travel to e.g. ieee.org is 100 ms. Assuming that it takes the answer the same time to get back, and that the tiny IOT device needs 50 ms to handle the job of receiving, handling, and answering the change, then we look into:

0.5 million*250ms = 125,000 seconds = 35 hours.

Surely you can do it all in parallel, taking this time downto 2 minutes, but handling 1000 sockets in parallel is far from trivial; the error-handling if e.g. a connection goes bad along the way is a nightmare. Even with a socket per device, we may need to handle the cases where we get halfway through the 100 objects, and then something happens. This leaves the given device in an unknown state - and then what? The alternative is to use a concept that saves on the round-trips and allows you to send all fully-equipped "objects" in one action. You might say "transaction", as you can design this so that a setup is either good or bad. This might take a little longer execution time in the actual IOT device, but still most is actually handling the in- and out-going messages, so let's raise this time from 50 to 100 ms - causing the round-trip-time to grow to 300 ms. That would take:

1000*300 ms = 300 s = 5 Minutes

If you insist on the 1000 parallel sockets; it would be done in 300 ms.
So the result of working with the bigger chunks of setup data is a lot less traffic, simpler error-handling and at least you have the choice of doing the 1000 sockets in parallel.

So how do you send a setup in one swoop? One idea is to have all setup-data centralized in the embedded device in an SQLite database. The setup simply sends a new version of this, the device stores it and basically does a reboot. If a reboot takes 1 minute, the last device will be ready 6 minutes after you started - not so bad. Even the parallel scenario becomes doable due to the transactional nature of the operation. SQLite has the advantage that it is file-based, and the file-format is the same on all platforms. This allows you to do a lot of tests in the protected PC-environment.

Now, someone in the team might start to realize that the users really don't update anything that often, and that they might use this approach for **every** update - not just the really big ones. This really depends on the application. If it is possible, it will guarantee that all devices are in the same, known configuration, that has been auto-tested in the test-lab for days[1] . It will mean that there are no other setup commands to implement than the upload of a new file. And the only state-change the device has to handle is to boot - something that was necessary anyway. So by looking at system-performance - not just the embedded performance - we end up with a much simpler and more robust system. Unless we have a one-man project, this is something that would never happen during the code-phase. Instead each developer would write more and more hard-to-read code in order to shave down the execution time. Good design decisions typically do not happen with a single guy at a PC. They happen in a cross-functional team around a whiteboard or beer.

Another possibility for larger chunks of setup-commands is to use REST with "sub-trees" in json or XML - this is discussed in Chapter 4.

## 7.2 The fear of the white paper

It can be terribly stressing to start from scratch. Most developers in the industry start in a project with a lot of "legacy" code that they extend and bug-fix. At the same time they learn from their peers, and along the way they become experts in the given architecture, hardware, language and the patterns used. But then - once-in-a-while - someone decides that the next product is made from scratch. Everyone is ecstatic, "finally we get to do it right!" On the inside a good part of the developers may be less ecstatic - what IS the right thing? The white paper can actually be very scary.

There are a few approaches. One way is to build on a framework from someone else. This makes really good sense, if you can find something that fits. Let's take some examples from the PC-World. When Microsoft moved from the MFC/C++ to .net/C# they moved to a better language and a better library - but not really a platform. There was no structured replacement for MFC, which provided the Document/View-structure, Serializing of files, Observer-Consumer pattern and a lot more. At the time a number of vendors of hardware - CPU's, FPGA's etc. typically programmed their own IDE - Integrated Development Environment for their customers to use in their development. Most of these used MFC/C++. The rest used Delphi, which sort of lost the battle. In either

case, the hardware-vendors were looking for a new application framework. The choice in this community has almost uniformly been Eclipse. Eclipse is an IDE-platform originally made for Java, but now has been extended for a huge number of tools. Eclipse is used for C/C++, but also for FPGA designs, DSP-development and a lot more. In this case the answer was another vendor, sometimes another language (java) - but for these hardware-vendors already half of their solution. Other vendors have found other similar narrow platforms, and the rest have found their way in the new world of C# and .net with less formalized platforms that was more like guidelines.

But what if there is no Eclipse for you? How do you actually start on the white paper? Like you eat an elephant - one byte at a time. It is often stated that the agile approach works best when there already is a foundation - or platform. That is probably right, but it is also possible to start from scratch in an agile way. Instead of thinking up the whole platform before coding - just start. However, the team and the management must be prepared to do some rewrites. In reality, you seldom start entirely from scratch. Wasn't there a requirement to be able to read files from the old product - or from the competitor - or some industrial standard? Shouldn't the new system interface to the old database, cloud or whatever? Why not start here then? Use this to learn the new language/compiler/IDE/library. And as demonstrated in Section 7.4 - files are a great place to meet. Starting without a complete plan is daring. But making a full plan of how you want to make a new product on top of new programming tools and idealistic guidelines is even more dangerous.

It can be very satisfying as a programmer to "see the light" and learn new technology. But once you have tried standing in front of a customer who cannot get his basic needs fulfilled, and you realize that it probably won't help to tell him that "it's programmed using XXX!" - then you realize that meeting the customer's requirements is at least as fun as trying out the latest programming fad. And if you focus on that first, there is no reason why you cannot try out some new technologies as well - but probably not ALL of them. In other words; don't start making all the layers in your new application in their full width from day one. Implement some "verticals" - some end-user features. When you have implemented a few of these, your team will start discussing how they really should isolate functionality "x", and put all "y"-related functions in a separate layer. And this is where that manager should fight for his team and the right to rewrite the code a couple of times. The argument could be that you saved all the first months of designs and discussions. This is the essence of the agile approach. However - all this has been good Latin in the embedded world for many years. Embedded programmers are used to a "bottom-up" approach. Before they can make any application-like coding, they need to be able to do all the basics; get a prompt on an RS-232 line, create a file-system, send out a DHCP message etc. This is probably one of the reasons why the agile approach has been received kind of "luke-warm" in many embedded departments. To the embedded developers it feels like old wine on new bottles - and in many ways it is. Still embedded developers can also learn from the formalized SCRUM process and its competitors.

*Some time ago I participated in a session where a number of project managers presented how they were doing SCRUM. One of the managers was managing an FPGA-team. It turned out that in many ways they were actually still following the waterfall-process. However, now they had stand-up meetings discussing their problems, they noted*

*and prioritized problems and they worked more as a team. To me this was not really agile, but it is a lot better than having four guys working in each their own world.*

## 7.3 Layers

There is a number of more or less interesting design patterns. You shall only be bothered with **one** architectural pattern here: **Layering**. The importance of this pattern cannot be overestimated. Basically it says that the upper layer knows about the lower layer - but the lower layer knows nothing about the upper layers. A well-known example of layering is the internet protocol stack as can be seen in 7.1.

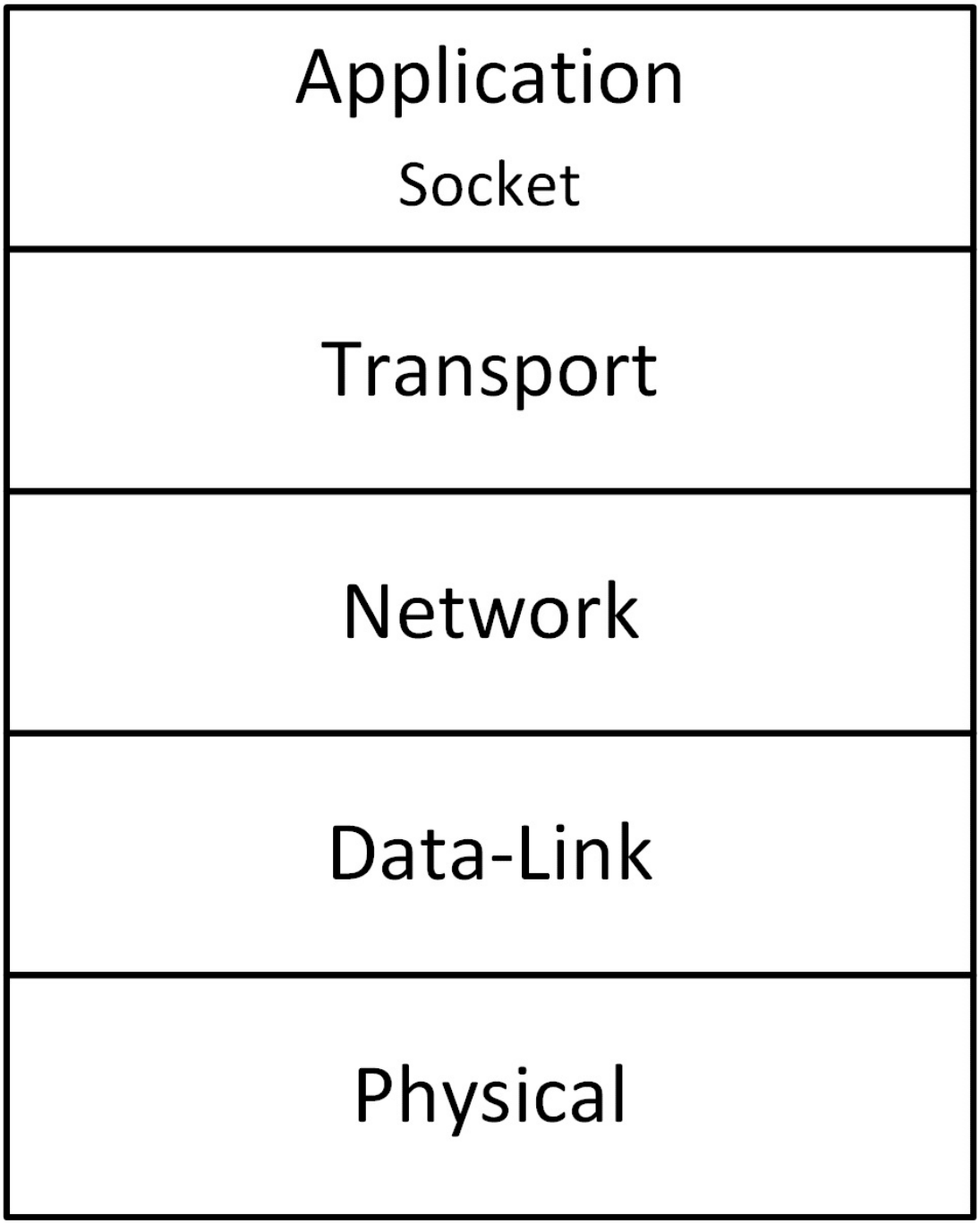



**Figure 7.1:** Layering

Here the Application Layer knows about the Transport Layer, but the Transport Layer

does not know which application is running, why, or how it works. It just accepts orders. Similarly the Network Layer knows about the Data-Link Layer and not the other way around. This means that the lower layers may be interchanged with USB or something else. Thus we have an incredible powerful modularity.

With the term “True Layering” we tighten the concept. This means that an upper layer only knows about the one directly below it - not something further down. The internet protocol stack is not completely truly layered; the socket interface simply knows too much about what goes on in the Network Layer - two stories down. On top of this comes that the TCP checksum calculation includes specific fields from the IP-header. This all means that once you have decided to use TCP or UDP you also get IP - so on this level we do not have full modularity. This also explains why many choose to refer to the Internet Protocol Stack as the TCP/IP stack.

## 7.4 Not just API’s - more files

In the embedded world as well as in the PC-world you are often required to create an API, that the next developer can use. And many times you need to do this - but not always. One of the problems with an API is that after the test, there are no traces of what has happened. Anything that is logged requires extra work by the test-programmer. You can learn a lot from Linux (Unix). Every time you make something new on Windows you have to dream up a new API. Not so on Linux. Here a huge part of output is written to files - and a huge part of input is read from files. Even things that are not really files can be read as files (/proc that show processes is a good example). In Unix, Linux, and actually also in DOS and Windows, a process can send data to *stdout* and read data from *stdin*. This is shown in the first two lines in Listing 7.1. It is however also possible to “pipe” data directly from the output of one process to the input of another - as the 3’rd line shows. In Unix and Linux the processes might even run on different computers.

```
01 progA > datafile
02 progB < datafile
03 progA | progB
```

**Listing 7.1:** stdin, stdout & pipes

This is not just a nice way of communicating - it also allows for some really great test-scenarios:

- Say that you have once and for all verified that the data out of *progA* into *datafile* is correct. You can now keep a copy of the datafile, and start making performance improvements on *progA* and run it as shown in line 1 of Listing 7.1. In between the updates of *progA*, you can make a simple “diff” (or use BeyondCompare or similar) between the old and the new data-file. Any changes will mean that you have made an error. This is a good example of **regression-testing**.
- You can test *progB* without even running progA. You can simply perform the second line of the listing. This allows for a much simpler test-setup - especially if running *progA* requires a lot of other equipment, or if *progA* is depending on more or less random “real-life” data - producing slightly varying output. The ability to test *progB* with exactly the same input is important.
- If the processes are running on different computers, and data is piped between them

as in the 3’rd line of the listing, you can use Wireshark to ensure that the data are indeed corresponding to the datafile.

This is really practical in tool-chains running on real PC’s, but it is actually also useful in the larger embedded systems.

## 7.5 Object Model (Containment Hierarchy)

In the days when C++ really peaked in PC-programming, many of us spent a lot of time on inheritance-hierarchies. Considering that this is implementation, and that implementation should be hidden, we spent way too much energy on this - and too little energy on “Containment Hierarchies” or “Aggregation”. This is e.g. “A car has two axles, each with two wheels, each consisting of a road-wheel and a tire. Inside the car there is …”. This is an object-model. Interestingly, the code does not even have to be object oriented. A good system has an object model that all the programmers understand. Very often this model is the same as the one the “world” sees via the network-interface, but it doesn’t have to be like this. In that case the outward model is known as a “façade”.

### 7.5.1 ITU SDH

A good example of an object model is found in the telecommunications world. SDH - Synchronous Data Hierarchy - describes & implements a way to integrate many data-streams with different data-rates into a high-speed stream. The great advantage that SDH has over the previous standard - PDH - is the ability to “pick out” or “inject” any data-rate from the “container” without the need to breakdown the entire container into its components. The device that does this is called a “Multiplexer”, and is distantly related to the multiplexers most hardware designers use in their designs, but is of course a lot more complex. The hierarchy is as shown in Figure 7.2.

4*VC11 (each 1.5Mbps) OR 3*VC12 (each 2 Mbps) equals one VC2. 7*VC2’s equals one VC3, and 3*VC3 equals one VC4. The names make more sense when you realize that VC11 and VC12 ought to be called VC1-1 and VC1-2. A VC4 is basically an STM-1 which is 155 Mbps. In other words 63*2Mbps becomes 155 Mbps.

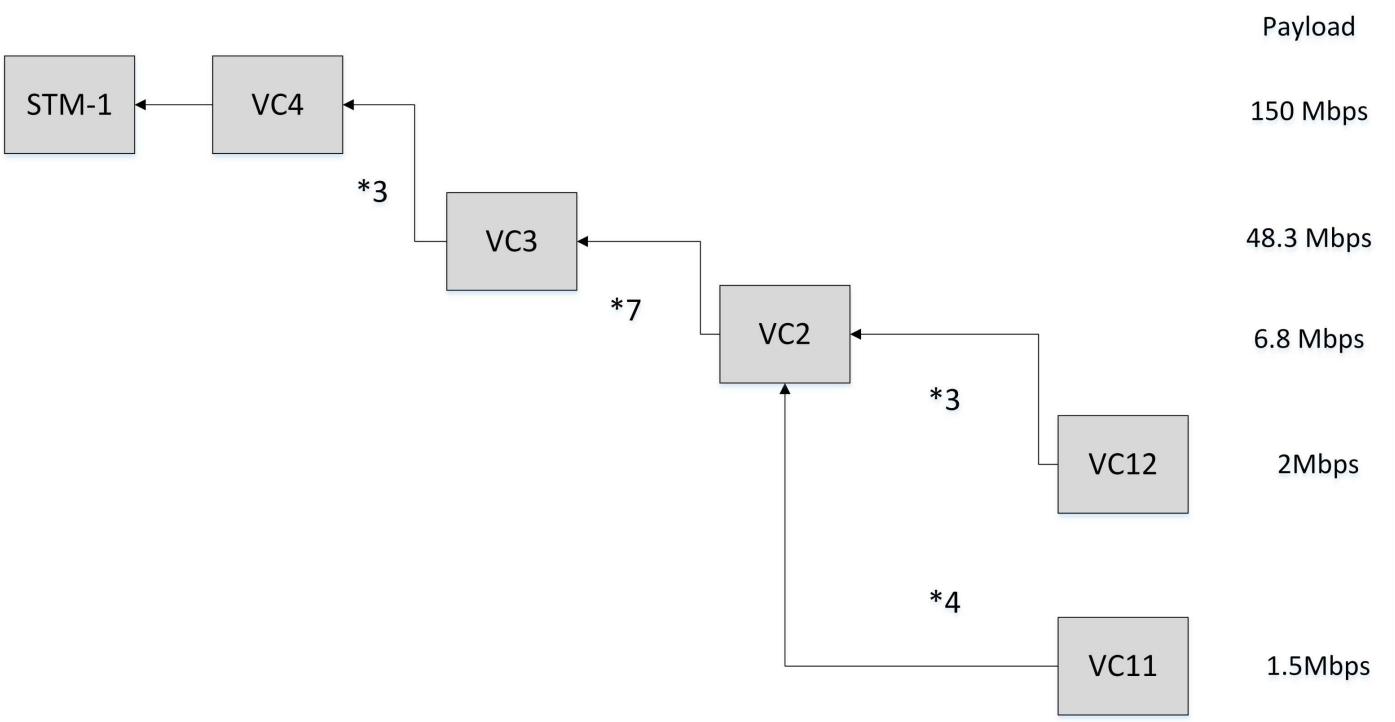



**Figure 7.2:** Simplified Virtual Container Structure

Managing all these data-streams, including a bunch of overhead-bits, statistics and alarms, requires a detailed understanding as well as a good overview. This is why SDH has an Object Model for its management interface.

*When I worked with this we could design our basic building blocks directly from the G.774 ITU Standard.*[2] *These were so well-described in the standard that a telecom operator could buy so-called "Network-Elements" from different vendors, and they could still manage them centrally with a single piece of software. A little "space" was left for the vendors in special functions.*

Figure 7.3 shows the Object Model for a Multiplexer. The gray boxes are the main Objects - typically realized as separate hardware elements, but not necessarily. Independent of the hardware realization, the embedded software appears to the management system as shown. The figure also shows a good application of the "Layers" pattern. A "Regenerator" is designed so that it can be used as a building block in a Multiplexer.

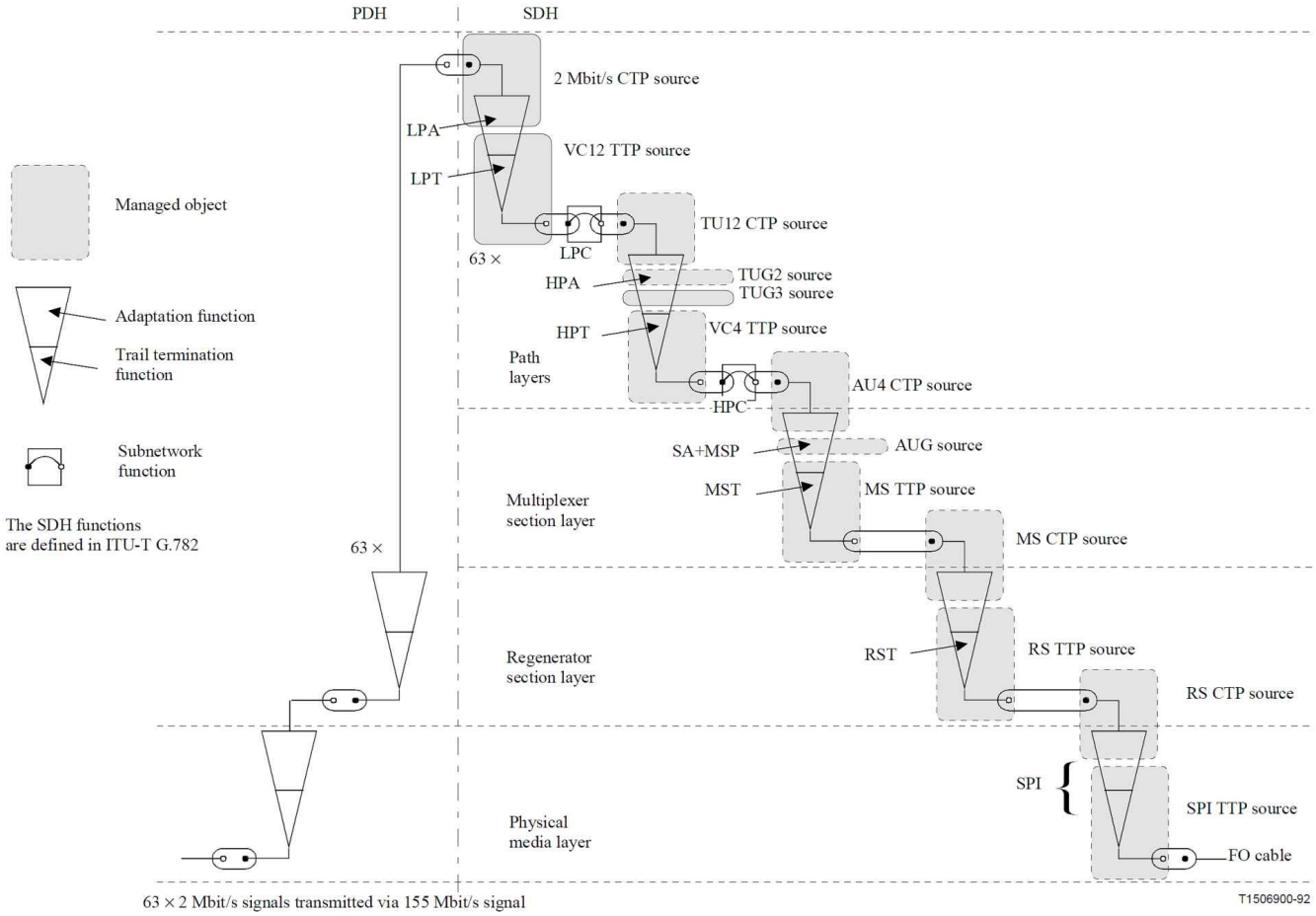



**Figure 7.3:** SDH Management Object Model. Courtesy of ITU

### 7.5.2 CANOpen

It is well known that modern cars have done away with a lot of point-to-point wiring, and replaced it with the simple, but clever, CAN-bus. It is however not so recognized that CAN only describes the physical layer and the datalink-layer - the rest is up to each manufacturer. Often even two different car models from the same manufacturer has different messages in the CAN-buses. CAN has an 11-bit CAN-ID - but the mapping of the possible values to measurement-types are not standardized in the automotive world (except a few in the OBD-II interface). When industrial manufacturers discussed to use CAN at the factory floor this was not an acceptable solution. They formed the group CiA - CAN in Automation - and defined CANOpen with the first standard DS301. This standard defines the use of the 11-bit CAN-ID, as well as message formats and a so-called “Object-Dictionary”. Figure 7.4 shows how the 11-bit CAN-ID is turned into a 4-bit Function ID - allowing for 16 functions, and a 7-bit Node-Id - allowing for 127 nodes in the network. Later standards define specific “profiles” for various usage. The more specific a profile is, the less there is to setup for the system integrator, naturally at the cost of flexibility. In CANOpen the Object Model lies in the Object Dictionary. This mimics a simple table where each entry, known by a 16-bit hexadecimal number, holds a specific function. It is however not that simple as each index may hold several sub-indices. Some indices are mandatory and completely specified, while others are vendor-specific. Each index starts with a number that describes the number of sub-indices. This provides the necessary

flexibility, but is still rigid enough to assure that the software from one vendor can handle the hardware from another.

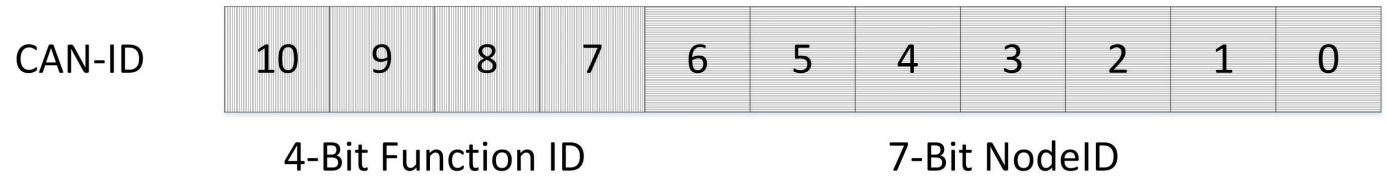



**Figure 7.4:** CANOpen usage of the CAN-ID

Figure 7.5 shows one vendors tool for managing the so-called Object-Dictionary. The GUI is crude, but it demonstrates the Object-Model very precisely. The 16-function-types are grouped int main “Message Types” shown in the figure and described in the following table:

| MsgType | Usage |
|---|---|
| NMT | Network Management. Heartbeat etc. |
| SYNC | Synchronization setup |
| EMCY | Emergency Messages |
| SDO | Service Data Objects. General Setup of e.g. trigger level for A/D-Converter etc. |
| RxPDO | Receive Process Data Objects. For a given CAN-ID sent to this node; which output to affect - and how? |
| TxPDO | Transmit Process Data Objects. What and when do we send something for a change on which input? |

The SDO-messages are what is used by the tool to setup the system. Re SYNC, Emergency and NMT you can often live with the defaults. The real workhorse is the PDO’s. These are sent during runtime when dictated by a timer, or when a data-value changes more than a specified value. By writing the right numbers in the Object-Dictionary you effectively wire inputs from one device to outputs of another. Using this concept, a CANOpen-system may be completely master-less, which was not the original default. Without a master a system is cheaper, more compact and has one less single point of failure that can bring down the system.

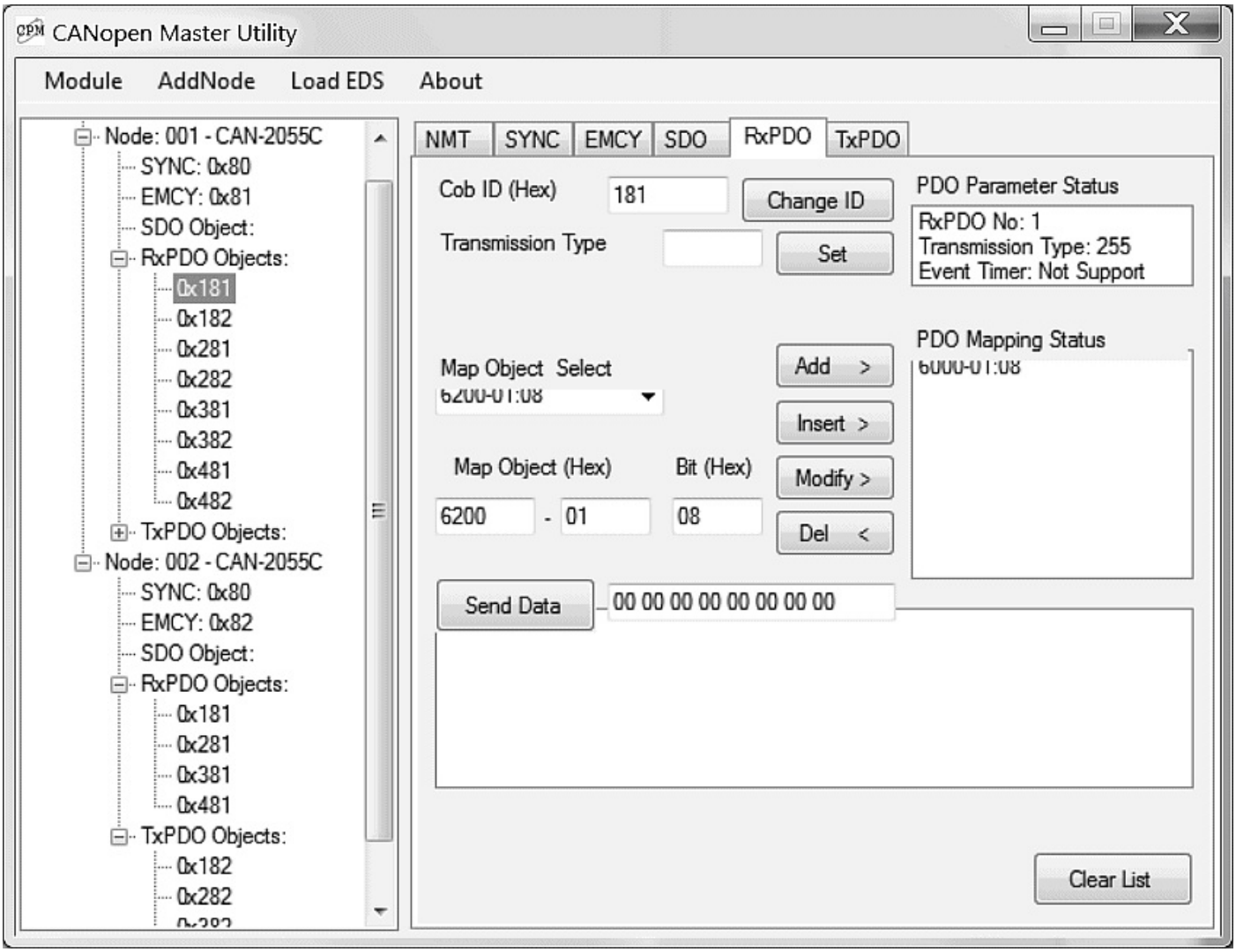



**Figure 7.5:** ICPDAS User interface for CANOpen

## 7.6 Message Passing

In the chapter on Operating Systems and especially Section 2.8 we looked at the various mechanisms that exists to help tasks avoid stepping on each other's toes. Multi-tasking can be problematic to debug on a single CPU and it only gets worse as we see more CPU's with multiple cores. In these we truly experience parallel processing. There is no way around the use of these complex constructs when it comes to writing code for the OS-core or drivers, but it is definitely worth **not** having to do this in normal user-space code.

If you use shared memory you need to use critical sections or mutexes to assure that you do **not** compromise data. If you forget one of these or make it too short you will compromise data - if you make it too long you will serialize execution and not get any value out of the multiple cores, and if you take the locks in different orders in the various threads, sooner or later you will experience a deadlock. It may be interesting, but it is absolutely not efficient.

For this reason *Message Passing* has been a popular solution for many years. When message passing is used there is typically a copy of data in the message sent from one task to another, but modern CPU's are very good at copying, so unless we are talking about

obscene amounts of data, this is a better deal than the error-prone technologies discussed earlier. Once each task is working on its own data there are no problems with data-sharing.

As stated before this also gives us a lot better debug possibilities. The fact that one or more tasks may be executed on a PC, while others are run in the real target should be enough for anyone to see the light.

To demonstrate that multi-threading problems are not just an embedded “feature” Figure 7.6 shows a so-called “race” in a C# program caught red-handed. In line 121 there is a check for “m_currentListenerSocket” member “Client” being null. We safely passed that on the way to the current line, 124, however as seen in the inspection window below, this is now null. It has been cleared by another thread. This was by design - it just happened at a bad time. This is in a PC-program using standard GUI and worker-threads. If instead the other thread had sent a message to our thread, that we should terminate this client, we would have read that information at a more proper time and removed the structure here where it is used (actually the code shows that we would have closed it anyway).

An OS like Eneas OSE is built around asynchronous message passing that works seamlessly between CPU’s in a kind of local network, and many small kernels support message passing within the same CPU. Today there are also a number of platform-independent concepts such as ZeroMQ. This is based on the well-known socket paradigm and is definitely worth looking into. At the moment it uses the LGPL - the Lesser GNU Public License - which means that you can link to the library from proprietary code.

---

```
107             }
108
109             // Apparently the loop has  exitted, so close the UDP listener
110             if (m_udpListener != null)
111                 m_udpListener.Close();
112         }
113
114
115         private UdpClient m_currentListenerSocket;
116         private void CloseCurrentListenerSocket()
117         {
118             if (this.m_currentListenerSocket == null)
119                 return;
120
121             if (this.m_currentListenerSocket.Client == null)
122                 return;
123
124             if (this.m_currentListenerSocket.Client.Connected)
125             {
126                 if (this.m_currentListenerSocket != null)
127                     this.m_currentListenerSocket.Close();
128
129                 this.m_currentListenerSocket = null;
130
131                 return;
132             }
133         }
134
135         // Default - handle socket exceptions
136         private byte[] ReceiveData(UdpClient udpClient, int timeOut)
137         {
```

100 %

Watch 1

| Name | Value |
|---|---|
| m.frametype | The name 'm' does not exist in the curr |
| m.framevariant | The name 'm' does not exist in the curr |
| m.ipaddr | The name 'm' does not exist in the curr |
| framevariant | The name 'framevariant' does not exist |
| bytes | The name 'bytes' does not exist in the c |
| 0x00 | 0 |
| result | The name 'result' does not exist in the c |
| m_idaArpDeviceCollection | The name 'm_idaArpDeviceCollection' |
| m_currentListenerSocket | {System.Net.Sockets.UdpClient} |
| Available | 'm_currentListenerSocket.Available' thr |
| Client | null |
| DontFragment | 'm_currentListenerSocket.DontFragmen |
| EnableBroadcast | 'm_currentListenerSocket.EnableBroadc |
| ExclusiveAddressUse | 'm_currentListenerSocket.ExclusiveAddr |
| MulticastLoopback | 'm_currentListenerSocket.MulticastLoop |
| Ttl | 'm_currentListenerSocket.Ttl' threw an e |
| Static members | |
| Non-Public members | |

**Figure 7.6:** A race condition in a C# program

## 7.7 Understanding C

The one C-book that stands out is still the “C Programming Language” by Kernighan & Ritchie. Even the second edition is old - from 1988 - but it includes the ANSI-C changes, which is important. Given the knowledge that it conveys it is amazingly slim. You should definitely read this if you haven’t already - and maybe even so. This does however not include the later updates. The latest is C11 (from 2011), and part of it was fixing problems that were introduced in C99. The biggest thing in C11 in relation to this book is probably the standardization om the multithreading functionality. This has been supported on all major platforms for many years, but not consistently. The two things from C99 that seems most popular are the inline comments (//) and the intermingled declaration of variables - both known from C++.

When it comes to learning about Do’s and Don’ts in C and other languages there is no book that compares with “Code Complete”. It only makes sense to throw in a few personal and well-proven guidelines:

1. Don’t be religious. There is no productivity in fighting over where the starting brace is. Live with a company standard.
2. The idea of reuse is a false goal (see further reading: “The Inmates are running the Asylum”). If your program does its job good it might be reused. If, on the other hand, you plan the code to be reusable far beyond the current context, then the code will be inefficient and thus never reused. There’s a good chance that if it finally comes to reuse, it will be in a completely other direction than originally anticipated.
3. Don’t make all coding about performance. Very few lines of code is typically run in inner-loops - making it eligible for performance-tuning. Repeating the mantra:

   *Design for Performance - Code for Maintainability*

4. In the cases where there are performance-problems, do not assume anything. Instead measure, analyze, change, measure etc.
5. Write for maintainability & readability. The single thing that can introduce most bugs, and hurt your performance most is when ignorance of the existing code causes the programmer to re-validate, recalculate, reread data etc. The next programmer does not even have to be another guy; it is amazing how fast you can forget your own code.
6. Use comments. Not the stupid kind that rewrites every line of code into other, but similar words. Instead make function headers that explains at a higher level what the function does, and when and how it is to be used. Especially whether it stands out from other functions in terms of being re-entrant or threadsafe or neither. If there is a specific order of initialization required etc. Comments inside the functions are also welcome as long as they do not just state the obvious. Typically, they can be about the general flow of the program, assumptions on the data, or in special cases explanation of a more intricate line of code.
7. It would be great to write the function-headers so that they can be used by Doxygen -

a very nice program that generates html-listings with links and graphics. Unfortunately, many projects generates this information, but never really uses it. Still the function headers are used, but mostly when the actual code is maintained. One thing that is good to have in a function header is some remarks on “who creates who” and “who kills who”.

8. Locally, in functions, single variables of the basic types are fine, but when it comes to data that lasts longer, the variables should be grouped in high-level structures such as *structs* and *arrays*. Especially in the embedded world *unions* and *bit-fields* are very popular - typically at the lower levels where you often need to deal with the same data in different ways. It can be really painful to read code where the programmer addresses everything as a byte array and “knows” that this is a flag, and these two bytes are a 16-bit signed number etc.
9. In a small embedded environment it is normally considered “bad” if the program calls *“malloc()”* or *“new”* after the initialization-phase. Allocating and de-allocating memory on the “heap” is very non-deterministic. Typically, an embedded program is not turned off in the evening, and often there is a long way to the nearest user that can do a reboot or similar. If you are programming in C++, consider making the constructors “private”, so that the compiler does not silently create objects on the heap.
10. Set your compiler to optimize for size instead of speed. Taking up less space is important in most embedded systems. On top of this, the code is very often cached, and if you can avoid a few cache-misses you will gain more performance than what a few compiler-tweaked instructions can bring.
11. “objdump -d -S <binary>” produces a listing of mixed source and assembly with gcc. Use this (or similar) before and after you make “clever” C-code. Sometimes the compiler is not as dumb as you think. Counting instructions is however not the full story. CISC CPU’s have instructions that take many cycles and some that takes a single.
12. If files are used, take care to only use random access when absolutely necessary.

    *Long ago I had implemented a Unix-program that scanned an “Intel Hex” object file, found a 26-character dummy-date, and replaced it with the current date, and then recalculated the checksum. I used random-access since I needed to go a little back and forth. This took an hour. I then changed the program to run with two passes - one random for the change of date and one sequential for the checksum, and now it took a minute.*

13. Take advantage of the fact that the logical operators && and !! are always evaluated left to right and by guarantee from the compiler only as much as needed. In other words, if you have more than one OR’ed expressions, evaluation stops as soon as one is true. Similar with a chain of AND’s; if the first is false the C-compiler guarantees that the rest are not evaluated. This is very, very practical, see Listing 7.2. Here we want to return a specific socket from an array in kernel code, it’s index must be within the bounds, and the sock in the array must contain an element (CID) matching the index. This could lead to a lot of embedded if-statements. The code in the listing is clever yes, but absolutely not too clever. Every C-programmer should know this.
14. Remember that most CPU’s will give you both the integer quotient and the remainder

when you perform integer division - but C does not pass these along. Should you need both in an inner loop this may be the reason to have a few instructions in assembly language in your code. You should do this as a macro - not a function - as a call may destroy your pipeline and take too many cycles.

```
01    if (cid < MAX_SOCKETS && cid >= 0 && all_socks[cid]
02       && all_socks[cid]->CID == cid)
03       return all_socks[cid];
04    else // errorhandling
```

**Listing 7.2:** Using limited, left to right evaluation of logic operator

## 7.8 Further Reading

- Alan Cooper: The Inmates are running the Asylum
  This is a fantastic, easy-to-read and funny book. It is mostly about GUI-design but you can easily draw parallels to general software design. The “inmates” are we - the developers - while the “asylum” is the company.
- Gamma & Helm: Design Patterns: Elements of Reusable Object-Oriented Software
  This book swept over the programming world with a refreshing view on coding: it’s not about the language but what you do with it. The authors identified a number of patterns that good programmers use independent of the language. It is now a classic.
- Pfeiffer, Ayre and Keydel: Embedded Networking with CAN and CanOpen
  This is a great book with the main focus on CANOpen, but it also covers the basic CAN very nicely. It is easy to read and has great illustrations. You can solve more problems with CAN than you may expect.
- Steve McConnel: Code Complete 2
  This is an improvement of what already was the best book ever written on the subject of how to code. Filled with good advice and examples of what not to do.
- Kernighan & Ritchie: The C Programming Language
  The absolute reference on C programming.
  Get the one with ANSI-C.
- Scott Meyers: Effective CPP
  This is not really a reference but a walk-through of 55 specific advises on coding. This really makes you understand the spirit of C++
- Jack Ganssle - Better Firmfare Faster
  Jack Ganssle has a fantastic down-to-earth approach.

# Chapter 8
# Tools

## 8.1 Simulator

*When I was studying assembly programming at the university, simulators were very popular. A simulator is a program that in our case simulated an 8080 microprocessor, and we would write our code in a "safe environment" were we could single-step, set breakpoints etc. Today I have the QEMU which simulates e.g. an ARM-environment running on my Linux desktop - which is actually running as a Virtual Machine on my Windows PC. But I really don't use it that much. Probably because I have better alternatives.*

If e.g. you are writing code for a small Linux system, you may find debugging a little crude. You may compile on a host or on the target, each with their challenges. Most Linux developers also use a Linux host (at least in the form of a virtual-machine). Often they find that there are parts of their code that they can run & debug directly on the host. If there's a lot of inputs to be simulated it becomes difficult, but if we are talking algorithms - like e.g. Digital Signal Processing - they may work on input and output files as shown in Section 7.4 - or synthetic data generated on the fly.

When you get a little further you may begin to need real-inputs from a target. This doesn't necessarily mean that you need to move all your code down in the target - yet. You may instead use a pipe or other network connection from the target to the host, and continue to do all the algorithm heavy stuff on your PC - with the nice & fast debugger and the direct connection to your version control system - while your target serves as a door to the physical world.

## 8.2 ICE - In-Circuit-Emulator

An Emulator can be fantastic. Essentially it's a computer system from where a connector can slide into a micro-processor socket. It can do everything that the microprocessor can - in real time. However - it can do a lot more. You may "trace" what happen's on specific memory cells, or break on specific patterns and sequences.

*In a project we had been looking for a "random" bug for a long time. We bought an ICE, and I set it up to break on "Read-Before-Write". Within an hour I had found the bug - an uninitialized variable. This was possible because this emulator had flags for each RAM-cell, and one of these were "W" for "written to". Whenever a read-operation was performed from a data location without this flag the ICE would break. Today such a bug may be found with static code analysis.*

Unfortunately Emulators have a number of shortcomings:

1. They are expensive, but not easy to share among team members.
2. It's not easy to apply an ICE on a finished product in the field. You need access to the PCB. This means that the box has to be opened, and maybe you need extension

PCB's to get the PCB in question "out in the open" - and extension PCB's can create all sorts of problems on their own.

3. Today most microprocessors are not even mounted in a socket. You MAY have a development version with a socket, but what if the problem always happens somewhere else?

So ICE's are rare - but they do exist, and sometimes they can be the answer to your problem.

## 8.3 Background or JTAG debugger

Many intelligent Integrated Circuits - as microprocessors and micro-controllers - have a JTAG interface. This is used during production for testing. Via JTAG a low-level program can assure that there are no short-circuits, and that IC A actually can "see" IC B. Due to the above mentioned drawbacks with ICE's, micro-controller-manufacturers decided to build in a few extra circuits in the standard CPU - often known as a "Background Debugger" - running in so-called "Background Debug Mode" - or BDM. The Background Debugger uses the JTAG-connection - only now not during production, but when debugging. If you have taken the extra cost of actually having a connector for JTAG, you can debug via this. And if the connector is even on the outside of your box, you can debug anywhere.

Typically the background debugger does not have all the features that an Emulator has. It may have one or two breakpoint registers for breaking on specific addresses, but typically they cannot break when e.g. a data-cell (variable) is changed. Sometimes you can buy extra equipment - like an extra box with RAM for tracing.

## 8.4 Target Stand-In

One virtue that the simulator has over the ICE and the background debugger is that it doesn't need a target. But, as discussed earlier, this is typically too detached from the real world. However, it is not uncommon to have embedded software developers ready to program - but no target to program in. As described earlier you can sometimes get away with using your PC as a stand-in for at least a part of the target. Another popular solution is the EVM-board. EVM stands for "Evaluation Module". This is a PCB that the CPU-vendor typically sells rather cheap. It has the CPU in question, along with some peripherals and some relevant connections to communicate with the outer world. This could e.g. be USB, HDMI and LAN - and often-times also some digital binary inputs and outputs. With such a board the embedded developers can try out the CPU for performance, functionality and even power-consumption - something that is not easy with a PC-version.

At Brüel & Kjaer we have worked 2/3 of a given project's development time, using 2 EVM's - one for the CPU and one for the DSP - together with some older hardware for interfacing to the real world. It was all tied together with a standard switch. In the final product we have internal LAN between the main components, and an FPGA that handles the switching - and then some. But for a very long time everything was running in a shoe-box with a mess of EVM's, cables etc. - a fantastic solution.

Another popular solution is to use e.g. Raspberry Pi or Beaglebone that we have seen in other chapters of this book. Another plus in the Beaglebone is that it is Open Source Hardware. This means that you have access to all diagrams etc. Do note that this involves the “Share-Alike” or “Copy-Left” license, so you cannot copy it, but it can definitely help understand the hardware. Neither the Raspberry Pi, nor the Beaglebone has the robustness to be put directly into an industrial product, but you can “train” very far, and write a lot of code.

## 8.5 Debugger

A debugger is one of the most important tools a developer can have, but it is easy to loose oversight. After debugging for an hour it may be a good idea to walk to the coffee-machine or similar - just to get away from the low-level thinking. Many people experience that they solve a problem when they are driving home from work. Nevertheless, you depend on a debugger where you can trust that if didn’t break at a given line, then it’s because it never came there. Nothing is more frustrating than almost having reproduced a problem - end then the program takes off. So make sure you know how it works.

*In my professional life I have been going back and forth between Linux and Windows as well as between programming and other activities. This may be the reason why I never have gotten comfortable with the GNU Debugger - GDB. A lot of programmers love GDB and are extremely productive with it, and as it works in almost any small or big installation this is very practical. But when it comes to debuggers I need a GUI. For this reason, I used to use Eclipse with C/C++ plugin when programming on Linux. Eclipse can run the GNU debugger very nicely - locally on the target if it is big enough and remotely on a host developer PC - see Figure 8.1. Still - in my opinion there is no debugging environment like Microsofts Visual Studio, and when I found out that there was a product called “VisualGDB” I was quite happy. Using this you can cross-develop a small Linux target from a Windows PC running Visual Studio - see Figure 8.2. Until now I have only tried this with the BeagleBone Black to which there is a standard Visual Studio plugin. This works very nice, and I am very interested to learn how it works on a custom target. The website “visualgdb.com” has a long list of 3’rd party HW that is supported.*

---

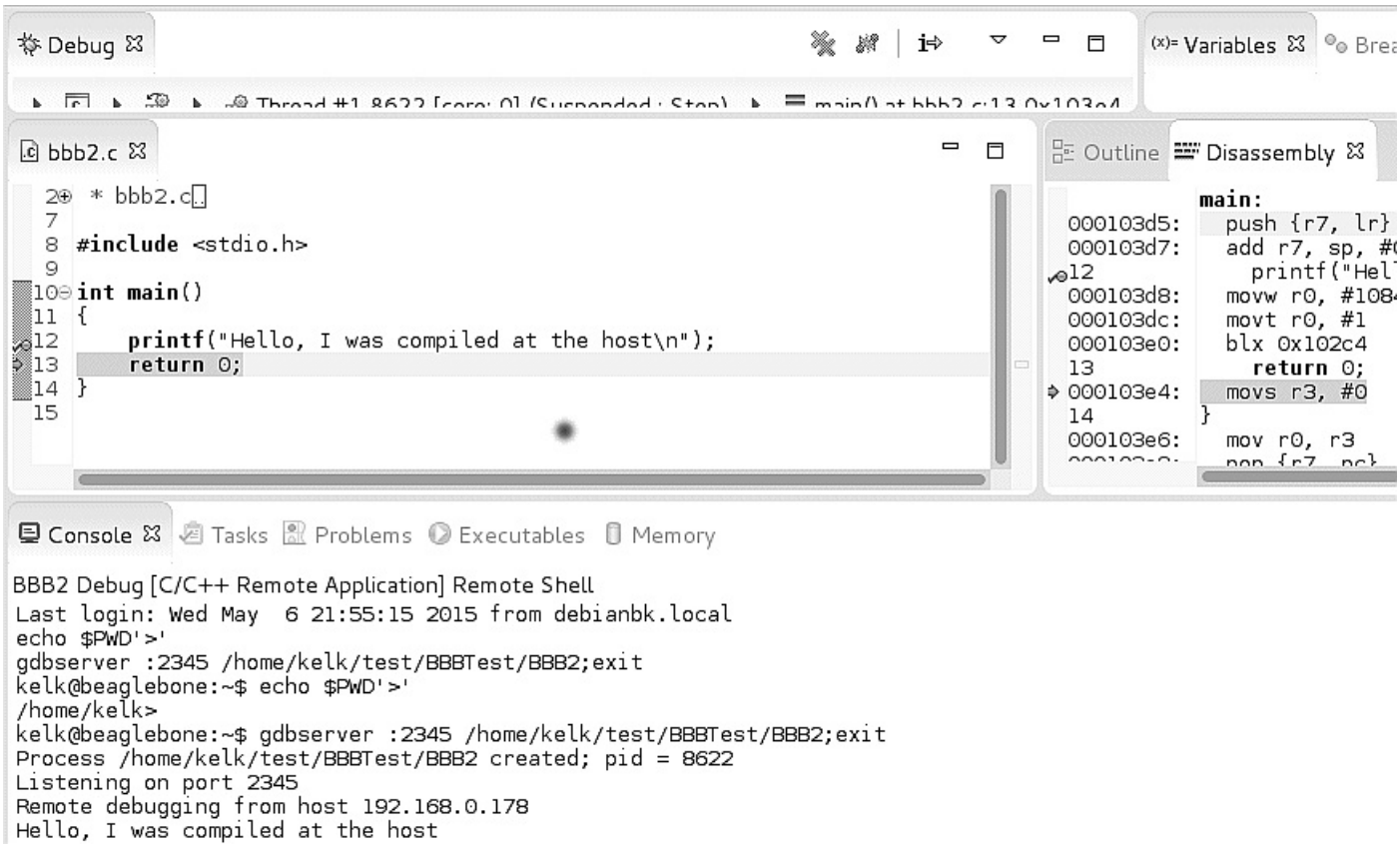



**Figure 8.1:** Cross Debugging Beaglebone using Eclipse

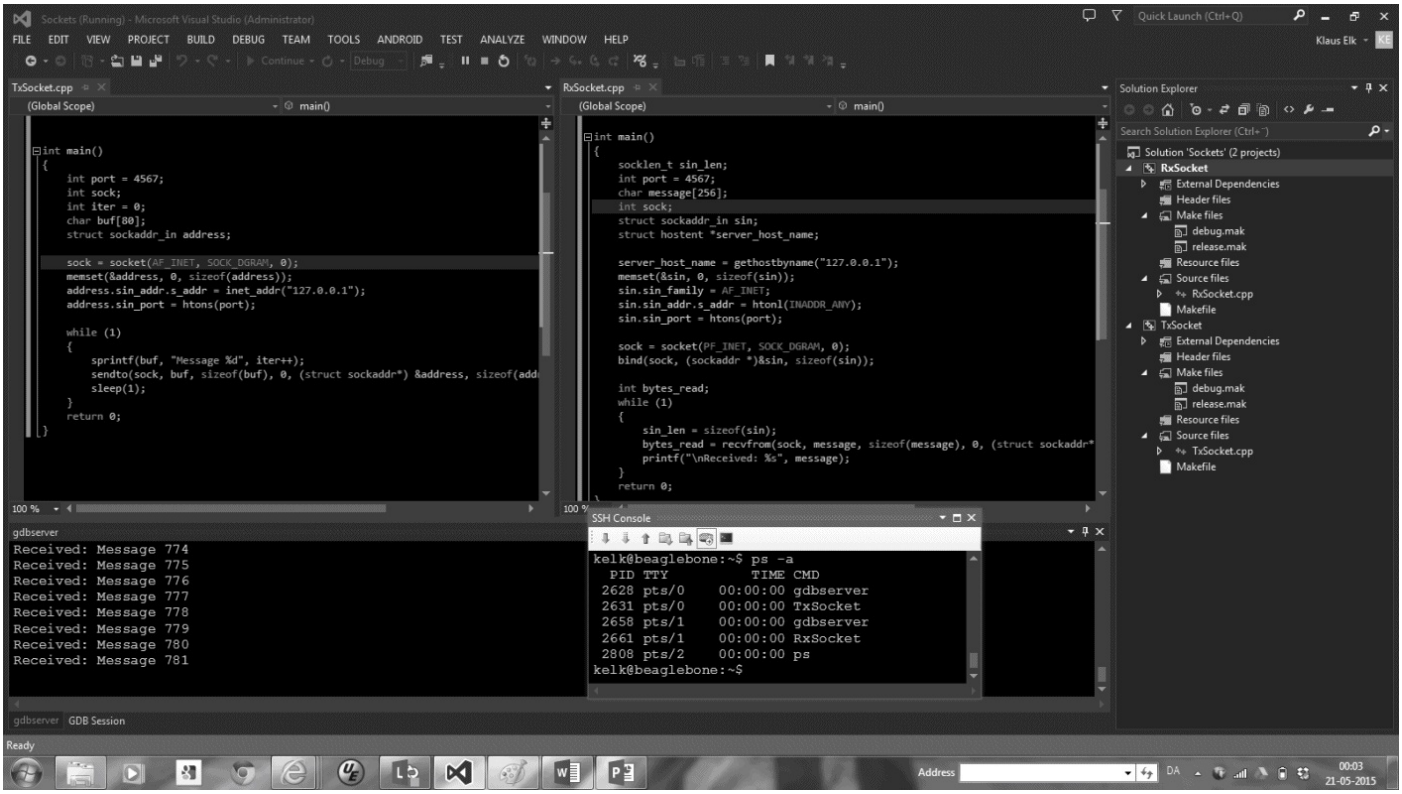



**Figure 8.2:** Cross Debugging Beaglebone using VisualGDB

## 8.6 Debugging without special tools

Single stepping is great when you know what you are looking for, but you **will** loose

overview. So it won't work when you have a problem and don't really know what is happening. *printf* -statements is often used because it requires no extra tools. It is also typically criticized (by tool-vendors and others) because it slows down the execution and requires recompiling with various compile flags set.

If you have the code-space for these statements, it can be very powerful. But don't spend a lot of time on writing *printf's* - only to delete them afterwards. Instead surround all your *printf's* with an if statement - or even two. Executing an if-statement will typically not slow anyone's code down (yes, yes - APART from that inner-loop). The "if" relates to a domain in your code - e.g. "Engine Control". So all *printf's* inserted in the application layers, as well as the allowed printouts in the driver that relate to Engine Control, are wrapped in an "if" with this flag. This can be done in C with a Macro. In your documentation (e.g. a wiki) you keep track of the flags. You also need to be able to set these flags at runtime (at least at power-up). This may be in a file or via network. You may also want to have a level of "verbosity". This is where there will be a second "if". If you have problems you typically want it all, but the CPU may simply not have the performance to give you full verbosity in all areas.

Now, if a system fails you can enable the flags in the relevant domain - without slowing everything else down. The great thing is that this can be in the field or at a customer's site - with the actual release he or she has. It is never fun to recreate a customer's problems back home. You may even ask the customer to turn on the flag, and collect the log and mail it to you. Alternatively you may login remotely or … Remember that the *printf's* should contain some description of the context - after all you may see them years after they were written. Do not forget to use the compiler directives _FILE_ and _LINE_. They will help you when you need to go back to the source to see **why** this was printed. If you do this along the way, and when there are new problems, then you will build up a helpful tool.

## 8.7 Monitoring Messages

Another great tool only makes sense if you use message passing as suggested earlier. Surely you will have a lot of functions calling functions as usual - but at some high level you have tasks that are mainly waiting on an "input-queue" for their next job. These tasks may once-in-a-while wait at another queue for a result of something that they started, but eventually they go back to the input queue for the next job. All this is handled by a system process - typically part of the kernel. Now - again at runtime - you can enable a switch that writes out - at the given time - how many messages are waiting at each inbox. This will tell you if someone is falling behind - maybe waiting for a locked resource or in an infinite loop. You may even get this printout to contain the most important fields in each of the waiting messages - that can tell you more about who is producing data and who is not consuming - and vice-versa.

In most message based systems, the messages comes from pools. Each pool is initialized to have a given number of messages of a given size in the pool. This way memory does not become fragmented. The tasks that are waiting at their input box will typically return the message to the pool when done with it. Often this is used as a throttle - aka back-pressure - so that no more than e.g. one message can be outstanding. Anyway, if

you can print out the number of buffers in each pool at a given time, you can also see if problems are building up - or have actually happened. Also here it makes sense to see the “type” field, or whatever it’s called, as that may tell you who is returning the buffers - and who is not.

## 8.8 Test Traffic

When testing, it is a common requirement to run through a lot of tests - varying only a single parameter. Listing 8.1 shows the full Python code for testing that a system accepts packets of all legal lengths - and handles those too short or too long correctly. Using normal sockets it can be difficult to create the ones with illegal length. Similarly, you may want to test with illegal hopcount, checksum etc. With the “ScaPy” library for Python this is a breeze. Note line 41 that simply holds all headers - and some data. The “/” separates the layers and if there is only the layer name it behaves “normally” and inserts counters with their right value etc. Listing 8.2 is another example where some TCP-defaults are overwritten.

An alternative to Python and ScaPy is the WinPcap library used from e.g. a C-program or tcl/tk. WinPcap comes with WireShark and is thus mainly for capturing data, but it can also send it.

```
01 #!//usr/bin/env python
02 #
03 # This is a small TCP-client that connects to a server.
04 # When connected it send the current time in text and closes.
05 # Call      : ./scanner.py IP-address [startlength] [endlength]]
06
07 import socket, sys, time
08 from scapy import *
09 conf.verb=0
10
11 defstartlen = 80
12
13 try:
14    shost=sys.argv[1]
15 except:
16    print "I␣need:␣<Target-IP>␣<IPLengthFirst>␣[<IPLengthLast>]"
17    sys.exit(1)
18
19 try:
20    startlen=int(sys.argv[2])
21 except:
22    startlen=defstartlen
23
24 try:
25    endlen=int(sys.argv[3])+1
26 except:
27    endlen=startlen+1
28
29 ip = IP()
30 ip.dst = shost
31
32 icmp = ICMP()
33
34 unanswered=0
35 try:
36    for len in range(startlen,endlen):
37       ip.len = len
38       icmp.seq=len
39       datalen = len-20-8
40
41       packet = ip /icmp/("X"*datalen)
42    #   print(hexdump(packet))
43    #   packet.show2()
44       ans,unans = sr(packet,retry=0,timeout=1)
45       for snd in unans:
46          print "No␣answer␣to␣packet␣with␣length", len
47          unanswered += 1
48 except KeyboardInterrupt:
49    print "Terminated␣by␣CTRL-C␣-␣closing␣down"
50    sys.exit(1)
51
52 print "Sent␣",endlen-startlen,"␣packets␣", unanswered,"␣unanswered"
```

**Listing 8.1:** Python and ScaPy

```
01 fin_packet = ip/TCP(dport=remote_port,sport=local_port,
02     flags="AF",seq=this_seq, ack=this_ack)/"Here␣are␣some␣data"
```

**Listing 8.2:** More ScaPy default overrides

## 8.9 The Switch as a Tool

### 8.9.1 Mirroring

If you are working with networks, you are used to switches. A switch is a nice plug'n play device that allows you to expand the number of devices you can connect. However, it is also a good tool. Wireshark is great, but it runs on your PC, and what if you want to measure between two embedded devices that are too small to run Wireshark? This is where we bring out the managed switch (or better, a TAP - see Section 8.10). You can easily buy managed switches for less than 200$. Managed switches typically has the ability to select a "Mirror Port". This means that you can ask the switch to output on this port, any data that goes in and/or out of one or several other ports. So if e.g. your two devices are connected to port 1 and 2, you set up the mirror to mirror one of these to port 3, which is where you connect the PC. This is seen in Figure 8.3 where port 1 Tx and Rx as well as Port 4 Tx only is mirrored to port 8. The biggest problem is that in a normal star-configuration with e.g. 1 Gbps connections, it is possible to have 1 Gbps in both directions at the same time which would mean that the mirror port would need to output 2 Gbps - not possible. Typically, this is not a real problem as most transmissions tend to have most traffic in one direction, but it is something that you must be aware of. If Wireshark reports lost frames this **could** be the reason.

Another practical problem is that a switch can be used in any network "as is" but when you want to manage it, this is typically via the built-in web-server. In order to be able to use this, the switch must be in your subnet. This means that you need to set the IP address, and typically to set it, you need to know the current IP-address - a classic "Catch 22". Often the easiest way to find this address is to reset the switch (typically with a pencil in a small hole), and then use the default IP-address written on a label somewhere.

Alternatively, you may connect it to your PC, fire up Wireshark, and then power up the switch. Normally, it will start to chat a little, and if the IP-address is fixed, it is easy to see what it is in Wireshark. If it is not fixed, it will be looking for a DHCP-server as we saw in Section 4.6, and we need to supply this - e.g. by connecting to the office network (or by having a server on the PC - see Section 8.11). Note that at this point you are probably not mirroring yet and you will only see the parts of the DHCP-conversation that are broadcast as your PC is connected to another port than the one connected to the DHCP-server via the office network[1] .

Most switches also have an RS-232 or USB-connection with a CLI - Command-Line-Interface, but if you only need to set a mirror, this can be cumbersome to setup, as these are very vendor-specific.

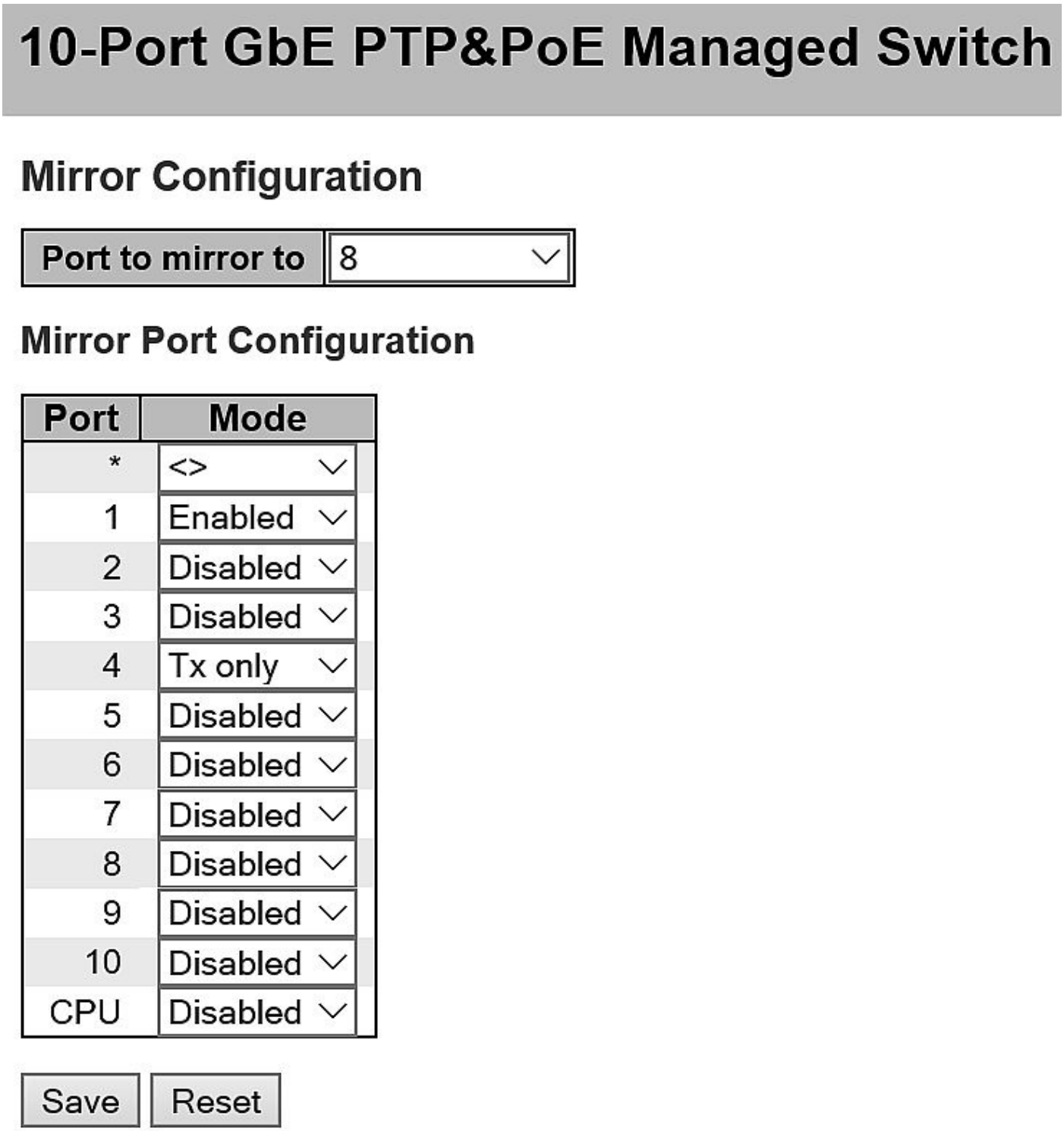



**Figure 8.3:** Mirror Setup on Port 8 in switch

### 8.9.2 Statistics

Managed switches also have a statistics page. If you have an embedded system that you suspect has problems on the physical layer, this statistics page is a good place to start - especially the “Receive Error Counters”. Naturally this only tells you about the quality of the **output** frames from the embedded system, although many TCP-retransmissions will also occur if there are problems with the **input**. To see information on the input (if possible at all)) or to dig deeper you may look into SNMP as described in Section 8.13.

10-Port GbE PTP&PoE Managed Switch

Detailed Port Statistics Port 8

Port 8 Auto-refresh

| Receive Total | | Transmit Total | |
|---|---|---|---|
| Rx Packets | 2651 | Tx Packets | 222 |
| Rx Octets | 303779 | Tx Octets | 141974 |
| Rx Unicast | 303 | Tx Unicast | 213 |
| Rx Multicast | 698 | Tx Multicast | 7 |
| Rx Broadcast | 1650 | Tx Broadcast | 2 |
| Rx Pause | 0 | Tx Pause | 0 |
| **Receive Size Counters** | | **Transmit Size Counters** | |
| Rx 64 Bytes | 1520 | Tx 64 Bytes | 18 |
| Rx 65-127 Bytes | 773 | Tx 65-127 Bytes | 26 |
| Rx 128-255 Bytes | 166 | Tx 128-255 Bytes | 11 |
| Rx 256-511 Bytes | 80 | Tx 256-511 Bytes | 85 |
| Rx 512-1023 Bytes | 104 | Tx 512-1023 Bytes | 15 |
| Rx 1024-1526 Bytes | 8 | Tx 1024-1526 Bytes | 67 |
| Rx 1527- Bytes | 0 | Tx 1527- Bytes | 0 |
| **Receive Queue Counters** | | **Transmit Queue Counters** | |
| Rx Q0 | 2651 | Tx Q0 | 0 |
| Rx Q1 | 0 | Tx Q1 | 0 |
| Rx Q2 | 0 | Tx Q2 | 0 |
| Rx Q3 | 0 | Tx Q3 | 0 |
| Rx Q4 | 0 | Tx Q4 | 0 |
| Rx Q5 | 0 | Tx Q5 | 0 |
| Rx Q6 | 0 | Tx Q6 | 0 |
| Rx Q7 | 0 | Tx Q7 | 222 |
| **Receive Error Counters** | | **Transmit Error Counters** | |
| Rx Drops | 0 | Tx Drops | 0 |
| Rx CRC/Alignment | 0 | Tx Late/Exc. Coll. | 0 |
| Rx Undersize | 0 | | |
| Rx Oversize | 0 | | |
| Rx Fragments | 0 | | |
| Rx Jabber | 0 | | |
| Rx Filtered | 452 | | |

**Figure 8.4:** Statistics page in switch

### 8.9.3 Simulating lost frames

Sometimes it is tempting to test retransmissions and general robustness by simply pulling out the Ethernet cable from the embedded device. Unfortunately - in this scenario - this typically causes a “link down” event on the device as well as the client PC, and you get to test something completely different. If, however, two switches are inserted between the device and the client PC, and the cable is unplugged between the switches, there are no “link down” events, and the cable can be swiftly inserted again.

### 8.9.4 Pause frames

Ethernet has a concept called “Pause Frames” - or rather 802.3**x** Flow Control. This has its pros and cons. Unmanaged switches typically use Pause frames if the other side of the connection does, but managed switches can be configured to turn this feature on or off. This can help investigate the pro’s and con’s in a given system. The really nice thing is that sometimes the statistics include the number of Pause frames sent. This can be seen in Figure 8.6. Here Flow control is configured on all ports on the switch. Only port 8 is connected and it has negotiated with its peer to use Flow Control in both directions. Note that this particular switch also allows you to set the max Frame size. This can be used to provoke and test the so-called IPv4 Fragmentation - see Section 4.13..

The arguments against Pause-frames are two-fold:

- Typically a connection is using TCP which has its own flow-control and the two types of flow-control may work against each other. The counter argument is that TCP’s flow-control is end-to-end and somewhat slow to react, whereas the Ethernet flow-control is on both sides of a link, and thus can assure against the packet losses that otherwise happens when a switch has full buffers and receives a frame.
- If “fast” gigabit devices and “slow” 10 Mbps devices are connected to the same switch, the slow device may cause back-pressure through the switch that stops all

other traffic through the switch for long intervals. This is a good argument for not using flow-control in such a mixed environment.

Figure 8.5 is a Wireshark capture where frame 1 ends the previous Pause, while frame 2 starts a new one. The “quanta” is a number corresponding to 512 bit times and is the timeout value. It comes into play if the Pause is not ended by a Pause with 0 as argument (as in the frame 1). The MAC source is always the transmitting port, while the destination can be the MAC-address at the other end - or the special PAUSE pattern seen in the capture - 01:80:c2:00:00:01. The Wireshark info field can be a little confusing by relating this to spanning trees. Figure 8.4 shows that a managed switch statistics page sometimes also counts the PAUSE frames sent back and forth between the given switch port and its peer (bottom of first section).

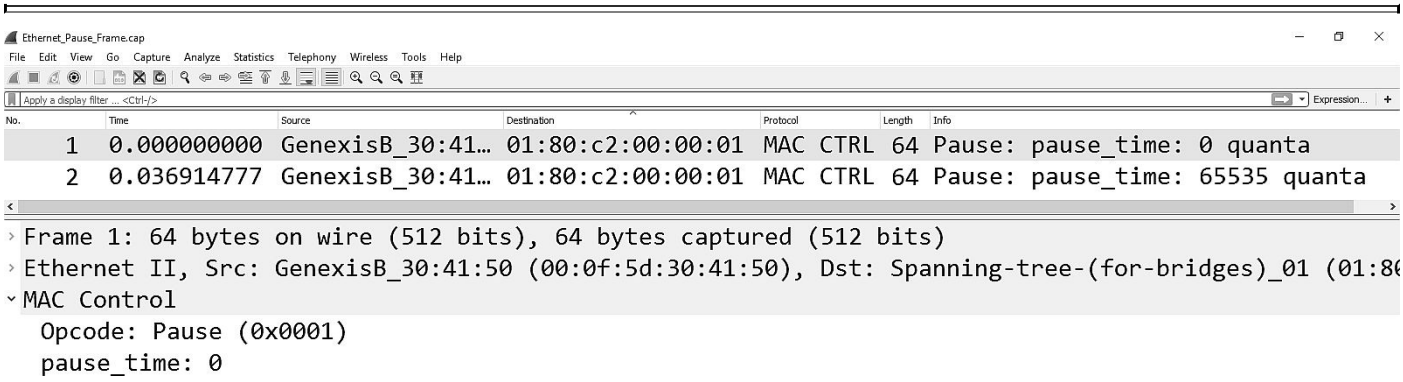



**Figure 8.5:** Ethernet Pause Frames

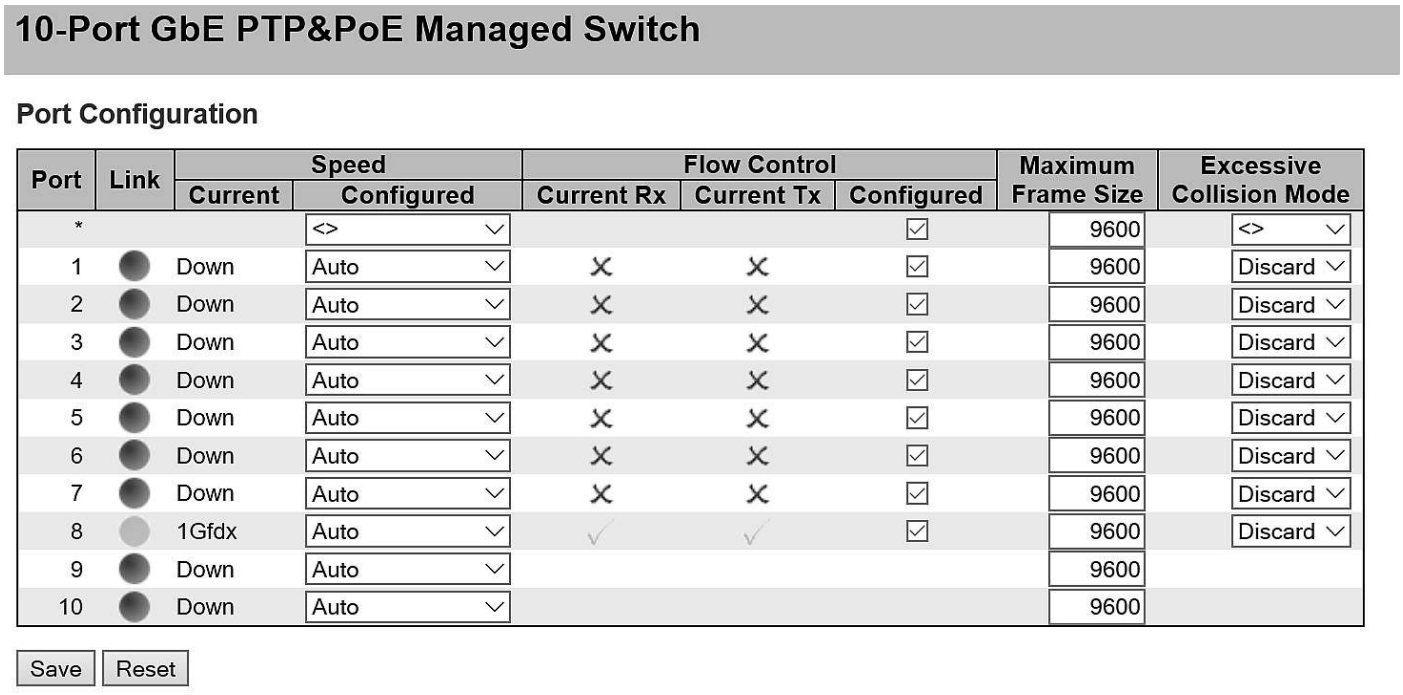



**Figure 8.6:** Port Setup with Flow Control

## 8.10 Tap

A tap is a device that can be inserted between two network-devices - with an Ethernet-

connector for each - and a third connector for a PC. See Figure 8.7. They come in different speeds - typically 100 Mbps or 1 Gbps per port. The third connector is typically a USB-connector with a driver for PC's so that Wireshark works with this. If it is USB3 it will be capable of handling the full 1+1 Gbps of traffic. Some of these taps even have built-in high-precision time-stamping that Wireshark understands. Thus it is a more professional and easy-to-plug-in tool than the managed switch in the previous section, but on the other hand something that you need to buy in good time before you need it, where a switch is readily available in most labs.


**Figure 8.7:** TAP Device

## 8.11 DHCP Server

A lot of equipment is configured as a DHCP client. If your PC is also a DHCP-client and is connected directly to such a device, then what? If the device can use "Link Local" as a fallback this is often the way to go. An alternative is to install a DHCP server on your PC. On Linux this is generally a question of setting a checkmark in your packet-manager or similar, but on Windows you need to find something that you can trust. A German developer - Uwe Ruttkamp - has made a nice one that can be found here: "dhcpserver.de". It comes with an easy-to-use setup wizard and is absolutely free to use.

Don't forget to turn it off, if you connect to e.g. a company network. The IT-guys are not too happy about competing DHCP-servers.

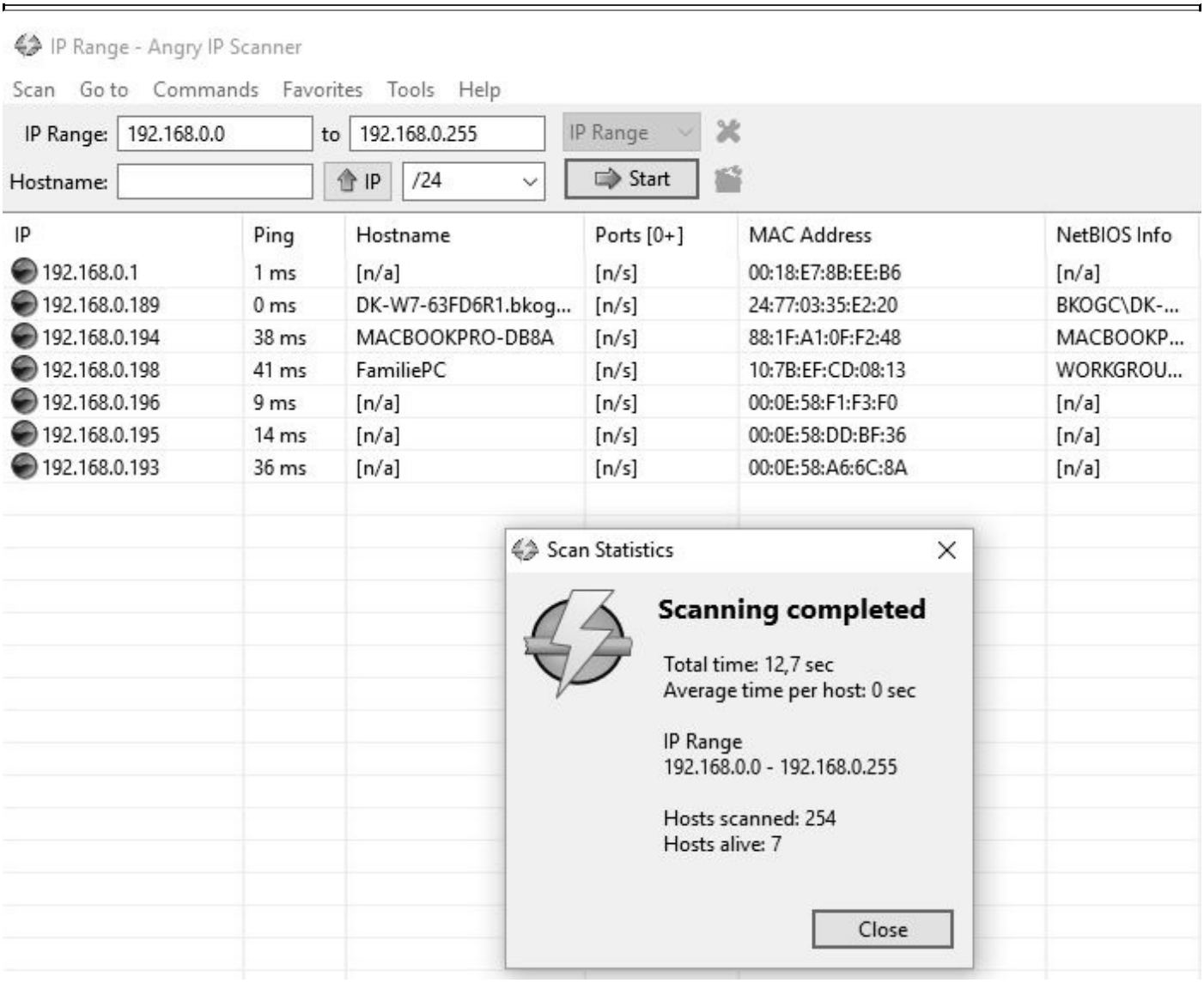
## 8.12 IP scanner

One of the problems using IP-addressing in embedded devices is that it is often

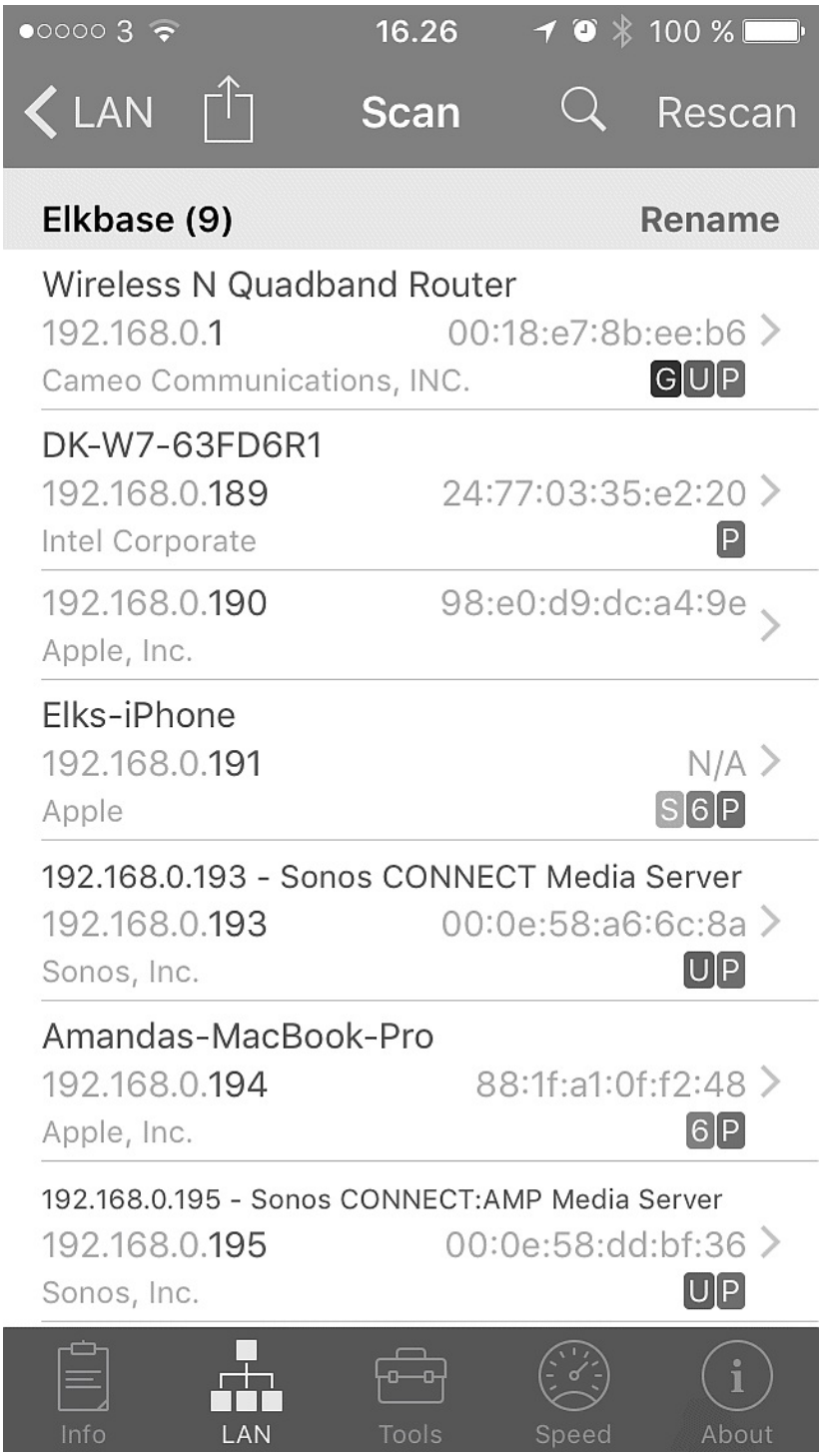
cumbersome to find or re-find the address used. Subsection 8.9.1 describes how to get around this problem on a switch. Not all devices are as chatty as a managed switch, and often the fastest way around the problem is to perform some kind of reset on the device. This may give it a known IP-address - but now you need to change the IP-address of your PC to match this IP, and first change the device IP-address and subnet, and then the PC's back to the intended subnet. This is also a cumbersome method.

If you are on a small network the fastest method is simply brute force - trying all addresses - using a small tool. Figure 8.8 - left - shows the popular "Angry IP scanner" in action. The right part of the same figure shows the result of a similar action - this time using a small app called "Net Analyzer" on an iPhone. Via small icons this tells us: **G**ateway, **S**canning Device (the iPhone itself), **P**ingable, **U**PNP/DLNA services available and IP**6** availability.

As this is all performed on a small home network a third alternative could be to look at the wireless router as this typically has a table showing devices on the network. But sometimes you just have a device connected directly to a PC, and in such a case only the first solution works.




(a) Angry IP scanner



(b) iPhone Net Analyzer

**Figure 8.8:** IP scanning in two ways

## 8.13 SNMP

An SNMP - Simple Network Management Protocol - server is implemented in many network devices such as switches and routers, but also in Operating Systems. The network device implements a MIB - Management Information Base. This is an Object Model, as described in Section 7.5. The server collects a huge amount of useful information that can be retrieved with the help of an SNMP-client. Figure 8.9 is a screen-shot from such a program from iReasoning (not free, but with a trial period of one month).

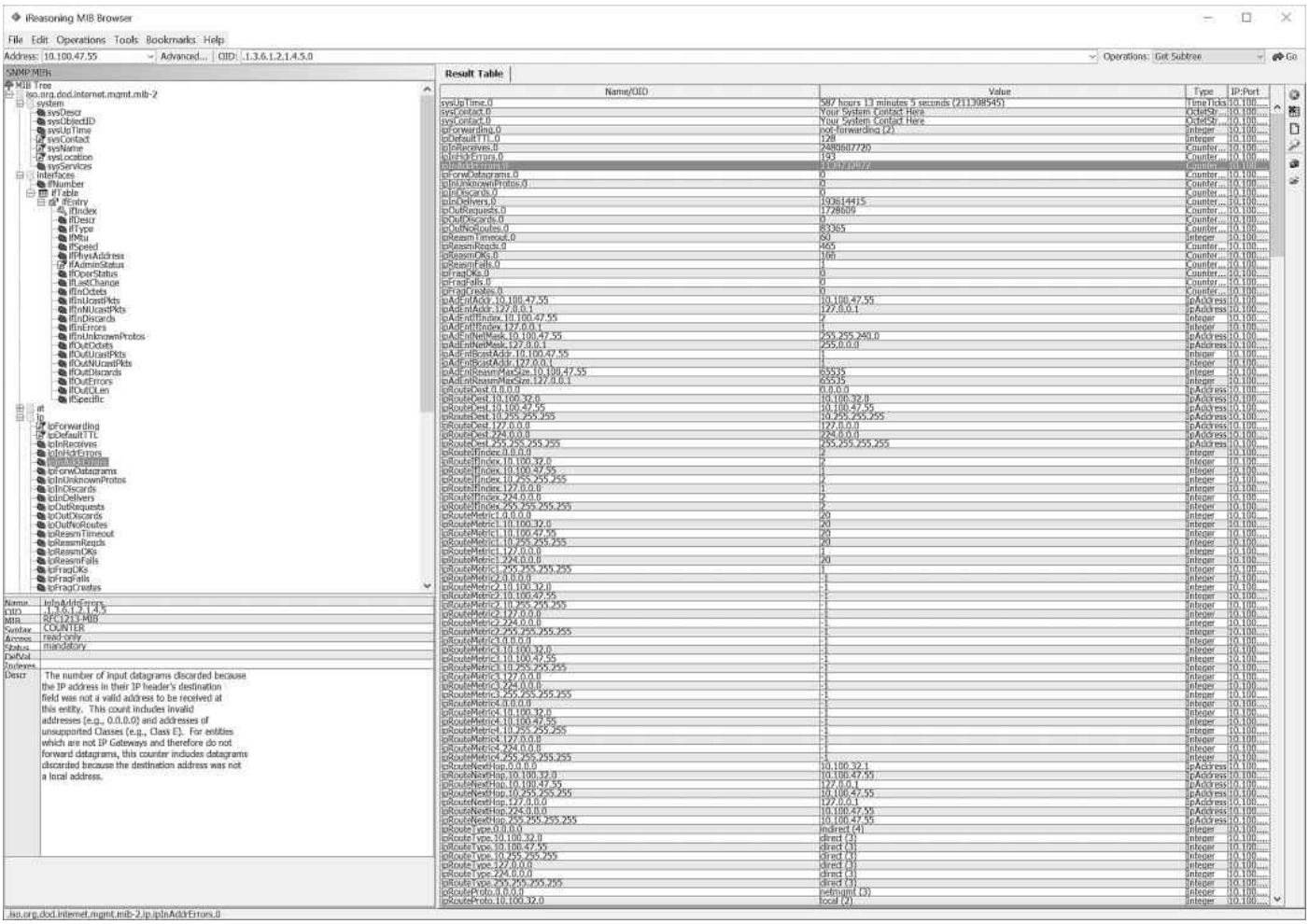


**Figure 8.9:** SNMP Client with lots of data from Windows CE

The figure is too busy and detailed to be read, so here's a description of it: At the very top left we have the IP-address of the network device. Just below the IP-address is the MIB in tree-form. Top right is the actual command sent, here "Get Subtree". This is very practical and is the one used to generate the right side with the actual current data from the MIB. The lower-left window shows information on the selected line. Clearly there is a lot of information to comprehend, but given the fact that it is served right under your nose with almost no work invested, it may very well be worth a try. As usual it is a good idea to have a working device to compare to when looking for the needle in the haystack.

## 8.14 strace

This is not a Linux book, and we will not go into all the nice tools on Linux, but *strace* is mentioned because its different from most others tools in that it can be used as an afterthought and it is also great as a learning tool.

When coding the IPv6 UDP-socket program in Section 4.19 something was wrong. To dig into the problem "*strace*" was used. This is a Linux tool that is used on the commandline in front of the program to be tested - e.g. "strace ./udprecv &". In this case it was used in two different terminals to avoid the printout to mix. Listing 8.3 shows the final output from the working programs (with some lines wrapped). The receiving program blocks while writing line 7 (*recvfrom*). In this case the problem was that one of the "sizeof" calls was working on a IPv4 type of structure (fixed in the listing shown).

```
01 ----- First Terminal: strace ./recvsock ----
02 socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
03 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
04 bind(3, {sa_family=AF_INET6, sin6_port=htons(2000),
05 inet_pton(AF_INET6, "::", &sin6_addr), sin6_flowinfo=0,
06 sin6_scope_id=0}, 28) = 0
07 recvfrom(3, "The␣center␣of␣the␣storm\n\0", 100, 0,
08 {sa_family=AF_INET6, sin6_port=htons(45697),
09 inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0,
10      sin6_scope_id=0}, [28]) = 25
11 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), …}) = 0
12 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
13     -1, 0) = 0x7fc0578cb000
14 write(1, "Received:␣The␣center␣of␣the␣stor"…, 34Received:
15 The center of the storm) = 34
16 exit_group(0)                           = ?
17 +++ exited with 0 +++
18 ----- Other Terminal: strace ./sendsock ----
19
20 socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
21 sendto(3, "The␣center␣of␣the␣storm\n\0", 25, 0,
22 {sa_family=AF_INET6, sin6_port=htons(2000),
23 inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0,
24 sin6_scope_id=0}, 28) = 25
25 exit_group(0)                           = ?
26 +++ exited with 0 +++
```

**Listing 8.3:** strace IPv6 recv & send

## 8.15 Wireshark

Wireshark is used extensively in the Network chapters in this book and surely it is **the** most important tool when it comes to Networks. It is not always easy to use and a given capture can look very different from one day to another if your preferences has somehow changed. Here are some simple guidelines:

- In the bottom right corner there is an innocent looking little field called “Profile”. Open the dialog and click the link that shows where the profiles are saved on your PC. Now create backups.
- It is normally very important to show the “info” and coloring according to the highest level protocol - see “View”-“Coloring Rules”. In this book the coloring has been removed to give the best black & white print, but normally the colors are a great help.
- In “Preferences”-“Protocols” you should select IPv4 and then uncheck “Validate the IPv4 checksum”. Do the same with TCP. As this validation is normally done in your Network Interface Card you will get false checksum errors unless this is done.
- In the main menu “Analyze” there is an “Expert Info”. This is a good place to start on a capture.
- In the main menu “Statistics” under “TCP Stream Graphs” there are some really valuable submenus that will help you get an overview of your TCP communication. Remember first to select a packet from the relevant flow - preferably in the relevant direction.
- In the middle view where a selected packet is dissected you can select many of the header fields, and with a right-click select to “Apply as Column” or “Apply as Filter”. This hidden gem is invaluable.
- In the top view where you see an overview you can e.g. select a HTTP-packet, right-click it and in the context-menu select “Follow TCP stream”. This gives you a filter on the stream - but also a nice window with the full Request and Response in clear text.
- If there is a lot of traffic it is a good idea to disable screen updates in order not to lose

frames in the capture.
- When analyzing traffic flow, the actual traffic is often irrelevant. You can ask Wireshark to only save e.g. the first 60 bytes of each frame. This will give the PC better performance.

## 8.16 Network Commands

The following table is a list of practical network-related commands. Most commands have the same name on Windows and Linux, but where they differ the Linux name is given in parentheses.

| Command | Usage |
|---|---|
| arp | Show or edit the arp table<br>(Known IP-addresses and their MAC-address)<br>“-a” shows all<br>“-s” can be used to add manually |
| ipconfig<br>(ifconfig) | Show current netcard configurations.<br>“/all” gives more info<br>“/renew” causes a DHCP update |
| netstat | Show current TCP/UDP connections,<br>their state and attached processes<br>“-a” shows all - including listeners<br>“-b” shows the executable (warning: slow!)<br>“-n” uses numbers instead of names for ports<br>“-o” shows process ID’s |
| nslookup | DNS-lookup. Shows the correspondence between IP-addresses, names etc. |
| pathping<br>(-) | Does a tracert - then pings each host on route and generates statistics |
| ping | Simplest way to see if there is “hole-through” to a<br>remote host |
| route | Show or edit the route table<br>(routes to networks and hosts)<br>“PRINT” shows the table<br>“ADD” allows adding |
| telnet | Client shell - redirects you keyboard commands to<br>remote - and see its output. |

| | |
|---|---|
| | Can be used with non-default portnumbers to do manual http etc<br>“Set localecho” is useful when simulating e.g. http |
| tracert<br>(traceroute) | Trace the route to a given host.<br>New packets are sent with increasing hopcount.<br>This is decremented at each router as usual.<br>When 0 a timing message is sent back home. |

## 8.17 Whiteboard

Right before we go into the subject of “Processes” it is fitting to bring up the Whiteboard as one of the most important tools - almost as important as an editor and a compiler. Surely a Whiteboard is indispensable in the start of projects when regular design brainstorms are needed. Also, in my team, if someone is stuck in a problem we call a meeting at the Whiteboard. This is not the same as a scrum meeting because it takes longer time, involves fewer people and is more in-depth, but it may have been triggered by a scrum meeting. We always start the meeting with a drawing. The person who is faced with the problem draws the relevant architecture. During the meeting other people will typically add surroundings to the drawing, and we will also come up with a list of actions on the board - typically of investigative nature. With modern phones it is no problem to take a picture and attach this to a mail as a simple “minutes”.

It is important that the team members themselves feel confident in the setup and will raise a flag when they need such a meeting. Should this not happen it falls back on the team leader to do so.

## 8.18 Further Reading

- technet.microsoft.com/en-us/sysinternals
  A great site for free windows tools.
- trac.edgewall.org
  A free tool that combines wikki, bug-tracking and source-control.
  Ideal for small teams.
- atlassian.com
  A larger complex of tools in the same department as “trac”.
- dhcpserver.de
  A free DHCP-server for Windows
- angryip.org
  An IP-scanner for Windows, Mac and Linux

# Chapter 9
# Processes

## 9.1 Poor Mans Backup

There are many great backup programs, but you can actually get far with the free stuff that is on your system. On Windows “robocopy” is very persistent in its attempts to store your data. The following text in a BAT-file (as a single line) copies “MyProject” to a Google drive - it could also be a network drive etc.

```
01 robocopy c:\Documents\MyProject
02 "C:\Users\kelk\Google␣Drive\MyProject"\
03 /s /FFT /XO /NP /R:0 /LOG:c:\Documents\MyProject\backup.txt\
```

**Listing 9.1:** Poor Mans Backup using Robocopy

The following table describes the options used - there are more.

| Option | Usage |
|---|---|
| /s | Include Subfolders |
| /XO | Exclude files older than destination |
| /R:10 | 10 Repetitive attempts if it fails |
| /LOG | Logfile - and LOG+ appends to file |
| /NP | No Percentage - do not show progress |
| /FFT | Assume FAT File Times (2-second granularity) |

In Windows you can schedule the above bat-file to be executed daily:

1. Open Control Panel
2. Select Administrative Tools
3. Select Task Scheduler
4. Create Basic Task
5. Follow wizard. When asked how often? - say once a week. And then check-mark the relevant days.
6. If you are copying to a network drive assure that the backup only tries to run when you are on the domain and have AC-power. Do not require the PC to be idle.
7. Select a time when you are normally e.g. at lunch

The Linux equivalent to the above is using the “tar” command to copy and shrink the data and “crontab” to setup the “cron” daemon that schedules the regular backup. Contrary to the above Windows concept this is described in numerous places and is skipped here.

There is a general pattern for scheduling jobs in this: Split the code in **what** and **when**.

## 9.2 Version Control

There is a lot of hype surrounding the use of Version Control. The most important thing however is that some kind of Version Control **is** indeed used. The second priority is that check-ins are performed daily or at least weekly, and that when the project is in the stabilization phase, single bug-fixes are checked in atomically with bug-number. This is important because the most error-prone type of code is bug-fixing. You - or someone else - would want to be able to:

1. Make a “shopping list” of all bug-fixes introduced between version A and version B. Most customers are mostly in a state where they want the exact same version as yesterday - only with a particular bugfix. This is not very practical for the developers as it is rarely the same bugfix the different customers want. The best thing we as developers therefore can do is to document exactly which bug-fixes are in the bag. When this is possible it may actually trigger “ah - yes I would actually like that fix too”, and at least it shows your customer that you as a company know what you are doing, and what you are asking him/her to take home.
2. Be able to answer exactly which version the customer needs in order to get a specific bugfix. This relates to the above - a minimal change-set.
3. Know which bugfix disappears if something is rolled back due to new bugs or unwanted side-effects.
4. Know that any roll-back will roll back a full fix - not just a part of it.

Another requirement to a version control system is the ability to make branches easily and equally easy merge them back.

## 9.3 Build and Virtualization

Virtualization is a fantastic thing. Many developers use it to run Linux inside their Windows PC. This type of usage was probably also the original intended one, but today it also makes a lot of sense to run a “Guest OS” inside a “Host OS” of the same type. When working professionally with software, FPGA or anything similar that needs to be built, it is important to have a “build PC” separate from the developers personal computers. It has the necessary tools to perform the build(s) and distribute the results and logs - but no more.

Most development teams today use a repository - Git, Subversion, PVCS or whatever (see Section 9.2. But typically this only covers the source, not compiler, linker and other binary tools, the environment settings, library folders etc. If you think about how much time it can take to setup a build-system that really works, it also makes sense to keep this whole setup as a single file that can be taken out at any time and used - with all the above mentioned files completely unchanged. Something that is built today may otherwise not be able to build on a PC two years from now. So preserving the complete image of the build PC over time is one good reason to virtualize it. Sometimes even the developers may prefer to use such a virtual PC. The best argument against this is performance, but if this is

not something that you build daily, but e.g. a small “coolrunner” or similar programmable hardware device that is changed once in a blue moon, then dusting off the virtual image really makes sense. It is amazing how much a standard developer PC actually changes in a year. The build-machine in the virtual PC must have all kinds of silent changes - like Windows Updates - turned off.

The best known tool for Virtualization is probably VMWare, but the free Oracle VirtualBox also works great.

## 9.4 Static Code Analysis

If possible, include some static code-analysis in your build. There is no need to spend days finding something that could be found in a second by a machine at night, while you sleep. Often these static tools can be configured to abide to e.g. the rules of MISRA (the Motor Industry Software Reliability Association). Figure 9.1 is from a tool called “PREfast” within the Windows CE environment (Courtesy Carsten Hansen). If you can include tools like this from the start of the project, you naturally get more value from it. But having it aboard from start also enables the team to use the tool instead of enforcing rigid coding rules. An example is when you want to test that variable a is 0 and you normally would write:

`if (a == 0)` - *but instead write:*

```
if (0 == a)
```

The first version is natural to most programmers, but some force themselves to the second, just to avoid the bug that happens if you only write “=”. We should be allowed to write fluent code and have a tool catch the random bug in the nightly build.

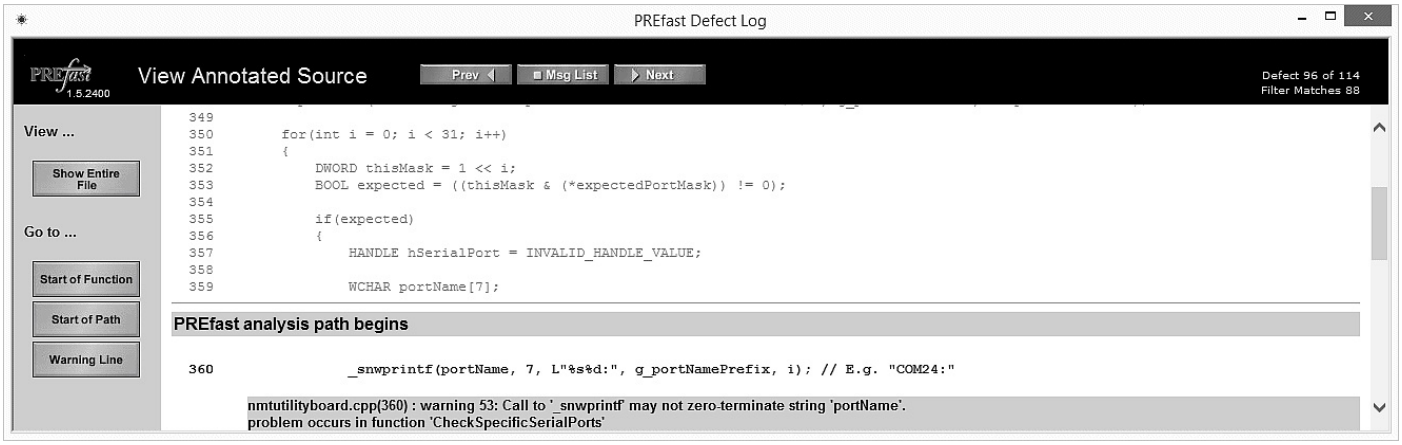



**Figure 9.1:** Prefast in Windows CE.

## 9.5 Inspections

Use inspections - at least on selected parts of the code. Even more important, inspect all the requirements and the overall design. Even if you are working with agile methods there will be something when you start. The mere fact that something will be inspected has a positive influence. On top of this comes the bugs found, and on top of this we have the silent learning that programmers get from reading each other’s code. Be sure to have a clear scope - are we addressing bugs, maintainability, test-ability or performance? The

most important rule in inspections is to talk about the code as it is there “on the table”. You say “…and then the numbers are accumulated, except the last”. You don’t say “…and then he accumulates the numbers, forgetting the last”. Programmers are like drivers - they all believe that they are among the best. Criticizing the code is one thing, but don’t criticize the programmer.

## 9.6 Defect Tracking

Tracking bugs - or defects - is an absolute must. Even as a single developer it can be difficult to keep track of all your bugs, prioritize them and manage stack-dumps, screen shots etc, but for a team this is impossible without a good defect tracking tool. With such a tool you must be able to:

- Enter bugs, assign them to a developer and prioritize them
- Enter metadata such as “found in version”, OS-version etc.
- Attach screen-dumps, logs etc.
- Assign the bug to be fixed in a given release
- Send bugs for validation and finally close them
- Receive mails when you are assigned - with a link to the bug
- Search in filtered lists

As described in Section 9.2; when you check in updated code in your software repository, you should do this in an “atomic” way. This means that all changes related to a specific bug are checked in together, and not with anything else.

It is extremely important that you can link to any bug from an external tool. This may be mails, a wikki or even an Excel-sheet. This means that each bug must have a unique URL[1] . This effectively shuts out tools that require you to open a database-application and enter the bug-number.

When you have established a base of bugs, and have prioritized and assigned them for specific releases you can start to apply statistics. You may e.g. plot the number of “Open, high-priority defects on project X for release Y” as a function of calendar time. This can be done in Excel. This is a “burn-down chart”. If you are on the right track it will have a down-going trend and you may extrapolate to see when your project is ready to be released. A graph of the same family is shown in Figure 9.2. It is from a small one-man hobby-project and it shows bugs created and resolved as function of time. The distance between the two curves is the net change - in this case positive although this is hard to see in black & white. This graph is generated directly by “Jira”- a tool from the Australian company “Atlassian”. It integrates with a wikki called ’Confluence’.

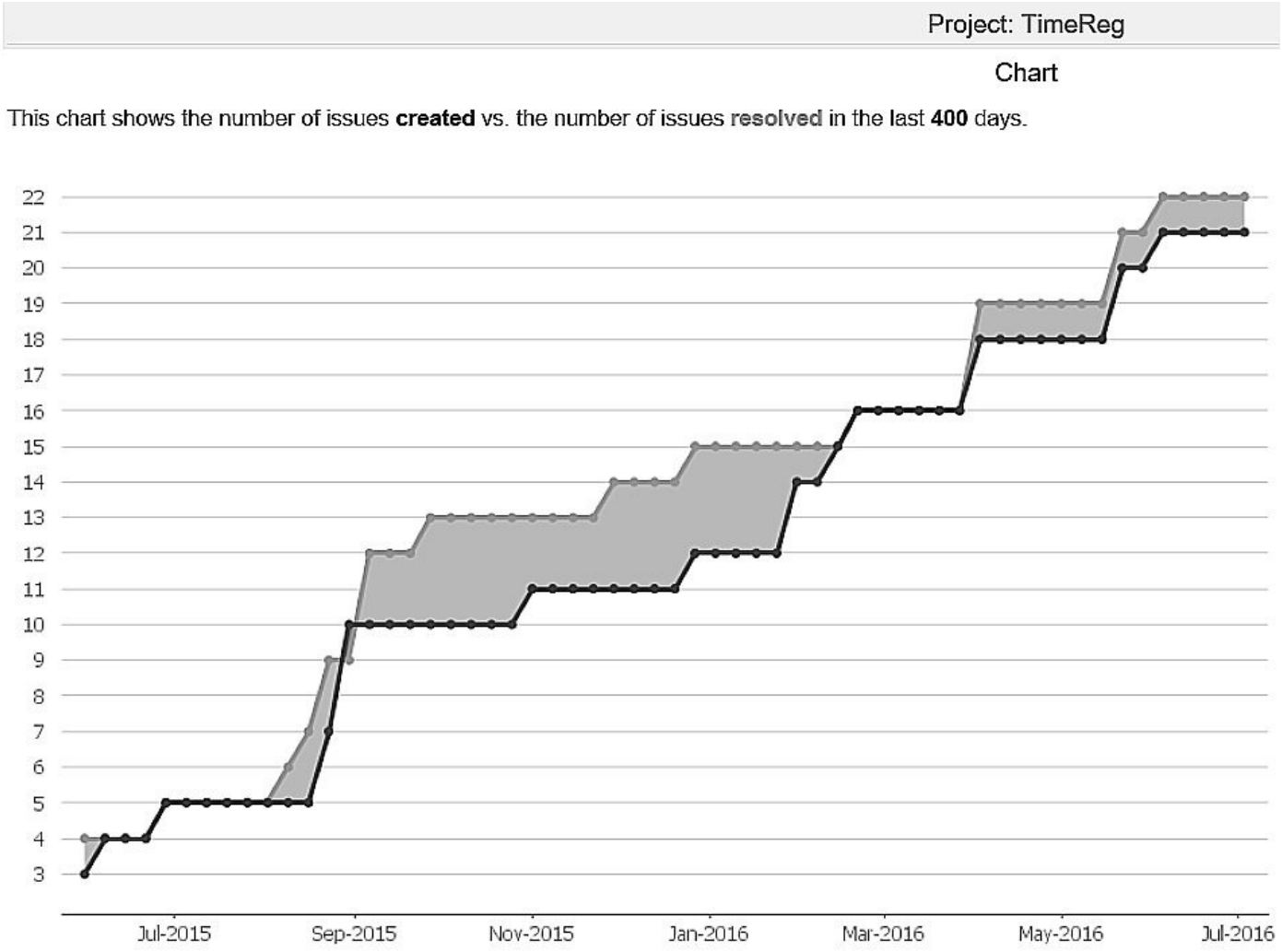



**Figure 9.2:** Incoming bugs versus fixed bugs in Atlassian Jira

Another recommended tool is the open source “Trac” found here: trac.edgewall.org.
This tool can do all of the listed requirements, and is integrated with a wikki and a source-code repository - you can choose between CVS, SVN (Subversion) and GIT - with bugs known as “tickets”. I used this in a “previous life” and it worked very well in our small team. It runs on top of an Apache server - which runs on Linux as well as on Windows.

## 9.7 Collaboration

Apart from the more formal documents, most projects need a place to collaborate. As described in Section 9.6 it is not uncommon to find tools that integrates a wikki with a bug-tracking system. This is nice, but not a must. As long as any bug can be referenced with a URL, you can link to it from any wikki - and that is important.

Documentation that you may need to dig out years from now, belong in e.g. Word-files in a repository that can manage versions and access rights. This may even be necessary for ISO-9001 requirements etc. But documentation that is meant for discussions and decisions in the near future are often better used in a wikki type of system.

For further reading - see the section on this in the tools chapter.

# Epilogue

The main discussion today among those who are working with IOT is how to actually make money. Many devices are still physically sold or rented to end customers and new opportunities are popping up in this field. If you e.g. buy a new heat system for your house, you may be offered a remote “app” for your phone - together with a box in the heat system that communicates with a cloud-server - that communicates with your phone wherever you are. The box is seen and maybe even toyed with by you as the user - although the app may be more interesting. Another example is a bath-weight - and in this case the design and physical interaction becomes more relevant.

A growing fraction of devices may however never be seen by end-users. These devices will enable “smart parking”, “smart traffic-control” etc. and live their life in anonymity - providing data to a cloud-server that shares its data to PC’s, phones and tablets - sometimes to the general public - sometimes to more closed groups. This means that the person who actually sees such a device is a technician installing it. He will prefer an easy setup as well as general robustness over a neat design. The owner of the device will be a professional who thinks in terms of TCO - Total Cost of Ownership. This future owner is right now considering how to get paid for data - or maybe how to use data to create more efficient workflows, thus saving money. The days where users would never accept to pay for something accessed via the internet are history - many people are buying subscriptions for e.g. music and movies - and what is the next thing?

I hope that this book has helped you on the technical side and that you will be giving new and interesting answers to the above questions.

*The author may be contacted at his facebook page:*
*facebook.com/klauselkbooks and homepage: klauselk.dk*

[1]Sometimes this happens indirectly via a Bluetooth bridge or similar

[1]The locater is typically integrated with the linker so don’t worry if you haven’t heard about it before

[2]Except for the NMI - Non-Maskable Interrupt if such exists

[1] A similar term is used in relation to TCP sockets

[1]We use the term “host” on the application layer, but devices communicating on the Ethernet layer are called “nodes”

[2]This is not so easy to spot as IPv4-addresses are given in decimal and MAC-addresses in hexadecimal

[3]It may not even be an Ethernet link

[4]Link-Aggregation aka “Teaming” is a practical concept that allows the use of several parallel cables - e.g. between a multiport NIC and a switch.

[5]Depending on the “coding scheme” the bits are almost sent “as is” or coded into softer waveforms - unrecognizable on an oscilloscope

[6]Confusingly *connect()* **is** possible. It does not make a connection but allows the use of *send()* instead of *sendto()* etc.

[7]Slimmed to fit on the page

[1]These terms may vary between vendors

[2]The graph says the capture is called full_rec_nomouse. This is because a wireless mouse affected the first recordings.

[1]Complex numbers is out of the scope of this book.

[2]This can be done using a concept called “Overlap-Add”

[3]With base 16 hexadecimal numbers you continue counting after 9: a, b, c, d, e, f - and then 10 which is 16 in the decimal system.

[4]Especially with IIR-filters we may need “Controlled Rounding” - yet a reason for a filter design program

[1]Things like IP-address and sensor-serial numbers will differ between devices

[2]Courtesy of ITU, found at https://www.itu.int/rec/T-REC-G.774-200102-I

[1]Yet an alternative is an IP-scanner - see Section 8.12

[1]One of the important rules in REST - see Section 4.11