EMBEDDED SOFTWARE FOR THE IoT

3rd Edition

KLAUS ELK

DE G PRESS

Klaus Elk

Embedded Software for the IoT

Klaus Elk

Embedded Software for the IoT

3rd edition

DE G PRESS

ISBN 978-1-5474-1715-5
e-ISBN (PDF) 978-1-5474-0102-4
e-ISBN (EPUB) 978-1-5474-0104-8

Library of Congress Control Number: 2018959434

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

See also https://klauselk.com
Published by Walter de Gruyter Inc., Boston/Berlin
Cover image: Tramino / iStock / Getty Images Plus
Typesetting: VTeX UAB, Lithuania
Printing and binding: CPI books GmbH, Leck

www.degruyter.com

About De|G PRESS

Five Stars as a Rule

De|G PRESS, the startup born out of one of the world's most venerable publishers, De Gruyter, promises to bring you an unbiased, valuable, and meticulously edited work on important topics in the fields of business, information technology, computing, engineering, and mathematics. By selecting the finest authors to present, without bias, information necessary for their chosen topic *for professionals*, in the depth you would hope for, we wish to satisfy your needs and earn our five-star ranking.

In keeping with these principles, the books you read from De|G PRESS will be practical, efficient and, if we have done our job right, yield many returns on their price.

We invite businesses to order our books in bulk in print or electronic form as a best solution to meeting the learning needs of your organization, or parts of your organization, in a most cost-effective manner.

There is no better way to learn about a subject in depth than from a book that is efficient, clear, well organized, and information rich. A great book can provide life-changing knowledge. We hope that with De|G PRESS books you will find that to be the case.

https://doi.org/10.1515/9781547401024-201

Contents

Preface

The internet of things is here, and soon 50 billion devices will be “connected.” So we are told. This raises the question: “Who is going to program all of these devices?”

In a major survey at “StackOverflow” in 2018, 5.2 % of the 100k responding developers claimed to work with embedded applications or devices. This is twice the percentage from the same survey in 2016. Still there is a large potential in attracting developers from the remaining 94.8 %.

Common to all these developers is the overwhelming amount of new domains they need to get into, on top of their basic programming skills.

The “2018 IoT Developer/Engineer Census and Analysis” from VDC Research states that “growth and demand has slowed for traditional engineers as engineering companies seek out ‘jack-of-all-trades’ IoT developers with domain-specific skills and the cloud/IT skills to build connected solutions and applications.”

It is exactly this multitude of skills that this book aims to bring to the reader. The author presents solid basic knowledge in the relevant domains in a structured way. This creates a strong foundation, onto which all the scattered details from the web may be attached.

Throughout the book, the author draws on 30+ years of good and bad real-life experiences from the private industry as well as from university teaching, in an informal and relevant way.

What is new in this edition?

Compared to the first edition, this book has two new chapters in the “IoT Technologies” part. It may not be a surprise that one of these is about internet security. This area is growing in importance as fast as the internet of things is growing in size. The other new chapter is about statistical process control (SPC). This is a less obvious choice. However, as the introduction to this book shows, SPC is an important part of “Industry 4.0,” another term closely related to IoT.

On top of these two new chapters, all the existing chapters are updated. The previous “Processes” chapter has been changed to “Code Maintenance” and now includes sections on Yocto and especially git, while similar changes have been made in other chapters. In terms of pages, this edition is more than 50% longer than the first.

The WireShark screenshots in the central network part are easier to read, while the many new figures and tables improve the reading experience.

This third edition is published by De Gruyter. This has meant countless improvements to the content as well as the print and design. Many details are updated to 2018 status using new information, and Python is now central in simulations.

https://doi.org/10.1515/9781547401024-202

About the author

Klaus Elk graduated as Master of Science in electronics from the Danish Technical University in Copenhagen in 1984, with a thesis in 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 Bachelor's degree in Marketing. In a period of 10 years, Klaus—besides his R&D job—taught at the Danish Technical University. The subjects were initially object oriented programming (C++ and Java), and later the Internet Protocol Stack. Today he is R&D Manager in Instrumentation at Brüel & Kjær Sound and Vibration.

Acknowledgments

I am very grateful to Stuart Douglas for his scouting—bringing this book into the De Gruyter family. Thanks to my editor, Jeffrey Pepper, for his patient scrutinizing work—my overuse of "initial caps" and hyphens alone did not make this easy. Jeffrey has provoked many improvements to the text and figures that enhances the reading experience. Likewise, I want to thank Paul Cohen for his good in-depth technical review. Finally, thanks to my family for listening patiently and for their acceptance of my many hours by the PC.

Klaus Elk

1 Introduction

1.1 The tale of the internet

Local Area Networks (*LAN*s) became accessible during the 1970s. This technology allowed computers within a building to connect to each other. The typical applications were printing and basic file sharing, leaving the rest to the user. With the appearance of the IBM PC in the mid-1980s, the market grew, and we saw a regular war between network vendors with each of their technology, for example, *Token-Ring, Token-Bus, Ethernet, Novell, AppleTalk,* and *DECnet*. Especially, the first three fought for the general market, whereas AppleTalk and DECnet, respectively, were specific to Apple and Digital. Novell quickly adopted Ethernet in its solutions—successfully focusing on network software for years. Even though Ethernet was not the technically most advanced solution, and only covered what we now know as the two lowest layers of the OSI stack, it won the war. This is a great demonstration of how a solution which is available and "good enough" can gain traction, and become popular, leading to lower prices, more sales, etc.

Before Ethernet became the clear winner, engineers and scientists were struggling with how to scale the LANs to become more than local. Different physical layers, addressing schemes, and protocols, almost made this impossible. Vinton Cerf and Robert Khan from DARPA presented, in a paper in 1974, a concept of common virtual addresses on top of the various physical-rooted addresses. They presented a new common protocol, *Transmission Control Protocol*, or *TCP*, on top of these. TCP connected the existing networks, and they called the concept *the internet*. The virtual addresses became known as Internet Protocol Addresses, or *IP* addresses. Later on, TCP and IP became separate layers, making it possible to run, for example, the simpler *UDP* on top of the IP network protocol. TCP and IP are treated in depth in Chapter 7.

Looking back, it is amazing how Ethernet has scaled from 10 Mbit/s to 10 Gbit/s, in affordable and still more robust solutions, *and* how TCP has been able to adapt to this. IP version 4 is growing out of addresses and IP version 6 is gradually taking over, as discussed in Chapter 7. The virtual addresses are hierarchical. This means that they are *routable*, and can change as we move. This is exactly like post addresses and has proven extremely useful.

In 1989, Tim Berners-Lee of CERN presented the *world wide web*. Many people think of *WWW* as being the same as the internet, but WWW is some 20+ years younger than the internet, while being simply one of many applications on it. The "hourglass" architecture shown in Figure 1.1 with the single IP in the center, millions of applications on top, and many physical network solutions at the bottom, has also proven itself to be extremely powerful. Software architecture is discussed in Chapter 4.

In the same period that the internet was developed, *Unix* was developed at *AT&T* using the new machine-independent language *C*. Unix and C were both created by Ken

https://doi.org/10.1515/9781547401024-001

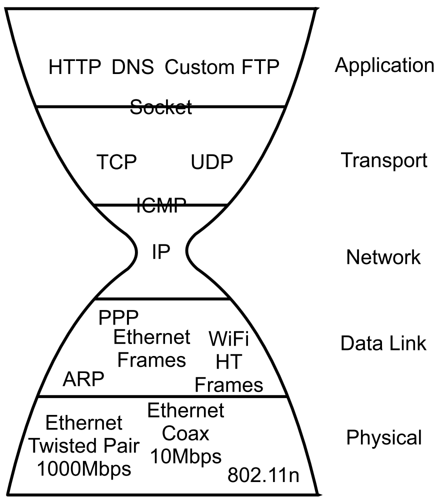


Figure 1.1: Internet stack – an hourglass.

Thompson and Denis Ritchie. Unix was also one of the first adopters of TCP/IP. In other words, many of the strengths *Linux* has today originates from the 1970s. Operating Systems are discussed in Chapter 2.

1.2 The cloud

While the early internet was mostly of academic interest,[1] the world wide web brought it to the people with its easy clickable *hyperlinks*. Web servers moved into companies and institutions, where they began to serve *fixed content*, text and figures, from file servers to all of us.

With the help of script languages such as *PHP* and *CGI*, the content became more dynamic, now often fetched from databases. This lasted for some years until we saw what was coined *Web 2.0*. Before Web 2.0, it was possible for a web server to update the content sent to the web browser, at the cost of a complete redraw of the page. Now web applications began to look like Windows or Mac applications, updating fields, graphs, etc. This was also the breakthrough of open source with the popular *LAMP*, which stands for Linux (the OS), *Apache* (the web server), *MySQL* (the database), and *Python* or *PHP* or *Perl* (all scripting languages).

As the complexity of web servers grew and the internet became faster and more robust, it became economically wise for many companies and institutions to use *web hotels*. The web hotels based their income on the scale: the constantly changing know-how required to run a single server is much better utilized running many. Now,

1 E-mail came before the WWW and was driving a slow growth of the internet.

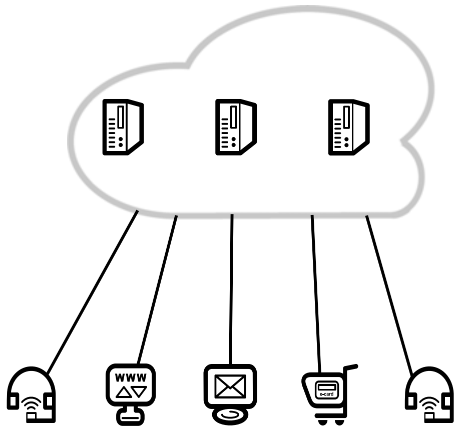


Figure 1.2: The cloud abstraction.

not even the employees of a given company knew where their company website was hosted, and they did not care. It was somewhere *in the cloud*. The cloud term became even more popular as we started to use smartphones, clearly connecting over the air—through the clouds. See Figure 1.2.

1.3 Internet of things

The first webcam was used at Cambridge University to show a coffee-machine, so that no one needed to go to an empty machine—a fun idea, but not the most practical. Instead you might consider building a web server into the coffee-machine, telling the user how many cups are left. The problem with this solution is that it is not very *scalable*. This is an important term that we will meet later. A web server in the small electronics of a cheap coffee-machine would typically be too limited to serve many web browsers. At this point in history, we actually did see a lot of *web-enabled* devices. It is practical that service technicians can connect to hardware using a standard web browser, and we still use this concept. At the time, however, the devices typically were not permanently connected to the internet, they just used the technology on rare occasions in a local connection, directly to the technician's PC.

The scalable solution to the coffee-machine problem would be a PC-based web server, reading many coffee machines, so that each machine only has a single client reading it, for example, twice a minute, and then many users could connect to the web server.

Let the coffee machine be any small internet-attached device, place the web server in the cloud, and let the users connect via a smartphone, PC, or tablet, and you have the *Internet of Things*, in short *IoT*. This is another application of the hourglass architecture, shown in Figure 1.3, only now the hourglass is not the implementation in

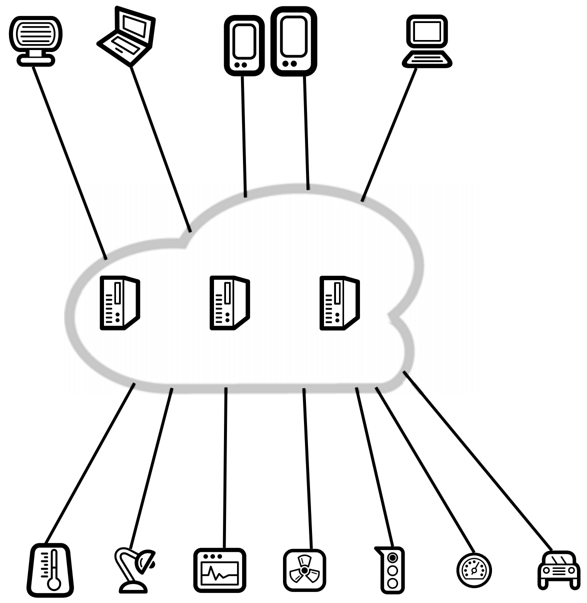


Figure 1.3: An unusual—but architectural relevant—view.

a single PC, but the entire internet. Rather unusual, the clients are placed *above* the cloud to demonstrate the architectural point. Another term sometimes used for all the devices “on the ground” is “the fog.”

The inventor of Linux, Linus Thorvalds, once stated in an interview that he created Linux for the desktop, and that it has overtaken the computerized world, except on the desktop. This refers to open source initially being extremely popular in the servers due to price, robustness, and ease of programming, while now it is moving into the small devices for the same reasons, plus a few more. Common to the (web)servers and devices is that they are only used directly by computer professionals. These people are not relying on crisp graphical interfaces, but prefer programs and scripts featuring automation.

Devices such as the *BeagleBone*, *Arduino,* and *Raspbery Pi* have opened the Linux-device world to new generations, but real-world devices typically need to be more robust. These are discussed in Chapter 3.

1.4 IoT related terms

– *Industry 4.0*
  Germany has been the most advanced European country in the automotive market for a long time. The main player, VW, was shaken in 2015 by *DieselGate*. Even before this, the Germans worried about *disruption*—the way new technologies may

turn a thriving business into bankruptcy in no time. They knew they could not afford to rest on their laurels. As one of ten *Future Projects* in a 2020 plan, the German government publicized *Industrie 4.0* in 2013.
The *fourth industrial revolution* has been claimed several times in history. This time it was an elaborate plan, envisioning the connectivity of IoT with its ability to bring real-time factory-floor data from, for example, Brazil, to production control in Germany. These data are typically in the form of statistics relating to specific assembly lines, product type, production gear, day/night shift, etc. Equally important, this plan focused on *decentralization*. This means more intelligent embedded systems at the assembly line, taking more autonomous and advanced decisions. These systems could possibly be calibrated remotely along the way, based on the statistics collected. Chapter 12 deals with *Statistical Process Control* (SPC) for embedded systems, and in Chapter 4 we are looking at *CANOpen*. This is one of several standards in factory automation.
Today, this advanced embedded software will often communicate with the product itself. It is no surprise that a car may be queried for its type, serial number, etc., and that calibrated production data may be written back to the car. However, via the car, the plant software can talk to its internal CAN buses, and via these to, for example, brakes, airbags, and other subsystems.

– *Industrial Internet of Things—IIoT*
  The *IIoT* term sounds very much like the Industry 4.0 concept, but it has a less clear birth certificate. The *Industrial Internet Consortium* has taken the term to heart. IIC was formed in 2014 by AT&T, Cisco, General Electric, IBM, and Intel, and since then companies from the entire world has joined. The vision is to assure the true interconnection of devices. The industries include manufacturing, energy, health care, transportation, and smart cities.
– *Internet of everything—IoE*
  Cisco is one of the members of the IIC. Cisco defines the *Internet of Everything* (IoE) as the “networked connection of people, process, data, and things.” In this way, it includes the internet of things, without forgetting the already existing internet of people. This makes sense, as Cisco’s market is the infrastructure, it does not really matter whether a person or a device is generating or subscribing to data. By including the word “process,” Cisco opens a door for expanding the current cloud with its web servers and databases with actual applications. This is something that also interests major players like Google, Microsoft, Facebook, and Amazon.
– *Big data*
  Big data is clearly not the same as IoT. However, all of these billions of internet-attached devices will create unprecedented amounts of data. Chapter 11 shows how some of these data may be filtered already at their birth in the devices. This will be essential in many scenarios. Processing power has grown at an impressive pace during the computer era, but the amounts of storage have grown even more. Big data may be fantastic, but if we want fast algorithms, they need to work on

less, but more relevant, data. We need to reduce data at the source and in the various process steps, to a certain limit. To quote Einstein: "Make things as simple as possible, but not simpler." The subject of treating big data in the cloud is not in the scope of this book. We will however see various ways to reduce data.

Should you want to browse through the amazing birth of the web and all the innovations that helped it along, take a look at *100 Ideas that Changed the Web* by Jim Boulton. This book spends only two pages per innovation, and one of these is typically a color image. This makes it entertaining, but still very informative.

Part I: **The basic system**

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, for example, decide to go for Linux, but this requires a MMU (memory management unit), and this may drive you toward too big and expensive CPUs. In that case, you may have to redo you original choice. Table 2.1 is a “kick-start” to this chapter. It shows an overall trend from simple to advanced.

Table 2.1: Task management—from simple to advanced.

OS/Kernel/Language	Type
Simple main	Strictly polling
Ruby	Co-routines
Modula-2	Co-routines
Windows 3	Nonpreemptive 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
VxWorks	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 toward a full *preemptive* real time operating system (RTOS).

2.1 No OS and strictly polling

The simplest embedded system has no operating system, leaving some low-level details to the programmer. If you are using “C” there is a `main()` function from which your “official” program starts at power-up. Since there is no OS, this must be assured by con-

https://doi.org/10.1515/9781547401024-002

figuring the compiler, linker, and locater.[1] It is necessary to initially call a small assembly program 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 the call 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.

In an OS-less system, `main()` has an infinite loop that could look like this:

Listing 2.1: Round-robin scheduling

```
int main(int argc, char *argv[])
{
   for(;;)
   {
      JobA();
      JobB();
      JobA();
      JobC();
   }
}
```

This is a "*round-robin*" scheme with the slight enhancement that JobA has received more "attention" (not really priority) by giving it access to the CPU with shorter intervals than the other processes. Within 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, is 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 doing something? In such a scenario, we 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 okay, but not in a system as simple as this.

Another concept, still without any operating system whatsoever, but a lot more clever, is to introduce finite state machines (FSMs) where you read all inputs, decide what has changed, and take action as shown in Listing 2.2.

1 The locater is typically integrated with the linker so do not worry if you have not heard about it before.

Listing 2.2: Main with finite state machine

```c
int main(int argc, char *argv[])
{
   for(;;)
   {
      ReadSensors();           // Read all inputs
      ExtractEvents();         // Is temp above limit?
      StateEventHandling();    // Take action
   }
}
```

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, via the FSMs. Later, we will look into sockets and TCP, and it can be seen that the listing very directly represents a good part of Figure 7.8, which is a graphic representation of the TCP connection states. For now, it is more relevant to look into the concept of a FSM.

Each column is a state of a TCP socket, that at a given time is the “current state.” Each row represents an event that occurs while in this state, for example, an ACK (acknowledge) has been received. Each element in the table contains the action to take, as well as 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 (it is coincidental that the number of states and events is the same). FSMs 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. FSMs are very popular among hardware designers, but not used by many software designers, which is a pity. Nevertheless, many modern frameworks contain FSMs inside, offering “event-driven” models.

Listing 2.3: One of three finite state machines for a TOE

```c
struct action connected_map[EV_MINOR(EV_ENDING_COUNT)]
                           [ST_MINOR(ST_ENDING_COUNT)] =
{
//st_normal      st_close_wait   st_last_ack     st_fin_wait_1  //EVENT
//NORM           CL_W            LACK            FW_1           // ev_end_
{{error,   NORM},{error,   CL_W},{error,  LACK},{error,  FW_1},// <error>
{{send_fin,FW_1},{send_fin,LACK},{no_act, LACK},{no_act, FW_1},// close
{{error,   NORM},{error,   CL_W},{req_own,OWN },{fw1_2,  FW_2},// ACK
{{ack_fin, CL_W},{error,   CL_W},{error,  LACK},{ack_fin,CL_G},// FIN
{{error,   NORM},{error,   CL_W},{error,  LACK},{ack_fin,TM_W},// FIN_ACK
{{error,   NORM},{error,   CL_W},{fin_to, CL  },{fin_to, CL  },// TimeOut
{{abort,   GHO },{abort,   GHO },{abort,  GHO },{abort,  GHO },// Exc_RST
};
// Table continuing here
//st_fin_wait_2  st_closing     st_time_wait                    //EVENT
//FW_2           CL_G           TM_W                            // ev_end_
{error,    FW_2},{error,  CL_G},{error,  TM_W}},                // <error>
{no_act,   FW_2},{no_act, CL_G},{no_act, TM_W}},                // close
{error,    FW_2},{cl_ack, TM_W},{error,  TM_W}},                // ACK
{ack_fin,  TM_W},{error,  CL_G},{error,  TM_W}},          // FIN
{error,    FW_2},{error,  CL_G},{error,  TM_W}},          // FIN_ACK
{req_own,  OWN },{req_own,OWN },{req_own,OWN }},          // TimeOut
{abort,    GHO },{abort,  GHO },{abort,  GHO }},          // Exc_RST
```

One nice thing about FSMs is that they are both implementation and documentation, and they typically can fit into a single page on the screen. Compared to a number of **`if-else`** or **`switch`** clauses, the use of FSMs is much "cleaner" and, therefore, much easier to create and keep error-free. With the good 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.

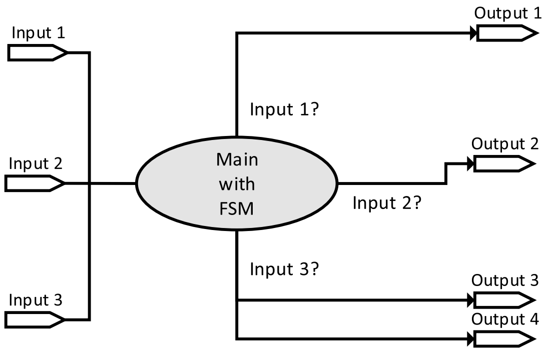


Figure 2.1: Finite state machine in main.

Figure 2.1 shows an OS-less system. The main advantage 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 practical 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 more times:

- Input 1 – leading to some processing and eventually to a change in output 1.
- Input 2 – leading to some processing and eventually to a change in output 2.
- Input 3 – leading 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, for example, an actor in a game or a cell in a bio-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, for example, 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, most CPUs have all interrupts disabled.[2]

In such a "purely interrupt-controlled system," the interrupt service routine may in principle do everything that is related to a given event. See Figure 2.2.

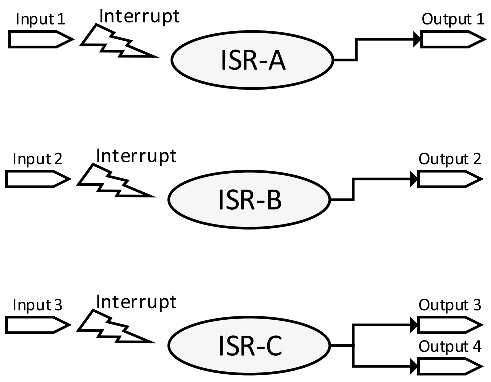


Figure 2.2: Purely interrupt controlled system.

There are many 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 program (stacking the registers it plans to use), and then this may become interrupted by the next, higher level

2 except for the NMI—nonmaskable interrupt—if such exists.

interrupt, etc. This is known as *nested* interrupts and typically only can happen if the original ISR has reenabled interrupts. This can be done by the OS, if such exists, before giving control to the programmer, or by the programmer himself in our current OS-less case. 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. This again depends on how far we got in the interrupted interrupt, which is almost impossible to know. This is one of the many reasons why you should defer as much of the action as possible, to something that runs later, at a lower priority. But then it's no longer a purely interrupt-based system. Furthermore, many systems do not have interrupt levels enough to match all the inputs.

2. As above, each input has its own interrupt service routine (ISR), and its own interrupt priority. In this system, however, nested interrupts are not allowed. This means that all other interrupts must wait until the first is done. This is really bad for the *interrupt latency*—the worst-case reaction time—on the other interrupts. Thus it normally becomes even more important to defer most work until later. This again is bad news for a purely interrupt-based system.
3. Both of the above scenarios may have the option for many inputs to trigger the same interrupt. The first thing the ISR must do then 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 "subpriority" so to speak.

From the above, it is clear that a purely interrupt-controlled system, with nothing deferred to low-priority handlers, has some huge challenges.

A particularly nasty problem with interrupts I once 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 CPUs.

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, for example, 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 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 does not offer any help to the developer in terms of multitasking (a concept we will see shortly).

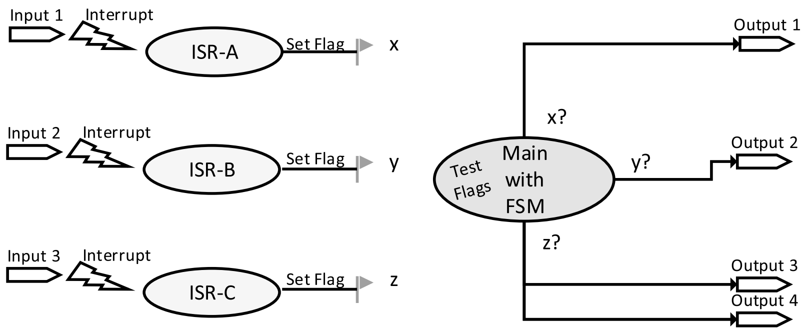


Figure 2.3: Interrupt system with finite state machine in main.

If, however, the main loop is broken up into small pieces of code, organized with the help of finite state machines, 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 ISRs are called events. ARM *mbed* prioritizes the interrupts in the usual way and also offers 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, the one with the highest priority is started first. Since all of these "pseudo threads" are started on a specific point and run to end, there is no *preemption*. One task in the application code never takes over the CPU from another; it is only interrupted by ISRs 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 example, for the small 32-bit Cortex M0 CPUs that have scarce resources (including no MMU). What makes *mbed* interesting is that it has a lot of the extras, normally seen on larger OS'es: TCP/IP stack, Bluetooth LE stack, etc. It also boasts a HAL (hardware abstraction layer), making the code the same for other CPUs in the ARM family. In this way, mbed is well positioned and does look very interesting.

Note that 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, nevertheless, not a good idea simply to assign a task to each developer, as this will require more coordination among tasks. The less coordination needed between tasks, the better. In fact, almost all the quirks that can make the use of a kernel complex are related to interaction between tasks. Figure 2.4 shows a system with a preemptive scheduler (we will get back to this shortly).

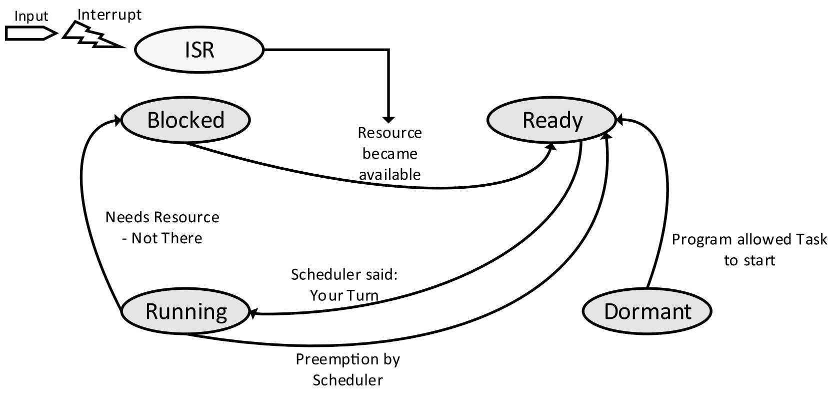

Input
Interrupt
ISR
Blocked
Ready
Resource became available
Needs Resource - Not There
Program allowed Task to start
Scheduler said: Your Turn
Running
Dormant
Preemption by Scheduler

Figure 2.4: OS with preemptive scheduling.

The states used in Figure 2.4 are:

– *Dormant*
  The task is not yet brought to life. This must be done explicitly by the application.
– *Ready*
  The task can run, only it's waiting for the current "running" task to "get off the CPU."
– *Running*
  Actually executing code. There can only be one running task per CPU—or rather per CPU *core*. Many modern CPU chips contain several cores. This is really like several CPUs in one house. Intel's hyperthreaded virtual cores also count here.
– *Blocked*
  The task is waiting for something to happen. This could, for example, be a socket in a `recv()` call, waiting for input data. When data arrives, the task becomes "Ready." A socket will also block if you write to it with `send()` and the assigned OS transmit buffer is full.

Most kernels today support *preemption*. This means that application code in tasks is not only interrupted by ISRs. 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.[3] 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) to make room for running the high-priority task.

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.

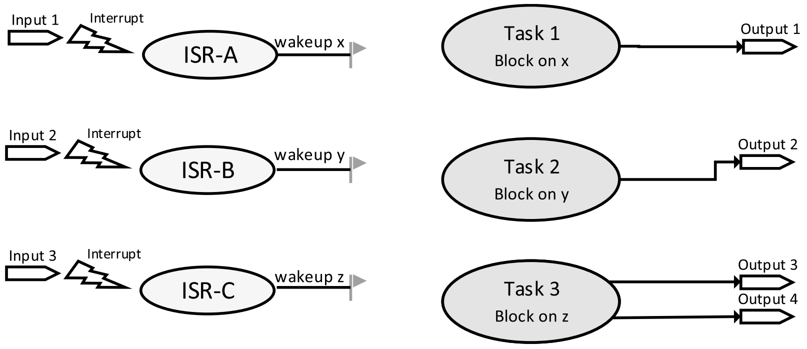


Figure 2.5: Tasks and interrupts.

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 ISRs 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, for example, 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, for example, CTRL-C from the keyboard. If the call returns nonzero, this is what has happened. In a normal embedded system, you are not so interested in the use of signals.

Tasks can 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. The queue is sometimes referred to as a mailbox.

3 A classic error is that the ISR programmer saves only the registers used. Then later adds code and forgets to save the extra register(s) used.

A highly recommended design is one where all tasks are waiting at their own specific queue, and are awakened when there is “mail.” A task may have other queues 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.” This is not much different from a manager, mainly driven by mails in his outlook in-box. One specific advantage in using messages is that some kernels seamlessly extend this to work between cores—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 supporting 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, and released in the opposite order. 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. Thus it is preferable to keep resource-sharing at a minimum. This is especially true in systems with multiple cores that actually do execute in parallel. Coordinating resources in this scenario is inefficient, and copying data may be preferred in many cases.

2.5 A nonpreemptive operating system

We typically talk about an *operating system* when there is a user interface as well as a scheduler. 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, released in May 1990. This was a huge and very complex operating system with a lot of new and exciting GUI stuff and tools. With Windows 3.1 (1992), 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 had a task concept, but contrary to a small RTOS kernel, it did not support preemption. Thus Windows 3 was like the system in Figure 2.4 without the preemption action. Another thing missing in Windows 3—which all good OSs have today—was 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. Even though 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. A task could not get to the CPU until the currently running task *yielded*. 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 the Windows 3 code, you would often see `sleep(0)`. This meant that the task *could* continue, but on the other hand was also prepared to leave the CPU at this point. Moreover, Windows 3 introduced a variant of Berkeley Sockets called *WinSock*. As we will see later, if you try to read data that hasn't arrived yet from a socket, your task will be blocked in a preemptive system.

In the days of Windows 3, this was standard in Unix, but Windows couldn't handle it—yet. Instead, 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 main reasons why Microsoft developed Windows NT—which in newer versions is known as Windows XP, Vista, 7, 8, or 10—"Windows" to all modern people. This is not just an anecdote. You may still see kernels for very small systems that are nonpreemptive, like ARM 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 we need to consider here:

– *Reentrant*
  A reentrant function can be used recursively; in other words, by the same thread. In order for this to happen, it must not use static data, instead it uses the stack. A classic example of a non-reentrant C-function is `strtok()`, which tokenizes a string very fast and efficiently, but keeps and modifies the original string while parsing it into tokens. It is called again and again until the original string is completely parsed. Should you want to start parsing another string, before the first is completely done, you will overwrite what's left of the first original string.
– *Thread-safe*
  A thread-safe function may be used from different threads of execution in parallel. This is accomplished by the use of OS locks or critical sections. If for instance, the function increments a 16-bit number in external memory, things can go wrong. To increment the number, we need to read it into the CPU, add one and write it back. If the CPU has a word size of 8-bits, it reads the low byte first, adds one to it, and writes it back. If this operation meant a *carry*, the next byte is incremented in the same way. Unfortunately, another thread, or interrupt service routine, might read the two bytes before the high-byte is incremented. Even with a 16-bit word size this can happen if data is not word aligned. In this case, the CPU needs to read two 16-bit words. You want to make the whole operation "atomic." This can be done by disabling interrupts for the duration of the complete *read-modify-write* operation, using relevant OS-macros. Many modern CPUs have specific assembler instructions for doing such atomic functions. C11 and C11++ compliant compil-

ers can use these functions—either explicitly like `std::atomic<>::fetch_add()`, or by declaring variables like `std::atomic<unsigned> counter`, and then simply write `counter++` in the code.
More complex scenarios, like, for example, engine control, may require several operations to be performed without something coming in between them. In this case, the OS-macros or manual interrupt disable/enable is needed.

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 as well as reentrant. Many kernels and operating systems supply two versions of their libraries, one for multitasking (thread-safe), and one not. The reason for having the latter at all, is that it is smaller and executes faster. It makes sense in nonpreemptive 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 nonblocking 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, creating an FSM within the OS—thus featuring 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, with these servicing the many real clients. On top of this, we often see a local Bluetooth or Wi-Fi control. 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 preemptive kernel gives you everything you need in order to multitask 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 commandline prompt, and typically a Graphical User Interface (GUI)—then we have a full OS.

Today Linux is the best known and used full OS in the embedded world. Windows also comes in versions targeting the embedded world, but somehow Microsoft never really had its heart in it. Windows CE is dying out. Very few hardware vendors are supporting it, thus there is not the wealth of drivers that we are accustomed to with desktop Windows. The development environment, "Platform Builder," can be very disappointing if you are used to Visual Studio. Microsoft marketed first Windows XP, later 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 and 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, known, 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.

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 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 remains 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, for example, be an FPGA collecting 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 SoCs containing two cores: one with a lot of horsepower, perfect for Linux, and another one that is small and well suited for handling interrupts. As the latter 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. The DS-MDK environment from ARM/Keil actually supports such a concept, for development hosted on Windows as well as Linux. In the simple example with an A/D converter, many CPUs are however capable of buffering data directly from an I^2S bus or similar.

Another problem with Linux is that it demands a memory management unit (MMU). This is, in fact, a very nice component in the larger CPUs that cooperate with the OS. 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. Processes are very relevant on a PC, and practical 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. Again, this may hurt software compatibility.

There is a lesson to learn from Ethernet. Ethernet is not perfect and it cannot guarantee a deterministic delay like Firewire can. Still Firewire has lost the battle, while Ethernet has survived since 1983 (i. e., at various speeds and topologies). The cheap "good-enough" solution—in this case the nonreal-time Linux—wins over the expensive perfect solution, *if* it can solve problems outside a small community.

2.7 Open source, GNU licensing, and Linux

It is a well-known fact that Linux brought the ways of Unix to the PC-world, mainly servers, and now is taking over the embedded world. Interestingly, Unix clones are old news to the embedded world. Many years before Linux was born, RTOSs such as OS-9, SMX, QNX, and many others, used the Unix style. It was even standardized as "POSIX." The idea was to make it feasible to switch from one to another. So why is Linux so successful? One explanation is the massive inertia it got via PC-hardware in embedded systems. Another explanation is that it is open source.

Kernels and OSs come in two major flavors: open or closed source. 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 some reasons why this is so important:

1. If you are debugging a problem, you may eventually end in the kernel/OS. If you have the source, you may be able to find out what is wrong. Often times you may be able to make a work-around in your own code. This would be pure guesswork using closed source.
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 official code base, so that the next update contains your fix. There is nothing more frustrating than finding a bug, and then realize that you have found it before. Also, the GNU Public License, GPL, requires you to make the improvement public, so getting it back into the official kernel makes life easier, which is the whole point.
3. As stated earlier, a lot of embedded codes lives for many years. It's a pain if the OS doesn't, and if it is open source you can actually maintain it, even if no one else does.

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 or 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 which includes a lot of open source code besides your application. This is okay, as long as the parts originating 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.” See Section 6.9 on 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. GNU is very focused on not being “fenced in” by commercial interests. The basic GNU license allows you to use all the open source programs. However, you cannot merge them into your source code, and not even link to them without being affected by the “copy-left” rule which means that your program source must then also be public.

Many sites claim that you are allowed to dynamically link without being affected by the copy-left clause. However, a FAQ on gnu.org raises and answers the question: *Does the GPL have different requirements for statically versus 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 “work-around” 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. The Linux philosophy states that programs should be “lean and mean” or in other words: do only one thing, but do it good. This, together with the fact that most programs use files, or rather *stdin* and *stdout*, allows you to really benefit from the GPL programs this way. This is very different from Windows where commandline programs are rarely used from within applications; see Section 4.4.

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” (LGPL) comes into the picture. The founders of GNU realized it would inhibit the spread of the open source concept if this was not possible. All the system libraries are under this license, allowing linking in any form. However, if you *statically* link, you must distribute your object file (not the source), thus enabling other people to update when newer libraries become available. This makes dynamic linking the preferred choice.

The GNU org are 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 separate—never start a proprietary code module based on open source. There are alternatives to the GPL, such as FreeBSD and MIT licenses, aiming to make it easier to make a living on products based on their code. Such libraries may also be used from the 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, dynamically loaded into the kernel. 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 ignored, not accepted by the GNU community.

2.8 OS constructs

Table 2.2 presents a short list and explanation of some OS constructs.

Table 2.2: OS primitives for scheduling.

Concept	Basic Usage
atomic	A Linux macro assuring atomicity on variables, not normally atomic, for example, a variable in external memory.
critical section	Generally a block of 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 which other threads/processes may wait on—blocking or not—in, for example, `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 releasing the resource, 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 -9`. A process can block until it is ready, 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, 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 4th.
- 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 3rd.
- 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 Parts 1 and 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 were, and still are, extremely helpful for a Windows developer. These guys knew more about Windows than Microsoft did, until they "merged" with Microsoft.
- 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, ISRs, etc. It includes a description of some specific low-level HW circuits. The book uses 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.
- Michael Kerrisk: *The Linux Programming Interface*
  This is a huge reference book—not something you read from cover to cover. Nevertheless, once you start reading, you may end up having read more than planned—simply because it is well written and filled with good examples.

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. This 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, as well as 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 executing code. Everything else was outside this in other chips. Then came the “microcontroller.” This was specifically made for smaller embedded systems, containing a few built-in peripherals such as timers, a little on-board RAM, interrupt-controller, and sometimes EPROM to store a program. As integration increased, we ended with the modern *SoC* (system-on-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 CPUs) and the term “CPU” can mean anything from the core to the full SoC chip.

An example of such a 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.

AM335x is particularly interesting as it is used in the BeagleBone Black Board. This is a hobbyist/prototyping board resembling the Raspberry Pi. The BeagleBone is referred to, in this book, in several contexts.[1] Inside the AM335x is an ARM Cortex-A8, and a *lot* of integrated peripherals. The core runs the actual program, dictating the instruction set, while the other components decide the overall functionality and interfaces. This chapter is dedicated to the full integrated chip concept which is referred to as the CPU. The term “core” is used for the part executing the code.

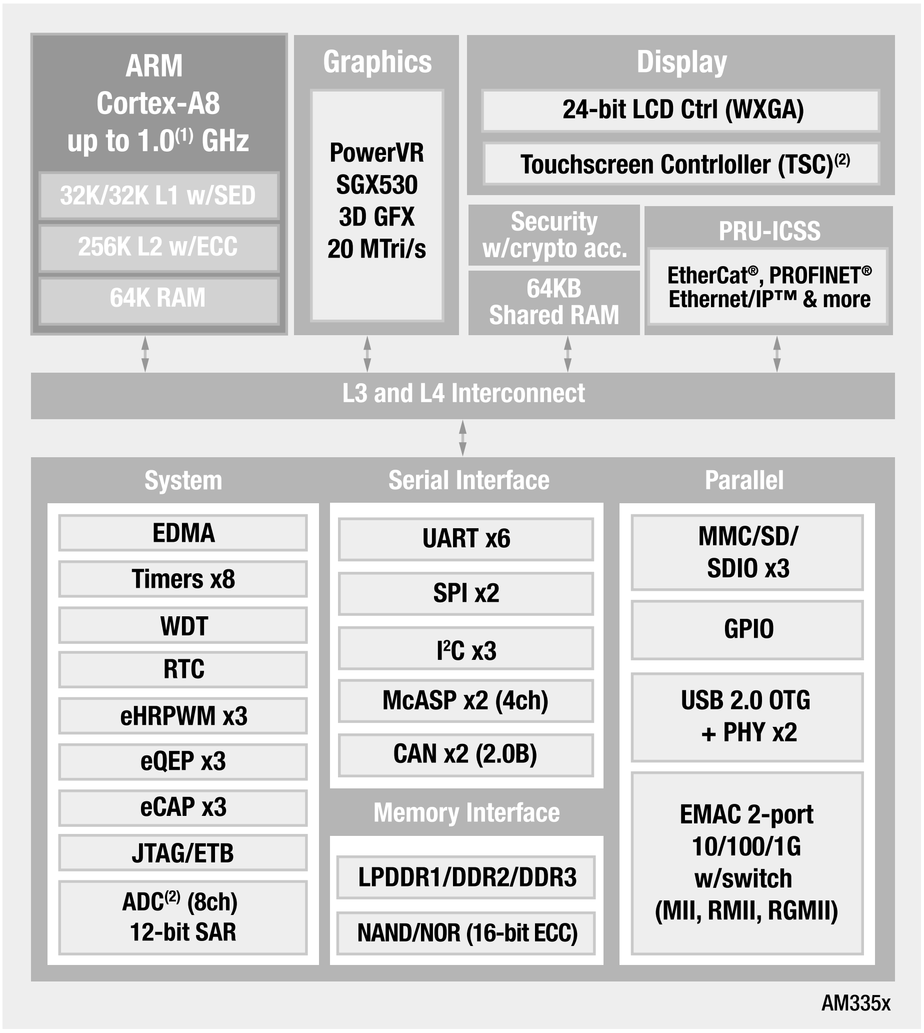
Modern computerized devices are thus created like “Russian dolls”[2] or “Chinese boxes”; see Figure 3.2.

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 alternatives to ARM. Table 3.1 gives an overview of the major concepts discussed in this chapter.

In the following, we will look a little closer into each of the subjects in Table 3.1.

1 The BeagleBone is also a reference hardware solution for the Yocto project; see Section 6.9.
2 The correct name is Matryoshka dolls.

https://doi.org/10.1515/9781547401024-003



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: AM335x—courtesy of Texas Instruments.

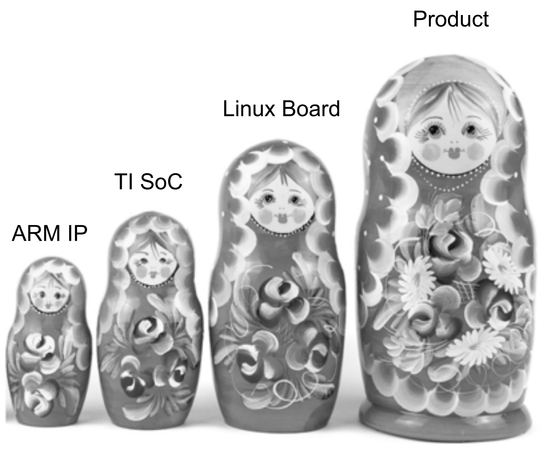


Figure 3.2: System design is like Russian dolls.

Table 3.1: Overview of concepts related to CPU-choice.

Concept	Purpose (short)
CPU core	Basic programmable unit
MMU support	Needed for high-end OSs
DSP support	Signal analysis
Power consumption	Battery powered, heat generation
Peripheral units	A/D, UART, MAC, USB, Wi-Fi…
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
RoHS	Compliance (typically) needed
JTAG debugger support	HW debug without ICE
OSs supporting	See Chapter 2
Upgrade path	If you grow out of current solution
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.

3.2 CPU core

Though not everything is about ARM, it is very popular in modern embedded designs. ARM has created three different “Cortex” (for “Core Technology”) “profiles”:

- *A* – for application
  These are the biggest, highest performing, and most integrated designs with "bells and whistles." These CPUs can do hardware control, but are better suited for number-crunching, including DSP and NEON media processing engines; see Figure 3.1. They integrate a lot of the peripherals as we shall see in this chapter. They are very well suited for running Linux or other high-end OSs. The Cortex-50 series even 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. They are a good choice for network routers, media players, and similar devices that do not require the performance of the Cortex-As, but still need data here and now.
- *M* – for microcontroller
  These are classic microcontrollers for handling external hardware. They are offered as soft cores for FPGA, but are also sold in numerous versions integrated with memory and peripherals. They cannot run standard Linux due to the missing MMU.

It's probably not a coincidence that the three types above spell "ARM." Still, they do reflect some major segments within the embedded world, which other microcontrollers and SoCs 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 ATMs ("R" profile) to advanced image processing ("A" profile).

As discussed before: Linux is not really well suited for real-time, 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, for example, both A- and M-profile CPUs in the same chip. This may allow us 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 Section 2.1 and Section 2.3. The integration thus continues, now toward multiple cores and/or DSPs.

3.3 CPU architecture

The best known CPU-architecture is "von Neumann." Here, program and data are accessed by the same databus and addressed via the same addressbus. The major advantage 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. In a very security-aware project, it may be preferable NOT to be able to change the instructions that the device executes.

The alternative to von Neumann is "Harvard." Here, data and program have separate buses. This is particularly popular with DSPs. A digital signal processor executes a lot of "multiply-accumulate" instructions, and typically (as we will see in Chapter 11) the multiply operands are one of each:

- A constant—from the program.
- A data value, for example, 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, for example, Cortex M0. The v7 family is Harvard and is used in the "application profile" CPUs that mostly also have 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 microprocessor architecture is whether it is CISC or RISC based. CISC (complex instruction set computing) is "the classic way." Here, a single assembly instruction may perform a complex operation, using many cycles. If you want to program in assembly-language, CISC has many instructions that resemble their C-equivalent. With RISC (reduced instruction set computing), each assembly instruction is simpler, executing much faster. However, the same complex operation demands more instructions, making assembly coding harder while also taking up more space for code, and using a little more performance on instruction decoding. With the instructions being simpler, it is possible for a clever programmer or compiler to do only what is needed in the current context.

Little code is written in assembly language today. C and higher-level languages are totally dominating, while CPUs are getting faster, memory cheaper, and compilers better. Thus the important thing is "how fast does my C program run?" Typically, RISC wins today. The main reason we still have a lot of CISC processors around, is the fact that Intel 80x86 CPUs (or AMD lookalikes) 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."[3] The execution of an instruction has several stages, for example, fetch the instruction, decode it, fetch the operands, execute the instruction, and store the result. To utilize the buses efficiently, the CPU may have several pipeline stages, handling 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.

CPUs come as either big-endian or little-endian; see Figure 3.3.

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 3.3):

3 A similar term is used in relation to TCP sockets.

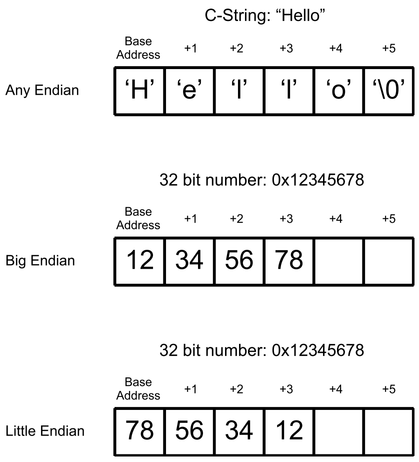


Figure 3.3: Endian-ness.

1. *Big-Endian*
   0x12 is placed at address 0x100, then 0x34 at address 0x101, etc. An argument for this is that when data is viewed in a debugger as bytes, they are placed in the same order as when viewed as 32-bit words. Thus it is relatively easy for a human to get an overview of memory content. 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 the most significant byte is at the highest address. Intel was on this side with most of their CPUs—including 80x86—although 8051 actually is big-endian.

Big-endian is also defined as "network byte order," meaning that it should be used for network data interchanged between platforms. We will see this 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. When sending application data between two PCs, which are both little-endian, there is quite a performance overhead in changing to network order and back again. At *least* the "endian-ness" of a protocol must be documented. Many modern CPUs can be configured to one or the other.

Similar to byte ordering, bit ordering could be a problem, but we are normally spared. All Ethernet PHYs transmit the LSB (least significant bit) first. Other serial buses, like CAN, send the MSB (most significant bit) first, but since the PHYs are standardized and bought for the purpose, we do not really have to care, unless we are debugging at the wire with an oscilloscope.

3.4 Word size

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

64-bit CPUs have not really taken off in the embedded world. The question is whether to use 8, 16, or 32 bits. Some may be surprised that 8 bits are still relevant. The site “embedded.com” had some interesting discussions on this. On January 6, 2014, Bernard Cole 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 CPUs.

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. Hardware vendors are still upgrading these designs with more built-in peripherals while 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 in, and efficient with, this architecture.

Consequently, if a design is extremely price sensitive, the 8-bit CPUs may be interesting. If, on the other hand, a highly integrated and high-performing modern System-on-Chip (SoC) solution is targeted, the 32-bit CPUs are definitely the place to look. It is a fact, however, that only recently have the 32-bit CPUs overtaken the 8-bit CPUs as the ones most sold.

That leaves the 16-bit CPUs in a kind of limbo and looking at Internet discussions, it certainly does seem like 16-bit designs are not really worth discussing.

I am not sure that 16-bit CPUs can be written off. Some Vendors such as Texas Instruments, Frescale/NXP and Microchip Technology supply to many designs, whereas we don’t hear much about designs from Renesas (the result of a merge between Hitachi and Mitsubishi). Nevertheless, this is one of the major vendors—offering many 16-bit based CPUs. 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, isolating the OS processes completely from each other. With this, it is impossible for one process to overwrite data in another process. In fact, it cannot even read it. This also has some overhead. If, for example, a process asks for a socket buffer to put data into, this buffer must be cleared by the OS 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, a MMU is our friend in catching bugs. A common, and sometimes very nasty problem, is *memory overwrite*. This comes in many flavors; some simply can write anywhere in memory, typically due to a "wild" or noninitialized C pointer. If this happens in a system with a MMU, it typically triggers an "exception." If this is caught, it may be easy to find the culprit.

3.6 RAM

No real program can run without RAM. Naturally, if one can fit the RAM inside the CPU, nothing is needed outside it. 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," constantly "visiting" 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 RAM 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, etc. DDR means "Double Data Rate," specifying that data is fetched on both the up- and down-going flank of the clock.

Microcontrollers like Intel 8051 and the newer Atmel Atmega128A, have internal SRAM. In some scenarios, this may be all that is needed. In other scenarios, we may, for example, in the Atmel case, utilize the fact that various parts of the chip may be put into sleep mode while the internal SRAM still works. The system stack and interrupt vectors are kept here. It may even be possible to turn external RAM completely off in sleep mode, but this requires the design to be able to restart almost from scratch.

3.7 Cache

Over the last many years, CPU performance has improved 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 intermediate memory, inside the CPU, which is a "shadow" of the latest blocks of data fetched from the external memory. In a Von Neumann system (see Section 3.3), a single cache is needed. In a Harvard system, a "split cache" is needed if both program and data is cached.

If, in a system without cache, code is executing a fast "inner loop," it is spending many clock cycles waiting for the same parts of the program to be fetched from external memory—with wait-states—again and again. If instead we have a cache big enough to hold all the code in the inner loop, the code executes much faster. Cache may come in several layers, and can be complex to set up. A common problem is that most modern CPUs are using "memory-mapped I/O." When we read from this, we 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-cacheable on a low level. It would seem that the C keyword "*volatile*" will help us, but it won't.

Many compilers offer the option to prioritize either speed or size, in the sense that the compiler either maximizes the speed of the program or minimizes 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 a program takes up, the bigger is the chance that important loops fit into the cache. This boosts speed much more than what any speed-optimizing compiler can do. Not all modern CPUs have cache. This is a feature typically only included in the larger CPUs, for example, 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 memory, it can store its own program here. Unlike the RAM case, we normally don't see CPUs that have both internal *and* external flash. It's either one or the other. EEPROM (Electrically 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 is a challenge as the EEPROM only allows a limited amount of writes.

3.9 FPU-floating point unit

There are many degrees of math assistance. For example, 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 have 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), respectively known as *float* and *double* in C.

When comparing FPUs, it is important to note how many cycles they need, in order to do the different kinds of multiplications and divisions. An accurate comparison is quite difficult as the various CPU structures don't fetch their variables in the same way or at the same speed. This is where the number of MFLOPS (million floating point operations per second) becomes interesting.

3.10 DSP

When doing digital signal processing, the number of possible MACs (multiply and accumulate) per second is a vital parameter. This must be measured in the relevant accuracy (integer, single, or double). As we shall see in Chapter 11, a multiply-accumulate with an output shifter is very important for filters implemented with integer arithmetic. If an FFT is implemented, so-called *bit-reversed addressing* can save a lot of cycles. Some of the high-performance CPUs include *SIMD* extensions. SIMD is single instruction on 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 has 32, 64-bit wide registers that can be seen as an array. This is particularly relevant in image processing. Note, however, that compilers do not support this directly.

3.11 Crypto engine

In Chapter 10, we dive into the huge subject of *security*. In order to, for example, participate in https, the system must support SSL (secure sockets layer) or TLS (transport layer security). It requires a lot of math to support encryption and decryption, etc. All of this can be done in software, but it is faster and, interestingly, also safer to do it in hardware.

As an example, the crypto engine in the TI AM335x boosts SSL performance a factor of two (blocksize 1 kB) to five (blocksize 8 kB). This engine can also work together with Linux DM-Crypt to encrypt a hard disk. In this case, it boosts performance a factor two to three. The crypto-engine in TI AM335x only supports a subset of the TPM specification,[4] and, for example, asymmetric key algorithms[5] are run in software.

3.12 Upgrade path

Most CPU cores are part of a family with 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, building 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 large differences between the various vendors versions. An upgrade path is very important as systems tend to grow. If the bigger and 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, while introducing better sleep modes.

4 TPM is discussed in Section 10.17.

5 Asymmetric key encryption is explained in Section 10.7.

3.13 Second sources

It is comforting to know that if the manufacturer of CPUs decides to stop production, or raise prices, there is at least one other manufacturer. This is not always possible. At least we need a process where we 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 the chance of second sourcing.

3.14 Price

Getting a good price sounds simple, but is in reality 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 the cheapest after your design is complete. However, when it comes to the larger semiconductor chips as microcontrollers, microprocessors, FPGAs, 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 wait until you are ready to go into production to start negotiating prices. The vendor already knows that you have based a product on this chip, and the basis for negotiation is lousy. It is more optimal to negotiate the prices up front. This is a very annoying way to work for many designers. They typically want to pick and choose themselves during the project. It is also difficult for professional purchasers, as they typically don’t have the overview that the designers have. This limits how much we really can discuss with our dealer before making choices. Obviously, all this is mostly relevant when producing in relatively large quanta. With small series, the development costs are typically much higher than production costs. This means that most choices that shorten the development phase, bring down time to market as well as overall costs.

3.15 Export control

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

If you are a non-US company, buying expensive chips, such as some CPUs and FPGAs, it may be advantageous to buy some non-US parts to stay on the safe side in De Minimis calculations. This is valid for licensed software as well.

Other rules are more technical, for example, stating that if a system is capable of sampling more than 100 Msamples/second for a given system (accumulating all channels) the end-user must be registered. There are also rules about encryption levels; see Section 10.20.

Finally, there are rules about *dual use*, which is technology that can be used in both military and nonmilitary 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 rely on my word but enlist someone that is competent on these matters.

3.16 RoHS-compliance

This topic is really not about embedded software. It's about electronics, but even more about chemistry. It relates to all components in an electronic design, not just the CPU, but everything—including the mechanics used. RoHS means "Restrictions of Hazardous Substances" and is an EU directive aimed at protecting people and the environment from Lead, Mercury, "Hexavalent" Chromium as well as a number of organic compounds. This protection is aimed mainly at the waste phase, but a hot electronic product will also emit contained organic softeners. Although RoHS originates in EU, it is adopted on other markets, including China, so even though it does not apply in the US, it actually does for US companies aiming for export. To have a RoHS compliant product, a company must be able to document that all parts going into an electronic product are RoHS compliant.

In other words, this is very much a documentation effort, forcing sub-vendors to do the same. Today RoHS is a part of the CE marking directive. RoHS has gradually been phased in, over a couple of years and is now fully implemented in the EU. At first, it could be very difficult to find RoHS compliant products. Especially the soldering processes, used many places, needed to be hotter now that Lead could not be used. This again was damaging for many components. Today RoHS-compliant components are the default, and it is becoming easier to "get on the train." Any new design should be RoHS compliant.

3.17 Evaluation boards

Evaluation boards are also known as EVMs, evaluation modules. 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 of a good estimate on the actual performance is to implement it, or rather the "fast path." The major vendors typically sell very cheap EVMs. This means that it is indeed doable to check some essential code on 2–3 platforms, if you know from start what is "essential." However, EVMs 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 EVMs.

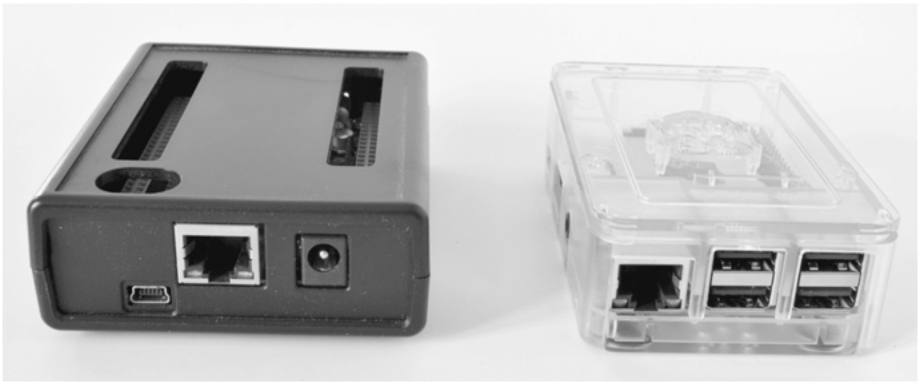


Figure 3.4: BeagleBone (left) and Raspberry Pi.

Cheap hobby devices such as Arduino, Raspberry Pi, and BeagleBone may serve the same purpose; see Chapter 5. The functionality and form factor of the BeagleBone and the Raspberry Pi are almost identical; see Figure 3.4.

The BeagleBone is slightly more practical than the Raspberry for prototyping, as it has two rows of connectors onto which it is easy to fixate a *cape*. You can buy all sorts of capes equipped with display, relays, batteries, motor-control, and much more.

3.18 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 workflow. If, for example, the team is used to working with integrated development environments (IDEs) and is forced to use the "vi" editor with a command-line debugger, one can expect some very unhappy faces. Typically, these tools are related to the chosen OS, as well as the CPU. The OS may demand specific file formats and specific setup of MMU, etc. The CPU may, on the other hand, dictate the location of interrupt vectors, special function registers, handling of internal versus external RAM, etc.

3.19 Benchmarking

It is clear that CPUs are implemented in many different ways. How do we compare their performance in, for example, number-crunching or network ability? Using evaluation boards is great, but this is only practical when we have homed in on a few options. Before getting 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). This 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 benchmark tests on full systems, but also on CPUs. It is interesting because a lot of these data are available without membership. The general score on CPUs is called *CoreMark*, whereas a benchmark on networking is called *NetMark*.

Another often used benchmark is *DMIPS* (Dhrystone MIPS) which is a test that runs a specific program performing a standard mix of classic operations, not including floating point. These tests are sometimes updated in order to "fool" the clever compilers that may optimize main parts away. DMIPS are practical when comparing different CPU-architectures.

Many benchmarks are given per MHz as the CPUs often come in many speed variants. Be aware that a system with a 1200 MHz CPU typically is not twice as fast as the same system running 600 MHz. The faster version may be waiting on RAM most of the time. A multicore solution running slow may thus be a better option than a fast single-core, exactly as we see in the PC world. As stated earlier, this is only possible if the program has parallel tasks, and these are not waiting on resources shared between them.

3.20 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 CPUs 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.

Again, the best way to compare is to buy evaluation modules of the most likely candidates and run benchmark tests resembling the target application. It is easy to measure the power consumption of an EVM-board by inserting a Watt-Meter in the supply line. Alternatively, a small serial resistance can be inserted in the supply line, and the voltage drop over this, divided by the resistance, gives us the current. By multiplying with the supply voltage, we have a good estimate of the wattage. The EVM may have a lot of irrelevant hardware. We do not want this to be a part of our calculations. It is therefore recommended to document the measurements in, for example, excel and calculate the deltas between running with and without this or that feature or algorithm. As stated, modern CPUs can speed up and down dynamically. Sometimes this is fine, at other times, we prefer to set the "policy" that fixates the CPU speed and voltage in the tests.

3.21 JTAG debugger

Many integrated circuits have a JTAG interface for testing the IC mounted in the *PCB* (printed circuit board) 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 CPUs. Some CPUs also have trace facilities.

Please note that an open JTAG in an embedded system is also a security risk; see Section 10.19.

3.22 Peripherals

Surely, if you need peripherals which are not included in your SoC, 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 any language extensions supporting your needs? As discussed earlier, you can find modern designs with an 8051 core on steroids, and 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 few teams have, and you would not be able to leverage older code without a number of "global substitutes" (followed by a lot of testing). Some peripherals even require support from the OS.

Building-blocks like DSPs and FPUs are listed in earlier sections. One 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" is, for example, 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, for example, allow nesting of interrupts with higher priority.
- *DMA – direct memory access*
  Direct memory access is a way to help the CPU move large sets of data more effectively. It may, for example, be able to move data from external memory to a disk 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, we 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 higher latency, there is power to save.
– *Switch*
  The classic 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 "spokes." The main drawback is the extra cabling. There are times where network-devices are laid out on a line, bandwidth requirements are small, and the coax solution would be nice to have. In such scenarios, the problem can be fixed 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 SoCs. The traffic load grows as we get closer to the main consumer/supplier of data. Thus it is important to specify a maximum number of devices, guaranteed to work in the application.
– *A/D converters*
  The best analog/digital converters are not built into microcontrollers, but sometimes we don't need the best. If you can live with 10–12 bits resolution, a relatively low sample rate and 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, needed for an RS-232 connection. RS-232 was the preferred physical connection for smaller devices for many years. Today UARTs are still used in development. There may be no external RS-232 connector on "the box," but many PCBs have a small connector for a serial connection used for logging, etc. during debugging. A good example is the BeagleBone, where you can order the cable as an accessory. Another area for UARTs are *IrDa* (infrared) connections and similar.

  Please note that an open UART in an embedded system is also a security risk; see Section 10.19.

  Unfortunately, few PCs today have an RS-232 port allowing for direct connection to the debug port. Help is provided in the form of small cheap USB/RS-232 converters.
– *USB Controller*
  USB became the preferred connection to PCs after RS-232. With the experience from RS-232, the inventors of USB did not stop at the physical layer. They also introduced a number of standard *Device Classes*, making simple usage "plug'n play."

It is possible to "tunnel" TCP/IP through USB, something which many smaller devices support. In the USB world there is a master (e. g., a PC) and one or more slaves. But advanced devices may be slave to a PC one day, and master to a set of headphones the next day. This is solved with USB *OTG* (on-the-go) allowing both roles—albeit not at the same time. The new USB-C standard is very promising. It will replace a lot of existing short cables.

– *CAN – controller area network*
  This is a small, cheap and very robust bus 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 4.5.
– *Wi-Fi*
  Some chips have the wireless equivalent of the MAC built in, but the antenna is naturally outside. Sometimes the antenna may be laid out directly on the PCB. Please see Chapter 9.
– *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 Wi-Fi devices and transfer of limited information at low rates; see Section 9.10. Version 5 of Bluetooth arrived at the end of 2016. It aims to blur the differences between Bluetooth Classic and Bluetooth Low Energy; see Section 9.10.
– *Buses*
  SPI (serial peripheral interface) and I^2S (inter-IC sound) are standard serial buses for interfacing to external on-board devices. Many CPUs have intelligent buffering of data on these buses, saving the core from numerous interrupts. Even more basic is GPIO (general purpose I/O) which is single pins (optionally configurable as groups of bytes) that may be controlled from software.
– *PRU – programmable realtime unit*
  Texas Instruments is using PRUs in some of their ARM derivatives, including the AM335x family. PRUs have a very limited instruction set, and can in no way compete with DSPs. 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 OSs like Linux are not good at handling low-latency real-time requirements. Programming the PRU requires a *PRU-assembler*—an additional build-step.
– *McAsp – Multichannel 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, able to keep and maintain calendar time when power is off. In the IoT case, the device may be connected more or less always, and can use NTP (Network Timing Protocol) to maintain calendar time. However, knowing correct time from boot will

help you to get better logs. In addition, there may be scenarios where the 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 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. Once started, 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 something "bad" has happened, for example, an infinite loop with disabled interrupts[6] 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 CPU. Similarly, flash controllers are built into many newer CPUs.
- *Display controller*
  LCD, touchscreen and other technologies typically require special control.
- *HDMI controller*
  HDMI is a protocol used by modern TVs for high quality video and audio. Video is an important part of most of ARM's A-profile CPUs.
- *Graphics Engine*
  This is an accelerator for OpenGL graphics. The popular QT GUI framework uses this, but also works without. Many graphics packages require OpenGL.
- *LCD Controller*
  This is a more low-level controller for graphics and text on dedicated LCD displays.
- *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 the position of a device, even if it is fixed on a location, 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. Be aware that professional systems providing these parameters, often combine the GPS/GNSS data with accelerometer data, to get better results.

6 Note that you can have an infinite loop that is neatly interrupted by something that kicks the watchdog, and then the system is not restarted. For this reason, you may consider a software version as well, requiring tasks to "report on duty" at intervals.

Most people do not realize that the accurate positions are the result of accurate timing. You can get the time with a precision better than 50 ns. In external devices, this typically comes in the form of a string on a serial bus, matched with a timing signal, typically 1 *PPS* (pulse-per-second). Incidentally, GPS needs to take both Einstein's 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 you will have to choose between "set A" and "set B" in a lot of sets. It is a good idea to ask the vendor for an overview of this.

3.23 Make or buy

The previous sections are all assuming 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 sell in large quanta, this is the way to go. If, on the other hand, your primary business is software, for example, mobile apps and cloud, then you may want a shorter route to the hardware controlling whatever sensors or actuators you are using. This discussion leads back to the Russian dolls in Figure 3.2. Not many companies are making CPU *intellectual property* like ARM, or the SoCs like Texas Instruments. A great deal are however making their own Linux boards while numerous other companies buy these "as is" from one of the many vendors of OEM products.

It is easy to get started with a Raspberry Pi, a BeagleBone, or an Arduino board. These are all fantastic: 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, for example, power to a BeagleBone is lost in a "bad moment" the device is "bricked." That is simply not good enough for real life products.

Similarly, you may be making the hardware and firmware, but what about a cloud solution? Here are a few industrial alternatives and teasers:

– *Libelium – devices*
  Libelium is a modern Spanish company selling hardware to system integrators. The embedded IoT devices are called "WaspMote." They connect to sensors and wirelessly, using ZigBee, connect to "Meshlium" routers that connect to the Internet and the cloud. Examples are numerous "smart" projects—parking, traffic, irrigation, radiation, etc.
– *Arduino Industrial 101 – low-level platform*
  This is not just an industrial temperature range and general robust version of an Arduino. The basic Arduino is great for I/O, but doesn't 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 Wi-Fi router implementations; see Section 6.10.

– *Hilscher netIOT – bridging the factory*
  Hilscher is an old German company, and is an interesting example of how IoT in many ways unite well-proven existing technologies with modern cloud technology. Hilscher has its core competence in creating ASICs for fieldbuses. This is, for example, EtherCAT and CANOpen (see Section 4.6), used for factory floor automation. Many solutions programmed "from the bottom" today could be done faster and cheaper with the help of, for example, CANOpen.
  Hilscher's product series "netIOT" allows integrators to connect their factory automation to the cloud via, for example, the MQTT protocol, suitable for smaller systems. You can thus design and implement systems without making any hardware. An example could be a car factory in Brazil, monitored from Germany. This is the core of "Industry 4.0."
– *Kvaser Ethercan Ligth HS – bridging the factory*
  Kvaser is a Swedish company with a history going back more than 60 years. It is one of the best known suppliers of CAN equipment. Kvaser has a product called "Ethercan Ligth HS." This is a bridge between standard CAN (thus including—but not limited to—CANOpen) and Ethernet. This is yet an example of including an existing range of products in the "Industry 4.0." This product can use *Power-over-Ethernet* (PoE) as the supply for the CAN, making it simpler to connect.
– *Z-Wave – intelligent home platform*
  Z-Wave is a full eco-system of systems for intelligent home control. It is backed up by a number of companies, and offers an infrastructure with a cloud server, phone apps, etc. An end-user can buy starter kits from one of the many vendors and easily make an existing house more or less remote-controlled. The starter kit includes a gateway which connects to the home router for connection to the Internet. From the gateway to the devices, we have a mesh network. This allows remote devices to jump through the ones nearer to the gateway. Playful users can toy around with a BeagleBone with a USB gateway for Z-Wave that may contain their local version of the webpage control.
  As a developer, you can buy an SDK with a 8051 derivate, Keil compiler, etc. You would need to build your own hardware with this inside in the longer run, but the eco-system is well established. The system is using the ISM band (868 Mzh in Europe, 908 MHz in the US+Canada, and similar in the rest of the world). The throughput is below 40 kbit/s, and if you can live with a lot less, the power-save functionality allows a battery to last for a year or more.
– *ThingWorx/KepWare – cloud interface and analytics*
  ThingWorx is a cloudbased data collection, user interface and analysis system, used for example, for factory control. In 2015, PTC—owner of ThingWorx—acquired KepWare, which essentially is a collection of protocols and middleware connecting to production facilities. With the combined system, customers have

access to current data from the factory, as well as trends and alarms for “outliers”; see Chapter 12. Access is not only available in the management offices, but also through augmented reality inside the factory.

– *TheThings.io – cloud interface*
  TheThings.io is yet another platform offering to publish your data in the cloud. One of the special features for this product is the interface to SigFox; see Chapter 9.
– *PLC – programmable logic controller*
  Many engineers tend to overlook PLCs when they are designing industrial solutions—maybe for historical reasons. PLCs were originally rather primitive devices. To make them more accessible to technicians and other people without a programming background, they used “ladder logic,” so that they resembled relays. However, PLCs has come a long way since then and may be a perfect solution to your challenges.

3.24 Further reading

The main source of information is the various vendors' websites. There are however, other sources like the following:

- 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.
- Derek Molloy: *Exploring Raspberry Pi*
  As above—but now for Raspberry Pi.
- Elecia White: *Making Embedded Systems – Design Patterns for Great Software*
  This book shows how to work on the hardware-near level. It introduces datasheets and block diagrams, and it explains how to work together with electrical engineers.
- 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 tests on systems—but also on CPUs. It is interesting because a lot of these data are available without membership. The general score on CPUs 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.
- bis.doc.gov/index.php/forms-documents/doc_view/1382-de-minimis-guidance
  This is a good starting point on export control.

Part II: **Best practice**

4 Software architecture

4.1 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, compared to a bad design, may be a factor of 10, whereas performance improvements in code optimization are 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 11. This all leads to the following mantra:

Design for Performance – Code for Maintainability

An example: In past years, object-oriented programming (OOP) has prevailed in PC programming. It is a fantastic concept for especially GUI programming. The idea of taking 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 which doesn't go well with a distributed system such as IoT.

The typical workflow in OOP is: you begin by instantiating 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 round trips. If you do a "tracert" (more on this in Chapter 7), you may see that a typical time for a packet to travel to, for example, ieee.org is 100 ms. Assuming 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, acting on, and answering the change, then we look into:

$$0.5 \text{ million} * 250 \text{ ms} = 125{,}000 \text{ seconds} = 35 \text{ hours}.$$

Surely you can do it all in parallel, reducing this time to 2 minutes, but handling 1,000 sockets in parallel is far from trivial, the error-handling if, for example, 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

https://doi.org/10.1515/9781547401024-004

you can design this so that the result of a setup action is either okay or not okay—not something in between. The execution time in the embedded device will be longer now, 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:

$$1000 * 300\,\text{ms} = 300\,\text{s} = 5\,\text{minutes}$$

If you insist on the 1,000 parallel sockets, it is done in 300 ms.

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 1,000 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 a SQLite database. The setup simply sends a new version of this; the device stores it and basically reboots. If a reboot takes 1 minute, the last device will be ready 6 minutes after you began—not so bad. Even the parallel scenario becomes doable due to the transactional nature of the operation. SQLite has the advantage of being file based, with the file format being the same on all platforms. This allows you to do a lot of tests in the protected PC environment.

Now, someone on the team might start to realize that the users really don’t update anything that often; they might use this approach for *every* update, not just the really big ones. This really depends on the application. If this is possible, it will guarantee that all devices are in the same, known configuration, which 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. The only state-change the device has to handle is to boot—something that was necessary anyway. Thus, by looking at system performance, instead of 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, but rather in a cross-functional team around a white board or beer.

Another—much more flexible—possibility for doing larger chunks of setup commands, is to use REST with “subtrees” in json or XML. This is discussed in Chapter 7.

4.2 The fear of the white paper

It can be terribly stressing to start from scratch. Most developers in the industry begin their career in a project with a lot of “legacy” code, which they extend and bug-fix.

1 Things like the IP address and sensor serial numbers will differ between devices.

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 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 had previously provided the document/view structure, serializing of files, observer-consumer pattern, and a lot more.

At the time a number of vendors of hardware—CPUs, FPGAs, 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. Eclipse was clearly the most popular choice in this community. Eclipse is an IDE platform originally made for Java, but since then extended for a number of tools. Eclipse is used for C/C++, but also for FPGA designs, DSP development, and a lot more.

Thus, in this case the answer was from another source, written for 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 break down the white paper? In the same way you would eat an elephant—one byte at a time. It is often stated that the agile approach works best when there is already a foundation—or platform. This 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 get going. 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 begin here then? Use this to learn the new language/compiler/IDE/library. And as demonstrated in Section 4.4, files are a great place to meet. Starting without a complete plan is daring. However, making a full plan of how you want to create 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: “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 begin by 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 of this has been good practice 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 in 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.

4.3 Layers

There are 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 layer below, whereas 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 Figure 4.1.

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 merely accepts orders. Similarly, the network layer knows about the data-link layer and not the other way around. Thus the lower layers may be interchanged with USB or something else. We have an incredible powerful modularity.

With the term "true layering," we tighten the concept. Now, an upper layer only knows about the one directly below it, not those further down. The internet protocol stack is not completely truly layered, the socket interface simply knows too much

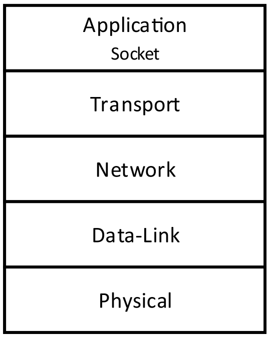


Figure 4.1: Layering.

about what goes on in the network layer, two stories down. On top of this, 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. On this level, we do not have full modularity. This also explains why many refer to the internet protocol stack as the TCP/IP stack.

4.4 Not just APIs—more files

In the embedded world, as well as in the PC world, you are often required to create an API, which the next developer can use. This is often necessary, but not always. One of the problems with an API is that after the test, there are no traces of what has happened. Any logging requires extra work by the test programmer. A lot can be learned from Linux (Unix). Every time something new is made for Windows, a new API is created. This is not the case with 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 showing processes is an 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 4.1. It is however, also possible to "pipe" data directly from the output of one process to the input of another, as the third line shows. In Unix and Linux, the processes might even run on different computers.

Listing 4.1: stdin, stdout, and pipes

```
1 progA > datafile
2 progB < datafile
3 progA | progB
```

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*, running it as shown in line 1 of Listing 4.1. 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 difference means 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 third line of the listing, you can use Wireshark (see Section 8.5 to ensure that the data is indeed corresponding to the datafile).

This is very practical in toolchains running on real PCs, but is also useful in the larger embedded systems as well.

4.5 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, for example, "A car has two axles,[2] 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 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 the latter case, the outward model is known as a "façade."

4.6 Case: 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 have different message-formats in the CAN buses. CAN has an 11-bit CAN-ID,[3] but the mapping of the possible values to measurement types is not standardized in the automotive world (with a few exceptions).

When industrial manufacturers discussed the use of CAN at the factory floor, this was not acceptable. They formed the group *CiA* (CAN in Automation) defining

2 It's a very primitive car.

3 CAN 2.0B also offers the choice of a 29-bit CAN-ID. This is not used in CANOpen.

CANOpen in 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 4.2 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.

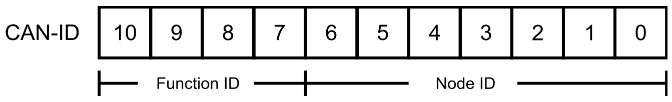


Figure 4.2: CANOpen usage of the CAN-ID.

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.

The object model in CANOpen is found in the object dictionary. This mimics a simple table where each entry, indexed by a 16-bit hexadecimal number, holds a specific function. It is, however, not that simple, as each index may hold several subindices. Some indices are mandatory and completely specified, while others are vendor specific. Each index starts with a number. This number indicates how many subindices the given index has. This provides the necessary flexibility, but is still rigid enough to assure that the software from one vendor can handle the hardware from another.

In my experience, engineers always asks for an export to Excel. Using a table as an object model in production is not a bad idea.

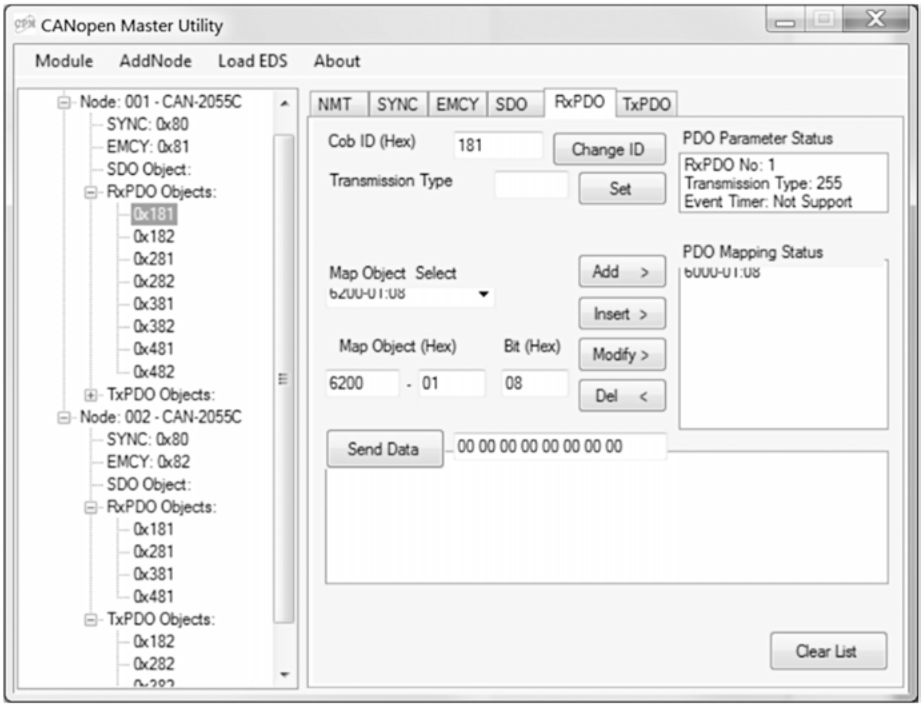
Figure 4.3 shows one vendor’s tool for managing the object dictionary. The GUI is crude, but it demonstrates the object model very precisely.

The 16 function types are grouped into main *message types*, shown in the figure and described in Table 4.1.

The tool uses the SDO messages to setup the system. When it comes to SYNC, Emergency, and NMT, you can often live with the defaults. This means that you may not have to bother about these at all. The real workhorses are the PDOs. These are sent during runtime when dictated by a timer, or when a measured value changes more than

Table 4.1: Important CANOpen message types.

MsgType	Usage
NMT	Network management. Heartbeat, etc.
SYNC	Synchronization setup
EMCY	Emergency messages
SDO	Service data objects. General setup of, for example, 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?



```
Node: 001 - CAN-2055C
    SYNC: 0x80
    EMCY: 0x81
    SDO Object:
    RxPDO Objects:
        0x181
        0x182
        0x281
        0x282
        0x381
        0x382
        0x481
        0x482
    TxPDO Objects:
Node: 002 - CAN-2055C
    SYNC: 0x80
    EMCY: 0x82
    SDO Object:
    RxPDO Objects:
        0x181
        0x281
        0x381
        0x481
    TxPDO Objects:
        0x182
        0x282
```

Cob ID (Hex): 181
Transmission Type:
Map Object Select: 6200-01:08
Map Object (Hex): 6200 - 01   Bit (Hex): 08
Send Data: 00 00 00 00 00 00 00 00

PDO Parameter Status:
RxPDO No: 1
Transmission Type: 255
Event Timer: Not Support

PDO Mapping Status:
6000-01:08

Figure 4.3: ICPDAS user interface for CANOpen.

a specified delta. By writing 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*.

4.7 Message passing

In the chapter on operating systems, especially Section 2.8, we looked at various mechanisms helping tasks to avoid stepping on each others toes. Multitasking can be problematic to debug on a single-core CPU, and it only gets worse as we see more CPUs with multiple cores. In these, we experience true parallel processing. There is no way around the use of these complex constructs, when it comes to writing code for the OS kernel or drivers. However, it is definitely worth *not* having to do this in normal user-space code.

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

For this reason, *message passing* has been a popular solution for many years. This typically involves copying data into the message sent from one task to another, which affects performance. Modern CPUs are, mostly, 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. There are no problems with data-sharing, once each task is working on its own data. As stated before, message passing also gives us better debug possibilities. The fact that one or more tasks may be executed on a PC, while others are run in the real target is an added bonus.

An OS like Eneas OSE is built around asynchronous message passing working seamlessly between CPUs in a kind of local network. Likewise, many small kernels support message passing within the same CPU. Today there are a number of platform-independent concepts such as ZeroMQ. ZeroMQ is based on the well-known socket paradigm and is definitely worth looking into. At the moment, it uses the LGPL (lesser GNU public license), meaning that you can link to the library from proprietary code.

Sometimes copying is not the solution. Streams of measurement data may be shared via standard “rotating buffers.” It is easy to make mistakes in these constructs—a pre-debugged version is practical. When it comes to data structures that must be shared, you may want to use a database, for example, SQLite. Again, someone else has spent resources on locking-concepts that work in embedded. As stated earlier, you also get some great possibilities for transporting default setups into the device, and exporting “strange cases” out of the device for debugging.

4.8 Middleware

Although features typically assigned to *middleware* have already been discussed, the term “middleware” has not. It is a confusing term, because it means so many different things, but it is important in a discussion on software architecture. Here are some examples of alternative views on what middleware is:

– *All the software between the Application Layer and the Operating System.*
  This includes a lot of libraries running in user-space.
– *An Inter/Intrasystem Communication Concept.*
  This could, for example, be *ZeroMQ* (zero message queue) or OS-specific constructs.
– *Remote Procedure Calls.*
  RPC may be a kind of middleware for an old-style “master-slave” architecture, but it has no place in IoT where devices may come and go, and a more services-based approach is needed.
– *An Object Broker.*
  This could be CORBA or something more modern based on REST; see Section 7.12.

– *A Full (application) Framework.*
  See below.

Modern smartphones contain a framework which typically helps with, but also dictates, many features. Examples are: gesture control, graphics handling, speech recognition, GPS access, power handling, interapplication communication, etc. All this is included in Android. In this case, Android is often talked about as being the OS. It is more correct to say that Android is a lot of preselected middleware on top of Linux.

Alternatively, you can pick libraries into your custom Linux-based, or completely home-grown system. Interestingly, all this belongs in the previously discussed “application layer” in the five-layer model. This means that the five-layer model may be okay when dealing with low-level datacommunication, but it really isn’t suited as a model for a full design. The same goes for the seven-layer OSI model: it too serves well for communication, but bundles all middleware together with the application in the top-most layer. In both models, the top layer is very crowded.

It seems like “middleware” in the phone industry is used as a synonym for “application framework,” whereas in the PC world it is a layer allowing programmers to work with distributed systems, without spending too much time on the fact that they are distributed.

In reality, it may be more difficult to switch middleware than to move to another operating system. Depending on the application, you may need a subset of typical middleware, like REST and an object model for the communication part, or you may need a full set like the one offered by Android.

4.9 Case: architectural reuse in LAN-XI

The very entertaining book *The Inmates are running the Asylum*[4] claims that reuse of software is a false goal. Most software designers have experienced the frustrations of creating something in the “right reusable way,” only to see it used once, or maybe not at all. The code gets bloated while you are preparing it for a lot of vaguely known future requirements, and this may make it unfit for the job at hand. This does, however, not mean that all reuse is bad.

A very well proven way to reuse code is *product-line reuse*. This is demonstrated in Figure 4.4.

The product shown is one of Brüel and Kjær’s *LAN-XI* modules. The first two modules were introduced in 2008, and since then approximately one module and/or frame (not shown) has been released per year. Each module basically measures, or generates, 300 kHz in channel $*$ measurement bandwidth, corresponding to 20 Mbps on the LAN. 100+ modules can work as one. Thanks to IEEE 1588 Precision Time Proto-

4 The inmates are the software designers and the asylum is the company they work for.

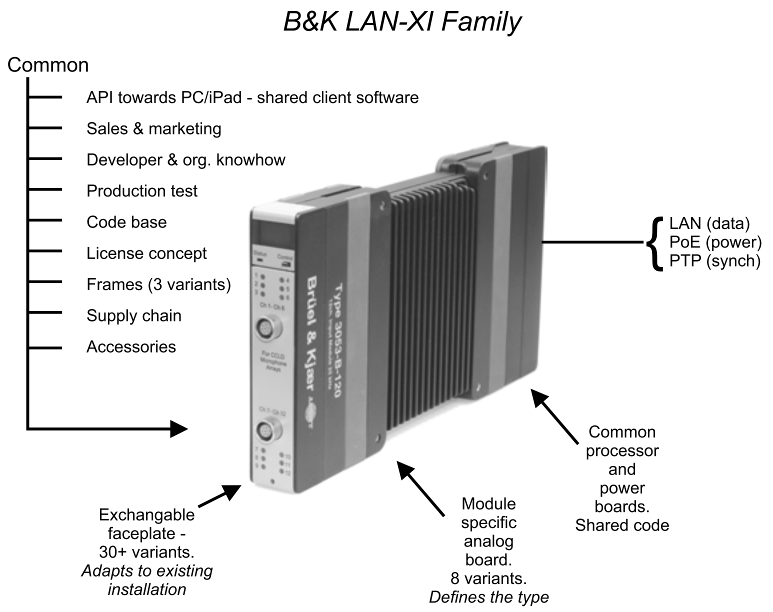

B&K LAN-XI Family
Common
API towards PC/iPad - shared client software
Sales & marketing
Developer & org. knowhow
Production test
Code base
License concept
Frames (3 variants)
Supply chain
Accessories
LAN (data)
PoE (power)
PTP (synch)
Common processor and power boards. Shared code
Module specific analog board. 8 variants. *Defines the type*
Exchangable faceplate - 30+ variants. *Adapts to existing installation*
Brüel & Kjær
Type 3053-B-120

Figure 4.4: Product-line reuse.

col (time-synchronization over Ethernet), sample-synchronization is better than 50 ns between any set of channels.

The modules may work “stand-alone,” powered via the Ethernet, or inside a frame with a built-in switch in the backplane. Thus, these modules are not your garden-variety IoT devices. Nevertheless, they are built on the technologies that many newer IoT devices utilize. The Wi-Fi measurements in Chapter 9, using Wireshark, are based on a LAN-XI module communicating with an iPad through a wireless router.

The initial launch required the work of many developers, spanning all design disciplines, while each later module is realized with considerable less effort due to reuse. As Figure 4.4 shows, the reuse inside the R&D department is only a part of the gain. Supply chain, licenses, production tests, etc. are almost the same for all modules. One of the biggest benefits is that once a customer knows one module, he or she knows them all.

The only thing that separates one module from its siblings, is the analog board as shown in the figure. Processor board, power board, display, mechanics, etc. are the same. The embedded code is also generic. It learns during initialization which module it is part of, based on parameters that are configured in the final step of production. FPGA code differs in some modules, but in this case it is part of the common downloaded image, and the device selects the part to use during the download. “Faceplates” with different connectors allow the basic system to fit within existing customer installations. For forward compatibility, each faceplate contains a picture of itself. This allows older client software to show the correct image of a new “unknown” set of connectors.

There can be no doubt that LAN-XI has been a success due to reuse. However, this is not the reuse of a software module or library. It is the reuse of an architecture. The development team has always considered the full product-line whenever a new feature is developed. This means that short-sighted benefits in the module, currently worked on, are never done at the cost of long-sighted architectural inconsistencies.

Even though the incremental work per new module has been much smaller than the original effort, far more than 50 % of the total work on the product line has taken place after the initial launch. This work has been partly on new modules, partly on general maintenance, as well as on new features, working on the already sold modules (within reasonable bounds). This is consistent with data from the Software Engineering Institute (SEI) at Carnegie Mellon University from 1999. SEI puts the similar number as high as 80 %. LAN-XI still spawns new family members, and may in time confirm this number. Another statement from SEI, confirmed in the work with LAN-XI, is that architecture defines:

- Division into teams
- Units for budgeting and planning
- Basis of work-breakdown structure
- Organization of documentation
- Basis of Integration
- Basis of test-plans and testing
- Basis of maintenance

So, still quoting SEI:

> *"Once decided, architecture is extremely hard to change. A good architecture is one in which the most likely changes are also the easiest to make."*

4.10 Understanding C

The *C Programming Language* by Kernighan and Ritchie still stands out. Even the second edition is old—from 1988—but it includes the ANSI-C changes, which is important. Given the knowledge 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 of C. The latest is C17 (from 2017), which mainly is a bug fix of C11, which again partly was fixing problems introduced in C99. The biggest thing in C11 is probably the standardization on the multithreading functionality, which we briefly touched upon in Section 2.5. This has been supported on all major platforms for many years, but not consistently. The two things from C99, which seem most popular, are the inline comments (//) and the intermingled declaration of variables, both already 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 to *Code Complete (2)*. It only makes sense to throw in a few experience-based guidelines:

1. Don't be religious. There is no productivity in fighting over where the starting brace is. Accept living with a company standard.
2. The idea of coding for vague, currently unplanned reuse, is a false goal (see further reading: *The Inmates are Running the Asylum*). If your program does its job well it might be reused. If, on the other hand, you plan the code to be reusable far beyond the current context, the code will be inefficient, and thus never (re)used. 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 are 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 and readability. The single thing that can introduce most bugs, and hurt your performance the most, is when ignorance of the existing code causes the programmer to revalidate, recalculate, reread data, etc. The next programmer does not even have to be another guy; it is amazing how quickly you can forget your own code.
6. Use comments. Not the silly kind that rewrites every line of code into other, but similar words. Instead make function headers that explain 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 reentrant or threadsafe or neither, whether a specific order of initialization is required, etc. Comments inside the functions are welcome as long as they do not just state the obvious. Typically, they can describe the general flow of the program, assumptions on the data, or in special cases explain a more intricate line of code. In a function header, it is practical to have remarks on "who creates who" and "who kills who."
7. Single variables of the basic types are fine when they live locally in functions, but when it comes to data lasting longer, the variables should be grouped into high-level structures such as *structs* and *arrays*. *Unions* and *bit fields* are especially popular in the embedded world, 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" this is a flag, and these two bytes are a 16-bit signed number, etc.
8. In a small embedded environment it is normally considered "bad" if the program calls `malloc()` or uses `new` after the initialization phase. Allocating and deallocating memory on the *heap* is very nondeterministic. Typically, an embedded program is not turned off in the evening, and often there is a long way to the nearest user who can do a reboot or other type of restart. If you are programming

in C++, consider making the constructors "private," so that the compiler does not silently create objects on the heap.

9. 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.
10. "objdump -d -S <binary>" produces a listing of mixed source and assembly-code with gcc. Use this (or similar) before and after you make a "clever" C-code. Sometimes the compiler is not as dumb as you think. Counting instructions is however not the full story. CISC CPUs have instructions taking many cycles and some taking only a single.
11. When accessing files, avoid random access if possible.

    Long ago I had implemented a Unix program that scanned an "Intel Hex" object file, found a 26-character dummy date, 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. This time it only took a minute.

12. 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 several OR'ed expressions, evaluation stops as soon as one is true. Similar with a chain of ANDs, once an expression evaluates to false, the C-compiler guarantees the rest are not evaluated. This is very, very practical; see Listing 4.2. This is actually C# code, but the concept is the same for all C-derived languages.[5] In this case, we want to access *"grid.CurrentRow.DataBoundItem"* but any part of this could be null. The code allows us to test from left to right in a single if-statement.
13. Remember that most CPUs will give you both the integer quotient and the remainder when you perform integer division. However, 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 use too many cycles.

Listing 4.2: Using limited, left to right evaluation of logic operator

```
if (grid != null && grid.CurrentRow != null &&
    grid.CurrentRow.DataBoundItem != null)
   // OK to use grid.CurrentRow.DataBoundItem
```

5 Note that especially in the C# case there is an alternative solution based on the so-called "null propagation."

4.11 Further reading

- Alan Cooper: *The Inmates are Running the Asylum*
  This is a fantastic, easy-to-read and entertaining 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 we work for.
- Gamma and 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 might 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 and 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++.
- Paul C Clements et al.: *Software Architecture in Practice*
  www.researchgate.net/publication/224001127
  This is a slide-show from SEI at Carnegie Mellon University related to a book of the same name. It has some great views about architecture and architects, as well as examples of reuse by product line.

5 Debug tools

5.1 Simulator

When I was studying assembly programming at the university, simulators were very popular. A simulator is a program which in our case simulated an 8080 microprocessor. We could 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 simulators that much. Probably because I have better alternatives.

If 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 and 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, for example, digital signal processing, they may work on input and output files as shown in Section 4.4, 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, still doing all the algorithm-heavy stuff on your PC, with the nice and fast debugger and the direct connection to your version control system, while your target serves as a door to the physical world.

5.2 ICE – in-circuit emulator

An emulator can be fantastic. Essentially, it's a computer system from where a connector can slide into a microprocessor socket. It can do everything that the microprocessor can, in real time. However, it can do a lot more. You may "trace" what happens 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 set, the ICE would break. Today such a bug might be found through static code analysis.

https://doi.org/10.1515/9781547401024-005

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 the box has to be opened. Maybe you need extension PCBs to get the PCB in question "out in the open," and extension PCBs 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 ICEs are rare, but they do exist, and sometimes they can be the answer to your problem.

5.3 Background or JTAG debugger

Many intelligent integrated circuits, as microprocessors and microcontrollers, have a JTAG interface. This is used during production for testing. Via JTAG a low-level program assures that there are no short-circuits, and that IC A actually can "see" IC B. Due to the above mentioned drawbacks with ICEs, microcontroller manufacturers decided to build in a few extra circuits into 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 this time 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 an emulator has. It may have one or two breakpoint registers for breaking on specific addresses, but typically they cannot break when, for example, a data cell (variable) is changed. Sometimes you can buy extra equipment, like an extra box with RAM for tracing.

Please note that an open JTAG in an embedded system is also a security risk; see Section 10.19.

5.4 Target stand-in

One virtue the simulator has over the ICE, and the background debugger, is that it doesn't need a target. Unfortunately, a simulator 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.

As described earlier, EVM stands for “evaluation module.” This is a PCB that the CPU vendor typically sells rather cheap. It has the relevant CPU, along with some peripherals and some connections to communicate with the outer world. This could, for example, be USB, HDMI, and LAN, and often also some 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.

In a project, we worked two-third’s of the development time, using two EVMs—one for the CPU and another for the DSP. We used some older hardware for interfacing to the real world. It was all tied together with a standard switch (possible since the final product uses LAN inside). For a very long time, everything was running in a shoe box with a mess of EVMs, cables, etc.—a fantastic solution.

Another popular solution is to use, for example, Raspberry Pi or BeagleBone as we have seen in other chapters of this book. A plus for the BeagleBone is that it is open source hardware. Thus, 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 you understand the hardware. It is also nice that Yocto[1] has a BeagleBone reference design. 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.

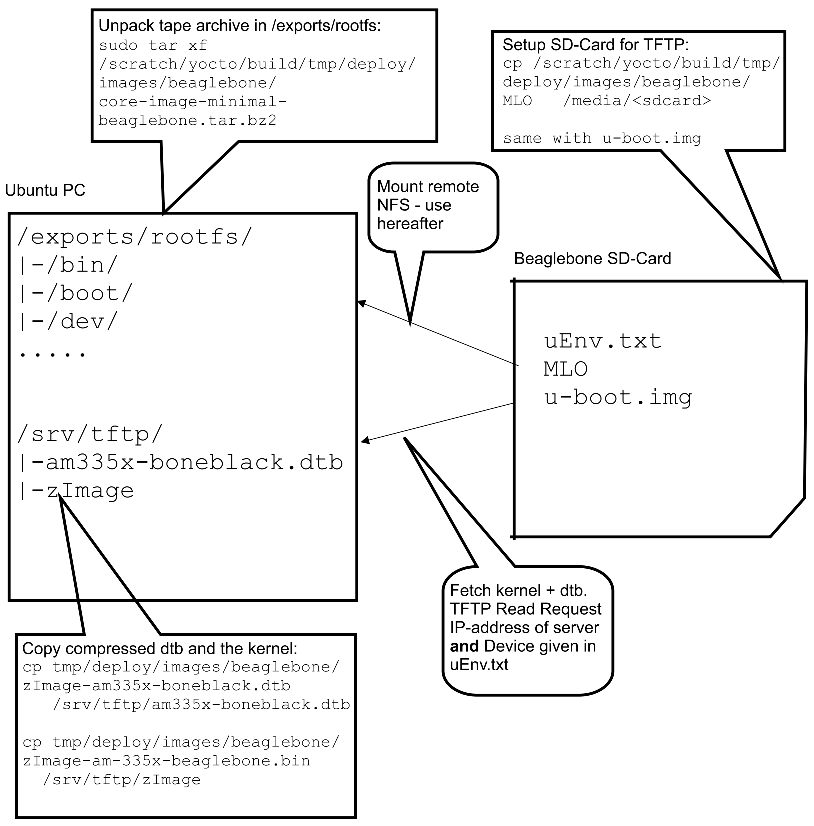
When you get to the point where you actually do have a target, it can be quite disappointing. From having the short turnaround time from editing a C-file to seeing it run, it now takes much longer. Getting the binary code into the target may, for example, require that an SD-card is programmed at the host, and then manually moved to the target. In such a case, it is often possible to mount the host system’s disk on the target. This allows you to have the fast compile on the host, and immediately after use it on the target. Using *TFTP,* the kernel is copied by the boot-loader to the target RAM, and hereafter the filesystem on the host is used via the Network File System (NFS).

This scenario is shown in Figure 5.1. The dashed, rounded boxes show how this happens at runtime, while the square boxes show how the output from a Yocto-build is copied to the relevant places.

5.5 Debugger

A debugger is one of the most important tools a developer can have, but it is easy to lose overview. 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

1 See Section 6.9.



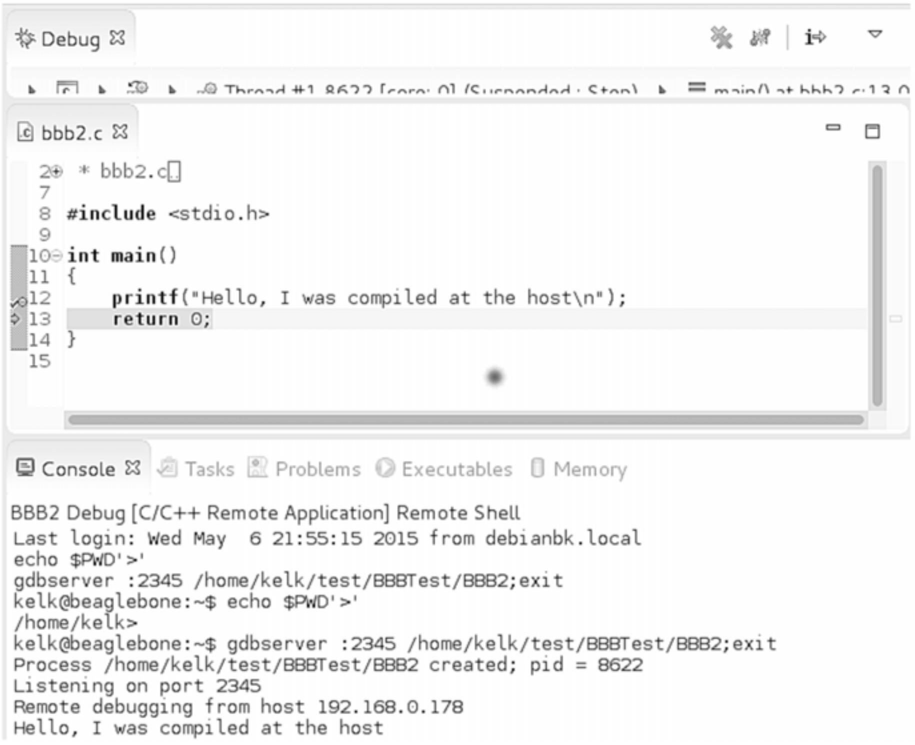
Unpack tape archive in /exports/rootfs:
sudo tar xf /scratch/yocto/build/tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.tar.bz2
Setup SD-Card for TFTP:
cp /scratch/yocto/build/tmp/deploy/images/beaglebone/MLO /media/<sdcard>
same with u-boot.img
Ubuntu PC
Mount remote NFS - use hereafter
Beaglebone SD-Card
/exports/rootfs/
|-/bin/
|-/boot/
|-/dev/
.....
/srv/tftp/
|-am335x-boneblack.dtb
|-zImage
uEnv.txt
MLO
u-boot.img
Fetch kernel + dtb. TFTP Read Request IP-address of server **and** Device given in uEnv.txt
Copy compressed dtb and the kernel:
cp tmp/deploy/images/beaglebone/zImage-am335x-boneblack.dtb /srv/tftp/am335x-boneblack.dtb
cp tmp/deploy/images/beaglebone/zImage-am-335x-beaglebone.bin /srv/tftp/zImage

Figure 5.1: BeagleBone target using host disk via TFTP and NFS.

experience that they debug all day, but solve the problem when they are driving home from work—I know I have.

Nevertheless, you depend on a debugger where you can trust that if it didn't break at a given line, it's because it never came there. Not because you have forgotten a detail in the setup. Nothing is more frustrating than almost having reproduced a problem, and then the program takes off. So make sure you know how it works. Figure 5.2 shows remote debugging on a BeagleBone.

In my professional life I have been going back and forth between Linux and Windows as well as between programming and management. 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 a 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 5.2.



```
 * bbb2.c

#include <stdio.h>

int main()
{
    printf("Hello, I was compiled at the host\n");
    return 0;
}
```

```
BBB2 Debug [C/C++ Remote Application] Remote Shell
Last login: Wed May  6 21:55:15 2015 from debianbk.local
echo $PWD'>'
gdbserver :2345 /home/kelk/test/BBBTest/BBB2;exit
kelk@beaglebone:~$ echo $PWD'>'
/home/kelk>
kelk@beaglebone:~$ gdbserver :2345 /home/kelk/test/BBBTest/BBB2;exit
Process /home/kelk/test/BBBTest/BBB2 created; pid = 8622
Listening on port 2345
Remote debugging from host 192.168.0.178
Hello, I was compiled at the host
```

Figure 5.2: Debugging BeagleBone target using Eclipse on the host.

5.6 strace

This is not a Linux book, and we will not go into all the nice tools on Linux, but *strace* is mentioned because it's 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 7.19, something was wrong. To dig into the problem, I used strace. This is a Linux command used on the command line in front of the program to be tested, for example, `strace ./udprecv`. In this case, it was used in two different terminal windows to avoid the printout to mix. Listings 5.1 and 5.2 show 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).

Listing 5.1: strace IPv6 receiver

```
1 strace ./recvsock
2 socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
3 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
```

```
bind(3, {sa_family=AF_INET6, sin6_port=htons(2000),
inet_pton(AF_INET6, "::", &sin6_addr), sin6_flowinfo=0,
sin6_scope_id=0}, 28) = 0
recvfrom(3, "The␣center␣of␣the␣storm\n\0", 100, 0,
{sa_family=AF_INET6, sin6_port=htons(45697),
inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0,
     sin6_scope_id=0}, [28]) = 25
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    -1, 0) = 0x7fc0578cb000
write(1, "Received:␣The␣center␣of␣the␣stor"..., 34Received:
The center of the storm) = 34
exit_group(0)                         = ?
+++ exited with 0 +++
```

Listing 5.2: strace IPv6 sender

```
strace ./sendsock
socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
sendto(3, "The␣center␣of␣the␣storm\n\0", 25, 0,
{sa_family=AF_INET6, sin6_port=htons(2000),
inet_pton(AF_INET6, "::1", &sin6_addr), sin6_flowinfo=0,
sin6_scope_id=0}, 28) = 25
exit_group(0)                         = ?
+++ exited with 0 +++
```

5.7 Debugging without special tools

Single-stepping is great when you know what you are looking for, but you *will* lose overview. 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. This concept is typically criticized (by tool vendors and others) for slowing down the execution, requiring 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 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, for example, "engine control."

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, collect the log and mail it to you. Alternatively, you may login remotely or ... Remember that the `printfs` 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 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, you will build a helpful tool.

5.8 Monitoring messages

Several places in this book, I have suggested message passing as a way to handle high-level interaction between tasks. If you are using this, you may have some great debugging facilities. Surely there will be a lot of functions calling functions as usual, but at some high level, tasks 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 they started, but eventually they go back to the input queue for the next job. All this is handled by a system process, often a part of the kernel.

Now, again at runtime, a switch may enable the program to write 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. This printout could even contain the most important fields in each of the waiting messages, telling more about who is producing data and who is not consuming, and vice versa.

In most message based systems, the messages come from pools. Each pool is initialized to have a given number of messages of a given size. This way memory does not become fragmented. The tasks 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, for example, one message can be outstanding.

If the number of buffers in each pool are printed out, it is possible to see problems building up, or at least spot them after the fact. Also here it makes sense to see the "type" field, or whatever it's called, as it may tell you who is returning the buffers correctly, and who is not.

5.9 Test traffic

When testing network features, scripts are often faster and more efficient to write than performance-optimal C-code. *Python* is a popular scripting language, and with the *ScaPy* library, testing network traffic can be a breeze.

Listing 5.3 is a simple, efficient example using TCP defaults, except for the ports which are previously set variables, and the TCP flags that are overwritten so that a TCP “FIN” packet is sent (also has the ACK flag set). The “/” separates the layers and if there is only the layer name in caps, it behaves “normally,” inserting counters and checksums with their right value, etc. The interesting thing is that it is also possible to override any default to generate faulty checksums, illegal lengths, etc.

Listing 5.3: ScaPy with IP and TCP layers

```
fin_packet = myip/TCP(dport=remote_port,
                      sport=local_port,
                      flags="AF",
                      /"Here␣are␣the␣last␣data"
```

Listing 5.4 is a full listing of a more elaborate script—used to test a DHCP server. It makes it possible to perform the client DHCP actions one-by-one (more on this in Chapter 7). The first call must be made with the “init” parameter and the output interface to be used. In this case, it was “wlp2s0.” This is found with the help of *ifconfig* on Linux or *ipconfig* on Windows. This initial call saves configuration data in a json file. The following calls use and update this file. These calls take as parameter a client’s DHCP action, for example, “discover.” The script sends the relevant command and dumps the DHCP “options” part of the response from the DHCP server.

The script also shows a few of the nice things in general Python:
– *actions* in the main—seen in the last lines
– Simple file handling
– Parsing of json via *eval*
– Logging

Listing 5.4: DHCP test client in Python

```
#!/usr/bin/env python3

#call ./dhcp.py <function>
# where function is one of init, discover, request or help
# Install Python3 and scapy to run this

import json

#Kill IPv6 warning:
import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)

import scapy
from scapy.all import *

#discover may be the first
def do_discover():
   print("In␣do_discover")

# VERY important: Tell scapy that we do NOT require
# IP's to be swapped to identify an answer
   scapy.all.conf.checkIPaddr = False

   settings = eval(open("dhcp.conf").read())
   iface    = settings["iface"]
   mac      = get_if_raw_hwaddr(iface)

   dhcp_discover = (
       Ether(dst="ff:ff:ff:ff:ff:ff") /
        IP(src="0.0.0.0",
        dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(chaddr=mac, xid=5678) /
        DHCP(options=[("message-type","discover"),"end"]))
#  dump what we plan to send (debug)
#  ls(dhcp_discover)

   disc_ans = srp1(dhcp_discover,iface=iface,
                 filter="udp␣and␣(port␣67␣or␣68)")

# The answer is a DHCP_OFFER - check it out
   print("Options:", disc_ans[DHCP].options)
#  print("xID:", disc_ans[BOOTP].xid)

# Save the offer to be used in a request
   settings["serverIP"] = disc_ans[BOOTP].siaddr
   settings["clientIP"] = disc_ans[BOOTP].yiaddr
   settings["XID"]      = disc_ans[BOOTP].xid

   with open('dhcp.conf','w') as file:
      file.write(json.dumps(settings))
   return

#this does a request without a discover first
def do_request():
   print("In␣do_request")
# VERY important: As before...
   scapy.all.conf.checkIPaddr = False

   settings = eval(open("dhcp.conf").read())
   iface    = settings["iface"]
   mac      = get_if_raw_hwaddr(iface)

   dhcp_request = (
       Ether(dst="ff:ff:ff:ff:ff:ff") /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(chaddr=mac) /
        DHCP(options=[("message-type","request"),
        ("server_id",settings["serverIP"]),
        ("requested_addr",settings["clientIP"] ),"end"]))
#  dump what we plan to send
#  ls(dhcp_request)

   ans_req = srp1(dhcp_request,iface=iface,
                  filter="udp␣and␣(port␣67␣or␣68)")
   print("Options:", ans_req[DHCP].options)
#  print("xID:", ans_req[BOOTP].xid)
   return

#this does a discover - then a request
def do_discover_request():
   print("In␣do_discover_request␣-␣not␣implemented␣yet")
   return

def do_none():
   print("Try␣./dhcp.py␣help")
   return

def do_init():
   try:
      iface = sys.argv[2]
   except:
      print("init␣must␣have␣the␣relevant␣interface␣" \
          + "as␣second␣parameter")
      return

   settings = {"serverI": "255.255.255.255",
               "clientIP": "0.0.0.0",
               "XID": "0",
               "iface": iface}
   with open('dhcp.conf','w') as file:
      file.write(json.dumps(settings))
   return

def do_help():
   print("Examples␣of␣usage:")
   print("␣␣␣./dhcp.py␣init␣wlp2s0␣␣␣␣␣␣" \
       + "Second␣Param:␣interface.␣This␣is␣stored␣in␣dhcp.conf")
   print("␣␣␣sudo␣./dhcp.py␣discover␣␣␣␣" \
       + "Send␣a␣DHCP␣DISCOVER␣and␣recv␣a␣DHCP␣OFFER")
   print("␣␣␣sudo␣./dhcp.py␣request␣␣␣␣␣" \
       + "Send␣a␣DHCP␣REQUEST␣and␣recv␣a␣DHCP␣ACK")
   return

```

```
action = {'discover': do_discover, 'request': do_request,
          'discover_request': do_discover_request,
          'init': do_init, 'help': do_help}

#main code

try:
   command = sys.argv[1]
except:
   command = "none"

action.get(command, do_none)()
```

The result from running the above on Linux is seen in Listing 5.5. Note that if it is run on a virtual PC, it is likely not to work. As discussed in Chapter 7, DHCP is often sent as broadcasts, and these may not pass through the host PC to the network. In this case, a bootable Linux partition on a Windows PC was used.

The output lines are broken for better readability. The following request message is not shown. Since we only output the options, it looks very much like the discover message. Only difference is that “message type” is now 5.

Listing 5.5: Output from DHCP test

```
kelk@kelk-Aspire-ES1-523:~/python$ ./dhcp.py init wlp2s0
kelk@kelk-Aspire-ES1-523:~/python$ sudo ./dhcp.py discover
In do_discover
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Options: [('message-type', 2),
 ('server_id', '192.168.1.1'),
 ('lease_time', 86400),
 ('renewal_time', 43200),
 ('rebinding_time', 75600),
 ('subnet_mask', '255.255.255.0'),
 ('broadcast_address', '192.168.1.255'),
 ('router', '192.168.1.1'),
 ('domain', b'home'), (252, b'\n'),
 ('name_server', '8.8.8.8', '8.8.4.4'), 'end']
```

Python can be problematic to install on Windows. Various versions of Python and its libraries do not always match. For this reason, the *Anaconda* environment is recommended. With this, you get versions that work together, as well as a nice integrated development environment called *Spyder* with editor and debugger. Do go into Tools-Preferences-Run in Spyder and put a check-mark on “Clear all variables before execution.” This will help you avoid strange phenomena such as a deleted variable that still works—until next time the program is loaded.

5.10 Further reading

- keil.com/mdk5/ds-mdk/
  This is the home of the DS-MDK heterogeneous environment from arm/Keil (not free). If you want to work with Linux as a target, but prefer to stay with the Windows development environment this could be a solution. There is also a smaller solution for homogeneous environments with a small RTOS. This solution is known as Keil μVision.
- anaconda.org/anaconda
  The home of Anaconda—a Python environment with free as well as paid versions.
- lauterbach.com
  Home of some very advanced emulators and debuggers.
- technet.microsoft.com/en-us/sysinternals
  A great site for free windows tools. Monitor processes, files, TCP, and much more.

6 Code maintenance

6.1 Poor man’s backup

There are many great backup programs, but you can actually get far with the free stuff already on your system. The modern cloud-based backup programs can—in my experience—be immature, dictating unreasonable requirements to your disk-structure. An older trusted solution is given here. On Windows, “robocopy” is very persistent in its attempts to copy 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.

Listing 6.1: Poor mans backup using Robocopy

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

Table 6.1 describes the options used—there are more.

Table 6.1: Robocopy parameters.

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 taskscheduler
4. Create basic task
5. Follow wizard. When asked how often—say once a week. And then check-mark the relevant days in the week.
6. If you are copying to a network drive make sure 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 at lunch or otherwise not using your PC.

The faithful Linux equivalent to the above, is to use the “tar” command to copy and shrink the data and “crontab” to setup the “cron” daemon, scheduling the regular

https://doi.org/10.1515/9781547401024-006

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 both the above procedures: Divide the code in *what* and *when*.

6.2 Version control—and git

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, and that when the project is in the stabilization phase, single bug-fixes are checked in atomically with reference to a bug-number. This is important because the most error-prone type of code is bug-fixing. We want to be able to:

1. Make a "shopping list" of all bug-fixes introduced between version A and version B. In business-to-business, most customers are mostly in a state where they want the exact same version as yesterday, only with a particular bug-fix. This is not very practical for the developers, as it is rarely the same bug-fix 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 trigger "ah, yes I would actually like that fix, too," and at least it shows the customer that we as a company know what we are doing, and what we are asking him or her to take home. Working with consumer products does not give the above customer requirements, however, the process still makes sense. If you have to produce a bugfix-release here and now, you may still want to find something with the minimum change. This requires less testing for the same amount of confidence.
2. Be able to answer exactly which version the customer needs in order to get a specific bug-fix. This relates to the above—a minimal change-set.
3. Know which bug-fix disappears if something is rolled back due to new bugs or unwanted side effects.
4. Know that any rollback will roll back full fixes, not just a parts of these.
5. Make branches and even branches on branches, and merge back. Manual assistance may be necessary with any tool.

On top of the above, there are more "nice-to-have" requirements such as a graphic tool integrated into Windows Explorer, integration with IDEs, etc.

In principle, all the above requirements are honored by "old-school" tools such as cvs, Subversion and Perforce. However, if you work with modern software development, you know that today *git* is it. The major challenge with git is that it has a philosophy quite different from the others. Git has a number of great advantages, at the cost of increased complexity. If you work with Windows software, or a completely custom embedded system, you and your team[1] can probably choose your tool freely, but if you work with Linux or other open source, it is not practical to avoid git. Most likely, you will love it in the long run, but there can be moments of cold sweat breaking before you get to that point.

Up until git, there was a central *repository*, often known as a *repo*, containing all the code, and developers could checkout whichever part they needed to work with. So either your code was *checked in*—or it wasn't. With git the repo is distributed, so that everybody has a full copy. You *commit* code from your working directory to your local repo, or *checkout* code the other way. But you need to synchronize your repo with a *push* to the central repo, in order to share code with other developers. You could say all repos are created equal, but one is more equal than the others; see Figure 6.1.

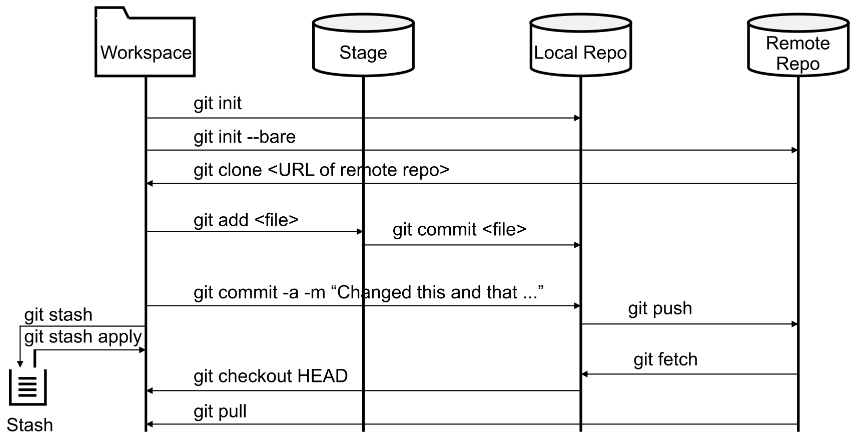


Figure 6.1: Main git actions.

Some major benefits of this concept is that it is lightning fast to commit or checkout, and a server meltdown will probably not lead to a disaster. Another benefit with git is that you can *stage* which parts of your changed code you want to commit. As stated, it is extremely important that bug-fixes or features are checked in atomically, so you either get the full fix, or nothing, when you later checkout the source. You may have tried the scenario where you are working on a feature and then asked to do a small and "easy" fix. The actual code change is quite simple, but you cannot commit all the now changed code without side effects. So you need to only commit the changes relating to the new bug-fix. Git introduces a "staging area" aka *index*. This is a kind of launch

1 Choosing version control is not like picking your favorite editor. It must be at least project-wide—maybe even company-wide.

ramp onto which you place the source files, or maybe even just parts of source files, you want to commit. This is great, but it does introduce one more level between your working directory and the shared repo.

Finally, git also introduces a so-called *stash*. This is a kind of parking lot, where you can park all the stuff you are working on while you briefly work on something else, like the fast bug-fix. If you have a newer version of a work-file you do not want to commit, it can obstruct a `git pull` from the remote repo. You can go through a merge, but instead it may be easier to `git stash` your workdir, do the `git pull` of the remote repo, and finally do a `git stash apply` bringing back your newer workfile. Now you can commit as usual, avoiding branches and merges. Basically, this is close to the standard behavior of, for example, Subversion.

Getting the grips of git is thus not so simple, but it is an investment worthwhile. John Wiegly has written a recommendable article "Git from the bottom up." He explains how git is basically a special file system with "blobs" (files) that each has a unique "safe" hash,[2] and how commits are the central workhorse, with branches being a simple consequence of these. John does some great demos using some scary low-level commands, showing the beauty of the inner workings of git. The scary part of these commands is not so much his usage, but the fact that they exist at all.

An extremely nice thing about git is that renaming files is simple. With other version tools, you often ponder that you ought to rename a file, or move it up or down a directory level, but you shy away from this, because you will lose history, and thus make it difficult for others to go back to older versions. Git's `git mv` command renames files. This command does all that is needed on the local disk as well as in the index/stage. Likewise, `git rm` removes files locally and in the index. The only thing left to do in both cases, is to commit. The history still works as expected.

There are a number of graphic tools for git, but you can go far with *gitk* which works on Linux as well as on Windows, and comes with a standard git installation. It is "view-only" which can be a comfort, not so easy to do something terrible. An interesting part of the standard installation on Windows is a *bash* shell. Like gitk this may be fired up via a right-click in Windows Explorer. This is a good place to practice the Linux command shell before you take the bigger leap to a real or virtual-machine Linux. It is perfectly possible to use git inside a normal Windows command shell, but it is a little weird to use "DOS" commands for standard file operations and use "rm," "mv," etc. when commands start with "git." Table 6.2 shows some of the most frequently used git commands.

Clearly, it is easy to make a repo. Therefore, it makes sense to make many smaller repos. It is almost just as easy to clone yet another repo as it is to get a zip file and unzip it. This is an important difference from the old version-control systems.

2 Hashes are discussed in Section 10.4.

Table 6.2: Selected git commands.

Command	Meaning
`git init`	Creates ".git" repo dir in the root of the workdir it is invoked in
`git init --bare`	Creates the repo without workdir files
`git clone <URL>`	Create local repo and workdir from remote repo
`git status`	Where are we? Note that *untracked* files will confuse if .gitignore is not used well
`git add <file>`	Untracked file becomes tracked. Changed files are staged for commit
`git commit <file>`	File committed to local repo. Use -m for message
`git commit -a`	All changed, tracked files committed – skipping stage. Use -m for message
`git checkout`	Get files from local repo to workdir
`git push`	Local repo content pushed to remote repo
`git fetch`	Remote repo content pulled to local repo – but not to workdir
`git merge`	Merge the local repo and the workdir – manually assisted
`git pull`	As fetch, but also merges into workdir
`git rebase`	Detach chain of commits and attach to another branch. Essentially a very clean way to merge
`git stash`	Copy workdir to a *stack* called "stash" (part of local repo but never synched remotely)
`git stash apply`	Copy back the latest workdir stashed

So how does git know which remote URL to use with which repo? In the best Linux tradition, there is a "config" file in the ".git"[3] directory. This contains the important remote URL, as well as the name of the current branch on the local and remote repo, instructions on whether to ignore case in file names (Windows) or not (Linux), merge strategy, and other stuff. Anywhere in the tree from the root of the workdir, git finds this file and is thus aware of the relevant context. This isolates the git features within the working dir, and thus allows you to rename or move the workdir without problems. It remains functional, referring to the same remote repo.

It is often stated that "git keeps the full copy of all versions." This is not entirely correct. The local repo is a full and valid repo and, therefore, can reproduce any version of a file. At the lower storage level, it stores a full copy of HEAD, the tip version. However, git runs a *pack* routine which stores deltas of text files. In this way, you get the speed of having it local, and still you have a compact system.

The problem comes with binary files. It is never recommended to store binary files in a version-control system, but it can be very practical and often is not really a problem. As with other systems, git cannot store the binary files as the usual series of deltas, each commit stores the full copy.[4] The difference from other systems lies in the full local copy of the repo. If you use, for example, Subversion, you typically checkout the *tip* (HEAD) of a given branch, and thus you only get one copy of the bi-

3 .git is the default name—you can call it something else.

4 Note that MS-Office files are considered binary.

nary files. With git you get the full chain of copies, including all versions of binary files in all branches. Even if the files are "deleted," they are still kept in order to preserve the commit history.

There are commands in git that can clean out such binary files, but it is not a single, simple operation (search for git "filter-branch"). There are also external scripts that will assist in deleting all but the newest of specific files. The actual reclaiming of space does not happen until a garbage collect is run with `git gc`. If you are using, for example, "BitBucket" as your remote repo, however, the final `git push` to this will *not* be followed by a garbage collection until the configured "safety time," usually 30 days, is passed. Instead the remote repo will take up the old space *plus* the new. If you performed a cleanup because you were close to the space limit, this is really bad news.

For this reason, you should always take the time to fill out the *.gitignore* file in the root of the workdir. It simply contains file masks of all files which git should ignore. This is a good help in avoiding the erroneous commit of a huge PNG or debug database. A nice feature of this file is that you may write, for example, "*.png" in a line, and further down write "!mylogo.png." This specifies that you generally do not want png files, except for your own logo file.

6.2.1 GitHub and other cloud solutions

Git itself is an open source tool that came with Linux, initially written by Linus Thorvalds. As stated earlier, it can be confusing and complex to use—partly because there are so many ways to do things. This challenge is to a degree handled by companies such as GitHub.

This Microsoft acquired company offers free storage to minor repos, and paid storage for the bigger ones. Nevertheless, GitHub probably owes a good part of its success to the process it surrounds git with. By using GitHub, you subscribe to a specific workflow that is known and loved by a lot of developers.[5]

Basically, any feature implementation or bugfix starts with a branch, so that the master remains untainted. Now the developer commits code until he or she is ready for some feedback. A *pull request* is issued to one or more other developers. They can now look at the code and pull it into their branch (which can even be in another repo). Comments are registered and shared.

Now the branch may be deployed to a test. If it survives the test, the branch is merged into the master.

Atlassian has a similar concept. Here, you can setup the system so that when a developer takes the responsibility for a bug created in *Jira* (see Section 6.6), a branch is automatically created. Still from within Jira, a pull request can be issued for feedback, and when a positive check-mark is received, a merge is done.

5 There are actually a few to choose between.

6.3 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, Perforce, or whatever (see Section 6.2). Typically, this only covers the source, not compiler, linker, and other binary tools, the environment settings, library folders, etc. Realizing how much time it can take to set up a build system that really works, it makes sense to keep this whole setup as a single file. This can be rolled out at any time and used, with all the above mentioned files completely unchanged. Something built today may otherwise not be able to build on a PC two years from now. Preserving the complete image of the build PC over time is one good reason to virtualize it.

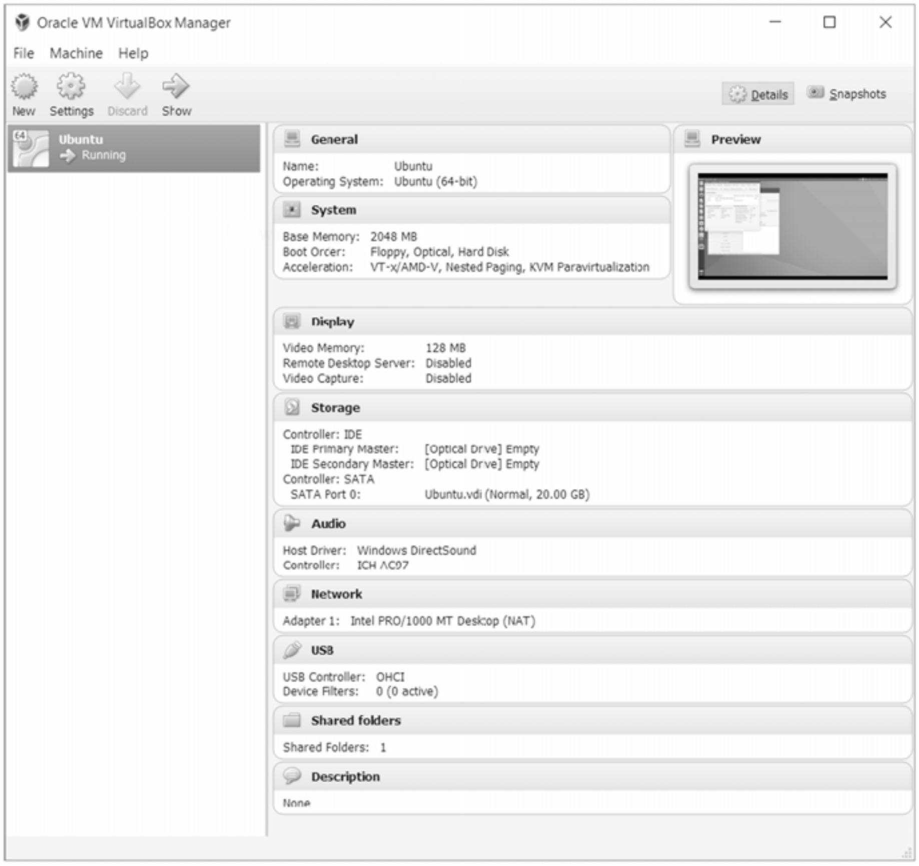
Sometimes even the developers may prefer to use such a virtual PC. The best argument against this is performance. However, if this is not something built daily but, for example, a small “coolrunner” or similar programmable hardware device, changed once in a blue moon,[6] then dusting off the virtual 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 following “manual” assumes that you run Linux as the guest OS on a Windows host.

The best known tool for Virtualization is probably VMWare, but the free Oracle VirtualBox also works great. Both VMWare and Oracle today integrate seamlessly with standard Linux distros as Ubuntu and Debian, when it comes to mouse, clipboard, access to local disk, etc. However, if you are using USB ports for serial debuggers, etc., you may find that you need to tell the virtual machine that the client Linux is allowed to “steal” the relevant port.

To get started, you download the Oracle VirtualBox or a VMWare system and install it. Next, you download an ISO image of the Linux system. Typically, there are more discs, just go for the simplest, you can always download packets for it later. From VirtualBox or VMWare, you now “mount” the ISO on the nonphysical optical/DVD drive. You can also choose the number of CPU cores and memory you want to “lend out” to the client system when it runs.

When all this is done, you start the virtual machine. Now you see Linux boot. It will ask you whether it is okay to wipe the disc. This is okay, it’s not your physical disc, just

6 A blue moon is the second full moon in a calendar month. Normally, there is two to three years between this happening, but in 2018 we had blue moon in both January and March.



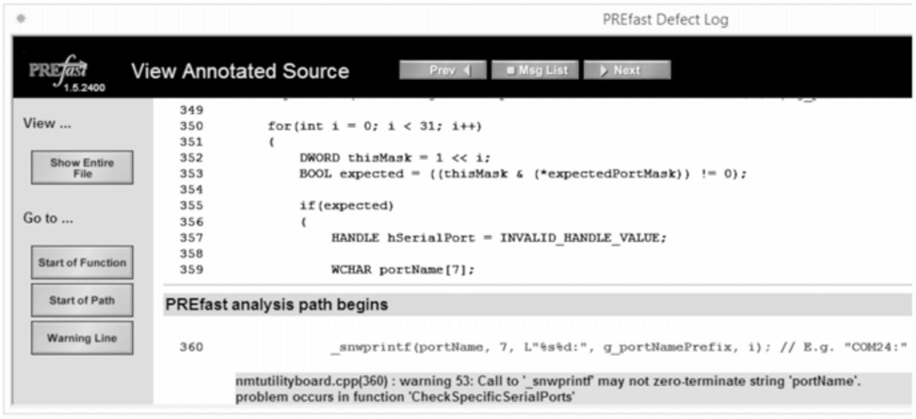
```
General
Name:              Ubuntu
Operating System:  Ubuntu (64-bit)

System
Base Memory:   2048 MB
Boot Orcer:    Floppy, Optical, Hard Disk
Acceleration:  VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Display
Video Memory:           128 MB
Remote Desktop Server:  Disabled
Video Capture:          Disabled

Storage
Controller: IDE
  IDE Primary Master:     [Optical Drive] Empty
  IDE Secondary Master:   [Optical Drive] Empty
Controller: SATA
  SATA Port 0:            Ubuntu.vdi (Normal, 20.00 GB)

Audio
Host Driver:  Windows DirectSound
Controller:   ICH AC97

Network
Adapter 1:  Intel PRO/1000 MT Deskcop (NAT)

USB
USB Controller:  OHCI
Device Filters:  0 (0 active)

Shared folders
Shared Folders:  1

Description
None
```

Figure 6.2: Oracle VirtualBox running Ubuntu.

the area you have set aside for Linux. There will also be questions on regional settings. These can be changed later, not a big deal if they are not set correctly. Finally, you can run Linux. After the initial try-outs, you probably want to go for the packet manager to install compiler, IDE and utilities. You should also remember to “unmount” the ISO disk. If not, the installation starts over next time you “power-up.” Figure 6.2 shows Ubuntu Linux running inside Oracle VirtualBox.

6.4 Static code analysis

Using static code analysis in the daily build is a good investment. 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, for example, the rules of *MISRA* (the Motor Industry Software Reliability Association). Figure 6.3 is from a tool called “PREfast” within the Windows CE environment.



```
349
350        for(int i = 0; i < 31; i++)
351        {
352            DWORD thisMask = 1 << i;
353            BOOL expected = ((thisMask & (*expectedPortMask)) != 0);
354
355            if(expected)
356            {
357                HANDLE hSerialPort = INVALID_HANDLE_VALUE;
358
359                WCHAR portName[7];
```

PREfast analysis path begins

```
360                _snwprintf(portName, 7, L"%s%d:", g_portNamePrefix, i); // E.g. "COM24:"
```

nmtutilityboard.cpp(360) : warning 53: Call to '_snwprintf' may not zero-terminate string 'portName'.
problem occurs in function 'CheckSpecificSerialPorts'

Figure 6.3: Prefast in Windows CE – courtesy Carsten Hansen.

If a tool like this is included in a project from Day 1, we naturally get more value from it. Having it on board from start also enables the team to use the tool instead of enforcing rigid coding rules. An example is when we want to test that variable a is 0 and 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 a single equals sign. We should be allowed to write fluent code and have a tool catch the random bug in the nightly build.

6.5 Inspections

Another recommendation is to use inspections, at least on selected parts of the code. Even more important, inspect all the requirements and the overall design. Even when working with agile methods, there will be something at the beginning. 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 programmers get from reading each others code. It is important 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." We say "...and then the numbers are accumulated, except the last." We don't say "...and then he accumulates the numbers, forgetting the last." Programmers are like drivers; we all believe we are among the best. Criticizing the code is one thing, but don't criticize the programmer. These are the main rules on inspections:

- Inspections are done with at least three people. They have roles: *moderator*, *reader*, *author,* and *tester*. The moderator facilitates the meeting, including preparations. He or she assures that people obey the rules during the meeting and are prepared. The reader is responsible for reading the code line-by-line, but rephrased to what it does in plain language. The author may explain along the way. In my experience, the tester role is a little vague.
- During the inspection, the moderator takes notes of all findings, with description and page- and line-number. Errors in comments and spelling of, for example, variables are noted as minor bugs.
- All participators must prepare by reading the code or spec first. This is typically handed out on print. Do not forget line numbers. It is convenient to prepare with code on the screen, allowing the inspectors to follow function-calls, header-files, etc., but to write comments on the handouts, bringing these to the meeting. Expect to use two hours to prepare and two hours to inspect 150–250 lines of code. It is possible to inspect block diagrams or even schematics in the same way.
- Have a clear scope. Typically, any code that does not cause functional errors is fine, but you may decide beforehand to inspect for performance, if this is the problem. The scope is very important to avoid religious discussions on which concept is best.
- Inspect code against a code guideline. A code guideline steers the inspection free of many futile discussions. If, however, you are using open source—adding code that will be submitted for inclusion in this open source, you need to adhere to the style of the specific code.
- End the small quality process by looking at the larger: what could we have done better? Should we change the guideline?

6.6 Tracking defects and features

Tracking bugs, aka defects, is an absolute must. Even for a single developer, it can be difficult to keep track of all 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, we 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.
- Export to Excel, or CSV (comma-separated-values), easily read into Excel.

As described in Section 6.2, when the updated code is checked into the software repository, this should be 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 we can link to any bug from an external tool. This may be mails, a wikki or even an Excel sheet. Each bug must have a unique URL.[7] This effectively shuts out tools requiring users to open a database application and enter the bug number.

Figure 6.4 shows a fairly basic "life of a bug." The bold straight lines represent the standard path, while the thinner curved lines are alternative paths.

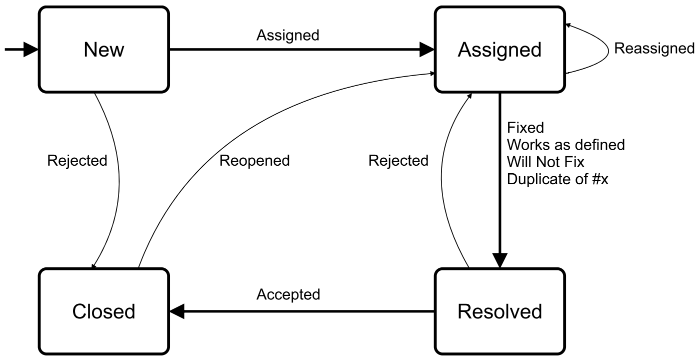


Figure 6.4: Simple defect/bug workflow.

A bug starts with a bug report in the "new" state. The team lead or project manager typically assigns the bug to a developer. The developer may realize that it "belongs" to a colleague and, therefore, reassigns it. Now the bug is fixed. Alternatively, it should not be fixed, because the code works as defined, the bug is a duplicate, or is beyond the scope of the project. In any case, it is set to "resolved." If possible, it is reassigned to the person who reported the bug, or a tester. This person may validate the decision and close the bug, or send it back to the developer.

This is not complex, but even a small project will have 100+ bugs, so we need a tool to help, and it should be easy to use.

If planned features and tasks are tracked in the same way as bugs, this can be used to create a simple forecasting tool. Figure 6.5 is an example of this.

Defects and tasks are exported from the defect-tracking tool, for example, every two weeks. Each export becomes a tab in Excel, and the sums are referenced from the main tab. "Remaining Work" (aka ToDo) is the most interesting estimates. However, "Total Work" is also interesting as it shows when the team has changed estimates or taken on new tasks. Excel is set to plot estimated man-days as a function of calendar time. Since exports from the tool does not take place at exact intervals, it is important that export dates are used as the x-axis, not a simple index.

7 One of the important rules in REST—see Section 7.12.

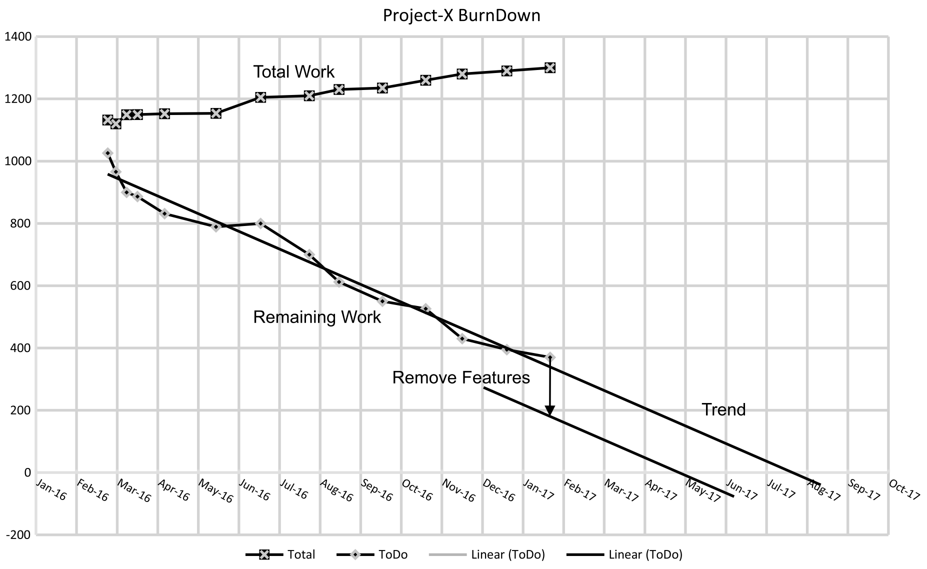


Figure 6.5: Forecasting a project—and taking action.

Excel is now told to create a trendline, which shows that the project will be done late July 2017. Now, suppose Marketing says this is a "nogo"—the product is needed in May. We can use the slope of the trendline as a measure of the project "speed," estimated man-days executed per calendar day, and draw another line with this slope, crossing the x-axis at the requested release date.[8] It is now clear for all that to get to that date, the project must cut 200 man-days worth of features. This is sometimes perfectly possible, sometimes not.

A graph of the same family is shown in Figure 6.6. It is from a small project, showing bugs created and resolved as function of time like before. This time it goes upwards and is therefore a "BurnUp chart."

The distance between the two curves is the backlog. This graph is generated directly by "Jira," a tool from the Australian company "Atlassian." It integrates with a wikki called "Confluence."

Finally, you may also create a "BurnUp Chart" based on simple time registration. Clearly, spending time does not assure that the job gets done. On the other hand, if less time than planned is spent in a project, it clearly indicates that the team is doing other things. This may be other projects competing for the same people. This is a clear sign for upper management to step in, and is the reason why we inside the project may prefer the BurnDown chart, or the first type of BurnUp chart, while it makes sense to use the latter BurnUp chart when reporting upwards.

8 Assuming that final build, tests, etc. are also estimated.

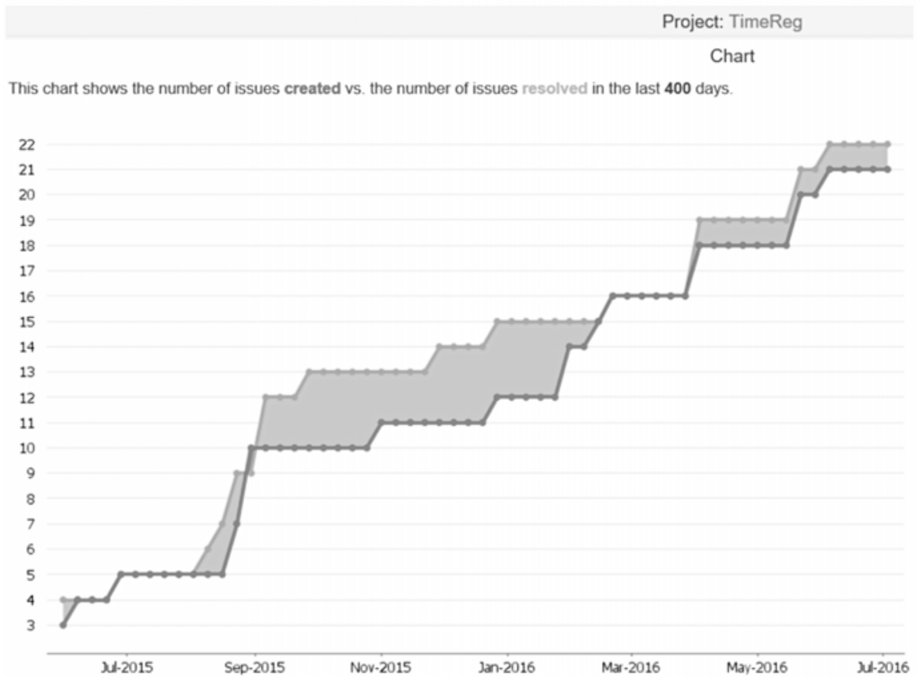


Figure 6.6: Incoming bugs versus fixed bugs in Atlassian Jira.

Another recommended tool is the open source "Trac." This can be downloaded at: 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.

6.7 Whiteboard

With all these fancy tools, let's not forget the simple whiteboard. This is one of the most important tools, almost as important as an editor and a compiler. Surely, a white-board is indispensable in the beginning of projects when regular design brainstorms are needed.

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 or similar. 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 smartphones 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.

6.8 Documentation

Documentation needed years from now, belong in, for example, Word files in a repository that can manage versions and access rights. This may even be necessary for ISO 9001 requirements, etc.

Apart from the more formal documents, most projects need a place to collaborate. As described in Section 6.6, 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.

6.9 Yocto

Yocto is a neat way to generate a customized Linux system. As discussed in Chapter 3, a modern CPU system is like a set of Russian dolls with parts from various vendors—Bluetooth Module, flash memory, Disc, SoC CPU, Ethernet MAC, USB controller, etc. There are drivers for Linux for many of these, from the vendors and from open source users. These must be combined with the best possible version of the Linux kernel, U-Boot, cross-compilers,[9] debuggers, and probably also with applications such as ssh, web server, etc.

If you buy a complete CPU board from a renown vendor, it comes with an SDK (Software Development Kit). This SDK may not contain every feature you need, even though you know it's out there, and it may not be updated as often as you would like it to be. If you create your own board, it is even worse; there is no ready-made SDK for you. This may not be so much different from the "old days" with a small kernel, but you probably chose Linux exactly to get into the ecosystem with all of these drivers, applications, and tools. The number of combinations of all these factors is astronomic, and it is a big task to assemble your special Linux installation from scratch.

On the internet, you can find many "recipes" for how to assemble such a system, always with "try version y if version x does not work," etc. So even though you can glue together something that works, the process is very nondeterministic, and even

9 Yocto generates target and tools for your development system.

a small update of a single component can trigger a redo of the whole trial-and-error process.

This is the reason for the *Yocto project*. Based on the work done in the *OpenEmbedded* organization, a number of vendors and volunteers have organized an impressive system of configuration scripts and programs, allowing you to tailor your own Linux system for any hardware. Yocto is using a layered approach where some layers pull code from official sites, while you may add other layers containing your code. The layers are not exactly architectural layers as described in Section 4.3. The top layer is the "Distro"—the distribution. Yocto supplies a reference distro—currently called "Poky," which may be used "as is," or replaced by your own. Another layer is the BSP (board-support-package) that addresses the specific hardware. Figure 6.7 shows a sample system.

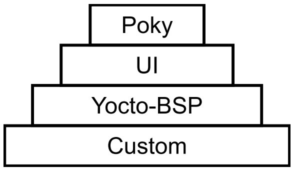


Figure 6.7: Yocto layers – sample.

Each layer may append or prepend various search paths as well as add actions to the generic workflow—mainly executed with the *bitbake* tool with the following main tasks:

– *Fetch*
  It is possible to configure your own local source mirror. This assures that you can find the software, also if it is (re)moved from a vendor's site. This saves bandwidth when many developers are working together, but also means that you control and document which open-source software is actually used. In an extreme situation, you might be accused of violating an open-source license, and in this case it is an advantage to be able to reproduce the source of the source.
– *Unpack*
  Sources may reside in a git repository, a tarball, or other formats. Naturally, the system can handle the unpacking needed.
– *Patch*
  Minor bugfixes, etc. to official software may be done until they are accepted "upstream."
– *Configure*
  The Linux kernel, busybox, and others have their own configuration programs, typically with semigraphical interfaces. The outcome of such a manual configuration can be stored in a file, and used automatically next time. Even better, the difference between the new and the default config is stored and reused.

- *Compile*
  Using the correct cross compiler with the correct basic architecture, 32/64-bit, soft or hard floating points, etc., the generic Linux source is compiled. The output is a complete image—with file system, compressed kernel image, boot-loader, etc. Yocto enforces license checks to help you avoid the traps of mixing open source with custom code.
- *QA*
  This is typically unit tests running on the build system. In order to run tests on what is typically a PC system, it is perfectly possible to compile code for a target as well as for the host itself, aka "native code." Not all target code may be run on the build machine without hard-to-create stubs, but as shown in Section 4.4, it is often possible to cut some corners and use pregenerated input data. An alternative to a native build, is to test the cross-compiled applications on the host machine using QEMU; see Section 5.1.
- *Install*
  The output from the build process is stored as needed. This may include the generation of "packages" in the various formats (RPM, DEB, or IPK), tar-balls or plain directory structures.

As great as Yocto is, it is definitely not a walk in the park. One of the most important features is therefore the ability to create an SDK. This allows some programmers to work with Yocto and low-level stuff, while others can develop embedded applications on top of the SDK, maintained by their colleagues. Alternatively, a software house that specializes in tailoring Yocto distros creates an SDK for your team.

We have a small dedicated team working on the custom hardware and basic Linux platform. They use Yocto to generate and maintain this. Yocto generates an SDK that they supply to the application team(s). This SDK contains the selected open-source kernel and drivers as well as the customized "Device Tree Overlay" and true custom low-level functionality. In this way the application programmers do not need to bother with Yocto or to experiment with drivers. They can focus on applying their domain knowledge, being productive with the more general Linux tools.

6.10 OpenWRT

Before the days of BeagleBone and Raspberry Pi, a single hobbyist developer could play around with Linux on his or her PC, but it was not easy to get your hands on embedded hardware to play with. In the 2004 August edition of *Linux Journal*, James Ewing describes the Linksys WRT54G Wi-Fi router. Through a bug, it was revealed that it runs on Linux. It was based on an advanced MIPS platform with plenty of flash space left unused. This was a very attractive platform to play with, and in the following months and years a lot of open-source software was developed.

The OpenWRT organization stems from this adventure. Here, software is developed for numerous Wi-Fi routers and other embedded Linux-based devices. End users can go here and look for open Linux firmware for their device, and developers may use it as inspiration. It is not uncommon to find better firmware here than what is supplied by the vendor of the device.

6.11 Further reading

- John Wiegly: *Git from the bottom up*
  https://jwiegley.github.io/git-from-the-bottom-up
  This is a great open-source 30-page article describing the inner-workings of git.
- Otavio Salvador and Daiane Angolini: *Embedded Linux Development with Yocto Project*
  An easy-to-read manual/cookbook on Yocto.
- trac.edgewall.org
  A free tool combining wikki, bug tracking and source control.
  Ideal for small teams.
- atlassian.com
  A larger complex of tools in the same department as “trac.”
- yoctoproject.org
  The homepage of the Yocto project with great documentation.
- openwrt.org
  The homepage of the OpenWRT Linux distribution.
- Jack Ganssle: *Better Firmware – Faster*
  Jack Ganssle has a fantastic way of combining the low-level hardware with the high-level software to create overview and efficient debugging.

Part III: **IoT technologies**

7 Networks

7.1 Introduction to the internet protocols

“The good thing about standards is that there are so many to choose from.”

The above quote is amusing because people agree that using standards is great, but nevertheless new standards keep popping up all of 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 will focus on the internet protocol stack, 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, the application protocol won’t work either. Unfortunately, a badly designed application protocol *can* slow a system down; see Section 7.20. This chapter also introduces REST in Section 7.12, an important concept used in many new protocols.

7.2 Cerf and Kahn-internet as net of nets

We honor Vinton Cerf and Robert Kahn as the inventors of the internet, which goes back to the 1960s, long before the world wide web. The simple beauty of their concept is the realization that instead of fighting over which local network is 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 7.1.

The term “protocol stack” means that a number of protocols are running on top of each other—using the layering pattern (see Section 4.3). The layers are normally numbered 1–5 from the bottom. Thanks to Cerf and Kahn, we have the “virtual” IP addresses on layer 3. The application is layer 5. Figure 7.1 here says “your code” as a typical application. However, the application layer can be very busy, consisting of

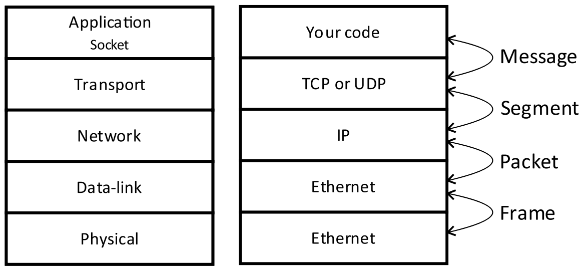


Figure 7.1: The generic layers, typical variant and PDU names.

https://doi.org/10.1515/9781547401024-007

many layers by itself. In a web server, you will find HTTP as one of the lower layers in the application, but nevertheless in layer 5 in this model. Devices that run applications that are communicating are called “hosts.” It doesn’t matter whether they act as client or server at the application layer.

Typically, we have the 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; it does not change even though the device, for example, 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. Today, in most cases these are Ethernet MAC addresses. Devices communicating on layer 2 are called “nodes.” A PC and a smartphone are both nodes and hosts. A router remains a node only, until we start communicating directly to an embedded web server inside it. Then it also becomes a host.

7.3 Life of a packet

In this section, we follow a packet from a web browser to a web server. The routing described, is no different from the communication between a cloud server and an embedded device in the field. The term “packet” is not precise, but very common. As shown in Figure 7.1, the various layers each have their own name for the packet, as it grows on its way down through the stack, while each layer adds a header, and in case of the Ethernet layer also a tail. The correct generic term for a “packet” on any layer is *PDU* (protocol data unit).

Figure 7.2 can be seen as a generic setup, demonstrating hosts, switches and routers. It can also be seen as very specific. If you have 4 to 6 ports on the switch and put it in the same box as a two-port router, you have a standard SOHO (small office home office) “router.” The LAN connecting the switch to the host might even be wireless. This is actually the setup used for the capture we will dig into now. The relevant interfaces are marked with circles. In Chapter 9 we will look more into the differences between wired and wireless LAN, but in this chapter there is no difference. 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, using the router’s MAC address. In the case of Figure 7.2, the router interface towards the LAN has IPv4 address 192.168.0.1 and MAC address 00:18:e7:8b:ee:b6.

Figure 7.2 shows the web browser’s request from a web client to a web server (from right to left). These are both termed hosts, because 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 messages seem to go from application to application, even though they in reality pass down the stack at the sender side and up at the receiver side. Likewise, the two TCP

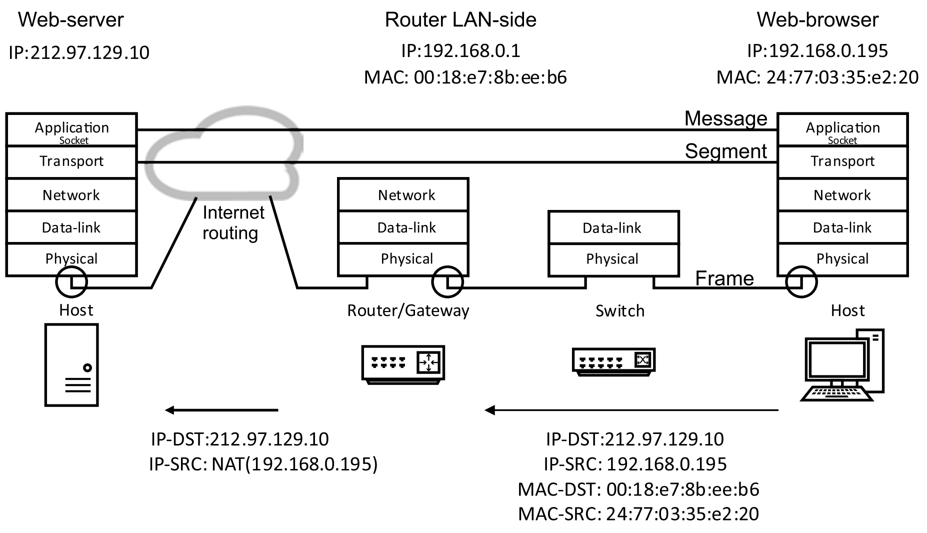


Figure 7.2: Internet with main network devices.

processes seem to communicate segments. The addresses of the interfaces are noted at the top. The packet follows the solid line, and is “stored and forwarded” by the switch as well as by the router/gateway. On the other side of the router we could have more LAN, but in this case we move into the actual internet. The web server is also connected to a gateway, but this is not shown. The router has its hands on several things in the packet from the client:

– The network-layer source containing the client’s IP address.
  IP addresses remain unchanged for the lifetime of the packet—*unless* the router contains a NAT, in which case the source IP will be changed when going towards the Internet—see Section 7.9. Most SOHO routers does contain a NAT, and this is no exception.
– The network-layer destination containing the web server’s IP address.
  IP addresses remain unchanged for the lifetime of the packet—*unless* the router contains a NAT, in which case the destination IP will be changed when going towards the local LAN—see Section 7.9. Our packet going from left to right will thus keep its destination IP address, but the answer coming back from right to left will be changed here. Generally a router looks up the remote IP address in its router table in order to decide which interface to transmit the package onto. In the case of a SOHO router there is only one choice.
– The link-layer source containing the client’s MAC address.
  If the router is connected to LAN on the outgoing interface, the source MAC address is replaced with the router’s own MAC address on this interface, as it is now the new link-layer source. In our case the router is an Internet gateway and thus uses Internet routing, which is not the subject of this book.

- The link-layer destination containing the routers MAC address.
  If the router is connected to LAN on the outgoing interface, the destination MAC address is replaced with the web server's MAC address—or the next router's MAC address—whichever comes first. In our case the router is an Internet gateway and thus uses Internet routing, which is not the subject of this book.
- The hop count.
  If IPv4 is used, the so-called "hop-count" is decremented. When it reaches zero, the packet is thrown away. This is to stop packets from circling forever.
- The checksum.
  Changing hop count or an IP address (due to NAT) introduces the need to change the IPv4 checksum.

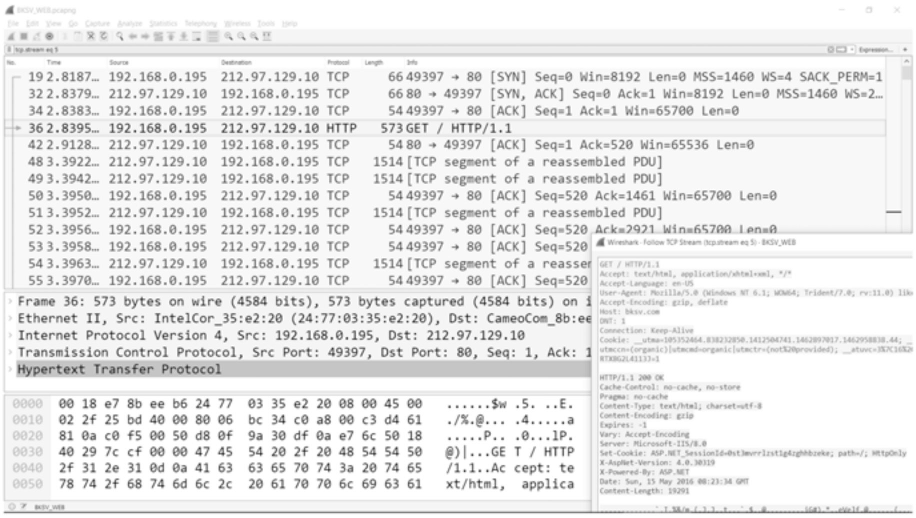
The bottom of Figure 7.2 shows the source and destination addresses in the packet, as it moves from right to left. The data is a simple GET request. IP4 addresses are used for simplicity, and the 192.168.x.y address is a typical NAT address; see Section 7.9.

The switch is completely transparent, changing nothing in the packet. The basic switch has no IP or MAC address. 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 "clocked out" on another port. This is actually a little confusing. One 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 configuration.

When the web server responds to the request from the client, it simply swaps sources with destinations on both levels, and the whole thing is repeated, now going from left to right. It actually swaps one more source/destination pair: the TCP ports. These are not mentioned above, as the router doesn't care about them. They are on a layer higher than what a router understands (also here an exception is made for the NAT case). 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 7.3 shows the scenario from Figure 7.2, 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 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 disappear due to the "stream" nature of TCP. A TCP segment may theoretically be up to 64 kBytes. However, TCP has a "Maximum Segment Size" parameter. This 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.



```
 19 2.8187… 192.168.0.195  212.97.129.10  TCP     66 49397 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 32 2.8379… 212.97.129.10  192.168.0.195  TCP     66 80 → 49397 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=2…
 34 2.8383… 192.168.0.195  212.97.129.10  TCP     54 49397 → 80 [ACK] Seq=1 Ack=1 Win=65700 Len=0
 36 2.8395… 192.168.0.195  212.97.129.10  HTTP   573 GET / HTTP/1.1
 42 2.9128… 212.97.129.10  192.168.0.195  TCP     54 80 → 49397 [ACK] Seq=1 Ack=520 Win=65536 Len=0
 48 3.3922… 212.97.129.10  192.168.0.195  TCP   1514 [TCP segment of a reassembled PDU]
 49 3.3942… 212.97.129.10  192.168.0.195  TCP   1514 [TCP segment of a reassembled PDU]
 50 3.3950… 192.168.0.195  212.97.129.10  TCP     54 49397 → 80 [ACK] Seq=520 Ack=1461 Win=65700 Len=0
 51 3.3952… 212.97.129.10  192.168.0.195  TCP   1514 [TCP segment of a reassembled PDU]
 52 3.3956… 192.168.0.195  212.97.129.10  TCP     54 49397 → 80 [ACK] Seq=520 Ack=2921 Win=65700 Len=0
 53 3.3958… 192.168.0.195  212.97.129.10  TCP     54 49397 → 80 [ACK] Seq=520
 54 3.3963… 212.97.129.10  192.168.0.195  TCP   1514 [TCP segment of a reasser
 55 3.3970… 192.168.0.195  212.97.129.10  TCP     54 49397 → 80 [ACK] Seq=520

> Frame 36: 573 bytes on wire (4584 bits), 573 bytes captured (4584 bits) on i
> Ethernet II, Src: IntelCor_35:e2:20 (24:77:03:35:e2:20), Dst: CameoCom_8b:ee
> Internet Protocol Version 4, Src: 192.168.0.195, Dst: 212.97.129.10
> Transmission Control Protocol, Src Port: 49397, Dst Port: 80, Seq: 1, Ack: 1
> Hypertext Transfer Protocol

0000  00 18 e7 8b ee b6 24 77  03 35 e2 20 08 00 45 00   ......$w .5. ..E.
0010  02 2f 25 bd 40 00 80 06  bc 34 c0 a8 00 c3 d4 61   ./%.@... .4.....a
0020  81 0a c0 f5 00 50 d8 0f  9a 30 df 0a e7 6c 50 18   .....P.. .0...lP.
0030  40 29 7c cf 00 00 47 45  54 20 2f 20 48 54 54 50   @)|...GE T / HTTP
0040  2f 31 2e 31 0d 0a 41 63  63 65 70 74 3a 20 74 65   /1.1..Ac cept: te
0050  78 74 2f 68 74 6d 6c 2c  20 61 70 70 6c 69 63 61   xt/html,  applica
```

```
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) lik
Accept-Encoding: gzip, deflate
Host: bksv.com
DNT: 1
Connection: Keep-Alive
Cookie: __utma=105352464.838232850.1412504741.1462897017.1462958838.44; _
utmccn=(organic)|utmcmd=organic|utmctr=(not%20provided); __atuvc=3%7C16%2
RTX8G2L4113J=1

HTTP/1.1 200 OK
Cache-Control: no-cache, no-store
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/8.0
Set-Cookie: ASP.NET_SessionId=0st3mvrrlzstlg4zghhbzeke; path=/; HttpOnly
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Sun, 15 May 2016 08:23:34 GMT
Content-Length: 19291
```

BKSV_WEB.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 5
Expression…
No. Time Source Destination Protocol Length Info
Wireshark · Follow TCP Stream (tcp.stream eq 5) · BKSV_WEB
BKSV_WEB

Figure 7.3: Transmission with HTTP using "follow TCP-Stream."

We only see the relevant conversation in Figure 7.3, which is why many frame numbers are missing in the left column in the top window. The filter for the conversation 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 "Protocol" field with relevant data from the "highest protocol" it knows, in this case HTTP. You can sort on this column by clicking the header, and thus quickly find a good place to start analyzing.

Thus finding the "GET" request was simple. Another result of the "Follow TCP-stream," is the overlaid window 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 we see the full conversation, not just the selected frame. Thus the dialog includes the following frames. In the info-field, these 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 nonexpanded 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 ranges, Wireshark often recognizes the leftmost 3 bytes, and inserts a vendor name in one of the two versions of the same MAC address it writes. This makes it easier to guess which device you are actually looking at. In this case, the client is a PC, with an Intel MAC.

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 TCP connection. You can start a Wireshark capture at any time during a transmission, but it's best to catch the three-way handshake first—it contains some important information, as we shall soon see.

Let's get some terms right: the *TCP client* is the host that sends 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 and 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 connection 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, 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 connection, application data may (and typically will) flow in both directions. It is up to the application layer to manage this. However, let's see what we may learn from the initial handshake, via the "info" field on frame 19 in Wireshark in Figure 7.3:

– *SYN Flag*
  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 Flag* [not in frame 19]
  Contained in all frames except the client's opening SYN. This allows you to tell the TCP client from the TCP server.
– *Seq=0*
  The 32-bit *sequence number* of the first byte in the segment sent. Many protocols number their packets, but TCP numbers its bytes. This allows for smarter retransmissions—concatenating smaller packets that were initially sent one-by-one as they were handed to TCP from the application layer. Note that the SYN flag and the closing FIN flag (outside the screen), count as bytes in this sense. Because we got the transmission from the initial three-way-handshake, Wireshark is nice, giving us relative sequence numbers (starting from 0). If you open the bottom hexadecimal view, you will see that sequence numbers do not start from 0. In fact, this is an important part of the security. Sequence numbers are unsigned and simply wrap.
– *Ack* [not in frame 19]
  The sequence number that this client or server expects to see next from the other side. As either side will sometimes send ACKs 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*
  This 16-bit *window size*, used by both sides, tells the other side how much space it currently has in its receive buffer. Thus the given sender knows when to stop trans-

mitting, 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 exactly once, without gaps, and in order, this buffer may easily become full if an early packet is 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*
  This is the *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* factor. This is an option in the opening three-way-handshake. If the client uses this option, the window scale 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 acknowledgment that the client's suggestion is accepted. Furthermore, the window scale sent from the server, is the scale it will use in its window size. Thus the two window scales 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*
  SACK_PERM means *selective acknowledge permitted*. The original implementation of TCP is rather simple. The ACK number can be understood as "This is how much data I have received from you without gaps." The transmitter may realize it has sent something, not seen by the receiver, and it will retransmit all bytes from this number and forward. However, with the "big-fat-pipes," the sender may need to resend a lot of data already caught by the receiver. With SACK_PERM, the host says 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 first not-seen 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. This means that less data is retransmitted, saving time at both ends, as well as utilizing the network better.
– *MSS=1460*
  MSS means *maximum segment size* and is the maximum size of the payload in the TCP segment (the data from the application) in bytes. This is related to the MTU; see Section 7.17.

If segments longer than MSS are sent, they may get there, but the overhead will affect performance. This is where packets on the IP[1] layer become “fragmented” into more frames in the Ethernet layer, as discussed in Section 7.14.

– *49397->80*

  These are the port numbers used with the sender’s first.

Amazing how much can be learned from studying the Wireshark “info” on a single frame. Wireshark shows us the TCP connection as well as its initiation and end. All this is clearly visible on the network between two hosts—and known by both ends.

A “TCP socket” is an OS construction offered to the application programmer, handling the TCP details on one end of a TCP connection. In Chapter 2, we saw a central part of a state-machine implementation of a socket. The socket does the book-keeping on sequence-numbers, the numbers of ACKs received, time left before next timeout and much more.

In our case, the client’s port number was 49397. This is the *ephemeral* port number—a random number selected by the operating system. It is a part of the security on plain connections (non-SSL) that this port number really *is* random. The socket is registered in the OS by the two IP addresses, the two port numbers and the TCP protocol. Together these five numbers constitute a *5-tuple*. Whenever the OS receives a packet, this 5-tuple is extracted from the packet and used to lookup the correct socket in the OS. The communicating hosts network interfaces dictate the IP addresses, and as the server is a public web server, the protocol is TCP and the server’s port number is 80. The client port number is thus the only of these five numbers not “written in stone.” If nobody knows the next port number, it makes the system somewhat less sensitive to “injection attacks,” where bandits send fake packets.

A real and a fake packet only differ in the payload part—the data from the application layer. The first packet received “wins,” and the other one is thrown away as an unneeded retransmission. A bandit on the network path might for example try to reroute the client to another website. If the client port number can be guessed, the bandit has good time to generate the fake packet and win the race. Clearly, this is not rock-solid security, but a small part of the picture. We will meet the much safer SSL in Section 10.12.

7.4 Life before the packet

When for example, 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[2] need to communicate via their MAC addresses. Typically, the PC knows the MAC address of the gateway router from the original DHCP, but what about other hosts?

1 This only happens in version 4 of IP, not in version 6.

2 We use the term “host” on the application layer, but devices communicating on the Ethernet layer are called “nodes.”

This is where ARP (address resolution protocol) comes into play. A host keeps a list of corresponding IP and MAC addresses, known as an ARP cache. If the IP address of the destination is not in the ARP cache, the host will issue an ARP request, effectively shouting: “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 7.1 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 marked “static” are all (except one) in the IP range from 224.x.y.z to 239.x.y.z. This is a special range reserved for “multicasting,” and this is emphasized by their MAC addresses, all starting with 01:00:5e and ending with a bit pattern, equal to the last part of the IP address.[3] 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 7.1 there is no entry for 192.168.0.198.

Listing 7.1: ARP cache with router at top

```
C:\Users\kelk>arp -a

Interface: 192.168.0.195 --- 0xe
  Internet Address      Physical Address      Type
  192.168.0.1           00-18-e7-8b-ee-b6     dynamic
  192.168.0.193         00-0e-58-a6-6c-8a     dynamic
  192.168.0.194         00-0e-58-dd-bf-36     dynamic
  192.168.0.196         00-0e-58-f1-f3-f0     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.0.1           01-00-5e-7f-00-01     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Listing 7.2 shows an ICMP “ping” request, the low level “are you there?” to the IP address 192.168.0.198, followed by a display of the ARP cache (in the same listing), now with an entry for 192.168.0.198.

Listing 7.2: ARP cache with new entry

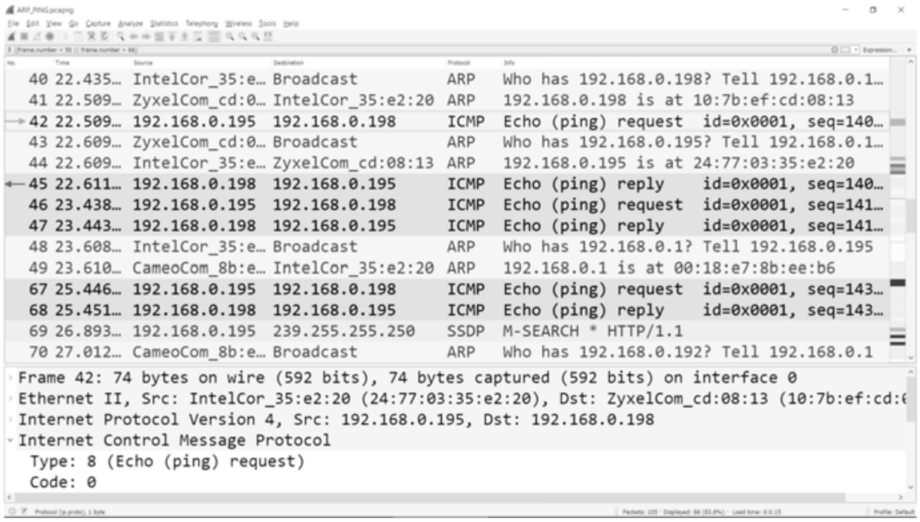
```
C:\Users\kelk>ping 192.168.0.198

Pinging 192.168.0.198 with 32 bytes of data:
Reply from 192.168.0.198: bytes=32 time=177ms TTL=128
Reply from 192.168.0.198: bytes=32 time=4ms TTL=128
```

3 This is not so easy to spot as IPv4 addresses are given in decimal and MAC addresses in hexadecimal.

```
Reply from 192.168.0.198: bytes=32 time=3ms TTL=128
Reply from 192.168.0.198: bytes=32 time=5ms TTL=128

Ping statistics for 192.168.0.198:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum = 177ms, Average = 47ms

C:\Users\kelk>arp -a

Interface: 192.168.0.195 --- 0xe
  Internet Address      Physical Address      Type
  192.168.0.1           00-18-e7-8b-ee-b6     dynamic
  192.168.0.193         00-0e-58-a6-6c-8a     dynamic
  192.168.0.194         00-0e-58-dd-bf-36     dynamic
  192.168.0.196         00-0e-58-f1-f3-f0     dynamic
  192.168.0.198         10-7b-ef-cd-08-13     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.0.1           01-00-5e-7f-00-01     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Figure 7.4 shows a Wireshark capture of the above 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



```
40 22.435… IntelCor_35:e… Broadcast          ARP   Who has 192.168.0.198? Tell 192.168.0.1…
41 22.509… ZyxelCom_cd:0… IntelCor_35:e2:20  ARP   192.168.0.198 is at 10:7b:ef:cd:08:13
42 22.509… 192.168.0.195  192.168.0.198      ICMP  Echo (ping) request  id=0x0001, seq=140…
43 22.609… ZyxelCom_cd:0… Broadcast          ARP   Who has 192.168.0.195? Tell 192.168.0.1…
44 22.609… IntelCor_35:e… ZyxelCom_cd:08:13  ARP   192.168.0.195 is at 24:77:03:35:e2:20
45 22.611… 192.168.0.198  192.168.0.195      ICMP  Echo (ping) reply    id=0x0001, seq=140…
46 23.438… 192.168.0.195  192.168.0.198      ICMP  Echo (ping) request  id=0x0001, seq=141…
47 23.443… 192.168.0.198  192.168.0.195      ICMP  Echo (ping) reply    id=0x0001, seq=141…
48 23.608… IntelCor_35:e… Broadcast          ARP   Who has 192.168.0.1? Tell 192.168.0.195
49 23.610… CameoCom_8b:e… IntelCor_35:e2:20  ARP   192.168.0.1 is at 00:18:e7:8b:ee:b6
67 25.446… 192.168.0.195  192.168.0.198      ICMP  Echo (ping) request  id=0x0001, seq=143…
68 25.451… 192.168.0.198  192.168.0.195      ICMP  Echo (ping) reply    id=0x0001, seq=143…
69 26.893… 192.168.0.195  239.255.255.250    SSDP  M-SEARCH * HTTP/1.1
70 27.012… CameoCom_8b:e… Broadcast          ARP   Who has 192.168.0.192? Tell 192.168.0.1

Frame 42: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: IntelCor_35:e2:20 (24:77:03:35:e2:20), Dst: ZyxelCom_cd:08:13 (10:7b:ef:cd:
Internet Protocol Version 4, Src: 192.168.0.195, Dst: 192.168.0.198
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
```

Figure 7.4: A ping that provokes ARP.

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 MACs). 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.[4]

Here, we see an explanation for the extended time for the first ping. The responding Windows 7 PC, decided to do its own ARP, 100 ms after having answered the original ARP. This is a defense against "ARP poisoning": If we see more than one device answering this second ARP, it is a sign that someone might be trying to take over one side of the communication. 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.

7.5 Getting an IP address

Network interfaces are born with a unique 48-bit MAC address. This is like a person's social security number, following the interface for life. Not so with IP addresses. IP addresses are hierarchical to facilitate routing. This is similar to letters addressed to <country>-<ZIP>-<street and number>-<floor>. This means that IP addresses may change. With mobile equipment they change all the time. 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 ready from power-up and does not change. Many systems use static addresses in one of the private IP ranges (see Section 7.8) in "islands" not connected to the internet, but to a dedicated network interface card (NIC) on a PC.
  Like many others, my company provides large systems with racks, and modules that go into slots in the racks, see Section 4.9. In our case, each module is IP addressable. I introduced a "best practice" addressing scheme:
  192.168.<rackno>.<slotno>, using network mask 255.255.0.0.
  This makes it easy to find a module from its address.
– *DHCP*
  Dynamic Host Configuration Protocol. This is practical with laptops and phones moving 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 fix the DHCP address handed out to a device, by linking it to the MAC address of the device. This is very practical as your PC, phone or whatever, can remain a

4 Unicasts, multicasts and broadcasts are explained in Section 7.18.

DHCP client wherever you go, and still you can be sure to always have the same IP address in your home. This is practical for, for example, development servers on a laptop.

- *Link Local*
  When a device is setup as a DHCP client, but cannot "see" a DHCP server, it waits some time and finally selects an address in the range 169.254.x.y. First, it tests that the relevant candidate address is free, using ARP, and then it announces the claim with a "Gratuitous ARP." See Section 7.4. This is a nice fall-back solution, allowing two hosts configured as DHCP clients, to communicate even without a DHCP server—*if* you can get past all the security in firewalls and other security devices. Microsoft calls link-local for "Auto-IP."
  Should the DHCP server become visible after a link-local address is selected, the device will change IP address to the one given from the server. This can be very upsetting.
- *Link-Local IP6*
  IPv6 includes another version of link-local that makes the address unique, based on the MAC address on the interface.

7.6 DHCP

Listing 7.3 shows how the DHCP process is easily provoked on a Windows PC.

Listing 7.3: DHCP release and renew

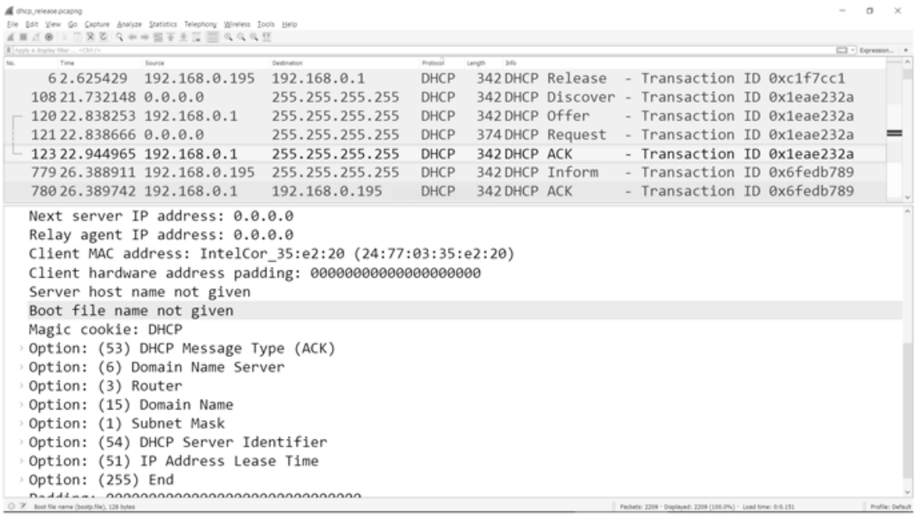
```
C:\Users\kelk>ipconfig /release
...skipped...
C:\Users\kelk>ipconfig /renew
```

Figure 7.5 shows the Wireshark capture corresponding to Listing 7.3. 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 secondarily by frame number. Frame by frame, we see:

- Frame 6: "DHCP Release" from the PC caused by line 1 in Listing 7.3. Here, the PC has the same address as before: 192.168.0.195. After this frame, it is dropped.
- Frame 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 in frame 6.
- Frame 120: "DHCP Offer" from the server. This is sent as a 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. It cannot be received by mistake by another client. The offer contains the future IP address

and mask, as well as the IP address of the DNS server (see Section 7.10) and the gateway router. It also includes the “lease time,” which is how long the PC may wait before it asks for a renewal.

- Frame 121: “DHCP Request” from the PC. Why, we just had the whole thing handed on a silver plate. The explanation is that another DHCP server might simultaneously have offered us an address. 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 PCs. This explains why my home router identifies my company laptop with my company’s domain, even though I am currently not on this domain.
- Frame 123: “DHCP ACK” from the server. This means that finally, the deal is sealed.
- Frame 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 passed with DHCP (DNS, etc.). Also according to IETF, this has been seen to be kind of misused. This appears to be one such case.
- Frame 780: “DHCP ACK” from the server. Answer to the above.


dhcp_release.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

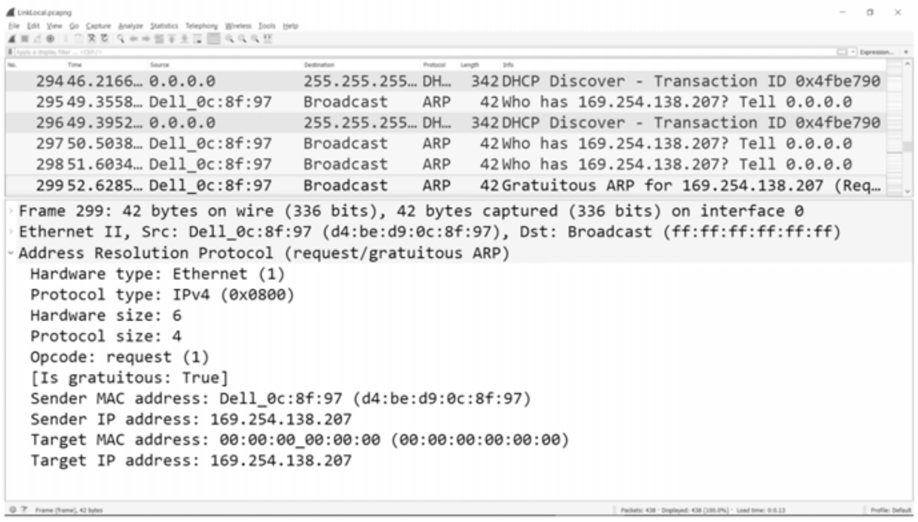
```
No.   Time      Source          Destination      Protocol Length Info
    6 2.625429  192.168.0.195   192.168.0.1      DHCP     342 DHCP Release  - Transaction ID 0xc1f7cc1
  108 21.732148 0.0.0.0         255.255.255.255  DHCP     342 DHCP Discover - Transaction ID 0x1eae232a
  120 22.838253 192.168.0.1     255.255.255.255  DHCP     342 DHCP Offer    - Transaction ID 0x1eae232a
  121 22.838666 0.0.0.0         255.255.255.255  DHCP     374 DHCP Request  - Transaction ID 0x1eae232a
  123 22.944965 192.168.0.1     255.255.255.255  DHCP     342 DHCP ACK      - Transaction ID 0x1eae232a
  779 26.388911 192.168.0.195   255.255.255.255  DHCP     342 DHCP Inform   - Transaction ID 0x6fedb789
  780 26.389742 192.168.0.1     192.168.0.195    DHCP     342 DHCP ACK      - Transaction ID 0x6fedb789
```

```
  Next server IP address: 0.0.0.0
  Relay agent IP address: 0.0.0.0
  Client MAC address: IntelCor_35:e2:20 (24:77:03:35:e2:20)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
> Option: (53) DHCP Message Type (ACK)
> Option: (6) Domain Name Server
> Option: (3) Router
> Option: (15) Domain Name
> Option: (1) Subnet Mask
> Option: (54) DHCP Server Identifier
> Option: (51) IP Address Lease Time
> Option: (255) End
```

Boot file name (bootp.file), 128 bytes
Packets: 2209 · Displayed: 2209 (100.0%) · Load time: 0:0.151
Profile: Default

Figure 7.5: DHCP commands and options.

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. In this case, they are unicast. In other words, the PC does not “forget” its address but uses it while it is renewed. If you would like to try some experiments, Section 5.9 contains a Python script for simulating the client side of DHCP.



```
No.  Time        Source           Destination     Protocol Length Info
294  46.2166…    0.0.0.0          255.255.255…    DH…      342    DHCP Discover - Transaction ID 0x4fbe790
295  49.3558…    Dell_0c:8f:97    Broadcast       ARP       42    Who has 169.254.138.207? Tell 0.0.0.0
296  49.3952…    0.0.0.0          255.255.255…    DH…      342    DHCP Discover - Transaction ID 0x4fbe790
297  50.5038…    Dell_0c:8f:97    Broadcast       ARP       42    Who has 169.254.138.207? Tell 0.0.0.0
298  51.6034…    Dell_0c:8f:97    Broadcast       ARP       42    Who has 169.254.138.207? Tell 0.0.0.0
299  52.6285…    Dell_0c:8f:97    Broadcast       ARP       42    Gratuitous ARP for 169.254.138.207 (Req…

Frame 299: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
Ethernet II, Src: Dell_0c:8f:97 (d4:be:d9:0c:8f:97), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request/gratuitous ARP)
  Hardware type: Ethernet (1)
  Protocol type: IPv4 (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)
  [Is gratuitous: True]
  Sender MAC address: Dell_0c:8f:97 (d4:be:d9:0c:8f:97)
  Sender IP address: 169.254.138.207
  Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Target IP address: 169.254.138.207
```

Figure 7.6: Link-local with gratuitous ARP.

Figure 7.6 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 instead informing its surroundings that the address is about to be taken—unless someone answers to it. 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 it has it. Hence the term "gratuitous."

7.7 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. However, when it comes to the term "subnet" it gets more fluffy. It might help to do a backward definition and say that a subnet is the network island you have behind a router. As discussed in Section 7.3, 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, defining the network ID, this being the subnet. The last 8 bits are what separates the hosts from each other—their host ID. Since we do not use addresses ending with "0", and the host ID address with all bits set is for broadcasts within the subnet, this leaves 254 possible host IDs 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 corporations were given a class B range to use at will. While everybody was waiting for IPv6 to help us out, CIDR and NAT were 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 bits of the given address is the subnet, and thus defines the subnet mask as the leftmost n bits, leaving 32 minus n bits 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 it quickly becomes complicated, which is probably why the old masks are die hards. There are however many apps and home-pages offering assistance with this.

7.8 Reserved IP ranges

We have seen addresses like 192.168.0.x numerous times. Now, let us take a look at what is known as "reserved IP ranges" (Table 7.1). Packets destined for an IP address in a "private range" will not get past a router. This makes them perfect for the home or company network, as they are not blocking existing global addresses. They 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 the ones shown in Table 7.1 are the most relevant.

Table 7.1: The most important reserved IP address ranges.

CIDR	Range	Usage
10.0.0.0/8	10.0.0.0–10.255.255.255	Private
169.254.1.0/16	169.254.1.0–169.254.255.255	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	Subnet only

7.9 NAT

NAT (network address translation) has been extremely successful, at least measured by how it has saved IP addresses. Most households need to be able to act as TCP clients against a lot of different servers, but very few households have a server which others need to be client against. These households 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 PCs, 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)—its source port number being 4603. The NAT opens the packet and swaps the internal IP address with the external. It also swaps the source port number with something of its own choice. In this case, port 9000; see Table 7.2. This scheme allows multiple internal IP addresses to be mapped to one external IP address by using some of the port number space. The packet moves to the web server where the request is answered and source and destinations of ports and IP addresses are swapped as usual. An answer is routed back to the external address where the NAT switches the destination IP and port back to the original (private) source IP and port. 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 in this case.

Table 7.2: Sample NAT table.

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

Table 7.2 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 answered. 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.

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, the NAT can throw away any segment that hasn't got the "ACK" bit set. Such a segment can only be a client opening "SYN." As the installation has no servers, we disallow all SYNs from the outside; see Section 7.13.

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 7.12. The mappings shown in the table are *dynamic*, as they are generated from traffic. It is possible to create static mappings. If we do want a PC on the intranet to act as a company/household web server, the static mapping will send all packets towards port 80 to this PC. This way we are able to have a server inside our network. In this case, we now *will* prefer that the external IP address is fixed, 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 4.3. If an application embeds information about ports and IP addresses in the application protocol, this is not translated, and will cause problems.

7.10 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 scaling ability assistance. Listing 7.4 was created on a Windows 10 PC.

Listing 7.4: Simple nslookup

```
C:\Users\kelk>nslookup google.com
Server:  UnKnown
Address:  192.168.0.1

Non-authoritative answer:
Name:    google.com
Addresses:  2a00:1450:4005:80a::200e
          195.249.145.118
          195.249.145.114
          195.249.145.98
          195.249.145.88
          195.249.145.99
          195.249.145.103
          195.249.145.113
          195.249.145.109
          195.249.145.119
          195.249.145.84
          195.249.145.123
          195.249.145.93
          195.249.145.94
          195.249.145.89
          195.249.145.108
          195.249.145.104
```

When looking up google.com, we get 16 IPv4 answers (and one answer with an IPv6 address). A small server park could use this to have 16 different servers, with separate IP addresses for the same URL. Most programmers are lazy and are simply using the first answer, and for this reason the next call will show that the answer is “cycled.” This is a simple way to “spread out” clients to different physical servers. 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.

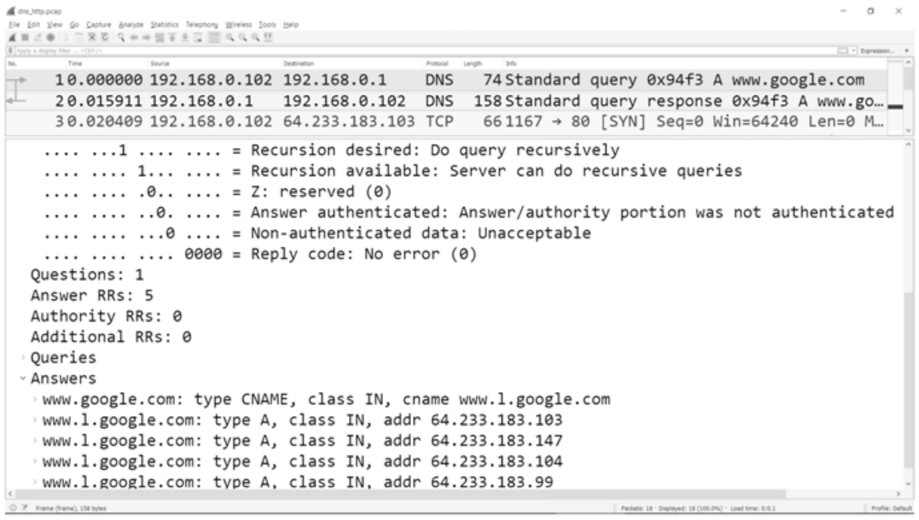
In order for a URL to be translated to an IP address by the DNS, it needs to be registered. This can be done via many organizations, as a simple search will show. Once registered, the URL and the belonging IP addresses are programmed into the 13 root DNS servers. When your program asks its local DNS 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” (see Section 7.18), and these requests are so small they can easily fit into a single UDP segment—no matter which protocol is beneath. DNS requests come in two forms, *recursive* or *iterative*. 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 iterative case, the local resolver must ask the next DNS server in the hierarchy itself, using an IP address for this which it got in the first answer, etc.

Figure 7.7 is a Wireshark caption of an older “googling,” with some completely different answers (compared to earlier).

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. In frame 3, we see the web browser 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.

Many smaller embedded systems operate directly on IP addresses, making the whole DNS concept irrelevant. A typical IoT system will have its cloud server registered in the DNS but normally not the devices. Alternatively, *mDNS* (multicast DNS) is used. This is used in *Bonjour* invented by Apple, and the Linux implementation of the same—called *Avahi*.



```
1 0.000000 192.168.0.102 192.168.0.1    DNS  74 Standard query 0x94f3 A www.google.com
2 0.015911 192.168.0.1   192.168.0.102  DNS 158 Standard query response 0x94f3 A www.go…
3 0.020409 192.168.0.102 64.233.183.103 TCP  66 1167 → 80 [SYN] Seq=0 Win=64240 Len=0 M…

  .... ...1 .... .... = Recursion desired: Do query recursively
  .... .... 1... .... = Recursion available: Server can do recursive queries
  .... .... .0.. .... = Z: reserved (0)
  .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated
  .... .... ...0 .... = Non-authenticated data: Unacceptable
  .... .... .... 0000 = Reply code: No error (0)
 Questions: 1
 Answer RRs: 5
 Authority RRs: 0
 Additional RRs: 0
› Queries
⌄ Answers
  › www.google.com: type CNAME, class IN, cname www.l.google.com
  › www.l.google.com: type A, class IN, addr 64.233.183.103
  › www.l.google.com: type A, class IN, addr 64.233.183.147
  › www.l.google.com: type A, class IN, addr 64.233.183.104
  › www.l.google.com: type A, class IN, addr 64.233.183.99
```

Figure 7.7: A DNS request for google.com.

7.11 Introducing HTTP

Telnet is not used very much anymore; there is absolutely no security in it. Nevertheless, it is great for showing a few basics on HTTP. Listing 7.5 shows a telnet session from a Linux PC. In modern Windows, Telnet is hidden. You need to enable it via “Control Panel”—“Programs and Features”—“Turn Windows features on or off.”

Listing 7.5: Telnet as webbrowser

```
kelk@debianBK:~$ telnet www.bksv.com 80
Trying 212.97.129.10...
Connected to www.bksv.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.bksv.com

HTTP/1.1 200 OK
Cache-Control: no-cache, no-store
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
Set-Cookie: ASP.NET_SessionId=nvyxdt5svi5x0fovm5ibxykq;
            path=/; HttpOnly
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Sun, 15 May 2016 11:46:29 GMT
```

```
19 Content-Length: 95603
20
21
22 <!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN"
23         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
24 <html id="htmlStart" xmlns="http://www.w3.org/1999/xhtml"
25           lang="en" xml:lang="en">
26 <head><META http-equiv="Content-Type" content="text/html;
27 ␣␣␣␣␣␣␣charset=utf-8">
28 <title>Home - Bruel &amp; Kjaer</title>
29 <meta http-equiv="X-UA-Compatible" content="IE=edge">
30 <meta name="DC.Title" content="......."
```

We are now basically doing exactly the same thing as was done before with a browser: a 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 the server, we can send the same lines we can see in the Wireshark communication in clear text in Figure 7.3. However, this time only the two mandatory headers are sent; see Listing 7.5, 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 substituted with an IP address before leaving the PC at all, thanks to the DNS. A modern web server installation can host many "sites" with different URLs, but with the same IP address. We need line 6 to tell the server which site we want to talk to.

Now we are done with the headers. Following the HTTP standard, we then 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 22, after the two times CR-LF. This body is HTML—no surprise there. Also note line 19, stating the "Content-Length," the number of bytes in the body, pretty much the same as a file-length.

Table 7.3 shows the well-known HTTP commands. 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?

Table 7.3: 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

7.12 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 an important set of ground rules for REST; see Table 7.4.

Table 7.4: REST rules.

Rule	Explanation
Addressable	Everything is seen as a resource that can be addressed, for example, with an 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.

With FTP, client and server need to agree on "where are we now?" In other words, which directory. This is very bad for scalability; you cannot have one FTP server answer one request (e. g., changing directory), and then let another in the server park handle the next. This is exactly what HTTP does allow. You can fetch a page from a web server, and when opening it up, it is full of links to embedded figures, etc. In the following requests for these, it can be other servers in the park that deliver 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 refetching of a page many times (apart from burning CPU cycles). So, out of the box HTTP delivers. An obvious trap, to avoid, is to have a command such as "Increment x." This suddenly adds state on the application level. So if you have received eight of something, and you want the ninth, then ask for the ninth, not the "next." This allows an impatient user (or application) to resend the command without side effects. It also means that the numbering of items must be well-defined.

Maybe REST is 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 of state-keeping on the client. Clients are often smaller machines, but they are many. Thus the solution 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. However, 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-to-be" 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 we use POST for, is performing actions. These are a little bit on the "edge" of REST, but hard to do without.

Listing 7.6: Sample REST GET

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

Listing 7.6 can be written directly into a browser's address bar and demonstrates another quality in REST: it is human-readable. In fact, it is not hard to create small test scripts in your favorite language, however, only GET can be written directly in the browser. As a consequence, REST is easier to debug in Wireshark than most other application protocols. Once you have decided to use REST on HTTP, there are still a few decisions to take:

- The data-format of 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 4.5. If you can create an object model, the whole development team understands and relates to, you are very far. If you can create an object model which third party developers, or customers, can understand and program against, you are even better off. The fact that you can piece together the URL by following the object model from its root is extremely simple, and very powerful.
- It is possible to address every node in the object tree specifically, reading or writing 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 typically depends on how you want to safeguard the system against unwanted changes, or the actions you need to do on a change.

  You may plan on having different user roles, each with a specific set of what they may read or change, and what not. In this case, it's a good idea to address the nodes individually, so that it is the plug-in in the web browser that uniformly manages user access, instead of the individual objects. However, as discussed in Section 4.1, there is a huge performance boost in setting a large subtree in one swoop. You might end up enabling the addressing of subnodes, as well as delivering/fetching whole subtrees in JSON or XML.

7.13 TCP sockets on IPv4 under Windows

In Section 7.3, 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, which is redrawn in Figure 7.8.

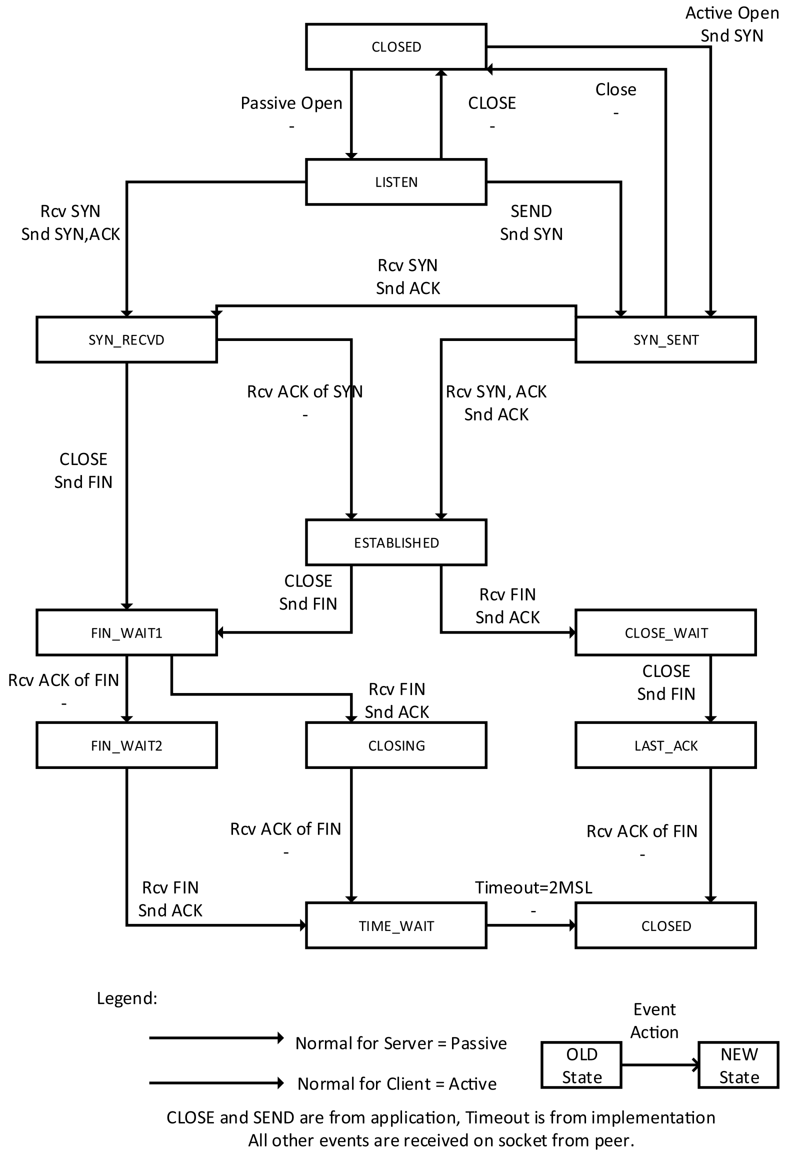


Figure 7.8: TCP state-event diagram.

The state where we want to be is “established.” This is where data is transferred. The rest is just building up or tearing down the connection. Similar to the opening three-way handshake, closing is normally done using two two-way handshakes. This is because the connection can send data in both directions. Both transmitters can independently state “I have no more to say,” which the other side can ACK. Older Windows stacks are lazy, they typically skip the two closing handshakes, instead sending 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,” responding to the first SYN with (SYN, ACK)—meaning that both flags are set. We have seen this earlier. When the client application is done with its job, it calls `shutdown()` on its socket. This causes TCP (still on the client) to send a FIN on the connection. The server TCP at the other end will answer this with an ACK when all outstanding retransmissions are done.

In the server application, `recv()` now unblocks. The server application sends any reply it might have, and now in turn calls `shutdown()` on its socket. This causes the TCP on the server to 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 sockets are closed on both sides (not simultaneously). This long sequence is to assure that when the given socket is reused there will be no interference from packets belonging to the old socket.

It is easy to see all the sockets existing 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. Listing 7.7 shows an example—with a lot of lines cut out to save space. Note the many sockets in “TIME_WAIT.”

Listing 7.7: Extracts from running netstat

```
C:\Users\kelk>netstat -a -b -p TCP

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:843          DK-W7-63FD6R1:0        LISTENING
 [Dropbox.exe]
  TCP    127.0.0.1:2559         DK-W7-63FD6R1:0        LISTENING
 [daemonu.exe]
  TCP    127.0.0.1:4370         DK-W7-63FD6R1:0        LISTENING
 [SpotifyWebHelper.exe]
  TCP    127.0.0.1:4664         DK-W7-63FD6R1:0        LISTENING
 [GoogleDesktop.exe]
  TCP    127.0.0.1:5354         DK-W7-63FD6R1:0        LISTENING
 [mDNSResponder.exe]
  TCP    127.0.0.1:5354         DK-W7-63FD6R1:49156    ESTABLISHED
 [mDNSResponder.exe]
```

```
  TCP    127.0.0.1:17600        DK-W7-63FD6R1:0         LISTENING
 [Dropbox.exe]
  TCP    127.0.0.1:49386        DK-W7-63FD6R1:49387     ESTABLISHED
 [Dropbox.exe]
  TCP    127.0.0.1:49387        DK-W7-63FD6R1:49386     ESTABLISHED
 [Dropbox.exe]
  TCP    192.168.0.195:2869     192.168.0.1:49704       TIME_WAIT
  TCP    192.168.0.195:2869     192.168.0.1:49705       TIME_WAIT
  TCP    192.168.0.195:2869     192.168.0.1:49706       TIME_WAIT
  TCP    192.168.0.195:52628    192.168.0.193:1400      TIME_WAIT
  TCP    192.168.0.195:52632    192.168.0.193:1400      TIME_WAIT
  TCP    192.168.0.195:52640    192.168.0.194:1400      TIME_WAIT
```

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 7.8 and 7.9 together form the skeleton of a web server on a Windows PC.[5] If you compile 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 always your local PC or other device. 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 browser view (typically with CTRL-F5), the port number will change as the operating system provides new ephemeral ports.

Listing 7.8: Windows server socket LISTEN

```c
#include "stdafx.h"
#include <winsock2.h>
#include <process.h>
#pragma comment(lib, "Ws2_32.lib")

static int threadCount = 0;
static SOCKET helloSock;
void myThread(void* thePtr);

int main(int argc, char* argv[])
{
   WSADATA wsaData; int err;
   if ((err = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
   {
      printf("Error␣in␣Winsock");
      return err;
   }

   helloSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (helloSock == INVALID_SOCKET)
   {
```

5 Please note that source code can be downloaded from https://klauselk.com

```
      printf("Invalid␣Socket␣-␣error:␣%d", WSAGetLastError());
      return 0;
   }

   sockaddr_in hello_in;
   hello_in.sin_family = AF_INET;
   hello_in.sin_addr.s_addr = inet_addr("0.0.0.0");  // wildcard
   hello_in.sin_port = htons(4567);
   memset(hello_in.sin_zero, 0, sizeof(hello_in.sin_zero));

   if ((err = bind(helloSock, (SOCKADDR*)&hello_in,
        sizeof (hello_in))) != 0)
   {
      printf("Error␣in␣bind");
      return err;
   }

   if ((err = listen(helloSock, 5)) != 0)
   {
      printf("Error␣in␣listen");
      return err;
   }
   sockaddr_in  remote;
   int  remote_len = sizeof(remote);

   while (true)
   {
      SOCKET sock = accept(helloSock, (SOCKADDR*)&remote,
                                      &remote_len);
      if (sock == INVALID_SOCKET)
      {
         printf("Invalid␣Socket␣-␣err:␣%d\n", WSAGetLastError());
         break;
      }

      printf("Connected␣to␣IP:␣%s,␣port:␣%d\n",
             inet_ntoa(remote.sin_addr), remote.sin_port);

      threadCount++;
      _beginthread(myThread, 0, (void *)sock);
   }

   while (threadCount)
      Sleep(1000);

   printf("End␣of␣the␣line\n");
   WSACleanup();
   return 0;
}
```

The following is a walk-through of the most interesting lines in Listing 7.8:

- Lines 1–8. Includes the socket library inclusion and global static variables.
- Line 13. Unlike Linux, Windows needs the WSAStartup to use sockets at all.
- Line 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 which is used as a handle in all future calls on the socket.
- Lines 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 this will work even if you change IP address, partly because it will listen to all your NICs. The port number must be specific. The macro `htons` means host-to-network-short, and is used to convert 16-bit **`short`**s, like a port number, from the host's "endian-ness" to the network's (see Section 3.3). The `inet_addr()` function outputs data in network order, so we do not need `htonl` here, which converts **`long`**s. Note that the macros change nothing if the CPU is big-endian.
- Line 32. We now handover the result of the above struggle to the socket in a `bind()` call.
- Line 39. 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 in stock for incoming connections. Our listening server is not a complete 5-tuple[6] as there is no remote IP address or port. `netstat -a` will show the status as "LISTENING."
- Line 49. The `accept()` call is the real beauty of the Berkeley TCP server. This call blocks until a client performs a SYN (connect). When this happens, the `accept()` unblocks. The original listening server socket keeps listening for new customers, and the new socket returned from the `accept()` call is a full-blown 5-tuple. We have used one of the 5 "stock" sockets, so the OS will create a new one in the background. `netstat -a` will show the status of this new socket as "ESTABLISHED."
- Line 61. A thread is spawned to handle the established socket, while the original thread loops back and blocks, waiting for the next customer.
- Line 68. When we break out of the loop, we need to clean-up.

Listing 7.9: Windows server socket ESTABLISHED

```
void myThread(void* theSock)
{
   SOCKET sock = (SOCKET)theSock;
   char rcv[1000]; rcv[0] = '\0'; // Allocate buffer
   int offset = 0; int got;

   sockaddr_in  remote;
```

6 5-Tuple: (Protocol, Src IP, Src Port, Dst IP, Dst Port).

```
 8     int  remote_len = sizeof(remote);
 9     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]; int headerlen =
33        _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     if (strstr(rcv, "quit"))
45        closesocket(helloSock);
46 }
```

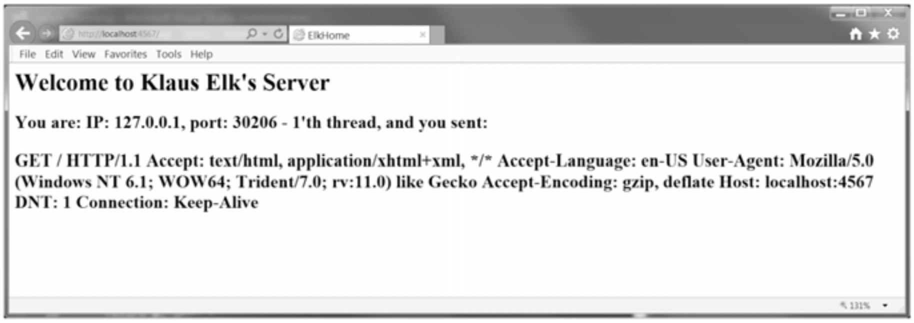
Continuing with Listing 7.9—again per line:

- Line 9. We use `getpeername()` so that we can write out IP address and port number of the remote client.
- Lines 13-17. The `recv()` call. If there is no data this statement blocks, otherwise it returns the number of bytes received so far. As this is a stream, there is absolutely no guarantee that we will 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 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.
- Line 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. Once we see two pairs of CR-LF, we are good.
- Line 23. We create the body of our HTML first. This is practical as it gives us the length of the body, which we will need in the header.
- Line 29. For printout we convert the IP address from a 32-bit number to the well--known x.y.z.v format in ascii using `inet_ntoa()`.
- Line 33. The header is generated—including the length of the body. Note that it ends with two CR-LF pairs.
- Line 36. We send the header.
- Line 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. We could have concatenated the header and the body before sending them. This would require an extra copy, but on the other hand save an OS call. You may want to experiment to find what is best for your embedded system.
- Line 41. We do a `shutdown()` of our sending side. This generates a FIN to the remote.
- Line 42. We do a `closesocket()`.
- Lines 44-45. As a small finesse, we close the parent socket if we have received "quit," for example, "http://localhost:4567/quit."

Figure 7.9 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 tiny web server's reply does not contain this header, it does not happen. If both parties agree on "Keep-Alive," we have the 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 af-



Welcome to Klaus Elk's Server

You are: IP: 127.0.0.1, port: 30206 - 1'th thread, and you sent:

GET / HTTP/1.1 Accept: text/html, application/xhtml+xml, */* Accept-Language: en-US User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko Accept-Encoding: gzip, deflate Host: localhost:4567 DNT: 1 Connection: Keep-Alive

Figure 7.9: Webbrowser with response from simple server.

ter 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. This is where a communication library can be handy.

A browser can also open parallel sockets. When you access a homepage your browser retrieves the "basic" HTML page. Embedded in this may be many 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 URLs 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."

7.14 IP fragmentation

Version 4 of IP is responsible for "adapting" the size of segments coming from "above" to what the underlying layer can handle. This is a little weird, since we already have seen how TCP breaks up large messages from the application layer to segments fitting into Ethernet frames further down. Why this adapter function?

Normally TCP MSS (maximum segment size) is created so that even after adding the necessary headers, it still fits within an Ethernet frame. But the packet is routed, sometimes over many different links, and one of these may not allow frames this big.[7] The router "looking into" such a link, must "fragment" the packet, in its IPv4 layer. The original packet stays fragmented for the rest of the journey until the fragments reach the destination IPv4 layer in the receiving host. Here, they are re-assembled, and not until all fragments are together, is the packet delivered to the TCP layer above.

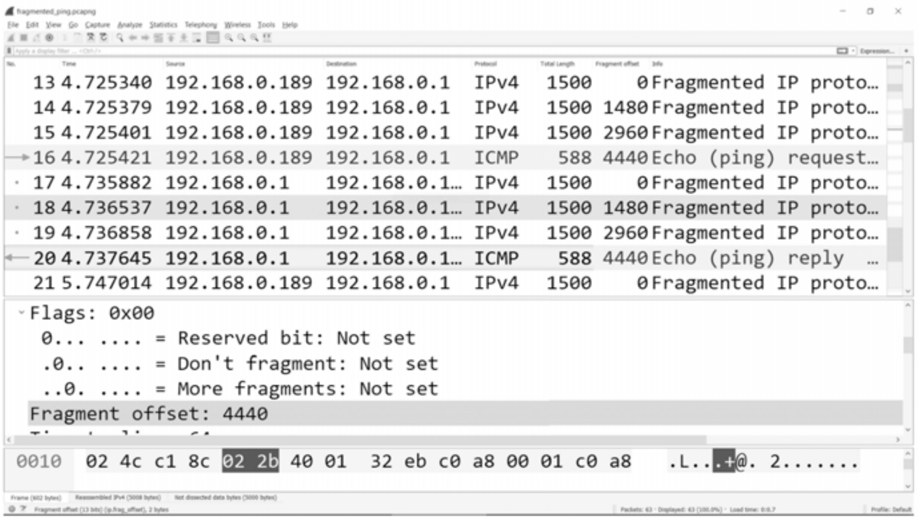
This is easily demonstrated with the help of the *ping* command. We tend to think of UDP and TCP as the only clients to the IP layer, but there is also ICMP. Listing 7.10 shows a normal ping command with the "-l" parameter that dictates the length of the data sent.

Listing 7.10: ICMP ping with 5000 bytes

```
C:\Users\kelk>ping -l 5000 192.168.0.1

Pinging 192.168.0.1 with 5000 bytes of data:
Reply from 192.168.0.1: bytes=5000 time=12ms TTL=64
Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64
Reply from 192.168.0.1: bytes=5000 time=7ms TTL=64
Reply from 192.168.0.1: bytes=5000 time=6ms TTL=64

Ping statistics for 192.168.0.1:
```

7 It may not even be an Ethernet link.

```
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 6ms, Maximum = 12ms, Average = 7ms
```



```
  13 4.725340 192.168.0.189 192.168.0.1    IPv4  1500    0 Fragmented IP proto…
  14 4.725379 192.168.0.189 192.168.0.1    IPv4  1500 1480 Fragmented IP proto…
  15 4.725401 192.168.0.189 192.168.0.1    IPv4  1500 2960 Fragmented IP proto…
→ 16 4.725421 192.168.0.189 192.168.0.1    ICMP   588 4440 Echo (ping) request…
  17 4.735882 192.168.0.1   192.168.0.1…   IPv4  1500    0 Fragmented IP proto…
  18 4.736537 192.168.0.1   192.168.0.1…   IPv4  1500 1480 Fragmented IP proto…
  19 4.736858 192.168.0.1   192.168.0.1…   IPv4  1500 2960 Fragmented IP proto…
← 20 4.737645 192.168.0.1   192.168.0.1…   ICMP   588 4440 Echo (ping) reply  …
  21 5.747014 192.168.0.189 192.168.0.1    IPv4  1500    0 Fragmented IP proto…

 Flags: 0x00
   0... .... = Reserved bit: Not set
   .0.. .... = Don't fragment: Not set
   ..0. .... = More fragments: Not set
 Fragment offset: 4440

0010  02 4c c1 8c 02 2b 40 01  32 eb c0 a8 00 01 c0 a8   .L...+@. 2.......
```

Figure 7.10: Fragmented ICMP ping with 5000 bytes of data.

Let’s examine the Wireshark capture of the ping in Figure 7.10:

- The “total length” column contains 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 “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 always occurs at 8-byte boundaries. In the figure, the offset is selected in the middle window, with the value 4440. This selects the same field in the bottom hex view. Here, it is 0x22b = 555 decimal—one-eighth of 4440.
- The info field contains an ID (outside the view) generated by the transmitting IPv4 layer. This is the same for all fragments from the same package, and incremented for the next.

Fragmentation is removed from IPv6. To simplify routers and improve performance, the router facing a link 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.

If you experience fragmentation, you should ask why and probably make sure it does not happen in the future, as it degrades performance.

7.15 Introducing IPv6 addresses

The main reason for introducing IPv6 was the shrinking IPv4 address space, 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 to 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, infrastructure in between. The removal of fragmentation in IPv6 simplifies routers. IPv4's dynamic header size requires routers 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.[8]

Table 7.5: 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 Table 7.5 states: IPv6 addresses are grouped in 16-bit chunks, written as 4-digit hexadecimal numbers. We are permitted to skip zeroes in front of each group and we can also skip *one* consecutive streak of zeros, using "::" notation for this. 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.

8 It has been suggested by Evan Jones that switches and routers simply recalculate the CRC and, therefore, might hide an internal malfunction. The actual case uncovered a Linux kernel error. Integrity checks as in Chapter 10 is a way to detect such problems.

Table 7.6: Functional differences between IPv4 and IPv6.

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

Table 7.6 shows the major differences between the two versions of the internet protocol.

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, for example, used in Link Aggregation.[9] The standard says that all packets in a given 5-tuple SHOULD be assigned the same 20-bit unique number—preferably a hash[10] of the 5-tuple. Traditionally, the 5-tuple has been used directly by network devices, but as some of these fields are 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 "%", helping the stack to use the right interface. On Windows this is the interface number (seen with "netstat -nr"), on Linux it could be, for example, "eth0."

Table 7.7 gives some examples on addresses in the two systems.

Table 7.7: 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

9 Link-Aggregation aka "Teaming" is a practical concept that allows the use of several parallel cables—typically between a multiport NIC and a switch.

10 Hashes are discussed in Section 10.4.

7.16 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, running as server in one instance, and as client in another. In our case, both run on a single Linux PC. This time we will try IPv6—it might as well have been the other way around.

Listing 7.11: Linux test main

```
....main...
   int sock_comm;
   int sock1 = do_socket();

   if (is_client)
   {
      do_connect(sock1, ip, port);
      sock_comm = sock1;
   }
   else // server
   {
      do_bind(sock1, 0, port);
      do_listen(sock1);
      sock_comm = do_accept(sock1);
   }

   TestNames(sock_comm);
   gettimeofday(&start, NULL);
```

The basic workflow is seen in Listing 7.11. 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 7.11 are shown in Listings 7.12 and 7.13. 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 OSs is that on Linux we do not need WSAStartup() or anything similar. This is nice, but not really a big deal.

What I find 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()`. The consequence is that you can open a socket and pass it on as a file handle to *any* process using files, for example, using `fprintf()`. This certainly can make life a lot easier for an application programmer.

Listing 7.12: Linux sockets part I

```
int do_connect(int socket, const char *ip, const bool port)
{
...
   struct sockaddr_in6 dest_addr;      // for destination addr
   dest_addr.sin6_family = AF_INET6;   // IPv6
   dest_addr.sin6_port = htons(port);  // network byte order

   int err = inet_pton(AF_INET6, ip, &dest_addr.sin6_addr);
   ...
   int retval = connect(socket, (struct sockaddr *)&dest_addr,
                        sizeof(dest_addr));
   ...
}

int do_bind(int socket, const char *ip, int port)
{
   struct sockaddr_in6 src_addr;       // for source addr
   src_addr.sin6_family = AF_INET6;    // IPv6
   src_addr.sin6_port = htons(port);   // network byte order
   if (ip)
   {
      int err = inet_pton(AF_INET6, ip, &src_addr.sin6_addr);
      {
         printf("Illegal␣address.\n");
         exit(err);
      }
   }
   else
      src_addr.sin6_addr = in6addr_any;

   int retval = bind(socket, (struct sockaddr *)&src_addr,
                     sizeof(src_addr));
   ...
}

int do_listen(int socket)
{
   int retval = listen(socket, 5); // Backlog of 5 sockets
    ...
}

int do_socket()
{
   int sock = socket(AF_INET6, SOCK_STREAM, 0);
   ...
}

int do_close(int socket)
```

```
{
   ...
   int retval = close(socket);
   ...
}
```

Looking at the code there are a number of differences between IPv4 and IPv6. Almost all relate to the functions 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 with `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 the unchanged TCP. However, we do not use `ntohl()` and `htonl()` on IP addresses, as they are too big for 32-bit words.
– There are new macros as `in6addr_any`.

*It can be a pain to get the address structures right at first. The most tricky parameter is the one inside a **`sizeof()`**, with the size of the structure which 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 name of the variable 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.*

Listing 7.13: Linux sockets part II

```
int do_accept(int socketLocal)
{
   ...
   struct sockaddr_in6 remote;
   socklen_t adr_len = sizeof(remote);

   int retval = accept(socketLocal,
                       (struct sockaddr *) &remote,&adr_len);
   ...
   char ipstr[100];
   inet_ntop(AF_INET6,(void*) &remote.sin6_addr,
             ipstr, sizeof(ipstr));
   printf("Got␣accept␣on␣socket␣%d␣with:"
          "␣%s␣port␣%d␣-␣new␣socket␣%d\n",
           socketLocal,ipstr,ntohs(remote.sin6_port),retval);
}

int do_send(int socket, int bytes)
```

```c
{
   ...
   if ((err = send(socket, numbers, bytes, 0)) < 0)
   ...
   return 0;
}

int do_recv(int socket, int bytes)
{
   int received = recv(socket, nzbuf,bytes-total, 0);
   ...
}

void TestNames(int socketLocal)
{
   struct sockaddr_in6 sock_addr;
   socklen_t adr_len = sizeof(sock_addr);
   char ipstr[100];

   getpeername(socketLocal,
               (struct sockaddr *) &sock_addr,&adr_len);

   inet_ntop(AF_INET6, (void*) &sock_addr.sin6_addr, ipstr,
             sizeof(ipstr));

   printf("getpeername:␣IP=␣%s,␣port=␣%d,␣adr_len=␣%d␣\n", ipstr,
           ntohs(sock_addr.sin6_port), adr_len);
}
```

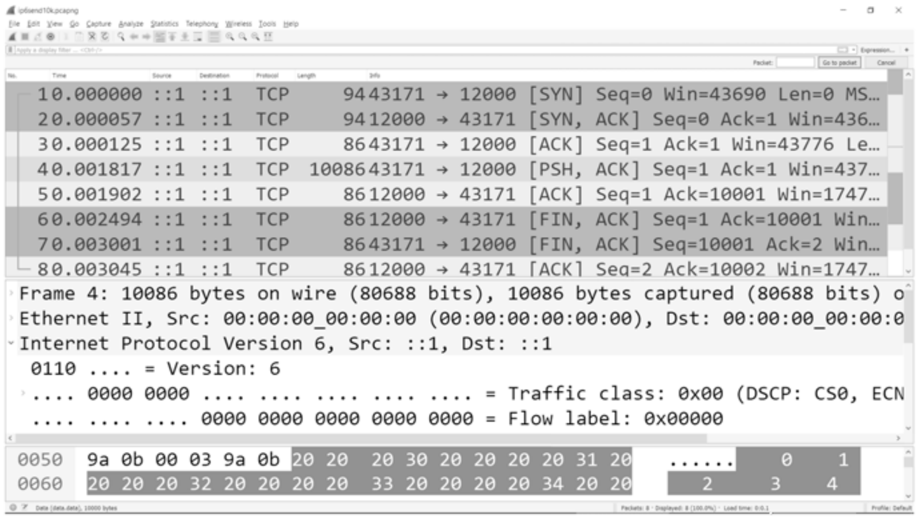
Listing 7.14 shows the execution.

Listing 7.14: Running Linux test

```
kelk@debianBK:~/workspace/C/ss6$ ./socktest -r --bytes=10000 &
[1] 1555
kelk@debianBK:~/workspace/C/ss6$ Will now open a socket
Binding to ip: localhost, port: 12000 on socket: 3
Listen to socket 3
Accept on socket 3

kelk@debianBK:~/workspace/C/ss6$ ./socktest -c -s --bytes=10000
Will now open a socket
Connecting to ip: ::1, port: 12000 on socket: 3
Got accept on socket 3 with: ::1 port 43171 - new socket 4
getpeername: IP= ::1, port= 43171, adr_len= 28
Loop     0
Plan to receive 10000 bytes on socket 4
getpeername: IP= ::1, port= 12000, adr_len= 28
Loop     0
Plan to send 10000 bytes on socket 3
```

```
Received 10000 bytes at address 0x7ffcb3b34e30:
   0    1    2    3    4    5    6    7    8    9
......
  7c8  7c9  7ca  7cb  7cc  7cd  7ce  7cf
Total sent: 10000 in 1 loops in 0.001910 seconds = 41.9 Mb/s
....
[1]+  Done                    ./socktest -r --bytes=10000
kelk@debianBK:~/workspace/C/ss6$
```

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 receives 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.



```
1 0.000000 ::1 ::1 TCP    94 43171 → 12000 [SYN] Seq=0 Win=43690 Len=0 MS…
2 0.000057 ::1 ::1 TCP    94 12000 → 43171 [SYN, ACK] Seq=0 Ack=1 Win=436…
3 0.000125 ::1 ::1 TCP    86 43171 → 12000 [ACK] Seq=1 Ack=1 Win=43776 Le…
4 0.001817 ::1 ::1 TCP 10086 43171 → 12000 [PSH, ACK] Seq=1 Ack=1 Win=437…
5 0.001902 ::1 ::1 TCP    86 12000 → 43171 [ACK] Seq=1 Ack=10001 Win=1747…
6 0.002494 ::1 ::1 TCP    86 12000 → 43171 [FIN, ACK] Seq=1 Ack=10001 Win…
7 0.003001 ::1 ::1 TCP    86 43171 → 12000 [FIN, ACK] Seq=10001 Ack=2 Win…
8 0.003045 ::1 ::1 TCP    86 12000 → 43171 [ACK] Seq=2 Ack=10002 Win=1747…

Frame 4: 10086 bytes on wire (80688 bits), 10086 bytes captured (80688 bits) o
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:0
Internet Protocol Version 6, Src: ::1, Dst: ::1
  0110 .... = Version: 6
  .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00 (DSCP: CS0, ECN
  .... .... .... 0000 0000 0000 0000 0000 = Flow label: 0x00000

0050  9a 0b 00 03 9a 0b 20 20  20 30 20 20 20 20 31 20   ......    0    1
0060  20 20 20 32 20 20 20 20  33 20 20 20 20 34 20 20      2    3    4
```

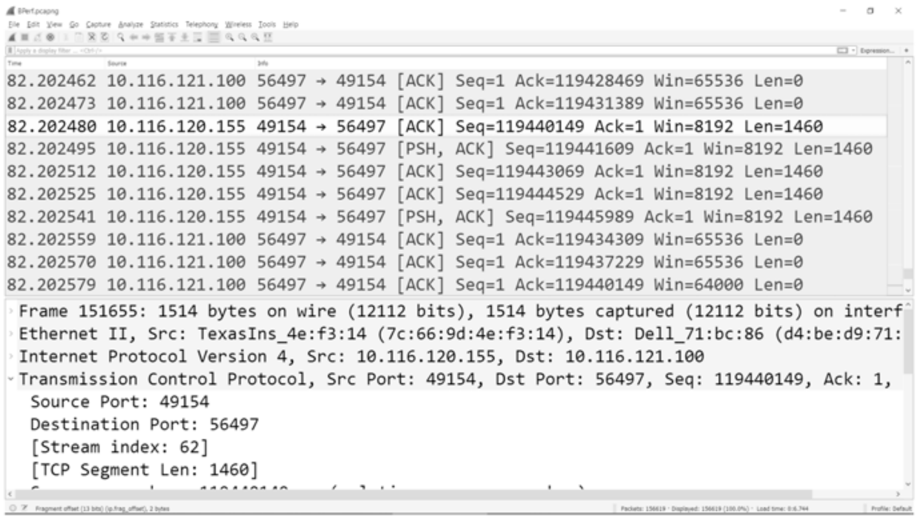
Figure 7.11: TCP on Linux localhost with IPv6.

Figure 7.11 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, because everything happens on the same PC. Interestingly, Wireshark still shows Ethernet in the stack, but with zero addresses at 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."

7.17 Data transmission

Until now, our focus on the TCP related stuff has been about how to get the transmission going. Now is the time to look at actual transmissions. When looking at TCP it makes little difference whether we have IPv4 or IPv6 beneath it—there is no "TCPv6." Figure 7.12 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). All frames are TCP segments between the two devices. To save space, only the Source column is shown.



```
Time      Source         Info
82.202462 10.116.121.100 56497 → 49154 [ACK] Seq=1 Ack=119428469 Win=65536 Len=0
82.202473 10.116.121.100 56497 → 49154 [ACK] Seq=1 Ack=119431389 Win=65536 Len=0
82.202480 10.116.120.155 49154 → 56497 [ACK] Seq=119440149 Ack=1 Win=8192 Len=1460
82.202495 10.116.120.155 49154 → 56497 [PSH, ACK] Seq=119441609 Ack=1 Win=8192 Len=1460
82.202512 10.116.120.155 49154 → 56497 [ACK] Seq=119443069 Ack=1 Win=8192 Len=1460
82.202525 10.116.120.155 49154 → 56497 [ACK] Seq=119444529 Ack=1 Win=8192 Len=1460
82.202541 10.116.120.155 49154 → 56497 [PSH, ACK] Seq=119445989 Ack=1 Win=8192 Len=1460
82.202559 10.116.121.100 56497 → 49154 [ACK] Seq=1 Ack=119434309 Win=65536 Len=0
82.202570 10.116.121.100 56497 → 49154 [ACK] Seq=1 Ack=119437229 Win=65536 Len=0
82.202579 10.116.121.100 56497 → 49154 [ACK] Seq=1 Ack=119440149 Win=64000 Len=0

› Frame 151655: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interf
› Ethernet II, Src: TexasIns_4e:f3:14 (7c:66:9d:4e:f3:14), Dst: Dell_71:bc:86 (d4:be:d9:71:
› Internet Protocol Version 4, Src: 10.116.120.155, Dst: 10.116.121.100
˅ Transmission Control Protocol, Src Port: 49154, Dst Port: 56497, Seq: 119440149, Ack: 1,
    Source Port: 49154
    Destination Port: 56497
    [Stream index: 62]
    [TCP Segment Len: 1460]
```

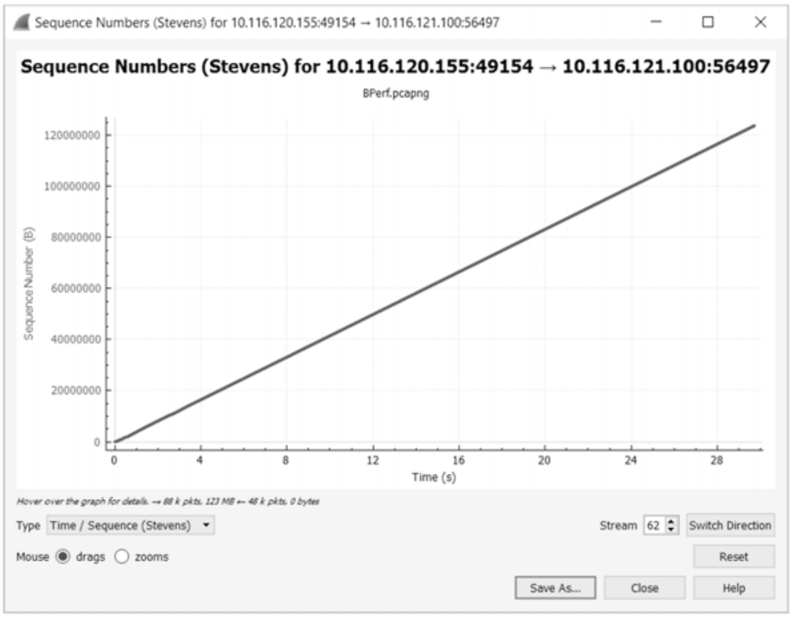
Figure 7.12: Transmitting many bytes.

All frames with data has 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 frames here are 1514 bytes according to Wireshark. The 14 bytes difference is the Source and Destination MAC addresses, each 6 bytes, plus the 2-byte "Type" Field with the value "0x0800" (not visible on the figure) for IPv4.

This is a pattern seen again and again. An incoming frame is received at the lowest layer. This layer needs to look at a given field to know where to deliver it. Here, Ethernet delivers it to IPv4, which, based on it's "Protocol" field, delivers it to TCP, which again delivers it to the application process, based on the port number.

Interestingly, the PC's sequence number is constantly "1" while the device is racing toward higher numbers. This means that the PC hasn't sent anything since its SYN flag. This is indeed a one-sided conversation. Outside the figure, at the bottom line in the "dissection" of the selected frame there is a line saying "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 frames through if they 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 7.13.



Hover over the graph for details. → 88 k pkts, 123 MB ← 48 k pkts, 0 bytes

Type: Time / Sequence (Stevens)

Mouse [✓ drags] zooms

Figure 7.13: Nice Stevens graph of sequence number versus time.

The Stevens graph, named after the author of the famous *TCP/IP Illustrated* 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, for example, X = 20 s with corresponding Y = 80 MBytes, we easily calculate the data rate to be 80 MByte/20 s = 4 MByte/s or 32 Mbps. The biggest challenge is counting the number of zeros. Inside the same window, you can also select, for example, “Roundtrip-Time.” Another menu-selection can give an I/O graph, but in this case they are as boring as Figure 7.13.

Figure 7.14 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 frame in question. Here, you should, for example, look for retransmissions, as these will occupy bandwidth without progressing sequence numbers.

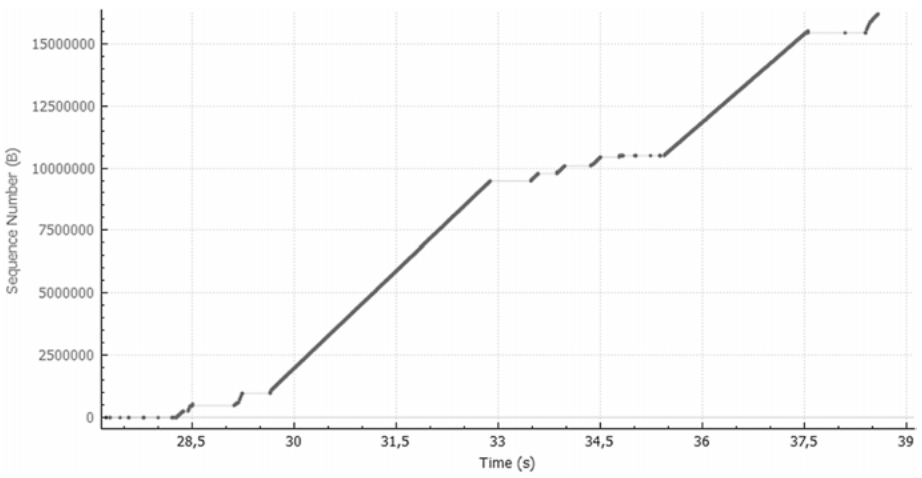


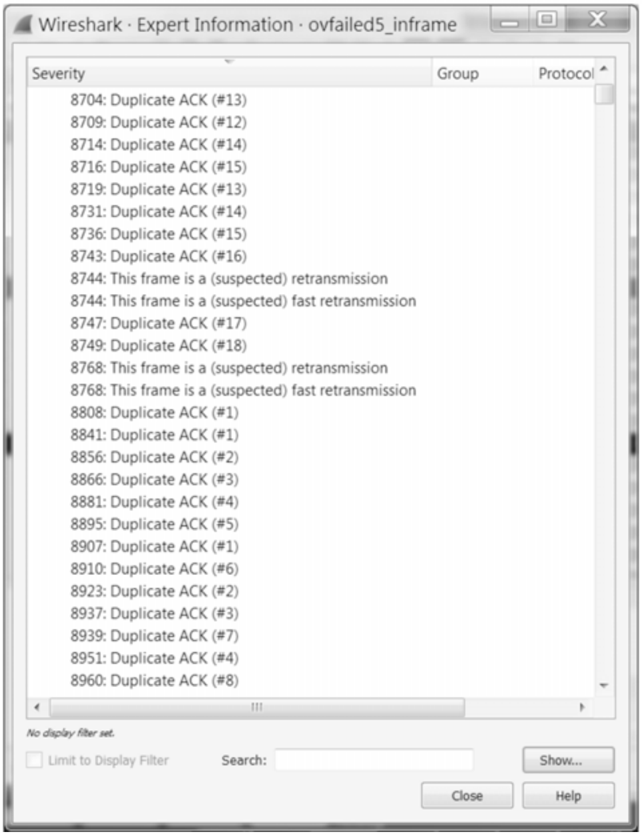
Figure 7.14: Stevens graph with many retransmissions.

You can also try “analyze” in the main menu, and then “expert information.” This is shown in Figure 7.15.

Here, the PC sends up to 20 Duplicate ACKs 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 7.23.

Note that when you capture 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, for example, 60 bytes per frame. This will allow the PC to “punch above its weight-class.”



```
8704: Duplicate ACK (#13)
8709: Duplicate ACK (#12)
8714: Duplicate ACK (#14)
8716: Duplicate ACK (#15)
8719: Duplicate ACK (#13)
8731: Duplicate ACK (#14)
8736: Duplicate ACK (#15)
8743: Duplicate ACK (#16)
8744: This frame is a (suspected) retransmission
8744: This frame is a (suspected) fast retransmission
8747: Duplicate ACK (#17)
8749: Duplicate ACK (#18)
8768: This frame is a (suspected) retransmission
8768: This frame is a (suspected) fast retransmission
8808: Duplicate ACK (#1)
8841: Duplicate ACK (#1)
8856: Duplicate ACK (#2)
8866: Duplicate ACK (#3)
8881: Duplicate ACK (#4)
8895: Duplicate ACK (#5)
8907: Duplicate ACK (#1)
8910: Duplicate ACK (#6)
8923: Duplicate ACK (#2)
8937: Duplicate ACK (#3)
8939: Duplicate ACK (#7)
8951: Duplicate ACK (#4)
8960: Duplicate ACK (#8)
```

Figure 7.15: Expert Information with lost data.

7.18 UDP sockets

We have used the terms broadcast (aka anycast), unicast, and multicast without really going into what the meaning is. Table 7.8 compares the three concepts.

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, restricted to the original subnet. A broadcast will never get through a router.

Table 7.8: The three types of -casts.

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

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, like broadcasts into a switch. Alternatively, it can support "group membership" and "spanning trees" and all sorts of advanced stuff, helping routers to only forward 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.

Now what about ARP, if several hosts have the same IP address? ARP is not used in the class D range. 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? Table 7.9 compares TCP and UDP on some of their main features.

Table 7.9: TCP versus UDP.

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

We have seen the stream nature of TCP in action. It is a two-edged sword giving programmers headaches trying to refind the "borders" in what was 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 7.10). These packets are rather small and can be resent 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 7.14. 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.

7.19 Case: UDP on IPv6

Listing 7.15 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.” The same rule applies to the members of the **struct**. We still use the macro htons on the port number, as this is UDP—not IP—and the transport layer is unchanged.

Listing 7.15: IPv6 UDP send on Linux

```
void sendit()
{
   int sock = socket(AF_INET6, SOCK_DGRAM, 0);
   if (sock <= 0)
   {
      printf("Opening␣socket␣gave␣error:␣%d\n",sock);
      exit(sock);
   }

   int err;
   char *mystring = "The␣center␣of␣the␣storm\n"; // Message

   struct sockaddr_in6 dst_addr;
   memset(&dst_addr, 0, sizeof(dst_addr)); // Clear scope & flow
   dst_addr.sin6_family = AF_INET6;
   dst_addr.sin6_port = htons(2000);       // TCP as usual

   if ((err=inet_pton(AF_INET6, "::1",&dst_addr.sin6_addr)) == 0)
   {
      printf("Illegal␣address.\n");
      exit(err);
   }

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  }
```

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.[11] Our next message may be `sendto()` someone else.

Just as we may call `bind()` before `connect()` in TCP, we may also do it in UDP before `sendto()`, but we normally don’t in either case. One reason is that it’s 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 the OS hands out, since we normally do not care about the client/active socket’s port number.

Listing 7.16 shows the code for a receiver, waiting for the data sent in the previous listing. Here, we set up the listener for “in6addr_any,” the same as “::0.”

Listing 7.16: IPv6 UDP receive

```
void recvit()
{
   int sock = socket(AF_INET6, SOCK_DGRAM, 0);
   if (sock <= 0)
   {
      printf("Opening␣socket␣gave␣error:␣%d\n",sock);
      exit(sock);
   }

   struct sockaddr_in6 listen_addr;
   memset(&listen_addr, 0, sizeof(listen_addr));
   listen_addr.sin6_family = AF_INET6 ;
   listen_addr.sin6_port = htons(2000);
   listen_addr.sin6_addr = in6addr_any;

   int on=1;
   if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
                         (char *)&on,sizeof(on)) < 0)
```

11 Confusingly, `connect()` *is* possible. It does not make a connection but allows the use of `send()` instead of `sendto()`, etc.

```
19     {
20        printf("setsockopt(SO_REUSEADDR)␣failed");
21        exit(0);
22     }
23
24     int err;
25     err = bind(sock, (struct sockaddr *) &listen_addr,
26                sizeof(struct sockaddr_in6));
27     if (err)
28     {
29        printf("Binding␣gone␣wrong␣-␣error␣%d\n", err);
30        exit(err);
31     }
32
33     char mystring[100];
34     struct sockaddr_in6 remote_addr;
35     socklen_t addr_len = (socklen_t) sizeof(remote_addr);
36
37     err = recvfrom(sock, mystring, sizeof(mystring),  0,
38          (struct sockaddr *) &remote_addr, &addr_len);
39     if (err < 0)
40     {
41        printf("Could␣not␣recv.␣Error:␣%d\n", err);
42        exit(err);
43     }
44     else
45        printf("Received:␣%s", mystring);
46  }
```

The strange code in lines 17–22 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 only run once, but somehow it never is.

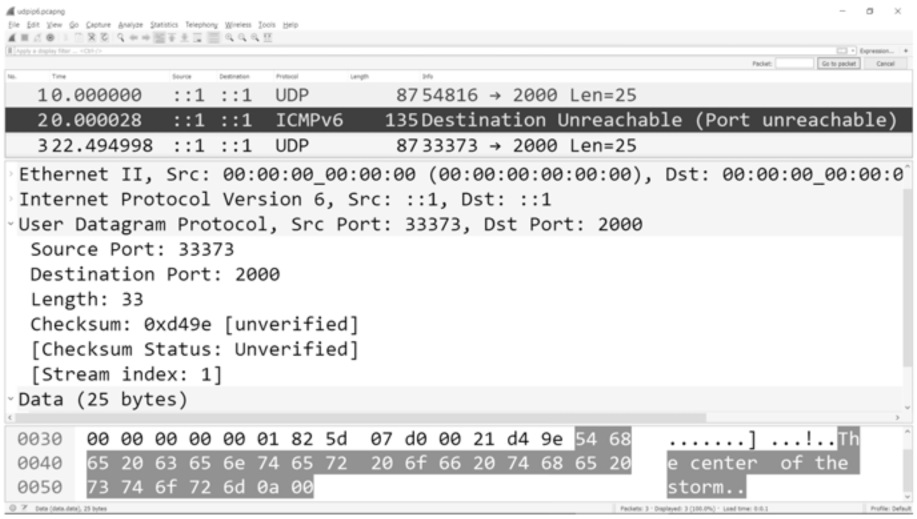
Now we need a bind for the listening socket. Naturally, it must be listening at the right port, and the "any" IP address is practical. If a packet makes it this far it must be to one of our interfaces, so why not wait for them all and do away with the need to find out which IP addresses we have where? Then we declare a "holder" for the remote address, which we could print out (but don't), and finally we call `recvfrom()`. This blocks until a message is received. When that happens it is written to the terminal.

Listing 7.17 shows the test of the two programs. First, `udprecv` is run in the background. We then use `netstat`[12] 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.

12 slimmed to fit on the page.

Listing 7.17: Running on localhost

```
kelk@debianBK:~/workspace/C/ss6$ ./udprecv &
[1] 2734
kelk@debianBK:~/workspace/C/ss6$ netstat -au -pn -6
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address PID/Prog name
udp6       0      0 :::48725      :::*            -
udp6       0      0 :::34455      :::*            -
udp6       0      0 :::10154      :::*            -
udp6       0      0 :::961        :::*            -
udp6       0      0 :::2000       :::*            2734/udprecv
udp6       0      0 :::111        :::*            -
udp6       0      0 :::5353       :::*            -
kelk@debianBK:~/workspace/C/ss6$ ./udpsend
Received: The center of the storm
[1]+  Done                    ./udprecv
```



```
1 0.000000   ::1 ::1 UDP      87 54816 → 2000 Len=25
2 0.000028   ::1 ::1 ICMPv6  135 Destination Unreachable (Port unreachable)
3 22.494998  ::1 ::1 UDP      87 33373 → 2000 Len=25

Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:0
Internet Protocol Version 6, Src: ::1, Dst: ::1
User Datagram Protocol, Src Port: 33373, Dst Port: 2000
  Source Port: 33373
  Destination Port: 2000
  Length: 33
  Checksum: 0xd49e [unverified]
  [Checksum Status: Unverified]
  [Stream index: 1]
Data (25 bytes)

0030  00 00 00 00 00 01 82 5d  07 d0 00 21 d4 9e 54 68   .......] ...!..Th
0040  65 20 63 65 6e 74 65 72  20 6f 66 20 74 68 65 20   e center  of the
0050  73 74 6f 72 6d 0a 00                               storm..
```

Figure 7.16: UDP on IP6 with ICMPv6 error.

Figure 7.16 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.

Twenty-two seconds later, the full scenario is run as in Listing 7.17, and this time there is no ICMP error as the waiting socket accepts it.

7.20 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. An overview is given in Table 7.10.

Table 7.10: Parameters on application layer protocols.

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, very good arguments are needed to pick another protocol. After all, protocols tie equipment together. Do you really want to be the person responsible, if the new product doesn't work in a system with the old product, or if the software developers are delayed another half year to support your new fancy protocol?[13]
– *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 reimplement 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 round trip 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.

13 There are times when the answer is yes.

If A and B are on either side of the globe, the round trip 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 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 lots of 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 we saw in Section 7.11, 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 7.11.

– *State*
  As discussed in Section 7.12 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 sometimes do, 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 data unique to this particular device.
– *Low Level*
  Many embedded developers prefer binary protocols as they are compressed compared to handling, for example, 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, for example, JSON or XML, it takes a lot more space as, for example, parameter names induces overhead, and in the case of single variables 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 or she has a lot of tools available for parsing data. Between the two parties, we have the wire. Watching a custom binary protocol in Wireshark can be a pain, whereas the same data in JSON or XML is close to self-explanatory.
  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, meaning that there is a trend toward 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 toward 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 why 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. 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 backward compatible manner.
– *Dependencies*
  Not so long ago, DCOM was popular for remote control. A major problem with DCOM was the 100 % tie to the Windows OS. Before this, CORBA was very popular in the Unix world. It required expensive tools that you could also get for PCs, but it never really caught on here. Both DCOM and CORBA were based on remote procedure calls. Neither performed very well, if one end of the communication suddenly wasn't available.
  A major reason for the success of REST on top of HTTP (see Section 7.12) is that it is easy to understand, runs without complex tools and on any OS. In the IoT world, we need a loosely coupled concept with clients and servers instead of masters and slaves. We also need something not tied to a specific OS or language. 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 4.1 goes into more details on protocol performance which is closely linked to power consumption.

7.21 Alternatives to the socket API

The socket concept used in this book is generally known as Berkeley sockets. The basic API is the same on all major platforms, with 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, for example, generating checksums, hopcount, type fields, etc. This is only relevant in test scenarios where you want to test error-handling. We saw this in Section 5.9.

In order to tweak the various socket options, you often end up using 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, see Section 10.12), 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 5.9. PHP on top of libcurl is also very effective. You may argue that Python and PHP are for web servers, but a larger IoT device may 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 Apache or Nginx 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.

7.22 Ethernet cabling

When it comes to Ethernet cables, there is a lot of baggage from the old days and names and terms that are not completely correct. The most used are given in Table 7.11.

One of the confusing things with Ethernet cables is when to use the EIA-568A wiring scheme and when to use the EIA-568B wiring scheme. Since most cables today are the “straight-over patch-type” it makes little difference. The really important thing is that all cables go straight through *and* that the pairs are maintained, so we do get them twisted pairwise, thus maintaining a high immunity to induced “common mode” noise. The pairs are easily identified by their colors. For example, one wire is green and the other wire in the pair is white with a green stripe.

Table 7.11: Names and terms used with Ethernet cables.

Term	Explanation
Coax	Old standard for multidrop. Not used anymore.
UTP	Unshielded Twisted Pair. This is the most normal type today—used in star configuration.
Fully Mounted	UTP normally has 4 pairs of two wires. Cheaper versions with less wires may work. Do not use them.
F/UTP S/UTP	Two variants with foil shield. This must be connected throughout the network.
CAT 5	Cables supporting 100 Mbps up to 100 m.
CAT 5e	Cables supporting 1 Gbps up to 100 m.
CAT 6	Cables supporting 1 Gbps up to 100 m.
CAT 6a	Cables supporting 10 Gbps/s up to 100 m
8P8C	Real name for connector—8 Positions and 8 Contacts
RJ45	Name almost always used for the connector on Ethernet cables—although not completely correct.
Patch Cable	Normal cables where all connections go straight through. Male connectors at both ends.
Cross-over	Older cables for connecting, for example, two PCs.
Auto M-Dix	Part of std. for 1 Gbps ports that negotiates automatic cross-over when needed.
EIA-568A/B	A and B are two different standards for which wire pair goes where. In a patch cable it really makes no difference. The important thing is that the pairs are maintained.

If you look at a male “RJ45” Ethernet connector from the connector end (cable going away from you) with the small tap/hook downwards and the contacts upwards, the pins are numbered 1–8 from the right side. Table 7.12 shows the connections in the EIA-568A setup. The EIA-568B is the same, except that the orange/white and the green/white pairs are swapped, but as stated, it makes little difference which is chosen. The strange layout is historic, enabling backwards compatibility with 4-pin connectors. It may be easier to understand when explained: There is a pair at the center

Table 7.12: EIA-568A wiring scheme.

Pair	Wire	Pin
1	Blue	4
Blue+White	White/Blue	5
2	Orange	6
Orange+White	White/Orange	3
3	Green	2
Green+White	White/Green	1
4	Brown	8
Brown+White	White/Brown	7

pins (4,5), then another pair on either side of this (3,6) and then one pair at one side (1,2) and another pair at the other side (7,8).

The initial wiring scheme, where pairs are symmetrically added at either side of the center was clever, until it wasn't. The growing space between the wires in a pair became a problem due to noise, and was abandoned. Anyway, if the above instruction is followed, a patch cable will work.

If you are making your own Ethernet cables, I recommend that you buy an Ethernet tester. Figure 7.17 shows a tester costing less than $10. The tester simply uses its battery to test one wire at a time, so if you see a "walking light" from 1 through 8 at the remote end, you are okay.

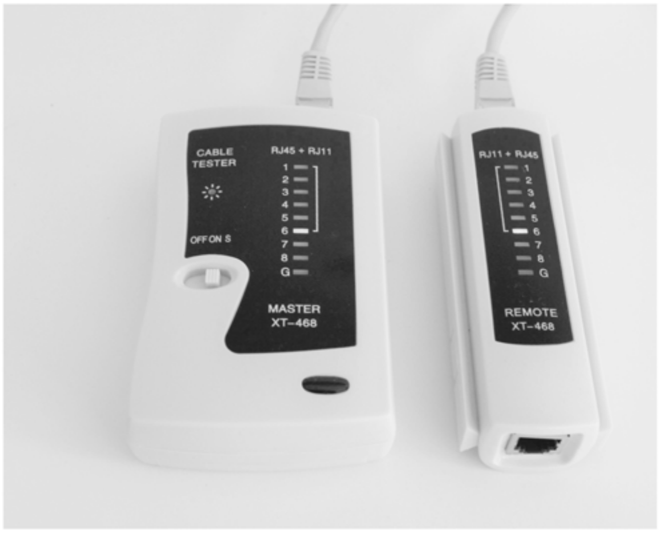


Figure 7.17: Low-cost Ethernet tester.

7.23 Physical layer problems

In Section 7.17, 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 7.18.

Basically, a test program is run that sends all "symbols" in many sequences. The symbols are overlaid on top of each other and variations in time and/or amplitude stands out. Thus, even though various parameters are measured, the diagram is very informative by itself.

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

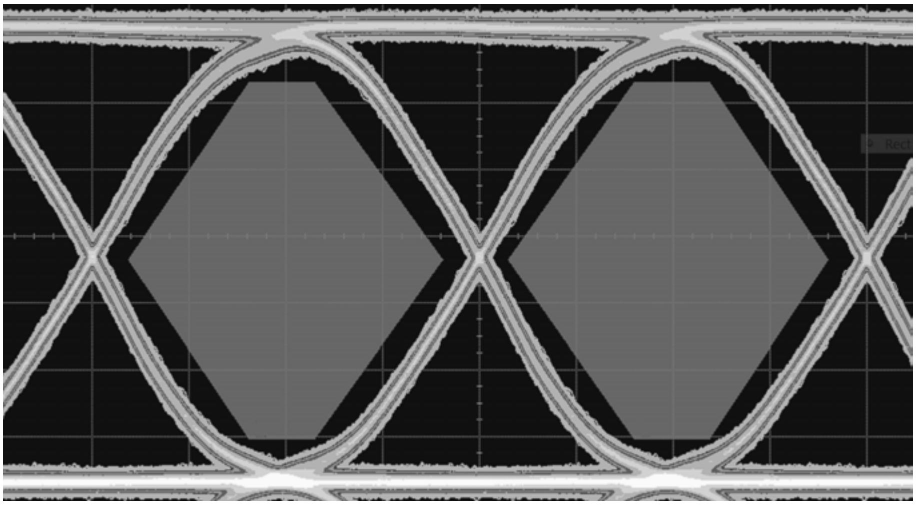


Figure 7.18: A good eye diagram.

My colleagues and I saw the problem with the physical layer in a prototype which 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 such a problem is diagnosed, however, it is a good idea to fire up the heavy tools 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. It has been stated earlier that modern network interface cards (NICs) have built-in checksum calculations. However, that statement is about the IP and TCP checksums. Even the oldest NICs have a hardware-based check of the Ethernet 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 support this. 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 8.4.

You can get libraries for SNMP, and thus build this kind of test into your own diagnostics. This will allow your customers to run the test and mail the result.

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 sitting between the MAC and the magnetics (and the connector), responsible for converting the more or less[14] analog waveform on the wire to a bit pattern. Many types of transients[15] can occur at the wire. They will go through the connector and the magnetics to the PHY, where they may kill the signal. In some scenarios, they may even kill the PHY. The correct protection circuitry is thus important here.

14 Depending on the "coding scheme" the bits are almost sent "as is" or coded into softer waveforms—unrecognizable on an oscilloscope.

15 Short peaks in currents and/or voltages, for example, electromagnetically induced.

7.24 Further reading

- Kurose and 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, and 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. 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 such as HTTP, and newer sources are recommended on these subjects.
- Richardson and 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.

8 Network tools

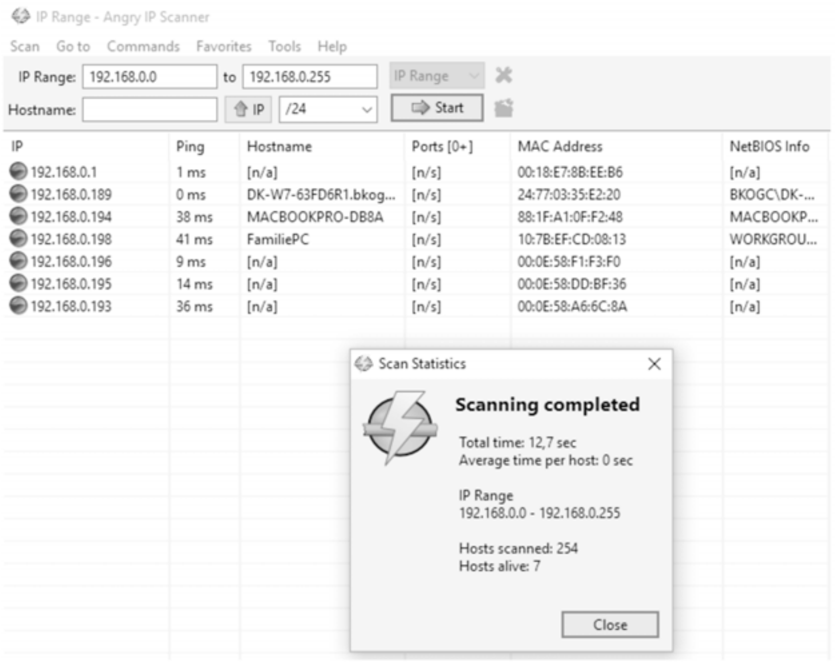
8.1 Finding the IP address

In Section 7.5, we discussed how a device gets an IP address. But how do we know what it is? This is a common problem when working with TCP/IP based devices. Normally, you know which TCP port you want to address, but what is the IP address?

A very common strategy is to give up on finding the IP address, and instead simply set it to its default. Many devices have a reset button of some kind, and a label specifying the default IP address and mask. The default is typically a static address like, for example, 192.168.1.1 with mask 255.255.255.0. In other words, an address in a private range in a /24 network. After resetting to this default, you configure an Ethernet port on your PC to, for example, 192.168.1.2 and the same mask. Now you connect the PC directly to the device, and you should be in business. Now you typically set the device to the preferred IP address and mask, or maybe to DHCP. When this is done, you must remember to set your PC back to whatever it was before, maybe DHCP.

A second strategy is brute force, using an IP scanner. This is a good strategy if you are on a small network like, for example, a home office, and you believe the device to be already inside the subnet of this network. An advantage here is that you do not need to change the settings on your PC back and forth.

Figure 8.1 shows the popular "Angry IP Scanner" in action.


IP Range - Angry IP Scanner
Scan Go to Commands Favorites Tools Help
IP Range
Start
Scan Statistics
Close

IP Range: 192.168.0.0 to 192.168.0.255
Hostname: /24

IP	Ping	Hostname	Ports [0+]	MAC Address	NetBIOS Info
192.168.0.1	1 ms	[n/a]	[n/s]	00:18:E7:8B:EE:B6	[n/a]
192.168.0.189	0 ms	DK-W7-63FD6R1.bkog...	[n/s]	24:77:03:35:E2:20	BKOGC\DK-...
192.168.0.194	38 ms	MACBOOKPRO-DB8A	[n/s]	88:1F:A1:0F:F2:48	MACBOOKP...
192.168.0.198	41 ms	FamiliePC	[n/s]	10:7B:EF:CD:08:13	WORKGROU...
192.168.0.196	9 ms	[n/a]	[n/s]	00:0E:58:F1:F3:F0	[n/a]
192.168.0.195	14 ms	[n/a]	[n/s]	00:0E:58:DD:BF:36	[n/a]
192.168.0.193	36 ms	[n/a]	[n/s]	00:0E:58:A6:6C:8A	[n/a]

Scanning completed

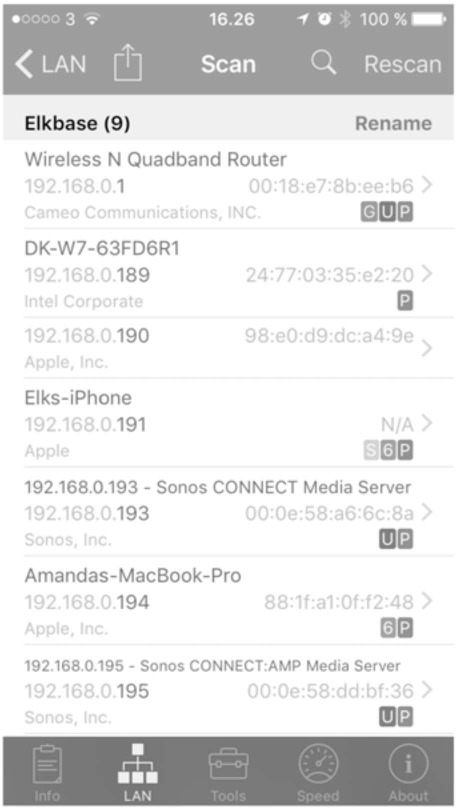
Total time: 12,7 sec
Average time per host: 0 sec

IP Range
192.168.0.0 - 192.168.0.255

Hosts scanned: 254
Hosts alive: 7

Figure 8.1: Angry IP scanner for Windows.

https://doi.org/10.1515/9781547401024-008


●○○○○ 3 16.26 100 %

< LAN Scan Rescan

Elkbase (9) Rename

Wireless N Quadband Router
192.168.0.1 00:18:e7:8b:ee:b6
Cameo Communications, INC. GUP

DK-W7-63FD6R1
192.168.0.189 24:77:03:35:e2:20
Intel Corporate P

192.168.0.190 98:e0:d9:dc:a4:9e
Apple, Inc.

Elks-iPhone
192.168.0.191 N/A
Apple S6P

192.168.0.193 - Sonos CONNECT Media Server
192.168.0.193 00:0e:58:a6:6c:8a
Sonos, Inc. UP

Amandas-MacBook-Pro
192.168.0.194 88:1f:a1:0f:f2:48
Apple, Inc. 6P

192.168.0.195 - Sonos CONNECT:AMP Media Server
192.168.0.195 00:0e:58:dd:bf:36
Sonos, Inc. UP

Figure 8.2: iPhone net analyzer.

Table 8.1: Network analyzer pro icons.

Icon	Meaning
G	**G**ateway
S	**S**canning Device (the iPhone itself)
P	**P**ingable
U	**U**PNP/DLNA services available
6	IP**6** availability

Figure 8.2 shows the result of a similar action; this time using a small app called “Network Analyzer Pro” on an iPhone. The explanation for the small icons is found in Table 8.1.

If this is all performed on a small home network, a third alternative could be to look at the wireless SOHO router. This typically has a table showing the devices on the network. If the device in question is connected to the network, you can find it here.

The fourth alternative is bringing out Wireshark. You connect the device to your PC, fire up Wireshark, and then power up the device. Often, it will start to chat a little, and if the IP address is static, it is easy to see what it is in Wireshark.

If the address is not static, it will be looking for a DHCP server as we saw in Section 7.6, and we need to supply this. If we connect the device to our company network or home-office network, we are back to the previous choices (IP scanner or table in SOHO router). An alternative is to setup our own PC as DHCP server. On Linux, this is generally a question of setting a checkmark in the packet manager or similar, but on Windows you need to find something you can trust. A German developer, Uwe Ruttkamp, has made a nice DHCP implementation for Windows 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 off your private DHCP server, before connecting to a company network. The IT guys are not too happy about "alternative" DHCP servers.

8.2 The switch as a tool

8.2.1 Mirroring

If you are working with networks, you are used to switches. A switch is a nice plug'n play device allowing 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 both are too small to run Wireshark?

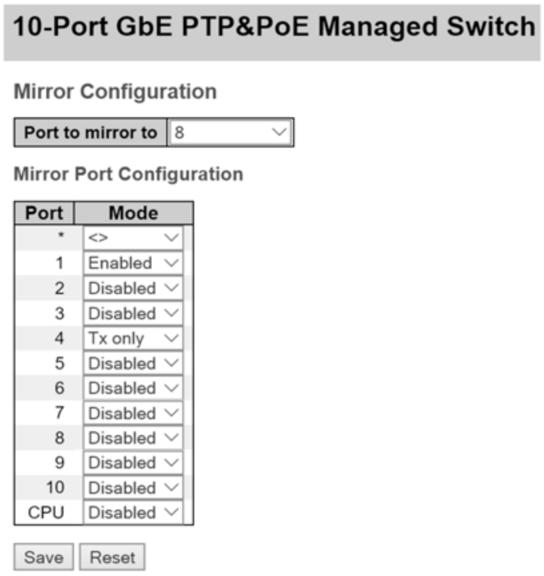
This is where we bring out the managed switch (or sometimes better, a tap, see Section 8.3). A managed switch can be bought for less than $200. Managed switches typically have the ability to select a "mirror port." This means that you can ask the switch to output on this port; any data going in and/or out of one or several other ports. So if, for example, your two devices are connected to ports 1 and 2, you can set up the switch to mirror one of these to port 3, which is where you connect the PC. A similar setup is seen in Figure 8.3 where port 1, Tx and Rx, as well as Port 4, Tx only, is mirrored to port 8.

In a normal star configuration with, for example, 1 Gbps connections, it is possible to have 1 Gbps in both directions at the same time, which means 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 the most traffic in one direction, but it is something we must be aware of. If Wireshark reports lost frames, this *could* be the reason.

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 addressable in your subnet. This means that you need to set the IP address of the switch, and typically to set it, you need to know the current IP address—a classic "Catch 22." See Section 8.1 on how to solve this problem.

Most switches have an RS-232[1] or more commonly a USB connection with a command-line interface that can be used directly, without knowing the switch IP

1 You can buy cables with a USB-to-RS232 converter.



10-Port GbE PTP&PoE Managed Switch

Mirror Configuration

Port to mirror to	8

Mirror Port Configuration

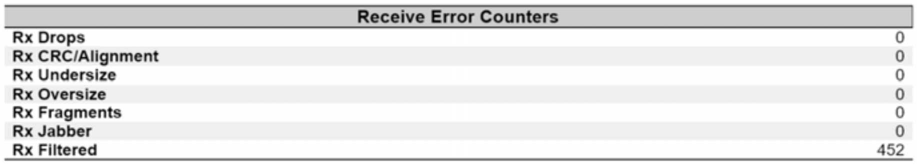
Port	Mode
*	<>
1	Enabled
2	Disabled
3	Disabled
4	Tx only
5	Disabled
6	Disabled
7	Disabled
8	Disabled
9	Disabled
10	Disabled
CPU	Disabled

Figure 8.3: Mirror setup on port 8 in switch.

address. However, these interfaces can be cumbersome to use, and very vendor specific.

8.2.2 Statistics

Managed switches have a statistics page. If an embedded system is suspected of having problems on the physical layer, this statistics page is a good place to start, especially the “Receive Error Counters” that Figure 8.4 has zoomed in on.



Receive Error Counters	
Rx Drops	0
Rx CRC/Alignment	0
Rx Undersize	0
Rx Oversize	0
Rx Fragments	0
Rx Jabber	0
Rx Filtered	452

Figure 8.4: Receive error counters from statistics page in switch.

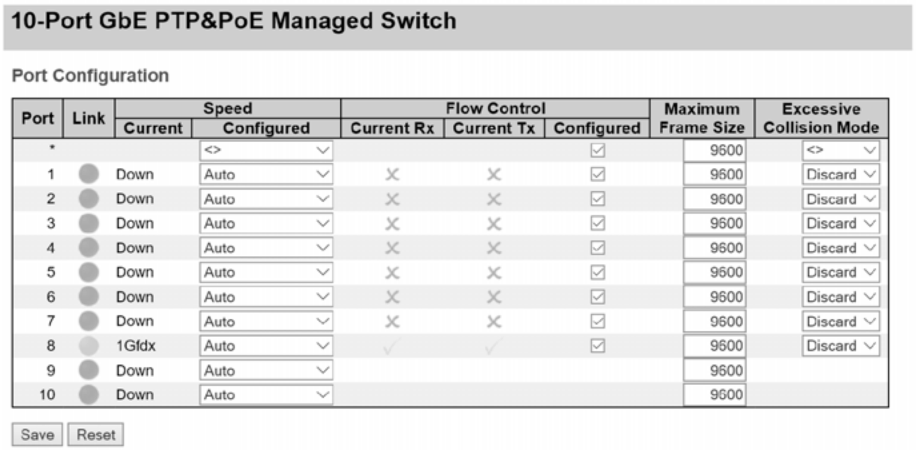
Naturally, this only tells us about the quality of the *output* frames from the embedded system, although many TCP retransmissions will also occur if there are problems with the *input*. To see information about the input or to dig deeper, SNMP may be used as described in Section 8.4.

8.2.3 Simulating lost frames

Sometimes it is tempting to test retransmissions and general robustness by simply pulling out the Ethernet cable from the embedded device and plugging it in after a few seconds. 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 between the switches is unplugged, there are no “link down” events, and the cable can be swiftly inserted again.

8.2.4 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. See Figure 8.5.


10-Port GbE PTP&PoE Managed Switch

Port Configuration

Port	Link	Speed		Flow Control			Maximum	Excessive
		Current	Configured	Current Rx	Current Tx	Configured	Frame Size	Collision Mode
*			<>			☑	9600	<>
1	●	Down	Auto	✗	✗	☑	9600	Discard
2	●	Down	Auto	✗	✗	☑	9600	Discard
3	●	Down	Auto	✗	✗	☑	9600	Discard
4	●	Down	Auto	✗	✗	☑	9600	Discard
5	●	Down	Auto	✗	✗	☑	9600	Discard
6	●	Down	Auto	✗	✗	☑	9600	Discard
7	●	Down	Auto	✗	✗	☑	9600	Discard
8	●	1Gfdx	Auto	✓	✓	☑	9600	Discard
9	●	Down	Auto				9600	
10	●	Down	Auto				9600	

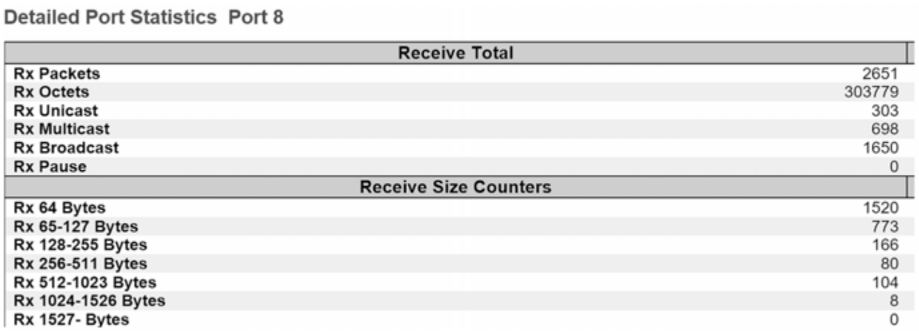
Figure 8.5: Port setup with flow control.

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 us to set the max frame size. This can be used to provoke and test the so-called IPv4 fragmentation; see Section 7.14.

The ability to toggle flow control on and off, can help investigate the pros and cons 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.

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 Ether-



Detailed Port Statistics Port 8

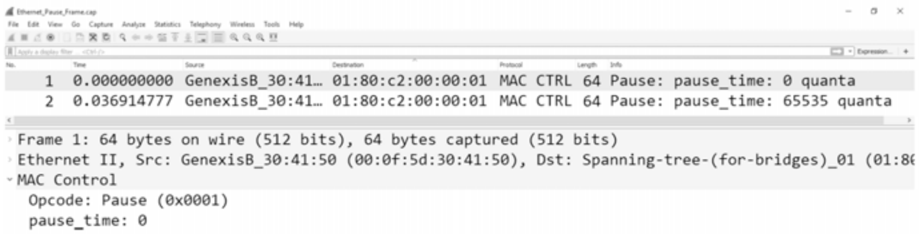
Receive Total	
Rx Packets	2651
Rx Octets	303779
Rx Unicast	303
Rx Multicast	698
Rx Broadcast	1650
Rx Pause	0
Receive Size Counters	
Rx 64 Bytes	1520
Rx 65-127 Bytes	773
Rx 128-255 Bytes	166
Rx 256-511 Bytes	80
Rx 512-1023 Bytes	104
Rx 1024-1526 Bytes	8
Rx 1527- Bytes	0

Figure 8.6: Detailed port statistics in switch.

net flow control is on both sides of a link, and thus can assure against the packet loss 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, stopping 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 8.7 is a Wireshark capture where frame 1 ends the previous pause, while frame 2 starts a new one.


Ethernet_Pause_Frame.cap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info

```
    1  0.000000000  GenexisB_30:41…  01:80:c2:00:00:01  MAC CTRL  64  Pause: pause_time: 0 quanta
    2  0.036914777  GenexisB_30:41…  01:80:c2:00:00:01  MAC CTRL  64  Pause: pause_time: 65535 quanta

› Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits)
› Ethernet II, Src: GenexisB_30:41:50 (00:0f:5d:30:41:50), Dst: Spanning-tree-(for-bridges)_01 (01:8
˅ MAC Control
    Opcode: Pause (0x0001)
    pause_time: 0
```

Figure 8.7: Ethernet pause frames.

The *quanta* parameter is multiplied by the transmission-time of 512 bits. It is a timeout, and comes into play if the pause is not ended by a pause frame with 0 as argument (as in frame 1) before this time has passed. 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 Wireshark capture—01:80:c2:00:00:01. The Wireshark info field can be a little confusing by relating this to spanning trees.

8.3 Tap

A tap is a device inserted between two network devices, with an Ethernet connector for each, and a third connector for a PC. See Figure 8.8.

Figure 8.8: Tap device.

Taps come in different speeds, typically 100 Mbps or 1 Gbps per port. The third connector is typically a USB connector, with a driver for PCs so that Wireshark works via this. If it is USB3—and if this is supported by your PC—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. A switch is readily available in most labs.

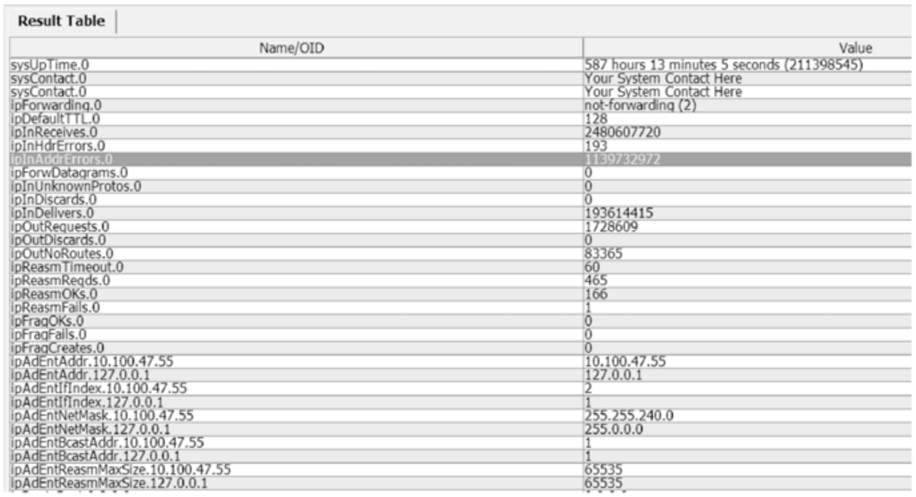
8.4 SNMP

An SNMP (simple network management protocol) server is implemented in many network devices such as switches and routers, but also in general embedded operating systems. The network device implements a MIB (management information base). This is an object model, as described in Section 4.5.

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 selected part of a screen-shot from such a program from iReasoning (not free, but with a trial period of one month). It is possible to define a group of devices, and thus pinpoint the weakest point in a network.

The figure is cut out of a bigger screenshot in order to be readable. The following is a description of the full screen, *not* shown here. At the very top left is the IP address of the network device—in this case an embedded device running Windows CE. Just below the IP address is the MIB in tree form. At the top right is the actual command sent—here, "Get Subtree." This is very practical and is the one used to generate the right side containing the actual current data from the MIB. The lower left window has static help information on the selected line.

The part of the screen, actually seen in the figure is thus a small part of the MIB, selected in the tree (outside the picture). Clearly, there is a lot of information to comprehend, but 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.



Result Table

Name/OID	Value
sysUpTime.0	587 hours 13 minutes 5 seconds (211398545)
sysContact.0	Your System Contact Here
sysContact.0	Your System Contact Here
ipForwarding.0	not-forwarding (2)
ipDefaultTTL.0	128
ipInReceives.0	2480607720
ipInHdrErrors.0	193
ipInAddrErrors.0	1139732972
ipForwDatagrams.0	0
ipInUnknownProtos.0	0
ipInDiscards.0	0
ipInDelivers.0	193614415
ipOutRequests.0	1728609
ipOutDiscards.0	0
ipOutNoRoutes.0	83365
ipReasmTimeout.0	60
ipReasmReqds.0	465
ipReasmOKs.0	166
ipReasmFails.0	1
ipFragOKs.0	0
ipFragFails.0	0
ipFragCreates.0	0
ipAdEntAddr.10.100.47.55	10.100.47.55
ipAdEntAddr.127.0.0.1	127.0.0.1
ipAdEntIfIndex.10.100.47.55	2
ipAdEntIfIndex.127.0.0.1	1
ipAdEntNetMask.10.100.47.55	255.255.240.0
ipAdEntNetMask.127.0.0.1	255.0.0.0
ipAdEntBcastAddr.10.100.47.55	1
ipAdEntBcastAddr.127.0.0.1	1
ipAdEntReasmMaxSize.10.100.47.55	65535
ipAdEntReasmMaxSize.127.0.0.1	65535

Figure 8.9: SNMP client with lots of data from Windows CE.

8.5 Wireshark

Wireshark is used extensively in the network chapters in this book, and 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 showing where the profiles are saved on your PC. Now create backups.
– It is normally important to show the "info" and coloring according to the highest level protocol; see "View"—"Coloring Rules."
– In "Preferences"—"Protocols" you should select IPv4 and then uncheck "Validate the IPv4 checksum." Do the same with TCP. Since validation is normally done in your network interface card, after Wireshark has seen the data, you will get false checksum errors unless this uncheck is done.
– In the main menu item "Analyze," there is an "Expert Info" item. This is a good place to start after a capture.
– In the main menu item "Statistics," there is a "TCP Stream Graphs" item, under which there are some very 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 we have an overview, you can, for example, select an 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 much 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, for example, the first 60 bytes of each frame. This will give the PC better performance.

8.6 Network commands

Table 8.2 is a list of the most general and usable network commands. You can go very far with this list.

Table 8.2: Network commands (Linux specific name in parentheses).

Command	Usage
arp	Show or edit the arp table (Known IP addresses and their MAC address) “-a” shows all “-s” can be used to add manually
ipconfig (ifconfig)	Show current netcard configurations. “/all” gives more info “/renew” causes a DHCP update
netstat	Show current TCP/UDP connections, their state and attached processes “-a” shows all – including listeners “-b” shows the executable (warning: slow!) “-n” uses numbers instead of names for ports “-o” shows process IDs
nslookup	DNS lookup. Shows the correspondence between IP addresses, names, etc.
ping	Simplest way to see if there is “hole through” to a remote host
route	Show or edit the route table (routes to networks and hosts) “PRINT” shows the table “ADD” allows adding
ssh	The modern “secure shell” replacement for telnet.
telnet	Client shell that redirects you keyboard commands to a remote device and see its output. Can be used with nondefault port numbers to do manual http etc. “Set localecho” is useful when simulating, for example, http
tracert (traceroute)	Trace the route to a given host. New packets are sent with increasing hopcount. This is decremented at each router as usual. When 0 a timing message is sent back home.

8.7 Further reading

- dhcpserver.de
  A free DHCP server for Windows
- angryip.org
  An IP scanner for Windows, Mac and Linux
- ireasoning.com
  Home of an SNMP browser (as well as an SNMP agent builder)
- Rich Seifert and Jim Edwards: *The All-New Switch Book*
  This is a fantastic brick of a book. If you read this, you will know all about switches, and then some. This book stays in the bottom two layers of the internet stack, which works very well.
- Wireshark.org
  The home of Wireshark.
- telerik.com
  Home of Fiddler. This is an analyzer for http only.

9 Wireless networks

9.1 Introduction

This chapter is mostly dealing with Wi-Fi, and secondly with Bluetooth. Before diving into these, we will look at the overall picture. You can structure the huge selection of wireless networks in many ways. In approximate descending order of link-distance:

– *Global*
  No system has a global wireless link that allows you to talk from anywhere to anywhere without some kind of relay station. Satellite phones are a good example. Here, we have an uplink from the transmitting satellite phone to the satellite, and a downlink to the receiver, typically connected to the normal telephone network (today part of the internet). Similarly, a mobile phone connects wirelessly to a *base station*, which via the internet is connected to the server you are addressing, or to another base station that completes the call to another mobile phone.
  "KNL Networks" (kyynel.net) has taken another approach. With the help of shortwave radio links and software defined radio, they provide internet to ships in the middle of the ocean. Over water, a single long link is necessary, while on land the distance of a single link is at maximum 600 km (375 miles). In this scenario, there are repeaters for longer distances.
– *LPWAN*
  Low power wide area networks are an interesting group of networks. They have a range of 15–50 km, which is comparable to mobile/cellular networks. The transfer rates are however just a small fraction of what is possible with mobile. The duty-cycle is typically also very low, resulting in as low as 1 kB data/day or even lower. Why is this suddenly interesting, when we now have high-definition video on our phones?
  Naturally, the key is low power consumption. These networks can run for a year or more on a single coin cell battery. As always, the sheer technical parameters are not the only interesting ones. Even more important is the way you become a member of the club, for example, the "Sigfox" company owns the technology of the same name, and is the only operator using it, so if you want Sigfox, you depend on them to make a network in your area. With "LoRa" you have the freedom to make your own network; however, Semtech is the only provider of the necessary LoRa transceiver chips.
  Most LPWANs use the ISM band. This is 868 Mzh in Europe, 908 MHz in the US and Canada, and similar in the rest of the world. This is pleasantly far away from 2.4/5 GHz used by Wi-Fi, and penetrates buildings a lot better. The major advantage of the ISM band is that it, like 2.4 GHz, requires no license, but since the ISM bands differ in the regions, you cannot use equipment from one region in another.

https://doi.org/10.1515/9781547401024-009

– *3G/4G*
  The original differences between the terms “cell phone” and “mobile phone” are sort of forgotten and they tend to mean the same. The “cell” term refers to the network. A given city is divided into cells with a base station which the phone connects to. The relatively short range inside a city is often perceived as a disadvantage, but it has an important upside. Since the air is essentially a shared medium, and the number of licensed frequency slots is limited, we would be in serious trouble if all phones had a reach of say 100 km. The relatively small cells allow the operator to reuse the same frequency over and over in a big city at the same time. If you want a device to use this network, the short range becomes a problem in rural areas.
  Another problem is the power usage that enforces mains-supply or large batteries. Finally, 3G/4G requires a SIM card giving access to the network, which is operated by whoever has bought the license. To a global device vendor, this means the need to negotiate numerous agreements with the various operators. E-SIM is an emerging standard that allows you to mount a chip at the factory that works globally. This is already used in Apple Watch Series 3.
– *Wi-Fi*
  The well-known Wi-Fi—IEEE 802.11—typically has a range of 30–100 m. The 2.4 GHz is crowded, but penetrates buildings a lot better than 5.0 GHz[1], where we have more channels. The 2.4 GHz band is free all over the world, while 5 GHz is only free in most of the world, with China being one of the major exceptions. Wi-Fi is extremely popular in homes and offices.
– *Bluetooth*
  Bluetooth also operates in the 2.4 GHz band, but not in the 5 GHz band. It uses a frequency-hopping scheme that makes it very tolerant to other networks, and vice versa. Contrary to all the aforementioned network types, Bluetooth originally could not be routed on the internet as it had no IP address. This changed with Bluetooth Low-Energy, facilitating the so-called 6LoWPAN. Bluetooth has a range comparable to Wi-Fi, but the transfer speed is much lower, especially when using BLE instead of Bluetooth Classic.
– *WPAN*
  IEEE 802.15.4 defines WPAN (wireless personal area networks). The range of these is comparable to that of Bluetooth and Wi-Fi. One of the best known is ZigBee, which applies a *mesh* technique, helping devices to act as routers for each other. In other words; a device may be to far away from the internet gateway itself, but can use devices closer to the gateway as stepping-stones. This allows for a longer range, but it also means that the devices “downstream” will use more power and need more bandwidth than those farther away. 6LoWPAN was created for this

1 Certain sub-bands in the 5 GHz band is for indoor use only.

group of networks, and thus also applies for ZigBee. ZigBee also uses the 2.4 GHz band.

– *Misc*
Z-Wave is a network, in many ways similar to ZigBee, but it is privately owned and not controlled in any IEEE standard. The basic Z-Wave cannot be routed, as it has no IP address, but also here there is a technology-patch, known as Z/IP. Z-Wave uses the ISM band, giving good penetration of walls, etc.
There is a large ecosystem of vendors with compatible products using Z-Wave. The products may be controlled via the cloud or via the “HomeSeer” application, which a user may run locally on, for example, a RaspBerry Pi.

Table 9.1 is a rough comparison of a broad selection of networks. The values stated must be taken with a grain of salt. A parameter such as the “Free Range” is typically exaggerated, and if it is possible to communicate anything at this range it will typically be at a very low rate.

Table 9.1: A selection of wireless networks.

Network	LoRa	SigFox	LTE 3/4G	Wi-Fi	BLE	ZigBee	Z-Wave
Free Range	15 km	50 km	35 km	80 m	200 m	10 m	30 m
Band	ISM	ISM	Licensed	2.4/5 GHz	2.4 GHz	2.4 GHz	ISM
Peak rate	50 kbps	1 kbps	10 Mbps	600 Mbps	2 Mbps	250 kbps	100 kbps
Good Rate	NA	NA	7 Mbps	150 Mbps	300 kbps	250 kbps	100 kbps
Application	City	City	Phones	LAN	Home	Home	Home
Proprietary	No	Yes	No	No	No	No	Yes
Topology	P2P	P2P	P2P	P2P	P2P	Mesh	Mesh
Sleep Power	15 uW	15 uW	20 mW**	10 uW	8 uW	4 uW	8 uW
Tx Power	200 mW	1 W	5 W	350 mW	25 mW	75 mW	75 mW
RF PHY	FSK	UNB	LTE	Many	Freq Hop	O-QPSK	GFSK
IP-Adress	Yes	Yes	Yes	Yes	Yes*	Yes*	No

*via 6WLoPan
**It really is mW

9.2 Wi-Fi basics

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

Many of the Wireshark captures in Chapter 7 were actually done on a PC connected to a Wi-Fi 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 *AP* (access point). All iPads, phones, PCs, and IoT devices on Wi-Fi, are known as *STA*s (stations). They all transmit on the same shared channel which means that only one AP or STA at a time can transmit. The whole interworking setup is a *basic service set* and similar systems nearby, that can interfere, are known as *overlapping basic service sets*.

The *BSSID* identifies the AP, and is the MAC address of the radio. The *SSID* (service set ID) identifies the WLAN and may be present on many APs. When an AP handles several SSIDs, each has a BSSID. Thus an AP is assigned as many MAC addresses as the number of SSIDs it can serve. All these terms are collected in Table 9.2.

Table 9.2: Parameters on application layer protocols.

Term	Explanation
Channel	Specific frequency
SSID	Service set ID
STA	Station. A device talking to an AP
AP	Access Point.
	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
	Neighbor on same channel
HT	High throughput defined in IEEE 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 channels 1, 6, and 11 at the same time. Wi-Fi is defined by IEEE 802.11, with the postfixed letters being very important. The first really used standard was 802.11*b*, then for some years 802.11*g*, and currently 802.11*n* and 802.11*ac* are where the action is, both including the 5 GHz band. If you have 802.11*b* equipment, you should get rid of it. It monopolizes the airtime as we shall see.

9.3 The access point as a repeater

The most common use of an access point is access to the internet; hence the name. SOHO (small office/home office) routers have a single wired connection to the internet, often marked *WAN* (wide area network), while offering local connection via Wi-Fi to a number of devices. Typically, these routers also have a small number of wired Ethernet ports, working like a switch connected to a 2-port router. This was shown earlier in Figure 7.2.

The users of iPads, phones, or PCs rarely see a need to interconnect these devices, they just need access to the internet. However, in a home or office you often need to connect to a printer or a NAS disk with movies or backups, and music systems like Sonos. The optimal way to connect these devices is via the wired Ethernet ports in the router, but this is not always possible. There is also a growing need to connect STAs like an iPad to wireless IoT devices like air quality measurement, heating, lights, etc. Such devices are also STAs. Thus STAs need to communicate with each other.

In the following, we will study the scenario where the AP (router) helps the STAs (devices) to communicate with each other. This gives a good background for understanding the possibilities and limitations in the Wi-Fi technology. The connection from a single wireless STA to the internet is basically a subset of this scenario.

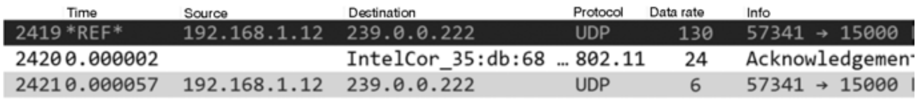
When we have an access point, we operate in infrastructure mode—the normal way to use Wi-Fi. On a wired network, nodes can communicate directly or through a switch—using unicasts, multicasts, or broadcasts. In a standard Wi-Fi infrastructure system, however, all communication must use the access point as a stepping stone. You may think: access point or switch—what's the difference? There are two major differences:

- In a wired switch, traffic can go in and out of all ports at the same time (if the switch is good). In the Wi-Fi scenario, the air is a shared medium, thus only one STA or the AP should "talk" at any given time. This is much like older Ethernet coax networks, or even classic radio communication sharing a single channel - forcing users to end all contributions with "over" (and "Roger" for acknowledge).
- In a wired switch, traffic is switched at layer 2 (Ethernet), based on MAC addresses. This is completely transparent, requiring no changes to the Ethernet frames. In the Wi-Fi case, we use a router, which means that traffic is routed at layer 3. Layer 2 traffic—the radio traffic using the 802.11 protocols—is opened by the router and repackaged into new layer 2 traffic.

We will now look at the multicast scenario, using Wireshark with "AirCap," a device that catches wireless traffic and gives information about the 802.11 frames as we shall see.

A STA "wants" to send a multicast frame to all other STAs in the basic service set. It does this by sending a message to the AP, which the AP then multicasts to all STAs; see Figure 9.1.

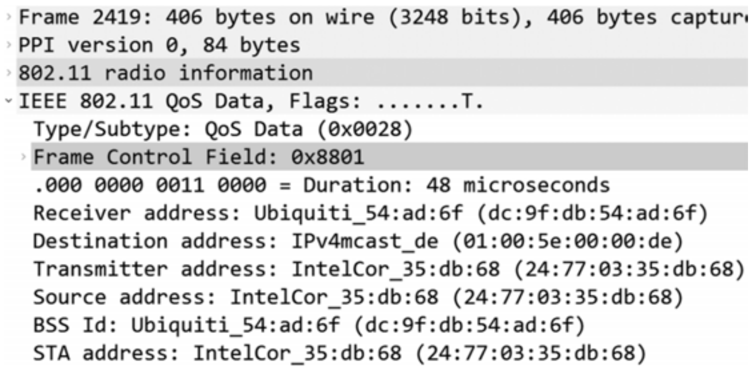
When one STA is sending anything, it is by definition to the AP, and the other STAs go into *sleep mode*, for the duration of the transmission, to save power. To help with



```
     Time        Source          Destination           Protocol  Data rate  Info
2419 *REF*       192.168.1.12    239.0.0.222           UDP       130        57341 → 15000
2420 0.000002                    IntelCor_35:db:68 …   802.11    24         Acknowledgemen
2421 0.000057    192.168.1.12    239.0.0.222           UDP       6          57341 → 15000
```

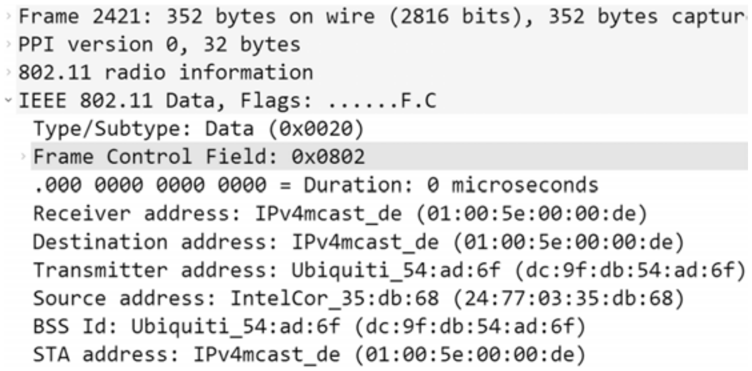
Figure 9.1: Multicast on Wi-Fi is two-fold.

this, there is generally information in all frames about the expected duration—we will see more of this in Section 9.9. Frame 2419 in Figure 9.1 is from the STA to the AP, while frame 2421 is the same data from the AP to all STAs. When setting up AirPcap, *PPI* (per-packet-information) was chosen which we see in the dissections of the two frames in Figures 9.2 and 9.3.



```
› Frame 2419: 406 bytes on wire (3248 bits), 406 bytes captur
› 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)
```

Figure 9.2: Frame 2419 from STA to AP.



```
› Frame 2421: 352 bytes on wire (2816 bits), 352 bytes captur
› 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)
```

Figure 9.3: Frame 2421 from AP to STAs.

Analyzing the two dissected frames tells us the following:

1. Frame 2419 is sent from our STA at 130 Mbps, which is an 802.11*n* speed. This is possible because the STA knows that the only receiver—the AP—can handle this speed (we will see later how it knows).
2. Frame 2421 is sent from the AP at 6 Mbps, an 802.11*g* speed—the lowest common denominator, assuring that older g devices can understand it.
   It would be even worse if 802.11*b* devices were present. Often the AP can be setup to be in *legacy mode*, where it complies with the old 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 seen. 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.[2]

3. When you look at wireless traffic in Wireshark, one of the first things you notice is how transmission speeds may change from one frame to the next, even if sender and receiver are unchanged. The next frame from the STA to the AP might be slower than the 130 Mbps, if noise is causing too many errors.
4. When right-clicking a line in Wireshark's overview window you get a pop-up menu. Here, frame 2419 is selected as a time reference. The following time stamps are now relative to this. Note that several REFerence stamps are allowed in the same capture. Be careful when reading a time difference: there could be an intermediate REF that you have missed. Here, we see that the time from when frame 2419 is sent and until frame 2421 is sent is 57 μs. Sending the 352 bits at 6 Mbps theoretically takes 58.7 μs.
5. The frames are identical on the UDP level as well as the IP level. The STA sent a normal multicast at these levels. However, on the 802.11 level, only frame 2421 is a multicast.
6. Both frames note the MAC source as IntelCor_35:db:68 and the MAC destination as the multicast address 01:00:5e:00:00:de.
7. There is a BSS ID which in both frames is ubiquiti_54:ad:6f—the MAC address of the access point.
8. Frame 2419 has a "Transmitter address," which is the same as its MAC source, IntelCor_35:db:68, and a "Receiver address," which is the access point (ubiquiti_54:ad:6f).
9. Frame 2421 has a "Transmitter address" which is the same as the AP's MAC, ubiquiti_54:ad:6f and a "Receiver address"—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 MAC addresses and "to DS" and "from DS" bits. Luckily, Wireshark interprets this for us—see Table 9.3.

In other words, Wireshark helps us keep the original "source" and "destination" terms, and is using "receiver" and "transmitter" for the participants on the radio level.

Table 9.3: Wireshark terms for Wi-Fi nodes.

Term	Explanation
Source	MAC of original source
Destination	MAC of original destination
Transmitter	MAC of transmitting radio now
Receiver	MAC of receiving radio now

2 These terms may vary between vendors.

9.4 How is speed calculated?

In Section 9.3, we saw different speeds listed by Wireshark. But how are these obtained? This is a very advanced subject—we will focus on the main parameters:

– *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 describes the number of states each transmitted symbol can have. A bit only has two states but, for example, 64-QAM has 64 states per symbol corresponding to 6 bits. The coding rate is the "true rate" of information transmitted when error-correction is accounted for. This information includes the original payload, as well as any headers added by the other layers.
– *Bandwidth*
  When we say that there are 11 channels in the 2.4 GHz range, each is 20 MHz wide. It is possible to combine two into a 40 MHz wide 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 the AirPcap used only supports the normal guard.
– *Subcarriers*
  As discussed, we may have a number of channels in, for example, 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 channels 1, 6, and 11 undisturbed. Using *OFDM* (orthogonal frequency division multiplexing), each channel is split into a number of subchannels, or rather subcarriers. Each of these is synthesized and decoded using respectively *iFFT* and *FFT* ([inverse] fast Fourier transform) technology.

If we have an MCS index of 7, a long guard interval and a 20 MHz channel, the data rate can be calculated as in Table 9.4.

Table 9.4: Wi-Fi data rate calculation.

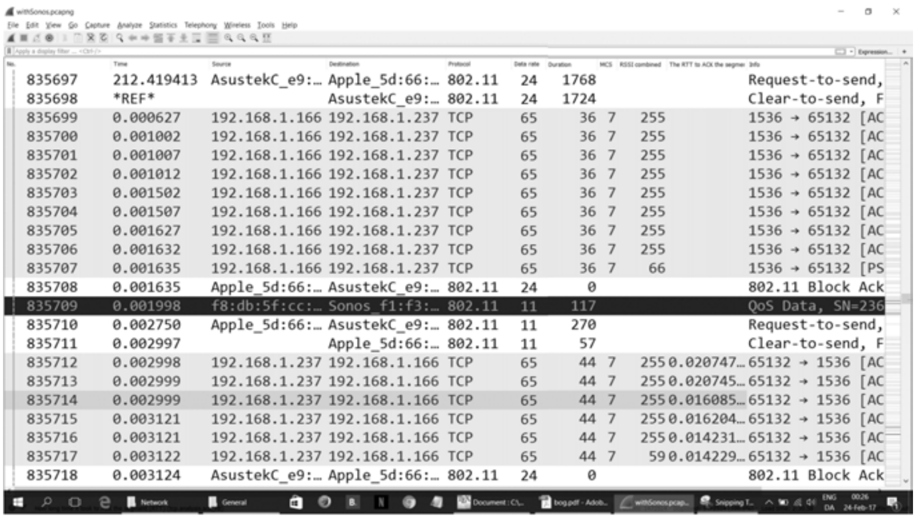
Factor	Value
Spatial streams	1 Flow
64-QAM = 64 states/symbol	6 bits/symbol
Time per symbol 4 μs incl. long guard	250 k/s
Subcarriers in 20 MHz	52
Left after forward correction	5/6
Total: 1 ∗ 6 bit ∗ 250 k/s ∗ 52 ∗ 5/6	65 Mbit/s

A short guard will cut off 0.4 µs of the 4 µs, increasing the speed by 11 %. You can find tables of MCS indexes at Wikipedia and other places. These are built into Wireshark.

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

9.5 Case: Wi-Fi data transmission

Figure 9.4 shows a scenario with a data-producing device,[3] wired to a small Asus router.


withSonos.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.     Time        Source            Destination       Protocol Data rate Duration MCS RSSI combined The RTT to ACK the segmer Info
835697  212.419413  AsustekC_e9:…     Apple_5d:66:…     802.11   24    1768                       Request-to-send,
835698  *REF*                         AsustekC_e9:…     802.11   24    1724                       Clear-to-send, F
835699  0.000627    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835700  0.001002    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835701  0.001007    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835702  0.001012    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835703  0.001502    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835704  0.001507    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835705  0.001627    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835706  0.001632    192.168.1.166     192.168.1.237     TCP      65      36  7   255              1536 → 65132 [AC
835707  0.001635    192.168.1.166     192.168.1.237     TCP      65      36  7    66              1536 → 65132 [PS
835708  0.001635    Apple_5d:66:…     AsustekC_e9:…     802.11   24       0                       802.11 Block Ack
835709  0.001998    f8:db:5f:cc:…     Sonos_f1:f3:…     802.11   11     117                       QoS Data, SN=236
835710  0.002750    Apple_5d:66:…     AsustekC_e9:…     802.11   11     270                       Request-to-send,
835711  0.002997                      Apple_5d:66:…     802.11   11      57                       Clear-to-send, F
835712  0.002998    192.168.1.237     192.168.1.166     TCP      65      44  7   255 0.020747…    65132 → 1536 [AC
835713  0.002999    192.168.1.237     192.168.1.166     TCP      65      44  7   255 0.020745…    65132 → 1536 [AC
835714  0.002999    192.168.1.237     192.168.1.166     TCP      65      44  7   255 0.016085…    65132 → 1536 [AC
835715  0.003121    192.168.1.237     192.168.1.166     TCP      65      44  7   255 0.016204…    65132 → 1536 [AC
835716  0.003121    192.168.1.237     192.168.1.166     TCP      65      44  7   255 0.014231…    65132 → 1536 [AC
835717  0.003122    192.168.1.237     192.168.1.166     TCP      65      44  7    59 0.014229…    65132 → 1536 [AC
835718  0.003124    AsustekC_e9:…     Apple_5d:66:…     802.11   24       0                       802.11 Block Ack
```

Figure 9.4: Full speed ahead on Wi-Fi.

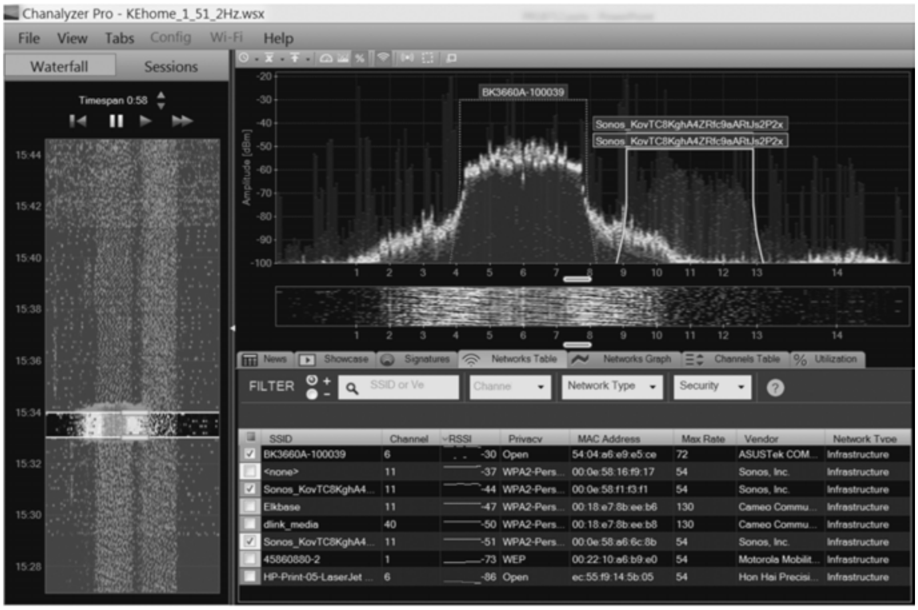
Via this, data is transmitted to a "pre-Air" iPad. The recording is made with AirPcap, using the PPI header from the 802.11 layer. A fantastic feature in Wireshark is that when you "open up" the dissector, you can right-click on an interesting value and select "Show as Column." In Figure 9.4, this has been done with all the columns shown between "protocol" and "info": Data Rate, Duration (µs), MCS Index, RSSI, and RTT.

Let's walk through the case:

– There is a little communication going on with a Sonos music system running on another AP—seen in frame 835709 and colored black. It does not seem to cause problems.

3 a LAN-XI module described in Section 4.9.

– All TCP communication happens within aggregated frames. Aggregated frames are a very important part of the newer wireless standards. As we shall see later (Section 9.8), there is a lot of overhead before a transmission can be started. It is therefore of great value to "keep going," once started.
  Disregarding the two first lines, we first have a block of aggregated TCP data in 9 frames with a total of 13890 ($8 * 1622 + 914$ in hidden column) 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. Based on earlier wired analysis of the same scenario, we know these are all ACKs. 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. This we know is 20 MHz band (from Table 9.4) and long guard (SGI—Short Guard Interval—false). The TxRate shows 65 Mbps which is consistent with the calculations in Section 9.4.
– 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 values are not so far apart.
– Before each datatransmission we see a *CTS* (clear-to-send) with a value in μs 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 CTS as REFerence, it is easy to check this estimated duration against the actual duration of the aggregated transmission. Section 9.9 goes into more details on this. The estimate is best for longer durations.
– The CTS is an answer to an RTS (request-to-send). When the iPad is about to send its ACKs, we see an RTS from the iPad to the Asus in frames 835710 and 835719 (outside the view), 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.
  Imagine, for example, that a neighboring network is so far from the Asus router that neither of the access points can see each other, nevertheless, one or more STAs in one network will experience interference from the neighbor AP or STAs. This is known as the *hidden terminal problem*. In such a case, the neighbor STA will still see the CTS or the RTS, and will understand when to keep quiet.
– 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 STAs 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 BSSs that this channel is going to be taken, partly to tell other



	SSID	Channel	RSSI	Privacy	MAC Address	Max Rate	Vendor	Network Type
✓	BK3660A-100039	6	-30	Open	54:04:a6:e9:e5:ce	72	ASUSTek COM...	Infrastructure
	<none>	11	-37	WPA2-Pers...	00:0e:58:16:f9:17	54	Sonos, Inc.	Infrastructure
✓	Sonos_KovTC8KghA4...	11	-44	WPA2-Pers...	00:0e:58:f1:f3:f1	54	Sonos, Inc.	Infrastructure
	Elkbase	11	-47	WPA2-Pers...	00:18:e7:8b:ee:b6	130	Cameo Commu...	Infrastructure
	dlink_media	40	-50	WPA2-Pers...	00:18:e7:8b:ee:b8	130	Cameo Commu...	Infrastructure
✓	Sonos_KovTC8KghA4...	11	-51	WPA2-Pers...	00:0e:58:a6:6c:8b	54	Sonos, Inc.	Infrastructure
	45860880-2	1	-73	WEP	00:22:10:a6:b9:e0	54	Motorola Mobilit...	Infrastructure
	HP-Print-05-LaserJet ...	6	-86	Open	ec:55:f9:14:5b:05	54	Hon Hai Precisi...	Infrastructure

Figure 9.5: Wi-Fi spectrum of our transmission.

STAs on our own network, that they might as well take a nap for the duration of the transmission, and thus save energy.

Figure 9.5 shows another tool applied—Chanalyzer with Wi-Spy from MetaGeek. We are measuring on channel 6, on the SSID called BK3660A-100039.

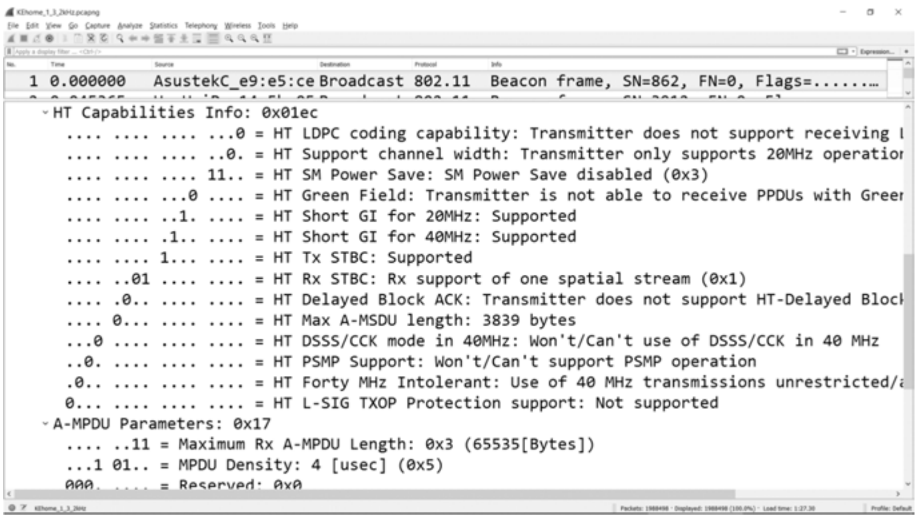
The information about the Wi-Fi SSIDs in the bottom table comes from the PCs wireless NIC. Based on this info, the tool draws a "profile" of a given network, the dotted line, over the actual measurement.

Apart from looking very fancy, this is actually a nice tool. We can see a lot of stuff on Wireshark, but it is all above the physical layer. You may have problems with connections that only show in Wireshark as retransmissions. Noise on the physical level from, for example, a microwave oven, a wireless mouse, or an infrared movement detector, may interfere. This can be seen with a tool such as Wi-Spy. The tool includes "standard signatures" of known noise sources. This may help find the culprit.

9.6 Case: beacons

We are not completely done with the transmission in the previous section. It is interesting to know why we ended up with MCS index 7, and why only 20 MHz Bandwidth, as the Asus AP should be able to support *channel bonding* where two 20 MHz channels are joined into one 40 MHz channel. Even in 20 MHz mode we should be able to get 72 Mbps according to Figure 9.5.

Access points are periodically (typically every 100 ms) emitting so-called *beacons*. These are broadcasts sent on a slow and robust data rate. This ensures that they are seen by all STAs in the vicinity, even older types. An important part of this broadcast is the SSID. This allows you to choose between networks. The AP also transmits a lot of information on its capabilities. Figure 9.6 shows the beacon from the Asus AP.


KEhome_1_3_2kHz.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time       Source            Destination Protocol Info
  1  0.000000   AsustekC_e9:e5:ce Broadcast   802.11   Beacon frame, SN=862, FN=0, Flags=.......…
```

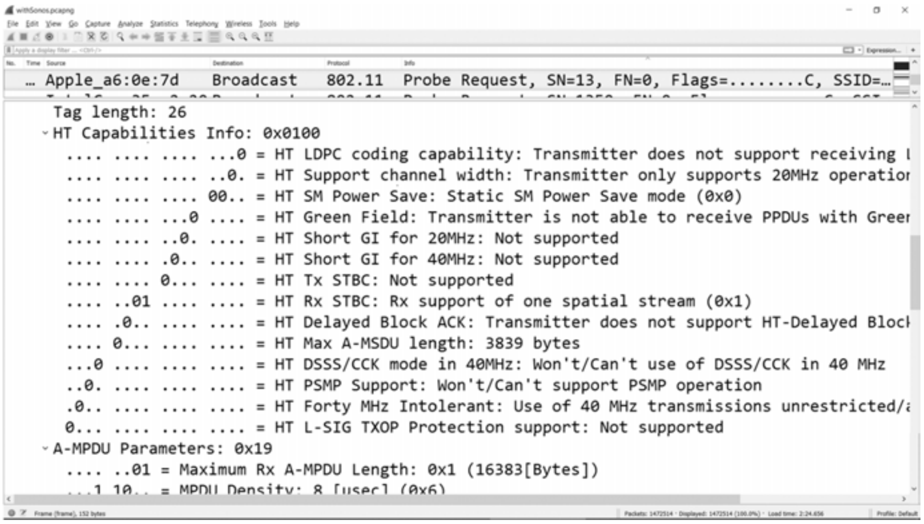
```
  ˅HT Capabilities Info: 0x01ec
     .... .... .... ...0 = HT LDPC coding capability: Transmitter does not support receiving
     .... .... .... ..0. = HT Support channel width: Transmitter only supports 20MHz operatio
     .... .... .... 11.. = HT SM Power Save: SM Power Save disabled (0x3)
     .... .... ...0 .... = HT Green Field: Transmitter is not able to receive PPDUs with Gree
     .... .... ..1. .... = HT Short GI for 20MHz: Supported
     .... .... .1.. .... = HT Short GI for 40MHz: Supported
     .... .... 1... .... = HT Tx STBC: Supported
     .... ..01 .... .... = HT Rx STBC: Rx support of one spatial stream (0x1)
     .... .0.. .... .... = HT Delayed Block ACK: Transmitter does not support HT-Delayed Bloc
     .... 0... .... .... = HT Max A-MSDU length: 3839 bytes
     ...0 .... .... .... = HT DSSS/CCK mode in 40MHz: Won't/Can't use of DSSS/CCK in 40 MHz
     ..0. .... .... .... = HT PSMP Support: Won't/Can't support PSMP operation
     .0.. .... .... .... = HT Forty MHz Intolerant: Use of 40 MHz transmissions unrestricted/
     0... .... .... .... = HT L-SIG TXOP Protection support: Not supported
  ˅A-MPDU Parameters: 0x17
     .... ..11 = Maximum Rx A-MPDU Length: 0x3 (65535[Bytes])
     ...1 01.. = MPDU Density: 4 [usec] (0x5)
     000. .... = Reserved: 0x0
```

Figure 9.6: Beacon from the Asus AP.

The “HT Capabilities Info” is the relevant section here; “HT” for “high throughput,” which is related to the 802.11*n* standard. The figure shows that the Asus AP says that it only supports 20 MHz operation, but why does it then claim support for “Short Guard” on 40 Mhz? It turns out that it actually *can* do the 40 MHz operation, but in this session this feature was disabled in its setup.

STAs may do *probe requests* where they advertise their capabilities. Not a single probe request was caught in this recording. A STA may run in *passive mode* where it does not emit probe requests, but simply connect to the AP with a common denominator of the two parties capabilities.

The iPad used at the time did not support neither 40 MHz channels or “short guard,” so the resulting traffic ends with a single 20 MHz channel and long guard, giving us an MCS index of 7 and from this the 65 Mbps speed. AirPcap does not support the short guard interval either. If both devices had supported the short guard interval, we would not have been able to follow the transmissions. In order to debug anyway, we would simply disable short guard on one of the devices. Figure 9.7 shows



```
… Apple_a6:0e:7d   Broadcast   802.11   Probe Request, SN=13, FN=0, Flags=........C, SSID=…
    Tag length: 26
  ˅HT Capabilities Info: 0x0100
     .... .... .... ...0 = HT LDPC coding capability: Transmitter does not support receiving L
     .... .... .... ..0. = HT Support channel width: Transmitter only supports 20MHz operatior
     .... .... .... 00.. = HT SM Power Save: Static SM Power Save mode (0x0)
     .... .... ...0 .... = HT Green Field: Transmitter is not able to receive PPDUs with Greer
     .... .... ..0. .... = HT Short GI for 20MHz: Not supported
     .... .... .0.. .... = HT Short GI for 40MHz: Not supported
     .... .... 0... .... = HT Tx STBC: Not supported
     .... ..01 .... .... = HT Rx STBC: Rx support of one spatial stream (0x1)
     .... .0.. .... .... = HT Delayed Block ACK: Transmitter does not support HT-Delayed Block
     .... 0... .... .... = HT Max A-MSDU length: 3839 bytes
     ...0 .... .... .... = HT DSSS/CCK mode in 40MHz: Won't/Can't use of DSSS/CCK in 40 MHz
     ..0. .... .... .... = HT PSMP Support: Won't/Can't support PSMP operation
     .0.. .... .... .... = HT Forty MHz Intolerant: Use of 40 MHz transmissions unrestricted/a
     0... .... .... .... = HT L-SIG TXOP Protection support: Not supported
  ˅A-MPDU Parameters: 0x19
     .... ..01 = Maximum Rx A-MPDU Length: 0x1 (16383[Bytes])
     ...1 10.. = MPDU Density: 8 [usec] (0x6)
```

Figure 9.7: Probe from the iPad STA.

the probe request from a similar unconnected iPad. Beacons and probes are also used when deciding which encryption to use; see Section 10.15.

9.7 Case: a strange lagging

The setup 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 9.8 was created.

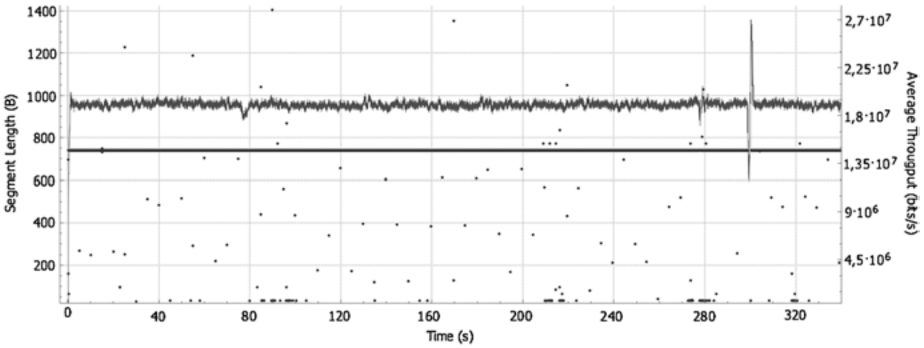


Figure 9.8: Wireshark throughput graph.

The top jagged line at almost 20M bit/s (y-axis is on the right of this, not easy to read) is data in the direction where data flows. The more steady line just below, is the segment length in bytes, which is not so relevant in this discussion (y-axis to the left). It is clear that something does happen almost exactly 5 minutes after starting the traffic and the Wireshark caption, as the hard-to-read neighboring main grid lines are at 280 s and 320 s. The area of the "dive" matches the area of the peak shortly after, meaning that the transmission catches up, and no data is lost. However, the dive verifies the "hesitation" experienced in the application. Making the similar graph on RTT (round-trip time) in Figure 9.9 verified the lagging.

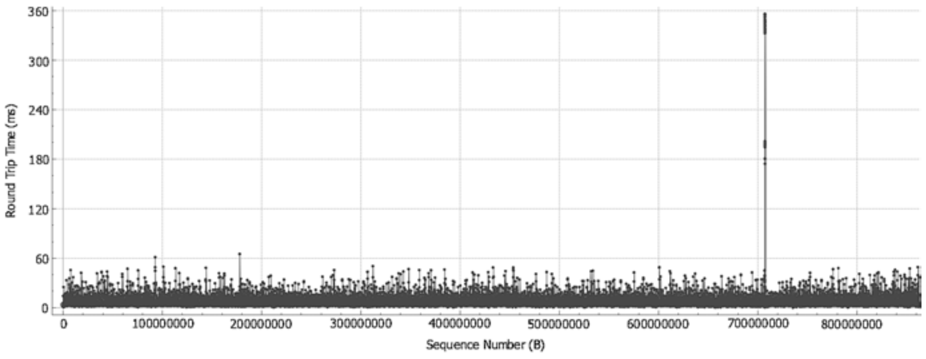


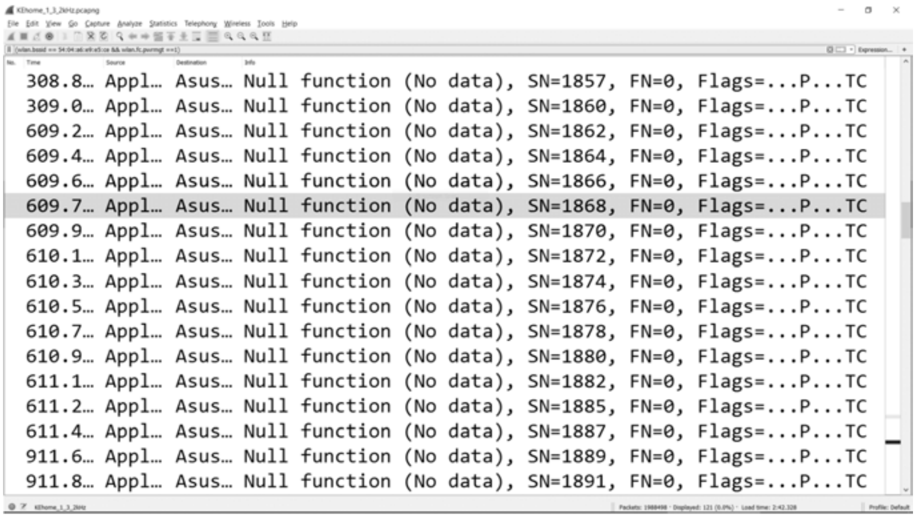
Figure 9.9: Wireshark RTT graph.

Here, the RTT goes from less than 60 ms to 360 ms for a short duration. The x-axis here is hard to read, but the closest main grid line says 700,000,000 Bytes. With a steady traffic at just below 20 Mbps, this grid line, shortly before the peak, is at a little more than (700,000,000 Bytes ∗ 8 bits/Byte)/20 Mbps = 280 s.

Browsing the standards, it became clear that according to 802.11n all STAs are required to scan all Wi-Fi channels every 300 seconds, and report their findings back to the AP. The AP collects this information and spreads it to all STAs so that they can update their "Navigation Vector." This assists in the handling of hidden terminals. Knowing what to look for now, Figure 9.10 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.

These power naps can effect any application. In this scenario, where data is continuously streamed, we see it clearly on the screen as a hesitation. Had we used a higher data rate we would have lost data. In more normal scenarios, it means that the maximum latency is longer than what a simple measurement suggests.



```
308.8… Appl… Asus… Null function (No data), SN=1857, FN=0, Flags=...P...TC
309.0… Appl… Asus… Null function (No data), SN=1860, FN=0, Flags=...P...TC
609.2… Appl… Asus… Null function (No data), SN=1862, FN=0, Flags=...P...TC
609.4… Appl… Asus… Null function (No data), SN=1864, FN=0, Flags=...P...TC
609.6… Appl… Asus… Null function (No data), SN=1866, FN=0, Flags=...P...TC
609.7… Appl… Asus… Null function (No data), SN=1868, FN=0, Flags=...P...TC
609.9… Appl… Asus… Null function (No data), SN=1870, FN=0, Flags=...P...TC
610.1… Appl… Asus… Null function (No data), SN=1872, FN=0, Flags=...P...TC
610.3… Appl… Asus… Null function (No data), SN=1874, FN=0, Flags=...P...TC
610.5… Appl… Asus… Null function (No data), SN=1876, FN=0, Flags=...P...TC
610.7… Appl… Asus… Null function (No data), SN=1878, FN=0, Flags=...P...TC
610.9… Appl… Asus… Null function (No data), SN=1880, FN=0, Flags=...P...TC
611.1… Appl… Asus… Null function (No data), SN=1882, FN=0, Flags=...P...TC
611.2… Appl… Asus… Null function (No data), SN=1885, FN=0, Flags=...P...TC
611.4… Appl… Asus… Null function (No data), SN=1887, FN=0, Flags=...P...TC
911.6… Appl… Asus… Null function (No data), SN=1889, FN=0, Flags=...P...TC
911.8… Appl… Asus… Null function (No data), SN=1891, FN=0, Flags=...P...TC
```

Figure 9.10: iPad power nap—13 breaks every 300 s.

9.8 Aggregated frames

As smarter types of coding were developed, it became 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. This is exactly what is done. This is called aggregation and in Section 9.5 we saw "aggregated frames."

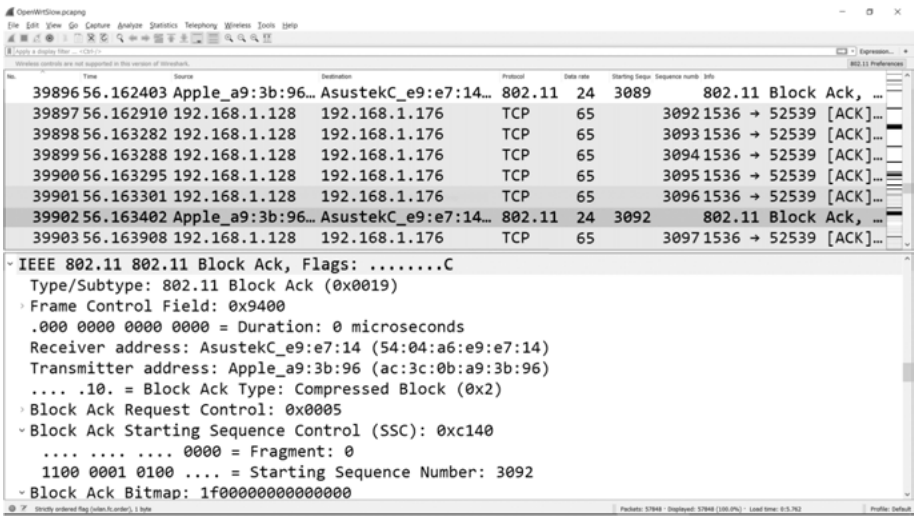
The thing that is aggregated is PDU (protocol data units). A PDU is loosely stated the "packet" at any level. As stated earlier the correct term for a PDU in Ethernet is a "Frame," while in IP it is a "Packet" and in TCP the PDU is a "Segment." At the application level, the PDU is a "Message" or simply "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 9.5. In a later test with similar hardware as before, but this time using OpenWRT firmware loaded into the same type of Asus router, a capture was done—see Figure 9.11—which is the case for this and the next section. OpenWRT is described in Section 6.10.

Figure 9.11 shows the following:

– Data is basically sent from 192.168.1.128, which is wired to the access point, the Asus, which is wirelessly connected to an iPad. In other words, the same setup as previously studied.
– Wireshark is setup to show the A-MPDU sequence numbers by looking in the dissection of one frame and clicking "Apply as Column." Similarly, "Data Rate"



```
No.   Time      Source           Destination        Protocol Data rate Starting Seque Sequence numb Info
39896 56.162403 Apple_a9:3b:96…  AsustekC_e9:e7:14… 802.11   24        3089               802.11 Block Ack, …
39897 56.162910 192.168.1.128    192.168.1.176      TCP      65                      3092 1536 → 52539 [ACK]…
39898 56.163282 192.168.1.128    192.168.1.176      TCP      65                      3093 1536 → 52539 [ACK]…
39899 56.163288 192.168.1.128    192.168.1.176      TCP      65                      3094 1536 → 52539 [ACK]…
39900 56.163295 192.168.1.128    192.168.1.176      TCP      65                      3095 1536 → 52539 [ACK]…
39901 56.163301 192.168.1.128    192.168.1.176      TCP      65                      3096 1536 → 52539 [ACK]…
39902 56.163402 Apple_a9:3b:96…  AsustekC_e9:e7:14… 802.11   24        3092               802.11 Block Ack, …
39903 56.163908 192.168.1.128    192.168.1.176      TCP      65                      3097 1536 → 52539 [ACK]…
```

```
IEEE 802.11 802.11 Block Ack, Flags: ........C
  Type/Subtype: 802.11 Block Ack (0x0019)
  Frame Control Field: 0x9400
  .000 0000 0000 0000 = Duration: 0 microseconds
  Receiver address: AsustekC_e9:e7:14 (54:04:a6:e9:e7:14)
  Transmitter address: Apple_a9:3b:96 (ac:3c:0b:a9:3b:96)
  .... .10. = Block Ack Type: Compressed Block (0x2)
  Block Ack Request Control: 0x0005
  Block Ack Starting Sequence Control (SSC): 0xc140
    .... .... .... 0000 = Fragment: 0
    1100 0001 0100 .... = Starting Sequence Number: 3092
  Block Ack Bitmap: 1f00000000000000
```

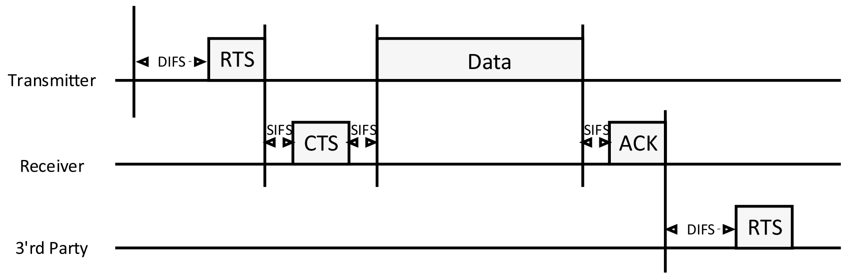
Figure 9.11: A-MPDUs and their contents.

and "Starting Sequence number" in the Block Acknowledge are chosen as columns.

– Frames 39897–39901 are in reality a single 802.11 A-MPDU with sequence numbers 3092–3096, in other words 5 MPDUs.
– Frame 39902 is an 802.11 block acknowledge and as it is the selected frame, it is dissected in the middle window. We see that 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. Thus we have a quick answer—a short RTT.
– Just below the "1f000…" (not visible), Wireshark states: "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 necessarily a statement about lost frames, only frames not seen yet.
– Data is sent at 65 Mbps with PHY type 802.11n because the Asus and the iPad have agreed that this is fine. The block ACKs are however, sent at 24 Mbps PHY type 802.11g. This is because the configuration 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 performance issue to send these relatively short frames at a third of the possible speed.

9.9 Channel assessment

We have seen the use of request-to-send and clear-to-send several times. The terms are ancient, dating back to at least RS-232, but the technology behind RTS/CTS in modern



DIFS = Distributed Interframe Space
SIFS = Short Interframe Space
RTS = Request-To-Send (incl. Data Transmission Time)
CTS = Request-To-Send (incl. Data Transmission Time)

Figure 9.12: Channel assessment with RTS and CTS.

Wi-Fi is much more sophisticated. 802.11n does not need these if it is alone in the world (greenfield), but the modern variant has proven very robust; see Figure 9.12.

The RTS is sent by the STA transmitter-to-be, to the AP after a period of silence in the air 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. This is just like classic Ethernet coax, where we also had a shared medium.

The RTS contains the expected transmission time, and the address of the receiver. This allows third 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 third parties (hidden terminal problem). Once we have a designated transmitter-receiver pair, the "dead-time" between frames can be shorter—resulting in higher transmission speed. This shorter dead-time is called *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.

9.10 Bluetooth low energy

Some years ago a war was fought between Wi-Fi and Bluetooth about being the preferred wireless connection. One of the main reasons why Wi-Fi won the war, is that 802.11 (Wi-Fi) is built on top of the existing Ethernet protocols. This gives the connectivity needed. With the assistance of a cheap Wi-Fi router connected to an existing network, the PCs, and later the phones and tablets, could connect to any server in the world.

Since then, Wi-Fi has been all about performance, getting more and more bytes across. Naturally, power savings has also been interesting as Wi-Fi is heavily used on laptops, but when, for example, 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 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 your phone. In the “small device” end of the market, Bluetooth is thus very 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. Wi-Fi is weaker than Bluetooth on the power side. This is suddenly very relevant, since IoT devices may not be connected to power as often as laptops and phones. Bluetooth Classic still really hasn’t got the connectivity, and is way behind on performance. On the other hand, Bluetooth Low Energy does have the connectivity. It is not created for long lasting peer-to-peer connections like Bluetooth Classic. It connects fast and tears down fast, much like UDP.

Table 9.5 is a comparison of Bluetooth Classic and BLE (Bluetooth Low Energy) a.k.a. Bluetooth Smart.

Table 9.5: Bluetooth Classic versus BLE.

Feature	Classic	BLE
Band	2.4 GHz	2.4 GHz
Distance	30 m	50 m
Data rate	2100 kByte/s	260 (650) kByte/s
Tx power max	100 dBm	10 dBm
Peak current max	30 mA	15 mA
Sleep current max	–	1 μA
Broadcast/beacon concept	No	Yes
Connect up+down	300 ms	3 ms

The data rates in Bluetooth Low Energy 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 up to 257 bytes. This explains why the theoretical max rate is given as two numbers in Table 9.5. The broadcast concept is new with BLE. Apple has an iBeacon concept which only broadcasts, and never does anything else. This is to be used in, for example, a department store, or a museum, where you walk around with your iPhone and catch different offers, anecdotes, or news.

Bluetooth 5 was announced very late in 2016, without the expected *mesh* technology, which came a little later. It boasts double speed or range (sometimes even more) due to new PHYs with more advanced coding of the symbols. It appears that the double speed can be obtained without using more energy than 4.2. Thus Bluetooth 5 may save energy.

The war between Wi-Fi and Bluetooth may be on again, but the rules have changed. The wireless IoT devices we hear about now, 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 do 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 some added means of connectivity; see Table 9.6.

Table 9.6: Bluetooth Smart connectivity.

Name	What it is	Usage
GATT	REST API	Server
HTTP	Proxy service	Client
6LoWPAN	IPv6 tunnel	Client and 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 (see Table 9.6) solution is for BLE devices, 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, for example, via Wi-Fi, 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. Via this, the device may check temperature, turn on heating, call up 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. 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 using 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 addendum 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.

9.11 Certification

The 2.4 GHz band is used all over the world for unlicensed radio traffic such as Wi-Fi and Bluetooth. The fact that this band is unlicensed, does not mean that it is unregulated. If a device is to be used in a given country, it must be certified for this country. With a single certification from the FCC[4] the US is open, along with a number of smaller countries that accept an FCC certification. There is a similar certification allowing usage in the EU countries. In the US and EU, a "DoC" (document of compliance) can be filled out by the manufacturer and submitted to the authorities. In this document, the manufacturer states the compliance of the product to the relevant standards, backed up by documentation. China, Brazil, and a few other countries demand an "in-country" certification—meaning that the measurement has to be performed by government appointed test-facilities inside the country.

Since June 13, 2017, the "Radio Equipment Directive"—(RED)—is enforced in the EU. All equipment marketed after this date must abide to the RED. The major new thing in RED, compared to the old rules, is that not only transmitters, but also receivers are regulated. As we have seen in this chapter: if a transmitter experiences too many transmission errors, it will select a more robust modulation type and lower the transmission speed. This means that a badly designed receiver will occupy the radio for a longer time.

The RED requires efficiency on both parties in order to get the overall best utilization of our shared medium. In an effort to make life simpler, the RED not only regulates the wireless performance but also takes the place of the EMC directive and the low-voltage directive, whenever there is a radio involved.[5] As other directives, RED only speaks in plain language about requirements, and defers all the technical stuff to a "Harmonised Standard" called "ETSI 300 328." This standard specifies the actual conformance tests (and results) needed to obtain a certificate. It has two main parts, one for frequency hopping (Bluetooth), and one for static channels (Wi-Fi and, e. g., Zigbee). The actual EU certification is a question of filling out the Document of Compliance, submitted together with documentation for the various measurements. In theory, you can perform these measurements yourself, if you have the lab, but most companies will use a commercial lab for these measurements.

In the US, the major sections are FCC § 15.247 for 2.4 GHz and FCC § 15.407 for 5 GHz. The latter contains some rules on *DFS* (dynamic frequency selection) in order to avoid interference with weather radar. The FCC rules do not yet mention receiver efficiency. In the EU as well as the US, a so-called "SAR" testing is needed if the device is handheld, or otherwise in close contact with human bodies.

If the device is based on a "precertified" wireless module, life becomes a lot easier. This module has been tested and documented on all digital and protocol related

4 Federal Communications Commission.

5 Thus even a GPS receiver is covered by the RED.

parameters such as, for example, SIFS and DIFS described in Section 9.9. Building the module into a box and adding an antenna does not change these specs. However, it is necessary to measure emission and input sensitivity.

Modern Wi-Fi also uses 5 GHz, and this is where things get complicated. Some countries, for example, China, does not allow for the use of 5 GHz. The European RED refers to "ETSI 301 893" for 5 GHz equipment.

9.12 Further reading

- Perahia and Stacey: *Next Generation Wireless LANs*
  Dry—but the best I have found.
- Laura Chappel: *Wireshark Network Analysis*
  An extremely detailed guide to Wireshark as stated in Chapter 7. There is also a good section on Wireless Networks.
- Wireshark.org
  The home of Wireshark.
- riverbed.com
  Home of AirpCap—Wi-Fi analyzer for Wireshark.
- metageek.com
  Home of Chanalyzer and WiSpy.

10 Security

10.1 Introduction

Many people use the words "safety" and "security" more or less as synonyms. However, while *safety* is about avoiding accidents and people getting hurt, *security* is about securing data and equipment against malicious use and theft. In this chapter, we will focus on *security*. Since this is a book about IoT, it is probably not a surprise to the reader that more and more embedded devices are connected to the internet. Strangely, this sometimes is a surprise to the embedded programmer, or rather the culture surrounding embedded programming.

Not so long ago, an embedded programmer could get away with "it's never going to be connected anyway." The message was that it was okay not to think about security, because a hacker would have to go through a lot of trouble to hack just a single device. That was then. Even though connected embedded devices don't show up on Google, they are still searchable. Hacker tools with such capabilities have existed for many years, and the "shodan.io" website brings this into plain view. At this site, you can browse IoT devices by, for example, type. You can find connected webcams with no password, or with the known default password. The site is scary, although it is mainly claimed to be a statement of how vulnerable the IoT world really is.

Now, for some years we have known about *denial of service* attacks. These might be characterized as "terrorism on a small scale," since the main goal is to harass a lot of people. This may attract certain types of people, but not the professionals. With *ransomware*[1] the situation is quite different. Here, the bad guys can actually make money, and since cash is fast being replaced by electronic transactions, criminals are "forced" into this area. As IoT devices originate from the embedded world, which hasn't yet gone through all the lessons the IT departments have, we may very well be in for a bumpy ride.

Many UK hospitals were subjected to what seemed a coordinated attack of the "Wanna Cry" ransomware on May 12, 2017. This attack exploited Windows and, therefore, mainly the systems for registration, planning and medical journal updates. The following day, attacks were reported from 100+ countries—in all sectors of public and private enterprises as well as homes.

The attack exploited holes in Windows network file-sharing SMB[2] protocol, and was spread—to the first machine in an installation via infected mails, and then "laterally" via the attacked organization's internal LAN. The weakness exploited did not exist in Windows 10. Two months earlier it was fixed by Microsoft on Windows 7 and 8, so that anyone using *Windows Update* on a supported version of Windows was safe. Unfortunately many public systems still use Windows XP which is not supported, or

1 All your files are suddenly encrypted. You might get them decrypted if you pay.

2 The Linux SAMBA-version was not affected.

https://doi.org/10.1515/9781547401024-010

they were not using Windows Update on Windows 7 and 8. Some larger IoT devices also run Windows, but they are not the only target—the next attack could be on Linux. What are the odds that these embedded installations are using automated updates?

Engineers in the security industry have published examples on medical equipment such as medicine dispensers and pacemakers, controlled with completely open protocols over the internet. These cases are wide open to ransomware.

How many companies have silently paid money to get out of trouble and avoid getting bad publicity? There is no way to know. To further complicate matters, ransomware is sometimes used as a way to "cover the tracks" of the real crime, which may be data theft. The first thing most of us will do, when struck by ransomware, is to unplug the computer, wipe the disk and install the backup. If there ever was a trail leading to a data-thief, it is certainly (and unfortunately) gone now.

Since the birth of the original "innocent" internet, secure protocols have been developed, and larger companies have been tightening security in their IT systems, but the devices in the field are still lagging behind. The main rule concerning security is normally:

We need to put so much work into security that the cost of breaking it is greater than the benefit for the villain.

With some sensitive data transmissions, this can be interpreted to mean:

We need to put so much work into security that it takes such a long time to breach, that the data is not interesting anymore.

Unfortunately, the above rules of thumb may lead someone to believe that they do not need to think about security in, for example, a toy sold on the mass market. However, if it is connected, it may be enrolled in a *botnet*. This term comes from "robot network," meaning that normal internet-connected devices are used to run hostile hacker "robot" software—against their original purpose. One example of a large botnet using default passwords on Linux systems is named *Mirai* (we will get back to this later). This is not directly a threat to the vendors of the IoT devices used, but to others. In the future, we may see certifications demanding a kind of social responsibility—just as we see with the Radio Equipment Directive; see Section 9.11. Until then, a careless vendor does not lose money directly, but may become the center of a "shitstorm," and thus lose brand value.

The social-professional site LinkedIn is a rich source of information for hackers. Knowing a flaw in a particular piece of software, hackers search for people or companies advertising themselves as users. Once they have found a user, they can typically guess the mail address from the name and the company. Now they send an infected mail. The mail could be created with the program in question—e. g. a PDF file, in which case it is used as the gate into the system. Alternatively the mail uses another *gate, but attacks the program which the user has special access rights to. If this is e. g. a SCADA (supervisory control and data acquisition) system for production, the damages can be huge.*

There are other ways to use the seemingly innocent personal details we keep at LinkedIn. More directly, LinkedIn was hacked in 2012 and the password database was stolen. The passwords were kept as "unsalted" SHA-1 hashes (see Section 10.4*) and were soon cracked. Similarly Dropbox passwords were stolen in the same year. You can check your own status at https://haveibeenpwned.com where security expert Troy Hunt freely tests a given account name (typically mail address) against databases from the black internet. I had two instances of broken passwords here. This finally convinced me to use a program for password and credit-card storage, stopping the constant reuse of a few passwords.*

10.2 The goals of a hacker

It is always good to know what you are up against. Let's take a look at what bandits might do:

- *Cloning*
  When the competitor buys your product, scrutinizing every detail in order to copy it, it is called *reverse engineering*. This used to be a matter of regenerating the wire diagram and the list of components used.[3] The appearance of embedded software, instead of pure hardware based on standard components, changed the game, but has not made this impossible.
- *Stealing Service*
  This is the well-known hacking of cable-TV, NetFlix, Spotify, etc. This is often driven by a few capable hackers that leave manuals for the masses. As fewer pay for the service, it becomes more expensive for the paying customer, and a deadly spiral begins.
- *Identity Theft*
  Typically using phishing mails, the hackers get access to a user account with access to trusted data. The data could be account numbers and credit-card information. The data is sold on the black internet. This creates a lot of trouble for the individuals whose information is used, and is also a potential brand killer once the word is spread that XX company was responsible for leaked personal information. Many public systems, like hospitals, are the target of data theft, the data being social security numbers and other personal information. This can be used, not only for credit-card fraud, but for a wider range of identity theft. It is not completely uncommon that when a hospital sends a bill after an expensive operation, the receiver can prove that he or she never had the operation.

3 This is often nicknamed "BOM" (Bill of Materials).

- *Taking Control of a System*
  Taking over a complete system will enable the bandit to do whatever the system is capable of. This is basically the "Die Hard 4.0 firesale," where we see manipulation of traffic, water- and gas-supply, as well as the entire bank system. On a more everyday level, you may be responsible for embedded software in a beer-brewing system used in an international brewery. The wrong temperature could destroy millions of dollars worth of beer. The bad guys may take over the system, threatening to change the temperature on all currently brewing beer—unless you pay. This is ransomware in the world of IoT.

We will look at physical devices as well as internet security. Interestingly, the math and concepts developed for network security, in many ways also can be used on the internals of a device. A device can be seen as a micro-network with, for instance, wires between the CPU and the storage. For this reason, we will start with network security.

10.3 Network security concepts

CCITT (the International Telegraph and Telephone Consultative Committee) has developed a security architecture in the standard X.800. Here, they introduce the following basic security services:

- *Confidentiality*
  This is classic "scrambling," where data is made unreadable before transmitted, and then recreated at the receiving end. This may relate to a single packet in a transmission, the full stream, or any block of data.
- *Integrity*
  Sometimes you don't need, or maybe don't even want data to be secret—you just need assurance that it's untampered with. A contract between two parties, or a software license, may be kept in clear text, but in a way ensuring that not a single letter may have been changed, added, or deleted.
- *Authentication*
  In the digital world, it is important to be able to prove that you are who you say you are—just as it is in the physical world. We want to assure the identity of the sender in a connectionless scheme such as email. This is *data origin authentication*. We may also need *peer authentication* where two parties in a conversation, or transaction, can be guaranteed that they are dealing with the expected peer.
- *Nonrepudiation*
  This is a kind of backward authentication. Just like we want to know the identity of a sender, we also want to assure that he cannot "go back on his words." Similarly, it is possible to assure that the receiver of a message can not deny having received it. The two cases are respectively known as *nonrepudiation with proof of origin* and *non-repudiation with proof of delivery*.

– *Availability*
  This is better known by the kind of attack it defends against—denial of service (DoS)—and is different from the previous four, in that remedies go far beyond cryptography and mathematical algorithms. Maybe this is the reason why it is deeply buried in X.800, but it is there. Since DoS attacks have been one of the most frequent types of attacks, availability is now pretty high on the list.
  In a DoS attack, the perpetrator does not steal secrets to use them, but renders systems unusable for a time. This does not sound so scary, but if we are dealing with large infrastructures, or hospitals, it becomes serious. As stated earlier, ransomware is one of the greatest threats to IT as well as IoT. This is exactly a case in making systems unavailable until someone pays.[4]
  Remedies are a mixture of algorithms in this chapter, and standard network protocols as described in Chapter 7 (e. g., TCP has a timeout to avoid the OS being pulled down by a load of half-open sockets, created by hackers in a DoS attack).

Of the above, nonrepudiation is the one people tend to forget, maybe because of the strange name, but probably also because it is not as frequently needed as the others. The three—*confidentiality*, *integrity,* and *availability*—are often referenced by the acronym “CIA” or even called the “CIA triad.”

The above services (partly with the exception of *availability*) are realized with the help of the technological cornerstones of modern digital security, which we will look into in the next sections. Some of the main terms are collected in Table 10.1.

Table 10.1: Security terms.

Term	**Explanation**
Plaintext	A message or document before encryption
Ciphertext	The encrypted message or document
Code	Informal word for combination of key and algorithm
Key	Used to make a general algorithm specific
Alice & Bob	Often used as the two legal parties in a conversation
Trudy	The illegal intruder on Alice and Bob
Shared Key	Used in classic encryption/decryption
K_A^+	Alices (A) public (+) Key
K_B^-	Bob’s (B) private (-) Key
Encryption algorithm	Outputs ciphertext from plaintext and Key
Decryption algorithm	Outputs plaintext from ciphertext and Key
Brute force	Using a fast computer to test all combinations
Lexicographic analysis	Using knowledge of the language to crack a code
Hash	Fixed-bit-length number representing data
SHA	Secure hash—generates, for example, a message digest
MAC	Message authentication code
Nonce	Number used once. Secures against replay

4 Do not count on getting your data back, even if you pay.

10.4 Hash function

A hash algorithm is a function that processes a binary block (or stream) of any length, and as a result produces a fixed-length binary number. A good hash statistically produces all possible outcomes equally often. It considers the position of characters so that "I owe You $100" generates a different hash than "I owe You $001" (this rules out a simple byte checksum). Another feature of a good hash is that a small change in the input, changes several bits in the output.

An example is the *cyclic redundancy check* (CRC) used in many protocols. Ethernet uses a CRC-32, meaning that it is 32 bits long, no matter how many bits of payload you have put inside the frame.

Hash tables are a well-known help when creating large key-value pair tables, where the hash of a long textual key is used as a way to organize the keys for fast retrieval of the value. For instance, TCP typically use hashes, to quickly find the process that matches an incoming packet with a 5-tuple consisting of a protocol, source port, destination port, source IP, and destination IP.

Similarly, computers never store the users passwords, but hashes of these. An administrator cannot "do things in your name" and if you use the same password in several systems and one is leaked, it doesn't necessarily mean that your password in general is revealed. SHA-1 is used as the ID for files in Git, see Section 6.2. Thus hashes is an interesting field with applications reaching far beyond cryptography.

With a good cryptographic hash, it is infeasible to create another input that produces the same output. Thus it is also infeasible to change a given message to something with the same hash. Examples of cryptographic hashes are MD5, SHA-1 and the newer SHA-256. The term *SHA* means "secure hash algorithm." Naturally, as time passes and computers get faster, and thus better at brute-force attacks, yesterday's secure hash may not be tomorrow's. MD5 is hopeless today, and SHA-1 is not considered safe anymore.

In the cases where password databases from, for example, LinkedIn or Dropbox were stolen, many passwords were regenerated. This was made easier by the fact that these hashes were not *salted*. A salt is additional information added to that supplied by the user. When this *is* used, it means that even if the user is using the same password on different sites, and these sites use the same hash algorithm, they do *not* have the same hash. Wi-Fi networks may, for example, use their SSID as a salt with the user-supplied password.

A *message digest* is simply a (secure) hash of a message. NIST[5] defines a number of secure hashes in FIPS[6]-180-4 from 2012; see Table 10.2.

5 US National Institute of Standards and Technologies.

6 Federal Information Processing Standards Publication.

Table 10.2: Secure hashes described in NIST FIPS-180-4.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

The hashes in Table 10.2 are standardized in ISO/IEC 10118-3 (along with others). FIPS-180-4 also describes the algorithms and constants needed to produce these hashes, and it is possible to create C-code directly from this description. This may be an interesting exercise, but it is recommended to use a standard well-tested implementation; see Section 10.16.

10.5 Symmetric key encryption

This is the classic encryption/decryption type, also known as *shared key* encryption. The general principle is known to most people and is shown in Figure 10.1, where Bob is sending a secret letter to Alice. The *plaintext* is sent through an encryption algorithm using a secret key.

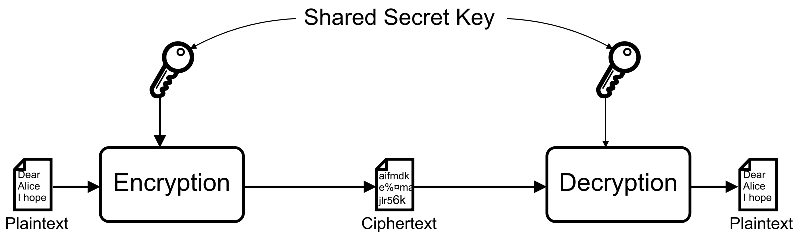


Figure 10.1: Confidentiality by symmetric key encryption.

The result of the encryption is the *ciphertext*. At the receiving end, the decryption algorithm performs the reverse process—using the same key. Thus the key is prior knowledge between the two participating parties.

Modern-day examples are DES (Data Encryption Standard) and triple-DES, now both obsolete, as well as the newer AES (Advanced Encryption Standard) standardized in ISO/IEC 18033-3. One of the earliest known algorithms is attributed to Caesar, but there are even earlier codes. The most basic example is, for example, that each letter in the plaintext is exchanged with the one before it (wrapping at the ends).

Plaintext:	`abcdefghijklmnopqrstuvwxyz`
Ciphertext:	`zabcdefghijklmnopqrstuvwxy`

Thus "IBM" becomes "HAL." This concept is an easy target for a *brute force* attack—simply try adding one, two, three, etc. to the index of the letters in the alphabet. It gets much more interesting when you substitute letters in a more random fashion. However, as every letter gets substituted with the same letter every time in the given message, or maybe for the whole given day, this concept is a sitting duck for *lexicographic analysis*. Knowing the original language and the frequency of the various letters in it, it is still fairly easy to break this code.

10.6 Case: enigma

More advanced algorithms include schemes for rotating the substitution. The German Enigma from the second world war is a great example of this; see Figure 10.2. It is mythical today, probably due to the advanced coding scheme realized before computers existed.

Enigma is used pretty much like a typewriter: you hit a key with a letter, and instead of printing it, Enigma lights up another letter. Pressing the key causes current

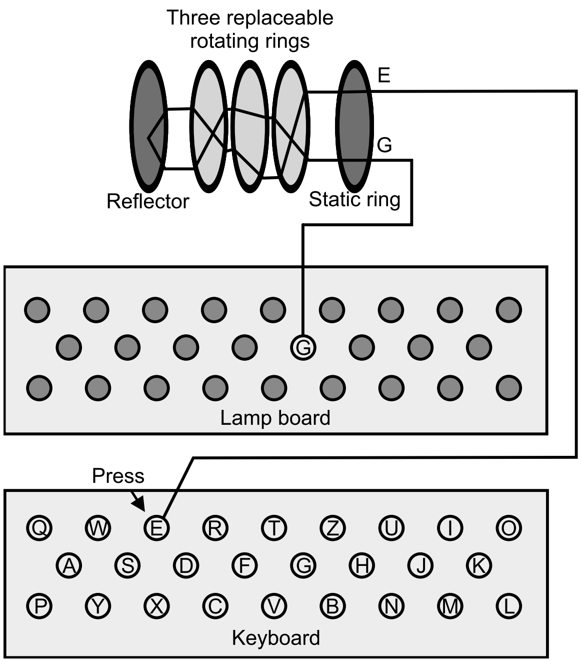


Figure 10.2: Enigma. Three rings rotate like a clock after each letter.

to run through wires, to a lamp. The path of the current goes through 3 wheels. Each wheel has the numbers 1–26 on its circumference, just as there are 26 letters on the keyboard and lamp-board. Spring-pins on one side of a wheel connect to plate contacts on the wheel next to it, on the other side. The internal connections between the two sides of a wheel are "shuffled" in different ways from one wheel to another. Thus every wheel has a fixed encoding of one letter to another. After encrypting a letter, the right wheel turns one position. When it comes to a number, specific for the given wheel, the middle wheel is turned one position; likewise with the next wheel. Any wheel of the 5–10 possible wheels can be mounted in any of the three positions on the axis.

To avoid having one machine for encoding and another for decoding, the static "reflector wheel" sends the current back through the three rotating wheels again. This creates a symmetry.

There were several Enigma variants, specific to submarines, navy and various geographic regions and time spans. The configuration of the machine changed every day according to a code book. This included the selection of the three wheels to use, their position (left, middle, right) and the numbered offset when mounting them on the spindle.

Some Enigmas were enhanced with an additional switchboard at the front. Here, the operator connected the wires for today's setting. This also had to be symmetrical, thus if A becomes G, then G must become A.

An important help for the code-cracker is to have parts of a message in both plaintext and cipher. It is said that due to the strict discipline in the German army, dispatches under WW2 tended to follow a pattern—say starting with the date and the rank and name of the officer sending the dispatch. In the movie, "The Imitation Game," the words "Heil Hitler" supposedly has this function.

Late in the war, Enigma was enhanced with a fourth wheel. Another story says that one of the first dispatches was sent by mistake on the old machine. Eager to correct the mistake, the communication officer rushed to send it with the new machine, thus helping the enemy tremendously. These stories may just be legends, but they paint a good picture of the traps you may fall into in this area.

Until 1976, this "family" of algorithms was the only one known, and thus all the exciting books and movies about WW2 relate to symmetric key encryption. Basically, the two parties, for example, Alice and Bob, share a secret key only they know. Once the encryption algorithm is known, the decryption algorithm can be deduced. This is not a problem: you should always use well-described and well-proven algorithms, as the home grown variants are extremely likely to be flawed.

The security must lie in the key. The basic problem with a shared key is the actual sharing. If you are to buy a pair of shoes over the internet, it is very impractical that you need to meet with the shoe seller and exchange a key first. A shared key is needed for every pair communicating with each other. With all the combinations of people doing business with each other, we need a huge number of keys.

When I taught TCP/IP at the Technical University in Copenhagen, I created a "multiple-choice" exam each year. I wanted a colleague to review this, but did not want to send it via the university's e-mail server. My good colleague suggested I should use Ax-Crypt to encrypt this with AES encryption, and then text the key to him. This we did for years, and it was a great alternative way to handle the key-sharing.

10.7 Asymmetric key encryption

In 1970, the concept of asymmetric keys was mathematically established in the UK, and immediately classified for military use. A similar scheme was published in 1976 by Diffie and Hellmann. In 1978, the RSA algorithm was published by Rivest, Shamir, and Adleman. This is now standardized as RFC 3447.

Based on factoring very large integers, it now became possible to create ciphertext from plaintext with one key, and to go the other way with a completely different key. Knowledge of one key cannot help you to know the other.

In the case of RSA, the encryption and decryption algorithm is the same. In fact, the RSA algorithm does not "know" whether a ciphertext comes in and a plaintext comes out, or vice versa. In principle, you generate a key-pair and you select one of these as your private and, therefore, secret key. The other key is the public key that you "publish"—not as easy as it sounds. K_A^+ and K_A^- is the notation for party A's public and private keys, respectively. There are other asymmetric key encryption concepts which *do* require a specific encryption algorithm and another algorithm for decryption. The major point is that all asymmetric key encryption algorithms use two different keys, a public and a private, hence the name "asymmetric."

Anyone who wants to send a letter to you, that only you can read, can use a commonly known algorithm with your public key to generate a ciphertext. This ciphertext becomes plaintext when decrypted, this time using your private key. This concept works because you are the only person with this key. Thus we have confidentiality. Surely, it is possible that a private key is known by the wrong people due to theft, extortion, or torture, but this is not different from shared or even physical keys, and is therefore ignored in these theories. Nevertheless, just like credit cards are renewed at intervals to minimize these problems, so are many digital keys. Figure 10.3 shows how Alice can encrypt her letter to Bob this way.

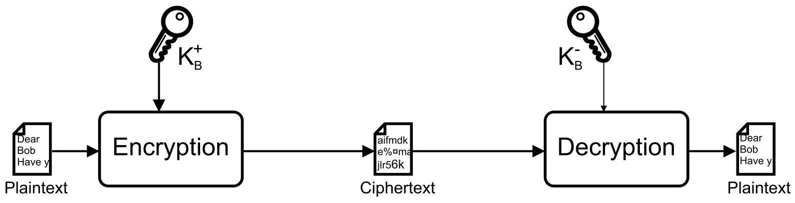


Figure 10.3: Confidentiality by public/private key-pair.

There is an interesting symmetry in these asymmetric keys: You can take any message and encrypt it with your own private key. Now anyone can use your public key to recreate the plaintext. There is not much secret keeping here, but the fact that your public key can “open” the message proves that it was created with your private key. Since this is only known by you, we thus have a way of authentication as well as nonrepudiation.

This gets even better, as we can chain the processes, like in the following scenario:

1. Alice creates a secret love letter to Bob.
2. Alice processes the message with her private key. This authenticates the message as being from Alice, and is one way of *signing* it (we will get back to signing).
3. Alice processes the output from the above with Bob’s public key. This creates confidentiality, now only Bob can read it.
4. Alice sends the message to Bob, or tweets it, or publishes it in another way if she doesn’t want to leave a trail pointing to Bob.
5. Bob grabs the message and processes it with his private key, then with Alice’s public key, or vice versa.
6. Finally, Bob, as the only person in the world, can read the message, knowing it’s from Alice.

Asymmetric key encryption, also known as public/private keys, has two major problems:

– *Performance*
  The amount of CPU power needed to encrypt/decrypt with this concept is roughly 1,000 times that of a symmetric key at the same level of security. This means that we are only using this concept where it really shines.
– *Publishing the Public Key*
  Suppose Alice starts the conversation with Bob by sending her public key first. Now Trudy could intercept this and make a new letter with her key-pair and send her own public key to Bob. Now Bob would believe the letter was authenticated as being from Alice. The lesson learned is that we need to take the publishing of public keys very seriously. The current solution is a *certificate authority*—or simply CA—which we will look into in Section 10.9.

10.8 Digital signature

Signing a long message or document by encrypting it with a private key, as described in Section 10.7, is a very costly process in terms of CPU time. Instead we prefer to first generate a safe hash of the document, and then encrypt this relatively short hash with our private key. This way it is really only the hash that is signed. Since it is unfeasible to create this hash from another message, the receiver of the message can simply recreate the hash from the plaintext message and compare it with the signature after decrypting this with our public key. This is shown in Figure 10.4 with Bob as the sender.

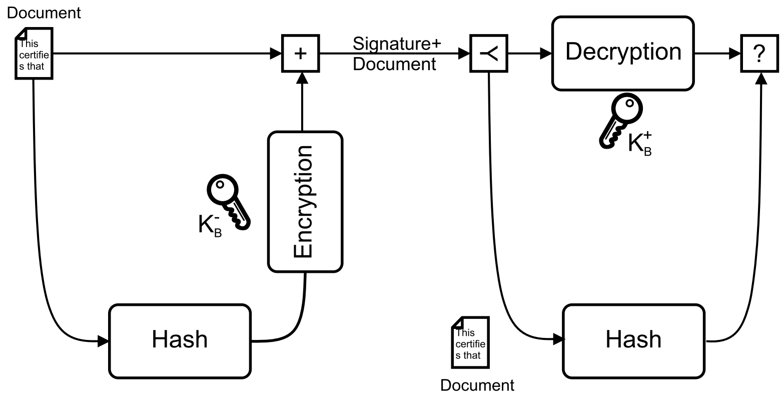


Figure 10.4: Integrity, authentication and nonrepudiation.

Using this scheme, we therefore obtain:

- *Authentication*
  Since the signature was confirmed after using Bob's public key, it must have been Bobs private key that generated it. Only Bob knows this.
- *Nonrepudiation with Proof of Origin*
  The same arguments as above mean that Bob cannot deny having generated the message.
- *Integrity*
  Since the hash of the plaintext was the same as the one sent with it, the message was untampered with. It might have been read along the way as it was sent in plaintext, but if anyone had changed it, they would have needed Bob's private key afterward. Only Bob has this.

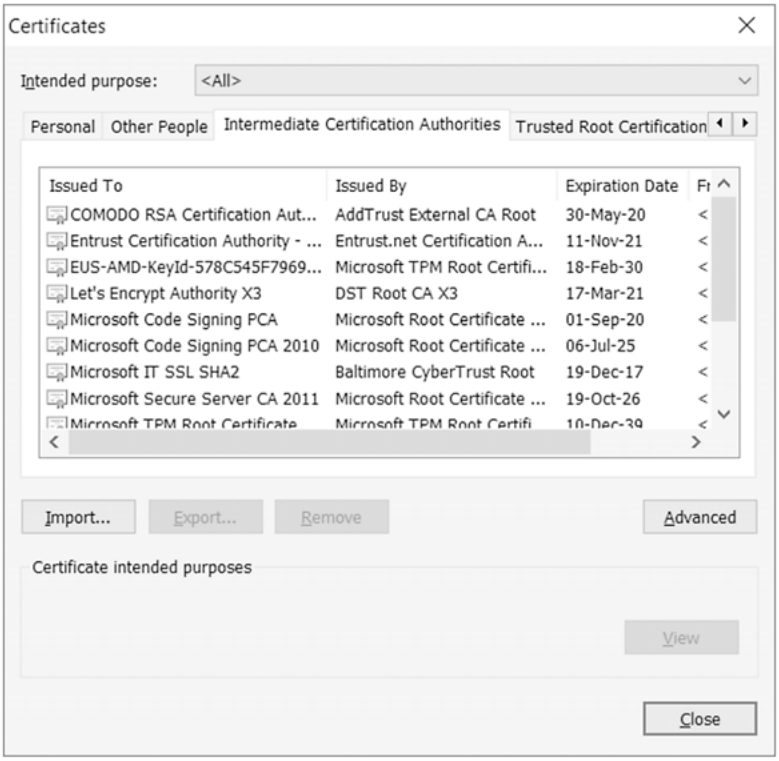
10.9 Certificates

Just like we are more willing to trust a stranger, say Jones, if our best friend vouches for him, more likely are we to accept a public key from Jones, if his public key is *signed* by a CA (certificate authority). This signature could simply be that the CA has encrypted Jones public key with the CA's private key. You already have the CA's public key. By using this, you authenticate that the key sent by Jones is indeed his. This is known as the *chain of trust* and is defined in X.509.

As stated in Section 10.5, the main problem with *symmetric* keys is sharing the common key, while actually using them is a lot faster than using asymmetric keys. For this reason, it makes a lot of sense to start a communication session by exchanging symmetric keys, generated on the fly for this session only, and encrypted with the help of public/private keys. Then use the symmetric keys for the rest of the conversation. Thus Alice can generate a random symmetric key to be used only for this session, encrypt it with Bobs public key, and send it to him. This concept is used in an advanced version in *SSL* (secure socket layer) and in its twin *TLS* (transport layer security) as we shall see later. Generally, asymmetric keys are the accepted way to transfer symmetric keys, and the rest is done using the symmetric keys. These asymmetric keys are sometimes generated with the use of Diffie–Hellman, but more often RSA is used. There is also a third type of public/private keys called elliptic curve cryptography. The math is different from RSA and Diffie–Hellman, but the processes surrounding it are the same.

When publishing keys to each other, or fetching them at websites, we rely on the certificates issued by the CA. So how do we get these? They can be downloaded from the CA's website, but very few end-users do this. Typically they are installed and updated together with the web browser. This is the weakest link in the chain—one more reason for updating your software from a trusted site. The largest CA's are, in descending order by market share: Comodo CA, Symantec/Verisign, GoDaddy, and GlobalSign.

Figure 10.5 shows the certificates stored in a Chrome browser on a Windows PC. Note the three columns; *Issued To*—who is certified, *Issued By*—who issued the certificate and *Expiration Date*—when does it expire. Using the "View" button, each certificate can be scrutinized. Some may be "self-signed" which means that they are not CA-certified, and if the user attempts to use them a warning is issued. The user may then acquire another verification of the certificate, abort, or ignore.


Certificates

Intended purpose: <All>

Personal | Other People | Intermediate Certification Authorities | Trusted Root Certification

Issued To	Issued By	Expiration Date	Fr
COMODO RSA Certification Aut...	AddTrust External CA Root	30-May-20	<
Entrust Certification Authority - ...	Entrust.net Certification A...	11-Nov-21	<
EUS-AMD-KeyId-578C545F7969...	Microsoft TPM Root Certifi...	18-Feb-30	<
Let's Encrypt Authority X3	DST Root CA X3	17-Mar-21	<
Microsoft Code Signing PCA	Microsoft Root Certificate ...	01-Sep-20	<
Microsoft Code Signing PCA 2010	Microsoft Root Certificate ...	06-Jul-25	<
Microsoft IT SSL SHA2	Baltimore CyberTrust Root	19-Dec-17	<
Microsoft Secure Server CA 2011	Microsoft Root Certificate ...	19-Oct-26	<
Microsoft TPM Root Certificate	Microsoft TPM Root Certifi...	10-Dec-39	<

Figure 10.5: Certificates in Chrome browser on Windows.

10.10 Message authentication code

The term *MAC* in the domain of security means “message authentication code,” and is completely unrelated to the DSP MAC (multiply accumulate) and the MAC addresses we know from the link layer in the network protocol stack. It comes in different flavors, of which the simplest requires no encryption at all. This can be used when we need authentication and identity, but not confidentiality. It is especially practical in small embedded devices. The concept is shown in Figure 10.6.

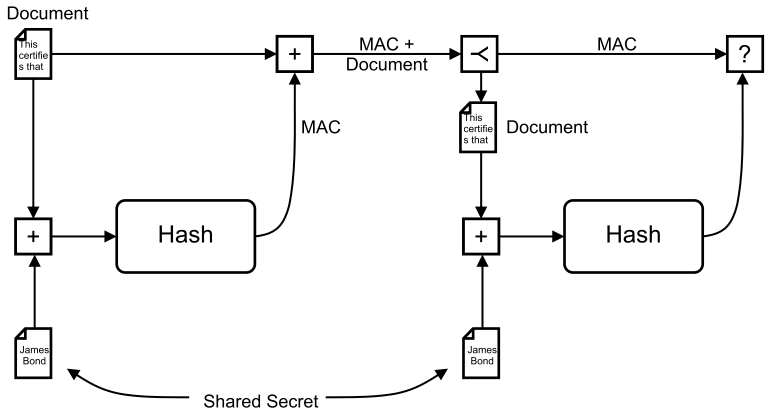


Figure 10.6: Message authentication code without encryption.

Just like with symmetric key encryption, the two parties have a shared secret. This is sometimes referred to as a “key”—sometimes simply a “shared secret.” This could, for example, be the text string “James Bond” as shown in Figure 10.6. Now we concatenate our plaintext document, which could be a software license, with the shared secret, and generate a safe hash of it all, for example, using SHA-256. We transmit the original plaintext and the hash of the combined text strings, but not the shared secret itself. The process is repeated at the receiving end, and if the two hashes are the same, we conclude that the message sent in plaintext is untampered with. Anyone might have seen the transmitted plaintext, and they might even have changed it, but then we would see a different hash after adding our shared secret. We have thus authenticated it without the use of advanced encryption algorithms. We only need the capability of generating a safe hash.

As discussed before, there are two problems with shared secrets, the actual sharing and the number of parties that need to share different secrets. If the text is indeed a software license, there really are only two parties; the back-office generating the license, and any of the same company’s embedded devices checking it. Thus we do not need loads of shared secrets, only one, or maybe one per product-type or family. The physical sharing is not a problem either, as we, the company, install the firmware with the shared secret in our own product before distributing it.

If the software license contains the serial number of the device, it is not a problem that it is in plaintext. On the contrary: imagine a customer receiving licenses for several embedded devices. This customer can look at the received text string, which typically is sent by mail, and immediately see which device it is meant for. Had we used encrypted keys, we would be tempted to pair each encrypted license with the serial number it was meant for. This would be a source of mix-ups, but even worse; we would be supplying the plaintext of a part of the ciphertext. Thus we would weaken our code in the same way as the soldier who starts the daily report with the date and his rank and name.

MACs without encryption are becoming increasingly popular. Without encryption, there are no export regulations, and existing high-performing libraries or hardware-engines can be used. It is also practical that a general standardized algorithm could be used with the secure hash as a callable function which may be replaced as requirements grow. This algorithm is the *HMAC* defined in ISO/IEC 9797-2. Figure 10.6 still serves as the overall principle for HMAC. There are a number of implementation details about padding bits in input and output and an XOR'ing function, but the basic concept remains the same.

10.11 Nonce

Even advanced cryptographic algorithms can be vulnerable to *playback attacks*.

An example is a shady shop that sells some very appealing stuff, at very low prices. You may buy something at the shop, ordering your bank to transfer a small amount of money to the shop's account. Now the shop has recorded this transaction and repeats it a number of times. This is done by contacting the bank and replaying your part of the conversation, while the bank goes though its predictable motions again. Both sides may send certificates, and advanced coding schemes are used, but the playback circumvents it all. The shop does this for a day with many customers, then withdraws the money and disappears.

The simple remedy is that the bank initially sends a nonce and asks that this is returned, encrypted with the relevant key and coding scheme. Now the recording is worthless, as it was based on the previous nonce. The practical solution is often to make the nonce part of the key used for the specific session.

10.12 Secure socket communication

With the previous sections, we now have all the tools needed to understand secure socket communication at an overall level. When using the TCP/IP protocol stack, everything passes through the IP layer. If security is implemented here, we may be able to secure all communication. This is exactly what *IPSec* does. However, IPSec is big and may be overkill for most embedded applications. When PCs were young they had the same problem: security at the IP level would require all PCs to have a new IP stack, which meant a new operating system that wasn't even there. Netscape/Mosaic realized that web traffic was the essential communication needed to allow internet-shopping for the masses. They created *SSL* (secure sockets layer). This only worked for socket-based TCP within the browser, but that was enough, and thus Netscape cut the Gordic knot, requiring only a new browser on the consumers PCs.

A later version of the protocol used in SSL was standardized as *TLS* (transport layer security). The protocol remained almost the same, but the service was now offered by the protocol stack to all applications using TCP. These newer versions support many different implementations of symmetric and asymmetric keys as well as MACs and nonces.

Depending on your embedded system, you may have one of the following scenarios:

– *No Built-in Security*
  You cannot perform any standardized communication using confidentiality, integrity, or authentication.
– *SSL in Web Server and/or Web Browser*
  If you only have secure sockets via your web solution, the recommended way to proceed is to use a REST-based protocol; see Section 7.12. Your code will be modules in the web server or browser and this will handle the security for you.
– *TLS*
  With TLS your possibilities extend to every protocol using TCP. This is typically all you need, unless you need to do broadcasts or multicasts; see Section 9.3. There is actually a separate TLS version for UDP, making even broadcasts and multicasts possible.
– *IPSec*
  With IPSec your embedded device can do *VPN* (virtual private network), and thus be a part of a company domain. IPSec does, however, not give you secure sockets. For this, you will still need TLS or SSL.

TLS as well as SSL show up as the well-known pad-lock symbol in your browser and as *https* instead of "http" before the URL, when using HTTP (the default port for SSL/TLS is 443). When analyzing which implementation to choose, it is necessary to look at what the other end will be able to support. You may have an advanced embedded device running Linux with bells and whistles, but it may need to communicate with smaller microcontroller devices. As in other matters, no chain is stronger than the weakest link.

Before sending encrypted and authenticated data in SSL/TLS, the parties need to go through a complex handshake, partly to negotiate which algorithms to use, partly to establish a "master key" for the session to be used as a symmetric key. This handshake has the following four phases, described here generically:

1. *Hello*
   The client contacts the server, informing it about the highest protocol version the client supports, which ciphers it supports and which compression algorithms. It also states a session ID (0—unless attempting to resume a communication) and a nonce—partly based on a time stamp. The list of supported ciphers includes which asymmetric key concepts are supported for sharing symmetric keys, as well as which symmetric key ciphers are supported. These are ordered after preference. Similarly, the client also informs the server about algorithms supported for hash and MAC.
   Based on the clients wishes and its own capabilities, the server sends the choice of protocol version, compression algorithm, ciphers, and MAC as well as its own nonce. If accepting a resumed communication, the session ID is sent.
   Knowledge Gained: Both parties now know both nonces, sent in plaintext as well as the basic configuration.
2. *Server Certificate*
   The server sends its relevant certificate containing its public key. The client verifies that this is indeed signed by a trusted CA, and is not too old. The server may request a corresponding certificate from the client.
   Knowledge Gained: Client now also knows the servers public key.
3. *Client Key Exchange*
   If the client was asked to send a certificate, the client now sends a *certificate verify* message—a signature of the previous communication, encrypted with the clients private key, showing the server that the client indeed has the key.
   The client now generates a random *PMK* (premaster-key) and sends it, encrypted with the servers public key. Both parties now generate the same *master-key* from a combination of this and the two nonces.
   Knowledge Gained: Both parties now share the master-key to be used in symmetric encryption for the rest of the session. If requested, the server now knows the clients public key.
4. *Change Cipher Spec*
   The main thing here is the "finished" message from the client, containing a MAC of the previous communication. If the server agrees, it sends a similar message which the client verifies. Normal communication now continues in the same way.
   Knowledge Gained: Both parties are now sure the original plaintext setup was untampered with by, for example, a "man-in-the-middle" attack.

10.13 OpenSSL

We have seen the theory behind SSL, but how is it implemented? This is where OpenSSL comes in. OpenSSL is a site, a library, and a tool. The following describes how the OpenSSL library from openssl.org is used to implement SSL in your code. It is perfectly possible to start with working standard socket-based code. First, the SSL library is initialized, and an SSL *context* is created with the relevant configuration and options. Within this context, an SSL *session* is created.

If this is a client, it now connects a normal TCP socket to the remote IP address, port 443 (instead of port 80 for a web server). A server does a normal `listen()` and `accept()` call on port 443.

This normal server or client socket (on port 443) is now attached to the previously created SSL session, and the session is used much like a normal socket, except that function calls start with "SSL_." We can, for example, read data with `SSL_read(session,...)`.

`SSL_get_peer_certificate(session)` is an important addition, allowing us to check the certificate from the web server (or optionally the web browser). There are many good samples on this procedure.

Openssl.org also supplies a command-line utility called *openssl*. This is a comprehensive tool that can generate keys, signatures, and certificates and a lot more. It can even generate a *CSR* (certificate signing request). A CSR is sent to a CA. It contains a public key that we have generated. From the CA, we get it back in a certificate, signed by the CA. We can now imagine a complete scenario on a production site:

1. On our Linux-based device, we use the openssl tool to generate a public/private keypair. It might not be possible to run openssl on the device, in which case it is run on a production PC.
2. We immediately store the private key in the write-only memory of the built-in TPM (trusted platform module)—if such exists. If the key was generated on the device it has never left it, and can never be extracted from it.
3. The public key is passed on to a production PC. The PC uses the openssl tool to digitally sign the key. It uses the company's very secret "golden" private key in this operation (this was originally also generated with openssl).
4. Again the openssl tool is used. This time to send the company-signed device public key in a CSR to the CA. The CA already have the public key that corresponds to our company golden private key, and uses this to verify that the CSR comes from us.
5. The answer from the CA contains the same certificate—now signed by the CA. This is stored in the device. The device is now ready to present a unique CA-signed certificate in an SSL-handshake in the field—typically using the OpenSSL library.

The above satisfies the X.509 "chain of trust." It can however be problematic to have the company's private key used in production on a daily basis. An alternative to steps 3 and 4 is a simpler approach where the production PC performs a login at an API at the CA instead of using sslopen. It delivers the newly generated key in a CSR structure as before, however, this is *not* signed by the company. The security of the login replaces the need for our company signing. In other words, the CA trust that we are who we say we are because we login, not because we present something that is signed with our private key. If security is breached we just need a new login instead of a new company certificate. The CA-signed certificate is delivered back on the same API.

It is possible to be less ambitious and instead of device-specific certificates use "wildcard" certificates. In both cases, a URL is certified not an IP address. If the device acts as a server, clients must use the URL when connecting, even though the IP is known This again means that the device might need to be registered in the DNS or use mDNS (multicast DNS) as known in Apple's Bonjour and the Linux Avahi implementation.

The unique URL could, for example, be "12345678.mydomain.com," where the digits are the serial number of the device. The corresponding wildcard URL is "*.mydomain.com." The main problem with wildcard URLs is that if one private key is compromised, all devices are threatened.

10.14 Case: heartbleed

With RFC 6520 in 2012, a simple addition to SSL was standardized, a *heartbeat*. This is a classic concept used in many protocols, allowing one party to ask the other party "are you still there?" at the same time telling this other party that it itself is alive, even though there has been no payload data for some time. This may avoid a timeout on either side followed by an elaborate resume scenario.

The *heartbeat message* contains the message type (stating that it is in fact a heartbeat message), a length field, and a dummy payload with as many bytes as stated in the length field (max 65535) and in some cases padding bytes. The receiver *must* reply with the exact same payload. This is basically a "ping" at the SSL protocol level.

One of the most used SSL implementations is OpenSSL described in the previous section. In 2012, this was used on almost all web servers offering SSL. When the heartbeat was implemented in OpenSSL, a bug was unfortunately introduced. This was found and described by Google and Finnish Codenomicon at almost exactly the same time (April 7, 2014) The problem is described here, because it is a great example of how easy it is to make these errors:

When the faulty OpenSSL server receives a heartbeat request, the payload is copied from the incoming packet to internal memory. This copy function uses the *actual* number of bytes in the payload. Now the reply is generated, this time copying out as many bytes as stated in the *length* field. The problem arises if a bad guy sends a packet, stating that there are, for example, 65535 bytes in the payload, while in reality there is only one. The single byte is copied to internal memory at the request, but at the reply this byte, and the next 65534 bytes in the storage, are copied to the reply message. Thus the server is dishing out 65534 bytes from its internal storage. This can be repeated several times, often with different "views" into the servers internal memory.

Codenomicon described how their test showed that they could dig out a lot of confidential data, including their own private keys. This did not even leave traces in log files, etc. No wonder it was soon baptized the "Heartbleed Bug," and even got its own logo; see Figure 10.7.

Figure 10.7: The logo for the Heartbleed bug.

It is a known fact that the bug has been exploited. On August 20, 2014, Reuters reported that a US Hospital had experienced a theft of data, including millions of social security numbers. This was due to Heartbleed. Nobody knows how many times the bug has been exploited, neither before, nor after the description of the problem was released. Naturally, publishing the bug has caused some incidents, but on the other hand it might also have urged administrators to get the bugfix, or at least recompile their system to disable heartbeat. This is the constant dilemma: how to tell the good guys they need an update, without telling the bad guys how to exploit a security breach. The "Wanna Cry" ransomware, discussed in the introduction to this chapter, became active only 2 months after the bug it exploits was fixed by Microsoft.

10.15 Case: Wi-Fi security

The early Wi-Fi systems used *WEP* which quickly turned out to be a very weak protection. It could be cracked just by observing wireless traffic for about 20 minutes. This is ironic since WEP means "wired equivalent privacy," which it certainly is *not*.

WPA (Wi-Fi protected access) was introduced as an intermediate fix. Today you should only accept WPA2 or better. This was standardized in 802.11i and became a part of 802.11-2007. Just as we have seen in other examples, the actual data transmission between stations and access point[7] uses symmetrical key encryption based on *AES-128*. In this case, a block-based scheme known as *CCMP* is used. This is a rather CPU-hungry process, and there may be Wi-Fi devices that need to drop to a lower data rate, than what would otherwise be possible with 802.11n (or newer).

WPA2 comes in two flavors, of which one is known as *WPA2 enterprise* or simply WPA2, while the other is known as *WPA2 private* or *WPA2-PSK*. PSK means "preshared

7 This section uses the terms from Table 9.2.

key.” You may experience that your laptop simply connects at work, while at home you need to type in a “pass-phrase” in your SOHO router or laptop before first-time use of either.

The difference lies in the authentication. Whether one or the other authentication method is used, is decided from the beacon/probe handshakes already discussed in Chapter 9. *At work* (the enterprise), the laptop is allowed to communicate to a RADIUS AS (authentication server) with help from the access point. The AP bridges the so-called EAPOL[8] protocol between STA and AP to the RADIUS protocol.

The AS authenticates the laptop as well as the access point. It generates a *master session key* for the given session which is sent to the participating access point and station. From this, they both derive the *PMK* (pairwise master key), still for this session only. This is possible because the station (the laptop) is able to present the credentials needed to login to the workplace domain.

At home, the laptop and the access point are each able to generate the PMK. The PSK might have been entered by the user as 64 hex digits, in which case it is used directly as the 256-bit PMK. This is probably not something many people want to do for a number of devices. If instead a pass-phrase has been used, the PMK is generated with the help of a *PBKDF2* algorithm which is a pseudo-random-generator. In the case of WPA2, it is called as:

$$PMK = PBKDF2(HMAC\_SHA1, passphrase, SSID, 4096, 256)$$

This means that SHA-1 is used iteratively 4096 times on the input, with the SSID as the so-called “salt” ensuring that two different networks with the same passphrase will have different keys, and that the resulting output is 256 bits long.

From this point on, the enterprise and the private scenarios are the same. Both parties now know the PMK (pairwise master key), which in the enterprise scenario is specific for the session-to-be, while in the private scenario it is constant for the given network, until the SSID or passphrase is changed. We now go through a “4-way-handshake”:

1. The AP (access point) sends a nonce to the STA (station) in plaintext.
2. The STA sends its nonce to the AP, also in plaintext, along with a *MIC – message integrity check* which is an MD5-HMAC of the plaintext as well as most of the header data. MICs are used in the next packets as well. Both sides now derive the *PTK* (pairwise transient key)—as the output from a pseudo-random-function like the above. This time based on: PMK, received nonce, own nonce, MAC address of the STA and MAC address of AP.
3. The AP now sends the encrypted *GTK* (group temporal key) which is used as the 128-bit AES-key for multicasts and broadcasts. Since there is a unique key for each STA-AP relation, it makes sense to have a separate key for multicasts/broadcasts. It also makes sense that this key is communicated to each STA, as broadcasting

8 Extensible authentication protocol over LAN.

it would require the key we are sending. The GTK-setup must be repeated whenever a device leaves the group so that it cannot eavesdrop on future conversations. Many routers have poor implementations of the multicast and broadcast scenarios. Section 9.3 shows how multicasts work on Wi-Fi.

4. An ACK is sent from the STA to the AP. This has no cryptographic meaning, it simply terminates this phase and synchronizes the use of the keys.

The above means that even in the private/PSK scenario where the pairwise master key is the same for all devices in the network, the pairwise transient key is unique for each STA-AP combination (due to the MAC addresses), and it is even different between each association due to the nonces. Hence the term “transient.” The PTK in WPA2 is 384 bits long. These bits are used as follows:

– First 128 bits: *KCK* (key confirmation key). Used for integrity inside the EAPOL protocol.
– Second 128 bits: *KEK* (key encryption key). Used for encryption within the EAPOL protocol.
– Last 128 bits: The AES-key used for the actual communication.

So how does WPA2 relate to an IoT device? Some devices may have a role similar to a laptop or a mobile phone. They will connect to a home network using WPA2-PSK. The developers of the device may expect to use the same in the enterprise for simplicity, but should be prepared that most IT departments will disallow this insisting on the enterprise scenario.

It is also possible that the IoT device is connected to the cloud via LTE, LoRa or similar, and that a mobile phone is used for local interaction. In this case, there is no authentication server, and the connection to the mobile device will be protected with WPA2-PSK. In this case, the IoT device may need some way to type in the preshared key. Alternatively, the developers may consider including it in the firmware, which carries some risks discussed in Section 10.19.

A third authentication alternative is *WPS (Wi-Fi protected setup)*. This is often used in SOHO-installations, because it is easy, simply requiring the user to press a button before a brief timeout, proving physical access. Another WPS-method is the use of a simple PIN-code, which is often factory defined and printed on a label on the device. These methods are basically considered unsafe.

10.16 Software Crypto libraries

It is generally a good idea to use a standard implementation of the Crypto algorithms. This avoids a lot of pitfalls, including export control violations as discussed in Section 10.20. CPU vendors such as Texas Instruments offer C algorithms suited for their processors. One of the most versatile and general free libraries is the C++ *Crypto++* library. This is available under the Boost license as a prebuilt library. It is also available as public domain source.

The Boost license used does not enforce “copy left” like many other open source licenses, but is comparable to the MIT license. Crypto++ supports a host of operating systems, among these are: Linux, Windows, OS X, iOS, and Android. Table 10.3 shows how Crypto++ supports the technologies described in this chapter. The Crypto++ library even supports a filter/pipeline concept, making it easy to “glue” algorithms together.

Table 10.3: Crypto++ – an overview.

Algorithm	Implementations
Random numbers	Several
Hash	MD5, SHA-1, SHA-256, and more
CRC	CRC-32
Symmetric key	DES, tripple-DES, AES, and more
Stream ciphers	Several
Asymmetric key	RSA, DSA, Diffie–Hellmann, and more
MAC	HMAC and more
Compression	GZIP and ZLIB
Encodings	Base-32, Base-64

10.17 Trusted platform module

The *Trusted Platform Module* is a standard for advanced hardware[9] Cryptoprocessors, originally developed by a consortium of companies and later standardized as ISO 11889. These modules were originally small PCBs (printed circuit boards) for use in PCs and servers. Later single-chip solutions emerged, and today at least parts of these are built into many SoCs. One of the main functions of a TPM in a PC, is to work together with “DM-Crypto” software on Linux, or the similar “BitLocker” software on Windows, to encrypt the entire disk in the PC.

The TPM can also generate a secure hash of the boot-loader and early loaded drivers, assuring the integrity of these. This is a necessary defense against so-called “rootkits,” since standard antivirus is loaded later in the boot process. This division of labor means that only the early boot is protected by the TPM (when enabled). It is up to the antivirus program to protect against virus in standard software on the PC, as well as plugins in the browser.

On Windows 8.1 and forward, the *SecureBoot* concept specifically keeps SHAs of: OS-loader, BootMgr, WinLoad, Windows Kernel Startup, antivirus-program signature, boot-critical drivers, “additional OS initialization,” and Windows Log-on Screen.

9 TPM is normally hardware-based, but software versions do exist.

By requiring antivirus vendors to have a Microsoft-generated signature, the handover from the SecureBoot to the antivirus is assured, while the Log-on Screen is controlled because this is where violations are reported. This is surely relevant if you are using “Windows Embedded,” but it is also interesting in general, since embedded systems tend to follow PC systems, with some delay.

The first wave of TPMs were mounted in PCs years ago, but were by default disabled. Most of these have probably never been activated. With TPM 2.0, the default is to be enabled. TPM 2.0 has “assigned” names to a lot of algorithms, meaning that they are optional. The most important mandatory ones are given in Table 10.4.

Table 10.4: Trusted platform module (TPM) v2.0.

Algorithm	Implementations
Random numbers	Key derivation methods
Hash	SHA-1, SHA-256
Symmetric key	128-bit AES
Asymmetric key	RSA, Diffie–Hellmann
MAC	HMAC
Misc	XOR

Various vendors are marketing chips containing subsets of the TPM features. They may, for example, contain symmetric key algorithms, but not asymmetric key algorithms. Common to all these chips is that they store the keys inside the chip as “write only” memory. “Write-only” memory used to be something to tease young developers with, once they had learned about read-only memory. It made no sense and did not exist. As it sometimes happens, it now does make sense.

You program the key during manufacturing or service. It is used by the algorithm inside the chip, but there is no command for retrieval of keys. At a later service, new keys may be set, but still never read, except by the internal hardware algorithms.

10.18 Embedded systems

Embedded systems are vulnerable via their internet connection in the same way PCs and servers are. The defenses against that is what this chapter has been about until now. However, embedded systems are not placed inside the company’s climatic room. Instead they are sold to end-users or left in the field. This makes them more vulnerable to direct physical access.

You can say that if an end-user has bought a $200 router legally, it is his right to toy around with it. In fact, that is how we got “OpenWRT”; see Section 6.10. However, what if this user applies his knowledge on all other devices of the same type? Such an attack will still be via the internet connection, but it is strengthened with the knowledge about the internal workings of the device.

Yet another interesting document from NIST is "FIPS-140-2." This describes the security requirements for physical cryptographic modules. It defines four levels of security, which, with some adaptations, can be viewed as general security levels for all embedded applications:

1. *No Security at All*
   The system has no defenses, and is not even able to detect any tampering.
2. *Proof of Tampering and Role-based Security*
   Broken seals or opaque tamper-evident coatings will show unintended access. Likewise pick-resistant locks on doors are clearly broken. Use, for example, pressure contacts to detect and log whenever someone opens the box, and other sensors for temperature and other environmental parameters outside the normal range. Role-based authentication assures that services are performed/utilized only by those authorized to do so.
3. *Critical Access Prevented and Identity-based Security*
   On top of level 2 requirements, level 3 demands that critical security parameters (CSPs) such as private keys, shared secrets, and shared keys inside the module are *zeroized* if unauthorized access is detected. Identity-based security must be used. Thus it is not enough that the log states that a "service technician" did this and that, instead it must name the technician.
   Strong removal-resistant and penetration resistant enclosure must be used. Various physical self-destruction methods may be applied even small explosions.
   The operating system can be of a general purpose type, but the installation must obey a list of specific demands. Any input/output ports allowing critical security parameters to be entered or output in plaintext, may not be used for any other purpose. Thus other ports are needed for general communication.
4. *Complete Envelope and Environmental Defense*
   All requirements for level 3 also apply here. At this level, the module is completely sealed. Intrusion will be detected with a "very high probability." A general purpose OS may be used, but the requirements are now even stricter and it is probably a better idea to design from the start with an *evaluated trusted OS*. The module will either resist unusual environmental changes, or zeroize all critical security parameters.

FIPS-140-2 specifically deals with the cryptographic components and therefore also divides systems into single and multichip solutions. The above is generalized to show how the four levels can be used, also when considering the safety of a full embedded system.

All project teams must consider the network security of their IoT device. Many teams need to consider signing and/or encryption of firmware and similar precautions as discussed in the next section, but hopefully only few of us need to work with secure encapsulations like the ones in level 3, and especially level 4. Such measures are outside the scope of this book. A good place to start is www.blackhat.com.

10.19 Vulnerabilities in embedded systems

A small catalog of known security problems related to the device or its firmware:

– *Firmware on Homepage*
  Many companies offer downloadable firmware for their devices on their homepage. This is in many ways practical. It allows customers to get fixes for their problems, as well as new functionality in frequent releases. This is so common that most of us look for new drivers when we get home from the store with our latest gadget. This also allows the vendor to distribute security patches. Not all users are going to use them, but at least they had the opportunity.
  The downside is that to the capable hacker this makes the binary image extremely easy to get access to. If it is unencrypted, it may be reverse-engineered with relative ease. You can remove debug symbols, and even "obfuscate," for example, Java code, but this only makes the code slightly harder to read. If, for example, a private key, a shared key, or a shared secret (for MAC) is kept within the firmware, it can be dug out and misused. A hacker can relatively easily make his own version of the firmware.
  Countermeasure: Not having firmware for download on the homepage will surely deter the less effective hacker, but it will also make it more expensive to upgrade users devices by requiring manual intervention. Surely this is not really safe anyway. Signing the firmware can protect against tampering, and if the code must be confidential, it must be encrypted. Compared to the time it takes to "burn" flash memory, etc. on-the-fly, decryption of a new firmware during download into the embedded system, will normally not burden the system much.
– *Simple Memory Read*
  Most embedded systems contain the binary code in, for example, flash outside the CPU. Similarly, FPGA systems are often loaded from external memory. Configuration data within, for example, EEPROM may be more sensitive than the basic firmware and as this is smaller, it is also faster to scan and interpret. These memories are all easy to read if the hacker has physical access. The memory circuit can be removed from the PCB and read by a standard flashing device or a custom-programmed CPU system.
  Countermeasure: It is possible to buy *cryptographic memory*. For example, Atmel's "CryptoMemory" is EEPROM with up to 256 kbit with built-in encryption engine. Similar devices from Maxim are called "DeepCover." Microchip has the more prosaic named "Hardware Crypto Engine" in different versions. These all have subsets of the contents of TPMs (see Section 10.17) but with built-in memory for more than just keys. In the future, we can expect TPMs in more embedded designs probably as part of larger general purpose SoCs. Another possibility is to put CPU and memory inside an FPGA, so that everything is inside the same chip. In 2015, Intel bought Altera—one of the major two players in this market (the other is Xilinx).

There can be many reasons for the acquisition—one could be to offer secure FPGAs with advanced CPUs inside for improved security.

– *UART Access*
  As described elsewhere, it is very convenient to have UART access to an embedded system for debug purposes. Sometimes this interface doesn't even require a password.
  Countermeasure: As a minimum a password must be required. This should not be the same for all devices. Instead the password can, for example, be a hash based on the serial number and a shared secret between the device firmware and a service application running on the company server, logging all requests.
– *JTAG Debugger*
  As described in Section 5.3, it is really practical to have a JTAG connection for debugging purposes. Unfortunately, this is also practical for the hacker.
  Countermeasure: Only allow this interface on R&D models.
– *Wireless 802.11b*
  Older wireless systems are not just slowing down your wireless, the WEP encryption is also easily broken.
  Countermeasure: Only allow WPA2 or better encryption on Wi-Fi.
– *Factory Defaults*
  A large-scale denial-of-service attack against the DNS system took place in October 2016. This was probably the first major attack using home routers, etc. in a large "bot-net." It turned out that the simple hack was based on a factory default username and password on a number of these devices. Only the ones on which username and/or password was unchanged were used, but that was enough.
  Countermeasure: For devices with user-login, there are alternatives: Generate unique passwords for each device, for instance written on a sticker on the device,[10] or insist on first-usage logins to change password. Alternatively, when possible, allow only local logins until the default password is changed.
– *Buffer Overflow*
  Many of the previously mentioned security holes are basically the downside of a convenience feature. A buffer overflow, on the other hand, is a bug, allowing hackers to access the system. This is typically based on programs written in classic C using, for example, copy functions without the *n* parameter for maximum length. The destination for the copy is typically on the stack or heap where other variables are also stored. Thus writing too long a string into one variable, will lead to another variable being overwritten. For the hacker, this is easiest if he knows the source, and thus open source is more vulnerable than closed source. The counterargument is that open source is reviewed and maintained by more people, and thus kept in better shape.

10 This is also a weakness—a sticker must be removed before usage.

Countermeasure: Avoid using C. This is probably not an option. Instead use static code analysis to catch the use of unsafe functions and flag this as errors.

– *Side-channel Analysis*
  Cryptography is in essence a mathematical science, assuming perfect implementation. Unfortunately, real-world devices, such as CPUs and FPGAs, leave traces from intensive algorithms on the power supply that can be used by the skilled hacker. Paul Kocher et al. described in 1998 how they were able to identify the 16 rounds of DES (see Section 10.5), and detect individual bits in a key, based on different branches in the software. This technology has since developed, now using statistics in order to meet the simple countermeasures first taken.
  Power is by no means the only side channel. Emitted sound and EMC (Electro Magnetic Conduction) are other known side channels which hackers may utilize.
  Countermeasure: Use the newest crypto-libraries or TPM-hardware, assuring that all paths through the algorithm result in equally many jumps, etc.

Common to many of the above vulnerabilities is that they require the perpetrator to have direct physical access to a system. This is sometimes used as defense: "The hackers probably don't want to physically access *all* our systems." Unfortunately, as stated before, it often only requires a single device to get the necessary knowledge to harm many. If the device is a commodity, like the $200 router, the hacker can buy one legally and bring it home. Here, he has all the time and all the tools needed. Even if the device is not commodity, for example, an aircraft, we cannot rule out the single disgruntled employee taking information or source code with him.

A good place to find more information is blackhat.com. This is a site for a community of security professionals hosting events and training sessions. From the events they publish videos on YouTube. A good part of the content relates to general IT, but there are also relevant sessions on IoT. A good example is a video from 2014 where Nitesh Dhanjani takes us through some disturbing examples. One example is a baby monitor that connects to a private Wi-Fi network. Once you have connected to this locally, you can always connect remotely. This means that anyone can listen to what goes on in your home, once they have passed it. Another example is a webcam which thoroughly implements SSL as described in Section 10.12. *Unfortunately this device also emits UDP packets stating in plaintext username and password, probably meant for debug, but not removed before going to market. There is much to learn about what not to do in these sessions, but you can actually also pick up great inspiration about what to do when connecting phones and devices via cloud solutions.*

For a catalog of defensive/secure coding guidelines, look at Steve McConnell's book *Code Complete (2)*, quoted in many places on the web. Likewise CERT at Carnegie Mellon has a list of "Top 10 Secure Coding Practices."

10.20 Export control

I am not a lawyer and the reader should not trust this simple text as more than informational. As of August 2018, the following is my layman's interpretation of the laws on export control.

The US laws on export control are relevant, not just for US citizens but for all of us, because they involve all (re)export from the United States or with US citizens or using dollars. Thus a program sold via Apple's Appstore or Google Play, is subject to these laws, even if sellers and buyers are all Norwegian. As if the above is not enough, many countries have rules that follow the US lead.

The US Bureau of Industry and Security (BIS) is responsible for the *EAR* (export administration regulation) rules. In the EAR, there are a number of categories of which category 5 is "Telecommunication," and of this "Part 2" is about information security, which is what is relevant for this chapter. This is often shortened to "Cat 5, Part 2" or simply "C5P2."

These rules are very hard to read, which is a shame considering the lengthy jail time it can involve to violate them. Nevertheless, they have been somewhat relaxed since 1992, and the aim is clearly to make rules that allow us to live normal private lives and not obstruct normal business.

As a result, a lot of stuff is fully excluded from Cat 5, Part 2 and in most other cases a self-registration e-mail notification to BIS is all that is required.

BIS Cat 5 Part 2 has a "Note 4" that *excludes* the following:

- *Consumer Applications – Piracy and Theft Prevention for Software and Music*
  This includes music players, TVs, HDMI devices as well as printers and cameras. It also includes household appliances.
- *Research*
  Specific research fields are stated here.
- *Business and Systems Applications for Operation, integration, and control*
  This includes transportation systems, fare collection, robotics, fire-alarms, etc.
- *Secure Intellectual Property Delivery and Installation*
  This includes software downloaders, installers, and updaters as well as software licenses and IP-protection. In other words, many of the security mechanisms this chapter has outlined. TPM is specifically mentioned here as a form of IP protection.
- *Specific Limited Communication*
  BIS examples are a child's laptop that only accesses a specific site for literacy education, or a vending machine emitting encrypted messages about what needs to be stocked.

It makes a lot of sense that even if you use advanced encryption in your firmware for IP protection, it does not really make it interesting as a means of communication for bad guys.

However, even though the protection of your intellectual property is accepted, the above still leaves a problem with the general use of a CPU system. Protecting firmware is one thing but general file encryption is specifically mentioned as being *included* in Cat 5, part 2.

This brings us to the "Decontrol Notes " dated 2016. These notes exempt smart-cards, civil mobile phones, and wireless personal area networks (e. g., Zigbee, see Chapter 9). That leaves us in the clear as general consumers.

Another exception is made for products for the "mass market." These are defined as over-the-counter sales, or similarly, of products for which the features and prices are either public or available on simple request.

The last part of an exemption in the "Decontrol Notes" is given here in italics with comments in normal font:

General purpose computing equipment or servers, where the "information security" functionality meets all of the following:

- *Uses only published or commercial cryptographic standards; and*
  All the algorithms in this chapter are standardized by NIST and are available commercially and for free. Open source algorithms and standard TPMs are thus also favored.
- *Is any of the following:*
  - *Integral to a CPU that meets the provisions of Note 3 in Category 5 – Part 2;*
    Note 3 is the previously described note on the mass-market, with the clause that cryptographic functions are not easily changed by the user. However, Note 3 states that if a symmetric key has a length greater than 64 bits, or an asymmetric key is more than 768 bits long, a classification request or self-classification report must be submitted to BIS. Since NIST has standardized AES-128, this could mean a lot of self-registrations.
  - *Integral to an operating system that is not specified by 5D002; or*
    5D002 is about software "specifically designed or modified for development, production, or use of cryptography."
  - *Limited to "OAM" of the equipment.*
    OAM is "operations, administration or maintenance." This includes handling of accounts and privileges on a computer and authentication to do so.

Remembering that I am not a lawyer, and this is my generalization: If you use standard or open source encryption for basic IP protection you are home free. If you use the same for file encryption or anything that may be generated and sent by users in a mass-market product, you need to send a self-registration mail. If you plan to go beyond the latter with, for example, products for bypassing security, or specifically for high-speed data encryption you need to contact the relevant authorities first.

10.21 Further reading

- openssl.org
  This site is the home of the openSSL library that may be used within your code. It is also the home of the openssl commandline utility.
- Paul Kocher, Joshua Jaffe, and Benjamin Jun: *Differential Power Analysis* (www.cse.msstate.edu/˜ramkumar/DPA.pdf)
  This is a classic article that started the DPA interests.
- www.blackhat.com
  Joe Grand: *Introduction To Embedded Security*
- HeartBleed.com
  Codenomicon's site with information about the Heartbleed bug
- William Stallings: *Cryptography and Network Security – principles and practice*
  This is a classic textbook from Pearson that goes into all the principles discussed in this chapter and a lot more. The newest edition is the 7th.
- www.ti.com/ww/en/embedded/security/index.shtml
  Texas Instruments here offers free downloads of C implementations of SHA-256, DES, 3-DES, and AES-128 as well as a manual for these. There is also a HW selection guide.
- www.cryptopp.com
  The home of the Boost Crypto++ library.
- FIPS.140-2
  The document from NIST that describes the Security Requirements for Cryptographic Modules.
- FIPS-180-4
  The document from NIST that describes secure hashes and all implementation details.
- SP 800-131A Rev. 1
  US government recommendations about what to use within US government offices. These are minimal requirements.
- Cisco 2017 Annual Cyber Security Report (free download)
  (Browse and fill out contact information)
  This is very interesting reading—filled with statistics and graphs on how the various threats affect industries, geographic regions, etc. It is mainly about server and client security and basically only covers IoT as a source of threats.
- processors.wiki.ti.com/index.php/AM335x_Crypto_Performance
  This benchmarks the crypto engine in the TI AM335x SoC used in the BeagleBone.
- wiki.sei.cmu.edu
  Top 10 Secure Coding Practices.
- haveibeenpwned.com
  Security expert Troy Hunt's site where anyone can test whether their account password has been leaked.

- shodan.io
  Website which crawls the internet and makes a database of IoT-devices that may be searched later.

11 Digital filters

11.1 Why 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" compensated for all the components that were not exactly the specified value. Somewhere at a production-site, someone had manually 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."

As embedded software developers, we tend to see the world as being full of analog signals which we want to measure or generate—digitally. Even when a sensor outputs a digital value, it's because another embedded programmer has digitized it for you. Much 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":

– With digital signal processing, there is little need for calibration. Imagine getting rid of (almost) all calibrations and instead have cloneable, unchanging specifications. This is what the digital world brings us.
– Digital signal processing is considerably more flexible than analog. You can fit many filters into a *DSP* (digital signal processor) or plain CPU. In the analog case, you need a lot of components, either completely separate or switchable.
– Field-upgradeable. Digital filters can be changed with a firmware update. This is mostly a good thing; see also Section 10.19.

Nevertheless, there are situations where we need analog circuits:

– Before A/D (analog to digital) conversion. According to the "sampling theorem," the sampling frequency must be at least twice that of any frequency contained in the signal. Typically, the only way to assure this, is to insert a lowpass filter before the A/D converter. Such a filter is called an *anti-aliasing filter*.
– Likewise, after D/A conversion we need a *reconstruction filter*, cutting off all frequencies generated by the conversion, above half the sampling frequency.
– Say you want to measure on a small 8 kHz signal hiding in noise with a large amplitude, for example, 50 Hz or 60 Hz power-line noise. You need an analog filter, removing, for example, 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.

https://doi.org/10.1515/9781547401024-011

– It is not always the case that the thing you want to measure, gives you a nice "stiff" voltage matching 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 need 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 your device relies heavily on DSP. If you have Wi-Fi, 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, for example, IEEE 802.11ac 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.

Analog signals are also called *continuous* signals, while digital signals are called *sampled* or *discrete* signals. We will get back to this.

11.2 Why filters?

There are reasons why 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 the *Intel IPP*[1] library. But how do we 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. The data is typically passed on to the cloud, where advanced digital signal processing may take place, depending on your application. To enable the implementation of filters, we will also look into number representations. These are the basics of any digital signal processing application.

There are scenarios where filtering alone is not enough—fast local action is preferred. One example is statistical process control, which is the subject of Chapter 12.

Although we will not discuss the implementation of FFTs in our embedded devices, we will see how the FFT can be used when designing and analyzing.

1 Integrated Performance Primitives is a library for image processing, etc.

11.3 About the sampling frequency

In general, we use the term f_s for the sampling frequency (or sample rate):

$$f_s = (1/\triangle T) \tag{11.1}$$

Here, $\triangle T$ is the time between samples from the A/D converter. In *POTS* (plain-old-telephone-system), $f_s = 8$ kHz. This means that $\triangle T = 125\,\mu$s.

Sometimes you will see the symbol ω used in formulas. This is the "*angular velocity* of the rotating unity vector." It is simply:

$$\omega = 2 * \pi * f \tag{11.2}$$

Another mathematical concept is to show a frequency axis with positive and negative frequencies. This is just a trick, but it very neatly explains phenomena such as aliasing. Aliasing is what happens when you violate the "sampling theorem": signal content from the original analog signal is "mirrored" around f_s, and spoils your data. In other words, the sampling theorem says: All signals above $f_s/2$ must be removed before sampling (or in reality must be below a certain level). This is also called the "Nyquist Criterion," and $f_s/2$ is known as the "Nyquist Frequency."

DSP algorithms, including filters, are completely relative to the sampling frequency. Thus, if you have created a lowpass filter, with a cutoff frequency at 3 kHz with $f_s = 20$ kHz, and you raise f_s to 40 kHz, the same filter will now cutoff at 6 kHz. For this reason, you will often see *normalized* filter characteristics where DC is 0 and f_s is 1.

11.4 Time and frequency domains

There is a one-to-one correspondence between a signal in the *time domain* and the *frequency spectrum* of the same signal in 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 the signal as a function of time. In the frequency domain, we have frequency on the x-axis, and the y-axis is typically magnitude and/or phase. A mathematician would thus say that we see the same signal in two different representations, whereas an engineer would say that in the frequency domain we see the (frequency) spectrum of the signal. Mathematically, you go from the time domain to the frequency domain with the help of the *forward Fourier transform*:

$$F(f) = \int_{-\infty}^{\infty} f(t) * e^{-i2\pi ft} dt \tag{11.3}$$

You use the *inverse Fourier transform* to go back:

$$f(t) = \int_{-\infty}^{\infty} F(f) * e^{i2\pi ft} df \tag{11.4}$$

When we work with sampled/discrete signals, we use the *DFT* (discrete Fourier transform), given in equation (11.5). There is a fast implementation of this known as the *FFT* (fast Fourier transform):

$$X(m) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) * e^{-i2\pi mn/N} \tag{11.5}$$

The inverse DFT—also known as the iDFT or the iFFT—is given by equation (11.5).

$$x(t) = \sum_{n=0}^{N-1} X(m) * e^{i2\pi mn/N} \tag{11.6}$$

In both expressions, n goes through the values from 0 to $N - 1$ (both included) for each m calculated. In other words, a 1024 point DFT uses 1024 samples in its calculation of each of the 1024 lines in the spectrum. Thus the calculations have an inner and an outer loop, each with 1024 steps. This means that a DFT calculation's execution time is proportional to N^2, whereas the FFT is proportional only to $N * \log(N)$. This makes a huge difference when analyzing long signals.

If first the DFT is applied, and following this the inverse DFT, you should be back to where you started. However, somewhere you need to divide all values by N, where N is the length of the signal (and spectrum), in order to get back to the right result.

This division is sometimes done in the DFT, sometimes in the iDFT and sometimes applications divide by $\sqrt{N}$ in both transformations. If you are an engineer caring about details such as the magnitude values, you want to divide after the DFT as done in equation (11.5).

It is often said that the forward transformation is an *analyzing* function because it allows you to analyze a spectral representation of a time signal. In the same spirit, the inverse transformation is called a *synthesizer* function, because it synthesizes a time signal from a frequency spectrum.

Due to the duality between the time and frequency domains, it sometimes makes sense to convert a signal from the time domain to its spectrum in the frequency domain, perform the relevant algorithm, and go back to the time domain.

Figure 11.1 shows an everlasting sine of 1 Hz in the time domain in the upper graph. The lower graph shows the magnitude of the two-sided spectrum of the sine as it comes out of a DFT/FFT. The two-sided spectrum is often used in signal processing.

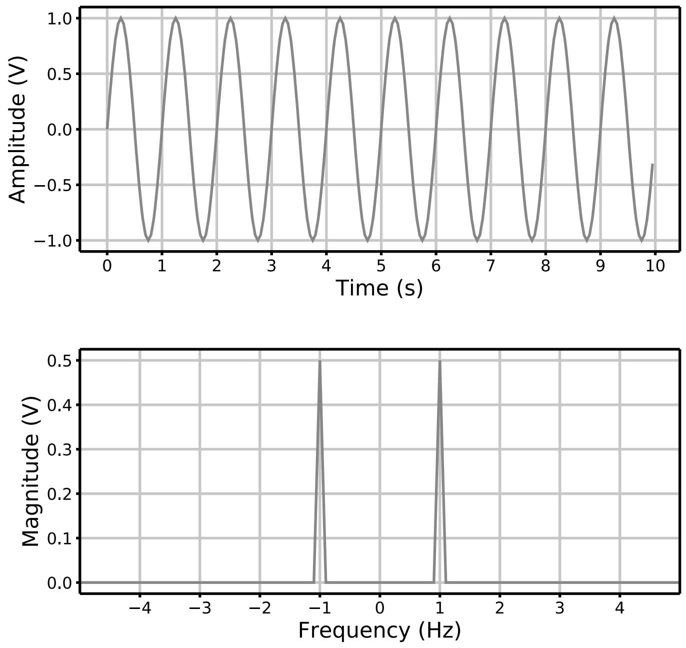


Figure 11.1: Everlasting sine in time and frequency domain.

You can go from the two-sided magnitude spectrum to the classic single-sided spectrum by adding all magnitude values at negative frequencies to their positive counterpart,[2] leaving the value at 0 unchanged. In Figure 11.1, this means that the magnitude of our sine becomes $0.5 + 0.5 = 1.0$, which indeed is the amplitude of the time signal.

If the spectrum is a single line located at $f = 0$, we have a completely horizontal line in the time domain, a DC signal. Likewise, a single vertical line in the time domain, corresponds to a flat spectrum—the same level on all frequencies. This is also known as "white noise." In fact, it is a good rule of thumb that if something is "edgy" or "spiky" in one domain, it is wide in the other. Electronic designers for instance, know that the *rise time* on various digital bus signals should not be steeper than necessary, as this radiates high-frequency noise, which may introduce unwanted signals elsewhere.

Signals originating in the time domain are real, whereas in the frequency domain they are normally complex.

Any signal can be decomposed to a number of *sinusoids*, "sines" in daily speak. We will see this in a moment.

Any repetitive signal—be it analog or digital—has a *discrete* spectrum—a number of lines, also known as *harmonics*. Each of these are found at the frequency $n * f$,

2 For real signals, this simply doubles the magnitude.

where “n” is an integer and f is the *repetition* frequency. This is also known as the *base* frequency or the *first harmonic*.

11.5 Analog and digital definitions

As previously stated, we deal with *continuous* signals in the analog world, whereas in the digital world they are *time-discrete*—another word for sampled. In the analog/continuous world, the time signal is noted $x(t)$, the frequency spectrum is $X(f)$ and the *transfer function* of a system is $H(f)$.

$H(f)$ is normally defined as $Y(f)/X(f)$ where $Y(f)$ is the frequency spectrum of the output of your system and $X(f)$ is the frequency spectrum of the input. Thus, the name “Transfer Function” is well chosen, describing how the spectrum of an input signal is transferred by going through the system.

In the digital/discrete world, the time signal is noted as $x[n]$, the transfer function is $H(z)$ and the 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 numerator and another in the denominator. The roots to the first are known as *zeros*, while the roots to the second are known as *poles*. Knowing the positions of these in a Cartesian coordinate system in the analog case, or in a polar coordinate system in the digital case, can tell you everything about the transfer function.

Take for instance the analog domain: imagine you are sitting on 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 ascend 20 dB/decade (or 6 dB/octave) and when you pass a pole you descend similarly. These values add up so that once you have passed an equal number of poles and zeros the transfer function is flat.

In exactly the same way a 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, $H(f)$. Also here we can use the Fourier transform and its inverse variant. This is true in both 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.

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 and/or phase, as a function of frequency. When we show both, it is often known as a Bode plot.

11.6 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 digital/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, becoming a *wave*. The spectrum of a single pulse is continuous, while the spectrum of the same repeated pulse is discrete (even in the analog world). Apart from that they are identical.

A single square pulse in the time domain becomes a $\sin(x)/x$ function in the frequency domain. The value of the spectrum at $f = 0$ is the DC of the signal, which is the same as the average. The magnitude spectrum has zero crossings at n/T, where n is an integer (not 0) and T is the period of the pulse.

Figure 11.2 shows two different square pulses in time and their spectra.

The wider the pulse is in time, the narrower the $\sin(x)/x$ becomes. The wider pulse also becomes higher in the frequency domain, as it contains more energy. As the height of the $\sin(x)/x$ at the center is the DC value, this is no surprise. The slimmest of the two pulses in Figure 11.2 is 1 high from 0.1 to 0.3 seconds. This gives us a DC of 0.2 V over the 1 s period. Similarly, the widest pulse has a DC of 0.5 V over the same period.

The first zero crossing of the slimmest pulse is found at $1/(0.3–0.1)s = 5$ Hz, while the first zero crossing of the widest pulse is at $1/(0.9–0.4)s = 2$ Hz.

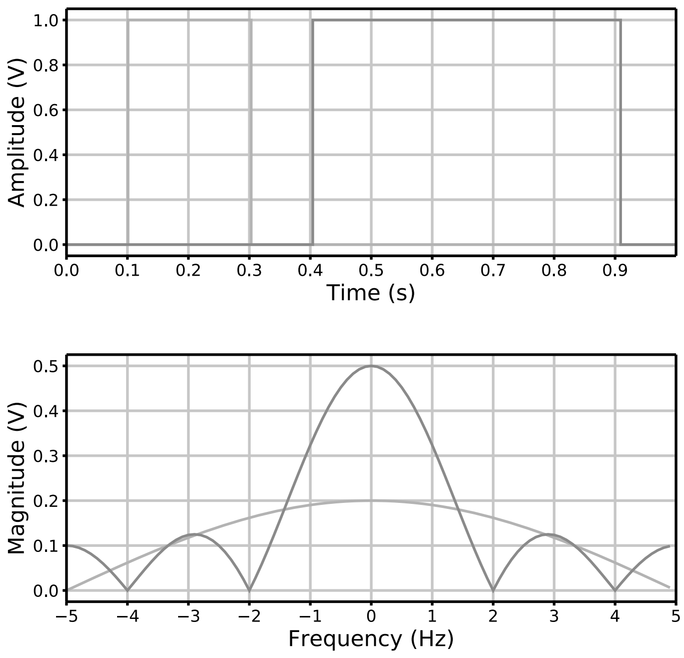


Figure 11.2: Two squares in time (10 s) and frequency domains.

In our discussion of the two pulses, we treat them as continuous signals. This means that to go from the time signals to the spectra we ought to use the Fourier integral from equation (11.3). To calculate that, we would need numerical integration. However, under certain conditions you can get away with using the DFT or FFT even though the signals are continuous.

The Python program in Listing 11.1 is created to demonstrate the theory but uses some tricks to get there with FFT. You might want to focus on the theory now, and go back to the listing later.

Listing 11.1: Fourier transform of square signals using Python

```python
#!/usr/bin/env python3
# This demonstrates the spectra of two square pulses
# Tricks are used to simulate a real Fourier integration
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages

Fs = 10  # sampling rate
Time = 10 # Trick 1: Use more samples than shown
Ts = 1.0/Fs # sampling interval
t = np.linspace(0,10.0,num=Time*Fs)
n = t.size

# 2 samples wide square
y1 = np.zeros([n],float)
y1[2] = y1[3] = 1.0

# 5 samples wide square
y2 = np.zeros([n],float)
y2[5] = y2[6] = y2[7] = y2[8] = y2[9] = 1.0

# Clear two complex arrays
Y1 = np.zeros([n],dtype=complex)
Y2 = np.zeros([n],dtype=complex)

Y1 = np.fft.fft(y1)/n # Normalization after FFT
Y2 = np.fft.fft(y2)/n

# Get the corresponding frequency axis
freqAxis = np.fft.fftfreq(n, Ts)

fig, ax = plt.subplots(2, 1, figsize=(10,10))

# Trick 2: drawstyle makes it look right but moved in time
ax[0].plot(t,y1,'r',t,y2,'g--',lw=3,drawstyle='steps-pre')
ax[0].set_xlim(0,1)  # Trick 3: Zoom in
ax[0].set_xlabel('Time␣(s)', fontsize=20)
ax[0].set_ylabel('Amplitude␣(V)', fontsize=20)
```

```
ax[0].xaxis.set_tick_params(labelsize=15)
ax[0].yaxis.set_tick_params(labelsize=15)
ax[0].set_xticks(np.arange(0.0, 1.0, step=0.1))

# fftshift assures that axes are monotonous
# Trick 4: Compensate for the long Time
ax[1].plot(np.fft.fftshift(freqAxis),
           Time*np.fft.fftshift(abs(Y1)),'r',
           np.fft.fftshift(freqAxis),
           Time*np.fft.fftshift(abs(Y2)),'g--',lw=3)
ax[1].set_xlim(-5,5)
ax[1].set_xlabel('Frequency␣(Hz)', fontsize=20)
ax[1].set_ylabel('Magnitude␣(V)', fontsize=20)
ax[1].xaxis.set_tick_params(labelsize=15)
ax[1].yaxis.set_tick_params(labelsize=15)
ax[1].set_xticks(np.arange(-5, 6, step=1))

ax[0].grid()
ax[1].grid()

# Adjust spacing
plt.subplots_adjust(left=None, bottom=None, right=None,
            top=None, wspace=None, hspace=0.4)

pp = PdfPages('TwoSquares.pdf')

plt.savefig(pp, format='pdf')
pp.close()
```

Having looked at a single square pulse, we now investigate a square wave. Figure 11.3 shows an everlasting square wave in the time and frequency domains.

The square wave in Figure 11.3 is the wide pulse from Figure 11.2, now repeated eternally. If you compare the spectra from the two figures it is clear that the spectrum in Figure 11.3 is a sampled version of the "dashed" one in Figure 11.2, originating from the wide single pulse. As stated earlier, all frequency content in a repeated signal is found at the repetition frequency multiplied by integer values. The repetition frequency is 1 Hz as we have 10 periods over 10 seconds. However, since the square wave has a *duty cycle*[3] of exactly 50 % there are only odd harmonics. The square wave can be said to be *composed of* sines of the base frequency multiplied by $1, 3, 5, 7, 9$, etc.

The Python code for the analysis of the square wave is shown in Listing 11.2.

Listing 11.2: Fourier transform of square wave using Python

```
#!/usr/bin/env python3
#This demonstrates the spectrum of a square wave
```

3 The percentage of the time that the signal is "high."

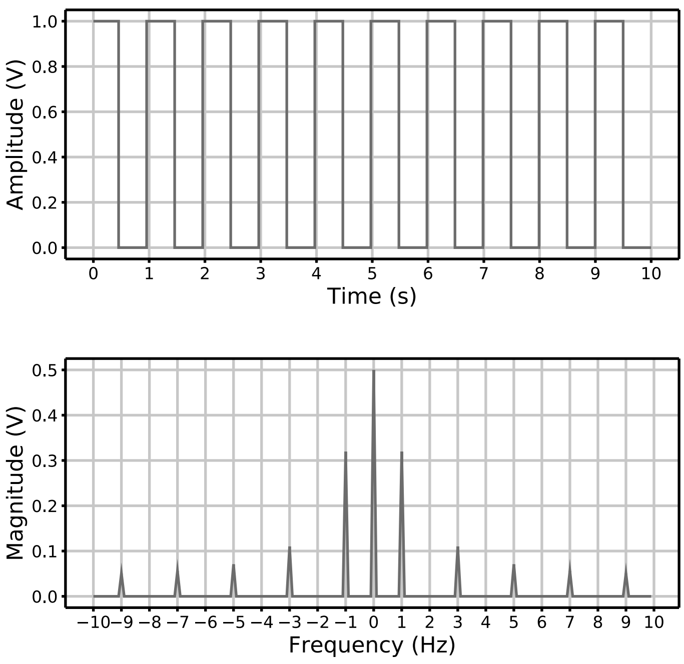


Figure 11.3: A square wave with 50 % duty cycle.

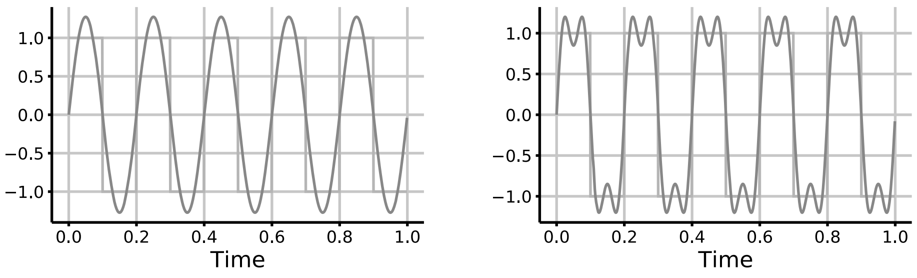
```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages

Fs = 20  # sampling rate. Max freq is Fs/2.
Time = 10
t = np.linspace(0,10.0,num=Time*Fs, endpoint=True)
n = t.size

freq = 1   # Frequency of square wave

y = np.zeros([n],float)
y2 = np.zeros([n],float)
# A sinus is a good starting point
y = np.sin(2*np.pi*freq*t)
# Make squares from the sine
for x in range(n):
    if y[x] >= 0:
        y2[x] = 1
    else:
        y2[x] = 0

Y2 = np.fft.fft(y2)/n # Normalization after FFT

```

```python
# Get the corresponding frequency axis
freqAxis = np.fft.fftfreq(n, 1/Fs)

fig, ax = plt.subplots(2, 1, figsize=(10,10))

ax[0].plot(t,y2,'b',drawstyle='steps-pre',lw=3)
ax[0].set_xlabel('Time␣(s)',fontsize=20)
ax[0].set_ylabel('Amplitude␣(V)',fontsize=20)
ax[0].xaxis.set_tick_params(labelsize=15)
ax[0].yaxis.set_tick_params(labelsize=15)
ax[0].set_xticks(np.arange(0, 11, step=1))

# fftshift assures that axes are monotonous
ax[1].plot(np.fft.fftshift(freqAxis),
  np.fft.fftshift(abs(Y2)),'b',lw=3)
ax[1].set_xlabel('Frequency␣(Hz)',fontsize=20)
ax[1].set_ylabel('Magnitude␣(V)',fontsize=20)
ax[1].xaxis.set_tick_params(labelsize=15)
ax[1].yaxis.set_tick_params(labelsize=15)
ax[1].set_xticks(np.arange(-10, 11, step=1))

ax[0].grid()
ax[1].grid()

# Adjust spacing
plt.subplots_adjust(left=None, bottom=None, right=None,
            top=None, wspace=None, hspace=0.4)

pp = PdfPages('SquareWave.pdf')

plt.savefig(pp, format='pdf')
pp.close()
```

The DFT/FFT assumes that the whole signal analyzed is repetitive. You can say that the first sample of the signal is placed after the last sample again and again. This "eternal extension" must be done so that the period of the signal is unchanged at the "stitches" and the ends meet correctly. You can try to change the length of the signal or change the frequency a bit and you will see noise creeping up from the "floor" of the spectrum. In fact, simply changing "endpoint=True" to "endpoint=False" in the "linspace" call is enough to make a visible difference. The Boolean "endpoint" decides whether or not the last sample in the range is included.

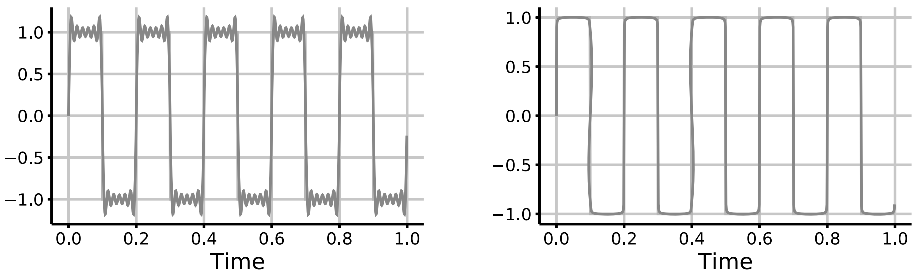
The fact that our square wave is built from sines is shown progressively in Figures 11.4 and 11.5, where we are going the other way and actually are building a square wave, from more and more sines.

In Figure 11.4(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 three harmonics added



(a) First harmonic (b) First 3 harmonics

Figure 11.4: Building up a square wave.



(a) First 11 harmonics (b) First 201 harmonics

Figure 11.5: Building up a square wave.

as one wave. Since all even harmonics are flat there are only two “active” harmonics in play. Figure 11.5(a) shows the first 11 harmonics added, and (b) shows the first 201 harmonics added—still together with the ideal square wave, which is now difficult to see. It is clear that the more harmonics we include, the closer we get to the ideal square wave. It can be proven that any “well behaving”[4] signal can thus be “built” from sines.

Listing 11.3 shows the Python code used to generate the harmonics buildup.

Listing 11.3: Square wave built from harmonics with Python

```python
#!/usr/bin/env python3
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

points = 1000
# Calculate amplitude of sine compared to square
fact = 4/np.pi # or np.sqrt(2)/1.11

```

4 Normal signals are well behaving. The requirement is to have only a finite number of maxima, minima, and discontinuities over a finite period.

```
# Generate the time-axis
t = np.linspace(0, 1, points, endpoint=False)

square = signal.square(2 * np.pi * 5 * t)

sines  = np.zeros([points],float)
sines2 = np.zeros([points],float)

#Step from 1, incrementing by 2 to xx
for x in range(1,202,2):
    sines += fact*1/x*np.sin(x*2 * np.pi * 5 * t)

fig, ax = plt.subplots()

ax.plot(t, square,'g--',lw=3)
ax.plot(t, sines,'r',lw=3)
ax.set_xlabel('Time',fontsize=20)
#plt.box(on=None)
#plt.ylim(-1.5, 1.5)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['left'].set_linewidth(0.5)
ax.xaxis.set_tick_params(labelsize=15)
ax.yaxis.set_tick_params(labelsize=15)
plt.subplots_adjust(left=None, bottom=0.2, right=None,
            top=None, wspace=None, hspace=0.4)
ax.grid()

pp = PdfPages('SquareOfSines201.pdf')
fig.savefig(pp, format='pdf')
pp.close()
```

11.7 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 complex 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 also *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 *saturation*. If you try adding two analog sines with an amplitude of 7 V each, and your power "rails" are +/−10 V, you will run into problems. The same thing can happen in the digital domain.

11.8 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 this type. As you may know, a simple RC circuit where a capacitor is charged via a resistor, has a *time-constant* $\tau = R * C$. This gives us the time it takes to charge 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 we will never get there. Of course this is theory, it doesn't bother an analog or 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 nonexisting. It was very practical that you could reach for your filter-table book on your shelf, find a filter with a fitting cutoff frequency, passband ripple, stopband damping and rolloff, and convert it to digital. This conversion is known as the *bilinear transformation*. It is not simple, 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 the 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. Very often 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, FIR, which we will get back to. Likewise, they also use 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 to name some. They all present an elegant mathematical solution to a filter challenge, such as having the minimum passband ripple or the steepest rolloff. Some of these filters have 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 may oscillate if designed poorly. This is typically due to quantification noise (roundoff errors) when fitting the filter coefficients into a limited number of bits. Oscillations can be really hefty as in a real oscillator, or, more commonly, so-called "limit cycles." This is when, for example, 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.

11.9 Implementing IIR

The general formula for an IIR filter is

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right). \tag{11.7}$$

Typically, the filter is scaled so that a_0 is 1. $y[n]$ is the current output sample, whereas $x[n]$ is the latest input sample fed into the filter. $y[n-1]$ is the previous output and $x[n-1]$ is the previous input sample. Thus, the latest $P+1$ input samples are each multiplied with a coefficient, specific to the “newness” of the sample. Likewise, the Q latest outputs are each multiplied with other coefficients specific to their “newness.” Finally, it is all added together. You don’t need to keep all the intermediate products along the way. It is more efficient to clear an “accumulator” first, adding the products to this one by one hereafter. Hence the focus on the time it takes to do a MAC—multiply-accumulate—operation.

Figure 11.6 is a typical way to visualize this.

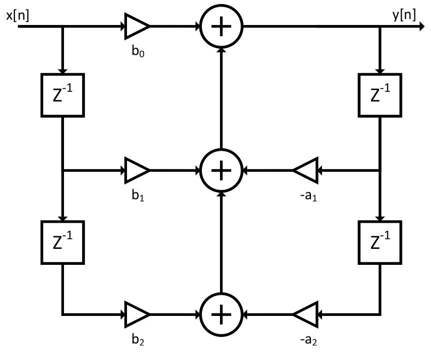


Figure 11.6: Biquad IIR filter implementation.

Z^{-1} is the mathematical way to show a delay of one sample. This means that the IIR filter shown needs to keep the latest three input samples and the previous two output samples, in order to do a calculation of the next output sample. We need to perform 5 MACs per output sample. This is not much, yet it is quite representative—known as a *biquad* filter. This name comes from *bi-quadratic*, which means that the filter formula contains two quadratic functions. In other words, a second-order polynomial in numerator as well as in denominator. This gives us two zeros and two poles.

Looking at this figure, it is clear 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 11.6 downwards. However, as stated earlier, IIR filters can become unstable, oscillating. It has proven more stable to cascade biquads using the output from one biquad as the input to the next. 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 11.4.

Listing 11.4: BiQuad IIR filter code. Created by Tom St. Denis

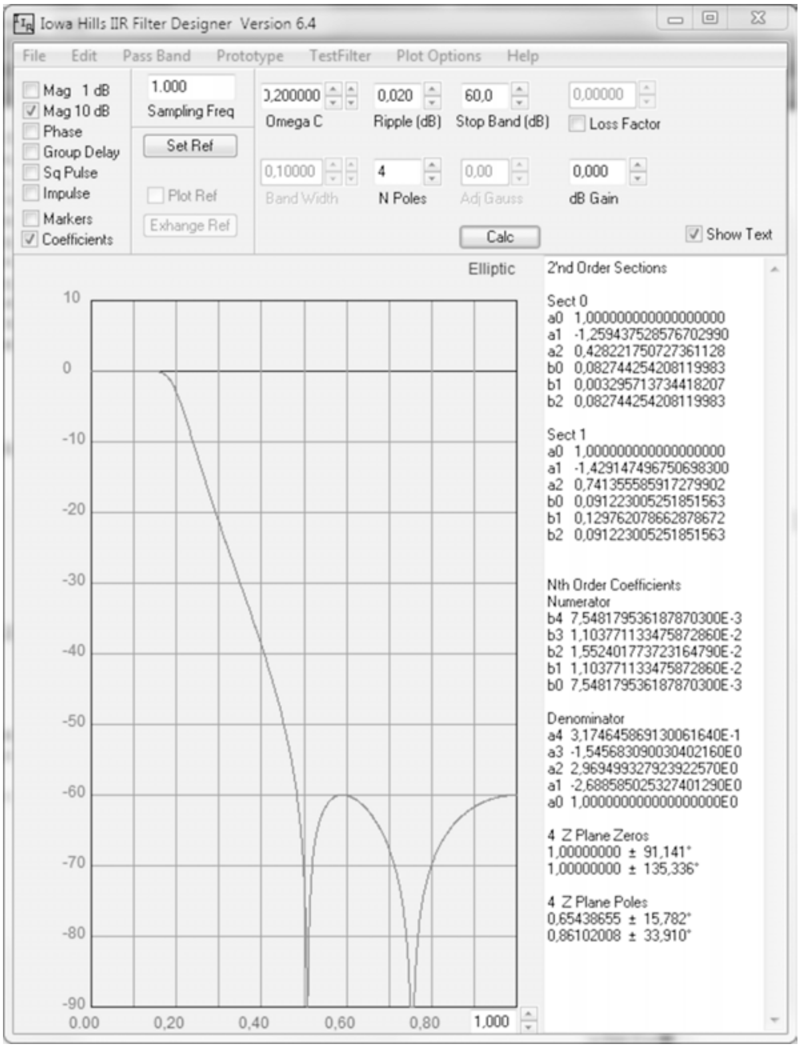
```c
...
typedef struct
{
        float a0, a1, a2, a3, a4;
        float x1, x2, y1, y2;
}
biquad;
...
float BiQuad(float sample, biquad * b)
{
    float result;

    /* compute result */
    result = b->a0 * sample + b->a1 * b->x1 + b->a2 * b->x2 -
             b->a3 * b->y1 - b->a4 * b->y2;

    /* shift x1 to x2, sample to x1 */
    b->x2 = b->x1;
    b->x1 = sample;

    /* shift y1 to y2, result to y1 */
    b->y2 = b->y1;
    b->y1 = result;

    return result;
}
```

There are many filter design programs. Before you go out and buy an expensive solution, which may even require a lot of programming and math understanding, scan some of the available free programs. Figure 11.7 is from a free program by IowaHills.

The example demonstrates many of the standard filter design criteria, with important parameters such as those given in Table 11.1.

In Figure 11.7, the “Coefficients” box is checked, leading to the text at the right side. This contains:

– Coefficients for two biquad sections in the optimal order.
– The n coefficients from the basic IIR formula.
– The four zeros. See Figure 11.8.
– The four poles. See Figure 11.8.



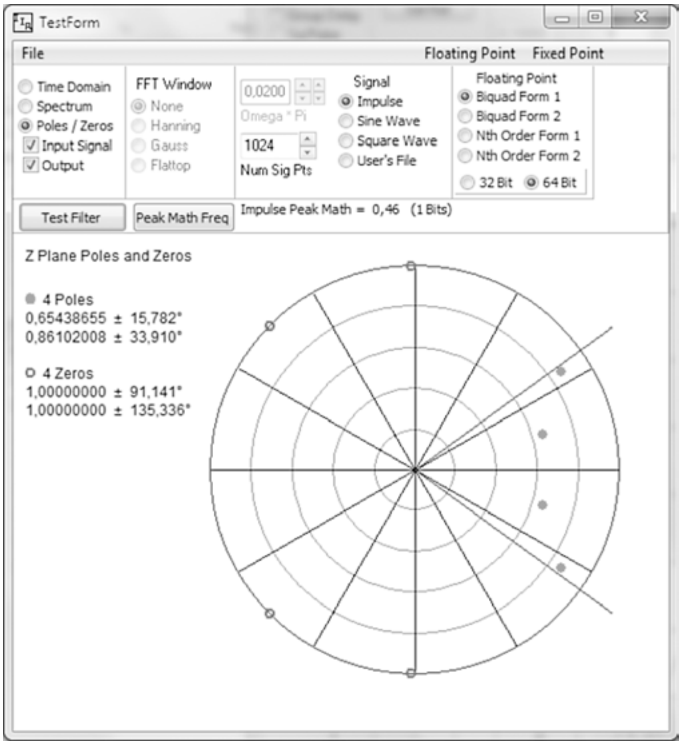
```
☐ Mag 1 dB
☑ Mag 10 dB
☐ Phase
☐ Group Delay
☐ Sq Pulse
☐ Impulse
☐ Markers
☑ Coefficients

1.000      Sampling Freq
0,200000   Omega C
0,020      Ripple (dB)
60,0       Stop Band (dB)
0,00000    ☐ Loss Factor
0,10000    Band Width
4          N Poles
0,00       Adj Gauss
0,000      dB Gain
☑ Show Text
```

```
2'nd Order Sections

Sect 0
a0  1,000000000000000000
a1  -1,259437528576702990
a2  0,428221750727361128
b0  0,082744254208119983
b1  0,003295713734418207
b2  0,082744254208119983

Sect 1
a0  1,000000000000000000
a1  -1,429147496750698300
a2  0,741355585917279902
b0  0,091223005251851563
b1  0,129762078662878672
b2  0,091223005251851563


Nth Order Coefficients
Numerator
b4  7,548179536187870300E-3
b3  1,103771133475872860E-2
b2  1,552401773723164790E-2
b1  1,103771133475872860E-2
b0  7,548179536187870300E-3

Denominator
a4  3,174645869130061640E-1
a3  -1,545683090030402160E0
a2  2,969499327923922570E0
a1  -2,688585025327401290E0
a0  1,000000000000000000E0

4  Z Plane Zeros
1,00000000  ±  91,141°
1,00000000  ±  135,336°

4  Z Plane Poles
0,65438655  ±  15,782°
0,86102008  ±  33,910°
```

Figure 11.7: IIR Filter design by IowaHills.

Table 11.1: Sample filter parameters.

Parameter	Value	Meaning
Basic type	Elliptic	Equiripple. Steep flank
Omega C	3 dB cutoff	Break-frequency
Gain	0 dB	Overall amplification
Sampling frequency	1	Normalized
Ripple	0.02 dB	Passband ripple
Stop band	60 dB	Stopband damping
Poles	4	2 BiQuads



TestForm

File | Floating Point | Fixed Point

Time Domain
Spectrum
[✓ Poles / Zeros]
[✓ Input Signal]
[✓ Output]

FFT Window
[✓ None]
Hanning
Gauss
Flattop

0,0200 Omega * Pi
1024 Num Sig Pts

Signal
[✓ Impulse]
Sine Wave
Square Wave
User's File

Floating Point
[✓ Biquad Form 1]
Biquad Form 2
Nth Order Form 1
Nth Order Form 2
32 Bit [✓ 64 Bit]

Test Filter | Peak Math Freq

Impulse Peak Math = 0,46 (1 Bits)

Z Plane Poles and Zeros

● 4 Poles
0,65438655 ± 15,782°
0,86102008 ± 33,910°

○ 4 Zeros
1,00000000 ± 91,141°
1,00000000 ± 135,336°

Figure 11.8: Pole-zero plot of the elliptic filter.

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.

Biquads are so popular you can find small DSPs or mixed-signal chips including them as ready-made building blocks—just waiting for your coefficients.

11.10 FIR filter basics

In contrast to IIR filters, FIR filters have no analog counterpart. They simply can only be realized digitally. We cannot use an analog filter as a starting point. FIR filters can basically also do lowpass, bandpass, and highpass—and combinations thereof. 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

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i]\right) \tag{11.8}$$

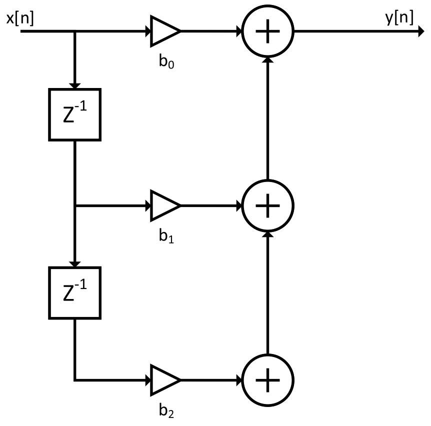


Figure 11.9: Second order FIR filter.

Formula (11.8) 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 11.9), only here there is no right side.

In this case, it is *not* a representative filter. Without the feedback, the filter has to contain many more elements. However, implementing a FIR filter is done by directly extending Figure 11.9 downwards—no cascading is needed.

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 designed 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 line. This again means that the filter delays the whole signal with a time equal to the slope of this line. This is typically not a problem. In most cases, it just means that, for example, 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. This means that the delay must be an integer number of samples.

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 each is 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:

Say that we have a signal sampled at $f_s = 96$ kHz. This is a common sample rate in professional audio. Let's say 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. This will save memory space as well as computer power in the upcoming calculations. We cannot go down to 12 kHz, since this would violate the sampling theorem (7.6 kHz being greater than 12/2 kHz). We might consider something that is not the original sampling frequency divided by 2^n, but this is typically more complicated and invites other problems.

So basically we want to throw away 3 out of every 4 samples. This we cannot 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 requiring 10 multiply accumulates (MACs) per sample for two biquad sections, at the high sample rate, we are effectively spending $4 * 10 = 40$ MACs 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 needed. This means that the FIR filter is not competing with 10 MACs per sample, but with 40 MACs per sample. So if we can do the FIR in 39 MACs it's actually faster. After each calculation, we move not one sample, but four. Then we do the next calculation.

Add to this, that the FIR filter is truly "vectorized." Once you have set it up correctly, many DSPs are very fast. If the DSP has hardware 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 lowpass filter, it will be centered around DC (remember the frequency axis with a negative side). If it's a highpass filter, it will be centered around $f_s/2$. A band-pass filter is somewhere in between.

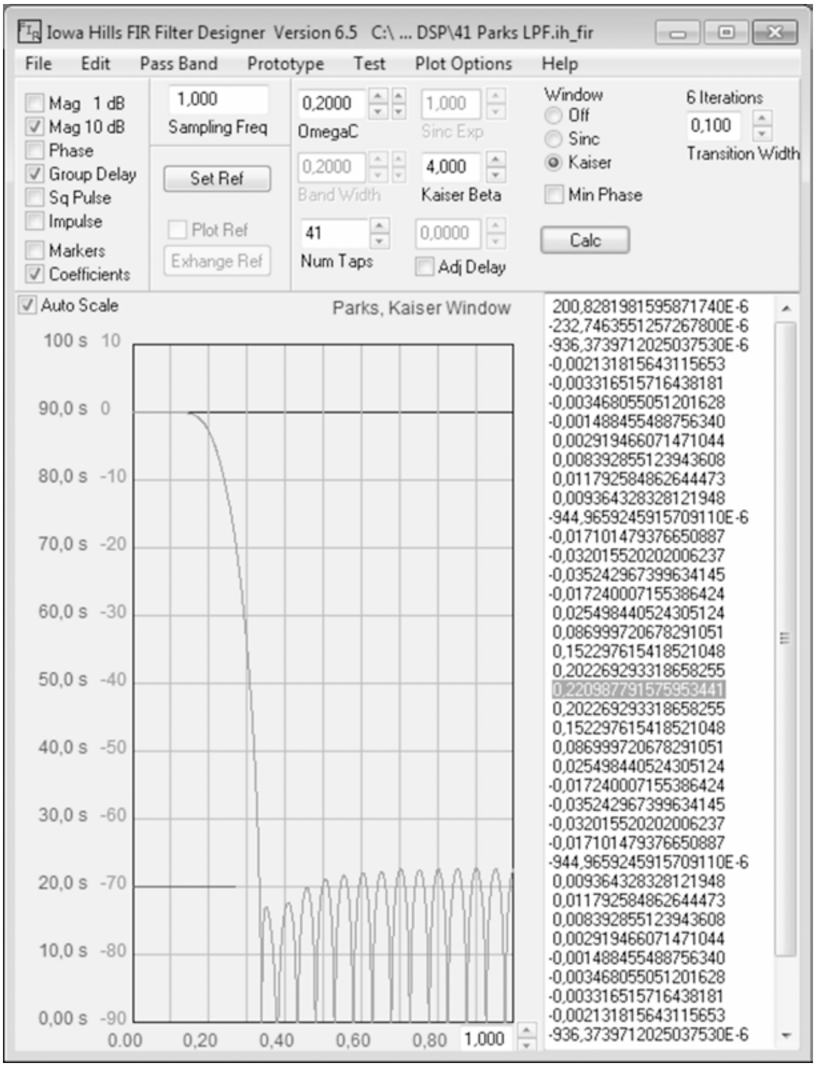
One way to implement an FIR filter is thus to do an FFT (fast Fourier transform), transforming the signal to the frequency domain, multiply it sample-by-sample with the square, and then perform an *iFFT (inverse fast Fourier transform)* to get back into the time domain. This can actually be done using short time "snippets," but it is not simple[5] 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 Figure 11.2, we saw a square pulse in the time domain, that became a $\sin(x)/x$ function in the frequency domain. If instead we take a square from the frequency domain located around DC—in other words a lowpass filter—to the time domain, it actually becomes a $\sin(x)/x$. Look at Figure 11.2, swap frequency and time domain, and you see the lowpass FIR filter as a $\sin(x)/x$ in the time domain. As we are working with discrete signals, the $\sin(x)/x$ is sampled. These samples are the filter coefficients. In digital signal processing, we call each of these coefficients from the FIR filter a *tap*.[6]

Figure 11.10 shows the FIR designer from IowaHills. This time there are no cascaded sections or poles and zeros, just a lot of coefficients. The center coefficient is marked with a cursor. Note that the other coefficients are symmetric around this. This is what gives us the linear phase.

5 This can be done using a concept called "overlap-add."

6 This name might stem from implementations using a "tapped delay line."



[ ] Mag 1 dB
[✓] Mag 10 dB
[ ] Phase
[✓] Group Delay
[ ] Sq Pulse
[ ] Impulse
[ ] Markers
[✓] Coefficients
[✓] Auto Scale

Sampling Freq: 1,000
OmegaC: 0,2000
Sinc Exp: 1,000
Band Width: 0,2000
Kaiser Beta: 4,000
Num Taps: 41
[ ] Adj Delay: 0,0000
Window: Off [✓ Kaiser] Sinc
[ ] Min Phase
6 Iterations
Transition Width: 0,100

```
 200,8281981595871740E-6
-232,7463551257267800E-6
-936,3739712025037530E-6
-0,00213181564311 5653
-0,00331651571643 8181
-0,00346805505120 1628
-0,00148845548875 6340
 0,00291946607147 1044
 0,00839285512394 3608
 0,01179258486264 4473
 0,00936432832812 1948
-944,9659245915709110E-6
-0,01710147937665 0887
-0,03201552020200 6237
-0,03524296739963 4145
-0,01724000715538 6424
 0,02549844052430 5124
 0,08699972067829 1051
 0,15229761541852 1048
 0,20226929331865 8255
 0,22098779157595 3441
 0,20226929331865 8255
 0,15229761541852 1048
 0,08699972067829 1051
 0,02549844052430 5124
-0,01724000715538 6424
-0,03524296739963 4145
-0,03201552020200 6237
-0,01710147937665 0887
-944,9659245915709110E-6
 0,00936432832812 1948
 0,01179258486264 4473
 0,00839285512394 3608
 0,00291946607147 1044
-0,00148845548875 6340
-0,00346805505120 1628
-0,00331651571643 8181
-0,00213181564311 5653
-936,3739712025037530E-6
```

Figure 11.10: IowaHills FIR Filter Designer.

11.11 Implementing FIR

Convolution is the process we apply to our data and the taps/coefficients of the FIR filter. This is the process Figure 11.9 shows in an abstract way. We “park” the coefficients next to the samples, multiply each sample with a filter coefficient, accumulate the products, and then we have a single output sample next to the center of the filter. There are two things 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 pointing ahead of the current sample—so to speak into the future? That is impossible. Yes, that's what the mathematicians call a *noncausal* filter. We need to work on a delayed signal, so that we have all input samples ready. This is another way to explain the filter delay. It is not difficult to realize that we need to delay the signal $((N-1)/2) * \triangle T$, 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, for example, 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 MACs either. Thus we need to "chop off the skirts" of the filter somewhere, hence the name "finite impulse response." To do this with minimal 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.

Just like with the IIR filter, using an existing filter or a filter-design program, for example, like Figure 11.10 is recommended. Excel is however not completely useless. Figure 11.11 shows the filter coefficients from the design program in an Excel graph.

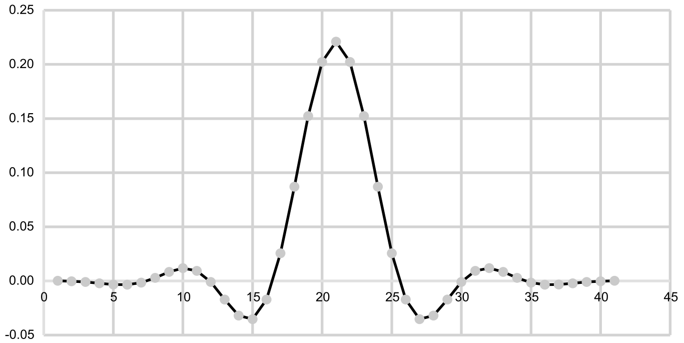


Figure 11.11: Coefficients from FIR design program.

It can be useful to plot the filter coefficients. A fellow student and I designed our first FIR filter during our masters project. It sounded awful. After many experiments, we plotted the coefficients. It turned out that the filter-program's print-function, which had neatly written the coefficients in a number of lines, for some reason never printed *a negative sign for the first number after a newline. This was easy to spot once the coefficients were plotted.*

Figure 11.12 shows a buffer, implementing a circular storage with the same number of samples as there are filter coefficients.

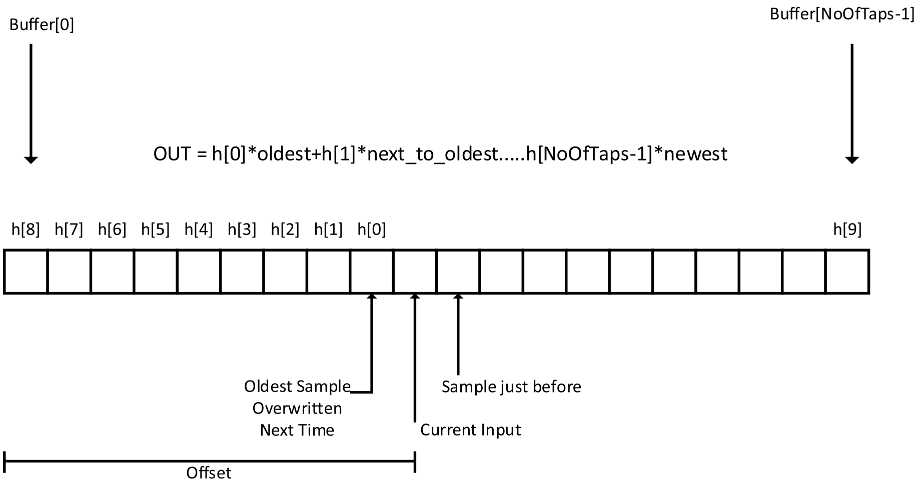

Buffer[0]
Buffer[NoOfTaps-1]
OUT = h[0]*oldest+h[1]*next_to_oldest.....h[NoOfTaps-1]*newest
h[8] h[7] h[6] h[5] h[4] h[3] h[2] h[1] h[0]
h[9]
Oldest Sample Overwritten Next Time
Sample just before
Current Input
Offset

Figure 11.12: Buffer usage for MIT's implementation.

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 latest. 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 11.12 is created to demonstrate the very efficient piece of C code made at MIT that implements a FIR filter with my comments inserted.

Listing 11.5: MIT FIR filter code

```
double Fir1::_Filter(double input)
{
   double *coeff     = coefficients;    // Point to index 0
   double *coeff_end = coefficients+taps; // Point to after array

   double *buf_val   = buffer +offset; // buffer is a class var

   *buf_val = input;                    // 'current' pointer
   double output = 0;                   // ready to accumulate

   while(buf_val >= buffer)            // Until left edge
      output += *buf_val-- * *coeff++; // MultAcc & move pointers

   if (++offset >= taps)               // At right edge ?
      offset = 0;

   return output;
}
```

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.

11.12 Dynamic range versus precision

FPUs (floating point units) are gaining ground, and they make coding easier by giving you a large *dynamic range*. You do not have to worry about overflow during calculations. With such a device you can write your embedded code in plain C, using **`float`** and **`double`** as you may be used to when programming on a PC. However, the smaller systems often do not have an FPU, 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 perfectionist may actually prefer the integer arithmetic. Say 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. As stated, this gives you a much larger dynamic range than a 32-bit integer does, 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 neatly, without getting overflows. This again may enable you to pick a smaller CPU and use less power and/or money.

11.13 Integers

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

$$\text{Value} = \sum_{n=0}^{p} a_n * \text{base}^n \tag{11.9}$$

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

The generic case: $a_2 * \text{base}^2 + a_1 * \text{base}^1 + a_0 * \text{base}^0$

$$\begin{aligned}
\text{Base 10, "256"} &: 2*10^2+5*10^1+6*10^0 &&= 2*100+5*10+6 &&= 256\\
\text{Base 2, "101"} &: 1*2^2+0*2^1+1*2^0 &&= 2*2+0+1 &&= 5\\
\text{Base 16, "}a2c\text{"} &: 10*16^2+2*16^1+12*16^0 &&= 10*256+2*16+12 &&= 2604\\
\text{Base 8, "452"} &: 4*8^2+5*8^1+2*8^0 &&= 256+40+2 &&= 298
\end{aligned}$$

The number systems are known as “decimal,” “binary,” “hexadecimal,” and “octal.” With base 16 hexadecimal numbers, you continue counting after 9: a, b, c, d, e, f, and then 0x10 which is 16 in the decimal system.

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 for 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 CPUs, it is common to use **`char`** for small numbers. There are also **`signed char`** and **`unsigned char`** here for exactly this purpose. But for some reason there is no default.

Make it a rule to always use the **`signed`** version when processing signals, for example, 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. 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:

Listing 11.6: Standard C data-sizes

```
sizeof(short) <= sizeof(int)  // Guaranteed - but no more
sizeof(int)   <= sizeof(long) // Guaranteed - but no more
```

This means that if the normal **`int`** is 16 bits, there's probably a good chance that a **`short int`** is 8 bits, 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 we use for unsigned integers. We want to be able to add -x to x, getting 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 examples shown in Table 11.2.

Table 11.2: Examples of 16-bit signed integers.

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

11.14 Fixed-point arithmetic

It is customary to work with a *virtual binary point* when you are using integer arithmetic to calculate noninteger numbers. This is typically placed just right of the sign bit. An example of a positive 8-bit number, with the virtual point written is

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

Hexadecimally, the above 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

$$0\text{x}5\text{a} = 5 * 16 + 10 = 90 \quad \text{and} \quad 90/2^7 = 0.703125$$

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[7] to a full integer and then converted to hexadecimal (or binary if you prefer). Traditionally, there have been several different ways to have a sign bit as MSB (most significant bit):

7 Especially with IIR filters, we may need "controlled rounding"—yet a reason for a filter design program.

- 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 CPUs, 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 about digital signal processing, 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, for example, the positive number 6 in a 4-bit representation. 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 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, 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 (affecting 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. In such cases, there is a *wrap* . The overflow flag can be checked in assembler, but not in C. In some systems, you can set this to generate an exception (aka a trap). In many DSPs, 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. See next bullet.
- If you are working with two's complement integer arithmetic, you may have designed a filter in a way that causes no overflow in the final sum of many products, but there may temporarily be an overflow along the way. Due to the way a two's complement number wraps you are saved—*unless* saturation mode was on. This is one of the reasons why integer DSP calculations are interesting; getting the maximum out of the dynamic range is an art form. But time consuming.

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

11.15 Q-notation and multiplication

If we multiply two “normal” signed 16-bit integers, we end up with a 32-bit integer. We have twice as many bits. There used to be one sign bit and 15 integer bits in each. 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. If we end up with two sign bits, our result is precisely half of what it should be. It becomes more complicated: the virtual binary point does not have to be right after the sign, you may want to set it so that you can have filter coefficients larger than one. In order to manage this, the “Q” notation was introduced. A 16-bit word using normal two’s complement fractions 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.[8]

If 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 to have Q31 and then keep the upper 16 bits as the Q15 result.

This results in truncation. Rounding might be preferred. You normally round decimal numbers to integers by adding 0.5 before you truncate. Here, we add 1 to the MSB of the bits that will be truncated. In the above case, we add 0x8000 to the Q31 number after the left shift. Alternatively, we may add 0x4000 to the Q30 number before any shifting. This gives us the choice afterwards between shifting one position left, using the upper 16 bits, or shifting 15 positions right, using the lower 16 bits.

A real DSP has the often mentioned multiply-accumulate operation with associated accumulator. If it’s a 16-bit DSP, this accumulator will be at least 32 bits wide. 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 wide accumulator, saving us the many round-off errors. Finally, when the result is read as the upper 16 bits, the fixed-point DSP will typically silently shift the result one bit left for you, assuming Q15 arithmetic. This may include automatic rounding as described, or even more advanced rounding.

There are variants where a so-called “barrel shifter” can be programmed to perform a specific number of shifts—depending on the number of digits before and after the decimal point in the two operands, as well as the extra bits the accumulator has. If you are using a standard, non-DSP, CPU with Q arithmetic, it is up to you to remember the last shift(s).

8 Another school of thought includes the sign bit. Thus Q15 is here known as Q1.15, and Q2.13 becomes Q3.13.

11.16 Division

Floating Point units also support division. 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. You should 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 performed 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.

11.17 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. Instead you would terribly much like to express, for example, 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.

11.18 Further reading

- Oppenheim and Schafer: *Discrete-Time Signal Processing*
  This is the authors 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 have 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 signal processing. The biquad filter by Tom St. Denis 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.
- Jan Gullberg: *Mathematics: From the Birth of Numbers*
  This is really not a book needed for DSP. But if you are a nerd with math and numbers you will probably love this book.
- dspillustrations.com
  This is a very competent site with great articles on DSP. “Approximating the Fourier transform with DFT” is very relevant.

12 Statistical process control

12.1 Introduction

As described in the Introduction, “Industry 4.0” has two main pillars. One pillar is the connectivity of manufacturing robots and machines. The other pillar is a higher degree of autonomy.

It makes sense that with billions of devices connected to the internet, we cannot expect them all to send raw data somewhere else for decision making. If the devices utilize their smarter CPUs, they can take decisions locally. This is faster and saves energy and bandwidth, and thereby costs. In many ways, this is also a more secure solution. Chapter 11 shows how the amount of data can be reduced by filtering sampled signals. In this chapter, we see how we can avoid a lot of communication and latency in manufacturing, by taking decisions locally. This means that the embedded systems programmer needs to know something about quality control and the applied statistics in this field. This chapter introduces the main ideas and provides links for further reading.

Walter Shewhart introduced statistical quality control and *control charts* almost one hundred years ago at Bell Laboratories. Many of the products used in the telephone networks were dug under ground, making it costly to repair them. Shewhart knew how variations in the production contributed to failures, but he also realized that minor random changes, within the acceptable limits, might lead operators to adjust machinery to a worse condition than it was before. There was a need for a way to distinguish the acceptable random variations from various causes of errors.

Shewhart introduced two terms for causes of variations in the output of a production system:

– *Common Causes*
  These are the random variations that exist in any production system as well as in nature. No two objects are completely identical. Once these common causes are measured and accepted, they should not lead to adjustments of the process.
– *Assignable Causes*
  These are the causes of variations that can be traced—sometimes through very hard work—to something not under control. This could be a window opened when measuring at “constant temperature,” or dirt brought into a clean-room in electronics production, or a new batch of components from a subsupplier behaving differently than the previous, etc.

Shewhart also separated the purpose of quality tests in two:

– *Capability*
  How well do we meet the specs? This is what people outside the field associate with quality control. If you have little control of the variations in your production

https://doi.org/10.1515/9781547401024-012

system, you will either have to give conservative specs, giving some customers more than what they pay for, or you will not have conservative specs, and then some customers will get less than what they paid for. Neither scenario is good.

– *Stability*
  Is the manufacturing process stable? Here, the purpose of the test is to detect whether, for example, wear and tear is degrading the production equipment. This will eventually lead to products out of spec. It is a well-known fact that the cost of fixing a problem grows, almost exponentially, as the product moves through the delivery chain. Fixing the machinery before it starts producing defective products is a lot cheaper than throwing away these bad products. The most expensive fix occurs when the defect is discovered by the customer. This induces logistic expenses (sending products to repair, etc.), but also damages the brand.

For most of this chapter, we are dealing with variables that can be measured such as weight, length, 3 dB bandwidth, volume, etc. These are normally given as real values (as opposed to integers). According to the *central limit theorem*, all statistical distributions can be treated as if they are *normally distributed*, when the number of samples is “large,” which is often interpreted to be at least 30 observations. The normal distribution is also known as a Gaussian distribution.

Figure 12.1 shows a special case of the normal distribution.

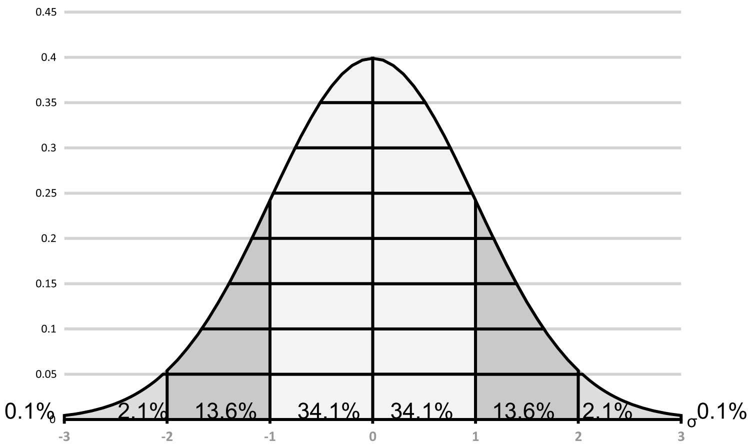


Figure 12.1: Standard Normal distribution.

The shape is as always the famous “bell shape,” however, the values on the x-axis are chosen so that the mean is zero and the standard deviation is one—the *standard normal distribution*. As in all probability distributions, the area under the curve equals 1. Any normal distribution can fairly easily be transformed into this, which was extremely practical in the days before the computer. Since the shape is symmetrical, it was only necessary with tables on one side, and from this all values could be transformed back to the current case. This is still a nice feature as it makes it easier for humans to relate to.

As shown in Figure 12.1, 68.2 % (34.1+34.1) of all outcomes fall within one standard deviation (plus or minus) from the mean value known as μ. The standard deviation is noted with the Greek letter σ known as *sigma*. Likewise, 95.5 % fall within $+/-2\sigma$ and 99.7 % fall within $+/-3\sigma$. Only 0.27 % fall outside the $+/-3\sigma$ area shown in the figure.

Another reason why the normal distribution is so attractive is that given μ and σ we can calculate the rest.

The available theory on *SPC* (statistical process control) generally deals a lot with "charts." This is a natural consequence of the existing workflow where measured raw-data is sent to a PC, typically for post-processing, where the SPC program is run on a graphical user interface. This allows the person responsible to follow the stability and capability of the system. The PC will flag "suspicious" behavior and perhaps take relevant actions. If this is instead performed in an embedded system, the charts may not actually be displayed, but the algorithms normally shown on the screen, will still make sense. Only now there is no direct operator watching the charts. We will stick to the "chart" term, treating it in the more abstract way.

Should you want to know more about statistical charts in general (not control charts), Alberto Cairo's book, *The Truthful Art,* is highly recommended. It is colorful and entertaining, but also very educational. It includes a general explanation of the basic terms in statistics.

An important parameter in SPC is *sample size* or *subgroup size*. It is very often assumed that a "sample" of, for example, 10 objects is taken from the assembly line every hour or day. Thus a "sample" in this context is 10 objects. This sample can be categorized by various statistical parameters, such as average, range and estimated variance of, for example, length or weight, etc. If you come from the world of digital signal processing, it can be confusing that a sample is more than one measurement and can have a variance. For this reason, we will generally use the term "subgroup." In the modern world of IoT, you might ask "why not measure on all objects?" Indeed this is probably going to be increasingly relevant in the future, but there are still a huge amount of scenarios where subgroups are preferred:

- The test is destructive. Naturally, we cannot test all objects if we have to break them. Some harsh tests will not necessarily break the *DUT* (device under test) but they may weaken it, so that a future harsh incident will break it. If it is required that all mobile phones from an assembly line can survive a 3 feet fall, this doesn't necessarily mean that you want to drop all phones 3 feet (not to mention that "survivors" may still carry scratches).
- The test is costly compared to the value of the product. A company, making modern high-end TVs, may test key parameters on each TV, whereas a company making screws will not necessarily want to measure (and store) length, weight, and thread parameters on all screws.
- The test takes a long time to run. This adds costs in itself.

In many ways, this boils down to the actual purpose of the test. If we stick to the purpose of monitoring the manufacturing process, instead of verifying each product against specs, subgroups make a lot of sense.

12.2 Important terms

One of the most intimidating features of statistics is all the symbols that mean almost the same and, therefore, in some situations can be treated alike, whereas in other situations they can't. Table 12.1 shows some symbols, very relevant to the subject of quality control.

Table 12.1: Statistical symbols.

Term	Explanation
μ	"mu." Theoretical mean value of a population
σ	"sigma." Theoretical standard deviation of a population
$\hat{\mu}$	Estimated mean value of a population
$\hat{\sigma}$	Estimated standard deviation of a population
$\bar{X}$	Average of a group of objects
$\bar{\bar{X}}$	Average of averages of several groups of objects
R	Range of a group of objects
$\bar{R}$	Average of ranges of several groups of objects
s	Standard deviation of a sample (subgroup)
T	Target – often the desired mean value

Estimating the mean value by averaging is not a surprise. The notation $\bar{X}$ is pronounced "X-Bar" and this is a term used a lot in control charts. Estimating σ is, however, not so easy. The *range* is interesting because it often is used to estimate σ. The range of a subgroup is simply the difference between the largest and smallest value measured in the subgroup. When, for example, measuring weight, the range of a subgroup is the weight of the heaviest object minus the weight of the lightest in the subgroup.

12.3 Control charts

The normal distribution can be used to show variations between individual products, for example, in histograms. However, histograms do not show changes over time. For this reason, Shewhart invented the control chart. The chosen *statistical parameter* (not measured value) is plotted as a function of time or test number. This allows us to spot trends in time.

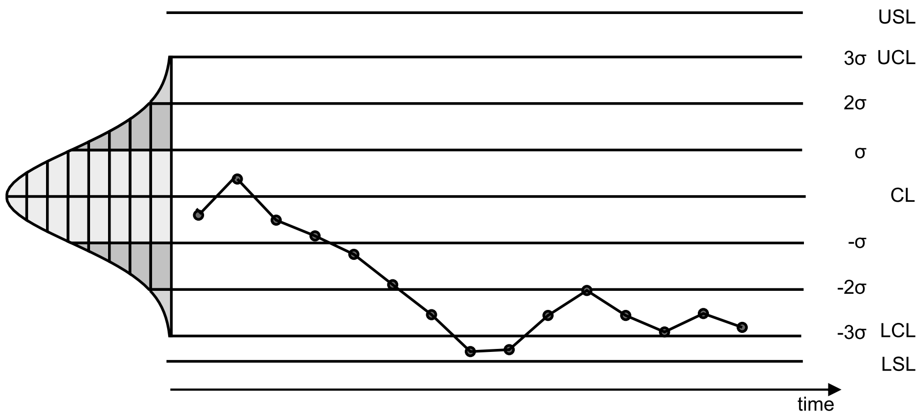


Figure 12.2: Control chart.

Figure 12.2 abstractly shows how the control chart has the normal distribution vertically placed left to the Y-axis, extending the sigma lines into the graph area. This is the generic concept.

Plotting *measured* values against upper and lower *specification* limits (USL and LSL) was obviously not unseen when the control chart was invented. The specification limits are simply the values which the given product must lie between, in order to be approved. However, this is *not* what the control chart generally plots. Instead, it plots statistical parameters. Furthermore, it does not plot against specification limits, but against control limits. Table 12.2 shows the values used in control charts.

Table 12.2: Control chart definitions.

Short	Full Name	Description
UCL	Upper control limit	$CL + 3 * \hat{\sigma}$
CL	Center line	Estimated mean value
LCL	Lower control limit	$CL - 3 * \hat{\sigma}$
USL	Upper spec limit	Max OK value
LSL	Lower spec limit	Min OK value

The upper and lower control limits are always three sigmas[1] away from the *centerline*, totally six sigmas spaced. This can be quite confusing if you have heard about *Six Sigma*, as it is almost totally unrelated. The upper and lower *control* limits are specific to the process control and the whole theory in control charts, whereas the upper and lower *spec* limits on the other hand, are specific to the product in question. If the two

1 There are alternatives, but three is predominant.

spec limits are placed on top of the respective control limits, we can expect 99.7 % of all products to be OK and only 0.27 % rejected, as shown in Figure 12.1 in the Introduction. However, there will be 2700 defective items if a million items are produced.

Motorola once decided to align the upper and lower spec limits with the +/−6σ lines—12 sigmas spaced. This is the basis for the "Six Sigma" name, producing only 0.002 defect items out of a million (also shown later in Table 12.9). This does *not* mean that the specs on the products are fantastic. It means that the manufacturing process is very well controlled—the production machines may be fantastic. Ironically, very few companies doing "Six Sigma" are actually abiding by this pretty tough rule. Instead, they are focusing on some of the more people-oriented processes in *DMAIC* (define-measure-analyze-improve-control).

Table 12.3 shows a set of *alarm rules*. They were originally implemented by *WECO* (Western Electric Company), and are now given by NIST as an example.

Table 12.3: WECO alarm rules from NIST.

Rule	Alarm Description
1	Any point above +3σ
2	2 out of the last 3 points above +2σ
3	4 out of the last 5 points above +1σ
4	8 consecutive points above the Center Line
	—— Center Line ——
5	8 consecutive points below the Center Line
6	4 out of the last 5 points below −1σ
7	2 out of the last 3 points below −2σ
8	Any point below −3σ

These rules are easily implemented in an embedded system, once the various limits are given. The actions taken by the embedded system could be anything from shutting down the machines, to sending a mail, or text to the person in charge.

Since 0.27 % of all measured values will fall outside the 3 sigma limits, we can expect a false alarm from rule 1, once per 370 points, if measuring on individuals. Some of the other rules are equally capable of generating false alarms. For this reason, we might treat these as merely warnings, and apply more methods.

12.4 Finding the control limits

When creating control charts, there are typically two phases: phase 1 for the original estimation of the limits, and phase 2 for normal production—using the limits. An example could be a day running with full inspections of all products. "Outliers" due to operator rookie-mistakes are removed before the control limits are calculated, based on the rest of the measurements from the full day in one big group. The following formulas show how to estimate the mean and sigma used when calculating the control limits.

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{12.1}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2} \tag{12.2}$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2} \tag{12.3}$$

$$\hat{\sigma} = s \quad \text{for } n > 30 \tag{12.4}$$

$$\hat{\sigma} = s/c_4 \quad \text{for } n <= 30 \tag{12.5}$$

$$\hat{\sigma} = \bar{R}/d_2 \quad \text{for } n <= 10 \tag{12.6}$$

Formula (12.1) is the simple calculation of an average. This works for any population or subgroup. Formula (12.2) is the true standard deviation for the full population. Formula (12.3) is the standard deviation for a subgroup (aka sample). As shown, it is technically called s and we divide by $n - 1$ – not n. This is only a good estimate for σ when there are more than 30 individuals in the subgroup—something easily forgotten. When this is *not* the case, we should divide by c_4, which can be found in Table 12.4.

When the sample size is 10 or below, we *may* instead use Formula (12.6), which is based on simple ranges (largest value – lowest value) divided by d_2, which is found in the same table.

The devil lies in the detail. In the “s” part of the X-Bar-s chart, the centerline is $\hat{\sigma}$, but then the control limits are found by adding +/–3 standard deviations of an esti-

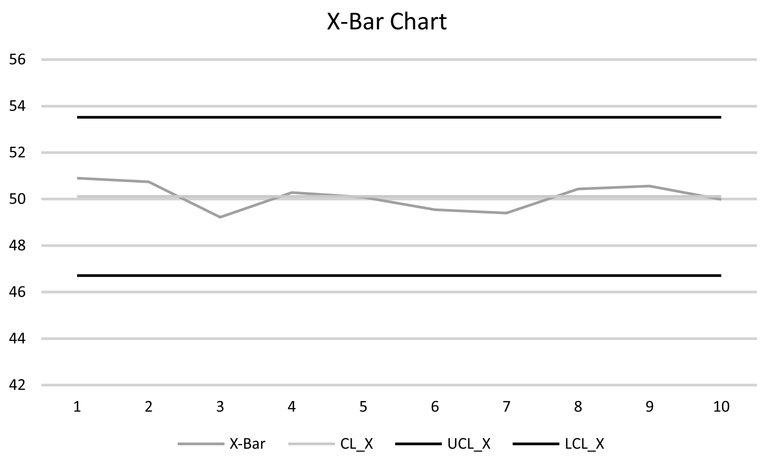
Table 12.4: Control chart constants for X-Bar, R- and s-charts.

Size	A_2	d_2	D_3	D_4	A_3	c_4	B_3	B_4
2	1.880	1.128	–	3.267	2.659	0.7979	–	3.267
3	1.023	1.693	–	2.575	1.954	0.8862	–	2.568
4	0.729	2.059	–	2.282	1.628	0.9213	–	2.266
5	0.577	2.326	–	2.114	1.427	0.9400	–	2.089
6	0.483	2.534	–	2.004	1.287	0.9515	0.030	1.970
7	0.419	2.704	0.076	1.924	1.182	0.9594	0.118	1.882
8	0.373	2.847	0.136	1.864	1.099	0.9650	0.185	1.815
9	0.337	2.970	0.184	1.816	1.032	0.9693	0.239	1.761
10	0.308	3.078	0.223	1.777	0.975	0.9727	0.284	1.716
15	0.223	3.472	0.347	1.653	0.789	0.9823	0.428	1.572
20	0.180	3.735	0.415	1.585	0.680	0.9869	0.510	1.490

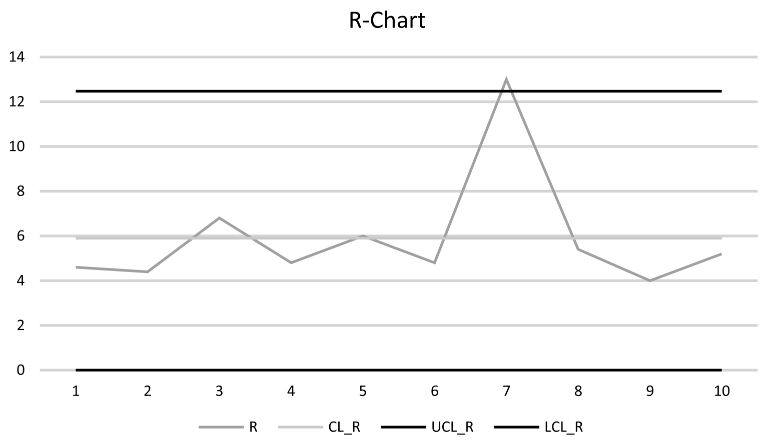
mation for sigma for the original distribution. In other words, standard deviations of standard deviations. This can be hard to relate to.

Calculating the above again involves *c*4 and calculating *this* involves *noninteger* factorials or rather the "Gamma" function, which is found in Excel. The hard-to-calculate constants are *d*2 and *d*3 (not shown in the formulas). These can be found by numeric simulations. They are, however, also given as simple function calls in "R"—the statistical program.

Since the whole charade is for numbers below 30, it is no wonder they are put into tables. The table constants are created to include as much of the real formulas as pos-



(a) X-Bar chart - monitoring mean of subgroups



(b) R chart - monitoring variations in subgroups

Figure 12.3: X-R control chart on ten subgroups of five objects.

sible. This explains why there are so many constants and why the resulting formulas end up rather simple as we shall see.

Due to all the complexities with the s-based calculations, it is no wonder simple ranges were preferred before the computer age. Today some might say that Formula (12.5) should be preferred, also for subgroup-sizes below 10, as it reflects on all the observations in the subgroup not just two of them (the min and max value). On the other hand, estimated standard deviations are still difficult to understand. Thus the X-Bar-R chart remains very popular.

The sharp distinction between phase 1 and phase 2 does not always exist. A PC program can take a historical set of data and calculate limits from this. From here, it continues to investigate the *same* dataset from these limits. You might think that if limits are based on a given dataset, there will be no data outside the limits but that is not correct. The X-R charts shown in Figure 12.3 are created as a small Excel exercise and clearly shows a violation in the R-chart on the seventh subgroup/sample.

The theory for the use of subgroups, and how to estimate σ, may be applied in both phases 1 and 2. Nevertheless, you can imagine that in phase 1, the quality engineer will be more prone to see the whole test production as one big group. It is also quite likely that phase 1 calculations are created in a PC SPC program, and that the results from this are control limits to be used in embedded software in phase 2.

12.5 Subgroups

In the introduction to this chapter, a number of scenarios were given that all lead to the use of subgroups. These often consist of only a few samples, which is why we need to introduce the calculated correction factors in Table 12.4. It is possible to calculate these “from scratch”; see, for example, the *Engineering Statistics Handbook* by NIST,[2] which provides some of the formulas. Table 12.4 was created using Excel and “R” as mentioned earlier.

Performing the calculations in standard C, using Table 12.4, is definitely doable, and may be preferred instead of an SPC library with a lot of code for drawing charts never used. Integrating such a program into a small embedded system might not be easy anyway.

It is customary to work with fixed-size subgroups containing something between 3 and 25 objects. Deciding the subgroup size and how to choose these samples is probably one of the most difficult parts of the whole process. Here are some rules of thumb:

- All observations within a subgroup must be from one single, stable process “stream.” In other words from the same assembly line, or machine, with the same operator, during the same shift, etc.

2 The US National Institute of Standards and Technology.

- If the subgroup is sampled from a larger set, the samples should not be random from this set, but consecutively sampled. In a scenario with 5 samples out of each 100 on an assembly line, it could, for example, be the last 5 in each set of 100.
- Observations must not influence each other. If, for example, we are evaluating a network by measuring roundtrip-time, it would *not* be a good idea to keep consecutive samples as described above, since we know how congestion leads TCP to "lower the speed." There is a high likelihood that two consecutive packets will experience similar delays. In this case, a sample every minute would make more sense. To avoid phenomena occurring exactly each minute, we could choose an interval of 59 s (a nice nearby prime).

12.6 Case: insulation plates

One of the most popular sets of control charts is the X-bar-R plot. Figure 12.3 (shown earlier) is an example of this, and is the output from this exercise. A very similar alternative is the X-bar-s plot, in which the X-bar part is almost the same.

The three plots are:

- *The X-Bar Chart*
  Here, we follow the development of the mean value in subgroups over time. When used together with the R-Chart, the limits for the X-Bar Chart is calculated from the sigmas estimated from R, whereas when used together with the s-Chart, we are using sigmas estimated from s. Since the mean of a subgroup is X-Bar (and this is what we plot) it follows that the centerline in this plot is a mean of means—X-Bar-Bar.
- *The R-Chart*
  Here, we follow the development in the roughly estimated standard deviation in the same subgroups over time. We use Formula (12.6), based on ranges that can be used for small subgroups (up to 10 individuals).
- *The s-Chart*
  Here, we also follow the development in the estimated standard deviation in the same subgroups over time. We use Formula (12.5), based on s that can be used for all subgroups.

The imaginary example follows the production of polystyrene insulation plates over 10 days. Each day a sample/subgroup of 5 plates is picked, allowing us to use range as the basis for estimation of sigma. The nominal thickness is 50 mm, however, the unit is left out in the following. The pseudo data is found in Table 12.5.

The formulas for the limits are shown in Table 12.6. Note that these simple formulas now use constants named A_2, D_3, and D_4. These are found back in Table 12.4. Together with other constants in this table, they are (as stated earlier) needed when

Table 12.5: Ten days of testing—five samples each day.

Day	1	2	3	4	5
1	53.4	51.2	48.8	48.9	52.2
2	51.1	52.3	51.2	47.9	51.2
3	48.1	52.4	47.7	45.6	52.3
4	52.2	48.9	50.3	47.6	52.4
5	48.4	52.1	47.5	53.5	48.9
6	50.4	49.6	48.3	52.1	47.3
7	48.4	52.1	47.1	43.2	56.2
8	49.9	52.1	52.7	47.3	50.2
9	48.6	51.2	52.6	49.2	51.2
10	48.5	51.6	50.2	52.4	47.2

Table 12.6: X-Bar and R control lines.

Parameter	Formula	Explanation
$\text{UCL}_{\bar{X}}$	$\bar{\bar{X}} + A_2 * \bar{R}$	Upper control means
$\text{CL}_{\bar{X}}$	$\bar{\bar{X}}$	Centerline means
$\text{LCL}_{\bar{X}}$	$\bar{\bar{X}} - A_2 * \bar{R}$	Lower control means
UCL_R	$D_4 * \bar{R}$	Upper control variations
CL_R	$\bar{R}$	Centerline variations
LCL_R	$D_3 * \bar{R}$	Lower control variations

working with formulas involving $\hat{\sigma}$ for populations smaller than 30. In this case, we need these constants from the row with Size = 5, as this is our subgroup size.

The similar formulas using s instead of R are shown in Table 12.7.

Table 12.8 shows the test-data again and now also the calculated R and X-Bar for each subgroup, as well as the means of these, giving us R-Bar and X-Bar-Bar. The right-most column is s, calculated as in Formula (12.3). This we will use later for an X-Bar-s chart.

Table 12.7: X-Bar and s control lines.

Parameter	Formula	Explanation
$\text{UCL}_{\bar{X}}$	$\bar{\bar{X}} + A_3 * \bar{s}$	Upper control means
$\text{CL}_{\bar{X}}$	$\bar{\bar{X}}$	Centerline means
$\text{LCL}_{\bar{X}}$	$\bar{\bar{X}} - A_3 * \bar{s}$	Lower control means
UCL_s	$B_4 * \bar{s}$	Upper control variations
CL_s	$\bar{s}$	Centerline variations
LCL_s	$B_3 * \bar{s}$	Lower control variations

Table 12.8: Subgroup data with means and ranges.

Day	1	2	3	4	5	R	X-Bar	s
1	53.4	51.2	48.8	48.9	52.2	4.6	50.90	2.03
2	51.1	52.3	51.2	47.9	51.2	4.4	50.74	1.66
3	48.1	52.4	47.7	45.6	52.3	6.8	49.22	3.01
4	52.2	48.9	50.3	47.6	52.4	4.8	50.28	2.08
5	48.4	52.1	47.5	53.5	48.9	6.0	50.08	2.58
6	50.4	49.6	48.3	52.1	47.3	4.8	49.54	1.86
7	48.4	52.1	47.1	43.2	56.2	13.0	49.40	4.96
8	49.9	52.1	52.7	47.3	50.2	5.4	50.44	2.13
9	48.6	51.2	52.6	49.2	51.2	4.0	50.56	1.63
10	48.5	51.6	50.2	52.4	47.2	5.2	49.98	2.15
All						5.9	50.11	2.41

Inserting the values from the bottom row as well as the constants, we get the following for the X-Bar-R chart:

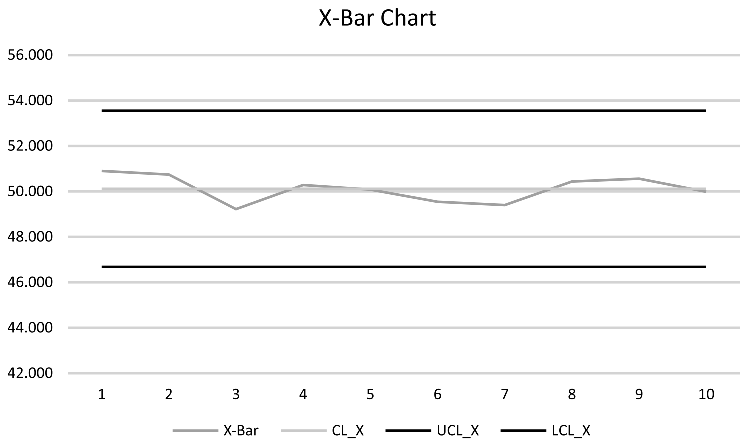
$$\begin{aligned}
\mathrm{UCL}_{\bar{X}} &= \bar{\bar{X}} + A_2 * \bar{R} && = 50.11 + 0.577 * 5.9 = 53.52\\
\mathrm{LCL}_{\bar{X}} &= \bar{\bar{X}} - A_2 * \bar{R} && = 50.11 - 0.577 * 5.9 = 46.71\\
\mathrm{UCL}_{R} &= D_4 * \bar{R} && = 2.114 * 5.9 = 12.47\\
\mathrm{LCL}_{R} &= D_3 * \bar{R} && = 0 * 5.9 = 0
\end{aligned}$$

These are the limits already used with the data in Figure 12.3. The similar values for the X-Bar-s are also calculated:

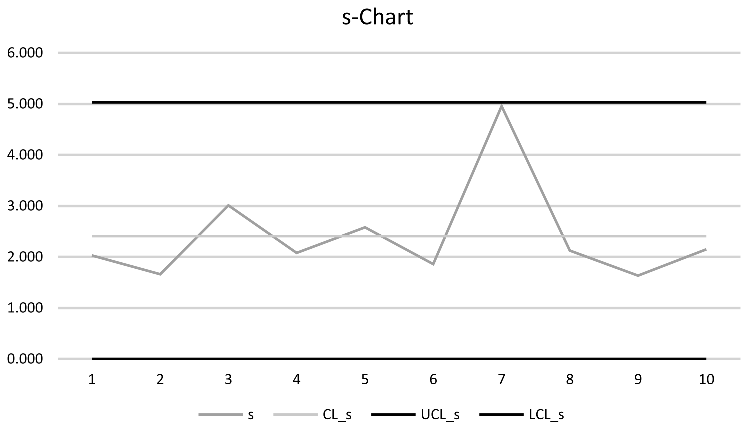
$$\begin{aligned}
\mathrm{UCL}_{\bar{X}} &= \bar{\bar{X}} + A_3 * \bar{s} && = 50.11 + 1.427 * 2.41 = 53.55\\
\mathrm{LCL}_{\bar{X}} &= \bar{\bar{X}} - A_3 * \bar{s} && = 50.11 - 1.427 * 2.41 = 46.68\\
\mathrm{UCL}_{s} &= B_4 * \bar{s} && = 2.09 * 2.41 = 5.03\\
\mathrm{LCL}_{s} &= B_3 * \bar{s} && = 0 * 2.41 = 0
\end{aligned}$$

Interestingly, the upper control limit for the R-value is more than two times the corresponding s-value. Thus, you might expect very different results from the two methods. We saw the X-Bar-R plot in Figure 12.3, and the X-Bar-s plot is shown in Figure 12.4.

So how does the X-Bar-R compare to the X-Bar-s control limits? As stated, the limits in the R- and s-charts are very different. However, looking at the corresponding charts, the results are quite similar, because the data-values in the two cases are scaled in the same way as the limits. The same thing is often stated in the literature. In our case we do, however, see a violation on day 7 in the R-based chart (Figure 12.3), which seems to be only an “almost violation” in the s-based chart (Figure 12.4).



(a) X-Bar chart - monitoring mean of subgroups



(b) s chart - monitoring variations in subgroups

Figure 12.4: X-s control chart on ten subgroups of five objects.

12.7 EWMA control charts

EWMA stands for *exponentially weighted moving average*. According to NIST, it is defined as

$$\text{EWMA}_t = \lambda * Y_t + (1 - \lambda) * \text{EWMA}_{t-1} \tag{12.7}$$

Here, Y_t is the current sample, while EWMA_{t-1} is the previous output from the formula. Lampda is a number between 0 and 1. Also according to NIST, this is typically set to something between 0.2 and 0.3. This formula is the same as the one from the simplest possible IIR filter, using only the current input sample and one older output sample; see Section 11.8. Indeed it is a simple lowpass filter, tracking the slow trends of the “input samples,” thus adapting to these, but still detecting any faster deviation. As usual in production control, the samples can be subgroups or individual measurements.

Unlike the other charts, we do not plot the variance, only the “lowpass-filtered” EWMA values. This is not very difficult. The difficulty here is that as we plot, we must also update the control-lines dynamically. NIST describes a simplification where the control-lines are calculated once after phase 1—based on the mean μ and standard deviation s calculated as formulas 12.1 and 12.3 (if there are more than 30 samples). Let’s leave it at that.

$$\mathrm{UCL}_U = \mu + 3 * \sqrt{\lambda/(2-\lambda)} * s \tag{12.8}$$

$$\mathrm{UCL}_L = \mu - 3 * \sqrt{\lambda/(2-\lambda)} * s \tag{12.9}$$

12.8 Process capability index

As stated earlier, we mainly deal with two things: process stability and process capability. In this section, we are finally dealing with the latter. Related to the control chart, the capability index is defined as 1 if the upper and lower spec limits are placed directly on top of the upper and lower control limit. As we know, this can be expected to result in 0.27 % defects.

The formula is

$$C_p = (\mathrm{USL} - \mathrm{LSL})/6\hat{\sigma} \tag{12.10}$$

The above formula will give over-optimistic estimates if the spec limits are not centered around the centerline. Instead, the following formula should be used:

$$C_{pk} = \min\left[(\mathrm{USL} - \hat{\mu})/3\hat{\sigma}, (\hat{\mu} - \mathrm{LSL})/3\hat{\sigma}\right] \tag{12.11}$$

Table 12.9 shows C_{pk} and related values. For example, the first line shows that if the spec limits are placed only one sigma below and above the centerline, then we can expect 68.27 % of all items to be okay and the number of *defects per million opportunities* will be 317311. From the definition of *Cpk*, it is no surprise that it is 1/3 in this case, and grows 1/3 each time we widen the control limit one sigma up and down. This table is simply calculated in Excel.

Table 12.9: C_{pk} and related values.

C_{pk}	Sigmas	Yield in %	DPMO
0.33	1	68.27	317311
0.67	2	95.45	45500
1.00	3	99.73	2700
1.33	4	99.99	63
1.67	5	99.9999	0.57
2.00	6	99.9999998	0.002

12.9 Further reading

- www.itl.nist.gov/div898/handbook/pmc/section3/pmc31.htm
  NIST/Sematech e-handbook of statistical methods—section on control charts.
- www.spcforexcel.com
  This site with Excel-extensions has a very extensive knowledge base.
- Alberto Cairo: *The truthful art – data, charts, and maps for communication*
  As the title says, this book is about presentation, not embedded software. Still, it introduces statistics in general. It is a very illustrative and entertaining book.
- Jan Gullberg: *Mathematics: From the Birth of Numbers*
  A fantastic nerdy brick of a book. Includes a chapter on probability theory.

Epilogue

In the epilogue to the first edition, I worried about the business model for IoT—how to generate a revenue. I also stated that although we do see some completely new stuff, most IoT solutions are probably going to make money by saving money. This seems confirmed over the past few years. Many IoT solutions are smarter implementations of existing solutions. Take, for example, the process-control in a beer-brewing company, where it was customary for employees to read meters at regular intervals and write them on paper. These readings would go into Excel spreadsheets, which could be used to regulate the process shortly after. Even later, data would be sent to management (e. g., as mail-attachments) for production planning.

Today, all of this can take place in realtime. The “inner loop” control in the factory, may be completely unmanned, and production plans in the cloud may be updated automatically. On top of this, data can be presented in the factory as augmented reality. Using the right glasses, you can look at a tank, and at the same time see temperature, alcohol content, shipping date, etc.

Making money by saving money, makes it easier for marketing. They do not need to develop completely new scenarios. Instead, they need to understand the customers’ workflow, and in the above case, the brewing-process and all that surrounds it, to help improve this. This has always been a core part of marketing.

https://doi.org/10.1515/9781547401024-013

List of Figures

List of Tables

Listings

Index