Constraint-Based Reverse Engineering and its Applications in Astrophysics

Diploma Thesis

by

Johannes Bauer

born December 6th , 1983 in Lichtenfels

Department of Computer Science 4
Distributed Systems and Operating Systems
University of Erlangen-Nuremberg
and
Dr.-Remeis Observatory Bamberg
Erlangen Center of Astroparticle Physics

Advisors:
Prof. Dr. rer. nat. Jörn Wilms
Dr.-Remeis-Sternwarte and Erlangen Center for Astroparticle Physics
Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
Lehrstuhl für Informatik 4

Start: 1st of March 2009
Submission: 31st of August 2009

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

....

Erlangen, den 31. August 2009

Abstract

In order to efficiently reverse engineer code tools are necessary which perform significantly more than simple disassembly. Such tools should aid the reverse engineer in the areas in which manual work is known to be tedious and error-prone. The engineer on his part aids the tool in the areas where automatic disassembly fails due to code obfuscation. This way the reverse engineer can concentrate on the actual work and can delegate the tedious parts to the utility at hand. To show how such an utility could look like and of what it could be capable of, a camera driver for an astronomical CCD camera is reverse engineered in the process of this work. The second part focuses on the reimplementation and astrophysical problems which need to be solved in order to create good imaging results.

Thanks

Ever since my father explained to his 10-year old son the process of code compilation as a glass which is shattered on the floor I have been fascinated of putting those pieces of debris back together. And even though the comparison is flawed it still captures well the picture of how difficult reverse-engineering machine code can be. This holds especially true when during the compilation process special care was taken to not let anyone know what really happens behind the looking glass. Even the more reason for me to try to develop tools and measures to effectively aid anyone interested in what happens when closed-source applications are run.

I would probably never have had enough endurance in pursuing my goal of becoming a computer scientist and writing this thesis without the help of many people very dear to me. Above all, I would like to thank my mother and father for enabling me to achieve this, who supported me firmly and who let me do my studies at my own pace – something I did benefit from substantially both socially and scientifically. I am also deeply indebted to my wife Julia, who greatly aided and encouraged me, who was always there for me when I needed her and who tolerated my stress-induced moodiness with a mere sigh and smile. My colleages at the Remeis Observatory also deserve my thanks – I always had a great and fun time when writing my thesis there.

This work would not be what it is today without the help and assistance of my advisors: I would like to thank Wolfgang Schröder-Preikschat for giving me the great opportunity to do an interdisciplinary thesis like this one and for setting the trust in me that I would achieve a worthwhile goal. Sincere thanks also to Jörn Wilms who constantly guided me throughout the development of this work, who gave me valuable advice and with whom I could discuss the ideas, thoughts and solutions presented in the thesis.

Contents

Chapter 1

Reverse Engineering

Today, every home computer runs vast amounts of binary code, be it the operating system or user applications. This code usually has been compiled from some language which has a higher abstraction level than machine code does. Compilation is far from trivial, but the other way around – reverse engineering binary code and trying to fully understand its meaning – is even more difficult. Sometimes for various reasons, as will be discussed in Sect. 1.1, it can be very important to have deep insight into the code which is run. Code analysis can be performed in multiple ways. The most conventional method and the method this work will focus on is *static disassembly*. This means a binary executable is analyzed while it is not run. In contrast to static disassembly, *dynamic disassembly* analyzes an executable during its runtime – this is what a debugger does. The two methods have both their own advantages and disadvantages. When using a static approach it is, for example, most difficult to analyze executables which are self-modifying such as compressed binaries. While a dynamic disassembler has access to all the register and memory values at each executed instruction, static disassemblers must rely on generalizations and assumptions. Extracting information statically is on the other way completely stealth for the application: An application has no way of defending itself against static disassembly, while in contrast there are various approaches for an application to detect a running debugger and take evasive action.

A concrete example is a CCD camera driver which is closed-source, as is the case with drivers of the Santa Barbara Instrument Group (SBIG). They produce highly efficient CCD cameras which are perfectly suited for use in astronomical applications – however, only having a Software Developers Kit (SDK) can be undesirable for a developer in terms of maintenance, as discussed in Sect. 1.1. Should the vendor decide to abandon driver support for a specific piece of hardware, the hardware becomes virtually unusable. From the customer's point of view, vendor independence in this aspect is preferable. The first step before reimplementation of a driver has to be some form of reverse engineering in order to find out what the driver does and how it operates.

Such a driver will be used as an example for reverse engineering work – it will be dissected with special utilities which were created in the process of this work. The field in which the acquired knowledge will be applied is the field of astrophysics. Specifically, this work focuses on utilities and algorithms which can be used to use CCD cameras effectively for stellar observations and caveats and workarounds employed to minimize the disturbing effects of Earth's rotation.

1.1 Motivation

Apart from the motivation of maintenance, there are further reasons why somebody would like to disassemble or reverse engineer binary code. These include:

1. **Compatibility**: This is probably the most obvious reason why someone would reverse engineer code – and probably the most important reason for people reversing code for the Open Source community. A vendor may choose to support one piece of hardware on a single platform, without publicizing details on how to communicate or interact with that hardware. For the user it might be desirable to actually use the device without having to rely on the vendor's support. The point to start is usually to reverse engineer the existing driver or control program on the proprietary platform in order to reimplement it for the desired target.

2. **Maintenance**: When drivers are released in closed-source form, the release is obviously closely bound to strict conditions. Those may be, for example, other libraries that are linked against. If a vendor decides to abandon a certain piece of hardware and will discontinue releasing new drivers for the product, the old drivers may not work on more recent systems anymore. The problem is extremely prominent when special library versions are used: A driver library which requires `libstdc++` in version 5 will not work on a recent system which has only version 6 installed. Reverse engineering is a lot easier for the user if he is aware of this fact beforehand: A dynamic, live approach is then possible without the need to install outdated, possibly deprecated files.

3. **Arbitrary Restrictions**: Some vendors choose to employ arbitrary restrictions on the software they sell, usually in the form of copy-protection. These do not add at all to the value of the software and are beneficial for the vendor alone. The reason for those restrictions are obviously to prevent users from violating copyright laws. While this is possibly true, the honest buyers of software are also affected by those restrictions. There are many good reasons to circumvent such protection, be it for performance improvement or convenience. An substantiated reason to circumvent copy protection could be, for example, the much higher speed of hard disks compared to optical media or the sheer convenience of not having to change DVDs constantly in order to run different applications.

4. **Documentation**: Poor API documentation is more common than not – either deliberately to prevent users to use an API, or because of a lack of time of the original developer. Sometimes it is therefore necessary to take a look at the interiors of what happens when a certain library call is made. This is especially interesting when it has to be determined if there are certain side effects to a call or maybe to see if a call is thread-safe. Such things can hardly be evaluated by trying out code, but must be verified by disassembly.

5. **Threat-Assessment and Auditing**: Most commercially available software is closed-source. Not all companies however deserve the trust that naïve users often put into them: Spying on its users, offering backdoors, and stealing sensitive information are popular among producers of commercial software for various reasons. These range from Digital Rights Management (DRM) over mechanisms to enforce copyright laws up to spyware. The threats that users of closed-source software have to face are manifold [Gra02]. If it is unavoidable and there are

concerns for security, it should be possible for the user to inspect the code to ensure it really is only doing what it is supposed to do.

6. **Performance profiling**: For any programmer the ideal programming language would be one in which he could express his thoughts and ideas in the most abstract way possible and the compiler would still create highly performant low-level code from that. As nice as this would be, reality looks different: Programming languages in many cases force the programmer to compromise between flexible, extensible, portable and highly problem-specific, machine-dependent programming. The assembly generated from high-level code is usually slower compared to code generated from the latter. As a rule of thumb, lower abstraction usually results in higher performance. For evaluation just how big of a performance hit a certain kind of abstraction level is, disassemblers are a viable solution. The programmer can use them to determine what the compiler actually does in order to translate the source code into binary form.

7. **Debugging**: While most high-level programmers deem it unnecessary to use a disassembler for doing their debugging work, they are generally making a mistake. Especially in the most high-level languages like C++ a disassembler can be the only option for tracking bugs in code efficiently. A certain error may manifest only under high optimization or when certain special language keywords (like `const`) are used. Without debugging the binary, it is virtually impossible to trace such programming errors. When templates are used which are inlined at compile time, an adequate tool becomes even more important, as the binary code tends to become highly unreadable without the appropriate means to aid the reverse engineer.

8. **Teaching and Research**: Comparing the binary output of the compiler helps to gain a greater understanding of what the machine does and how it does it – interesting questions like how is a system call actually invoked, how does the loader work together with dynamic libraries, and how does compilation work in general are most easily answered when taking a look on the generated binary and understanding what it does.

1.2 Differences in Assembly Representation

It is necessary to clarify exactly on what kind of reverse engineering work this thesis will focus on: It is mainly the reversing of assembly code written for the `x86` or `x86-64` architecture. While many things pointed out in this work will be almost identical on many architectures as the `ARM`, `MIPS`, `PPC`, `AVR` or `8051`, there are subtle differences. These differences become apparent when comparing the assembly generated for those architectures to bytecode assembled by a Java compiler, for instance. In order to see just how far those will go, in this section a short comparison will be drawn between `x86` assembly generated from C source code and Java bytecode. The assembly generated by the `x86` compiler will be called *low-level* assembly in contrast to Java bytecode, which will be called *high-level* assembly code in the following sections. A brief comparison between Java bytecode and compiled `x86` machine code will be given in order to explain just how different the difficulty of disassembly can be depending on the complexity of the underlying architecture.

1.2.1 Symbolic Names

As humans are far better dealing with names than with numbers, in high-level code functions are called by their name instead of the address of the function's location. To finally execute the code, the compiler must break those names down to function pointers (i.e., resolve them). These names of functions or variables are called *symbols* and some of them are embedded in the final executable, while others are not [Com95].

In low-level assembly code, symbol names can be completely stripped unless necessary for interface access. This means in a program compiled from C there must be a reference to the location of the `main()` function contained in the metadata of the executable. It is, however, not necessary for the compiler to actually include the `main()` symbol, i.e., the *name* "main" into the ELF file. The reference to `main()` will be used by the loader to jump to the entry point after the program has been loaded into memory in order to start its execution. Code compiled from C also needs to contain symbol information when functions are exported as a library: Programs linking against the library need to resolve the function pointers from the given name. For static libraries this will be done at compiletime by the linker while for dynamic libraries it will be performed during runtime by the loader.

Either way, assembled code originating from C needs to contain very little symbol information – all symbol names for subroutines which are only used within the program can be stripped. As a result they are not part of the actual assembly representation and only contained in the executable metadata (e.g., the ELF header for Linux executables). Also, calling convention is purely up to the compiler for languages like C. Obeying the established standards [St08] is optional as long as it is consistent. Someone trying to disguise the true purpose of code may therefore have interest in breaking such conventions in order to defend against someone trying to figure out what the code actually does.

In Java, things are different: Symbol names and type information are retained completely unless they are explicitly renamed [LY99]. Even when obfuscations are applied as Batchelder et al. describe them [BH07], this will not change the fact that function prototypes are preserved and so is the class structure. Also, Java is type safe on machine level, meaning that anybody reverse engineering Java bytecode has a huge advantage over reverse engineering, for example, code compiled from C.

1.2.2 Function Prototypes

As stated before, function prototypes are completely eliminated in the process of compiling low-level assembly code. Every instruction may be the start of a subroutine – subroutines can only be identified by the targets of `call` opcodes in the program. This may prove to be difficult if indirect `call` instructions are used. Even if it is known that common calling conventions are obeyed, it is next to impossible to determine the number of parameters which are passed to a subroutine for a general case. However, it is very well possible that an obfuscating compiler uses different calling conventions for various functions in order to complicate automatic disassembly. In contrast, Java bytecode preserves the whole prototype of every assembled method in its bytecode by mangling all types of formal parameters into the names of the methods. For the reverse engineer this is yet another hint on what operation the method may perform which not present in low-level assembly code.

1.2.3 Exceptions

In low-level assembly code, exception handling is a quite complicated matter. Consider the piece of C++ code presented in List. 1.1 and its assembly equivalent (which, due to its length, is only presented in the Appendix on page 65).

Listing 1.1: Exception thrown in C++

```
class moo {
    private:
        int val;
    public:
        moo(int val) : val(val) { }
        int get() const { return val; }
};
int main() {
    try {
        throw moo(0x1234);
    } catch (moo &e) {
        return e.get();
    }
}
```

Low-level assembly code requires extensive measures to be taken in order to ensure that all memory is properly cleaned up after the exception object has been constructed – even in the case the constructor of the exception object itself throws further exceptions. The code generated by the compiler is hard to understand without substantial knowledge about the process of stack unwinding and how exception handler tables are organized (i.e., in the `.eh_frame_hdr` section of the ELF binary) [Boo05].

In contrast to that, Java bytecode provides exception support as part of the JVM: There is an exception table stored in every class file which contains information about which handlers apply for code regions protected by `try` clauses, e.g.:

```
from   to  target type
   0   51    74   Class java/lang/RuntimeException
   0   60    91   any
```

An exception is thrown through creation of an exception object and afterwards execution of the special `athrow` opcode. The JVM looks up the table, matches class types and performs a jump to the given address if a match is found. Otherwise the JVM handles the stack unwinding in (emulated) hardware.

1.2.4 Structure Member Access

Structures are virtually invisible in low-level assembly which has been compiled from C source code. Consider the following example:

Listing 1.2: Structure access in C

```
struct strukt {
    int a; int b; int c; int d;
};
int main() {
    struct strukt bar;
    bar.b = bar.d;
    return 0;
}
```

Listing 1.3: Structure access in low-level assembly

```
mov -0x4(%rbp), %eax
mov %eax, -0xc(%rbp)
```

The access of the members `bar.b` and `bar.d` has been eliminated completely by the compiler. During compilation it determines the offset of the two structure members from the start of the structure. Those are, in this case, 4 and 12, respectively. The generated assembly code indicates that the compiler has laid out the memory so that the structure starts at `%rbp - 16`. The address of the members `bar.b` and `bar.d` is therefore determined at compiletime to be `%rbp - 12` and `%rbp - 4`. As the `x86-64` architecture does not allow a direct move instruction with both operands using the register indirect with displacement addressing, a detour over register `%eax` is used.

The corresponding Java equivalent could not be more different as the JVM bytecode provides own opcodes for accessing structure members:

Listing 1.4: Structure access in Java

```
class strukt {
    public int a; public int b;
    public int c; public int d;
}
public class foo {
public static void main() {
    strukt bar = new strukt();
    bar.b = bar.d;
}
}
```

Listing 1.5: Structure access in Java bytecode

```
aload_1
getfield #4 (int strukt.d)
putfield #5 (int strukt.b)
```

The references #4 and #5 are meaningless without the layout of the method and constant table – but these are available as part of the class file. It is only necessary to trace those by following the indices in the table, which is exemplary done for index #4:

```
const #4 = Field #2.#19;

const #2 = class #18;
const #18 = Asciz strukt;

const #19 = NameAndType #25:#26;
const #25 = Asciz d;
const #26 = Asciz I;
```

Although the names can be removed from such a structure in Java bytecode, the types of members and their positions in the structure cannot. Hence the information that access to an integer within a structure is performed is always preserved – giving any reverse engineer yet another advantage compared to reversing code compiled to low-level assembly.

1.2.5 Concurrent operation

Changing the current lightweight thread of execution in low-level assembly is usually performed by a dispatch function. The dispatcher copies the function pointer of the thread start function to the lower end of a previously prepared memory region. The address of this memory region is then copied into the stack pointer. When the dispatcher returns from the subroutine, it therefore jumps to the kickoff function. For an inexperienced person, such assembly code may look unusual and confusing. It is, however, easy to spot, as the direct manipulation of the stack pointer (apart from arithmetic operations) is a rather uncommon operation. Locking is seldomly implemented directly, but instead delegated to special functions within thread-handling libraries as the `libpthread`. In Linux 2.6 those in turn delegate the call to the operating system's `futex(2)` call. In

Java's high-level assembly bytecode threads are far easier to spot and locking of critical sections is obvious: Threads are created by a class inheriting from the special Thread class. They are started by the virtual method call start(). When a thread enters a critical section by using the language keyword synchronized, code is generated which contains the monitorenter and monitorleave opcodes – yet another example of how high the JVM's abstraction level really is.

1.2.6 Memory Allocation

Memory allocation in code compiled into low-level assembly can generally be differentiated between stack and heap memory allocation. Stack space is requested by the application by decrementing the stack pointer. Heap memory is allocated on Linux systems by calling the brk(2) system call which is called, for example, by the library function malloc(3). When creating new objects, all these steps can be seen on assembly level. Things become interesting when the compiler uses optimizations on arithmetic operations performed on the stack pointer when the total stack size required by the subroutine is known in advance. Memory allocation can then be performed in a single operation by decrementing the stack pointer by the sum of required memory – all objects or variables then lie next to each other with a reverse engineer having no idea which one could be which until they are actually used. In contrast, Java does type-safe memory allocation: The new opcode takes the index of a class type, for which then memory is reserved. The reverse engineer therefore always knows what type of memory a certain object refers to. With arrays it is similar, except for the fact that the newarray opcode is used by the JVM for that purpose: Here, not only is the exact size known for each array, but also the types of elements contained in the array – very much unlike low-level assembly code. Both facts make it much easier for any reverse engineer to spot interesting portions of code in Java bytecode than it is to spot a similar code in low-level assembly.

1.2.7 Return Values of Functions

Similar to the prototype of functions, there really is no strict procedure on how values should be returned in low-level assembly code – there is merely a calling convention of the application binary interface (ABI) [St08]. It varies greatly depending on platform, programming language and compiler used. Any software developer not interested in interoperability can choose the internal ABI to his own liking as long as it is kept consistent. With the GCC compiler and an x86 machine, integral data types will be returned in %eax and floats will be returned in a FPU special register. GCC on x86-64 behaves differently: Here, integral data types are returned in %rax, floats are in the %xmm0 register.

Java again does things very differently: As type safety is a very important aspect of the Java language, each returning opcode also has its own associated type. There are the return, ireturn, lreturn, freturn, dreturn and areturn opcodes to return either nothing, integers, long integers, float values, double values or pointers. Although type safety in general is very important, in this case it once again gives the reverse engineer an edge over reversing low-level assembly code.

1.3 From Assembly to Higher-Level Abstraction

1.3.1 Goals

First off, it needs to be clarified what the goals are which can be expected from a tool aiding the user in his reverse engineering work. Only when those are clearly defined in advance is it possible to measure how well they perform. Although there are works in literature which refer to the complexity of reverse engineering and particularly the deobfuscation process, there usually is no practical solution presented. This also is the result of Appel [App02], who concludes that that given knowledge about the way an obfuscation works, it "should be possible to make specialized execution-analysis tools tuned to" particular obfuscation algorithms. Although Barak et al. [BGI$^+$01] prove that it is generally impossible to obfuscate programs in a way so their original form cannot be restored efficiently, i.e. $O(P)$, many forms of obfuscation exist which cannot be reversed by any algorithm as the necessary problem which needs to be solved is undecidable. Useful applications of such algorithms are shown, e.g., by Collberg et al. [CTL97] in the form of aggregation transformations or the merge of scalar variables.

As shown in Sect. 1.2, there are remarkable differences between the level of abstraction different architectures provide. Therefore it has to be formalized what information is definitely lost (irretrievably) and what information is only converted during the process of compilation. As explained before, this is, however, highly dependent on the architecture. As Müller has shown [Mü93], it is possible to create a fully-functional Turing-complete machine which contains only 8 opcodes. It is obvious that the abstraction level of such a machine is grossly different from the one a high-level assembly language like Java bytecode provides.

The goals for the reverse engineering tool created in this work shall therefore be:

- **Aiding Data Flow**: Data flow shall be tracked by the disassembler in order to evaluate register values at certain code locations.
- **Arbitrary Addressing**: Indirect addressing shall be possible for arbitrary arithmetic expressions.
- **Intermediate Code**: The disassembler shall represent the disassembled code internally in a machine independent, high level intermediate code.
- **Code Transformations**: On the used intermediate code operations code transformations shall be possible to increase the level of abstraction.
- **Extensibility**: Updates of the disassembler like retargetation shall be possible easily.
- **Memory Models**: The utility shall support an infinite amount of separate memory spaces.
- **User Interactivity**: The user shall have the possibility to interfere with the disassemblers results at any code location and at any abstraction level (e.g., by splitting up Maximal Basic Blocks or by changing register values).

From now on this work will only refer to `x86` or `x86-64` code when discussing assembly representation unless explicitly stated so.

1.3.2 Useful Premises for Offline Analysis

To be able to decipher the meaning of static code effectively, it is very important to realize what premises are assumed when looking at code and what the implication of such premises is. Many authors merely imply these premises – one of the authors which do explicitly state the assumptions their tools rely on are Kruegel et al. [KVRV04]. The reason so few authors explicitly state the prerequisites on assembly code is probably because any code which differs from the standard case of sequential deterministic execution is difficult to handle and requires special attention.

Moreover, it is essential to keep in mind that heavily obfuscated code may explicitly break simple assumptions in order to complicate disassembly. Thus the design of a reverse engineering utility should be kept modular enough to be able to handle such deviating code. The basic assumptions made in standard code are:

- The internal state of the processor will change only in the way specified by the performed instructions.
- Except for control transfer instructions code will execute in a sequential fashion.

In practice neither of those assumptions is completely true, but systems mostly behave as if they were. Code that is analyzed will rarely run on bare hardware, but rather on top of an operating system. The operating system will most likely be of preemptive kind and continuously generate interrupts by a system timer. The Interrupt Service Routine (ISR) then might occasionally preempt the running process in favor of another – in this process not also will the control flow change to a completely undefined location, but also the internal state of the processor will. However, in order to keep everything as simple as possible for the application developer, such operating systems will usually restore the environment completely before returning flow control to the user application. It could be possible, however, that for example specialized types of operating systems make use of the possibility to change the applications registers. Since the above assumptions would then be broken, the constraints for analysis would necessarily have to be reevaluated. To show that such cases really exist in real-world applications, consider the piece of code presented in List. 1.6, which was compiled for a AVR microcontroller.

Listing 1.6: AVR assembly example using a volatile register

```
0000004c <main>:
  4c:   88 24          eor r8, r8

0000004e <check>:
  4e:   88 20          and r8, r8
  50:   f1 f3          breq    .-4        ; 0x4e <check>
  52:   01 d0          rcall   .+2        ; 0x56 <done>
```

If it would be executed in a non-concurrent (i.e., sequential) manner, the `breq` opcode at `0x50` would always branch since `r8` would always remain 0. If an Interrupt Service Routine (ISR) would however change that register value, the loop would break and `done` would be called. This shows how important it is that assumptions made about code are true in all cases – anything may leads to a false disassembly result.

1.3.3 Control Flow Representation

For all further discussion the control flow representation of any given program will be represented by control flow graphs, or CFGs for short. The formal representation of a

CFG is the graph tuple

$$G(V,E) \tag{1.1}$$

$$V := \{v_1, v_2, \ldots, v_n\} \tag{1.2}$$

$$E := \{(x_1,y_1),(x_2,y_2),\ldots,(x_m,y_m)\}, \quad x \in V \wedge y \in V \tag{1.3}$$

where V denotes the vertices and E denotes the edges in the graph. The graph is directed and possibly cyclic. Any vertex in that a CFG represents a Maximal Basic Block. Maximal Basic Blocks are sections of code which fulfill the following requirements:

- Except for the last instruction in the block they do not contain any control flow instruction.
- Except for the first instruction in the block they are not referenced by any other control flow instruction in the CFG as a target.
- The block cannot be enlarged in either direction without violating the other requirements.

The edges between blocks represent control flow transitions from one MBB into another.

The algorithm to identify the first instruction of MBBs is as follows:

- The first instruction in the program (entry point) starts a MBB.
- Any instruction referenced by a control flow instruction in the program starts a MBB.
- Any instruction directly following a control flow instruction in the program starts a MBB.

This algorithm will determine all static entry points for MBBs from which on the actual MBB can be deduced by simply increasing the MBB, sequentially adding all instructions which do not branch control flow. Dynamic entry points of MBBs (i.e., entries induced by indirect `call` or `jmp` opcodes) can not be determined in that manner – a constraint-based analysis can detect some, but not all.

1.4 Pitfalls and Obfuscation

1.4.1 Clarification

Software developers may have various reasons to inhibit reverse engineering of deployed code. In order to make it more difficult for reverse engineers to perform their work some anti-reversing techniques have been developed. These are called *obfuscations*. There are many kinds of obfuscations – some are targeted against the disassembler and others against the reverse engineer. How such obfuscations work and how a constrained-based disassembler may cope with them is explained in the next sections.

1.4.2 Overlapping Instructions

A problem of disassembling binary code is to separate instructions from data – which may both very well reside in the same segment. When the location of the instructions has been determined, another problem is to identify which of these instructions are actually valid. Valid in this context means instructions which might actually executed during runtime and which do not serve the sole purpose of confusing disassemblers and thus obfuscating code. Consider the following piece of x86-64 code:

Listing 1.7: Overlapping assembly instructions

```
4004a0 <main>:
  4004a0:       eb 09                   jmp    4004ab <safe>

4004a2 <illegal>:
  4004a2:       48 c7 c0 0f 0b 00 00    mov    $0xb0f, %rax
  4004a9:       eb fa                   jmp    4004a5 <illegal+0x3>

4004ab <safe>:
  4004ab:       c3                      retq
```

The particular piece of code shown in List. 1.7 will be successfully processed by a disassembler using either a linear-sweep- or a recursive-descent-approach. The interesting part lies in the code started by the "illegal" label. The jump instruction jumps "into" the mov instruction, effectively decoding the partial mov as an ud2 opcode, which will result in an illegal instruction trap to be thrown by the processor. This is actually no problem – the problem is that a disassembler will continue disassembling from that instruction on and might skip or even crash when completely illegal opcodes are found on that path.

As Linn et al. have shown [LD03], constructs like the above are highly artificial and can seldomly be constructed from regular code. Therefore, although they may in theory contain useful code, they will usually only appear in order to confuse disassemblers.

1.4.3 Opaque Constraints

Conditional branches in machine code are always split into two parts (although these two may be combined into the same opcode): First some sort of comparison occurs, which yields a boolean value, followed by the branch, which depends on the outcome of the comparison. Code emitted which does a comparison of which the outcome is always known beforehand is called a *opaque constraint* or *opaque predicate*. It is used solely to obfuscate code. When the result of the condition evaluation is constant, the conditional branch effectively becomes an unconditional branch with the added benefit of confusing the decompiler with a dead branch target. This dead branch may point to valid code sections with an invalid offset as described in Sect. 1.4.2, or it may point into a data section. In order to trigger false detection of control flow in functions, it may also simply point to a main code line – how a disassembler can try to evade such maneuvers can be found in Sect. 1.6.4. The trick aims at fooling the disassembler into the assumption that code might branch when in practice it never will because of the opaque predicate. This might give the programmer the advantage of leading the disassembler of a reverse engineer into code regions which contain illegal instructions – resulting in skipped instructions, a wrong CFG and possibly even a disassembler crash.

Collberg et al. describe many different types of opaque constraints [CTL98]. Not all opaque constraints can be identified as such by sole offline analysis. This is the case, e.g., when concurrent threads modify memory in a pseudo-random manner [Eil05]. In order to fulfill the requirement of stealth, however, the opaque constraint has to be as inconspicuous as possible – therefore usually very short portions of code are used for this purpose.

The simplest form of an opaque constraint is a comparison of two known values and a following jump, as seen in List. 1.8.

Listing 1.8: Most basic form of an opaque constraint

```
4004a4:    48 c7 c0 00 0f 00 00      mov    $0xf00, %rax
4004ab:    48 c7 c3 a2 0b 00 00      mov    $0xba2, %rbx
4004b2:    48 39 c3                  cmp    %rax, %rbx
4004b5:    75 00                     jne    <somewhere>
```

These constructs can be easily beaten by static analysis of a constraint-based disassembler. There are more complex forms of opaque constraints, however, which are only slightly more complicated to write, but much more complicated to solve, as in List. 1.9. Here, first a register is cleared, effectively setting the ZF to 0. The condition codes contained in the `RFLAGS` register are pushed onto the stack using the `pushf` opcode [Int07b]. From there it is manually retrieved in order to check the zero flag, which is located at the 6th bit [Int07a]. Such constraints can also be beaten by the disassembly tool developed in the course of this work.

Listing 1.9: More complicated, but still solvable, opaque constraint

```
400514: 48 31 c0                 xor    %rax, %rax
400517: 9c                       pushfq
400518: 58                       pop    %rax
400519: 48 83 e0 20              and    $0x20, %rax
40051d: 75 07                    jne    <never>      ; Will never be
    taken
```

Finally, there are opaque constraints which are so complicated that a constraint-based disassembly tool cannot determine if they are true or not. Consider the piece of code in List. 1.10 for example.

Listing 1.10: Unsolvable opaque constraint

```
unsigned int cnt = 0;
for (unsigned int i = 0; i < 1000; i++) {
    if (randomfloat(1.0) < 0.1) cnt++;
}
if (cnt) {
    // Live code branch
} else {
    // Dead code branch
}
```

This opaque constraint is not solvable because it has nondeterministic behavior. Assuming the random source is really random, it can, in fact, not be predicted which branch will be taken. However, the probability the live code branch is taken is much grater than the probability of taking the dead code branch. In order to reach the dead code branch the random value $0 \leq r < 1$ would have to be greater than 0.1 for 1000 consecutive times. The chance of that happening are approximately

$$p = (1-0.1)^{1000} \approx 1.74787 \cdot 10^{-46}$$

Collberg et al. describe in the highly interesting possibility of including a theorem prover into the disassembler in order to be able to decode even more complicated types of opaque predicates [CTL97]. One example they give is

$$x^2(x+1)^2 = 0 \pmod 4$$

Another interesting possibility might be to introduce numerical problems which can only be solved algebraically or by iteration, but which cannot be calculated directly, such as:

$$\lim_{x \to 0} \frac{\sin x}{x} = 1$$

or

$$\min_{x>0} x^x = e^{-e^{-1}}$$

The complexity when using such theorems leaves great room for experimentation. Collberg et al. [CTL97] describe the idea of using a simple, yet unproven, conjecture as an opaque predicate such as the Collatz problem, seen in Eq. 1.4. The conjecture states that recursive application of the formula will eventually converge to 1 [Lag96].

$$T(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases} \tag{1.4}$$

As the Collatz conjecture holds true for values well beyond 2^{32}, it is possible to choose an arbitrary integer $i \leq 2^{32}$ as an input value to introduce nondeterminism. In that range, the loop will terminate within 1050 steps (this worst-case value occurs, e.g., for $i = 2610744987$ as can be calculated easily on any modern computer within a hour).

1.4.4 Abstruse Code

A final example for the difficulty of reverse engineering is the generation of abstruse code. It is a highly efficient method to confuse a human reading the disassembled output. Abstruse code can be generated in an extremely performant way while deobfuscation requires a powerful disassembler. Consider this piece of code:

Listing 1.11: Abstruse code

```
40060c <abstruse>:
  40060c:       48 c7 c0 ee ff c0 00    mov    $0xc0ffee, %rax
  400613:       48 3d 0d f0 ad 0b       cmp    $0xbadf00d, %rax

400619 <i1>:
  400619:       74 02                   je     40061d <i3>

40061b <i2>:
  40061b:       75 02                   jne    40061f <i4>

40061d <i3>:
  40061d:       77 04                   ja     400623 <target1>

40061f <i4>:
  40061f:       73 0a                   jae    40062b <target2>

400621 <i5>:
  400621:       eb 10                   jmp    400633 <target3>
```

```
400623 <target1>:
  400623:       48 c7 c0 01 00 00 00    mov    $0x1, %rax
  40062a:       c3                      retq

40062b <target2>:
  40062b:       48 c7 c0 02 00 00 00    mov    $0x2, %rax
  400632:       c3                      retq

400633 <target3>:
  400633:       48 c7 c0 03 00 00 00    mov    $0x3, %rax
  40063a:       c3                      retq
```

At the start of List. 1.11 the comparison between `0xc0ffee` and `0xbadf00d` is performed – after the comparison instruction all condition codes of the CPU are known and can be calculated by a disassembler using a constraint-based approach. Therefore, when it can be assumed that the labels `i1` through `i5` are never referenced from outside the `abstruse` function, it can be calculated in advance what value the function will return in advance. However, for a human reader the prediction is error-prone and tedious.

1.5 The Constraint-Based Approach

1.5.1 Nomenclature

The internal state of a CPU usually is composed of many subunits:

- General-purpose registers,
- Special-purpose registers: Stack Pointer and Instruction Pointer,
- Flags Register (Condition Codes).

This list is incomplete – however for the analysis of static code it usually is unnecessary to include other parameters, such as, for example, the internal instruction counter a CPU may provide. Any subset of these variables might be taken into account for constraint-based analysis. The set of analyzed state parameters will be called A, the set of so-called *aspects* which are taken into consideration for analysis. For classical static disassembly, $A = \emptyset$.

For any set of CPU instructions, $I = \{i_1, \ldots, i_n\}$, which are detected during disassembly there exists a unambiguous relation which assigns every instruction i to exactly one Maximal Basic Block of the set $B = \{b_1, \ldots, b_m\}$. B is a partition of I:

$$\forall i \in I \quad \exists b \in B : i \in b, \quad b_i \cap b_j = \emptyset \text{ if } i \neq j, \quad \cup B = I \tag{1.5}$$

The instructions contained in any Maximal Basic Block are then referred to again in indices $b_{i,j}$, where $1 \leq i \leq m$ determines the number of the MBB and $1 \leq j \leq |b_i|$ determines the actual instruction number in order of natural execution.

The state of the processor is called a *condition* c. It is represented as a set of tuples $(\text{aspect}, \text{value})$ which includes all known processor aspects at that time. Aspects not contained in such a set are considered undefined:

$$c = \{(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)\} \tag{1.6}$$

Of particular interest are the *preconditions* and *postconditions* associated with every MBB $b \in B$. All preconditions are guaranteed to be satisfied before the block is executed (no matter by which edge of the CFG) and all postconditions are guaranteed to be satisfied after the block has completed its execution.

For clarification an example of the above is given in Appendix A.2 on page 61.

1.5.2 Principle of Operation

The idea of doing constraint-based reverse engineering is in theory simple and similar to the data flow analysis provided by some disassemblers. After the code has been disassembled and the MBBs have been identified, the code is internally annotated by a set of constraints C which reflect a subset of the processor's internal state. In practice, this subset will contain the most important condition codes and registers the CPU provides. All elements of A will initially be set to an undefined value. Consider the piece of code shown in List. 1.12. For simplicity, in this example A was chosen to be

$$A = \{\mathrm{ZF}, \mathrm{OF}, \mathrm{CF}, \mathrm{rax}, \mathrm{rbx}, \mathrm{rcx}, \mathrm{rdx}\} \tag{1.7}$$

The set includes four general-purpose registers `%rax` through `%rdx` and three condition codes (processor flags) which are the zero, overflow, and carry flag. For the short example List. 1.12 the CFG is shown in Figure 1.1. Assuming that MBB_3 can only be reached through MBB_1 or MBB_2, the precondition of MBB_3 is the intersection of the postconditions MBB_1 and MBB_2. In general this means

$$cr_i = \bigcup_j co_j \quad \forall j | \exists E(b_i, b_j) \tag{1.8}$$

Listing 1.12: Constraint-Based disassembly showing all conditions C

```
; { }
MBB1:
  mov $9, %rax
    ; +{ rax = 9 }
  mov %rax, %rbx
    ; +{ rbx = 9 }
  mul %rbx
    ; +{ rax = 81, rdx = 0, OF = 0 }
  jmp MBB3
; { rax = 81, rbx = 9, rdx = 0, OF = 0 }


; { }
MBB2:
  xor %rdx, %rdx
; { rdx = 0, ZF = 1, OF = 0, CF = 0 }


; { rdx = 0, OF = 0, CF = 0 }
MBB3:
  jmp MBBx
; { rdx = 0, OF = 0, CF = 0 }
```

1.6 Possible Extensions

While the principle of operation discussed in Sect. 1.5.2 for itself is not very spectacular, it is merely the description of the most basic data flow analysis. This basic approach can be extended in various ways to yield far more impressive results aiding the reverse engineer in his work.

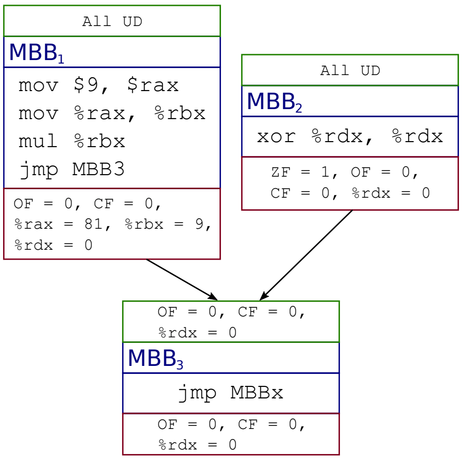


Figure 1.1: Maximal Basic Blocks arranged in the CFG

1.6.1 Extending Data-Flow to Arithmetic Expressions

Updating conditions in the data-flow analysis like in Sect. 1.5.2 is only of limited use: Usually most of the variables will be undefined firsthand and therefore the disassembler will yield little helpful information. This behavior can be improved when the disassembler is programmed in a way to use variables optionally instead of undefined values. In the representation chosen, the user will always use the `assume` statement borrowed from Maple to tell the disassembler to assume certain premises at locations in code.

Listing 1.13: CRC-8 calculation with polynomial $x^8 + x^5 + x^4 + 1$

```
8048430:       55                      push   %ebp
8048431:       89 e5                   mov    %esp, %ebp
8048433:       0f b6 4d 08             movzbl 0x8(%ebp), %ecx
8048437:       53                      push   %ebx
8048438:       0f b6 5d 0c             movzbl 0xc(%ebp), %ebx
804843c:       89 c8                   mov    %ecx, %eax
804843e:       89 ca                   mov    %ecx, %edx
8048440:       83 e0 01                and    $0x1, %eax
8048443:       81 e2 e6 00 00 00       and    $0xe6, %edx
8048449:       31 c3                   xor    %eax, %ebx
804844b:       89 d8                   mov    %ebx, %eax
804844d:       d1 fa                   sar    %edx
804844f:       c1 e0 07                shl    $0x7, %eax
8048452:       09 d0                   or     %edx, %eax
8048454:       89 ca                   mov    %ecx, %edx
8048456:       83 e2 10                and    $0x10, %edx
8048459:       83 e1 08                and    $0x8, %ecx
804845c:       c1 fa 04                sar    $0x4, %edx
804845f:       31 da                   xor    %ebx, %edx
8048461:       c1 f9 03                sar    $0x3, %ecx
8048464:       31 d9                   xor    %ebx, %ecx
8048466:       c1 e2 03                shl    $0x3, %edx
8048469:       09 d0                   or     %edx, %eax
804846b:       c1 e1 02                shl    $0x2, %ecx
804846e:       09 c8                   or     %ecx, %eax
8048470:       5b                      pop    %ebx
8048471:       5d                      pop    %ebp
8048472:       c3                      ret
```

A good example for this technique can be seen in List. 1.13, a function which computes the CRC-8 checksum of a single bit. Deducing manually what operation this function performs exactly is tedious and, much more importantly, extremely error-prone. When using a constraint-based approach all the work can be left to the computer algebra system (CAS) running in the background of the disassembler. The full computation is available in the appendix as List. A.1 on page 62. Here only the result will be discussed:

```
eax := OR(OR(OR(SHL(XOR(Par2, AND(Par1, 1)), 7), SHR(AND(Par1, 230), 1)),
  SHL(XOR(SHR(AND(Par1, 16), 4), XOR(Par2, AND(Par1, 1))), 3)),
  SHL(XOR(SHR(AND(Par1, 8), 3), XOR(Par2, AND(Par1, 1))), 2))
```

This can be written as:

Listing 1.14: Return code written in C-like fashion

```
eax = ((Par2 ^ (Par1 & 0x01)) << 7
       | (Par1 & 0xe6) >> 1
       | (((Par1 & 0x10) >> 4) ^ (Par2 ^ (Par1 & 0x01))) << 3
       | (((Par1 & 0x08) >> 3) ^ (Par2 ^ (Par1 & 0x01))) << 2
```

If this still looks confusing or unclear, the original code should be considered from which it was initially assembled:

Listing 1.15: Actual C source code of the listing

```
unsigned char CRC8_Bit(unsigned char OldCRC, unsigned char Bit) {
    unsigned char NewCRC;
    NewCRC = (((OldCRC & 0x01) ^ Bit) << 7) |
        ((((OldCRC & 0x10) >> 4) ^ ((OldCRC & 0x01) ^ Bit)) << 3) |
        ((((OldCRC & 0x08) >> 3) ^ ((OldCRC & 0x01) ^ Bit)) << 2) |
        ((OldCRC & 0xe6) >> 1);
    return NewCRC;
}
```

So by telling the disassembler to use the computer algebra system on a certain register of a function by usage of variables instead of undefined values essentially the whole function could be restored into a compilable, readable form.

Listing 1.16: Comparison of global variable with a constant

```
80484b5:       a1 18 a0 04 08         mov    0x804a018, %eax
80484ba:       3d ef be 00 00         cmp    $0xbeef, %eax
80484bf:       75 07                  jne    80484c8 <main+0x24>
80484c1:       e8 ce fe ff ff         call   8048394 <code1>
80484c6:       eb 05                  jmp    80484cd <main+0x29>
80484c8:       e8 cc fe ff ff         call   8048399 <code2>
80484cd:       [...]
```

This fact is particularly interesting when considering that the condition codes the architecture provides are included in the disassembler aspects *A*. Their inclusion does not make much sense when only representation of explicit values is possible as this will almost never be the case in production code. However, when using a CAS capable of boolean expressions in the background, the code in List. 1.16 can be transformed in an abstract way: The important parts are shown in List. 1.17, where the `DInt()` function is the CAS representation of dereferencing an address pointing to a 32-bit signed integer value.

Listing 1.17: Impact of the comparison on the ZF aspect

```
; { }
mov     0x804a018, %eax
; +{ eax = DInt(0x804a018) }
cmp     $0xbeef, %eax
; +{ ZF = ((DInt(0x804a018) - $0xbeef) == 0) }
jne     80484c8 <main+0x24>
```

As the disassembler knows that the `jne` opcode is essentially an instruction which translates to

```
if (ZF == 0) goto Target
```

this means, knowing about the aspect ZF, the disassembler can translate the `jne` instruction into

```
if (((DInt(0x804a018) - $0xbeef) == 0) == 0) goto 0x80484c8
```

which any capable CAS will optimize to

```
if (DInt(0x804a018) - $0xbeef) goto 0x80484c8
```

Knowing the preference of programmers, such constructs present in intermediate code can be further translated into the final representation

```
if (*((int*)0x804a018) != $0xbeef) goto 0x80484c8
```

Such an aid is great for any reverse engineer, as he must not concern himself with the low-level details, but immediately sees what work the code is actually performing.

1.6.2 Maximal Stack Regions

Indirections in code relative to the stack pointer are difficult to read for humans as the stack pointer value constantly changes due to `push` or `pop` instructions. Many times, the actual value of the stack pointer is not relevant, whereas the value relative to the entry point of a function is. As a solution, Maximal Stack Region analysis is presented: A MSR is a portion of code in which the stack pointer does only change in predeterminable ways. These might be `push` or `pop` instructions or arithmetic computations. Analysis of MSRs is done in much the same way as the standard analysis of MBBs described in Sect. 1.3.3. The MBB analysis is done beforehand and is used to calculate the MSRs:

1. At first, the first instruction of each MBB is assigned its own MSR with offset 0.

2. Then, each MSR is processed sequentially. The instructions encountered can be one of the following kind:

   (a) The instruction does not modify the stack pointer in any way: Assign the instruction the same MSR with same offset as the instruction right before it.

   (b) The instruction modifies the stack pointer in a predeterminable way: Assign the instruction the same MSR of the instruction right before it, but adjust the MSR offset to reflect the stack pointer change inflicted by the instruction itself.

   (c) The instruction modifies the stack pointer in an indeterminable way: Assign the instruction a new MSR with offset 0 and break the current MSR up into two separate MSRs.

After this is done the analysis yields a consistent memory block table, but the memory blocks are not yet maximal. To get the final MSRs, optimization is necessary: Any MSR of which all predecessor MSRs are equal can be optimized away. This is especially the case when there is only one predecessor node. When only an edge $\text{MSR}_0 \rightarrow \text{MSR}_1$ exists, all references of MSR_1 can be replaced by MSR_0. The offset has to be adjusted so the new offset of all instructions in MSR_1 is the sum of the MSR offset of the last instruction of MSR_0 and the MSR_1 instruction offset. The optimization is iteratively applied until there are no more changes in the layout of the MSRs.

If an analysis is done that way, it is possible to assign to each instruction within every MSR a offset. The initial MSR offset is 0. Every instruction $p_{i,j+1}$ has the offset $p_{i,j} + \text{offset}(p_{i,j+1})$, where the offset function returns the relative stack pointer change of an instruction. An example is given in the appendix on page 63. This makes it possible to calculate at any place in the code where the address of a the stack pointer is relative to the current MSR. The advantage for the reverse engineer is that it is not necessary to think relative to the stack pointer (which may change throughout code by, e.g., `push` or `pop` instructions), but in the contiguous memory segments which are represented by the MSRs.

1.6.3 Extension by Specialized Constraints

When disassembling code for a specific target, there are instances in which certain code sequences or instructions are used which have a specific high-level equivalent. Usually those are introduced not in order to obfuscate code, but simply because of the limitations of the architecture. Consider the code shown in List. 1.19 which is from a shared object compiled for the `x86`.

Listing 1.18: Position-independent in source code of a library

```
extern int cup;

void foobar() {
    cup = 0xc0ffee;
}
```

Listing 1.19: Position-independent code on `x86`

```
469 <__i686.get_pc_thunk.cx>:
469:   8b 0c 24                  mov    (%esp),%ecx
46c:   c3                        ret

0000043c <foobar>:
43c:   e8 28 00 00 00            call   469 <__i686.get_pc_thunk.cx>
; { ecx = 0x441 }
441:   81 c1 b3 1b 00 00         add    $0x1bb3,%ecx
; { ecx = 0x1ff4 }
447:   8b 81 f8 ff ff ff         mov    -0x8(%ecx),%eax
44d:   c7 00 ee ff c0 00         movl   $0xc0ffee,(%eax)
453:   c3                        ret

1ff4 <_GLOBAL_OFFSET_TABLE_>:
[...]
```

After the call of the rather peculiar function `__i686.get_pc_thunk.cx`, the return address of the next instruction after the `call` opcode is pushed on the stack. This address then is moved to the `%ecx` register before returning to the caller. The value is then added with a precalculated offset in order to retrieve the address of the global offset table (GOT). This table then is indirected with a displacement of -8 in order to get the address of the effective target. The indirection over the GOT is necessary because the shared library is relocatable and therefore needs to consist of position independent code (PIC). The absolute addresses may change, but the relative address of the instruction at `0x43c` relative to the GOT will not change when the library is mapped to memory in one contiguous block. During runtime the address of `cup` is then resolved and written to `GOT - 0x08`. This address then is finally assigned its value during the call of `foobar`.

The way in which a constraint-based disassembler can aid the reverse engineer in this scenario is the following: When `__i686.get_pc_thunk.cx` is identified as a function which solely moves the return address to `%ecx`, the reverse engineer can tell the disassembler that the postcondition of the MBB should be:

```
assume((%esp) = %eip, trait=postcall)
```

This is not performed automatically, as it is not always ensured that the function is really reached via a `call` opcode. Therefore the assumption will have to be stated explicitly with the added `trait` keyword – this tells the disassembler that the statement is only effective after a call has returned. In this case it is necessary because the referenced value `%eip` is not yet available within the MBB `__i686.get_pc_thunk.cx`. When the disassembler is allowed to safely assume that `%ecx` holds the value of `%eip` after the call it can precalculate the addresses as shown in List. 1.19. It then can also identify the indirection with displacement instruction to read a value from the GOT, as the size of the GOT has to be known beforehand. This way it is possible to transform the four instructions from `0x43c` to `0x45e` into the single pseudocode instruction `mov %eax, Global[2]` (the 2 being the size of the GOT plus the displacement in the GOT divided by four). When comparing the code generated for `x86` to that generated by `x86-64`, the above function call will not occur: The `x86-64` allows, in contrast to the `x86`, indirect register access relative to the instruction pointer `%rip`.

Another use case of specialized constraints is the access of 16-bit words on the 8-bit `AVR` architecture, as shown in List. 1.20.

Listing 1.20: AVR code accessing a 16 bit `int` variable

```
ce: cf ef           ldi r28, 0xFF   ; 255
d0: d0 e1           ldi r29, 0x10   ; 16
d2: de bf           out 0x3e, r29   ; 62
d4: cd bf           out 0x3d, r28   ; 61
```

For the reverse engineer it is quite apparent that the registers `r28` and `r29` belong together as one word and so do `r24` and `r25`. However, the disassembler cannot simply assume this is true, but must rely on the aid of the reverse engineer. He can instruct the disassembler to assume that two bytes belong together to form a double word by use of the `typeof` command:

```
assume(typeof(r29:r28, dword))
```

Then, in the ongoing code transformations, the disassembler will always join `r28` together with `r29` to form the virtual 16-bit register `r28:29`. The same is possible with memory-mapped I/O addresses:

```
assume(typeof(0x3e:0x3d, dword))
```

So that through the iterative optimization process, in the end the internal intermediate code will yield

```
(uint16_t*)(0x3e:0x3d) = 0x10ff
```

which is exactly the original form of the code in question.

1.6.4 Determining the Extent of Subroutines

To be able to group logically coherent parts of the disassembled output together into subroutines, it is an important goal to identify those sections of code which belong together. For this purpose, the following algorithm can be used:

1. Identify all opcodes which are directly referenced by `call` instructions.
2. Annotate the MBBs which are targeted by `call` opcodes with a sequentially increasing function number f_0 to f_{n-1} with n being the number of subroutines detected in the control flow.
3. For each function f_i start a depth-search following the CFG at control at all control transfer instructions except for `call` opcodes. If is it possible to determine that a control transfer instruction will never take a branch, do not follow it.
4. Annotate each MBB in the way of the depth-search with the label f_i.
5. Stop when a `ret` opcode is encountered.

This mapping can be ambiguous, i.e., it is possible that an MBB belongs to more than one function. An obfuscating compiler can also obstruct the correct functionality of this mapping by use of opaque constraints. To solve this problem, the use of a constraint-based disassembler can help as described in Sect. 1.6.1.

Chapter 2

Astrophysical Premises

After the theoretical considerations of the previous chapters, we now turn to the practical application in astronomy. In this chapter, the astronomical and instrumental background is set. The following chapter 3 then describes the application of reverse engineering on a special piece of hardware. Observation of the skies may seems like a straightforward endeavour – however, it can become almost arbitrarily complicated when the requirements of equipment and software need to fulfill scientific standards. In order to give a quick insight on how telescopes and imaging devices work, the theory will be discussed in the next few sections. An understanding of these theoretical basics is important as many effects impact the process of creating high-quality images.

2.1 Basics and operation

2.1.1 Telescopes

There are many ways how optical instruments can be built in order to achieve magnification of objects. A relative simple design is a simple lens construction in a so-called *refractor* telescope. For larger telescopes, refractors are unsuitable as a cell can only sustain its lens in the circumference – large lenses therefore deform under their own weight leading to optical aberrations. Mirror telescopes or *reflector* telescopes are a viable solution to this problem, as a mirror can be attached with its whole back surface to the telescope tube, leading to much better stability. One design used for a reflector telescope is the Cassegrain design, which is shown in Fig. 2.1. The parallel rays of light coming from the sky fall onto the primary mirror which has a hole in the middle and has been ground into parabolic shape. The light is from there reflected onto the secondary mirror which has hyperbolic shape. It reflects the light back through the hole in the primary mirror onto the ocular or CCD camera used for imaging.

For doing imaging, the mount carries a telescope which has specific optic parameters. Among the most important are the focal length f and the objective diameter D or telescope *aperture*. For calculations the aperture ratio F is interesting:

$$F = \frac{D}{f} \tag{2.1}$$

while in digital photography often the inverse is used, called the f-number or relative

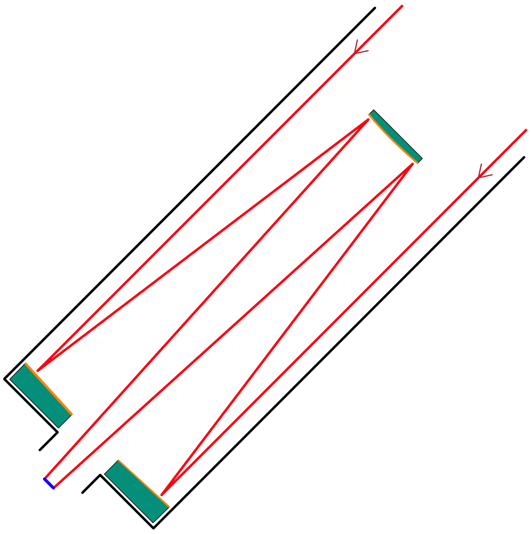


Figure 2.1: Schematic drawing of a Cassegrain telescope

aperture [Smi05]:

$$\kappa = \frac{1}{F} = \frac{f}{D} \tag{2.2}$$

κ is a dimensionless unit but, especially in digital photography, often is written as f/κ (e.g., $f/8$). An object seen under the angle u forms an image of height s [KKO+96]:

$$s = f \tan u \approx f u \tag{2.3}$$

The simplification is possible since the angle u is usually very small: For $u = \frac{100}{3600}^{\circ} = 100''$ the relative error is in the region of 10^{-6}, for $u = 1''$ it is already less than 10^{-10}.

The resolution of the system is not only limited by the imaging device, but also by the telescope itself. Because for high focal lengths as they occur in telescopes, not only the lens imperfections become relevant, but also diffraction of light. Diffraction limits the maximal achievable optical resolution of an optical instrument. The so-called Airy Diffraction Disk is the pattern which is generated by these limitations. The angle between the maximum in the center and the first minimum can be calculated by [McL97]:

$$\theta = 1.22 \frac{\lambda}{D} \,\mathrm{rad} \tag{2.4}$$

The Raleigh criterion uses the Airy Disk to calculate the minimal distance two objects can have in order to be able to distinguish them during observation. In the visible range from around 350 nm to 750 nm diffraction is reasonably small, as can be seen in Fig. 2.2.

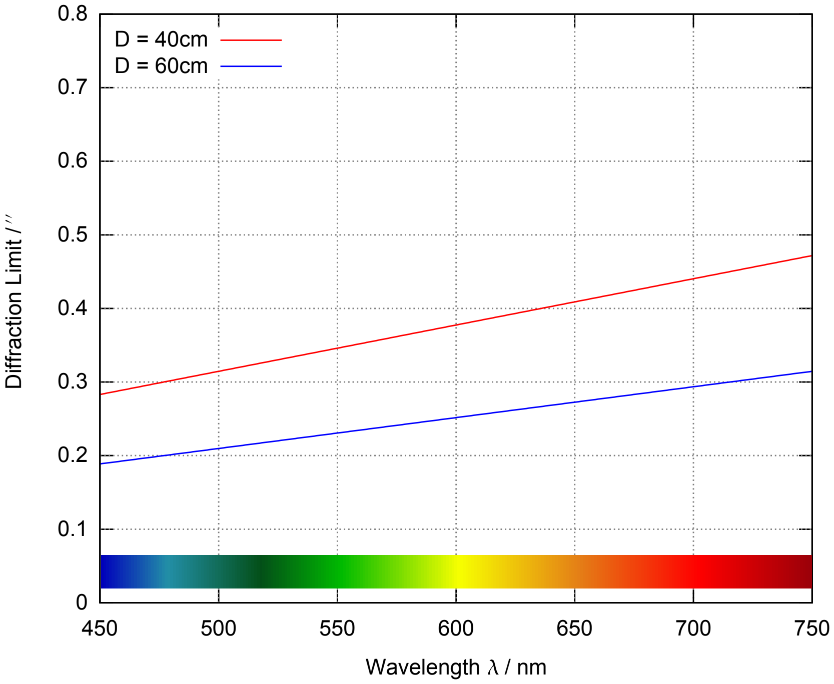


Figure 2.2: Diffraction limited resolution of 40 cm and 60 cm telescopes in visible light

2.2 Imaging Detectors

In order to use telescopes to make images, astronomers originally used photography. Since the early 1990s, this technique has been superseded by the use of charge coupled devices (CCDs). Starting after their invention in 1969, these detectors have today become the single most important imaging technique used in astrophysical imaging [McL97].

A CCD chip is an array of photosensitive pixels which is usually arranged in rectangular shape. Every pixel itself is a *bin* for charge. A photon hitting the CCD carries a specific energy

$$E = h\nu = \frac{hc}{\lambda} \tag{2.5}$$

where h is Planck's constant, c is the speed of light and λ is the wavelength of the photon. When the photon is absorbed in the CCD chip, if E is larger than the band gap of Silicon (a few electron volts for typical CCDs), then an electron is excited from the valence into the conduction band and eventually stored in the charge bin. The resulting charge distribution over the CCD is therefore a representation of the distribution of light hitting the CCD, i.e., it is an image. The major advantage of CCDs is that they have a very high quantum efficiency in the 80–90% range, i.e., 80–90% of all optical photons result in a detectable charge. This efficiency is much higher than that of photographic emulsions with an equivalent quantum efficiency of 1–2%. Therefore CCDs are significantly more sensitive than photographic film [McL97]. Charge bins are not of infinite size – they can overflow into neighbouring pixels. This effect is called *blooming* and it occurs, for example, when a CCD is overexposed.

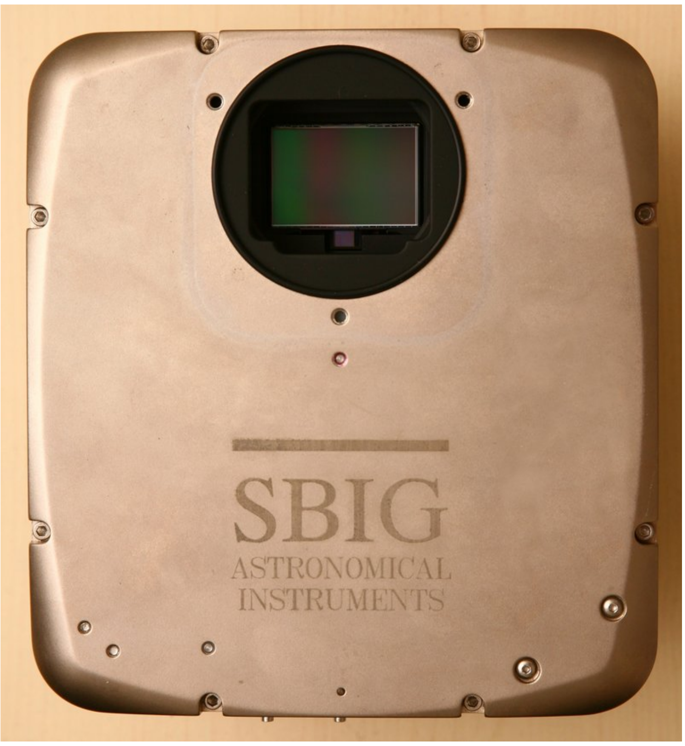
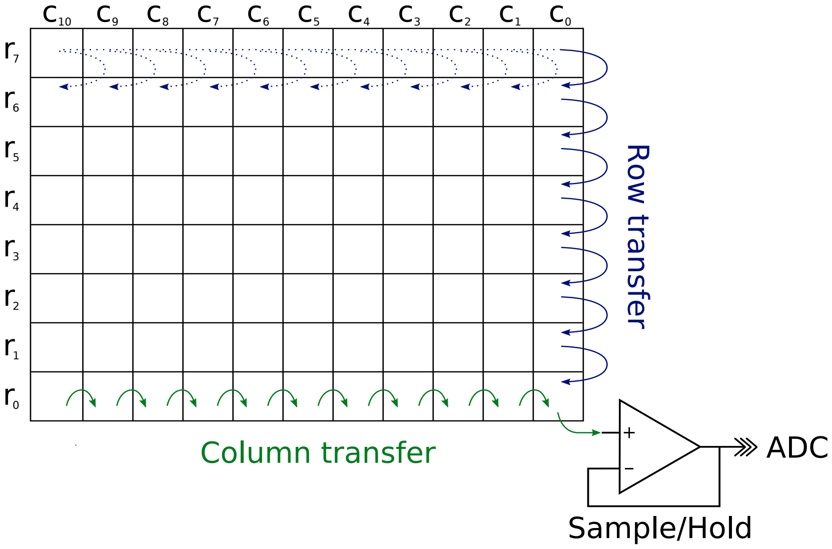


Figure 2.3: SBIG STL-11000 camera with open shutter



c_{10} c_9 c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0
Row transfer
Column transfer
+
−
ADC
Sample/Hold

Figure 2.4: Schematic drawing of a CCD

Every CCD chip has two methods of shifting pixels in the picture, the *row-* and *column-transfer*, as shown in Fig. 2.4. When a row transfer occurs, the content of r_n is replaced by the charge of r_{n+1}. The charge of r_0 is discarded and the first row (r_7) is filled with a zero charge. Much the same happens during column-transfer although here only the single row r_0 is affected: Each bin r_0, c_n is replaced by the charge of bin r_0, c_{n+1}, the charge of r_0, c_{10} is filled with a zero charge. The charge in r_0, c_0 is not discarded, but stored in a sample and hold circuit until it has been converted to a digital value by the analog-digital converter (ADC) contained in the CCD camera. This is the way the charge distribution on the CCD can be measured and the image is taken.

For an ideal CCD, row and column transfer operations would be completely lossless. They are almost lossless in a real-world device: The so-called *charge transfer efficiency* of the, for example, KAI-11002 is about 99.999% [Kod06]. Furthermore in an ideal CCD bins would only fill up when they are actually hit by photons. Real-world CCDs differ in this aspect again: Even with no photons hitting the photosensitive layer charge carriers are generated. This effect is known as the *dark current*. The dark current is an thermally induced component and becomes less at sufficiently low temperatures [Rie03]. This effect is called *dark noise* and increases exponentially over temperature and almost linear over time.

Readout and analog-to-digital conversion is not lossless, but associated with a certain error called the *bias error*. Both effects can be seen in the measurements in Fig. 2.5.

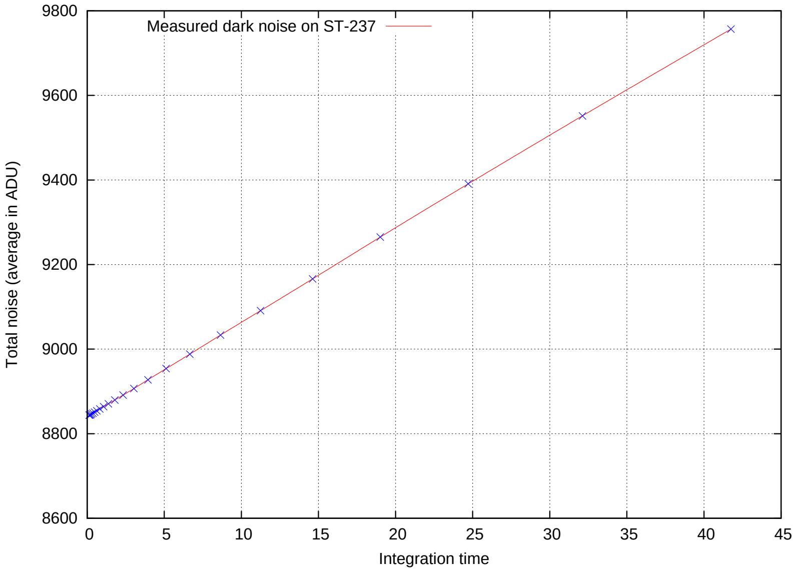


Figure 2.5: Dark current measurements on TC-237

The bias error is around 8850 ADU (Analog-Digital Units) and dark noise increases in an almost linear fashion with around 22 $\frac{\text{ADU}}{\text{s}}$.

2.3 Telescope Mountings

2.3.1 Telescope mount

The earth revolves around its own axis once approximately every 24 hours. Therefore for a stationary observer it appears as if the stars, of which we can assume their distance is infinite, rotate in a circumpolar fashion around earth. When trying to take a digital image of the stars using a CCD camera, the exposure time has to be quite long relative to the earth's rotational velocity. Would an observer therefore point his optical instrument against the sky without any further technical equipment and integrate an image over tens of minutes, the result would be a rotationally smeared image. The solution is obviously to make arrangements so the optics follow the apparently moving star in order to get a sharp image.

To achieve this goal, it is the easiest approach to mount the optics in a so-called *equatorial mount*. The equatorial mount basically ensures that the suspension of the optics are parallel to earth's axis. The two telescope axes are called *right ascension* (RA) and *declination* (DEC), where RA is usually measured in hours and ranges from 0 h to 24 h and DEC is measured in degrees and ranges from -90° to $+90^\circ$. This does not resolve the problem of having to move the optics together with the rotation of the earth – it simplifies it, however: Since the axis of the earth and the optics are parallel, the telescope's tracking only needs to move the optics in RA direction.

Another approach commonly found in telescopes suspension is the *azimuthal mount*, where one axis is orthogonal to the horizontal plane. The telescope can be moved in its azimuth (i.e., the bearing) and in its altitude. Telescopes mounted on an azimuthal mount can track objects, but always have to be guided on both axes in order to follow the movement. There is a major problem, however: When tracking an object on an azimuthally mounted telescope, the object stays in the center of the field of view – but in contrast to equatorial mount, the field of view is rotating around its center. While this is no problem for visual observation, it completely removes the ability of a telescope mounted in such a way to integrate images over long periods of time. Trailing becomes a problem at integration times of as short as 30 seconds. As telescopes mounted azimuthally are therefore unsuitable for imaging without further technical equipment (e.g, a device rotating the imaging camera together with the telescope, introducing a third axis), this work will from now on focus solely on equatorially mounted telescopes.

Even when using an equatorially mounted telescope there are still various pitfalls: One has to keep in mind that telescopes used for professional observation of the sky are large in both dimension and weight. For example the mirror alone of the 60 cm Zeiss Cassegrain Telescope in the observatory of Bamberg, Germany, has a weight of 125 kg – the counterweight of the telescope is 250 kg. The main tube has a diameter of almost 2 m and a length of 3.3 m. An image of the telescope can be seen in Fig. 2.6. The suspension which carries such a large telescope needs to be suited to move around such a great weight – therefore large gears are manufactured and used in conjunction with a transmission. Large gears have a significant disadvantage for this purpose: Manufacturing the center drilling with high precision is difficult and expensive. Therefore gears used in mounts for astrophysical equipment usually are slightly eccentric. This manifests in the core problem of guiding: The velocity with which the gear has to be moved cannot be constant, but must vary with the position of the gear in a sinusoidal manner in order to achieve constant angular velocity ω.

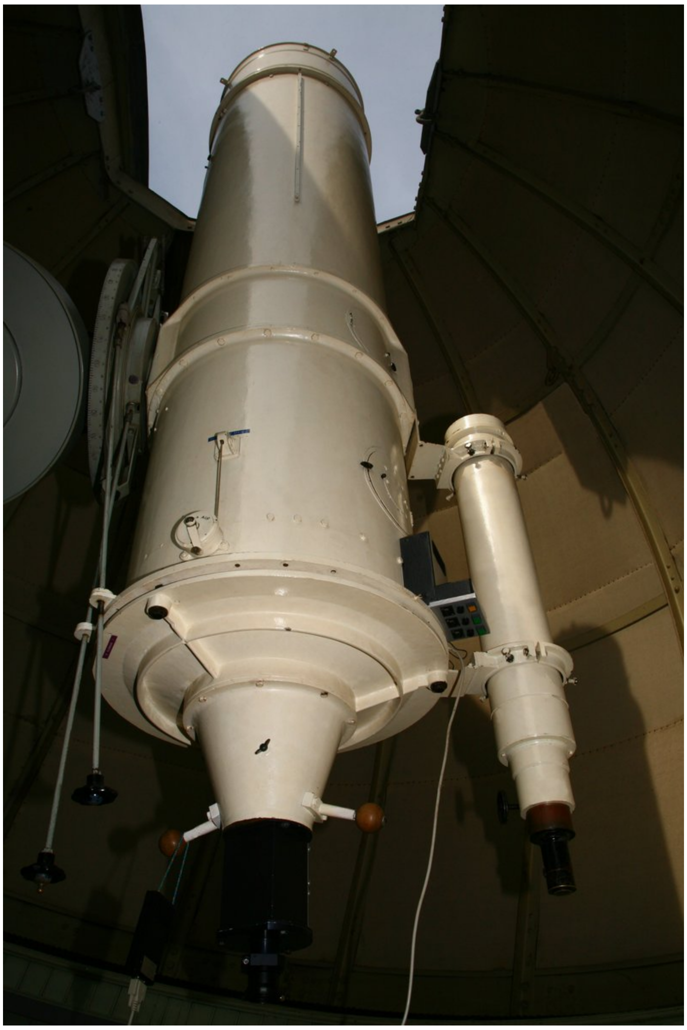


Figure 2.6: 60 cm Carl Zeiss telescope of the Remeis Observatory, Bamberg

2.3.2 Mount Gear Inaccuracy

A gear can be approximated by its ideal center point $C = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and its outer radius r. The eccentricity of an actual drilling C_e can be represented as e, which we define as

$$e = \frac{|C_e|}{r} = \frac{1}{r}\sqrt{C_{ex}^2 + C_{ey}^2}, \quad 0 \leq e < 1 \tag{2.6}$$

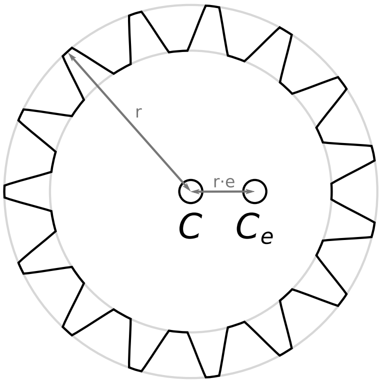


Figure 2.7: Schematic drawing of a gear, showing r and $r \cdot e$

This means a perfectly centered gear has an eccentricity coefficient of 0. As the eccentric drilling rotates around C during motion of the gear there will always be a position in which the eccentric drilling is at $\begin{pmatrix} r \cdot e \\ 0 \end{pmatrix}$.

When the gear is set in motion, the effective radius (i.e. the distance between C_e and a fixed point on the outer radius where the transmission occurs) varies in a sinusoidal manner:

$$r_{\text{eff}(\omega)} = \left| \begin{pmatrix} r\cos\omega \\ r\sin\omega \end{pmatrix} - \begin{pmatrix} r \cdot e \\ 0 \end{pmatrix} \right| = r \left| \begin{pmatrix} \cos\omega - e \\ \sin\omega \end{pmatrix} \right| \tag{2.7}$$

normalizing this by dividing by r yields die relative effective radius s (which is proportional to the transmitted gear velocity as the circumference of a circle is proportional to $2\pi r$)

$$s_{(\omega)} = \left| \begin{pmatrix} \cos\omega - e \\ \sin\omega \end{pmatrix} \right| = \sqrt{(\cos^2\omega - 2e\cos\omega + e^2) + \sin^2\omega} = \sqrt{e^2 - 2e\cos\omega + 1} \tag{2.8}$$

As can be easily guessed from Fig. 2.8 the gear eccentricity e is directly related to the maximal relative error. To verify these maxima we calculate the values at $\omega = 0$ and $\omega = \pi$:

$$s_{\max} = \sqrt{e^2 - 2e\cos\pi + 1} = \sqrt{e^2 + 2e + 1} = \sqrt{(1+e)^2} = 1 + e \tag{2.9}$$

$$s_{\min} = \sqrt{e^2 - 2e\cos 0 + 1} = \sqrt{e^2 - 2e + 1} = \sqrt{(1-e)^2} = 1 - e \tag{2.10}$$

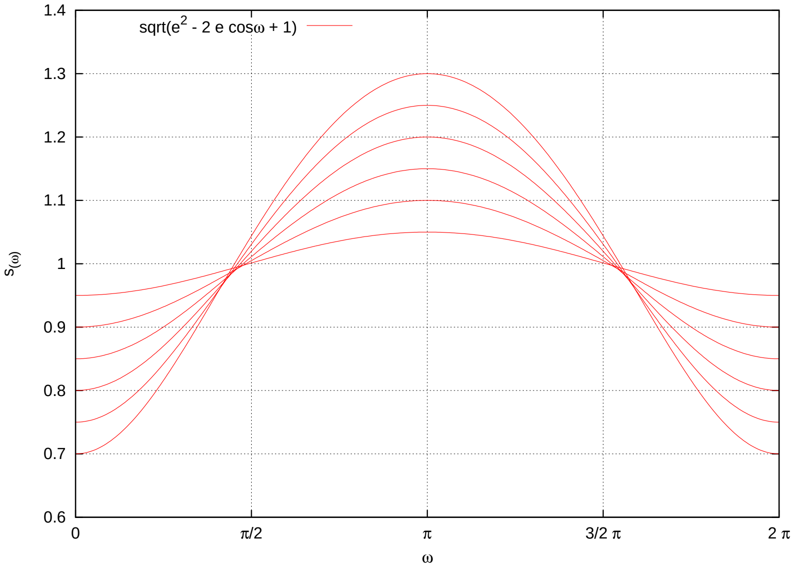


Figure 2.8: Relative radius $s_{(\omega)}$ with e ranging from 0.05 to 0.30

In any transmission there is usually more than one gear involved. When considering many cascaded gears with radii $r_0, \dots, r_n$ and their according gear eccentricities $e_0, \dots, e_n$, the total relative error S accumulates:

$$S = \prod_{i=0}^{n} \sqrt{e_i^2 - 2e_i \cos\left(\frac{\omega r_0}{r_i} + \Delta\omega_i\right) + 1} \tag{2.11}$$

where $\Delta\omega$ is the initial angular offset to the first gear which has $\Delta\omega_0 = 0$. It usually will not play a role as the ratio of radii is fractional – if it would be integral, however, there is the chance that gear eccentricities cancel each other out. In practice, this is highly unlikely. The absolute error can be gained by multiplying S with all radii.

In the worst-case scenario the maximal eccentricity would be

$$S_{\max} = \prod_{i=0}^{n} \sqrt{e_i^2 - 2e_i \cos\pi + 1} = \prod_{i=0}^{n} \sqrt{e_i^2 + 2e_i + 1} = \prod_{i=0}^{n} (1 + e_i) = \prod_{i=0}^{n} s_{\max} \tag{2.12}$$

and $S_{\min}$ analogously. However, these numbers are just upper and lower bounds for the errors – the true formulae become so complicated with even two gears that numerical calculation is more viable than a symbolic approach. For the variables used in Fig. 2.9 the true minimal and maximal total relative errors are $E_{\min} = 0.855$ and $E_{\max} \approx 1.127$.

When comparing Fig. 2.9 to the actual drift which occurs in a real system as in Fig. 2.10, the resemblance is remarkable.

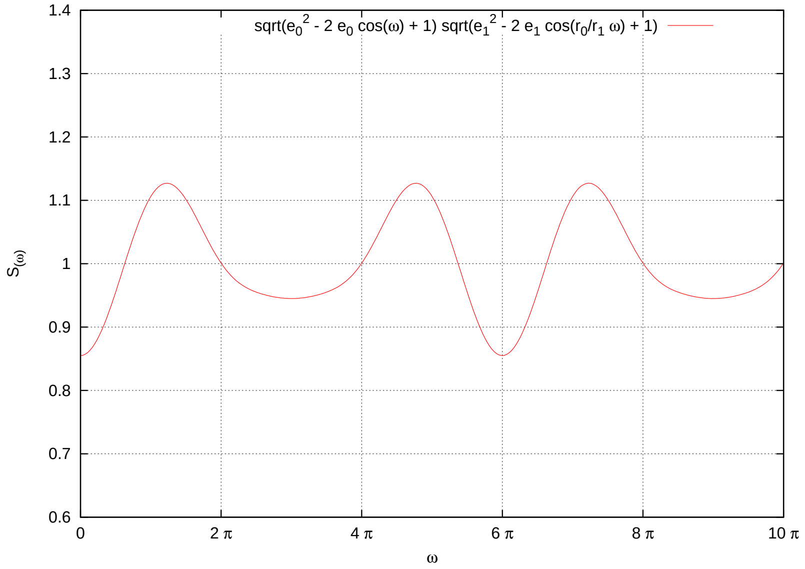


Figure 2.9: $S_{(\omega)}$ with two gears: $e_0 = 0.05, \quad e_1 = 0.1, \quad r_0 = 1, \quad r_1 = 1.1, \quad \Delta\omega_1 = 0$

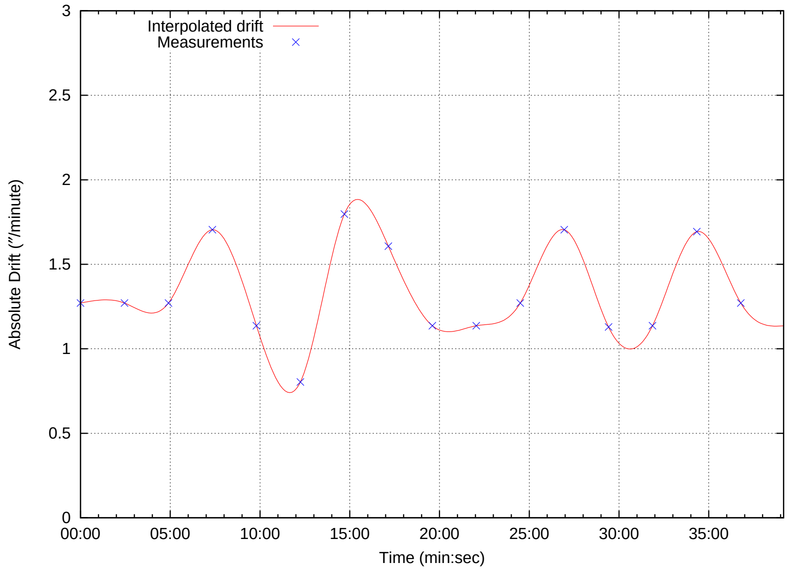


Figure 2.10: Measured total system drift of the 40 cm telescope mount in the Remeis Observatory, Bamberg

Chapter 3

Autoguiding with Astrophysical Imaging Detectors

The work which was done in order to show that the developed reverse engineering tools described in Ch. 1 function properly as described was to disassemble and reimplement a CCD camera driver together with the autoguiding procedures. The next section gives a description of how this was done.

3.1 Reverse Engineering the CCD Camera Driver

Reverse engineering is each time something new – until the assembly code has actually been reviewed, there is no way of knowing how obfuscated code is or what special tricks are used by the original developer. Therefore, this description can only be an example of what actual reverse engineering work might look like – different circumstances might call for different measures or, maybe, a completely different approach.

3.1.1 Getting to Know the Target

The target in question is the SBIG unified CCD camera driver included in the Linux SBIG SDK. The driver is available from the SBIG website in different versions and for different platforms – it is important to download all of them and keep them safe. In case of the SBIG driver there were two Linux versions, one for x86 and another for x86-64, but at the time of writing this thesis, the x86-64 driver has disappeared from their website. Although by special request, SBIG will send customers this driver via email, this still raises the concerns mentioned in the "Maintenance" argument presented in Sect. 1.1. The driver is provided both as a shared and a static library. This is useful as the static library consists of object-files which are usually broken down into semantically similar parts – an information that might be helpful when reversing the code. It is also interesting to look at what external libraries the driver links against:

```
$ ldd libsbigudrv.1.4.60.so
    linux-gate.so.1 =>  (0xb8065000)
    libusb-0.1.so.4 => /lib/libusb-0.1.so.4 (0xb8017000)
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eb4000)
    /lib/ld-linux.so.2 (0xb8066000)
```

The second thing which should be checked is which calls are used from these external libraries – they will be prominent in the shared library as unresolved symbols. In this particular example, of particular interest are:

```
$ nm -u libsbigudrv.1.4.60.so
         U memcpy@@GLIBC_2.0
         U strcmp@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
[...]
         U usb_bulk_read
         U usb_bulk_write
[...]
```

Library calls are of special interest because they cannot be obfuscated – their name needs to be present in the library in order to be resolved by the loader and their calling convention is precisely known. Especially the very low-level functions for memory comparison (like `memcpy(3)`) are often inlined by compilers – not in this case. This gives the reverse engineer the advantage of being able to produce high-level code with least effort around the regions where these calls are made.

3.1.2 Intercepting Library Calls

In order to get an impression on what the driver does, first a working example program has to be created which calls the SBIG driver library functions. In this case this has already been done by SBIG, who included an example application for use with their driver. As the ST-9XE and STL-11000 are both USB cameras, the most obvious way communication could be performed via USB bulk transfers (namely, the `usb_bulk_read` and `usb_bulk_write` calls provided by the `libusb` library). To see what the example program transmits and receives on USB, these calls now need to be trapped. For this, the special purpose tool *LibTrapper* has been developed, which is merely a code-generator: It receives a configuration file input which states what the names of functions to be intercepted are and what prototypes they have. Then it emits code which compiles to a shared object file and which defines exactly these symbols.

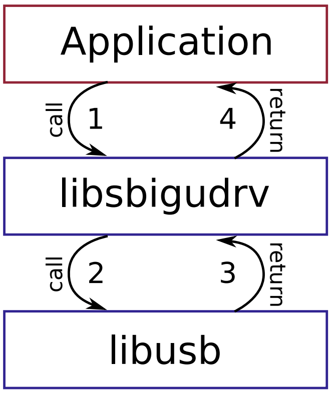


Figure 3.1: Standard driver library call

The intention is as follows: The loader can be instructed to load the generated trapper library before it loads any other libraries (via means of `LD_PRELOAD`). When symbolic names such as `usb_bulk_read` are encountered in the loader resolve stage, the order in which symbols are defined matters – preloaded libraries have precedence over ordinarily linked libraries. Therefore, the loader will resolve the library call to the function contained in trapper library. The trapper library consists of carefully designed trampolines which in their turn call `dlsym(2)` to find out where the functions are located in memory that the application actually wanted to call (via means of `RTDL_NEXT`). Then, it is designed to call a user-implemented prehook, which is called before the actual library call: This hook can modify the parameters passed to the actual library call and specify if this call should be executed. If the prehook decides not to let the actual call pass, it can simply instruct the call to return an arbitrary value. If it does call the actual library, the trapper library executes it and can afterwards execute a posthook within the trapper library: This once again can modify all parameters and can also modify the value returned to the application. This is shown graphically in Fig. 3.2 – path "a" is the one taken when the prehook decides the actual library call should not be made, path "b" is taken otherwise.

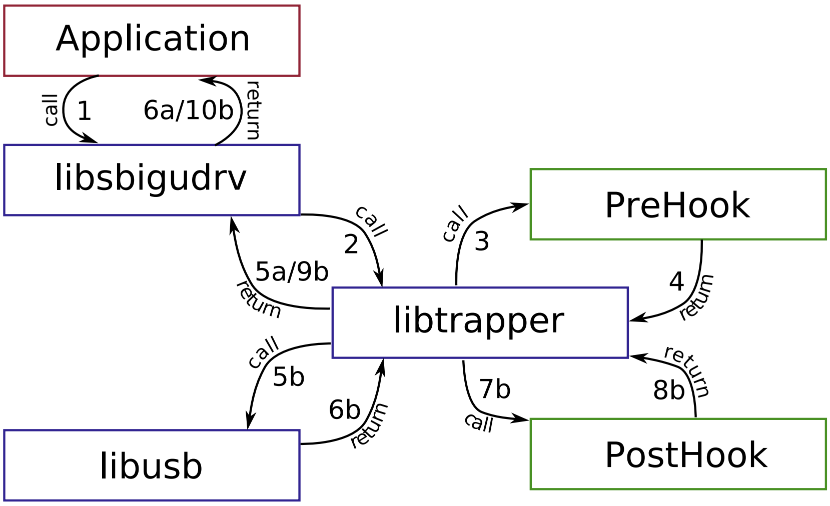


Figure 3.2: Library call intercepted and mangled by `libtrapper`

The reason for having the prehook not make the actual library call might not be so obvious: Doing that it becomes possible to emulate the camera's behavior in software. This is a real advantage: When rudimentary camera behavior is emulated only so far that the driver thinks it is talking to a real camera, the camera itself becomes obsolete for studying the driver. it is also possible to emulate camera behavior incrementally – all commands which are not implemented yet are passed to the actual camera in the prehook, all others are emulated. These are some of the methods used the gain knowledge about how the driver and error codes within the protocol work.

3.1.3 A Peek into the SBIG Protocol

One of the first things that can be done when intercepting camera communication is writing hooks which dump the communication to a file. Using the Packet Capture-format (PCAP) is (using the `libpcap` [LSW08]) just little more work than dumping directly into a file in binary mode, but it gives the reverse engineer another advantage: There are many powerful tools available freely to inspect PCAP dumpfiles, the most popular probably being Wireshark [LSW08]. For analysis of the SBIG protocol a Wireshark plugin was written. During the beginning of the reverse engineering process little to nothing is known about the protocol and its structure. However, the early dumps still remain valid as they are interpreted by the Wireshark plugin. When it occurs that the reverser finds out later that an opcode was misinterpreted or takes parameters in a different fashion, only the plugin needs to be updated and all previous recordings are then interpreted correctly. In contrast, when writing to a textfile and doing interpretation of the dump on the fly during packet capture the process of reinterpretation of those dumpfiles becomes tedious. The following is an example of a dump, showing what is transmitted to the camera and what the camera responds with:

```
→ a560
← a5620218
→ a5738089a0
← a5710318
→ a5738189a0
← a571fb18
→ a5738289a0
← a5710118
→ a5738389a0
← a5710f18
→ a5738489a0
← a5710118
→ a5738589a0
← a5710118
...
```

The dump immediately suggests that commands start out with the constant sequence `0xa5` and responses are of fixed size depending on the request. Also, the protocol operates in a question-answer manner: The driver requests something, the camera responds. For each request there is exactly one response.

3.1.4 Understanding the Protocol Semantics

A complete dump of the protocol is the premise to the engineering of a new driver, but the work is not finished there: Knowledge about what was transmitted is worth nothing when the *meaning* of the protocol parts remains hidden, as it is then not possible to easily write code that does something different than replaying the dumped commands. Therefore, more high-level information must be extracted. The SBIG library API provides only one function:

```
short SBIGUnivDrvCommand(short Command, void *Params, void *Results);
```

where the `Command` parameter is actually of type `enum PAR_COMMAND`, a list of the 60 distinct commands the driver accepts. The easiest approach is now to simply add the call to `SBIGUnivDrvCommand()` to the number of trapped libraries and also dump this information in the PCAP file which is generated during run of the application. It will be embedded as metadata, giving the reverse engineer the possibility to take a look into the call stack at the time of the data transfer. This way it can be determined quickly which API command corresponds to which driver command or commands.

3.1.5 Hardware Disassembly

After getting a general overview of what is communicated in the SBIG protocol, it can be beneficial to take a look at the used hardware. Disassembly of the SBIG STL-11000 enclosure reveals a prominent IC in a 80-pin Plastic Quad Flat Package (PQFP). Removal of the sticker covering the top the IC makes it possible to read the imprint: The controlling MCU is a Cypress EZ-USB AN2131. The Cypress EZ-USB is a unique MCU design featuring a 8051 core and an USB transceiver in a System-on-Chip (SoC) [Cyp02b]. It is the first chip generation by Cypress featuring this design and has since been superseded by the more advanced EZ-USB FX and EZ-USB FX2, also known as the CY7C464XX [Cyp01a] [Cyp00] and CY7C68XX [Cyp02a] [Cyp01b] series. Between all three series there is little difference in the basic architecture, although the capabilities of each chip vary greatly. The MCU almost completely handles the complicated USB communication, leaving a very high level abstraction in assembly. The firmware loader is present in hardware – since the firmware is loaded via USB into the SRAM of the SoC, it can be put in its original state by a simple power-cycle. The USB transceiver also has remarkable hardware-capabilities including Isochronous Transfer and Direct Memory Access (DMA).

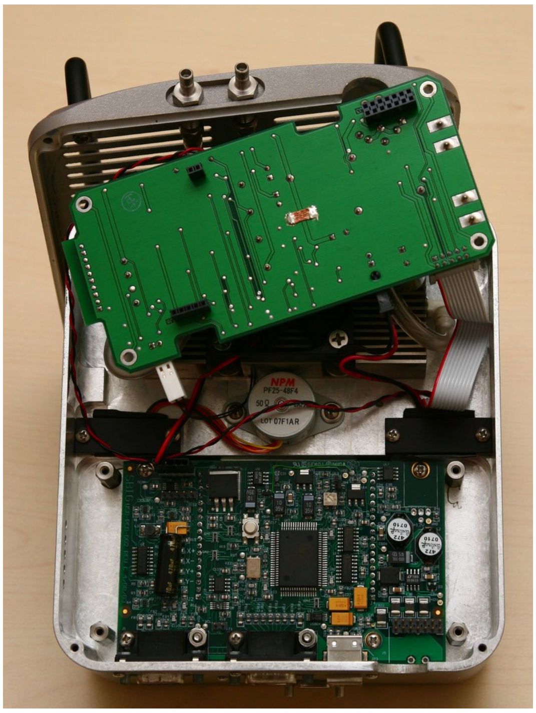


Figure 3.3: Disassembly of the STL-11000 showing the EZ-USB AN2131 in the center

For the purpose of reverse engineering, a 8051 core is ideal: There are few instructions and registers. The device has 8kB of SRAM of which the lower 128 byte are memory mapped into the address space for direct access. Special device registers (e.g., for port access) are also accessed using memory-mapped I/O in the high memory region from `0x7b00` and upwards.

3.1.6 Firmware Disassembly

After disassembling the hardware, the firmware becomes interesting because it is now possible to disassemble it meaningfully. The first thing to do is just letting the whole firmware pass through a disassembler for the 8051 architecture. Then it is most useful to take a sweep through and start out reversing in striking code regions – as it is hard to define what exactly is meant by that, here is an example:

Listing 3.1: First sweep through the disassembled SBIG ST-L firmware

```
[...]
L0106:
    MOV DPTR, #07DC1h
    MOVX A, @DPTR
    ANL A, #0F0h
    CJNE A, #0F0h, L0107
    LCALL L0108
    LJMP L0109

L0107:
    CJNE A, #0h, L0210
    LCALL L0211
    SJMP L0109

L0210:
    CJNE A, #10h, L0226
    LCALL L0227
    SJMP L0109

L0226:
    CJNE A, #20h, L0230
    LCALL L0231
    SJMP L0109
```

What happens at `L0106` is that the number `0x7dc1` is loaded in the pointer register of the MCU. Then the value from address `0x7dc1` is loaded by indirection of this pointer register into the accumulator. Only the most significant nibble of the accumulator is needed, which is why a bitwise `AND` is performed with `0xf0`. Then, a series of comparisons starts, which can be expressed by pseudocode of the form

```
msn = (*0x7dc1) & 0xf0
if (msn == 0xf0) l0108();
    else if (msn == 0x00) l0211();
    else if (msn == 0x10) l0227();
    else if (msn == 0x20) l0231();
[...]
```

Revealing the meaning of this code is easy when taking a look in the MCU handbook [Cyp02b]. When looking at the RAM table, the area form `0x7dc0` to `0x7dff` is a USB buffer for endpoint 2. Therefore the nibble that is checked is that of the second byte of the buffer (i.e., `buf[1]`).

Keeping in mind what the protocol looked like when taking a short peek into it, the meaning is quite obvious: The code in question is processing the received buffer. Is it checking which command code was issued by the host.

When performing the disassembly for embedded devices a trivial feature which has a great effect is automatic resolving of special registers. This means the disassembler is told what names special regions in the memory map have. The user can override the architectural names to give them more meaning. In this particular example, by disassembly of the SBIG binary it was found that the memory region from 0x1b4b onwards was used for sending USB responses and the memory region from 0x7dc0 was used to receive USB requests – from the EZ-USB architecture, those two would have been called EP07OUT[11] and EP02OUT respectively. As per user request, however, they will simply be called sendbuf and recvbuf, which is far more intuitive for the reverse engineer.

It also is possible to rename all labels as per user request, so that they actually have meaning. To aid the reverse engineers work, it is useful to enable certain code transformations: Due to hardware restrictions, memory indirections above 0x80 always are performed indirectly by use of the dptr register. When labeling all the above memory to the discoveries found and enabling code transformations which improve readability and were performed in a fully automatic manner, the code becomes much clearer:

Listing 3.2: Disassembled SBIG ST-L firmware after code transformations

```
interpret-recvd-data:
    a = ([recvbuf + 1])
    a &= 0xf0
    cjne a, 0xf0, 10107
    call interpret-recvd-data-0xf_
    goto interpret-recvd-data-finished

10107:
    cjne a, 0x0, 10210
    call interpret-recvd-data-0x0_
    goto interpret-recvd-data-finished
[...]
```

While reverse engineering more and more code, some other interesting examples can be encountered:

Listing 3.3: Disassembly of high-level arithmetic routines on the 8051

```
10116:
    a = r7
    b = r5
    mul ab
    r0 = b
    xch a, r7
    b = r4
    mul ab
    add a, r0
    xch a, r6
    b = r5
    mul ab
    add a, r6
    r6 = a
ret
```

For which the constraint-based disassembler gives the following result as a postcondition of the MBB:

```
R6    (((r6 * r5) & 0xff) + (((r7 * r4) & 0xff) + (((r7 * r5) & 0xff00) >> 8)))
R7    ((r7 * r5) & 0xff)
```

While this might look strange at the first look, it becomes apparent when thinking about how integer multiplication works. When multiplying two numbers, A and B, both of word width w, the result is an integer of length $2 \cdot w$. This result cannot be represented in one register as it exceeds the machine word width. Therefore, it it split up into two registers, $(A \cdot B)_{\mathrm{H}}$ and $(A \cdot B)_{\mathrm{L}}$, where

$$(A \cdot B) = w \cdot (A \cdot B)_{\mathrm{H}} + (A \cdot B)_{\mathrm{L}} \tag{3.1}$$

Now consider two double word integers x, y at word width w. Let

$$x = A \cdot w^1 + B \cdot w^0, \quad y = C \cdot w^1 + D \cdot w^0 \tag{3.2}$$

The multiplication $x \cdot y$ is

$$x = (A \cdot C) \cdot w^2 + (A \cdot D + C \cdot B) \cdot w + (B \cdot D) \tag{3.3}$$

As each of these multiplications can overflow, substitution as in Eq. 3.1 yields

$$\begin{aligned} x = ((AC)_{\mathrm{H}} \cdot w^2 + ((AD)_{\mathrm{H}} + (CB)_{\mathrm{H}}) \cdot w + (BD)_{\mathrm{H}}) \cdot w^1 \\ +((AC)_{\mathrm{L}} \cdot w^2 + ((AD)_{\mathrm{L}} + (CB)_{\mathrm{L}}) \cdot w + (BD)_{\mathrm{L}}) \cdot w^0 \end{aligned} \tag{3.4}$$

and simplification then

$$\begin{aligned} x = (AC)_{\mathrm{H}} \cdot w^3 \\ +((AC)_{\mathrm{L}} + (AD)_{\mathrm{H}} + (CB)_{\mathrm{H}}) \cdot w^2 \\ +((AD)_{\mathrm{L}} + (CB)_{\mathrm{L}} + (BD)_{\mathrm{H}}) \cdot w^1 \\ +(BD)_{\mathrm{L}} \cdot w^0 \end{aligned} \tag{3.5}$$

Knowing this and again taking a close look at the above assembly code, that reads:

$$\begin{aligned} r6 = (r6 \cdot r5)_{\mathrm{L}} + (r7 \cdot r4)_{\mathrm{L}} + (r7 \cdot r5)_{\mathrm{H}} \\ r7 = (r7 \cdot r5)_{\mathrm{L}} \end{aligned} \tag{3.6}$$

This means the function in questions performs a multiplication of two 16-bit integers, resulting in the 16 least significant bits of the multiplication. The upper 16 bit are discarded.

Another interesting part appears during disassembly of the response routine:

Listing 3.4: Disassembly of the send command routine

```
send-cmd-((0x16), ([sendbuf + 2])...):
    (0x52) = 0x0
    (0x33) = 0x0
    dptr = [sendbuf + 0]
    (0x31) = dph
    (0x32) = dpl
    a := @dptr = 0xa5
    dptr++
; assume((0x16) := ResponseID
    a := @dptr = (0x16)
    a &= 0xf
    add a, 0x3
    a &= 0xfe
    rr a
    (0x2d) = a
    (0x2c) = 0x0
ret
```

This is the command which does the final preparation of the send buffer right before it is sent out the USB port. The packet has already been initialized starting from the byte with offset 2 and the response ID has been put at memory address `0x16`. The disassembler calculates the value of the accumulator register `a` in the postcondition:

```
a = ROR(((ResponseID & 0x0f) + 3) & 0xfe)
```

and simplifies it to

```
a = ((ResponseID & 0x0f) + 3) / 2
```

This might look strange, as instead of a regular logical shift right instruction the combination of the bitwise `AND` with `0xfe` is made followed by a rotate right opcode. The reason is simply that the 8051 MCU core is lacking a `SHR` opcode. In practice, however, the `SHR` operation is required far more often than the `ROR` opcode – yet this is not surprising, but a clever design decision: Expressing a `SHR` by using `ROR` requires two instructions, as shown above. The other way around, however, would cause four instructions and require a byte of memory:

```
SHR A, 1
CLR (0x20)
MOV C, (0x20.7)
ORL A, (0x20)
```

By using these techniques of which only selected examples were presented it was possible to reverse engineer the whole SBIG CCD camera protocol. The complete results of this work are summarized in the Appendix B.

3.2 Forms of Object Tracking

Guiding the telescope is possible in different ways, ranging from hardware-only solutions to software-only solutions – of greatest interest are methods which are cost-effective and are easy to setup for the user: Those are mostly hardware/software-combinations. Some possible ways of solving the tracking problem will be discussed in the next few sections.

3.2.1 Off-Axis Guiding

The most popular form of autoguiding is the *off-axis guiding*. In this method a second CCD chip (the guiding chip) is fixed within the CCD camera itself next to the imaging CCD (i.e., off the optical axis). The optical instrument in front of the CCD is obviously the same for both imaging and guiding, being both an advantage and disadvantage at the same time: As can be seen in Eq. 3.9 using this technique the focal length cancels out and the achievable accuracy is only determined by the ratio of pixel sizes of the imaging- and guiding CCD. As the imaging CCD's pixels are usually designed to be smaller than the imaging CCD's pixels, the achievable performance is outstanding. As both CCDs are attached to the PCB in a rigid manner, the distance between them stays the same during guiding. This means any inadvertent movement or change of the telescopes image (for example thermal expansion of the mirror) affects both CCDs in the same manner. However, these advantages come at a price: Small pixels in the guiding CCD mean that the total dimension of the guiding CCD is also quite small – in practice it is often difficult to find an object illuminating the guiding CCD as the field of view is extremely small. Using a TC-237 guiding sensor on, for example, a telescope with 4 m focal length gives a field of view of around $4' \times 3'$ (see table 3.1). Not all objects that are selected for imaging have appropriately bright guiding stars in their direct proximity which could be used for off-axis guiding. If one can be found, however, off-axis guiding is by far the most effective solution to the guiding problem.

3.2.2 Auxiliary Telescope Guiding

When off-axis guiding is impractical or undesirable, a remote guiding head can be used to guide the main telescope. This guiding head will then be attached to the auxiliary telescope, a second telescope which is rigidly attached to the main tube. As both telescopes view the same direction, errors due to guiding impact the images generated by both telescopes in the same manner. There are some pitfalls, however: Some mirrors tend to move slightly when the position of the tube changes. If this happens on either the imaging or guiding device the images of the running exposure must be discarded: The movement offset will clearly be visible. Having a second telescope also increases the necessary overhead that has to be made before starting any observation. Both devices need to have precise focus in order to get good image quality. The main reason why auxiliary telescope guiding can be a great advantage over off-axis guiding is that the focal lengths of both instruments can be different – and generally are. The auxiliary device usually has a focal length at least 10 times smaller than the main telescope has. That way, the field of view is much larger and it will almost always be possible to find an object bright enough for guiding purposes.

3.2.3 Software Virtual Guiding

The gear inaccuracy described in Sect. 2.3.2 is not constant over time as gears rotate. This means that with sufficiently small integration times the guiding error of some images will be small compared to other systemic influences like seeing. As Berry et at. describe in [BB06], with a standard 360-tooth gear the worm driving it will rotate once in 4 minutes. When integrating pictures of 60 seconds per exposure, this means that without guiding two out of four images will be reasonably well-tracked while the other two will be slightly trailed. When imaging the same object over and over again, the trailed pictures can be discarded. The good images can then be cross-correlated against each other in order to determine their shift. In software it is then possible to add the shift-corrected images together, yielding a higher signal to noise ratio. However each readout will have its own readout noise which adds to the degradation of the summed image. Therefore it is usually best to make exposures as long as possible (limited by the equipment precision) and only resort to software virtual guiding if nothing else is available.

3.3 Guiding Accuracy

For the imaging device itself the width and height of the pixel array (number of pixels), n_w and n_h, and the width of height of each pixel, d_w and d_h, is of interest. Not all imaging devices provide square pixels, but the ones used in this work do. It has to be kept in mind that if the imaging device has rectangular shaped pixels the resolution in x and y direction is different.

Table 3.1: Different devices and their characteristics when behind optics

Device	d	r in $\frac{''}{\text{px}}$	f_w	f_h
$f = 48\,\text{cm}$				
ST-9	$9\,\mu$m	3.87	$33'0''$	$33'0''$
STL-11K	$9\,\mu$m	3.87	$4°0'4''$	$2°0'2''$
TC-237	$7.4\,\mu$m	3.18	$34'49''$	$26'14''$
$f = 400\,\text{cm}$				
ST-9	$9\,\mu$m	0.46	$3'58''$	$3'58''$
STL-11K	$9\,\mu$m	0.46	$31'0''$	$20'40''$
TC-237	$7.4\,\mu$m	0.38	$4'11''$	$3'9''$

Table 3.2: Maximal guiding performance for focal widths on main/guide optical instruments

f_G \ f_I	48 cm	400 cm	1000 cm
48 cm	0.82 px	6.85 px	17.13 px
400 cm	0.10 px	0.82 px	2.06 px
1000 cm	0.04 px	0.33 px	0.82 px

The resolution a imaging device provides is usually given in arcseconds per pixel and can easily be calculated by solving Eq. 2.3 for u and inserting $s = d_x$ into the equation:

$$r_x = \frac{d_x}{f} \tag{3.7}$$

When using auxiliary telescope guiding, as described above, two optical instruments are used: The first captures the actual object and is solely used for imaging while the second captures nearby stars and is used for guiding the main instrument. The focal lengths of the instruments will be called f_I and f_G. They are usually not equal. Assuming the smallest possible change detectable by the guiding sensor is ±1px and under the assumption of square pixels on both the guiding and imaging device, according to Eq. 2.3 a angular guiding error results,

$$u_\text{err} = \pm\frac{d_\text{G}}{f_\text{G}} \tag{3.8}$$

which then results in a pixel error on the imaging sensor of

$$n_\text{err} = \pm\frac{f_\text{I} u_\text{err}}{d_\text{I}} = \pm\frac{d_\text{G} f_\text{I}}{d_\text{I} f_\text{G}} \tag{3.9}$$

When inserting realistic values for the Bamberg instrumentation $f_\text{I} = 400\,\text{cm}$, $f_\text{G} = 48\,\text{cm}$, $d_\text{I} = 9\,\mu\text{m}$ and $d_\text{G} = 7.4\,\mu\text{m}$, the minimal achievable pixel error on the imaging sensor is

$$n_\text{err} = \pm\frac{7.4\,\mu\text{m} \cdot 400\,\text{cm}}{9\,\mu\text{m} \cdot 48\,\text{cm}} \approx \pm 7\,\text{px} \tag{3.10}$$

Considering that both the telescope diffraction and the guiding precision are far smaller than the weather conditions in middle Europe permit it can be concluded that guiding with these parameters is generally possible and brings a significant improvement in image quality over long exposure times. The remaining factor dominating image quality is seeing, i.e., turbulence in Earth's atmosphere causing objects to appear flickering.

3.4 Cross Correlation

The telescope slew velocity is variable over time as described in Sect. 2.3.2. Therefore the most commonly used approach is to first make one reference image with the guiding CCD sensor. Then, integration of the main image is started. During that same time, the guiding sensor integrates images of the same object over and over again. These pictures are then cross-correlated against the reference image in order to determine the shift. The shift can then be corrected in a negative feedback loop by changing the slew of the telescope mount's motors accordingly.

3.4.1 Efficient Implementation

Since for determination of the movement shift guiding images have to be calculated frequently, cross correlation must necessarily be implemented efficiently. The discrete cross correlation formula for two images A and B with according geometries $W_A \times H_A$ and $W_B \times H_B$ is [Rus98]

$$(A \star B)_{(i,j)} = C_{(i,j)} = \sum_{x=0}^{W_A-1} \sum_{y=0}^{H_A-1} A_{(x,y)} \cdot B_{(x-i,y-j)} \tag{3.11}$$

where $0 \leq i < W_A + W_B - 1$ and $0 \leq j < H_A + H_B - 1$. The complexity of this spatial domain cross correlation algorithm is $O((W_A + W_B) \cdot (H_A + H_B) \cdot W_A \cdot H_A)$. This can be be simplified if we assume that the dimensions of the two images A and B are equal:

$$2W \cdot 2H \cdot W \cdot H = 4(WH)^2 \to O(n^2) \tag{3.12}$$

where n is a measure for the number of pixels in each image. This performance is quite poor, but can be improved by using the equivalent algorithm in the frequency domain. The convolution theorem is applied and yields:

$$\widetilde{(A \star B)}_{(x)} = \tilde{A}_{(x)} \cdot \overline{\tilde{B}_{(x)}} \tag{3.13}$$

Here, $\tilde{X}$ indicates the discrete Fourier transform of X while $\overline{X}$ is used as a notation for the complex conjugate of X. This immediately gives the required result:

$$(A \star B) = C = \text{IDFT}(\tilde{A} \cdot \overline{\tilde{B}}) \tag{3.14}$$

The algorithm which performs a DFT or IDFT can be implemented in $O(n \log n)$ using the fast Fourier transformation (FFT) algorithm originally introduced by Cooley and Turkey [CT65] [FJ05]. As the complexity of a multiplication in frequency space is of magnitude $O(n)$, the total algorithmic computational complexity is $O(n \log n)$.

For finding the actual shift the cross correlation image has to be analyzed. By simply taking the position in the image where the magnitude is greatest, the shift can be determined. When the cross correlation has been calculated without zero padding, it must be noted that because of the Nyquist sample theorem only movement over half the height and width can be detected: Any motion that moves further will be falsely detected as motion in the opposite direction. This is due to the fact that the Fourier transform actually analyzes a function which is periodically continued beyond the actual limits of the image. Both images used for the cross-correlation are mathematically tiled to infinite size – any feature which moves more than half an image dimension could therefore also have moved a shorter distance in opposite direction. This can nicely be seen when the elliptic feature of Fig. 3.4a is moved so that the result is Fig. 3.4b. In the cross-correlation Fig. 3.5 the detected movement is highlighted by the strait line while the actual movement (dotted line) remains undetected.

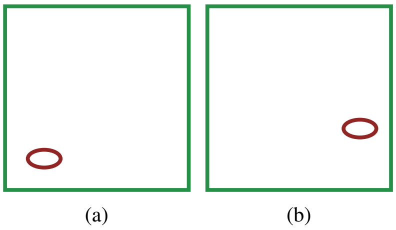


Figure 3.4: (a) shows the original image, (b) with the feature shifted

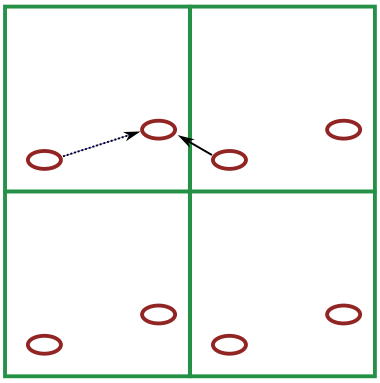


Figure 3.5: Infinite continuation of images in overlay

3.4.2 Avoiding False Positive Detection by Normalization

It may prove difficult to determine the actual shift from the resulting values as one of the main problems is the locationally invariant noise (see Sect. 2.2) which covers the complete background of the picture. As this noise is not random, but due to the fact that certain pixels simply have a higher sensitivity than others, the noise of two images is correlated. This will contribute a substantial part to the shift vector $(0,0)$. This vector cannot simply be ignored either because it often will be the case that the image actually has not moved at all. Similar problems are also discussed by Lewis in [Lew95] where it is mentioned that the classical cross-correlation may fail under certain circumstances –

the alternative is the *normalized* cross-correlation function:

$$\gamma_{(i,j)} = \frac{\sum\left((A_{(x,y)} - \hat{A}_{(i,j)})(B_{(x-i,y-j)} - \hat{B})\right)}{\sqrt{\sum(A_{(x,y)} - \hat{A}_{(i,j)})^2 \cdot \sum(B_{(x-i,y-j)} - \hat{B})^2}} \tag{3.15}$$

where each of the sums is actually the implicit double sum ranging over B, i.e.:

$$\sum \mathrm{Expr} = \sum_{x=i}^{i+W_B-1} \sum_{y=j}^{j+H_B-1} \mathrm{Expr} \tag{3.16}$$

This term can thus be written differently:

$$\gamma_{(i,j)} = \frac{\sum_B\left((A_{(x+i,y+j)} - \hat{A}_{(i,j)})(B_{(x,y)} - \hat{B})\right)}{\sqrt{\sum_B(A_{(x+i,y+j)} - \hat{A}_{(i,j)})^2 \cdot \sum_B(B_{(x,y)} - \hat{B})^2}} \tag{3.17}$$

This normalized cross-correlation function will yield significantly better results. Unfortunately, however, it is a much more expensive operation than the naïve cross-correlation implementation, but thankfully can be implemented efficiently when using the algorithm presented by [Lew95]. This is done by first building *summed-area tables* (SAT) over A, in a way so that

$$s^k_{P(x,y)} = \begin{cases} P^k_{(x,y)} + s^k_{P(x-1,y)} + s^k_{P(x,y-1)} - s^k_{P(x-1,y-1)} & \text{if } (x \geq 0) \wedge (y \geq 0) \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

By convention, the exponent k of the summed-area table s^k_P can be omitted if $k = 1$, i.e.:

$$s^1_P = s_P \tag{3.19}$$

Note that for the general case

$$s^k_{P(x,y)} \neq (s_{P(x,y)})^k \tag{3.20}$$

Summed-area tables are an old concept [Cro84] which will be immensely helpful in efficiently solving the normalized cross correlation function. Each pixel in the SAT is the sum over all pixels to the top and left of it to the power of k, including itself:

$$s^k_{P(i,j)} = \sum_{x=0}^{i} \sum_{y=0}^{j} (P_{(x,y)})^k \tag{3.21}$$

Calculation of the SAT is of order $O(n)$, as the operation in Eq. 3.18 can be performed efficiently using the algorithm in List. 1. Once calculated, the sum over any arbitrary rectangular region of A with size w, h and offset i, j can be retrieved in $O(1)$ using the formula:

$$\sum_{x=i}^{i+w-1} \sum_{y=j}^{j+h-1} P^k_{(x,y)} = s^k_{P(i-1,j-1)} + s^k_{P(i+w-1,j+h-1)} - s^k_{P(i+w-1,j-1)} - s^k_{P(i-1,j+h-1)} \tag{3.22}$$

To prove this is true one needs only to plug the basic premise Eq. 3.21 into Eq. 3.22:

$$\begin{aligned}
&\sum_{x=0}^{i-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \sum_{x=0}^{i+w-1}\sum_{y=0}^{j+h-1} P^k_{(x,y)} - \sum_{x=0}^{i+w-1}\sum_{y=0}^{j-1} P^k_{(x,y)} - \sum_{x=0}^{i-1}\sum_{y=0}^{j+h-1} P^k_{(x,y)} = \\
&= \sum_{x=0}^{i-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \left(\sum_{x=0}^{i+w-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \sum_{x=0}^{i+w-1}\sum_{y=j}^{j+h-1} P_{(x,y)^k} \right) - \sum_{x=0}^{i+w-1}\sum_{y=0}^{j-1} P^k_{(x,y)} - \sum_{x=0}^{i-1}\sum_{y=0}^{j+h-1} P^k_{(x,y)} = \\
&= \sum_{x=0}^{i-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \sum_{x=0}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} - \sum_{x=0}^{i-1}\sum_{y=0}^{j+h-1} P^k_{(x,y)} = \\
&= \sum_{x=0}^{i-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \sum_{x=0}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} - \left(\sum_{x=0}^{i-1}\sum_{y=0}^{j-1} P^k_{(x,y)} + \sum_{x=0}^{i-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} \right) = \\
&= \sum_{x=0}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} - \sum_{x=0}^{i-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} = \\
&= \left(\sum_{x=0}^{i-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} + \sum_{x=i}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} \right) - \sum_{x=0}^{i-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} = \\
&= \sum_{x=i}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} \qquad (3.23)
\end{aligned}$$

Efficient calculation of such an integral over the area $[(i,j),(i+w-1,j+h-1)]$ using a previously calculated SAT will be notated using:

$$\sum_{x=i}^{i+w-1}\sum_{y=j}^{j+h-1} P^k_{(x,y)} = \int_{i,j}^{w,h} P^k \qquad (3.24)$$

Algorithm 1 `GetSATk`: Precalculate the summed-area table s_P^k over P

```
$sPk[0,0] := P[0,0]^k$
for $1 \le x < W_P$ do
    sPk[x, 0] := $P[x,0]^k + sPk[x-1,0]$
end for
for $1 \le y < H_P$ do
    sPk[0, y] := $P[0,y]^k + sPk[0,y-1]$
end for
for $1 \le x < W_P$ do
    for $1 \le y < H_P$ do
        sPk[x, y] := $A[x,y]^k + sPk[x-1,y] + sPk[x,y-1] - sPk[x-1,y-1]$
    end for
end for
```

As Lewis explains, the numerator of the normalized cross-correlation function can be calculated efficiently using the FFT convolution of $A-\hat{A}$ with $B-\hat{B}$. The problematic portion of the denominator, ν, is

$$\nu = \sum_B (A_{(x+i,y+j)} - \hat{A}_{(i,j)})^2 = \sum_B (A_{(x+i,y+j)} - \eta)^2 \qquad (3.25)$$

Calculation of ν can first be simplified by using the summed-area table in Eq. 3.18 in order to efficiently calculate η, the local average of A under i, j:

$$\eta = \frac{1}{W_B \cdot H_B}\left(s_{A(i-1,j-1)} + s_{A(i+W_B-1,j+H_B-1)} - s_{A(i+W_B-1,j-1)} - s_{A(i-1,j+H_B-1)}\right) \tag{3.26}$$

The remaining portion of ν can be expressed as follows:

$$\begin{aligned} \nu = \sum_B (A_{(x+i,y+j)} - \eta)^2 &= \sum_B (A^2_{(x+i,y+j)} - 2\eta A_{(x+i,y+j)} + \eta^2) \\ &= \sum_B A^2_{(x+i,y+j)} - 2\eta \sum_B A_{(x+i,y+j)} + \sum_B \eta^2 \end{aligned} \tag{3.27}$$

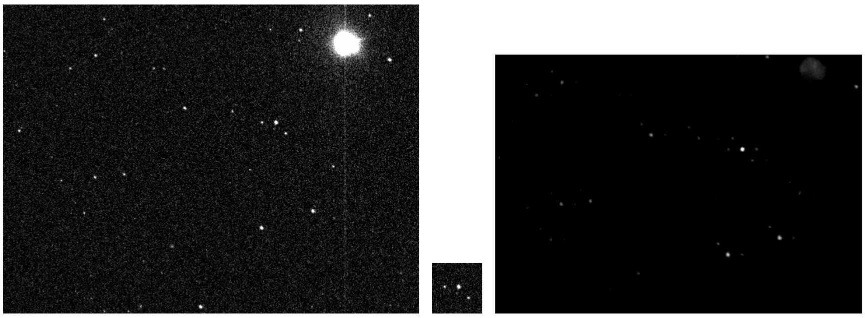
The two first sums can then again be represented using the previously calculated summed-area tables s_A and s_A^2. The third sum is invariant of x, y and can be expressed directly:

$$\nu = \left(\int_{i,j}^{W_B,H_B} A^2\right) - 2\eta\left(\int_{i,j}^{W_B,H_B} A\right) + W_B H_B \eta^2 \tag{3.28}$$

The remaining denominator sum

$$\sigma_B = \sum_B (B_{(x,y)} - \hat{B})^2 \tag{3.29}$$

is invariant over the shift i, j and must therefore only be calculated once.



(a) (b) (c)

Figure 3.6: (a) shows the image and (b) the feature which it is cross correlated against, (c) the normalized cross correlation function

3.4.3 Subpixel Accuracy

Using the approach presented in Sect. 3.4.2 the accuracy gained by cross correlation is ±1px. It might be desirable to improve that resolution to subpixel accuracy. The naïve approach is to simply upsample the images A, B to A', B' so that

$$W_{X'} = \kappa W_X, \quad H_{X'} = \kappa H_X \tag{3.30}$$

and then cross correlating A' against B'. The achieved minimal accuracy

$$\tau = \frac{1}{\kappa} \tag{3.31}$$

Algorithm 2 `GetInt`: Retrieve the integrated area over P^k using its summed-area table *sPk*

```
Require: Summed-area table $sPk$, offsets $x, y$ and integral dimensions $w, h$
  if $x \neq 0 \wedge y \neq 0$ then
    $A := sPk[x-1, y-1]$
  else
    $A := 0$
  end if
  if $y \neq 0$ then
    $B := sPk[x+w-1, y-1]$
  else
    $B := 0$
  end if
  $C := sPk[x+w-1, y+h-1]$
  if $x \neq 0$ then
    $D := sPk[x-1, y+h-1]$
  else
    $D := 0$
  end if
  return $A + C - B - D$
```

Algorithm 3 Calculation of the normalized cross correlation function γ

```
Require: $W_A > W_B \wedge H_A > H_B$
  $\text{PadX} := (2 \cdot W_A) - 1$
  $\text{PadY} := (2 \cdot H_A) - 1$
  $A_{\text{Pad}} := \text{PadImage}(A - A.Avg(), PadX, PadY)$
  $B_{\text{Pad}} := \text{PadImage}(B - B.Avg(), PadX, PadY)$
  $A_{\text{Freq}} := \text{DFT}(A_{\text{Pad}})$
  $B_{\text{Freq}} := \text{DFT}(B_{\text{Pad}})$
  $A_{\text{Freq}} *= \overline{B_{\text{Freq}}}$
  $Conv := \text{IDFT}(A_{\text{Freq}})$
  $sA1 := \text{GetSATk}(A, 1)$
  $sA2 := \text{GetSATk}(A, 2)$
  $SigmaB := B.\text{SqrSum}() - 2 \cdot B.Avg() \cdot B.\text{Sum}() + W_B \cdot H_B \cdot B.\text{Avg}()^2$
  $ASqr := A.\text{SqrSum}()$
  for $0 \leq i < W_A - W_B$ do
    for $0 \leq j < H_A - H_B$ do
      $\eta := \text{GetInt}(sA1, i, j, W_B, H_B) / W_B / H_B$
      $Numerator := Conv[(i - (W_A/2) + (W_B/2)) \mod PadX, (j - (H_A/2) + (H_B/2)) \mod PadY]$
      $\nu := \text{GetInt}(sA2, i, j, W_B, H_B) - (2 \cdot \eta \cdot \text{GetInt}(sA1, i, j, W_B, H_B) + (W_B \cdot H_B \cdot \eta^2)$
      $Denominator := \sqrt{\nu \cdot SigmaB}$
      $\gamma[i, j] := Numerator / Denominator$
    end for
  end for
```

However, as Guizar-Sicairos et al. note [GSTF08], this is in practice impossible to calculate for large κ, as the complexity of the upsampled cross correlation operation is $O(\kappa^2 n \log \kappa^2 n)$. These authors instead present an algorithm which performs dynamic upsampling during the computation of the Fourier transformation which is only insignificantly slower than the basic $\kappa = 1$ version.

Other approaches include the fitting of a Gaussian function onto the cross correlated image in order to determine the interpolated peak [NDT04]. Such an approach is much faster than the dynamic upsampling variant, but also less accurate [GSTF08].

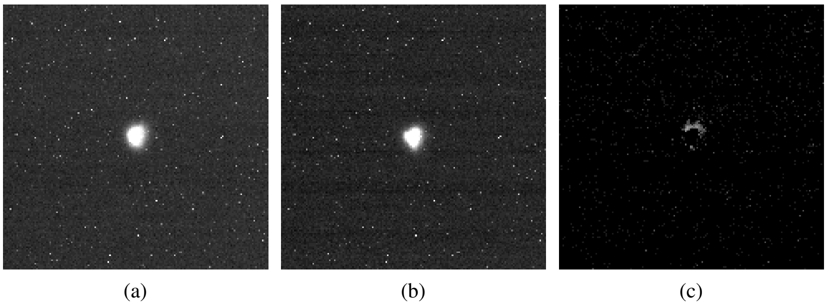

(a) (b) (c)

Figure 3.7: (a) and (b) show two consecutive images, (c) the absolute difference

Although guiding with subpixel accuracy if often advertised for commercial applications, it is goal which is difficult to achieve: Fig. 3.7 shows two images which were taken with the SBIG TC-237 remote guide head behind a $f = 48\,\mathrm{cm}$ auxiliary guiding telescope on an observation night with bad seeing. They were integrated for 1 second each and taken right after each other. The difference image shown in 3.7c shows the catastrophic extent of the seeing influence. In the star central region, no change is visible (hence it is black). Regions around the star, which are changed by seeing, are smeared up to 5 pixels, the average being around 1 pixel. On the instruments used this corresponds to

$$u = \frac{1\,\mathrm{px} \cdot 7.4\mu\mathrm{mpx}^{-1}}{480\,\mathrm{mm}} \mathrm{rad} \approx 3.2'' \tag{3.32}$$

Given the fact that the accuracy of the instruments is heavily dominated by seeing makes using a subpixel-accuracy guider an unrealistic endeavour for many observations in Central Europe. This is why subpixel determination has not been implemented during the reimplementation of the CCD software.

3.5 Determining the movement field

In order to determine how far and in which direction the telescope has to be moved to achieve a certain pixel shift, the software has to be calibrated first. Most telescope mounts provide only very simple means of controlling the slew velocity: An external interface is supplied at which four relays can be connected. These relays then change the slew in one of the four possible directions. Two of these have influence of the right ascension direction while the other two influence slew on the declination axis. As the camera is connected at an angle φ to the telescope, the change in the image by the CCD camera caused by slew is also seen under this angle φ. Another aspect making the determination of the movement field more difficult are the nonlinearities of the motors.

They have a phase in which they accelerate after the slew change was initiated, a phase in which their velocity is constant, and a phase in which they decelerate after the slew change was stopped. Due to hysteresis caused by the motor mechanics the motors also tend remain in their position for a certain time when the direction is reversed.

The first step in the calibration process is to find a suitable guide star and setting up the focus. After this has been done, the guiding star should be centered on the guiding chip manually to give the greatest possible movement ability in all directions. Then the software calibration process is started. The calibration routine integrates a picture of the guiding star for use as reference and then moves the telescope in a specific direction for a different time periods. After each movement another image is integrated and the feature (i.e., the guiding star) is cross correlated against the reference image. This is performed for all four directions, so that a movement field as seen in Fig. 3.8 is determined.

The axis designation is relatively arbitrary: Any axis can be chosen to be the X+ axis, it is only important that the other three axes are named in a consistent way: The Y+ axis is the axis rotated $\frac{\pi}{2}$ against X+ and X- is the one rotated π against X+. These designations are not to be confused with those of a regular Cartesian coordinate system.

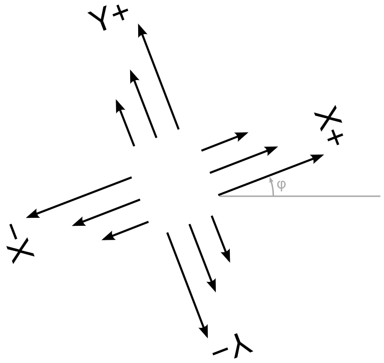


Figure 3.8: Vector field determined by cross correlation

The movement field of the example in Fig. 3.8 contains vectors which show perfect linear dependence on each other within the same axis and an exact phase angle of $\frac{\pi}{2}$ inbetween axes. In practice, this is often different: Due to bitmap rasterization of the images, the vector components will be integral values. Since this in an inaccurate representation, it is very likely that the vectors even along the same axis in the same direction will have different polar angles due to rounding errors.

With the values recorded the process of finding the correct movement times for the X and Y direction at a given object displacement (which is calculated by continuous imaging and cross correlation against the reference) is straightforward: The angle of the displacement α is first calculated

$$\alpha = \operatorname{atan2}(y,x) \tag{3.33}$$

where atan2 is defined as

$$\operatorname{atan2}(y,x) = \begin{cases} \operatorname{sgn}(y) \cdot \arctan(|\frac{y}{x}|) & y \neq 0 \wedge x > 0 \\ \operatorname{sgn}(y) \cdot \frac{\pi}{2} & y \neq 0 \wedge x = 0 \\ \operatorname{sgn}(y) \cdot (\pi - \arctan(|\frac{y}{x}|)) & y \neq 0 \wedge x < 0 \\ 0 & y = 0 \wedge x < 0 \\ \text{undefined} & y = 0 \wedge x = 0 \\ \pi & y = 0 \wedge x > 0 \end{cases} \tag{3.34}$$

The next step then is to determine which of the four directions is closest to α, i.e., the smallest angular difference between α and $\varphi + k \cdot \frac{\pi}{2}, k = 0 \ldots 3$. This is the direction out of which a movement vector will definitely be chosen. The second field would naturally be the one second closest to α. However, in practice this is numerically unstable when the angular difference between α and $\varphi + k \cdot \frac{\pi}{2}$ is sufficiently small. Therefore both directions of the orthogonal axis are taken into account.

Out of the vector field three vectors are chosen which are in their length closest to the length of the displacement vector. Then, the linear combinations

$$\vec{d} = \lambda\vec{x} + \mu\vec{y} \tag{3.35}$$

are calculated for x, y_1 and x, y_2 through

$$\begin{aligned} \lambda &= \frac{y_1 d_0 - y_0 d_1}{y_1 x_0 - y_0 x_1} \\ \mu &= \frac{x_0 d_1 - x_1 d_0}{y_1 x_0 - y_0 x_1} \end{aligned} \tag{3.36}$$

under the constraint that

$$\lambda \geq 0 \wedge \mu \geq 0 \tag{3.37}$$

These coefficients λ and μ then are multiplied with the movement duration that was recorded during the calibration process. This way, the directions and movement durations can be determined which cause the corrective motion that has to be performed.

3.6 Design of a Distributed System

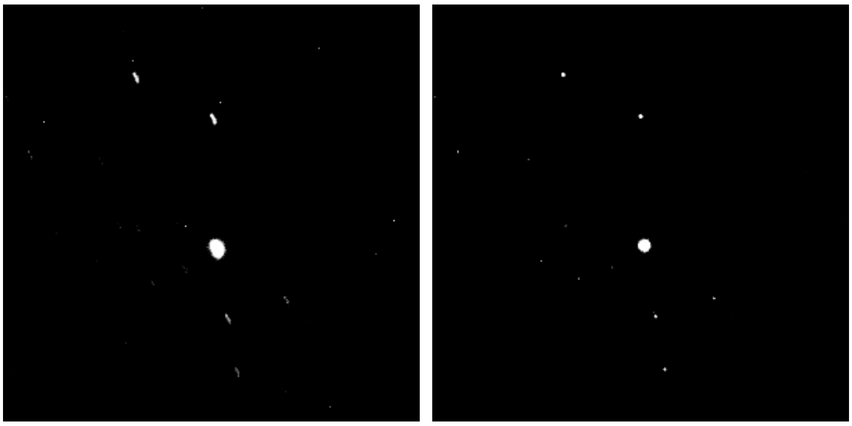
Astronomical imaging devices are often split up into many components like the telescope motors, the dome motor, and guiding and imaging CCD cameras. Not all interfaces of these components are controlled by the same computer. Unification of the control mechanism is desirable for the operator as it relieves him of the need to know where exactly interface points to the devices are. For a transparent access to all devices, a special device server protocol was developed. It operates on a very low abstraction layer and leaves enough room for the implementation to be able to handle devices with very different capabilities and properties. It operates in a simple request/response style manner and is in its design similar to that of the Post Office Protocol (POP) used for mail transfer [MR96]. However, in order to ensure that transparent access to devices is possible with only a single device server, the protocol knows the special `LISTEN` command. It is different than all other commands because it reverses the roles of the server and client during an active client-server session if it is executed successfully. Consider two computers, «UrsaMajor» and «UrsaMinor». While «UrsaMajor» controls all major functionality of the astronomical equipment, the dome control is connected to «UrsaMinor». For command unification, the protocol therefore can perform the following function: In the configuration of the «UrsaMinor» the dome driver is registered and also a reference to the controlling host «UrsaMajor» is stored. When the «UrsaMinor» server is first brought up it connects to «UrsaMajor» after device initialization and then issues the `LISTEN` command. This command tells «UrsaMajor» that the connected client can provide resources (i.e., devices) available which it is willing to share. If «UrsaMajor» accepts the resources, it acknowledges the command. Immediately after the `LISTEN` command has been issued the roles are reversed: The connection is now in a state where «UrsaMajor» is the client and «UrsaMinor» is the server. «UrsaMajor» then performs commands to look up the resources which «UrsaMinor» has to share and incorporates these into the list of available devices. When a client then connects to «UrsaMajor» and requests device operations which are actually in hardware available on «UrsaMinor», all commands are transparently relayed back and forth. This way, a connecting client does not have to have knowledge about the actual hardware or network layout, but can issue the commands to a single device server which takes care or the rest.

The advantage of using the `LISTEN` command in contrast to «UrsaMajor» simply connecting to «UrsaMinor» in the first place is that this way in the configuration of «UrsaMajor» no references to the clients have to be made. Therefore, the layout and location of the client servers may change, but the server configuration will always stay the same. Details about how the protocol is implemented and what parameters are used can be found in the Appendix C.

3.7 Imaging Examples

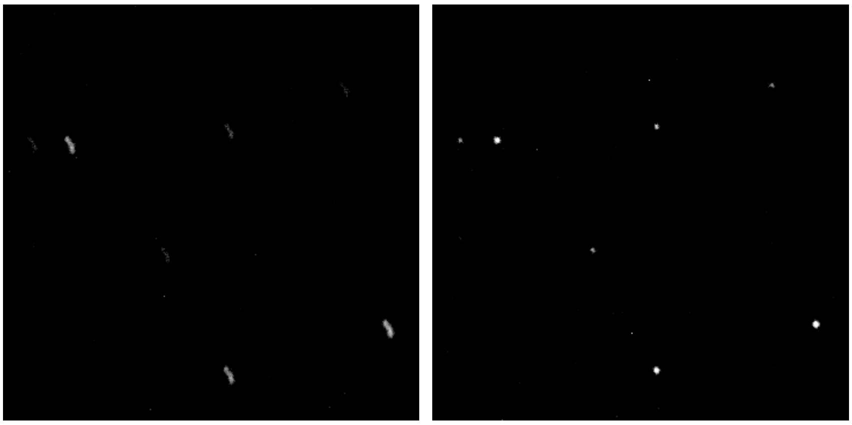
In Fig. 3.9 the difference between enabled and disabled guiding can be seen. Paradoxically, the image in Fig. 3.9a with disabled guiding might appear to show more details (i.e., some objects close to the right border). This is not not the case, however: Its brightness has only been scaled to match the apparent brightness of Fig. 3.9b. As the total maximal brightness of the guided picture is much higher (and so is its signal-to-noise ratio), through the scaling the faint objects in Fig. 3.9b become invisible (although they are indeed there). This becomes clear when looking at Fig. 3.10, which shows a different area of the same image, this time scaled equally. The non-guided picture 3.10a is much less bright than the guided equivalent in Fig. 3.10b.

An example of real imaging can be seen in Fig. 3.11, an image of the Ring Nebula (Messier Object 57). Images with red, green and blue filters were integrated to combine them into a single RGB picture. When the brightness is raised so much that the details of the Ring Nebula become invisible, a spiral galaxy can be seen to its right, shown on Fig. 3.11b. That image also shows a strait line from top to bottom which is probably caused by a airplane or satellite crossing the field of view during the integration period.



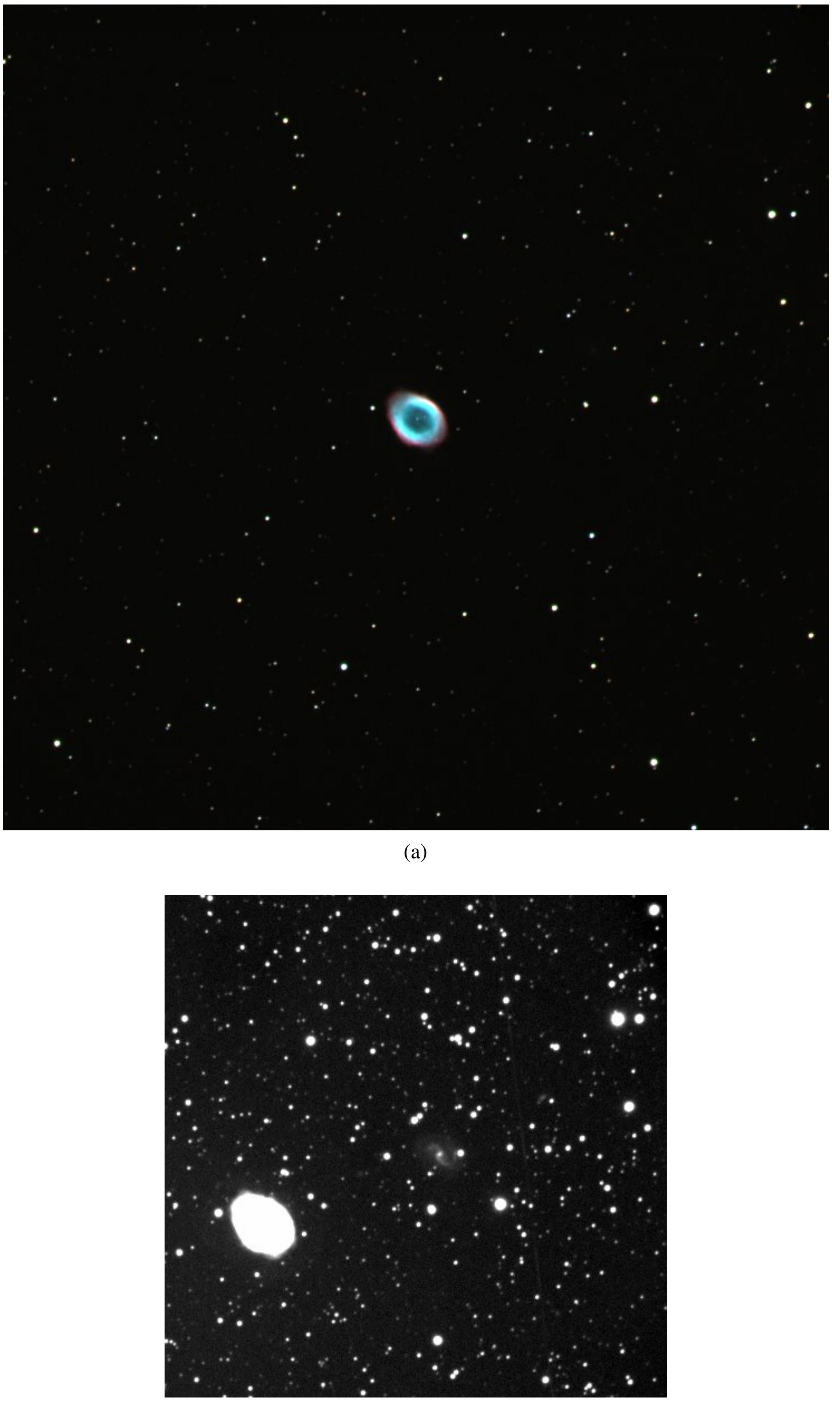
(a) (b)

Figure 3.9: Two images of 5 minutes each, (a) without and (b) with guiding both scaled to match same impression



(a) (b)

Figure 3.10: Two images of 5 minutes each, (a) without and (b) with guiding both scaled equally



(a)

(b)

Figure 3.11: (a) Messier 57 (Ring Nebula) taken with reverse engineered software, (b) shows same picture with luminance channel scaled for greater intensity

Chapter 4

Conclusion and Outlook

In the course of this work it was shown that the reverse engineering of binaries can be simplified significantly when appropriate utilities are used. Such a constraint-based reverse engineering tool has been developed and tested in the field to dissect code for various platforms. While the improvement of readability of disassembled code is good on complex instruction set computers (CISC) like the `x86` or `x86-64`, it performs outstandingly well with code for reduced instruction set computers (RISC). The expectations that were aimed at were definitely met: The disassembler performs those tasks that a machine can perform well, while leaving the important part of deciphering the semantic meaning to the user.

Through an iterative process it is possible to dissect code efficiently on the assembly level. Guided by other tools for analysis of library calls and writing this information intermingled with received and transmitted data into dumpfiles makes the reverse engineering of drivers like the used one relatively easy. However, there is much work still to do: The reverse engineering tool has no complete support of the supported instruction sets. The meaning of many opcodes is not understood by the disassembler and therefore all conditions have to be reset to an undefined state when such opcodes are encountered. It would be no problem to add support for these opcodes, too, but this was not yet done due to the time consuming nature of that task.

The practical work performed was the reimplementation of the CCD camera driver and the appropriate guiding implementation. It was explained that this is a rather nontrivial task with many constraints. Through use of efficient algorithms working in the frequency domain a solid solution could be presented which enables astrophotographers to take images of the skies with integration times far beyond 15 minutes. More than that, having an open implementation of the imaging framework gives users the possibility to write scripts performing astronomical imaging – something which was not possible before. This is an important step towards automated imaging. The framework was also designed for remote operability, something which is not only convenient in cold winter nights but also has positive impact on the imaging itself as the turbulence generated by the operator's body temperature disappears.

Only few things are perfect, however, and on this side there is also still work to be done: The usability of the imaging server is not as comfortable as could be expected. Setting up the guiding on a telescope is a lot of work and the tools designed are often not very verbose: Should, for example, a cable get loose on the connection from the telescope to the computer, the frontend will inform about the device error without hinting to the possible reasons. For experienced users such messages are annoying but for beginning astrophotographers they might save a lot of time searching for the error.

In this work on both the computer science and astrophysical side two very different but equally interesting topics were addressed. The work on the computer science side will aid reverse engineers in a manner in order to make static disassembly of binary code becomes less complicated – thus increasing productivity. On the astrophysical side it is now possible to perform high-quality imaging with completely free open source software. Both are merely tools which require capable hands to achieve good results – hopefully this work is a valuable contribution for both scientific fields.

Appendix A

Notation and Examples

A.1 Mathematical Notation

For all mathematical formulae used throughout this document, the following notation is used:

$$\tilde{x} = \mathrm{FFT}(x)$$

$$\bar{x} = \mathrm{conj}(x)$$

Should X be an image, then W_X is its width and H_X is its height. Image indices start from 0, the dimensions of X are therefore from $0 \ldots W_X - 1$ in X-direction and from $0 \ldots H_X - 1$ in Y-direction. When summing over a picture, the following explicit syntax will be used:

$$\sum_{x,y=\dim(A)} \mathrm{expr} = \sum_{x=0}^{W_A-1} \sum_{y=0}^{H_A-1} \mathrm{expr}$$

If it is clear which picture is meant, this may be omitted:

$$\sum_{x,y} A_{(x,y)} = \sum_{x=0}^{W_A-1} \sum_{y=0}^{H_A-1} A_{(x,y)}$$

If indices are not ambiguous, x and y will be used:

$$\sum_{A} \mathrm{expr} = \sum_{x=0}^{W_A-1} \sum_{y=0}^{H_A-1} \mathrm{expr}$$

Averaging of a picture is notated by:

$$\hat{A} = \mathrm{avg}(A) = \frac{1}{W_A \cdot H_A} \sum_{x,y} A_{(x,y)}$$

The *local average* as it appears in [Lew95] is notated by:

$$\hat{A}_{(x,y)}$$

and it has only then a meaning when there is a convolution of A with another (implied) image B(at an implied position i,j: Then, it denotes the average of the area under the feature B, i.e.

$$\frac{1}{W_B \cdot H_B} \sum_{x=i-\frac{1}{2}W_B}^{i+\frac{1}{2}W_B-1} \left(\sum_{y=j-\frac{1}{2}H_B}^{j+\frac{1}{2}H_B-1} A_{(x,y)} \right)$$

A.2 Notation of MBBs

As described in Sect. 1.5.1 on page 17 an example of the nomenclature will be given here. Consider the code List. 1.12. Formally, the analysis includes the aspects A:

$$A = \{\mathrm{ZF}, \mathrm{OF}, \mathrm{CF}, \mathrm{rax}, \mathrm{rbx}, \mathrm{rcx}, \mathrm{rdx}\} \tag{A.1}$$

The code which is analyzed contains 6 instructions:

$$\begin{aligned} i_1 = \text{mov \$9, \%rax}, \quad i_2 = \text{mov \%rax, \%rbx}, \quad i_3 = \text{mov mul \%rbx} \\ i_4 = \text{jmp MBB3}, \quad i_5 = \text{xor \%rdx, \%rdx}, \quad i_6 = \text{jmp MBBx} \end{aligned} \tag{A.2}$$

$$I = \{i_1, i_2, i_3, i_4, i_5, i_6\} \tag{A.3}$$

Those instructions are assembled into three Maximal Basic Blocks:

$$b_1 = (i_1, i_2, i_3, i_4), \quad b_2 = (i_5), \quad b_3 = (i_6) \tag{A.4}$$

$$B = \{b_1, b_2, b_3\} \tag{A.5}$$

Reference of the instructions is also possible by use of the index of the MBB in question:

$$b_{1,1} = i_1, \quad b_{1,2} = i_2, \quad \ldots, \quad b_{2,1} = i_5, \quad b_{3,1} = i_6 \tag{A.6}$$

The pre-conditions for the three blocks are

$$cr_1 = \emptyset, \quad cr_2 = \emptyset, \quad cr_3 = \{(\mathrm{OF}, 0), (\mathrm{CF}, 0), (\mathrm{rdx}, 0)\} \tag{A.7}$$

while the post-conditions are:

$$co_1 = \{(\mathrm{OF}, 0), (\mathrm{CF}, 0), (\mathrm{rax}, 81), (\mathrm{rbx}, 9), (\mathrm{rdx}, 0)\} \tag{A.8}$$

$$co_2 = \{(\mathrm{ZF}, 1), (\mathrm{OF}, 0), (\mathrm{CF}, 0), (\mathrm{rdx}, 0)\} \tag{A.9}$$

$$co_3 = \{(\mathrm{OF}, 0), (\mathrm{CF}, 0), (\mathrm{rdx}, 0)\} \tag{A.10}$$

A.3 CRC-8 Listing

Listing A.1: CRC-8 calculation with polynomial $x^8 + x^5 + x^4 + 1$

```
; { }

movzbl 0x8(%ebp),%ecx
; assume(ecx := Par1)
; +{ ecx := Par1 }

push   %ebx
movzbl 0xc(%ebp),%ebx
; assume(ebx := Par2)
; +{ ebx := Par2 }

mov    %ecx,%eax
; +{ eax := Par1 }

mov    %ecx,%edx
; +{ edx := Par1 }

and    $0x1,%eax
; +{ eax := AND(Par1, 1) }

and    $0xe6,%edx
; +{ edx := AND(Par1, 230) }

xor    %eax,%ebx
; +{ ebx := XOR(Par2, AND(Par1, 1)) }

mov    %ebx,%eax
; +{ eax := XOR(Par2, AND(Par1, 1)) }

sar    %edx
; +{ edx := SHR(AND(Par1, 230), 1) }

shl    $0x7,%eax
; +{ eax := SHL(XOR(Par2, AND(Par1, 1)), 7) }

or     %edx,%eax
; +{ eax := OR(SHL(XOR(Par2, AND(Par1, 1)), 7), SHR(AND(Par1, 230), 1))
      }

mov    %ecx,%edx
; +{ edx := Par1 }

and    $0x10,%edx
; +{ edx := AND(Par1, 16) }

and    $0x8,%ecx
; +{ ecx := AND(Par1, 8) }

sar    $0x4,%edx
; +{ edx := SHR(AND(Par1, 16), 4) }

xor    %ebx,%edx
; +{ edx := XOR(SHR(AND(Par1, 16), 4), XOR(Par2, AND(Par1, 1))) }

sar    $0x3,%ecx
; +{ ecx := SHR(AND(Par1, 8), 3) }

xor    %ebx,%ecx
```

```
; +{ ecx := XOR(SHR(AND(Par1, 8), 3), XOR(Par2, AND(Par1, 1))) }

shl     $0x3,%edx
; +{ edx := SHL(XOR(SHR(AND(Par1, 16), 4), XOR(Par2, AND(Par1, 1))), 3) }

or      %edx,%eax
; +{ eax := OR(OR(SHL(XOR(Par2, AND(Par1, 1)), 7), SHR(AND(Par1, 230), 1)),
;      SHL(XOR(SHR(AND(Par1, 16), 4), XOR(Par2, AND(Par1, 1))), 3)) }

shl     $0x2,%ecx
; +{ ecx := SHL(XOR(SHR(AND(Par1, 8), 3), XOR(Par2, AND(Par1, 1))), 2) }

or      %ecx,%eax
; +{ eax := OR(OR(OR(SHL(XOR(Par2, AND(Par1, 1)), 7), SHR(AND(Par1, 230), 1)),
;      SHL(XOR(SHR(AND(Par1, 16), 4), XOR(Par2, AND(Par1, 1))), 3)),
;      SHL(XOR(SHR(AND(Par1, 8), 3), XOR(Par2, AND(Par1, 1))), 2)) }

pop     %ebx
pop     %ebp
ret
```

A.4 Maximal Stack Regions

As described before, in the first step, each MBB is assigned its own MSR:

Listing A.2: Determining the MSRs

```
i1:                     ; MSR0 + 0x00
    push %eax           ; MSR0 - 0x04

i2:                     ; MSR1 + 0x00
    pop %eax            ; MSR1 + 0x04
    pop %eax            ; MSR1 + 0x08
    mov (%rsp), %rsp    ; MSR2 + 0x00
    push %rsp           ; MSR2 - 0x04

i3:                     ; MSR3 + 0x00
    pop %eax            ; MSR3 + 0x04
```

then, optimizations can be performed, by looking at the MSR graph (a modified MBB CFG), which only has the edges MSR$_0$ → MSR$_1$ and MSR$_2$ → MSR$_3$. Note that there is no edge between MSR$_1$ and MSR$_2$ as the block MSR$_2$ was created due to an indeterminable instruction. Therefore, only two MSRs remain, MSR$_0$ and MSR$_2$, with the summed offsets:

Listing A.3: Determining the MSRs after the optimization step

```
i1:                     ; MSR0 + 0x00
    push %eax           ; MSR0 - 0x04

i2:                     ; MSR0 - 0x04
    pop %eax            ; MSR0 + 0x00
    pop %eax            ; MSR0 + 0x04
    mov (%rsp), %rsp    ; MSR2 + 0x00
    push %rsp           ; MSR2 - 0x04
```

```
i3:                     ; MSR2 - 0x04
    pop %eax            ; MSR2 + 0x00
```

Here is a real-world example:

Listing A.4: Analysis of MSRs in code

```
; Entry from bar: Active MSR = M0 - 0x40
; Entry from xyz: Active MSR = M1 - 0x08
; Conflict -> Assign new Active MSR := M2 + 0x00
4004a0 <foo>:                        ; M2 + 0x00
4004a0: push   %rbp                  ; M2 - 0x08
4004a1: xor    %edx, %edx            ; %rdx = 0
4004a3: mov    %rsp, %rbp            ; %rbp = M2 - 0x08
4004a6: sub    $0x188, %rsp          ; M2 - 0x190
4004ad: lea    -0x200(%rbp), %rcx    ; %rcx = M2 - 0x208

; Entry from foo   : Active MSR = M2 - 0x190
; Entry from 4004c4: Active MSR = M2 - 0x190
; Identical -> M2 - 0x190
i1:
4004b4: mov    (%rdi, %rdx, 4), %eax  ; %eax = (int*)%rdi[%rdx]
4004b7: mov    %eax, (%rcx, %rdx, 4)  ; (int*)(M2 - 0x208)[%rdx] = %eax
4004ba: inc    %rdx
4004bd: cmp    $0x80, %rdx
4004c4: jne    4004b4 <foo+0x14>     ; Active MSR = M2 - 0x190

; Entry from 4004c4: Active MSR = M2 - 0x190
; Identical -> M2 - 0x190
i2:
4004c6: xor    %dl, %dl

; Entry from 4004c6: M2 - 0x190
; Entry from 4004c8: M2 - 0x190
; Identical -> M2 - 0x190
i3:
4004c8: mov    (%rdi, %rdx, 4), %eax  ; %eax = (int*)%rdi[%rdx]
4004cb: mov    %eax, (%rsi, %rdx, 4)  ; (int*)%rsi[%rdx] = %eax
4004ce: inc    %rdx
4004d1: cmp    $0x80, %rdx
4004d8: jne    4004c8 <foo+0x28>     ; Active MSR = M2 - 0x190

; Entry from 4004d8: Active MSR = M2 - 0x190
; Identical -> M2 - 0x190
i4:
4004da: xor    %dl, %dl
4004dc: nopl   0x0(%rax)             ; Active MSR = M2 - 0x190

; Entry from 4004dc: Active MSR = M2 - 0x190
; Identical -> M2 - 0x190
i5:
4004e0: mov    (%rcx, %rdx, 4), %eax  ; %eax = (int*)(M2 - 0x208)[%rdx]
4004e3: mov    %eax, (%rdi, %rdx, 4)  ; (int*)%rdi[%rdx] = %eax
4004e6: inc    %rdx
4004e9: cmp    $0x80, %rdx
4004f0: jne    <i5>                  ; Active MSR = M2 - 0x190

; Entry from 4004f0: Active MSR = M2 - 0x190
; -> M2 - 0x190
4004f2: leaveq
4004f3: retq
```

A.5 Exception Handling

Listing A.5: Exception low-level Assembly

```
; Get memory for exception to be thrown (4 bytes) in %eax
80486bc:      c7 04 24 04 00 00 00    movl   $0x4, (%esp)
80486c3:      e8 70 fe ff ff          call   <
    __cxa_allocate_exception@plt>

; Instantiate moo()
80486c8:      c7 44 24 04 34 12 00    movl   $0x1234, 0x4(%esp)
80486cf:      00
80486d0:      89 04 24                mov    %eax, (%esp)
80486d3:      89 c3                   mov    %eax, %ebx
80486d5:      e8 a6 ff ff ff          call   <moo::moo(int)>

80486da:      c7 44 24 08 00 00 00    movl   $0x0, 0x8(%esp)
80486e1:      00
80486e2:      c7 44 24 04 10 88 04    movl   $0x8048810, 0x4(%esp)
80486e9:      08
80486ea:      89 1c 24                mov    %ebx, (%esp)
80486ed:      e8 66 fe ff ff          call   <__cxa_throw@plt>

80486f2:      89 1c 24                mov    %ebx, (%esp)
80486f5:      89 c6                   mov    %eax, %esi
80486f7:      89 d7                   mov    %edx, %edi
80486f9:      e8 4a fe ff ff          call   <__cxa_free_exception@plt
    >

; Is this the correct exception handler? Then handle it
80486fe:      83 ef 01                sub    $0x1, %edi
8048701:      74 0e                   je     8048711 <main+0x71>

; No exception handlers match: Continue stack unwinding
8048703:      89 34 24                mov    %esi, (%esp)
8048706:      e8 8d fe ff ff          call   <_Unwind_Resume@plt>

804870b:      89 c6                   mov    %eax, %esi
804870d:      89 d7                   mov    %edx, %edi
804870f:      eb ed                   jmp    80486fe <main+0x5e>

8048711:      89 34 24                mov    %esi, (%esp)
8048714:      e8 5f fe ff ff          call   <__cxa_begin_catch@plt>

; %ebx = ((moo*)%eax)->get()
8048719:      89 04 24                mov    %eax, (%esp)
804871c:      e8 6f ff ff ff          call   <moo::get() const>
8048721:      89 c3                   mov    %eax, %ebx

; Clean up
8048723:      e8 40 fe ff ff          call   <__cxa_end_catch@plt>
```

Appendix B

SBIG Camera Protocol

B.1 General Notes

All commands which transmit parameters which are more than one byte in length encode the values with big endianness. Almost all commands and responses have the constant prefix `0x5a` as the byte with offset 0. The only exception is the `ACK` command, which is always `00 06`. The command number is encoded in the byte with offset 1 (e.g, command `0x60` is the *Establish Link* command, described in Sect. B.2.1). Command bits which are set in the hex display of the graphics are always set, even if that is not mentioned explicitly each time.

Response: Acknowledge

```
 0  1
06 00
```

B.2 Command Reference

B.2.1 Establish Link

The *Establish Link* command is called before all other commands and before an exposure is started.

Command: 0x60: Establish link
Expects: Response 0x62

```
 0  1
a5 60
```

Response: 0x62: Establish link

```
 0  1  2  3
a5 62 00 00
         Firmware version
```

B.2.2 Temperature Regulation

These commands either query the current CCD camera temperature or set the temperature regulation. The ADU values for the regulation are 12 bits in length, but only the 8 most significant bits are transmitted. The formula with which the values are converted can be seen in the SBIG SDK source code. It is

$$r_{(T)} = R_0 \cdot \mathrm{RR}_{\mathrm{Dev}}^{\left(\mathrm{DT}_{\mathrm{Dev}}^{-1}\cdot(T_0-T)\right)} \tag{B.1}$$

$$ADU_{(T)} = \frac{\mathrm{MAXAD}}{\mathrm{RB}_{\mathrm{Dev}}\cdot r_{(f)}^{-1}+1.5} \tag{B.2}$$

where MAXAD, R_0 and T_0 are constant values for all curves:

$$\mathrm{MAXAD} = 4096, \quad R_0 = 3, \quad T_0 = 25$$

and RR, RB and DT are the curve specific parameters which are different for the CCD temperature calculation and the calculation of the ambient temperature:

$$\mathrm{RR}_{\mathrm{CCD}} = 2.57, \quad \mathrm{RB}_{\mathrm{CCD}} = 10, \quad \mathrm{DT}_{\mathrm{CCD}} = 25$$

$$\mathrm{RR}_{\mathrm{Ambient}} = 7.791, \quad \mathrm{RB}_{\mathrm{Ambient}} = 3, \quad \mathrm{DT}_{\mathrm{Ambient}} = 45$$

Not all cameras seem to support the ambient temperature calculation – the ST-9 and STL-11k always return 25° C.

Command: 0x30: Query temperature status
Expects: Response 0x35

```
 0  1
a5 30
```

Response: 0x35: Temperature Status

```
 0  1  2  3  4  5  6  7
a5 35 00 00 00 00 00 00
      |  |  |  |  |
      |  |  |  |  └───────┤Current Peltier power from 0-255
      |  |  |  └──────────┤Ambient temperature >> 4 (in ADU)
      |  |  └─────────────┤CCD temperature >> 4 (in ADU)
      |  └────────────────┤Temperature setpoint >> 4 (in ADU)
      └───────────────────┤Temperature regulation state (0 = Off, 1 = On)
```

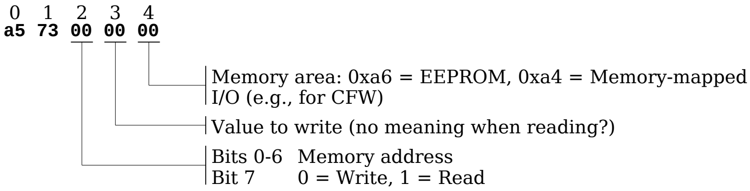
Command: 0x23: Set temperature regulation
Expects: ACK

```
 0  1  2  3  4
a5 23 00 00 e6
      |  |
      |  └─────┤Setpoint in ADU >> 4
      └────────┤Status of temperature regulation (0 = Off, 1 = On)
```

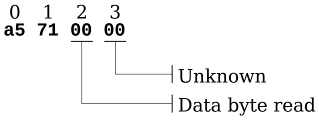
B.2.3 I^{2}C Access

Warning: This command can damage your camera if used improperly! It implements memory read/write access to 128 bytes of specific memory areas. In the command `0x73` an address and a memory area is requested. The areas known are `0xa6` which refers to the internal EEPROM on chip and `0xa4` which is used to control the position of the color filter wheel in the STL-11k. This is done internally via I^2C access (both the CFW and EEPROM are connected to the I2C bus of the EZ-USB device). In the EEPROM there is information stored about the chip (i.e., internal specifics), the serial number of the camera, the device and vendor ID of the USB device, and the user memory area (which is the only area accessible through the official API). When writing to the EEPROM, it is very possible to brick the camera, for example if the USB ID is overwritten – the camera will then no longer correctly do USB enumeration and only manual upload of the firmware is possible. Use with extreme caution.

Command: 0x73: Memory access
Expects: Response 0x71 (to read) or ACK (to write)


0 1 2 3 4
a5 73 00 00 00
Memory area: 0xa6 = EEPROM, 0xa4 = Memory-mapped I/O (e.g., for CFW)
Value to write (no meaning when reading?)
Bits 0-6 Memory address
Bit 7 0 = Write, 1 = Read

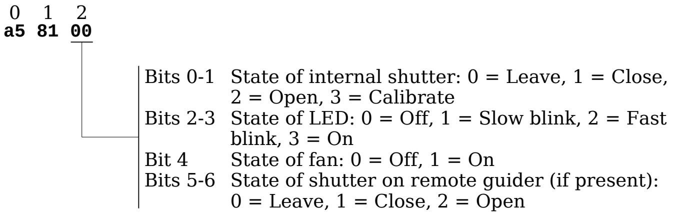
Response: 0x71: Memory Read Result



B.2.4 Shutter Control

The manual shutter control is only necessary for use with the remote guide head, which does not automatically close the shutter when integrating an image. When taking a dark frame on the remote guide head, this command is useful.

Command: 0x81: Manual shutter control
Expects: ACK


0 1 2
a5 81 00
Bits 0-1 State of internal shutter: 0 = Leave, 1 = Close, 2 = Open, 3 = Calibrate
Bits 2-3 State of LED: 0 = Off, 1 = Slow blink, 2 = Fast blink, 3 = On
Bit 4 State of fan: 0 = Off, 1 = On
Bits 5-6 State of shutter on remote guider (if present): 0 = Leave, 1 = Close, 2 = Open

B.2.5 Exposure Control

The *Start Exposure* command is passed the desired integration time in units of $\frac{1}{100}$ seconds. It is also given the sensor ID on which integration is desired and if the shutter should be opened (otherwise a dark frame will be taken). Note that to take a dark frame on the remote guiding head, the *Manual Shutter Control* command has to be used.

Command: 0x04: Start Exposure
Expects: ACK

```
0  1  2  3  4  5
a5 04 00 00 00 02
```

Bit 0	Always 0
Bit 1	Always 1
Bits 2-3	1 = Dark frame, 2 = Light frame
Bit 4	0 = Main sensor, 1 = Guide sensor (internal or remote)
Bit 5	Always 0
Bit 6	0 = Internal sensor, 1 = Remote guide head
Bit 7	Always 0

Numer of ticks (i.e. 1/100th seconds) the integration takes

Command: 0x11: End Exposure
Expects: ACK

```
0  1  2
a5 11 00
```

0 = Imaging sensor, 1 = Guiding sensor

B.2.6 Command Status

Response: 0x93: Command Status

```
0  1  2  3  4  5
a5 93 00 05 08 80
```

Bits 0-1	State of imaging sensor 0 = Idle, 1 = Busy, 2 = Integrating, 3 = Complete
Bits 2-3	State of guiding sensor 0 = Idle, 1 = Busy, 2 = Integrating, 3 = Complete

Command: 0x90: Query Command Status
Expects: Response 0x93

```
0  1
a5 90
```

B.2.7 CCD Readout Control

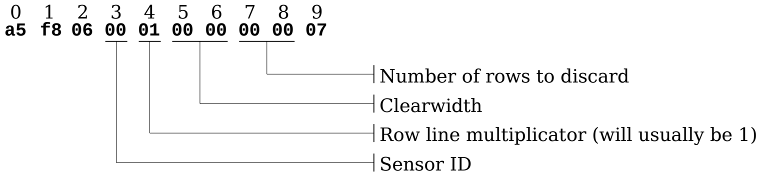
The prepare readout command is issued after the *End Exposure* command. Then, lines can be discarded via the *Discard Lines* command until they are read out via the *Request Data* commands.

Command: 0xf8 / 0x05: Prepare readout
Expects: ACK

```
0  1  2  3  4  5  6  7  8  9
a5 f8 05 00 00 00 00 00 00 00
```

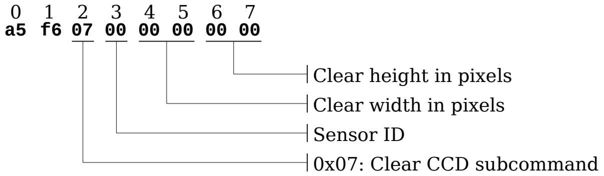
Magic 0xf805 value Y
Magic 0xf805 value X
Binning Y
Binning X
Sensor ID

Command: 0xf8 / 0x06: Discard lines
Expects: ACK

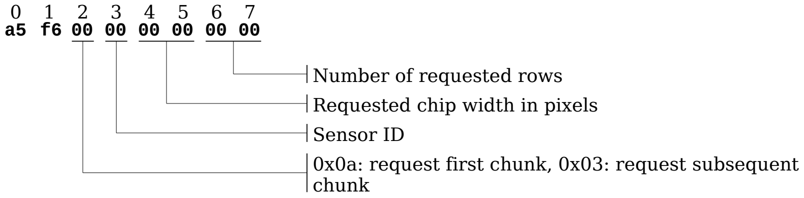

0 1 2 3 4 5 6 7 8 9
a5 f8 06 00 01 00 00 00 00 07
Number of rows to discard
Clearwidth
Row line multiplicator (will usually be 1)
Sensor ID

The `0xf6` command series are commands which are split into somewhat similar subcommands. Known are the `0xf607`, the `0xf60a` and `0xf603` subcommand which are used for clearing the CCD and requesting data from it. When requesting data, for the first block the `0xf60a` subcommand has to be used while all subsequent blocks are retrieved via the `0xf603` subcommand.

Command: 0xf6 / 0x07: Clear CCD
Expects: ACK

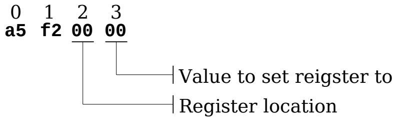


Command: 0xf6 / 0x0a, 0x03: Request Data
Expects: Raw data packet of requested length


0 1 2 3 4 5 6 7
a5 f6 00 00 00 00 00 00
Number of requested rows
Requested chip width in pixels
Sensor ID
0x0a: request first chunk, 0x03: request subsequent chunk

The purpose of the `0xf2` command is unknown. On both the ST-9 and ST-11k the register `0x08` is set to `0x01` before the begin of the exposure (i.e., before the *Start Exposure* command) and set to `0x00` again after the last block has been read from the CCD. On the ST-11k only there is also access to the `0x0b` register after the `0x08` register has been set. The value passed is the sensor ID which was just read out. The `0xf208` command seems to respond with the request itself (although sometimes it responds with the request except that the value returned is different from the one set) while the `0xf20b` command always responds with ACK.

Command: 0xf2: Set unknown register
Expects: Different, depending on register: sometimes ACK, sometimes the request itself



B.3 Camera Parameters

All known camera readout mode parameters which have fixed values are recorded in Tables B.1 and B.2. Some of these are "magic" values which are always sent in the same commands, other are general readout parameters. The chip width and height give the amount of pixels which are read from the chip while the image dimensions give the width and height of the actual picture (after cropping away separation pixels). The offset of the crop is also given.

Table B.1: High level readout mode parameters

Camera	Sensor	Binning Mode	Image	Chip	Crop
ST-9XE	Imaging	1×1	512×512	516×527	$(4,0)$
	Guiding	1×1	662×495	662×495	$(0,0)$
	Remote	1×1	662×495	662×495	$(0,0)$
ST-11k	Imaging	1×1	4012×2672	4012×2672	$(0,0)$
		2×2	2008×1336	2008×1336	$(0,0)$
		3×3	1340×890	1340×890	$(0,0)$

Table B.2: Low level readout mode parameters

Camera	Sensor	Bin	Clear Width	Row Discard	F8-05-X	F8-05-Y	F3
ST-9XE	Imaging	1	534	4	12	10	22
	Guiding	1	683	3	26	0	N/A
	Remote	1	683	3	26	0	N/A
ST-11k	Imaging	1	4076	24	36	32	44
		2	4076	24	18	16	13
		3	4076	24	12	10	13

B.4 Typical Imaging Process

Algorithm 4 Integrate a picture and retrieve it

```
0x62 ⇒ EstablishLink
0x11 ⇒ EndExposure
0xf806 ⇒ DiscardLines(Mode.RowDiscard)
0xf607 ⇒ ClearCCD
0xf2 ⇒ MagicF2(0x08 := 1)
0x04 ⇒ StartExposure
repeat
  Wait
until (0x90 ⇒ QueryCommandStatus) = Exposure finished
0x11 ⇒ EndExposure
0xf806 ⇒ DiscardLines(Mode.RowDiscard)
if Camera.MagicF3 ≠ N/A then
  0xf3 ⇒ PrepareReadout(Mode.MagicF3)
end if
0xf1 ⇒ PrepareReadout
0xf805 ⇒ PrepareReadout(Mode.MagicF805X, Mode.MagicF805Y)
while RowsRemaining do
  if First chunk then
    0xf60a ⇒ Fetch first chunk
  else
    0xf603 ⇒ Fetch subsequent chunk
  end if
end while
0xf2 ⇒ MagicF2(0x08 := 0)
if Camera = ST-11k then
  0xf2 ⇒ MagicF2(0x0b := SensorID)
end if
```

Appendix C

Device Server Protocol

C.1 Identifiers

All identifiers are case-sensitive except if explicitly stated. If more than one parameters of the same type are included in a command or query, those may be written as `<Type-[0-9]+>` for have further reference within the description. If a multiline answer is given with multiple types those may be written as `<Type-[0-9]+-[0-9]+>` where the first integer value indicates the line reference and the latter indicates the argument reference (if necessary for disambiguation).

- Identifier: [-_A-Za-z0-9]+
- String: [_/.,:$?%-=A-Za-z0-9]+
- Text [_/.,:$?%-=A-Za-z0-9]+
- HexString: [0-9a-f]+
- Code: [0-9]+
- Command: `<Identifier>` (case insensitive)

C.2 Commands

C.2.1 Requests

- Request := `<Command>` `<Code>` `<Identifier>`(`<String>`)*

Where arguments are indexed starting from 1 and command may be referred to as argument 0.

- Example: `EXECUTE 4 INTEGRATE 60`
- Command: `EXECUTE`
- Code (Device ID): `4`
- Subcommand: `INTEGRATE`
- Argument 0: `INTEGRATE`
- Argument 1: `60`

C.2.2 Responses

- CommentResponse := `<Text>`
- SingleResponse := (+OK|-ERR) `<Identifier>` `<Text>`?
- MultiResponse := (+OKDATA|-ERR `<Identifier>` `<Text>`?)

The server may output `<CommentResponse>` responses at any time except inbetween "+OKDATA" and "." for the sole purpose of providing additional output to a connected user. The "+OK" and "-ERR" responses always give the error code as described in Sect. C.7 as the first parameter and can also provide a textual, possibly more detailed description of the reason.

C.3 Line format

All lines terminate with 0x0a (\n), although lines terminated with 0x0d 0x0a (\r\n) shall be tolerated by any server conforming to the protocol.

C.4 Command Format

Single line answer commands:

- → `<Request>`
- ← `<SingleResponse>`

Multi line answer commands when response is "+OKDATA":

- → `<Request>`
- ← `<MultiResponse>`
- ← `<Text>`
- ← `.`

C.5 Protocol States

1. Unauthenticated (`NOTAUTH`)
2. Authentication requested (`AUTHREQ`)
3. Authenticated (`AUTH`)

Conforming implementations shall always return -ERR upon receiving any command which is not valid in the current state.

C.6 Commands

C.6.1 RQAUTH

Requests authorization challenge from server.

Valid in	`NOTAUTH`, `AUTH`
Request	→ `RQAUTH`
Response	← `+OK <String> YYYYMMDDHHmmSS-<Identifier>`
Where	`<String>` are the server supported authentication hash function identifiers, separated by comma. Any conforming server must support at least `SHA1`. `YYYY` is the current year `MM` is the current month `DD` is the current day of month `HH` is the current hour `mm` is the current minute `SS` is the current second `<Identifier>` is a string identifying the server host.
Transitions	→ `AUTHREQ`
Example	→ `RQAUTH` ← `+OK MD5,SHA1 20081224123456-imageserver`

C.6.2 AUTH

Authenticates against server with previously acquired challenge

Valid in AUTHREQ

Request → AUTH <Identifier-1> <Identifier-2> <HexString>

Where <Identifier-1> is the user name of the user trying to authenticate
<Identifier-2> is the hash function used for authentication
<HexString> is the response r to the previously given challenge c with password k (shared secret) using the previously given hash function h with block length b (e.g. $b = 64$ bytes for MD5 or SHA1) calculated as of RFC 2104 [KBC97]. A short explanation:
ipad = 0x36 repeated b times
opad = 0x5c repeated b times
$k = pad(\text{password}, b)$
$r = h(k \oplus \text{opad}, h(k \oplus \text{ipad}, c))$
Padding is is left-aligned with pattern 0x00. For details if $length(k) > b$ consult RFC 2104 [KBC97].

Response ← +OK if authentication succeeded.

Response ← -ERR EAUTHUSR/EAUTHPW/EAUTH if authentication failed.

Transitions → AUTH or → NOTAUTH

Example Password for joe is foobar in the following example:

```
→ RQAUTH
← +OK SHA1,MD5 20081022220035-myhost
→ AUTH joe MD5 11111111122222222233333344444444
← -ERR EAUTHPW Your password is not correct.
→ RQAUTH
← +OK SHA1,MD5 20081022220044-myhost
→ AUTH joe MD5 9c64d0990ee92a637d9cb68734a2b937
← +OK Welcome user joe.
```

Comments The server may choose to close the connection, if authentication fails, instead of making the transition to the → NOTAUTH state.

C.6.3 LIST

Lists the devices accessible to server.

Valid in	`AUTH`
Request	→ `LIST`
Response	← `+OKDATA` ← `<Code-1> <Identifier-1> <String-1>`(LOCKED)? ← [...] ← `<Code-N> <Identifier-N> <String-N>`(LOCKED)? ← `.`
Where	`<Code>` is the number of device N which the client can get a `LOCK` on. `<Identifier>` is the string representation of device N which is not necessarily unique and only for user convenience. It usually contains the type of device. `<String>` is the location of the device. For local devices this has to be "LOCAL", for remote devices it will be the name of the remote host. The LOCKED keyword indicates that the listed device is currently locked.
Example	→ `LIST` ← `+OKDATA` ← `0 mysbig-st9-1 LOCAL` ← `3 mysbig-st9-2 LOCAL LOCKED` ← `4 mysbig-st12 LOCAL` ← `8 telescope ngc7293.sternwarte.de LOCKED` ← `.`

C.6.4 LOCK

Lock a connected device or set of connected devices.

Valid in	`AUTH`
Request	→ `LOCK <Code>( <Code>)*`
Where	`<Code>` are the device codes of the devices to be locked. Lock acquirement shall happen atomically. Either the current user gets a lock for all devices or doesn't acquire any lock at all.
Response	← `+OK` if all devices could be locked ← `-ERR ELOCK` if at least one of the requested devices is already locked. ← `-ERR ENODEV` if at least one of the requested devices does not exist.

C.6.5 UNLOCK

Unlock a connected device.

Valid in	`AUTH`
Request	→ `UNLOCK <Code>`
Where	`<Code>` is the device code of the devices to be unlocked.
Response	← `+OK` if the device could be unlocked. ← `-ERR EBUSY` if the device could not be unlocked because it is currently performing an operation. ← `-ERR ENODEV` if the device number is invalid. ← `-ERR ENOLOCK` if the device could not be unlocked because it does not currently have a lock on by the current user.

C.6.6 QUIT

Close connection to server.

Valid in	NOTAUTH, AUTHREQ, AUTH
Request	→ QUIT
Response	← +OK before the server closes the connection.

C.6.7 LISTPARAMS

Lists the parameters which a device has.

Valid in	AUTH
Request	→ LISTPARAMS <Code>
Where	<Code> is the device code of the device of which the available parameters should be listed.
Response	← +OKDATA ← <Identifier-1> <String-1> -> <Text-1> ← [...] ← <Identifier-N> <String-N> -> <Text-N> ← . ← -ERR ENODEV if the device number is invalid.
Where	<Identifier> is the name of the parameter. <String> is the prototype of the parameter, separated by commas. The first value shall always be "RW" for a read-write-property or "RO" for a read-only property. <Text> is the description of the parameter.
Example	→ LISTPARAMS 1 ← +OKDATA ← POSITION RW,float,float -> Get/set RA and DEC ← VELOCITY RO,unsigned int -> Get slew velocity ← SWAPRADEC RW,bool,bool -> Swap RA with DEC ← .

C.6.8 SETPARAM

Sets a device parameter to a specified value.

Valid in	`AUTH`
Request	→ `SETPARAM <Code> <Identifier> <Text>`
Where	`<Code>` is the device code of the device of which the parameters should be modified. `<Identifier>` is the parameter name to modify. `<Text>` is the new value or new values the parameter shall have, separated by spaces.
Response	← `+OK` ← `-ERR ELOCK` if the device is currently locked by a different user. ← `-ERR ENODEV` if the device number is invalid. ← `-ERR ENOPARM` if the device number does not offer the requested parameter. ← `-ERR ENOLOCK` if the parameter requires a lock to be held, but none exists. ← `-ERR EWRPROT` if the parameter is write protected.
Example	→ `SETPARAM 0 CFWPOSITION RED` ← `+OK`

C.6.9 LISTCAPABILITIES

Lists the capabilities which a device offers.

Valid in	`AUTH`
Request	→ `LISTCAPABILITIES <Code>`
Where	`<Code>` is the device code of the device of which the available capabilities should be listed.
Response	← `+OKDATA` ← `<Identifier-1> <String-1>` -> `<Text-1>` ← [...] ← `<Identifier-N> <String-N>` -> `<Text-N>` ← `.` ← `-ERR ENODEV` if the device number is invalid.
Where	`<Identifier>` is the name of the capability. `<String>` are the capability prototype parameters. `<Text>` is a textual description of the capability.
Example	→ `LISTCAPABILITIES 0` ← `+OKDATA` ← `RESET -> Reset the USB port and the camera` ← `SHUTTER [INT|EXT],[OPEN|CLOSE] -> Shutter control` ← `.`

C.6.10 EXECUTE

Execute a capability on a previously opened (and possibly locked) device.

Valid in	AUTH
Request	→ EXECUTE <Code> <Identifier> <Text>
Where	<Code> is the device code of the device of which the capability should be executed. <Identifier> is the name of the capability call to be executed. <Text> contains the parameters of the capability, separated by spaces.
Response	← +OK if the capability could be executed successfully. ← -ERR ELOCK if the device is currently locked by a different user. ← -ERR ENOLOCK if capability requires a locked device, but user does not hold lock. ← -ERR ENODEV if the device number is invalid. ← -ERR ENOCAP if the capability identifier is invalid. ← -ERR EDEVFAIL if the capability cannot be executed because of device failure.

C.6.11 LISTRESULTS

Lists the results which a device produced.

Valid in	AUTH
Request	→ LISTRESULTS <Code>
Where	<Code> is the device code of the device of which the available results should be listed.
Response	← +OKDATA ← <String-1> ← [...] ← <String-N> ← . ← -ERR ENODEV if the device number is invalid.
Where	<String> is the name of the result.

C.6.12 FETCH

Fetch a result from a device.

Valid in	`AUTH`
Request	→ `FETCH <Code> <Identifier-1> <Identifier-2>`
Where	`<Code>` is the device code of the device of which the available results should be fetched. `<Identifier-1>` is the name of the result to be fetched. `<Identifier-2>` is the name of the filter to be applied by the server before sending out the data. Conforming implementations need to only support the PLAIN filter, which does not process the data in any way before Base64-encoding is performed. However this standard suggests that GZIP should be supported as a filter which performs GZIP compression as of RFC 1952 [Deu96].
Response	← `+OKDATA` ← [...] ← .
Where	The data is Base64-encoded binary data as of RFC 3548 [Jos03]. ← `-ERR ENODEV` if the device number is invalid. ← `-ERR ENORES` if the result name is invalid. ← `-ERR EINVCONT` if the server does not understand the filter identifier. ← `-ERR EFILEERR` if the server cannot serve the requested result file (e.g., possibly due to I/O errors).

C.6.13 LISTEN

Switch roles after authentication in order to relay resources. This means the connected client will, after the LISTEN command has been performed successfully, act as the server and the server will send client commands. The same connection will be used. Usually the host acknowledging the LISTEN command will first perform a LIST and LISTPARAMS command in order to update the own list of known devices. It then will take care of relaying.

Valid in	`AUTH`
Request	→ `LISTEN <Identifier>`
Where	`<Identifier>` is the name of the connected client which offers access to all of its devices.
Response	← `+OK` ← `-EPERM` if the user does not have permission to share devices. ← `-ENAMETK` if the supplied hostname is already taken by another connection. ← `-ENOTIMPL` if the `LISTEN` operation is not implemented in the server.

C.7 Error Codes

1. `EAUTHUSR` Authentication failed, unknown username

2. `EAUTHPW` Authentication failed, wrong password

3. `EAUTH` Unspecified authentication failure

4. `EPERMS` Permission denied

5. `EDEVBUSY` Device or resource busy
6. `ELOCK` Device is locked
7. `ENOLOCK` Device is not locked
8. `ENOSUCHDEV` No such device
9. `ENOCAP` No such capability
10. `ENOPAR` No such parameter
11. `ENORES` No such result
12. `ENAMETK` The name is already taken
13. `ECMDEX` Command expected
14. `ECMDINV` Command unknown or invalid in current protocol state
15. `ECMDPAR` Command has not supplied the exact number of required parameters
16. `ECMDMLFRM` Command has malformed or unsupported syntax or parameters
17. `ENOTIMPL` The operation is not implemented in the server
18. `EINVCONT` Invalid or unsupported content type requested
19. `EWRPROT` Property is write-protected (read-only)
20. `EDEVFAIL` Device failure or device unresponsive
21. `EFILEERR` Unable to serve the requested result file

Bibliography

[App02] Andrew W. Appel. Deobfuscation is in NP. August 2002.

[BB06] Richard Berry and James Burnell. *The Handbook of Astronomical Image Processing*. Willman-Bell, Inc., 2006.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs, 2001.

[BH07] Michael Batchelder and Laurie J. Hendren. Obfuscating java: The most pain for the least gain. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2007.

[Boo05] Neil Booth. Cpplib internals, 2005. `http://gcc.gnu.org/onlinedocs/cppinternals.pdf`.

[Com95] Tool Interface Standard Committee. Executable and linking format (elf) specification 1.2. May 1995. `http://refspecs.freestandards.org/elf/elf.pdf`.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM Press.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html`.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97c/index.html`.

[Cyp00] Cypress Semiconductor. EZ-USB FX technical reference manual, 2000. `http://www.keil.com/dd/docs/datashts/cypress/fx_trm.pdf`.

[Cyp01a] Cypress Semiconductor. Cy7c64601/603/613 EZ-USB FX USB microcontroller datasheet, September 2001. http://www.keil.com/dd/docs/datashts/cypress/cy7c646xx_ds.pdf.

[Cyp01b] Cypress Semiconductor. EZ-USB FX2 technical reference manual, 2001. http://www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf.

[Cyp02a] Cypress Semiconductor. Cy7c68013 EZ-USB FX2 USB microcontroller high-speed USB peripheral controller, June 2002. http://www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf.

[Cyp02b] Cypress Semiconductor. EZ-USB technical reference manual, 2002. http://download.cypress.com.edgesuite.net/design_resources/datasheets/contents/an21xx_8.pdf.

[Deu96] P. Deutsch. Rfc1952 – gzip file format specification version 4.3, May 1996. http://www.ietf.org/rfc/rfc1952.txt.

[Eil05] Eldad Eilam. *Reversing – Secrets of Reverse Engineering*. Wiley, 2005.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[Gra02] Volker Grassmuck. *Freie Software – Zwischen Privat- und Gemeineigentum*. Bundeszentrale für politische Bildung, Bonn, 2002.

[GSTF08] Manuel Guizar-Sicairos, Samuel T. Thurman, and James R. Fienup. Efficient subpixel image registration algorithms. *Opt. Lett.*, 33(2):156–158, 2008.

[Int07a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*. August 2007. http://www.intel.com/design/processor/manuals/253665.pdf.

[Int07b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2B: Instruction Set Reference, N-Z*. May 2007. http://www.intel.com/design/processor/manuals/253667.pdf.

[Jos03] S. Josefsson. Rfc 3548 – the base16, base32, and base64 data encodings, July 2003. http://www.ietf.org/rfc/rfc3548.txt.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. Rfc2104 – hmac: Keyed-hashing for message authentication, February 1997. http://www.ietf.org/rfc/rfc2104.txt.

[KKO$^+$96] H. Karttunen, P. Kröger, H. Oja, M.Poutanen, and K. J. Donner. *Fundamental Astronomy*. Springer, Heidelberg, 1996.

[Kod06] Kodak Image Sensor Solutions. Product summary of Kodak KAI-11002 progressive scan interline ccd image sensor, 2006. http://www.kodak.com/global/plugins/acrobat/en/business/ISS/productsummary/Interline/KAI-11002ProductSummary.pdf.

[KVRV04] Christopher Kruegel, Fredrik Valeur, William Robertson, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *13th Usenix Security Symposium*, August 2004.

[Lag96] Jeffrey C. Lagarias. The $3x+1$ problem and its generalizations, January 1996.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *In ACM Conference on Computer and Communications Security (CCS)*, pages 290–299. ACM Press, 2003.

[Lew95] J. P. Lewis. Fast normalized cross-correlation. In *Vision Interface*, pages 120–123. Canadian Image Processing and Pattern Recognition Society, 1995.

[LSW08] Ulf Lamping, Richard Sharpe, and Ed Warnicke. Wireshark users' guide 29371 for wireshark 1.2.0, 2008. http://www.wireshark.org/download/docs/user-guide-a4.pdf.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, Palo Alto, California, USA, April 1999.

[McL97] Ian S. McLean. *Electronic Imaging in Astronomy – Detectors and Instrumentation*. Wiley, Chichester, England, 1997.

[MR96] J. Myers and M. Rose. Rfc 1939 – post office protocol version 3, May 1996. http://www.ietf.org/rfc/rfc1939.txt.

[Mü93] Urban Müller. Brainfuck – an eight-instruction turing-complete programming language. http://www.muppetlabs.com/~breadbox/bf/, 1993.

[NDT04] H. Nobach, N. Damaschke, and C. Tropea. High-precision sub-pixel interpolation in piv/ptv image processing. In *Proceedings of the 12th International Symposium on Applications of Laser Techniques to Fluid Mechanics*, Lisbon, Portugal, July 2004.

[Rie03] George Rieke. *Detection of Light – From the Ultraviolet to the Submillimeter*. Cambridge University Press, 2003.

[Rus98] John C. Russ. *The Image Processing Handbook*. CRC Press, Boca Raton, FL, USA, 1998.

[Smi05] Warren J. Smith. *Modern Lens Design*. McGraw-Hill, New York, NY, USA, 2005.

[St08] Richard M. Stallman and the GCC Developer Community. GNU compiler collection internals, 2008. http://gcc.gnu.org/onlinedocs/gccint.pdf.