**Analyse von Dateien und Dateiformaten**

**Fachbereichsarbeit Informatik**

**Abstract**

Diese Arbeit beschäftigt sich mit der Untersuchung von Inhalten von Dateien in Bezug auf ihr Dateiformat, ohne diese in ihr bestimmungsgemäßes Programm zu laden. So soll zum Beispiel anhand des PDF-Standards der Inhalt von PDF-Dateien untersucht werden ohne diese in ein PDF-Programm zu laden.

Das PDF Format wurde für diese Arbeit gewählt, weil es sehr viele verschiedenartige Informationen enthalten kann und durch das Einbetten von ausführbaren Inhalten (Java-Script) auch Möglichkeiten zum Missbrauch bietet.

Parallel zur theoretischen Betrachtung dieser Problemstellung, entsteht während der Arbeit ein Analysewerkzeug, dass Dateien, egal welchen Ursprungs, unabhängig von ihrem Dateiformat in binärer Form einließt und eine Analyse erlaubt ohne Gefahr zu laufen schadhafte Teile auszuführen.

**Vorwort**

Dateien in gespeicherter Form dienen primär zur Verarbeitung durch Software. Vor allem das Wiedergeben gespeicherter Informationen ist ein Vorgang, der jeden Tag millionenfach stattfindet.

Durch den Wunsch, auch herkömmliche Textdokumente mit zusätzlichen Interaktionen auszustatten, hat man sehr vielen Programmen die Möglichkeit gegeben, Aktionen dynamisch aufgrund des Dateiinhalts durchzuführen, also das Programm nicht nur zum Erstellen und Wiedergeben der gespeicherten Daten zu verwenden, sondern die Möglichkeit einer Interaktion ohne das Zutun des Benutzers zu schaffen.

Was auf der einen Seite eine sehr effiziente Erweiterung der Funktionalität für nützliche Prozesse ist, kann auf der anderen Seite auch erhebliche Gefahren für den Anwender, bzw. dessen System darstellen.

Ein Großteil der breit angelegten Angriffe auf Einzelnutzer und der gezielten Angriffe auf Unternehmen geschieht mithilfe so genannter Scripts, also Anweisungen die in herkömmlichen Dateien versteckt werden und Schadprogramme auf dem Zielsystem installieren, oder selber schadhaft sein können.

Viele solcher Scripts werden durch das Programm nicht direkt dargestellt und das herausfiltern der fraglichen Teile durch herkömmliche Texteditoren ist nicht zielführend, da diese öfters nicht in Form von darstellbaren Zeichen enthalten sind oder einfach der Umfang der Datei für eine konventionelle Analyse zu groß ist.

Der Ansatz der meisten konventionellen Virenscanner besteht darin, Dateien nach Signaturen bekannter Schädlinge zu durchsuchen. Was in den meisten Fällen, und vor allem für den Heimanwender, durchaus ausreichend ist, ist für andere Fälle bei Weitem zu wenig. Gerade bei „lohnenswerten Zielen“ wie Banken, staatlichen Einrichtungen oder größeren Firmen ist die Wahrscheinlichkeit eines gezielten professionellen Angriffes sehr groß. Die Werkzeuge die hier von Angreifern benutzt werden, sind meist Unikate, welche speziell für diesen Angriff ausgelegt sind oder dahingehend vorbereitet wurde. Da diese Werkzeuge noch unbekannt sind, werden sie von konventionellen Abwehrmechanismen meist nicht erkannt.

Um diesen Bedrohungen Herr zu werden, braucht es genau den Gegenteiligen Ansatz. Man ignoriert während der Analyse alles, das nicht gefährlich sein kann und lenkt seinen Fokus auf die Inhalte, die grundsätzlich ein Gefahrenpotential beinhalten (z.B. Skripts), unbekannte Inhalte (z.B. nicht zu klassifizierende Anhänge) oder Anomalien (Inhalte die offensichtlich nicht Bestandteil des jeweiligen Dateiformates sind).

Inhaltsangabe

[Grundlagen 4](#_Toc478940217)

[Textdateien 4](#_Toc478940218)

[Binärdateien 5](#_Toc478940219)

[Executables 8](#_Toc478940220)

[Byte Order 9](#_Toc478940221)

[Container 9](#_Toc478940222)

[Analysieren mit dem „Analyzer“ 11](#_Toc478940223)

[Eine Sache der Interpretation 12](#_Toc478940224)

[Quellenverzeichnis 13](#_Toc478940225)

# 

# Grundlagen

Generell kann man zwischen binär geschrieben Daten und Daten im Textformat unterscheiden. Allerdings ist es in vielen Fällen auch so, dass eine Mischform vorliegt. Dateien die mehrere, eventuell auch unterschiedliche, Dateien vereinen werden in dieser Arbeit als Container bezeichnet.

## Textdateien

Wenn man von Textdateien spricht, sind Dateien gemeint, in denen so genannte druckbare Zeichen benutzt werden. Die Bytes werden in Form definierter Zeichen interpretiert und können so als darstellbarer Text ausgegeben werden.

Der wohl am häufigsten vorkommende und einer der ältesten Zeichensätze ist der ASCII (American Standard Code for Information Interchange). Er definiert die 10 arabischen Ziffern, das lateinische Alphabet in Groß- und Kleinschreibung, einige druckbare Zeichen und einige nicht druckbare Zeichen, die meist als Steuerzeichen für Anwendungen verwendet werden, wie zum Beispiel ein Zeilenumbruch, Tabulator, … .

Insgesamt sind in diesem Standard 128 Zeichen definiert, welche über eine 7 Bit Codierung realisiert werden. Das 8. Bit des Bytes bleibt.



ABC als 7 Bit Code[[1]](#footnote-1)

Da man allerdings schon bei den Umlauten in der deutschen Sprache auf Probleme stößt, gibt es eine Reihe von Erweiterungen und Lösungen, bi denen sich ein Zeichen aus mehreren Bytes zusammensetzen kann. Man diese Formate auch Multibyte.

Der weltweite De-facto-Standard ist die UTF-8[[2]](#footnote-2) Codierung, vielfach auch als Unicode bezeichnet.

Bei dieser bleibt der ASCII Zeichensatz vollständig gültig. Werden mehr Bytes zur Darstellung benötigt, wir mit Hilfe der Anfangsbits des Bytes bekannt gegeben wie viele Bytes für die Definition des Zeichens verwendet werden.

|  |  |
| --- | --- |
| (0)0000000 | Einzelnes Byte wird als ASCII Zeichen interpretiert |
| (110)00000 (10)000000 | Unicode mit zwei Bytes, wobei im ersten Byte die Gesamtzahl der Bytes festlegt wird und weiteren Bytes mit 10 beginnen. |
| (11110)000 (10)000000(10)000000 (10)000000 | Zeichen mit 4 Bytes |

Die Bits in Klammern werden genutzt um die Anzahl der Bytes pro Zeichen festzulegen, die Bits außerhalb dienen zur Definition des eigentlichen Zeichens.

So können maximal 8 Bytes zur Definition eines Zeichens verwendet werden.

Bei der Analyse unbekannter Dateiformate kann man z.B. auf dieses Muster prüfen um herauszufinden um welchen Zeichensatz es sich handelt!

## Binärdateien

Anders als die Textformate verhält es sich mit Binärdateien. Hier kann jedes einzelne Bit durch die Anwendung ausgewertet werden, was eine Analyse sehr schwierig bis nahezu unmöglich macht.

Meist werden jedoch reguläre Größen für die jeweiligen Datentypen verwendet.

In der nachfolgenden Tabelle sind einige Beispiele für so genannte „native“ Datentypen aufgelistet:

|  |  |
| --- | --- |
| Allgemeine Bezeichnung (Datentyp gemäß C-Standard) | Größe |
| Byte (char) | 1 Byte |
| Int16(short) | 2 Byte |
| Wide Character (wchar\_t) | 2 Byte |
| In32 (int) | 4 Byte (auf 32 Bit-Systemen) |
| Floating Point(float/double) | 4, bzw. 8 Byte |
|  |  |

Man kann sehen, dass aufgrund der Größen nicht eindeutig festgestellt werden kann, um welchen Datentyp es sich handelt. Es ist Sache der jeweiligen Anwendung festzulegen ob 2 Byte als zwei einzelne Bytes in Form von Characters, als ein Unicode Character oder als short interpretiert werden.

Hinzu kommt, dass z.B. ein Byte bzw. char, nicht zwangsläufig als Zeichen verwendet wird, sondern auch bitweise verwendet werden kann um so genannte Flags zu speichern und dann ist es natürlich so, dass die obigen Typen, bis auf die Gleitkommawerte, vorzeichenbehaftet sein können.

Der große Vorteil im Speichern von Daten in binärer Form ergibt sich durch den Platzbedarf.

Muss z.B. die Zahl 1.000.000 gespeichert werden, ergibt sich bei der binären Speicherung eines Integers ein Platzbedarf von 4 Byte, Speichert man die Zahl als Text benötigt man 7 Byte.

Dieses kleine Beispielprogramm in C++ soll das kurz demonstrieren. Es werden zwei Dateien, einmal „asInt“ und einmal „asChar“ gespeichert. In der Datei „asInt“ wird der Wert als 4Byte Integer gespeichert, in „asChar“ als Text im ASCII Format.

#include <fstream>

int main(int argc, char argv[])

{

int asInt = 1000000;

char asChar[7]{'1', '0', '0', '0', '0', '0', '0'};

std::ofstream ofInt("c:/dev/fileSize/asInt");

//der int wird in ein char-Array kopiert

char intBytes[4]{0};

memcpy(&intBytes, &asInt, 4);

ofInt.write(intBytes, 4);

ofInt.flush();

std::ofstream ofChar("c:/dev/fileSize/asChar");

ofChar.write(asChar, 7);

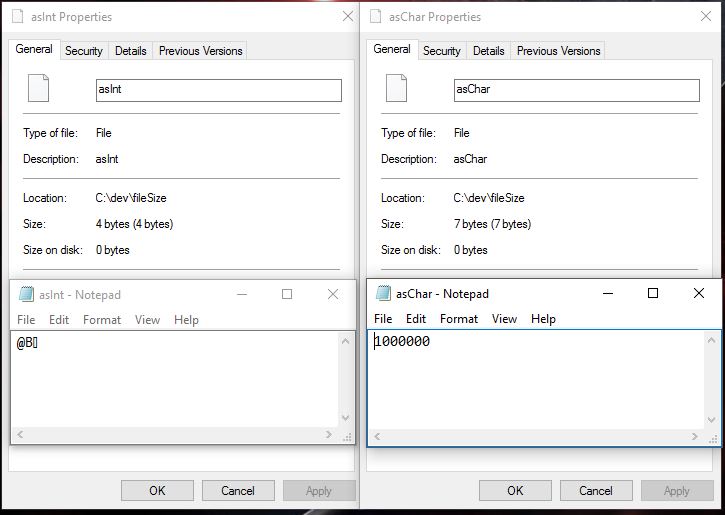
ofChar.flush();

return 0;

}

Das Lesen der Datei würde nun so funktionieren, dass man die jeweiligen Bytes in ihre gewünschten variablen kopiert. Dazu muss man allerdings wissen, dass z.B. in der Datei „asInt“ die ersten vier Bytes als Integer interpretiert werden müssen.

Das Ergebnis:



Wie erwartet sehen wir die entsprechenden Dateigrößen. Natürlich kann der Texteditor den Integer nicht interpretieren und wir sehen nur kryptische Zeichen, während die ASCII Version lesbar ist.

Die Tatsache, dass man bei binären Dateien nur schwer bis gar nicht erahnen kann, welche Rolle die einzelnen Bytes spielen, stellt uns bei der Analyse vor eine große Herausforderung.

Eine Möglichkeit ist, sich an veröffentlichte Standards zu halten, die definieren wie die Datei aufgebaut ist. Stehen diese Standards nicht zur Verfügung, kann man sich mit den sehr zeitraubenden Techniken des Revers Engineerings[[3]](#footnote-3) helfen.

Ein sehr bekanntes Beispiel für binäre Dateien ist das JPEG Format:

struct JpegHeader

{

char SOI[2]; //Start of Image Marker

char APP0[2]; //Use Marker

char Length[2]; //File Length

char Identifier[5]; //Id String

char Version[2]; //Format Revision

char Units; //Units for the X and Y densities

char Xdensity[2]; //Horizontal pixel density

char Ydensity[2]; //Vertical pixel density

char XThumbnail; //Thumbnail horizontal pixel count

char YThumbnail; //Thumbnail vertical pixel count

};

Der Header einer JPEG Datei als C++ Struktur[[4]](#footnote-4)

Beim Einlesen legt man eine Instanz dieser Struktur im Speicher an, liest die ersten 21 Bytes aus der Datei und kopiert sie in die Objekt Instanz.

### Executables

Eine Form der Binärdaten welche wir hier noch speziell beachten werden sind ausführbare Dateien. Sie liegen in einer Form vor, die vom System in den Speicher geladen und abgearbeitet werden kann, also alle Programme die auf dem Rechner laufen können.

In Windows Systemen erkennt man diese Dateien an den Endungen .exe, .dll, .lib oder .sys.

Sie sind so aufgebaut, dass sie eine Startkennung haben, die abhängig vom Betriebssystem ist und in weiterer Folge natürlich das eigentliche Programm, das von der jeweiligen Hardware Architektur verwendet wird.

MS Windows benutzt für ausführbare Dateien das Common Object Fileformat (COFF).

Offset Size Field Description

------ ---- ------------------- -----------------------------------------------

0 2 Machine Number identifying type of target machine.

2 2 Number of Sections Number of sections; indicates size of the

Section Table, which immediately follows

the headers.

4 4 Time/Date Stamp Time and date the file was created.

8 4 Pointer to Symbol Offset, within the COFF file, of the symbol

Table table.

12 4 Number of Symbols Number of entries in the symbol table.

This data can be used in locating the

string table, which immediately follows

the symbol table.

16 2 Optional Header Size of the optional header, which is

Size included for executable files but not

object files. An object file should have a

value of 0 here.

18 2 Characteristics Flags indicating attributes of the file.

COFF-Header[[5]](#footnote-5)

Die Programmlogik selber ist in Maschinensprache, in so genanntem „Opcode“ (operation code) geschrieben.

Bei der Analyse geht es darum, die Dateien auf die Evidenz von Opcodes zu prüfen. Ein gängiges Verfahren zum Einschleusen von Schadsoftware besteht darin, Teile dieser Programme in offensichtlich ungefährlichen Dateien zu übermitteln. Die Zielanwendung ignoriert diese Teile, weil sie sie nicht interpretieren kann und so bleiben sie meist unentdeckt.

### Byte Order

Eine Sache mit der man noch konfrontiert werden könnte und die hier deshalb kurz erwähnt werden sollte ist die Byte Order.

Diese legt fest, in welcher Reihenfolge die Bytes integraler Werte im System gespeichert ist.

Big-Endian setzt das höchstwertige Byte an die Anfangsadresse, also der kleinsten Speicheradresse, Little-Endian hingegen setzt das niederwertigste Byte an die Anfangsadresse.

Die meisten PCs verwenden als Basis die 8086er Assembly Struktur, welche Little-Endian benutzt, somit wird das, soweit es diese Arbeit betrifft, nicht zu Problem werden. Achtgeben sollte man jedoch insbesondere im industriellen Umfeld, da vor allem für maschinennahe Steuerungen gerne Big-Endian verwendet wird und eine Übersetzung nicht immer gleich erfolgt (z.B. in Pufferspeichern bei Echtzeitsystemen).

## Container

Dieses Dokument zum Beispiel ist im Programm MS Word© verfasst. Die Textbereiche inklusive ihrer Steuerzeichen sind im Textformat, genauer gesagt im XML Format, gespeichert, es enthält jedoch auch Bilddateien, die in binärer Form vorliegen. In diesem Fall handelt es sich um das **Open Office XML** Format[[6]](#footnote-6), das von der Firma Microsoft© entwickelt wurde. Es beinhaltet im Prinzip 2 Arten von Dateien, die als ZIP-Archiv[[7]](#footnote-7) zusammengefügt sind. So genannte „Parts“, die den Inhalt des Dokuments enthalten, und „Items“, die beschreiben wie der Inhalt dargestellt wird.

Zur Demonstration habe ich die Endung dieser Datei von .docx in .zip umbenannt und sie mit dem Programm 7zip[[8]](#footnote-8) entpackt. Das Resultat sieht man in Bild 1-1.

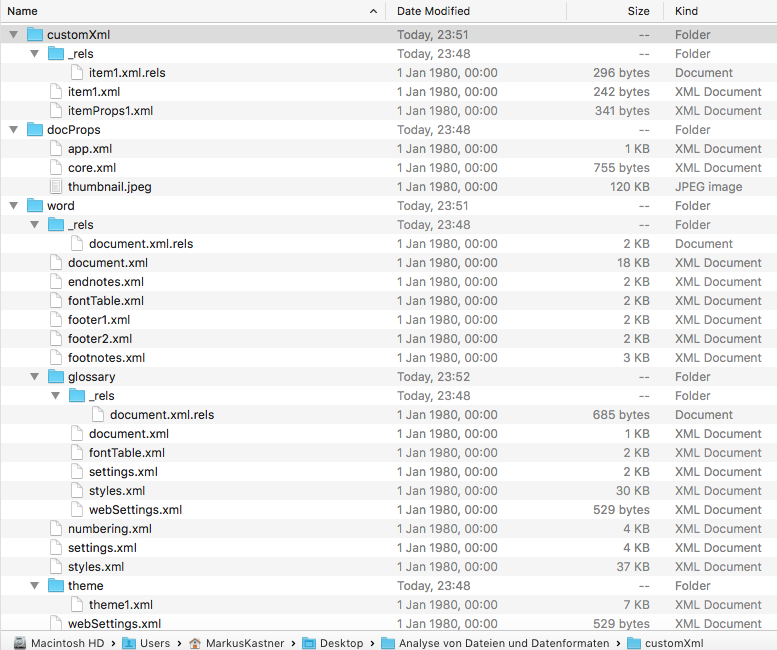


Bild 1-1 Inhalt einer .docx Datei

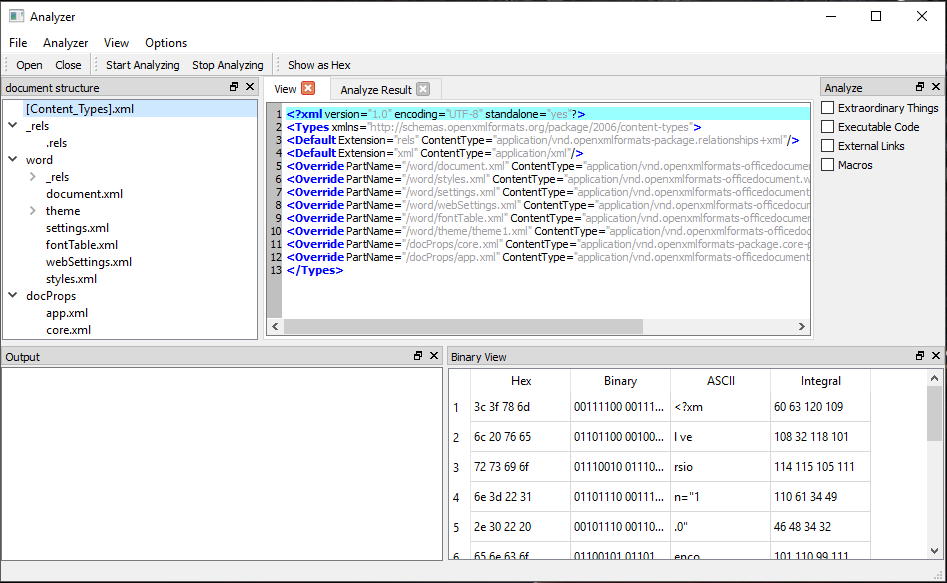
Der Text befindet sich in der Datei „document.xml“. Zur Veranschaulichung hier der Anfang bis zur Überschrift des Dokuments:



Bild 1-2 Auszug aus dem XML-formatierten Textteil eines Word Dokuments

Auf das Thema XML (Extensible Markup Language)[[9]](#footnote-9) wird in dieser Arbeit nicht näher eingegangen, da es sicher hierbei lediglich um einen Standard zur Textformatierung handelt. Das einzige, das unser Interesse wecken wird, sind Verweise auf andere Dokumente.

# Analysieren mit dem „Analyzer“

Parallel zu dieser Arbeit erstelle ich ein Programm das es erlaubt, Dateien in verschiedenen Formaten zu betrachten. So kann der Inahlt unabhängig von der eigentlichen Zielanwendung untersuchen und einen eingehender Blick auf Aufbau und enthaltene Informationen geworfen werden.

Der „Analyzer“ beim Öffnen eines MS Word Dokuments

Ziel des Analyzers ist es, alle Inhalte einer Datei einzulesen und unbekannte Teile, Teile die Unregelmäßigkeiten enthalten und Teile die als Bedrohung erkannt wurden zu lokalisieren. Darüber hinaus soll sich der Anwender einen Überblick über die Gesamtstruktur der Daten verschaffen und einzelne oder Sequenzen von Bytes in verschienen Interpretationen betrachten können.

Technische Eckdaten:

Sprache: C++

Bibliotheken: Google Test Framework (gtest)

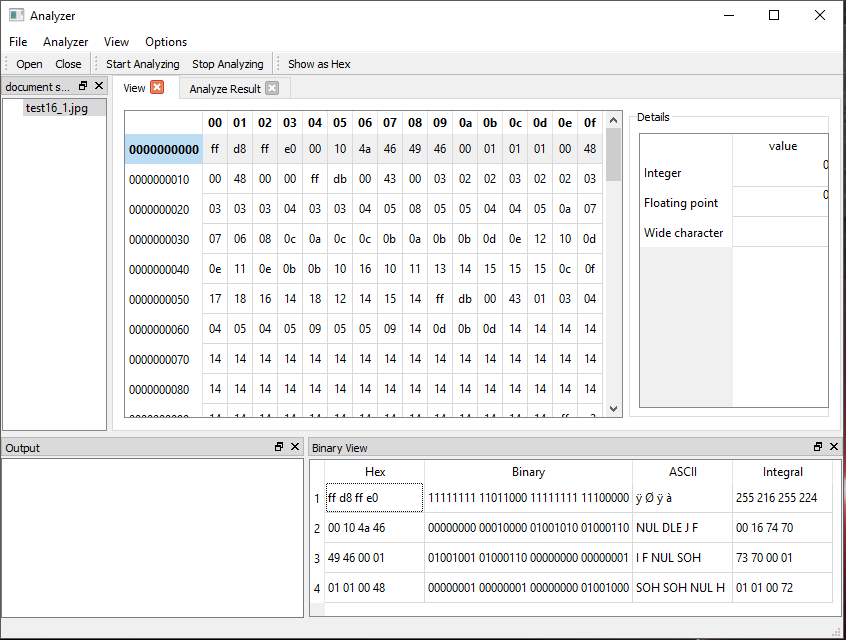
Qt (ausschließlich für die graphische Oberfläche)

IDE: VisualStudio 2015 Pro

Das Projekt wird unter <https://github.com/MarkusKastner/Analyzer> verwaltet.

# Eine Sache der Interpretation

Wie eingangs schon erwähnt, werden binäre Daten vom jeweiligen Programm interpretiert. Da die Datei aber unabhängig des Zielprogrammes gelesen werden soll und das Format oft nicht klar, bzw. bei der Analyse auf schadhafte Teile auch weitgehend irrelevant ist, braucht man eine Möglichkeit die Daten anders lesbar zu machen. Diese Aufgabe kommt dem „Hex-Editor zu.



Öffnen der Datei einer .jpg Datei in hexadezimaler Darstellung.

Alle Daten die in der Datei „test16\_1.jpg“ enthalten sind, wurden in das Programm geladen und werden in einem Raster als Hexadezimalwerte angezeigt. Da man sich darunter aber eher weniger vorstellen kann, werden die selektierten Werte in verschiedenen Interpretationen angezeigt.

Im Fenster unten rechts ist eine Standartansicht. Alle selektierten Bytes werden in Hex, binär, als ASCII Code und als numerischer Wert angezeigt.

Im Fenster „Details“ rechts des Rasters werden, so es eine Interpretation gibt, die möglichen Interpretationen der selektierten Bytes angezeigt.

Eine Interpretation, bzw. Reinterpretation von Daten nennt sich Typumwandlung, oder auf Englisch Typecasting.

Typumwandlungen für primitive Datentypen wie int, double oder wchar\_t hat die Programmiersprache C++, und natürlich alle anderen Hochsprachen, standartmäßig und diese kommen im Analyzer auch zur Anwendung. Umwandlungen zu komplexeren Typen müssen allerdings implementiert werden.

Wenn von einer „Umwandlung“ die Rede ist, heißt das nicht, dass die Daten selber verändert werden, es wird lediglich die Interpretation geändert, die „Bits und Bytes“ bleiben gleich.

## reinterpret\_cast und memcpy

Zwei Operationen kommen bei Typumwandlungen in C++ sehr oft vor. Das ist zum ersten die durch den C++ Standard definierte Funktion reinterpret\_cast und die, aus der Sprache C stammende, memcpy Funktion.

Ein Array von 2 chars, also insgesamt 16 Bit, wird mit reinterpret\_cast in einen integralen 16 Bit Typen „umgewandelt“. Man beachte, dass hier kein 16 Bit typ im Speicher anglegt wird, sondern nur ein Zeiger auf das erste Byte des Arrays:

char buffer[2];

buffer[0] = 0;

buffer[1] = 0;

int16\_t \* value = reinterpret\_cast<int16\_t\*>(&buffer);

Ein wenig anders verhält es sich bei der Funktion memcpy. Hier wird erst ein Speicher angelegt und diesen wird das Array dann physikalisch hineinkopiert.

char buffer[2];

buffer[0] = 0;

buffer[1] = 0;

int16\_t value;

memcpy(&value, buffer, 2);

## RGB Farben

Das RGB Frabmodel ist ein additives Farbmodel. Das heißt, dass die jeweiligen Farbtöne von rot, grün und blau übereinandergelegt werden und sich aus dem Resultat der Farbton ergibt.

Für rot grün und blau steht jeweils ein Byte zur Verfügung, somit hat ein RGB Datensatz 3 Byte.

In numerischen Werten, die des Öfteren in Editoren für Unser Interfaces zu finden sind würde das so aussehen:

[255][0][0] … ergibt ein sattes rot

[0][0][0] … ist schwarz

…

## Opcodes

Ein Opcode hat in den meisten Fällen folgenden Aufbau:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Byte | 2. Byte | 3.&4. Byte | 5.&6. Byte |
| Opcode | Adressmodus | Adresse | Daten |

Opcode:

Bit 2 bis 7 stehen für die Bezeichnung des Codes. Bit 1 entscheidet ob die nachstehende Registeradresse als Ziel oder Quelle der Operation verwendet wird. Bit 0 gibt an ob es sich um eine 8 oder 16 Bit Operation handelt.

# Quellenverzeichnis

**„Das PDF Howto“:** Eine Seite von Christof Bürgi, zu finden unter

http://www.p2501.ch/pdf-howto/

1. https://de.wikipedia.org/wiki/American\_Standard\_Code\_for\_Information\_Interchange [↑](#footnote-ref-1)
2. https://tools.ietf.org/html/rfc3629 [↑](#footnote-ref-2)
3. Revers Engineering bezeichnet das Verfahren ein erstelltes Produkt zurück zu entwickeln. [↑](#footnote-ref-3)
4. https://www.w3.org/Graphics/JPEG/jfif3.pdf [↑](#footnote-ref-4)
5. https://support.microsoft.com/de-de/kb/121460 [↑](#footnote-ref-5)
6. https://de.wikipedia.org/wiki/Office\_Open\_XML

   https://msdn.microsoft.com/de-de/office/bb906068.aspx [↑](#footnote-ref-6)
7. https://de.wikipedia.org/wiki/ZIP-Dateiformat [↑](#footnote-ref-7)
8. http://www.7-zip.org/ [↑](#footnote-ref-8)
9. https://de.wikipedia.org/wiki/Extensible\_Markup\_Language [↑](#footnote-ref-9)