QR-Codes lesen – mit Stift & Papier

Einführung in den Aufbau von QR-Codes für Neugierige

Armin Hanisch

Mail: mail@arminhanisch.de | Mastodon: @Linkshaender@bildung.social | www.ArminHanisch.de

Den QR-Code unten werden wir lesen, ohne Computer, Smartphone oder eine Scanner-App. Sondern wie Code-Archäologen mit Stift, Papier und etwas Kopfrechnen. Kann ja nicht so schwer sein, wenn ein dummes Telefon das lesen kann, oder? 😀

Hinweise zu diesem Tutorial

Ziel

Warum will jemand QR-Codes „zu Fuß“ decodieren, wenn es doch auf fast jedem Smartphone eine App gibt, mit der diese Codes gelesen werden? Es gab mehrere Gründe für dieses analoge Tutorial zu einem digitalen Thema.

- Erkennung grundlegender informatischer Prinzipien anhand einer praktischen Anwendung
- Die Vermittlung digitaler Kompetenzen erfordert nicht digitale Lehrmittel
- Motivierende Lernmöglichkeit für Schülerinnen und Schüler
- OER-Material für Lehrerinnen und Lehrer (QR-Codes sind „hip“ im Lehrbetrieb)
- „Weil ich es kann“ 😉 – Selbstermächtigung und Technikmündigkeit

Voraussetzungen

Was muss ich wissen oder können, um dieses Tutorial durcharbeiten und verstehen zu können?

- QR-Codes schon mal gesehen und/oder mit einer App gelesen haben
- Wissen, was Bits sind und wie binäre Zahlen in das Dezimalsystem umgerechnet werden
- Grundlegendes Verständnis, wie Zeichen kodiert abgelegt werden („ASCII-Codes“ etc.)
- Neugier und Bleistift oder Farbstifte 😉

Aufbau des Tutorials

Dieses Tutorial ist schrittweise aufgebaut, um beispielsweise über mehrere Unterrichtseinheiten hinweg durchgearbeitet zu werden. Dieser Ansatz hat sich bei Workshops bewährt, aber natürlich kann das Material hier jederzeit in anderer Reihenfolge präsentiert oder bearbeitet werden.

Hinweis für die IT-Profis & QR-Code Spezialisten

Ja, ich bin mir bewusst, dass dieses Tutorial einige Aspekte vereinfacht darstellt. Wir betrachten in diesem Tutorial auch nicht das Thema Fehlerkorrektur oder das Erstellen von QR-Codes. Das ist auch nicht Ziel des Tutorials. Einige Begriffe führe ich dann ein, wenn sie benötigt werden, einige Details (z.B. Versionsblöcke bei QR-Codes ab Version 7) lasse ich weg. Das Tutorial soll für einen „Aha!-Effekt“ sorgen, Interesse wecken und die Grundlagen aufzeigen.

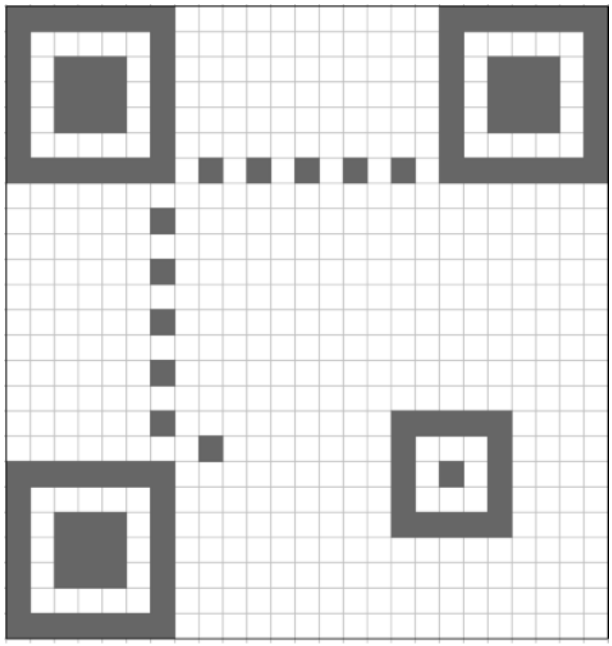
Verwendung des Materials / OER-Einsatz

Der grundlegende Aufbau eines QR-Codes

Sieht man sich QR-Codes an, dann fällt auf, dass bei allen Code einige Elemente gleich sind, ganz gleich, ob es sich um gedruckte Codes, Grafiken oder sogar Tattoos handelt.

Bilder: Mike Caruso, http://mikejcaruso.blogspot.com und Nikhil Jhingan auf quora.com

Das sind beispielsweise die drei großen Quadrate. Manche Codes (wie der unsere) enthalten auch noch ein oder mehrere kleine Quadrate. Wer genauer hinsieht, erkannt, dass zwischen den großen Quadraten immer eine Linie verläuft, die abwechselnd hell und dunkel ist. Ich habe das für den Beispiel-Code links farblich hervorgehoben und rechts diese Strukturen freigestellt.



Welche Bedeutung haben diese einzelnen Elemente? Sehen wir sie und der Reihe nach an …

Module

Module (*modules*) werden die Datenpunkte/Datenblöcke genannt. Pixel wäre irreführend, da ein Modul aus mehreren Pixels besteht, wenn der Code größer ist. Also Datenpunkt = Modul.

Positionsmarker (die großen Quadrate)

Sie zeigen an, in welcher Richtung der Code gedruckt ist (um ihn bei Bedarf drehen zu können). Wir werden gleich noch sehen, dass diese Ausrichtung wichtig ist.

Ausrichtungsmarker (das kleine Quadrat)

Je größer der QR-Code ist, um so mehr dieser Marker sind enthalten, um die Prüfung der Ausrichtung des Codes zu erleichtern.

Taktzellen / Timing-Marker (die abwechselnd schwarz-weißen Linien)

Diese Felder dienen dem Barcode-Leser dazu, die Größe der Matrix festzustellen und somit auch die Größe eines „Modules" (eines Datenpunktes).

Diese Elemente tragen keine Daten und werden beim Auslesen des Codes nicht benötigt. Sie sind allerdings unerläßlich, um den Code dauber lesen zu können. Oder aber, um festzustellen, ob der Code zu verzerrt ist, dass ein Lesen nicht möglich ist.

Formatinformation

Zusätzlich existieren noch zwei Formatfelder, die in der Abbildung rechts rot/gelb markiert sind. Dabei handelt es sich um zwei Kopien mit je 15 Bits, die die gleiche Information tragen. Einmal um den linken, oberen Positionsmarker herum und einmal rechts neben dem unteren Positionsmarker (Bits 0 bis 6) und unter dem rechten Positionsmarker (Bits 7 bis 14). Diese Felder enthalten Informationen über Fehlertoleranz und die **Datenmaske** (keine Angst, erkläre ich etwas weiter unten).

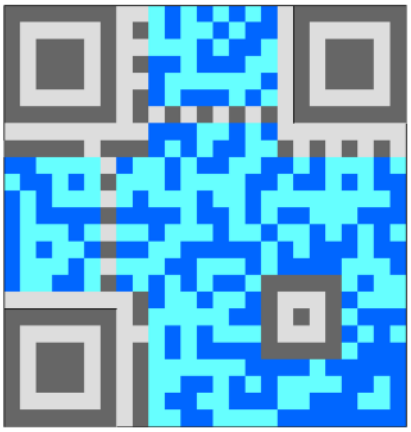
Dark Module

Das ist der etwas einsam liegende Punkt rechts vom unteren großen Quadrat.

Ruhezone

Zusätzlich benötigt ein QR-Code noch eine „**Ruhezone**" um den Code, damit dieser vom Hintergrund getrennt werden kann. Diese muss so breit sein wie vier Datenpunkte.

Damit bleibt als **Datenbereich** (den wir lesen werden) der Bereich übrig, den ich in der Abbildung rechts blau hinterlegt habe. In diesem **Datenbereich** werden die Zeichen abgelegt, die im QR-Code enthalten sind. Der grau markierte Rest ist also „Verwaltung & Infrastruktur" und trägt keine Daten.



Decodierung Teil 1 – Formatfelder

Formatbits finden

Nachdem der grundlegende Aufbau klar ist, so dass wir uns im QR-Code „zurecht finden", können wir mit der Decodierung beginnen. Dazu muss zuerst die Formatinformation gelesen werden. Ich habe den Teil, der um den linken, oberen Positions-Marker herum geht, an dieser Stelle „gerade gebogen", damit wir den Inhalt leichter lesen können.

Wichtig! In diesem Bereich finden sich auch zwei Punkte der Timing-Marker, diese dürfen nicht mit gelesen werden (waren ja auch oben grau markiert, weil sie eben nicht zum Datenbereich gehören). Aus diesem Grund zeigt die Abbildung unten auch nur 15 Bits (17 Kästchen rundherum und zwei davon gehören zu den Timing-Markern).

Für uns interessant sind nur die ersten fünf Bits (die in unserem Beispiel praktischerweise alle den Wert „1" besitzen, so dass die leicht erkennbar sind). Diese fünf Bits können wir aber nicht direkt verwenden, sondern müssen die erst konvertieren.

Fehlerkorrektur- und Masken-Bits berechnen

Das geschieht, indem diese fünf Bits mit dem Wert `10101` mit einer **Exklusiv-Oder** Operation kombiniert werden. ***Exklusiv-Was-Bitte?*** 🤔 Das ist einfacher, als es sich anhört. Exklusiv-Oder bedeutet, dass ein Bit **oder** das andere, aber **nicht beide** den Wert 1 haben dürfen, damit als Ergebnis wieder eine 1 entsteht. Beispiel dafür aus dem realen Leben: man kann entweder mit dem Rad ODER mit dem Auto in die Berge fahren, aber nicht mit beiden gleichzeitig. 😉 Haben beide Bits den Wert 1, dann ist das Ergebnis ebenfalls 0.

0 0 ➡ 0 1 0 ➡ 1 0 1 ➡ 1 1 1 ➡ 0

Um aus unseren fünf Einsen also den wirklichen Wert zu erhalten, führen wir diese Operation jetzt für jedes Bit aus:

```
    1 1 1 1 1
XOR 1 0 1 0 1
    ----------
=   0 1 0 1 0
```

Die ersten beiden Bits geben den Fehlerkorrektur-Level an. Das gibt an, wie viel Prozent eines QR-Codes zerstört sein dürfen und der Code trotzdem noch lesbar bleibt:

L = 01 (ca. 7%) / M = 00 (ca. 15%) / Q = 11 (ca. 25%) / H = 10 (ca. 30%)

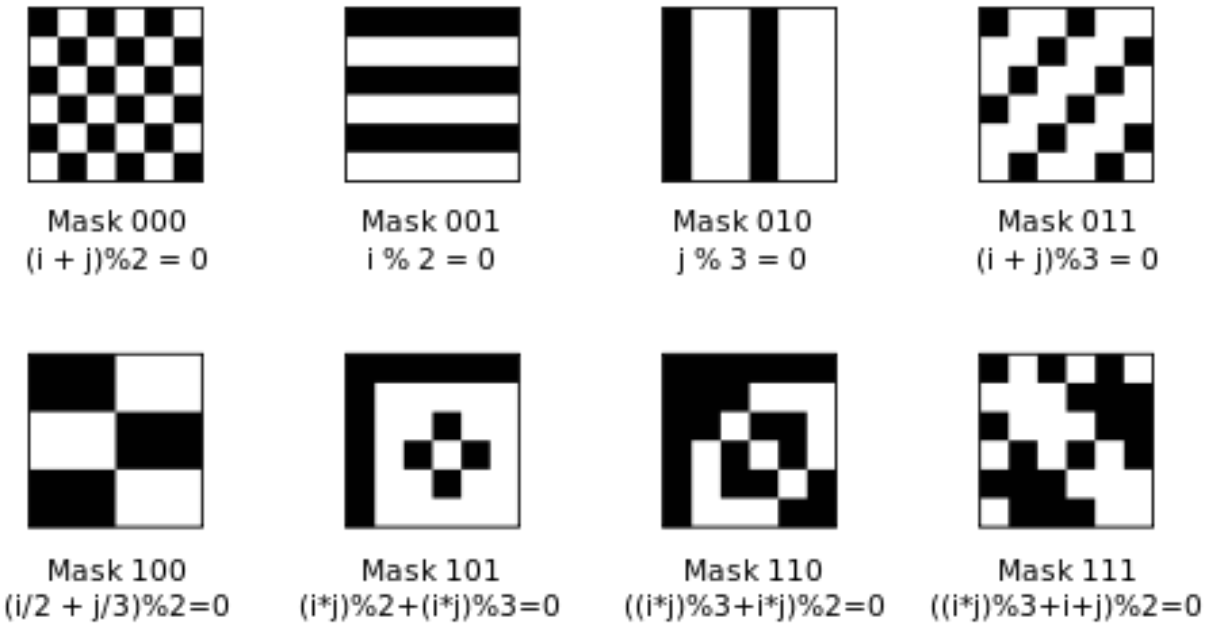
Interessante Information, aber derzeit benötigen wir diese nicht, wir sind an den restlichen Bits interessiert …

Datenmaske feststellen

Die restlichen drei Bits definieren die **Datenmaske**. Diese Information ist wichtig, denn die benötigen wir für das Decodieren des Codes. Aber wieso ist die **Datenmaske** denn notwendig? Sie soll verhindern, dass große, gleichmäßig helle oder dunkle Flächen in einem QR-Code entstehen, denn das kann den Lesevorgang stören oder unmöglich machen. Dazu werden alle Daten in einem QR-Code nochmals mit einem Punktemuster überlagert, eben der Datenmaske. So werden große, einfarbige und strukturlose Gebiete verhindert.

Diese Masken sind Muster aus je 6x6 Datenpunkten, die über den kompletten QR-Code gelegt werden, aber nur für den Datenbereich des QR-Codes gelten. Immer, wenn ein gesetzter (schwarzer) Datenpunkt der Maske auftritt, wird der Wert des Datenpunktes umgedreht (invertiert). Aus schwarz (1) wird also weiß (0) und umgekehrt.

In unserem Beispiel ergibt sich der Wert `010` binär (also 2 in dezimal). Wir haben drei Bits für die Maske, damit können wir Werte von 000 bis 111 erzeugen (8 Werte). Hier sind die möglichen Muster für die Datenmasken. Dabei steht i für die Zeilennummer (beginnen ab 0!) und j für die Spaltennummer (beginnend ab 0!). Der Operator „%“ ist der Modulo-Operator (der Rest einer Division, kennt eigentlich jeder noch aus der Schule).



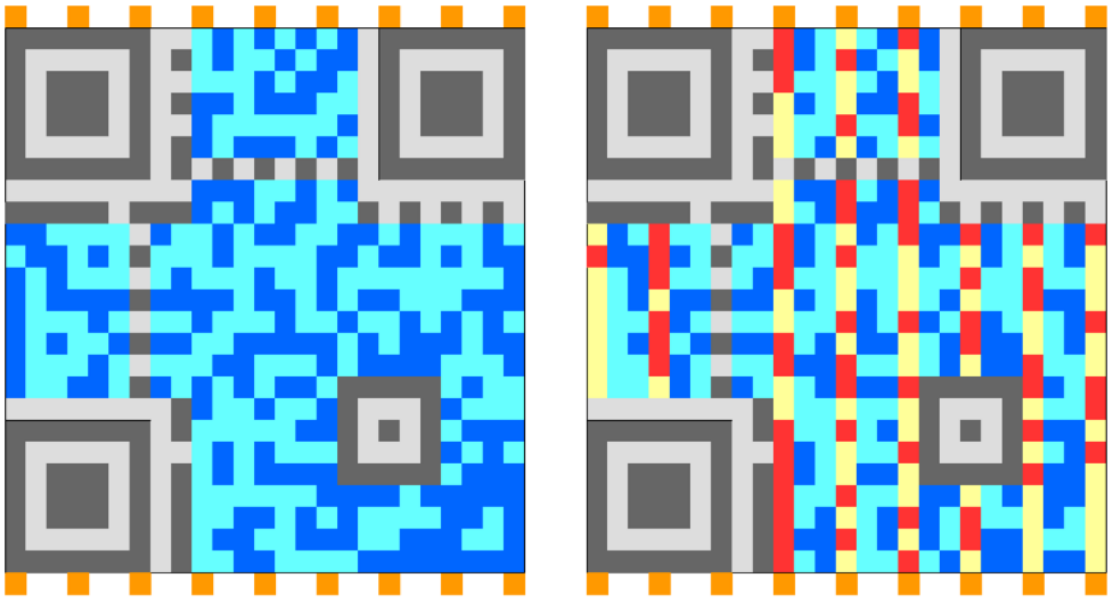
Ich habe für das Beispiel die Datenmaske 2 gewählt, denn die ist relativ einfach abzubilden und umzurechnen, denn wir müssen einfach nur die Datenpunkte in jeder dritten Spalte invertieren. Das besagt der math. Ausdruck `j % 3 = 0` – immer wenn die Nummer der Spalte ohne Rest durch 3 geteilt werden kann, muss die Maske angewendet werden. Sehen wir uns das auf dem Beispiel-Code einmal an …

Decodierung Teil 2 – Maske anwenden

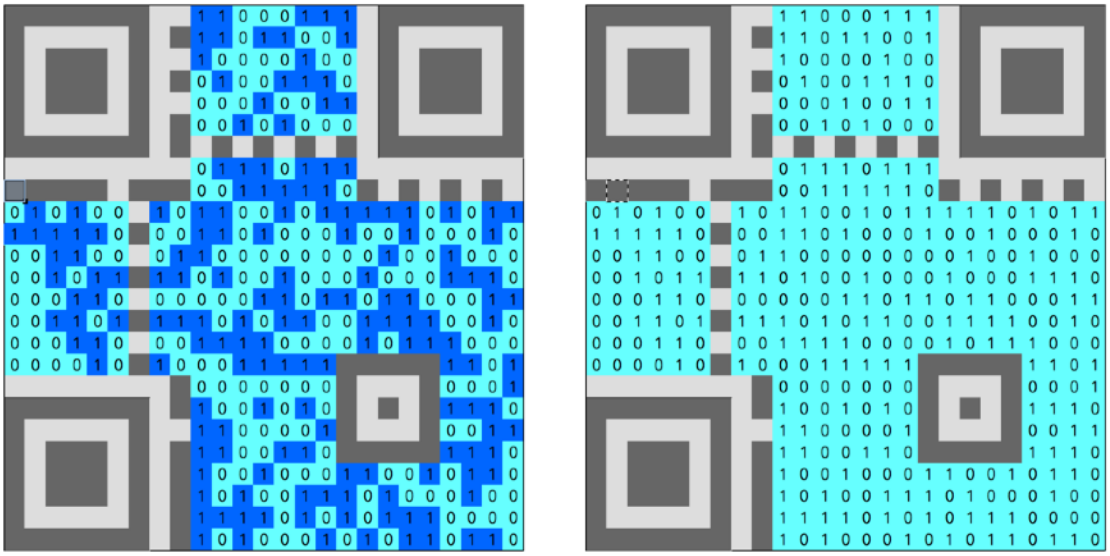
Für unser Beispiel müssen wir also die Werte in jeder dritten Spalte invertieren. Ich habe in der Abbildung unten jeweils die entsprechenden Spalten oberhalb und unterhalb des QR-Codes mit einem orangefarbenen Punkt markiert.

Wichtig! Bitte darauf achten, dass nur die Werte invertiert werden, die im (türkis gefärbten) Datenbereich liegen. Alles grau hinterlegte darf nicht geändert werden!

Die linke Abbildung ist der originale QR-Code. Die Abbildung rechts zeigt die umgedrehten Bits in jeder dritten Spalte (rot=1 und gelb=0).

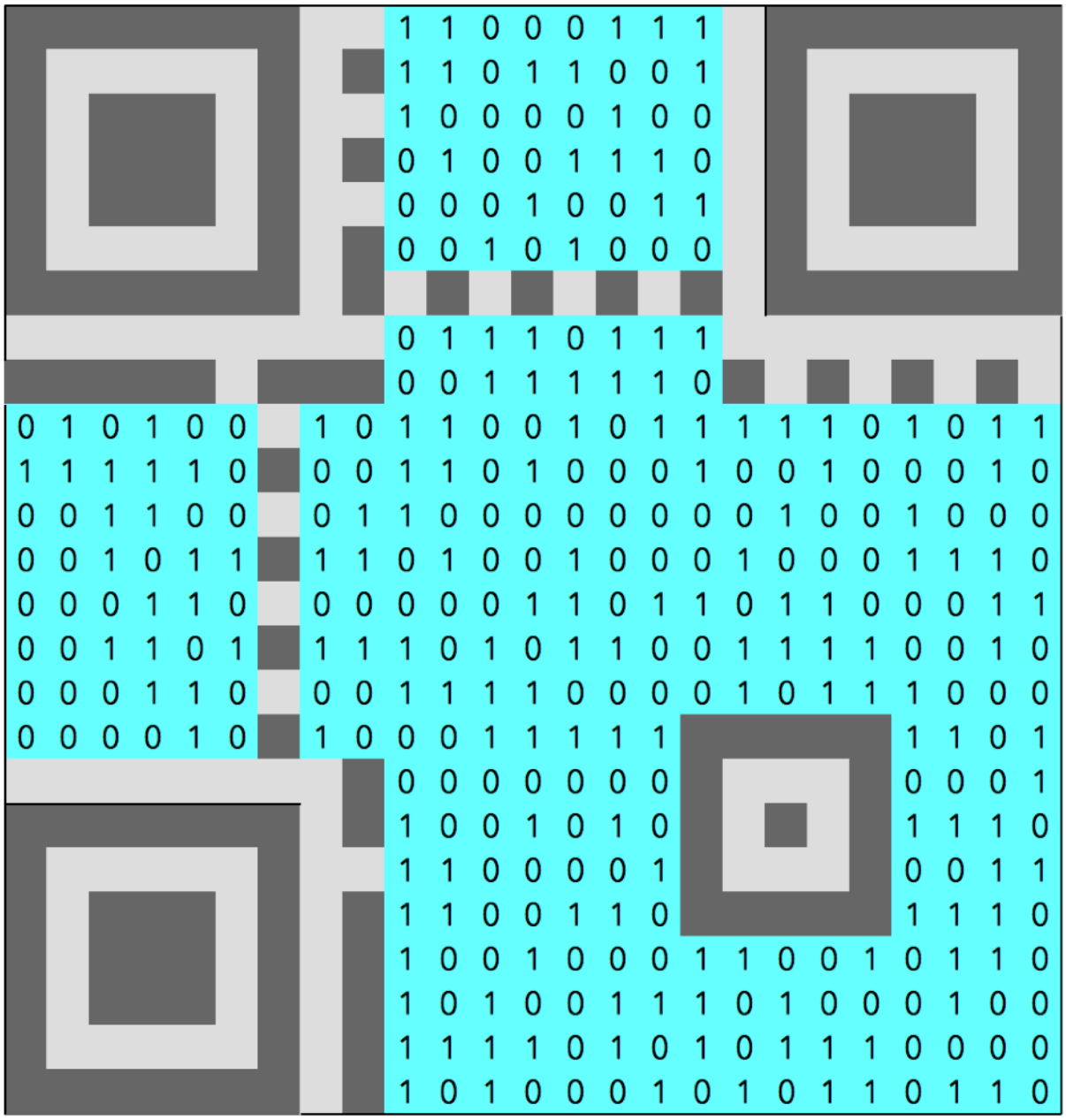


Damit wir es etwas leichter haben, trage ich nun für die Farben die Bitwerte (0 oder 1) ein, wie in der Abbildung links unten zu sehen. Anschließend wird der Datenbereich einheitlich eingefärbt, denn jetzt haben wir die Nullen und Einsen direkt in den Datenpunkten.



Nochmal in Nahaufnahme, bitte!

Nun haben wir endlich die tatsächlichen Bits, in denen die Daten stecken, die wir aus dem Code lesen wollen. Um darin auch gut lesen, herum malen und Anmerkungen anbringen zu können, hier nochmal der fertig konvertierte QR-Code mit markiertem Datenbereich in Großdarstellung.



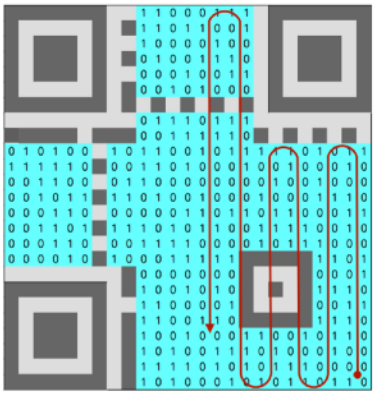
Dieser Code sollte am Besten abgedruckt werden, dann lässt sich leichter darauf zeichnen.

Jetzt kommt als nächster Schritt das Feststellen, in welcher Form die Daten da überhaupt im Code stecken (sind es nur Zahlen, Buchstaben oder irgendwelche Daten wie eine URL). Dazu muss auch geklärt werden, in welche Reihenfolge diese Bits aus dem Code gelesen werden.

Decodierung Teil 3 – Leserichtung & Datenlänge

Die Leserichtung eines QR-Codes

Begonnen wird mit dem Lesen eines QR-Codes immer rechts unten. Damit wissen wir auch, warum das Erkennen der Positionierung über die Positions-Marker so wichtig ist. Anschließend werden die Daten immer in zwei Bit breiten Bahnen schlangenförmig gelesen.



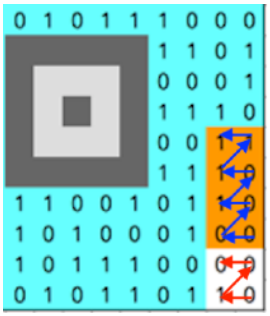
Dabei wird (weil wir ja rechts unten beginnen), dieses Bit zuerst gelesen, dann das Bit links daneben, dann geht es wieder eine Reihe hoch zum rechten Bit, dann wieder das Bit links daneben und so weiter. Wenn es „um die Kurve" geht und wir von oben nach unten lesen, bleibt es trotzdem dabei, dass immer zuerst das rechts Bit gelesen wird. Das führt zu einer Art „Zick-Zack-Muster" beim Lesen des Codes. Beim Lesen darf allerdings nur der Datenbereich verarbeitet werden (hier türkis hinterlegt). Treffen wir beim Lesen auf einen Nicht-Daten-Bereich, dann überspringen wir den einfach.

Inhaltstyp feststellen

Mit dieser Information können wir feststellen, wie diese Bits kodiert wurden, also welche Inhalte in unserem QR-Code stecken. Die drei wichtigsten Typ-Codes in Binärdarstellung:

- 0001 = Numeric (der QR-Code enthält nur Ziffern)
- 0010 = Alphanumeric (nur `0–9`, `A–Z`, Leerzeichen, `$`, `%`, `*`, `+`, `-`, `.`, `/`, `:`)
- 0100 = Byte encoding (Zeichen im Zeichensatz ISO 8859-1)

Weitere Informationen zu den Typen finden sich z.B. in der Wikipedia: https://en.wikipedia.org/wiki/QR_code



Diese Information finden wir im Feld für den Kodierungstyp (*encoding type*). Platziert sind diese Daten in den vier ersten Bits, also unten rechts. Die Grafik rechts zeigt den Ausschnitt aus unserem Code, bei dem diese vier Bits weiß hinterlegt sind. Da wir oben gelernt haben, wie die Bits gelesen werden, erhalten wir folgende Bitfolge: 0 1 0 0 (dezimal 4).

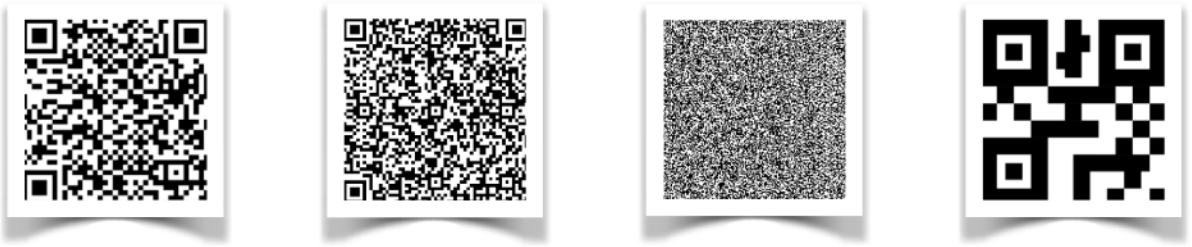
Wir haben also einen QR-Code vor uns, bei dem die Daten Byte-kodiert sind. Für jedes Datenwort müssen wir also immer 8 Bits zusammenfassen. Abhängig vom Datentyp sind die Datenwörter verschieden lang:

- Bei numerischer Kodierung 10 Bits (immer drei Ziffern werden in 10 Bits codiert)
- Bei alpanumerischer Kodierung 11 Bits (je zwei Zeichen werden in 11 Bits codiert)
- Bei Koderierung in Bytes 8 Bits, da ein Byte aus **8 Bits** besteht

Bevor wir weitermachen können, müssen wir uns kurz mit dem Begriff der „**Version**" eines QR-Codes beschäftigen, dann können wir weitermachen.

QR-Code Versionen

QR-Codes existieren in vielen verschiedenen Größen, aber das allein ist nicht das einzige Unterscheidungskriterium. Wichtig ist vor allem, wie viele Datenpunkte und damit Bits ein QR-Code enthält. Die Abbildung unten zeigt vier verschiedene QR-Codes, die alle gleich groß sind, aber unterschiedliche Datenmengen enthalten. Mit „Größe" ist also hier die Anzahl der Datenpunkte (Module) gemeint und nicht die Druck- oder Anzeigegröße.



Der kleinste QR-Code besitzt 21x21 Module und hat die Version 1. Die maximale Kapazität an Daten hängt von der Fehlerkorrektur-Einstellung ab (ja, da war was zu Beginn dieses Tutorials!): die folgende Tabelle zeigt die Datenkapazität für einen Version 1-Code (21x21 Module).

ECC	Bits
L	152
M	158
Q	104
H	72

Wird eine größere Kapazität benötigt, benötigen wir mehr Module. Mit jeder Version werden sowohl in X- als auch in Y-Richtung vier Module angefügt. Wir können also ganz leicht die „Version" eines QR-Codes auszählen. Die Anzahl der Datenpunkte an einer Kante minus 17 und dieses Ergebnis durch 4 teilen.

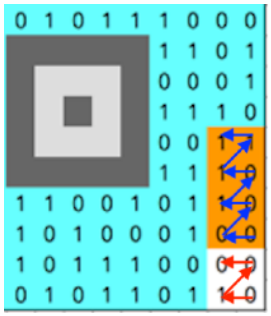
Beispiel: 25x25 => 25 -17 = 8 => 8 / 4 => 2 => Version 2

Die maximale Versionsnummer ist übrigens 40. Damit lassen sich bei einer hohen Fehlertoleranz von „Q" (bei der 25% des Codes zerstört oder unlesbar sein dürfen) in Byte-Kodierung 1663 Zeichen oder in alphanumerischer Codierung 2420 Zeichen oder aber 3993 Ziffern bei numerischer Codierung speichern.

Hinweis: Die komplette Tabelle liegt unter der Adresse http://www.qrcode.com/en/about/version.html

Warum war das wichtig? Weil die Anzahl der Bits für das Feld mit der Datenlänge von der Version des QR-Codes abhängt. Daher müssen wir zuerst wissen, das die Version die Anzahl der Module angibt, bevor wir weiter machen können. Jetzt, da wir wissen, dass unser Beispiel-Code die Version 2 besitzt, können wir auch mit dem Decodieren weiter machen …

Zeichenanzahl im QR-Code feststellen



Die Bits für die Anzahl der Datenwörter folgen gleich nach den Bits für den Datentyp. Diese Anzahl, das haben wir gerade gelernt, hängt von der Version des QR-Codes ab. Die Tabelle unten gibt an, wie viele Bits für das Längenfeld verwendet werden müssen.

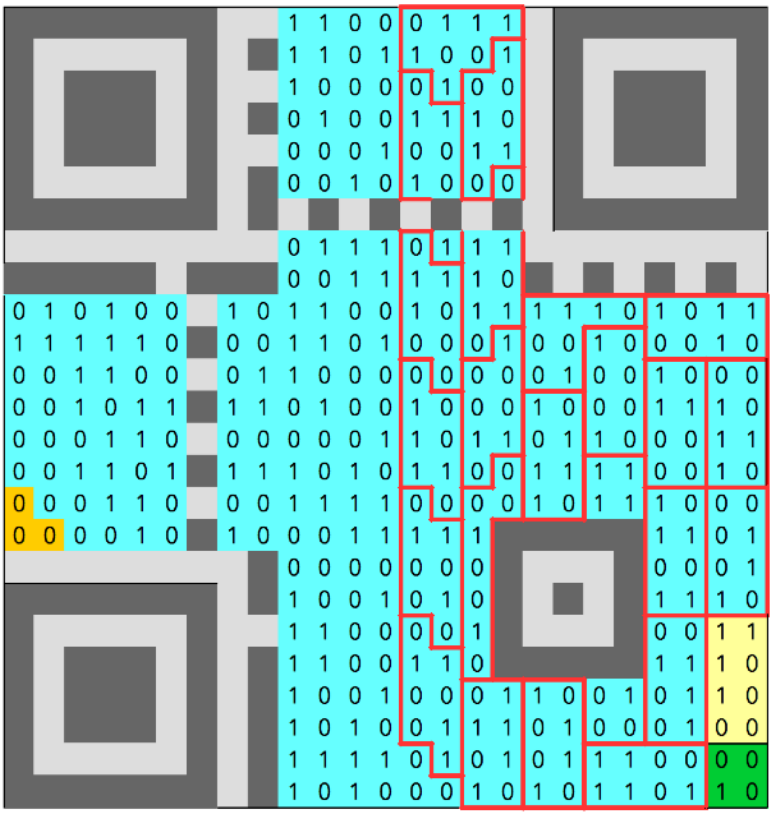
	1–9	10–26	27–40
Numerisch	10	12	14
Alphanumerisch	9	11	13
Bytes	8	16	16

Für unseren Version 2 Code haben wir den Datentyp „Byte", also benötigen wir für das Längenfeld 8 Bits. Diese sind in der Grafik rechts orangefarben hinterlegt (und wieder mit Pfeilen für das Lesen der Bits versehen). Wem die Darstellung zu klein ist, ein paar Seiten weiter oben gab es den kompletten Code in Großdarstellung.

Wenn wir die Bits wie gezeigt hintereinander schreiben, erhalten wir `00010111`, das sind:

0 x 128 + 0 x 64 + 0 x 32 + 1x 16 + 0 x 8 + 1 x 4 + 1 x 2 + 1 x 1 => 23. Wir haben 23 Zeichen!

Diese 23 Datenwörter mit je 8 Bit müssen wir jetzt dem Lesemuster folgend lesen. Da wir nur im Datenbereich lesen dürfen, sind die Blöcke je nach QR-Code nicht immer schön rechteckig. Ich habe das für unseren Beispiel-Code in der Abbildung unten eingezeichnet.

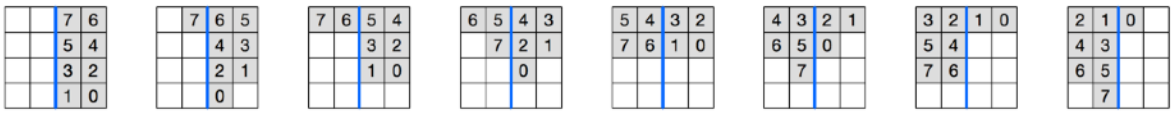


Decodierung Teil 4 – Datenwörter lesen

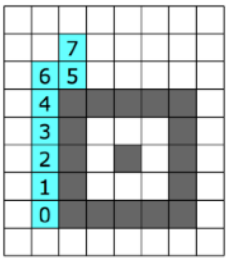
Datenwort-Formen sind nicht immer Rechtecke

Wir haben gerade gelernt, dass wir zum Auslesen unseres Codes 23 Datenwörter lesen müssen. Aus der Abbildung auf der letzten Seite sehen wir auch, dass die Form eines Datenwortes nicht immer ein Rechteck ergibt. Dann muss besonders darauf geachtet werden, die Bits in der richtigen Reihenfolge zu lesen.

Die Abbildung unten zeigt als Beispiel, welche Formen von Bitblöcken sich an einem Umkehrpunkt von „nach oben lesen" (rechts Bahn) und „nach unten lesen" (links Bahn) ergeben. Das 8Bit-Wort „kriecht" sozusagen um die Kurve. Analog dazu geschieht das an deren Umkehrpunkten oder an Hindernissen. Die Bits werden in der Reihenfolge von 0–7 gelesen.



Hier noch ein Beispiel aus unserem QR-Code, wenn ein Ausrichtungs-Marker (das kleine Quadrat im rechten, unteren Viertel) im Weg steht. Diese Layout/Infrastruktur-Bereiche müssen ja übersprungen werden, daher geht es zuerst fünf Bits lang nur nach oben, bis wieder zwei Reihen für die Bahn zur Verfügung stehen. Da dann wieder die Regel „rechts vor links" gilt, ergibt sich die Bitreihenfolge aus der Abbildung rechts.



Zum Decodieren anderer Codes hilft es, diesen auf einer DIN A4-Seite auszudrucken und die entsprechenden Datenwörter mit einem farbigen Stift zu umranden, wie ich es bei der Abbildung auf der letzten Seite getan habe.

Nun kennen wir die Datenwörter und schreiben die in der Binärdarstellung in einer Liste untereinander. Dann rechnen wir das in eine Dezimalzahl um (das geht mit etwas Übung leicht per Kopfrechnen) und erhalten den Zeichencode. Dazu benötigen wir eine Tabelle, in der zu jedem Zeichencode das entsprechende Zeichen steht. Da wir keinen Zeichencode über 128 haben, ist der Zeichensatz unerheblich und es kann jede ASCII-Tabelle oder Code-Tabelle für PC-Zeichen genutzt werden. Eine ist hier zu finden: `http://www.ascii-tabelle.org/ascii_tabelle.htm`

Das tun wir auf der nächsten Seite …

Konvertierung Zeichencode nach Zeichen

Diese 23 Pakete mit je 8 Bit müssen wir dann in eine Dezimalzahl umwandeln. Mit diesem Zeichencode können wir in einer Zeichencode-Tabelle nachschlagen, welches Zeichen hier gespeichert wurde.

Die Tabelle rechts zeigt das Ergebnis für die 23 Datenwörter, jeweils die Bitfolge (zum Vergleich mit den Bits im QR-Code), als Dezimalzahl und mit dem Zeichen, das dem Zeichencode entspricht. Da wir keinen Zeichencode über 128 haben, ist der Zeichensatz unerheblich und es kann jede ASCII-Tabelle oder Code-Tabelle für PC-Zeichen genutzt werden.

Nr.	Bits	Dezimal	Zeichen
1	`01101000`	104	h
2	`01110100`	116	t
3	`01110100`	116	t
4	`01110000`	112	p
5	`01110011`	115	s
6	`00111010`	58	:
7	`00101111`	47	/
8	`00101111`	47	/
9	`01000001`	65	A
10	`01110010`	114	r
11	`01101101`	109	m
12	`01101001`	105	i
13	`01101110`	110	n
14	`01001000`	72	H
15	`01100001`	97	a
16	`01101110`	110	n
17	`01101001`	105	i
18	`1110011`	115	s
19	`1100011`	99	c
20	`1101000`	104	h
21	`101110`	46	.
22	`1100100`	100	d
23	`1100101`	101	e

Die ersten Zeichen, die wir erhalten, bilden die Zeichenfolge „`https:`“. Daran (bzw. `http:`) erkennt ein QR-Code-Leser, dass es sich bei dem Inhalt des QR-Codes um eine URL handelt und kann den Browser öffnen.
Es gibt also gar keinen QR-Code vom Typ „URL“! Ebenso wenig wie es einen Typ „Visitenkarte“ gibt. Diesen erkennt die Software einfach daran, dass die ersten Zeichen `BEGIN:VCARD` lauten.

Für den QR-Code ist das einfach „Nutzlast“ aus byteweise kodierten Zeichen. Falls jemand auf einer Website diese Unterscheidung bei einer Anwendung zum Erzeugen von QR-Codes findet, dann deshalb, weil damit die Eingabeformulare für die Daten unterschiedlich gestaltet werden.

Ergebnis der Decodierung

Wir erhalten die URL
`https://ArminHanisch.de`
als Ergebnis der Decodierung. Voilá! 😊

Herzlichen Glückwunsch! Sie haben gerade erfolgreich einen QR-Code mit einer URL manuell zerlegt, dekodiert und in Text verwandelt. Ohne Computer oder eine App!

Da sind aber noch Bits übrig? Wozu dienen diese denn? Das finden wir gleich raus…

Und der Rest der Bits?

Von den Bits, die unseren QR-Code ausmachen, haben wir bis jetzt etwa die Hälfte verarbeitet. Welche Bedeutung haben dann die restlichen Bits? Diese dienen der Fehlerkorrektur, denn einer der Vorteile eines QR-Codes ist eine Fehlertoleranz. Wie wir bereits wissen, gibt es vier verschiedene Fehlerkorrektureinstellungen. So können bis zu etwa 30% des Codes unlesbar sein (z.B. zerstört oder durch ein Bildmotiv überdruckt) und der Code kann von einer Applikation trotzdem gelesen werden. Diese Hilfsdaten für die Fehlerkorrektur stecken in den restlichen Daten des QR-Codes. Durch das Einfügen dieser zusätzlichen Daten lassen sich über einen Algorithmus fehlende Daten „zurückrechnen". Dies ist der gleiche Ansatz, der beispielsweise auch bei Audio-CDs verfolgt wird.

Eine detaillierte Beschreibung des Fehlerkorrektur-Verfahrens ist nicht das Ziel dieses Tutorials, da die mathematischen Grundlagen hierfür anspruchsvoller sind und ohne Unterstützung durch Computercode nicht mehr sinnvoll nachvollzogen werden können. Im Anhang finden sich aber eine Reihe von Internet-Adresse für Interessierte. Der verwendete Ansatz nennt sich „Reed-Solomon-Code" und wurde von Irving Reed und Gustave Solomon Anfang der 60er Jahre des letzten Jahrhunderts am MIT Lincoln Lab inLexington, Massachusetts entwickelt.

Anhang

Weiterführende Links zum Erstellen, Lesen und Restaurieren von QR-Codes

`https://www.thonky.com/qr-code-tutorial/introduction`

`http://blog.qartis.com/decoding-small-qr-codes-by-hand/`

`https://blauerbildschirm.wordpress.com/2012/03/05/wie-ein-qr-code-codiert-wird-turtorial-qr-code-encoding-tutorial/` (sic! – da steht „Tu**r**torial")

`http://www.ams.org/publicoutreach/feature-column/fc-2013-02`

`http://datagenetics.com/blog/november12013/index.html`

`https://medium.com/@r00__/decoding-a-broken-qr-code-39fc3473a034`

`http://www.swetake.com/qrcode/qr1_en.html`

`https://www.robertxiao.ca/hacking/ctf-writeup/mma2015-qrcode/`

`http://aioo.be/2015/07/28/Decoding-a-partial-QR-code.html`

`https://en.wikiversity.org/wiki/Reed–Solomon_codes_for_coders`

`https://www.thonky.com/qr-code-tutorial/numeric-mode-encoding`