ESP32CAM PROJEKT MIT QR-CODES

EINFÜHRUNG

Basketball ist ein Sport, in dem Zahlen eine ziemlich große Bedeutung haben. Vielleicht ist es sogar der Teamsport mit den meisten Zahlen, denn wo sonst wird der Punktwert eines Tor- oder Korberfolgs noch so genau gemessen – abhängig von der Position der Spielerin bzw. des Spielers auf dem Feld? Und welche andere Sportart zählt die Fouls so genau für jeden einzelnen Mitwirkenden, mit ausdrücklichem Einfluss auf das Spielgeschehen und damit auch auf das Spielergebnis?

Aber auch im Lande des aktuellen Weltmeisters hat mittlerweile die Digitalisierung Einzug gehalten: Seit ungefähr zwei Jahren werden zumindest Punkte, Fouls und Auszeiten, die jahrzehntelang nur mit einem Kugelschreiber manuell auf einem Anschreibebogen festgehalten wurden, nun digital auf einem Tablet oder notfalls auch Smartphone „angeschrieben". Bei den Wettbewerben des Deutschen Basketball Bunds (DBB) werden dafür die Apps verschiedener Anbieter im Markt eingesetzt, in den nicht-professionellen Ligen hat man sich allerdings im Sommer 2022 für einen spanischen Anbieter entschieden, die Firma NBN23 („NBN" ist die Abkürzung für den Basketball-Slang-Ausdruck „nothing but net", was ein anerkennender Ausruf für einen besonders präzise erzielten Korberfolg ist, bei dem der Ball weder das Brett noch den Ring berührt). NBN23 stellt dabei zweierlei Apps kostenlos bereit: Einerseits die App „NBN23 InGame" (verfügbar für iOS und Android), welche während eines laufenden Spiels vom Anschreiber benutzt wird, um alle Vorkommnisse im Spiel elektronisch zu erfassen, und andererseits die App „Swish" (bzw. unter der Marke „DBB.Scores" vom DBB gebrandet) für iOS oder Android, die jeder Mitwirkende, aber auch jeder Fan auf seinem Smartphone (oder Tablet) installieren kann, um schon während des Spiels laufend die aktuellen „Plays" bzw. Statistiken mitzuverfolgen oder nach dem Spiel die Daten je Spiel, Spieltag, Saison usw. abzurufen[1].

Die für die Trainer wichtigsten Kennzahlen zum Spiel (Spielstand, Restzeit, Foulverteilung, genommene Auszeiten) werden in vielen Hallen zwar vom Zeitnehmer oder auch vom Anschreibe-Assistenten auf einer großen Anzeigetafel projiziert, aber die gibt es erstens nicht überall und zweitens fehlen oftmals die Details zum Coaching des Spielverlaufs. Konnte früher ein Coach einen raschen Blick auf den Anschreibebogen werfen und bei Unstimmigkeiten die Schiedsrichter um Klärung bitten bzw. der „Crew Chief" in jeder Spielpause die handschriftlich auf dem Bogen notierten Spielsituationen kontrollieren, so ist das heute nicht mehr so einfach möglich, denn die Darstellung der App „NBN23 InGame" auf dem Tablet des Anschreibers orientiert sich natürlich an dessen User Experience (siehe Bild).

PROBLEMSTELLUNG

Genau aus diesem Grund präsentiert NBN23 eine elektronische Version dieses Anschreibebogens im Internet – aufrufbar für alle Mitwirkenden mit digitalem Endgerät und Anschluss an das Netz. Die URL zu diesem „digitalen Anschreibebogen" kann der Anschreiber zwar per Menübefehl als QR-Code auf seinem Tablet anderen Beteiligten sichtbar machen (siehe Bild), aber natürlich nur vor dem Spiel oder in den Spielpausen. Läuft das Spiel gerade oder gerät es sogar in seine „Crunch Time" (so werden die letzten zwei Spielminuten genannt, in denen nicht nur besondere Spielregeln gelten, sondern oft auch das Spiel erst entschieden wird), dann ist keine Transparenz mehr möglich.

[1] Von manchen Fans wird als Manko empfunden, dass detaillierte Statistiken hinter einer „Paywall" versteckt sind. NBN23 versucht, seine Software-Entwicklung und die Bereitstellung der Cloud-Infrastruktur durch die Einnahmen aus dieser Paywall zu refinanzieren.

Die Problemstellung war also, größere Transparenz herzustellen, den „digitalen Anschreibebogen“ also zu jedem Zeitpunkt des Spiels und für alle in der Halle einfach und zuverlässig einsehbar zu machen. Eine weitere große Anzeigetafel (einfache Flachbildschirme scheiden aus, da die Schrift schon von den Mannschaftsbänken aus kaum noch zu entziffern wären) würde große Kosten verursachen und hätte obendrein einen zusätzlichen Energiebedarf zur Folge. Es stellt sich also die Ersatzfrage, wie man die oben erwähnte URL bzw. den QR-Code dazu bequem und jederzeit an alle Mitwirkenden bzw. Zuschauer verteilen könnte, denn auf diese Weise ist jeder Interessierte in der Lage, spontan sein eigenes mobiles und stromsparendes Endgerät zu benutzen.

Die erste intuitive Idee zur Verteilung war es, einfach mit einem Smartphone ein Foto des QR-Codes aufzunehmen und dieses auf eine Webseite hochzuladen. Doch hat dieser Lösungsvorschlag drei Schwächen: Erstens besteht das Risiko von Reflexionen und Unschärfe bei der Aufnahme, welche dann zu einer fehlerhaften URL und mithin zu Frustrationen führen könnten. Zweitens muss dann obendrein noch ein temporärer Link oder ein geeigneter Permalink zu der Webseite verteilt werden, auf der sich nun das hochgeladene Foto des Codes befindet. Und drittens benötigt jeder Interessent für den Scan dieses QR-Codes ein zweites Endgerät oder zumindest einen Partner – ein separater Flachbildschirm in der Halle (mit Browser und Anschluss ans Internet) allein zum Zwecke des Scannens kommt schon aus Gründen der Nachhaltigkeit und Kosten auf keinen Fall in Frage.

Die Aufgabe haben wir dadurch gelöst, dass wir mit Hilfe einer preiswerten kleinen Kamera den QR-Code gleich selbst auf die darin enthaltene URL scannen, diese prüfen und anschließend korrekt als Schwarz-weiß-Grafik permanent (und stromsparend) auf einem E-Ink Display vergrößert abbilden, das dann während des gesamten Spiels deutlich sichtbar am Anschreibetisch aufgebaut stehen soll. Im Grunde wird der Code einmal geschickt von einem Gerät auf ein besser geeignetes Zweites kopiert.

SOFTWARE-ARCHITEKTUR

Nun gibt es leider keine preiswerten E-Ink Displays mit bereits fest eingebauter Kamera, so etwas bleibt den deutlich teureren und stromfressenderen Tablets vorbehalten. Unsere Wahl fiel daher auf eine Kombination aus einer ESP32-CAM (siehe make Sonderheft „ESP32-CAM SPECIAL“) und zusätzlich einem 6-Zoll E-Ink Display von Inkplate (mittlerweile „soldered.com“), weil gerade beides zur Hand, aber als Hardware kommen sicher auch andere Modelle in Frage. Stellt sich nun die Frage, wie die eingescannten Daten aus dem einen Gerät mit möglichst wenig Aufwand in das andere Gerät übertragen werden. „Klassisch“ per Verdrahtung und I2C oder SPI? Per ESP-NOW? Oder doch via „sneaker-net“, also per Micro-SD-Karte?

Wir haben uns für MQTT entschieden, allerdings unter Zuhilfenahme eines (kostenlosen) Cloud Brokers, weil wir nicht zusätzlich noch ein drittes Gerät spendieren wollten: Die ESP32-CAM spielt hierbei den „Publisher“, veröffentlicht also die gerade eingescannte und geprüfte URL (bzw. den interessanten Wert daraus) als neue „Message“ zu einem vorher vereinbarten „Topic“ über den Cloud Broker. Und das E-Ink Display empfängt als „Subscriber“ eben diese Message vom Cloud Broker und wandelt diese anschließend über ein passendes Restful API in die schwarzen Pixel um, die nun permanent auf dem Display angezeigt werden sollen.

Vorteil dieser pfiffigen Lösung ist, dass beide Komponenten „out of the box“, also ohne weitere Anbauten (wenn man mal von den – mit einem 3D-Drucker erstellten – Gehäusen absieht), Lötarbeiten usw. zum Einsatz kommen, das Handling sehr einfach ist und auch keine weiteren laufenden Kosten entstehen. Einziger Nachteil einer solchen „Zwei-Geräte-Architektur“ ist, dass irgendein drahtloses Netzwerk für die Kommunikation benötigt wird. Aber weil die App „NBN23 InGame“ ohnehin auf ein WLAN angewiesen ist (welches mittlerweile in den meisten Spielhallen verfügbar sein sollte), um die Daten gleich während des laufenden Spiels in Echtzeit zur NBN23 Cloud-Infrastruktur hochzuladen, lag gleich ziemlich nahe, das örtliche WLAN einfach mitzuverwenden.

Das folgende Schaubild zeigt noch einmal die gesamte Architektur der Lösung:

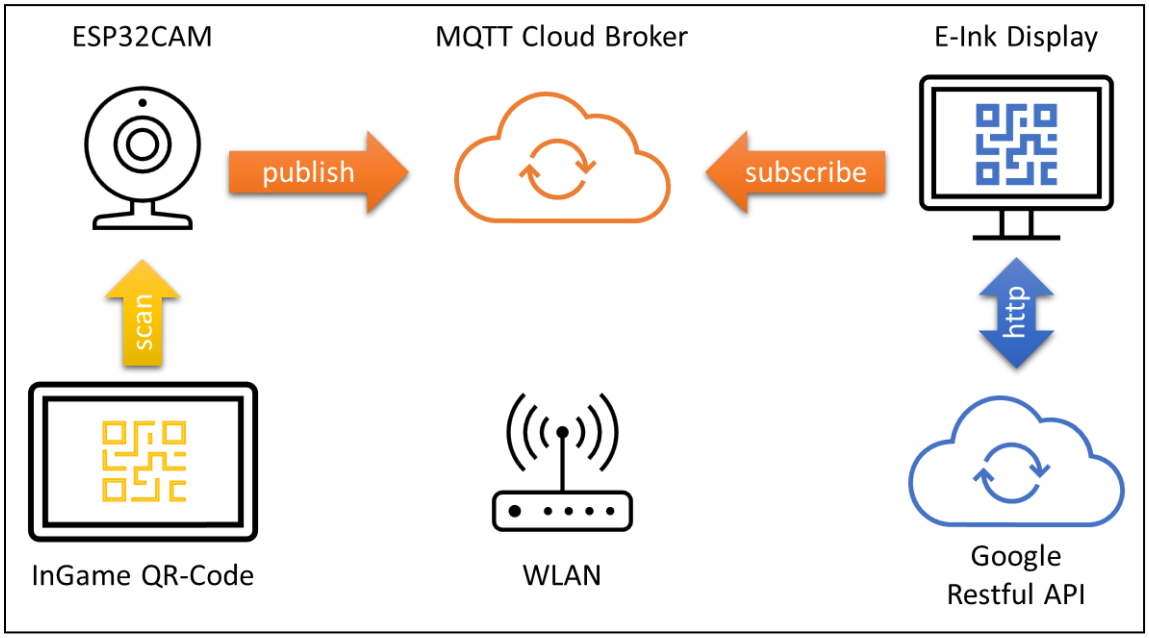

Abbildung 1: Schaubild der Software-Architektur

Das Format jeder „Message“ in unserem einzigen Topic haben wir auf die sogenannte „`matchId`“ (ein Teilstring aus der URL) reduziert. Wir hätten zwar auch die komplette URL übertragen können, aber das Meiste davon bleibt ohnehin stets unverändert. Durch Analyse einiger Beispiele konnten wir nämlich ermitteln, dass die von der App „NBN23 InGame“ angebotenen URIs grundsätzlich folgende einfache Syntax besitzen (es gibt ja keine Authentisierung und damit auch keine Tokens oder Passwörter zu beachten):

```
https://scoresheets.nbn23.com/scoresheet-5v5?matchId=0123456789012345
https://scoresheets.nbn23.com/scoresheet-3v3?matchId=0123456789012345&language=en
```

Abbildung 2: Syntax der „NBN23 InGame“ URIs

Die „`matchId`“ (der grau hinterlegte Teilstring) ist immer eine 16-stellige Ziffernfolge ohne weiter Struktur, welche das Basketballspiel in der Datenbank von NBN23 eindeutig identifiziert, aber nicht „erraten“ werden kann.

Außerdem muss der Subscriber nach dem Empfang der Message erst noch per separater URI ein Restful API von Google aufrufen, um den eigentlichen QR-Code als Grafik frisch zu erzeugen, denn eine von unserer Kamera eingescannte Grafik hätten wir selbst komprimiert kaum in einer MQTT-Message unterbekommen.

CODE-STRUKTUR

Wir haben den Code mit Hilfe der bekannten Arduino IDE (Version 2) erstellt, getestet und auf die Boards hochgeladen. Da es zwei verschiedene Boards gibt, brauchen wir auch zwei getrennte Folder mit jeweils einer ino-Datei gleichen Namens. Der Einfachheit halber haben wir diese Ordner kurz “pub” (für den MQTT-Publisher) und “`sub`” (für den MQTT-Subscriber) genannt, demzufolge gibt es in “pub” auch die Codedatei “`pub.ino`” und in “`sub`” die Codedatei “`sub.ino`”, jeweils verfasst im C/C++-Dialekt der Arduino IDE.

Beide Anwendungen teilen sich eine ganze Reihe von Credentials (zum Beispiel die Zugangsdaten für das WLAN, aber auch die Zugangsdaten für den MQTT Cloud Broker, sowie ein paar wichtige Konstanten), die wir aus diesem Grund in einer gemeinsamen Include-Datei „`credentials.h`“ hinterlegt haben. Logischerweise muss sich je eine Kopie dieser Datei im Ordner “pub” und eine im Ordner “sub” befinden. Dabei spielt es für die Software-Entwicklung, das Kompilieren und den Software-Upload auf die Boards keine Rolle, welche Datei das Original und welche nur die Kopie ist. Wer das möchte, kann auch mit Symlinks oder Hard Links arbeiten, wenn sie/er sich nach jeder Änderung eine manuelle Synchronisation ersparen möchte.

In der ersten Generation unserer beiden Software-Pakete besaßen sowohl der MQTT-Publisher als auch der MQTT-Subscriber beide ihre eigenen Routinen, um das jeweilige Board zuerst mit einem WLAN und anschließend mit dem MQTT Cloud Broker zu verbinden. Wir haben dann aber festgestellt, dass sich die Routinen sehr ähnlich waren – schließlich verwenden sie ja auch das gleiche WLAN und die gleiche Broker-Instanz. Was lag also näher, als auch diese Routinen in einer gemeinsamen Codedatei auszulagern? Die C++-Codedatei heißt „`reconnect.cpp`” und wird ebenfalls in beiden Ordnern “pub” und “sub” abgelegt (bzw. vom einen in den anderen verlinkt). Sie enthält eigentlich nur eine einzige, exportierte Funktion:

```
void reconnect(bool fromsetup, bool ispub)
```

Abbildung 3: Header der exportierten Funktion

Durch die beiden booleschen Parameter kann man aber steuern, wie die Diagnoseinformationen korrekt heißen müssen, die über den „Seriellen Monitor“ ausgegeben werden sollen: Beim Aufruf aus einer `setup`-Funktion wird `fromsetup` auf `True` gesetzt, später in einer `loop`-Funktion auf `False`. Und der zweite Parameter `ispub` besitzt im Publisher (ESP32-CAM) den Wahrheitswert `True`, im Subscriber (E-Ink Display) demgegenüber `False`.

Ein echter „Reconnect“ ist aber in den meisten Fällen gar nicht erforderlich. Denn sobald der Publisher den gescannten QR-Code als Message publiziert bzw. der Subscriber aus der Message einen neuen QR-Code erzeugt und angezeigt hat, können die jeweiligen Geräte eigentlich ausgeschaltet werden, um Strom oder Batterieladung zu sparen. Nur im seltenen Fall, dass nach dem Programmstart die Verbindung zum Cloud Broker verloren ginge, bevor die Arbeit erledigt ist, muss die Verbindung aus der `loop` Funktion heraus erneut hergestellt werden.

MQTT CLOUD BROKER

Wie oben schon erwähnt, fiel die Entscheidung leicht, einen Cloud Broker aus dem Internet anstelle eines lokalen Brokers im WLAN zu verwenden. Wählt man einen solchen Cloud Broker geschickt aus, dann spart das Kosten und zusätzliche Hardware. Viele Nachrichten werden nicht ausgetauscht jeden Monat während der Saison, das sollte im Kontingent der meisten Cloud Broker ohne kostenpflichtige Anmeldung enthalten sein.

Nach der Registrierung bei einem passenden Dienst wird dort erst einmal eine erste Broker-Instanz angelegt und von dieser werden lediglich folgende Informationen benötigt und in der Include-Datei „`credentials.h`“ fest eingebrannt:

Tabelle 1: MQTT Cloud Broker Credentials

`mqtt_broker`	Das ist die statische IPv4-Adresse der MQTT Cloud Broker-Instanz. Geschickter ist es aber, hier ihren FQDN (Fully Qualified Domain Name) im Code zu hinterlegen, denn dann funktioniert unsere Lösung immer noch, selbst wenn sich die IP-Adresse später einmal ändern sollte.
`mqtt_topic`	Da wir nur ein einziges Topic benötigen, über das alle `matchId`s als Messages ausgetauscht werden sollen, spielt die hierarchische Struktur dieses Topics keine Rolle. Jeder Name tut es, wir setzen daher „`nbn23.com/5v5/matchId`“ ein, obwohl die Lösung genauso gut für 3x3 Basketball-Spiele funktionieren sollte.
`mqtt_username`	Man vergibt einen Nutzernamen für die Anmeldung an der Broker-Instanz.
`mqtt_password`	Und man generiert sich ein ausreichend starkes statisches Passwort zur späteren Autorisierung aller Message-Transfers.
`mqtt_port`	Als nächstes muss man noch die TCP-Portnummer für mit TLS gesicherte Verbindungen in Erfahrung bringen, das ist typisch 8883 oder 443 oder ein anderer Privileged oder Registered Port.
`mqtt_cert`	In dieser Variablen wird schließlich das Serverzertifikat zur MQTT Cloud Broker-Instanz hinterlegt, welches beim ersten Verbindungsaufbau überprüft werden soll.

Eine Liste von geeigneten Cloud-Diensten finden Sie im Anhang bzw. in der Online-Seite zu diesem Artikel.

LIBRARIES

So klein, wie die vier benötigten, eher kurzen Codedateien suggerieren mögen, sind die kompilierten Programme dann aber auch wieder nicht. Es werden eine ganze Reihe von umfangreichen Bibliotheken hinzugebunden, welche die eigentliche „Schwerstarbeit“ erledigen.

Der Publisher benötigt für das Einscannen eines QR-Codes die Library „`ESP32QRCodeReader`“ und für den Datenaustausch mit dem Cloud Broker die Arduino-Librarys „`WiFiClientSecure`“ (die ihrerseits wiederum auf „`WiFi`“ aufsetzt) sowie „`PubSubClient`“.

Und der Subscriber setzt für das Anzeigen einer Grafik auf dem E-Ink Display zusätzlich zu „`PubSubClient`“ nur die mitgelieferte Library „`Inkplate`“ des Herstellers Inkplate (mittlerweile „soldered.com“) ein, die aber ziemlich universell ist und sowohl den http-Download einer PNG-Datei aus dem Internet als auch das Rendern einer Grafik aus dem PNG-Format in ein Bitmap-Format elegant miterledigt.

PUBLISHER

In vielen MQTT Publisher Arduino-Anwendungen werden Messages typischerweise direkt aus den klassischen Funktionen `setup()` oder `loop()` heraus versendet. Das ist hier nicht der Fall, stattdessen arbeiten wir über einen „Task Scheduler“: Die Anwendung legt sich quasi schlafen und wartet nur darauf, von einem QR-Code „geweckt“ zu werden, die ihr zufällig vor die Linse kommt. Der Anwender muss keinen Auslöser drücken, sondern nur die Kamera so vor dem QR-Code platzieren, dass dieser im richtigen Schärfebereich und gleichmäßig fokussiert werden kann. Hält man die Kamera zu weit weg oder zu nahe am Code oder verkantet diese (so dass die Schärfeebene der Kamera und die Bildebene des QR-Codes nicht parallel zueinander sind), dann kann trotz der Markierungen im zweidimensionalen Code und der eingebetteten Fehlerkorrektur-Maßnahmen nichts dekodiert werden, was der erwarteten Syntax entspricht.

Doch wie genau funktioniert dieses „Wecken“ und was passiert danach?

Sobald in der Funktionen `setup()` alle Initialisierungen (Serielles Interface, LED zur Statusanzeige, Verbindung zum MQTT Cloud Broker) abgeschlossen sind, wird eine `ESP32QRCodeReader` Instanz auf dem Core #1 der ESP32-CAM angelegt und anschließend nur noch dem Task Scheduler eine Callback-Funktion übergeben mit der Bitte, diese Funktion doch einfach nur aufzurufen, sobald ein QR-Code entdeckt, gescannt und korrekt decodiert wurde.

Die bedeutsame Programmlogik steckt in jener Callback-Funktion: Sie analysiert die Payload des QR-Codes und parst diesen auf eine gültige „`matchId`“, also einen Suchstring innerhalb der Payload, an die eine 16-stellige Ziffernfolge direkt als Wert angehängt ist. Natürlich kann es dabei zu Fehlern kommen und der Scan-Vorgang muss wiederholt werden (beispielsweise muss die Kamera besser ausgerichtet werden).

Für das Parsen reicht in unserem Fall noch eine primitive Mustersuche nach dem Muster „`?matchId=`“: Ab direkt danach bis zum Ende der URI oder bis zum nächsten Parameter wird alles als 16-stellige Zeichenfolge interpretiert und an den MQTT-Broker übergeben.

In der Sporthalle finden wir naturgemäß keinen PC mit Arduino IDE und Seriellem Monitor vor, so dass es nicht möglich sein wird, irgendwelche Diagnoseinformationen live mitzuverfolgen. Aus diesem Grund haben wir uns dazu entschieden, die in die ESP32-CAM fest eingebaute LED dazu zu verwenden, wenigstens eine erfolgreiche Verarbeitung eines QR-Codes (scannen, parsen, senden) durch ein kurzes Aufblitzen zu signalisieren. Der Anwender kann anschließend die Kamera ausschalten und wieder von der Stromversorgung trennen, die `matchId` ist ja wohlbehalten im Broker angekommen.

(Blitzt im Ernstfall nichts auf, dann sollte man den QR-Code mit seinem Smartphone dokumentieren, vielleicht wurde ja vor kurzem erst die Syntax des URI vom Hersteller geändert?)

Die Library „`PubSubClient`“ bietet zwar dem Publisher keine Möglichkeit, beim Versand einen QoS-Wert mitzugeben (in unserem Fall böte sich die 1 an, weil es pro Spiel nur genau eine `matchId` zu übertragen gibt), aber wir haben uns entschieden, wenigstens das Flag für „`Retained`” zu setzen. Dieses sorgt dafür, dass eine Message so lange im MQTT-Broker zwischengespeichert wird, bis sich der erste Subscriber anmeldet und diese Message abgerufen hat. In der Praxis wird es vorkommen, dass die ESP32-CAM bereits zum Einsatz kommt, bevor das E-Ink Display überhaupt ausgepackt und mit der Stromversorgung verbunden wurde.

SUBSCRIBER

Unser Subscriber ist noch einmal ein ganzes Stück einfacher aufgebaut als der Publisher. Die `setup()` Funktion besteht aus ganzen vier Codezeilen Initialisierung: Serieller Monitor, E-Ink Display und zwei Aufrufe für die schon bekannte Library „`PubSubClient`“, nämlich zunächst die Registrierung einer Callback-Funktion für die spätere Verarbeitung von empfangenen Messages und dann die Subskription auf unser vordefiniertes Topic. Das war’s auch schon. Auch hier steckt die eigentliche Programmlogik in der Callback-Funktion selbst, die wir auch gleich `callback()` genannt haben.

Wird nun tatsächlich irgendeine Message aus dem Topic empfangen, dann ruft die Library „`PubSubClient`“ implizit die Callback-Funktion auf und übergibt ihr alle Parameter, die zur Verarbeitung der Nachricht notwendig sind. In unserem Fall handelt es sich schlicht um die 16 Ziffern, aus denen die `matchId` besteht. Stimmt die Länge der Nachricht nicht, dann passiert gar nichts (außer einer Warnmeldung auf der seriellen Konsole). Ansonsten wird die `matchId` per `snprintf()` C-Standardfunktion in ein String-Template hineinkopiert und das Ergebnis einfach an ein Restful API von Google zur Erzeugung und zum anschließenden Download einer Grafik im Format PNG übergeben. Dieser Aufruf wird nicht verschlüsselt, denn das ist nicht notwendig bei einer öffentlichen URL.

Die geeignete Größe des QR-Codes (möglichst raumfüllend, um auch aus größerer Entfernung noch „scanbar“ zu sein) und seine Basis-Koordinaten auf dem E-Ink Display muss man einmal durch manuelles Ausprobieren herausfinden und im Code „einbrennen“.

Interessant zu erwähnen ist vielleicht noch, dass bei der Registrierung einer Callback-Funktion über die Library „`PubSubClient`“ ein QoS-Parameter gesetzt werden kann. Wir entschieden uns hier für den QoS-Wert 1, was so viel bedeutet wie „At least once“, also genauer gesagt:

> You need to get every message and your use case can handle duplicates. QoS level 1 is the most frequently used service level because it guarantees the message arrives at least once but allows for multiple deliveries. Of course, your application must tolerate duplicates and be able to process them accordingly.

Abbildung 4: Was bedeutet QoS = 1?

Duplikate unserer Message mit der gleichen `matchId` sind ohnehin kein Problem: Denn wenn sich der „Bildschirminhalt“ eines E-Ink Displays nicht ändert, dann wird es auch nicht aktualisiert, benötigt also keine weitere Energie. Überhaupt kann man das E-Ink Display direkt nach dem Anzeigen eines neuen QR-Codes (also schon vor Spielbeginn) auch komplett von der Stromversorgung trennen, denn bei solchen Displays verschwindet der Inhalt auch dann nicht und bleibt „kostenlos“ dauerhaft sichtbar und scannbar.

Beginnt aber später am Tag ein neues Basketball-Spiel, das naturgemäß seine eigene `matchId` hat, dann verbindet man das E-Ink Display einfach erneut mit der Stromversorgung und wartet einfach, bis sich ein neuer QR-Code neu aufgebaut hat.

WEITERENTWICKLUNG

Wir haben selbstverständlich noch ein paar Ideen im Hinterkopf, wie wir unser kleines Projekt weiter aufhübschen könnten: Zum Beispiel sollte man nach einem erfolgreichen „scan & publish“ bzw. „subscribe & display“ das zugehörige Gerät automatisch ausschalten oder zumindest in einen Tiefschlafmodus versetzen, um noch mehr Energie zu sparen. Auch die Tatsache, dass momentan ein Satz von WLAN-Zugangsdaten fest im Code eingebrannt ist, ist doch sehr unschön: Wie geht man am besten damit um, wenn ein Basketball-Spiel mal in einer anderen Spielhalle stattfindet, wo das WLAN einen anderen Namen (SSID) und ein anderes WLAN-Passwort hat? Wir denken derzeit in die Richtung „ESPAsync_WiFiManager“ Library, WPS oder OTA-Update der Zugangsdaten, aber selbst SPIFF auf einer Micro-SD-Karte ist noch eine aktuelle Option. Überhaupt ließe sich dann per OTA („over the air“) sogar der komplette Programmcode drahtlos und damit bequemer aktualisieren…

ANHANG

FOTOS

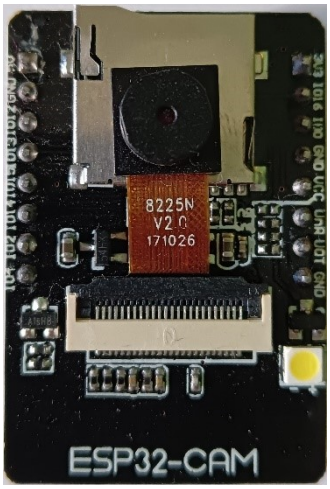

Abbildung 5: ESP32 Kamera (ohne „Motherboard“ und Gehäuse)

Abbildung 6: E-Ink Display mit QR-Code (ohne Gehäuse)

CODEBAUM

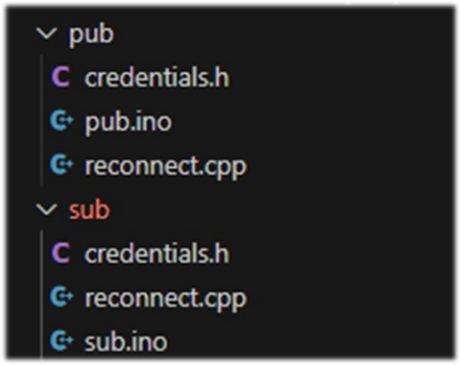

Abbildung 7: Codebaum

LINKS

MQTT CLOUD BROKER

https://www.emqx.com/en/cloud
https://mqtt.one/
https://www.cloudmqtt.com/
https://www.hivemq.com/products/mqtt-cloud-broker/

EMPFOHLENE GEHÄUSE

https://www.thingiverse.com/thing:5638294
https://github.com/SolderedElectronics/Inkplate-6-hardware/tree/master/3D%20printable%20case/Original%20case

LIBRARYS

https://pubsubclient.knolleary.net/
https://www.arduino.cc/reference/en/libraries/pubsubclient/
https://github.com/alvarowolfx/ESP32QRCodeReader

https://github.com/SolderedElectronics/Inkplate-Arduino-library
https://inkplate.readthedocs.io/en/latest/arduino.html
https://github.com/espressif/arduino-esp32/tree/master/libraries/WiFiClientSecure