Smart Schloss System 1.0

Dieser Teil dient dazu, den Source Code von dem ESP32 System zu verstehen und bearbeiten zu können. Das System ist gebaut mit dem Ziel, einfach skalierbar zu sein.

Klassen und ihre Struktur

In der Codebase ist jede Klasse gemäß der Standard-C++-Konvention strukturiert und besteht aus zwei verschiedenen Dateien: einer Header-Datei (.h) und einer Implementierungsdatei (.cpp). Dieser modulare Ansatz verbessert die Codeorganisation und -wartbarkeit, indem Klassen Deklarationen und -definitionen separat voneinander gehalten werden.

Header-Datei (.h)

Die Header-Datei dient als Schnittstelle zur Klasse und enthält Klassendeklarationen, Prototypen von Memberfunktionen und Deklarationen von Datenelementen. Sie definiert die öffentliche Schnittstelle der Klasse, sodass andere Teile des Codes mit ihr interagieren können, ohne die Implementierungsdetails zu kennen.

Die allgemeine Struktur einer Header-Datei umfasst:

- Include Guards: Präprozessor-Direktiven, die mehrfache Einbindungen derselben Header-Datei verhindern.
- Klassendeklaration: Der Klassenname und eventuelle Basisklassen, von denen geerbt wird.
- Öffentlicher Abschnitt: Dieser Abschnitt enthält die öffentlichen Memberfunktionen und Datenelemente der Klasse.
- Privater Abschnitt: Private Memberfunktionen und Datenelemente, die vor externem Zugriff geschützt sind.

Implementierungsdatei (.cpp)

Die Implementierungsdatei enthält die tatsächlichen Definitionen der in der Header-Datei deklarierten Memberfunktionen. Diese Trennung zwischen Schnittstelle und Implementierung trägt dazu bei, die Abhängigkeiten zu reduzieren und die Wartbarkeit des Codes zu verbessern.

Die Implementierungsdatei enthält in der Regel:

- Header-Einbindung: Einbinden der entsprechenden Header-Datei am Anfang.

- Funktionsdefinitionen: Implementierung der in der Header-Datei deklarierten Memberfunktionen.
- Implementierung privater Funktionen: Implementierung etwaiger privater Memberfunktionen.
- Definitionen von Datenelementen: Initialisierung und Verwaltung der Datenelemente der Klasse.

Klassen Dokumentation

Klassen:

1. DB
2. DoorState
3. Lock
4. ManagingWifi
5. PinsExpander
6. PN532
7. RealTimeManager

Klassenbeschreibung:

1. DB

Die Klasse DB ist verantwortlich für die Verwaltung der Drupal-Datenbank. Sie bietet eine Schnittstelle zur Kommunikation mit dem Server und zur Verarbeitung von Daten. Die Klasse ist aufgeteilt in Enums, Namespaces und Structs, die die Lesbarkeit des Sourcecodes verbessern.

Bibliotheken

Die Klasse verwendet folgende Bibliotheken:

1. Arduino Bibliothek: Grundlegende Funktionen und Datentypen
2. C++ Standard Bibliothek: Standard-C++-Funktionen
3. HTTPClient: Für HTTP-Anfragen an den Server
4. ArduinoJson: Für das Verarbeiten von JSON-Daten

Enums

1. QueryName: Enthält verschiedene Abfragetypen, die von der runQuery-Funktion verwendet werden.
2. BoxDoorState: Zustände der Kastentür (offen, geschlossen)
3. KeyState: Zustände eines Schlüssels (reserviert, verfügbar, abgeholt, verloren)
4. BuchungZustand: Zustände einer Buchung (gebucht, abgeholt, zurückgegeben, storniert, abgelaufen, spaet)

Structs

1. BookingData: Struktur für Buchungsdaten
2. UpdateKeyStateQuery: Struktur für die Aktualisierung des Schlüsselzustands
3. FetchBookingDataQuery: Struktur für die Abfrage von Buchungsdaten
4. InsertBoxAccessQuery: Struktur für das Einfügen von Kastenzugriffsdaten
5. UpdateBookingStateQuery: Struktur für die Aktualisierung des Buchungszustands
6. UpdateBoxDoorState: Struktur für die Aktualisierung des Kastentürzustands
7. UpdateKastenZugangState: Struktur für die Aktualisierung des Kasten-Türzugangszustands

Implementierungsdetails

Der folgende Abschnitt enthält detaillierte Informationen zur Implementierung des vorliegenden Codes:

1. Konstruktor und Destruktor:
   - Der Konstruktor `DB::DB(const char* serverURL)` initialisiert die Klasse und setzt die Serveradresse.
   - Der Destruktor `DB::~DB()` löscht das Objekt, wenn es Out-of-scope

2. `runQuery`-Funktionen:
   - `runQuery` verarbeitet verschiedene Abfragen basierend auf den übergebenen Parametern.
   - Alle mögliche Abfragen sind als Enum “QueryName” gelistet.
   - Abhängig von der Abfrage werden HTTP-Anfragen an den Server gesendet, um Daten abzurufen, zu speichern oder zu aktualisieren.

3. `deserializeJsonObj`-Funktion:
   - Diese Funktion deserialisiert die JSON-Antwort vom Server und erstellt einen Vektor von JSON-Objekten.
   - Die Funktion verwendet die ArduinoJson-Bibliothek zur Verarbeitung von JSON-Daten.

4. `processBookingData`-Funktion:
   - Diese Funktion verarbeitet die abgerufenen Buchungsdaten aus dem JSON-Format.
   - Sie extrahiert relevante Informationen und speichert sie in einer Vektorstruktur für spätere Verwendung.

5. `updateKeyState`, `updateBookingState`, `updateBoxDoorState`, `updateKastenZugangState`-Funktionen:
   - Diese Funktionen senden HTTP-Anfragen an den Server, um Daten in verschiedenen Tabellen zu aktualisieren.
   - Die Daten werden als JSON-Objekte in den Anfragen übergeben.
   - `update KeyState`: aktualisiert den Schlüssel Zustand.

- `updateBookingState`: aktualisiert den Buchung Zustand.
- `updateBoxDoorState`: aktualisiert den Türzustand des Kasten.
- `updateKastenZugangState`: aktualisiert den Türzustand abhängig von dem User, der die Tür geöffnet/geschlossen hat, damit es geloggt wird, ob die Tür zu gemacht wurde.

6. String-Manipulationsfunktionen:
   - Die Funktionen `urlEncode`, `correctDrupalTimestamp`, `correctDrupalTimestamp2`, `keyStateToString` und `bookingZustandToString` bearbeiten und formatieren Zeichenketten und Enums Werte.

7. `getCurrentBookings`-Funktion:
   - Diese Funktion gibt die Liste der aktuellen Buchungen zurück, die im aktuellen Kontext gespeichert sind.

8. `clearCurrentBookings`-Funktion:
   - Diese Funktion leert die Liste der aktuellen Buchungen.

Bitte beachte, dass dieser Text nur eine Zusammenfassung der Implementierungsdetails ist. Weitere Details und Erklärungen können im eigentlichen Code gefunden werden.

2. DoorState

Die Klasse DoorState ist verantwortlich für die Zuordnung von Pins zu Kasten-IDs basierend auf den Sensor-Pins. Diese Klasse erleichtert die Identifizierung von Kästen anhand der Sensordaten. Die Pins sind als Input initialisiert.

Bibliotheken

Die Klasse verwendet die folgenden Bibliotheken:

- Arduino Bibliothek: Grundlegende Funktionen und Datentypen.

Implementierungsdetails

1. Konstruktor und Destruktor
   - Die Klasse DoorState hat keinen expliziten Konstruktor oder Destruktor, da keine speziellen Initialisierungs- oder Freigabeaufgaben erforderlich sind.
2. MapBoxSensorPin - Funktion
   - Die Methode MapBoxSensorPin nimmt den Sensor-Pin als Eingabe und überprüft, welcher Kasten-Node-ID auf Drupal diesem Pin zugeordnet ist.
   - Basierend auf der Zuordnung wird die entsprechende Kasten-ID zurückgegeben.
   - Falls der Sensor-Pin ungültig ist oder nicht zugeordnet werden kann, wird der entsprechende Fehlercode zurückgegeben, um Fehlerzustände zu kennzeichnen.

Die Klasse bietet eine einfache Möglichkeit, Sensordaten in Kasten-IDs umzuwandeln, was in Szenarien nützlich ist, in denen Sensordaten zur Identifizierung von Kästen verwendet werden müssen.

Bitte beachten Sie: Aktuell ist nur ein Sensor auf Pin PE_B0 angeschlossen, der nur einem Kasten-Node-ID von drupal zugeordnet ist. Diese Klasse muss bearbeitet werden wenn neue Sensoren angeschlossen werden und einen Kasten-Node-ID in Drupal zuordnen sollen.

3. Lock

Die Klasse Lock ist für die Verwaltung der Schlossfunktionen verantwortlich. Sie ermöglicht das Validieren von Buchungen und das Zuordnen von Schloss-Pins zu Kasten-Node-IDs on Drupal.

Bibliotheken

Die Klasse verwendet die folgenden Bibliotheken:

- Arduino Bibliothek: Grundlegende Funktionen und Datentypen
- RealTimeManager.h: Eine Klasse für die Verwaltung von Echtzeitdaten(mehr dazu später)
- DB.h: Diese Klasse wurde oben detailliert erklärt.

Implementierungsdetails

1. Konstruktor
   - Die Klasse Lock verfügt über einen Konstruktor, der beim Initialisieren eines Lock-Objekts aufgerufen wird.
2. validateBooking - Funktion
   - Die Methode validateBooking vergleicht den Buchungszeitraum mit dem aktuellen Zeitpunkt und überprüft den Buchungszustand. Wenn die Buchung gültig ist, gibt die Methode true zurück, andernfalls false.
3. MapBoxLockPin - Funktion
   - Die Methode MapBoxLockPin ordnet einer gegebenen Kasten-ID den entsprechenden Schloss-Pin zu. Dies ermöglicht die Steuerung des Schlosses für den angegebenen Kasten.

Die Klasse bietet eine Möglichkeit zur Buchungsüberprüfung und zur Zuordnung von Schloss-Pins zu Kasten-IDs.

Bitte beachten Sie: Aktuell sind 8 Ausgänge auf Pins PE_A0 - PE_A7 deklariert, die die Kasten Nummer 1 - 8 in Drupal entsprechen. Falls ein neuer Pins-Expander angeschlossen wird muss diese Funktion erweitert werden.

4. ManagingWifi

Die Klasse ManagingWifi ist für die Verwaltung der WLAN-Verbindung und Netzwerkinformationen verantwortlich. Sie bietet Methoden zum Herstellen und Trennen von WLAN-Verbindungen, zum Abrufen von IP-Adressen und Signalstärken sowie zur Überwachung des Verbindungsstatus.

Bibliotheken

Die Klasse verwendet die folgenden Bibliotheken:

- WiFi: Für die WiFi-Verbindungsfunktionen

Implementierungsdetails

1. Konstruktor
   - ManagingWifi(const char* ssid, const char* password): Ein Konstruktor, der eine WLAN-Verbindung mit den angegebenen SSID und Passwortparametern herstellt.

2. connectToDifferentWifi - Funktion
   - Die Methode connectToDifferentWifi ermöglicht das Herstellen einer WLAN-Verbindung zu einem anderen Netzwerk.

3. Disconnect - Funktion
   - Die Methode disconnect trennt die aktuelle WLAN-Verbindung.

4. getWifiStatus - Funktion
   - Die Methode getWifiStatus gibt den aktuellen Verbindungsstatus des WLANs zurück.

5. getIPAddress - Funktion
   - Die Methode getIPAddress gibt die IP-Adresse des Geräts im WLAN-Netzwerk zurück.

6. getSignalStrength - Funktion
   - Die Methode getSignalStrength gibt die Signalstärke der aktuellen WLAN-Verbindung zurück.

7. setIPAddress - Funktion
   - Die Methode setIPAddress aktualisiert die IP-Adresse des Geräts.

8. setSignalStrength - Funktion
   - Die Methode setSignalStrength aktualisiert die Signalstärke der WLAN-Verbindung.

9. handleConnectionStatus - Funktion
   - Die Methode handleConnectionStatus überwacht den Verbindungsstatus des WLANs und gibt entsprechende Statusinformationen aus.

Die Klasse ManagingWifi bietet Methoden zur Verwaltung von WLAN-Verbindungen, zur Abruf von Netzwerkinformationen und zur Überwachung des Verbindungsstatus. Sie ermöglicht das Herstellen und Trennen von Verbindungen, das Abrufen von IP-Adressen und Signalstärken sowie das Überwachen von Änderungen im Verbindungsstatus.

5. PinsExpander

Die Klasse PinsExpander ermöglicht die Steuerung eines MCP23017 I/O-Expander über I2C. Sie bietet Methoden zur Konfiguration von Pins, zum Setzen von Pins auf HIGH oder LOW und zum Lesen der Zustände von Eingangspins.

Bibliotheken

Die Klasse verwendet die folgenden Bibliotheken:

- Adafruit_MCP23X17.h: Für die Steuerung des MCP23017 I/O-Expanders über I2C.
- Wire.h: Für die I2C-Kommunikation.

Implementierungsdetails

1. Konstruktoren
   - PinsExpander(uint8_t address, int sda_pin, int scl_pin): Ein Konstruktor, der die I2C-Adresse des MCP23017 und die Pins für SDA (Serial Data) und SCL (Serial Clock) festlegt.
2. setup - Funktion
   - Die Methode setup initialisiert den MCP23017 I/O-Expander und die I2C-Kommunikation.
3. setPinModeOutput - Funktion
   - Die Methode setPinModeOutput konfiguriert die angegebenen Pins als Ausgänge.
   - Parameter:
     pins: Ein Array von Pins, die als Ausgänge konfiguriert werden sollen.
     numberOfPins: Die Anzahl der Pins im pins-Array.
4. setPinModeInput - Funktion
   - Die Methode setPinModeInput konfiguriert die angegebenen Pins als Eingänge.
   - Parameter:
     pins: Ein Array von Pins, die als Eingänge konfiguriert werden sollen.
     numberOfPins: Die Anzahl der Pins im pins-Array.
5. TurnHigh - Funktion
   - Die Methode TurnHigh setzt den angegebenen Pin auf HIGH.
   - Parameter:
     pin: Der Pin, der auf HIGH gesetzt werden soll.
     numberOfPins: Nicht verwendet in dieser Methode.
6. TurnLow - Funktion
   - Die Methode TurnLow setzt den angegebenen Pin auf LOW.
   - Parameter
     pin: Der Pin, der auf LOW gesetzt werden soll.
     numberOfPins: Nicht verwendet in dieser Methode.
7. readAllPins - Funktion
   - Die Methode readAllPins liest den Zustand der angegebenen Eingangspins.
   - Parameter
     inputPins: Ein Array von Eingangspins, deren Zustand gelesen werden soll.
     numberOfPins: Die Anzahl der Pins im inputPins-Array.

Die Klasse PinsExpander ermöglicht die Steuerung eines MCP23017 I/O-Expander über I2C. Sie bietet Methoden zur Konfiguration von Pins, zum Setzen von Pins auf HIGH oder LOW und zum Lesen der Zustände von Eingangspins. Die Klasse erleichtert die Interaktion mit dem I/O-Expander und ermöglicht die Erweiterung der digitalen Ein- und Ausgänge.

Bitte beachten Sie: Falls andere Pins Expander auf dem gleichen I2C Bus angeschlossen werden, muss man in main.cpp ein Objekt dafür definieren und die Adresse davon im Konstruktor abgeben. Zwei Objekte dürfen nicht die gleiche Adresse haben. Man muss die Adresse der anderen Pinsexpander ändern.
Overview | Adafruit MCP23017 I2C GPIO Expander | Adafruit Learning System

6.PN532

Die Klasse PN532 ermöglicht die Interaktion mit einem NFC-Modul (PN532) zur Kommunikation mit MIFARE-Tags oder -Karten. Sie bietet Methoden zum Initialisieren des Moduls, zum Lesen von NFC-Tags und zum Abrufen der UID (Unique Identifier) des aktuell erkannten Tags.

Bibliotheken

Die Klasse verwendet die folgenden Bibliotheken:

- Adafruit_PN532.h: Für die Steuerung des NFC-Moduls (PN532) zur Kommunikation mit MIFARE-Tags oder -Karten.
- Wire.h: Für die I2C-Kommunikation.

Implementierungsdetails

1. Konstruktor

Die Klasse PN532 verfügt über einen Konstruktor:

- PN532(uint8_t sdaPin, uint8_t sclPin): Ein Konstruktor, der die SDA- und SCL-Pins für die I2C-Kommunikation und den PN532 initialisiert.

2. Begin
   - Die Methode begin initialisiert das PN532 NFC-Modul und die I2C-Kommunikation.
3. readCard
   - Die Methode readCard liest die UID (Unique Identifier) eines erkannten NFC-Tags und gibt sie als hexadezimalen String zurück.
4. getCurrentUID
   - Die Methode getCurrentUID gibt die zuletzt gelesene UID (Unique Identifier) des NFC-Tags als String zurück.
5. resetCurrentUID
   - Die Methode resetCurrentUID setzt die gespeicherte UID zurück, um Platz für die nächste Leseaktion zu machen.

Die Klasse PN532 ermöglicht die Kommunikation mit einem NFC-Modul (PN532) zur Erfassung von MIFARE-Tags oder -Karten. Sie bietet Methoden zum Initialisieren des Moduls, zum Lesen von NFC-Tags und zur Verwaltung der erfassten UID. Die Klasse ermöglicht die einfache Integration von NFC-Funktionen in Anwendungen.

7. RealTimeManager

Die Klasse RealTimeManager ermöglicht die Verwaltung der Echtzeit durch die Synchronisierung mit einem NTP-Server (Network Time Protocol). Sie stellt Methoden zur Verfügung, um das aktuelle Datum, die aktuelle Uhrzeit und das aktuelle Datum und die Uhrzeit als String abzurufen. Darüber hinaus ermöglicht sie die Konvertierung zwischen einem Datum/Uhrzeit-String und einem time_t-Wert.

Bibliothek:

Die Klasse verwendet die folgenden Bibliotheken:

- WiFi.h: Für die WiFi-Kommunikation.
- NTPClient.h: Für die Kommunikation mit einem NTP-Server zur Zeitsynchronisierung.
- WiFiUdp.h: Für die WiFi-UDP-Kommunikation.

Implementierungsdetails

1. Konstruktor
    - RealTimeManager(const char* ntpServer, int timeZoneOffset): Ein Konstruktor, der den NTP-Server und den Zeitversatz (Zeitzonen-Offset) für die Initialisierung des RealTimeManager festlegt.
2. Destruktor
    - Die Klasse RealTimeManager hat einen Destruktor, der aufgerufen wird, wenn ein RealTimeManager-Objekt außerhalb des Gültigkeitsbereichs geht, um Ressourcen freizugeben und Aufräumarbeiten durchzuführen.
3. begin
    - Die Methode begin initialisiert den RealTimeManager und synchronisiert die Zeit mit dem NTP-Server.
4. update
    - Die Methode update aktualisiert die Zeit vom NTP-Server.
5. getCurrentDate
    - Die Methode getCurrentDate gibt das aktuelle Datum als String im Format "YYYY-MM-DD" zurück.
6. getCurrentTime
    - Die Methode getCurrentTime gibt die aktuelle Uhrzeit als String zurück.
7. getCurrentDateTime
    - Die Methode getCurrentDateTime gibt das aktuelle Datum und die Uhrzeit als String im Format "YYYY-MM-DD HH:MM:SS" zurück.
8. convertStringToTime
    - Die Methode convertStringToTime konvertiert einen Datum/Uhrzeit-String im Format "YYYY-MM-DD HH:MM:SS" in einen time_t-Wert.

Die Klasse RealTimeManager ermöglicht die Verwaltung der Echtzeit durch die Synchronisierung mit einem NTP-Server. Sie stellt Methoden zur Verfügung, um das aktuelle Datum, die aktuelle Uhrzeit und das aktuelle Datum und die Uhrzeit als String abzurufen. Die Klasse erleichtert die Arbeit mit Datum und Uhrzeit und ermöglicht es, Zeitstempel zu erstellen und zu verarbeiten.