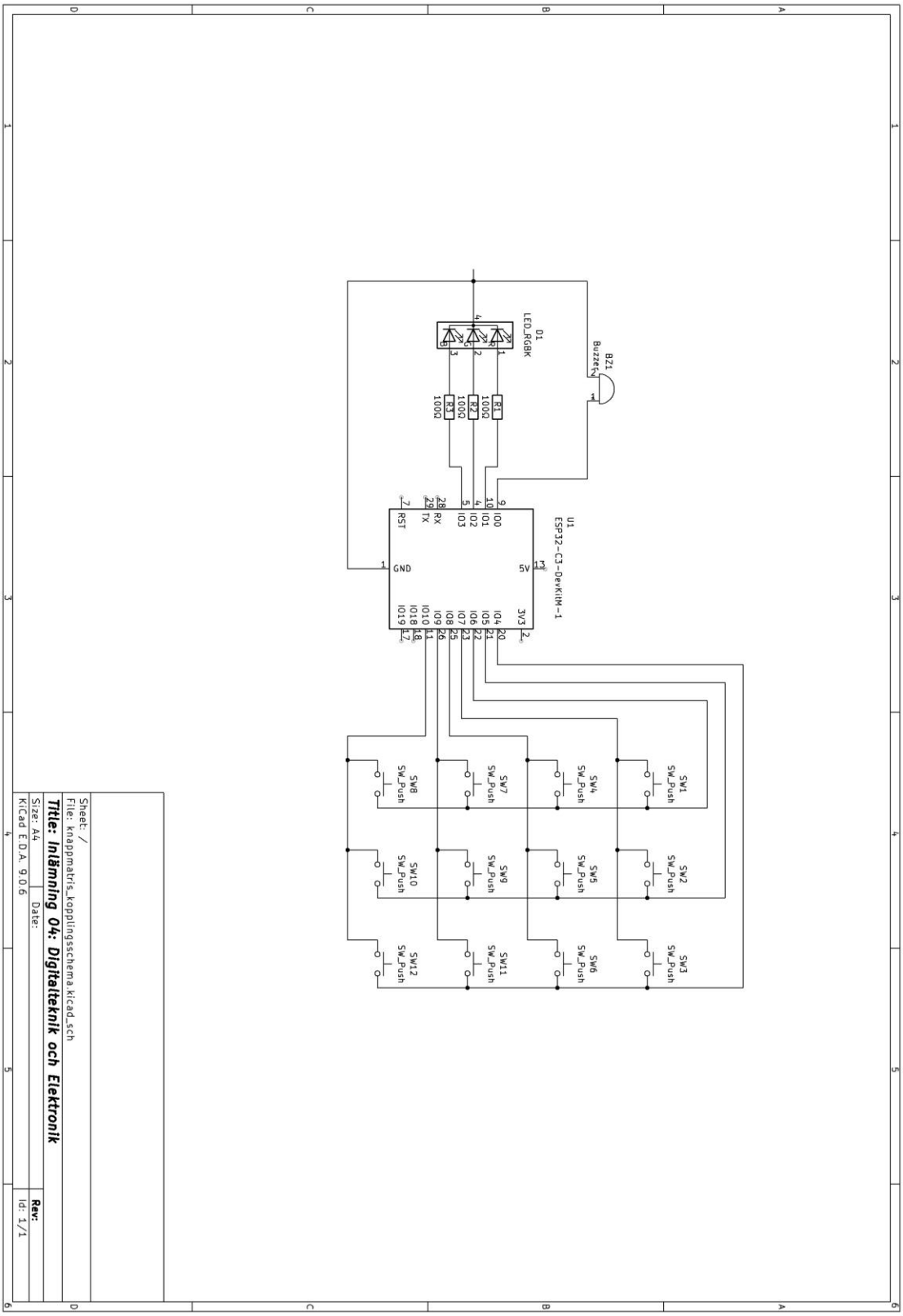
Inlämning 04 – Digitalteknik och Elektronik



Komponenter / Datablad

- ESP32-C6-DevKitM-1
  Kolla vilken logikspänning och maxström GPIO hanterar, och vilka pins som kan nyttjas som GPIO.
  Logik spänning: 3.3V
  Max ström pins: 40mA
  Använda GPIOs: 0, 1, 2, 3, 4, 5, 6, 7, 8, 13, 14.

- Buzzer MI-TH12-0323-RK
  Till buzzern så var det relevanta att se om den är aktiv eller passiv, samt vilken spänning den kräver.
  Spänning: 3V
  Max ström: 30mA

- RGB LED 392-100105
  Ta reda på vilka motstånd som behövs till varje färg.
  Spänning: Grön och blå 3.4V, Röd 2.3V
  Ström: Max 20mA per färg
  Ohms lag används för att beräkna exakt motstånd, men vi vet att 100-330 Ω per färg duger.

- Tactile Button Switch 6mm
  Ta reda på vilka pins som är kopplade.

Kod

Koden ligger strukturerad under en SystemManager-struct som hanterar flödet i koden. Under den är logik och funktionalitet nedbrytet i diskreta funktioner. Vi använder Adafruit KeyPad-bibliotek, men vi har skrivit en egen utbruten funktion nedan för att visa hur det funkar:

- Matris:
  Koden för matrisen är uppbyggd på rader och kolumner.

  setup() sätter kolumnerna till INPUT_PULLUP, vilket ger dom ett viloläge på HIGH, trots att kopplingarna är flytande. Raderna sätts till OUTPUT och HIGH.

  Konceptet är att vi skannar hela matrisen varje varv av loop(). Vid knapptryck så kopplas en rad och en kolumn ihop elektroniskt, och kolumnens PULLUP

övervinns av radens tillfälliga OUTPUT = LOW, vilket gör att läsningen av aktuell kolumn blir LOW. Vi kan då se kombinationen av rad och kolumn för att räkna ut vilken knapp som tryckts: keypadStates[row][column].

```
for (int r = 0; r < ROW; r++) {
  digitalWrite(rowPins[r], LOW);
  for (int c = 0; c < COLUMN; c++) {
    if (digitalRead(columnPins[c]) == LOW) {
      keypadStates[r][c]) = true;
    } else keypadStates[r][c]) = false;
  }
  digitalWrite(rowPins[r], HIGH);
}
```

Matrisflöde:

1. Sätt första rad till LOW.
2. Skanna varje kolumn och se om någon läses LOW.
3. Om LOW, då vet vi kombinationen rad och kolumn för vår nedtryckta knapp.
4. Annars sätt tillbaka aktuell rad till HIGH, och gå vidare till nästa rad och börja om från steg 1.

- LED:
  Placerad i en egen klass “Led”, med flera metoder som [turnOn], [turnOff], [turnOnFor(uint16_t duration)].
  Vi skriver varje pin vi vill ska lysa [digitalWrite(n, HIGH)], och genom att R, G och B är separerade så kan vi skapa färgblandningar. T.ex rött + blått = lila.

```
void Led::turnOn() {
  _state = true;
  digitalWrite(_pinR, _r);
  digitalWrite(_pinG, _g);
  digitalWrite(_pinB, _b);
}
```

- Buzzer:
  Buzzern är aktiv, vilket gör att den endast kan vara på [digitalWrite(n, HIGH)] eller av [digitalWrite(n, LOW)]. Vi får den att låta genom att sätta dess GPIO till HIGH, och stänger av med LOW. Även den är placerad i en klass med flera metoder som speglar Led-klassen.

```
void Buzzer::toggle() {
  if (_state) this->turnOff();
  else this->turnOn();
```

```
  }
```

- Arduino Cloud:
  Tre variabler är skapade som endast Read så att vår keypad skickar info till dashboarden där man kan se status.
  Variabler:
  - isLocked
  - numUnlocks
  - numWrongAttempts

  Dessa exponeras i “thingProperties.h” och kan sedan användas i resten av koden som:

```
int numUnlocks;
int numWrongAttempts;
bool isLocked;
```

  I vår loop() körs sen funktionen ArduinoCloud.update() som uppdaterar variablerna mot molnet.

SOURCE CODE:

“MaxSecurityLock.ino”

```
#include "thingProperties.h"
#include "SystemManager.h"
// #ifdef CLOSED
// #undef CLOSED
// #endif

void setup() {
  Serial.begin(115200);
  delay(500);
  Serial.println("BOOT!");
  SystemManager::setup();
  initProperties();
  ArduinoCloud.addCallback(ArduinoIoTCloudEvent::CONNECT,
  SystemManager::onCloudConnect);
  ArduinoCloud.begin(ArduinoIoTPreferredConnection);
  setDebugMessageLevel(2);
  ArduinoCloud.printDebugInfo();
}

void loop() {
  ArduinoCloud.update();
  SystemManager::run();
}
```

“SystemManager.h”

```
#pragma once
#include <Keypad.h>
#ifdef CLOSED
#undef CLOSED
#endif
#include <Arduino.h>
#include "thingProperties.h"
#include "pinout.h"
#include "Buzzer.h"
#include "Led.h"
#include "KeyLock.h"

enum Mode {
  LOCKED,
  UNLOCKED,
  SET,
  CONFIRM
};

#define ROWS 4
#define COLS 3
char keyMap[ROWS][COLS] = {{'1','2','3'}, {'4','5','6'}, {'7','8','9'},
{'*','0','#'}};
uint8_t rowPins[ROWS] = {PIN_ROW_1, PIN_ROW_2, PIN_ROW_3, PIN_ROW_4};
uint8_t colPins[COLS] = {PIN_COL_1, PIN_COL_2, PIN_COL_3};
Keypad keypad = Keypad(makeKeymap(keyMap), rowPins, colPins, ROWS, COLS);
char clearedDigits[PASSCODE_MAX_LENGTH] = {' ',' ',' ',' ',' ',' ', ' ', ' '};
char passcode[PASSCODE_MAX_LENGTH] = { '1', '2', '3', '4', ' ', ' ', ' ', ' '
};

const int passcodeLength = 8;
int passcodeIdx = 0;
char digits[PASSCODE_MAX_LENGTH];
char confirm[PASSCODE_MAX_LENGTH];
KeyLock keylock(passcode, passcodeLength);
Mode currentMode = Mode::LOCKED;

// Led & Buzzer
unsigned long now;
unsigned long nextLedTrigger = 0;
unsigned long ledLoopDuration = 1000;
Buzzer buzzer(BUZZ_PIN);
Led led(RGB_RED, RGB_GRN, RGB_BLU);
int blinkCount = 3;
static int color = 0;
```

```
unsigned long blinkingLEDDurationMs = 100UL;
unsigned long autoLockDurationMs = 10'000UL;

struct SystemManager {
  static void clearDigits() {
    passcodeIdx = 0;
    memcpy(digits, clearedDigits, PASSCODE_MAX_LENGTH);
  }

  static void clearConfirm() {
    passcodeIdx = 0;
    memcpy(confirm, clearedDigits, PASSCODE_MAX_LENGTH);
  }

  static void handleKeyEntry(char key) {
    switch (currentMode) {
      case Mode::SET:
        led.setColorGreen();
        digits[passcodeIdx++];
        break;
      case Mode::CONFIRM:
        led.setColorGreen();
        confirm[passcodeIdx++];
        break;
      default:
        led.setColorBlue();
        digits[passcodeIdx++];
      }
    buzzer.turnOnFor(100);
    led.turnOnFor(100);
  }

  static void handleSubmission() {
    const bool isMatch = keylock.passcodeMatch(digits);
    if (currentMode == Mode::LOCKED && isMatch) {
      unlock();
    } else if (currentMode == Mode::SET) {
      setPasscode();
    } else if (currentMode == Mode::CONFIRM &&
KeyLock::passcodeCompare(digits,
    confirm)) {
      passcodeConfirmed();
    } else {
      incorrectPasscode();
    }
    passcodeIdx = 0;
  }
```

```
static void handleClearDigits() {
  clearDigits();
  buzzer.turnOnFor(200);
  led.setColorBlue();
  led.turnOnFor(200);
  Serial.println("Cleared");
}

static void incorrectPasscode() {
  lock();
  led.setColorRed();
  led.turnOnFor(100);
  buzzer.turnOnFor(100);
  Serial.println("-- Incorrect Passcode --");
  // update numWrongAttempts in Arduino Cloud
  numWrongAttempts += 1;
}

static void setPasscode() {
  buzzer.turnOnFor(500);
  led.setColorBlue();
  led.turnOnFor(500);
  confirmPasscode();
}

static void changePasscode() {
  clearDigits();
  clearConfirm();
  currentMode = Mode::SET;
    led.setColorBlue();
    led.turnOn();
    Serial.println("## Set Passcode ##");
}

static void confirmPasscode() {
  currentMode = Mode::CONFIRM;
  led.setColorBlue();
  led.turnOn();
  Serial.println("## Confirm Passcode ##");
}

static void passcodeConfirmed() {
  keylock.changePasscode(passcode, digits);
  lock();
  buzzer.turnOnFor(250);
```

```
    led.setColorGreen();
    led.turnOnFor(250);
    Serial.println("## Passcode Set ##");
    clearConfirm();
    clearDigits();
  }

  static void unlock() {
    currentMode = Mode::UNLOCKED;
    autoLockDurationMs = millis();
    led.setColorGreen();
    led.turnOnFor(100);
    buzzer.turnOnFor(100);
    Serial.println("** Correct Passcode **");
    // Update numUnlocks on Arduino Cloud
    isLocked = false;
    numUnlocks += 1;
  }

  static void lock() {
    currentMode = Mode::LOCKED;
    buzzer.turnOnFor(500);
    led.setColorBlue();
    led.turnOnFor(500);
    Serial.println("Locked");
    handleClearDigits();
    // Update isLocked boolean in Arduino Cloud
    isLocked = true;
  }

  static void autoLock() {
    unsigned long limit = 10'000UL;
    if (millis() - autoLockDurationMs >= limit && currentMode ==
Mode::UNLOCKED) {
      lock();
    }
  }

  static void updateKeypadStates() {
    char key = keypad.getKey();
    KeyState state = keypad.getState();
    if (key && state != HOLD) {
      Serial.println(key);
      if (key >= '0' && key <= '9') {
        handleKeyEntry(key);
      } else if (key == '#') {
        handleSubmission();
```

```
        passcodeIdx = 0;
      } else if (key == '*') {
        handleClearDigits();
      }
    }
    if (state == HOLD && currentMode == Mode::UNLOCKED) {
      changePasscode();
      passcodeIdx = 0;
    } else if (key == '*' && state != HOLD && currentMode != Mode::UNLOCKED) {
      lock();
    }
  }

  static void setup() {
    keypad.setHoldTime(1000);
    clearDigits();
    clearConfirm();
    keylock.debug();
    lock();
  }
  static void run() {
    updateKeypadStates();
    buzzer.update();
    led.update();
    autoLock();
    led.blink(100, 3);
  }
  static void onCloudConnect() {
    led.setColorBlue();
    led.turnOnFor(100);
    buzzer.turnOnFor(100);
    Serial.println("Connected to Arduino Cloud");
  }
};
```

“Buzzer.h”

```
#pragma once
#include <Arduino.h>
class Buzzer {
  uint8_t _pin;
  bool _state;
  unsigned long _timeOff;
  bool _timerOn;
public:
  Buzzer(uint8_t pin);
  void turnOnFor(uint16_t duration);
  void turnOn();
  void turnOff();
  void toggle();
  void update();
};
```

“Buzzer.cpp”

```
#include "Buzzer.h"

Buzzer::Buzzer(uint8_t pin) : _pin(pin), _state(false), _timeOff(0),
_timerOn(false) {
  pinMode(_pin, OUTPUT);
}
void Buzzer::turnOn() {
  _state = true;
  digitalWrite(_pin, _state);
}

void Buzzer::turnOff() {
  _state = false;
  digitalWrite(_pin, _state);
}

void Buzzer::turnOnFor(uint16_t duration) {
  this->turnOn();
  _timerOn = true;
  _timeOff = millis() + duration;
}

void Buzzer::toggle() {
  Serial.print(_state);
  Serial.print("
  ");
  if (_state) this->turnOff();
  else this->turnOn();
  Serial.println(_state);
}

void Buzzer::update() {
  if (_timerOn && millis() >= _timeOff) {
    this->turnOff();
    _timerOn = false;
  }
}
```

“Led.h”

```h
#pragma once
#include <Arduino.h>

enum LEDColor {
  RED,
  GREEN,
  BLUE
};

class Led {
  uint8_t _pinR;
  uint8_t _pinG;
  uint8_t _pinB;
  bool _state;
  unsigned long _timeOff;
  bool _timerOn;
  bool _r;
  bool _g;
  bool _b;
public:
  Led(uint8_t pinR, uint8_t pinG, uint8_t pinB);
  void turnOnFor(uint16_t duration);
  void turnOn();
  void turnOff();
  void blink(uint16_t duration, int count);
  void toggle();
  void update();
  void setColorRed();
  void setColorGreen();
  void setColorBlue();
  void setColor(LEDColor color);
};
```

“Led.cpp”

```
#include "Led.h"

Led::Led(uint8_t pinR, uint8_t pinG, uint8_t pinB)
: _pinR(pinR), _pinG(pinG) ,_pinB(pinB), _state(false), _timeOff(0),
_timerOn(false), _r(HIGH), _g(HIGH), _b(HIGH) {
  pinMode(_pinR, OUTPUT);
  pinMode(_pinG, OUTPUT);
  pinMode(_pinB, OUTPUT);
}

void Led::turnOn() {
  _state = true;
  digitalWrite(_pinR, _r);
  digitalWrite(_pinG, _g);
  digitalWrite(_pinB, _b);
}

void Led::turnOff() {
  _state = false;
  _r = LOW;
  _g = LOW;
  _b = LOW;
  digitalWrite(_pinR, _r);
  digitalWrite(_pinG, _g);
  digitalWrite(_pinB, _b);
}

void Led::turnOnFor(uint16_t duration) {
  this->turnOn();
  _timerOn = true;
  _timeOff = millis() + duration;
}

void Led::blink(uint16_t duration, int count) {
  for (int i = 1; i <= count; i++) {
    int duration_multiplier = duration * i;
    this->turnOn();
    _timerOn = true;
    _timeOff = millis() + duration_multiplier;
  }
  turnOff();
}

void Led::toggle() {
  Serial.print(_state);
```

```
  Serial.print("
  ");
  if (_state) this->turnOff();
  else this->turnOn();
  Serial.println(_state);
}

void Led::update() {
  if (_timerOn && millis() >= _timeOff) {
    this->turnOff();
    _timerOn = false;
  }
}

void Led::setColorRed() {
  _r = HIGH;
  _g = LOW;
  _b = LOW;
}

void Led::setColorGreen() {
  _r = LOW;
  _g = HIGH;
  _b = LOW;
}

void Led::setColorBlue() {
  _r = LOW;
  _g = LOW;
  _b = HIGH;
}

void Led::setColor(LEDColor color) {
  switch (color) {
    case LEDColor::RED:
      setColorRed();
      break;
    case LEDColor::GREEN:
      setColorGreen();
      break;
    case LEDColor::BLUE:
      setColorBlue();
      break;
    default:
      break;
  }
}
```

“KeyLock.h”

```
#pragma once
#include <Arduino.h>
#include <cstddef>

constexpr std::size_t PASSCODE_MAX_LENGTH = 8;
class KeyLock {
  private:
  int _passcode_length;
  char _passcode[PASSCODE_MAX_LENGTH];
public:
  KeyLock() = delete;
  KeyLock(const char (&passcode)[PASSCODE_MAX_LENGTH],
  std::size_t passcode_length = PASSCODE_MAX_LENGTH);
  bool passcodeMatch(const char (&input)[PASSCODE_MAX_LENGTH]);
  static bool passcodeCompare(const char* input1, const char* input2);
  bool changePasscode(char (&previous)[PASSCODE_MAX_LENGTH],
  const char (&new_pass)[PASSCODE_MAX_LENGTH]);
  bool try_lock();
  bool try_unlock();
  void debug();
};
/*
EXPLANATION:
The array storing the passcode can hold up to 8 integers
so we only need to check the passcode until the end OR until
we reach a -1
*/
// [-1, -1, -1, -1, -1, -1, -1, -1]
// [1, 2, 3, 4]
```

“KeyLock.cpp”

```
#include "KeyLock.h"
#include <Arduino.h>
#include <cstddef>
#include <cstring>

KeyLock::KeyLock(const char (&passcode)[PASSCODE_MAX_LENGTH],
std::size_t passcode_length)
: _passcode_length(passcode_length) {
  std::memset(this->_passcode, -1, sizeof(this->_passcode));
  // Copying one digit at a time
  for (int i = 0; i < PASSCODE_MAX_LENGTH; i++) {
    _passcode[i] = passcode[i];
  }
}

bool KeyLock::passcodeMatch(const char (&input)[PASSCODE_MAX_LENGTH]) {
  // Copying one digit at a time
  for (int i = 0; i < PASSCODE_MAX_LENGTH; i++) {
    if (input[i] != this->_passcode[i]) {
      return false;
    }
  return true;
}

bool KeyLock::passcodeCompare(const char *p1, const char *p2) {
  // Copying one digit at a time
  for (int i = 0; i < PASSCODE_MAX_LENGTH; i++) {
    if (p1[i] != p2[i]) {
      return false;
    }
  }
  return true;
}

bool KeyLock::changePasscode(char (&previous)[PASSCODE_MAX_LENGTH],
const char (&new_pass)[PASSCODE_MAX_LENGTH]) {
  if (this->passcodeMatch(previous)) {
    // Copying one digit at a time
    for (int i = 0; i < this->_passcode_length; i++) {
      _passcode[i] = new_pass[i];
    }
      memcpy(previous, new_pass, PASSCODE_MAX_LENGTH);
    } else {
    return false;
    }
```

```
  return true;
}

bool KeyLock::try_lock() { return true; }
bool KeyLock::try_unlock() { return true; }
void KeyLock::debug() {
  Serial.println(" ");
  Serial.print("Passcode Length: ");
  Serial.println(this->_passcode_length);
  Serial.print("Passcode is ");
  for (int i = 0; i < this->_passcode_length; i++) {
    Serial.print(this->_passcode[i]);
  }
  Serial.println(" ");
}
```

“pinout.h”

```
#pragma once
#include <Arduino.h>

const uint8_t RGB_RED = 14;
const uint8_t RGB_GRN = 7;
const uint8_t RGB_BLU = 6;
const uint8_t PIN_COL_1 = 4;
const uint8_t PIN_COL_2 = 3;
const uint8_t PIN_COL_3 = 2;
const uint8_t PIN_ROW_1 = 8;
const uint8_t PIN_ROW_2 = 1;
const uint8_t PIN_ROW_3 = 0;
const uint8_t PIN_ROW_4 = 5;
const uint8_t BUZZ_PIN = 13;
```

“Secrets.h”

```
#define SECRET_SSID <username>
#define SECRET_OPTIONAL_PASS <passcode>
#define SECRET_DEVICE_KEY <unique-code>
```

“thingProperties.h”

```
// Code generated by Arduino IoT Cloud, DO NOT EDIT.
#pragma once
#include <Arduino.h>
#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>
#include "secrets.h"

const char DEVICE_LOGIN_NAME[] = <OMITTED>;
const char SSID[] = SECRET_SSID; // Network SSID (name)
const char PASS[] = SECRET_OPTIONAL_PASS; // Network password (use for WPA, or
use as key for WEP)
const char DEVICE_KEY[] = SECRET_DEVICE_KEY;  // Secret device password

int numUnlocks;
int numWrongAttempts;
bool isLocked;

void initProperties(){
  ArduinoCloud.setBoardId(DEVICE_LOGIN_NAME);
  ArduinoCloud.setSecretDeviceKey(DEVICE_KEY);
  ArduinoCloud.addProperty(numUnlocks, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(numWrongAttempts, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(isLocked, READ, ON_CHANGE, NULL);
}

WiFiConnectionHandler ArduinoIoTPreferredConnection(SSID, PASS);
```