Design och tjänsteutveckling för
Internet of Things, vt23

# Laboration 1

John Berge

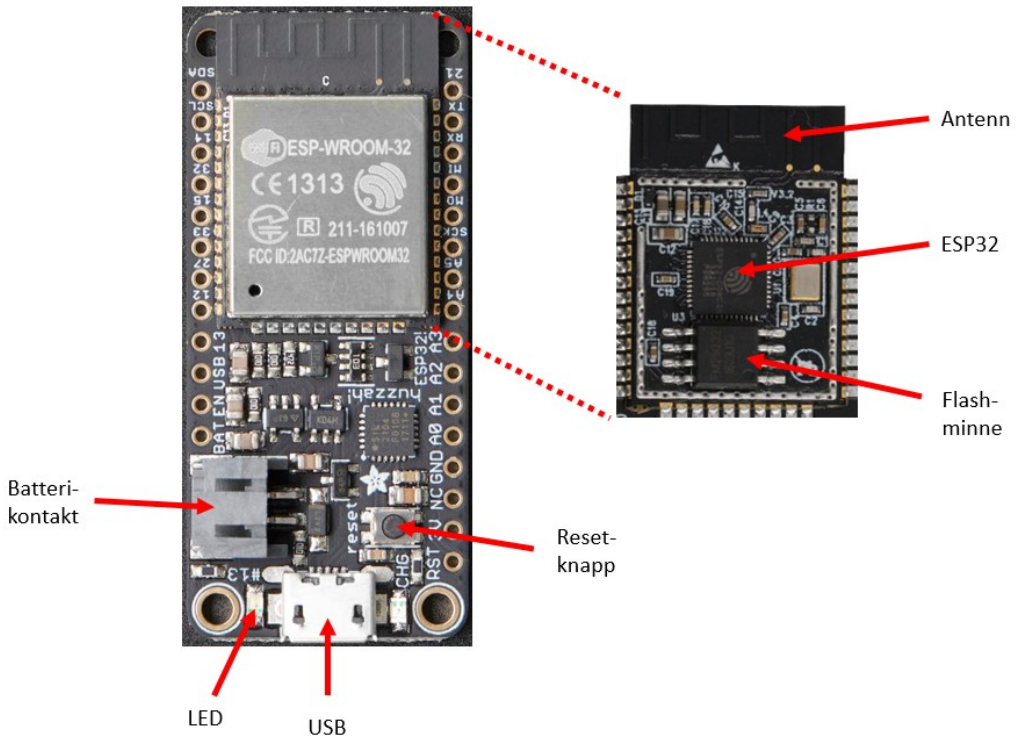
Syftet med dessa inledande laborationsuppgifter är att ge en känsla för grundläggande begrepp inom inbyggda system. En microcontroller-modul (ESP32) används som plattform för att ge praktisk erfarenhet av hårdvarunära programmering, digital I/O, AD-omvandling, seriell kommunikation, etc. Externa komponenter som lysdioder, tryckknappar och en temperatursensor ansluts med hjälp av kopplingsdäck. Efter avslutad laboration har ett temperatur-övervakningssystem med audiovisuellt larm och justerbar larmgräns konstruerats.

Uppgifterna II, III, IV och eventuell spetsuppgift redovisas/demonstreras för lärare i labbsalen. Förbered er på att diskutera lösningarna och se till att alla frågeställningar är behandlade, innan ni ber om att få redovisa.

Spetsuppgiften är en frivillig fördjupning för den som vill lära sig mer och/eller eftersträvar högre betyg. För betyg VG på laborationen ska samtliga grund- och spetsuppgifter vara redovisade och klara innan utsatt deadline.

# *I. ESP32, Mongoose OS*

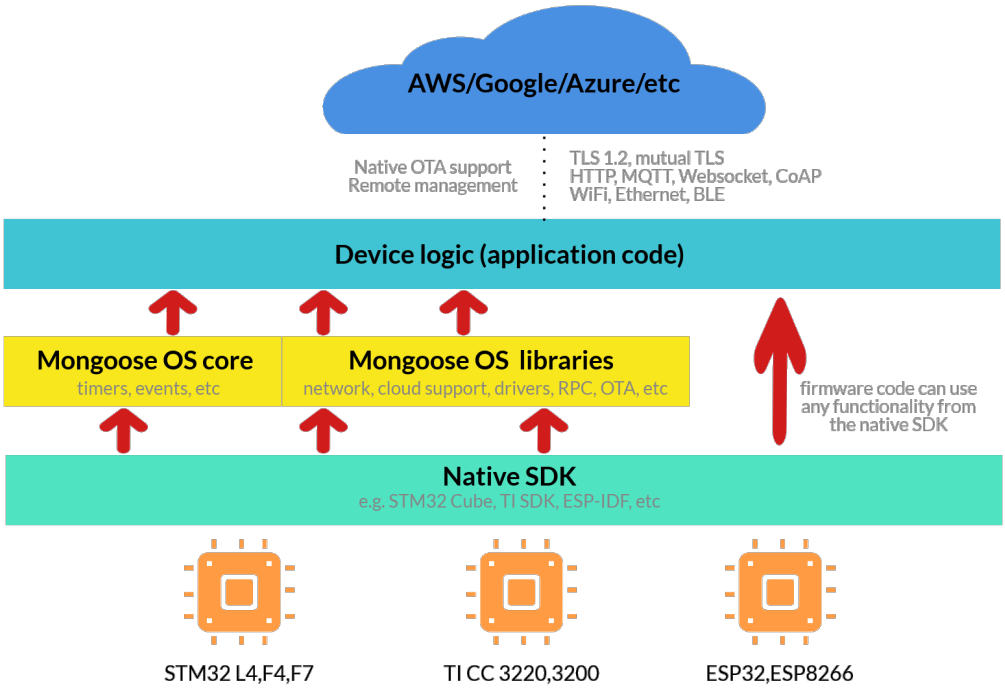
ESP32 är en relativt avancerad microcontroller med inbyggt WiFi- och Bluetooth-stöd. I denna kurs kommer vi att arbeta med en modul som utöver själva ESP32-chipet innehåller bl.a. flash-minne för programkod och en antenn för WiFi/Bluetooth. Denna modul är i sin tur monterad på ett kretskort som är bestyckat med praktiska anslutningspinnar, kontakt för batteri, USB-kontakt, etc.




ESP32-modulen kan programmeras på ett flertal olika sätt. Tillverkaren *Espressif* tillhandahåller ett utvecklingspaket, *Espressif IoT Development Framework (ESP-IDF)*, som består av ett grundläggande mjukvarubibliotek och de verktyg som krävs för att kompilera C-kod för ESP32-processorn.

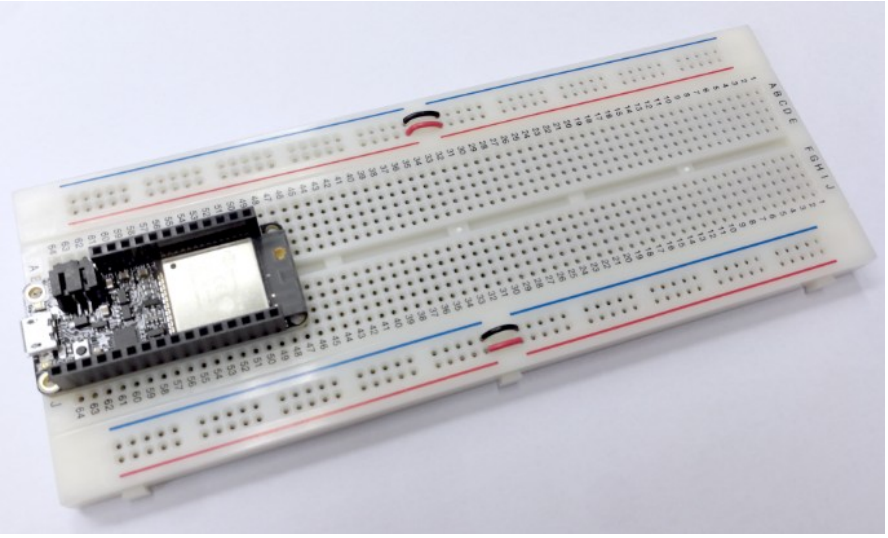
ESP-IDF kan användas direkt för att utveckla applikationer för ESP32. Utvecklaren har då mesta möjliga kontroll över hur hårdvarans resurser används och kan skapa applikationer som är optimerade enligt önskade parametrar. Nackdelen med detta angreppssätt är att utvecklingstiden kan bli omfattande och att vissa vanliga funktioner, som nätverkshantering, filsystem och säkerhetslösningar måste implementeras på nytt om och om igen.

I denna kurs kommer vi att arbeta med ett av flera tillgängliga mjukvarulager som ligger ovanpå *ESP-IDF*: *Mongoose OS*. Mongoose OS möjliggör snabb utveckling av avancerade applikationer med antingen C eller JavaScript (eller en kombination).

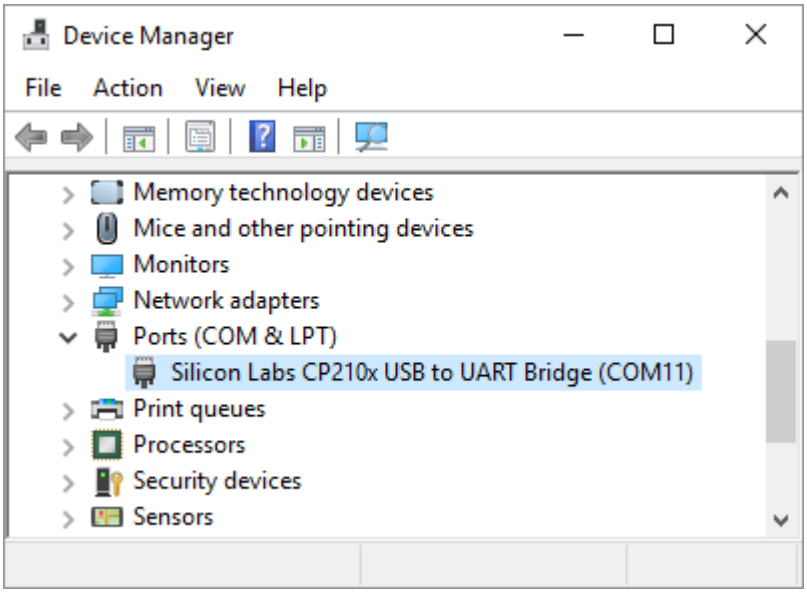


*Uppgift a)*

Montera försiktigt ESP32-kortet på ett kopplingsdäck enligt bild nedan. (Kopplingsdäcket förhindrar att anslutningspinnarna på undersidan skadas eller kortsluts, och kommer att användas framöver för att ansluta ytterligare komponenter.)



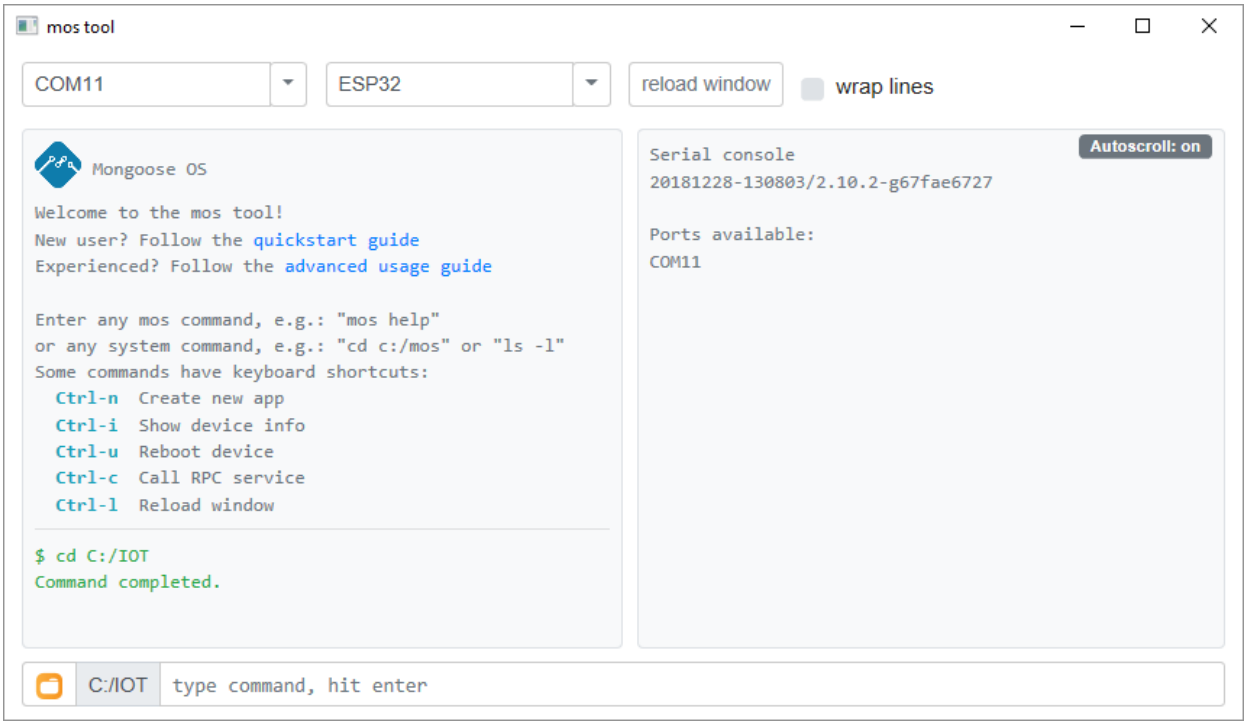
Anslut en USB-kabel mellan ESP32-kortet och den PC du har tänkt arbeta med. ESP32-kortet kommer nu att spänningsmatas via PCns USB-port, och dyka upp som en COM-port i Windows enhetshanterare (Device Manager).[1]




1 Om du arbetar med en egen dator behöver troligtvis en drivrutin installeras. Om den inte installeras automatiskt kan den hämtas här: https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

Mongoose OS kommer med ett verktyg, *mos tool*, som används för att kommunicera med ESP32-processorn, kompilera kod, och föra över firmware till ESP32-modulens flash-minne. Ladda ner *mos.exe* från Mongoose OS hemsida[2] och placera filen i den katalog där du tänkt samla materialet för denna laboration.

Starta *mos.exe* och välj rätt COM-port (enligt Windows enhetshanterare) och ESP32 i drop-down-menyerna i programfönstrets överkant.




Skapa ett nytt minimalt projekt-skelett genom att ange följande kommando i prompten i programmets nederkant:

```
mos clone https://github.com/mongoose-os-apps/empty iot_labb
```

Klicka på mapp-ikonen till vänster om kommando-prompten för att öppna filhanteraren och se de nyskapade filerna. Filen *mos.yml* används för att konfigurera upp systemet, lägga till mjukvarubibliotek som behövs, etc. Applikationskoden kan skrivas antingen i C eller i JavaScript.

2 https://mongoose-os.com/docs/

Skapa en fil *fs/init.js* med valfri text-editor[3] och lägg in följande JavaScript-kod:

```
load('api_timer.js');
let count = 0;
function min_timer_callback(){
    print('hej:', count++);
}
Timer.set(1000, Timer.REPEAT, min_timer_callback, null);
```

Komplettera *mos.yml* med följande rad, som aktiverar stödet för utveckling i JavaScript.

```
- origin: https://github.com/mongoose-os-libs/mjs
```

Filen *src/main.c* kan tas bort, eftersom vi i denna laboration ska arbeta med JavaScript.

Kör följande kommando i kommando-prompten. Kommandot laddar upp konfigurations- och källkodsfilerna till Mongoose OS moln, där filerna kompileras och bakas ihop till en firmware-fil *build/fw.zip*.

```
mos build
```

Kör därefter följande kommando, som överför innehållet i *fw.zip* från PCn, via USB-kabeln, till ESP32-modulens flash-minne.

```
mos flash
```

Om allt fungerar som det ska bör uppstartsmeddelanden från Mongoose OS efter en stund scrolla förbi i *mos tool*s logg-fönster (till höger), följt av periodiska utskrifter från det lilla scriptet ovan. Gratulerar, du har just byggt och installerat din första applikation för Mongoose OS på ESP32! :)

*Uppgift b)*

Studera exempelscriptet ovan och försök förstå hur det fungerar. Bekanta dig med dokumentationssidorna på Mongoose OS hemsida. Här finns t.ex. information om hur Timer-funktionen används, *Mongoose OS -> API Reference -> Core -> Timers (JS API)*
Bekanta dig även med dokumentationen för den minimala JavaScript-implementationen *mJS*, länk finns under *Mongoose OS -> API Reference -> Misc*

Modifiera det givna scriptet så att utskriften till logg-fönstret visar antalet minuter och sekunder som förflutit. Exempel på utskrift: `2 minuter och 34 sekunder`
(Tips: om endast *init.js* uppdateras behöver inte hela den ibland aningen tidskrävande cykeln *mos build*, *mos flash* köras. Det räcker att föra över det nya scriptet med *mos put fs/init.js* och därefter trycka på den lilla svarta reset-knappen på ESP32-kortet.)

3 En plugin för Mongoose OS finns till Visual Studio Code.
https://mongoose-os.com/docs/mongoose-os/quickstart/ide.md

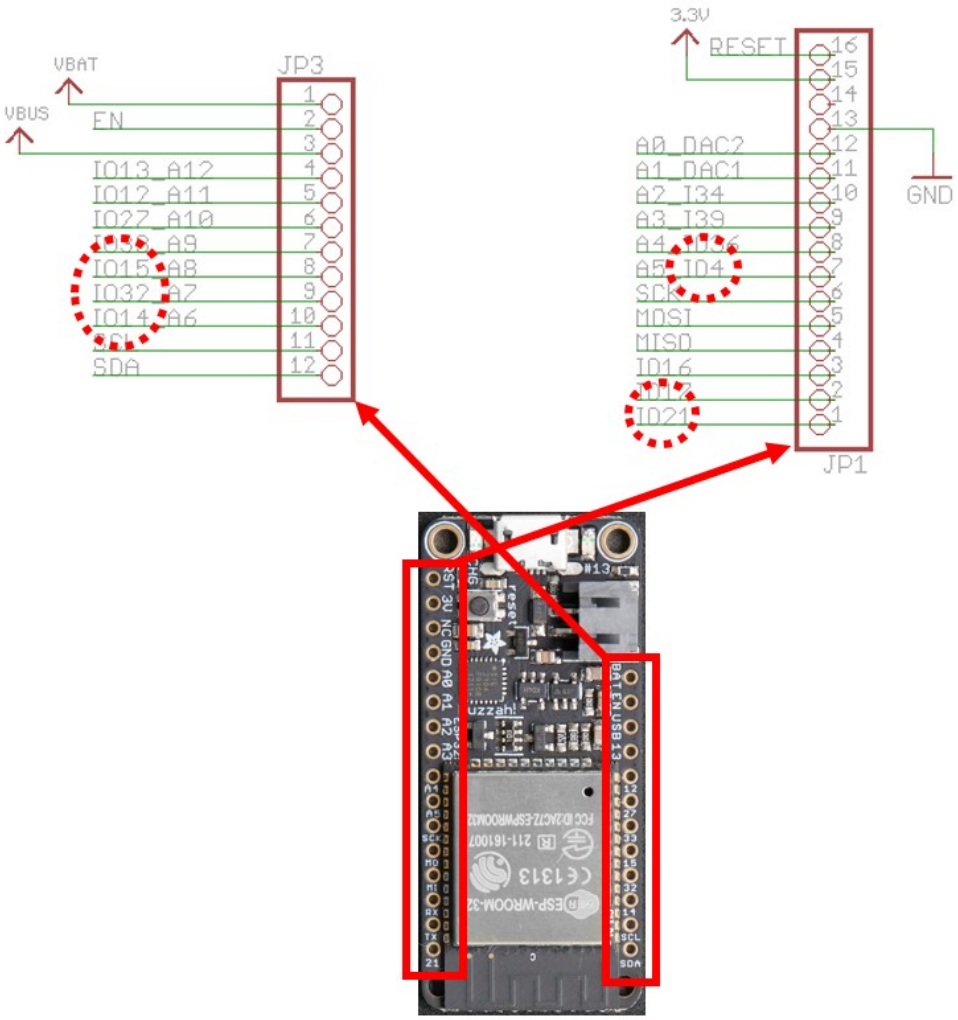
# *II. Digital I/O*

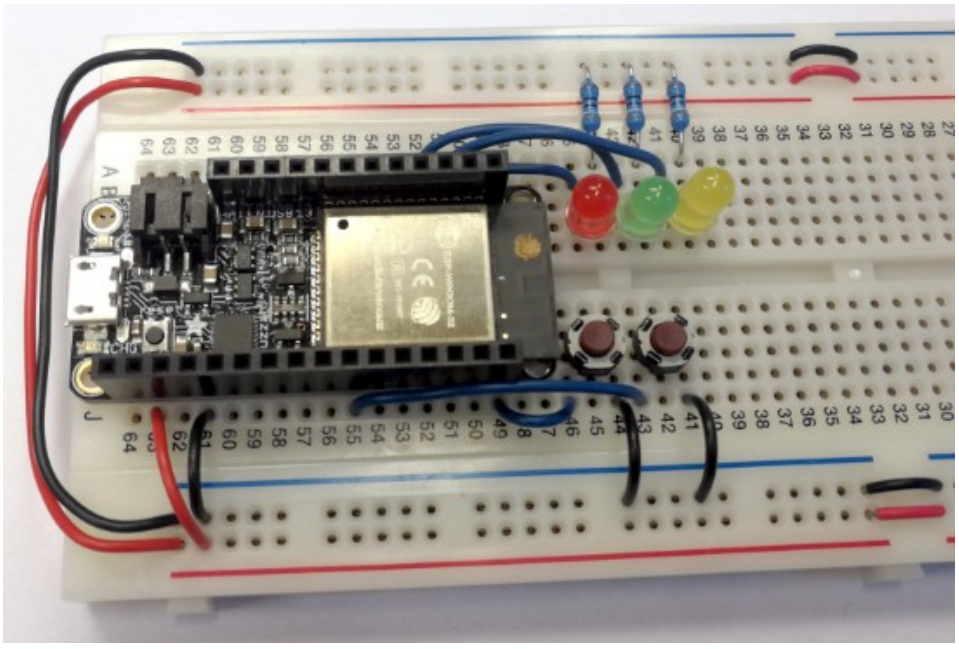
Det enklaste sättet för en microcontroller att kommunicera med omvärlden är sk. digital I/O (digital Input/Output), även benämnt GPIO (General Purpose I/O). Med *digital* menas att bara två spänningsnivåer används, antingen 0V som representerar en logiskt låg signal (logisk 'nolla') eller 3.3V som representerar en logiskt hög signal (logisk 'etta')[4] . En digital *utgång* kan användas för att driva spänningsnivån på en anslutningspunkt antingen hög (3.3V, '1') eller låg (0V, '0'). En digital *ingång* kan användas för att läsa in vilken spänningsnivå som ligger på en anslutningspunkt. 3.3V kommer att läsas in som logiskt hög ('1') och 0V kommer att läsas in som logiskt låg ('0').

”*hello, world*” inom hårdvarunära programmering brukar oftast innebära att tända och släcka en lysdiod med hjälp av en digital utgång, följt av att läsa av en tryckknapps tillstånd med hjälp av en digital ingång. Kopplingsdäcket ska nu att användas för att ansluta dessa komponenter.

- Dra ur USB-kabeln ur ESP32-modulen (eller datorn). Tag för vana att alltid arbeta med icke-spänningssatt system när förändringar i uppkopplingen ska göras.
- Anslut kopplingsdäckets blå jord- och röda matningsspännings-banor till ESP-modulens *GND*- och *3V*-anslutningar. Externa komponenter som inte drar alltför mycket ström kan därefter i och med detta enkelt spänningsmatas via dessa banor. Tag för vana att använda röda kablar för anslutningar till matningsspänning (3.3V) och svarta kablar för anslutningar till jord (0V).
- Koppla tre lysdioder (gärna olika färger) mellan jord och ESP32-processorns anslutningar *IO15*, *IO32*, och *IO14*. För att veta var dessa anslutningar finns tillgängliga på det svarta kretskortet måste kortets kopplingsschema konsulteras, se utsnitt på nästa sida. Koppla 390Ω motstånd i serie med lysdioderna för att begränsa strömmen. Kom ihåg att lysdioderna måste vara vända åt rätt håll, det korta benet ska anslutas mot jord.
- Koppla två tryckknappar mellan jord och ESP32-processorns anslutningar *IO21* och *IO4*.
- ***Dubbelkolla att alla anslutningar är korrekta. Felkoppling kan medföra att komponenter går sönder när matningsspänningen slås på. Be en handledare eller kamrat om hjälp om du är osäker.***
- Anslut USB-kabeln.

4 Digitala system kan arbeta med olika *logiknivåer*, beroende på vilka komponenter som ingår. 1.8V, 3.3V, 5V, etc. I denna kurs kommer vi att arbeta med 3.3V.


VBAT
VBUS
EN
IO13_A12
IO12_A11
IO27_A10
IO33_A9
IO15_A8
IO32_A7
IO14_A6
SCL
SDA
JP3
3.3V
RESET
A0_DAC2
A1_DAC1
A2_I34
A3_I39
A4_IO36
A5_IO4
SCK
MOSI
MISO
IO16
IO17
IO21
GND
JP1



Skapa ett nytt minimalt projekt och fyll på *mos.yml/init.js* enligt instruktionerna för uppgift Ia. Verifiera att du kan bygga och flasha systemet.

```
cd ..
mos clone https://github.com/mongoose-os-apps/empty iot_labb_gpio
  -- redigera mos.yml/init.js --
mos build
mos flash
```

Komplettera *init.js* med kod som konfigurerar LED-anslutningspunkterna som *digitala utgångar*. Se *Mongoose OS -> API Reference -> Core -> GPIO*

```
load('api_timer.js');
load('api_gpio.js');

let PIN_LEDR = 15;      // red LED
let PIN_LEDG = 32;      // green LED
let PIN_LEDY = 14;      // yellow LED
GPIO.setup_output(PIN_LEDR, 0);
GPIO.setup_output(PIN_LEDG, 1);
GPIO.setup_output(PIN_LEDY, 0);

let count = 0;
function min_timer_callback(){
    print('hej:', count++);
    GPIO.toggle(PIN_LEDR);
    GPIO.toggle(PIN_LEDG);
    GPIO.toggle(PIN_LEDY);
}
Timer.set(1000, Timer.REPEAT, min_timer_callback, null);
```

Uppdatera firmware med den nya koden. Fungerar det som det ska?

Anslutningspunkter kan konfigureras som *digitala ingångar* på ett par olika sätt, beroende på om vi periodiskt vill gå in aktivt och kontrollera logiknivån, eller om vi vill att systemet automatiskt ska meddela oss när en viss övergång sker. Ofta behöver också inbyggda sk. pull-motstånd aktiveras för att säkerställa väldefinierade logiska nivåer, t.ex. när en ansluten tryckknapp inte är intryckt. Bekanta dig med följande API-funktioner.

```
GPIO.setup_input
GPIO.read
GPIO.set_button_handler
```

Komplettera *init.js* med kod för att läsa av de två tryckknapparna och programmera systemet enligt följande specifikation. I och med att vi i vårt system anslutit knapparna till jord, måste pull-up-motstånd aktiveras.

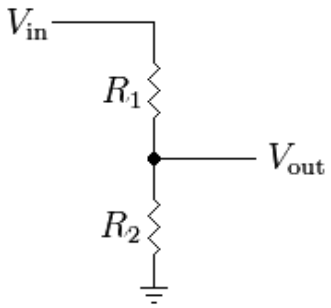
- Lysdiod 1 blinkar hela tiden.
- Lysdiod 2 blinkar när knapp 1 är nedtryckt, annars inte.
- Lysdiod 3 kontrolleras på/av med hjälp av knapp 2. Ett tryck sätter på, nästa tryck slår av, osv.
- En logg-utskrift visar hur många gånger knapp 2 har tryckts ned.

# *III. ADC*

Ibland behöver ett i huvudsak digitalt system även förhålla sig till analoga signaler, dvs. signaler där spänningsnivån kan variera fritt utan att vara begränsad till de digitala logiska nivåerna (0V/3.3V i vårt fall). Många microcontrollers är därför utrustade med inbyggda analog-till-digital-omvandlare (ADC) och digital-till-analog-omvandlare (DAC). En ADC konverterar en analog spänningsnivå på en av microcontrollerns anslutningspunkter till ett digitalt tal som representerar spänningsnivån. På motsvarande sätt kan en DAC generera en analog spänningsnivå utifrån ett digitalt tal.

Sensorer och andra kringkomponenter kan ha analoga gränssnitt, och i denna uppgift ska en enkel potentiometer användas för att demonstrera den ADC som finns tillgänglig på ESP32.

Koppla upp en tre-pinnars vridpotentiometer på kopplingsdäcket. En potentiometer är ett reglerbart motstånd som fungerar som en resistiv spänningsdelare.[5] Anslut de två yttre benen till jord respektive 3.3V ($V_{in}$). Mät spänningen på potentiometerns mittersta ben ($V_{out}$) med en multimeter eller ett oscilloskop samtidigt som du vrider på ratten. Hur varierar den uppmätta spänningen när du vrider ratten från ändläge till ändläge?


Om du kopplat rätt ska spänningen på potentiometerns mittersta ben variera från nära 0V vid rattens ena ändläge till omkring 3.3V vid andra ändläget. Potentiometern kan med hjälp av denna analoga signal användas som en input-enhet till ett microcontroller-baserat system för att på ett enkelt sätt låta användaren interagera med systemet. (En joystick på t.ex. en PlayStation-handkontroll fungerar enligt exakt samma princip, två vridpotentiometrar, en för x-axeln och en för y-axeln.)

5 https://sv.wikipedia.org/wiki/Spänningsdelare

Anslut potentiometerns mittersta ben till ESP32-processorns anslutning *IO36.*

Komplettera *mos.yml* med följande rad, som aktiverar stödet för ADC.

```
- origin: https://github.com/mongoose-os-libs/adc
```

Bekanta dig med JavaScript-funktionerna för hantering av ADC. Se *Mongoose OS -> API Reference -> Core -> ADC*

Siffervärdet som ADC.read() returnerar kommer att ligga i spannet 0-4095, där 0 motsvarar 0V och 4095 motsvarar 3.3V. Den ADC som finns i ESP32 är en 12-bitars omvandlare, vilket innebär att den kan dela upp ett analogt spänningsintervall i $2^{12}$-1 diskreta intervall. Upplösningen blir alltså i teorin 3.3V/($2^{12}$-1) ≈ 0.8mV, men många ytterligare faktorer påverkar den verkliga noggrannheten.

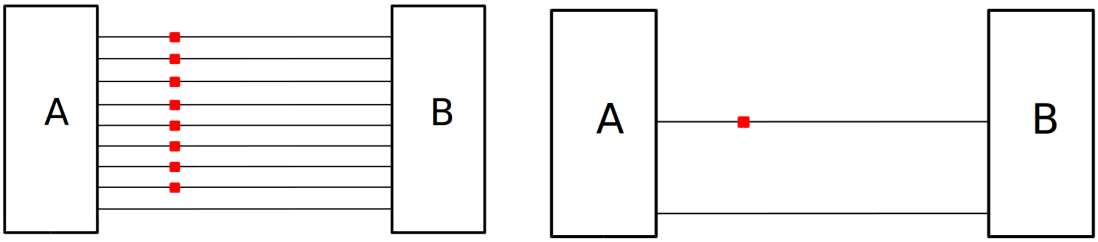
Programmera systemet så att det uppfyller följande specifikation.

- Inläst ADC-värde samt motsvarande spänning i mV skrivs ut en gång per sekund i logg-fönstret.
- Lysdiod 1, 2, 3 tänds/släcks när inläst analog nivå passerar 1V, 2V, 3V.
  0-1V => ingen LED tänd
  1-2V => LED1 tänd
  2-3V => LED1 och LED2 tända
  3-3.3V => LED1, LED2 och LED3 tända

# *IV. Seriekommunikation, temperatursensor*

Digital I/O är ett mycket användbart och rättframt sätt att interagera med enklare komponenter som tryckknappar och lysdioder. Överföring av diverse status-signaler med hjälp av digital I/O är också vanligt förekommande. Vid denna typ av kommunikation behöver bara en bit information överföras åt gången. '1' => lysdiod på, '0' => lysdiod av, exempelvis. Digital I/O kan även användas i mer komplexa sammanhang där flera bitar överförs *parallellt*. Åtta I/O-ledningar kan t.ex. användas för att föra över ett åtta bitars binärt tal från en enhet till en annan.

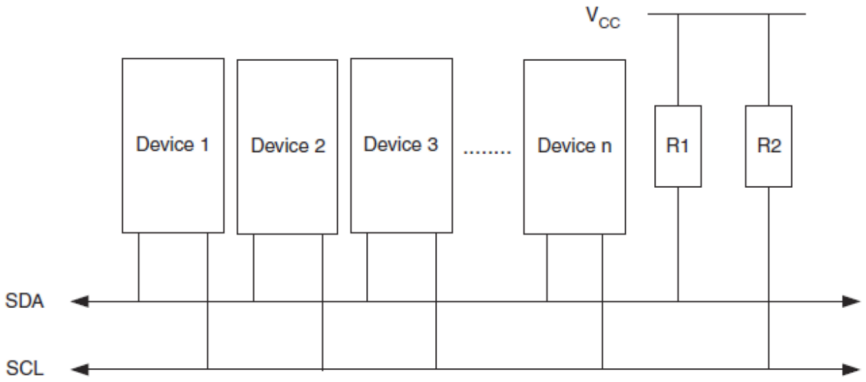
När det kommer till överföring av lite större datamängder (flera bytes) är dock *seriella* kommunikationssätt helt dominerande. Med seriell överföring menas att flera databitar skickas över en efter en från ett delsystem till ett annat. En seriell överföring av ett åtta bitars tal kan då ske över en eller ett fåtal dataledningar, istället för över åtta dataledningar som i det parallella fallet.



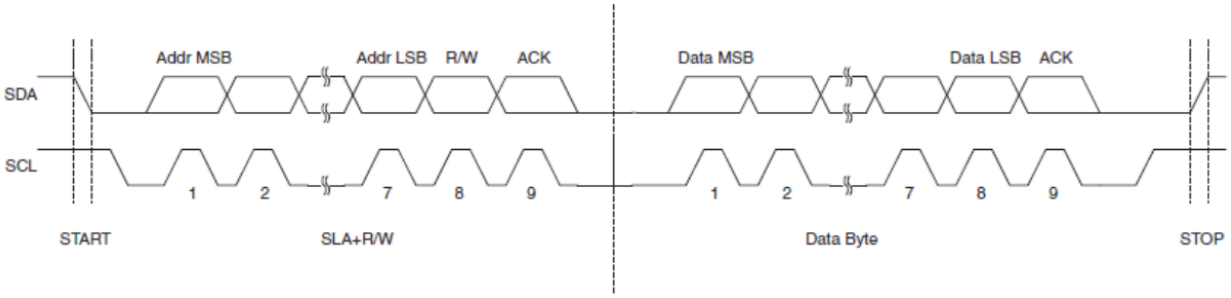

Det finns en uppsjö olika varianter av protokoll/gränssnitt för seriell kommunikation. Inom inbyggda system är de tre grundläggande *UART*, *SPI* och *$I^2C$*, och de flesta microcontrollers har inbyggt hårdvarustöd för dessa. I denna laboration ska vi använda oss av en temperatursensor som kommunicerar via $I^2C$.

I²C står för *Inter-Integrated Circuit* och är en adressbaserad seriell sk. *buss*. Hårdvarumässigt består I²C-bussen av två ledningar, dataledningen SDA och klockledningen SCL. Två pull-up-motstånd ska finnas på bussen, från SDA/SCL till systemets matningsspänning. På en I²C-buss kan upp till 127 enheter kopplas på samma SDA/SCL-ledningar. Varje enhet ska ha en unik sju bitars adress, som används vid varje dataöverföring för att särskilja de anslutna enheterna.

Vid en överföring agerar en enhet *master*, och de övriga *slavar*. Typiskt är det en microcontroller i systemet som agerar master, och t.ex. en eller flera sensorer som agerar slavar.[6] Mastern initierar överföringen och instruerar den adresserade slaven på vilket sätt överföringen ska ske; om data ska skickas från mastern till slaven, eller från slaven till mastern. Mastern genererar klocksignalen på SCL-ledningen och kontrollerar på så vis även hur snabbt överföringen ska ske.




På grund av de två pull-up-motstånden ligger de två I²C-ledningarna på logiskt hög nivå när ingen överföring sker på bussen. Illustrationen nedan visar hur spänningsnivåerna varierar vid en typisk överföring. Mastern genererar klocksignalen som sätter takten. På dataledningen skickar mastern först sju adress-bitar som anger vilken slav-enhet den vill kommunicera med. Därefter skickas en bit (R/W) som anger om mastern är intresserad av att läsa en byte från slaven, eller skriva en byte till slaven. Om det finns en slav-enhet på bussen med den angivna adressen skickar den enheten en ACK-bit. Därefter kan själva dataöverföringen starta.




6 I princip kan flera master-enheter vara anslutna till samma buss, och en enhet kan växla mellan att agera master och slav.

MCP9808 är en temperatursensor från Microchip med I²C-gränssnitt. Via I²C-bussen kan en master-enhet läsa av uppmätt temperaturdata samt göra diverse inställningar, t.ex. konfigurera strömsparläge och alert-funktioner. En I²C-slav som MCP9808 innehåller typiskt ett antal interna dataregister för att lagra inställningar och mätdata. I och med detta måste vi på något sätt kunna ange vilket av slavens interna register vi är intresserade av att interagera med. Hur detta ska gå till återfinns i komponentens datablad. För att hämta hem temperatur-data från MCP9808 ska ett 16-bitars register läsas av. Klipp från databladet:

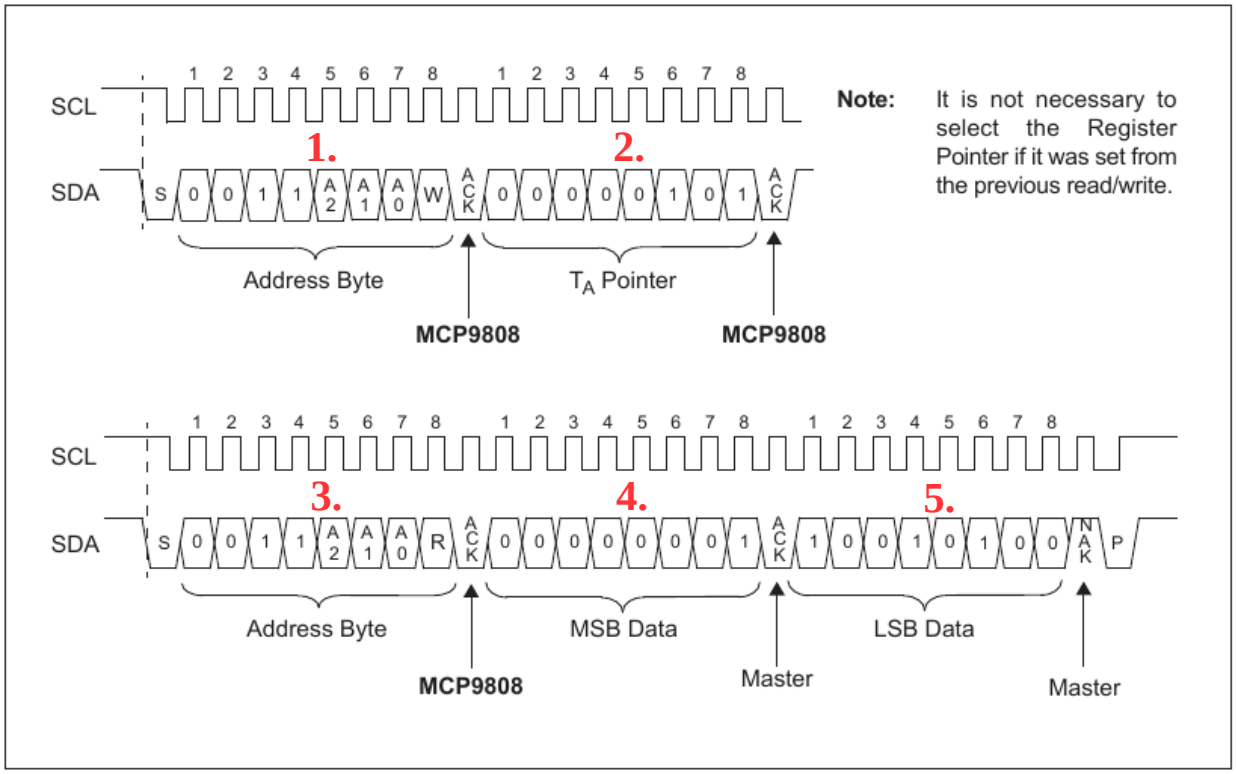



***FIGURE 5-5:*** *Timing Diagram for Reading +25.25°C Temperature from the $T_A$ Register (see **Section 4.0 "Serial Communication"**).*

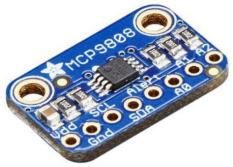
Här framgår att denna sekvens ska följas:

1. Mastern skickar slav-adress samt Write-bit, vilket indikerar att mastern vill skriva till slaven. Sensorn svarar med ACK.
2. Mastern skickar adressen till registret som vi är intresserade av att interagera med (Register Pointer). Sensorn svarar med ACK.
3. Mastern skickar slav-adressen på nytt, denna gång med Read-bit, vilket indikerar att mastern vill läsa från slaven. Sensorn svarar med ACK.
4. Sensorn skickar de åtta *mest signifikanta* bitarna av temperatur-registret. Mastern svarar med ACK, vilket indikerar att mastern vill läsa mer data från slaven.
5. Sensorn skickar de åtta *minst signifikanta* bitarna av temperatur-registret. Mastern svarar med NACK, vilket indikerar att mastern är nöjd och överföringen är klar.

Hur sekvensen ovan utförs rent praktiskt beror på vilken microcontroller som används och hur utvecklingsmiljön för applikationskoden ser ut. För ESP32 finns som vi strax ska se praktiska funktioner i Mongoose OS för att hantera olika typer av I²C-kommunikation.

MCP9808-sensorn sitter monterad på ett litet blått kretskort. På kretskortet finns pull-up-motstånd för I²C-bussens ledningar förmonterade. Koppla upp sensorn på kopplingsdäcket, fyra anslutningar behövs:

Vdd -> 3.3V
Gnd -> jord
SCL -> ESP32: SCL/IO22
SDA -> ESP32: SDA/IO23




Skapa ett nytt minimalt projekt med *mos tool*. Om du vill kan du därefter kopiera över *mos.yml* och *init.js* från föregående uppgift och bygga vidare på den koden. Komplettera *mos.yml* med följande rad, som lägger in mjukvarustödet för I²C.

```
- origin: https://github.com/mongoose-os-libs/i2c
```

I *mos.yml* måste vi även ange att I²C ska aktiveras (som standard avstängt), samt ange vilka av ESP32-processorns IO-anslutningar som används för SDA/SCL. (Denna information hittar vi i kopplingsschemat för ESP32-kortet.)

```
config_schema:
  - ["i2c.enable", true]
  - ["i2c.scl_gpio", 22]
  - ["i2c.sda_gpio", 23]
```

Det behövs inte mycket kod i *init.js* för att utföra en avläsning av det 16 bitar stora temperaturregistret enligt sekvensen på föregående sida, tack vare de funktioner som finns tillgängliga via Mongoose OS. Se *Mongoose OS -> API Reference -> Core -> I2C*
Funktionen I2C.readRegW() utför precis det som vi behöver. Den läser två bytes (ett *word*) från en slav-enhet, med start på en given slav-register-adress. Exempel på användande:

```
load('api_i2c.js');

let MCP9808_I2CADDR = 0x18;          // 0x00011000 std slave address
let MCP9808_REG_AMBIENT_TEMP = 0x05; // 0b00000101 temp data reg

let i2c_h = I2C.get();               // I2C handle
let t = I2C.readRegW(i2c_h, MCP9808_I2CADDR, MCP9808_REG_AMBIENT_TEMP);
```

Efter anropet till I2C.readRegW() har data förts över via I²C-bussem från sensorns temperatur-register till ESP32 och variabeln *t*. Utskrift av variabeln *t* kommer dock inte att visa temperaturen i grader Celsius som vi kanske förväntar oss. Åter till databladet för sensorn, som visar vilken data som temperatur-registret egentligen innehåller:

**REGISTER 5-4: $T_A$: AMBIENT TEMPERATURE REGISTER (→ ADDRESS '0000 0101'b)(1)**

| R-0 | R-0 | R-0 | R-0 | R-0 | R-0 | R-0 | R-0 |
|---|---|---|---|---|---|---|---|
| $T_A$ vs. $T_{CRIT}$(1) | $T_A$ vs. $T_{UPPER}$(1) | $T_A$ vs. $T_{LOWER}$(1) | SIGN | $2^7$ °C | $2^6$ °C | $2^5$ °C | $2^4$ °C |
| bit 15 | | | | | | | bit 8 |

| R-0 | R-0 | R-0 | R-0 | R-0 | R-0 | R-0 | R-0 |
|---|---|---|---|---|---|---|---|
| $2^3$ °C | $2^2$ °C | $2^1$ °C | $2^0$ °C | $2^{-1}$ °C | $2^{-2}$ °C(2) | $2^{-3}$ °C(2) | $2^{-4}$ °C(2) |
| bit 7 | | | | | | | bit 0 |

Här ses att de tre mest signifikanta bitarna inte innehåller temperaturdata utan används i samband med sensorns alert-funktioner. Vi ser också att de fyra minst signifikanta bitarna används för att ange delar av grader. Se nedan för ett kod-exempel som rensar bort icke-relevanta bitar, och konverterar till ett decimaltal. (Här bryr vi oss inte om SIGN-biten som indikerar minusgrader.)

```
let tempC = t & 0x0fff;       // bitwise AND to strip non-temp bits
tempC = tempC/16.0;           // convert to decimal
print("Temperature:", tempC);
```

*Uppgift temperaturlarm*

Konstruera ett system för temperaturövervakning enligt följande specifikation.

- Temperaturen ska kontinuerligt mätas (hela plusgrader).
- Aktuell temperatur ska jämföras med en inställbar övre temperaturgräns.
- En vridpotentiometer ska användas för att justera temperaturgränsen.
- Aktuell temperatur och temperaturgräns ska en gång per sekund skrivas ut till logg-fönstret.
- När aktuell temperatur överstiger temperaturgränsen ska systemet larma.
- Vid larm ska logg-utskriften kompletteras med ”VARMT!”.
- Vid larm ska en lysdiod blinka.
- Vid larm ska en piezoelektrisk ”buzzer” ljuda.[7]
- Om temperaturen återgår till tillåtet värde ska larmet återställas.

7 https://en.wikipedia.org/wiki/Piezoelectric_speaker
*Mongoose OS -> API Reference -> Core -> PWM*

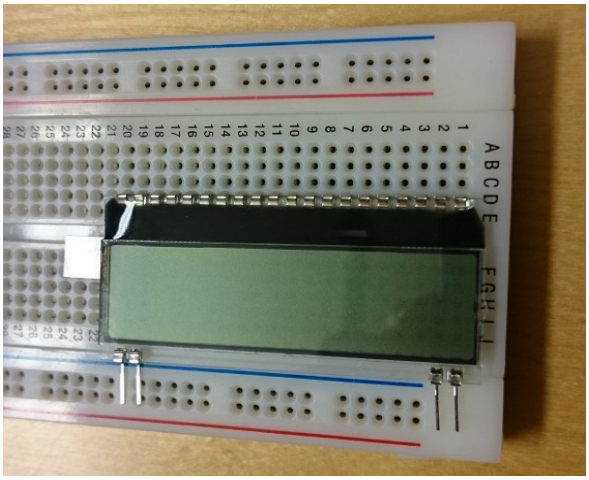
# *Spetsuppgift – LCD-panel*

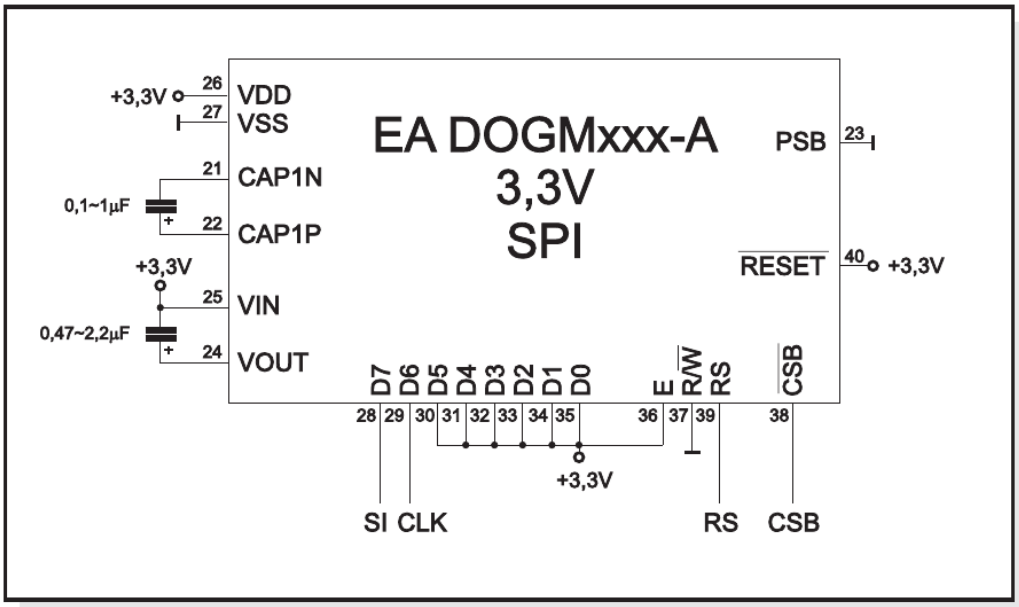
Utöka temperaturlarmet från föregående uppgift med en enkel LCD-panel för att visa text. Informationen som tidigare visades med hjälp av logg-utskrifter ska nu i stället visas på LCD-panelen (aktuell temperatur, temperaturgräns samt eventuell VARMT!-indikation).


LCD-panelen, med modellbeteckning *DOGM163*, är utrustad med en styr/driv-krets *ST7036* som tar hand om drivningen av de individuella pixlarna. (Chippet sitter inbakat i panelens svarta kant och syns som en ~5x1 mm remsa.) LCD-panelen används genom att låta microcontrollern skicka åtta bitars instruktioner/kommandon till styr/driv-kretsen, som sedan i sin tur sköter uppdateringen av display-innehållet. Kommunikationen mellan MCU och *ST7036* kan ske på tre olika sätt. Via ett 8-bitars parallellt interface, ett 4-bitars interface eller ett seriellt interface (SPI). *I denna uppgift ska SPI-läget användas.*

I princip måste två olika datablad konsulteras för att få fullständig information om hur panelen fungerar. Databladet från tillverkaren av LCD-panelen (*Electronic Assembly*) samt databladet för den inbakade styr/driv-kretsen *ST7036*. Som tur är innehåller det enklare databladet från *Electronic Assembly* den information som vi behöver för denna uppgift. Skumma igenom databladet så att du får en känsla för vilken typ av information det innehåller.

Koppla upp LCD-panelen på kopplingsdäcket, lämpligen längst till höger så att panelens pinne 21 sammanfaller med kopplingsdäckets rad 1. Kom ihåg att lämna plats så att kablar kan anslutas. De fyra pinnarna längst ner på panelen behöver inte användas, och kan böjas försiktigt så att de ligger längs med kopplingsdäckets yta. Ta det försiktigt med panelen, den består främst av glas. Flytta den helst inte efter det att du satt ner den i kopplingsdäcket. Pinnarna längs panelens övre kant ska anslutas enligt databladet s. 4. Följ kopplingsschemat för 3.3V SPI. Många anslutningar, dubbelkolla att du kopplat rätt!






SI -> ESP32: MOSI
CLK -> ESP32: SCK
CSB -> ESP32: IO33
RS -> ESP32: valfri ledig GPIO

*mos.yml:*

```
config_schema:
  - ["spi.enable", true]
  - ["spi.cs0_gpio", 33]
  - ["spi.miso_gpio", 19]   # från kopplingsschema
  - ["spi.mosi_gpio", 18]
  - ["spi.sclk_gpio", 5]

libs:
  - origin: https://github.com/mongoose-os-libs/spi
```

Innan text kan skrivas ut måste LCD-panelen initieras. Initieringen utförs genom att via SPI skicka en sekvens av åtta bitars instruktioner till panelen. Denna initieringssekvens beskrivs i databladen, och hur den ser ut beror på vilket interface som ska användas, exakt vilken LCD-panel vi har, vilken spänningsnivå vi arbetar med, etc. Ett exempel på en lämplig sekvens av kommandon för att initiera vår panel med tre rader text (*DOGM163*) finns i databladet, se utdrag nedan. Varje kommando måste följas av en fördröjning, så att ST7036-chipet hinner utföra instruktionen. Mongoose OS-funktionen *Sys.usleep()* kan vara lämplig.

**Example of initialisation: 8 bit / 3.3V**

**EA DOGM163**

| Befehl | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | Hex | Bemerkung |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Function Set | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | $39 | 8 bit data length, 2 lines, instruction table 1 |
| Bias Set | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | $15 | BS: 1/5, 3 line LCD |
| Power Control | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $55 | booster on, contrast C5, set C4 |
| Follower Control | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | $6E | set voltage follower and gain |
| Contrast Set | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | $72 | set contrast C3, C2, C1 |
| Function Set | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | $38 | switch back to instruction table 0 |
| Display ON/OFF | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | $0F | display on, cursor on, cursor blink |
| Clear Display | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | $01 | delete display, cursor at home |
| Entry Mode Set | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | $06 | cursor auto-increment |

Skriv kod som genomför initieringssekvensen enligt tabellen ovan. Se till att det fungerar som det ska innan du går vidare. Vid lyckad initiering rensas displayen och en blinkande markör syns på första raden uppe till vänster. Några *init.js*-klipp som kanske kan vara inspirerande:

```
load('api_spi.js');
load('api_sys.js');

let spi_h = SPI.get();
let spi_param = {cs: 0, mode: 0, freq: 100000, hd: {tx_data: "", rx_len: 0}};

function lcd_init(){
    lcd_cmd("\x39");
    Sys.usleep(30*1000);
      ... osv, osv ...
}

function lcd_cmd(cmd){
    GPIO.write(PIN_LCD_RS, 0);   // RS low for cmd
    spi_param.hd.tx_data = cmd;
    SPI.runTransaction(spi_h, spi_param);
}
```

Efter initieringen kan tecken skrivas till displayen genom att skicka ASCII-koder med hjälp av kommandot *Write Data to RAM*. Observera att RS-anslutningen ska ligga hög om ett tecken ska skrivas, och låg för övriga kommandon.

Tips: För att flytta cursorn/markören används kommandot *Set DDRAM Address.*