MODUL 3

Digital input

Knapp, buzzer, logik — och Arduinons första ingång från omvärlden.

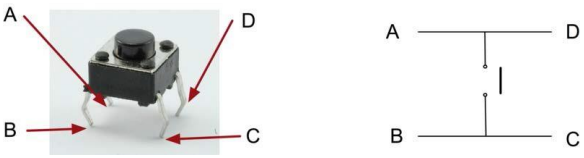
Vad du lärde dig idag

Den tredje träffen var ett paradigmskifte. Fram till nu har era program kört rakt fram: setup, loop, `digitalWrite`, `delay`. Ingen del av koden har reagerat på omvärlden. I dag fick programmen **läsa av** en knapp — och göra olika saker beroende på om den var tryckt eller inte.

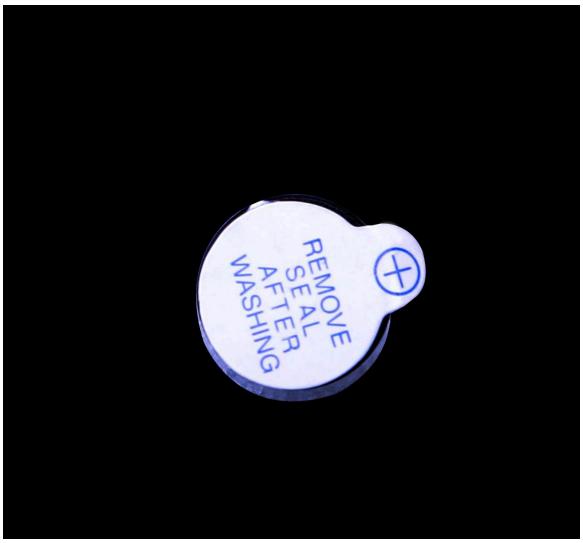
Kvällens nyheter:

- **`digitalRead(pin)`** läser av en pinne och returnerar `HIGH` eller `LOW`.
- **`pinMode(pin, INPUT_PULLUP)`** aktiverar en intern pullup-resistor i Arduinon, så du slipper koppla en egen. Sido-effekt: knappens logik blir **inverterad** — tryckt knapp läses som `LOW`, släppt som `HIGH`.
- **`if` och `else`** låter programmet välja olika väg beroende på ett villkor. Operatorerna `==`, `!=`, `<`, `>` jämför värden.
- **Edge-detection** är mönstret för att agera **en gång** på en knapptryckning, även om knappen hålls nere i hundra loop-varv. Nyckeln: spara förra värdet, jämför med nuvarande, agera bara när de skiljer sig åt på rätt sätt.
- **Active buzzer** är plug-and-play — `digitalWrite(buzzerPin, HIGH)` ger ljud, `LOW` ger tyst. Ingen `tone()`, ingen frekvens, ingen resistor. Men: **klisterlappen stannar på**.
- **Larm-mönstret** — ert första riktiga sense-act-loop **med minne**. Knappen togglar `larmPaslaget` via en flank, och buzzern speglar tillståndet med `digitalWrite(buzzerPin, larmPaslaget ? HIGH : LOW)`. Det är samma kärna som hackathonens tjuvlarm bygger på.

Ni byggde två kretsar parallellt (knapp + buzzer), kopplade ihop dem i en enda sketch, och såg Arduinons första sense-act-loop: den läser något från omvärlden, kommer ihåg ett tillstånd mellan loop-varven, och agerar därefter. Det är i princip allt embedded-programmering handlar om.



Tactile switch — fyra ben, men elektriskt två par. Tryckning kortsluter paren.



*Active buzzer — svart cylinder med klisterlapp och plusmarkering på ovansidan. Lappen **stannar kvar**.*

Repetition av de viktigaste begreppen

VARIABLER SOM KAN ÄNDRAS

I Modul 1 mötte ni `const int` — namngivna värden som aldrig ändras. I dag kom den första `int` utan const, och den första `bool`:

```
const int knappPin = 9;     // ändras aldrig
int tryckCount = 0;         // kan öka
bool larmPaslaget = false;  // kan togglas
```

Regeln är enkel: om värdet **ska kunna ändras under tiden programmet kör**, släpp `const`. Om det är bestämt från början (ett pin-nummer, en tröskel, en maxtid), behåll `const` — det gör koden lättare att läsa och hindrar dig från att råka ändra värdet av misstag.

Datatyper ni sett hittills:

`int`	Ett heltal. Kan vara positivt, negativt eller noll. På Uno: −32768 till 32767.
`const int`	Heltal som inte ändras efter deklaration.
`bool`	`true` eller `false`. Används för till/från-tillstånd.
`byte`	Ett heltal 0–255. Bra för PWM-värden och byte-fält.

En djupare genomgång av datatyper, scope och operatorer finns i Bilaga A.

`pinMode(pin, INPUT_PULLUP)` — tekniken förklarad

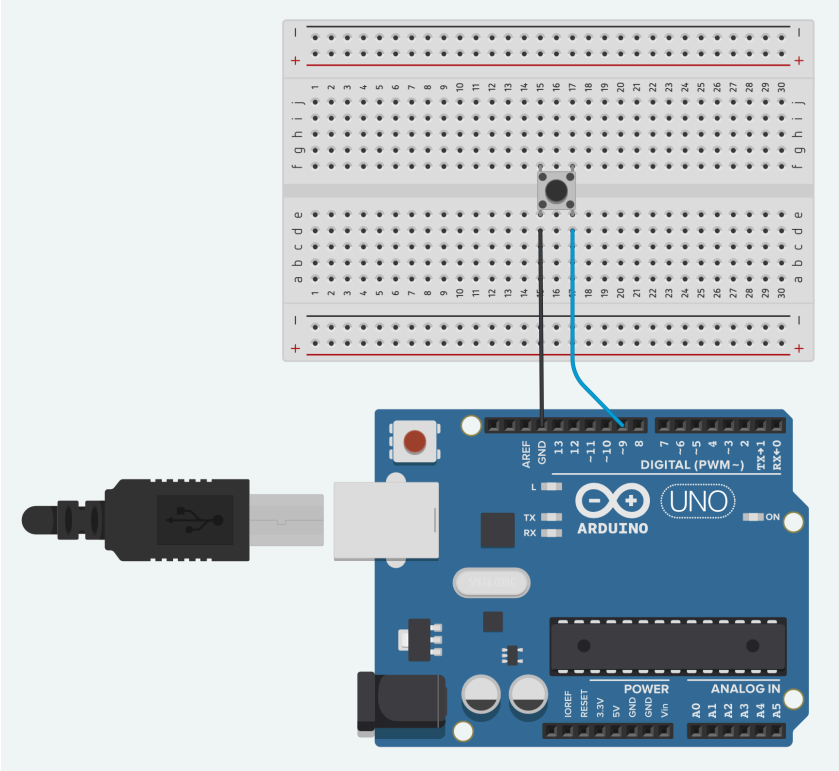
En Arduino-pinne som inte är kopplad till någonting **flyter**. Det betyder att dess spänning inte är bestämd — den kan vara något slumpmässigt mellan 0 och 5 V, ibland 2,4 V, ibland 0,1 V, ibland 4,7 V. `digitalRead` på en flytande pinne ger slumpmässiga resultat. Inte bra för en knapp.

Lösningen är en **pullup-resistor**: en resistor som kopplar pinnen till +5 V. När knappen är släppt, ”drar” pullup:en pinnen till HIGH. När knappen trycks, kortsluter den pinnen till GND och drar den till LOW. Alltid ett definitivt värde, aldrig flytande.

`INPUT_PULLUP` säger åt Arduinon att aktivera en **intern** pullup-resistor — den finns inbyggd i chippet. Du behöver aldrig löda, aldrig koppla en extern resistor. En rad kod: `pinMode(knappPin, INPUT_PULLUP);`.

Konsekvensen är den inverterade logiken. **Tryckt knapp = LOW**, **släppt knapp = HIGH**. Svårt att vänja sig vid. Varje gång ni läser koden, tänk högt: ”LOW betyder tryckt, LOW betyder tryckt”.

> **ÖVERSÄTTNINGSREGEL**
> När du läser en `INPUT_PULLUP`-pinne:
> - `LOW` → tänk **”tryckt”**
> - `HIGH` → tänk **”släppt”**
>
> Gör den substitutionen **innan** du läser resten av raden. Med tiden sker den automatiskt; i början är den medveten.



Knappkretsen på breadboard — knappen pluggas i mittspåret, ena benet via en kort kabel till `GND`-skenan, andra benet med en blå kabel till `D9`. Som output används Arduinons inbyggda LED på pin 13 — ingen extra LED behövs.

> **BRYGGA TILL MODUL 4**
>
> `INPUT_PULLUP` är i själva verket en intern **pullup-resistor** — en ”osynlig” resistor till +5 V som gör att en släppt knapp blir stabilt `HIGH`. I Modul 4 kommer du bygga en **riktig spänningsdelare** själv, med två resistorer, för att läsa av en fotocell. Skillnaden: i Modul 3 får du bara två digitala lägen (HIGH/LOW). I Modul 4 får du en mellanspänning som kan variera och som `analogRead` kan mäta.

`IF` / `ELSE`

Syntaxen är:

```
if (villkor) {
  // kör det här om villkoret är sant
} else {
  // kör det här om det är falskt
}
```

Villkoret är ett uttryck som är antingen sant eller falskt. De vanligaste jämförelserna:

`a == b`	**lika med**. Glöm inte det andra likhetstecknet — `=` är tilldelning, `==` är jämförelse.
`a != b`	**olika**

`a < b`	**mindre än**
`a > b`	**större än**
`a <= b`	mindre eller lika
`a >= b`	större eller lika

Den vanligaste nybörjarbuggen är att skriva `if (x = 5)` när man menar `if (x == 5)`. Kompilatorn gnäller inte — det är giltig C++ — men programmet gör inte det du väntar dig. Två likhetstecken.

EDGE-DETECTION: AGERA PÅ FLANKEN

Problemet: en knapp som hålls ner är `LOW` i hundratals loop-varv innan användaren släpper den. Om du gör något vid **varje** `LOW`, händer det hundratals gånger.

Lösningen: reagera bara på **övergången** från HIGH till LOW — **flanken**. Spara förra värdet, jämför, agera bara när de skiljer sig åt:

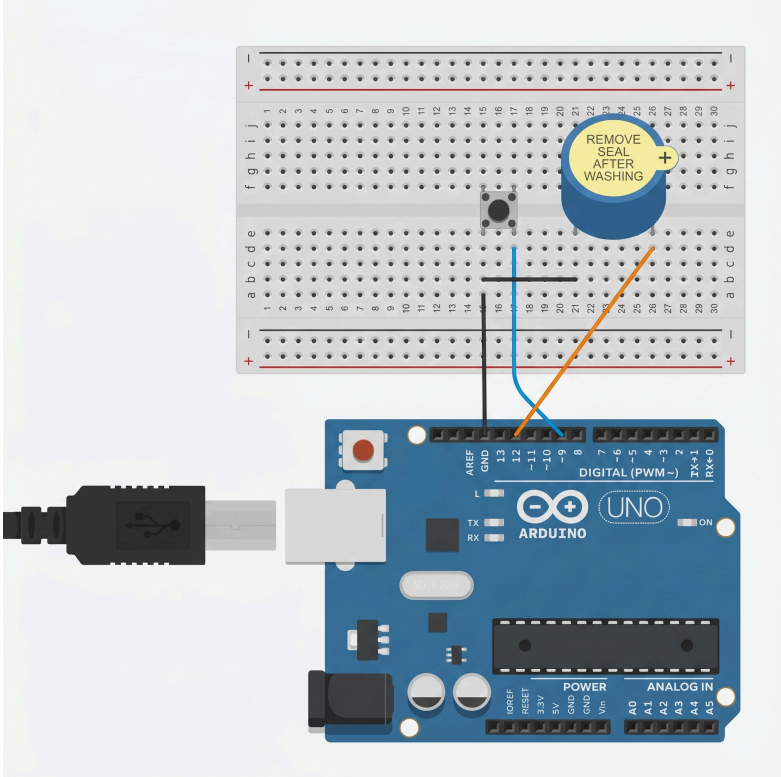
```
const int knappPin = 9;
bool larmPaslaget = false;
int lastState = HIGH;

void loop() {
  int state = digitalRead(knappPin);
  if (state == LOW && lastState == HIGH) {
    // flanken just nu – agera en gång
    larmPaslaget = !larmPaslaget;
  }
  lastState = state;
  delay(10);  // liten paus mot studs
}
```

Uttrycket `state == LOW && lastState == HIGH` fångar **fallande flanken** — ögonblicket då pinnen går från HIGH (släppt) till LOW (tryckt). Samma tekniker används i all digital elektronik: man reagerar på **flanken**, inte på nivån.

`!larmPaslaget` är **logisk inversion** — utropstecknet betyder ”det motsatta”. Om `larmPaslaget` var `false`, blir det `true`. Om det var `true`, blir det `false`. Det är så vi **togglar** ett tillstånd med en knapp.

`delay(10)` är en enkel **debounce**. Knappens metallblad studsar fysiskt några millisekunder när de möts, vilket ger flera falska flanker i rad. Tio millisekunder räcker för att släta över det i den här kursen. I produktion används fler tekniker — se Bilaga A.



Buzzern läggs på samma breadboard som knappen — `+`-benet (höger, markerat med `+`) går via en orange kabel till `D12`, `-`-benet via `GND`-skenan tillbaka till Arduinon. Ingen resistor.

ACTIVE BUZZER

Buzzern i kittet är **active** — svart cylinder med inbyggd oscillator och en liten klisterlapp på ovansidan. Det betyder att `digitalWrite(pin, HIGH)` räcker för att få ljud; ingen `tone()`, ingen frekvens, ingen resistor.

Active buzzer (vår)	Inbyggd oscillator. `digitalWrite(pin, HIGH)` → pip. `LOW` → tyst. Fast frekvens.
Passive buzzer (saknas i kittet)	Ingen oscillator. Kräver `tone(pin, hz)` för att ge ljud. Kan spela olika tonhöjder — bra för melodier.

Vi använder den aktiva för dess enkelhet. Om ni skulle köpa en lös buzzer i elektronikbutik kan det alltså vara motsatsen — fråga alltid vilken typ.

Dra inte av klisterlappen. Den är en fabriksdämpare från tillverkningsprocessen. Tekniskt fungerar buzzern utan, men den blir obehagligt hög. Lappen sitter på.

Bygg från minnet

Skriv från scratch larm-mönstret från lektionen — utan att titta. Krav: en knapptryckning ska **toggla** larmet (på/av), inte bara pipa medan knappen hålls nere. Det innebär edge-detection på knappen, en `bool larmPaslaget` som globalt tillstånd, och `digitalWrite` av buzzern utifrån det tillståndet.

Tryck en gång → larm på, buzzern tjuter. Tryck igen → tyst. Det här är det mönster ni byggde live på lektionen och som hackathonens tjuvlarm vidareutvecklar.

SKELETT

```
const int knappPin  = 9;
const int buzzerPin = 12;

bool larmPaslaget = false;
int  lastState    = HIGH;

void setup() {
  pinMode(knappPin,  INPUT_PULLUP);
  pinMode(buzzerPin, OUTPUT);
}

void loop() {
  int state = digitalRead(knappPin);
  if (state == LOW && lastState == HIGH) {
    larmPaslaget = !larmPaslaget;          // flank → toggla
  }
  digitalWrite(buzzerPin, larmPaslaget ? HIGH : LOW);
  lastState = state;
  delay(10);  // mot studs
}
```

Ladda upp. Tryck. Det tjuter — och stannar på tills du trycker igen. Du har byggt larm-mönstret från minnet — **kärnan** i hackathonens tjuvlarm.

OM FLANK-MÖNSTRET KRÅNGLAR

Som sista utväg: skala ner till hold-varianten — `if (digitalRead(knappPin) == LOW) digitalWrite(buzzerPin, HIGH); else digitalWrite(buzzerPin, LOW);`. Den tjuter bara medan du håller knappen — det är inget larm, men bekräftar att kopplingen fungerar. Testa, fixa flanken, gå vidare.

Hemma-övningar

ÖVNING 1 — TYSTARE VARIANT: TOGGLE MED LED + FLANK-PÅ-SLÄPP

Två extensioner till larm-mönstret från ”Bygg från minnet”. Båda i samma sketch.

Extension A — byt utgång till LED. Byt `buzzerPin` mot `LED_BUILTIN` (pin 13). Användbart för sen-natt-övande utan att bli osams med grannarna — och bekräftar att samma flank-mönster funkar på vilken digital utgång som helst.

Extension B — toggla på släppet **istället för trycket.** I ”Bygg från minnet” agerade du på fallande flanken (`state == LOW && lastState == HIGH`). Skriv om så larmet växlar när du **släpper** knappen (stigande flanken: `state == HIGH && lastState == LOW`). Skillnaden är liten i kod men stor pedagogiskt — du ser direkt att flank-detektion finns i två varianter, och att det är samma mönster med roller bytta.

> **TESTA DIG FRAM**
> Börja från ”Bygg från minnet”-sketchen. Byt `buzzerPin` mot `LED_BUILTIN` och lägg till `pinMode(LED_BUILTIN, OUTPUT);` i setup. Kör — det ska bete sig identiskt med klassens larm, fast tyst. Sedan vänd jämförelsen i if-satsen och se hur det känns att toggla på släppet.

ÖVNING 2 — RÄKNA KNAPPTRYCK OCH SKRIV UT

Skriv en sketch som räknar antalet knapptryckningar (med edge-detection) och skriver ut det totala antalet på Serial Monitor varje gång någon trycker. Du behöver lägga till `Serial.begin(9600);` i setup och `Serial.println(tryckCount);` i rätt del av loopen. (Serial Monitor gås igenom ordentligt nästa vecka i Modul 4 — den här övningen är en förhandstitt.)

Nyckelinsikten: `Serial.println` ska INTE ligga i `loop()` utanför `if` -satsen. Då printar den 10 000 gånger per sekund. Den ska köras **bara** när en ny tryckning har registrerats — alltså inne i det lilla blocket där flanken detekteras.

ÖVNING 3 — KNAPP + BUZZER + MORSE

Skriv ett program där du kan morsa med knappen och buzzern piper direkt. Snabbt tryck = kort pip. Långt tryck = långt pip. Detta är faktiskt lättare än det låter — följ knappens tillstånd direkt: `if knappen är LOW → buzzer HIGH, annars LOW` . Ingen edge-detection behövs.

För extra credit: lägg till Arduinons inbyggda LED som ”visuell bekräftelse” så det tänds samtidigt som buzzern.

Vanliga fel och snabblösningar

Knappen gör ingenting	Glömt `pinMode(pin, INPUT_PULLUP)` . Utan den flyter pinnen. Eller: kabeln sitter i fel Arduino-pin.
Knappen ”sitter fast” vid tryck	Du läste av `HIGH` som ”tryckt”. Kom ihåg: med INPUT_PULLUP är **LOW** = tryckt.
Larmet växlar av/på random när jag håller knappen	Du saknar edge-detection. Agera på FLANKEN, inte på tillståndet.

Buzzern tjuter konstant	Glömt `pinMode(buzzerPin, OUTPUT)`. Eller: du har skrivit `HIGH` utanför en if/else som aldrig vänder det tillbaka till `LOW`.
Serial Monitor skriver ut miljoner rader	Du printar direkt i loop utan if-skydd. Flytta `Serial.println` inuti if-blocket där det bara kör vid en ny händelse.
Knappen ger flera tryck åt gången	Studs. Lägg `delay(10);` på slutet av loopen — enklaste formen av debounce.

Snabbreferens

`pinMode(p, INPUT_PULLUP)`	Sätt upp som ingång med intern pullup.
`digitalRead(p)`	Returnerar `HIGH` eller `LOW`.
`if (x == y) { ... } else { ... }`	Villkorssats. Två likhetstecken för jämförelse.
`&&` `||` `!`	Logiska operatorer: och, eller, inte.
`!x`	Logisk inversion. `!true` = `false`, `!false` = `true`.
Active buzzer	`digitalWrite(pin, HIGH)` = pip. Pin 12 i kursen. Klisterlappen stannar på.
Edge-detection-idiom	Spara `lastState`, jämför med nuvarande, agera bara på övergång.
Larm-mönstret (kärnan)	Edge-detection togglar `larmPaslaget`; `digitalWrite(buzzerPin, larmPaslaget ? HIGH : LOW)` speglar tillståndet. Full sketch under ”Bygg från minnet”.

Inför nästa träff

Modul 4 är den sista innan hackathonen. Tre nyheter står på schemat: **analogRead** (att läsa ett tal, inte bara HIGH/LOW), **spänningsdelaren** (tekniken som låter Arduinon mäta motstånd), och **Serial Monitor på allvar** (Arduinons verktygslåda för felsökning).

Ni får också två nya komponenter: en **fotocell** (en resistor vars motstånd sjunker i ljus) och en **tilt-sensor** (en metallkula som kortsluter två ben när den lutas). Den sista är vår anti-stöld-trigger i det slutliga larmet.

Glöm inte dator eller kitet hemma!