FRO ÅNGE · NYBÖRJARKURS · VÅREN 2026

Elektronik & Programmering med Arduino

Kompendium · fem träffar · från första kretsen till fungerande tjuvlarm

Baserat på ELEGOO UNO Project Basic Starter Kit

FÖRORD

Detta är ditt kompendium.

Det du håller i handen är inte en lärobok i elektronik. Det är inte heller en bruksanvisning till Arduino. Det är en **referens** — den plats där du slår upp det ni gick igenom i klassrummet när det där ordet du behövde försvann igen.

Kompendiet följer samma struktur som de fem träffarna. Varje modulkapitel börjar med vad ni gjorde den kvällen, repeterar de viktigaste begreppen, och ger dig tre hemma-övningar att öva på i lugn och ro vid köksbordet. Sist i varje kapitel finns en snabbreferens du kan peka på när något trasslar.

Efter modulerna följer sex bilagor — en fördjupning av saker som inte fick plats på slidesen: syntax-grammatik (vad `const int` egentligen betyder), hur du läser felmeddelanden (Arduino-IDE:s felpanel är din bästa vän), en komponentlista med gotchor, en fullständig hackathon-lösning för den som fastnar, en säkerhets-checklista, och en ordentligt genomtänkt förklaring av Ohms lag och framspänningsfall.

Tonläget är avsiktligt direkt. Jag antar att du är en vuxen människa som kan lite redan, som inte är rädd för att ha fel, och som hellre får svaret rakt än paketerat i floskler. Ni är radioamatörer. Ni klarar det.

Om något är oklart: fråga vid nästa träff. Eller öppna Serial Monitor, skriv ut värdet, och se med egna ögon vad Arduinon faktiskt tänker. Det sista är ofta det som löser gåtan.

— kursledaren, april 2026

INNEHÅLL

Vad du hittar var.

MODUL 1

LED & krets

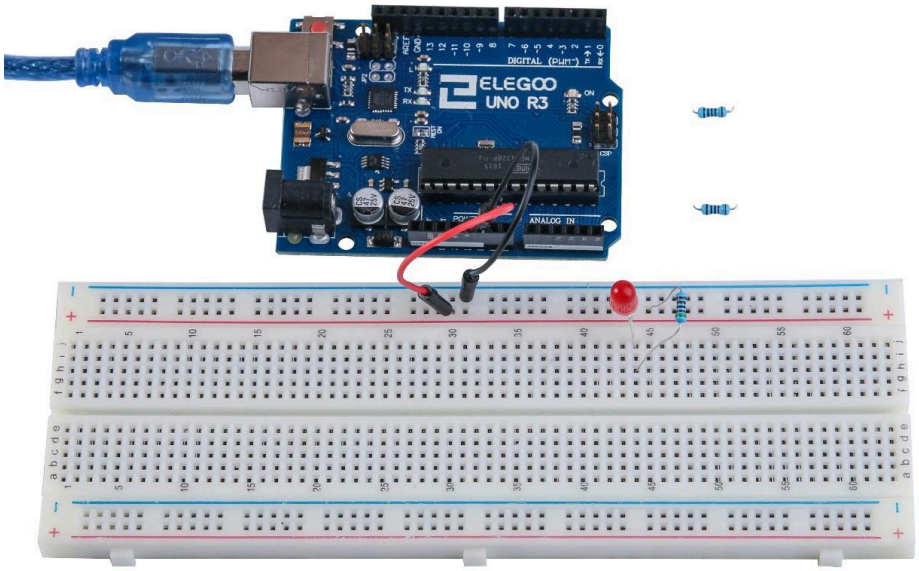
Digital output, Ohms lag, och en lampa som faktiskt blinkar.

Vad du lärde dig idag

Den första träffen handlade om att få något — vad som helst — att hända. En lampa som tänds, släcks, och blinkar i en rytm du själv bestämmer. Bakom det där enkla resultatet ligger fem idéer som hela resten av kursen vilar på.

- **En krets är en ring**. Strömmen måste lämna en pinne på Arduinon, passera genom dina komponenter, och komma tillbaka till GND. Bryts ringen händer ingenting.
- **Spänning och ström hör ihop men är inte samma sak**. Spänning är tryckskillnad (volt), ström är flöde (ampere). Ohms lag binder ihop dem via motstånd: $U = I \cdot R$.
- **En LED har polaritet**. Långt ben är plus (anod), kort ben med platt kant är minus (katod). Kör du den baklänges lyser den inte — men går inte sönder.
- **En LED behöver en resistor i serie**. Utan något som bromsar strömmen brinner dioden upp. En 220 Ω-resistor är ett säkert allround-val för 5 V-matning.
- **`digitalWrite` styr en pinne mellan två lägen**: `HIGH` (5 V ut) och `LOW` (0 V). Det är allt som händer i `Blink` under huven.

Ni kopplade blink-kretsen på breadboarden, laddade upp IDE:ns färdiga `Blink`-exempel, och ändrade sedan koden för att få er egen rytm — många valde SOS i morse.

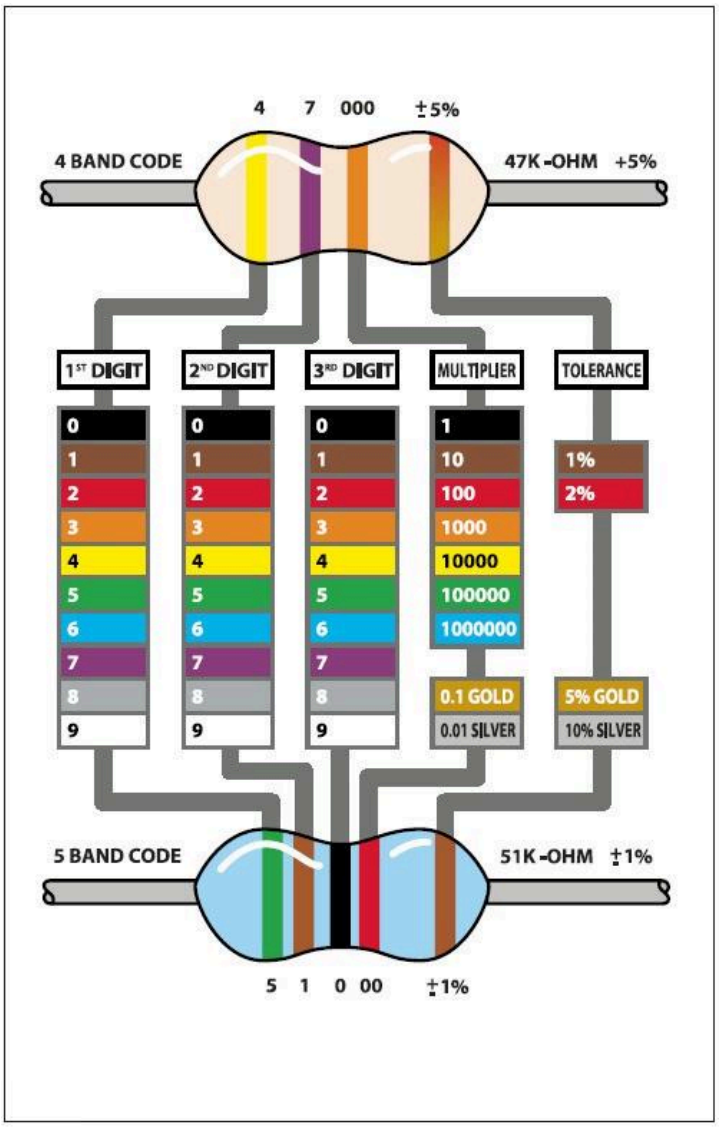


Arduino + LED + 220 Ω-resistor på breadboard. Lägg märke till att 1 kΩ och 10 kΩ som syns i bilden bara är alternativa värden — kursen använder 220 Ω.

Repetition av de viktigaste begreppen

RÄKNA BANDEN FÖRST

Kittet innehåller två sorters resistorer: **4-bands** (digit-digit-multiplikator-tolerans) och **5-bands** (digit-digit-digit-multiplikator-tolerans, 1 % precision). Titta på **din** resistor och räkna bandens antal innan du börjar dekoda. 220 Ω är **röd-röd-brun-guld** (4-band) eller **röd-röd-svart-svart-brun** (5-band). Osäker? Mät med multimeter.



Färgkodschart. Överst: 4-band (första två siffror, multiplikator, tolerans). Underst: 5-band (tre siffror, multiplikator, tolerans). Samma tabell sitter inuti locket på ditt kit.

KRETSEN

Du har fyra saker på breadboarden: Arduinon, en LED, en 220 Ω-resistor, och två kablar. Strömmen går ut från digital pin 13 när den är `HIGH`, in i LED:ens långa ben, ut genom korta benet, in i ena änden av resistorn, ut från andra änden, och tillbaka in i Arduinons GND-pinne. Det är en sluten ring.

Bryts ringen händer ingenting. En lös kabel, ett ben som sitter i fel hål, en resistor som glömts bort — alla tre är samma fel: ringen är inte sluten. Felsök genom att följa strömmen med fingret, hål för hål, från pin 13 tillbaka till GND.

> **VATTENANALOGIN**
>
> Om elektricitet känns abstrakt: tänk vatten. **Spänningen** (volt) är trycket i systemet — som hur högt en vattentank står över ditt kök. **Strömmen** (ampere) är flödet — hur

mycket vatten som faktiskt rör sig per sekund. **Motståndet** (ohm) är hur mycket röret stryper flödet — som en halvöppen kran. En **resistor** är en strypventil.

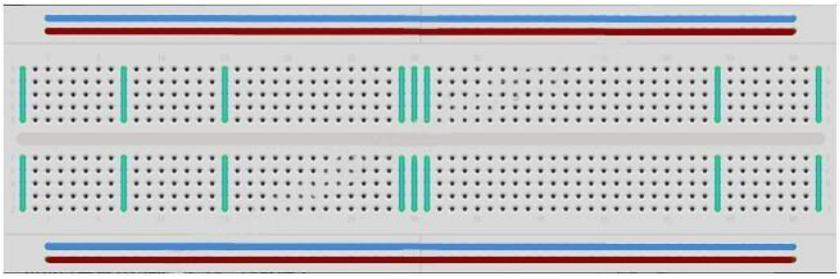
Ohms lag säger: för ett givet tryck bestämmer strypningen flödet. Stryp mer → mindre flöde. Lossa på → mer flöde. Vi kommer återvända till den här bilden flera gånger under kursen, bland annat när vi bygger spänningsdelaren i Modul 4.

BREADBOARDEN

Plastbiten du byggde på är ihålig med små metallklämmor invändigt. **Fem hål i rad är samma elektriska nod.** Raden bredvid är en helt annan nod. Gapet i mitten av brädan bryter förbindelsen mellan övre och nedre halvan.

Längs sidorna löper två par långa rälar: $+$ och $-$. Dessa **power rails** kommer bli viktiga först i Modul 2, men tänk på dem som ”ett långt hål vardera” redan nu.

När LED:en inte lyser: kolla först att kabeln och komponentbenet faktiskt sitter på **samma rad**. Nio av tio nybörjarfel är ett hål fel.



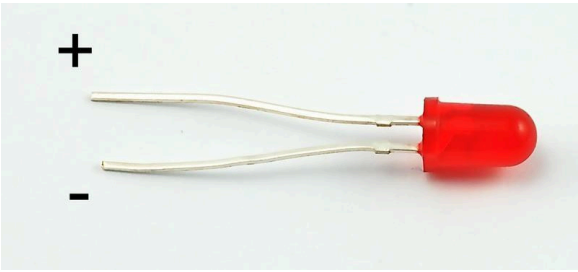
Breadboarden ovanifrån, med röda och blåa power rails längs sidorna och de gröna streck som visar vertikala nod-kolumner i mittsektionen. Gapet i mitten bryter förbindelsen mellan övre och nedre halvan.

OHMS LAG

Tre storheter, en formel:

$$U = I \cdot R$$

Omformulerat: $I = U/R$, $R = U/I$. Håll för den storhet du vill räkna ut, så ser du vilka de andra två ska göras av. Fullständig förklaring med exempel finns i Bilaga F.



LED:ens polaritet — långt ben (anod) till plus, kort ben med platt kant (katod) till GND.

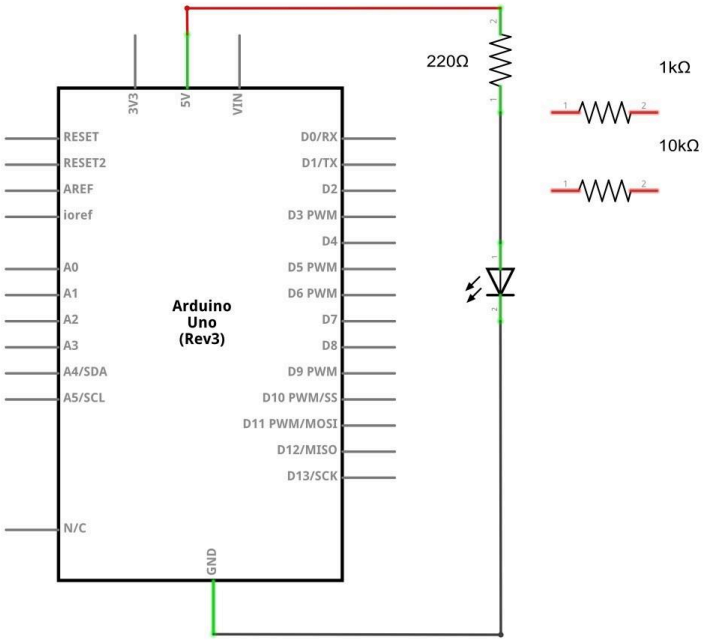
Resistorn läses från den ände där färgbanden ligger tätast. 220 Ω = röd-röd-brun-guld (4-band) eller röd-röd-svart-svart-brun (5-band). Bilaga C har full tabell.

VARFÖR 220 Ω?

En vanlig röd LED har ett **framspänningsfall** på ungefär 2 V. Det betyder att när du skickar 5 V över kombinationen LED + resistor, så ”äter LED:en upp” 2 V och resistorn måste ta hand om resten. Vi siktar på en ström runt 15 mA genom LED:en — ett säkert riktvärde långt under datablads-maxvärdet på 20 mA, men tillräckligt för att lysa synligt:

$$R = (5\ \text{V} - 2\ \text{V}) / 15\ \text{mA} = 200\Omega$$

220 Ω är det närmaste standardvärdet, och det ger en ström runt 13,6 mA — inom säkert område för varje LED i kittet. Djupare förklaring av varför LED:en inte följer Ohms lag direkt finns i Bilaga F.



Schematisk krets för Blink: Arduino Uno → D13 → LED (anod) → katod → 220 Ω → GND. Den här symboliska ritningen är densamma som breadboard-kopplingen — bara på ett annat språk.

`DIGITALWRITE`, `PINMODE` OCH `LED_BUILTIN`

`pinMode(LED_BUILTIN, OUTPUT)` talar om för Arduinon: ”den här pinnen är en utgång, jag ska styra den”. `digitalWrite(LED_BUILTIN, HIGH)` sätter pinnen till 5 V. `digitalWrite(LED_BUILTIN, LOW)` sätter den till 0 V. `delay(ms)` är en paus i antal millisekunder. Allt `Blink` gör är att växla mellan HIGH och LOW med pauser emellan.

`LED_BUILTIN` är Arduinons eget **alias** för pin 13 — exakt samma pinne, bara ett annat namn. När du skriver `digitalWrite(LED_BUILTIN, HIGH)` går samma ström ut genom pin 13-hålet som när du skriver `digitalWrite(13, HIGH)`. Varför två namn? Pin 13 är fysiskt kopplad till en liten ytmonterad LED på kortet (märkt ”L”) — permanent, via sitt eget motstånd fabriksvägen. Arduinons example-sketchar använder `LED_BUILTIN` så att samma kod fungerar

på alla deras kort, även de där inbyggda LED:en inte sitter på pin 13. Vi använder samma namn i kursen för att matcha Arduino IDE:s exempel.

När ni blinkar `LED_BUILTIN` blinkar alltså både den lilla inbyggda LED:en på kortet **och** er externa LED på breadboarden — båda är kopplade till samma pin. Den externa LED:en bygger ni för att se en krets ni själva monterat.

> **OM UPLOAD INTE GÅR IGENOM**
> – Tools → Port → välj den port som dyker upp när ni stoppar in USB-kabeln (`/dev/cu.usbmodem*` på Mac, `COM*` på Windows).
> – Tools → Board → ”Arduino Uno”.
> – Byt USB-kabel om porten inte dyker upp alls — vissa billiga kablar saknar dataledare.
>
> Fullständig felsökning i Bilaga B, avsnitt ”Upload-fel”.

Bygg från minnet

Lägg undan slidesen. Ta fram Arduinon, en LED, en resistor och två kablar. Försök koppla blinkkretsen utan att titta på bild. När lampan blinkar: grattis — du kan det.

Fastnar du, peka ut dessa fyra frågor:

1. Vilken pinne ska ström gå **ut** ur?
2. Vilket ben på LED:en ska strömmen in i?
3. Var i ringen ska resistorn sitta?
4. Vilket ben på Arduinon ska strömmen tillbaka till?

Svaren: pin 13, långa benet, i serie någonstans i ringen, GND.

Hemma-övningar

ÖVNING 1 — HJÄRTSLAG

Ändra `Blink` så att lampan blinkar i ett hjärtslagsmönster: två korta blink tätt efter varandra, sedan en längre paus, och om igen.

> **LEDTRÅD**
> Två `digitalWrite(HIGH)` / `digitalWrite(LOW)`-par med korta `delay()` emellan, och sedan ett längre `delay()` innan loopen börjar om.

ÖVNING 2 — NAMNGIVNA TIDER

Skriv om din hjärtslags-kod så att alla tidsvärden ligger i `const int`-variabler i toppen av filen. Ändra sedan bara variablerna för att testa ett snabbare tempo. Syftet: du ska aldrig behöva leta upp fem `delay`-siffror och ändra dem var för sig.

> **EXEMPEL**
> `const int kort = 80;` `const int lang = 200;` `const int paus = 1000;` — och sedan använda variabelnamnen i stället för siffror inne i `loop()`.

ÖVNING 2½ — MAMMA-IGENKÄNNING

Innan morse — bygg en övergång. Skriv ett mönster som är tydligt igenkännbart från vanlig Blink: **tre snabba blink** (100 ms på, 100 ms av) följt av en **lång paus** (1500 ms). Använd de `const int`-variabler du skapat i övning 2.

Detta kluster-av-blink är faktiskt morse-S — och byggstenen i nästa övning. Om du kan kluster-av-tre kan du morsa vilken bokstav som helst.

ÖVNING 3 — DINA INITIALER I MORSE

Slå upp morsealfabetet och få lampan att blinka dina initialer. `·` = en kort blinkning, `–` = en lång blinkning (tre gånger så lång som en kort). Mellan två bokstäver: en kort paus. I slutet av hela meddelandet: en längre paus innan det börjar om.

> **BÖRJA ENKELT**
> Du behöver bara `digitalWrite` och `delay` — ingen ny syntax. Skriv ut varje dit och dah som ett par: `digitalWrite(LED_BUILTIN, HIGH)`, sedan `delay(...)`, sedan `digitalWrite(LED_BUILTIN, LOW)`, sedan `delay(...)`. Bokstaven ”S” blir tre korta blink i följd. Det blir många rader kod — det är OK, du lär dig att upprepa. Om du senare vill rensa upp det, introducerar vi **funktioner** i Bilaga A — en funktion är ett namngivet kodblock man kan anropa om och om igen.

Vanliga fel och snabblösningar

LED lyser inte	Vänd LED:en — långt ben mot pin 13, kort mot resistorn.
LED lyser väldigt svagt	Kontrollera resistor-värdet. 220 Ω = röd-röd-brun-guld (4-band) eller röd-röd-svart-svart-brun (5-band). Om du dekodat fel kan du fått 2,2 kΩ (10× för hög) — dubbla multiplikatorn. Mät med multimeter vid tvivel.
`Blink` kompilerar men inget händer	Kabeln till pin 13 sitter i pin 12 eller GND (hålet precis bredvid pin 13). Räkna hålen.
`expected ';' before...`	Glömt semikolon på föregående rad. Arduino-IDE:ns felpanel pe-

	kar på nästa rad, felet är på raden ovanför.
`'digitalWrit' was not declared in this scope`	Stavfel — funktionen heter `digitalWrite`, du missar `e` på slutet. Arduino är också skiftlägeskänsligt, så `digitalwrite` (litet w) är också fel.
Upload misslyckas med `avrdude: stk500_getsync()`	Välj rätt port under Verktyg → Port. USB-kabeln (Uno använder Type B, den kvadratiska kontakten) måste vara en **datakabel**, inte en ren laddkabel.

Snabbreferens

`pinMode(pin, OUTPUT)`	Sätter upp pinnen som utgång. Görs en gång, i `setup()`.
`digitalWrite(pin, HIGH)`	Slår på pinnen (5 V). Strömmen går ut.
`digitalWrite(pin, LOW)`	Slår av pinnen (0 V). Ingen ström.
`delay(ms)`	Paus i millisekunder. `delay(1000)` = en sekund.
`const int name = 13;`	Ger ett tal ett namn. Används för pin-nummer och andra värden som inte ändras.
`LED_BUILTIN`	Arduinons inbyggda namn på pin 13. En liten ytmonterad LED (med eget seriemotstånd på kortet) är kopplad dit fabriksvägen — men den gäller **bara** den. Kopplar du en egen LED i header-hålet för pin 13 måste du fortfarande ha ett eget 220 Ω-motstånd i serie.

Inför nästa träff

Idag har Arduinon bara **pratat** till omvärlden — skickat ström ut genom en pinne. Det kallas **act** i embedded-världen. Från och med Modul 3 lär vi den också **lyssna** — läsa tillstånd in från knappar och sensorer. Mönstret **sense → act** — ”läs av → agera” — är grunden för allt ni kommer att bygga. Tänk på det hädanefter.

Men innan sense: en ny utgångs-teknik. Nästa vecka släpper vi `digitalWrite` i två minuter och lär oss `analogWrite` — ett kommando som inte bara kan sätta pinnen PÅ eller AV, utan kan ställa den någonstans **däremellan**. Det är det som gör det möjligt att blanda färg på en RGB-LED.

Ingen ny matematik, men en ny idé: Arduinon kan inte göra analog ström på riktigt — den **fuskar**, genom att blinka pinnen jättesnabbt. Hur snabbt? Det ska vi titta på.

Glöm inte dator eller kitet hemma!

MODUL 2

PWM & RGB

analogWrite, färgblandning, och en pixel som ni programmerar själva.

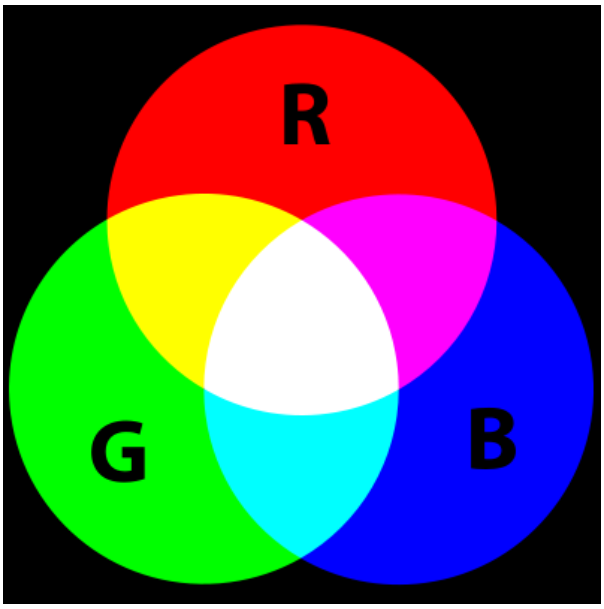
Vad du lärde dig idag

Den andra träffen handlade om att ge Arduinon ett nytt verb. Fram till nu kunde den bara säga ”PÅ” och ”AV” — `digitalWrite(pin, HIGH)` eller `digitalWrite(pin, LOW)`. I dag fick den säga ”någonstans emellan”.

Kvällens nyheter i punktform:

- `analogWrite(pin, värde)` tar ett tal från **0** till **255**. 0 = helt släckt. 255 = helt tänt. 128 = ungefär halvstyrka.
- **Arduinon kan inte göra halvstyrka på riktigt**. Den fuskar genom att blinka pinnen väldigt snabbt — flera hundra gånger i sekunden. Ögat hinner inte med och upplever blandningen som en mellannivå. Tekniken heter **PWM**, Pulse Width Modulation.
- **Bara vissa pinnar stödjer PWM**. På Arduino Uno är det pinnarna markerade med `~` på kretskortet: **3, 5, 6, 9, 10, 11**.
- **RGB-LED:en blandar färg ur tre kanaler**: röd, grön, blå. Precis som en pixel på skärmen ni sitter framför. Alla färger ni ser är kombinationer av dessa tre — det finns ingen egen gul lysdiod inblandad.
- **Kittets RGB-LED är en common cathode**. Det betyder att det längsta benet — katoden — går till GND, och de tre andra till PWM-pinnar via varsin 220 Ω-resistor.

Ni kopplade RGB-LED:en, upptäckte att pin-ordningen är rött/katod/grönt/blå (inte intuitivt — katoden är **andra** benet, inte i mitten), och experimenterade med att blanda fram lila, gammelrosa, cyan, skolgul.



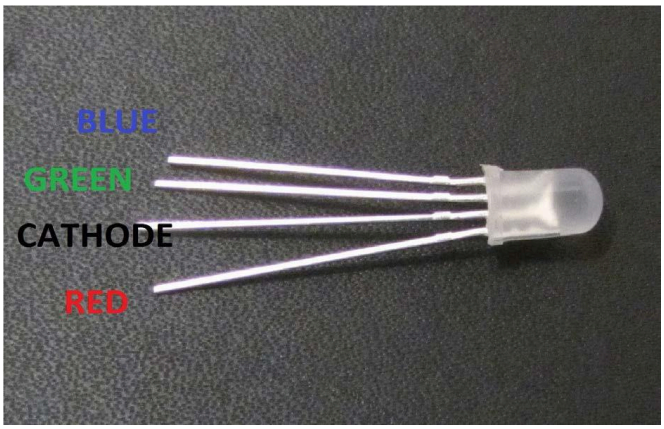
Additiv färgblandning — tre färgkanaler (rött, grönt, blått) som överlappar ger gult, cyan, magenta och (alla tre) vit. Samma princip som för skärmens pixlar.

> **TÄNK SKÄRM, INTE FÄRGBURK**
> RGB-LED:ens blandning är **additiv** — du lägger till ljus och närmar dig vitt. Målarens färglära är **subtraktiv** — du lägger till pigment och närmar dig svart. Därför: röd + grön = **gul** på er LED, men **brun-mudd** i akvarell. Den som bygger intuition åt fel håll här kommer att fastna senare i kursen.

Repetition av de viktigaste begreppen

RGB-LED:ENS PINOUT

Ditt första möte med en komponent som INTE tål att kopplas baklänges utan att krångla. Håll LED:en med den platta sidan mot dig:



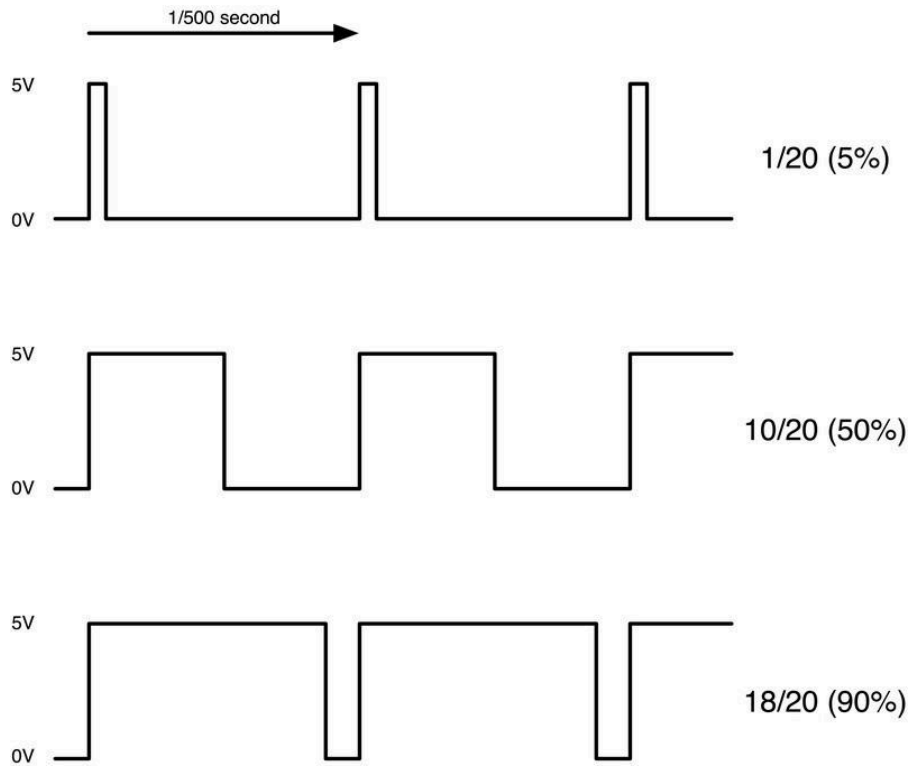
*RGB-LED:ens fyra ben, ordning från **platta sidan**: **röd · katod · grön · blå**. Katoden är det **andra** benet från platta sidan och längst av de fyra.*

Ben 1 (från platta sidan)	Röd anod — via 220 Ω → **D6**
Ben 2 — längst	**Gemensam katod** — direkt till **GND**
Ben 3	Grön anod — via 220 Ω → **D5**
Ben 4	Blå anod — via 220 Ω → **D3**

> **INTE COMMON ANODE**
> Det finns två sorters RGB-LED:er: **common cathode** (katoden är gemensam) och **common anode** (anoden är gemensam). Kittets variant är common cathode. Kopplar du som common anode lyser den inte. Värt att veta om du köper en LED lös i elektronikbutiken — fråga alltid vilken typ.

PWM OCH DUTY CYCLE

PWM fungerar så här: pinnen blinkar mellan HIGH och LOW flera hundra gånger i sekunden. Andelen tid den är HIGH kallas **duty cycle**.



PWM vid tre olika duty cycles: 5 %, 50 % och 90 %. Bilden visar en generaliserad 500 Hz. Verklig frekvens på Uno är 490 Hz (de flesta pinnar) eller 980 Hz (pin 5 och 6). Ögat hinner inte med och upplever medeleffekten.

- `analogWrite(pin, 0)` → pinnen är alltid LOW → duty cycle 0 % → släckt.
- `analogWrite(pin, 64)` → HIGH ungefär 25 % av tiden → svag lyster.
- `analogWrite(pin, 128)` → HIGH 50 % av tiden → halvstyrka.
- `analogWrite(pin, 255)` → pinnen är alltid HIGH → duty cycle 100 % → full.

På Arduino Uno växlar det mellan HIGH och LOW flera hundra gånger i sekunden. Pin 5 och pin 6 kör på ca **980 Hz**, de övriga PWM-pinnarna på ca **490 Hz**. För ögat är båda omöjliga att urskilja. För vissa känsliga sensorer eller kameror kan PWM-flimmer däremot synas — men det påverkar inte den här kursen.

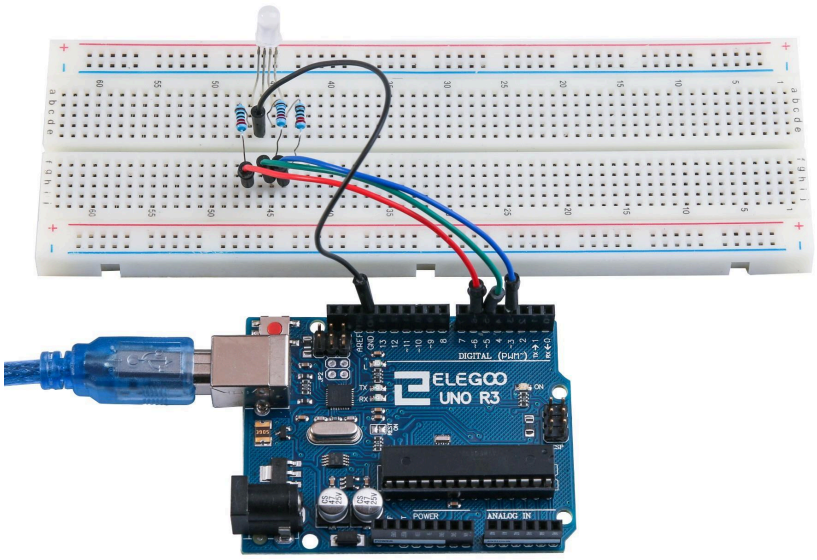
> **DÄRFÖR `~`**
> Pinnarna med tilde-tecken på Arduinon (3, 5, 6, 9, 10, 11) har hårdvarustöd för PWM. Om du av misstag använder `analogWrite` på pin 7 händer inget synligt fel, men LED:en kommer antingen vara helt av eller helt på — ingen mellannivå.

FÄRGBLANDNING I PRAKTIKEN

Tänk så här: varje kanal (röd, grön, blå) är en tonskala från 0 till 255. Du är en målare med tre tuber. Du kan klämma ut hur mycket du vill av varje, och resultatet blir en mix. Några startvärden att testa:

Röd	R 255, G 0, B 0
Grön	R 0, G 255, B 0
Blå	R 0, G 0, B 255
Vit	R 255, G 255, B 255
Gul	R 255, G 255, B 0
Cyan	R 0, G 255, B 255
Magenta	R 255, G 0, B 255
Lila	R 180, G 0, B 220
Gammelrosa	R 255, G 140, B 180
Skolgul	R 255, G 200, B 0
Mörk cyan	R 0, G 80, B 80

Du kommer märka att ögat INTE upplever 128 som ”hälften så ljust” som 255. Det beror på att ljusuppfattning är logaritmisk — vi är mer känsliga för skillnader i mörker än i ljus. Räkna inte med matematisk exakthet, experimentera fram tills det ser rätt ut.



En RGB-LED kopplad på breadboarden via tre 220 Ω-resistorer till PWM-pinnarna D3, D5, D6 och katoden till GND. Samma koppling som ni byggde.

Bygg från minnet

Skriv en sketch från scratch som cyklar mellan tre färger var fjärde sekund: röd → grön → blå → röd igen. Du behöver:

1. Tre `const int`-variabler för pin-numren (`ledR`, `ledG`, `ledB`).
2. En `setup()` som sätter alla tre som OUTPUT.
3. En `loop()` som anropar `analogWrite` på alla tre kanalerna med rätt värden, pausar, och byter färg.

SKELETT

```
const int ledR = 6;
const int ledG = 5;
const int ledB = 3;

void setup() {
  pinMode(ledR, OUTPUT);
  pinMode(ledG, OUTPUT);
  pinMode(ledB, OUTPUT);
}

void loop() {
```

```
  analogWrite(ledR, 255); analogWrite(ledG,   0); analogWrite(ledB,
0);
  delay(4000);
  // ... fortsätt med grön och blå
}
```

Hemma-övningar

ÖVNING 1 — DIN EGEN FÄRG

Hitta en kombination av R, G, B-värden som matchar en färg du tycker om i verkligheten. Åk ut och titta på en solnedgång, bilens instrumentbräda, tapeten hemma. Försök återskapa. Spara värdena i en sketch med kommentarer: `// solnedgång 19:42`.

ÖVNING 2 — FYRA-FÄRGS-ROTATION

Utöka ”Bygg från minnet”-övningen till fyra färger som roterar i loop: röd → gul → grön → blå → röd. Varje färg i två sekunder. Notera: gul är R 255 + G 255, inte en egen färg.

ÖVNING 3 — LÅNGSAM ÖVERGÅNG

Skriv en sketch som långsamt **tonar** över från röd till blå under 5 sekunder, istället för att hoppa. Detta kräver en `for`-loop — en ny konstruktion vi inte tagit upp på lektionen men som är lätt att läsa:

`FOR`-LOOP FÖR ÖVERGÅNGAR

```
for (int i = 0; i <= 255; i++) {
  analogWrite(ledR, 255 - i);   // röd sjunker
  analogWrite(ledB, i);         // blå stiger
  delay(20);                    // total tid: 255 × 20 ms ≈ 5 s
}
```

`for (int i = 0; i <= 255; i++)` — tre delar separerade med semikolon:

- `int i = 0` → **start:** skapa räknaren `i` och sätt till 0.
- `i <= 255` → **villkor:** kör så länge detta är sant.
- `i++` → **steg:** efter varje varv, öka `i` med 1.

Fullständig förklaring av `for` finns i Bilaga A.

Vanliga fel och snabblösningar

RGB-LED:en lyser bara i en färg	Två av tre ben är i fel hål, eller två resistorer saknas. Kontrollera att VARJE färg-ben har sin egen 220 Ω-resistor till sin PWM-pinne.

LED:en blinkar märkligt	Du har råkat koppla den som common anode — du har dragit katoden till +5 V istället för GND. Vänd på kopplingen.
`analogWrite(ledR, 200)` gör inget	Du har använt en pinne som inte stödjer PWM. Kolla att det finns ett `~` framför pin-numret på Arduinon. Giltiga: 3, 5, 6, 9, 10, 11.
Fel färg kommer ut	Du har blandat ihop vilket ben som är röd, grön, blå. Kom ihåg: från platta sidan är ordningen röd, katod, grön, blå.
Färgerna är inverterade (0 = full, 255 = av)	Du har en **common anode**-RGB istället för common cathode. Kittets LED är common cathode, men om du köpt en lös LED i elektronikbutik kan det vara motsatsen. Koppla längsta benet till `+5 V` istället för GND — då fungerar den, men du måste också invertera logiken i koden: `analogWrite(ledR, 255 - värde)`.
LED:en blinkar inte alls, men är ansluten	Glömt `pinMode(ledR, OUTPUT)` för en eller flera kanaler. Alla tre måste upp som OUTPUT i `setup()`.
Programmet fungerade, nu gör det inget	Du har glömt att ladda upp den nya versionen efter en ändring. Klicka på pilen uppe till vänster i IDE:n.

Snabbreferens

`analogWrite(pin, v)`	Skickar ut PWM med duty cycle $v/255$. Pinnen måste vara en PWM-pinne (`~`).
`pinMode(pin, OUTPUT)`	Samma som tidigare — fungerar för alla digitala pinnar, även PWM-pinnar.
PWM-pinnar på Uno	3, 5, 6, 9, 10, 11 — markerade med `~` framför pin-numret.
Kittets RGB-LED	Common cathode. R/K/G/B från platta sidan. Varje färg-ben via 220 Ω.
Kursens pin-ordning	R → D6, G → D5, B → D3.

Inför nästa träff

Modul 1 och 2 har varit rent **act** — Arduinon har bara skickat ström ut till LED-ar. Modul 3 lägger till den andra halvan: **sense**. Arduinon får sitt första öra.

Vi introducerar `digitalRead` och `INPUT_PULLUP`, bygger en första sketch som reagerar på knapptryck, lägger till en buzzer (den med den viktiga klisterlappen), och slutligen bakar in

edge-detection — den tekniska detalj som gör det möjligt att toggla ett tillstånd på och av utan att knappen skriver över sig själv 50 gånger per sekund.

Glöm inte dator eller kitet hemma!

MODUL 3

Digital input

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

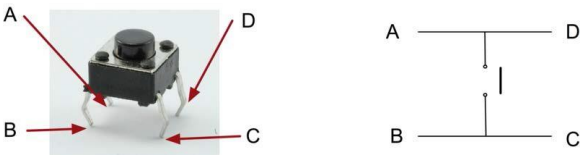
Vad du lärde dig idag

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.

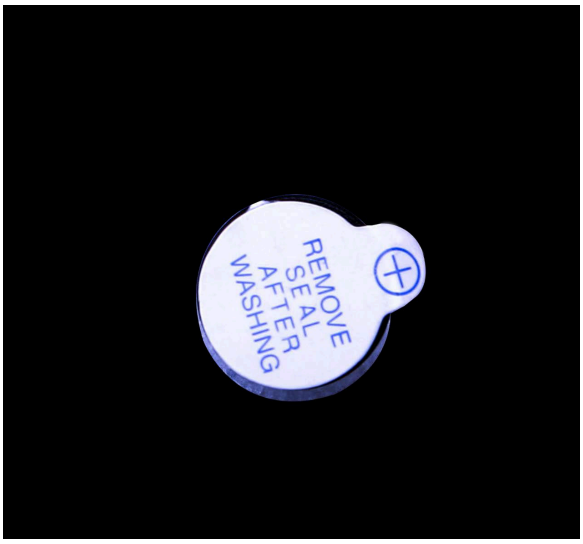
Kvällens nyheter:

- **`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

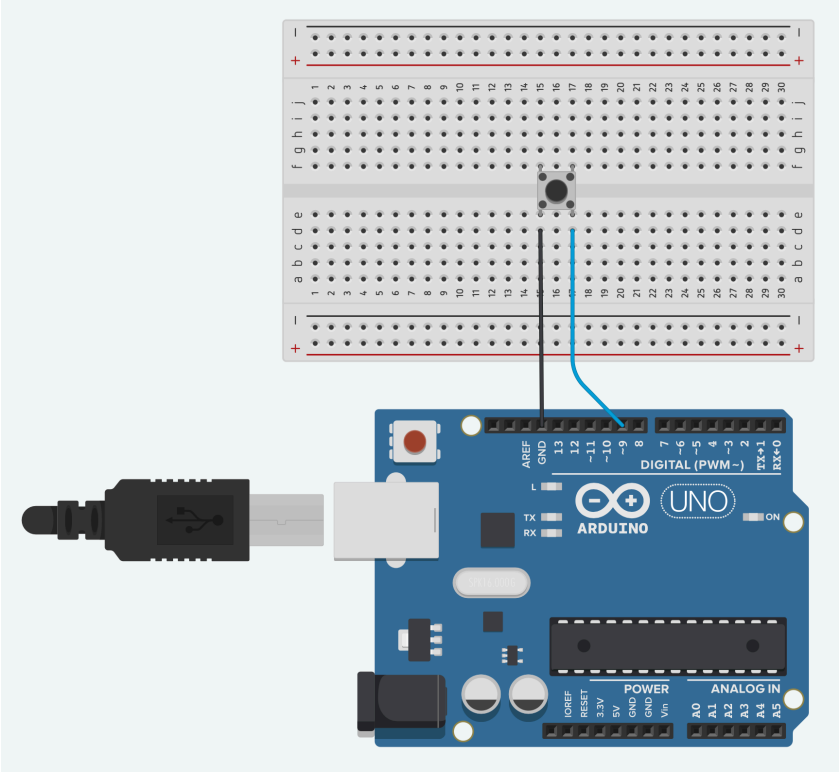
En Arduino-pinne som inte är kopplad till någonting **flyter**. Det betyder att dess spänning inte är bestämd — den kan vara något slumpmässigt mellan 0 och 5 V, ibland 2,4 V, ibland 0,1 V, ibland 4,7 V. `digitalRead` på en flytande pinne ger slumpmässiga resultat. Inte bra för en knapp.

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:

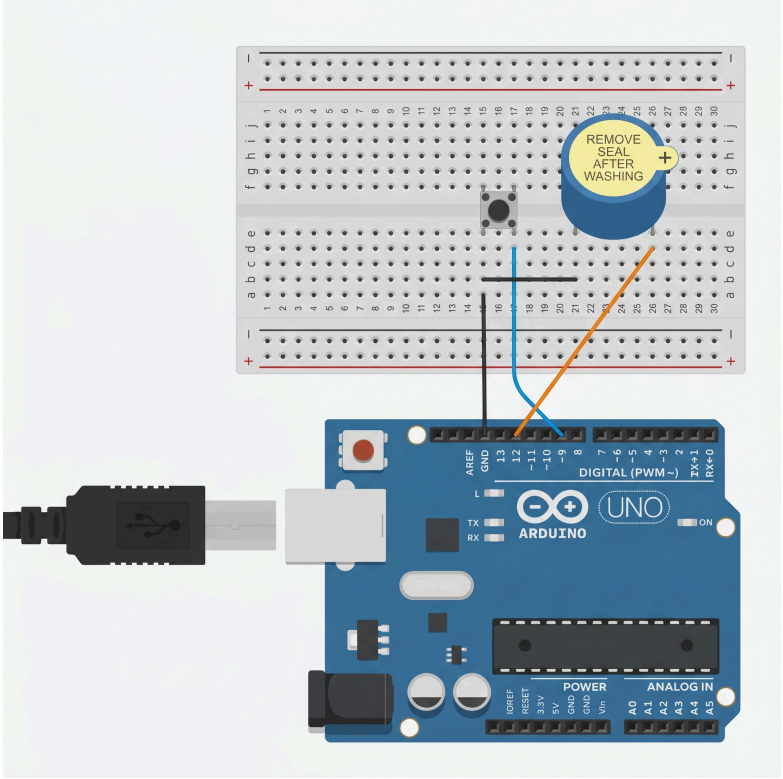
```
const int knappPin = 9;
bool larmPaslaget = false;
int lastState = HIGH;

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!

MODUL 4

Analog input

Sensorer, spänningsdelare, Serial Monitor — Arduinons fönster mot världen.

Vad du lärde dig idag

Fjärde träffen öppnade upp en ny dimension. Modul 3 lärde Arduinon säga ”tryckt / inte tryckt” — en av två möjliga svar. Modul 4 lär den säga ”500 av 1023” — **graderade** svar.

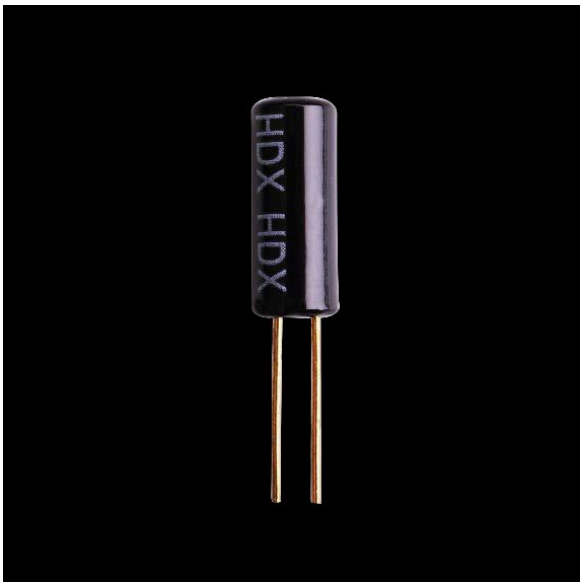
Kvällens nyheter:

- `analogRead(pin)` mäter spänningen på en analog pinne (A0–A5 på Uno) och returnerar ett heltal **0–1023**. 0 = pinnen på 0 V, 1023 = pinnen på 5 V, däremellan linjärt.
- `Serial.begin(9600)` och `Serial.println(värde)` låter Arduinon skicka text tillbaka till datorn över USB. Du ser det i **Serial Monitor** — förstoringsglas-ikonen uppe till höger i IDE:n.
- **Spänningsdelaren** är tekniken som gör det möjligt att läsa av ett motstånd med en pinne som bara kan mäta spänning. Två resistorer i serie, mellanpunkten går till A0.
- **Fotocellen** (LDR) är en resistor som ändrar värde med ljus: ca 50 kΩ i mörker, ca 500 Ω i starkt ljus. I serie med en fast 1 kΩ-resistor bildar den en spänningsdelare du kan läsa av.
- **Tilt-sensorn** (tekniskt en **ball-tilt switch** — en kula i en hylsa, inte en accelerometer) är **digital**, trots att den ofta kallas ”sensor”. Läses med `digitalRead`, inte `analogRead`. En knapp som gravitationen trycker.

Ni byggde spänningsdelaren med fotocellen, öppnade Serial Monitor tillsammans för första gången i klassrummet, och såg siffrorna ändra sig i realtid när ni höll handen över cellen. Det var Arduinons första riktiga mätinstrument.

Idag fullbordades er **sense-katalog**: en knapp (digital), en tilt-switch (digital), en fotocell (analog), plus Serial-kanalen tillbaka till datorn. Modul 5 är att sätta ihop **sense** och **act** till en riktig reaktiv maskin.

Fotocell (LDR) — en liten rund skiva med gult "ormögon-mönster" på ovansidan. Motståndet sjunker med ökande ljus.



Tilt-sensor — cylindrisk kapsel med en lös metallkula inuti. Lutad → kulan rullar mot två stift → kortslutning.

Repetition av de viktigaste begreppen

ANALOG VS DIGITAL

En **digital** ingång har två tillstånd: HIGH eller LOW. En knapp är digital. En tilt-sensor är också digital — även om vi i dagligt tal kallar den ”sensor”.

En **analog** ingång har en graderad skala. Spänningen på pinnen kan vara 0 V, 2,4 V, 4,7 V eller vad som helst däremellan. `analogRead` omvandlar den spänningen till ett tal 0–1023.

> **VARFÖR JUST 1024 STEG?**
> Arduinons analog-till-digital-omvandlare (ADC) har **10 bitars upplösning**. Det betyder $2^{10} = 1024$ möjliga utvärden, från 0 till 1023. Upplösningen blir $5\ \text{V}\ / 1024 \approx 4,88\ \text{mV}$ per steg — finkornigt nog för de flesta kursens uppgifter.

Bara pinnarna **A0–A5** på Uno har ADC-hårdvara. `analogRead` på en digital pinne returnerar skräpvärden.

SPÄNNINGSDELAREN

Bryggan: ett motstånd → två motstånd

Tänk dig först ett **enda lååångt motstånd**. Det inre materialet — i kursens resistorer en tunn kolfilm runt en keramisk kärna — är jämnt fördelat. Att skicka ström genom det är som att låta vattnet kämpa sig genom en lång trång kanal: trycket sjunker **linjärt** längs vägen. `+5 V` i ena änden, `0 V` i andra. Vid mitten är spänningen exakt `2,5 V`. Vid en fjärdedel: `3,75 V`. Vid tre fjärdedelar: `1,25 V`.

Två motstånd i serie är samma sak — bara delat på en specifik punkt. Skarven mellan dem är en mätpunkt, och det är där `A0` läser av. Värdet du får på `A0` är helt enkelt spänningen vid den brytpunkt du valde.

En **vridpotentiometer** är denna idé i hårdvara: tre ben (yttre två = ändarna av motståndsmaterialet, mittben = wipern), och ratten flyttar wipern längs det inre motståndsspåret. Kopplar du yttre benen till `+5 V` och `GND` och mittbenet till `A0`, får du ett `analogRead`-värde som sveper jämnt 0–1023 när du vrider — exakt det fenomen vi nu utnyttjar med fotocellen, fast där är det ljuset som flyttar mätpunkten.

Vatten-metaforen

Tänk dig ett genomskinligt **vattenrör som höjdskala**: `+5 V` är uppe vid kranen, `GND` är nere vid utloppet. Två motstånd i serie är två rörsegment i rad, och skarven mellan dem är en **tapp** på skalan — det är där `A0` sitter och mäter.

Regeln är enkel när man ser det så här:

- Två lika stora motstånd → skarven hamnar mitt på → `A0 ≈ 2,5 V`.
- Mer motstånd **ovanför** A0 → skarven pressas ner → `A0` **sjunker**.
- Mer motstånd **nedanför** A0 → skarven dras upp → `A0` **stiger**.

Byt nu ut det **övre** motståndet mot en fotocell. Fotocellens motstånd ändras med ljus — högt i mörker, lågt i ljus. Det flyttar skarven upp och ner:

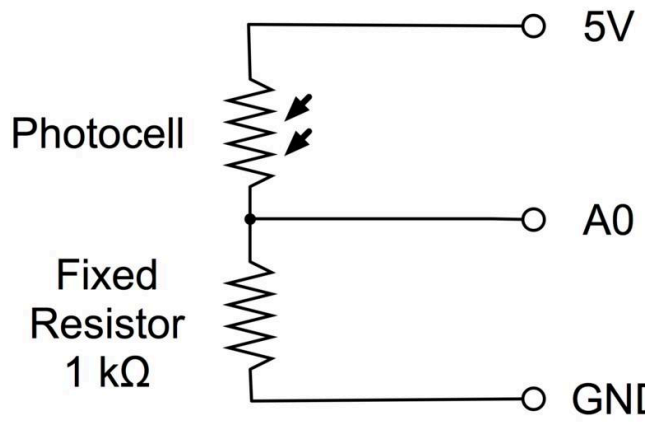
- **Mörker** → högt motstånd ovanför → skarven pressas ner → `A0` sjunker.
- **Ljus** → lågt motstånd ovanför → skarven dras upp → `A0` stiger.

`A0` läses med `analogRead(A0)` och ger ett tal **0–1023** — Arduinons 10-bits linjal över området 0–5 V.

Formellt: om R_{foto} är fotocellens motstånd och R_{under} är den fasta 1 kΩ-resistorn, så är mellanspänningen

$$V_{\text{mid}} = \frac{R_{\text{under}}}{R_{\text{foto}} + R_{\text{under}}} \cdot 5\ \text{V}$$

Fullständig härledning av formeln finns i Bilaga F.



Schematisk ritning av fotocellens spänningsdelare. +5 V → fotocell → A0 → 1 kΩ → GND. A0 mäter mellanpunktens spänning.

> **ORDNINGEN SPELAR ROLL**
> I kursens koppling sitter fotocellen **överst** (mot +5 V) och 1 kΩ **nederst** (mot GND). Detta ger: mörker = LÅG siffra, ljus = HÖG siffra. Byter du plats på dem **inverteras** skalan. Ingen fysik skadas, men logiken i koden måste följa med. Vi kör fotocellen överst genomgående.

TYPISKA A0-VÄRDEN MED 1 KΩ

Beror på ditt rum, ditt exemplar, och mängden ljus som faller in. Men som riktvärden:

Hand över cellen (nästan mörkt)	A0 ≈ **20–100**
Rumsljus, taklampa på	A0 ≈ **150–400**
Lampa nära, ljust arbetsbord	A0 ≈ **500–700**
Ficklampa direkt mot cellen	A0 ≈ **700–900**

Kalibrera själv: öppna Serial Monitor, lek med ljuset, och notera vilka värden ni ser. Den tröskel ni väljer för ”det är mörkt nu” ska ligga någonstans mellan era rumsljus- och mörker-värden.

> **VARFÖR JUST 1 KΩ?**
> Fotocellen går från 50 kΩ (mörker) till 500 Ω (ljust). Med **1 kΩ** som motvikt hamnar mätvärdet mitt i A0:s 0–1023-skala för **rumsljus** — det är där ni jobbar. 10 kΩ skulle pressa det mesta mot toppen av skalan, 220 Ω skulle pressa allt mot botten. 1 kΩ är storleksordningen som ger bäst läsbarhet för ljusnivåerna i ett klassrum.

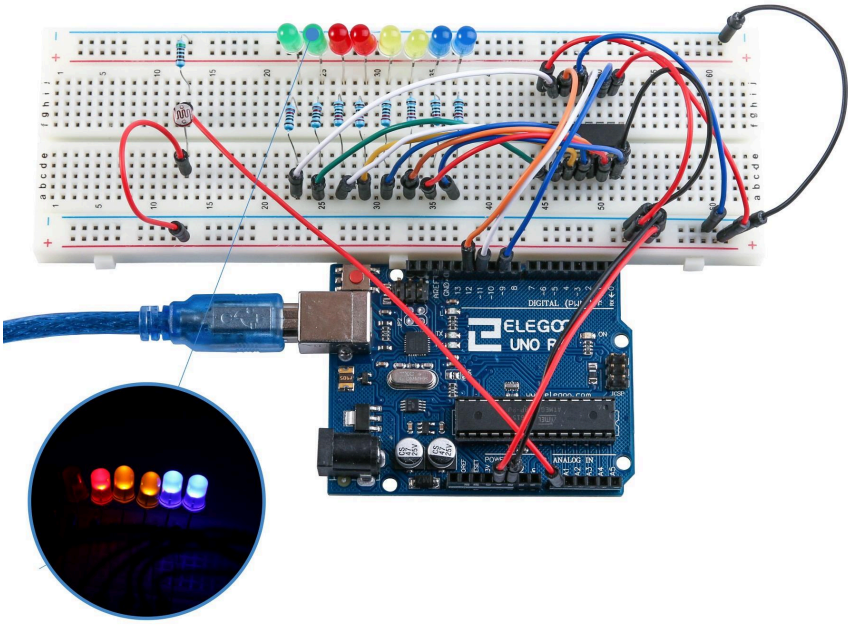
SKALA OM VÄRDEN MED `MAP()`

Ibland vill ni använda A0:s 0–1023 för att styra något som tar 0–255 (PWM) eller 0–100 (procent). Arduino har en inbyggd funktion för just detta — `map()`:

```
int pwmVarde = map(ljus, 0, 1023, 0, 255);
```

Läs det som: ”Ta `ljus` som ligger i intervallet 0–1023, skala om det linjärt till 0–255, och spara i `pwmVarde`.” Fungerar lika bra med inverterade intervall — `map(ljus, 0, 1023, 255, 0)` ger 0 vid 1023 och 255 vid 0 (användbart när mörkare ska ge starkare ljusstyrka).

Signaturen: `map(värde, frånMin, frånMax, tillMin, tillMax)`. Ni kommer se den igen i hemma-övning 2 och senare i hackathonen.



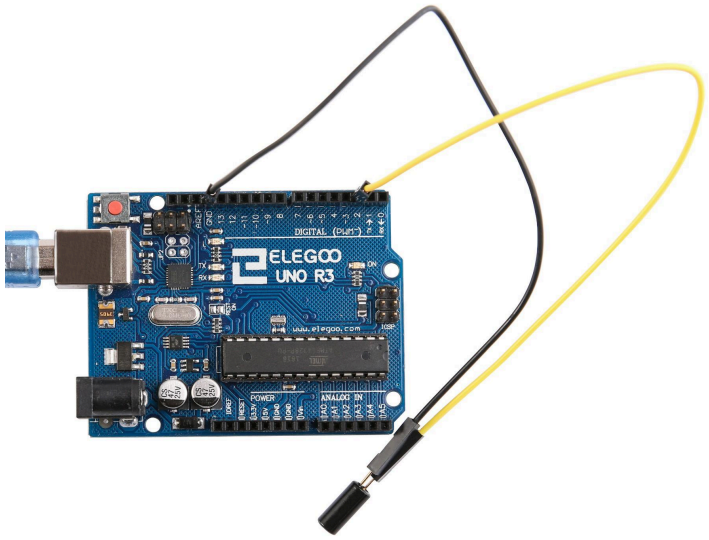
Fotocell-kretsen byggd på breadboard med RGB-LED:er som lyser i bakgrunden. Den här bilden är från en hackathon-variant där fotocellen styr LED:erna.

TILT-SENSORN

Tilt-sensorn är en liten metallcylinder med en lös kula inuti — tekniskt en **ball-tilt switch**, inte en accelerometer. I upprätt läge rör kulan bara ett av benen — kretsen är bruten. Lutas sensorn rullar kulan åt sidan och kortsluter båda benen — kretsen sluts.

Elektriskt är det alltså **identiskt med en knapp**. Kopplas som knappen: ena benet till `D2` (i kursen), andra benet till GND, och pinnen sätts upp som `INPUT_PULLUP`. Läses med `digitalRead`. (Du kan använda en annan digital pinne, men då måste du ändra pin-variabeln i koden.)

Tilten studsar fysiskt — kulan bouncar inuti sin hylsa i en tiondels sekund efter en rörelse. En `delay(50);` efter läsningen räcker som debounce i kursens sammanhang.



Tilt-sensorn kopplad direkt i Arduino-headers med F-M DuPont-kablar — ingen breadboard. Ena benet till D2, andra till GND.

SERIAL MONITOR

Första gången på kursen kan Arduinon kommunicera **tillbaka** till er. Fram till nu har ni varit envägs-kommunicerande: ni skriver kod, laddar upp, kollar om LED:en blinkar. Nu kan Arduinon skriva text till er.

Syntaxen är tre saker:

```
void setup() {
  Serial.begin(9600);     // starta kommunikationen, 9600 bit/s
  // analoga pinnar är INPUT som default —
  // följande är onödigt men inte fel:
  // pinMode(A0, INPUT);
}

void loop() {
  int ljus = analogRead(A0);
  Serial.println(ljus);   // skriv värdet + ny rad
  delay(100);             // inte för snabbt, annars drunknar du
}
```

Öppna Serial Monitor genom att klicka på **förstoringsglaset** uppe till höger i Arduino IDE. En ny ruta dyker upp där siffror rullar förbi. Du kan ändra raden där det står ”9600 baud” till samma värde som du skrev i `Serial.begin()` — de måste matcha.

`Serial.print()` skriver utan radbrytning. `Serial.println()` skriver med radbrytning. För flera värden på samma rad:

```
Serial.print("ljus=");
Serial.print(ljus);
Serial.print(" tilt=");
Serial.println(tilt);
```

SERIAL MONITOR ÄR DIN FELSÖKNINGSVANA

Serial Monitor är den viktigaste felsökningsvanan i hela kursen. När ett program inte gör vad du väntar, **printa** det du tror är fel. Printa villkorsvariabler. Printa mätvärden. Printa vilken gren av en if/else som körs. Arduinon är inte svår att felsöka — den behöver bara skriva ut vad den ser.

Bygg från minnet

Skriv från scratch en sketch som printar både ljus- och tilt-värden i Serial Monitor varje hundratals millisekund. Du behöver:

1. Tre `const int` för pin-nummer: `ldrPin = A0`, `tiltPin = 2`, plus eventuellt `buzzerPin`.
2. `Serial.begin(9600)` och `pinMode(tiltPin, INPUT_PULLUP)` i setup.
3. En loop som läser båda värdena, printar dem på en rad, pausar.

Målet: se siffrorna i realtid och använda dem för att kalibrera tröskeln ”det är mörkt”.

Hemma-övningar

ÖVNING 1 — STÄMNINGSLAMPA

Skriv en sketch som tänder den inbyggda LED:en (pin 13) **endast** när fotocellen läser under en tröskel du själv väljer. När det är mörkt, tänds lampan. När det ljusnar, släcks den. Använd din egen kalibrering från lektionen.

STRUKTUR

```
if (ljus < troskel) {
  digitalWrite(LED_BUILTIN, HIGH);
} else {
  digitalWrite(LED_BUILTIN, LOW);
}
```

Där `troskel` är ett `const int` i toppen av filen. Ändra det tills beteendet känns lagom.

ÖVNING 2 — PROPORTIONAL RGB

Använd fotocellens värde som direktstyrning av RGB-LED:ens ljusstyrka. När det är mörkt → LED:en är full röd. När det är ljust → LED:en är svag röd. Formeln: `analogWrite(ledR, map(ljus, 0, 1023, 255, 0));`

`map(värde, frånMin, frånMax, tillMin, tillMax)` är en inbyggd funktion som skalar om ett tal från ett intervall till ett annat. Här: 0–1023 → 255–0 (inverterad, så att mörker ger starkt ljus).

ÖVNING 3 — TILT-ALARM

Skriv en sketch som bara piper buzzern när tilt-sensorn läser `LOW` (lutad). När den är upprätt, tystnad. Det är ett **bekräftelsetest** — visar att tilt-sensorn fungerar och styr buzzern. Hackathonens tjuvlarm bygger sedan vidare med larm-mönstret från Modul 3 (knappens toggle av `larmPaslaget`) och låter `larmPaslaget && tiltLutad` styra buzzern.

För extra credit: lägg till att buzzern piper i ett mönster istället för kontinuerligt — till exempel 200 ms på, 200 ms av, så länge tilten är lutad.

Vanliga fel och snabblösningar

analogRead ger alltid 0	Pinnen är inte kopplad (flyter, men drar mot noll). Kontrollera att fotocellen har ström från 5 V och att 1 kΩ-resistorn sitter på plats.
analogRead ger alltid 1023	Du har kopplat +5 V direkt till A0 utan resistor-par. Kolla att spänningsdelaren är komplett.
Siffrorna darrar	Normalt — sista biten på ADC:n är alltid lite brusig. Om du vill stabilisera: ta medel av flera avläsningar innan du agerar.
Serial Monitor är tom	Glömt `Serial.begin(9600)` i setup, eller Serial Monitor står på fel baud rate. Matcha samma siffra i båda.
Serial Monitor visar konstig text	Baud rate matchar inte. Båda ska vara 9600 i kursen.
Tilt-sensorn verkar inte reagera	Kontrollera `pinMode(tiltPin, INPUT_PULLUP)`. Utan pullup är det slumpmässigt.
Programmet printar 10 000 gånger per sekund	Saknar `delay(100)` i loopen. Lägg in en paus.

Snabbreferens

`analogRead(pin)`	Läser en analog pinne (A0–A5), returnerar 0–1023.
`Serial.begin(9600)`	Startar kommunikation med datorn, 9600 baud.
`Serial.print(x)`	Skriver utan radbrytning.
`Serial.println(x)`	Skriver med radbrytning.
`map(v, lo1, hi1, lo2, hi2)`	Skalar om ett värde från ett intervall till ett annat.
Fotocell-koppling	5 V → fotocell → A0 → 1 kΩ → GND. Mörker = låg siffra.
Tilt-sensor	D2 + GND, `INPUT_PULLUP`. Lutad = LOW.
Typisk debounce	`delay(50)` efter läsningen räcker.

Inför nästa träff

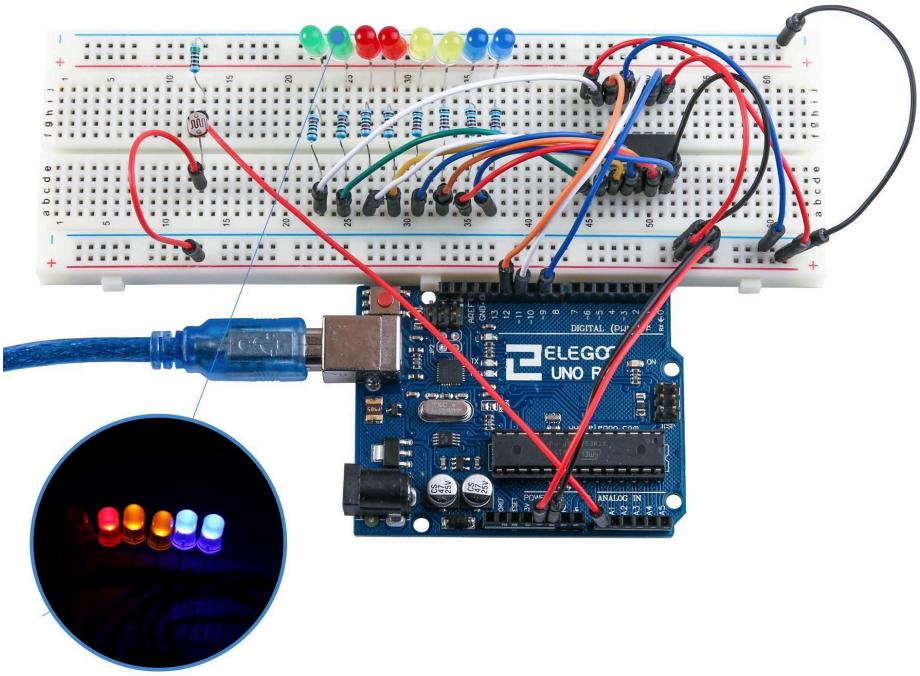
Nästa vecka är **hackathon**. Inga nya begrepp — allt ni behöver kan ni redan. I stället sätter vi ihop alla fyra moduler till ett fungerande tjuvlarm: knappen togglar larmläget, tilt-sensorn triggar tjut + rött blink, fotocellen styr stämningsljus i mörker.

Glöm inte dator eller kitet hemma! Hackathon brukar bli längre än man tror.

MODUL 5

Integration · Hackathon

Alla fyra modulerna, en krets, ett fungerande tjuvlarm.



Den typ av sammansatt krets ni byggde på hackathonen — RGB-LED:er, fotocell, buzzer, sammanvävda på en enda breadboard. Här en variant där fotocellen styr LED:erna direkt.

Vad du gjorde i dag

Sista träffen är en sammansmältning. Ingen ny teori, ingen ny syntax — allt ni behövde kunde ni redan. I stället satte ni ihop bitarna ni byggt separat och gjorde ett fungerande system av dem.

Systemet ni byggde:

- **En knapp** (Modul 3) som togglar ett ”larm-läge”.
- **En tilt-sensor** (Modul 4) som upptäcker om larmet rörs.
- **En fotocell** (Modul 4) som mäter omgivningsljus.
- **En RGB-LED** (Modul 2) som lyser stämningsljus i mörker.
- **En buzzer** (Modul 3) som tjuter vid tilt-event.
- **Ett program** som binder ihop alltihop.

Reglerna var enkla, men programmet som realiserade dem krävde att varje bit från de föregående fyra modulerna sammanvävdes till en enda loop:

1. Knappen togglar `larmPaslaget` med **edge-detection**.
2. Om `larmPaslaget && tilt-sensorn rörs` → **tjut + rött blink**.
3. Om `!larmPaslaget && fotocellen är mörk` → **stämningsljus**.
4. Annars → **tyst och mörkt**.

Det är i princip ett minimalt embedded-system. Ni har byggt ett sådant. Grattis — ni kan nu beskriva grunderna för allt från disktermometrar till enkla RC-bilar i samma ordlista.

Systemets arkitektur

PIN-TILLDELNING

Samma pinnar som i varje lektion — återanvänds direkt:

`knappPin`	D9 — från Modul 3
`tiltPin`	D2 — från Modul 4
`ldrPin`	A0 — från Modul 4
`buzzerPin`	D12 — från Modul 3
`ledR`	D6 — från Modul 2
`ledG`	D5 — från Modul 2
`ledB`	D3 — från Modul 2

Alla dessa pinnar är PWM (`~`-markerade) för R/G/B-kanalerna så att ni kan variera ljusstyrkan. Digital2 och Digital12 är vanliga digital-pinnar — inga krav på PWM där.

SENSE-ACT-LOOPEN

Strukturen för hela programmet är densamma som alla embedded-system någonsin skrivna:

1. **Läs av omvärlden** — fotocell, tilt, knapp.
2. **Uppdatera internt tillstånd** — hantera edge-detection för knappen.
3. **Bestäm vad som ska hända** — if/else baserat på `larmPaslaget` och mätvärdena.
4. **Styr utvärlden** — skriv till RGB-LED och buzzer.
5. **Paus** — kort `delay` för att inte bränna CPU:n.
6. Börja om.

Nedan bygger vi upp den här loopen **i tre steg**, precis som ni själva ska göra under hackathonen. Den fullständiga sammansatta sketch:en ligger i Bilaga D, som ni får efter hackathonen — kursledaren har den om ni kör fast helt och behöver titta på en referens.

Bygg inkrementellt

Den vanligaste nybörjarfällan är att försöka skriva hela programmet direkt. Bygg **ett steg i taget** och bekräfta att varje steg fungerar innan ni går vidare. När något går sönder vet ni exakt vad det är — det är det ni nyligen la till.

STEG 1 — KNAPPEN TOGGLAR LARM-LÄGET

Börja med att bara läsa knappen med edge-detection och printa till Serial Monitor varje gång `larmPaslaget` ändras. Inga LED:ar, ingen buzzer, ingen sensor. Bara: ”fungerar togglet-ringen?”

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

void setup() {
  Serial.begin(9600);
  pinMode(knappPin, INPUT_PULLUP);
  Serial.println("Start. Larm av.");
}

void loop() {
  int knappState = digitalRead(knappPin);
  if (knappState == LOW && lastKnappState == HIGH) {
    larmPaslaget = !larmPaslaget;
    Serial.print("Larm nu ");
    Serial.println(larmPaslaget ? "PÅ" : "AV");
  }
  lastKnappState = knappState;
  delay(10);
}
```

Ladda upp. Tryck. Serial Monitor ska skriva ”Larm nu PÅ” och ”Larm nu AV” omväxlande — en rad per tryck. **Om den flippar flera gånger per tryck har du en bug. Fixa först. Gå inte vidare.**

STEG 2 — LÄGG TILL TILT-SENSOR OCH BUZZER

Utöka: när larmet är på **och** tilten lutas → buzzern tjuter. När larmet är av eller tilten står upprätt → tyst.

```
// Lägg till överst:
const int tiltPin   = 2;
const int buzzerPin = 12;

// I setup(), efter pinMode för knappen:
pinMode(tiltPin,   INPUT_PULLUP);
pinMode(buzzerPin, OUTPUT);

// I loop(), efter edge-detection-blocket men före delay:
bool tiltLutad = (digitalRead(tiltPin) == LOW);
if (larmPaslaget && tiltLutad) {
  digitalWrite(buzzerPin, HIGH);
} else {
  digitalWrite(buzzerPin, LOW);
}
```

Ladda upp. Toggla larmet på med knappen. Luta tilten → det ska tjuta. Återställ larmet till av → tyst även om tilten lutas. Bekräfta båda innan nästa steg.

STEG 3 — LÄGG TILL STÄMNINGSLJUS I MÖRKER

Sista biten: när larmet är **av** och fotocellen läser under en tröskel → RGB-LED:en tänds mjukt. Annars mörk.

```
// Lägg till överst:
const int ldrPin       = A0;
const int ledR = 6, ledG = 5, ledB = 3;
const int morkTroskel  = 300;     // kalibrera själv

// I setup(), efter buzzerPin-raden:
pinMode(ledR, OUTPUT);
pinMode(ledG, OUTPUT);
pinMode(ledB, OUTPUT);

// I loop(), utöka den if/else som styr buzzern till tre grenar:
int ljus = analogRead(ldrPin);
if (larmPaslaget && tiltLutad) {
  digitalWrite(buzzerPin, HIGH);
  analogWrite(ledR, 255); analogWrite(ledG, 0); analogWrite(ledB, 0);
} else if (!larmPaslaget && ljus < morkTroskel) {
  digitalWrite(buzzerPin, LOW);
  analogWrite(ledR, 120); analogWrite(ledG, 60); analogWrite(ledB, 20);
} else {
```

```
  digitalWrite(buzzerPin, LOW);
  analogWrite(ledR, 0); analogWrite(ledG, 0); analogWrite(ledB, 0);
}
```

Ladda upp. Täck fotocellen (larmet av) → LED lyser varmt. Toggla larmet på och luta → tjut + rött. Testa alla kombinationer.

Den färdiga, rensade och kommenterade sammansatta sketch:en med en hjälpfunktion för att sätta färg: **Bilaga D** (delas ut efter hackathonen).

Tips för hackathon-formatet

SERIAL MONITOR ÄR INTE VALFRI

Varje gång du lägger till ett nytt villkor eller en ny variabel, printa den. Printa `larmPaslaget`. Printa `ljus`. Printa `state`. Printa varje gren av varje if/else. Om beteendet är mystiskt — läs vad Serial Monitor säger, inte vad du **tror** står i koden.

DEBUG-PRINT SNABBRECEPT

```
Serial.print("larm=");
Serial.print(larmPaslaget);
Serial.print(" ljus=");
Serial.print(ljus);
Serial.print(" tilt=");
Serial.println(tilt);
```

Kör det på slutet av varje loop (efter ett lämpligt `delay(200)`) så har du en kontinuerlig ström av sanning.

EDGE-DETECTION ÄR ICKE FÖRHANDLINGSBAR

För knappen: **alltid** edge-detection, aldrig direkt `if (digitalRead(knappPin) == LOW)`. Utan edge-detection hoppar `larmPaslaget` fram och tillbaka hundratals gånger per tryckning, och du har ingen aning om vilket värde den slutade på när du släppte.

För tilten: edge-detection är mer valfritt. Ni kan agera direkt på ”tilt är LOW nu”, och det fungerar oftast bra. Men samma problem med studs kan uppstå — lägg `delay(50)` efter läsningen.

ORDNA KODEN I BLOCK

När stegen 1–3 ovan är hopfogade ser loopen typiskt ut så här:

1. **Läs inputs** — `analogRead` och `digitalRead` för alla tre sensorer.
2. **Uppdatera internt tillstånd** — edge-detection för knappen.
3. **Bestäm utfall** — en if/else if/else med tre grenar (larm-trigg, stämningsljus, tyst).

4. **Kort paus** — `delay(10)`.

Den fullständiga, rensade och kommenterade sammansatta sketch:en ligger i **Bilaga D** (som delas ut efter hackathonen) — inklusive en `sattFarg(r, g, b)`-hjälpfunktion som gör if/else-grenarna mycket lättare. Behöver ni en referens under hackathonen, fråga kursledaren.

Tröskeln 300 är bara en gissning. Kalibrera själv med Serial Monitor innan hackathonen börjar — värdet hör hemma som en `const int` i toppen så du enkelt kan ändra det.

Vanliga problem och snabblösningar

Larmet flimrar när knappen hålls	Du saknar edge-detection. Återvänd till ”Reagera på flanken” i Modul 3.
Tilten triggar slumpmässigt	Studs. Lägg `delay(50)` efter `digitalRead(tiltPin)`.
RGB-LED blir aldrig riktigt mörk	Glömt att sätta alla tre kanalerna till 0 i ”tyst och mörkt”-grenen. Alla tre kanaler ska vara `analogWrite(pin, 0)`.
Stämningsljuset lyser alltid	Din tröskel är för hög — `ljus < 300` matchar även rumsljus. Sänk till 150 och testa.
Buzzern vägrar bli tyst	En av grenarna sätter `HIGH` men ingen annan återställer till `LOW`. Säkerställ att **varje** gren av if/else skriver både buzzer och LED.
Kompilerar inte — ”variable not in scope”	Variabler deklarerade **inne** i setup() är bara synliga där. Flytta `lastKnappState` och `larmPaslaget` upp till global nivå (utanför setup och loop).

Bygg vidare

Hackathonen är ungefär 2 timmar med paus. Räkna med 60–80 minuter för att få grundlösningen (stegen 1–3 ovan) att köra stabilt. Resterande tid räcker till ett av förslagen nedan för den som vill djupare. Varje förslag är 15–30 minuters extra kod isolerat.

Allt ni behöver är redan i kittet:

- **Dubbelt pip för larm-på/larm-av**. Ett kort pip när larmet aktiveras, två snabba när det stängs av. Förtydligande till användaren vad som hände.
- **Dimning i stället för hopp**. När stämningsljuset tänds — tona upp det över en sekund i stället för att bara slå på. `for`-loop med stegvis ökande `analogWrite`.
- **Hysteres**. Dela tröskeln i två: en ”tänd vid 300, släck vid 400”. Det hindrar LED:en från att blinka av och på när ljuset ligger precis på tröskeln.

– **Förvarning**. När larmet varit på i 30 sekunder, blinka buzzerN i ett mönster som varning om att du glömde stänga av den.
– **Eget morse-meddelande** via `LED_BUILTIN` varje gång tilten triggar.

Vilken som helst av dessa är 15–30 rader extra kod. Prova.

Snabbreferens

Pin-karta	Knapp D9, Tilt D2, LDR A0, Buzzer D12, R/G/B D6/D5/D3.
Kod-skelett	setup: `Serial.begin` + `pinMode` för alla. loop: läs, edge-detect, if/else, styr.
Grundläggande debug	`Serial.print(variabel)` på allt som är mystiskt.
`!larmPaslaget`	Logisk inversion — togglar `true` / `false`.
Full sketch	Bilaga D — delas ut efter hackathonen.

Efter kursen

Fem veckor sedan visste ni inte vad en GND-pinne var. I dag har ni byggt ett system som läser av omvärlden och reagerar på den. **Det är i princip vad embedded-ingenjörer gör på jobbet** — bara i mindre skala och med billigare komponenter.

Nästa steg — om ni vill fortsätta — är inte ett enda steg utan en skog av dem:

– **TinkerCAD Circuits** (`tinkercad.com/circuits`). Autodesks gratis online-simulator för Arduino och breadboard. Ni bygger kretsen i browsern, skriver samma Arduino-kod som i IDE:n, och trycker play — den simulerar hela kretsen inklusive kod, LED:ar, sensorer, serial monitor, allt. Perfekt när ni vill prova en idé på bussen, när kittet står kvar på kontoret, eller när ni vill testa ett koncept innan ni köper en ny komponent. Fungerar i vilken browser som helst.
– **Arduino-projekt online**. Sidor som Hackster, Instructables, Make: Magazine är fulla av recept för allt från väderstationer till MIDI-instrument. Börja enkelt. Kopiera något. Förstå vad du kopierade.
– **Lokala makerspaces**. Det finns ofta ett i närmsta större stad. Gå en kväll. Andras projekt är smittsamt.
– **Fråga er själva: vad i mitt eget hem skulle jag vilja automatisera?** Ett litet problem i verkligheten är mycket bättre motivation än en abstrakt tutorial. ”Lampan vid hallspegeln tänds när jag kommer hem i mörkret.” ”Brevlådan skickar en notis när posten kommit.” ”Kaffebryggaren pausar två minuter efter att jag stängt den av.” Bygg det lilla. Lär dig det stora på vägen.

– **Gå längre in i elektroniken**. De fyra operatorerna från Bilaga A räcker som programmerarpotential för nästan allt Arduino-relaterat. Begränsningen är snart inte koden — den är era idéer om vad som borde göras.

Om ni kommer tillbaka och visar upp något ni byggt: det är kursens högsta beröm. Detta kompendium är er utgångspunkt, inte er slutpunkt.

BILAGA A

Syntax-grammatik

Datatyper, operatorer, scope, funktioner — det lilla språket under Arduinons huva.

Vad ”kod” egentligen är

En Arduino-sketch är ett program skrivet i **C++**. Det är ett fullfjädrat programmeringsspråk med egna regler för hur orden får kombineras — precis som svenska har grammatik och stavningsregler. Felar du mot dem vägrar kompilatorn att översätta din text till något Arduinon kan köra.

Kursens slides presenterar syntax genom exempel, men ingen formell översikt. Den här bilagan är den formella översikten. Den är inte uttömmande — den täcker **det ni använder på kursen**, inte hela C++.

Variabler och datatyper

DEKLARATION

En variabel är ett namngivet utrymme där du kan lagra ett värde. För att skapa en skriver du dess **datatyp**, dess **namn**, och (valfritt) ett **initialvärde**:

```
int count = 0;
bool larmPaslaget = false;
const int knappPin = 9;
```

- `int` är datatypen — heltal.
- `count` är namnet du hittat på.
- `= 0` är initialvärdet — ditt val.
- `;` avslutar raden. Glömmer du semikolonet klagar kompilatorn.

PRIMITIVA DATATYPER SOM KURSEN ANVÄNDER

`int`	Heltal. −32768 till 32767 på Uno (16-bit). Använd när du räknar något: räknare, mätvärden, pin-nummer.
`long`	Stort heltal. Upp till ±2 miljarder. Använd `unsigned long` för `millis()`-tid eftersom den växer snabbt.
`byte`	Heltal 0–255. Perfekt för PWM-värden, bytes från sensorer, bitfält.
`bool`	Antingen `true` eller `false`. Använd för till/från-tillstånd.
`float`	Decimaltal. 3,14, −0,001 osv. Långsammare än `int`, undvik om du inte måste.
`char`	Ett enstaka tecken, typ `'a'`. Används sällan i denna kurs.

`CONST` — "DET HÄR VÄRDET ÄNDRAS ALDRIG"

Prefixet `const` låser värdet efter det tilldelats. Försöker du skriva över det senare, vägrar kompilatorn.

```
const int ledPin = 13;     // bestämt nu, för evigt
int pressCount = 0;        // kan ändras
```

Använd `const int` för pin-nummer och andra fasta värden. Det gör koden läsbar och hindrar dig från att råka ändra värdet av misstag.

`CONST INT` VS `#DEFINE` — SAMMA RESULTAT, OLIKA VÄGAR

I Arduino-exempel ser ni båda skrivsätten:

```
const int ledPin = 13;     // modern C++
#define LED_BUILTIN 13     // äldre C-makro
```

De fungerar ungefär likadant — båda ger namnet `ledPin` eller `LED_BUILTIN` värdet 13 — men under huven är de olika saker.

`const int` deklarerar en **riktig variabel** som lagras i minnet och som har en datatyp (`int`). Kompilatorn vet att det är ett heltal och kan stoppa dig från att använda det fel, till exempel skriva `ledPin = "text"`.

`#define` är ett **textmakro**: preprocessorn går igenom filen innan kompilatorn ser den, och ersätter varje förekomst av `LED_BUILTIN` med `13`. Ingen datatyp, ingen typkontroll. Det är ett äldre C-arv från 70-talet.

> **VARFÖR BÅDA?**
> Arduino-biblioteket (`Arduino.h`) använder `#define` av historiska skäl — makron fanns i C långt innan `const` blev standardiserad och typstark. För **er egen kod** på kursen använder vi `const int`, eftersom det är tydligare, typstarkt, och uppmärksammar dig

> direkt om du råkar göra något galet. Att Arduino-biblioteket självt använder `#define` är helt OK — det är etablerat och välbeprövat.

Operatorer

ARITMETISKA

```
a + b     // addition
a - b     // subtraktion
a * b     // multiplikation
a / b     // division (heltal!)
a % b     // modulo – resten efter division
a++       // öka a med 1
a--       // minska a med 1
a += 5    // kortare form av: a = a + 5
```

> **HELTALSDIVISION TRICKAR MÅNGA**
> `int a = 5 / 2;` ger `a = 2`, inte `2.5`. Heltal avrundas nedåt. För decimalsvar måste du använda `float`: `float a = 5.0 / 2.0;` ger `2.5`.

JÄMFÖRELSE

```
a == b    // lika
a != b    // olika
a < b     // mindre än
a > b     // större än
a <= b    // mindre eller lika
a >= b    // större eller lika
```

Det vanligaste nybörjarfelet: att skriva `=` (tilldelning) istället för `==` (jämförelse).

```
if (x = 5) { ... }   // FEL: tilldelar 5 till x, alltid sant
if (x == 5) { ... }  // RÄTT: jämför x med 5
```

Kompilatorn varnar ibland, men inte alltid. Läs raden högt: ”om x är lika med 5” = `==`. ”Sätt x till 5” = `=`.

LOGISKA

```
a && b    // OCH – båda måste vara sanna
a || b    // ELLER – minst en måste vara sann
!a        // INTE – vänder på sanningsvärdet
```

Används i `if`-villkor och tillstånd:

```
if (larmPaslaget && lutad) {
  // både larm på OCH tilt lutad
}

if (!larmPaslaget || ljus > 500) {
  // larm AV, ELLER det är ljust
}
```

`!` framför en variabel är **logisk inversion** — det ”motsatta”. `!true` = `false`, `!false` = `true`. Används ofta för att toggla: `larmPaslaget = !larmPaslaget;`.

Villkorssatser

`IF` / `ELSE`

```
if (villkor) {
  // kör om villkor är sant
} else {
  // annars
}
```

Flera alternativ kedjas med `else if`:

```
if (ljus < 200) {
  // kolsvart
} else if (ljus < 500) {
  // halvmörkt
} else {
  // ljust
}
```

Körs i ordning — första matchande gren vinner, resten hoppas över.

KORT IF-SYNTAX (TERNÄR)

```
int värde = (villkor) ? omSant : omFalskt;

// exempel:
digitalWrite(LED_BUILTIN, larmPaslaget ? HIGH : LOW);
```

Läses: ”om larmPaslaget, sätt HIGH, annars sätt LOW”. En rad istället för fyra.

Loopar

`FOR` — UPPREPA ETT BESTÄMT ANTAL GÅNGER

```
for (int i = 0; i < 10; i++) {
  // kör tio gånger, med i = 0, 1, 2, ..., 9
}
```

Tre delar, åtskilda med semikolon:

1. **Initiering**: `int i = 0` — körs en gång innan loopen.
2. **Villkor**: `i < 10` — testas före varje varv. När det blir falskt, avslutas loopen.
3. **Uppdatering**: `i++` — körs efter varje varv.

Användbart för till exempel att tona en LED från släckt till tänd:

```
for (int v = 0; v <= 255; v++) {
  analogWrite(ledR, v);
  delay(10);
}
```

`WHILE` — UPPREPA TILLS NÅGOT ÄNDRAS

```
while (villkor) {
  // körs så länge villkor är sant
}
```

Kursens `void loop()` är i själva verket ett inbyggt `while`-loop som aldrig avslutas. Du skriver det inte själv, Arduino-ramverket gör det åt dig.

Funktioner

En **funktion** är ett namngivet stycke kod du kan anropa. `setup` och `loop` är funktioner. Så är `digitalWrite` och `delay`. Du kan skriva egna också:

```
void blink(int tid) {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(tid);
  digitalWrite(LED_BUILTIN, LOW);
  delay(tid);
}
```

Delar:

- `void` är **returtypen** — vad funktionen ger tillbaka. `void` betyder ”inget”.
- `blink` är namnet.
- `(int tid)` är **parameterlistan** — värden som funktionen tar emot. Här ett heltal som heter `tid`.

– `{ ... }` är funktionens kropp — koden som körs.

Anropa den senare med `blink(200);` eller `blink(1000);`.

En funktion som **ger tillbaka** ett värde har annan returtyp:

```
int dubbelt(int x) {
  return x * 2;
}

// Användning:
int y = dubbelt(7);   // y blir 14
```

Scope — var en variabel syns

Variabler deklarerade **inne i** en funktion är **lokala** — de syns bara där.

```
void setup() {
  int temp = 5;
  // temp finns bara här inne
}

void loop() {
  // temp finns INTE här – kompilerings-fel om du försöker använda det
}
```

Variabler deklarerade **utanför** alla funktioner — längst upp i filen — är **globala** och syns överallt.

```
int tryckCount = 0;    // global – syns i både setup och loop

void setup() {
  tryckCount = 5;   // OK
}

void loop() {
  tryckCount++;     // OK
}
```

Regeln för kursen: deklarera `const int` för pin-nummer och andra konstanter globalt, överst i filen. Deklarera tillståndsvariabler (`larmPaslaget`, `lastState`) globalt så att de överlever mellan `loop`-varv. Tillfälliga hjälpvärden (räknare, nuvarande mätvärde) deklareras lokalt där de behövs.

VARFÖR 'LOKAL'?
Lokala variabler försvinner när funktionen återvänder — minnet frigörs för nya variabler. Globala lever hela programmets livstid. Det är därför

> `int state = digitalRead(knappPin);` kan stå inne i loop — den skapas på nytt varje varv, används, och kastas bort. `lastState` däremot måste vara global så att den överlever mellan varven.

Kommentarer

```
// En en-rads-kommentar. Allt efter // på raden ignoreras.

/*
  En flera-rads-kommentar.
  Allt mellan tecknen ignoreras.
*/
```

Använd dem för att förklara **varför** något görs, inte **vad** koden gör — det ser man ändå. Bra kommentar: `// edge-detection – annars flippar larmet 10000 ggr/sek`. Dålig: `// sätt pin 13 till HIGH`.

Arduino-specifika konstanter

Några värden Arduino-biblioteket definierar åt dig och som ni redan sett:

`HIGH`, `LOW`	De två logiska nivåerna. Används med `digitalWrite` och `digitalRead`.
`INPUT`, `OUTPUT`, `INPUT_PULLUP`	Används med `pinMode`.
`LED_BUILTIN`	Vanligtvis pin 13 — där den inbyggda LED:en sitter.
`A0` – `A5`	Analoga pinnar, används med `analogRead`.
`true`, `false`	Booleska värden.

Dessa är `#define`-s i `Arduino.h` — samma historiska arv som diskuterades tidigare. Du kan inte ändra dem, bara använda dem.

Användbara Arduino-funktioner

Utöver de ni redan använt rakt av finns några hjälpfunktioner i Arduino-biblioteket som sparar mental ansträngning:

`MAP()` — SKALA OM ETT VÄRDE

```
map(värde, frånMin, frånMax, tillMin, tillMax)
```

Skalar om ett tal linjärt från ett intervall till ett annat. Exempel — gör om fotocellens 0–1023 till PWM:s 0–255:

```
int pwmVarde = map(ljus, 0, 1023, 0, 255);
```

Eller **inverterat** — mörkt ger starkt ljus:

```
int pwmVarde = map(ljus, 0, 1023, 255, 0);
```

Mycket användbar i hackathonen när sensor-värden ska styra PWM-utgångar.

`CONSTRAIN()` — KLIPP TILL ETT INTERVALL

```
int sakert = constrain(värde, minsta, största);
```

Om `värde` är under `minsta` → returnera `minsta`. Om över `största` → returnera `största`. Annars oförändrat. Användbart tillsammans med `map()` för att vara säker på att resultatet aldrig går utanför ett önskat intervall.

`MILLIS()` — TID SEDAN UPPSTART

```
unsigned long nu = millis();
```

Returnerar antal millisekunder sedan Arduinon fick ström. Till skillnad från `delay()` blockerar `millis()` inte loopen — den är grunden för ”non-blocking” kod som kan göra flera saker samtidigt. Används sällan i kursen men är nästa steg för den som vill djupare.

Vanliga kompilatorsfel

`expected ';' before...`	Glömt semikolon på föregående rad. IDE:n pekar på **nästa** rad — felet är raden innan.
`'foo' was not declared in this scope`	Stavfel, eller variabel används utanför sitt scope. Kontrollera att variabeln är global eller lokal där den används.
`expected '\}' at end of input`	Glömt stänga ett block. Räkna måsvingarna — varje `\{` måste ha en `\}`.
`left operand of ',' has no effect`	Du har skrivit `,` istället för `;` mellan två uttryck.

`invalid operands of types 'const char*' and 'int'`	Du har försökt göra matematik på en sträng. Kontrollera vad du skriver in i operatorn.

En uttömmande genomgång av felmeddelanden finns i Bilaga B.

BILAGA B

Felmeddelanden

Hur du läser Arduino-IDE:ns gnäll, och vad det faktiskt betyder.

Varför meddelandena ser kryptiska ut

Arduino-IDE:n använder en riktig C++-kompilator under huven (`avr-g++`). Den är ett verktyg för proffs som skrivit miljoner rader kod och accepterat att felmeddelanden ibland ser akademiska ut. De är långa, tekniska, och verkar peka på fel plats — men det finns logik bakom dem.

Grundregeln: felmeddelanden pekar på **den rad där kompilatorn insåg att något var fel**, inte nödvändigtvis **den rad där du gjorde misstaget**. Om du glömmer ett semikolon på rad 10 kommer kompilatorn klaga på rad 11.

Andra grundregeln: läs alltid det **första** felmeddelandet, inte det sista. Senare fel är ofta följdfel från det första.

Kompilatorsfel du kommer stöta på

`EXPECTED ';' BEFORE ...`
Det vanligaste felet. Du har glömt ett semikolon.

```
digitalWrite(LED_BUILTIN, HIGH)    // saknar ;
delay(1000);                       // kompilatorn klagar HÄR
```

Kompilatorn läste rad 1, förväntade sig `;` men hittade `delay` istället. Felet pekas ut på rad 2, men fixa rad 1.

`'VARIABEL' WAS NOT DECLARED IN THIS SCOPE`
En variabel används utan att ha deklarerats — ofta stavfel, eller variabeln är lokal och du försöker nå den någon annanstans.

```
void setup() {
  int temp = 5;
}

void loop() {
  Serial.println(temp);   // FEL: temp finns bara i setup()
}
```

Fixa genom att flytta `int temp = 5;` till global nivå (överst i filen), eller deklarera en ny `temp` i loop.

Eller — oftast — stavfelet. Två varianter av samma fel:

```
digitalWrit(13, HIGH);    // FEL: saknar 'e' på slutet
digitalwrite(13, HIGH);   // FEL: fel skiftläge på 'w'
```

Rätt är `digitalWrite` — med **stort W och e på slutet**. Arduino är **skiftlägeskänsligt**: `digitalwrite` och `DigitalWrite` är olika ord i kompilatorns ögon. Det vanligaste felet är att tappa `e`:t; det näst vanligaste är att glömma skiftläget.

`expected '}' at end of input`

Du har glömt att stänga ett block någonstans i filen. Räkna alla `\{` och se till att det finns lika många `\}`.

> **HITTA OBALANSERADE KLAMMER**
> Arduino-IDE:n markerar matchande klammer när du klickar på en. Klicka på ett `\{` — den motsvarande `\}` ska markeras automatiskt. Om det inte finns en, har du hittat problemet.

`'LED_BUILTIN' was not declared in this scope` (ovanligt)

Brukar bero på att `#include <Arduino.h>` saknas — men det inkluderas automatiskt av Arduino IDE. Ser du det här felet har du sannolikt skapat en fil manuellt utanför en vanlig `.ino`-sketch. Starta en ny sketch från IDE:n istället.

`void value not ignored as it ought to be`

En funktion som inte returnerar något (`void`) har använts som om den gav tillbaka ett värde:

```
int x = delay(100);    // FEL: delay returnerar void
```

Rätt är att bara anropa den:

```
delay(100);
```

`ASSIGNMENT OF READ-ONLY VARIABLE`
Du har försökt ändra värdet på en `const` variabel:

```
const int ledPin = 13;
ledPin = 12;            // FEL: const går inte att ändra
```

Antingen: ta bort `const` om värdet **ska** kunna ändras. Eller: hitta den riktiga variabeln du menade att ändra.

`INVALID OPERANDS OF TYPES ... TO BINARY 'OPERATOR'`
Du har blandat inkompatibla typer i ett uttryck:

```
int x = "hej" + 5;      // FEL: sträng + int
```

Arduino är strikt med typer. Du kan inte addera ett tal till en textsträng rakt av. Fixa genom att tänka igenom vad du faktiskt vill göra — oftast menade du någonting helt annat.

`TOO MANY ARGUMENTS TO FUNCTION`
Du har skickat in fler argument än funktionen förväntar:

```
digitalWrite(LED_BUILTIN, HIGH, 100);   // FEL: digitalWrite tar bara 2
argument
```

Kolla funktionens signatur. Oftast är det `delay` du egentligen ville ha på slutet:

```
digitalWrite(LED_BUILTIN, HIGH);
delay(100);
```

Upload-fel (inte kompileringsfel)

Här kör kompilatorn klart, men programmet kan inte överföras till Arduinon.

`AVRDUDE: STK500_GETSYNC(): NOT IN SYNC`
IDE:n kan inte prata med Arduinon. Fyra vanliga orsaker:

1. **Fel port i Tools → Port**. Välj den port som heter något med ”Arduino” eller ”usbserial”.
2. **Fel board i Tools → Board**. Välj ”Arduino Uno”.
3. **USB-kabeln är en laddkabel, inte en datakabel**. Arduino Uno R3 använder en USB Type B-kontakt (den kvadratiska, ”printer-style”). Kontrollera att kabeln faktiskt överför data — många billiga kablar är bara ström.
4. **Arduinon är inte ansluten till datorn**. Låter självklart, men händer.

`SERIAL PORT 'COMX' NOT FOUND`
Porten försvann. Koppla ur och i Arduinon, vänta några sekunder, välj porten igen i Tools → Port.

`PORT BUSY` ELLER `PORT IN USE`

Serial Monitor är öppet och blockerar porten. Stäng Serial Monitor innan du uppladdar. IDE:n öppnar den automatiskt igen efter uppladdningen.

Runtime-beteenden (inte fel — men mystiska)

Ibland kompilerar programmet, laddas upp utan problem, men beter sig inte som du väntar. Här är några klassiker:

"PROGRAMMET GÖR INGENTING"

Vanligaste orsaker:

1. Du använder fel pinne. Kontrollera att kabeln faktiskt sitter i pin 13 och inte pin 12.
2. Komponenten är inte ansluten. Lös kabel? Har LED:en fallit av breadboarden?
3. Polaritet fel. LED:en vänd åt fel håll? Resistorn saknas?
4. `pinMode` saknas. Alla utgångar måste sättas som `OUTPUT` i `setup`, alla ingångar som `INPUT` eller `INPUT_PULLUP`.

"PROGRAMMET GÖR NÅGOT MEN INTE DET JAG VILL"

Lägg in `Serial.println()` överallt där du misstänker att det går fel. Printa villkorsvariabler, mätvärden, vilken gren av if/else som körs. Arduinon är inte mystisk — den är bara tyst. Gör den pratig.

```
Serial.print("larmPaslaget=");
Serial.print(larmPaslaget);
Serial.print(" ljus=");
Serial.println(ljus);
```

"BUZZERN TJUTER KONSTANT"

En av grenarna i din if/else skriver `HIGH` men ingen återställer till `LOW`. Se till att **varje** gren skriver både `HIGH` och `LOW` på buzzern.

"KNAPPEN GER FLERA TRYCK ÅT GÅNGEN"

Knappens metallblad studsar fysiskt några millisekunder. `delay(10)` eller `delay(50)` i slutet av loopen är enklaste formen av debounce och **räcker** för att släta över studsen — men bara om du också har edge-detection på knappen. Utan flank-mönstret (`state == LOW && lastState == HIGH`) toggas larmPaslaget hundratals gånger per tryck, vilket inget delay i världen räddar. Modul 3-sektionen "Reagera på flanken" är icke förhandlingsbar för larmet.

"SERIAL MONITOR ÄR TOM"

Tre saker att kontrollera:

1. `Serial.begin(9600);` finns i setup?
2. Baud rate i Serial Monitor står på 9600 (matchar siffran i `Serial.begin`)?
3. Skriver du faktiskt till Serial någonstans (`Serial.print` eller `Serial.println`)?

"SERIAL MONITOR VISAR KONSTIG TEXT"

Baud rate matchar inte. Kontrollera att både `Serial.begin()` och Serial Monitor står på samma siffra. 9600 är kursens standard.

"LED-STYRKAN ÄNDRAS INTE MED ANALOGWRITE"

Du använder en pinne som inte stödjer PWM. Kolla att det finns `~` framför pin-numret på Arduino-kortet. Giltiga PWM-pinnar på Uno: **3, 5, 6, 9, 10, 11**.

Felsöknings-checklista

När något inte fungerar, gå igenom i ordning:

1. **Läs felmeddelandet högt**. Ofta räcker det.
2. **Kontrollera första felmeddelandet**, inte det sista.
3. **Kolla pin-numren**. Stämmer de med vad du kopplat?
4. **Kolla polariteten**. LED, RGB-LED, buzzer — alla har riktning.
5. **Kolla att kabeln sitter i rätt hål**. ”Samma rad på breadboarden” är en favoritbugg.
6. **Printa allt**. Lägg `Serial.println` i varje gren av varje if/else och läs vad som händer.
7. **Ladda om Arduino IDE**. I sällsynta fall får IDE:n hicka och måste startas om.
8. **Fråga någon**. Inte kursledaren — grannen. Två par ögon hittar fel snabbare än ett.

> #### FELSÖKNINGSFILOSOFI
>
> När något inte fungerar är det frestande att **gissa**. Det är näst intill alltid fel strategi. **Ta reda på** — printa värden, kolla polariteten, läs felmeddelandet. Varje gång du fixar en bugg genom gissning istället för att verifiera, har du bara haft tur. Verifiera, vinn.

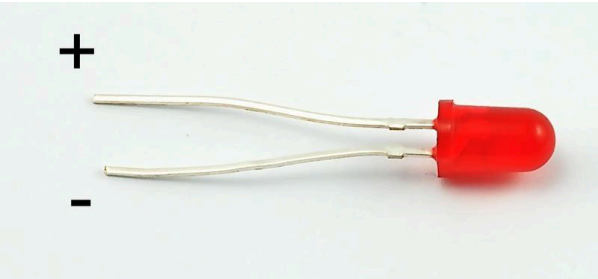
BILAGA C

Komponentreferens

Snabbreferens för varenda bit ni kopplat i kursen.

Denna bilaga är en pek-och-slå-upp-referens. Varje komponent får en kort beskrivning, pinout/polaritet om det spelar roll, vanliga fel, och hur den ska kopplas in mot Arduinon.

LED (standard, genomskinlig)



Lysdiod med markerad anod (+, långt ben) och katod (–, kort ben med platt kant).

En lysdiod. Skickar man ström den rätta vägen lyser den. Fel väg: ingenting. För mycket ström: den brinner upp permanent.

Polaritet	Långt ben = anod (+). Kort ben med platt kant = katod (–).
Max ström	Ca 20 mA. Över det → bränns upp.
Framspänningsfall	Röd ≈ 1,8–2 V. Grön/gul ≈ 2–2,2 V. Blå/vit ≈ 3–3,4 V. Se Bilaga F.
Seriemotstånd	Alltid. 220 Ω är ett säkert allround-val för 5 V.
Arduino-koppling	Pin 13 → anod. Katod → resistor → GND.

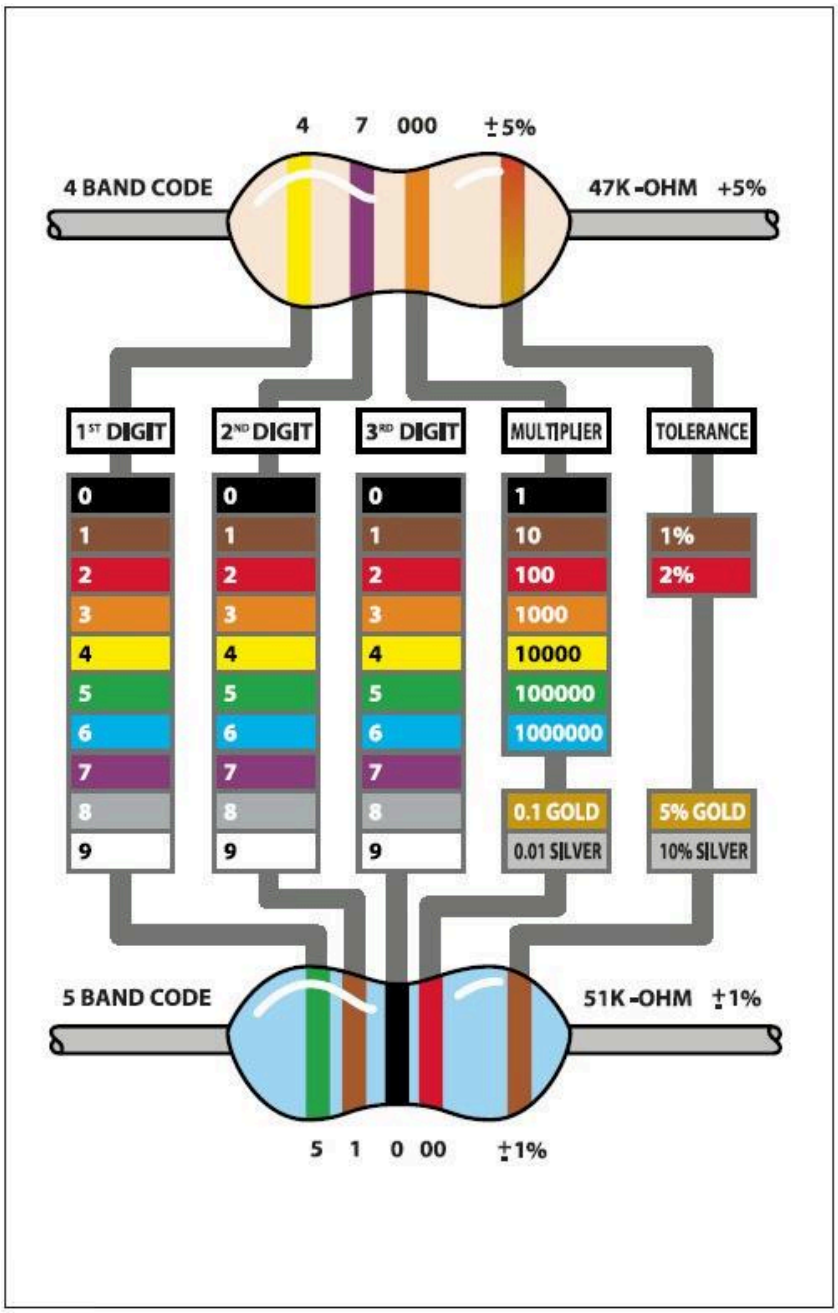
Vanliga fel: LED kopplad baklänges (lyser inte, men går inte sönder), resistor saknas (lyser en kort stund, sedan bränns upp), benet ligger i fel hål på breadboard.

Resistor

Resistor med färgband. Läs från den ände där banden ligger tätast.

Passiv komponent som ”bromsar” ström. Inget att lysa, inget att blinka — men den är ofta det som skiljer en fungerande krets från en brunnen komponent.

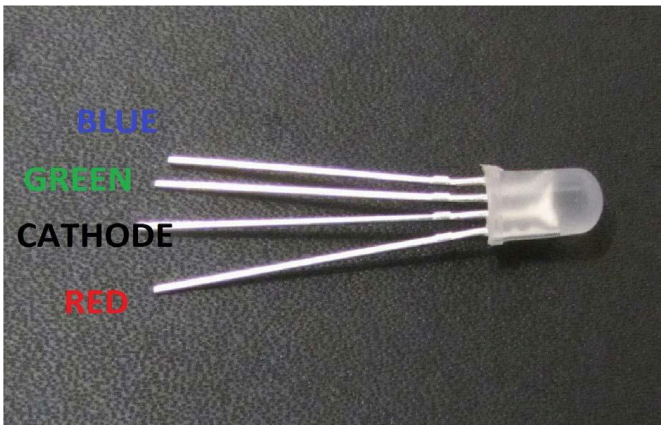
Polaritet	Ingen. Kopplas hur som helst.
Värden i kittet	220 Ω, 1 kΩ, 10 kΩ (och några varianter däremellan)
Avläsning	Färgband. Räkna först **hur många band** din resistor har — kittet innehåller både 4-bands (normal precision) och 5-bands (1 % precision).
4-band	första två siffror · multiplikator · tolerans. 220 Ω = röd-röd-**brun**-guld.
5-band	tre siffror · multiplikator · tolerans. 220 Ω = röd-röd-svart-**svart**-brun.
1 kΩ (4-band)	brun · svart · röd · guld
1 kΩ (5-band)	brun · svart · svart · brun · brun
10 kΩ (4-band)	brun · svart · orange · guld
10 kΩ (5-band)	brun · svart · svart · röd · brun
Osäker?	Mät med multimeter i kontinuitet/Ω-läge. 220 Ω ger exakt ”220” på skärmen. Alltid säkrast när du tvivlar.



Färgkodstabell. För 4-band: digit-digit-multiplikator-tolerans. För 5-band (1 %-precision): digit-digit-digit-multiplikator-tolerans. Räkna banden innan du börjar dekoda.

Tabellen för färgkoder sitter också inuti locket på kittet. Du är välkommen att använda den eller mäta med multimeter. Med tiden lär man sig de vanligaste värdena utan att tänka.

RGB-LED (common cathode)



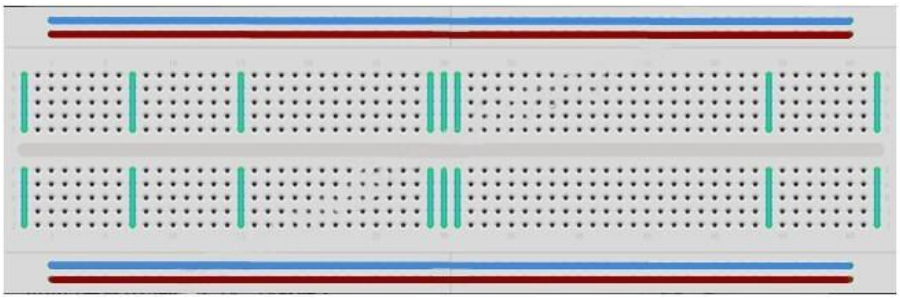
RGB-LED med de fyra benen markerade. Från platta sidan: röd, katod (längst), grön, blå.

Fyra ben, tre kanaler (röd, grön, blå) som delar en gemensam katod. Kittets variant är **common cathode** — det längsta benet går till GND.

Pin-ordning från platta sidan	röd · katod · grön · blå
Katoden	Andra benet från platta sidan. Längst av de fyra. Till GND.
De tre andra	Via varsin 220 Ω-resistor till PWM-pinne.
Kursens pinnar	R → D6, G → D5, B → D3.
Varför PWM?	För att kunna variera ljusstyrkan per kanal (0–255).

Vanligt fel: kopplad som common anode (med gemensam + istället för GND). Då fungerar inget. Om du bara får en färg lysa: ett av benen är i fel hål eller saknar resistor.

Breadboard



Breadboard ovanifrån med interna förbindelser markerade: röda/blåa power rails längs sidorna, gröna kolumn-noder i mitten, gapet som bryter förbindelsen.

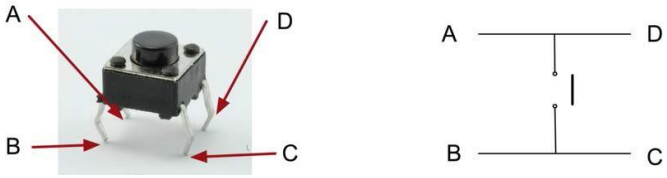
Plastbit med dolda metallklämmor som låter dig koppla ihop komponenter utan att löda.

Horisontella rader i mitten	Fem hål i rad är en nod. Raden bredvid är en egen nod.
Gapet i mitten	Bryter förbindelsen mellan övre och nedre halvan.

Power rails (+/–) längs sidorna	Hela raden är en enda lång nod. Används för att dela ut 5 V och GND.
Hål diameter	Passar genomgångshåls-komponenter och DuPont-kablar (0,6 mm).

Vanligaste nybörjarfelet: ett komponentben och en kabel ligger i olika rader. Kolla först att de **faktiskt sitter på samma rad**.

Knapp (tactile switch)

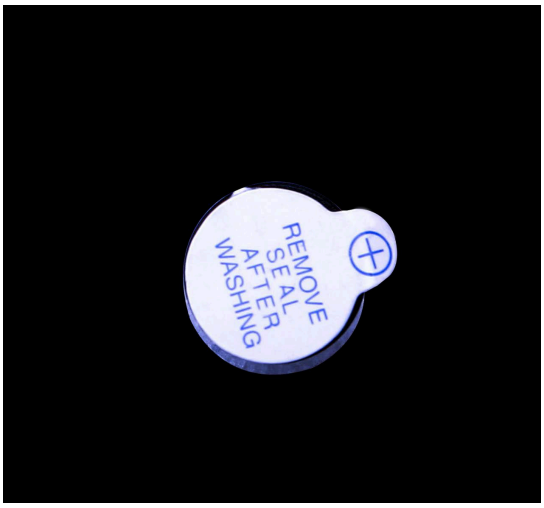


Tactile switch — fyra ben, men internt bara två noder (A–D hopkopplade, B–C hopkopplade). Tryckning sluter de två noderna.

En liten mekanisk brytare med fyra ben. När du trycker ner den kortsluts två par av benen.

Pinnar	Fyra ben, men elektriskt bara två noder: A–D är internt ihopkopplade, B–C likaså. Tryckning kortsluter båda paren.
Koppling i kurs	Ena benet → D9, andra benet → GND. Ingen extern resistor.
Läs med	`digitalRead(pin)` efter `pinMode(pin, INPUT_PULLUP)`.
Tryckt = LOW	På grund av inverterad logik med INPUT_PULLUP.
Studs	Fysisk studs några millisekunder. Lägg `delay(10)` eller `delay(50)` för enkel debounce.

Active buzzer



*Active buzzer — svart cylinder med inbyggd oscillator. Klisterlappen på ovansidan **stannar kvar**.*

En liten svart cylinder som ger ifrån sig ljud när du skickar ström genom den. **Active** betyder att den har en inbyggd oscillator — `digitalWrite(pin, HIGH)` ger direkt pip, ingen `tone()` behövs.

Polaritet	Långa benet = plus (markerat med `+` på ovansidan). Kort ben = minus.
Koppling i kurs	`+`-ben (höger, markerat) → `D12`, `-`-ben → `GND`-skenan på breadboarden tillsammans med knappen.
Resistor	Behövs inte. Buzzern är självreglerande.
Klisterlappen	Stannar på. ”Remove after washing” är en dämpare från tillverkningsprocessen.
Frekvens	Fast (2 kHz). Inte ställbar.

Varning: passive buzzer (blå, låg) finns också i kittet och ser liknande ut. Den kräver `tone(pin, frequency)` för att låta. I kursen använder vi active buzzer — svart, hög, med klisterlapp.

Fotocell (LDR)

Fotocell (LDR) — rund skiva med karakteristiskt gult snake-eye-mönster på ovansidan.

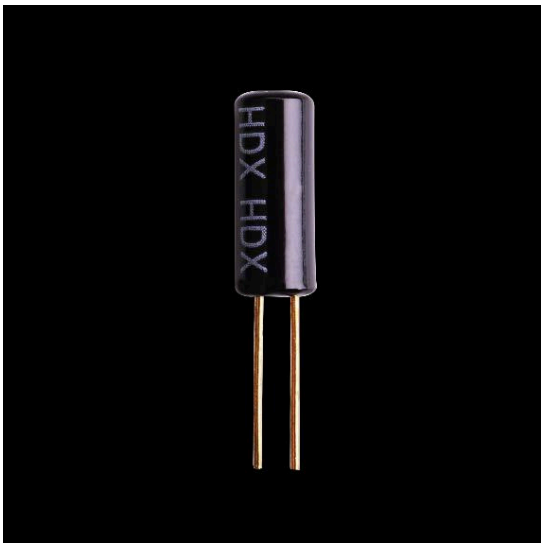
”Light-dependent resistor”. En resistor vars värde ändras beroende på ljuset som träffar den.

Polaritet	Ingen. Kopplas hur som helst.
Motståndsområde	Ca 50 kΩ i mörker, ca 500 Ω i starkt ljus.
Måste ha partner	Ingår i en spänningsdelare — behöver en fast resistor (1 kΩ) för att bilda mellanpunkten.
Koppling i kurs	5 V → fotocell → A0 → 1 kΩ → GND.

Läses med	`analogRead(A0)` — ger 0–1023.

Mätvärden (ungefärliga, med 1 kΩ mot GND): mörker ≈ 20–100, rumsljus ≈ 150–400, lampa nära ≈ 500–700, ficklampa direkt ≈ 700–900. Kalibrera alltid själv.

Tilt-sensor



Tilt ball switch — cylindrisk kapsel med lös metallkula inuti.

En liten cylinder med en lös metallkula inuti. I upprätt läge rör kulan bara ett ben — kretsen är bruten. Lutad: kulan rullar till det andra benet och kortsluter dem.

Polaritet	Ingen för själva funktionen. Men kopplas som en knapp: ena benet till pinne, andra till GND.
Digital trots att det är en sensor	Läses med `digitalRead`, inte `analogRead`. Bara två lägen.
Koppling i kurs	Ena benet → D2, andra → GND. Rakt i Arduino-headers.
Pinmode	`pinMode(tiltPin, INPUT_PULLUP)`.
Lutad = LOW	Inverterad logik från INPUT_PULLUP.
Studs	Kulan bouncar. `delay(50)` efter läsningen räcker som debounce.

Vridpotentiometer (10 kΩ) — demonstrationskomponent

Kittet innehåller en **blå vridpotentiometer** på 10 kΩ. Ni kopplar den inte själva i kursen — men i Modul 4 visar läraren den som **fysisk modell av en spänningsdelare**, för att göra det konkret vad fotocellen sedan gör automatiskt.

Vad det är	Ett ”långt motstånd” där en glidande arm (wiper) kan ställas var som helst längs det inre motståndsmaterialet.
Pinnar	Tre ben. Två yttre = ändarna av motståndsspåret. Mittben = wipern.
Demo-koppling	Yttre ben → +5 V och GND. Mittben → A0. Vrid → A0 sveper 0–1023 jämnt.
Maxvärde	10 kΩ totalt mellan de två yttre benen. Wipern delar upp denna resistans i två andelar.
Läses med	`analogRead(A0)`. Ger samma 0–1023 som fotocell-kopplingen.

Varför den ligger utanför hands-on-laben: en pot löser samma uppgift som en knapp i många nybörjarprojekt (välja ett värde) men introducerar mekanik som tar fokus från koden. Den är dock perfekt som **brygga** till spänningsdelar-konceptet — wipern är mätpunkten, ni kan se den flyttas med handen.

Vill ni leka mer: prova att ersätta fotocellen i Modul 4 med pot:en och styra LED-ljusstyrkan med vridning istället för ljus. Allt i Modul 4-koden fungerar oförändrat.

Arduino Uno R3

Själva styrkretsen. En mikrocontroller i ett ”development board”-format med inbyggd USB-kontroller, spänningsreglering och stift-kontakter.

Processor	ATmega328P (16 MHz, 8-bit)
Minne	32 KB flash (din kod), 2 KB SRAM (variabler), 1 KB EEPROM (persistent)
Digital-pinnar	D0–D13, varav **D3, D5, D6, D9, D10, D11** stödjer PWM (markerade med `~`)
Analoga pinnar	A0–A5, kan läsas med `analogRead` (0–1023).
Strömförsörjning	USB (5 V) eller externt via DC-jacket (7–12 V → regleras till 5 V internt).
5 V-pin	Ger ut 5 V vid USB-strömförsörjning. Totala budgeten är ca 450 mA efter att kortets egna komponenter tagit sitt. Mer än så och USB-portens polyfuse (500 mA) bryter.
3,3 V-pin	Ger ut 3,3 V. Användbar för vissa sensorer. Max 50 mA.
GND	Tre stift, alla elektriskt samma punkt. Vilken som helst fungerar.

Reset-knappen	Startar om din sketch från början. Användbar om den krashar.
Inbyggd LED	Pin 13, alltid kopplad till en intern LED. Använd `LED_BUILTIN` i koden.

Kabelbröderna — DuPont-kablar

De färgade trådarna du använder för att koppla mellan breadboard och Arduino.

M-M (hane-hane)	Stift i båda ändar. För breadboard-till-breadboard eller breadboard-till-header.
M-F (hane-hona)	Ena änden stift, andra änden hylsa. För Arduino-header till breadboard, eller till direkta komponenter som tilt-sensor.
F-F (hona-hona)	Hylsa i båda ändar. Används sällan i kursen.
Typisk längd	10–20 cm. Inte kritiskt men kortare är snyggare.

74HC595 skiftregister — den komponenten vi inte använder

Kittet innehåller **en** komponent som kursen inte rör: ett 74HC595-skiftregister. Det är en 16-bens IC (integrerad krets) som låter Arduinon styra 8 LED-utgångar med bara 3 pinnar.

74HC595 skiftregister	16-bens DIP-IC. Tre styrpinnar (`data` , `clock` , `latch`) kan styra 8 utgångar. Principen: skiftregistret ”kommer ihåg” en 8-bits sekvens du matar in en bit i taget, och håller dem alla samtidigt på sina utgångar.

Det är en utmärkt första fördjupning efter kursen — prova att blinka åtta LED:ar i ett löpande mönster. Sök efter ”74HC595 shift register Arduino” för exempel.

Vill du ha fler komponenter?

Kittet är medvetet minimalt — de komponenter ni byggt med under fem träffar är i princip precis de som ryms. Om ni vill gå vidare:

- **Utökningskit** (Super Starter Kit, Most Complete Kit) innehåller saker som servomotor, LCD-display, IR-mottagare, temperatursensor, ultraljudsavståndsmätare, m.fl.
- **Lösa komponenter** från Kjell & Company, Electrokit eller liknande när ni har ett specifikt projekt i åtanke. Ofta mer prisvärt än att köpa ett nytt kit.
- **Online-tutorials** — Hackster.io och Instructables förklarar nästan alltid vad de använder och var man får tag på det.

BILAGA D

Hackathon-lösning

En fullständig, kommenterad referens-sketch för tjuvlarmet.

När du ska titta hit

Den här bilagan innehåller en **fullständig körbar** version av hackathon-koden. Den finns här av två skäl:

1. **Om du fastnar hopplöst under hackathonen** och kursledaren redan har hjälpt tre gånger, så att du inte blockeras av en bug som inte är pedagogiskt värdefull att lösa.
2. **Som referens efter kursen** när du vill bygga vidare eller när du lägger en gammal sketch åt sidan och vill återvända.

Titta inte hit för tidigt. Halva värdet av hackathonen är att brottas med integrationen själv. Att läsa svaret innan du provat är som att titta på baksidan av ett korsord innan du försökt.

Regler för larmet (repeterade)

1. **Knappen** på pin 9 togglar `larmPaslaget` med edge-detection.
2. Om **larmet är på och tilt-sensorn är lutad** → buzzern tjuter och RGB-LED blinkar rött.
3. Om **larmet är av och fotocellen läser under en tröskel** → RGB-LED tänds mjukt som stämningsljus.
4. Annars → allt är tyst och mörkt.

Komplett sketch

```
// ─── Pin-tilldelning ──────────────────────────────────────
const int knappPin  = 9;   // Modul 3 – digital in
```

```
const int tiltPin   = 2;   // Modul 4 – digital in
const int ldrPin    = A0;  // Modul 4 – analog in
const int buzzerPin = 12;  // Modul 3 – digital out
const int ledR      = 6;   // Modul 2 – PWM out
const int ledG      = 5;   // Modul 2 – PWM out
const int ledB      = 3;   // Modul 2 – PWM out

// ─── Tröskel för fotocellen ──────────────────────────
// Kalibrera själv i Serial Monitor. Låg siffra = mörkt.
const int morkTroskel = 300;

// ─── Tillståndsvariabler (globala, lever mellan loopar) ─
bool larmPaslaget = false;
int  lastKnappState = HIGH;  // HIGH = släppt (INPUT_PULLUP)

// ─── Hjälpfunktion för att skriva RGB ────────────────
void sattFarg(int r, int g, int b) {
  analogWrite(ledR, r);
  analogWrite(ledG, g);
  analogWrite(ledB, b);
}

// ─── SETUP ───────────────────────────────────────────
void setup() {
  Serial.begin(9600);
  pinMode(knappPin,  INPUT_PULLUP);
  pinMode(tiltPin,   INPUT_PULLUP);
  pinMode(buzzerPin, OUTPUT);
  pinMode(ledR, OUTPUT);
  pinMode(ledG, OUTPUT);
  pinMode(ledB, OUTPUT);

  Serial.println("Larmet startat. Larm av som default.");
}

// ─── LOOP ────────────────────────────────────────────
void loop() {
  // 1. LÄS INPUTS
  int  knappState = digitalRead(knappPin);
  bool tiltLutad  = (digitalRead(tiltPin) == LOW);
  delay(50);  // debounce tilt – kulan bouncar i hylsan
  int  ljus       = analogRead(ldrPin);

  // 2. EDGE-DETECTION för knappen (toggla larmläget)
  if (knappState == LOW && lastKnappState == HIGH) {
    larmPaslaget = !larmPaslaget;
    Serial.print("Larm nu ");
    Serial.println(larmPaslaget ? "PÅ" : "AV");
  }
  lastKnappState = knappState;

  // 3. BESTÄM VAD SOM SKA HÄNDA
  if (larmPaslaget && tiltLutad) {
```

```
    // LARM AKTIVT + RÖRT → tjut + rött blink
    digitalWrite(buzzerPin, HIGH);
    sattFarg(255, 0, 0);
    delay(100);
    digitalWrite(buzzerPin, LOW);
    sattFarg(0, 0, 0);
    delay(100);
  } else if (!larmPaslaget && ljus < morkTroskel) {
    // LARM AV + MÖRKT → stämningsljus (mjukt varmt)
    digitalWrite(buzzerPin, LOW);
    sattFarg(120, 60, 20);
  } else {
    // TYST OCH MÖRKT
    digitalWrite(buzzerPin, LOW);
    sattFarg(0, 0, 0);
  }

  // 4. Kort paus mot kortkrets-looping
  delay(10);
}
```

Rad-för-rad-förklaring

PIN-TILLDELNING (RAD 2–8)

Alla pin-nummer samlade överst i filen. `const int` så de inte kan råka ändras. Den här bara-deklarations-stilen är standard bland Arduino-sketchar — allt läsbart från toppen, inga magiska siffror gömda i mitten av koden.

`MORKTROSKEL` (RAD 12)

`300` är en gissning. Du ska kalibrera själv med Serial Monitor. Lämna värdet som en `const int` i toppen så du enkelt kan ändra det utan att leta genom hela filen.

`LASTKNAPPSTATE` (RAD 16)

Startas på `HIGH` eftersom `INPUT_PULLUP` gör att en **släppt** knapp läses som HIGH. Om du startar på `LOW` kan edge-detection tro att knappen var tryckt första loopen och toggla larmet direkt.

> **NAMING-SKIFTE MOT MODUL 3**
>
> I Modul 3:s slides och ”Bygg från minnet” hette variablerna `state` och `lastState`. Här i hackathon-koden står det `knappState` och `lastKnappState`. Anledningen är att Bilaga D läser flera tillstånd (knapp, tilt, ljus) i samma loop — då är `state` för otydligt. När du själv skriver hackathon-koden kan du behålla `state` / `lastState` om du bara har en knapp; namnen är bara etiketter, mönstret är detsamma.

`sattFarg()` hjälpfunktion (rad 19–23)

Att skriva `analogWrite(ledR, ...)` + `analogWrite(ledG, ...)` + `analogWrite(ledB, ...)` på varje rad blir tröttsamt. En funktion som tar tre tal och skriver dem i en svep är både kortare och lättare att läsa. `sattFarg(255, 0, 0)` betyder ”helt rött”.

Det är det första ”egenskrivna” verktyget i koden. Du kunde klarat dig utan, men det **gör koden läsbar**. Det är i princip vad programmering i större skala handlar om — att skapa små, tydliga verb för vad du vill göra.

Edge-detection (rad 46–51)

Samma mönster som i Modul 3: **agera bara när knappen just nu övergår från HIGH till LOW**. Utan detta skulle larmet toggla 100+ gånger per tryckning, och värdet skulle vara slumpmässigt vid släpp.

`larmPaslaget ? "PÅ" : "AV"` är en kort ternär if — ”om larmPaslaget, använd ’PÅ’, annars ’AV’”. En rad istället för fyra.

Tre grenar av if/else (rad 55–73)

Den första grenen har en liten detalj värd att notera: den alternerar mellan rött och svart med `delay(100)` — det är hur blink skapas inne i ett `loop`-varv. Varje gång loopen kommer tillbaka till den här grenen, kör det ett fullt blink-cykel på 200 ms.

Det betyder att knapptrycket inte reagerar förrän **efter** en blink-cykel. 200 ms är knappt märkbart för en människa, men om du vill att larmet ska svara snabbare på knappen kan du göra blinkningen med `millis()` istället för `delay()`. Det är en ren utökning — ingår inte i kursen men finns i alla online-tutorials under sökordet ”non-blocking Arduino”.

`delay(10)` sist (rad 77)

En liten paus mellan loop-varv. Gör att Arduinon inte snurrar CPU:n på tomgång. Tio millisekunder är kort nog att interaktionen känns omedelbar men tillräckligt för att knappens studs lägger sig innan nästa läsning.

Varianter att prova

Variation 1 — dubbelt pip vid larm på/av

Istället för att bara ändra `larmPaslaget` tyst, låt buzzern säga till med ett kort pip:

```
if (knappState == LOW && lastKnappState == HIGH) {
  larmPaslaget = !larmPaslaget;

  // Pip-bekräftelse
  digitalWrite(buzzerPin, HIGH);
  delay(50);
  digitalWrite(buzzerPin, LOW);
  if (!larmPaslaget) {
    delay(50);
```

```
    digitalWrite(buzzerPin, HIGH);
    delay(50);
    digitalWrite(buzzerPin, LOW);
  }
}
```

Ett pip = larm på, två pip = larm av. Bekräftar aktiveringen utan att säga något.

VARIATION 2 — HYSTERES PÅ MÖRKERTRÖSKELN

Ett problem: om `ljus` ligger precis på `morkTroskel = 300`, kan värdet darra mellan 299 och 301 och lampan blinkar av och på. Lösningen är två olika trösklar — en för att tända, en för att släcka:

```
const int tandVid = 280;
const int slackVid = 320;

// Statisk variabel som behåller sitt värde mellan loopar
static bool stamningslampaPa = false;

if (ljus < tandVid) {
  stamningslampaPa = true;
} else if (ljus > slackVid) {
  stamningslampaPa = false;
}
// (lampan behåller sitt senaste tillstånd när ljus är mellan trösklarna)

if (stamningslampaPa) {
  sattFarg(120, 60, 20);
} else {
  sattFarg(0, 0, 0);
}
```

`static` betyder att variabeln behåller sitt värde mellan loop-varv trots att den är lokal. Det är ett alternativ till global `bool stamningslampaPa;` längst upp.

Hysteres är konceptet att två trösklar (tänd, släck) ligger olika, så att systemet inte oscillerar på gränsen.

VARIATION 3 — FADE-IN PÅ STÄMNINGSLJUS

Istället för att LED-en bara **tänds** när det blir mörkt, låt den långsamt tona upp. Detta kräver en lokal `int`-variabel som minns nuvarande ljusstyrka mellan loop-varv:

```
static int stamningsStyrka = 0;

if (!larmPaslaget && ljus < morkTroskel) {
  if (stamningsStyrka < 120) stamningsStyrka++;
} else {
  if (stamningsStyrka > 0) stamningsStyrka--;
}
```

```
analogWrite(ledR, stamningsStyrka);
analogWrite(ledG, stamningsStyrka / 2);
analogWrite(ledB, stamningsStyrka / 6);
```

Varje loop-varv justerar styrkan med 1. Över 120 loop-varv (ca 1,2 sekunder vid `delay(10)`) tonar LED:en från 0 till 120. Mjukare beteende, inget hopp.

Sista rådet

Om du använder den här koden rakt av: **skriv om den för hand**. Inte för att den här versionen är dålig, utan för att du lär dig genom att skriva. En Arduino-bok med 500 sidor är inte lika mycket värd som 200 rader kod du skrivit själv från minnet.

Och om något här känns onödigt komplicerat — **skit i det**. En fungerande enkel lösning är värt hundra eleganta som inte kompilerar.

BILAGA E

Säkerhet

De saker som gör att du inte bränner dig, LED:en, eller Arduinon.

Ofarligt för dig — farligt för utrustningen

Spänningen du jobbar med på kursen är **5 V**. Det är samma spänning som en USB-kabel. **För människor är det helt ofarligt** — du kan hålla båda ledningarna mellan tummen och pekfingret och känna ingenting alls. Du jobbar inte på vägguttags-nivå.

Men **din utrustning** — laptopen, Arduinon, komponenterna — är inte lika robust. En kortslutning mellan `+5 V` och `GND`, en LED utan resistor, en pinne som dras över sin maxgräns — alla är saker som kan stänga av USB-porten (bra) eller rosta komponenter (mindre bra). Fem små saker att veta:

1. LED:en är riktad — kontrollera polariteten

Skickar du ström åt fel håll genom en LED: den lyser inte. Skickar du för mycket ström åt rätt håll: den brinner upp på mindre än en sekund.

- **Långt ben = plus (anod)**. Mot signal-pinnen.
- **Kort ben + platt kant på plastkuddens rand = minus (katod)**. Mot GND.
- **Seriemotstånd är obligatoriskt**. 220 Ω duger för 5 V.

Utan resistor går mer än 20 mA genom LED:en, och den hinner lysa i högst några sekunder innan den bränns upp permanent. Den är inte farlig för dig — men du får byta. Kittet har ett par extra, men regeln är enkel: **aldrig en LED utan seriemotstånd**.

2. Ingen pinne tål hur mycket ström som helst

Arduinons digitala pinnar har en **rekommenderad maxström på 20 mA per pinne** (absolut maximum enligt databladet är 40 mA, men det sliter på chippet). Totalt genom chippet

(alla pinnar tillsammans): absolut max är **200 mA** — håll dig klart under det. Drar du mer bränner du en liten effekttransistor inne i chippet. Det märks inte direkt — men efter ett tag slutar pinnen svara på `digitalWrite`.

För kursen är det inget problem. En LED via 220 Ω drar ca 13 mA. En RGB-LED med alla tre kanaler full = ca 40 mA. En buzzer = ca 30 mA. Allt med god marginal.

Fel att undvika:

- En LED direkt till 5 V utan resistor → drar 25+ mA direkt, bränner pinnen eller LED:en eller båda.
- Två LED:ar parallellt på samma resistor → delar strömmen olika, en tar skada.
- Försök driva en motor direkt från pinnen → motorer drar hundratals mA, Arduinon överlever inte.

Regel: om du kopplar något som inte är uttryckligen beskrivet i kursen, räkna om strömmen med Ohms lag först.

3. Kortslut inte 5 V och GND

Om du direkt kopplar `+5 V` till `GND` med en ren kabel, börjar Arduinon dra hela strömmen den kan leverera. Kabeln blir varm; i värsta fall smälter dess isolering eller en spårledning på kortet bränns.

Det låter allvarligare än det är. USB-porten har en skyddskrets som bryter vid ca 500 mA. Men: kabeln blir VÄLDIGT varm, laptop-varningen poppar upp, och du får koppla ur i rätt tid.

Vanligaste sättet att göra det av misstag:

- En lös kabel som hamnar i GND-raden och rör plus-raden på breadboardens power rails.
- En resistor med brutna ben där ena änden rör plus och andra änden rör minus utanför kretsen.
- Förvirrande fingerplats när du sticker in en kabel.

Om du märker att Arduinon blir varm — **koppla ur USB:n direkt**. Kolla vart du senast drog en kabel.

4. Statisk elektricitet är en verklig sak

Torr vinterluft + nyfallen snö + syntetiska tröjor → du bygger upp några tusen volt statisk elektricitet på kroppen. När du rör Arduino-chippets pinnar med ett finger kan det ladda ur sig genom chippet och permanent skada det.

Det är en liten risk på Arduino Uno (den är byggd för att vara robust), men risken finns för andra IC-komponenter. I kursen är det sällsynt att detta händer, men värt att veta.

Förebygg: ta tag i något jordat (en metallkran, värmeelementet, laptopens USB-kontakt) innan du börjar koppla om du misstänker att du är uppladdad. Eller: **rör aldrig bara själva metall-pinnarna på chipen** — håll i plastkanterna eller i själva kretskortet.

För kursens komponenter spelar det mindre roll. LED, resistor, knapp, buzzer, fotocell — ingen av dem är känslig för statisk. Men IC-kretsar som 74HC595 **är**.

Säker felsökning

När något inte fungerar är det ibland frestande att ”testa med att snabbt koppla fram och tillbaka”. **Koppla inte om medan Arduinon är strömförsörjd.** Dra ur USB:n, koppla om, koppla i igen. Det skyddar mot oväntade kortslutningar och gnistor.

Ordning:

1. Koppla ur USB.
2. Ändra kopplingen.
3. Kontrollera att inget oväntat är kortslutet (titta, tänk efter).
4. Koppla i USB.
5. Ladda upp ny kod om det behövs.

Det tar 5 extra sekunder per iteration och sparar miljoner sekunder av felsökning.

Och till slut: ofarligt för människan

Säg det högt en gång till: 5 V via USB är inte farligt för **dig**. Du kan hålla båda ledningarna, röra vid chipet med handen, droppa vatten på breadboardet (nej, bara skämt — torka upp det), och inget händer med dig. Det är komponenterna och laptopen man skyddar — inte fingrarna.

Det värsta som kan hända på kursens kompetensnivå är:

1. En LED brinner upp → du får byta till en av de andra i kittet.
2. En USB-port i laptopen stänger av p.g.a. överskydd → du kopplar i igen.
3. En Arduino slutar svara på en pinne → byt till en annan pinne och anteckna det.

Ingenting brinner. Ingen skadas. Värsta utfallet är att du får köpa en ny 29-kronors LED från Kjell & Company. **Slappna av. Bygg.**

BILAGA F

Ohms lag & framspänningsfall

Den tyngsta bilagan. Här utreder vi varför 220 Ω är 220 Ω och ingenting annat.

Vad kapitlet handlar om

Det här är inte en rundtitt. Det är ett djupdyk. Om ni satt på träff 1 och undrade ”men **varför** 220 Ω?” utan att hinna fråga, är det här svaret. Om ni redan kunnat svaret i detalj, är det en repetition på ert modersmål.

Målet är att efter läsningen ska ni kunna:

1. **Räkna ut** vilken resistor en given LED behöver utan att googla.
2. **Förklara** varför en LED inte följer Ohms lag direkt.
3. **Förstå** vad som händer om resistorn är för liten eller för stor.
4. **Läsa** en spänningsdelar-koppling som fotocellens och veta varför mellanpunkten rör sig när ljuset ändras.

Allt med svenska ord, inga latinska uttryck som jag inte vill förklara.

Ohms lag — den ultimata tre-storheten

Tre storheter som alltid hör ihop i en elektrisk krets:

Spänning (U)	Tryckskillnaden mellan två punkter. Enhet: volt (V). Exempel: 5 V.
Ström (I)	Hur mycket laddning som rör sig per sekund. Enhet: ampere (A). I småsensorik räknas det ofta i milliampere (mA).

Motstånd (R) Hur mycket ett material bromsar strömmen. Enhet: ohm (Ω). Exempel: 220 Ω.

FORMELN

$$U = I \cdot R$$

I ord: **spänningen över en resistor är lika med strömmen genom den, gånger resistorns värde**.

Omformulerat för de gånger du söker annan storhet:

$$I = \frac{U}{R} \qquad R = \frac{U}{I}$$

Eller med den klassiska **triangeln**. Rita en pyramid med U på toppen och I och R i botten. Håll över den storhet du vill räkna ut, så ser formeln sig själv.

```
  U
-----
I · R
```

Håll för U ⟶ ser du $I \cdot R$. Håll för I ⟶ ser du $\frac{U}{R}$. Håll för R ⟶ ser du $\frac{U}{I}$. Fungerar **varje gång**.

VARFÖR BOKSTÄVERNA U OCH I?

U kommer från tyskans ”Spannung” (spänning). **I** från franskans ”intensité du courant” (strömintensitet, Ampères terminologi).

Anglosaxiska böcker använder ibland V för spänning istället för U — samma sak, olika lokala traditioner. I Sverige och Tyskland: U. I Storbritannien och USA: V.

FÖRSTA RÄKNEEXEMPLET

Om du har en resistor på 220 Ω och strömmen genom den är 15 mA, hur stor är spänningen över den?

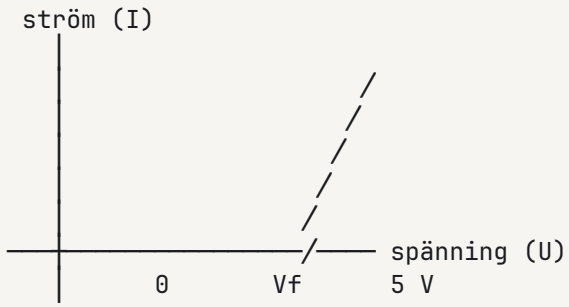
$$U = I \cdot R = 0,015 \text{ A} \cdot 220\Omega = 3,3 \text{ V}$$

Bra att komma ihåg: **ström ska omvandlas till ampere (inte milliampere) innan du räknar**. 15 mA = 0,015 A.

Varför en LED **inte** följer Ohms lag

Här kommer det som knyter ihop allt. En resistor är en snäll komponent — den följer Ohms lag slaviskt. Dubbla spänningen, dubbla strömmen. Halvera spänningen, halvera strömmen. Linjärt.

En **LED är inte en resistor**. Den är en halvledar-diod med en trösklad I-V-karakteristik. På en graf av ström mot spänning ser den ut så här (ungefär):



Upp till `Vf` (framspänningen): nästan ingen ström alls. LED:en lyser inte.

Vid `Vf`: LED:en ”vaknar” och börjar leda ström. Den lyser.

Strax över `Vf`: ström stiger **dramatiskt** med minimal ökning i spänning. Det här är farligt — en liten ökning av spänningen ger stor ökning av strömmen.

Så vad är Vf? Det är **framspänningsfallet** — den spänning LED:en ”kräver” för att leda ström. För en röd LED är Vf ungefär 2 V.

KONSEKVENSEN

Om du bara kopplar en röd LED till +5 V utan resistor, försöker LED:en dra så mycket ström den kan. Den lägger 2 V över sig själv (sin Vf) och de återstående 3 V har ingenstans att ta vägen — förutom att driva STRÖM genom LED:en. Massiv ström. Dioden bränns upp på under en sekund.

Lösningen: en resistor i serie som ”äter upp” de återstående 3 V:en och håller strömmen på en säker nivå.

Räkneexemplet — hur man kommer fram till 220 Ω

Krav:

- Matningsspänning: **5 V** från Arduinons pin 13.
- LED: **röd**, framspänningsfall ≈ **2 V**.
- Önskad ström: **15 mA** = 0,015 A. (Säker nivå långt under LED:ens 20 mA-max.)

Tanken är: resistorn och LED:en ligger i **serie**. Strömmen genom dem är samma. Spänningen fördelas så att LED:en tar sina 2 V och resistorn tar resten. Det ska bli 5 V totalt.

$$U_{\text{resistor}} = 5\ \text{V} - 2\ \text{V} = 3\ \text{V}$$

Nu vet vi spänningen över resistorn och strömmen genom den. Ohms lag löser resten:

$$R = \frac{U}{I} = 3\frac{\text{V}}{0},015\ \text{A} = 200\Omega$$

Exakt svar: 200 Ω. Men resistor-värden säljs bara i fasta standardstegstorlekar — E12-serien, som innehåller 100, 120, 150, 180, **220**, 270, 330, 390, 470, osv. Det närmsta tillgängliga värdet över 200 är **220 Ω**.

Kolla vad strömmen faktiskt blir med 220 Ω:

$$I = \frac{5\ \text{V} - 2\ \text{V}}{220}\Omega = 13,6\ \text{mA}$$

Lite lägre än vårt ideal på 15 mA. **Det är helt OK.** LED:en lyser fortfarande starkt och håller sig inom säkra gränser. Skillnaden är ungefär som om en lampa kördes på 85 % av sin fulla styrka — synligt, men marginellt.

Det är **därför** 220 Ω valdes. Inget gissningsval — ren matematik.

Vad händer om resistorn är för **liten**?

Säg att du byter till **120 Ω** istället. Strömmen blir:

$$I = \frac{5\ \text{V} - 2\ \text{V}}{120}\Omega = 25\ \text{mA}$$

25 mA > 20 mA (LED:ens max). Den börjar värmas upp, lyser starkare initialt, och tappar ljusstyrka efter någon minut som kiselet inuti degraderar. Efter några timmar eller dagar är den obrukbar — ljuset blir märkbart svagare, slut.

Säg att du använder **ingen resistor alls** (0 Ω). Då begränsar bara LED:ens egen fysik strömmen. Den lägger 2 V över sig själv, men det finns inget som tar upp de andra 3 V — strömmen skenar iväg och LED:en bränns upp på **mindre än en sekund**.

> **EN LED UTAN SERIEMOTSTÅND ÄR EN DÖD LED INOM FEM SEKUNDER**
> Låt det här vara en av de första reglerna ni lär er utantill. **Varje LED behöver en resistor i serie**. Utan undantag på kursens nivå. När ni sitter hemma och experimenterar och det finns en dag då ni är trötta och stressade och hoppar över resistorn ”bara den här gången” — det är dagen ni får köpa nya LED:ar.

Vad händer om resistorn är för **stor**?

Säg att du använder **2,2 kΩ** (2200 Ω) istället för 220. Strömmen blir:

$$I = \frac{5\ \text{V} - 2\ \text{V}}{2200}\Omega = 1,4\ \text{mA}$$

Det är tio gånger **mindre** än 15 mA. LED:en lyser — men **svagt**. Ungefär som en stearinlåga i dagsljus. Du kan knappt se att den är tänd. Ingen skada, bara svag.

Om resistorn är ännu större — säg **10 kΩ** — blir strömmen 0,3 mA. LED:en lyser kanske, kanske inte, beroende på LED:ens tröskelström. Osynlig för praktiska ändamål.

Så: **för liten resistor = LED dör**. **För stor resistor = LED lyser svagt eller inte alls**. Mellan dem finns ett lagom intervall på säg 180–330 Ω för röda LED:er på 5 V. 220 Ω ligger mitt i mjuka delen.

Framspänningsfall för andra färger

En viktig detalj: **olika LED-färger har olika Vf**. Det betyder att räkningen för en grön eller blå LED ger ett annat resistor-värde.

Röd	Vf ≈ 1,8 – 2,0 V. Resistor för 15 mA på 5 V: runt 200 Ω → välj 220.
Gul	Vf ≈ 2,0 – 2,2 V. Nästan samma som röd. 220 Ω fungerar.
Grön	Vf ≈ 2,0 – 2,2 V (klassisk grön) eller 3,0 – 3,2 V (modern ”ren grön”). 220 Ω för den klassiska.
Blå	Vf ≈ 3,0 – 3,4 V. Resistor: (5 − 3,2) / 0,015 ≈ 120 Ω. Men 220 Ω fungerar också, bara något svagare.
Vit	Vf ≈ 3,0 – 3,4 V (samma som blå, eftersom vita LED:er är blåa med fosforcoating). 220 Ω duger.

Varför 220 Ω är ett säkert allroundval: det är lagom för alla färger i kittet. Röda LED:ar hamnar på 13 mA (säker). Blåa LED:ar hamnar på 7 mA (något svaga men fungerar). Ingen färg får för mycket ström.

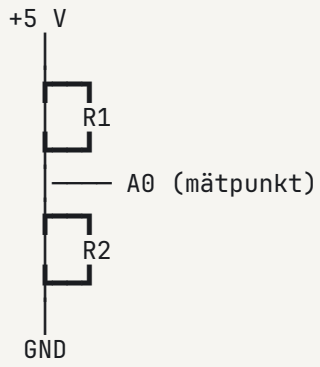
Om du vill optimera: räkna om med varje LED:s Vf och välj närmaste E12-värde. För kursen: 220 Ω överallt, klart.

Spänningsdelaren — ohms lag för **två** resistorer

I Modul 4 byggde ni en **spänningsdelare** för att läsa av fotocellens motstånd. Så här fungerar den på papper.

KOPPLINGEN

Två resistorer i serie mellan +5 V och GND. Mätpunkten är **mellan** dem.



Strömmen genom hela serien är samma (det är vad ”serie” betyder):

$$I = \frac{5\ \text{V}}{R_1 + R_2}$$

Spänningen vid mätpunkten, mätt mot GND, är spänningen **över R2**:

$$V_{\text{mid}} = I \cdot R_2 = \left(\frac{R_2}{R_1 + R_2}\right) \cdot 5\ \text{V}$$

Detta är **spänningsdelar-formeln**. Den säger: andelen av inkommande spänning som hamnar på mätpunkten är samma som andelen R_2 i den totala resistansen.

EXEMPEL 1 — TVÅ LIKA STORA RESISTORER

Om $R_1 = R_2 = 1$ kΩ:

$$V_{\text{mid}} = \left(\frac{1000}{1000 + 1000}\right) \cdot 5 = 0,5 \cdot 5 = 2,5\ \text{V}$$

Mellanpunkten ligger exakt halva. Intuitivt vettigt: lika bromsning på båda sidorna = spänningen delas lika.

EXEMPEL 2 — FOTOCELL I MÖRKER

Fotocell i mörker ≈ 50 kΩ. Fast resistor = 1 kΩ. Fotocellen är R1 (över), 1 kΩ är R2 (under):

$$V_{\text{mid}} = \left(\frac{1000}{50000 + 1000}\right) \cdot 5 = 0,0196 \cdot 5 \approx 0,098\ \text{V}$$

A0 visar ett väldigt lågt värde. Arduinons analog-till-digital-omvandlare mäter 0,098 V → ungefär 20 på 0–1023-skalan.

EXEMPEL 3 — FOTOCELL I STARKT LJUS

Fotocell i starkt ljus ≈ 500 Ω. Fast resistor = 1 kΩ:

$$V_{\text{mid}} = \left(\frac{1000}{500 + 1000}\right) \cdot 5 = 0,667 \cdot 5 \approx 3,33\ \text{V}$$

A0 visar cirka 682 (3,33 V / 5 V × 1023 ≈ 682).

SÅ VARFÖR PRATAR NI OM SIFFROR SOM 150 / 400 / 900?

Verkligt rumsljus ligger någonstans mellan mörk och strålande sol. Arduinons ADC är 10-bits (1024 steg) vilket innebär runt 5 mV upplösning. I praktiken:

Mörker, hand över cellen	A0 ≈ 20–100
Rumsljus, taklampa	A0 ≈ 150–400
Direkt bordslampa	A0 ≈ 500–700
Ficklampa direkt	A0 ≈ 700–900
Midsommarsol direkt	A0 ≈ 900 – 1023 (mättad)

Dessa är **riktvärden**. Din exakta fotocell, din breadboard, ditt rum — alla påverkar. Kalibrera alltid med Serial Monitor innan du sätter en tröskel i koden.

Serie-resistorer, parallell-resistorer — snabbregler

För den som vill veta lite mer:

Två resistorer i serie: den totala resistansen är summan.

$$R_{\text{total}} = R_1 + R_2$$

Två resistorer parallellt (ena änden ihop, andra änden ihop): den totala resistansen är **mindre** än den minsta. Formeln:

$$R_{\text{total}} = \frac{R_1 \cdot R_2}{R_1 + R_2}$$

Ett specialfall värt att komma ihåg: två **lika stora** resistorer parallellt = halva värdet. Två 220 Ω parallellt = 110 Ω.

I kursen använder vi bara resistorer i serie. Men när ni googlar Arduino-projekt online kommer ni stöta på parallell-kopplingar också — nu vet ni vad de betyder.

Sammanfattning — det mest centrala

1. **Ohms lag:** $U = I \cdot R$. Tre variabler, en formel, tre omformuleringar.
2. **En LED kräver resistor i serie.** Alltid. Varje gång.
3. **Räkningen för 5 V + röd LED + 15 mA:** `(5 - 2) / 0,015 = 200 Ω` → välj 220 Ω.
4. **Olika färger = olika Vf**, men 220 Ω funkar som allroundval på 5 V.

5. **Spänningsdelaren** delar upp en spänning mellan två seriekopplade resistorer. Mellanpunktens spänning bestäms av förhållandet mellan resistorerna.

Och det viktigaste: **när något inte fungerar i en krets är oftast svaret ”räkna om”**. Inte ”gissa igen”.