

Physical computing, sensing and control

Sensing, decisions and action
Conditions, loops and Arduino I/O

Prof. Dr. Mehmet Göktürk · Gebze Technical University

What we will be able to explain

How a physical measurement becomes a value in a program.

Why sampling, noise and response time affect a system.

How conditions and loops describe behavior in C.

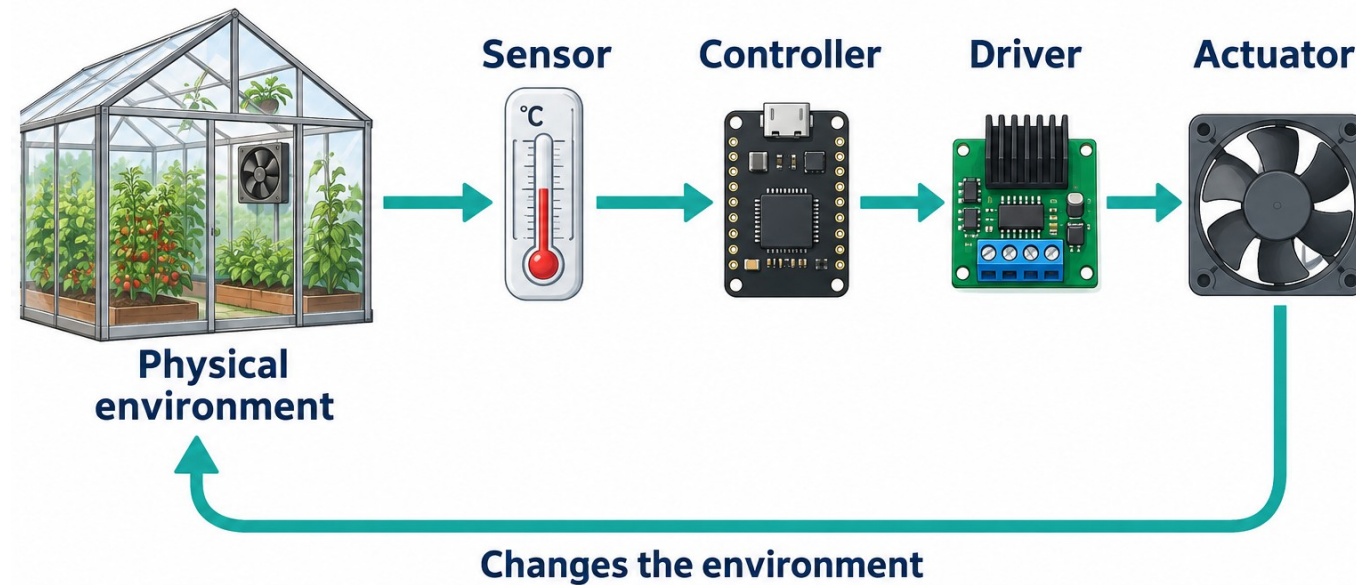
How an Arduino sketch controls an observable output.

Computers outside the desktop

System	What it observes	What it changes
Thermostat	Temperature	Heating or cooling
Automatic door	Presence or position	Door motion
Washing machine	Water level and selected cycle	Valves, pump and motor
Street lighting	Light level or a schedule	Lamp output

A useful system needs an input, a decision rule and an appropriate output.

Feedback in a physical system



A sensor reports the result of an earlier action. The next decision uses the new measurement.

Sensors and actuators

Component	Role	Example
Sensor	Turns a physical quantity into a signal	Temperature sensor, microphone
Input interface	Makes the signal usable by the computer	Digital input or ADC
Actuator	Changes the physical environment	Motor, valve, heater
Output interface / driver	Connects a command to the load	Motor driver or LED circuit

The program handles a representation of the world, not the physical quantity itself.

Open-loop and closed-loop control

Example	Decision	Limitation / benefit
Fan runs for 10 minutes	Use elapsed time only	Does not check whether cooling succeeded
Fan responds to temperature	Measure again and adjust	Responds to disturbances and changing conditions
Onboard LED pattern	Follow a programmed sequence	Makes program behavior visible

Feedback means measuring the effect and using that information in later decisions.

A microcontroller is a small computer

A microcontroller typically integrates a CPU, memory, timers and I/O peripherals on one chip.

An embedded system puts computing inside a larger device with a particular purpose.

An Arduino board adds connections, power circuitry and other support around a microcontroller.

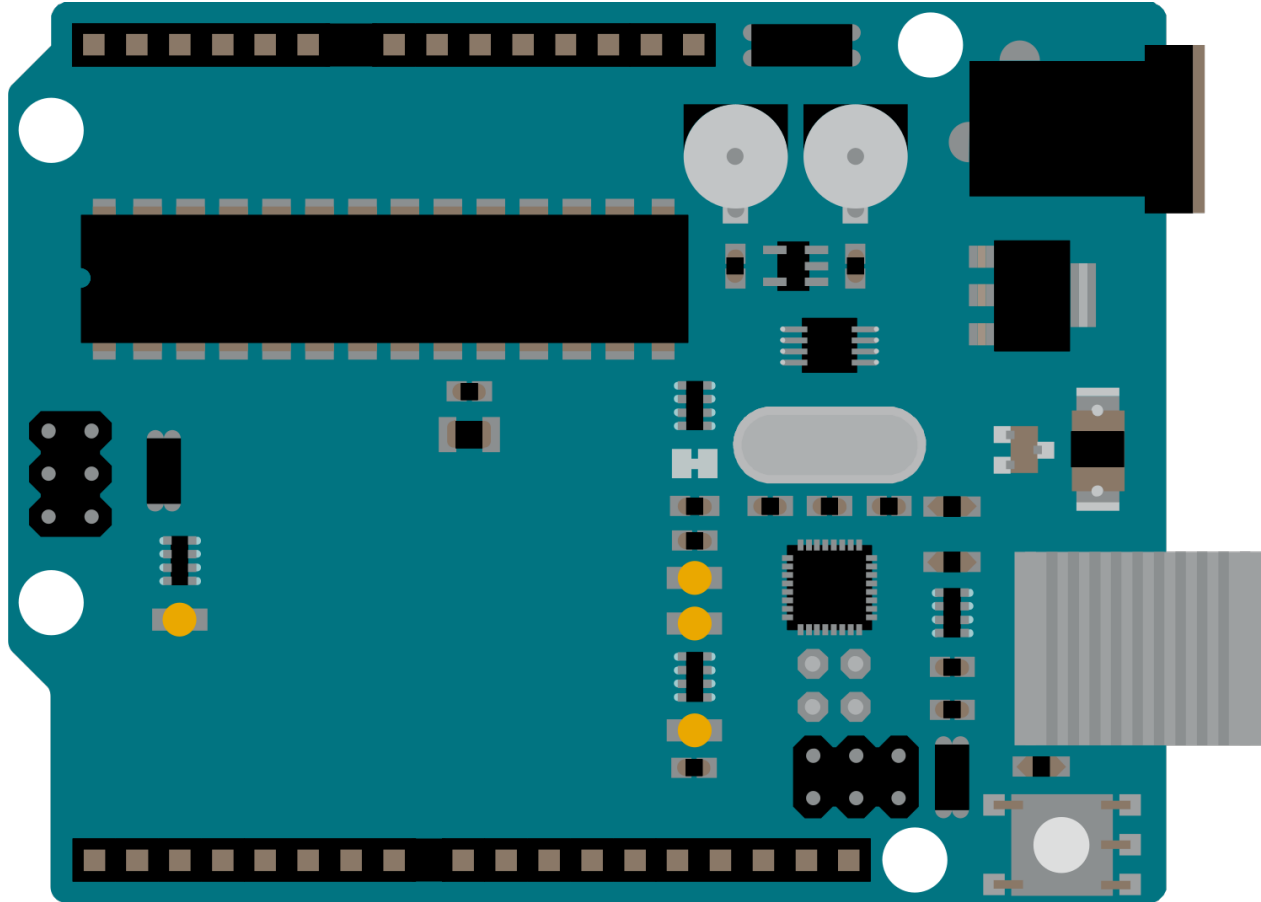
The board and the microcontroller are different objects.

Embedded design is a set of trade-offs

Constraint	Design question	Consequence
Energy	Battery, mains or harvested power?	Affects sensing and communication frequency
Memory	Store every reading or a summary?	Changes how long history can be retained
Time	How late may a response arrive?	Changes scheduling and communication choices
Environment	What heat, moisture or vibration occurs?	Changes component and enclosure choices

The best design meets the application's needs. The fastest processor is not automatically the best fit.

An example board: Arduino UNO R3



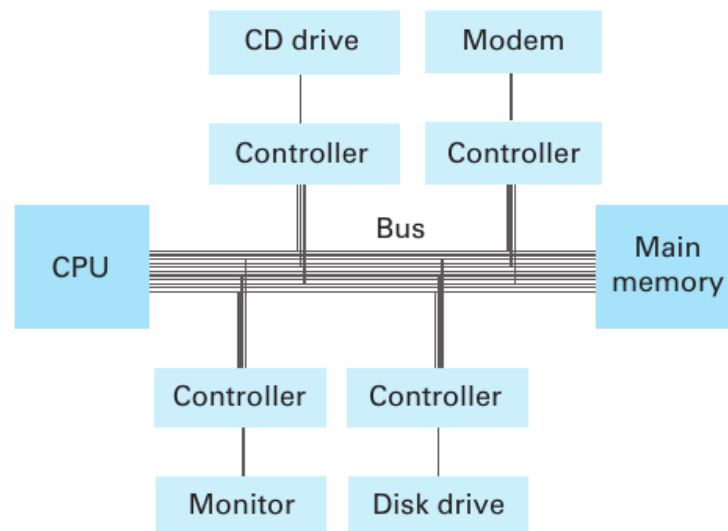
ATmega328P is the main microcontroller.

USB supports programming and communication.

Digital pins, analog inputs and power connections have distinct roles.

UNO R3 is the reference example here. Pin capabilities and voltage levels differ between board models.

I/O hardware connects software to devices



A CPU does not directly understand light, sound or movement.

Controllers and interfaces bridge the computer and external devices.

Brookshear Figure 2.13. Historical device examples illustrate the controller's role; a microcontroller integrates many peripherals.

Analog and digital signals

Aspect	Analog signal	Digital representation
Possible values	A range of physical levels	A finite set of symbols or codes
Example	Sensor voltage changes with temperature	An ADC reading stored as an integer
Physical reality	Noise can alter the voltage	Digital circuits also use physical voltages

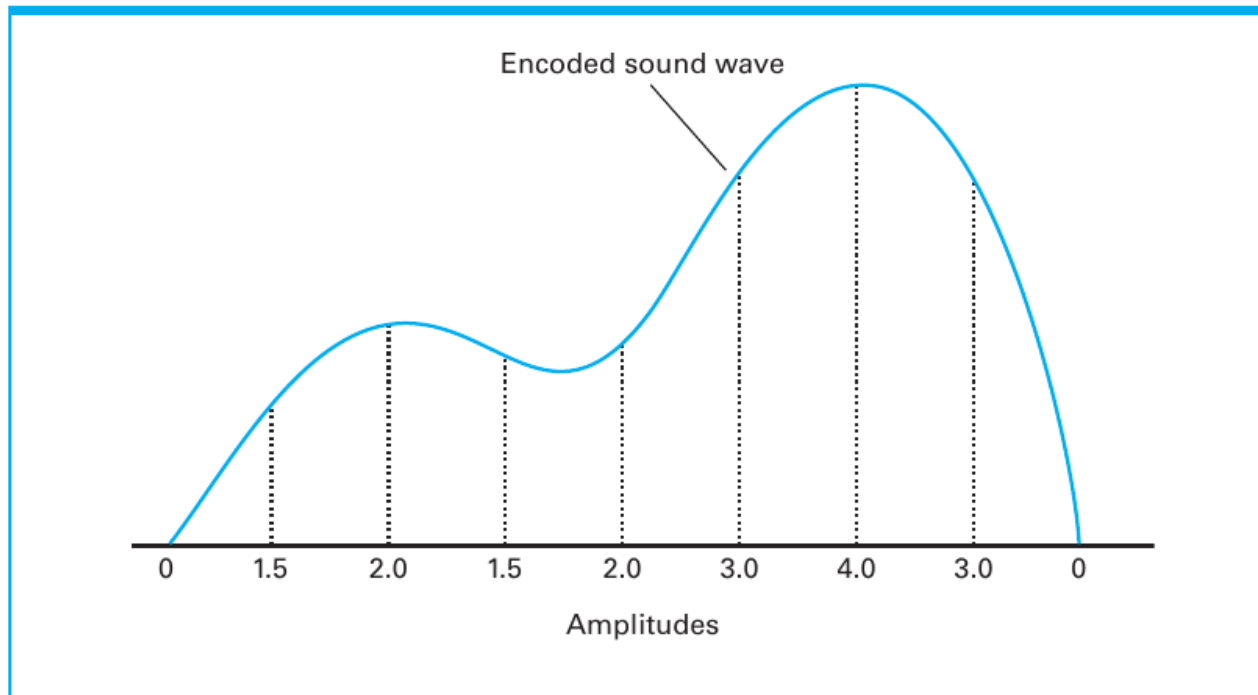
An analog-to-digital converter (ADC) assigns codes to voltages. Digital inputs instead classify levels as HIGH or LOW.

Sampling and quantization are different choices

Choice	What becomes discrete?	What can be lost?
Sampling interval	Time of observation	Events or changes between samples
Quantization step	Represented amplitude	Small differences inside one code interval
Faster sampling	More observations per second	Does not by itself correct a sensor offset
More ADC bits	Smaller ideal voltage increments	Does not by itself detect a missed pulse

A measurement needs both a time scale and a value scale.

Sampling records values at selected times



Each sample captures one observation.

The stored sequence omits what happened between observations.

Brookshear Figure 1.12: a waveform and its sampled amplitudes. Horizontal position follows time; printed numbers give amplitudes.

A short event can fall between samples

Time (ms)	Actual input	Program observation
0	LOW	Sample: LOW
40 to 70	HIGH	No sample in this interval
100	LOW	Sample: LOW
200	LOW	Sample: LOW

Sampling every 100 ms misses this 30 ms pulse. The observation “always LOW” would be misleading.

Aliasing: a moving wheel can appear stationary

Observation	Actual motion	Recorded position
First sample	Mark starts at the top	Top
Second sample	Exactly one full turn later	Top
Third sample	Another full turn later	Top

A stationary wheel gives the same samples. These observations cannot distinguish the two motions.

Choosing a sampling interval

Sampling every 100 ms gives 10 samples per second.

A slowly changing room temperature and a short button press need different observation rates.

Faster sampling costs processing, storage and sometimes power.

Choose the interval from the events you must detect, not just how fast the CPU can run.

Quantization groups values into codes

Illustrative 3-bit ADC	Input interval	Output code
Range: 0 to less than 8 V	0 to less than 1 V	0
8 possible codes	1 to less than 2 V	1
One code per 1 V interval	3 to less than 4 V	3
Highest interval	7 to less than 8 V	7

This is a simplified ADC model. Both 3.1 V and 3.8 V produce code 3, so some detail is lost.

UNO R3: a concrete ADC example

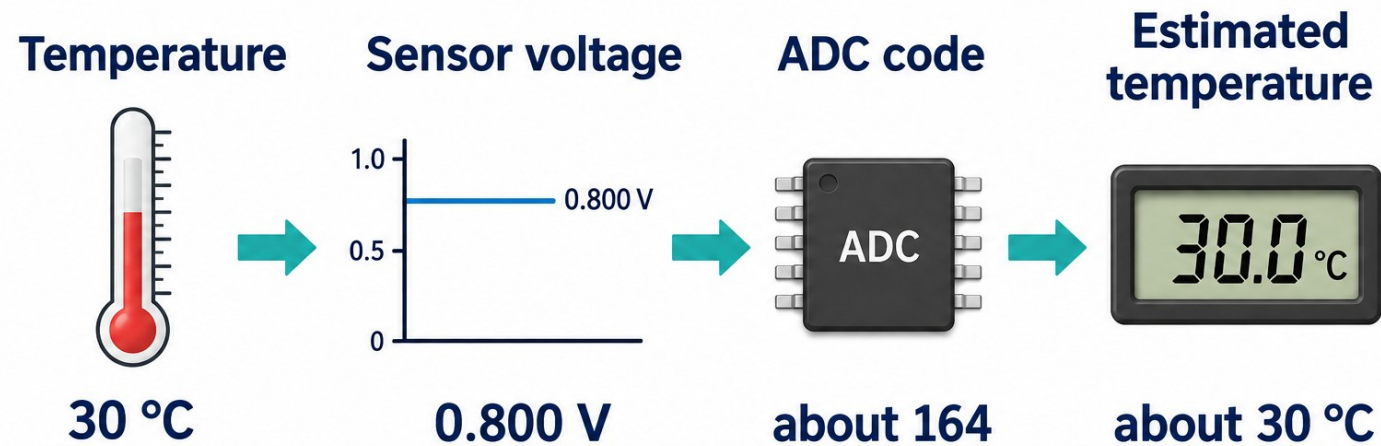
Its 10-bit ADC has 1024 possible codes, numbered 0 through 1023.

With an assumed 5.00 V reference, one ideal step is $5.00 / 1024 \approx 4.88$ mV.

`analogRead(A0)` gives a code. It does not directly give degrees Celsius.

A sensor's conversion rule and the actual reference voltage determine the physical estimate.

A physical measurement becomes a number



Example: 10 mV/°C sensor with 0.500 V offset; 5.00 V ADC reference

A sensor produces a voltage; software applies a conversion model to estimate the physical quantity.

Worked example: a temperature estimate

Step	Calculation	Result
Sensor model	$V = 0.500 + 0.010 \times T$	At 30 °C: $V = 0.800 \text{ V}$
ADC scale	1024 codes over 5.00 V	About 4.88 mV per code
Observed code	Suppose the ADC returns 164	Estimated $V \approx 164 \times 5 / 1024$
Convert the unit	$T \approx (0.8008 - 0.500) / 0.010$	About 30.1 °C

About 0.49 °C per ADC step in this example. Decimal digits in the output do not guarantee that much accuracy.

Resolution is different from accuracy

Term	Meaning	Example
Resolution	Smallest represented increment	ADC step of about 4.88 mV
Accuracy	Closeness to the true value	A sensor consistently reads 2 °C too high
Repeatability	Consistency under the same conditions	Repeated readings cluster together
Calibration	Comparison and correction using a reference	Estimate and correct a systematic offset

More bits do not automatically remove noise, sensor error or a poor reference voltage.

Calibration: correcting an offset

Reference temperature	Sensor reports	Observed error
20.0 °C	22.0 °C	+2.0 °C
30.0 °C	32.0 °C	+2.0 °C
New reading	27.0 °C	Subtract 2.0 °C: estimate 25.0 °C

This correction assumes a stable offset. Two points do not prove accuracy over all temperatures.

Smoothing trades noise for responsiveness

Reading sequence	Four-sample mean	Interpretation
29, 31, 29, 31 °C	30 °C	Fluctuations partly cancel
20, 20, 20, 28 °C	22 °C	A real change first appears smaller
28, 28, 28, 28 °C	28 °C	The estimate catches up

Averaging can reduce some noise, but delays changes. It does not remove a constant calibration offset.

Noise near a decision boundary

Measured temperature	Rule: fan ON at 30 °C or above	Output
29.9 °C	Below threshold	OFF
30.1 °C	At or above threshold	ON
29.8 °C	Below threshold	OFF
30.2 °C	At or above threshold	ON

Small fluctuations can cause repeated switching even when the room hardly changes.

Hysteresis uses two thresholds

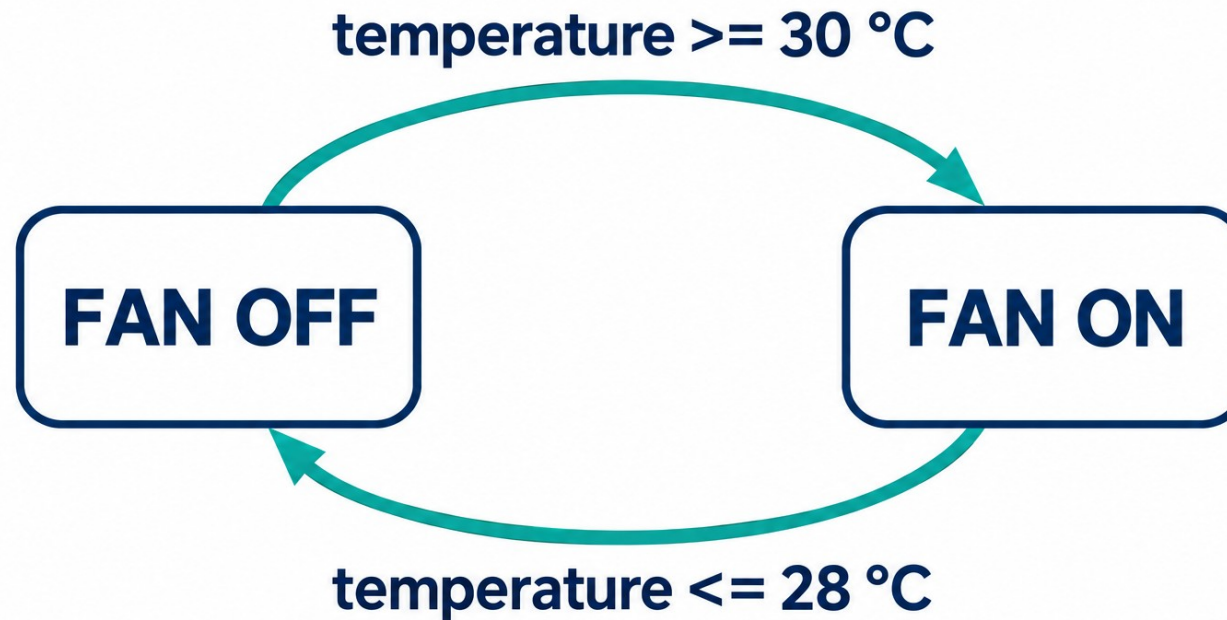
Turn the fan ON at 30 °C or above.

Turn it OFF at 28 °C or below.

Between 28 °C and 30 °C, keep the previous state.

Now the decision depends on both the measurement and the system's state.

The thermostat has two states



Between thresholds: keep the current state

At 29 °C the fan may be ON or OFF. The previous state determines the answer.

A trace of hysteresis

New measurement	State before	State after
29 °C	OFF	OFF
31 °C	OFF	ON
29 °C	ON	ON
27 °C	ON	OFF

The first and third measurements are identical, but their histories differ. Input alone does not determine output.

A command and its electrical effect

A general-purpose input/output (GPIO) pin provides a logic signal with limited current capability.

A motor or heater needs an appropriate driver and power source.

An external LED needs current limiting. The board's built-in LED already has its supporting circuit.

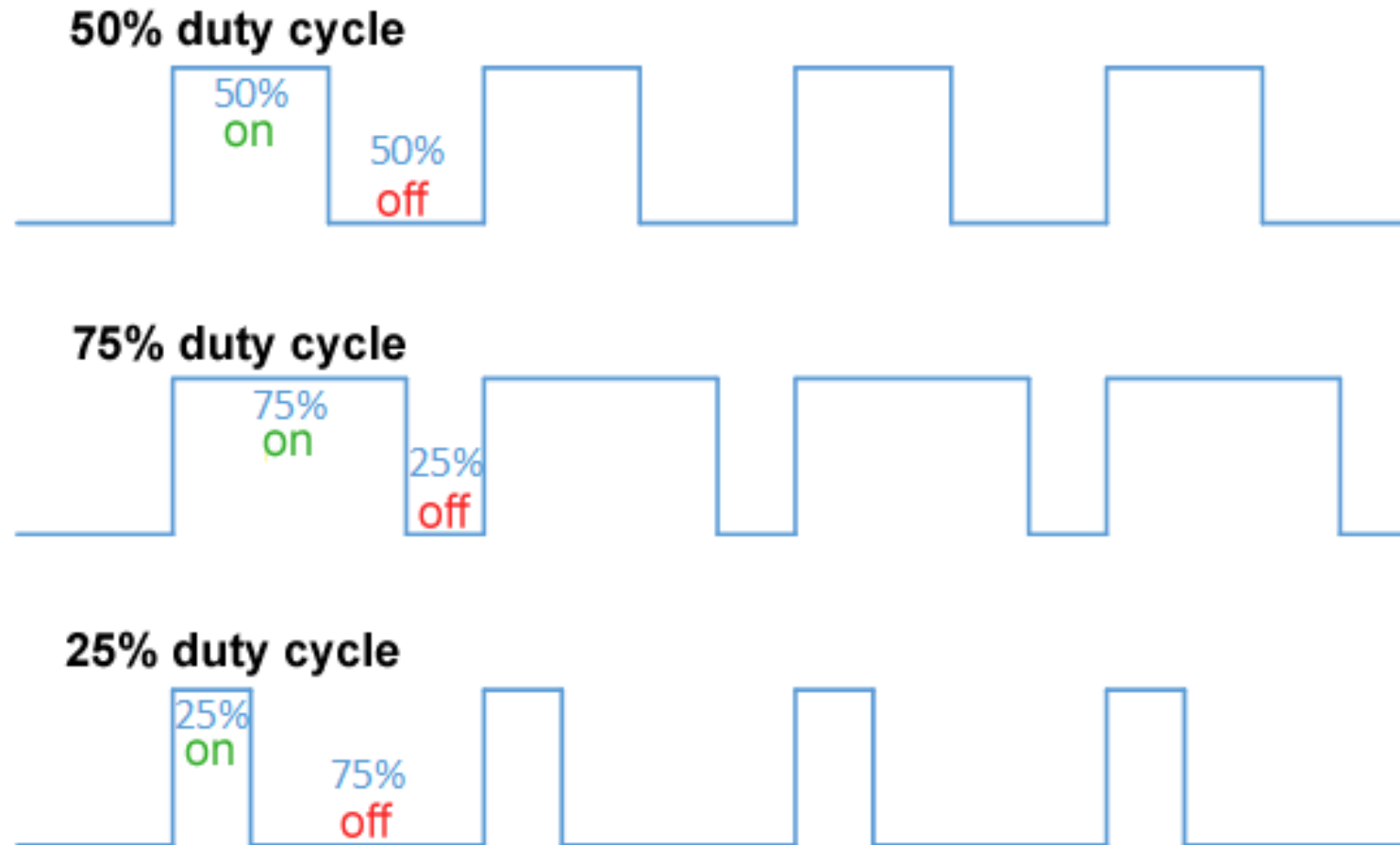
This week's required hardware example uses only the built-in LED.

Pulse-width modulation (PWM)

Duty cycle	Time HIGH in a 4 ms period	Time LOW
25%	1 ms	3 ms
50%	2 ms	2 ms
75%	3 ms	1 ms

On UNO R3 PWM pins, `analogWrite` controls pulses. It does not create a continuously adjustable DC voltage.

PWM waveforms



Same period and voltage levels, different ON time. Figure: Thewrightstuff / Nyq, CC BY-SA 4.0.

Response time belongs to the whole system

A fast CPU cannot observe an input that the program never reads.

With regular 100 ms polling, a persistent change may wait almost 100 ms for the next sample.

Processing, communication and actuator motion add further delay.

For a physical system, “correct” can include responding before a deadline.

A response deadline has a budget

Stage	Illustrative worst-case time
Wait for next sample	100 ms
Read input and decide	5 ms
Send the command	10 ms
Actuator begins the required response	80 ms
Total	195 ms

A hypothetical 150 ms requirement fails. Reducing computation from 5 to 1 ms still leaves 191 ms.

Energy changes the observation strategy

Idealized operating plan	Average current	Ideal battery life
Always active at 20 mA	20 mA	2000 mAh / 20 mA = 100 h
Active 10%, asleep 90%	$0.10 \times 20 + 0.90 \times 0.1 = 2.09$ mA	About 957 h

Sleeping saves energy but can miss events. Real life also depends on batteries, regulators, peripherals and wake-up costs.

Failure detection belongs in the design

Possible problem	What the system can observe	Possible response
Sensor disconnected	Invalid or out-of-range reading	Reject the reading and report a fault
Old reading reused	Timestamp exceeds an allowed age	Treat the measurement as stale
Actuator does not respond	Command and measured effect disagree	Flag the fault and use a defined fallback

A safe fallback depends on the application. “Turn everything OFF” is not a universal policy.

A washing cycle is a sequence of states

State	Transition condition	Next state
FILL	Required water level reached	WASH
WASH	Wash interval completed	DRAIN
DRAIN	Low-water condition confirmed	SPIN
Any active state	Fault condition detected	Defined fault-handling state

Sensors and time govern transitions. An elapsed-time-only design cannot confirm that filling or draining succeeded.

A small design problem: automatic lighting

Design question	One possible answer
What is measured?	Ambient light near the lamp
What is controlled?	Lamp output through a suitable driver
What could go wrong?	Lamp light reaches the sensor and changes its own decision
What improves the design?	Sensor placement, separate thresholds and testing

Before coding, specify the input, desired behavior, operating conditions and possible failure cases.

Different applications need different priorities

Application	Important question	Design consequence
Room thermostat	How quickly does room temperature change?	Slow sensing may be adequate
Automatic door	Was a person detected in time?	Latency and fault behavior need close attention
Washing machine	Which stage is active and has it completed?	State, sensing and interlocks matter

Choose requirements first. Then decide what to sense, how often to observe, and what action is permitted.

Design discussion: a late temperature alarm

A sensor is sampled once per second. A dangerous excursion lasts only 0.2 seconds. Could the alarm miss it?

Would changing from a 10-bit ADC to a 12-bit ADC solve that problem?

What if the reading arrives quickly, but the actuator responds too slowly?

Answer: sampling rate, value resolution and response delay solve different problems. Identify which requirement is failing.

Check: the physical system

A 12-bit ADC has 4096 codes. Does that guarantee a more accurate sensor? No: accuracy also depends on sensor and reference errors.

A 30 ms pulse lies between samples 100 ms apart. Can it be missed? Yes.

Is an LED blinking on a schedule feedback control? No: the sketch does not measure whether its light reached a target.

Does a sensor value of 600 mean 600 °C? No: the unit and conversion rule are missing.

Describing behavior in code

A decision chooses a path.

A loop repeats a path while its rule allows it.

An if or while condition treats zero as false and a nonzero value as true.

Comparison operators produce 0 or 1.

`temperature >= 30` compares a value with a threshold.

Braces group the statements controlled by a decision or a loop.

Comparisons

Operator	Meaning	Example when x is 30
== and !=	Equal / not equal	x == 30 is true
< and >	Strictly less / greater	x > 30 is false
<= and >=	Include the boundary	x >= 30 is true

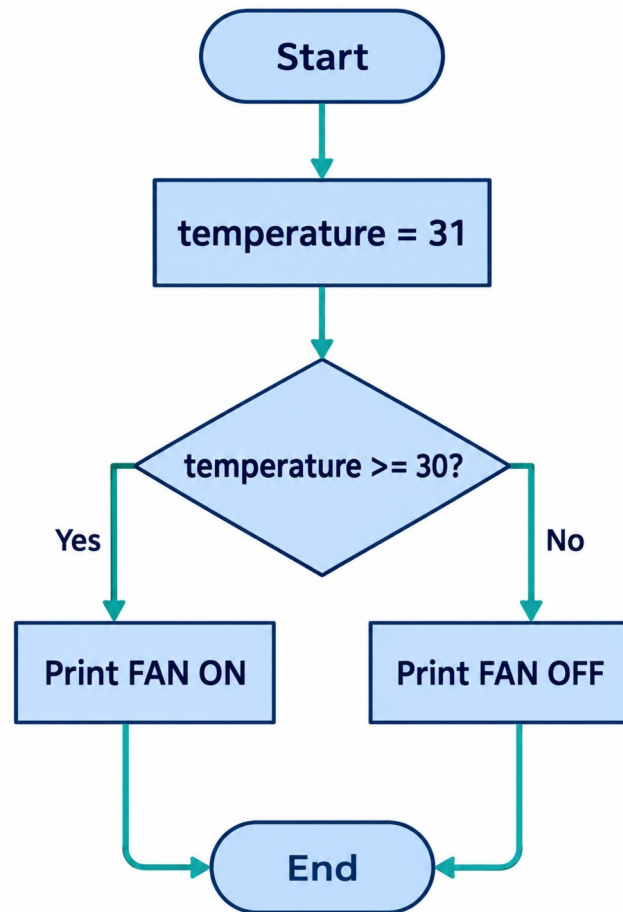
The boundary value is a test case. “At least 30” means ≥ 30 .

Combining conditions

Operator	Meaning	Example
&&	Both conditions hold	<code>x >= 20 && x <= 30</code>
	At least one condition holds	<code>x < 0 x > 100</code>
!	Logical negation	<code>!ready</code>

C short-circuits && and ||: it skips the right operand when the left already determines the result.

A decision before its C syntax



The diamond states a question with two outcomes.

Each arrow label identifies the outcome.

The two branches rejoin after the action.

With temperature = 31, the Yes branch prints FAN ON. Both paths then reach End.

A complete threshold program

```
#include <stdio.h>

int main(void) {
    int temperature = 31;
    if (temperature >= 30) {
        printf("FAN ON\n");
    } else {
        printf("FAN OFF\n");
    }
    return 0;
}
```

The fixed value represents a reading.

No sensor is needed to test the decision.

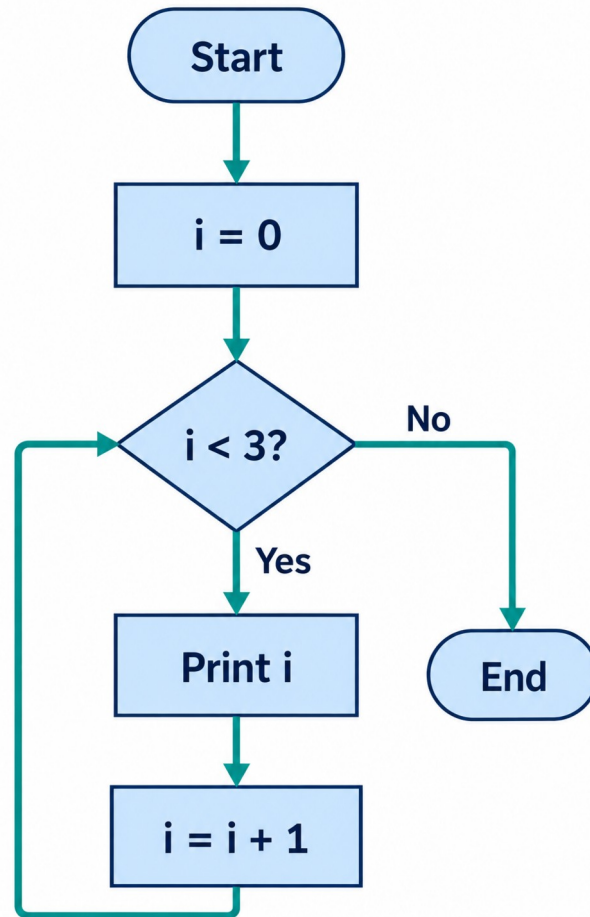
Expected output: FAN ON

Boundary tests for the decision

temperature	temperature ≥ 30	Expected output
29	False	FAN OFF
30	True	FAN ON
31	True	FAN ON

Test below, exactly at, and above the boundary. Changing $\geq$ to $>$ changes the result at 30.

A loop before its C syntax



Initialize the state.

Test before each iteration.

Update so the condition can eventually become false.

The body prints 0, 1 and 2. When i becomes 3, the No branch reaches End.

The while loop

```
#include <stdio.h>

int main(void) {
    int i = 0;
    while (i < 3) {
        printf("%d\n", i);
        i = i + 1;
    }
    return 0;
}
```

Test, execute the body, then test again.

A false initial condition means zero iterations.

The update is inside the body.

A trace includes the final failed test

i before test	$i < 3$	Action
0	True	Print 0, set i to 1
1	True	Print 1, set i to 2
2	True	Print 2, set i to 3
3	False	Leave the loop

Three body executions require four condition tests.

The same repetition with for

```
#include <stdio.h>

int main(void) {
    for (int i = 0; i < 3; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Initialization runs once.

The condition runs before each body execution.

The update runs after each completed body.

How an Arduino sketch runs

The Arduino environment builds a C++ program for the selected board.

`setup()` runs once after startup or reset.

The framework repeatedly calls `loop()`. You provide its body.

These are supplied entry points. Designing your own functions comes next week.

Blink: one ON interval and one OFF interval

```
void setup() {  
    pinMode(LED_BUILTIN, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(LED_BUILTIN, HIGH);  
    delay(500);  
    digitalWrite(LED_BUILTIN, LOW);  
    delay(500);  
}
```

500 ms ON and 500 ms OFF.

One cycle takes about one second.

The framework calls loop() again.

loop() and a loop statement

Construct	Who repeats it?	What stops?
Arduino loop()	The Arduino framework calls it repeatedly	A return ends this call, not the framework
for (int i = 0; i < 3; i++)	The C++ loop statement	Stops when $i < 3$ becomes false
while (condition)	The C/C++ loop statement	Stops when the condition becomes false

One loop() call can contain a finite loop. The names refer to different levels of control.

The cost of delay()

delay() makes a simple blink easy to read.

During that pause, the sketch does not proceed to its next input-reading statement.

The board is not entirely inactive: interrupts and some hardware functions can continue.

A system that must remain responsive needs a different timing design, such as checking elapsed time.

Preparation for the first Arduino lab

Be able to upload and explain an onboard-LED sketch.

Trace a condition and a bounded repetition before changing them.

Compare expected and observed behavior after one small change.

External buttons and additional sensors are not needed for this first result.

Brookshear and Brylow, Computer Science: An Overview, 12th Global Edition: sampling (1.4), I/O (2.5), sensors and actuators (p. 492).

Arduino documentation; Analog Devices on ADCs; TI TMP235 sensor model; NIST on physical-system timing.

Arduino Blink Without Delay: an extension on responsive timing.

Instructor Sets 4 and 6: decisions and loops. The source notes identify individual figures and examples.

Code examples and extensions

Worked examples for review and practice.
These slides are outside the core lecture sequence.

An ordered chain of decisions

```
#include <stdio.h>

int main(void) {
    int temperature = 30;
    if (temperature < 20) {
        printf("COLD\n");
    } else if (temperature < 30) {
        printf("COMFORTABLE\n");
    } else {
        printf("HOT\n");
    }
    return 0;
}
```

Only the first matching branch runs.

The second branch means $20 \leq \text{temperature} < 30$.

Independent if statements behave differently.

Two small mistakes change the meaning

```
if (x = 30) {  
    /* assigns 30, then tests 30 */  
}
```

Use == for an equality comparison.

```
if (x >= 30);  
{  
    /* this block is unconditional */  
}
```

A semicolon can create an empty statement.

Use compiler warnings and consistent braces.

Off-by-one errors

Loop condition, starting at $i = 0$	Values printed	Count
$i < 3$	0, 1, 2	3
$i \leq 3$	0, 1, 2, 3	4
$i < 0$	None	0

Write the intended values before writing the comparison.

The APIs in our first sketch

Name	Purpose
LED_BUILTIN	Board-defined identifier for the built-in LED
pinMode(pin, OUTPUT)	Configure a pin for output
digitalWrite(pin, HIGH / LOW)	Request a digital output level
delay(millisecods)	Pause the sketch's normal instruction sequence

UNO R3 is the reference board. HIGH turns its built-in LED on.

Reading the timing of Blink

Approximate time	Action	LED state afterward
0 ms	Write HIGH	ON
500 ms	Write LOW	OFF
1000 ms	Next loop() call writes HIGH	ON
1500 ms	Write LOW	OFF

The delays dominate this example. Instructions and framework work also take a small amount of time.

A condition can select the blink rate

```
const int reading = 31;

void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
    int pauseMs = 700;
    if (reading >= 30) {
        pauseMs = 200;
    }
    digitalWrite(LED_BUILTIN, HIGH);
    delay(pauseMs);
    digitalWrite(LED_BUILTIN, LOW);
    delay(pauseMs);
}
```

reading is synthetic test data.

31 gives a 400 ms cycle.

29 gives a 1400 ms cycle.

A bounded burst inside loop()

```
void setup() {  
    pinMode(LED_BUILTIN, OUTPUT);  
}  
  
void loop() {  
    for (int i = 0; i < 3; i++) {  
        digitalWrite(LED_BUILTIN, HIGH);  
        delay(200);  
        digitalWrite(LED_BUILTIN, LOW);  
        delay(200);  
    }  
    delay(1000);  
}
```

Three short flashes, then a longer pause.

The for loop stops after three iterations.

The next loop() call repeats the whole burst.

The same burst logic on a PC

```
#include <stdio.h>

int main(void) {
    for (int i = 0; i < 3; i++) {
        printf("LED ON\n");
        printf("LED OFF\n");
    }
    printf("PAUSE\n");
    return 0;
}
```

This models the order of actions.

It does not reproduce real timing or voltage.

Use it to trace repetition without a board.

Optional: a button input

```
// Inside setup(), with suitable wiring:  
pinMode(2, INPUT_PULLUP);  
  
// Inside loop():  
int pressed = (digitalRead(2) == LOW);  
if (pressed) {  
    digitalWrite(LED_BUILTIN, HIGH);  
} else {  
    digitalWrite(LED_BUILTIN, LOW);  
}
```

A switch connects pin 2 to GND.

The pull-up makes an open switch read HIGH.

Pressed reads LOW.
Mechanical contacts can bounce.

Optional: do ... while

```
#include <stdio.h>

int main(void) {
    int i = 3;
    do {
        printf("%d\n", i);
        i++;
    } while (i < 3);
    return 0;
}
```

Prints 3 once.

The first test happens after the body.

Notice the semicolon after while (...).

Optional: switch selects a discrete case

```
#include <stdio.h>

int main(void) {
    int mode = 1;
    switch (mode) {
        case 0: printf("OFF\n"); break;
        case 1: printf("SLOW\n"); break;
        case 2: printf("FAST\n"); break;
        default: printf("INVALID\n");
    }
    return 0;
}
```

Each case names a constant value.

break leaves this switch.

Without break, execution may continue into later cases.

Optional: remembering a state

```
// fanOn already holds the previous state.  
if (temperature >= 30) {  
    fanOn = 1;  
} else if (temperature <= 28) {  
    fanOn = 0;  
}  
// In between, leave fanOn unchanged.
```

Initial state must be defined.

Starting OFF: 29, 31, 29, 27
gives OFF, ON, ON, OFF.

State lets the same reading
produce different actions
depending on history.