ZENITH STUDY HUB

Microprocessor & Micro-controller

UNIT V

Applications of Microcontrollers in Robotics & Automation

From a sensor reading to a safe, repeatable robot action

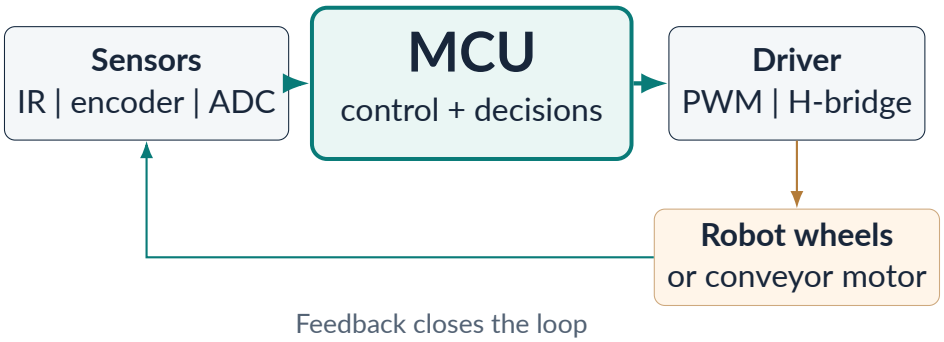

Figure 1. Original conceptual illustration: the MCU processes sensor data and commands a protected driver; it does not directly power a motor.

Official syllabus coverage

1. Microcontrollers in Robotics.
2. Microcontroller-Based Robot Control.
3. Sensor Interfacing for Robots.
4. Microcontroller-Based Automation.

Study path: read the concept → examine the labelled figure → work the numerical → solve the self-check.

Course: PCC302-ROA | TE Robotics & Automation | 2024 Pattern

Prepared for learning review: Om Deshmukh | 21 November 2026

Start here: the four questions this unit answers

Syllabus topic	Question you should be able to answer
Microcontrollers in robotics	What does the embedded controller do inside a robot, and why is a motor driver necessary?
Robot control	How do sensors, timers, PWM and feedback make a wheel or arm follow a target?
Sensor interfacing	How does an actual voltage, pulse or digital level become a trustworthy control input?
Microcontroller-based automation	How do conveyors and sorting machines sequence actions, handle faults and restart safely?

How to recognize a technically complete answer

A strong answer defines the term, identifies the MCU peripheral involved, draws a labelled signal-flow diagram, explains a step-by-step operating sequence, solves a simple numerical when applicable, and names an important failure or safety limitation.

Do not confuse a teaching example with a deployment schematic. Motor current, grounding, logic levels, encoder input thresholds, overcurrent protection, emergency-stop circuits, and pin assignments depend on the chosen hardware. The exercises show control principles, not certified wiring or safety designs.

Contents

1 Microcontrollers in Robotics 3

2 Microcontroller-Based Robot Control 6

3 Sensor Interfacing for Robots 12

4 Microcontroller-Based Automation 16

5 Exam Workshop: solved numericals and answer patterns 21

6 Rapid Revision and Troubleshooting 24

1 Microcontrollers in Robotics

In simple language
A microcontroller is the robot's local decision-making element. It repeatedly reads the world through sensors, compares readings with the desired task, and generates commands to actuators through protected interface circuits. Its peripherals let it perform this work at predictable time intervals.

1.1 Where a microcontroller fits in a robot

A useful robot can be separated into six interacting layers:

1. **Power:** battery or supply, protection, voltage regulation and distribution.
2. **Sensing:** limit switches, encoders, proximity sensors, analog sensors and serial modules.
3. **Input electronics:** filtering, pull-ups, level shifting, ADC or digital capture.
4. **Controller:** CPU, flash program, RAM, timer, interrupt, communication and watchdog.
5. **Output electronics:** transistor driver, power H-bridge, stepper driver or isolated relay output.
6. **Physical motion:** DC motor, stepper, servo, valve or gripper; motion changes sensor signals.

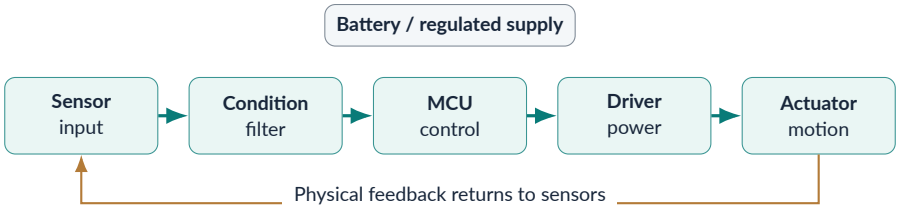

Figure 2. Complete robot signal chain with feedback. Each arrow indicates an interface requirement, not necessarily a direct wire.

1.2 What the MCU peripherals do

MCU part	Robotics purpose
GPIO	Read a switch; drive a low-power logic control line after protection
Timer/counter	Schedule periodic control updates; measure pulse spacing or intervals
PWM	Command an external motor driver or controlled power stage
ADC	Measure a voltage from a conditioned analog sensor
External interrupt	Notice selected edges such as an encoder event or limit transition
UART/SPI/I^2C	Exchange data with a sensor, display, another controller or host
Watchdog	Help recover from software that stops making expected progress
Nonvolatile memory	Keep program or calibrated parameters across power cycles

1.3 8051, PIC18F458 and modern controllers: do not mix device capabilities

The original classic 8051 has GPIO, two timers and serial communication but *no built-in ADC and no dedicated PWM peripheral*; these functions need external hardware or carefully designed software. The syllabus PIC18F458 provides a 10-bit ADC and hardware CCP PWM modes. Both can implement robot control when the external electronics and timing requirements fit. A newer MCU may integrate more features; check the exact part-number datasheet rather than borrowing register names from a similar part.

An exam-grade distinction: A PWM pin is a *command signal*; the driver supplies motor current. The commanded duty cycle is not the measured speed, and a watchdog is not a substitute for a hardware emergency stop.

1.4 Open-loop versus closed-loop operation

In **open-loop control**, the MCU sends a command without observing the resulting motion. Example: run a conveyor at fixed PWM for two seconds. In **closed-loop control**, a sensor reports the actual outcome and the software updates its command. Example: use an encoder to keep wheel speed near 100 rpm as battery voltage changes.

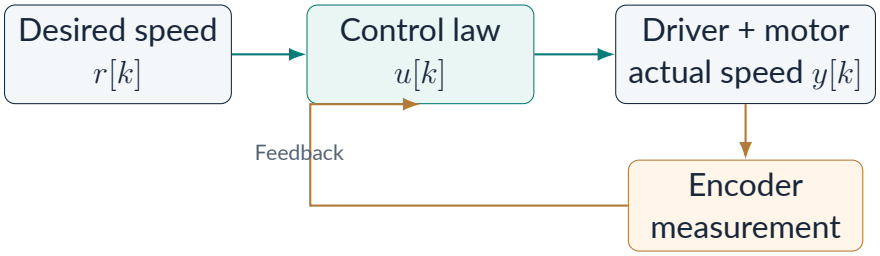

Figure 3. Conceptual closed-loop speed control. The sign of error is $e[k] = r[k] - y[k]$.

WORKED EXAMPLE: Choose the controller architecture

A simple lamp needs one digital output. An obstacle-avoiding rover needs multiple inputs, timers and two protected motor commands. A balancing robot additionally needs fast inertial measurement and low-latency feedback. More CPU speed alone does not guarantee better balance: sensor quality, control period, timing jitter and actuator dynamics also matter.

1.5 Real-time behaviour and deadline terminology

A controller is *real-time* when correctness depends on both the value of an output and *when* it is delivered. For a 10 ms periodic control loop, finishing computation in 3 ms leaves time for interrupts and other tasks. If worst-case computation takes 14 ms, the design cannot meet a strict 10 ms deadline without redesign. **Hard real-time** means a missed deadline may invalidate the operation; **soft real-time** may tolerate occasional late responses with degraded quality.

WORKED EXAMPLE: Deadline utilization

One task needs 1.5 ms every 10 ms, a second 2 ms every 20 ms, and a background monitor 4 ms every 100 ms. Their ideal CPU utilizations are $1.5/10 + 2/20 + 4/100 = 0.29$, or 29%. This is *not* proof that deadlines are schedulable: blocking, interrupt latency, non-preemptive regions and scheduler design must also be examined.

2 Microcontroller-Based Robot Control

In simple language
Robot control is the process of converting a desired motion into timed motor commands. A practical loop needs a target, measurement, error calculation, control algorithm, output saturation and another measurement.

2.1 Typical wheel-speed control sequence

1. Read an encoder counter at a fixed interval T_s.
2. Subtract the previous count to obtain pulse change ΔN.
3. Convert counts to wheel revolutions or angular speed with the correct encoder definition.
4. Calculate $e[k] = \omega_{\text{target}} - \omega_{\text{measured}}$.
5. Update a control law such as proportional control or PI.
6. Limit output to a safe command and send PWM/direction to the driver.
7. Monitor current, supply voltage, faults and timeout conditions; repeat.

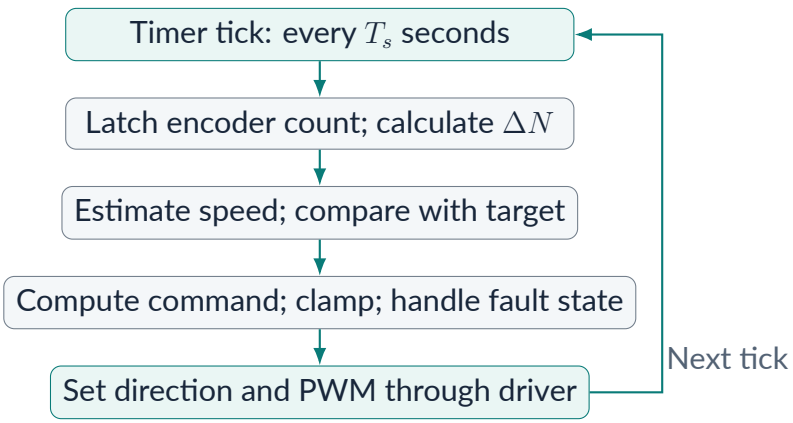

Figure 4. Periodic sampled-data controller. Use a fixed measured T_s, not an assumed software-delay duration.

2.2 Motors and driver types

Actuator	Method and limitation
Brushed DC motor	H-bridge reverses polarity; PWM commands a driver. Load, voltage and current affect speed.
Stepper motor	Step/direction interface to a suitable driver. Open-loop steps may be lost under excessive load.
Position servo	Use the device's specified pulse or digital interface. Internal feedback and permitted range vary.
Brushless motor	Use a dedicated electronic commutator or appropriate inverter. MCU GPIO must never drive windings directly.
Solenoid or relay	Use a transistor/relay driver, protection from inductive flyback and appropriate power sizing.

2.3 A brushed DC motor and H-bridge

Four controlled switches form two legs. Turning on one diagonal pair places one polarity across the motor; the other diagonal reverses it. Real driver ICs add dead time or shoot-through prevention, flyback handling and fault protection. **Never intentionally close both high-side and low-side switches in the same leg.**

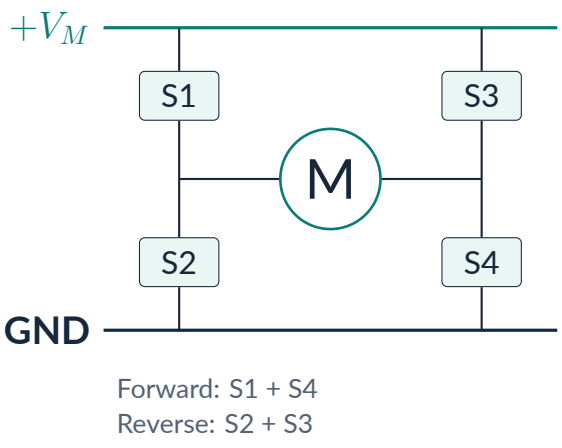

Figure 5. Conceptual H-bridge; a real driver needs safe switching and motor-current protection.

WORKED EXAMPLE: How duty cycle relates to ON time

A PWM signal runs at 5 kHz with commanded duty $D = 0.35$. The period is $T = 1/5000 = 200\,\mu$s; the nominal ON time is $T_{on} = DT = 70\,\mu$s and OFF time is $130\,\mu$s. The command requests 35% ON time; it does not guarantee the wheel runs at 35% of its free speed.

2.4 PWM signal, clearly distinguished from motor speed

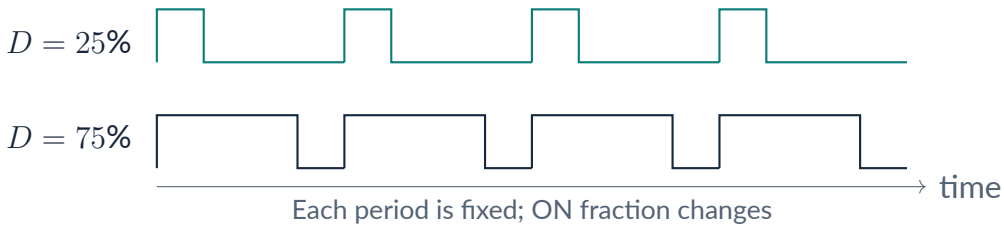

Figure 6. Two idealized PWM waveforms. Hardware timing, driver topology and load determine the actual motor current and response.

2.5 Encoder pulse counts, quadrature and speed

An incremental encoder generates transitions as the shaft turns. With quadrature channels A and B, their 90° electrical phase relationship allows direction estimation. Depending on the decoder, one counts one, two or four edges per fundamental encoder cycle ($\times 1$, $\times 2$ or $\times 4$). Always define whether your encoder's PPR means periods, edges, or gearbox output counts.

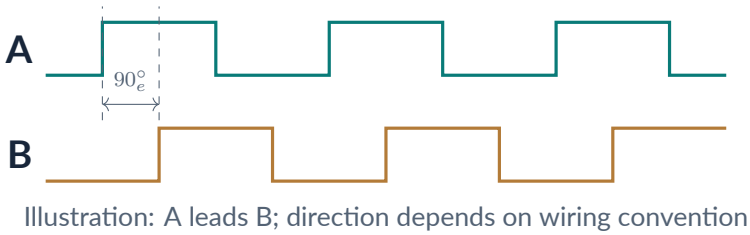

Figure 7. Original two-channel quadrature timing. Read actual decoder phase conventions instead of assuming clockwise or anticlockwise labels.

WORKED EXAMPLE: Wheel speed from an encoder

An encoder is specified as 300 fundamental A-channel cycles per wheel revolution, and the MCU performs $\times 4$ quadrature decoding. Thus $C = 300 \cdot 4 = 1200$ counts per wheel revolution. If $\Delta N = 240$ counts arrive in $T_s = 0.100$ s, revolutions per second $= 240/(1200 \cdot 0.1) = 2$; speed $= 120$ rpm, and angular speed $= 4\pi \approx 12.57$ rad/s. **Note:** These counts refer to the *wheel*; a motor-shaft encoder before a gearbox needs the gear ratio.

WORKED EXAMPLE: Correcting for a gearbox

A shaft encoder produces 40 cycles per motor revolution, $\times 4$ counting is used, and the gear reduction is 30 motor revolutions per wheel revolution. Counts per wheel turn $= 40 \cdot 4 \cdot 30 = 4800$. A 480-count interval therefore represents $480/4800 = 0.1$ wheel revolution.

2.6 Discrete proportional and PI speed control

At sample k, define error $e[k] = r[k] - y[k]$. A proportional controller is $u[k] = K_p e[k]$. A simple positional PI form is

$$I[k] = I[k-1] + K_i T_s e[k], \qquad u_{\text{raw}}[k] = K_p e[k] + I[k].$$

The actuator receives $u[k] = \text{clip}(u_{\text{raw}}[k], u_{\text{min}}, u_{\text{max}})$. If saturation persists and the integrator continues accumulating, *integral windup* can cause a long recovery. Basic anti-windup strategies include freezing integration when it would drive the saturated command further outward, or using back-calculation. Safety limits and reset behaviour must be specified.

WORKED EXAMPLE: One PI update

Suppose $K_p = 0.5$, $K_i = 2\,\text{s}^{-1}$, $T_s = 0.05$ s, $r = 100$ rpm, $y = 90$ rpm and previous integrator contribution $I[k-1] = 12$ command units. Error $= 10$ rpm; new contribution $I[k] = 12 + 2(0.05)(10) = 13$; raw command $= 0.5(10) + 13 = 18$ command units. If the permitted output is 0–100, no clipping is needed. **Units:** K_p and K_i must be calibrated to the chosen command/rpm units.

2.7 Differential-drive mobile robot kinematics

Two driven wheels of radius r, separated by track width L, produce wheel angular velocities ω_R and ω_L. With ideal pure rolling on a flat surface,

$$v = \frac{r}{2}(\omega_R + \omega_L), \qquad \Omega = \frac{r}{L}(\omega_R - \omega_L).$$

Here v is forward speed in m/s and Ω is yaw rate in rad/s. The inverse relations are $\omega_R = (v + \Omega L/2)/r$ and $\omega_L = (v - \Omega L/2)/r$.

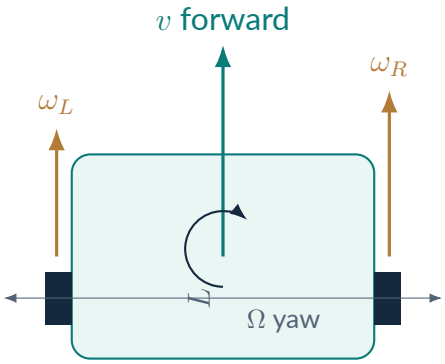

Figure 8. Top-view differential-drive schematic (not to scale). Track width is measured between wheel contact centres.

WORKED EXAMPLE: Straight movement and turning

Take $r = 0.05$ m, $L = 0.30$ m, $\omega_R = 12$ rad/s and $\omega_L = 8$ rad/s. Forward speed is $0.05(12 + 8)/2 = 0.50$ m/s; yaw rate is $0.05(12 - 8)/0.30 = 0.6667$ rad/s. If both wheel rates are equal, the ideal yaw rate is zero. This result neglects slipping and compliance.

2.8 Servo command pulse timing: not the same as motor PWM

Many hobby position servos accept a repeating pulse train whose width approximately indicates target angle. *The permitted width, repetition interval, direction and angular range depend on the specific servo.* A commonly illustrated convention is a pulse around 1–2 ms repeated near 20 ms; do not treat it as universal or drive a high-current servo from the MCU supply pin.

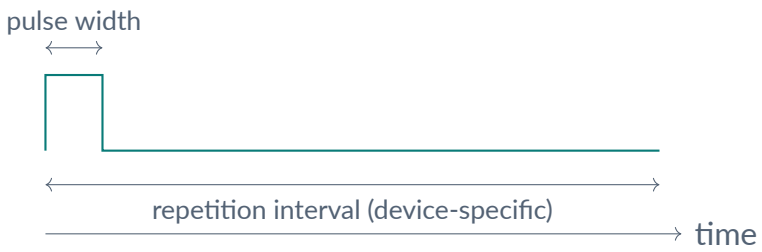

Figure 9. Servo-command pulse-width concept; values are examples, not a universal calibration.

2.9 Timing architecture: interrupts, DMA and periodic scheduling

An encoder edge may trigger a counter or interrupt; a periodic hardware timer should trigger the speed-control update. Short ISRs can capture timestamps and update counters; slower calculations can execute in an appropriately scheduled task. Avoid blocking delay loops inside interrupts and share multibyte counters safely (atomic access or interrupt masking when necessary).

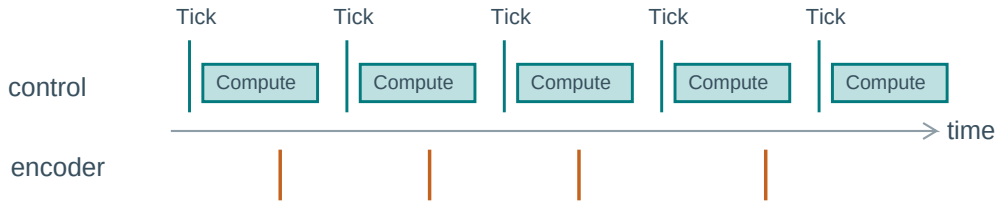

Figure 10. Idealized periodic control ticks and asynchronous encoder transitions. Real execution and latency must be measured.

2.10 Practical control pseudocode (not tested or compiled)

```
// Periodic 20 ms callback (illustrative pseudocode)
void control_tick(void) {
  int32_t now = atomic_encoder_snapshot();
  int32_t delta = now - previous;
  previous = now;
  float speed_rps = delta / (COUNTS_PER_REV * 0.020f);
  float error = target_rps - speed_rps;
  float command = PI_update_with_antiwindup(error, 0.020f);
  if (fault_active() || command_timeout()) {
    driver_disable();  // software layer, not safety-rated
  } else {
    driver_set_bounded_command(command);
  }
}
```

Firmware readiness checklist: Choose an actual MCU and peripherals; prove counter atomics; implement the hardware driver truth table; specify safe states and resets; compile with the chosen toolchain; test timing and current on the real board. The example deliberately omits hardware-specific register writes.

3 Sensor Interfacing for Robots

In simple language
Interfacing means making the sensor's physical output *electrically compatible*, *measurable*, and *trustworthy* before the MCU decides what to do. A sensor is more than a function returning a number.

3.1 Choose the right input path

Sensor and output	MCU interface and important condition
Pushbutton / limit contact (digital)	GPIO input with bias, debounce and input protection; certified safety signals need separate design.
Optical / proximity switch (digital or industrial voltage)	Protected GPIO and level shifting or isolation where sensor output exceeds logic ratings.
Potentiometer / analog range sensor (voltage)	Analog conditioning and filter, then compatible ADC with known reference.
Hall-effect pulse sensor (logic edges)	Counter or timer capture, or bounded-load GPIO interrupt.
Quadrature encoder (phase A/B)	Hardware quadrature decoder or carefully specified interrupt/counter logic.
Inertial measurement unit, IMU (digital packets / analog axes)	I^2C, SPI or ADC according to sensor; scale and calibrate axes.
Temperature sensor (voltage / resistance / packets)	Excitation and conditioning plus ADC, or a supported digital interface.

3.2 Power, grounding and logic levels first

1. Compare sensor output voltage range with MCU absolute maximum and valid input thresholds.

2. Give analog sensors a suitable reference, decoupling, and return path.
3. Use a level shifter, divider, buffer or optocoupler where appropriate. A plain resistor divider may be unsuitable for bidirectional or fast buses.
4. Protect inputs against transients, reverse polarity and inductive environments.
5. Separate high-current motor return paths from the sensitive sensor reference as required by the hardware layout.

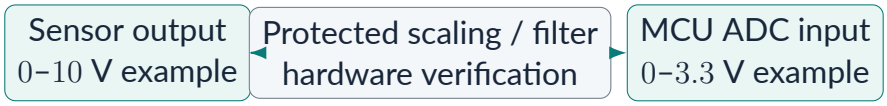

Figure 11. Analog signal-conditioning concept. Voltage ranges are illustrative, not device-specific allowable limits.

3.3 Analog-to-digital conversion and code width

For an ideal N-bit ADC with reference span $V_{\text{span}} = V_{\text{ref}+} - V_{\text{ref}-}$, the quantization-bin width is approximately

$$\Delta V = \frac{V_{\text{span}}}{2^N}.$$

For an ideal code-to-voltage endpoint interpolation, $V \approx V_{\text{ref}-} + (\text{code}/(2^N - 1))V_{\text{span}}$ is often used. They are *different stated models*. Practical accuracy also depends on offset, gain error, reference noise, input settling and nonlinearity.

WORKED EXAMPLE: 10-bit ADC with 3.3 V span

$2^{10} = 1024$ codes; bin width $= 3.3/1024 = 0.00322265625\,\text{V} = 3.223\,\text{mV}$. With endpoint mapping and code 512, estimated voltage $= 512(3.3)/1023 \approx 1.652\,\text{V}$. The ADC is not magically accurate to 3.223 mV; that is an ideal quantization *step*. The input must stay inside the device's permitted limits.

WORKED EXAMPLE: Voltage divider range selection

A 0–$10\,\text{V}$ sensor needs a maximum nominal scale factor of at most $3.3/10 = 0.33$ for a 0–$3.3\,\text{V}$ ADC. A resistor divider with $R_{\text{top}} = 20\,\text{k}\Omega$ and $R_{\text{bottom}} = 10\,\text{k}\Omega$ gives $V_{\text{out}} = V_{\text{in}}/3$, so 10 V becomes 3.333 V: **slightly above 3.3 V and therefore unacceptable without additional margin/protection**. Change the ratio and check resistor tolerance, source impedance and input transients; never assume a divider alone is a robust industrial interface.

3.4 Sampling rate and aliasing

A sample rate f_s can represent sinusoidal content below the Nyquist frequency $f_s/2$ under appropriate sampling and bandlimiting assumptions. Frequencies higher than that may *alias* to misleading lower frequencies. Physical low-pass filtering before the ADC is usually required; digital filtering after sampling cannot remove an alias that has already folded into the passband.

WORKED EXAMPLE: An aliasing example

A pure 70 Hz interference sampled at 100 Hz may appear at $|70 - 100| = 30\,\text{Hz}$ in the sampled sequence. This is not a real 30 Hz vibration. Reduce out-of-band interference with analog filtering or choose a faster and appropriately conditioned sample rate. For motor-control loops also consider delay, latency, sensor noise and mechanical bandwidth.

3.5 Switch debounce and edge-based sensing

A mechanical contact can bounce several times during a single press. If the firmware counts every transition, one press may appear as many events. In a time-based debounce scheme, accept a new state only after it has remained stable for a chosen window. The window is application-specific; it must not hide a required fast protective event. Safety-rated limit switches are handled by a separate safety architecture.

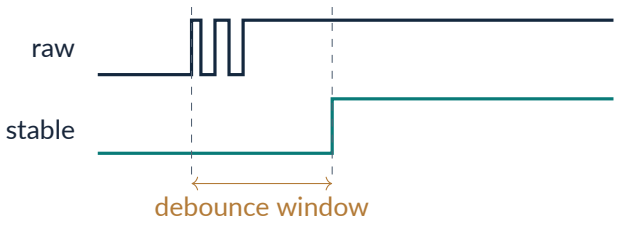

Figure 12. Illustrative button bounce and a stability-based acceptance scheme; not a safety-

stop implementation.

3.6 Encoder interrupts and counter integrity

At high edge rates, a software ISR for every edge may overrun the CPU. Prefer a hardware timer/encoder decoder when available, or calculate worst-case interrupt load. When the counter is wider than the MCU's atomic read width, create a consistent snapshot; otherwise a race condition can produce a corrupted measurement. An encoder also needs noise-immune wiring and correct input thresholds.

WORKED EXAMPLE: Interrupt load budget

A quadrature encoder generates 4,800 counts per wheel revolution in $\times 4$ mode. At 300 rpm $= 5$ revolutions/s, the worst nominal event rate is $4{,}800 \cdot 5 = 24{,}000$ events/s. An ISR costing $12\ \mu$s per event would consume $24{,}000 \cdot 12\ \mu\text{s} = 0.288$ s of CPU time per second, or **28.8%**. This is before other work or signal noise; it strongly motivates a hardware counter or more efficient design.

3.7 Calibration, scaling and uncertainty

A sensor may have an *offset* (zero error), *gain* error (wrong slope), nonlinear response and temperature drift. For a simple linear calibration with two known measurements (x_1, y_1) and (x_2, y_2), the corrected physical estimate can be modelled as $y = ax + b$ with $a = (y_2 - y_1)/(x_2 - x_1)$ and $b = y_1 - ax_1$. This is a model, not a substitute for a real calibration certificate or uncertainty analysis.

WORKED EXAMPLE: Two-point calibration

An ADC system reports raw codes 100 and 900 at physical positions 0 mm and 80 mm. Scale $a = (80 - 0)/(900 - 100) = 0.1$ mm/code and intercept $b = -10$ mm. At code 550 the estimated position is $0.1(550) - 10 = 45$ mm. The model assumes the sensor is sufficiently linear between the reference points.

3.8 A robust sensor-read pseudocode

```
// Pseudocode: details vary by device
sensor_sample() {
  configure_reference_and_input_if_needed();
  select_ADC_channel();
  wait_for_input_acquisition();
  start_ADC_conversion();
  wait_until_conversion_done_or_timeout();
  raw = read_ADC_result_consistently();
  if (!range_plausible(raw)) { set_sensor_fault(); return; }
  filtered = application_filter(raw);
  physical = calibration_map(filtered);
  publish_with_timestamp_and_quality(physical);
}
```

Check your understanding
Why must an ADC input settle before conversion? Why is a low-pass filter *before* sampling more important for alias prevention than one placed only in code? Why is a time-stamped measurement preferable when a robot changes speed quickly?

4 Microcontroller-Based Automation

In simple language
Automation is a repeatable sequence of actions driven by sensors, interlocks and timing rules. An MCU can implement the logic, but the industrial machine also needs properly rated power electronics, fault detection and independent protective measures.

4.1 Basic automation system

A small conveyor sorting station may contain an object-detection sensor, conveyor motor, pneumatic diverter or servo, item counter and indicator. The controller must decide *when* to stop or divert, not just whether an object is present. A good sequence declares explicit states so that the outputs are predictable even during power-up and faults.

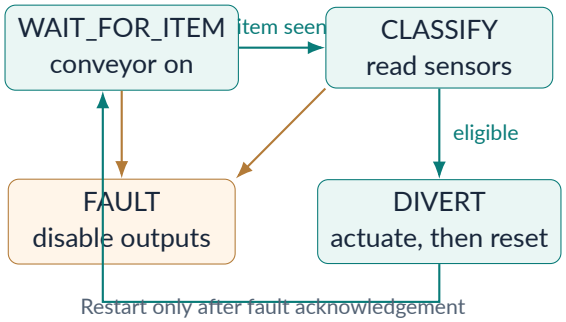

Figure 13. Simplified conveyor sorting finite-state machine. Return transitions require specified guards and timing; not a complete industrial safety logic diagram.

4.2 Finite-state machines: the safest way to explain a sequence

A finite-state machine (FSM) uses a named *state*, input *conditions*, an explicit *transition*, and outputs determined by the current state. Example states: `IDLE`, `RUN`, `DETECT`, `SORT`, `FAULT`. In the `FAULT` state, outputs must be selected according to a risk assessment: *motor-off* is common but not automatically safe for vertical loads or spring-loaded mechanisms.

State	Condition / action	Next state
IDLE	Start request, guards valid	RUN
RUN	Move conveyor; item sensor active	DETECT
DETECT	Sample classifier; valid item	SORT or RUN
SORT	Issue timed diverter command; detect completion	RUN
Any operational state	Overcurrent, guard error or timeout	FAULT
FAULT	Disable relevant commands; log fault; require deliberate reset	IDLE after safe acknowledgement

4.3 Input debouncing versus interlocking

Debouncing removes contact chatter in ordinary logical inputs. *Interlocking* prevents an unsafe or incompatible action, such as commanding conveyor motion while a guarded access panel is open. Safety interlocks may require independent rated hardware and standards compliance; a few lines of MCU code alone are not sufficient for certified machinery protection.

4.4 Automated traffic-barrier case study

Inputs: vehicle sensor, fully-open switch, fully-closed switch, manual request and fault input. **Outputs:** motor driver command, status lamp and alarm. A sensible FSM separates `CLOSED`, `OPENING`, `OPEN`, `CLOSING`, `FAULT`. If an obstruction is detected, software may request stop or reverse according to the validated system requirements. A final product needs hardware force limits and applicable safeguards.

4.5 Batch filling and a timed valve

A filling station may read a container-present sensor, measure mass or flow, and command a valve. Open-loop timing can be inaccurate because flow changes with pressure and temperature. Closed-loop filling reads a calibrated sensor, reduces flow near the target and stops at a bounded threshold; residual liquid

in the pipe causes overshoot. This illustrates the difference between *repeatable sequencing* and accurate *feedback control*.

WORKED EXAMPLE: Conveyor throughput

A belt moves at 0.25 m/s and objects arrive approximately every 0.50 m along the belt. Ideal arrival interval $= 0.50/0.25 = 2$ s, or 30 objects per minute. If processing takes 2.4 s per object in a single-station machine, the station is the bottleneck at at most 25 objects/min in steady state. Buffering, parallelism and real transient behaviour would require further modelling.

4.6 Event scheduling, timers and polling

A simple loop that repeatedly waits 500 ms can miss short sensor pulses. Instead, a timer-driven scheduler maintains elapsed ticks and small non-blocking tasks. Input-capture hardware can timestamp fast events. Polling remains reasonable for slow signals with known minimum pulse widths; it is not inherently wrong, but requires an explicit responsiveness bound.

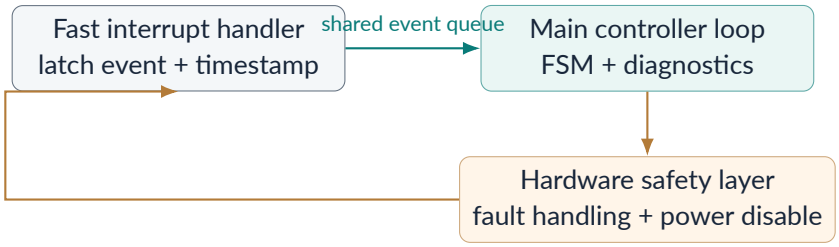

Figure 14. Separate timing-critical event capture, application logic, and independent protection. Interconnections are conceptual.

4.7 Watchdogs, brownout and restart policy

A hardware watchdog monitors program progress and may reset the MCU if it is not serviced correctly. Brownout detection handles insufficient supply voltage. Neither determines whether restarting a motor is safe; the firmware must define an intentional reset state, clear commands, record fault cause when feasible, and respect a qualified restart policy. Never make a motor begin moving automatically merely because the MCU rebooted.

Safety boundary: Emergency stop, protective guarding and fault tolerance require a hazard assessment and appropriately rated hardware. Teaching

pseudocode is not a safety-certified controller. A stuck output transistor or welded contact can defeat an MCU-only disable command.

4.8 Integrated case study: line-following warehouse cart

A line-following cart detects left and right reflectance sensors, controls two motors and uses front proximity detection to reduce speed. A better architecture separates the fast wheel-speed loop (periodic), slower navigation decisions (state machine), and fast fault monitoring (hardware + short software path). Example actions:

1. Sample and calibrate reflectance sensors; determine whether the line is left or right.
2. Convert lateral line error into a desired yaw or left-right speed difference.
3. Estimate each wheel's actual speed from encoder counts.
4. Update per-wheel PWM commands through motor drivers with limits and anti-windup.
5. If an obstruction is measured, enter a guarded stop state; only resume under defined conditions.

WORKED EXAMPLE: Simple steering command with proportional line error

Let normalized line error $e_\ell = +0.20$ mean the line is to the cart's right. Let base wheel speed be 100 rpm and steering gain $K = 50$ rpm/unit error. Set the left speed target $r_L = 100 + Ke_\ell = 110$ rpm and right target $r_R = 100 - Ke_\ell = 90$ rpm. This produces a right-turn tendency under the chosen wheel and sign convention. Actual turning depends on slip and each wheel's speed-control response.

4.9 Automation state-machine pseudocode

```
// Educational pseudocode; outputs need hardware drivers.
enum State { IDLE, RUN, DETECT, SORT, FAULT };
State state = IDLE;
void update_state(void) {
  if (critical_fault()) state = FAULT;
  switch (state) {
    case IDLE:
      all_outputs_to_defined_safe_state();
      if (start_requested() && guards_valid()) state = RUN;
```

```
      break;
    case RUN:
      conveyor_enable();
      if (item_present()) state = DETECT;
      break;
    case DETECT:
      if (!classification_valid()) state = FAULT;
      else if (divert_required()) state = SORT;
      else state = RUN;
      break;
    case SORT:
      command_bounded_divert();
      if (diverter_done()) state = RUN;
      if (diverter_timeout()) state = FAULT;
      break;
    case FAULT:
      stop_commands_and_report();
      // Physical protective circuits operate independently.
      if (authorized_reset() && all_guards_valid()) state = IDLE;
      break;
  }
}
```

4.10 Engineering verification: how to test a controller

A strong test plan includes unit checks for scaling and state logic; integration checks with simulated sensors and a motor driver; stress tests of interrupts and timing; fault injection (sensor open-circuit, stuck contact, power loss, stalled motor, communications timeout); and finally hardware tests with current limiting, guarded motion and supervised safety arrangements. Measure worst-case latency instead of relying only on average loop time.

5 Exam Workshop: solved numericals and answer patterns

5.1 A five-mark answer structure

For *microcontroller-based robot control*: draw sensor → signal interface → MCU → motor driver → motor → feedback; define closed loop; explain fixed-period sampling and compare; give one encoder or PWM numerical; state safety and interface limitations. For *sensor interfacing*: classify digital/analog/pulse interfaces, mention signal conditioning, calculate ADC resolution, show a flow dia-

gram, and discuss noise/calibration. For *automation*: draw a clear FSM, identify input/output conditions, timers, interlocks and fault transitions.

5.2 Further worked numerical: wheel distance

A wheel has radius 0.04 m and 1,000 verified counts per revolution after decoding. A move produces 2,500 counts. Revolutions $= 2.5$; ideal travel $= 2\pi(0.04)(2.5) = 0.2\pi \approx 0.6283$ m. Encoder-derived travel is *odometry*; slip and wheel-radius error affect physical distance.

5.3 Further worked numerical: sample latency

A controller samples every 20 ms, with a processing latency of at most 3 ms. An event arriving just after a sample may wait almost 20 ms before measurement and then 3 ms for processing, giving approximately **23 ms worst-case software response** under these idealized assumptions. Additional sensor/filter/driver delays and scheduling may increase it. A truly protective event needs a dedicated risk-based architecture.

5.4 Further worked numerical: timer tick

On a classic 12-clock 8051 at 11.0592 MHz, nominal timer rate when incrementing per machine cycle is $11{,}059{,}200/12 = 921{,}600$ ticks/s, so one tick is approximately $1.08507\,\mu$s. For 10,000 ticks the ideal timer interval is about 10.8507 ms. ISR latency, timer setup and clock tolerance must be considered separately; other 8051 variants may have different clock ratios.

5.5 Further worked numerical: current and motor power

A 12 V motor draws 0.80 A during an example steady state. Electrical input power $P = VI = 9.6$ W. This is not mechanical output power and not a statement about startup/stall current. Driver thermal design must consider current profiles, power losses and worst-case stall scenarios from the actual datasheet.

5.6 Further worked numerical: sensor scaling

A sensor maps 0–100 mm to 0.5–2.5 V ideally. The span is 2 V; at 1.7 V the fraction is $(1.7 - 0.5)/(2.5 - 0.5) = 0.60$, so estimated distance is 60 mm under

the assumed linear model. State the offset explicitly; incorrectly multiplying $1.7/2.5$ by 100 would ignore the nonzero zero-point.

5.7 Self-check questions, aligned to syllabus

1. Sketch the blocks of an MCU-controlled mobile robot, labeling the driver and feedback.
2. Differentiate open-loop and closed-loop motor control, with a practical example for each.
3. Explain GPIO, timer, PWM, ADC, UART, watchdog and interrupt roles in a robot.
4. Explain H-bridge forward/reverse operation and the forbidden same-leg switching case.
5. An encoder yields 900 decoded counts/revolution. Find rpm from 135 counts in 0.1 s.
6. A PWM signal runs at 4 kHz and 60% duty. Find period, ON time, OFF time.
7. For a 12-bit ADC spanning 3.3 V, calculate ideal code-bin width.
8. Explain quadrature phase, $\times 4$ decoding and why gearbox ratio matters.
9. Derive forward speed and yaw rate for a differential-drive robot.
10. Explain the need for anti-aliasing, debounce and calibration.
11. Design a five-state conveyor sorter with a fault transition.
12. Compare interrupt-driven capture and polling; name one failure mode of each.
13. Explain what a watchdog can and cannot guarantee.
14. Identify three reasons a control system that works in simulation may fail on hardware.

5.8 Answer key for numerical checks

Question	Worked result
Q5: speed	$135/(900 \cdot 0.1) = 1.5$ rps $= 90$ rpm
Q6: PWM	$T = 1/4000 = 250\,\mu$s; ON $= 0.6(250) = 150\,\mu$s; OFF $= 100\,\mu$s
Q7: 12-bit ADC	$3.3/4096 \approx 0.0008057$ V $= 0.8057$ mV/bin
Q9: differential drive	$v = r(\omega_R + \omega_L)/2$, $\Omega = r(\omega_R - \omega_L)/L$

6 Rapid Revision and Troubleshooting

Problem	First engineering checks
Motor does not start	Motor power rail, driver enable, fault pin, current limit, control signal levels
Speed drifts under load	Is control open-loop? Encoder availability, gain, supply and saturation
Encoder counts jump	Input thresholds, wiring, noise, bounce, wrong edge definition, counter races
ADC readings fluctuate	Reference noise, source impedance, acquisition, grounding and filtering
Conveyor misses items	Sampling period vs pulse width, capture interrupts, FSM timing and sensor placement
Firmware intermittently freezes	Blocking ISR, buffer bounds, race, watchdog reason, supply dips
Servo moves unexpectedly	Device pulse specification, common reference, power brownout, failsafe state

The central idea of Unit V
A good embedded robotics system is not simply a microcontroller with wires. It is a well-defined physical measurement path, repeatable control timing, a protected output stage, a documented state machine and explicit

fault handling. Sensors make behaviour observable; feedback makes it correctable; testing makes it dependable.

Source, image and verification ledger

Official course scope: Savitribai Phule Pune University, TE Robotics and Automation 2024 Pattern, PCC302-ROA Unit V "Application of Microcontrollers", supplied *TE Syllabus.pdf*, printed page 17 (PDF physical page 17). The official four headings are reproduced on the cover.

Engineering sources consulted:

1. Microchip, *PIC18FXX8 Data Sheet*, DS41159E (device capabilities). https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/41159e.pdf
2. Microchip, *Motor Control Sensor Feedback Circuits*, AN894 (sensor feedback considerations). https://www.microchip.com/en-us/application-notes/an894
3. Texas Instruments, *DRV8833 Dual H-Bridge Motor Driver*, product documentation (driver and protection). https://www.ti.com/product/DRV8833
4. National Instruments, *NI Hardware Encoder Measurements: How-To Guide* (quadrature modes). https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000x1riCAA&l=en-US
5. Analog Devices, *Filter Basics: Anti-Aliasing* (sampling and analog-filter concepts). https://www.analog.com/en/resources/technical-articles/guide-to-antialiasing-filter-basics.html

Figure licensing: All numbered diagrams are independently authored TikZ vectors based on technical concepts and the cited device material. Open-licensed web photo candidates were reviewed, including Wikimedia Commons "Two hobby servo motors.jpg" by Kborer (CC0 1.0); original image bytes were unavailable from this environment, so no outside photographs were embedded. Link: https://commons.wikimedia.org/wiki/File:Two_hobby_servo_motors.jpg. No third-party book diagram was copied.

Verification boundary: The calculations and figure/page structure are programmatically checked in the accompanying QA files. The firmware examples are explicitly illustrative pseudocode; they have not been compiled or tested on a real 8051/PIC board. Real hardware requires vendor datasheets, test fixtures and a safety review.

Suggested personal learning sequence: Read Unit V sections 1–4, reproduce the circuit and state diagrams in your own notebook, solve every worked numerical without looking, and then answer the fourteen self-check questions.