UNIVERSITY OF BUEA

College of Technology

CEC405

Level 400 · Semester 1 · 4 credits

Microprocessor and Microcontroller Practicals

Lecture notes and laboratory sheets

PIC16F877A with CCS C · Arduino Uno · Proteus VSM · Web and MQTT IoT

Lecturer: Prof. Moffo Lonla Bertrand
Volume: 60 hours (Lectures 10 h, Tutorials 10 h, Practicals 40 h)
Assessment: Practical mini project (CA) 30% · Written examination 70%
Edition: 2026/2027 academic year

These notes accompany the CEC405 course outline. They are for the use of registered students of the University of Buea and may be revised during the semester.

Contents

About this course **iv**

I Lecture Notes **1**

1 Introduction to Microprocessor and Microcontroller Systems **2**

1.1 What is an embedded system? . . . 2
1.2 The microcomputer model . . . 2
1.2.1 The instruction cycle . . . 3
1.3 Microprocessor versus microcontroller . . . 3
1.3.1 The 8085 as a reference 8-bit microprocessor . . . 3
1.4 Architectures . . . 3
1.5 The two microcontrollers used in this course . . . 4
1.5.1 Microchip PIC16F877A . . . 4
1.5.2 Microchip (Atmel) ATmega328P — the Arduino Uno . . . 4
1.6 The minimum system . . . 4
1.7 From C source to a running chip: the build process . . . 5
1.7.1 The Intel HEX format . . . 6
1.7.2 Configuration bits (fuses) . . . 6
1.8 Debugging techniques . . . 6
Tutorial sheet 1 . . . 6

2 Interfacing Microcontrollers **8**

2.1 Digital input/output ports . . . 8
2.2 Connecting LEDs . . . 9
2.3 Connecting switches . . . 9
2.4 Driving loads: transistors and relays . . . 10
2.5 Seven-segment displays (reminder) . . . 10
2.6 Analogue inputs and the ADC . . . 11
2.7 The HD44780 character LCD . . . 12
2.8 The 1-Wire bus and the DS18B20 . . . 12
2.9 Serial communication: UART and RS232 . . . 14
2.10 I^2C and SPI (overview) . . . 14
2.11 Timers, PWM and servomechanisms . . . 15
2.12 Interrupts and real-time issues . . . 16
Tutorial sheet 2 . . . 16

II Laboratory Sheets **18**

1 Initiation to the CCS C Compiler and Proteus **19**
L1.1 Equipment . . . 19
L1.2 Part A — First CCS C project, compiled from Proteus . . . 19
L1.3 Part B — Proteus schematic and simulation . . . 20
L1.4 Part C — Source-level debugging . . . 21
L1.5 Part D — Exercises . . . 21
L1.6 Questions . . . 22

2 Case Study 1: Controlling LEDs with Switches **23**
L2.1 Circuit . . . 23
L2.2 Part A — Direct mirror . . . 23
L2.3 Part B — Toggle with debouncing . . . 24
L2.4 Part C — Pattern selector (state machine) . . . 24
L2.5 Questions . . . 25

3 Case Study 2: LM35 Temperature Monitoring with LEDs **27**
L3.1 Circuit . . . 27
L3.2 Program . . . 27
L3.3 Measurements . . . 29
L3.4 Exercises . . . 29
L3.5 Questions . . . 29

4 Case Study 3: DS18B20 Temperature Monitoring with an LCD **31**
L4.1 Circuit . . . 31
L4.2 Program . . . 31
L4.3 Measurements and tests . . . 34
L4.4 Exercises . . . 34
L4.5 Questions . . . 34

5 Case Study 4: Communication Between a Microcontroller and a Terminal (RS232) **36**
L5.1 Circuit . . . 36
L5.2 Part A — Command interpreter (polling) . . . 37
L5.3 Part B — Interrupt-driven reception and periodic reporting . . . 38
L5.4 Part C — Connecting the simulation to a real PC terminal . . . 38
L5.5 Questions . . . 38

6 Case Study 5: Arduino Projects in Proteus — Source Code and Flowchart **40**
L6.1 From CCS C to Arduino . . . 40

L6.2 Part A — Source code project: pedestrian crossing 41
L6.3 Part B — Flowchart project (Proteus Visual Designer) 43
L6.4 Questions 43
L6.5 Mini project brief (continuous assessment, 30%) 44

7 Case Studies 6 and 7: Web-Based IoT Project with Arduino and Proteus 46
L7.1 Background 46
L7.2 The device: Arduino firmware (used in Labs 7 and 8) 47
L7.3 Part A (week 10) — Gateway and live monitoring page 48
L7.4 Part B (week 11) — Remote control, history and automation 50
L7.5 Questions 51

8 Case Studies 8 and 9: MQTT-Based IoT Project with Arduino and Proteus 52
L8.1 Background: MQTT 52
L8.2 Part A (week 12) — Broker, bridge and first clients 53
L8.3 Part B (week 13) — Distributed control and dashboard 55
L8.4 Questions 55

A Quick Reference: CCS C and Arduino 57

B Proteus Components Used in the Labs 58

C PIC16F877A Pin Assignment (DIP-40) 59

D Lab Report Template 60

About this course

Course description

CEC405 is the practical companion of the microprocessor and microcontroller design course. You will programme real 8-bit microcontrollers (the Microchip PIC16F877A and the Atmel/Microchip ATmega328P on the Arduino Uno) in C/C++, simulate complete circuits in Proteus VSM before building them, and connect them to the outside world: switches, LEDs, sensors, displays, serial terminals and finally the Internet through Web and MQTT gateways.

The course follows the same path as an engineering project: understand the hardware, write the firmware, build it, simulate it, debug it, and only then build the real circuit.

Learning outcomes

On successful completion of this course you will be able to:

LO1 analyse the needs of an electronic application and design an appropriate microcontroller-based solution;

LO2 use the hardware units of a modern microcontroller (I/O ports, ADC, timers, UART, interrupts);

LO3 create microcontroller applications with the appropriate software tools (Proteus 9.2 VSM with the CCS C compiler and its built-in Arduino compiler, Python);

LO4 interface external devices (switches, LEDs, sensors, LCDs, PCs and networks) to a microcontroller.

Weekly plan

Week	Topic	Type	Hours	Where in these notes
1	Introduction to microprocessor and microcontroller systems	L	4	Chapter 1
2	Interfacing microcontrollers	L	4	Chapter 2
3	Initiation to the CCS C compiler and Proteus	P	4	Lab 1
4	Case study 1: LEDs controlled by switches (PIC)	P	4	Lab 2
5	Case study 2: LM35 temperature monitoring with LEDs	P	4	Lab 3
6	Case study 3: DS18B20 temperature monitoring with LCD	P	4	Lab 4
7	Case study 4: microcontroller–terminal communication (RS232)	P	4	Lab 5
8	Case study 5: Arduino in Proteus (source code and flowchart projects); mini project issued	P	4	Lab 6, § L6.5
9	Mini project assessment (CA)	P	4	Section L6.5
10	Case study 6: Web-based IoT project, part A	P	4	Lab 7
11	Case study 7: Web-based IoT project, part B	P	4	Lab 7
12	Case study 8: MQTT-based IoT project, part A	P	4	Lab 8
13	Case study 9: MQTT-based IoT project, part B	P	4	Lab 8

The 10 hours of tutorials are covered by the tutorial sheets at the end of Chapters 1 and 2 and by the review questions of each lab.

Assessment

- **Continuous assessment (30%)**: practical mini project, issued in week 8 and assessed in week 9 (Section L6.5). Lab sheets are checked at the end of each session; a missing lab sheet may reduce the CA mark.
- **Written examination (70%)**: covers the lecture chapters, the tutorial sheets and the theory behind every lab (calculations, code reading, code writing, interfacing diagrams).

Software and hardware

Tool	Use
Proteus 9.2 SP1 Professional (Labcenter)	Schematic capture, VSM simulation and debugging; firmware projects compiled from inside Proteus; built-in Arduino compiler and Visual Designer (Labs 6–8)
CCS C Compiler (PCWHD or PCM, v5)	C compiler for the PIC (Labs 1–5); installed on the same PC and called by Proteus
Arduino IDE 2.x (optional)	Only to upload sketches to a real Arduino board
Python 3.10+ with `pyserial`, `flask`, `paho-mqtt`	PC-side gateway for the IoT labs
com0com (Windows)	Virtual null-modem COM port pair linking Proteus to the PC
Mosquitto	MQTT broker (Labs 8)
Hardware (when available)	PIC16F877A board or breadboard, PICkit 3/4, Arduino Uno, LM35, DS18B20, 16×2 LCD, LEDs, push-buttons, resistors, USB–serial adapter

Copying the programs

The listings are numbered so that the programs can be discussed line by line in class. Every complete program carries its file name above the listing (*File: Lab1_blink.c*, ...). These files are attached to this PDF: open the attachments panel of your PDF reader (paper-clip icon in Adobe Reader, Edge or Foxit) and save or open the file you need. They are also distributed in the folder `CEC405_Code`. Take the code from these files rather than copying it from the listing: depending on the PDF reader, a copied listing may include the line numbers or lose its indentation (macOS Preview drops leading spaces, which breaks the Python programs `gateway.py` and `bridge.py`).

How a lab session works

1. **Before the session** complete the pre-lab box of the lab sheet. You will not be allowed to start the practical work without it.
2. **During the session** work in groups of two or three. Build the circuit in Proteus, write and compile the code, simulate, debug, then (when hardware is available) test on a real board.
3. **Record** your observations directly on the lab sheet: tables, screenshots of the simulation, answers to the questions.
4. **At the end** show the working simulation to the instructor, who signs the sheet. Hand in the sheet and the project folder (`.c`/`.ino`, `.hex`, `.pdsprj`) named `CEC405_LabN_GroupX.zip`.

Laboratory rules and safety

- No food or drinks near the computers and benches.
- Never connect or disconnect components while the board is powered.
- Check polarity (electrolytic capacitors, LEDs, sensors, power supply) before switching on.
- The microcontroller pins work at 5 V and can supply about 20–25 mA only: never connect a motor, relay coil or lamp directly to a pin; use a transistor driver.
- Mains (230 V) circuits are *never* handled in this lab. Relays are tested with low-voltage loads only.
- Report any damaged equipment to the lab technician immediately.

Recommended reading

- R. Barnett, L. O'Cull, S. Cox, *Embedded C Programming and the Microchip PIC*, Thomson Delmar Learning, 2004, ISBN 1-4018-3748-4 (course textbook).
- Microchip, *PIC16F87XA Data Sheet* (DS39582).
- Microchip/Atmel, *ATmega328P Data Sheet*.
- CCS Inc., *CCS C Compiler Manual* (built-in functions reference).
- M. R. Thakur, *Arduino Projects Vol. I with Proteus Simulation*, Circuits4you, 2016 (source of mini-project ideas).
- Labcenter Electronics, *Proteus VSM and Visual Designer help files*.

Part I

Lecture Notes

CHAPTER 1

Introduction to Microprocessor and Microcontroller Systems

Learning objectives

After this chapter (week 1, 4 h lecture) you should be able to:

- describe the building blocks of a microcomputer: CPU, memory, I/O and buses;
- distinguish a microprocessor from a microcontroller, and Harvard from von Neumann architectures;
- list the resources of the PIC16F877A and the ATmega328P and choose between them;
- explain every step of the firmware build process, from C source to a `.hex` file in the chip;
- draw the minimum circuit that makes a microcontroller run.

1.1 What is an embedded system?

An *embedded system* is a computer built into a product to perform a dedicated function: a washing machine controller, a prepaid electricity meter, a car engine ECU, a temperature logger. Unlike a personal computer it has a fixed program (the *firmware*), limited resources, real-time constraints and often runs for years on the same power supply. Most embedded systems are built around a *microcontroller*.

1.2 The microcomputer model

Every programmable digital system, from the 1970s 8085 trainer to today's ARM boards, follows the same model: a central processing unit (CPU) fetches instructions from memory, executes them, and exchanges data with input/output (I/O) devices through buses. Figure 1.1 shows this bus-oriented organisation; the role of each block is described below.

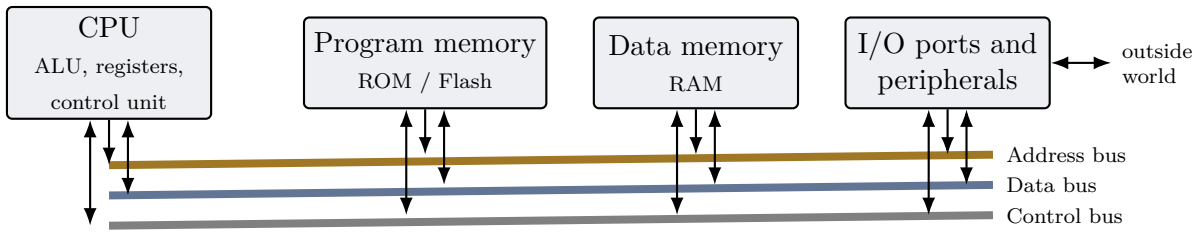


Figure 1.1: Bus-oriented microcomputer.

CPU contains the *arithmetic and logic unit* (ALU), working registers (accumulator, W register, general-purpose registers), the *program counter* (PC) that points to the next instruction, the *stack pointer*, the *status register* (flags Z, C, DC, N, ...) and the control unit that decodes instructions.

Program memory holds the instructions. Today it is Flash (electrically reprogrammable, non-volatile); older systems used ROM, EPROM (UV erasable) or OTP.

Data memory RAM for variables and the stack (volatile) and EEPROM for parameters that must survive a power cut (calibration constants, set points).

I/O parallel ports, timers, ADC, serial interfaces, etc.

Buses the *address bus* selects a memory location or I/O register (unidirectional); the *data bus* carries the data (bidirectional); the *control bus* carries RD, WR, clock, reset and interrupt lines. An n-bit address bus addresses 2^n locations: 16 address lines give $2^{16} = 65\,536 = 64\,\text{KB}$.

1.2.1 The instruction cycle

The CPU repeats forever: **fetch** the instruction at PC → increment PC → **decode** → **execute** (and write back the result). A jump, call, return or interrupt simply loads a new value into PC. On the PIC16 one instruction cycle lasts four oscillator periods: at $f_{osc} = 20\,\text{MHz}$, $T_{cy} = 4/f_{osc} = 200\,\text{ns}$, i.e. 5 MIPS. The AVR executes most instructions in a single clock: 16 MIPS at 16 MHz.

1.3 Microprocessor versus microcontroller

A **microprocessor** (Intel 8085, Zilog Z80, Motorola 6800, and today the x86 or ARM Cortex-A application processors) contains only the CPU. Memory and I/O chips must be added on the board through the external buses. A **microcontroller** integrates CPU, Flash, RAM, EEPROM, clock circuitry and many peripherals on one chip; it only needs a power supply and (often) a crystal.

	Microprocessor system	**Microcontroller**
Integration	CPU only; external ROM, RAM, I/O chips	CPU + memory + peripherals on one chip
Buses	Exposed on pins (e.g. 8085: multiplexed AD0–AD7, A8–A15)	Internal; pins used as I/O ports
Typical use	General-purpose computing, large memory	Dedicated control, low cost, low power
Cost / board size	Higher / larger	Lower / single chip
Examples	8085, Z80, 68000, x86, Cortex-A	PIC16F877A, ATmega328P, STM32, ESP32

1.3.1 The 8085 as a reference 8-bit microprocessor

The Intel 8085 (1976) is still used in many teaching trainers. It has an 8-bit data bus, a 16-bit address bus (64 KB address space), the lower address byte multiplexed with the data bus (AD0–AD7, demultiplexed with the ALE signal and a 74LS373 latch), the accumulator A, six general registers B, C, D, E, H, L (usable as pairs BC, DE, HL), a 16-bit stack pointer and PC, five interrupt inputs (TRAP, RST7.5, RST6.5, RST5.5, INTR) and a separate 256-port I/O space accessed with the `IN` and `OUT` instructions. To build a working 8085 system you need at least a latch, an EPROM, a RAM, an address decoder (74LS138) and an I/O chip (8255 PPI). A microcontroller replaces this whole board.

1.4 Architectures

Von Neumann versus Harvard. In a *von Neumann* machine (8085, x86) program and data share one memory and one bus: the CPU cannot fetch an instruction and access data at the same

time. In a *Harvard* machine (PIC, AVR) program and data memories have separate buses, possibly of different widths: the PIC16 program memory is 14 bits wide while its data memory is 8 bits wide. This allows instruction fetch and data access in parallel (*pipelining*).

CISC versus RISC. *Complex instruction set computers* (8085: 74 instructions, x86) have many instructions of variable length and duration. *Reduced instruction set computers* (PIC16: 35 instructions; AVR: about 130 simple instructions with 32 working registers) have fixed-length instructions that execute in one cycle (or a few), which simplifies the hardware and makes timing predictable — an important property for real-time control.

1.5 The two microcontrollers used in this course

1.5.1 Microchip PIC16F877A

Core	8-bit RISC, Harvard, 35 instructions, 8-level hardware stack
Clock	DC to 20 MHz (200 ns instruction cycle); HS crystal mode for 4–20 MHz
Memory	8 K words × 14 bits Flash; 368 bytes RAM (4 banks); 256 bytes EEPROM
I/O	33 pins in 5 ports: PORTA (RA0–RA5), PORTB (RB0–RB7), PORTC, PORTD (8 bits each), PORTE (RE0–RE2); 25 mA sink/source per pin
Analogue	10-bit ADC, 8 channels (AN0–AN7); 2 analogue comparators
Timers	Timer0 (8-bit), Timer1 (16-bit), Timer2 (8-bit), watchdog timer
Capture/PWM	2 CCP modules (capture, compare, 10-bit PWM)
Serial	USART (RS232 via MAX232), MSSP (SPI or I^2C)
Interrupts	15 sources (RB0/INT, RB4–RB7 change, timers, ADC, USART, . . .)
Package	40-pin DIP (most common in labs)

1.5.2 Microchip (Atmel) ATmega328P — the Arduino Uno

Core	8-bit AVR RISC, Harvard, 32 general registers, most instructions in one clock
Clock	16 MHz crystal on the Arduino Uno (up to 20 MHz)
Memory	32 KB Flash (0.5 KB used by the Arduino bootloader); 2 KB SRAM; 1 KB EEPROM
I/O	23 GPIO; on the Uno: digital pins D0–D13, analogue pins A0–A5; 20 mA recommended per pin
Analogue	10-bit ADC, 6 channels on the Uno (A0–A5)
Timers	Timer0 and Timer2 (8-bit), Timer1 (16-bit); 6 PWM outputs (D3, D5, D6, D9, D10, D11)
Serial	USART (D0 = RX, D1 = TX, also connected to the USB bridge), SPI, I^2C (A4 = SDA, A5 = SCL)

Choosing a microcontroller

Start from the application: number and type of I/O (digital, analogue, PWM), communication interfaces, memory needed by the program and data, speed, power budget, supply voltage, package, cost, availability of tools and of local know-how. A 40-pin PIC16F877A suits a stand-alone controller with many I/O lines; the Arduino ecosystem shortens development time thanks to its libraries.

1.6 The minimum system

A microcontroller needs only four things to run: (1) a regulated supply (5 V, with a 100 nF decoupling capacitor close to each pair of supply pins), (2) a clock source, (3) a reset circuit, and (4) a program in Flash. Figure 1.2 shows these elements for the PIC16F877A: the 10 kΩ resistor

holds $\overline{\text{MCLR}}$ high (the RESET button pulls it low), and the 20 MHz crystal with its two 22 pF capacitors forms the oscillator.

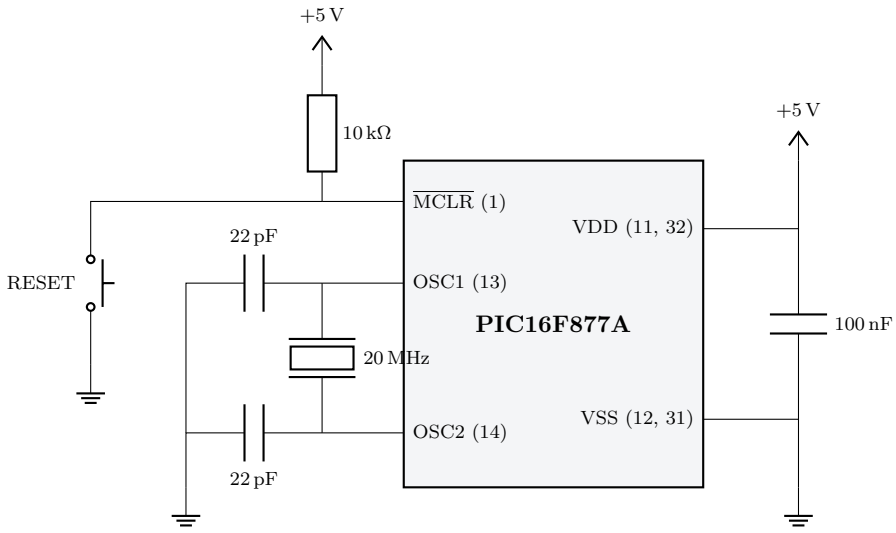


Figure 1.2: Minimum circuit of the PIC16F877A (supply, 20 MHz crystal, MCLR reset).

In Proteus

Proteus connects the power pins of microcontrollers automatically and does not simulate the crystal: the clock frequency is a *property* of the component. You may omit the crystal and the decoupling capacitors in the simulation schematic, but you **must** draw them on a real board.

1.7 From C source to a running chip: the build process

Understanding the tool chain lets you read compiler manuals, interpret error messages and debug efficiently. Figure 1.3 summarises the chain; each step is detailed below.

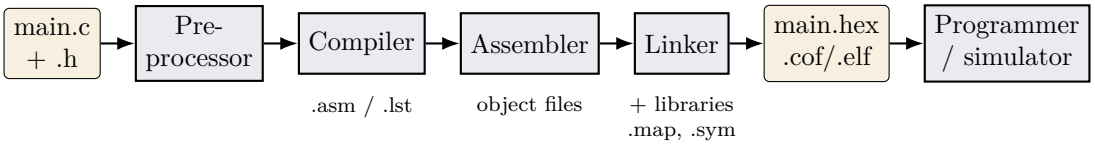


Figure 1.3: Build chain of microcontroller firmware.

1. **Preprocessor** handles the lines starting with `#`: it includes the device header (`#include <16F877A.h>`), replaces macros (`#define LED PIN_D0`) and evaluates conditional compilation. CCS adds its own pre-processor directives that configure the hardware: `#fuses`, `#use delay`, `#use rs232`.
2. **Compiler** translates C into assembly language for the target and optimises it. Read the *list file* (`.lst`) to see which instructions each C line produced.
3. **Assembler** translates mnemonics into machine code (object file).
4. **Linker** places code and variables at absolute addresses, adds library routines (`printf`, floating point, ...) and produces the output files: `.hex` (machine code), `.cof`/`.elf` (code + debug symbols for source-level debugging in Proteus), `.map`/`.sym` (memory map). The compiler also reports ROM and RAM usage: check it!

5. **Programming**: the `.hex` file is written into Flash with an in-circuit programmer (PICkit through the ICSP pins MCLR, PGD = RB7, PGC = RB6) or, on the Arduino, by the *bootloader* that receives the program through the USB-serial link. In Proteus, the `.hex` or `.cof` file is attached to the microcontroller's *Program File* property. In a Proteus 9.2 *firmware project* this is automatic: Proteus calls the compiler (CCS C for the PIC, its built-in Arduino compiler for the Uno) and loads the result itself.

1.7.1 The Intel HEX format

A `.hex` file is plain text. Each line (record) has the form `:LLAAAATT[DD...]CC`: byte count LL, 16-bit address AAAA, record type TT (00 = data, 01 = end of file, 04 = extended linear address), the data bytes, and a checksum CC equal to the two's complement of the sum of all the previous bytes. Example:

```
:0300300002337A1E      3 data bytes 02 33 7A stored from address 0x0030
:00000001FF            end-of-file record
```

Check: $03 + 00 + 30 + 00 + 02 + 33 + 7A = E2_{16}$; two's complement of E2 is 1E. ✓

1.7.2 Configuration bits (fuses)

Some options are not set by the program at run time but by *configuration bits* programmed with the code: oscillator type (`HS` for a crystal above 4 MHz, `XT` up to 4 MHz), watchdog timer (`NOWDT`), code protection (`NOPROTECT`), low-voltage programming (`NOLVP`, which frees RB3), brown-out reset (`BROWNOUT`) and power-up timer (`PUT`). A wrong oscillator fuse is the most frequent reason why "the program does nothing" on a real board.

1.8 Debugging techniques

- **Simulation** (Proteus VSM): run, pause, single-step through the C source when the `.cof`/`.elf` file is loaded, watch variables and registers, use virtual instruments (oscilloscope, logic analyser, virtual terminal, voltmeter).
- **Instrumented code**: toggle a spare LED, send values with `printf` to a terminal.
- **In-circuit debugging**: PICkit/ICD or debugWIRE with breakpoints on the real chip.
- **Measurements**: multimeter (supply, logic levels), oscilloscope (clock, PWM, serial frames), logic analyser (protocols).
- **Method**: reproduce the fault, divide the system (power → clock → reset → program → peripheral), change one thing at a time, keep notes.

Tutorial sheet 1

1. Define: embedded system, firmware, microprocessor, microcontroller, bus, program counter.
2. Draw the block diagram of a bus-oriented microcomputer and describe the role of each bus.
3. An 8085 has 16 address lines. What is its maximum memory size? How many address lines are needed to address an 8 KB EPROM, and which address lines drive the 74LS138 decoder if the EPROM is placed at 0000H–1FFFH?

4. Compare the von Neumann and Harvard architectures. Why can the PIC16 program memory be 14 bits wide while its data is 8 bits wide?
5. A PIC16F877A runs with a 4 MHz crystal. Compute the instruction cycle time. How long does a loop of 250 iterations take if each iteration executes 4 single-cycle instructions?
6. Compare the resources of the PIC16F877A and the ATmega328P (memory, I/O, ADC, timers, serial interfaces). Which would you choose for a 6-zone alarm with 8 sensors, a keypad and an LCD? Justify.
7. List the steps of the build process and the file produced by each one. Which file do you load into Proteus to step through the C source?
8. Verify the checksum of the record `:0400100012345678E8`. Is the record valid?
9. Draw the minimum circuit of a PIC16F877A with a 8 MHz crystal. Which oscillator fuse do you use? What happens if MCLR is left floating?
10. A student's board does nothing after programming. Propose a systematic fault-finding sequence.

CHAPTER 2

Interfacing Microcontrollers

Learning objectives

After this chapter (week 2, 4 h lecture) you should be able to:

- configure I/O pins and respect their electrical limits;
- compute the components that connect LEDs, switches and loads to a microcontroller;
- convert an analogue sensor voltage into a physical quantity with the ADC;
- explain the protocols used in the labs: HD44780 LCD, 1-Wire, UART/RS232, I^{2}C, SPI;
- generate PWM and drive a servo; choose between polling and interrupts for real-time tasks.

2.1 Digital input/output ports

Each port pin is controlled by three registers: the *direction* register (PIC: `TRISx`, 1 = input, 0 = output; AVR: `DDRx`, 1 = output), the *output latch* (`PORTx`/`LATx` on PIC, `PORTx` on AVR) and the *input* register (reading `PORTx` on PIC, `PINx` on AVR). Figure 2.1 shows how these registers act on one pin: the PORT latch holds the value to output, the TRIS latch enables or disables the output driver, and an input buffer always reads the actual pin level.

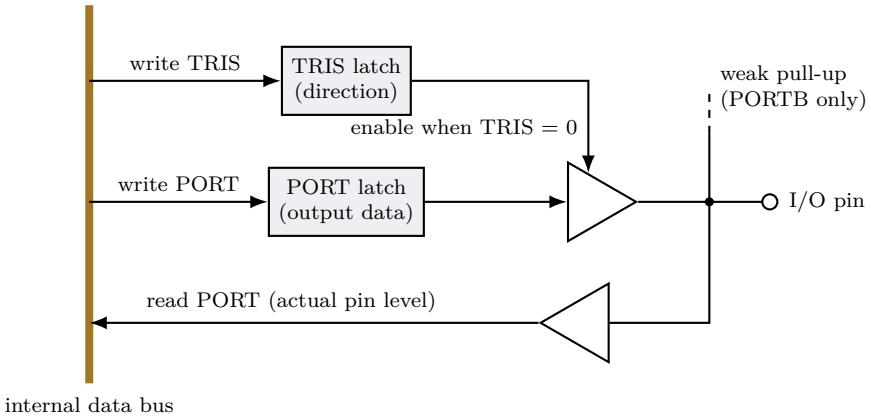


Figure 2.1: Simplified structure of one I/O pin of the PIC16F877A.

High-level functions hide these registers:

Action	CCS C (PIC)	Arduino
Set pin high / low	`output_high(PIN_D0);` `output_low(PIN_D0);`	`digitalWrite(13, HIGH);`
Toggle	`output_toggle(PIN_D0);`	`digitalWrite(13, !digitalRead(13));`
Write a whole port	`output_d(0x55);`	`PORTD = 0x55;` (direct register access)
Read a pin	`if (input(PIN_B0)) {...}`	`if (digitalRead(2)) {...}`
Internal pull-ups	`port_b_pullups(TRUE);` (PORTB only)	`pinMode(2, INPUT_PULLUP);`

Electrical limits. At $V_{DD} = 5\,\text{V}$ a PIC16F877A pin can source or sink 25 mA; each port has a total limit (about 200 mA for PORTA+B+E). The AVR recommends 20 mA per pin and 200 mA for the whole chip. A logic input reads '1' above about $0.8V_{DD}$ (Schmitt-trigger inputs) or 2 V (TTL inputs) and '0' below about $0.2V_{DD}$/0.8 V. A floating input reads random values: **every input must be pulled up or down**.

2.2 Connecting LEDs

An LED has a forward voltage V_F (red ≈ 1.8–2 V, green ≈ 2.1 V, blue/white ≈ 3 V) and needs a few mA to be visible. It can be connected in *source* mode, where the pin supplies the current and '1' lights the LED (Figure 2.2a), or in *sink* mode, where the pin absorbs the current and '0' lights the LED (Figure 2.2b). In both cases the series resistor limits the current:

$$R = \frac{V_{OH} - V_F}{I_F} \qquad \text{e.g. } R = \frac{5-2}{10\,\text{mA}} = 300\,\Omega \rightarrow 330\,\Omega \text{ (E12 value).} \tag{2.1}$$

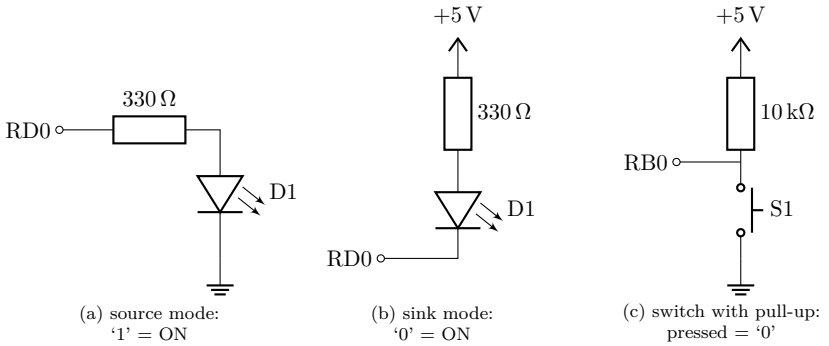


Figure 2.2: LED and switch interfaces.

2.3 Connecting switches

With a pull-up resistor (external 4.7 kΩ to 10 kΩ, or the internal weak pull-ups of PORTB or of the AVR) the input reads '1' when the switch is open and '0' when it is pressed (*active-low*), as in Figure 2.2c. With a pull-down resistor the logic is reversed.

Contact bounce. A mechanical contact bounces for 1–20 ms before settling: a single press can be read as several presses (Figure 2.3). Software debouncing: when a change is detected, wait 20–50 ms and read the pin again; act only if the new state is confirmed. For a *toggle* function, act on the *edge* (transition from released to pressed), not on the level, and wait for release.

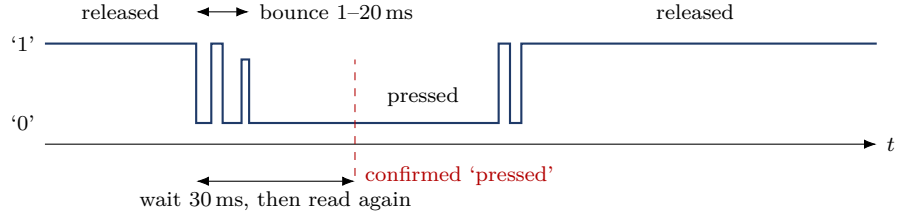


Figure 2.3: Contact bounce of a push-button connected as in Figure 2.2c, and the debouncing principle.

```
if (!input(PIN_B0)) {              // pressed (active-low)
   delay_ms(30);                   // debounce
   if (!input(PIN_B0)) {
      output_toggle(PIN_D0);       // act once
      while (!input(PIN_B0));      // wait for release
   }
}
```

2.4 Driving loads: transistors and relays

Loads that need more current or a different voltage (relays, motors, buzzers, lamps) are switched by a transistor (Figure 2.4). For an NPN transistor (BC547, 2N2222) used as a switch, choose a base current about ten times larger than I_C/β_{min} to guarantee saturation:

$$I_B = \frac{10\, I_C}{\beta_{min}}, \qquad R_B = \frac{V_{OH} - V_{BE}}{I_B}. \tag{2.2}$$

Example: relay coil 5 V, 70 Ω ⇒ $I_C \approx 71$ mA; $\beta_{min} = 100 \Rightarrow I_B = 7.1$ mA, $R_B = (5 - 0.7)/7.1\,\text{mA} \approx 600\,\Omega \rightarrow 560\,\Omega$. A **flyback diode** (1N4148/1N4007) across the coil, cathode to $+V$, absorbs the inductive voltage spike when the transistor turns off. The relay contacts then switch the load circuit, which is electrically isolated from the microcontroller. For larger currents use a logic-level MOSFET (IRLZ44N) or a driver array (ULN2803: 8 Darlington drivers with built-in diodes).

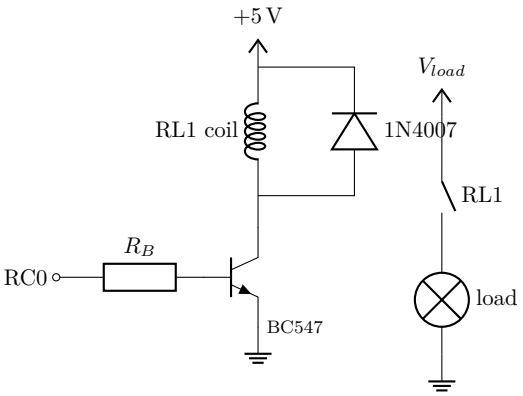


Figure 2.4: Relay driver with flyback diode; the relay contact RL1 switches a separate load circuit.

2.5 Seven-segment displays (reminder)

A digit has 7 segments a–g (+ decimal point). In a common-cathode display a segment lights when its anode receives '1' (through a 330 Ω resistor). Several digits are *multiplexed* (Figure 2.5): the segment lines of all digits are connected in parallel to one port, and each common cathode is switched by its own transistor. The microcontroller outputs the pattern of digit 1 and turns on its transistor only, then does the same for digit 2, and so on, fast enough (each digit refreshed at more than 50 Hz) for the eye to see all digits lit at the same time.

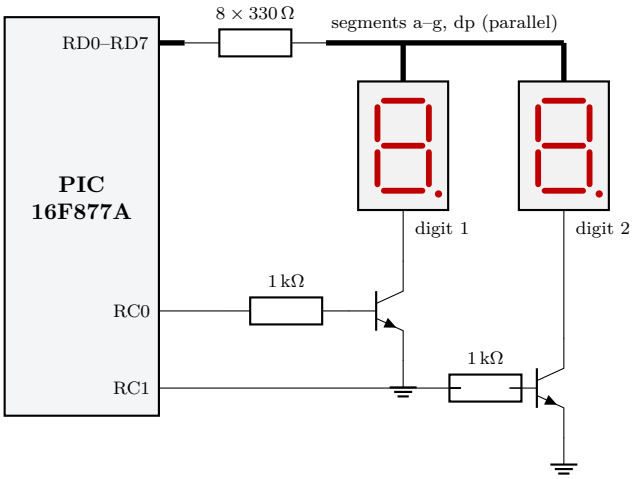


Figure 2.5: Two multiplexed common-cathode seven-segment digits.

2.6 Analogue inputs and the ADC

A successive-approximation ADC converts an input voltage V_{in} between V_{REF-} and V_{REF+} into an n-bit number N:

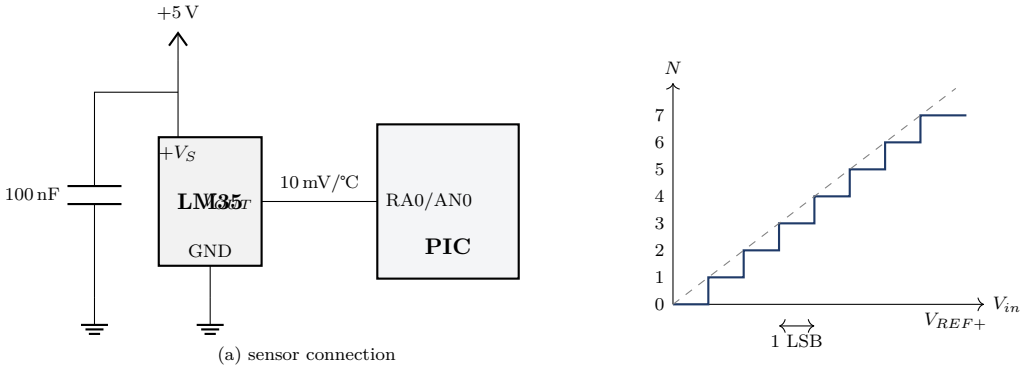
$$N = \left\lfloor \frac{V_{in} - V_{REF-}}{V_{REF+} - V_{REF-}} \left(2^n - 1\right) \right\rfloor, \qquad \text{resolution (LSB)} = \frac{V_{REF+} - V_{REF-}}{2^n - 1} \ (\text{or } /2^n). \tag{2.3}$$

With 10 bits and $V_{REF} = 5\,\text{V}$, 1 LSB $\approx 4.89\,\text{mV}$. The conversion characteristic is a staircase (Figure 2.6b): every input voltage inside one step gives the same code, which is the origin of the *quantisation error* of up to 1 LSB.

The LM35 temperature sensor delivers $V_{out} = 10\,\text{mV}/^\circ\text{C} \times T$ ($0\,^\circ\text{C} \to 0\,\text{V}$, $25\,^\circ\text{C} \to 250\,\text{mV}$), accuracy $\pm 0.5\,^\circ\text{C}$ at $25\,^\circ\text{C}$, pins $+V_S$, V_{OUT}, GND. It is connected directly to an analogue input (Figure 2.6a). Combined with a 10-bit ADC at 5 V:

$$T\,[^\circ\text{C}] = \frac{N \times 5000\,\text{mV}/1023}{10\,\text{mV}/^\circ\text{C}} = N \times 0.4888. \tag{2.4}$$

The resolution is about $0.5\,^\circ\text{C}$. To improve it, lower V_{REF+} (e.g. 1.1 V internal reference on the AVR gives $0.1\,^\circ\text{C}$) or amplify the sensor signal.



(a) sensor connection

(b) 3-bit ADC characteristic

Figure 2.6: (a) LM35 connected to analogue input AN0; (b) staircase transfer characteristic of an ADC.

ADC in practice

Select the analogue pins (`setup_adc_ports`), the conversion clock (`setup_adc`), the channel (`set_adc_channel`), wait the acquisition time (about 20 µs) after a channel change, then convert (`read_adc`). Use `#device ADC=10` in CCS to obtain 10-bit results (the default is 8 bits!). Average several readings to reduce noise.

2.7 The HD44780 character LCD

The 16×2 alphanumeric LCD (Proteus model `LM016L`) is driven by an HD44780-compatible controller. Its pins are listed in the table below, and Figure 2.7 shows the 4-bit connection to PORTD used in Lab 4.

Pin	Name	Function	Typical connection
1, 2	VSS, VDD	Supply 0 V, 5 V	
3	VEE (V0)	Contrast	10 kΩ potentiometer wiper
4	RS	0 = command, 1 = data (character)	MCU pin
5	RW	0 = write, 1 = read	GND (write only) or MCU pin
6	E	Enable: data latched on falling edge	MCU pin
7–10	D0–D3	Data bus low nibble	not used in 4-bit mode
11–14	D4–D7	Data bus high nibble	MCU pins
15, 16	A, K	Backlight LED	5 V via resistor, GND

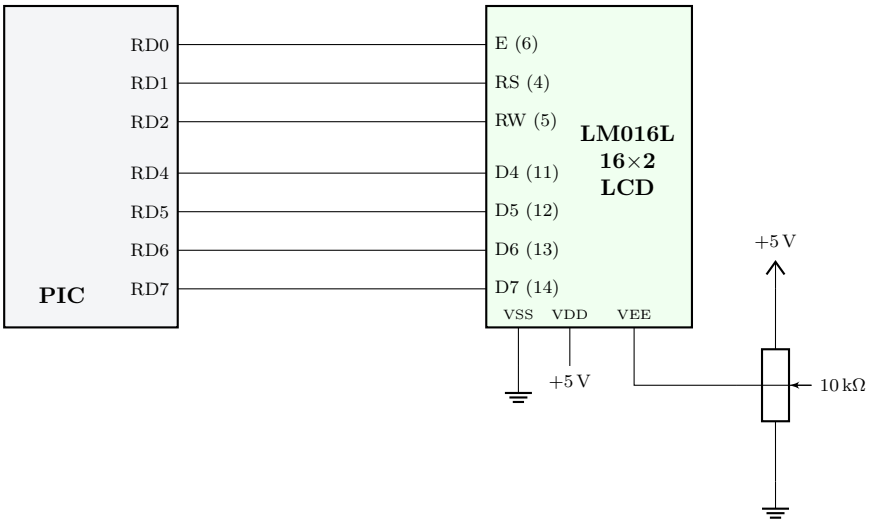


Figure 2.7: HD44780 LCD connected to PORTD in 4-bit mode (D0–D3 unconnected).

In **4-bit mode** each byte is sent as two nibbles (high then low) on D4–D7, saving four pins. Common commands: `0x01` clear display, `0x02` cursor home, `0x0C` display on / cursor off, `0x28` 4-bit, 2 lines, 5×8 font, `0x80+addr` set cursor (line 1 starts at address 0x00, line 2 at 0x40). Libraries do this for you: CCS `lcd.c` (`lcd_init`, `lcd_putc`, `lcd_gotoxy`, `printf(lcd_putc, ...)`) and Arduino `LiquidCrystal` (`begin`, `setCursor`, `print`).

2.8 The 1-Wire bus and the DS18B20

The DS18B20 is a digital thermometer (−55 to +125 °C, ±0.5 °C from −10 to +85 °C, 9- to 12-bit resolution, 12-bit default = 0.0625 °C) that communicates over a single data line DQ with a 4.7 kΩ

pull-up resistor. Each device has a unique 64-bit ROM code, so many sensors can share one wire (Figure 2.8).

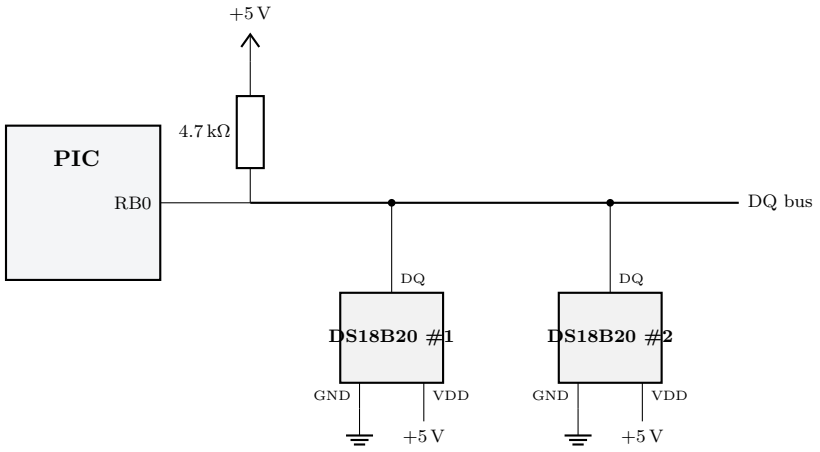


Figure 2.8: 1-Wire bus: one pull-up resistor and several DS18B20 sensors on a single data line.

All communication is made of *time slots* started by the master pulling DQ low (Figure 2.9):

- **Reset/presence**: master pulls DQ low $\geq 480\,\mu s$, releases it; the sensor answers with a low *presence pulse* 60–240 µs long.
- **Write 1**: low for 1–15 µs, then release, slot $\geq 60\,\mu s$. **Write 0**: low for the whole 60 µs slot.
- **Read**: master pulls low $\approx 2\,\mu s$, releases, samples DQ before 15 µs: '1' if high, '0' if the sensor holds it low.
- Bytes are sent **LSB first**.

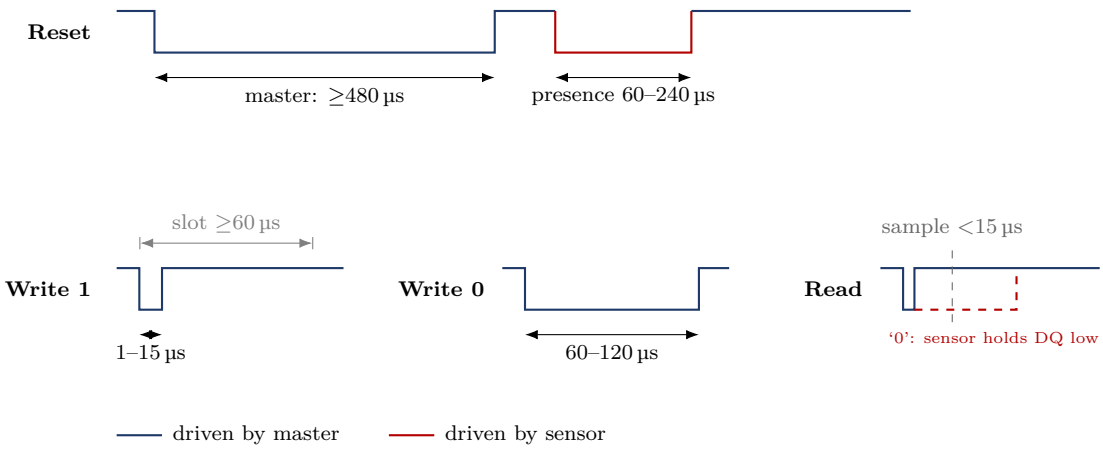


Figure 2.9: 1-Wire time slots (DQ idles high thanks to the pull-up resistor).

Measurement sequence with a single sensor: `Reset`, `0xCC` (Skip ROM), `0x44` (Convert T), wait 750 ms (12-bit); `Reset`, `0xCC`, `0xBE` (Read scratchpad), read byte 0 (LSB) and byte 1 (MSB). The 16-bit result is a two's-complement number in units of 0.0625 °C:

$$\texttt{0x0191} = 401 \Rightarrow 401 \times 0.0625 = 25.0625\,^{\circ}\text{C}; \qquad \texttt{0xFF5E} = -162 \Rightarrow -10.125\,^{\circ}\text{C}.$$

To drive DQ in open-drain fashion, a pin is either *output low* or *input* (released, the pull-up gives '1'). In CCS: `output_low(PIN)` and `output_float(PIN)`.

2.9 Serial communication: UART and RS232

A UART sends each character as an asynchronous frame (Figure 2.10): one *start bit* (0), 5–9 data bits (LSB first), optional parity, one or two *stop bits* (1). Both sides must use the same settings, e.g. **9600 8N1**: 9600 baud, 8 data bits, no parity, 1 stop bit. One bit lasts 1/9600 = 104 µs; a 10-bit frame takes 1.04 ms, i.e. at most 960 characters/s.

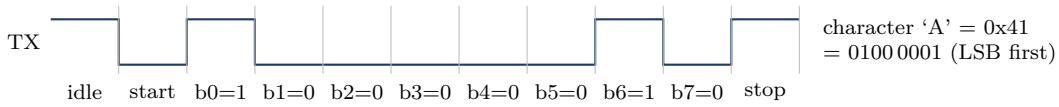


Figure 2.10: UART frame of the character 'A' in 8N1 format (idle = 1).

RS232 is the electrical standard of the PC COM port: '1' (mark) is −3 to −15 V, '0' (space) is +3 to +15 V. The microcontroller's TTL levels (0/5 V) must therefore be converted by a **MAX232** with four 1 µF charge-pump capacitors (Figure 2.11). Modern PCs have no COM port: use a USB–serial adapter (FT232, CH340, CP2102) at TTL level, connected directly to RC6/RC7 (TX of the adapter to RX of the PIC and vice versa). The Arduino Uno already has a USB–serial bridge on D0/D1. In CCS:

```
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, parity=N, bits=8, stop=1)
printf("T = %4.1f C\r\n", temp);   // send formatted text
c = getc();                        // wait for and read one character
if (kbhit()) c = getc();           // non-blocking: read only if a character arrived
```

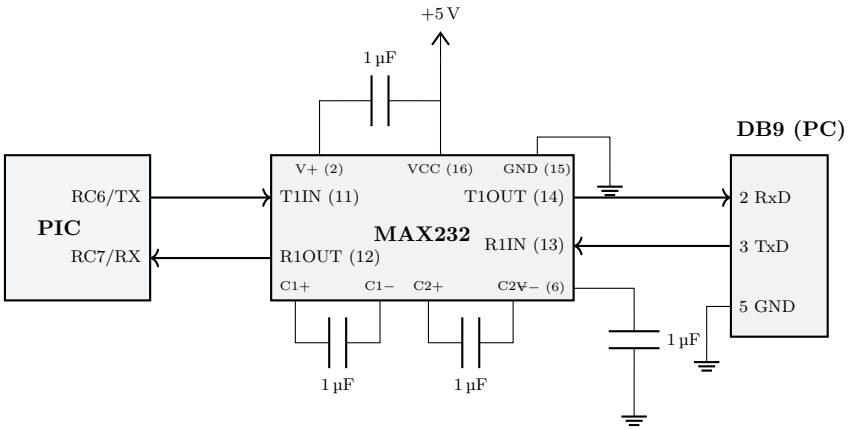
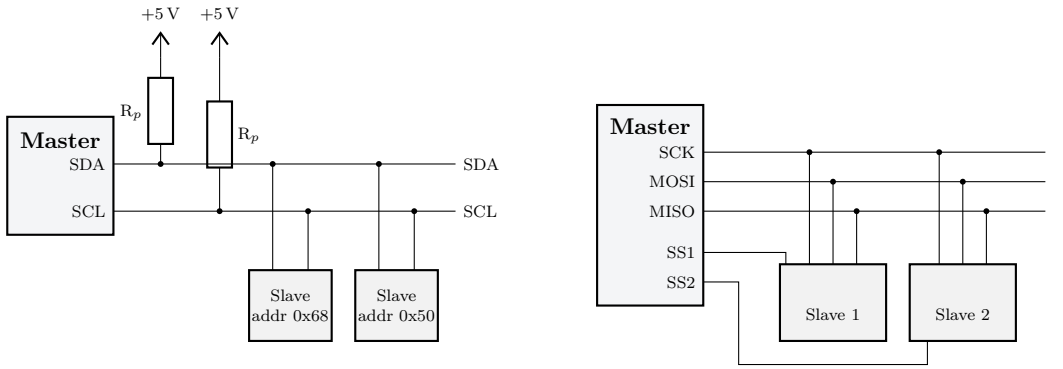


Figure 2.11: RS232 interface between the PIC USART and a PC COM port using a MAX232.

2.10 I²C and SPI (overview)

Both are synchronous serial buses: a clock line generated by the master times every bit. Their wiring is compared in Figure 2.12 and their characteristics in the table below.

	I²C (TWI)	**SPI**
Wires	2: SDA, SCL (open-drain + pull-ups)	4: SCK, MOSI, MISO, $\overline{\text{SS}}$ (one SS per slave)
Addressing	7-bit address in the first byte	Hardware chip select
Speed	100 kHz, 400 kHz (typ.)	Several MHz
Duplex	Half	Full
Examples	RTC DS1307, EEPROM 24C02, LCD backpack PCF8574, BMP280	SD card, ENC28J60/W5100 Ethernet, nRF24L01



(a) I^{2}C: 2 shared lines, addressing by software

(b) SPI: shared SCK/MOSI/MISO, one $\overline{\text{SS}}$ per slave

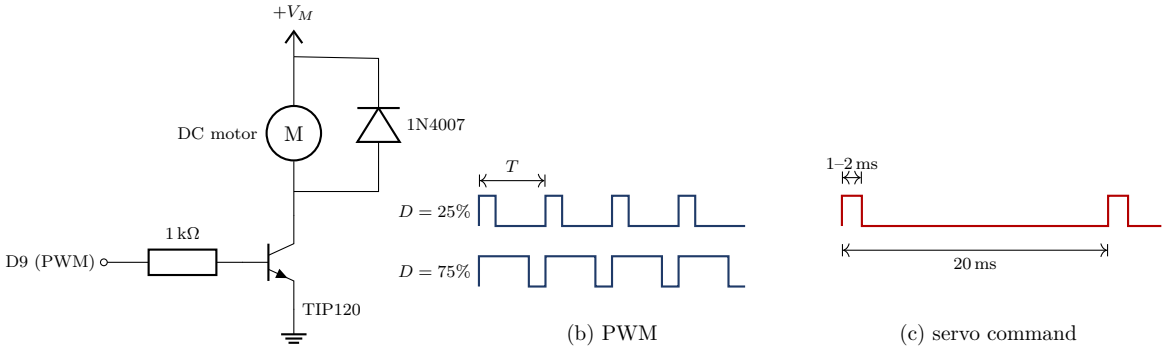
Figure 2.12: Wiring of (a) an I^{2}C bus and (b) an SPI bus with two slaves.

2.11 Timers, PWM and servomechanisms

A **timer** is a counter incremented by the instruction clock (through a prescaler) or by an external pin. When it overflows it sets a flag and may trigger an interrupt: this is the basis of accurate time bases, frequency/RPM measurement (counting pulses during a gate time) and pulse-width measurement (capture mode).

Pulse-width modulation (PWM) outputs a rectangular wave of fixed period T and variable high time t_{on}; the average voltage is $V_{avg} = D \cdot V_{DD}$ with duty cycle $D = t_{on}/T$ (Figure 2.13b). It controls LED brightness and DC motor speed through a transistor (Figure 2.13a); the flyback diode is mandatory because the motor is inductive. Arduino: `analogWrite(pin, 0..255)` on pins 3, 5, 6, 9, 10, 11 (about 490 or 980 Hz). PIC: CCP module with Timer2, `setup_ccp1(CCP_PWM);` `set_pwm1_duty(value);`

A hobby **servo** is a closed-loop servomechanism (motor + gearbox + potentiometer + controller). It has three wires (GND, +5 V, signal) and its position is commanded by a pulse repeated every 20 ms (Figure 2.13c): 1 ms ≈ 0°, 1.5 ms ≈ 90°, 2 ms ≈ 180°. Arduino: `#include <Servo.h>`, `myservo.attach(9); myservo.write(90);`



(a) motor speed control

(b) PWM

(c) servo command

Figure 2.13: (a) PWM speed control of a DC motor; (b) PWM signals with two duty cycles; (c) servo command pulse.

2.12 Interrupts and real-time issues

- **Polling** — the main loop checks each device in turn. Simple, but the response time equals the loop time, and every `delay_ms()` blocks everything else.
- **Interrupts** — a hardware event (pin edge, timer overflow, byte received) suspends the main program and runs an *interrupt service routine* (ISR), after which the main program resumes exactly where it was stopped (Figure 2.14). Keep ISRs short: set a flag or store a byte, and let the main loop do the work. Variables shared with an ISR must be declared `volatile` (Arduino/GCC).
- **Non-blocking timing** — instead of `delay()`, compare the elapsed time with a period: `if (millis() - last >= 1000) { last += 1000; ...}`. Several tasks can then run "in parallel" with different periods (cooperative multitasking).
- **Watchdog timer** — resets the microcontroller if the program stops refreshing it, recovering from software lock-ups.

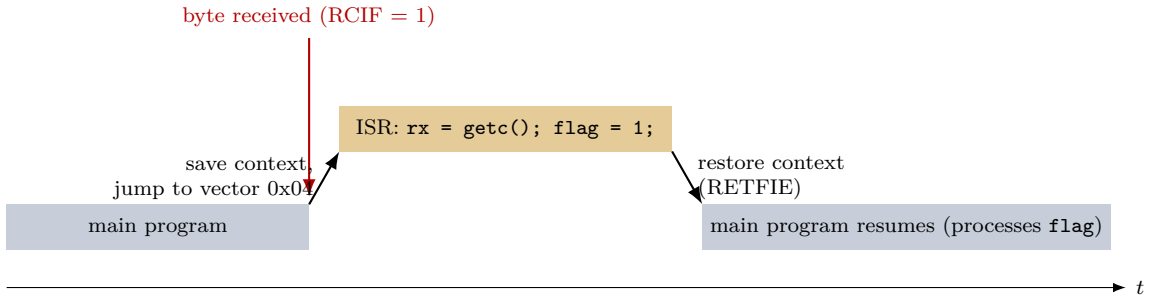


Figure 2.14: Execution of an interrupt: the main program is suspended, the ISR runs, then the main program resumes.

CCS example of a serial receive interrupt:

```
char rx; int1 rx_flag = 0;
#int_rda                          // USART receive interrupt
void serial_isr(void) { rx = getc(); rx_flag = 1; }

void main(void) {
   enable_interrupts(INT_RDA);
   enable_interrupts(GLOBAL);
   while (TRUE) {
      if (rx_flag) { rx_flag = 0; /* process rx */ }
      /* other tasks run here without waiting for the serial line */
   }
}
```

Tutorial sheet 2

1. Compute the series resistor for a blue LED ($V_F = 3.1\,\text{V}$, $I_F = 8\,\text{mA}$) driven by a 5 V pin. Choose the nearest E12 value and compute the actual current.
2. Eight red LEDs (15 mA each) are connected to PORTD in source mode, all ON. Is the PIC within its limits? Propose a safer solution.
3. Draw a switch connected to RB4 with an external pull-down resistor. What does the program read when the switch is pressed? Why is a pull resistor necessary?

4. Explain contact bounce and write a CCS C function `int1 pressed(void)` that returns 1 only once per press of the switch on RB0.
5. A 12 V relay has a 400 Ω coil. Design the BC547 driver ($\beta_{min} = 110$) from a 5 V pin. Why is the diode needed?
6. An LM35 is connected to AN0 of a PIC (10-bit ADC, $V_{REF} = 5$ V). The ADC returns 63. Compute the temperature. What value is expected at 37 °C? What is the resolution in °C?
7. Give the HD44780 command sequence to write "OK" at the start of line 2.
8. A DS18B20 returns LSB = 0x50, MSB = 0x05. Compute the temperature. Same question for LSB = 0x90, MSB = 0xFC.
9. Draw the frame of the character '5' (0x35) sent at 9600 8N1. How long does it take to send the string "T=25.0\r\n"?
10. Compare I^2C and SPI. Which would you use for a W5100 Ethernet controller, and for a DS1307 RTC?
11. A servo must move to 45°. Give the pulse width and period of the command signal.
12. Rewrite a "blink every 500 ms and read a button" program so that the button is never missed. Use polling with `millis()`, then an interrupt.

Part II

Laboratory Sheets

LAB 1

Initiation to the CCS C Compiler and Proteus

Week:	3	**Duration:**	4 hours
Platform:	PIC16F877A · CCS C · Proteus 9.2 SP1		
Assessment:	Checked at end of session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Create a Proteus 9.2 project with a CCS C firmware project for the PIC16F877A; compile it from Proteus and inspect the files produced by the CCS compiler.
- Draw the schematic in Proteus and simulate the firmware.
- Debug at source level: breakpoints, single step, watch window.

Pre-lab work (to be done before the session)

1. Read Sections 1.6–1.8 (minimum system, build process, debugging).
2. Compute the series resistor of a red LED ($V_F = 2\,\text{V}$) for $I_F = 10\,\text{mA}$ at 5 V.
3. Compute the instruction cycle time of a PIC16F877A at 20 MHz.
4. Write, on paper, a program that turns RD0 on for 500 ms and off for 500 ms forever.

L1.1 Equipment

PC with Proteus 9.2 SP1 Professional and the CCS C Compiler (PCWHD or PCM, v5) installed on the *same* computer. Proteus 9.2 calls the CCS compiler itself: you write, compile, simulate and debug without leaving Proteus. Proteus parts: `PIC16F877A`, `LED-RED`, `RES`, `BUTTON`.

L1.2 Part A — First CCS C project, compiled from Proteus

1. Create a folder `D:\CEC405\Lab1` (no spaces or accents in the path).
2. Start Proteus 9.2 SP1. **File** → **New Project**: name `Lab1`, path = your folder; *Create a schematic from the selected template* (DEFAULT); *Do not create a PCB layout.*
3. On the *Firmware* page select **Create Firmware Project**, then set the family to `PIC16`, controller `PIC16F877A`, compiler `CCS C Compiler`. Tick *Create Quick Start Files*. Finish. (If CCS is not in the compiler list, or is marked as not configured, click **Compilers...**, select the CCS line and set its installation folder, usually `C:\Program Files (x86)\PICC`. This is done once per PC.)

4. Proteus places the PIC on the schematic and opens the **Source Code** tab with a skeleton `main.c`. Replace its whole content with the program:

File: `Lab1_blink.c` attached to this PDF

```
#include <16F877A.h>
#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT, PUT
#use delay(clock = 20000000)      // 20 MHz crystal, used by delay_ms()

#define LED PIN_D0

void main(void)
{
   output_low(LED);
   while (TRUE)                    // an embedded program never ends
   {
      output_high(LED);
      delay_ms(500);
      output_low(LED);
      delay_ms(500);
   }
}
```

5. **Build** → **Build Project**. Proteus runs the CCS compiler and shows its messages (errors, warnings, ROM and RAM usage) in the output window at the bottom of the **Source Code** tab. Note the ROM and RAM usage below. Open the firmware sub-folder of the project: identify the `.hex`, `.cof`, `.lst`, `.sym` and `.err` files produced by CCS.
6. Open the `.lst` file. Find the assembly instructions generated for `output_high(LED)`.

ROM used: _____

RAM used: _____

Instructions for `output_high(LED)`: _____

Compiling outside Proteus

The same program can also be compiled in the CCS IDE (**Compile**, F9). The `.cof` or `.hex` file is then attached by hand to the **Program File** property of the PIC in a project created *without* a firmware project. You will need the `.hex` file in any case to program a real chip with a PICkit. In Labs 2–5, create every project as in steps 2–5 above (one Proteus project with a CCS firmware project per lab).

L1.3 Part B — Proteus schematic and simulation

1. Open the **Schematic Capture** tab: the PIC16F877A is already placed. Press `P` (Pick Devices) and add `PIC16F877A`, `LED-RED`, `RES`. Place them and wire: RD0 (pin 19) → 330 Ω → LED anode; LED cathode → GROUND terminal (left toolbar **Terminals Mode**). This is the source-mode connection of Figure 2.2a; the complete circuit is given in Figure L1.1.
2. Double-click the resistor and set its value to 330. Double-click the PIC and set **Processor Clock Frequency** = 20MHz. Do not change **Program File**: it is linked to the firmware project, so Proteus loads the CCS output itself. Click OK.
3. Start the simulation with the **Play** button (bottom left). If the source has changed since the last build, Proteus recompiles it first. The LED must blink at 1 Hz. Watch the small red/blue squares on the pins: they show the logic state.

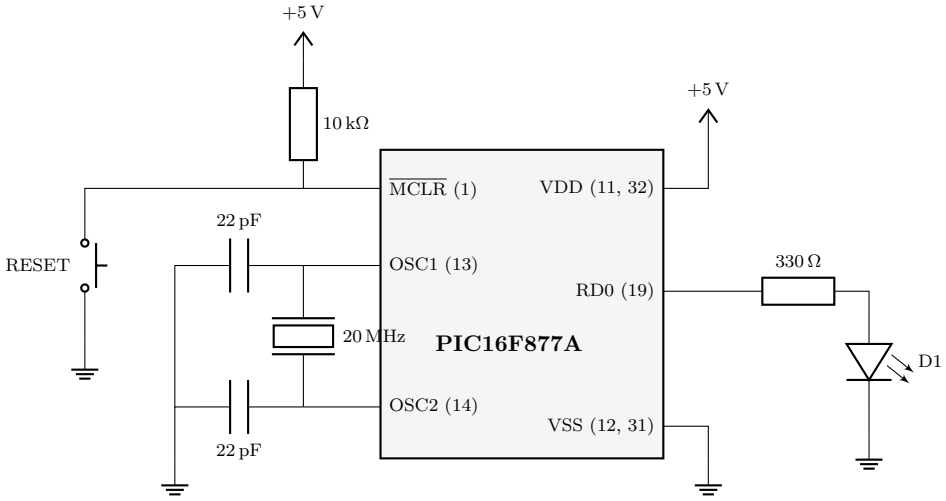


Figure L1.1: Complete circuit of Lab 1: PIC16F877A minimum system and LED on RD0. For the exercises of Part D, connect RD1–RD7 in the same way (one 330 Ω resistor and one LED per pin).

Why the .cof file?

The `.cof` file produced by CCS contains the machine code *and* the link between addresses and C source lines. Proteus loads it, so it can show your C source during debugging. With a `.hex` file alone you would only see disassembled code.

L1.4 Part C — Source-level debugging

1. Start the simulation and press **Pause**. Proteus switches to the **Source Code** tab and highlights the current C line. Open **Debug** → **PIC CPU Registers**.
2. Put a breakpoint on the line `output_low(LED);` inside the loop (click in the margin to the left of the line). Run: the simulation stops on the breakpoint. Use **Step Over** (F10) and observe the LED and the PORTD register.
3. Open **Debug** → **Watch Window**, right-click **Add Items (By Name)** and add PORTD and TRISD. Note their values when the LED is ON and OFF.
4. Place an **Oscilloscope** (**Virtual Instruments Mode**) on RD0. Measure the period.

	LED ON	LED OFF
PORTD (hex)	_____	_____
TRISD (hex)	_____	_____

Measured period: _____

L1.5 Part D — Exercises

1. Connect 8 LEDs to PORTD (RD0–RD7) and write a **running light**: one LED ON moving from RD0 to RD7 every 200 ms, then back ("Knight Rider"). Use `output_d()` and the shift operators `«` and `»`.
2. Write an 8-bit **binary counter** on PORTD incrementing every 250 ms.

3. Change the **Processor Clock Frequency** of the PIC to 4 MHz *only*, without changing `#use delay`. What happens to the blink period? Explain.
4. Change the fuse `HS` to `XT` and simulate. Proteus does not complain; would the real board work at 20 MHz? Why?

L1.6 Questions

1. What does `#use delay(clock=20000000)` do? What goes wrong if the value differs from the real crystal?
2. Why does `main()` contain an infinite loop?
3. What is the role of each fuse in the program?

What to hand in

The Proteus project folder (`.pdsprj` with its firmware sub-folder: `main.c`, `.hex`, `.cof`), the running-light and counter programs, the completed tables and answers. Instructor's signature: _____

LAB 2

Case Study 1: Controlling LEDs with Switches

Week:	4	**Duration:**	4 hours
Platform:	PIC16F877A · CCS C · Proteus 9.2 SP1		
Assessment:	Checked at end of session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Read digital inputs with pull-up resistors and drive digital outputs.
- Debounce switches and detect edges.
- Implement a small state machine that selects LED patterns.

Pre-lab work (to be done before the session)

1. Read Sections 2.1–2.3.
2. Draw the circuit of Figure L2.1 with the component values.
3. With active-low switches, what does `input_b() & 0x0F` return when only S2 (RB2) is pressed?

L2.1 Circuit

Part (Proteus name)	Connection	Note
`PIC16F877A`	clock 20 MHz	
4 × `BUTTON` (S0–S3)	RB0–RB3 to GND	internal PORTB pull-ups
8 × `LED-RED`/`LED-GREEN`	RD0–RD7 via 330 Ω to GND	source mode
`RES`	330 Ω × 8	

L2.2 Part A — Direct mirror

Each switch S0–S3 lights the corresponding LED RD0–RD3 while pressed.

File: `Lab2_mirror.c` attached to this PDF

```
#include <16F877A.h>
#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT, PUT
#use delay(clock = 20000000)

void main(void)
{
   int8 sw;
```

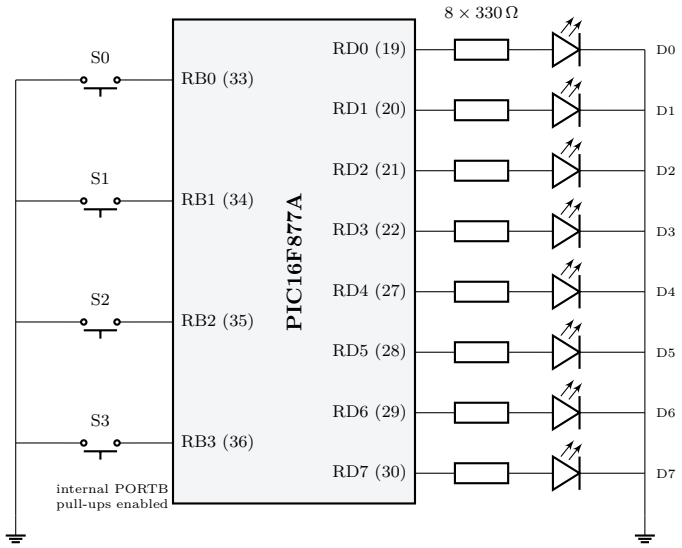


Figure L2.1: Complete circuit of Lab 2 (Case Study 1): four push-buttons on RB0–RB3 and eight LEDs on RD0–RD7. Supply, crystal and $\overline{\text{MCLR}}$ as in Figure L1.1.

```
8     port_b_pullups(TRUE);          // RB0..RB7 weak pull-ups: open = 1, pressed = 0
9     set_tris_d(0x00);              // PORTD all outputs
10    output_d(0x00);
11    while (TRUE)
12    {
13       sw = ~input_b() & 0x0F;     // invert (active-low) and keep RB0..RB3
14       output_d(sw);
15    }
16 }
```

Task: simulate and fill the table.

S3	S2	S1	S0	PORTD (binary)
0	0	0	1	_____
0	1	0	1	_____
1	1	1	1	_____

(1 = pressed)

L2.3 Part B — Toggle with debouncing

Modify the program so that each *press* of S0 toggles RD0 (press once: ON; press again: OFF). First write it *without* debouncing and without waiting for release: what happens when you keep the button pressed? Then use the debounce pattern of Section 2.3.

Simulating a bouncing switch

Proteus buttons do not bounce. To observe the problem, hold the button down: without edge detection the LED toggles continuously. On the real board the bounce makes the toggle unreliable even with a short press.

L2.4 Part C — Pattern selector (state machine)

Write a program where the switches select the pattern displayed on the 8 LEDs:

Switch	Mode	Pattern (every 150 ms)
S0	0	running light left (RD0 → RD7)
S1	1	running light right (RD7 → RD0)
S2	2	all LEDs flash (0x00 / 0xFF)
S3	3	8-bit binary counter

Skeleton:

File: `Lab2_patterns_skeleton.c` attached to this PDF

```
int8 mode = 0, pattern = 1;

void read_switches(void)
{
   if (!input(PIN_B0)) mode = 0;
   if (!input(PIN_B1)) mode = 1;
   if (!input(PIN_B2)) mode = 2;
   if (!input(PIN_B3)) mode = 3;
}

void main(void)
{
   port_b_pullups(TRUE);
   while (TRUE)
   {
      read_switches();
      switch (mode)
      {
         case 0: pattern = (pattern == 0x80) ? 0x01 : pattern << 1; break;
         case 1: /* to complete */ break;
         case 2: /* to complete */ break;
         case 3: /* to complete */ break;
      }
      output_d(pattern);
      delay_ms(150);
   }
}
```

Tasks: complete the cases. Then explain why a very short press may be missed, and fix it by replacing `delay_ms(150)` with 15 iterations of `delay_ms(10)` that each call `read_switches()`.

L2.5 Questions

1. Why are the PORTB pull-ups needed? What would the LEDs show without them?
2. What is the difference between reading a *level* and detecting an *edge*?
3. What is the worst-case reaction time to a switch in Part C (original version and fixed version)?
4. How could the RB0/INT interrupt be used to react immediately to S0?

What to hand in

The three programs, the Proteus project, the completed table and answers. Instructor's signature: _____

LAB 3

Case Study 2: LM35 Temperature Monitoring with LEDs

Week:	5	**Duration:**	4 hours
Platform:	PIC16F877A · CCS C · Proteus 9.2 SP1		
Assessment:	Checked at end of session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Configure and use the 10-bit ADC of the PIC16F877A.
- Convert ADC readings into temperature and evaluate the measurement resolution and error.
- Display a temperature range on an LED bar graph and implement an alarm with hysteresis.

Pre-lab work (to be done before the session)

1. Read Section 2.6 (ADC and LM35).
2. Compute the ADC values expected at 20, 30 and 45 °C (10 bits, $V_{REF} = 5\,\text{V}$).
3. The bar graph shows one LED per 5 °C above 15 °C. How many LEDs are lit at 32 °C?
4. Explain in two sentences what *hysteresis* is and why an alarm needs it.

L3.1 Circuit

Connect the LM35 to AN0 as in Figure 2.6a and the eight bar-graph LEDs to PORTD in source mode (Figure 2.2a). The complete circuit is shown in Figure L3.1 and the parts are listed below.

Part (Proteus name)	Connection	Note
`LM35`	$V_{OUT} \rightarrow$ RA0/AN0 (pin 2); $+V_S \rightarrow 5\,\text{V}$; GND	adjust temperature with its arrows
8 × `LED-*` (green, yellow, red)	RD0–RD7 via 330 Ω	bar graph
`LED-RED` (alarm)	RC0 via 330 Ω	
`BUZZER` (optional)	RC1	active buzzer model
`DC VOLTMETER`	on LM35 output	reference measurement

L3.2 Program

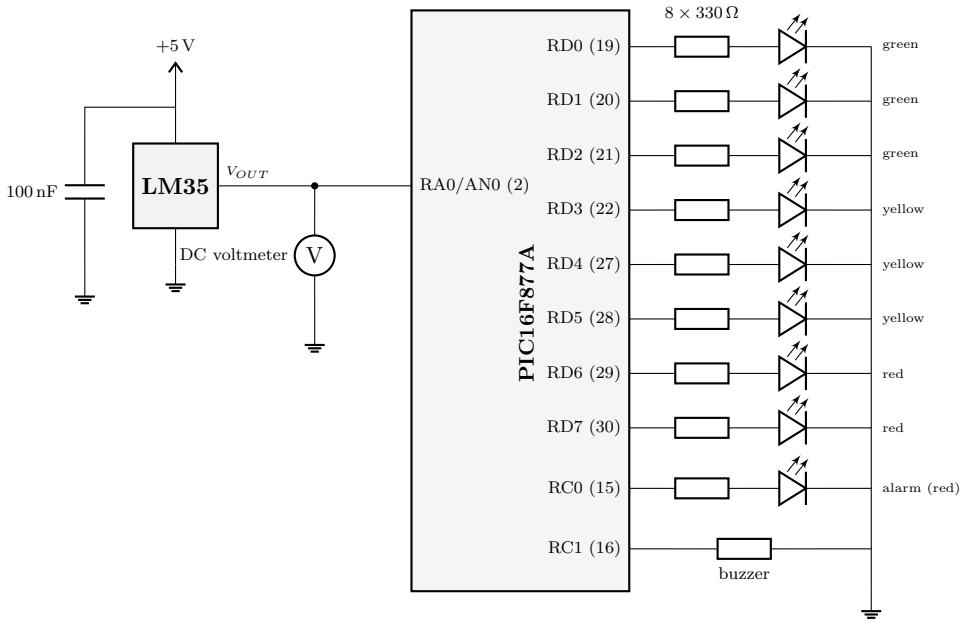


Figure L3.1: Complete circuit of Lab 3 (Case Study 2): LM35 on AN0 with a reference voltmeter, LED bar graph on PORTD, alarm LED on RC0 and optional buzzer on RC1. Supply, crystal and $\overline{\text{MCLR}}$ as in Figure L1.1.

File: `Lab3_lm35_thermometer.c` attached to this PDF

```c
#include <16F877A.h>
#device ADC=10                    // 10-bit results (default would be 8-bit!)
#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT, PUT
#use delay(clock = 20000000)

#define ALARM_LED   PIN_C0
#define T_ALARM_ON  40.0          // degC
#define T_ALARM_OFF 38.0          // hysteresis of 2 degC

float read_temperature(void)
{
   int16 sum = 0;
   int8 i;
   for (i = 0; i < 8; i++) {      // average 8 conversions
      sum += read_adc();
      delay_us(50);
   }
   return (sum / 8.0) * 500.0 / 1023.0;   // 10 mV/degC, Vref = 5 V
}

int8 bargraph(float t)
{
   int8 n, bits = 0;
   if (t < 15.0) return 0;
   n = (int8)((t - 15.0) / 5.0) + 1;      // one LED per 5 degC above 15 degC
   if (n > 8) n = 8;
   while (n--) bits = (bits << 1) | 1;    // n ones: 0x01, 0x03, 0x07 ...
   return bits;
}

void main(void)
{
   float t;
   int1 alarm = 0;

```

```
36    setup_adc_ports(AN0);            // RA0 analogue, others digital
37    setup_adc(ADC_CLOCK_DIV_32);     // Tad = 1.6 us at 20 MHz
38    set_adc_channel(0);
39    delay_us(20);                    // acquisition time
40
41    while (TRUE)
42    {
43       t = read_temperature();
44       output_d(bargraph(t));
45
46       if (!alarm && t >= T_ALARM_ON)  alarm = 1;
47       if ( alarm && t <= T_ALARM_OFF) alarm = 0;
48       output_bit(ALARM_LED, alarm);
49
50       delay_ms(200);
51    }
52 }
```

L3.3 Measurements

Set the LM35 to the temperatures of the table (click its arrows). Read the voltmeter and the number of LEDs lit. Use the watch window (add the variable `t` — it requires the `.cof` file) to read the computed temperature.

T_{set} (°C)	V_{LM35} (mV)	ADC expected	t computed	Error (°C)	LEDs lit
10					
20					
27					
35					
39					
41					
55					

Hysteresis test: raise the temperature from 35 to 42 °C then lower it to 36 °C. At which temperatures does the alarm turn ON and OFF? _____

L3.4 Exercises

1. Replace the bar graph by a "thermometer" with three zones: green LEDs below 25 °C, yellow 25–35 °C, red above 35 °C.
2. Add a second LM35 on AN1 and display on the bar graph the *difference* of the two temperatures. (Use `setup_adc_ports(AN0_AN1_AN3)` and switch channels.)
3. Improve the resolution: connect a 2.5 V reference to RA3/VREF+ (`AN0_AN1_VSS_VREF`) and adapt the conversion formula. What is the new resolution and the new maximum temperature?
4. Replace the floating-point computation by integer arithmetic (tenths of degree, `int16`/`int32`). Compare the ROM usage reported by the compiler.

L3.5 Questions

1. Why is `#device ADC=10` necessary?

2. What is the theoretical resolution of the measurement in °C? Is the averaging useful in simulation? On the real board?
3. Why does the alarm need two thresholds?

What to hand in

Programs, Proteus project, measurement table, answers. Instructor's signature: _____

LAB 4

Case Study 3: DS18B20 Temperature Monitoring with an LCD

Week:	6	**Duration:**	4 hours
Platform:	PIC16F877A · CCS C · Proteus 9.2 SP1		
Assessment:	Checked at end of session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Implement the 1-Wire protocol with precise software timing.
- Read and convert the two's-complement result of a DS18B20.
- Use the CCS LCD driver in 4-bit mode and format numbers with `printf`.

Pre-lab work (to be done before the session)

1. Read Sections 2.7 (LCD) and 2.8 (1-Wire).
2. Draw the timing diagrams of the reset/presence sequence, a write-0, a write-1 and a read slot.
3. Convert the raw values 0x0550, 0x0008 and 0xFE6F into °C.

L4.1 Circuit

Connect the LCD to PORTD in 4-bit mode as in Figure 2.7 and the DS18B20 to RB0 with its 4.7 kΩ pull-up resistor as in Figure 2.8 (one sensor only). The complete circuit is shown in Figure L4.1 and the parts are listed below.

Part (Proteus name)	Connection	Note
`DS18B20`	DQ → RB0; VDD 5 V; GND	4.7 kΩ pull-up DQ–5 V (`RES`)
`LM016L` (16×2 LCD)	E = RD0, RS = RD1, RW = RD2, D4–D7 = RD4–RD7	D0–D3 unconnected
`POT-HG` 10 kΩ (optional)	VEE contrast	not needed in simulation
`LED-RED`	RC0 via 330 Ω	"sensor error" indicator

L4.2 Program

File: `Lab4_ds18b20_lcd.c` attached to this PDF

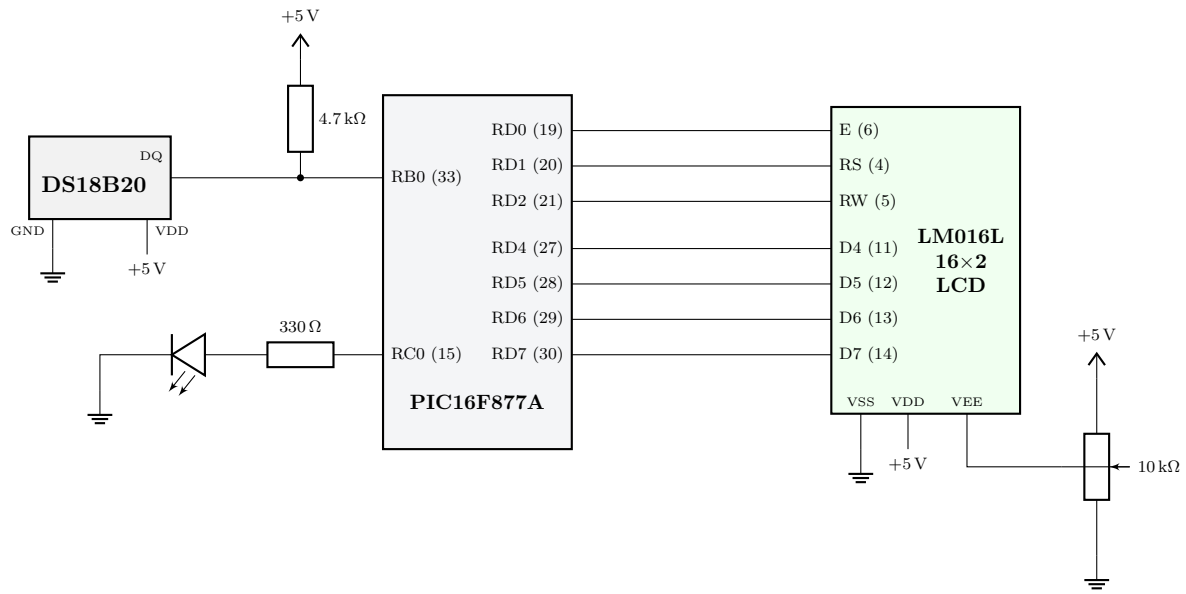


Figure L4.1: Complete circuit of Lab 4 (Case Study 3): DS18B20 on RB0 with its pull-up resistor, 16×2 LCD on PORTD in 4-bit mode, error LED on RC0. Supply, crystal and $\overline{\text{MCLR}}$ as in Figure L1.1.

```c
#include <16F877A.h>
#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT, PUT
#use delay(clock = 20000000)

// ---- LCD on PORTD (must be defined BEFORE including lcd.c) ----
#define LCD_ENABLE_PIN PIN_D0
#define LCD_RS_PIN     PIN_D1
#define LCD_RW_PIN     PIN_D2
#define LCD_DATA4      PIN_D4
#define LCD_DATA5      PIN_D5
#define LCD_DATA6      PIN_D6
#define LCD_DATA7      PIN_D7
#include <lcd.c>

#define DQ        PIN_B0
#define ERR_LED   PIN_C0

// ---------------- 1-Wire primitives ----------------
int1 ow_reset(void)                 // returns 1 if a device answered
{
   int1 presence;
   output_low(DQ);
   delay_us(500);                   // reset pulse >= 480 us
   output_float(DQ);                // release: pull-up brings DQ high
   delay_us(70);
   presence = !input(DQ);           // device pulls low = present
   delay_us(430);                   // end of presence time slot
   return presence;
}

void ow_write_bit(int1 b)
{
   output_low(DQ);
   if (b) { delay_us(3);  output_float(DQ); delay_us(60); }
   else   { delay_us(62); output_float(DQ); delay_us(2);  }
}

int1 ow_read_bit(void)
```

```
{
   int1 b;
   output_low(DQ);
   delay_us(2);
   output_float(DQ);
   delay_us(10);                    // sample before 15 us
   b = input(DQ);
   delay_us(50);
   return b;
}

void ow_write_byte(int8 v)
{
   int8 i;
   for (i = 0; i < 8; i++) { ow_write_bit(v & 0x01); v >>= 1; }   // LSB first
}

int8 ow_read_byte(void)
{
   int8 i, v = 0;
   for (i = 0; i < 8; i++) { v >>= 1; if (ow_read_bit()) v |= 0x80; }
   return v;
}

// ---------------- DS18B20 ----------------
int1 ds18b20_read(float *t)
{
   int8 lsb, msb;
   signed int16 raw;

   if (!ow_reset()) return 0;
   ow_write_byte(0xCC);             // Skip ROM (single sensor)
   ow_write_byte(0x44);             // Convert T
   delay_ms(750);                   // 12-bit conversion time

   if (!ow_reset()) return 0;
   ow_write_byte(0xCC);
   ow_write_byte(0xBE);             // Read scratchpad
   lsb = ow_read_byte();
   msb = ow_read_byte();

   raw = (signed int16)make16(msb, lsb);
   *t = raw * 0.0625;
   return 1;
}

void main(void)
{
   float t, tmin = 125.0, tmax = -55.0;

   lcd_init();
   printf(lcd_putc, "\fCEC405  DS18B20");
   delay_ms(1000);

   while (TRUE)
   {
      if (ds18b20_read(&t))
      {
         output_low(ERR_LED);
         if (t < tmin) tmin = t;
         if (t > tmax) tmax = t;
         lcd_gotoxy(1, 1);
         printf(lcd_putc, "T = %6.2f%cC  ", t, 223);     // 223 = degree sign
         lcd_gotoxy(1, 2);
```

```
103            printf(lcd_putc, "m%5.1f M%5.1f ", tmin, tmax);
104         }
105         else
106         {
107            output_high(ERR_LED);
108            printf(lcd_putc, "\fSensor error!");
109         }
110      }
111   }
```

Timing

1-Wire timing is critical. The CCS `delay_us()` is exact only if `#use delay` matches the real clock. If interrupts are used elsewhere, disable them during each time slot (`disable_interrupts(GLOBAL)` ... `enable_interrupts(GLOBAL)`).

L4.3 Measurements and tests

1. Simulate. Change the DS18B20 temperature with its arrows and complete the table.
2. Disconnect DQ (delete the wire) while the simulation runs: what happens?
3. Place a **Digital Oscilloscope** or **Logic Analyser** on DQ. Capture a reset/presence sequence and measure the reset pulse and presence pulse durations; compare them with Figure 2.9.

T_{set} (°C)	Raw value expected (hex)	LCD reading	Error
25			
−5			
85			
−30.5			

Reset pulse	_____
Presence pulse	_____

L4.4 Exercises

1. Add a push-button on RB1 that resets the min/max values.
2. Read the 64-bit ROM code with the `0x33` (Read ROM) command and display it in hexadecimal. Verify that the first byte is the family code 0x28.
3. Configure 9-bit resolution (write the configuration register with `0x4E`) and measure the new conversion time required. What is the resolution?
4. *Challenge:* two DS18B20 on the same wire, addressed with `0x55` (Match ROM).

L4.5 Questions

1. Why must DQ never be driven *high* by the microcontroller?
2. Compare the LM35 and the DS18B20 (interface, resolution, accuracy, wiring length, cost of pins).
3. The display flickers if `\f` is used at every refresh. Why? How is it avoided above?

What to hand in

Program, Proteus project, tables, oscilloscope screenshot, answers. Instructor's signature: _____

LAB 5

Case Study 4: Communication Between a Microcontroller and a Terminal (RS232)

Week:	7	**Duration:**	4 hours
Platform:	PIC16F877A · CCS C · Proteus 9.2 SP1 · PC terminal		
Assessment:	Checked at end of session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Configure the USART with `#use rs232` and exchange text with a terminal.
- Write a command interpreter (polling, then interrupt-driven).
- Link a simulated microcontroller to a real PC program through a virtual COM port (preparation for the IoT labs).

Pre-lab work (to be done before the session)

1. Read Sections 2.9 (UART and RS232) and 2.12 (interrupts).
2. How many characters per second can be sent at 9600 8N1? At 115200?
3. Draw the connection of a PIC to a PC DB9 COM port through a MAX232 (pins 2, 3, 5 of the DB9).
4. Install a terminal program on your PC (Tera Term, PuTTY or the Arduino Serial Monitor) and `com0com` (Windows) if you want to do Part C at home.

L5.1 Circuit

In simulation the USART pins are connected directly to the Proteus Virtual Terminal. On a real board a MAX232 level converter is required between the PIC and the PC (Figure 2.11). The frames exchanged have the format of Figure 2.10. The complete simulation circuit is shown in Figure L5.1.

Part	Connection	Note
`VIRTUAL TERMINAL` (Instruments)	RXD ← RC6/TX (pin 25); TXD → RC7/RX (pin 26)	9600, 8, N, 1
`LED-RED`	RD0 via 330 Ω	controlled by commands
`LM35`	AN0	reused from Lab 3
`COMPIM` (Part C)	TXD/RXD to RC6/RC7 (see note)	physical/virtual COM port

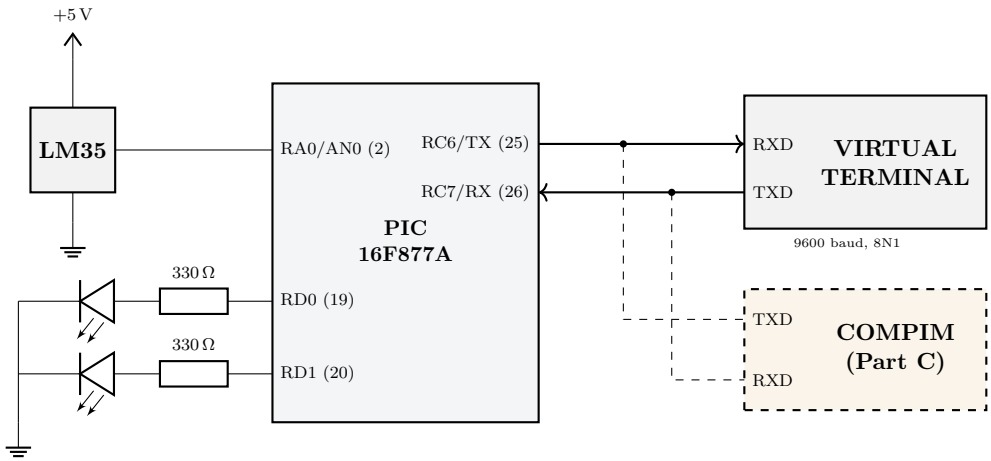


Figure L5.1: Complete circuit of Lab 5 (Case Study 4): USART to the Proteus Virtual Terminal, LM35 on AN0, command LED on RD0 and blinking LED on RD1 (Part B). In Part C the COMPIM (dashed) replaces the terminal; swap its TXD/RXD wires if no data is received. Supply, crystal and $\overline{\text{MCLR}}$ as in Figure L1.1.

L5.2 Part A — Command interpreter (polling)

File: `Lab5_serial_commands.c` attached to this PDF

```c
#include <16F877A.h>
#device ADC=10
#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT, PUT
#use delay(clock = 20000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, parity=N, bits=8, stop=1)

#define LED PIN_D0

void help(void)
{
   printf("\r\n--- CEC405 Lab 5 ---\r\n");
   printf("1 : LED on\r\n0 : LED off\r\nt : temperature\r\nh : this help\r\n> ");
}

float temperature(void)
{
   return read_adc() * 500.0 / 1023.0;
}

void main(void)
{
   char c;
   setup_adc_ports(AN0);
   setup_adc(ADC_CLOCK_DIV_32);
   set_adc_channel(0);
   delay_us(20);

   help();
   while (TRUE)
   {
      c = getc();                          // waits for a character
      putc(c);                             // echo
      switch (c)
      {
         case '1': output_high(LED); printf("\r\nLED ON\r\n> ");  break;
         case '0': output_low(LED);  printf("\r\nLED OFF\r\n> "); break;
```

```
            case 't': printf("\r\nT = %5.1f C\r\n> ", temperature()); break;
            case 'h': help(); break;
            default : printf("\r\n?\r\n> ");
        }
    }
}
```

Tasks: simulate; if the Virtual Terminal window does not appear, open it with **Debug → Virtual Terminal**. Click inside it and type commands. Then answer: while the program waits in `getc()`, can it blink another LED? Why not?

L5.3 Part B — Interrupt-driven reception and periodic reporting

Modify the program so that:

- a second LED on RD1 blinks every 500 ms *continuously*;
- the temperature is sent automatically every 2 s in the format `T:25.4` followed by `\r\n` (one value per line, easy to parse by a PC program);
- the commands are received by the `#int_rda` interrupt (see Section 2.12): the ISR stores the character and sets a flag; the main loop executes the command.

Time base suggestion: a main loop with `delay_ms(10)` and a counter (`if (++ticks == 50)` ...).

L5.4 Part C — Connecting the simulation to a real PC terminal

The `COMPIM` component connects the simulated UART to a COM port of the PC. With **com0com** you create a pair of virtual ports linked by a virtual null-modem cable, e.g. `COM3`↔`COM4`:



1. Install com0com and create the pair (Setup utility). Check in the Device Manager.
2. In Proteus add `COMPIM`; set Physical port = COM3, Physical and Virtual baud rate = 9600, 8 data bits, no parity, 1 stop bit.
3. Wire COMPIM to RC6/RC7. *If nothing is received, swap the TXD and RXD wires*: the pin names of COMPIM follow the DB9 connector convention and are a frequent source of confusion.
4. Open Tera Term on COM4 at 9600 and repeat the tests of Parts A and B.

On a real board

Insert a MAX232 between RC6/RC7 and the DB9 connector as in Figure 2.11 (T1IN ← RC6, R1OUT → RC7, T1OUT → DB9 pin 2, R1IN ← DB9 pin 3, GND → DB9 pin 5), or connect a 5 V TTL USB–serial adapter directly (adapter TX → RC7, adapter RX ← RC6, common GND).

L5.5 Questions

1. What is the difference between `getc()` and `kbhit()`? When must each be used?

2. Why must an ISR be short? What happens if a second character arrives while the main loop is busy for 5 ms (the PIC USART has a 2-byte receive FIFO)?
3. Why is a line-based, fixed format (`T:25.4\r\n`) preferable for communication with a PC program?
4. Explain the roles of the MAX232 and of its four capacitors.

What to hand in

Programs of Parts A and B, Proteus project, terminal screenshot of Part C, answers. Instructor's signature: _____

LAB 6

Case Study 5: Arduino Projects in Proteus — Source Code and Flowchart

Week:	8	**Duration:**	4 hours
Platform:	Arduino Uno · Proteus 9.2 SP1 (built-in compiler, Visual Designer)		
Assessment:	Checked; mini project issued		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Move from the PIC/CCS environment to the Arduino environment and compare them.
- Simulate an Arduino Uno in Proteus with (a) C++ source code and (b) a Visual Designer flowchart.
- Write non-blocking code organised as a finite-state machine.

Pre-lab work (to be done before the session)

1. Read Section 2.12 (non-blocking timing with `millis()`).
2. Give the Arduino equivalents of `output_high`, `input`, `delay_ms`, `read_adc` and `printf` (see Appendix A).
3. Draw the state diagram of the pedestrian crossing described in Part A.

L6.1 From CCS C to Arduino

An Arduino *sketch* is C++ compiled by `avr-g++`. The IDE hides `main()`: it calls `setup()` once and then `loop()` forever. Pins are numbered as on the board (D0–D13, A0–A5) instead of port bits. Libraries (`LiquidCrystal`, `Servo`, `OneWire`, `DallasTemperature`, ...) replace the drivers you wrote in Labs 1–5.

	CCS C / PIC16F877A	**Arduino / ATmega328P**
Program structure	`main()` with `while(TRUE)`	`setup()` + `loop()`
Clock / fuses	`#fuses`, `#use delay`	fixed by the board (16 MHz), bootloader
Digital I/O	`output_high(PIN_D0)`	`pinMode(13,OUTPUT);` `digitalWrite(13,HIGH)`
ADC	`setup_adc...; read_adc()`	`analogRead(A0)` (10 bits, 0–1023)
Serial	`#use rs232`, `printf`, `getc`	`Serial.begin`, `print`, `read`
Time	`delay_ms()` only	`delay()`, `millis()`, `micros()`
Compiler in Proteus 9.2	CCS C, called by Proteus	Proteus built-in Arduino compiler
Output files	`.hex`, `.cof`	`.hex`, `.elf`

L6.2 Part A — Source code project: pedestrian crossing

Specification

A road crossing has car lights (red D10, yellow D9, green D8), pedestrian lights (red D12, green D11) and a request push-button on D2 (to GND, internal pull-up). Cars have green by default. When a pedestrian presses the button, cars keep green for at least 5 s after the last change, then yellow 2 s, then red; pedestrians get green for 6 s, then all red for 2 s (clearance), then cars get green again. The current state is printed on the serial monitor. Figure L6.2 gives the corresponding state diagram, implemented by the program below, and Figure L6.1 the complete circuit.

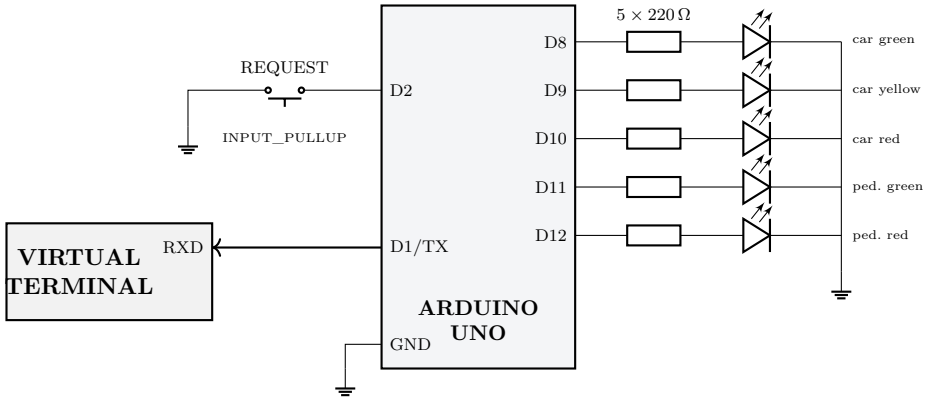


Figure L6.1: Complete circuit of Lab 6 (Case Study 5): car and pedestrian lights on D8–D12, request button on D2, state messages on D1/TX. The same circuit is used for the flowchart project of Part B.

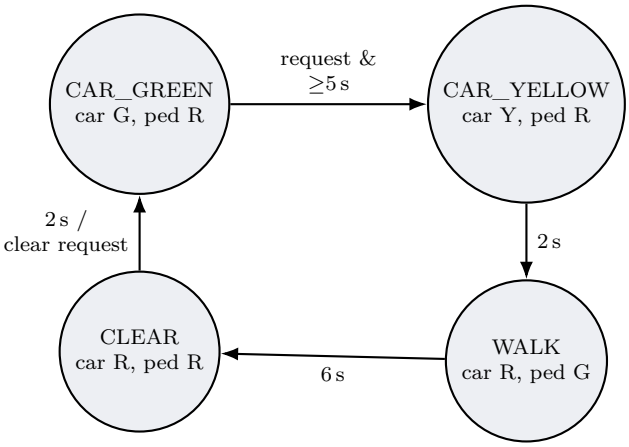


Figure L6.2: State diagram of the pedestrian crossing.

File: `Lab6_crossing.ino` attached to this PDF

```
// CEC405 Lab 6A - pedestrian crossing, non-blocking state machine
const byte CAR_G = 8, CAR_Y = 9, CAR_R = 10, PED_G = 11, PED_R = 12, BTN = 2;

enum State { CAR_GREEN, CAR_YELLOW, WALK, CLEAR };
State state;
unsigned long t0;            // time of entry into the current state
bool request = false;

void lights(byte cg, byte cy, byte cr, byte pg, byte pr) {
  digitalWrite(CAR_G, cg); digitalWrite(CAR_Y, cy); digitalWrite(CAR_R, cr);
  digitalWrite(PED_G, pg); digitalWrite(PED_R, pr);
```

```
}

void enter(State s) {
  state = s;
  t0 = millis();
  switch (s) {
    case CAR_GREEN:  lights(1,0,0, 0,1); Serial.println(F("CAR_GREEN"));  break;
    case CAR_YELLOW: lights(0,1,0, 0,1); Serial.println(F("CAR_YELLOW")); break;
    case WALK:       lights(0,0,1, 1,0); Serial.println(F("WALK"));       break;
    case CLEAR:      lights(0,0,1, 0,1); Serial.println(F("CLEAR"));      break;
  }
}

void setup() {
  for (byte p = 8; p <= 12; p++) pinMode(p, OUTPUT);
  pinMode(BTN, INPUT_PULLUP);
  Serial.begin(9600);
  enter(CAR_GREEN);
}

void loop() {
  if (digitalRead(BTN) == LOW) request = true;     // remember the request
  unsigned long dt = millis() - t0;                // overflow-safe

  switch (state) {
    case CAR_GREEN:  if (request && dt >= 5000) enter(CAR_YELLOW); break;
    case CAR_YELLOW: if (dt >= 2000) enter(WALK);  break;
    case WALK:       if (dt >= 6000) enter(CLEAR); break;
    case CLEAR:      if (dt >= 2000) { request = false; enter(CAR_GREEN); } break;
  }
}
```

Method 1 (used in this course) — Proteus firmware project, built-in Arduino compiler

Proteus 9.2 SP1 has its own Arduino compiler: the Arduino IDE is *not* needed to write, compile, simulate or debug the sketch.

1. **File** → **New Project**, name `Lab6A`; default schematic; no PCB.
2. On the *Firmware* page choose **Create Firmware Project**: family `Arduino`, controller `Arduino Uno`, compiler `Arduino AVR` (the compiler supplied with Proteus). Tick *Create Quick Start Files*. (If the compiler is marked as not configured, click **Compilers...** on the same page to set it up.)
3. Proteus places the board on the schematic and opens the **Source Code** tab with a sketch skeleton. Paste the program, then **Build** → **Build Project**. Note the Flash and RAM usage shown in the output window.
4. In the **Schematic Capture** tab, add 5 LEDs (`LED-RED`, `LED-YELLOW`, `LED-GREEN`) with 220 Ω resistors, a `BUTTON` and a `VIRTUAL TERMINAL` (RXD ← D1/TX). Run.

Method 2 (optional) — compile in the Arduino IDE, simulate in Proteus

Use this method only if you also want to upload the same sketch to a real Uno board, which requires the Arduino IDE.

1. In the Arduino IDE select **Tools** → **Board** → **Arduino Uno**, then **Sketch** → **Export Compiled Binary**. The `.hex` (and `.elf`) files appear in the `build` sub-folder of the sketch.

2. In Proteus place an `ATMEGA328P`. Properties: **Program File** = the exported `.ino.hex` (or `.elf` for source-level debugging), **CKSEL Fuses** = external crystal, **CKDIV8** = unprogrammed, **Clock Frequency** = 16MHz.
3. Arduino pin → ATmega328P pin: D0–D7 = PD0–PD7, D8–D13 = PB0–PB5, A0–A5 = PC0–PC5.

Observations: record the sequence of states printed on the terminal after one button press and the measured durations (use the Proteus **Graph** or the oscilloscope on D9).

L6.3 Part B — Flowchart project (Proteus Visual Designer)

Visual Designer lets you program the Arduino with flowchart blocks and ready-made peripheral models (LEDs, buttons, LCDs, sensors, shields, Grove modules). Proteus generates the C++ code automatically, which you can inspect.

1. **File** → **New Flowchart** (or New Project → **Create Flowchart Project**), family `Arduino`, controller `Arduino Uno`.
2. In the **Projects** pane, right-click **Arduino Uno** → **Add Peripheral**: add LEDs and a button (e.g. from the *Breakout Peripherals* or *Grove* categories). They appear on the schematic and their methods (`on()`, `off()`, `read()`...) appear in the Projects tree.
3. The flowchart already contains two event blocks: `SETUP` and `LOOP`. Drag *method calls* from the Projects tree onto the flowchart, add *Delay*, *Decision*, *Assignment* and *Loop* blocks from the toolbox and connect them.
4. Use the circuit of Figure L6.1 (LEDs and button added as peripherals). Implement a **simplified** crossing (blocking version): LOOP: car green; wait until the button is pressed (decision block looping on itself); delay 3 s; yellow 2 s; red + walk 6 s; red 2 s. The expected flowchart is shown in Figure L6.3.
5. Run. Then open the generated code (**Source Code** view) and compare it with your Part A code.

L6.4 Questions

1. Why can the Part B version not react to the button during the 6 s walk phase, while the Part A version could (if required)? Which version satisfies the specification?
2. Why is `millis() - t0 >= 5000` correct even when `millis()` overflows after 49.7 days, whereas `millis() >= t0 + 5000` is not?
3. Give two advantages and two limitations of flowchart programming.
4. Compare the ROM/RAM used by your Part A sketch (Proteus build output) with a CCS program of similar function. Comment.

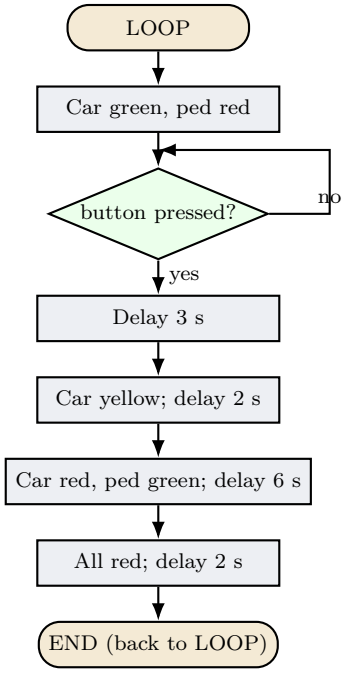


Figure L6.3: Flowchart of the simplified (blocking) crossing.

What to hand in

Lab6A sketch and Proteus project, Lab6B flowchart project, answers. Instructor's signature: _____

L6.5 Mini project brief (continuous assessment, 30%)

Organisation

- Groups of 2–3 students. Project chosen or allocated in week 8; assessment in week 9.
- Platform: Arduino Uno or PIC16F877A, simulated in Proteus. A hardware prototype earns a bonus.

- Each group delivers (1) the Proteus project, (2) the commented source code *or* the Visual Designer flowchart, (3) a 5–8 page report, (4) a 10-minute demonstration with questions to every member.

Project list

Ideas adapted from M. R. Thakur, *Arduino Projects Vol. I*, and from local needs. Every project must include at least one sensor or input device, one display, one actuator and non-blocking code.

No.	Project	Minimum requirements
1	Digital code lock	4×3 keypad, LCD, relay (door), 3 wrong attempts → alarm and lock-out, code changeable and stored in EEPROM
2	Temperature controller	LM35/DS18B20, LCD, two set-point buttons, relay (heater) with hysteresis, set point in EEPROM
3	Object counter	IR/ultrasonic sensor, LCD, count up/down, reset button, buzzer at a preset count
4	DC digital voltmeter	0–30 V via divider, auto-range or two ranges, LCD, over-voltage alarm
5	Water-level controller	4 level probes, pump relay, dry-run protection, level shown on LCD/LEDs
6	Automatic street-light controller	LDR, hysteresis, PWM dimming after midnight (simulated clock), manual override
7	Solar power monitor	voltage and current (ACS712 or shunt) of a PV panel, power and energy (Wh) on LCD, serial logging
8	Ultrasonic distance meter / parking aid	HC-SR04, LCD, buzzer beeping faster as distance decreases
9	Programmable timer	keypad or buttons, LCD countdown, relay output, start/pause/reset
10	Automatic irrigation	soil-moisture sensor, pump relay, LCD, watering only at night (LDR)
11	RPM meter	IR/hall pulse input with interrupt, RPM on LCD, over-speed alarm
12	Prepaid energy meter (simplified)	pulse input from a meter, credit entered by keypad, relay cut-off at zero credit, LCD

Report structure

1. Problem statement and specification 2. Block diagram and choice of components (with calculations: resistors, dividers, transistor drivers) 3. Circuit diagram 4. Flowchart / state diagram 5. Code description (main functions) 6. Tests and results (screenshots, measurement tables) 7. Limitations and improvements 8. Contribution of each member.

Assessment grid (week 9)

Criterion	**Marks**	**Score**
Analysis, specification and design (block diagram, calculations, flowchart)	20	
Simulation works and meets all minimum requirements	30	
Code quality (structure, comments, non-blocking design, naming)	15	
Report (clarity, completeness, figures, tests)	15	
Demonstration and individual oral questions	20	
Total (/100, scaled to /30)	**100**	
Bonus: working hardware prototype	+10	

LAB 7

Case Studies 6 and 7: Web-Based IoT Project with Arduino and Proteus

Week:	10 (Part A) and 11 (Part B)	**Duration:**	2 × 4 hours
Platform:	Arduino Uno · Proteus 9.2 SP1 · com0com · Python/Flask		
Assessment:	Checked at end of each session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Explain the architecture of an IoT system: device, gateway, server, client.
- Define a simple, robust message format between a microcontroller and a PC (JSON lines).
- Publish live sensor data on a web page and control actuators from a browser (any phone or PC on the same network).

L7.1 Background

IoT architecture used in Labs 7 and 8

A small microcontroller like the ATmega328P has no network interface. In real products it is connected to the network either by a communication module (Ethernet W5100/ENC28J60 via SPI, Wi-Fi ESP8266/ESP32 via UART, GSM/LTE modem) or through a *gateway*: a more powerful computer that speaks to the device over a local link and to the Internet over TCP/IP. We use the gateway approach because it can be simulated completely: the Arduino runs in Proteus, its UART is exported through COMPIM and a virtual COM port, and a Python program on the PC acts as the gateway. Figure L7.1 shows the complete chain.

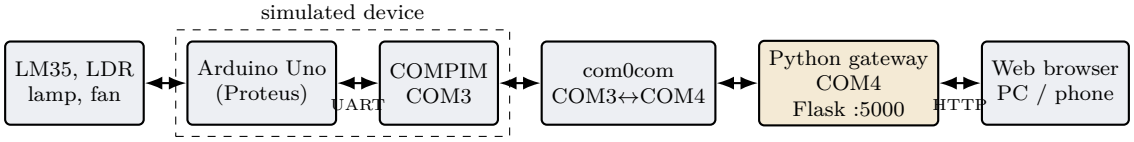


Figure L7.1: Web-based IoT architecture of Lab 7.

HTTP in five lines

The browser (client) sends a *request* to the server: a method (`GET` to read, `POST` to send data), a path (`/api/data`) and optional headers and body. The server returns a *response*: a status code (200 OK, 400 Bad Request, 404 Not Found), headers and a body (HTML for pages, JSON for data). A web page can request data in the background with JavaScript `fetch()` and update itself without reloading: this is how our dashboard refreshes every two seconds.

Device–gateway protocol

- Device → gateway, every 2 s, one JSON object per line: `{"t":27.4,"light":612,"lamp":0,"fan":1}`
- Gateway → device, one command per line: `LAMP=1`, `LAMP=0`, `FAN=1`, `FAN=0`, `GET` (send data now).

Pre-lab work (to be done before the session)

1. Install Python 3, then: `pip install pyserial flask paho-mqtt`.
2. Install com0com and create the pair COM3↔COM4 (any free pair will do; note yours).
3. Explain the difference between `GET` and `POST`. What is JSON?
4. Find the IP address of your PC (`ipconfig` on Windows).

L7.2 The device: Arduino firmware (used in Labs 7 and 8)

Part (Proteus)	Connection	Role
`LM35`	A0	temperature
`TORCH_LDR` (or `LDR`) + 10 kΩ	divider to A1	light level
`LED-YELLOW` + 220 Ω	D13	"lamp"
`LED-BLUE` + 220 Ω (or relay + motor via BC547)	D8	"fan"
`COMPIM`	D0/RX, D1/TX	to COM3, 9600 8N1
`VIRTUAL TERMINAL` (optional)	RXD on D1/TX	spy on the traffic

File: `Lab7_iot_device.ino` attached to this PDF

```
// CEC405 Labs 7-8 - IoT device firmware
const byte LAMP = 13, FAN = 8;
const unsigned long PERIOD = 2000;      // report period (ms)
unsigned long last = 0;
String rx;                              // incoming command line

float readTemp() {
  long s = 0;
  for (byte i = 0; i < 8; i++) s += analogRead(A0);
  return s / 8.0 * 500.0 / 1023.0;      // LM35: 10 mV/degC, Vref 5 V
}

void sendData() {
  Serial.print(F("{\"t\":"));     Serial.print(readTemp(), 1);
  Serial.print(F(",\"light\":")); Serial.print(analogRead(A1));
  Serial.print(F(",\"lamp\":"));  Serial.print(digitalRead(LAMP));
  Serial.print(F(",\"fan\":"));   Serial.print(digitalRead(FAN));
  Serial.println('}');
}

void handle(String cmd) {
  cmd.trim();
  if      (cmd == "LAMP=1") digitalWrite(LAMP, HIGH);
  else if (cmd == "LAMP=0") digitalWrite(LAMP, LOW);
  else if (cmd == "FAN=1")  digitalWrite(FAN, HIGH);
  else if (cmd == "FAN=0")  digitalWrite(FAN, LOW);
  else if (cmd != "GET")    { Serial.println(F("{\"error\":\"unknown command\"}")); return; }
```

```
  sendData();                                   // immediate feedback
}

void setup() {
  pinMode(LAMP, OUTPUT);
  pinMode(FAN, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  while (Serial.available()) {                  // non-blocking line reader
    char c = Serial.read();
    if (c == '\n')            { handle(rx); rx = ""; }
    else if (c != '\r' && rx.length() < 32) rx += c;
  }
  if (millis() - last >= PERIOD) { last += PERIOD; sendData(); }
}
```

The complete circuit of the device is shown in Figure L7.2.

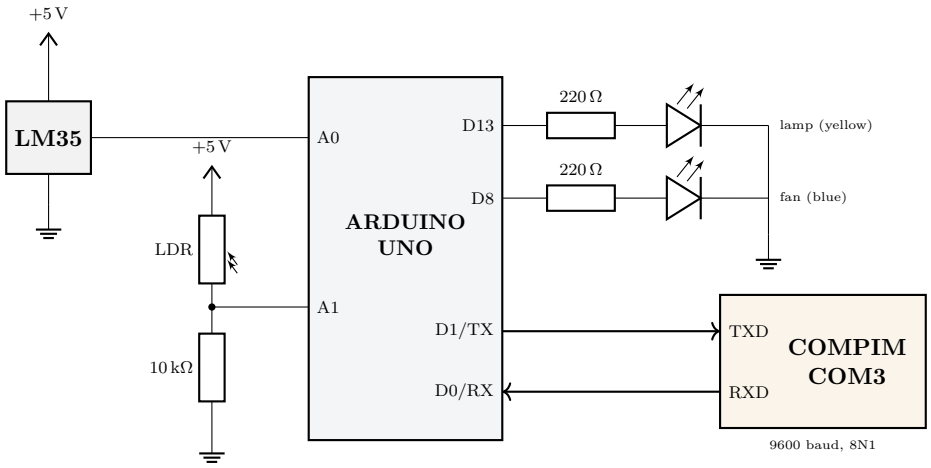


Figure L7.2: Complete circuit of the IoT device used in Labs 7 and 8 (Case Studies 6–9): LM35 on A0, LDR divider on A1, lamp LED on D13, fan LED on D8 and COMPIM on the serial port (swap TXD/RXD if no data is received). For a real fan, replace the D8 LED by the transistor and relay driver of Figure 2.4.

Test before going further: run the simulation with a Virtual Terminal (or Tera Term on COM4). You must see one JSON line every 2 s. Type `LAMP=1` + Enter in Tera Term: the lamp LED turns on.

L7.3 Part A (week 10) — Gateway and live monitoring page

Save as `gateway.py` (change `PORT` to your com0com port):

File: `gateway.py` attached to this PDF

```
# CEC405 Lab 7 - serial-to-web gateway
import json, threading, time
from collections import deque
import serial
from flask import Flask, jsonify, request, render_template_string

PORT, BAUD = "COM4", 9600
ser = serial.Serial(PORT, BAUD, timeout=1)

```

```
state = {"t": None, "light": None, "lamp": 0, "fan": 0, "time": None}
history = deque(maxlen=150)              # last 5 minutes at one sample / 2 s
lock = threading.Lock()

def reader():
    """Background thread: read JSON lines from the device."""
    while True:
        line = ser.readline().decode(errors="ignore").strip()
        if not line.startswith("{"):
            continue
        try:
            data = json.loads(line)
        except ValueError:
            continue                     # ignore corrupted lines
        if "t" not in data:
            print("device:", data)
            continue
        data["time"] = time.strftime("%H:%M:%S")
        with lock:
            state.update(data)
            history.append(dict(data))

threading.Thread(target=reader, daemon=True).start()
app = Flask(__name__)

@app.route("/")
def index():
    return render_template_string(open("page.html", encoding="utf-8").read())

@app.route("/api/data")
def api_data():
    with lock:
        return jsonify(state)

@app.route("/api/history")
def api_history():
    with lock:
        return jsonify(list(history))

@app.route("/api/cmd", methods=["POST"])
def api_cmd():
    body = request.get_json(silent=True) or {}
    dev, value = body.get("dev"), body.get("value")
    if dev not in ("lamp", "fan") or value not in (0, 1):
        return jsonify(ok=False, error="bad command"), 400
    ser.write(f"{dev.upper()}={value}\n".encode())
    return jsonify(ok=True)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)   # reachable from the LAN
```

Save in the same folder as `page.html`:

File: `page.html` attached to this PDF

```
<!doctype html>
<html><head><meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>CEC405 IoT dashboard</title>
<style>
 body{font-family:sans-serif;max-width:480px;margin:auto;padding:16px}
 .card{border:1px solid #ccc;border-radius:8px;padding:12px;margin:8px 0}
 .big{font-size:2.2em;font-weight:bold}
 button{font-size:1em;padding:8px 16px;margin:4px}
</style></head>
```

```
<body>
<h2>CEC405 - Group ___</h2>
<div class="card">Temperature<div class="big"><span id="t">--</span> &deg;C</div></div>
<div class="card">Light level<div class="big" id="light">--</div></div>
<div class="card">Lamp: <b id="lamp">-</b>
  <button onclick="cmd('lamp',1)">ON</button><button onclick="cmd('lamp',0)">OFF</button></div>
<div class="card">Fan: <b id="fan">-</b>
  <button onclick="cmd('fan',1)">ON</button><button onclick="cmd('fan',0)">OFF</button></div>
<small>Last update: <span id="time">-</span></small>
<script>
async function refresh(){
  const d = await (await fetch('/api/data')).json();
  for (const k of ['t','light','time']) document.getElementById(k).textContent = d[k] ?? '--';
  document.getElementById('lamp').textContent = d.lamp ? 'ON' : 'OFF';
  document.getElementById('fan').textContent  = d.fan  ? 'ON' : 'OFF';
}
async function cmd(dev, value){
  await fetch('/api/cmd', {method:'POST', headers:{'Content-Type':'application/json'},
                           body: JSON.stringify({dev, value})});
  setTimeout(refresh, 300);
}
setInterval(refresh, 2000); refresh();
</script>
</body></html>
```

Procedure

1. Start the Proteus simulation *first* (it opens COM3), then run `python gateway.py`.
2. Open `http://localhost:5000`. Vary the LM35 and the LDR torch in Proteus: the page follows.
3. Open `http://localhost:5000/api/data` directly: what do you see?
4. From a phone on the same Wi-Fi network open `http://<PC-IP>:5000`. (Allow Python through the Windows firewall if asked.)
5. Measure the delay between a change in Proteus and its display in the browser (worst case and average). Explain it.

Worst-case delay: _____

Explanation: _____

L7.4 Part B (week 11) — Remote control, history and automation

1. Use the ON/OFF buttons; verify with the LEDs in Proteus. Test from the phone.
2. Send an invalid command with `curl` and note the HTTP status code:
   ```
   curl -X POST -H "Content-Type:  application/json"
       -d "{\"dev\":\"pump\",\"value\":1}" http://localhost:5000/api/cmd
   ```
3. **History chart.** Add to `page.html` a chart of the last 5 minutes of temperature using `/api/history` (an HTML `<canvas>` drawn by hand, or Chart.js loaded from a CDN).
4. **Local automation.** Modify the *Arduino* firmware so that, when an `AUTO=1` command has been received, the fan is switched on above 30 °C and off below 28 °C (hysteresis), and the lamp follows the LDR. Add `"auto":0/1` to the JSON and an AUTO switch to the page.

5. **Robustness.** What happens to the page when the Proteus simulation is stopped? Make the gateway mark the data as *stale* (field `"online": false`) if no line has arrived for 6 s, and show it on the page.

L7.5 Questions

1. Why does the gateway read the serial port in a separate thread?
2. Where should the automation (fan thermostat) run — device, gateway or browser? Discuss reliability when the network or the PC fails.
3. List three security weaknesses of this system and how a real product would address them (authentication, HTTPS, input validation, firmware update…).
4. In a real deployment, which module would you add to the Arduino to remove the PC gateway? Which interface would connect it to the ATmega328P?

What to hand in

Arduino sketch, Proteus project, `gateway.py`, `page.html` (final version), screenshots of the page on a PC and a phone, answers. Instructor's signature (A): _____ (B): _____

LAB 8

Case Studies 8 and 9: MQTT-Based IoT Project with Arduino and Proteus

Week:	12 (Part A) and 13 (Part B)	**Duration:**	2 × 4 hours
Platform:	Arduino Uno · Proteus 9.2 SP1 · Python · Mosquitto		
Assessment:	Checked at end of each session		
Name:	_____	**Matricule:**	_____
Group:	_____	**Date:**	_____

Learning objectives

- Explain the publish/subscribe model and the main MQTT features: topics, wildcards, QoS, retained messages, last will.
- Connect the simulated device to an MQTT broker through a serial–MQTT bridge.
- Build distributed monitoring and control with several independent clients.

L8.1 Background: MQTT

MQTT (Message Queuing Telemetry Transport, ISO/IEC 20922) is a lightweight protocol over TCP (port 1883, or 8883 with TLS) designed for constrained devices and unreliable networks. Figure L8.1 shows the clients used in this lab around the broker.

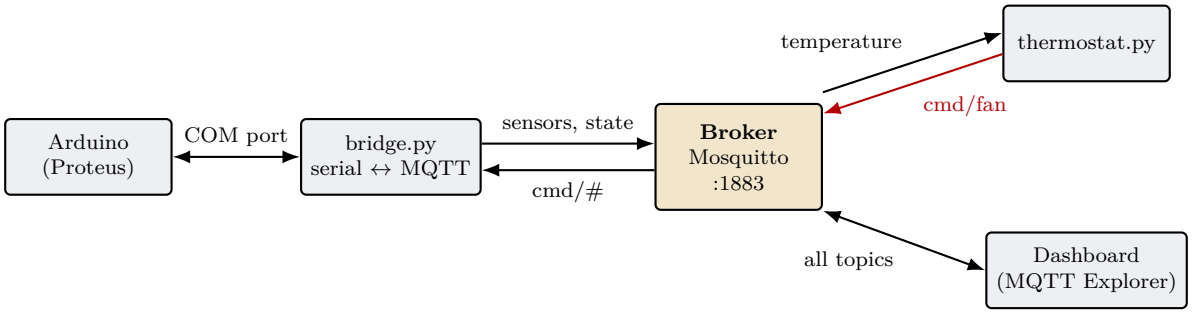


Figure L8.1: Publish/subscribe architecture of Lab 8: every client talks only to the broker.

- **Broker**: the server (Mosquitto, EMQX, HiveMQ) that receives every message and forwards it to the interested clients. Clients never talk to each other directly.
- **Publish/subscribe**: a client *publishes* a message (payload, any bytes) on a *topic*; every client *subscribed* to a matching topic receives it. Publisher and subscribers do not need to know each other or be online at the same time.
- **Topics** are hierarchical strings: `cec405/g01/sensors/temperature`. Wildcards in subscriptions: `+` matches one level (`cec405/+/sensors/temperature`), `#` matches all remaining levels

(`cec405/g01/#`).

- **QoS**: 0 = at most once (fire and forget), 1 = at least once (acknowledged, possible duplicates), 2 = exactly once (four-step handshake).
- **Retained message**: the broker keeps the last retained message of a topic and sends it immediately to new subscribers — ideal for states ("the lamp is ON").
- **Last Will and Testament (LWT)**: a message the broker publishes on behalf of a client that disconnects abnormally (detected through the *keep-alive* timeout) — ideal for "offline" status.

Topic (<g> = your group, e.g. g01)	**Payload**	**Publisher / options**
`cec405/<g>/sensors/temperature`	`27.4`	bridge, QoS 0
`cec405/<g>/sensors/light`	`612`	bridge, QoS 0
`cec405/<g>/state/lamp`, `.../state/fan`	`0` / `1`	bridge, retained
`cec405/<g>/cmd/lamp`, `.../cmd/fan`	`0/1/ON/OFF`	any client, QoS 1
`cec405/<g>/status`	`online` / `offline`	bridge (LWT), retained

Pre-lab work (to be done before the session)

1. Install Mosquitto (`mosquitto.org`) and MQTT Explorer (or another MQTT client).
2. Which subscription receives the temperature of *all* groups? All messages of group g03?
3. Why should a state topic be retained but not a command topic?
4. What is the difference between the HTTP request/response model of Lab 7 and publish/subscribe?

L8.2 Part A (week 12) — Broker, bridge and first clients

1. Start the broker

Mosquitto 2.x accepts only local connections by default. To accept the whole lab network, create `lab.conf`:

File: `lab.conf` attached to this PDF

```
listener 1883
allow_anonymous true
```

and run `mosquitto -c lab.conf -v` (the `-v` option prints every connection and message). In the lab, the instructor's PC may host a shared broker for all groups; its address will be given in class.

2. Test with the command-line clients

In two terminals:

```
mosquitto_sub -h localhost -t "cec405/#" -v
mosquitto_pub -h localhost -t "cec405/g01/test" -m "hello"
```

3. Serial–MQTT bridge

Keep the Arduino firmware and the Proteus circuit of Lab 7 (Figure L7.2). Save as `bridge.py`:

File: `bridge.py` attached to this PDF

```python
# CEC405 Lab 8 - serial-to-MQTT bridge (paho-mqtt 2.x)
import json
import serial
import paho.mqtt.client as mqtt

PORT, BAUD = "COM4", 9600
BROKER = "localhost"          # or the instructor's broker address
GROUP = "g01"
BASE = f"cec405/{GROUP}"

ser = serial.Serial(PORT, BAUD, timeout=1)

def on_connect(client, userdata, flags, reason_code, properties):
    print("connected:", reason_code)
    client.publish(f"{BASE}/status", "online", qos=1, retain=True)
    client.subscribe(f"{BASE}/cmd/#", qos=1)

def on_message(client, userdata, msg):
    dev = msg.topic.rsplit("/", 1)[-1]               # lamp or fan
    val = msg.payload.decode(errors="ignore").strip().upper()
    if dev in ("lamp", "fan") and val in ("0", "1", "ON", "OFF"):
        v = "1" if val in ("1", "ON") else "0"
        ser.write(f"{dev.upper()}={v}\n".encode())
        print("command:", dev, v)

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2, client_id=f"{GROUP}-bridge")
client.will_set(f"{BASE}/status", "offline", qos=1, retain=True)
client.on_connect = on_connect
client.on_message = on_message
client.connect(BROKER, 1883, keepalive=30)
client.loop_start()                                  # network loop in a thread

while True:
    line = ser.readline().decode(errors="ignore").strip()
    if not line.startswith("{"):
        continue
    try:
        d = json.loads(line)
        client.publish(f"{BASE}/sensors/temperature", d["t"])
        client.publish(f"{BASE}/sensors/light", d["light"])
        client.publish(f"{BASE}/state/lamp", d["lamp"], qos=1, retain=True)
        client.publish(f"{BASE}/state/fan", d["fan"], qos=1, retain=True)
    except (ValueError, KeyError):
        continue
```

4. Procedure and observations

1. Start Proteus, then the bridge. In `mosquitto_sub` (or MQTT Explorer) observe the topics.
2. Switch the lamp: `mosquitto_pub -h localhost -t cec405/g01/cmd/lamp -m ON -q 1`.
3. Stop `mosquitto_sub` and restart it: which topics are received immediately? Why?
4. Kill the bridge brutally (close its window). After how many seconds does `cec405/g01/status` become `offline`? Relate it to the keep-alive value.
5. Subscribe to `cec405/+/sensors/temperature` on the shared broker: list the groups seen.

Topics received immediately on restart: _____

Delay before “offline”: _____

Groups seen: _____

L8.3 Part B (week 13) — Distributed control and dashboard

1. **Thermostat client.** Write `thermostat.py`, a separate MQTT client that subscribes to `cec405/<g>/sensors/temperature` and publishes `cec405/<g>/cmd/fan` = `ON` above 30 °C and `OFF` below 28 °C — *only when the state must change*. The device and the bridge are not modified: this is the strength of publish/subscribe.
2. **Dashboard.** Build a dashboard with one of: MQTT Explorer (charts on topics), Node-RED with `node-red-dashboard` (gauge, chart, switches), or a mobile MQTT app (IoT MQTT Panel, MQTT Dash) connected to the broker over Wi-Fi.
3. **Cross-group test.** With the instructor's broker, let another group switch your lamp (they publish on your command topic). Then discuss what should prevent this in a real system.
4. **QoS experiment.** Publish 100 numbered messages with QoS 0 and QoS 1 while the subscriber's network is briefly interrupted; count the received messages.
5. **Security (discussion or demonstration).** Disable anonymous access: create users with `mosquitto_passwd -c passwd g01`, add `allow_anonymous false` and `password_file passwd` to `lab.conf`, and pass `client.username_pw_set()` in the bridge.

Going further: MQTT on the device itself

With an ESP8266 or ESP32 (programmed with the Arduino IDE and the `PubSubClient` library), the device connects to Wi-Fi and to the broker directly, without a PC gateway. The topic design, retained states and LWT used here remain exactly the same.

L8.4 Questions

1. Draw the sequence of messages from the moment `thermostat.py` detects 31 °C to the moment the fan LED turns on in Proteus.
2. Compare the Web (Lab 7) and MQTT (Lab 8) architectures: number of connections, push vs. pull, bandwidth, suitability for battery-powered devices, adding a new client.
3. Which QoS would you choose for (a) a temperature sent every 2 s, (b) a command to open a valve, (c) an alarm? Justify.
4. What would you change in the topic structure for 500 devices spread over several sites?

What to hand in

`bridge.py`, `thermostat.py`, `lab.conf`, dashboard screenshots, observation tables, answers.
Instructor's signature (A): _____ (B): _____

APPENDIX A

Quick Reference: CCS C and Arduino

Function	CCS C (PIC16F877A)	Arduino (Uno)

Device / clock	`#include <16F877A.h>`
	`#fuses HS,NOWDT,NOLVP`
	`#use delay(clock=20000000)`
		`(board selected in the IDE)`

Pin direction	`set_tris_d(0x00); (automatic with output_x)`	`pinMode(13, OUTPUT);`

Write pin	`output_high(PIN_D0);`
	`output_low(PIN_D0);`
	`output_bit(PIN_D0, v);`
		`digitalWrite(13, HIGH);`

Write port	`output_d(0xFF);`	`PORTD = 0xFF;`
Read pin	`input(PIN_B0)`	`digitalRead(2)`
Read port	`input_b()`	`PIND`
Pull-ups	`port_b_pullups(TRUE);`	`pinMode(2, INPUT_PULLUP);`
Delays	`delay_ms(n); delay_us(n);`	`delay(n); delayMicroseconds(n);`
Elapsed time	`(use a timer)`	`millis(), micros()`

ADC	`#device ADC=10`
	`setup_adc_ports(AN0);`
	`setup_adc(ADC_CLOCK_DIV_32);`
	`set_adc_channel(0); v=read_adc();`
		`v = analogRead(A0);`

PWM	`setup_ccp1(CCP_PWM);`
	`setup_timer_2(T2_DIV_BY_16,255,1);`
	`set_pwm1_duty(512);`
		`analogWrite(9, 128);`

Serial setup	`#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)`	`Serial.begin(9600);`
Serial out	`printf("x=%d\r\n", x); putc(c);`	`Serial.print(x); Serial.println();`
Serial in	`if (kbhit()) c = getc();`	`if (Serial.available()) c = Serial.read();`

LCD	`#define LCD_... pins; #include <lcd.c>`
	`lcd_init(); lcd_gotoxy(1,2);`
	`printf(lcd_putc,"\fT=%4.1f",t);`
		`#include <LiquidCrystal.h>`
		`LiquidCrystal lcd(rs,e,d4,d5,d6,d7);`
		`lcd.begin(16,2); lcd.setCursor(0,1);`
		`lcd.print(t,1);`

EEPROM	`write_eeprom(addr, v); v = read_eeprom(addr);`	`#include <EEPROM.h>` `EEPROM.update(addr, v); v = EEPROM.read(addr);`

Interrupts	`#int_ext void isr(){...}`
	`enable_interrupts(INT_EXT);`
	`enable_interrupts(GLOBAL);`
		`attachInterrupt(digitalPinToInterrupt(2), isr, FALLING);`

Types	`int1, int8, int16, int32, signed int16, float`	`bool, byte, int (16 bit), long (32 bit), float`

Integer sizes

In CCS C an `int` is **8 bits and unsigned** by default; on the Arduino an `int` is 16 bits and signed. Always use explicit types (`int16`, `signed int16`, `uint16_t`) when porting code between the two.

APPENDIX B

Proteus Components Used in the Labs

Component	Keyword in Pick Devices	Remarks
PIC microcontroller	`PIC16F877A`	set Processor Clock Frequency; Program File is automatic in a firmware project
AVR / Arduino	`ATMEGA328P`	or Arduino Uno via firmware/flowchart project
Resistor	`RES`	set the value by double-click
LEDs	`LED-RED, LED-GREEN, LED-YELLOW, LED-BLUE`	animated
Push-button	`BUTTON`	momentary; click to press
Switch	`SWITCH, DIPSW_8`	latching
Temperature sensors	`LM35, DS18B20`	interactive temperature arrows
LCD 16×2	`LM016L`	HD44780 compatible
Light sensor	`TORCH_LDR, LDR`	torch distance adjustable
Potentiometer	`POT-HG`	interactive
Transistor	`BC547, 2N2222`	
Relay	`RELAY`	choose coil voltage
Motor	`MOTOR-DC`	
Buzzer	`BUZZER, SOUNDER`	
Keypad	`KEYPAD-PHONE, KEYPAD-SMALLCALC`	4×3 / 4×4
Seven-segment	`7SEG-COM-CATHODE, 7SEG-MPX4-CC`	
Serial port	`COMPIM`	link to a real or virtual COM port
Level converter	`MAX232`	only for a real-board schematic
Instruments	`VIRTUAL TERMINAL, OSCILLOSCOPE, LOGIC ANALYSER, DC VOLTMETER`	Virtual Instruments Mode

Common Proteus problems

Symptom	Probable cause / remedy
Build fails at once	Compiler not configured: **Compilers...** in the New Project wizard; set the CCS folder
Nothing happens	Firmware not built, no Program File, or wrong path (avoid spaces and accents in paths)
Timing is wrong	Clock frequency property differs from `#use delay` / board clock
ADC reads 0–255 only	`#device ADC=10` missing (CCS)
Input always 0 or 1	Missing pull-up/down; pin configured as analogue (`setup_adc_ports`)
LCD shows blocks / nothing	Pin definitions do not match wiring; `lcd_init()` missing
Nothing on COMPIM	Wrong COM port or baud; TXD/RXD swapped; port opened by another program
"Simulation not running in real time"	Too many probes/graphs or delays too short; reduce instruments

APPENDIX C

PIC16F877A Pin Assignment (DIP-40)

Pin	Function	Pin	Function
1	MCLR/VPP	40	RB7/PGD
2	RA0/AN0	39	RB6/PGC
3	RA1/AN1	38	RB5
4	RA2/AN2/VREF−	37	RB4
5	RA3/AN3/VREF+	36	RB3/PGM
6	RA4/T0CKI	35	RB2
7	RA5/AN4/$\overline{\text{SS}}$	34	RB1
8	RE0/AN5/$\overline{\text{RD}}$	33	RB0/INT
9	RE1/AN6/$\overline{\text{WR}}$	32	VDD
10	RE2/AN7/$\overline{\text{CS}}$	31	VSS
11	VDD	30	RD7/PSP7
12	VSS	29	RD6/PSP6
13	OSC1/CLKI	28	RD5/PSP5
14	OSC2/CLKO	27	RD4/PSP4
15	RC0/T1OSO/T1CKI	26	RC7/RX/DT
16	RC1/T1OSI/CCP2	25	RC6/TX/CK
17	RC2/CCP1	24	RC5/SDO
18	RC3/SCK/SCL	23	RC4/SDI/SDA
19	RD0/PSP0	22	RD3/PSP3
20	RD1/PSP1	21	RD2/PSP2

RA4 is open-drain: it needs a pull-up to output a ‘1’.

Arduino Uno pin map

Arduino pins	ATmega328P	Special functions
D0, D1	PD0, PD1	RX, TX (USB serial)
D2, D3	PD2, PD3	INT0, INT1; D3 = PWM
D4–D7	PD4–PD7	D5, D6 = PWM
D8–D13	PB0–PB5	D9, D10, D11 = PWM; D10–D13 = SS, MOSI, MISO, SCK; D13 = on-board LED
A0–A5	PC0–PC5	ADC0–ADC5; A4 = SDA, A5 = SCL

APPENDIX D

Lab Report Template

Use this structure for any lab report requested by the instructor (and for the mini project report).

1. **Title page**: course, lab number and title, group members (names, matricules), date.
2. **Objectives** (in your own words, 3–5 lines).
3. **Pre-lab answers**.
4. **Circuit**: schematic exported from Proteus, list of components with calculated values.
5. **Program**: flowchart or state diagram, commented listing (or its key parts).
6. **Results**: completed tables, screenshots (simulation, terminal, web page), measured values with units.
7. **Discussion**: comparison with expected values, errors and their causes, problems met and how they were solved.
8. **Answers to the questions**.
9. **Conclusion**: what was learnt, possible improvements.

Good practice

Name your files consistently (`CEC405_Lab4_G07.c`), comment every function, keep the wiring and the `#define` pin names in agreement, and keep a backup of each working version before modifying it.