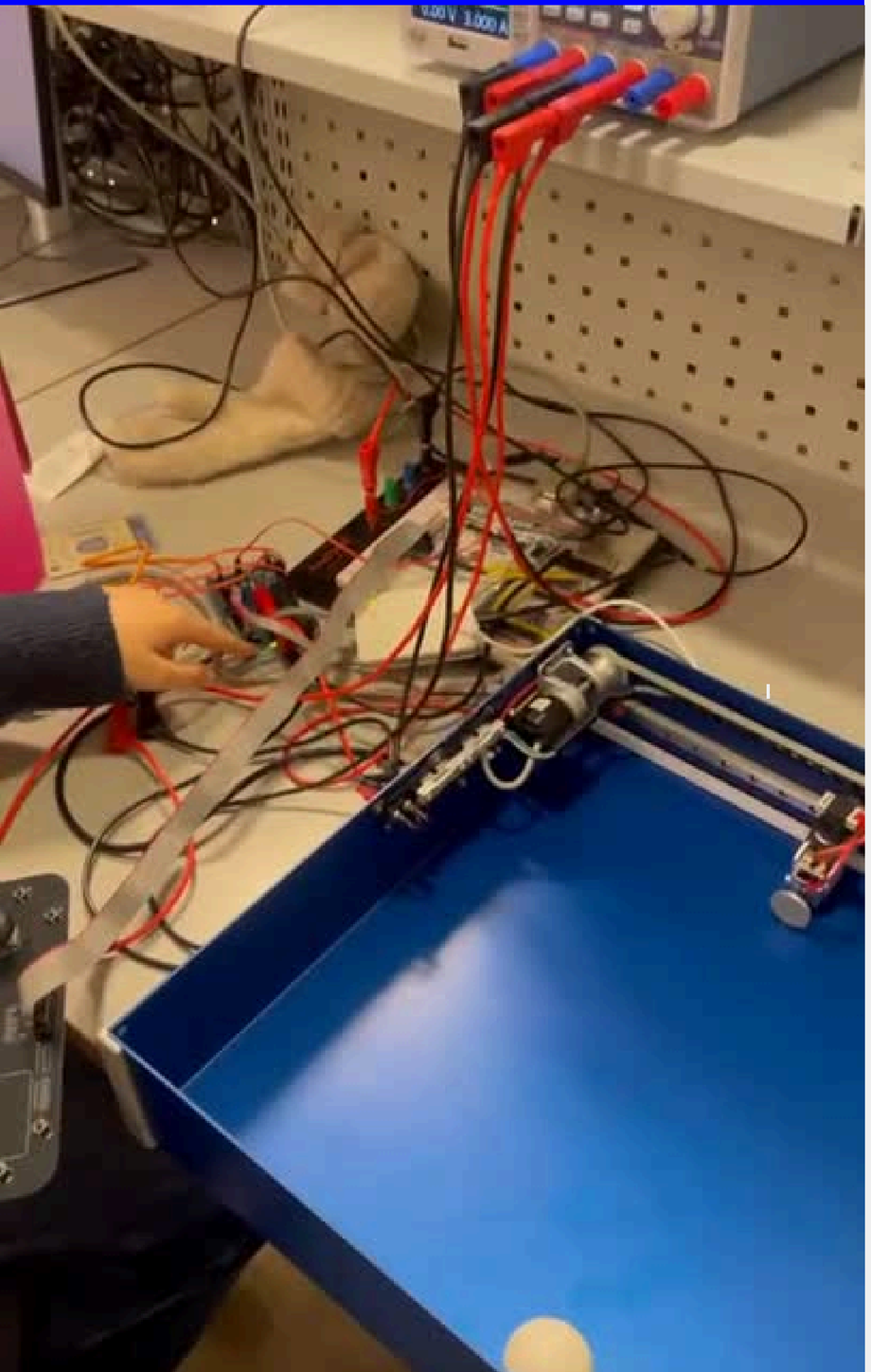
Pong game with a Distributed Embedded Control System

Presented by Sam Chhabra



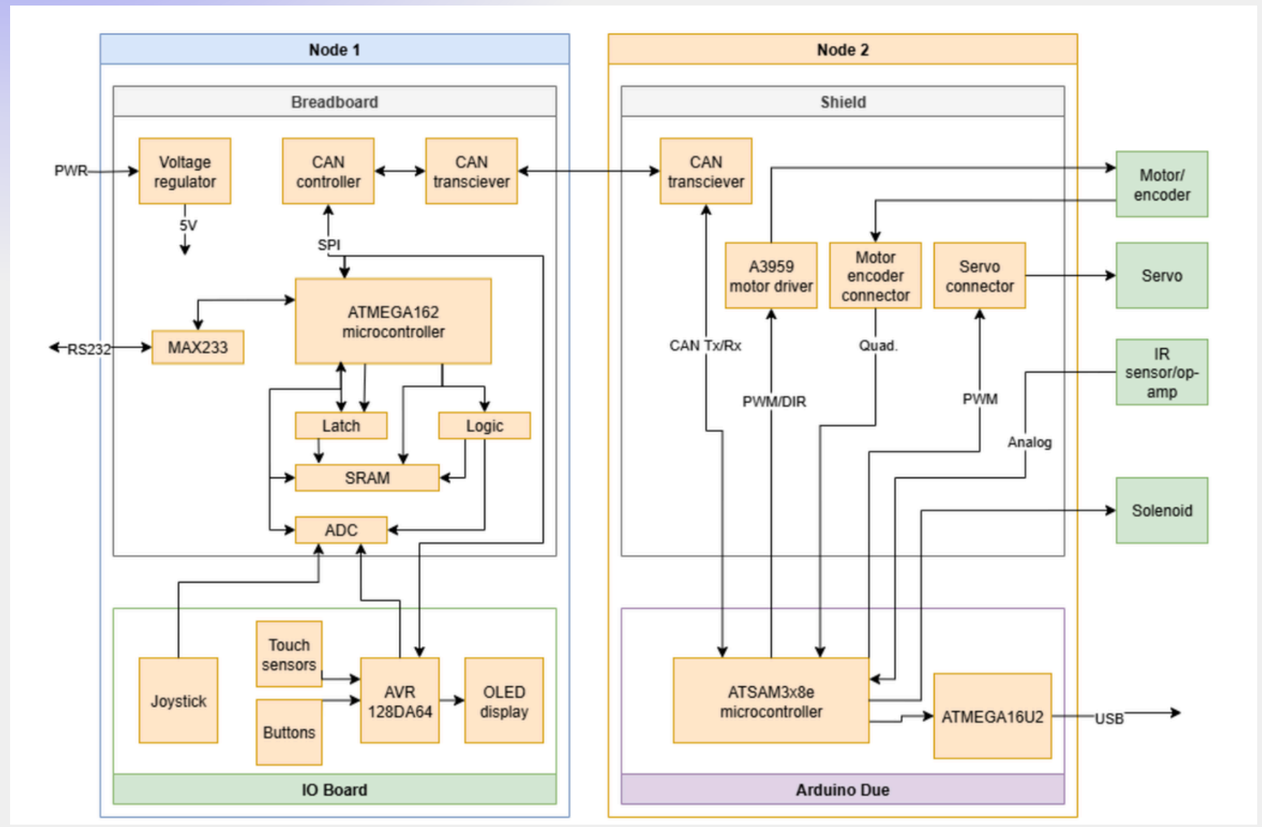
Outline

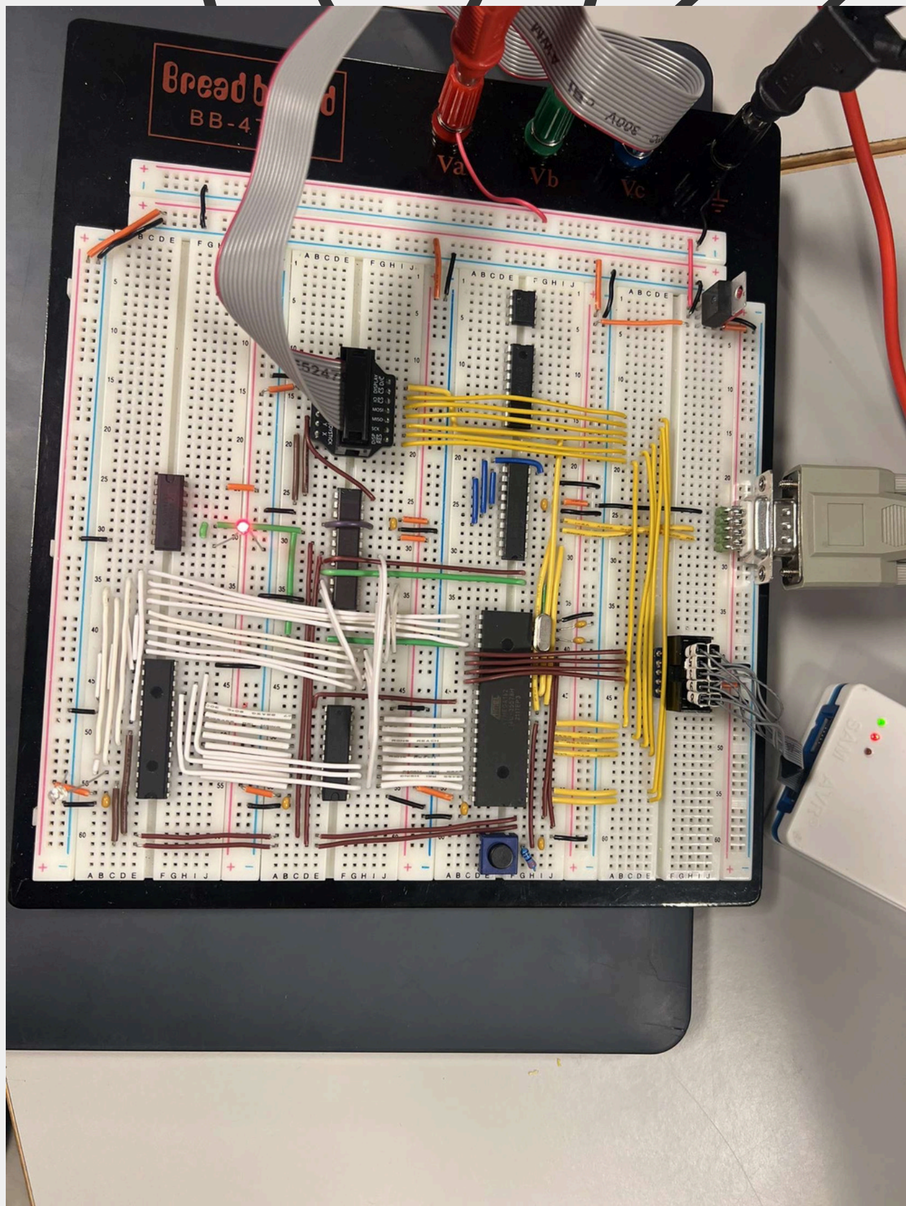
01

System architecture

- How did we program the components
- Components Used
- Node 1: structure and functionality
- Node 2: structure and functionality
- Communication between nodes
- Interrupts and PMW use in the project
- Control loop and feedback
- Conclusions

Project Architecture





Components Used

03

Component	Node	Purpose / Function	Reference Manual
ATmega162	Node 1	Main controller for game logic, user input, display control, ADC handling, and CAN communication	ATmega162 datasheet
ATSAM3X8E (Ardui 2)	Node 2	Motor control, PWM generation, encoder decoding, solenoid control, closed-loop control	SAM3X datasheet
ATmega16U2	Node 2	USB-to-serial interface for programming and debugging	Arduino Due documentation
MCP2515	Node 1	Implements CAN 2.0B protocol, message handling via SPI	MCP2515 datasheet
MCP2551	Node 1&2	Converts logic-level CAN signals to differential CAN bus signals	MCP2551 datasheet
MAX233	Node 1	Enables serial communication with a PC for debugging	MAX233 datasheet
MAX156 (8/4-channel)	Node 1	Converts analog joystick and touch sensor signals to digital values	MAX156 datasheet
SRAM	Node 1	Stores external data such as display buffers	SRAM datasheet
SSD1309 OLED	Node 1	Drives the 128×64 OLED display	SSD1309 datasheet
OLED 128×64	Node 1	Visual output for game state and information	OLED module manual
Joystick	Node 1	Analog directional user input	Joystick module
Buttons / Touch sensors	Node 1	Additional user interaction inputs	I/O board documentation
A3959	Node 2	Drives DC motor using PWM and direction control	A3959 datasheet
DC Motor	Node 2	Moves the racket	Motor datasheet
Quadrature Encoder	Node 2	Provides motor position and direction feedback	SAM3XTC
Servo Motor	Node 2	Controls mechanical inclination using PWM	RC servo standard
Solenoid	Node 2	Shoots the ping-pong ball	Solenoid datasheet
IR Sensor + Op-Amp	Node 2	Detects goal events via beam interruption	Sensor documentation
Voltage Regulator	Node 1	Generates stable 5 V supply	Regulator datasheet

Development process

04

The project was implemented using a **low-level embedded systems approach**, without the use of Microchip Studio or the Arduino IDE.

All programming and debugging tasks were carried out through command-line toolchains and external debugging tools.

During development, each node was connected to a computer to enable:

- Firmware flashing
- Code debugging
- Serial communication and logging

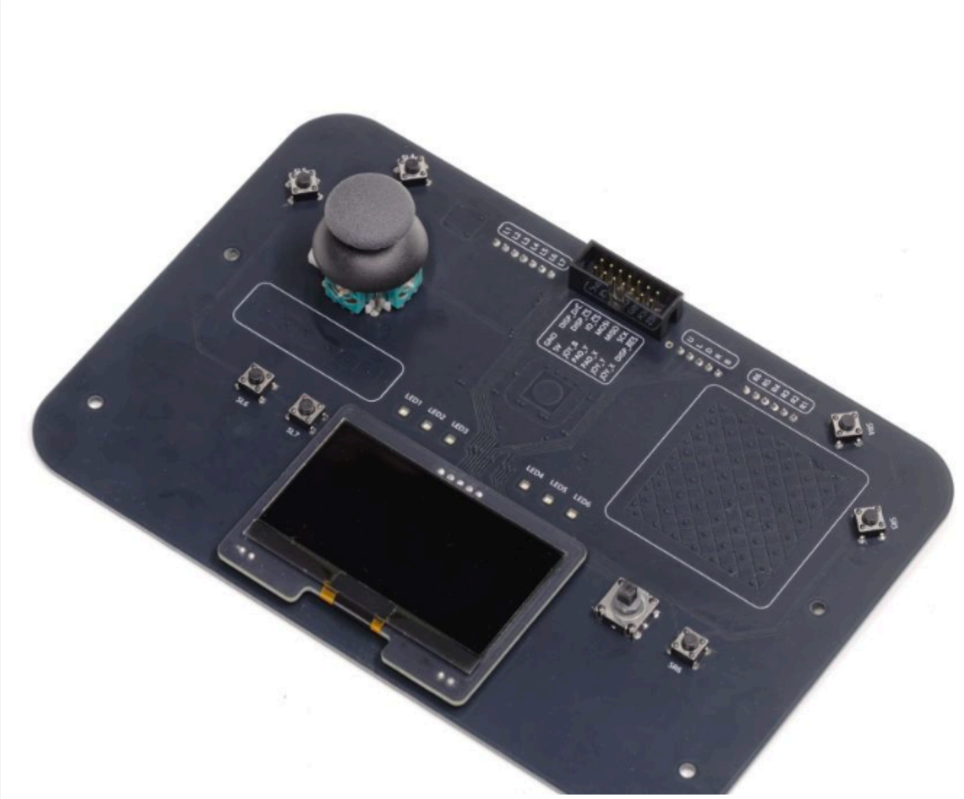
The primary tools utilized were:

- avr-gcc → for compilation
- avrdude → for flashing AVR microcontrollers
- openOCD → for flashing and debugging ARM-based microcontrollers
- Atmel-ICE → hardware debugger/programmer used for:
- ATmega162 (Node 1) via JTAG
- ATSAM3X8E (Node 2) via JTAG

The entire development process was **conducted on Linux**, with Makefiles used to automate both compilation and firmware flashing.

External Components used

05



- I/O Board
- Joystick
- Buttons
- Touch sensors
- AVR128DA64 (ucontrol)

Manages the I/O board peripherals and communicates with the ATmega162 via SPI.

- OLED display

Node 1: structure and functionality

06

- **Voltage regulator:**

Converts the external supply voltage to a stable 5 V for all Node 1 components.

- **Processing**
- **ATmega162 microcontroller**

Main controller of Node 1. Handles user input processing, game logic, display control, and CAN communication.

- **External Memory and Data Acquisition**
- **Latch**

Demultiplexes address and data lines for the external memory interface.
through address decoding Selects which external device (SRAM or ADC) is active based on the address.

- **SRAM:** External memory used for data storage (e.g. display buffers).
- **ADC (MAX156):** Converts analog signals from joystick and touch sensors into digital values.
- **Communication and Debug**

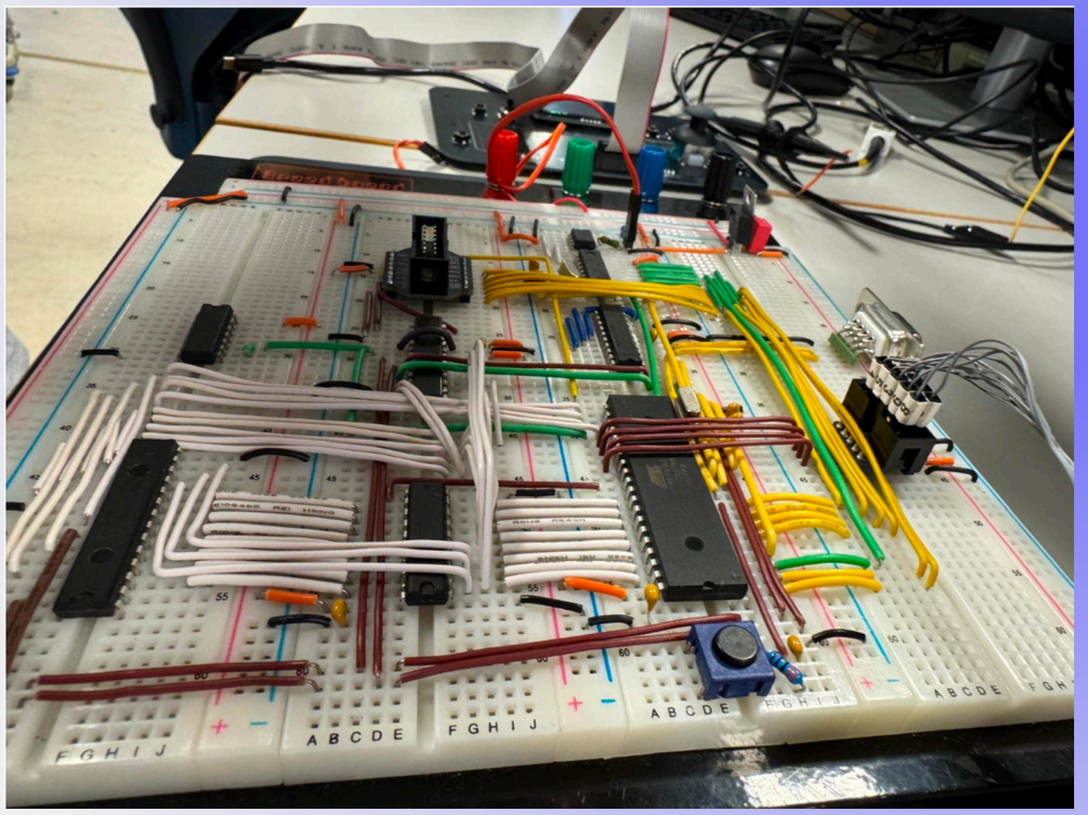
MAX233 (RS-232 driver)
Enables serial communication between the microcontroller and a PC for debugging.

- **CAN controller (MCP2515)**

Implements the CAN protocol logic and message handling.

- **CAN transceiver**

Converts logic-level CAN signals to differential bus signals.



Node 2: structure and functionality

07

- **Processing**
- **ATSAM3X8E microcontroller (Arduino Due)**

Main controller of Node 2: Executes motor control, servo control, solenoid triggering, sensor acquisition, and closed-loop control algorithms.

- **ATmega16U2**

Acts as a USB-to-serial interface for programming and debugging the Arduino Due.

- **Communication**
- **CAN transceiver**

Interfaces the Arduino Due with the CAN bus for communication with Node 1.

Motor and Actuators

- **A3959 motor driver**

Drives the DC motor using PWM and direction control.

- **Motor encoder connector**

Interfaces the quadrature encoder for position feedback.

- **Servo connector**

Provides PWM and power signals to the servo motor.

- **Solenoid**

Electromechanical actuator used to shoot the ping-pong ball.

- **IR sensor / operational amplifier**

Detects goal events by sensing interruption of an infrared beam.

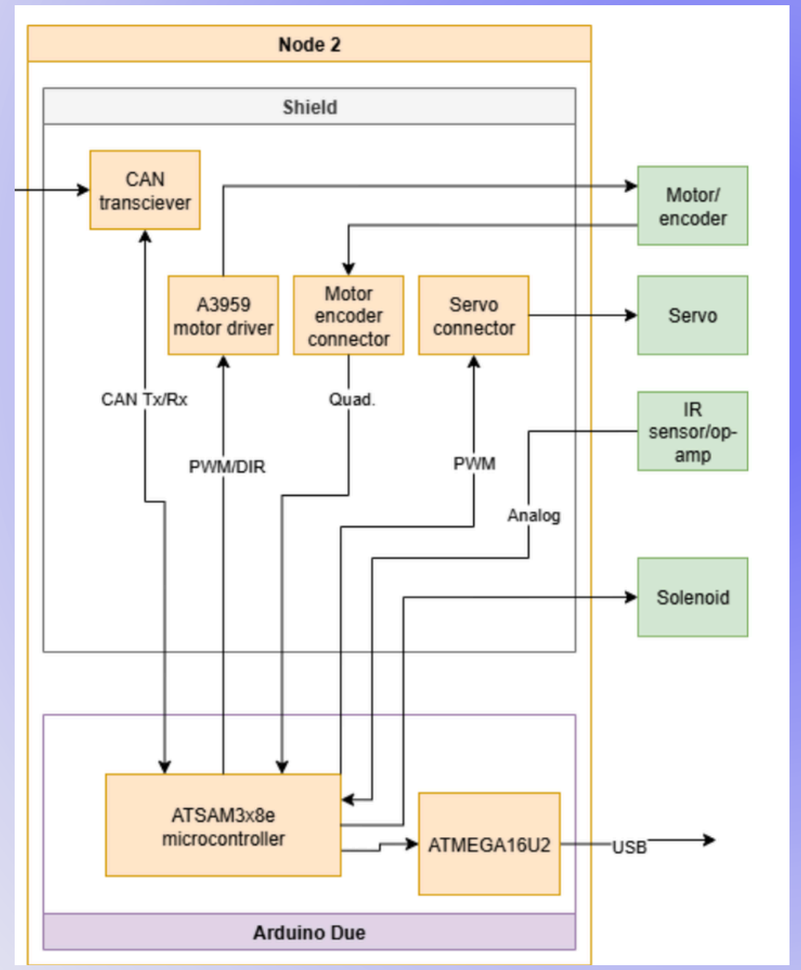
External Components:

- **Motor with quadrature encoder**

Moves the racket and provides precise position feedback.

- **Servo motor**

Controls an additional mechanical degree of freedom.



CAN

- **Purpose**

CAN is used for reliable communication between Node 1 and Node 2, Node 1 needs to send to Node 2 the direction of the joystic sampled by the ADC.

- **Main Components**

ATmega162: main microcontroller on Node 1
MCP2515:CAN controller
MCP2551: CAN transceivers
Differential CAN bus with termination resistors

- **Setup and Configuration**

MCP2515 connected to ATmega162 via SPI
CAN bit timing configured consistently on both nodes
Multi-master CAN bus using message IDs for priority and filtering
Interrupt line used to signal received CAN messages

- **Software Implementation**

We developed a custom CAN library
Low-level driver handles MCP2515 registers via SPI
High-level CAN interface is hardware-independent
Initialization only requires setting clock, bitrate, bit timing (CNF1–3), and mode

```
CAN_init_normal_16TQ();

//start transmission with 0x100
CanFrame tx = {
      .id  = 0x100,
      .dlc = 1,
      .data = {0x00}
};

CAN_send(&tx);

_delay_ms(100);

CanFrame rx;

while (!run_menu)
{
    update_joystick();
    //update_slider();
    oled_play();


    uint8_t dir = encode_direction(joystick.dir);

    CanFrame tx = {
        .id  = 0x111,
        .dlc = 4,
        .data = {dir, joystick.x_val_perc, joystick.y_val_perc, (uint8_t) joystick.button}
    };


    // printf("Sending direction:code: %02X)\n\r", dir);

    CAN_send(&tx);
```

```
uint8_t BRP = (F_CPU / (16*BAUD)) - 1; // Baud Rate Prescaler

uint32_t reg = ((0<<24) | (BRP<<16) | (3<<12) | (1<<8) | (6<<4) | 5);
/*
 * - Target bit rate: 250 kbps
 * - Clock frequency: 84 MHz
 * - Time quanta (TQ) per bit: 16
 *
 * BRP = (F_CPU / (BAUD * TQ)) - 1 = (84,000,000 / (250,000 * 16)) - 1 = 20
 *
 * Bit timing segments (all decremented by 1 for register):
 * PROPAG = 1 TQ
 * Tprs = 2 * (50+30+DLine) ns = 2 Tcsc
    => PROPAG = Tprs/Tcsc - 1 = 1

 * Phase1 + Phase2 = 16 TQ - Sync(1 TQ) - PropSeg (2 TQ) = 13 TQ
 * PHASE1 = 7 TQ                -> 6 in register
 * PHASE2 = 6 TQ                -> 5 in register
 *
 * SJW = 4 TQ                   -> 3 in register
 * SMP = 0 (single sampling)
 *
 */
```

Interrupts and PMW usage in the project

To efficiently handle CAN message reception and transmission, the system uses a **hardware interrupt generated by the MCP2515**.
The **INT pin of the MCP2515** is connected to the **external interrupt pin INT1 of the ATmega162**.

Whenever the MCP2515 receives a valid CAN message or completes a transmission, it pulls this pin low, triggering an interrupt.

If a receive interrupt (MCP_RX0IF) is set, the CAN message is read from the MCP2515 buffer, made available to the game logic, and the flag is cleared.
If a transmit interrupt (MCP_TX1IF) is set, the system detects that the message was successfully sent and clears the flag to allow new transmissions.

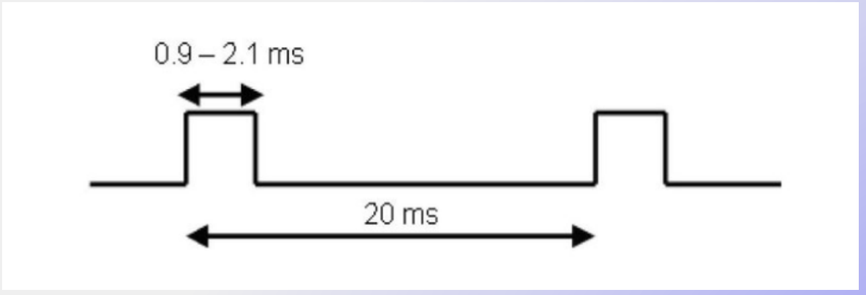
This **event-driven approach** avoids continuous polling of the CAN bus.

```
ISR(INT1_vect)
{
    uint8_t flags = MCP_get_interrupt_flags();

    if (flags & MCP_RX0IF) {
        CanFrame rx;

        CAN_receive(&rx);
        MCP_clear_interrupt_flags(MCP_RX0IF);


        //handle received frame

        if (DEBUG_CAN) {
            printf("Received frame!\n");
            printf("ID: 0x%03X, DLC: %u, DATA:", rx.id, rx.dlc);
            for (uint8_t i = 0; i < rx.dlc; i++)
                printf(" %02X", rx.data[i]);
            printf("\n");
        }
    }

    if (flags & MCP_TX1IF) {
        MCP_clear_interrupt_flags(MCP_TX1IF);
        printf("transmission complete");
    }
}
```

TC0 generates a periodic interrupt (control tick)
and the motor control update is performed within the ISR.

```
void control_timer_init(void)
{
    PMC->PMC_PCER0 |= (1 << ID_TC0);

    TcChannel *ch = &TC0->TC_CHANNEL[0];

    ch->TC_CMR =
        TC_CMR_TCCLKS_TIMER_CLOCK4 |   // MCK/128 → 84 MHz / 128 = 656 25
        TC_CMR_WAVE |                  // waveform mode
        TC_CMR_WAVSEL_UP_RC;           // reset on RC compare

    ch->TC_RC = 13125;

    ch->TC_IER = TC_IER_CPCS;          // interrupt on RC match
    ch->TC_IDR = ~TC_IER_CPCS;

    NVIC_EnableIRQ(TC0_IRQn);

    ch->TC_CCR = TC_CCR_CLKEN | TC_CCR_SWTRG;
}

void TC0_Handler(void)
{

    TC0->TC_CHANNEL[0].TC_SR;
    update_motor();
}
```

Interrupts and PMW usage in the project



In **Node2**, the **SAM3X microcontroller** uses hardware PWM to control both a servo motor and a DC motor.
The servo is driven using **PWM Channel 1 (PB13)** with a **20 ms period**, which is the standard frequency for RC servos. By changing the pulse width (around 1.5 ms for the center position), the system precisely controls the servo position.
The **DC motor** is controlled using **PWM Channel 0 (PB12)** with the same time base. In this case, the duty cycle represents the motor speed, starting from zero and increasing as needed during runtime.

```
void init_pmw_motor(void){

    PMC->PMC_PCER0 |= (1u << ID_PIOB) (int)4 PIOB manipulation
    PMC->PMC_PCER1 |= (1u << (ID_PWM - 32));   //  PWM clock

    PIOB->PIO_PDR   = PIO_PDR_P12;              // disable GPIO control
    PIOB->PIO_ABSR |= PIO_ABSR_P12;             // select Peripheral B (PWMH0)

    //Disable channel during setup
    PWM->PWM_DIS = PWM_DIS_CHID0;

    PWM->PWM_CH_NUM[0].PWM_CMR  = PWM_CMR_CPRE_CLKA | PWM_CMR_CPOL;  // to have high pulses

    PWM->PWM_CH_NUM[0].PWM_CPRD = 20000;              // 20 ms
    PWM->PWM_CH_NUM[0].PWM_CDTY = 0;                //motor speed

    PWM->PWM_ENA = PWM_ENA_CHID0;
}
```

```
static inline void servo_write(uint32_t ch, uint16_t us)
{
    if (us != PWM->PWM_CH_NUM[ch].PWM_CDTYUPD) {
        if (us < 900)  us = 900;
        if (us > 2100) us = 2100;
        PWM->PWM_CH_NUM[ch].PWM_CDTYUPD = us;
    }

    //printf("servo: %d\n\r", PWM->PWM_CH_NUM[ch].PWM_CDTY);
}
```

Control loop and feedback

```c
void update_motor(void)
{
    int32_t pos = qdec_tc2_get_position();

    // Errore scelto per avere u > 0 ⇒ destra (dir=0), u < 0 ⇒ sinistra (dir=1)
    int32_t err = pos - latest_setpoint;

    // Dead-band
    if (err > -DEAD_BAND && err < DEAD_BAND) {
        I *= 0.98f;
        if (I > -1.0f && I < 1.0f) I = 0.0f;
        motor_write(0, 2, 0);
        return;
    }

    // ===== PI con EULER ESPLICITO sull'integrale =====
    // I[k+1] = I[k] + Ts * err[k]

    I += Ts * (float)err;
    if (I >  I_ABS_MAX) I =  I_ABS_MAX;
    if (I < -I_ABS_MAX) I = -I_ABS_MAX;

    float u = Kp * (float)err + Ki * I;   // sforzo di controllo

    int dir, duty;
    if (u >= 0.0f) { dir = 0; duty = (int)u; }   // 0 = destra
    else           { dir = 1; duty = (int)(-u); }// 1 = sinistra

    if (duty > DUTY_MAX) duty = DUTY_MAX;

    if (pos >= left_limit  && dir == 1) duty = 0;  // a sinistra e chiedi più sinistra
    if (pos <= right_limit && dir == 0) duty = 0;  // a destra e chiedi più destra

    motor_write(0, dir, duty);
}
```

The motor control system in Node 2 is implemented as a **closed-loop control system**, where the motor position measured by the encoder is continuously compared to a reference value received from Node 1. The goal is to minimize the error between the desired and the actual position.

At each control step:

1. The encoder provides the current motor position.
2. The position error is computed.
3. A control law updates the motor command accordingly.

This control loop is executed **periodically inside a timer interrupt (TC0_Handler)**, ensuring a fixed and deterministic control period.

The system uses a **discrete time integration approach**, based on the **Euler method**, to update the motor state over time. In practice, the continuous motor dynamics are approximated by updating the control output using small time steps and integral error accumulation.

=> the system achieves stable motor positioning, the greater t**he error the faster the correction and vice versa.**

$$u[n] = K_p e[n] + TK_i \sum_{i=0}^{n} e[i] + \frac{K_d}{T}(e[n] - e[n-1])$$

Thank You

Sam Chhabra