The PWM drawer

Project description:

A **PWM (Pulse Width Modulation) Drawer** functions as a **mini oscilloscope** designed to visualize and analyze PWM signals. Its main purpose is to **measure, display, and print the frequency and waveform** of a given PWM signal.

Microcontroller and Pin Configuration

The project is built using an **ATmega32 microcontroller**, where:

- **Pin D6** is used to **measure or read the PWM signal** from an external source.
- **Pin D7** is used to **generate and output an internal PWM signal** from the ATmega32 itself.

Sources of the PWM Signal

The PWM signals that the drawer analyzes can originate from two main sources:

1. **External Sources:**
    - PWM signals generated by **other microcontrollers or external circuits**.
    - The PWM Drawer captures these signals through **Pin D6**, processes them, and visualizes their frequency and waveform.
2. **Internal Source:**
    - The **ATmega32 itself generates the PWM signal** internally using **Pin D7**.
    - This allows the system to **self-check and debug** its own PWM output without requiring an external generator.
    - The internally generated PWM signal can also be used for **testing and comparison** against external sources.

Functionalities of the PWM Drawer

- **Frequency Measurement:** It calculates and displays the frequency of the detected PWM signal.
- **Duty Cycle Analysis:** Determines the percentage of the high time relative to the full period of the waveform.
- **Waveform Display:** Shows the shape of the PWM waveform in real time, allowing for monitoring and debugging.
- **Signal Comparison:** If needed, the system can compare internal and external signals to ensure consistency.
- **Real-Time Monitoring:** Continuously updates the display, allowing users to observe changes in PWM signals dynamically.
- **Debugging Tool:** Helps in verifying PWM outputs from microcontrollers, motor controllers, and other PWM-based circuits.

PWM functions:

1. PWM generating function.

```
void PWM_voidInitChannel2(void)
{
	// Set PD7 (OC2) as output
	DIO_voidSetPinDirection(DIO_PORTD, DIO_PIN7, DIO_PIN_OUTPUT);

	// Select Fast PWM Mode (Mode 3)
	SET_BIT(TCCR2_REG, WGM20);
	SET_BIT(TCCR2_REG, WGM21);

	// Select Non-Inverting Output Mode
	CLR_BIT(TCCR2_REG, COM20);
	SET_BIT(TCCR2_REG, COM21);
}

void PWM_voidGeneratePWMChannel2(u8 copy_u8DutyCycle)
{
	if (copy_u8DutyCycle <= 100)
	{
		// Calculate OCR2 value for duty cycle
		OCR2_REG = ((copy_u8DutyCycle * 256) / 100) - 1;

		// Select Prescaler = 64
		SET_BIT(TCCR2_REG, CS20);
		SET_BIT(TCCR2_REG, CS21);
		CLR_BIT(TCCR2_REG, CS22);
	}
	else
	{
		// Return error state (Invalid duty cycle)
	}
}

void PWM_voidStopChannel2(void)
{
	// Stop PWM by clearing prescaler bits
	CLR_BIT(TCCR2_REG, CS20);
	CLR_BIT(TCCR2_REG, CS21);
	CLR_BIT(TCCR2_REG, CS22);
}
```

2. PWM Reading function.

```
void PWM_Read(u8* dutyCycle)
{
	u16 RisingEdge = 0, FallingEdge = 0, Ton = 0, Ttotal = 0;
	u16 timeout = 50000;  // Timeout counter to prevent hanging

	// Set PD6 (ICP1) as input
	DIO_voidSetPinDirection(DIO_PORTD, DIO_PIN6, DIO_PIN_INPUT);

	// Enable Noise Canceler, Capture on Rising Edge, Prescaler = 64
	TCCR1B_REG = (1 << ICES1) | (1 << CS11) | (1 << CS10);

	// Wait for First Rising Edge with Timeout
	timeout = 50000;
	while ((GET_BIT(TIFR_REG, ICF1) == 0) && (timeout > 0)) {
		timeout--;
	}
```

```
        if (timeout == 0) {
               *dutyCycle = 0;   // No signal detected, set duty cycle to 0%
               return;
        }

        RisingEdge = ICR1_REG;
        SET_BIT(TIFR_REG, ICF1);  // Clear flag for next capture

        //  Capture Falling Edge with Timeout
        timeout = 50000;
        CLR_BIT(TCCR1B_REG, ICES1);  // Switch to falling edge
        while ((GET_BIT(TIFR_REG, ICF1) == 0) && (timeout > 0)) {
               timeout--;
        }
        if (timeout == 0) {
               *dutyCycle = 0;
               return;
        }

        FallingEdge = ICR1_REG;
        SET_BIT(TIFR_REG, ICF1);  // Clear flag

        //  Capture Next Rising Edge with Timeout
        timeout = 50000;
        SET_BIT(TCCR1B_REG, ICES1);  // Switch back to rising edge
        while ((GET_BIT(TIFR_REG, ICF1) == 0) && (timeout > 0)) {
               timeout--;
        }
        if (timeout == 0) {
               *dutyCycle = 0;
               return;
        }

        Ttotal = ICR1_REG - RisingEdge;
        SET_BIT(TIFR_REG, ICF1);  // Clear flag

        // Calculate Ton
        Ton = FallingEdge - RisingEdge;

        // Prevent division by zero
        if (Ttotal == 0) {
               *dutyCycle = 0;  // No valid PWM detected
               return;
        }

        // Calculate Duty Cycle
        *dutyCycle = ((Ton * 100) / Ttotal);
}
```

3. LCD PWM Displaying function.

```
void LCD_voidDisplayPWM(u8 dutyCycle)
{
              float Ton_value=0,T_total=0;
              float F_pwm=0;
               // Calculate PWM Frequency and Time
               F_pwm = (F_CPU / 1000.0) / (prescaler * 256); // Convert to
kHz
               T_total = 1.0 / F_pwm;
               Ton_value = (dutyCycle/100.0) * T_total;
               // Clear LCD before displaying
                     LCD_voidClear();
                     // Display Frequency and Duty Cycle
```

```
                        LCD_voidDisplayString("Frequency:");
                        LCD_voidDisplayFloat(F_pwm);
                        LCD_voidDisplayString("KHz ");

                        LCD_voidDisplayString(" Duty Cycle:");
                        LCD_voidDisplayNumber(dutyCycle);
                        LCD_voidDisplayChar('%');
                // Move to the first line to draw waveform
                LCD_voidGoToSpecificPosition(LCD_LINE_TWO, 0);
                LCD_voidDisplayString("PWM:");
                float value_top=(dutyCycle/100.0)*4.0;
                u8 value_button=4-value_top;

                for (u8 counter = 0; counter <5; counter++)
                {
                        for (u8
counter_TOP=0;counter_TOP<value_top;counter_TOP++){
                                LCD_voidDisplayChar(0b10110000); // Draw upper
horizontal segment (?)
                        }
                        for (u8
counter_button=0;counter_button<value_button;counter_button++){
                                LCD_voidDisplayChar(0b01011111); // Draw lower
horizontal segment (__)
                        }
                }
                //display the time on
                LCD_voidDisplayString(" TIME:");
                LCD_voidDisplayFloat(Ton_value);
                LCD_voidDisplayString("ms");
                // Loop to shift text to the left
                for (u8 i=0; i<20;i++)
                {
                        _delay_ms(500);  // Delay for smooth movement
                        LCD_voidSendCommand(0b00011000); // Shift display left
                }

}
```

1. PWM_voidInitChannel2();

Purpose:
Initializes Timer2 in **Fast PWM Mode** to generate a PWM signal on **PD7 (OC2 pin)**.

Key Configurations:

- Sets **PD7 as an output** (where the PWM signal will be generated).
- Configures **Fast PWM Mode (Mode 3)** using `WGM20` and `WGM21` bits.
- Sets **Non-Inverting Mode** (PWM output is active high, meaning the duty cycle represents the ON time).

---

2. PWM_voidGeneratePWMChannel2(u8 copy_u8DutyCycle);

Purpose:
Generates a PWM signal on **PD7** with a specific **duty cycle** (0%–100%).

Key Operations:

- **Calculates OCR2 value** to set the duty cycle:

  $$OCR2=((100*dutyCycle)\times256)-1$$

- **Prescaler = 64** for proper PWM timing.

Why Prescaler 64?
Using a prescaler of **64** balances the **PWM frequency** and resolution.

---

3. PWM_voidStopChannel2();

Purpose:
Stops PWM generation on **PD7**.

How?

- Clears **CS20, CS21, and CS22** bits in **TCCR2** to stop the timer.

4. PWM_Read(u8 dutyCycle);*

Purpose:
Reads an **external PWM signal on PD6 (ICP1 – Input Capture Pin)** and calculates its **duty cycle**.

How It Works:

1. **Configures PD6 as Input** to capture PWM signals.
2. **Uses Timer1 Input Capture Mode** to detect signal edges.
3. Captures:
   - **First Rising Edge** → Stores time as `RisingEdge`.
   - **Falling Edge** → Stores time as `FallingEdge`.
   - **Next Rising Edge** → Stores time as `Ttotal` (full period).
4. Calculates:
   - **Ton = FallingEdge - RisingEdge** (ON time).
   - **Ttotal = Next RisingEdge - First RisingEdge** (Full Period).
   - **Duty Cycle = (Ton / Ttotal) * 100**.
5. Uses a **timeout mechanism** to prevent infinite loops in case of no signal.

5. LCD_voidDisplayPWM(u8 dutyCycle)

Purpose:
Displays **PWM Frequency, Duty Cycle, and Waveform** on an **LCD**.

- **Calculates Frequency**:

$$FPWM= FCPU/(Prescaler\times256)$$

- **Calculates Period (Ttotal)**:

  $$Ttotal=1/FPWM$$

- **Calculates ON Time (Ton)**:

$$Ton=(dutyCycle/100)\times Ttotal$$

Project in proteus.