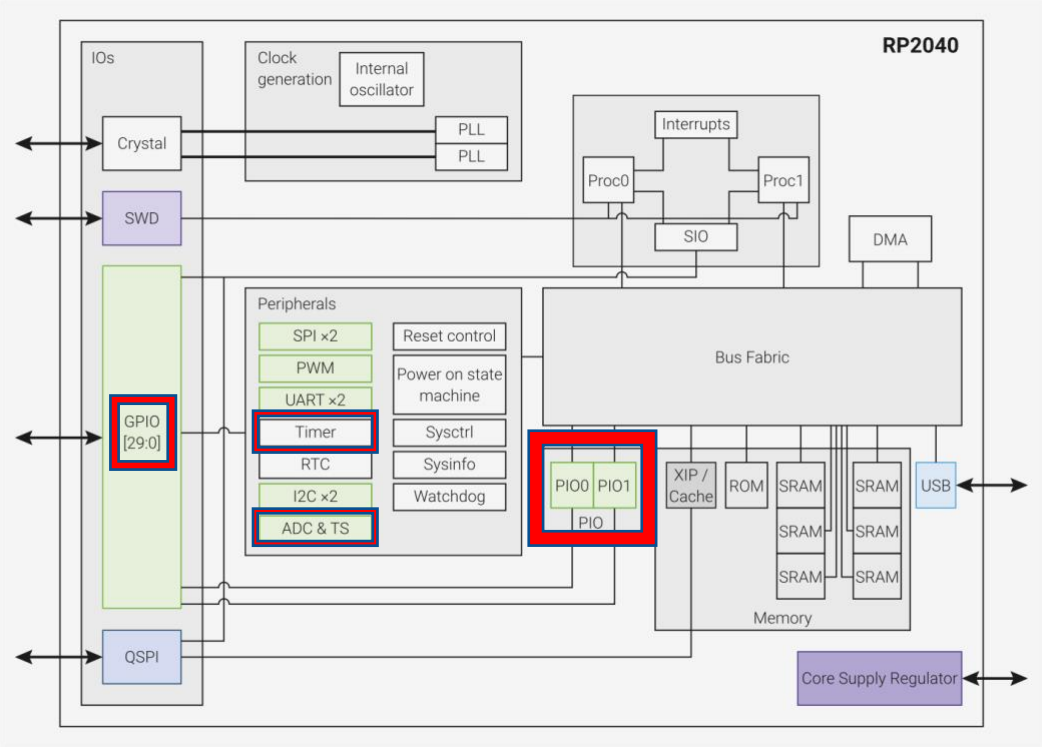
Ultrasonic sensor program

For this piezo transducer I've used the RP2040-zero dev board for its convenient size, layout ,power distribution and practicality of the RP2040 chip being able to do parallel tasks without an OS or RTOS.
Before going into the software its best to check the RP2040 block diagram.
This chip has many hardware peripherals which can be used with memory I/O mapping and some programmable blocks apart of the bus fabric.
To know which one to use for this projects the intentions must be defined:

1. Send pulses through 2 pins (1,0) (0,1) (PWM, PIO, GPIO)
2. Time measurement (program counter, (delays + interrupt)
3. Pin monitoring, pulse received (GPIO, ADC)

For this project the PIO will be used for the pulses since it allows a single program to toggle 2 pins after the function is called once. For time measurement I will use a counter since its always running and I only need to dereference a pointer into a long and subtracts the values between memory calls.
For receiving the pulse I will use the GPIO for simplicity and will include ADC for range:



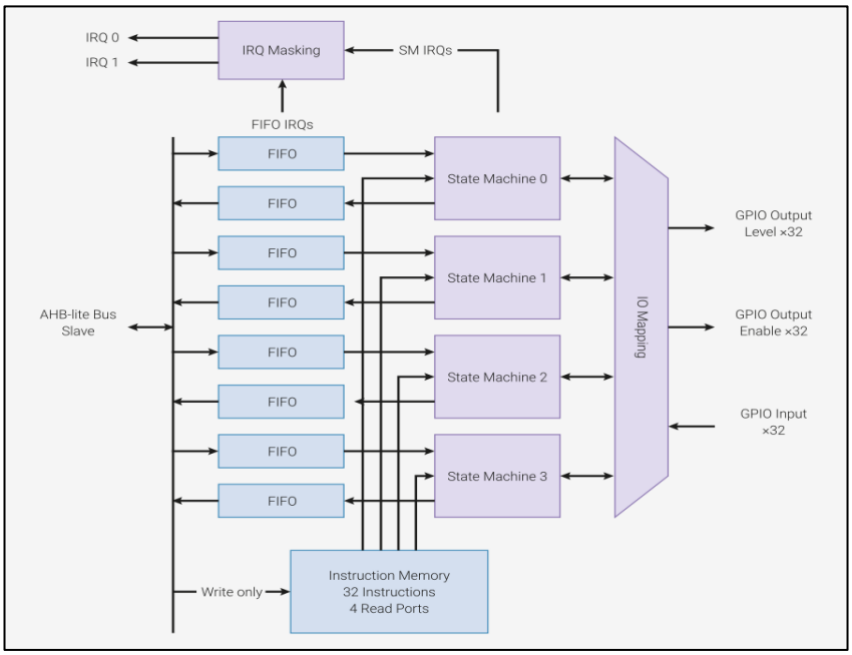
Program logic 1:

1. Send 8 pulses (using PIO) make a interrupt for after pulses finish
2. Use pulse interrupt to dereference counter T0
3. GPIO pin waits to be toggled
4. Once GPIO pin is toggles dereference counter T1
5. Use time difference to get distance of wave travelled.

Program logic 2:

1. Send 8 pulses (using PIO) make a interrupt for after pulses finish
2. Use pulse interrupt to dereference counter T0
3. ADC waits to read a threshold value
4. Once ADC above or below value, dereference counter T1
5. Use time difference to get distance of wave travelled.

PIO program



The PIO is similar to a FPGA in practice. Its programmed in a very simple assembler with only 9 instruction set which is very limiting for anything out of hardware control or simple Boolean operations. For the case of toggling pins its perfect. Only 1 PIO state machine is needed using both input and output shift registers.
Take in account the PIO has a seemingly complex setup to configure before programming, the code has (PIO_C_SDK_init) explained in the notes which is consist of memory flow management, here I will go over the actual PIO program only.

PIO memory:

Memory	Description	Function
Offset	address instruction memory	Configures program (where its loaded and executed from)
TX & RX	input and output buffers TX and RX between PIO & CPU	Temporary 4 word Queue for receiving and sending (32Bit string)
ISR & OSR	input shift register & output shift register	Used for shifting data/ common in assemblers (pull, push, in, out)
Scratch X & Y	scratch registers	Temporary loop variables inside PIO program.

Direction	CPU function	PIO function	FIFO used	Shift register used
CPU -> PIO	Pio_sm_put_blocking()	Pull	TX fifo	OSR
PIO -> CPU	Pio_sm_get_blocking()	Push	RX fifo	ISR

For this memory procedure the PIO program can be written, the C-sdk sends the number of pulses to TX FIFO the PIO pulls the TX fifo into OSR and the Pio moves OSR to X using the mov command. This gives a variable which can be used, then the loop starts. The loop for pins has a nop instruction, so empty instruction of 1 clock to pull a pin high and low through changing 2 bit values (0b01, 0b10) in the loop there's 2 instructions once completed the x register gets reduced by 1 and jumped back to loop start, (jmp x-- , loop) when x = 0 reducing it goes to -1 and for PIO this only works if X > 1 otherwise the program continues, this means that the loop cycles the n times which x is, so if X = 10 it will cycle 10 times, fist cycle x = 9 2nd it will be 8 and so on till zero then it continues to next instruction set.

```
.program pulse
.side_set 2 opt  ; bits used to control pins (2 bits)

; The PIO waits to provide a pulse count
.wrap_target
   pull block                ; block until CPU writes a word to TX FIFO, so pulls to OSR when block is full
   mov x, osr                ; mov from OSR to scratch X to be used in program

loop:
   nop     side 0b01 [1]     ; first half of pulse from side set = 0b[2] = (pin1,pin2) = (01) [true]
   nop     side 0b10 [1]     ; second half of pulse from side set = 0b[2] = (pin1,pin2) = (01) [true]
   jmp x--, loop             ; decrement X and loop if not zero so after each pulse it - the x register and jumps back
                             ; after X = 0 decrement becomes -1 which instructs to move next, (strange but native to PIO)

                             ; After finishing all pulses, notify CPU
   mov isr, null             ; clear ISR (just to be safe)
   set pindirs, 0            ; optionally release the pins (if needed)
   push block                ; push ISR (0) to RX FIFO as completion flag
```

.wrap

C – SDK

The SDK is divided by 2 parts the major part is setting up the PIO but wont be covered here but can be found on the PICO datasheet in **chapter 3,** and the code it self has it explained step by step. (The program has this order USING I/O):

GPIO:

(Setup Pio → Start PIO SM → interrupt → take counter values & wait for GPIO to go high → GPIO High counter value 2 → (dt * a)

ADC:

(Setup Pio → Start PIO SM → interrupt → take counter values & wait for GPIO to go high → GPIO High counter value 2 → (dt * a)

For this section the most important part is the use of the ADC and GPIO. There's 2 options when reading a high Pin. The first Is using the GPIO peripherals which logic operates between (1.8-3.3 +- 0.5 V) these values are Boolean which is more intuitive for the programming side but requires more hardware for reliable readings. The second is the ADC, using the ADC is slightly more complex than using the GPIO since it needs to be configured to get triggered by a value within a range meaning more core power is needed with the benefit of over 1m range with amp and 30cm using no amplifier.

GPIO_program:

```
void pulse_gpio(){
    const uint8_t ECHO_PIN = 27;
    const double SOUND_SPEED_CM_PER_US = 0.0344;
    gpio_init(ECHO_PIN);
    gpio_set_dir(ECHO_PIN, GPIO_IN);
    while (1) {
        // Send the pulse
        pulse_setup();
        uint64_t t0 = read_timer_raw_macro2();  // Immediately after pulse sent
        // Wait for echo rising edge (GPIO goes HIGH)
        uint64_t timeout = t0 + 50000;  // 50 ms timeout
        uint64_t t1 = 0;
        while (gpio_get(ECHO_PIN) == 0) {
            if (read_timer_raw_macro2() > timeout) {
                printf("Timeout waiting for echo (rising edge)\n");
                goto done;
            }
        }
        t1 = read_timer_raw_macro2();  // Time when echo is detected (rising edge)
        // Calculate time difference and distance
        double dt_us = (double)(t1 - t0);
        double distance_cm = (dt_us * SOUND_SPEED_CM_PER_US) / 2.0;
        printf("Echo delay = %.0f us → distance = %.2f cm\n", dt_us, distance_cm);
    done:
        oled_print_value(distance_cm);
        sleep_ms(10);  // avoid flooding
    }
}
```

SDK high level functions:

gpio_init();	gpio_set_dir();	gpio_get();
Initiation	Sets pins	Gets status

PICO_LSD higher level functions:

pulse_setup();	Read_timer_raw_macro2();	oled_print_value();
Sets PIO Pulse	Reads timer value	Prints to SSD1036 LCD

Using the high level functions of the SDK is fast and reliable enough for a rising edge detection, the pulse setup function is within the main file and is just to setup the memory for PIO which executes a like mentioned in the (PIO_PROGRAM_SECTION), read timer macro is also within the file and takes ([TL (*(volatile uint32_t*)(0x40054024))] & [(*(volatile uint32_t*)(0x40054028))]) and buffers those register values into uint64_t which allows it to count to 580000 years.

For the rising edge, the detection loop is stuck at (gpio == 0) comparing current time and measured time + threshold. This is so that only useful time measurements get calculated, another reason keeping it on a low loop gives the processor less work since detecting a rising edge using (Gpio == 1) causes the program to only execute when its high and not after its ben high, alternatively using a variable which saves the value till loop is completed is more intuitive but requires a extra variable and processing power (negligible but elegance!)
Given there's a rising edge detected the program leaves the loop and takes the timer value for t1 and calculates distance using the speed of sound. These values are passed on to the printf(); and to oled_print_value(); afterward

ADC program:

```
void adc_read_dist() {
    adc_init();
    adc_gpio_init(26);
    adc_select_input(0);
    while (1) {
        uint64_t t1 = 0;
        uint8_t raw = 10;
        const int max_samples = 200;
        pulse_setup();
        uint64_t t0 = read_timer_raw_macro2();
        for (int i = 0; i < max_samples; i++){
            raw = read_stable_adc(2);
            printf("raw value = %u\n", raw);
            if(raw < min_v){
                t1 = read_timer_raw_macro2();
                float dt = (t1 - t0 - 200);
                float distance = (dt * 0.0344)/2;
                printf("distance = %.2f cm\n", distance);
                oled_print_value(distance);
                break;
            }
        }
        sleep_ms(10);
    }
}
```

adc_init();	adc_gpio_init();	Adc_select_input();
Initiation	Sets pins for ADC	set ADC block

PICO_LSD higher level functions:

pulse_setup();	Read_timer_raw_macro2();	oled_print_value();
Sets PIO Pulse	Reads timer value	Prints to SSD1036 LCD

The higher level function of the SDK are mainly used to setup the ADC hardware for the RP2040, the PICO_LSD libs are used for timing, setting the pulses and printing on the LCD display.

The ADC program consist of the ADC_INPUT coming directly from the BC547 amplifier before the optocoupler, the reason is because the optocoupler threshold is much lower than the lowest readable signal from the ADC, this allows to have a much larger range for input signal detection giving a doubled increase in range.

The main difference between the GPIO input and ADC input is instead of waiting for a rising edge it waits for a value to be below a threshold, the reason its below and not above a threshold is due to ADC pulled high, the GP PIN is pulled high internally so sending a AC signal will offset the ADC above and below the current reading value. Main rule is if pulled down use a above value and if pulled up use a below value but generally can very depending on the setup. (small changes to be made to follow the same flow as GPIO)

Main Loop:

```
int main() {
    stdio_init_all();

    // gpio input for dip switch
    gpio_init(SET_ADC);
    gpio_set_dir(SET_ADC, GPIO_IN);
    gpio_init(SET_PULSE);
    gpio_set_dir(SET_PULSE, GPIO_IN);
    gpio_init(SET_PULSE_OUT);
    gpio_set_dir(SET_PULSE_OUT, GPIO_IN);
```

```
    oled_init();
    while(1){
        if(gpio_get(SET_ADC) == 0  && gpio_get(SET_PULSE) == 1) {
            pulse_gpio();
        } else if(gpio_get(SET_ADC) == 1 && gpio_get(SET_PULSE) == 0){
            adc_read_dist();
        } else {
            // print no setting selected when i hav
        }

    }
}
```

The main loop is just the execution of the previous functions, to show the functionality of both ADC & GPIO input there's a set of input pins which are pulled low using 10K resistors and pulled high via a 1K and dip switches, this allows for a program selection which directs to a loop for pulse_gpio or adc_read_dist the PICO has a hardware reset so I skipped a setup loop for now and keep it to turning the switches then doing a hardware reset.

Extra info:

There's a few functions which aren't mentioned here such as PIO program initiations and how its memory is managed this information can be found in (3.2 Getting started with PIO) on the SDK documentation, (Chapter 3. PIO) on RP2040 datasheet and in the PICO-SDK examples for example showcases on how to use the PIO. Its extremely well documented and rather simple compared to alternatives like FPGA's.

PICO_LSD higher level functions are libraries I've made specifically for the PICO, these are personal libraries I use for my projects which I haven't documented properly such as the SD1306 which uses the ASCII format to print Char strings onto a LCD, the concept behind this is out of scope since its more heavy on computer science side than in control theory or electrical engineering.

Sources:

https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html#pico-1-family