Fiavi Yang

Embedded Systems

LAB 6
ARM ASSEMBLY LANGUAGE AND TIMERS

W Booth School of Engineering Practice and Technology

ABSTRACT

This lab consists of two parts. In Part 1, you will make a busy-wait timer using ARM Assembly language. In the next part, you will use hardware timers to generate desired delays. Using hardware timers will release the CPU and make it available to attend to other tasks. It is the requirement of complex projects that require multi-tasking. Using timers, you will generate music notes and listen to them via a passive buzzer.

MATERIALS

1. A Windows-based workstation with a USB port available to connect to the Renesas Synergy kit.
2. SSP and e2Studio software.
3. Renesas Synergy SK-S7G2 Starter Kit and the micro-USB cable.
4. One **Passive** Buzzer
5. Male-Female Jumper wires (Brown, Red, Yellow)
6. One Resistor (1k Ohm)
7. One NPN Transistor (S8050)

INSTRUCTION

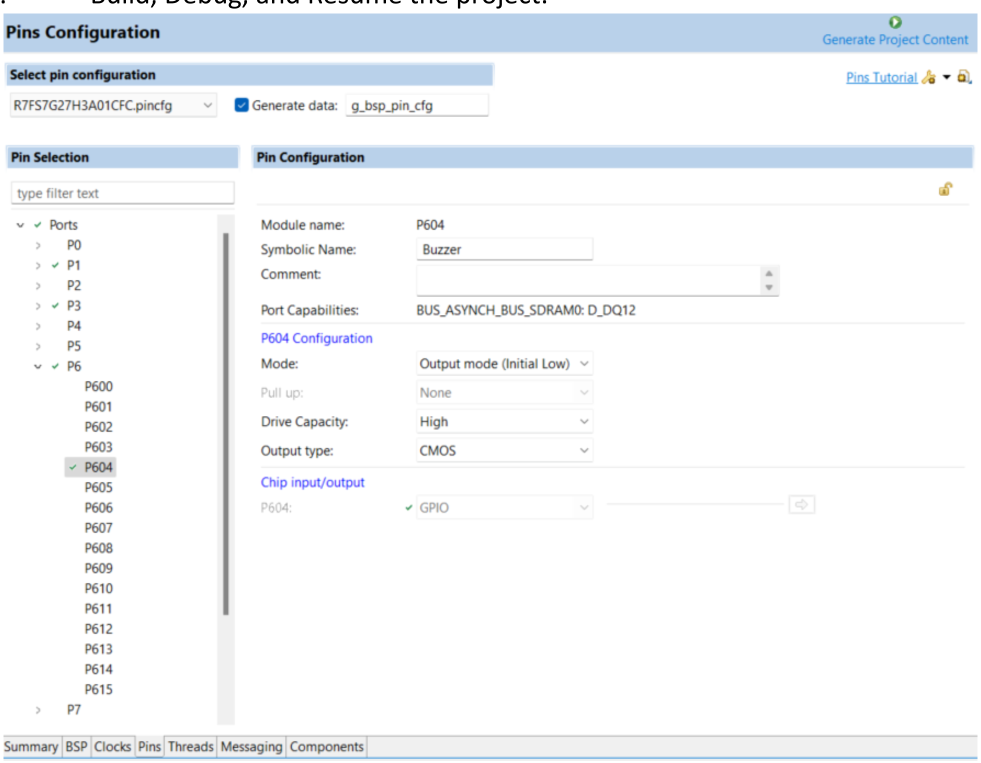
A. Test the MCU and SSP

A.1. Make a new project (Lab7) with the **Blinky** template, build, debug, and download the program to the Renesas Synergy SK-S7G2 kit. Resume twice to see the LEDs blinking. If you face any errors in this step, check the board, device, and cable connections.

Board: S7G2 SK
Device: R7FS7G27H3A01CFC

B. Configure Buzzer I/O pin.

B.1. Select the **Synergy Configuration** Perspective.
B.2. From **Project Explorer** select **Lab7** and expand it. Then select **configurations.xml** and select **Pins** tab.
B.3. Expand **Ports** to see the list of available ports.
B.4. Select **P6**.
B.5. Select **P604** (Pin 04 of Port 6).
B.6. From Pin Configuration, Change the Symbolic name to **Buzzer**.
B.7. Change the P604 Configuration Mode to **Output mode (Initial Low).**
B.8. Change the P604 Configuration Drive Capacity to **High.**
B.9. Do not change any other settings!
B.10. **Generate Project Content** by clicking on the green icon (Top-left corner).
B.11. Build, Debug, and Resume the project.

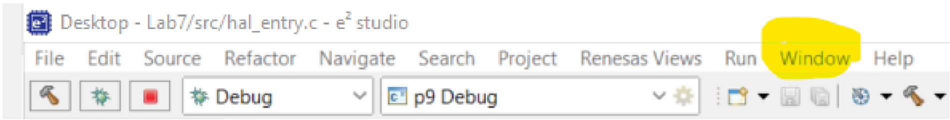


C. Debug

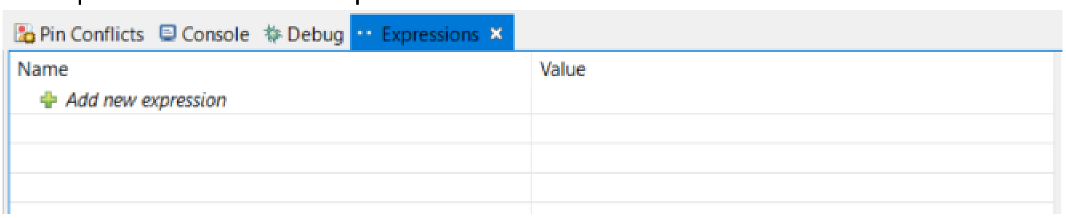
C.1. From **Project Explorer** open **Lab7/src/hal_entry.c**.

C.2. Add a regular breakpoint in line 58 of file **hal_entry.c**. Select the line and right-click to add the breakpoint.

C.3. On the top of the page, you can see the **Window** menu, select it.

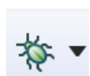


C.4. From the **Window** menu, select **Show View, Others … -> Debug - > Expressions**. You will notice the Expression Window will open in e2Studio.

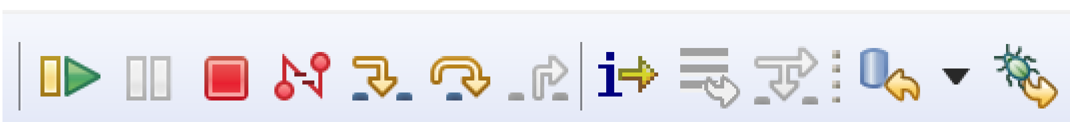


C.5. In the **Expressions** Window, click on the plus sign to ***Add new expression***.

C.6. Type **level** (It is the name of the variable we want to monitor, hence, case-sensitive)

C.7. Click on debug. 

C.8. Use the debug tools (**Resume,** F8, and **Step Into**, F5) to learn how the code is being executed and debug your code when necessary.

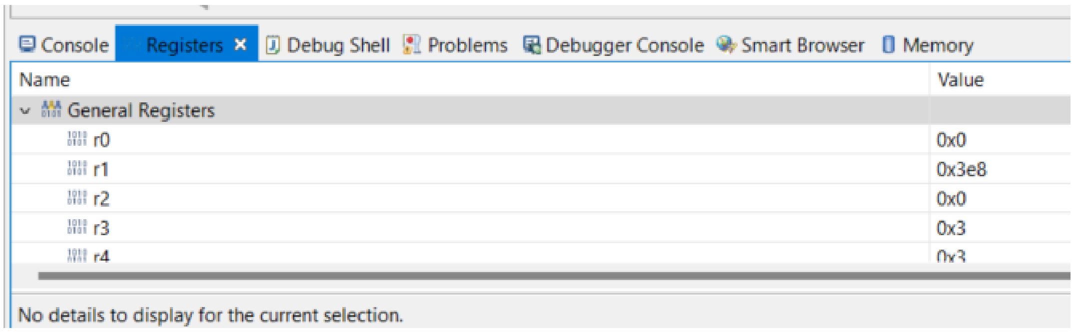


D. Registers

The ARM processor has general purpose and special purpose registers. You may find the name and specifications of the registers in the Datasheet or User Manual of the processor. In the e2studio, you can monitor the values of the registers and program them directly with Assembly language.

D.1. From the **Window** Menu select **Show View -> Registers**.

D.2. In the **Registers** Window, Select **General Registers**. Double-click on it to see the list of the registers.



D.3. The value of the registers will be updated while debugging the code. Continue Resuming the code and watch r5.
R5 alternating between 0 and 1

D.4. Restart the debug by pressing the Restart icon . If the Restart icon is missing, you can add it from Window / Perspective / Customize Perspective / Renesas Debug / Restart.

D.5. Watch the value of **pc** Register right after the restart.

Q1. What value is stored in the **pc** register?
0x2bc8

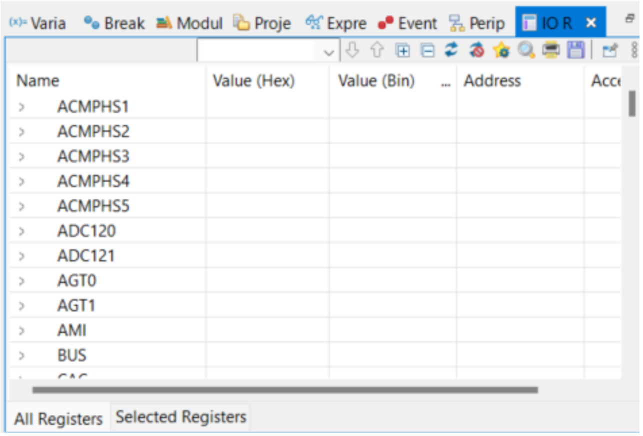
Q2. Which line of the code is being executed?
Line 58 of hal_entry.c

Q3. In which address of the program memory is this code stored?
0x2bc8

D.6. Press F5  to run the code step by step. Watch the ***program counter*** **(pc register)** to find out which line is being processed at each step.

0x2bde line 58
0x2bea line 58
0x2bf0 line 68
0x269c line 70
0x26aa line 71

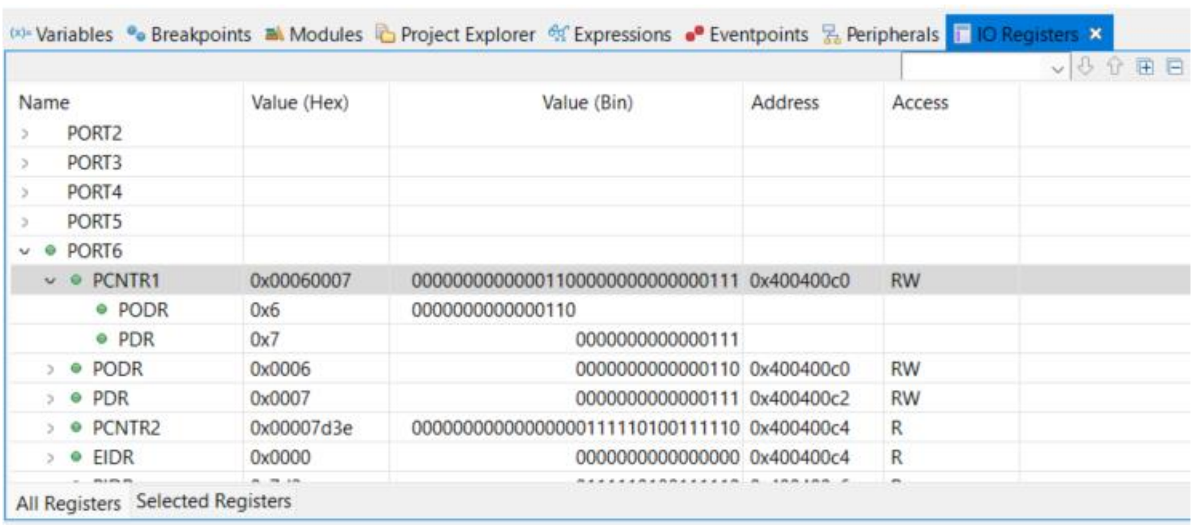
D.7. From the **Window** menu, select **Show View, Others … -> Debug - > IO Registers.** The **IO Register** Tab will appear beside the **Expressions** Window.



You can see a list of registers that control the IO ports.

D.8. Scroll down to find PORT6. Right-click on PORT6 and add it to the list of the selected registers (**Add to Selected Registers**).

D.9. Expand the PORT6 register to see the PCNTR1 register.



Q4. Terminate and debug again. Resume the code to initiate the system (Run SystemInint();) What are the Hex and Binary values of PDR of PORT 6 after the SystemInit function is processed?

Hex: 0x17

Binary: 00000000 00010111

Q5. Which pins of Port 6 are configured as inputs and which are outputs?

Hint: To answer this question, compare the values of each bit of **PDR** with the port control register description from User Manual (next page).

1 means output, so the three on-board LEDs (P600, P601, P602) are outputs, also the P604 we configured in part B. The rest are input pins.

20.2 Register Descriptions

20.2.1 Port Control Register 1 (PCNTR1/PODR/PDR)

Address(es): PORT0.PCNTR1 4004 0000h, PORT1.PCNTR1 4004 0020h, PORT2.PCNTR1 4004 0040h, PORT3.PCNTR1 4004 0060h, PORT4.PCNTR1 4004 0080h, PORT5.PCNTR1 4004 00A0h, PORT6.PCNTR1 4004 00C0h, PORT7.PCNTR1 4004 00E0h, PORT8.PCNTR1 4004 0100h, PORT9.PCNTR1 4004 0120h, PORTA.PCNTR1 4004 0140h, PORTB.PCNTR1 4004 0160h

PORT0.PODR 4004 0000h, PORT1.PODR 4004 0020h, PORT2.PODR 4004 0040h, PORT3.PODR 4004 0060h, PORT4.PODR 4004 0080h, PORT5.PODR 4004 00A0h, PORT6.PODR 4004 00C0h, PORT7.PODR 4004 00E0h, PORT8.PODR 4004 0100h, PORT9.PODR 4004 0120h, PORTA.PODR 4004 0140h, PORTB.PODR 4004 0160h

PORT0.PDR 4004 0002h, PORT1.PDR 4004 0022h, PORT2.PDR 4004 0042h, PORT3.PDR 4004 0062h, PORT4.PDR 4004 0082h, PORT5.PDR 4004 00A2h, PORT6.PDR 4004 00C2h, PORT7.PDR 4004 00E2h, PORT8.PDR 4004 0102h, PORT9.PDR 4004 0122h, PORTA.PDR 4004 0142h, PORTB.PDR 4004 0162h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	PODR 15	PODR 14	PODR 13	PODR 12	PODR 11	PODR 10	PODR 09	PODR 08	PODR 07	PODR 06	PODR 05	PODR 04	PODR 03	PODR 02	PODR 01	PODR 00
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	PDR15	PDR14	PDR13	PDR12	PDR11	PDR10	PDR09	PDR08	PDR07	PDR06	PDR05	PDR04	PDR03	PDR02	PDR01	PDR00
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Symbol	Bit name	Description	R/W
b15 to b0	PDRn	Pmn Direction	0: Input (functions as an input pin) 1: Output (functions as an output pin).	R/W
b31 to b16	PODRn	Pmn Output Data	0: Output low 1: Output high.	R/W

m = 0 to 9, A, B
n = 00 to 15

Port Control Register 1 is a 32- and 16-bit readable/writable register that controls port direction and port output data.

PCNTR1 specifies both the port direction and the output data, in 32-bit units. PDR (PCNTR1 bits [15:0]) and PODR (PCNTR1 bits [31:16]) specify port direction and port output data, respectively, and are accessed in 16-bit units.

The PDRn bits select the input or output direction for individual pins on the associated port when the pins are configured as general I/O pins. Each pin on port m is associated with a PORTm.PCNTR1.PDRn bit. The I/O direction can be specified in 1-bit units. Bits associated with non-existent pins are reserved. Reserved bits are read as 0. The write value should be 0. P000 to P007 and P200 are input only, so PORT0.PCNTR1 bits [7:0] and PORT2.PCNTR1 bit [0] are reserved. The PDRn bit in the PORTm.PCNTR1 register serves the same function as the PDR bit in the PFS.PmnPFS register.

The PODRn bits hold data to be output from the general I/O pins. Bits associated with non-existent pins are reserved. Reserved bits are read as 0. The write value should be 0. P000 to P007 and P200 are input only, so PORT0.PCNTR1 bits [23:16] and PORT2.PCNTR1 bit [16] are reserved. Writes to P000 to P007 and P200 have no effect. The PODRn bit in the PORTm.PCNTR1 register serves the same function as the PODR bit in the PFS.PmnPFS register.

Q6. Resume the program until the values of the **level** Variable become high. Write down the Hex value of PDR and PODR registers from PORT6 while level = high.

PDR = 0x17
PODR = 0x7

Q7. Resume the program until the values of the **level** Variable become low. Write down the Hex value of PDR and PODR registers from PORT6 while level = low.

PDR = 0x17
PODR = 0x0

Q8. What would be the status of the LEDs when the **PCNTR1** Register of **Port 6** is equal to **0x00060007**? You can verify your answer in the next step.
0000000000000110   0000000000000111

0 means ON, that turn the LED 1 ON

E. Assembly Code

In this step, we want to write a code in Assembly. The objective is to get familiar with the Assembly language. Assembly language is useful in writing high-performance codes with optimal runtimes, such as real-time signal processing (For example, live video processing applications).

E.1. Replace the code in **hal_entry.c** with the following code. Here, we use **in-line Assembly,** which means using Assembly language within a C code. To do that, we put the assembly instructions inside `__asm("");`

```
#include "hal_data.h"
void hal_entry(void)
{
    // Reset pin 0 to 0 to turn green led ON (110 = 6),
    //pins 0,1,2 are output (111 = 7),
    R_IOPORT6->PCNTR1 = 0x00060007;

    __asm("     ldr r4, =5");
    __asm("loop:                ");
    __asm("     subs r4,r4,#1   ");
    __asm("     bne loop        ");

    // pins 0,1,2 are output, set pin 0 to 1 to turn green led OFF
    R_IOPORT6->PCNTR1 = 0x00070007;
}
```

E.2. Read the code description to understand how it works:

- `R_IOPORT6->PCNTR1` refers to the PCNTR1 register from PORT6, or Port 6 control register 1.

- `ldr r4, =5` : load the `decimal` value 5 to register r4.
- `loop:` determines a lable for this line of code.
- `subs r4,r4,#1` : subtract 1 from r4 and save the result in r4.
- `bne loop` : compares the result of the previous calculation (r4) with zero, if the result is not equal to zero, jump to the line labelled as *loop*. Here, *loop* is a lable. Note in a code you must have unique labels. You can choose other labels for example loop1, loop2, ….

E.3. Build the project.

E.4. Add two breakpoints beside lines 10 and 11:

```c
#include "hal_data.h"
void hal_entry(void)
{
    // Reset pin 0 to 0 to turn green led ON (110 = 6),
    //pins 0,1,2 are output (111 = 7),
    R_IOPORT6->PCNTR1 = 0x00060007;

    __asm("     ldr r4, =5");
    __asm("loop:                ");
    __asm("     subs r4,r4,#1   ");
    __asm("     bne loop        ");

    // pins 0,1,2 are output, set pin 0 to 1 to turn green led OFF
    R_IOPORT6->PCNTR1 = 0x00070007;
}
```

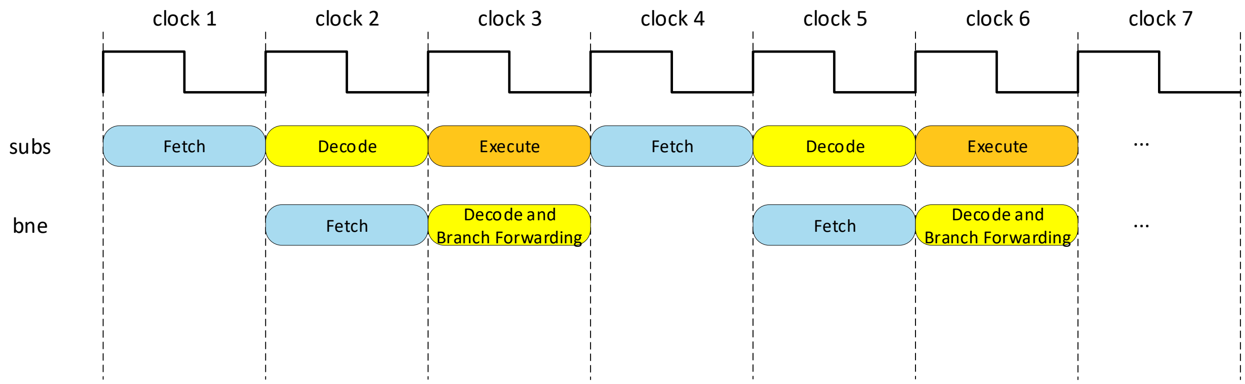
E.5. Debug the project.

Q9. Write down the value of the register r4. Resume. When paused, write down the value of r4 (From general-purpose registers). Resume, and write down the value of r4 in each step until r4=0.

r4:
0x1ffe0964
0x1ffe0964
0x5
0x4
0x4
0x3
0x3
0x2
0x2
0x1
0x1
0x0

Q10. How does this code affect the statuses of the built-in LEDs?
While the programming is counting down 4r, the green LED stays on. After r4=0, all LEDs are set to 1, so the green LED turns off.

E.6. When the processor is busy with executing the assembly lines, it cannot do anything else. So, we will have a delay. The delay made with this method is called a busy wait. It is the simplest form of making delays in microprocessors (But not the best one). The delay function you used earlier is using a similar assembly code. This loop has two instructions, subs (Substruction) and bne (branch not equal). This is the Timing Diagram for the execution of the loop in the assembly code. Due to branch forwarding at the decode stage of the BNE instruction, each loop takes 3 clock cycles.


clock 1
clock 2
clock 3
clock 4
clock 5
clock 6
clock 7
subs
Fetch
Decode
Execute
Fetch
Decode
Execute
...
bne
Fetch
Decode and Branch Forwarding
Fetch
Decode and Branch Forwarding
...

E.7. From the project configuration.xml, navigate to Clocks and find the frequency of the ICLK.

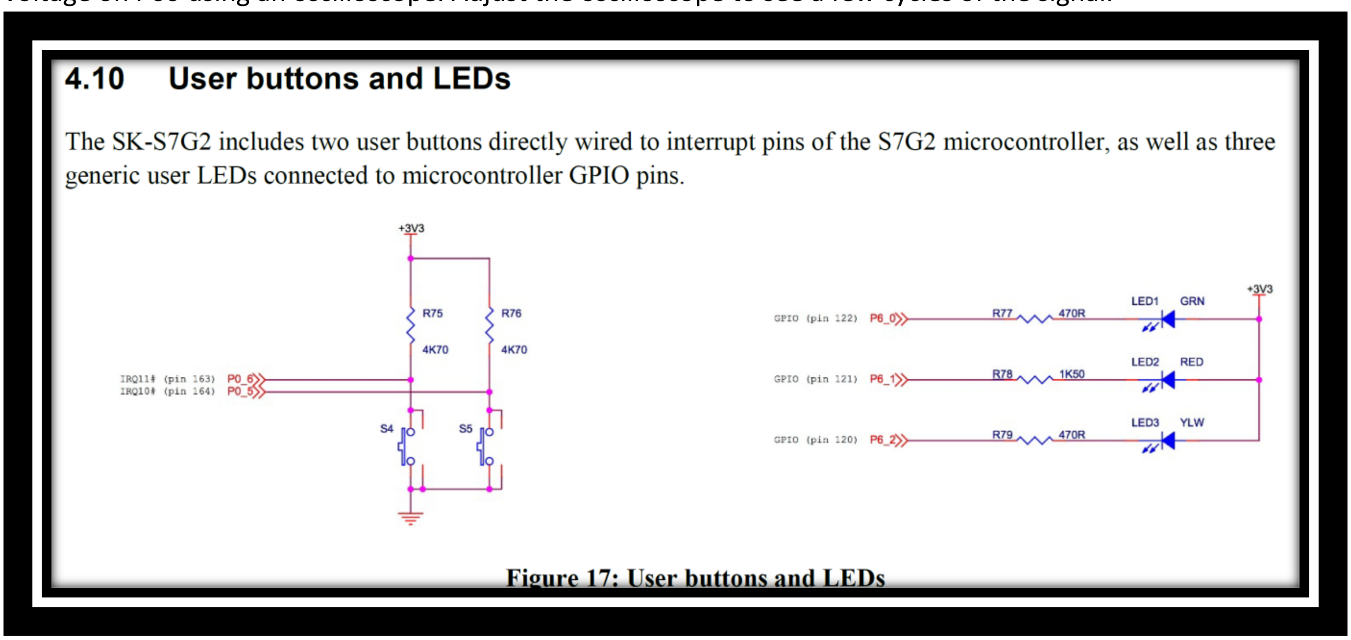
Q11. How much is the frequency of the ICLK?
240MHz

Q12. How long is one period of the ICLK? Express your answer in nanoseconds.
1/ (240e6)= 4.167 nanoseconds

Q13. Calculate the time it takes for the processor to process the instructions in the loop (processing time of each loop in ns).
3*4.167 = 12.501 nanoseconds

Q14. Calculate the number of times the loop must be executed to keep the CPU busy for 0.9s.
0.9/12.5e-9 = 72,000,000

E.8. Modify line 8 of the code to write a very large number to r4. Build and debug the code. Remove the breakpoint so the code will be executed with the CPU speed without any pause. Measure the voltage on P60 using an oscilloscope. Adjust the oscilloscope to see a few cycles of the signal.

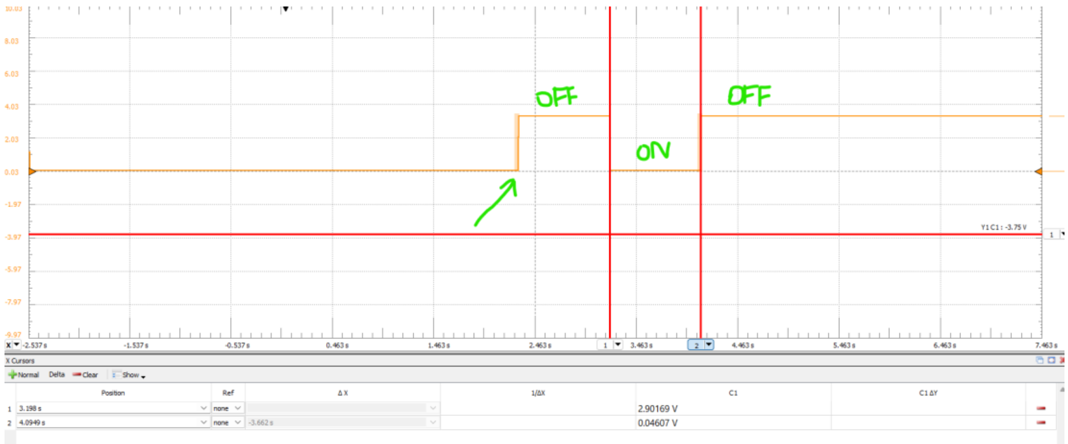


4.10 User buttons and LEDs

The SK-S7G2 includes two user buttons directly wired to interrupt pins of the S7G2 microcontroller, as well as three generic user LEDs connected to microcontroller GPIO pins.

Figure 17: User buttons and LEDs

Q15. Take a snapshot of the oscilloscope.



The program enter hal_entry() at green arrow. Before that all LEDs are ON. As shown below, to see the effects, set all LEDs OFF for 0.9s before setting the green LED to ON for also 0.9s.

```
R_IOPORT6->PCNTR1 = 0x00070007;
__asm("      ldr r4, =72000000");
__asm("loop0:                 ");
__asm("      subs r4,r4,#1   ");
__asm("      bne loop0        ");

R_IOPORT6->PCNTR1 = 0x00060007;

__asm("      ldr r4, =72000000");
__asm("loop:                 ");
__asm("      subs r4,r4,#1   ");
__asm("      bne loop        ");

// pins 0,1,2 are output, set pin 0 to 1 to turn green led OFF
R_IOPORT6->PCNTR1 = 0x00070007;
```

Q16. Measure the period of the signal on P60.

0.8908s

Q17. In the previous step, you calculated the time it took for the processor to process the instructions in the loop. Verify your calculations using the oscilloscope measurements.

The result matches the theoretical value of 0.9s.

Q18. Do you notice any errors in calculating the processing time of the loop? Correct your calculations if possible.

The error is 0.01s less than ideal assuming no measurement error, then the corrected result would be 72000000+0.01/12.5e-9 = 72800000 loops for 0.9 seconds

E.9. Use the assembly instructions you learned in this lab to write a program to generate a PWM signal. To keep it simple, generate a PWM signal with a duty cycle of 50%, and a period of 1.8s. Show the output on built-in LED3. Keep LED1 off, and LED2 ON.

0000000000000001 0000000000000111 Turns the LED3 and LED2 ON. -> 0x00010007

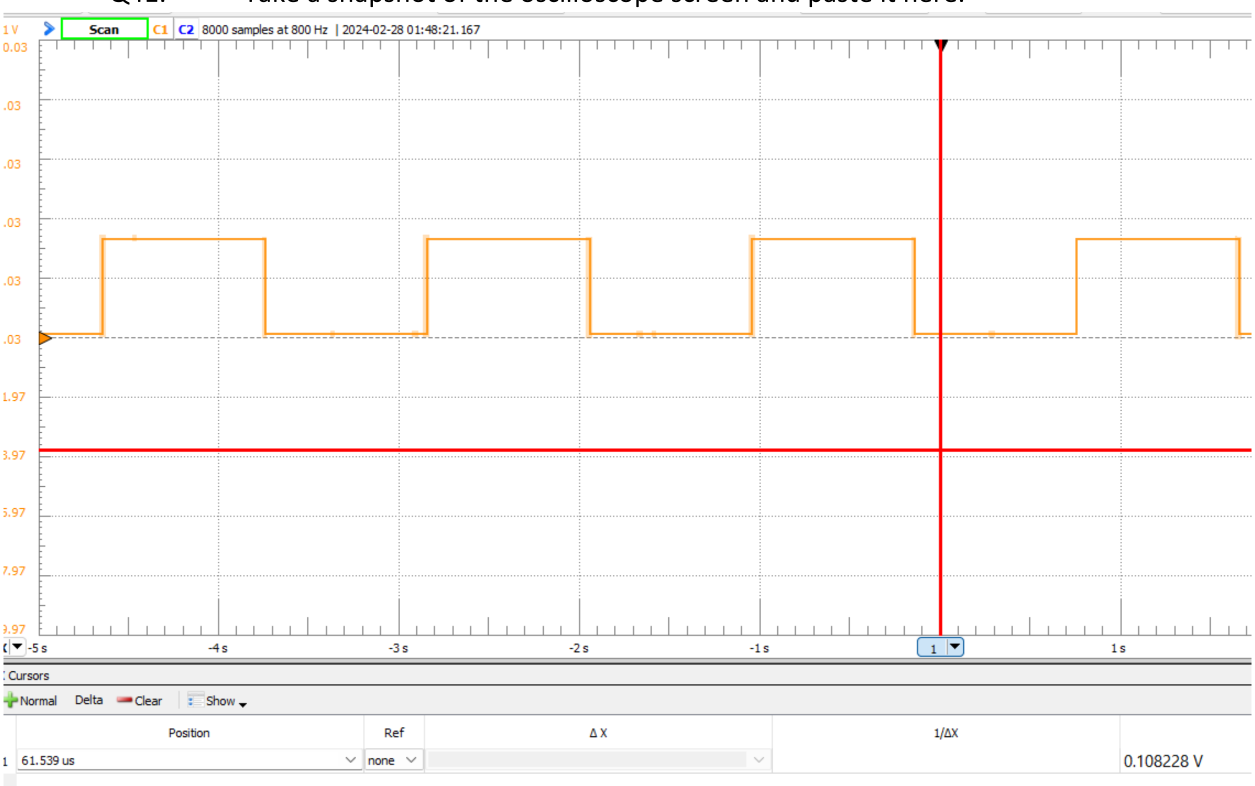
0000000000000101 0000000000000111 Turns the LED2 ON. -> 0x00050007

Q19. Take a snapshot of your assembly lines and paste it here.

```
Q20.    #include "hal_data.h"
Q21.    void hal_entry(void)
Q22.    {
Q23.        while (1)
Q24.        {
Q25.            R_IOPORT6->PCNTR1 = 0x00010007;
Q26.
Q27.            // delay 0.9s
Q28.            __asm(" LDR r4, =72000000");
Q29.            __asm(" ON_LOOP: ");
Q30.            __asm(" SUBS r4, r4, #1"); // r4-=1
Q31.            __asm(" BNE ON_LOOP"); // if r4==0: break
Q32.
Q33.            // turn LED OFF
Q34.            R_IOPORT6->PCNTR1 = 0x00050007;
Q35.            __asm(" LDR r4, =72000000");
Q36.            __asm(" OFF_LOOP: ");
Q37.            __asm(" SUBS r4, r4, #1");
Q38.            __asm(" BNE OFF_LOOP");
Q39.        }
Q40.    }
```

E.10. Measure the PWM signal you generated using an oscilloscope.

Q41. Take a snapshot of the oscilloscope screen and paste it here.



Q42. What is the advantage and disadvantage of a busy-wait software delay?

It’s easy to implement. It has minimum delay to start the delay and after it finishes.
It consumes all the CPU computing resources, so no multi-tasking would be available during busy-wait.

F. Clocks

F.1. Select the **Clocks** tab from the project Configuration. You will see the clocks used in the system. **XTAL** is giving you the source of oscillation in your circuit. It is like the heart of your system. Find the **XTAL** frequency. Do not change it.

Q43. How much is the Chrystal (XTAL) frequency?

24MHz

F.2. In the clock configuration, you can see frequency dividers and multipliers used to produce different clocks in your system. Hover your mouse on the frequency values to find more information about **ICLK, PCLKA**, and **PCLKC**.

Q44. How much is the ICLK frequency?

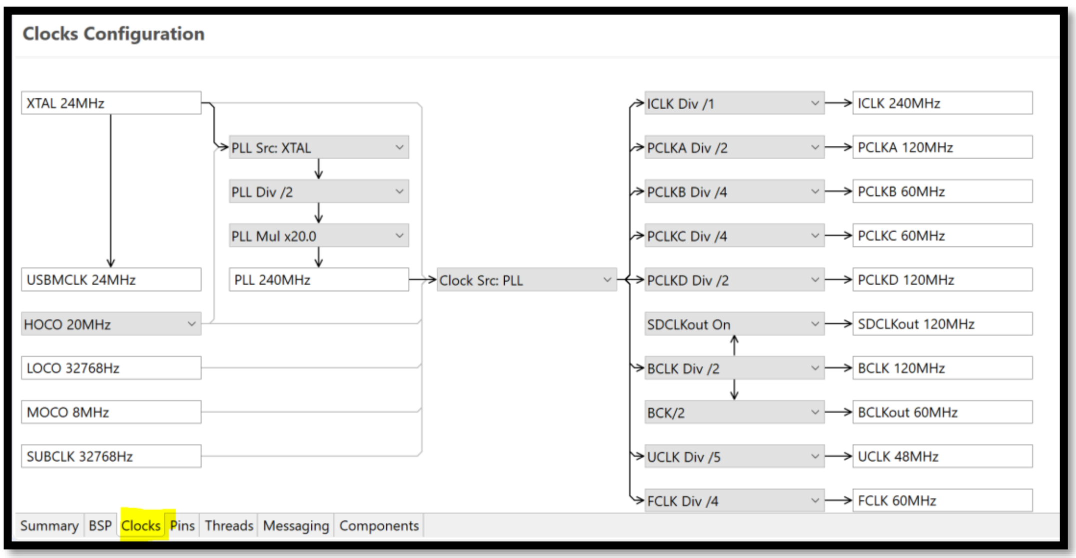
240MH

Q45. Which clock is used by ADC module?

PCLKC

Q46. Which clock is used by CPU?

ICLK, which means internal clock.



F.3. Change the XTAL to 12MHz.

F.4. Build and run the project.

Q47. Measure the frequency of the square wave on P60. (But P62 is one that's generating square wave based on previous steps.)

Period = 1.8s, f=0.556Hz

F.5. Return the XTAL to 24MHz, build and run the project.

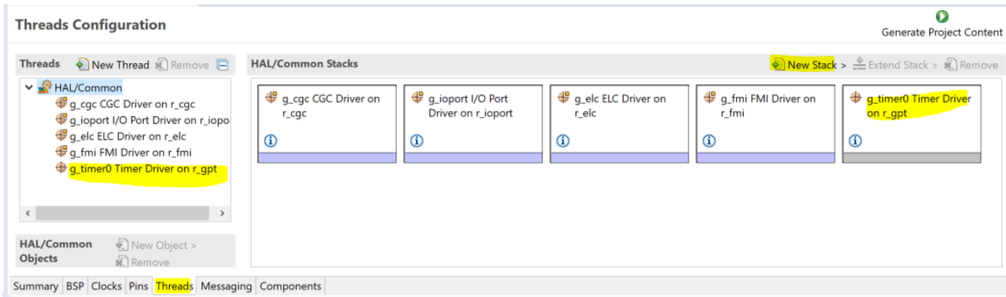
Q48. Measure the frequency of the square wave on P60.

It stays the same.

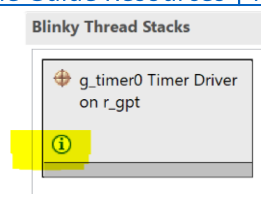
G. Timer

In this step, we want to configure an internal hardware timer and use it to generate different frequencies.

G.1. In the e2Studio, from the File Explorer, open **configurations.xml**.
G.2. Select HAL/Common Thread from Threads tab.
G.3. Click on the **New Stack** to add a timer. New Stack/Driver/Timers/Timer Driver on r_gpt.



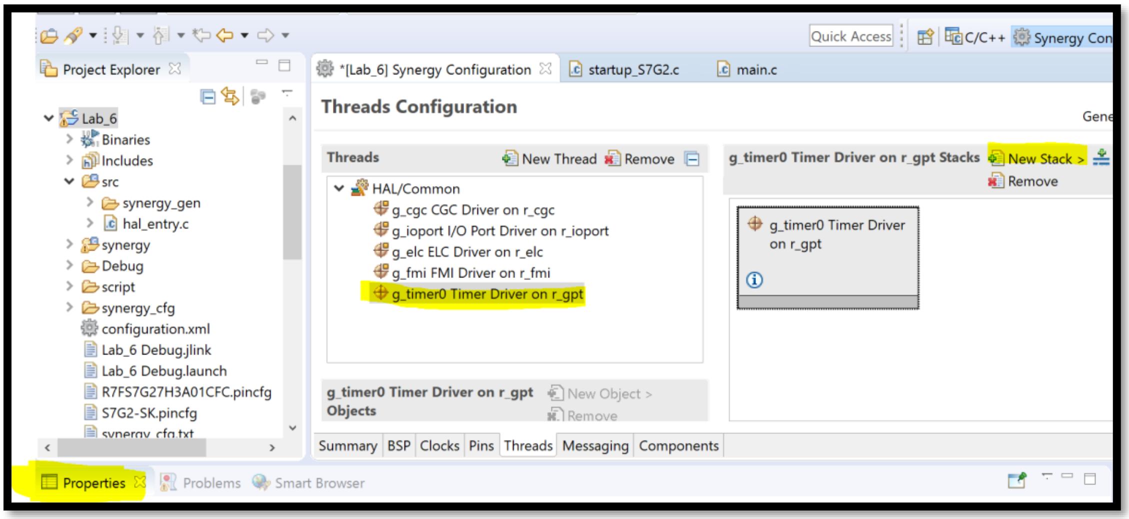
G.4. Click on the info icon on the timer stack to find **r_gpt Module Guide Resources** online. The following link will open: [r_gpt Module Guide Resources | Renesas Customer Hub]



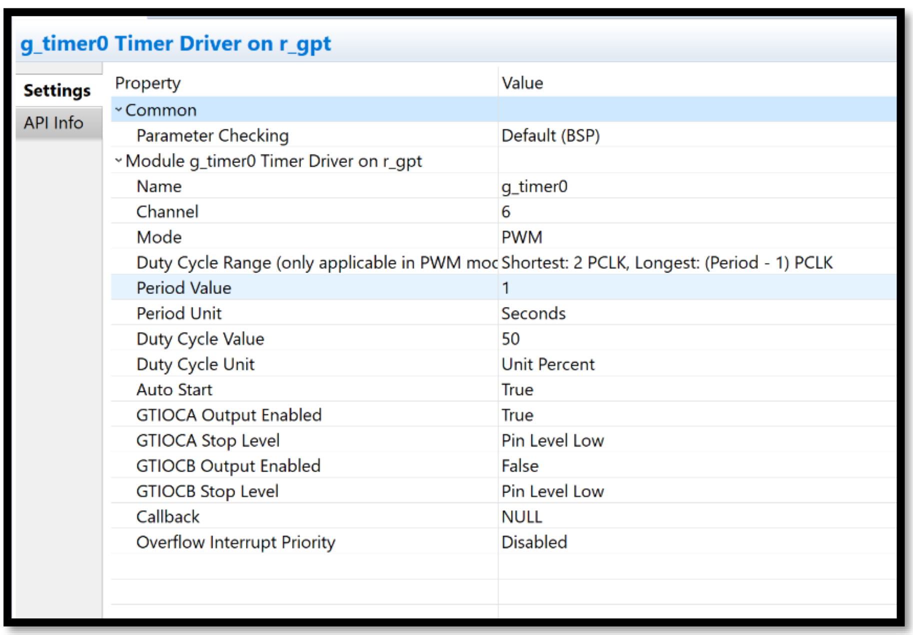
G.5. Based on the information provided in the r_gpt Module guide, **PCLKD** is the core clock of this module. Refer to the Clock tab, and find the value of this clock source.

Q49. How much is PCLKD=? Refer to the Clock tab and find the value of this clock source (Ensure XTAL = 24MHz).
120MHz

G.6. Select the Timer stack and open its **Properties**. If the Property window was not available, select it from Window -> Show View -> Properties.



G.7. Select the timer Thread, then Click on the Properties. Edit the Properties of the timer as shown in the following figure. Hovering the mouse on each property reveals more information about them.



G.8. Replace the code in the hal_entry.c by the following code. Read the comments in the code. Hover on the functions new to you to learn about their syntax.

```
/* HAL-only entry function */
#include "hal_data.h"

// Define the number of counts per millisecond (1 count per clock tick, clock rate is
120MHz)
// So, there are 120E6 ticks per second.
// Divide by 1000 to get ticks / millisecond
#define COUNTS_PER_MILLISECOND (120E6 / 1000)
bsp_leds_t leds;

void hal_entry(void)
{
     ioport_level_t led_level = IOPORT_LEVEL_HIGH; //off
     R_BSP_LedsGet(&leds);
     g_ioport.p_api->pinWrite(leds.p_leds[0],led_level);
     g_ioport.p_api->pinWrite(leds.p_leds[1],led_level);
     g_ioport.p_api->pinWrite(leds.p_leds[2],led_level);


    // Variable to hold counts
     timer_size_t counts = 0;

    // Open the timer using the configured options from the configurator
     g_timer0.p_api->open(g_timer0.p_ctrl, g_timer0.p_cfg);
    //Reset the counter to initial value
     g_timer0.p_api->reset(g_timer0.p_ctrl);

    // Main Loop
    while(1)
    {
     g_timer0.p_api->counterGet(g_timer0.p_ctrl, &counts);
    // Check if 500ms has elapsed => This should be a helper function at some point
    // Need to look if the PBCLK settings are stored in a define somewhere...
            if (counts > (500*COUNTS_PER_MILLISECOND))
            {

            /* Determine the next state of the LEDs "!" implements the logic not*/
                led_level = !led_level;

                // Reset the timer to 0
                g_timer0.p_api->reset(g_timer0.p_ctrl);
            }

      /* Update LED1 */
      g_ioport.p_api->pinWrite(leds.p_leds[0],led_level);

    }
}
```

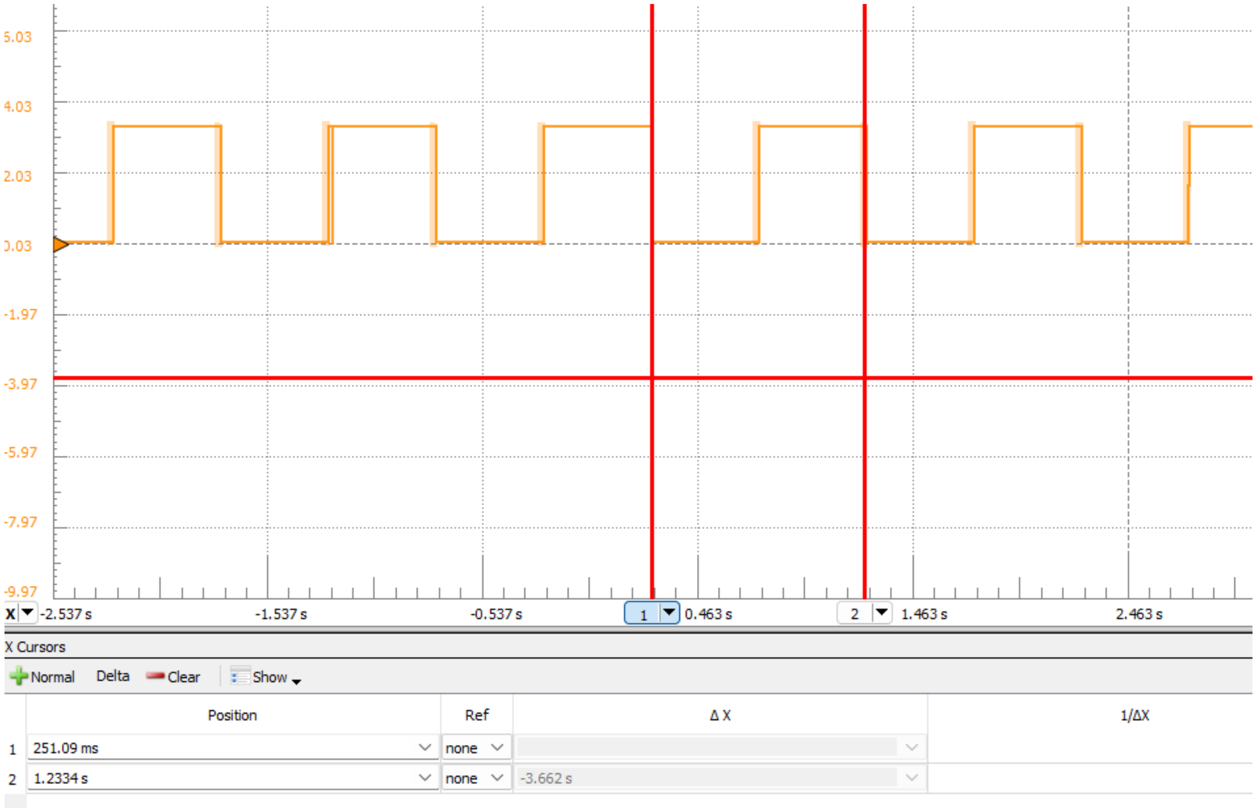
G.9. Build, debug, and run the code. One of the LEDs should start blinking.

Q50. Which LED is blinking?

The green LED, or LED 1

Q51. Measure the blinking frequency.

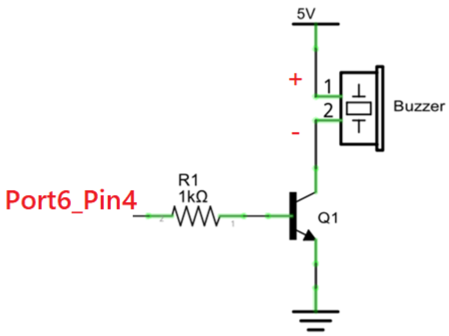
1Hz



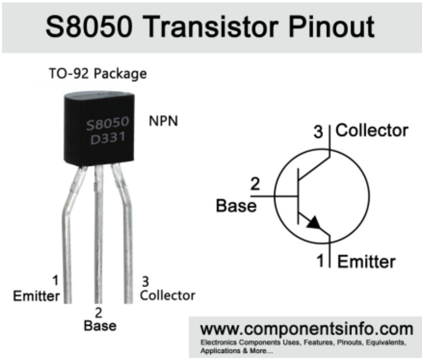
H. Buzzer Circuit

The human eye can detect low-frequency blinking, but when the frequency is high, we do not sense blinking (Movie vs picture!). So, to check frequencies in the order of hundreds of Hz, to a few kHz, we can use a buzzer and hear its sound.

H.1. Disconnect the board from the computer (USB cable). Build the buzzer circuit.



In this circuit, Q1 is S8050 NPN Transistor that is used to amplify the pin's current. Make sure you connect the transistor pins correctly to the circuit. Refer to the S8050 Transistor Pinout and ask for help when in doubt.



Note: The buzzer is a component with polarity. The positive pin is indicated by a + sign on the top of the component.

H.2. To check the circuit, you can temporarily connect the input (R1) to Port6 Pin 0 (P60) instead of P64. You should be able to hear a low sound while LED blinks. After you ensure the circuit is working, connect the R1 to P64 as shown in the circuit.

I. Music Notes

I.1. Modify the *hal_entry.c* code to do the following tasks:

- Generate a square wave (50% duty cycle) with the frequency of freq[1]=440 Hz on the output pin (P64).

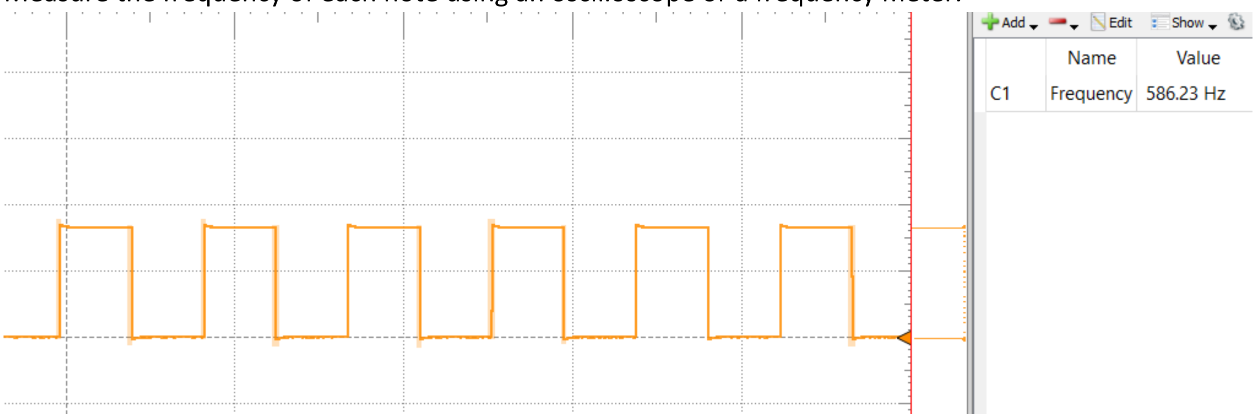
Q52. How much is the period value and unit you selected for the timer?

Period = 1sec, or 1000us

I.2. Use an array to easily modify your code to play other music notes.

uint32_t freq[7] = {440,494,523,587,659,698,783};

I.3. Measure the frequency of each note using an oscilloscope or a frequency meter.



Q53. Summarize the measurements in a table.

Nominal Frequency (Hz)	Frequency (Hz) From measurement
440	439.17
494	492.09
523	521.23
587	586.23
659	657.92
698	695.24
783	778.67

Q54. **OPTIONAL (BONUS*):** Write a program to play a melody. Submit your project, and a video (with sound) showing the result. You shall not use software delay.

Successfully play do re mi fa sol la ti do by implementing an extra counter for the timer.

Q55. Zip your project and upload it in the Lab Folder.

SUBMISSION

Please submit the following to the Lab 7 folder:

1. A document containing your answers to the Lab questions and the screenshots.
2. The Zip folder of your last project.

```
/* HAL-only entry function */
#include "hal_data.h"

// Define the number of counts per millisecond and microsecond (1 count per clock tick, clock rate is 120MHz)
// So, there are 120E6 ticks per second.
// Divide by 1000 to get ticks / millisecond
#define COUNTS_PER_MILLISECOND (120E6 / 1000)
      #define COUNTS_PER_MICROSECOND (120E6 / 1000000)
      bsp_leds_t leds;


      // Frequency array, freq[0] is for testing
      uint32_t freq[9] = {2, 440, 494, 523, 587, 659, 698, 783, 830};
      uint32_t f_index = 1;


      void hal_entry(void)
      {
           ioport_level_t led_level = IOPORT_LEVEL_HIGH; //off
           ioport_level_t buzz_level = IOPORT_LEVEL_HIGH; //off
           R_BSP_LedsGet(&leds);
           g_ioport.p_api->pinWrite(leds.p_leds[0],led_level);
           g_ioport.p_api->pinWrite(leds.p_leds[1],led_level);
           g_ioport.p_api->pinWrite(leds.p_leds[2],led_level);
           g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_04,buzz_level);


          // Variable to hold counts
           timer_size_t counts = 0;
           uint32_t duration_counts = 0;

          // Open the timer using the configured options from the configurator
           g_timer0.p_api->open(g_timer0.p_ctrl, g_timer0.p_cfg);
          //Reset the counter to initial value
           g_timer0.p_api->reset(g_timer0.p_ctrl);



          // Main Loop
          while(1)
          {
           g_timer0.p_api->counterGet(g_timer0.p_ctrl, &counts);
           uint32_t timer_period = (1000000  / (freq[f_index]*2));
```

```
            // Check if 500ms has elapsed => This should be a helper function at some point
            // Need to look if the PBCLK settings are stored in a define somewhere...
                    if (counts > (timer_period*COUNTS_PER_MICROSECOND))
                    {

                    // determine the next state for the led and buzzer PWM
                        led_level = !led_level;
                        buzz_level = !buzz_level;

                        // Reset the timer to 0
                        g_timer0.p_api->reset(g_timer0.p_ctrl);
                    }

              //write the state
              g_ioport.p_api->pinWrite(leds.p_leds[0],led_level);
              g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_04,buzz_level);
              duration_counts +=1;

              if (duration_counts >300000){
                  f_index +=1;
                  duration_counts = 0;
               }
              if (f_index>9){
                  f_index = 1;
               }

            }

        }
```

REFERENCE

https://components101.com/buzzer-pinout-working-datasheet

https://www.componentsinfo.com/s8050-pinout-equivalent/

https://pages.mtu.edu/~suits/notefreqs.html

https://en.wikipedia.org/wiki/Music_and_mathematics

Advanced Debugging: https://www.youtube.com/watch?v=HmKiCZA0XoU&feature=youtu.be

ARM registers: https://developer.arm.com/documentation/dui0068/b/writing-arm-and-thumb-assembly-language/overview-of-the-arm-architecture/registers#:~:text=ARM%20processors%20have%2037%20registers,processor%20exceptions%20and%20privileged%20operations.

User manual of S7G2 from Renesas