

Technical Considerations regarding Timer Interrupts for Development Boards: M5Stack, BBC micro:bit and Arduino

Atsushi Matsubara

Production Division

1 Introduction

Timer interrupt means to accomplish the desired processing or the designed purpose depending on the elapsed time. This interrupt can analyze data, handle events, measure physical quantities using sensors and control or operate hardware devices at the specific timing in computers for numerical analysis and embedded systems using development boards. This perspective leads to the view that the timer interrupt can be regarded as one of important software technologies for online or interactive processing in measurement and control of home appliances and industrial devices.

A timer is configured using the clock signal with the unique frequency to drive the processor into a computer. Operating effectively and efficiently electronic circuits such as prescaler, comparator and counter - hardware components (peripherals) of a computer - from a software implementation perspective, leads to the generation of the interrupt events.

This paper provides typical and highly versatile skills for timer interrupt using three development boards: M5Stack, BBC micro:bit and Arduino. The development environment for creating programs to install in these boards is unified as “Arduino IDE”, which has libraries to support them. The following attempts allow their timers to drive: M5Stack utilizes variables and functions of “hw_timer” structure, micro:bit does those of “NRF_TIMER N ” (where $N \in \{0, \dots, 4\}$) one and Arduino sets the appropriate parameter values to the registers of the hardware Timer (e.g., Timer0, Timer1 or Timer2).

The paper introduces three examples for timer interrupt systems: first is a countdown system using a 4-Digit Display module, next creation of musical scales (synchronous superposition of them: chords) using PWM signals and last “one-to-one” data transmission / reception using wireless communication modules named “XBee”. These examples will suggest that interrupt skills lead to be widely useful for a system development. In addition, the adopted boards are M5Stack Basic V2.7, micro:bit v2.21 and Arduino Leonardo, respectively.

2 Timer Interrupt

2.1 Mechanism: Essential Elements

Although there are several mechanisms for timer interrupt depended on development boards, the paper presents the most popular or orthodox one as a fundamental concept in Figure 1.

A development board has not only CPU (Central Processing Unit, consequently processor) and memories but also peripherals, which are the mentioned above electronic circuits (prescaler, comparator, counter, etc.) and also include capabilities of timer and input/output (serial) ports. Clock signal with the frequency f_{pre} to drive the peripherals is generated via circuits with crystal oscillator using one with the frequency $f_{clk}(> f_{pre})$ to do the processor.

To occur timer interrupt, the following three circuits are operated appropriately based on the peripheral clock signal with f_{pre} : 1) prescaler creates a clock signal for driving counter, which has the frequency f_{cnt} that is $(1/p)$ times lower than the frequency f_{pre} of the peripheral clock signal using division ratio p , 2) counter updates count number m at regular time intervals $t_{cnt} = 1/f_{cnt}$, 3) comparator takes on the role of matching the number m with preset threshold C , and the condition for raising the interrupt is thus $m = C$ (the m is denoted as m_C).

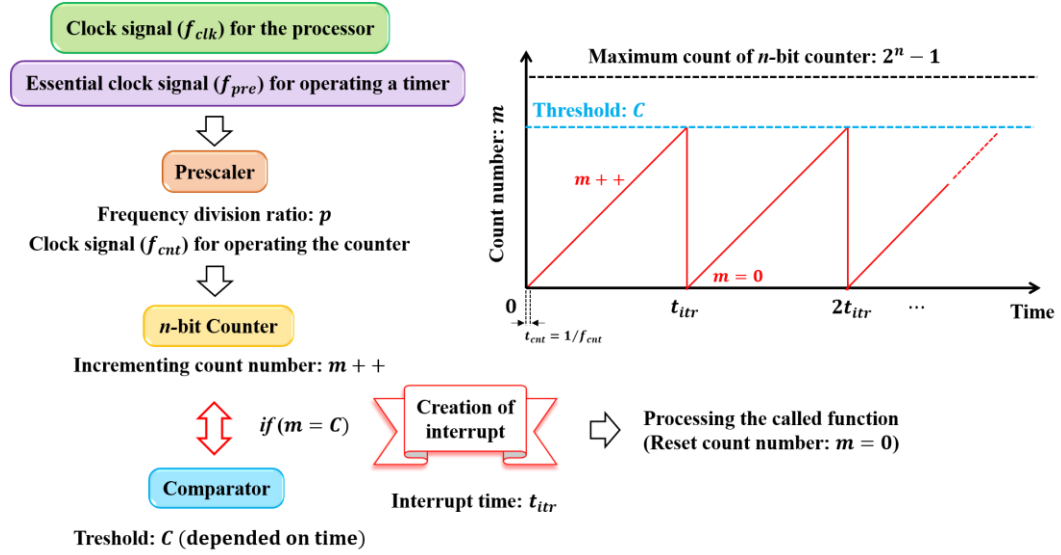


Figure 1. Concept of timer interrupt

Therefore, the frequency f_{cnt} of the clock signal to drive the counter is expressed as follows:

$$f_{cnt} = \frac{f_{pre}}{p} \quad (1)$$

Additionally, time t_{itr} to raise the interrupt is related to the threshold C as follows:

$$t_{itr} = (C + 1) \frac{1}{f_{cnt}} = (C + 1) \frac{p}{f_{pre}} \quad (2)$$

After processing tasks in function called when triggering the interrupt, the interrupt is resumed or continued if the count m is reset to 0.

2.2 M5Stack

M5Stack Basic ver. 2.7 has processor “ESP32-D0WDQ6-V3” of Espressif Systems ESP32 Series¹⁾, uses the structure “hw_timer_t” for timer management defined in the header file “esp32-hal-timer.h”. Table 1 shows the commonly-used functions closely related to the structure.

timerBegin() sets a timer number (0 to 3: 4 types configurable) to 1st argument, division ratio p

Function	Description
timerBegin()	Setting initial values (e.g., division ratio, countdown type [up or down])
timerAttachInterrupt()	Registering the function called when an interrupt is raised
timerAlarmWrite()	Setting time (t_{itr}) when an interrupt is raised
timerAlarmEnable()	Starting a timer
timerAlarmDisable()	Stopping a timer

Table 1. Typical functions for timer interrupt

to 2nd one and count mode (set count-up as “true”, count-down as “false”) to 3rd one. The return value is stored as a pointer to an “hw_timer_t” structure.

timerAttachInterrupt() sets the pointer of the structure as 1st argument, user function name called when raising an interrupt as 2nd one and interrupt method mode (“edge” / “level” mode inputs “true” / “false”) as 3rd one. The “edge” and the “level” modes mean that an interrupt is fired if $m = C$ and $m \geq C$, respectively.

timerAlarmWrite() assigns the same pointer to 1st argument, the threshold C to 2nd one and the logic that an interrupt is generated cyclically (“true”) or once (“false”) to 3rd one. **timerAlarmEnable()** / **timerAlarmDisable()** specifies the pointer, and starts/ stops its timer.

2.3 micro:bit

BBC micro:bit v2.21 with processor “nRF52833” of Nordic Semiconductor’s nRF52 series has five timers (TIMER0 ~ TIMER4 instances) as the peripheral elements and defines structures of NRF_TIMER series (NRF_TIMER0 ~ NRF_TIMER4) in the header file “nrf52833.h”.

Member Variables	Description
TASKS_START/STOP	Starting / Stopping a timer
MODE	Selecting an operating mode (“Timer” or “Counter”)
BITMODE	Setting a bit mode for counter (e.g., 8bit, 16bit, ...)
PRESCALER	Setting a division ratio for prescaler
CC [·]	Setting thresholds for comparator
INTENSET	Setting a type of interrupt (e.g., comparing count with CC [·])
SHORTS	Taking a shortcut between interrupt event and task (e.g., clear)

Table 2. Typical member variables for NRF_TIMER structure

Member variables for the

structures shown in Table 2 are normally used²⁾.

“TASKS_START / STOP” variable starts / stops a timer when its value is set to “1”. “MODE” one is mode for driving a timer and uses “TIMER_MODE_MODE_Timer” defined as an unsigned long type in the header file “nrf52833_bitfields.h”.

“BITMODE” specifies the bit size for a counter, adopts “TIMER_BITMODE_BITMODE_16bit” with “0UL” if used 16 bits counter and does “TIMER_BITMODE_BITMODE_08bit” with “1UL” if done 8bits one. Moreover, “PRESCALER” is used to set power value “ q ” for division ratio $p = 2^q$ $q \in [0, \dots, 9]$ of prescaler. Array CC[·] sets the threshold C of comparator, which means the number of counts m_c at interrupt time t_{itr} . The number of the elements in the array CC[·] is here 6 as $[0, \dots, 5]$, which means that the paper selects one of the six thresholds. Finally, “INTENSET” specifies a mode for comparing with comparator, “SHORTS” does one for resetting the count number of counter ($m = 0$) after raising a timer interrupt.

2.4 Arduino

Arduino Leonardo, which has processor “ATmega32U4” of Microchip Technology’s ATmega series, contains four timers: TIMER0, TIMER1, TIMER3 and TIMER4. These timers are respectively limited to 8bits, 16bits, 16bits and 10bits counters as available counters. For reference, Arduino UNO R3 with “ATmega328P” has three timers: TIMER0, TIMER1 and TIMER2, which can use 8bits, 16bits and 8bits counters, respectively.

Although parameters for registers depend on timers, the paper shows bit patterns of typical registers for “TIMER1 (16bits counter is enabled)” applied to the following examples in Figure 2³⁾.

“TCCR1A” and “TCCR1B” registers are used to specify how to interrupt. “WGM” one is set to “CTC (Clear Timer on Compare Match)” mode, which raises an interrupt when the counter (m_c) matches the threshold C (stored in “OCR1A” register, not shown in the figure) and clears the count number ($m = 0$) after the interrupt. “CS” one specifies binary information assigned according to division ratio p of prescaler.

“OCIE1A” in “TIMSK1” is the bit that enables the compare-match interrupt with the condition: $m_c = C$.

TCCR1A	7	6	5	4	3	2	1	0	CS12	CS11	CS10	Description
							WGM11	WGM10	0	0	0	-
TCCR1B	7	6	5	4	3	2	1	0	0	0	1	$f_{pre}/2^0$
				WGM13	WGM12	CS12	CS11	CS10	0	1	0	$f_{pre}/2^3$
									0	1	1	$f_{pre}/2^6$
TIMSK1	7	6	5	4	3	2	1	0	1	0	0	$f_{pre}/2^8$
			ICIE1				OCIE1B	OCIE1A	1	0	1	$f_{pre}/2^{10}$

Figure 2. Bit patterns of typical registers for TIMER1

3 First Example: count-down timer using Grove 4-Digit Display

First, this section describes a count-down timer using Seeed Studio’s Grove 4-Digit Display. This embedded system has the control capabilities that switch “counting” to “pause” and vice versa depending on a digital input using a tact switch with a pull-up resistor “10kΩ” externally connected to a development board.

3.1 M5Stack

<Hardware configuration>

Figure 3 is the hardware configuration. M5Stack is connected to the 4-Digit Display forming P2–DIO and P5–CLK lines besides Vcc terminals and GND ones, and is also done to the tact switch using P35 for digital input. An event triggered through the switch is handled as an interrupt in the implemented, later mentioned program.

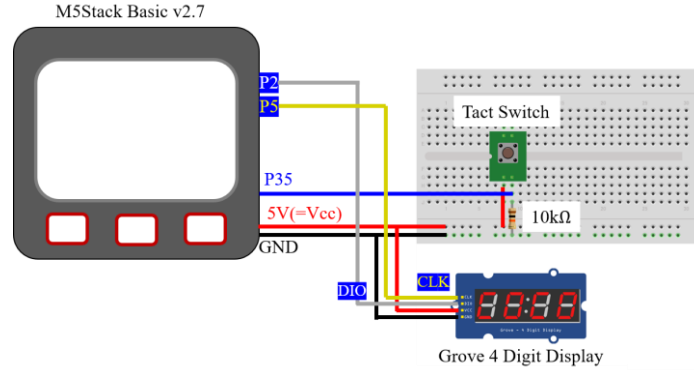


Figure 3. Outline of hardware (M5Stack)

<Program implementation>

Code 1 is the implemented program for the system. The program includes the header file “TM1637.h” because this Display has an IC named “TM1637”. This file is available after installing the library “Grove 4-Digit Display (by Seed Studio)” using “LIBRARY MANAGER” in “Arduino IDE”.

The parameters for raising an interrupt event once a second is set as shown in Table 3 using the peripheral clock signal with frequency $f_{pre} = 80\text{MHz}$. The timer number for timerBegin () function is set to “1”.

Information shown on the Display is provided from “minutes” and “second” variables that respectively store updated values depending on “minutes” and “seconds” as units in user function “onTimer” called at a timer interrupt event.

These variables are globally declared applying “volatile” keywords. The “volatile” keyword is a qualifier that helps notify the compiler to avoid optimizing the variables when compiling a program. In addition, this qualifier can prevent the loss of the updated information that is occasionally or often likely to arise when calling the variables from cache memories in cases of interrupted and multi-threaded processing.

Furthermore, the user function “onTimer ()” is essential to have `IRAM_ATTR` attribute. This is because storing the compiled function in a RAM instead of a flash memory allows it to call smoothly and quickly.

Variable name		Value
Defined on the paper	Defined on Code 1	
f_{pre}	peripheral_f	80MHz
p	division_ratio	80
f_{cnt}	counter_f	1MHz
C	counter_number	1×10^6

Table 3. Set parameters (M5Stack)

Information provided to each digit of the Display is specifically processed as follows in user function “arrange()” called from “onTimer()”: the upper 2 digits are assigned the quotient and the remainder when dividing the variable “minutes” by “10”, which are stored in “frt” and “snd” variables, respectively, while the lower ones are assigned them when using the same logic for “second”, which are done in “thd” and “fth”. The indicator module is thus updated using display () function of TM1637 class in user function “display ()” called from loop() function after storing these variables “frt”,

```

#include<M5Stack.h>
#include <TM1637.h>
#define DIO 2
#define CLK 5
#define settime 3
#define PIN 35

hw_timer_t *timer=NULL;
float peripheral_f=80e6;
float interval_time=1.0;
uint16_t division_ratio=80;
volatile SemaphoreHandle_t timerSemaphore;
portMUX_TYPE timerMux=portMUX_INITIALIZER_UNLOCKED;

TM1637 tm1637(CLK,DIO);
volatile int minutes=settime;
volatile unsigned int second=60;
volatile int frt,snd,thd,fth;
bool btn_flag=false;
int8_t TimeDisp[]={0x00,0x00,0x00,0x00};
void IRAM_ATTR onTimer(){
    if(btn_flag==false){
        portENTER_CRITICAL_ISR(&timerMux);
        if(second==0) second=60;
        countdown();
        arrange();
        portEXIT_CRITICAL_ISR(&timerMux);
    }
    xSemaphoreGiveFromISR(timerSemaphore,NULL);
}

void setup() {
    attachInterrupt(digitalPinToInterrupt(PIN),Statement,RISING);
    timerSemaphore=xSemaphoreCreateBinary();
    Serial.begin(115200);
    pinMode(PIN,INPUT);

    M5.begin();
    M5.Power.begin();
    M5.Lcd.setCursor(0,0);
    M5.Lcd.setTextSize(4);
    delay(100);
    tm1637.init();
    tm1637.set(BRIGHT_TYPICAL);
    tm1637.point(POINT_ON);

    arrange();
    int oo,ss,tt,ff;
    oo=frt;ss=snd;tt=thd;ff=fth;
    display(oo,ss, tt, ff);
    timer=timerBegin(1,division_ratio,true);
    float counter_f=peripheral_f/division_ratio;
    u_int64_t count_number=(u_int64_t)(counter_f*interval_time);
    timerAttachInterrupt(timer,&onTimer,true);
    timerAlarmWrite(timer,count_number,true);
    timerAlarmEnable(timer);
}

```

Code 1. M5Stack's program for count-down (first half)

competition).

The functions for the semaphore and the critical section used in the system are listed in Table 4.

Figure 4 shows one example scene.

“snd”, “thd” and “fth” in the array “TimeDisp[]”

The system implements control mechanisms called “semaphore” and “critical section” to access these four variables synchronously in loop () function. The former mechanism coordinates or manages access to a resource through multi-tasks/threads, while the latter one exclusively controls access to common resources.

These mechanisms thus enable the whole system to operate safely and efficiently even if a common resource is accessed simultaneously (that is, access

```

void loop() {
    if(xSemaphoreTake(timerSemaphore,0)==pdTRUE){
        if(btn_flag==false){
            M5.Lcd.clear();
            M5.Lcd.setCursor(0,0);
            M5.Lcd.setTextColor(GREEN);
            M5.Lcd.printf("CountDown");
            portENTER_CRITICAL(&timerMux);
            int oo,ss,tt,ff;
            oo=frt;ss=snd;tt=thd;ff=fth;
            portEXIT_CRITICAL(&timerMux);
            display(oo,ss, tt, ff);
        }
        else{
            M5.Lcd.clear();
            M5.Lcd.setCursor(0,0);
            M5.Lcd.setTextColor(RED);
            M5.Lcd.printf("Pause!");
        }
    }
}

void countdown(){
    if(second==60) minutes--;
    if(minutes<0) minutes=settime;
    else second--;
}

void arrange(){
    frt=minutes/10;
    snd=minutes%10;
    if(second<60){
        thd=second/10;
        fth=second%10;
    }
    else{
        thd=0;
        fth=0;
    }
}

void display(int oo,int ss, int tt, int ff){
    TimeDisp[0]=oo;
    TimeDisp[1]=ss;
    TimeDisp[2]=tt;
    TimeDisp[3]=ff;
    tm1637.display(TimeDisp);
}

void Statement(){
    btn_flag=!btn_flag;
}

```

Code 1. M5Stack's program for count-down (second half)

Function	Description
SemaphoreHandle_t	Preparing the queue handle for created semaphore
xSemaphoreCreateBinary()	Creating the semaphore for one user
xSemaphoreTake()	Acquiring a semaphore
xSemaphoreGiveFromISR()	Releasing a semaphore
portENTER_CRITICAL() / portEXIT_CRITICAL()	Disabling timer interrupt between these two functions (e.g., used in the loop () function)
portENTER_CRITICAL_ISR() / portEXIT_CRITICAL_ISR()	Disabling timer interrupt between these two functions (used in the function called when raising interrupt event)
portMUX_TYPE portMUX_INITIALIZER_UNLOCKED	Data type of exclusive control for one user The variable initialized to non-exclusive control state

Table 4. Introduced functions for Semaphore and Critical Section

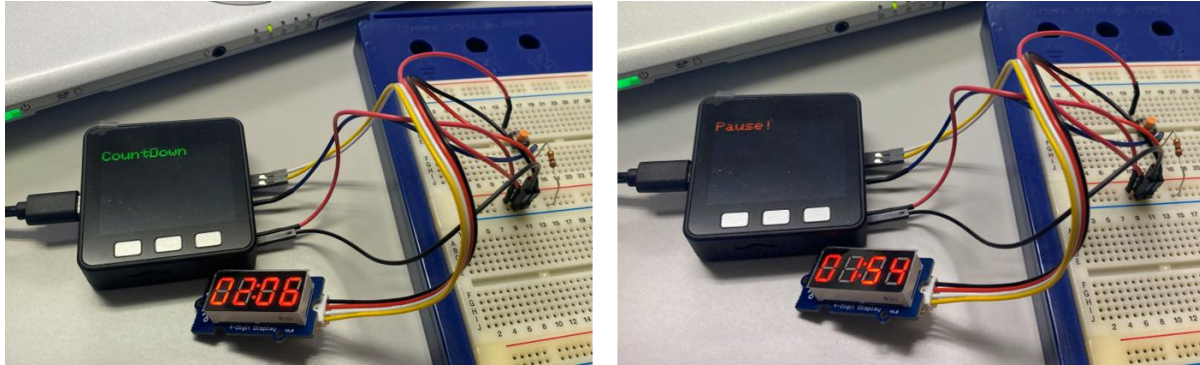


Figure 4. One scene (M5Stack; left: counting down, right: pausing)

3.2 Arduino

<Hardware configuration>

Hardware configuration of Arduino conforms to that of M5Stack as shown in Figure 5.

The Arduino is firmly connected to the Display on P12–DIO and P13–CLK lines and also done to the tact switch via the P2 which supports an external interrupt.

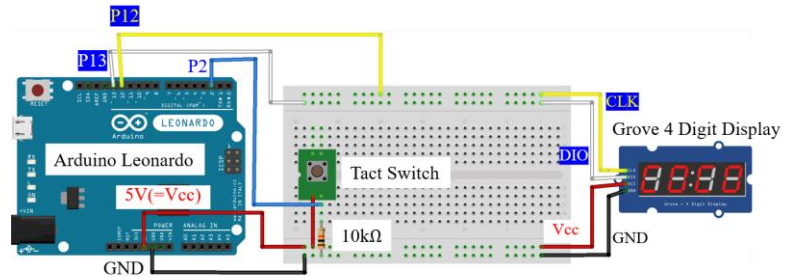


Figure 5. Outline of hardware (Arduino)

<Program implementation>

Code 2 is the program uploaded to the Arduino Leonardo's ATmega32U4. Table 5 lists the indispensable parameters to raise an interrupt once a second using the clock signal with the peripheral frequency $f_{pre}=16\text{MHz}$.

In order to generate the timer interrupt event with CTC mode, the system respectively sets the combination of the **WGM** registers and that of the **CS** ones as “(WGM13, WGM12, WGM11, WGM10) = (0,1,0,0)” and “(CS12, CS11, CS10) = (1,0,0)”. It also does the bit configurations for the **TCCR1A** and the **TCCR1B** registers as follows:

TCCR1A="0b00000000", TCCR1B="0b00001100"

That of the **TIMSK1** register for “compare match” is:

TIMSK1=0b00000010

where OCIE1A=1.

Variable name		Value
Defined on the paper	Defined on Code 2	
f_{pre}	fclk	16MHz
p	pre (=2 ^{bits})	256
f_{cnt}	—	62500Hz
OCR1A	val	62499

Table 5. Set parameters (Arduino)

```

#include <TM1637.h>
#define fclk (16e6)
#define bits 8
#define DIO 13
#define CLK 12
#define settime 3
#define PIN 2
float freq=1.0;

TM1637 tm1637(CLK,DIO);
int8 t TimeDisp[]={0x00,0x00,0x00,0x00};
volatile int minutes=settime;
volatile unsigned int second=60;
volatile int frt,snd,thd,fth;
volatile bool btn_flag=false;

void setup() {
  attachInterrupt(digitalPinToInterrupt(PIN),Statement,RISING);
  pinMode(PIN,INPUT);
  tm1637.init();
  tm1637.set(BRIGHT_TYPICAL);
  tm1637.point(POINT_ON);
  arrange();
  display();

  int pre=pow(2,bits);
  unsigned int val=(unsigned int)(fclk/(pre*freq))-1;

  TCCR1A=0b00000000;
  TCCR1B=0b00001100;
  OCR1A=val;
  TIMSK1=0b00000010;
}

```

Code 2. Arduino's program for count-down (first half)

```

ISR(TIMER1_COMPA_vect){
  if(btn_flag==false){
    if(second==0) second=60;
    countdown();
    arrange();
    display();
  }
}

void countdown(){
  if(second==60) minutes--;
  if(minutes<0) minutes=settime;
  else second--;
}

void arrange(){
  frt=minutes/10;
  snd=minutes%10;
  if(second<60){
    thd=second/10;
    fth=second%10;
  }
  else{
    thd=0;
    fth=0;
  }
}

void display(){
  TimeDisp[0]=frt;
  TimeDisp[1]=snd;
  TimeDisp[2]=thd;
  TimeDisp[3]=fth;
  tm1637.display(TimeDisp);
}

void Statement(){
  btn_flag=!btn_flag;
}

```

Code 2. Arduino's program for count-down (second half)

The function called when raising a timer interrupt event is defined as **ISR()** function (ISR: interrupt service routine) in the header file “interrupt.h”, whereas in M5Stack, it is done using the user-configured “onTimer” function. The variable “**TIMER1_COMPA_vect**”, defined in the header file “iom32u4.h” and specified in the argument of the **ISR()** function, is a mode for realizing an interrupt due to “compare match” of **TIMER1**. These two header files, however, do not explicitly need including for the reasons that they are automatically referred and recognized when specifying the board for Arduino Leonardo in Arduino IDE.

A series of processes related to the updated variables “frt”, “snd”, “thd” and “fth” with the volatile keyword to display on each digit of the Display for Arduino conform to those for M5Stack.

Figure 6 is, thus, one of the examples for the Arduino.

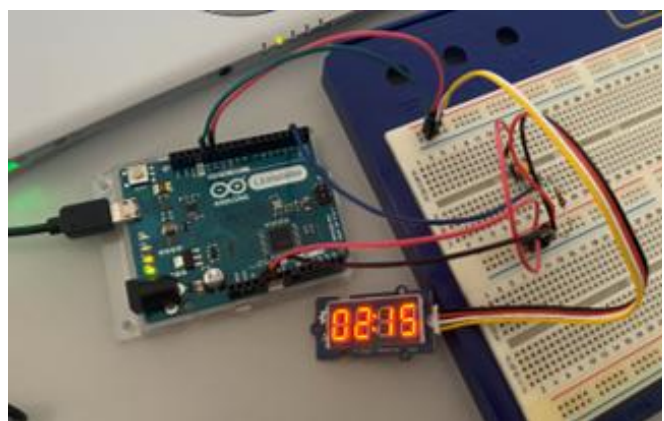


Figure 6. One scene (Arduino)

On the other hand, the paper omits discussion of micro:bit's system because the authors provide the similar system using **TIMER2** in Journal of technical reports in Advanced Technology Institute, Yamaguchi University, No.5 (2025): “Technical Introduction to Embedded System with BBC micro:bit” (page 14).

4 Next Example: creating musical scales and chords

Next, the paper discusses mechanism for creating chords by superimposing or harmonizing of the musical scales synchronously. Music released from speakers adopts the piano version of Johann Pachelbel's "Canon in D"⁴⁾, which includes a lot of situations (occasions) to strike respectively piano keys of low frequency band with player's left-hand fingers and those of high frequency band with his/her right-hand fingers, to create chords. The speaker externally connected to a development board is SPL (HONG KONG) LIMITED's Mylar Speaker "MSI28-12R"⁵⁾ or External Drive Piezo Transducer "SPT08". It is also acceptable to use the built-in speaker on a development board: 1W-0928 of M5Stack Basic V2.7, or MLT-8530 from Jiangsu Huaneng Elec. of micro:bit v2.21.

4.1 Musical scales⁶⁾

The frequency f_X of musical scale signal X in the k th octave area is represented as follows using concert pitch "A4" with f_{A4} (=440Hz) and positional difference d of it and other scale:

$$f_X = f_{A4} \times 2^{\left[\frac{(k-4)+d}{12}\right]} \quad (3)$$

Calculating the frequencies of musical scales in the 4th octave area based on the difference d of two scales, Table 6 is completed. Those of ones in other octave area can also be sequentially derived using combination (k, d) of the variables. However, they can be efficiently and smoothly done using bit shift operators in the point of view of program implementation for the reason that they depend on power of 2 and relate to bit rules for binary.

scale X	scale difference d	f_X [Hz]
C4	-9	262
C4#	-8	277
D4	-7	294
D4#	-6	311
E4	-5	330
F4	-4	349
F4#	-3	370
G4	-2	392
G4#	-1	415
A4	0	440
A4#	1	466
B4	2	494

Table 6. musical scales in the 4th octave area

The effect of the left shift operator "<<" leads f_X to " $2^n f_X$ " as a result, whereas that of the right one ">>" does it to " $f_X/2^n$ ", where n is the number of bits to shift. Hence, the f_X for the former operator fulfills the role of "multiplicand", whereas it for the latter one does that of "dividend". For instance, $f_{A5} = f_{A4} \ll 1$ and $f_{A2} = f_{A4} \gg 2$.

The tune of the adopted Pachelbel's Canon requires the musical scales of 2nd ~ 6th octave areas.

4.2 Creating mechanism for musical scales / chords

The signal provided to the speaker is in PWM (Pulse Width Modulation) form, which can precisely control motor and LED modules. Specifying available pins for digital output and calling the dedicated or customized function in accordance with development boards leads to creating PWM signals. Specifically, Arduino and micro:bit use analogWrite() function which specifies the parameters such as one pin and pulse width (duty ratio) of signal, and M5Stack effectively introduces useful functions of "ledc" series.

PWM signal created in this example modulates pulse width of a carrier signal, a rectangular wave with an inaudible frequency and 50% duty ratio, with musical scales.

Figure 7 is an outline of the mechanism that reflects the musical scales to the PWM signal. Creating the PWM signal using the modulation wave as the j th musical scale $X_{j,k}$ where $j \in \{1, \dots, 12\}$, $k \in \{2, \dots, 6\}$ of twelve-tone equal temperament in the k th octave area, the updated value $S_{j,k} \in \mathbb{N}$ and the added one $S_{j,k}^{\text{add}} \in \mathbb{N}$ depending on the frequency $f_{j,k}$ of the scale $X_{j,k}$ are related as follows:

$$S_{j,k}^{\text{new}} = S_{j,k}^{\text{old}} + S_{j,k}^{\text{add}}, \quad S_{j,k}^{\text{add}} = I_{\text{max}} \frac{f_{j,k}}{f_{\text{cr}}} \quad (4)$$

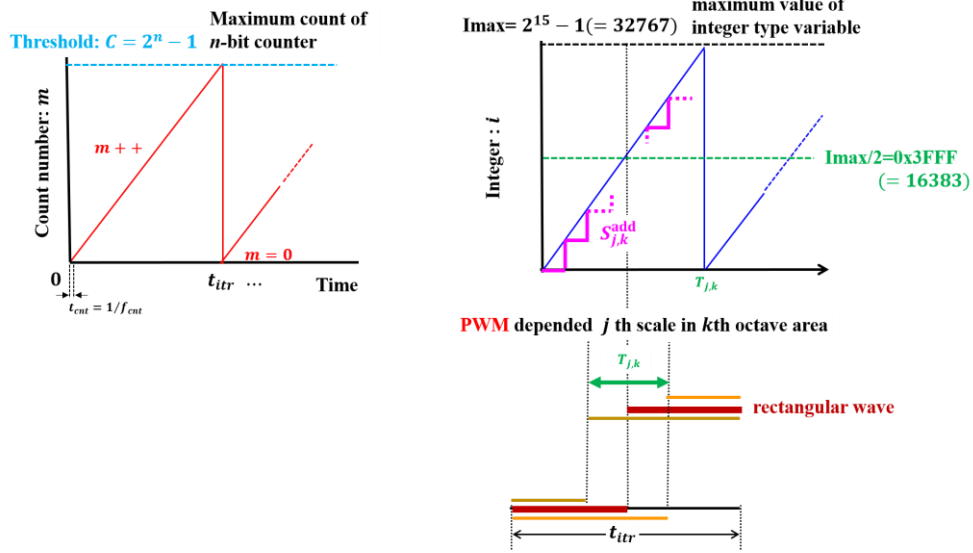


Figure 7. Creating mechanism for musical scale

f_{crr} is the frequency of the carrier signal and the subscripts “old” and “new” of the updated value $S_{j,k}$ mean pre-update and post-update, respectively. $S_{j,k}$ is updated at timer interrupt event, which is triggered by the overflow of count number m of n -bit counter. This occurs when the threshold C for comparator is set to the possible maximum ($2^n - 1$) for the counter. The I_{max} is the possible maximum of an integer variable (int data type), which is $2^{15} - 1 = 32,767$ in Arduino IDE. The scale signal $X_{j,k}$ is created when $S_{j,k}$ additionally updated by $S_{j,k}^{\text{add}}$ every timer interrupt event reaches or exceeds the I_{max} (i.e., overflow event). The system controls to extend the pulse width (i.e., enrich the duty ratio) of the carrier wave if $S_{j,k} \geq \frac{I_{\text{max}}}{2}$, whereas to shorten that (reduce it) of this wave if $S_{j,k} < \frac{I_{\text{max}}}{2}$. The paper, for convenience, defines the variation of the pulse width under this situation as $\Delta W_{j,k}$.

Conforming the time t_{itr} to the period $t_{\text{crr}} (= 1/f_{\text{crr}})$ of the carrier wave, the number of additions $N_{j,k}^{\text{over}}$ of $S_{j,k}^{\text{add}}$ (in other words, that of the interrupts) when the updated value $S_{j,k}$ reaches an overflow can be represented as follows:

$$\begin{aligned} N_{j,k}^{\text{over}} &= (\text{int}) \frac{f_{\text{crr}}}{f_{j,k}} = \frac{1}{t_{\text{itr}} f_{j,k}} \\ \therefore N_{j,k}^{\text{over}} t_{\text{itr}} &= \frac{1}{f_{j,k}} = T_{j,k} \end{aligned} \quad (5)$$

It will be caught that the difference of the scales $\mathbf{X}_k = (X_{1,k}, \dots, X_{12,k})^T$ depends on that of $\mathbf{N}_k^{\text{over}} = (N_{1,k}^{\text{over}}, \dots, N_{12,k}^{\text{over}})^T$ $k \in \mathbb{N}, (k \neq 0)$, where $k \in \{2, \dots, 6\}$ in the system. The effect of “(int)” means to cast the calculated result to the integer value.

Therefore, synchronously superimposing (or harmonizing) the variations of the pulse width based on multiple musical scales results in a chord.

$$\Delta W^{\text{chord}} = \sum_j \Delta W_{j,k} \quad (6)$$

4.3 M5Stack

<Hardware configuration>

Figure 8 shows a hardware configuration for performing the music harmoniously. M5Stack is, through P16 and P17, respectively connected to the speakers that release musical scales or chords of low and high frequency areas performed with left and right-hand fingers. The paper positions the former / the latter speaker as “left / right-hand fingers side”. The

performance for the music starts when click event of the M5Stack's button assigned to P39 is triggered.

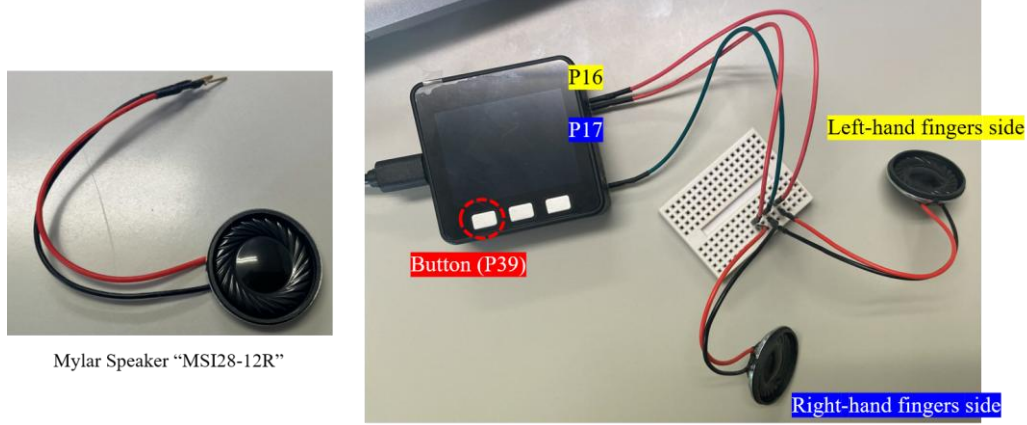


Figure 8. Hardware configuration for musical performance (M5Stack)

<Program implementation>

Having the voluminously and complexly bloated codes, the program installed on M5Stack is concisely explained with pseudo representation in this paper.

Function	Description
ledcSetup ()	Setting initial values (channels, frequency for carrier wave, bits)
ledcAttachPin ()	Linking a channel with a specified pin
ledcWrite ()	Releasing PWM signal with a specified duty ratio

Table 7. Typical functions for creating PWM

M5Stack utilizes the functions of “ledc” series shown in Table 7 to create a PWM signal.

ledcSetup() specifies a channel to 1st argument. The number of channels is equal to that of used speakers. This function sets the frequency of a carrier signal for PWM to 2nd argument and the number of bits related to duty ratio to 3rd one. **ledcAttachPin()** specifies pin number for signal output to 1st one, channel number linked with pin to 2nd one. PWM signal is created using **ledcWrite()** that sets the channel to 1st and duty ratio of it to 2nd.

The program of Code 3 includes pseudo representations written in italics. Table 8 means that the variables declared in the code defined correspond to ones defined for timer interrupt and PWM and also store the specified and calculated values.

Assigning index “m” for variable “M” of “left / right-hand fingers side” to “0 / 1”, the combination (PWMPin[m],CH[m]) of the pin and the channel numbers specified in the ledc’s functions are (16,0) and (17,1) [see setup() function].

Beat is a temporal length to strike the key of a musical scale. The tune of the Pachelbel’s Canon has the five temporal lengths that are whole, half, quarter, eighth and sixteenth notes of a beat. The program stores these values in array variable “t_scale[]”. The scale’s frequencies of 2nd ~ 6th octave areas calculated using the equation (3) and bit shift operators are stored in two-dimensional array “freq[][]”. However, the necessary

Variable name		Value
Defined on the paper	Defined on Code 3	
f_{pre}	peripheral_f	80MHz
p	division_ratio	5
f_{cnt}	—	16MHz
C	counter_number	256
f_{crr}	fcrr	62,500Hz
f_{A4}	fA4	440Hz

Variable name		State
Defined on the paper	Defined on Code 3	
$S_{j,k}$	operator_ori [][]	updated once a timer interrupt
$S_{j,k}^{add}$	operator_add [][]	depending on eq. (4) and a scale
ΔW^{chord}	output []	updated once a timer interrupt

Table 8. Correspondence between two variables (M5Stack)

```

#include<M5Stack.h>
#define PWMbase 50
#define btnPIN 39
#define bits 8
#define M 2
#define N 5
#define Mscale_num 12
#define octave_low 2
#define octave_high 6
#define octave_len (octave_high-octave_low+1)
#define BPM 74

int PWMPin[M],CH[M];
int fcrr;
hw_timer_t *timer=NULL;
float peripheral_f=80e6;
uint16_t division_ratio=5;
uint16_t reso;
int **freq;
float fA4=440.0;
int Min_d=-9;
unsigned int beat=(unsigned)(1000*60/BPM);
unsigned int *t_scale;

int N1,N2;
int delay_time;
int **F,**Vol;
volatile int **operator_ori,**operator_add;
int B1_f,A1_f;
int Volume1=63;
int Volume2=63;
bool flag=false;

void IRAM_ATTR onTimer(){
    int output[M];
    for(int m=0;m<M;m++) output[m]=0;
    int scale_number;
    for(int m=0;m<M;m++){
        if(m==0) scale_number=N1;
        else scale_number=N2;
        for(int i=0;i<scale_number;i++){
            operator_ori[m][i] += operator_add[m][i];
            operator_ori[m][i] &= 0x7FFF;
        }
    }
    for(int m=0;m<M;m++){
        if(m==0) scale_number=N1;
        else scale_number=N2;
        for(int i=0;i<scale_number;i++){
            if(operator_ori[m][i] < 0x3FFF) output[m] += -Vol[m][i];
            else output[m] += Vol[m][i];
        }
    }
    for(int m=0;m<M;m++){
        if(output[m]>0) output[m] >>= 1;
        else output[m]=-( (-output[m]) >> 1);
    }
    for(int m=0;m<M;m++) ledcWrite(CH[m],reso+output[m]);
}

```

Code 3. M5Stack's Program for musical performance
(Global variables, function called at timer interrupt event)

scales extracted from its array as the tune progresses are temporarily stored in two-dimensional array “F[m] [.]” and used to calculate the additional values “operator_add[m][.]”

The “onTimer()” function with IRAM_ATTR attribute

```

void setup() {
    Storing the pin information for connecting to speakers in array
    “PWMPin[]={16,17}”.
    Storing the channels in array “CH[]={0,1}”.
    Setting 2bits - 1 to variable “val”.
    Converting duty ratio to bits using map() function → Stored in variable “reso”.
    Calculating fcrr using peripheral_f, division_ratio and val.
    setup1();
    setup2();
    for(int m=0;m<M;m++){
        ledcSetup(CH[m],fcrr,bits);
        ledcAttachPin(PWMPin[m],CH[m]);
        ledcWrite(CH[m],reso);
    }
    timer=timerBegin(1,division_ratio,true);
    timerAttachInterrupt(timer,&onTimer,true);
    uint64_t count_number=val+1;
    timerAlarmWrite(timer,count_number,true);
}

void setup1(){
    Preparing array “freq[][]” to store the frequencies of the musical scales in the
    2nd to 6th octave area using eq. (3) and bit shift operators("<<" and ">>").
}

void setup2(){
    Storing values of beat length to play musical scales in “t_scale[]”.
    Preparing array “F[][]” to store the frequencies of the scales required to match
    the rhythm of the score.
    Storing Volumes to provide to the speakers in array “Vol[][]”.
    Preparing array “operator_ori[][]” to store Sj,k
    Preparing array “operator_add[][]” to store Sj,kadd
}

```

Code 3. M5Stack's Program for musical performance
[setup () functions: includes pseudocode]

```

void sound(){
    int nmax;
    for(int m=0;m<M;m++){
        if(m==0) nmax=N1;
        else nmax=N2;
        for(int n=0;n<nmax;n++){
            operator_add[m][n]=(int)(32767*F[m][n]/fsamp);
        }
        timerAlarmEnable(timer);
        delay(delay_time);
        timerAlarmDisable(timer);
    }
}

void loop() {
    int bval=digitalRead(btnPIN);
    if(bval==0) flag=true;
    if(flag==true){
        Introduction();
        Chorus();
        Center();
        ComingDown();
        Ending();
        exit(0);
    }
}

```

Code 3. M5Stack's Program for musical performance
[sound () and loop () functions]

called at a timer interrupt makes it possible to create scales or chords. The “operator_ori[m][i]” for a scale “i” of “m”-hand fingers side is masked by the value “0x7FFF” using the bitwise AND operator “&” after additionally updated with “operator_add[m][i]”. Although the temporal transition of the scale “i” signal as a modulation wave is formed by decreasing the variation of the pulse width “output[m]” if “operator_ori[m][i]” < ”0x3FFF” and increasing that of it otherwise, its periodic or cyclical features are achieved through the reset operation that operator_ori[m][i] is set to “0” if more than ”0x7FFF”: the result due to the “overflow” of this array variable is derived from the above masking.

The system creates chords as follows: first, it synchronously superimposes the variations of the pulse width based on the updated variables “operator_ori[m][.]”, then stores them in array “output[.]”, and finally provides them to the ledcWrite() function. “N1” and “N2” variables are the number of scales to superimpose on left and right-hand fingers sides, respectively. It should be noted that the output signal is one scale if “N1=1” or “N2=1”.

The musical performance is based on the following procedure: 1) setting the number of the musical scales “N1” “and” “N2” for left and right-hand fingers sides, 2) selecting a durational time (stored in “delay_time”) that releases the scales or chords, 3) specifying the frequencies “F[m][.]” of the released scales, 4) calling the sound() function to calculate “operator_add[.]” and start / stop a timer [timerAlarm-Enable () / Disable () function], Furthermore, 5) sequentially calling user functions “Introduction”, “Chorus”, ..., and “Ending” due to the music score in the loop() function in order to move the performance for the music forward (the total performance time is approximately 6 minutes).

Figure 9, for reference, shows a bar of the music. This bar includes the chords that the maximum number of the scales (N1, N2) are (2,3) on left and right sides. The performance is executed applying individually and sequentially the above procedure 1)~ 4) to the parts (epoch) which are temporally or synchronously segmented the bar by the musical scale that has the shortest released time in left and right-hand fingers sides. The figure is composed of six epochs. The program regards rest part (● in the left illustration) as keeping the state like a silence using F[m][.] = 0.

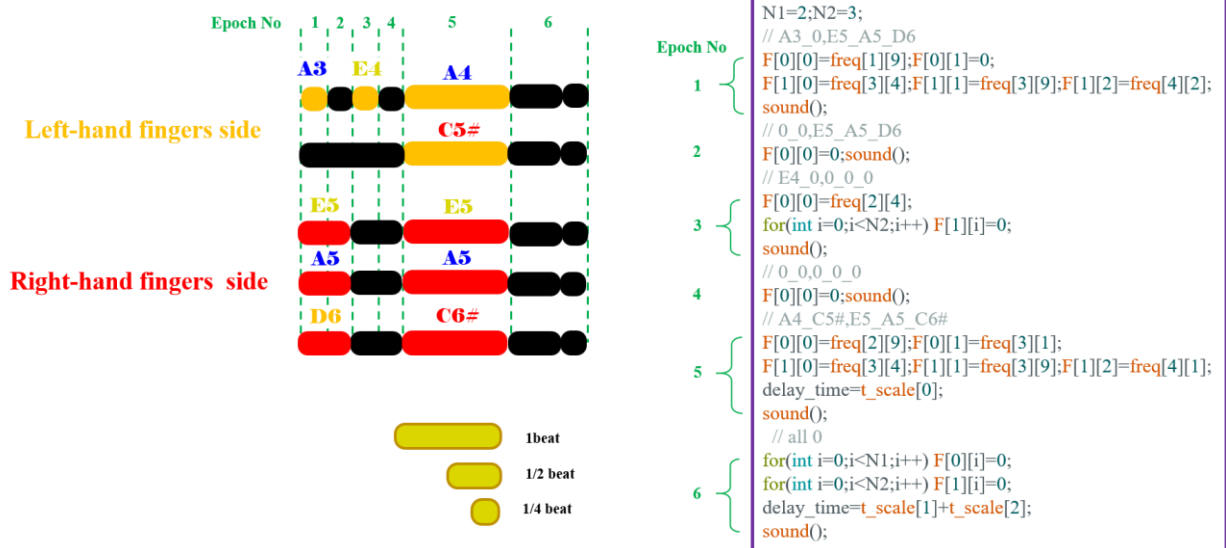


Figure 9. a performance example for musical scales or chords: a musical bar

4.4 micro:bit

<Hardware configuration>

Hardware configuration for micro:bit is shown in Figure 10. The micro:bit links to the speaker of left-hand fingers side

over P0, while it does to the one of right side over P1. The tune starts after pushing the button A (assigned to P5) of the micro:bit.

<Program implementation>

The program installed on the micro:bit is presented in Code 4, but most of the codes are represented in the pseudocode. Since the implemented algorithm for the micro:bit conforms to that of M5Stack, this section discusses the elements that are different from M5Stack, such as generating timer interrupt and PWM.

The correspondence between the parameters defined for timer interrupt and PWM and those declared in Code 4 is summarized in Table 9.

The micro:bit utilizes two NRF_TIMER structures, that is, “NRF_TIMER1” using TIMER1 for left-hand fingers side and “NRF_TIMER2” using TIMER2 for right

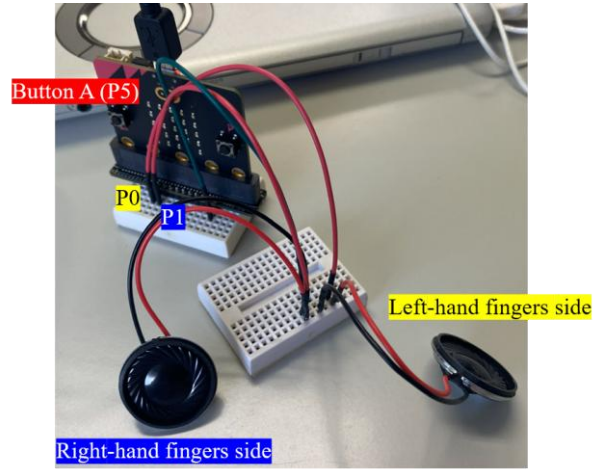


Figure 10. Hardware configuration for musical performance (micro:bit)

Variable name		Value
Defined on the paper	Defined on Code 4	
f_{pre}	fcik	16MHz
p	$2^{NRF_TIMER1/2 \rightarrow PRESCALER}$	$2^0 = 1$
f_{cnt}	—	16MHz
C	$NRF_TIMER1/2 \rightarrow CC [0]$	256
f_{crr}	fcrr	62,500Hz
f_{A4}	fA4	440Hz

Table 9. Correspondence between two variables (micro:bit)

```
#include <stdlib.h>
#include "nrf.h"
#define IPIN 0
#define rPIN 1
#define PWMbase 50
#define bits 255
#define fcik (16e6)
:
#define btn 5
int fcrr;
uint16_t val;
int prescaler=0;
int **freq;
float fA4=440.0;
:

extern "C" void TIMER1_IRQHandler(void){
    NRF_TIMER1->EVENTS_COMPARE[0] = 0;

    Update operator_ori[0][:] and variable output

    analogWrite(IPIN,val+output);
}

extern "C" void TIMER2_IRQHandler(void){
    NRF_TIMER2->EVENTS_COMPARE[0] = 0;

    Update operator_ori[1][:] and variable output

    analogWrite(rPIN,val+output);
}
```

Code 4. Micro:bit’s program for musical performance (Global variables and function called at timer interrupt)

```
void sound(){
    Calculating operator_add[m][:]
    NRF_TIMER1->TASKS_START=1;
    NRF_TIMER2->TASKS_START=1;
    delay(delay_time);
    NRF_TIMER1->TASKS_STOP=1;
    NRF_TIMER2->TASKS_STOP=1;
}
```

Code 4. Micro:bit’s program for musical performance [sound () function]

side. This design was adopted for the following reasons:

1) the micro:bit cannot utilize the channels unlike M5Stack, 2) one NRF_TIMER structure would not have enough time to create all scales (or chords) of both sides within one interruption. The 2) means that the tasks in the function called when raising a timer interrupt event cannot be completed before the next one, which has been confirmed as loss or break of the musical performance by the author.

For the micro:bit, the functions called when a timer interrupt event is triggered are the handlers named “TIMER1_IRQHandler” and “TIMER2_IRQHandler”,

```

void setup() {
    pinMode(IPIN,OUTPUT);
    pinMode(rPIN,OUTPUT);
    val=(uint16_t)map(PWMbase,0,100,0,bits);
    fcr=(int)(fclk/(pow(2,prescaler)*(bits+1)));

    analogWrite(IPIN,val);
    analogWrite(rPIN,val);

    Using NRF_TIMER1 for left performance
    NRF_TIMER1->TASKS_STOP=1;
    NRF_TIMER1->TASKS_CLEAR=1;
    NRF_TIMER1->MODE=TIMER_MODE_Timer;
    NRF_TIMER1->PRESCALER=prescaler;
    NRF_TIMER1->BITMODE=TIMER_BITMODE_BITMODE_08Bit;
    NRF_TIMER1->CC[0]=bits+1;

    NRF_TIMER1->INTENSET = (TIMER_INTENSET_COMPARE0_Enabled << TIMER_INTENSET_COMPARE0_Pos);
    NRF_TIMER1->SHORTS = (TIMER_SHORTS_COMPARE0_CLEAR_Enabled << TIMER_SHORTS_COMPARE0_CLEAR_Pos);
    NVIC_SetPriority(TIMER1_IRQn, 3);
    NVIC_ClearPendingIRQ(TIMER1_IRQn);
    NVIC_EnableIRQ(TIMER1_IRQn);

    Using NRF_TIMER2 for left performance
    The parameters set in variables of NRF_TIMER2 structure conform to those of NRF_TIMER1 one.
    :
}

```

Code 4. Micro:bit's Program for musical performance [setup () function]

which are defined in the startup routine file “gcc_startup_nrf52833.S” included in the header file “nrf52833.h” via “nrf.h”. Moreover, these handlers are modified using **extern “C”** linkage so that they can be processed with the rule of C language.

The system prepares two carrier waves with 50 % duty ratio and $f_{\text{cr}}=62,500\text{Hz}$ for left and right sides in setup () function, and creates the musical scales (or chords) as modulation waves in the two handlers. Each PWM output is thus realized using **analogWrite ()** function.

Setting 1 (“= true”) to the member variables “TASKS_START” and “TASKS_STOP” of the structure starts and stops the timer interrupt, the procedures for which are performed in sound () function.

Although the chapter shows the cases of M5Stack and micro:bit, the Arduino system can also be realized by applying a comparable algorithm and the techniques for timer interrupt [registers and ISR () function] introduced in 2.4 section.

5 Final Example: “one-to-one” wireless data communication using XBee modules

Finally, the paper focuses on a data transmission system using Digi International Inc.’s wireless communication modules “XBee”, in a one-to-one communication configuration that exchanges temperature and humidity information measured periodically using DHT20 temperature and humidity sensors manufactured by Guangzhou Aosong Electronic Co., Ltd.

The programming skills for this communication system, from a broad perspective, have the following noteworthy points: 1) transmitting data once every specified time interval (e.g., 1 second) using the mechanism of timer interrupt, 2) adapting to I2C communication used to get temperature and humidity information from a DHT20 sensor, 3) Building a multi-threading state that processes procedures for data transmission and reception as mutually exclusive tasks on different (two separate) threads.

Having not been able to confirm a safe and reliable method to build a multi-threading state in the implementation of

program for micro:bit on the development environment “Arduino IDE”, the paper applies the single-threading system , which runs sequentially the tasks for transmitting and receiving, to it.

5.1 XBee wireless communication modules

The appearance of the wireless communication module “XBee” which emits electromagnetic waves in the 2.4 GHz band is shown in Figure 11⁷⁾. “XBee” module has a serial communication system that sequentially and wirelessly propagates data one bit at a time. It additionally adopts an asynchronous protocol called “UART (Universal Asynchronous Receiver/Transmitter)”, which transmits and receives data asynchronously without a clock signal (TX: Transmit; RX: Receive).

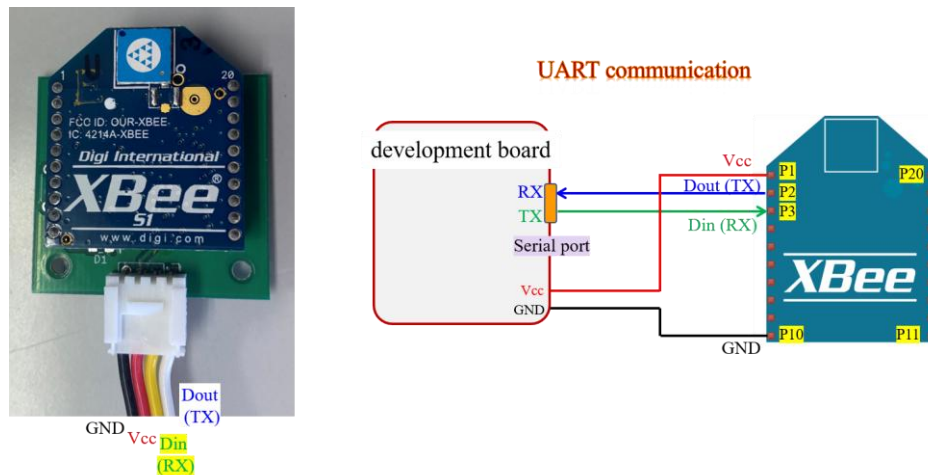


Figure 11. XBee module

Although Din (RX) and Dout (TX) provided from XBee form signal lines with TX and RX pins used for the serial port on development board, respectively, these lines can be built easily and quickly by using an expanded board for enhanced functionality with a Grove connector shown on the left of the figure. Especially, Arduino Leonardo connects RX’s P0 and TX’s P1 to Dout and Din of XBee using XBee shield for Arduino (Zhiwei Robotics Corp’s DFR0015), respectively.

Serial communication uses the instance “[Serial](#)” of [HardwareSerial](#) class defined in the header file “[HardwareSerial.h](#)” for each board. Table 10 shows the number of instances that can be generated from the class, and their functions used in the example. It should be noted that the number of available instances and parameters to specify in the `begin ()` function depend on a board.

The `begin ()` function initializes the instance, specifies the baud rate to transfer data; provided, however, that M5Stack also allows the user to decide whether to add a parity bit and to specify the pin numbers for transmission and reception. By appending the code “`#define ARDUINO_GENERIC`” to the header file “`Uart.h`”, BBC micro:bit can specify its pin numbers using the `setPins()` function. Furthermore, Arduino cannot explicitly set them and does not need to set them because they are fixed to (RX, TX) = (0, 1) as mentioned above.

	M5Stack	micro:bit	Arduino (Leonardo)
instances	Serial, Serial0, Serial1, Serial2	Serial	Serial1
functions	begin (BAUD, parity, RX, TX)	setPins(RX,TX) and begin(BAUD)	begin (BAUD)
	write(str)		
	available ()		
	readStringUntil(char)		

Table 10. Functions with Serial instances

In the functions with common arguments and parameters regardless of board type, the `write ()` function can transmit string data to the established destination, and `readStringUntil()` reads string data from a buffer until a terminator character (stored in variable “char”) specified in its argument, which is not included in its return value. `Available ()` can be used to check the number of the received data bytes, which indicates that data has been received if “available (>0)” is satisfied.

5.2 Temperature & Humidity Sensor (DHT20)⁸⁾

Figure 12 shows that the DHT20 sensor has four pins: Vcc, GND, SDA (Serial Data) and SCL (Serial Clock), and the left side of the figure presents that it has a Grove connector to bundle these pins.

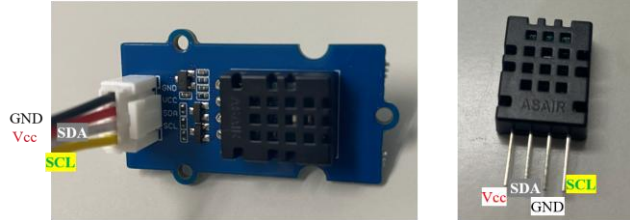


Figure 12. Temperature & Humidity Sensor (DHT20) (left: Grove type)

Temperature and humidity information measured using the sensor are provided to a development board through I2C (Inter-Integrated Circuit) communication system, which complies with a clock-synchronized protocol under the condition that the roles of controller and device are divided between nodes (case of this example: the development board and the sensor). The former role corresponds to “master” and the latter to “slave”. This example follows the steps that the sensor (slave side) transmits temperature and humidity values to the board (master one) on the SDA line when requested through the SCL line to provide data

However, receiving data requires the following process: 1) the master transmits command signal “0x71” to the slave, provided that the upper seven bits of the signal “0x71(=’0b01110001’)” represents the I2C address “0x38” assigned uniquely to the sensor and the lower one bit indicates a mode for reading data, 2) after waiting 10ms, the master transmits sequentially trigger signal “0xAC” and its parameters “0x33” and ”0x00” to the slave, which controls timing to start the measurement, 3) the system is waited for 80ms.

From the above procedure, the master - slave communication line is established, and the master can demand data from the slave. In fact, data, as shown in Figure 13, is obtained as a total of 7 bytes of information, which sequentially concatenates 1 byte for state (status), 2.5 bytes for humidity, 2.5 bytes for temperature and 1 byte of CRC (Cyclic Redundancy Check).

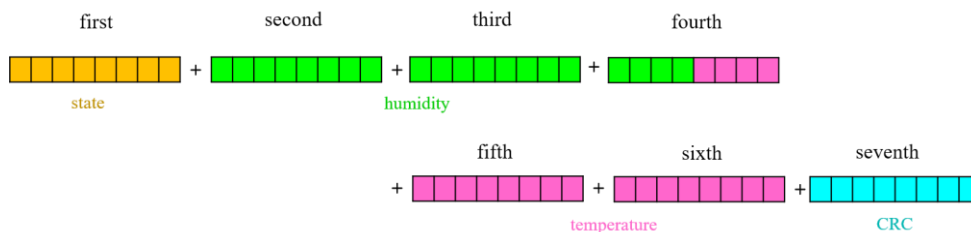
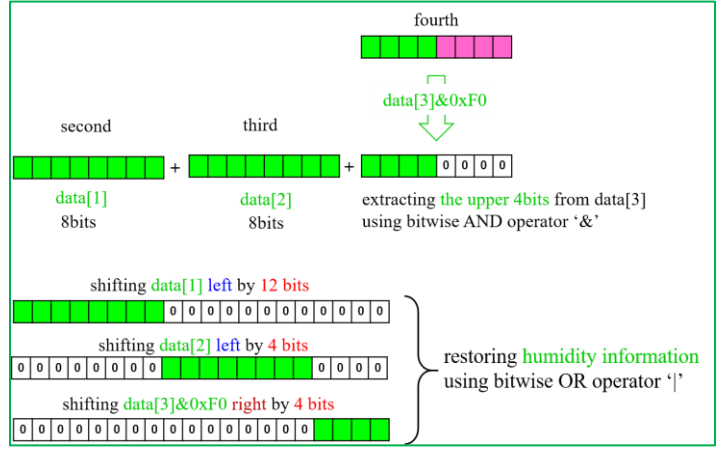


Figure 13. Received data configuration

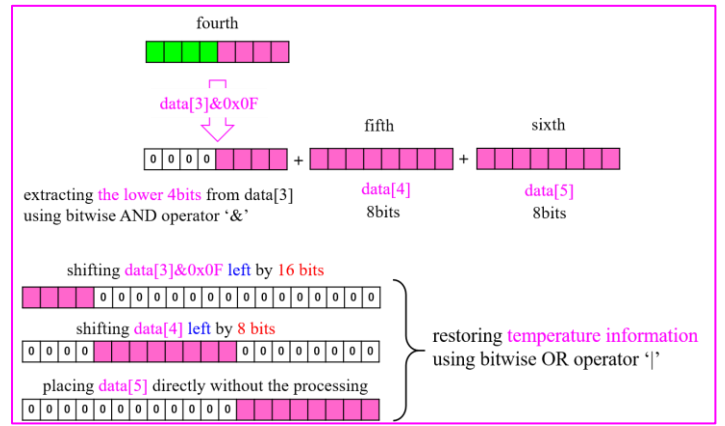
Temperature and humidity information must be restored as 20-bit data though they are obtained as multiple bytes of data (see Figure 14).

Assuming that the seven bytes of information are sequentially stored in each element [index: 0, ..., 6] of array variable “data []” once one byte, the humidity information is restored according to the following procedure [the figure (a)]:

1. First, operating to bit-shift the second and the third byte data (“data [1]” and “data [2]”) to the left direction by 12 bits and 4 bits, respectively, the operated results, for convenience, are represented by “ α ” and “ β ”.
2. Next, extracting the upper 4 bits of the fourth data (data [3]) through performing bitwise AND (“&” operator) with “0xF0”.
3. Then, bit-shifting the extracted data to the right direction by 4bits, whose result is denoted as “ γ ”.
4. Finally, performing bitwise OR (“|” operator) of the three result: “ α ”, “ β ” and “ γ ”.



(a) Humidity



(b) Temperature

Figure 14. Data restoration

The temperature information, shown in the figure (b), also is restored conforming to the humidity one as follows: 1) operating to shift the “data [3] & 0x0F”, the lower 4 bits of the fourth data which extracted by “0x0F” using bitwise AND operator, to the left by 16 bits. 2) doing to shift the fifth data “data [4]” to the left by 8 bits. 3) summing logically the sixth data “data [5]” and the two results [1) and 2)].

In the implemented program, tasks as shown in Code 5(a) leads the temperature and humidity information to restore. However, it is necessary to convert their information to the values with units of “[Celsius]” and “[%]”, which depends on the following relations. In addition, the same cord (b) shows converting them.

$$H = \frac{H_{res}}{2^{20}} \times 100, \quad T = \frac{T_{res}}{2^{20}} \times 200 - 50 \quad (7)$$

H_{res} and T_{res} are respectively the restored values for humidity and temperature, and also the exponent part “20” for a power of 2 is synonymous with the whole length of the restored information. Dividing the restored value by divisor 2^{20} is equivalent to operating to shift it to the right direction by 20 bits, but the double type temperature value is processed adopting a hybrid approach, where it is divided after bit-shifting.

Turning to the seventh byte data “data[6]”, CRC, an error-detecting code to determine whether data was correctly received or not, utilizes the remainder value after dividing transmitting signal by a certain regularized value. The evaluation indicator, which is positioned as the “success” of receiving data, is that the data [6] matches the calculated remainder (variable “crc”) using the other six bytes of data (data [0]~data [5]).

```
uint32_t h1,h2,h3,t1,t2,t3;
h1 = data[1];h2 = data[2];h3 = data[3] & 0xF0 ;
uint32_t Hu = (h1 << 12) | (h2 << 4) | (h3 >> 4);
t1 = data[3] & 0x0F;t2 = data[4];t3 = data[5];
uint32_t Te = (t1 << 16) | (t2 << 8) | t3;
```

(a) Restoration

```
int Humi = (Hu * 100) >> 20;
float Temp = ((Te * 200) >> 18)/4.0 - 50.0;
```

(b) Conversion

Code 5. Restoration and conversion for data

The above regularized value is called CRC check polynomial, one of the generating polynomials, whose G_x for the sensor is defined as follows:

$$G_x = 1 + x^4 + x^5 + x^8 \quad (8)$$

In the power items of x , the part of bit x is given in binary ("0" or "1"), and that of exponent indicates the bit position. The right side of eq. (8) means that the terms of degree 0 (= constant term), 4, 5 and 8 in x are set to "1" in generally formed eighth-degree polynomial: $\sum_{k=0}^8 x^k$.

The implemented program uses, specifically, the lower 8 bits "0b^{7 6 5 4 3 2 1 0}00110001(=0x31)" of the generator polynomial G_x , excluding the 9th bit (8-dimensional terms).

CRC calculation utilizes not only bit-shifting but also frequently the algorithm shown in Code 6, based on Ref. 9, using the processing of exclusive OR (XOR), which is considered equivalent to that of dividing. This algorithm depends on the processing flow that the variable "crc" is sequentially updated for each byte data from data [0] to data [5].

```
uint8_t crc = 0xFF;
for (int i = 0; i < 6; i++) {
    crc ^= data[i];
    for (int x = 0; x < 8; x++) {
        if (crc & 0x80) {
            crc <<= 1;
            crc ^= 0x31;
        }
        else crc <<= 1;
    }
}
```

*Successful of communication for nodes
if the crc=the received data [6] for CRC*
Code 6. CRC's algorithm

5.3 Multi-Threading

The tasks related to the transmitting and the receiving generally carry out not preferentially either one but parallelly. Therefore, the implemented program needs to form and maintain the multi-threading state which performs them individually on separated and independent threads.

FreeRTOS, one of multi-threading methods that can be implemented in embedded system using development boards, is a real-time OS with the capacity of task scheduling, which can manage and control tasks efficiently to prevent specific one from consuming time. The program for M5Stack uses FreeRTOS without requiring an additional library, whereas that for Arduino does it if installed them, in Arduino IDE. Since the author has not been able to confirm any libraries for micro:bit which can safely and reliably form the states of multi-threading, the communication system using micro:bit discussed later operates on a single thread which successively performs tasks for the transmitting and receiving.

xTaskCreate series functions are used to spawn a task (that is, leading to creating a thread) based on the FreeRTOS. A development board executes each task as a thread on a core of the CPU. M5Stack has the advantage of being able to process tasks parallelly because it has multiple cores in the CPU (dual core in Basic v2.7), whereas Arduino alternately processes them since it has only one core in its CPU. Considering consequently the specifications of these boards, the former board uses xTaskCreatePinnedToCore () to specify core number, while the latter one does xTaskCreate () with core-selection free.

xTaskCreatePinnedToCore () takes seven parameters, among which the important elements especially are: first argument, which specifies the name of a function called from newly thread; third one, which sets the stack size; fifth one, determines the priority of a task; and seventh one, indicates the core number. The stack size is that of a memory area allocated temporarily to store the values that are necessary to perform a task (e.g., receiving related task). The possible values for the priority of a task range from 0 (low) to 255 (high). The core number is set to "0" or "1" in case of Dual Core.

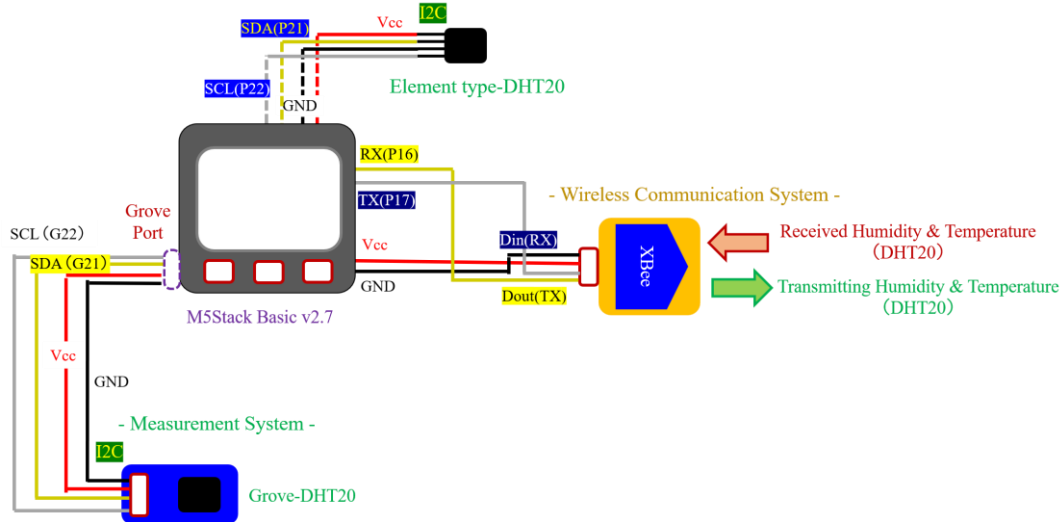
xTaskCreate () behaves like xTaskCreatePinnedToCore (), but the number of its arguments is six, one less than that of them for the latter function, which dues to independent of core selection.

FreeRTOS-related libraries for M5Stack become available when including the header file "task.h" via "M5Stack.h".

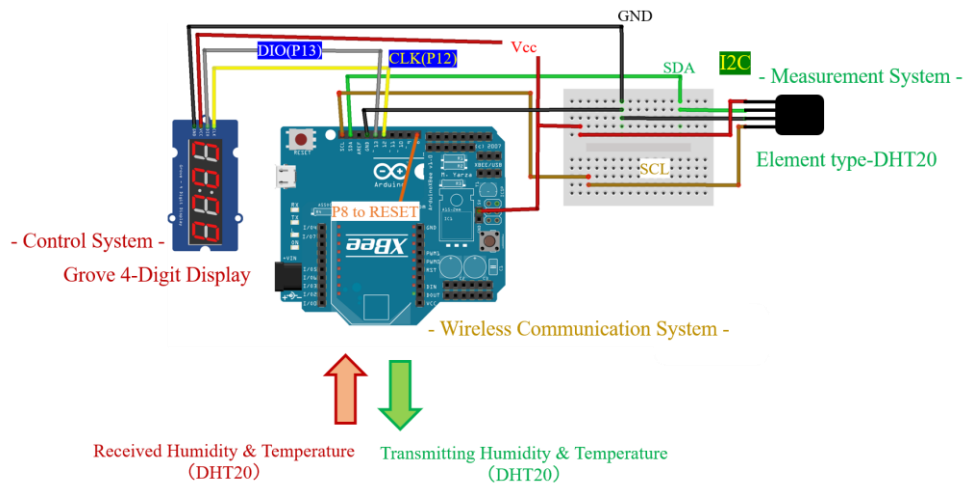
Those for Arduino require the file “Arduino_FreeRTOS.h”, which can be used after installing the “FreeRTOS” library through “LIBRARY MANAGER” window accessed via “Sketch” → “Include Library” → “Manage Libraries...” in Arduino IDE.

5.4 M5Stack - Arduino communication line

<Hardware configuration>



(a) M5Stack side



(b) Arduino side

Figure 15. Hardware configuration (M5Stack - Arduino line)

Figure 15 shows the hardware configurations for M5Stack and Arduino of a one-to-one wireless communication system. The system exchanges (transmits and receives) temperature and humidity information obtained from DHT 20 sensors between these boards using XBee modules.

On M5Stack side, as shown in the figure (a), M5Stack interfaces with a XBee module based on (the former, the latter) = (P16, Dout) and (P17, Din) pin-connected lines. P16 and P17 are the pins of RX and TX for UART communication, respectively. Additionally, it links to the Grove type DHT20 sensor through Grove port with SDA, SCL, etc. For reference, element type DHT20 sensor should be connected to M5Stack's P21(SDA) and P22 (SCL).

Arduino, another side depicted in the figure (b), is connected to XBee through a dedicated shield: Zhiwei Robotics

Corp.'s "Xbee Shield for Arduino", and also to a DHT20 sensor by SDA and SCL pins, which can be designated as I2C communication. It is further tied to Grove 4-Digit Display to show the received temperature and humidity values using the "P12-CLK" and the "P13-DIO" lines.

It is noted that this example system does not have thoroughly exception handlings since the author could not sufficiently have confirmed and verified the potential issues and the risks that might occur in a communication system.

However, the system drives based on the policy that the shortcut line linking P8 for digital output to the RESET pin allows the implemented program to restart through a digital signal via P8 when occurring these issues.

<Program implementation>

The implemented program in M5Stack is Code 7. `Wire` instanced the `TwoWire` class is prepared to obtain data of the DHT20 sensor, and the instance is initialized in the `setup()` function using `begin()` with SDA and SCL pins.

The user function "readDHT20" takes on the role of obtaining data from the sensor, restoring it to temperature and humidity values, and translating them to values of "[°C]" and "[%]" units. Sending some command signals to the sensor enables M5Stack to communicate with it, specifically following this protocol: 1) specifying the sensor-unique address "0x38" (stored in variable "address") using `beginTransmission()`

```
#include <stdio.h>
#include <string.h>
#include <M5Stack.h>
#include <Wire.h>
#define SDA 21
#define SCL 22
#define RX 16
#define TX 17
#define BAUD 9600
#define address 0x38

hw_timer_t *timer=NULL;
float peripheral_f=80e6;
float interval_time=0.5;
uint16_t division_ratio=80;
volatile SemaphoreHandle_t timerSemaphore;
portMUX_TYPE timerMux=portMUX_INITIALIZER_UNLOCKED;

volatile bool flag=false;
char str1[7];
char str2[7];
char us[]=".";
char del[]=" ";

int start=0;
int step=30;
struct DHT20_s {
    int Humi;
    float Temp;
};
DHT20_s DHT20;

void IRAM_ATTR onTimer() {
    portENTER_CRITICAL_ISR(&timerMux);
    flag=!flag;
    portEXIT_CRITICAL_ISR(&timerMux);
    xSemaphoreGiveFromISR(timerSemaphore,NULL);
}
```

Code 7. M5Stack-Arduino line (M5Stack side)
(global variables and function called when a timer interrupt is fired)

```
bool readDHT20() {
    uint8_t data[8];
    Wire.beginTransmission(address);
    Wire.write(0x71);
    Wire.endTransmission();
    delay(10);
    Wire.beginTransmission(address);
    Wire.write(0xAC);
    Wire.write(0x33);
    Wire.write(0x00);
    Wire.endTransmission();
    delay(80);
    Wire.requestFrom(address, 7);
    int k=0;
    while(1) {
        if(k==7) break;
        if (Wire.available() > 0) {
            data[k] = Wire.read();
            k++;
        }
    }
    uint8_t crc = 0xFF;
    for (int i = 0; i < 6; i++) {
        crc ^= data[i];
        for (int x = 0; x < 8; x++) {
            if (crc & 0x80) {
                crc <<= 1;
                crc ^= 0x31;
            }
            else crc <<= 1;
        }
    }
    uint32_t h1,h2,h3,t1,t2,t3;
    h1 = data[1];h2 = data[2];h3 = data[3] & 0xF0;
    uint32_t Hu = (h1 << 12) | (h2 << 4) | (h3 >> 4);
    t1 = data[3] & 0x0F;t2 = data[4];t3 = data[5];
    uint32_t Te = (t1 << 16) | (t2 << 8) | t3;
    DHT20.Humi = (Hu * 100) >> 20;
    DHT20.Temp = ((Te * 200) >> 18)/4.0 - 50.0;
    uint8_t cr = data[6];
    return (cr == crc);
}
```

Code 7. M5Stack-Arduino line (M5Stack side)
(user function "readDHT20")

function, 2) transmitting each command signal through `write ()` function, 3) completing the transmission-related processing by calling `endTransmission ()`.

When M5Stack requests data from the sensor: 4) asking it to provide the desired number of bytes of data using `requestFrom ()` function, 5) checking whether readable data have been received (the number of bytes $\neq 0$) or not through `available()` function, 6) reading data by `read()` and storing it in a memory allocated as array “data [.]”, that is, reading sequentially seven bytes of data byte by byte and storing each byte in its corresponding element of the array variable.

The purpose of the system is to measure temperature and humidity information and deliver it to the recipient when a timer interrupt event is triggered. It thus is the neutral approach or viewpoint to implement the procedure of measuring and transmitting data in `onTimer ()` called when a timer interrupt is triggered. However, it is essentially prohibited or severely limited to perform I2C communication – related processing within the called function after raising the event. This is due to the fact that I2C communication also utilizes interrupts and synchronous processing.

On the other hand, I2C communication will be exclusively executed until completed to prevent loss of data. The program, therefore, which can use both interrupt events and I2C

communication, should briefly suppress the former events using appropriate compensation codes. In addition, the processing of I2C, such as obtaining data from the DHT20 sensor in the example, needs some time.

Essentially, in a regularly raised interrupt event including the functionality of a timer like this example, the time delay due to other interrupt-unrelated independent processing could be a factor that blocks the path to the goal of the whole system. This leads to the conclusion that the other processing must be completed before the next interrupt event.

Since coexistence of timer interrupt processing and I2C communication is, restricted, the system attempts to obtain data from the DHT20 sensor via I2C based on the following protocol: 1) declaring the boolean variable “flag” in global declaration scope, 2) toggling the variable “flag” (that is, switching the state of it) in `onTimer ()` whenever an interrupt

```
void setup() {
    Wire.begin(SDA,SCL);
    M5.begin();
    M5.Lcd.setCursor(0,0);
    M5.Lcd.setTextSize(3);
    M5.Lcd.setTextColor(GREEN);
    timerSemaphore=xSemaphoreCreateBinary();
    xTaskCreatePinnedToCore(Received,"ReceivedTask",1024,NULL,1,NULL,1);
    Serial2.begin(BAUD, SERIAL_8N1,RX,TX);
    timer=timerBegin(1,division_ratio,true);
    float counter_f=peripheral_f/division_ratio;
    u_int64_t count_number=(u_int64_t)(counter_f*interval_time);
    timerAttachInterrupt(timer,&onTimer,true);
    timerAlarmWrite(timer,count_number,true);
    timerAlarmEnable(timer);
}

void loop() {
    if(xSemaphoreTake(timerSemaphore,0)==pdTRUE){
        portENTER_CRITICAL(&timerMux);
        bool fla=flag;
        portEXIT_CRITICAL(&timerMux);
        if (fla==true){
            if (readDHT20()==true) {
                M5.Lcd.clear();
                M5.Lcd.setCursor(0,start);
                M5.Lcd.printf("Humi=%d %%%\n",DHT20.Humi);
                M5.Lcd.setCursor(0,start+step);
                M5.Lcd.printf("Temp=%.2f *C\n",DHT20.Temp);
                int tem=(int)(DHT20.Temp+0.5);
                if(DHT20.Humi>99) DHT20.Humi=99;

                sprintf(str1,"%d",tem);
                strcat(str1,us);
                sprintf(str2,"%d",DHT20.Humi);
                strcat(str1,str2);
                strcat(str1,del);
                char *moji;
                moji=str1;
                Serial2.write(moji);
            }
        }
    }
}

void Received(void *arg){
    while(1){
        if(Serial2.available()>0){
            String Rx_str=Serial2.readStringUntil(0x5f);
            M5.Lcd.setCursor(0,start+3*step);
            M5.Lcd.print(Rx_str);
        }
    }
}
```

Code 7. M5Stack-Arduino line (M5Stack side)
(setup, loop and Received functions)

event fires, 3) referring to “flag” in loop (), which is independent of onTimer() with timer interrupt-related processing, and calling readDHT20() if the value of “flag” is “true”.

In the example system, measurement and communication are performed at one second intervals under the condition that the variable “flag”, toggles with each interrupt (0.5 seconds), is “true”. Thus, the timer interrupt-related parameters are set conforming to Table.3, while threshold C is set to 0.5×10^6 considering to switch the state of the variable “flag”.

The system utilizes [Serial2](#) instance for serial communication with XBee, and the configuration of transmitting data is as follows: 1) converting temperature and humidity values to integers after rounding them, 2) casting them to strings, 3) concatenating the former value and the latter one with the period character ‘.’ as delimiter, 4) appending underscore ‘_’, which a recipient will recognize as terminator of data, to the concatenated strings. The configuration will specifically be:

Temperature value + ‘.’ + Humidity value + ‘_’

The total length of this string data is 6 (those of four strings are 2, 1, 2, 1, respectively).

```
#include<Arduino_FreeRTOS.h>
#include<stdio.h>
#include<string.h>
#include "DHT.h"
#include<TM1637.h>
#define DHTTYPE DHT20
#define DIO 13
#define CLK 12
#define fclk (16e6)
#define bits 8
#define BAUD 9600
#define resetPIN 8
void Received(void *arg);
DHT dht(DHTTYPE);
TM1637 tm1637(CLK,DIO);
int8_t TimeDisp[]={0x00,0x00,0x00,0x00};
TaskHandle_t ReceiveHandler;
int frt,snd,thd,fth;
float freq=1.0;
char str[10],str1[10];
char us[]="&";
char del[]=" _ ";
float temp_hum_val[2];
int fw,bw;
int flag;
void setup() {
    digitalWrite(resetPIN,HIGH);
    pinMode(resetPIN,OUTPUT);
    Serial1.begin(BAUD);
    Wire.begin();
    dht.begin();
    for(int i=0;i<2;i++) temp_hum_val[i]=0.0;
    tm1637.init();
    tm1637.set(BRIGHT_TYPICAL);
    tm1637.point(POINT_ON);
    int pre=pow(2,bits);
    unsigned int val=(unsigned int)(fclk/(pre*freq))-1;
    TCCR1A=0b00000000;
    TCCR1B=0b00001100;
    OCR1A=val;
    TIMSK1=0b00000010;
    xTaskCreate(Received,"ReceivedTask",128,NULL,1,&ReceiveHandler);
}
```

Code 8. M5Stack-Arduino line (Arduino side: first half)

```
ISR(TIMER1_COMPA_vect){
    sei();
    flag=dht.readTempAndHumidity(temp_hum_val);
    if(flag==1) digitalWrite(resetPIN,LOW);
    if(flag==0) {
        fw=(int)(temp_hum_val[1]+0.5);
        bw=(int)(temp_hum_val[0]+0.5);
        if(fw<0) fw=1;
        if(bw>=99) bw=99;
        if(fw>-49){
            sprintf(str,"%d",fw);
            strcat(str,us);
            sprintf(str1,"%d",bw);
            strcat(str,str1);
            strcat(str,del);
            Serial1.write(str);
        }
        else digitalWrite(resetPIN,LOW);
    }
}
void loop() {
}
void Received(void *arg){
    while(1){
        if(Serial1.available()>0){
            String str=Serial1.readStringUntil(0x5f);
            int fwd=str.substring(0,2).toInt();
            int bwd=str.substring(3).toInt();
            arrange(fwd,&frt,&snd);
            TimeDisp[0]=frt;
            TimeDisp[1]=snd;
            arrange(bwd,&thd,&fth);
            TimeDisp[2]=thd;
            TimeDisp[3]=fth;
            tm1637.display(TimeDisp);
        }
    }
}
void arrange(int val, int *a,int *b){
    *a=val/10;
    *b=val%10;
}
```

Code 8. M5Stack-Arduino line (Arduino side: second half)

The user function “Received ()”, specified in `xTaskCreatePinnedToCore ()`, processes the received data from Arduino in a while loop,

which is the same the functionality of an iteration as the loop(). It drives on a separate thread from the one being responsible for transmission-related processing.

The received data is shown in the Lcd monitor of M5Stack. `Serial2.readStringUntil()` function stores the substring before the underscore '_' as one beneficial segment in the string variable "Rx_str", because the segment contains temperature and humidity information.

The program installed on Arduino is shown in Code 8. Arduino introduces the DHT class defined in the header file "DHT.h" to operate the DHT20 sensor as it can, unlike M5stack, benefit from the libraries of DHT series. Instantiating of the `DHT` class (the instance "dht") results from specifying "DHT20", the variable which is defined to suit the DHT sensor, to variable "DHTTYPE". Although the `setup()` function sequentially initializes the "dht" instance and the "Wire" one for I2C communication, the author has confirmed that I2C system will not properly drive if these initialization tasks are not completed before starting a timer (or its interrupt).

`readTempAndHumidity()` function of DHT class can obtain humidity and temperature values, which are respectively stored in the first (index: 0) and second (1) elements of double array "temp_hum_val[]" specified to its argument (unit of the former value H is [%], and that of the latter T is [°C]).

It is easier to make timer interrupts and I2C communication coexist on Arduino than on M5Stack. Especially, the program for Arduino includes the `sei()` function, which enables an interrupt throughout the program. In short, the function allows the program to communicate using I2C in the `ISR()` called when an interrupt event is triggered.

Processing of received data for Arduino, similar to M5Stack, is performed in `Received()` function, which runs on a newly created thread using `xTaskCreate()`. For data display, the program assigns the temperature value T to the upper two digits of the Grove 4-Digit Display while the humidity one H to the lower ones of it. The parameters for a timer interrupt are also according to Table 5 because its event is fired once a second.

Figure 16 shows a shot of communication. The figure indicates that the result measured using the Grove type DHT20 sensor on M5Stack side is shown as $(T [^{\circ}\text{C}], H [\%]) = (26, 49)$ on 4-Digit Display of Arduino while one done using the element type DHT20 on the latter side is displayed as $(T, H) = (25, 47)$ in the bottom row of the former's Lcd monitor. This would thus be considered a successful verification of data communication using two development boards.

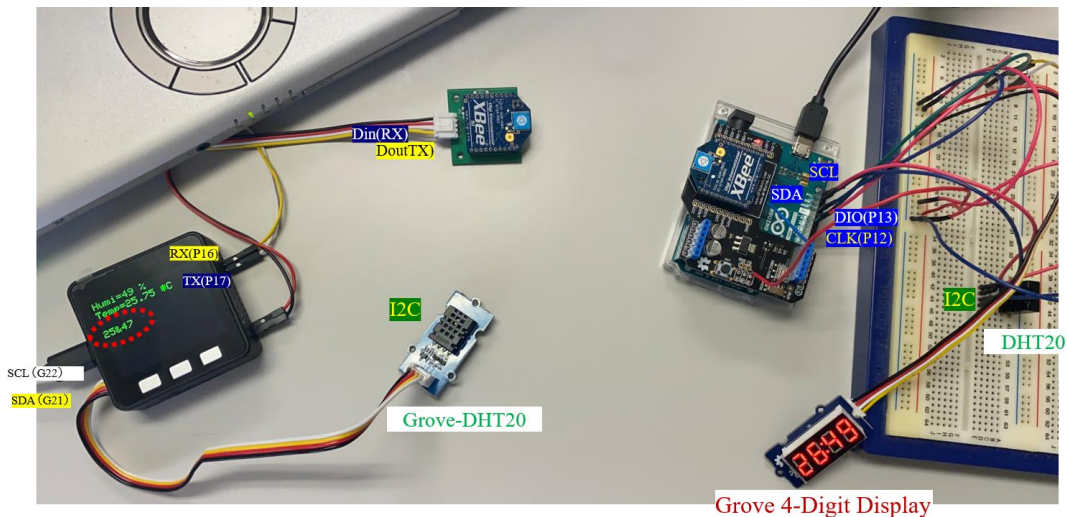


Figure 16. A case of M5Stack-Arduino communication line

5.5 M5Stack - micro:bit communication line

As discussed earlier, since the author has not confirmed a way to create and maintain safe and reliable multi-threading state in Arduino IDE, the paper discusses a one-to-one communication system for M5Stack-micro:bit line that micro:bit

performs sequentially the transmitting and receiving processes on a single thread. The M5Stack side of this communication system is exactly the same as that of M5Stack-Arduino line discussed in section 5.4. This section thus focuses on the micro:bit side and omits the M5Stack one.

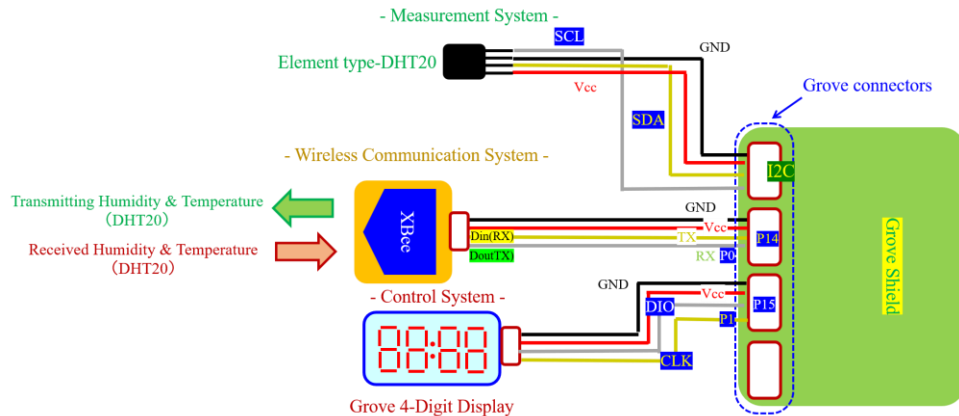


Figure 17. Hardware configuration (micro:bit)

<Hardware configuration>

Figure 17 shows the hardware configuration of the micro:bit side. The micro:bit with a Grove Shield is connected to element-type DHT20 sensor through the Grove connector for I2C communication on the shield, to XBee module forming the P14-Din and the P0-Dout lines, and to Grove 4-Digit Display utilizing the P15-DIO and the P1-CLK lines.

<Program implementation>

Code 9 is the program for micro:bit. The

```
#include <Wire.h>
#include <TM1637.h>
#define RX 0
#define TX 14
#define BAUD 9600
#define CLK 1
#define DIO 15
#define SDA 20
#define SCL 19
#define address 0x38
TM1637 tm1637(CLK,DIO);
int8_t TimeDisp[]={0x00,0x00,0x00,0x00};
char str0[6], str1[6];
char us[]=".";
char del[]="_";
int Temp,Humi;
struct DHT20_s {
    int Humi;
    float Temp;
};
DHT20_s DHT20;
```

Code 9. M5Stack - micro:bit line
(micro:bit side: global variables)

```
bool readDHT20() {
    uint8_t data[8];
    Wire1.beginTransmission(address);
    Wire1.write(0x71);
    Wire1.endTransmission();
    delay(10);
    Wire1.beginTransmission(address);
    Wire1.write(0xAC);
    Wire1.write(0x33);
    Wire1.write((byte)0x00);
    Wire1.endTransmission();
    delay(80);
    Wire1.requestFrom(address, 7);
    int k=0;
    while(1) {
        if(k==7) break;
        if (Wire1.available() > 0) {
            data[k] = Wire1.read();
            k++;
        }
    }
    uint8_t crc = 0xFF;
    for (int i = 0; i < 6; i++) {
        uint8_t b = data[i];
        crc ^= b;
        for (int x = 0; x < 8; x++) {
            if (crc & 0x80) {
                crc <<= 1;
                crc ^= 0x31;
            } else {
                crc <<= 1;
            }
        }
    }
    uint32_t h1,h2,h3,t1,t2,t3;
    h1 = data[1];h2 = data[2];h3 = data[3] & 0xF0;
    uint32_t Hu = (h1 << 12) | (h2 << 4) | (h3 >> 4);
    t1 = data[3] & 0x0F;t2 = data[4];t3 = data[5];
    uint32_t Te = (t1 << 16) | (t2 << 8) | t3;
    DHT20.Humi = (Hu * 100) >> 20;
    DHT20.Temp = ((Te * 200) >> 18)/4.0 - 50.0;
    uint8_t cr = data[6];
    return (cr == crc);
}
```

Code 9. M5Stack - micro:bit line (micro:bit side: "readDHT20")

approximate flow of processing is to show the received data from M5Stack on Grove 4-Digit Display, after which to measure the temperature and the humidity values and transmit them to it.

The `Wire1` instance, generated from `TwoWire` class to obtain data from the DHT20 sensor, is initialized via `begin()` function after designating the SDA and the SCL pins using `setPins()` in `setup()`. micro:bit v2.21 is assigned to their pins to P19 and P20 as the suitable ones for them, respectively. The procedure and the algorithm for the obtaining data in `readDHT20()` conform to those of M5Stack.

The only available instance in this micro:bit's communication system is the `Serial` one, which is initialized using `begin()` function after specifying P14 and P0 pins (for transmitting and receiving, respectively) through `setPins()`, similar to the `Wire1`.

It should be noted that the `Serial` instance cannot be used to output values on Arduino IDE's serial monitor because it is used exclusively XBee communication.

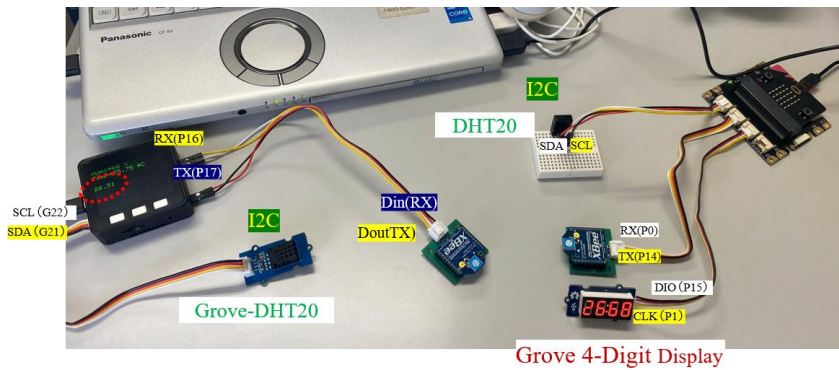


Figure 18. A case of M5Stack - micro:bit communication line

Figure 18 shows one successful communication where M5Stack displays the received information $(T, H)=(26,51)$ on its Lcd monitor while micro:bit shows it $(T, H)=(26,68)$ on the Grove 4-Digit Display. Note: the difference between the humidity values is due to an intentional increase on the M5Stack side.

6 Conclusion

This paper focuses on timer interrupts in embedded system technology using development boards, and reports mechanism and usefulness of them through the three system examples: count-down timer, generation of musical scales (chords), and one-to-one wireless data communication. While development boards make use of their own specifications and have unique definitions and policies for timer interrupts, the underlying mechanism to trigger the interrupt events is commonly based on operating efficiently and effectively their CPU and peripheral electronic circuits (prescaler, comparator and counter).

When implementing timer interrupts, this section here describes the noteworthy points which could be confirmed through three systems.

```
void setup() {
    Wire1.setPins(SDA, SCL);
    Wire1.begin();
    Serial.setPins(RX,TX);
    Serial.begin(BAUD);
    tm1637.init();
    tm1637.set(BRIGHT_TYPICAL);
    tm1637.point(POINT_ON);
}

void loop() {
    if(Serial.available()>0){
        String
        str=Serial.readStringUntil(0x5f);
        split(str);
        int frt,snd,thd,fth;
        arrange(Temp,&frt,&snd);
        arrange(Humi,&thd,&fth);
        TimeDisp[0]=frt;
        TimeDisp[1]=snd;
        TimeDisp[2]=thd;
        TimeDisp[3]=fth;
        tm1637.display(TimeDisp);
        delay(20);
        if (readDHT20()==true) {
            int te=(int)(DHT20.Temp+0.5);
            int hu=DHT20.Humi;
            if(hu>99) hu=99;
            sprintf(str0,"%d",te);
            strcat(str0,us);
            sprintf(str1,"%d",hu);
            strcat(str0,str1);
            strcat(str0,del);
            char *moji;
            moji=str0;
            Serial.write(moji);
        }
    }
}

void split(String str){
    String fwd,bwd;
    fwd=str.substring(0,2);
    bwd=str.substring(3,str.length());
    Temp=fwd.toInt();
    Humi=bwd.toInt();
}

void arrange(int num, int *t,int *f){
    *t=num/10;
    *f=num%10;
}
```

Code 9. M5Stack - micro:bit line

First, the delay () function is utilized to pause the execution of subsequent instructions temporarily in the programming development environment “Arduino IDE”. The functionality of it is managed in TIMER0 of each board. Just as timer number “1” for M5Stack, “TIMER1” and “TIMER2” for micro:bit and “TIMER1” for Arduino are preferentially used in the paper, a timer except for TIMER0, which manages the defaulted functions such as delay(), millis(), etc., would be utilized for timer interrupts.

Next, one of technical tips for Arduino to realize coexistence of timer interrupts and I2C communication in the third example, one-to-one wireless communication system, is to utilize sei() function, which enables interrupt events. However, this function is suspected to have the two following drawbacks: 1) the loss of received data in the unlikely event that the next timer interrupt is triggered before completing the measurement and communication processing, 2) the data race and deadlock that could be caused when multiple threads simultaneously access the same data and resources in multi-threading state. Thus, the system will efficiently prioritize interrupts (e.g., timer interrupt and I2C) under scheduling policy of FreeRTOS and must sufficiently take steps to prevent its overall performance from decreasing. The author will consider these points when building this type of system in the future.

Finally, the author would also like to make a plan to build a safe multi-threading environment for micro:bit in Arduino IDE. In contrast, “MakeCode” for micro:bit, an online development platform that is available on the web, can easily and quickly build such an environment. This is due to the fact that the number of threads equal to that of multiple blocks performing loop processing can be parallelly launched. The author created the one-to-one communication systems for M5Stack and Arduino using micro:bit’s program for multi-threading created in MakeCode, and confirmed their safe operation omitting their details.

For this paper, the author received much knowledge and advice of not only embedded techniques but also measurement, control and communications from the Doctors of Engineering listed below. He has been deeply grateful for their support since completing this paper, and looks forward to their continuous support and feedback for building the system.

<Embedded systems, Measurement and Control>

Shota Nakashima, Graduate School of Sciences and Technology for Innovation, Yamaguchi University.

< Measurement and Control >

Ryosuke Adachi, Graduate School of Sciences and Technology for Innovation, Yamaguchi University.

Thanks very much to the academic scientists!

References

- 1) ESP32 Series Datasheet ver.4.9, https://m5stack-doc.oss-cn-shenzhen.aliyuncs.com/644/esp32_datasheet_en.pdf
- 2) nRF52833 Product Specification v1.7, NORDIC SEMICONDUCTOR, nRF52833_PS_v1.7.pdf, pp.672-684,
https://docs.nordicsemi.com/bundle/ps_nrf52833/page/keyfeatures_html5.html
- 3) ATmega16U4/ATmega32U4 DATASHEET, pp.111-139,
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U432U4_Datasheet.pdf
- 4) For instance,
https://www.youtube.com/watch?v=ZHCjU_rcQno
KANON Johann Pachelbel, MUSIC FOR PIANO NO.448, Zen-On Music Company Limited, 1987
- 5) Mylar Speaker MSI28 Series, SPL (HONG KONG) LIMITED,
<https://www.buzzer.com.hk/member/129287/diy/MSI28-12R.pdf>
- 6) Meinard Müller, Fundamentals of Music Processing -Audio, Analysis, Algorithms, Applications-, Springer, pp.13-24, 2015
- 7) XBee®/XBee-PRO® RF Modules, <https://cdn.sparkfun.com/assets/4/9/0/0/f/XBee-Datasheet.pdf>
- 8) Data Sheet DHT20 Humidity and Temperature Module, ASAIR,
<https://cdn.sparkfun.com/assets/8/a/1/5/0/DHT20.pdf>
- 9) For instance, websites for CRC Algorithm,
<https://e2e.ti.com/support/sensors-group/sensors/f/sensors-forum/1061837/hdc3020-crc-c-calculation>
https://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch41