Networked Embedded Systems

Practicum 1: Accessing GPIOs

Group number: 8

Name	Student ID	Email
Bernberger Sarah	k12112018	sarahbernberger.sb@gmail.com
Grundner Simon	k12136610	simon.grundner@gmail.com

Friday, April 19, 2024

Map of Content

Theory Questions 4
Task A: Blink an LED 5
a) Calculations 5
b) Implementation 5
c) Results 6
d) Discussion 6
Design Decisions 6
Code Reusability 6
Task B: Vary the Cycle Time 7
a) Calculations 7
b) Implementation 7
c) Discussion 7
Task C: Vary the Duty Cycle 8
a) Calculations 8
b) Implementation 8
c) Discussion 8
Task D: Vary the Blinking Frequency with Push Buttons 9
a) Implementation 9
b) Discussion 12
Approach And Initial Thoughts 12
Limits 12
Possible Solutions 12
Issues 12
Task E: Display the Current Steps 13
a) Calculations 13
c) Implementation 14
b) Results 16
d) Discussion 16
Bit Math 16
Index 17

Figures 17
Code Segments 17
Tables 17

Theory Questions

a) What does the abbreviation GPIO mean?

GPIO is short for **G**eneral **P**urpose **I**nput **O**utput and describes one of the most basic pin functionalities of a microcontroller besides other basic pin-functions such as:

- Power Pins (Beige)
- GPIO Pins (Green or Gray)
- Strapping Pins (Olive Green)

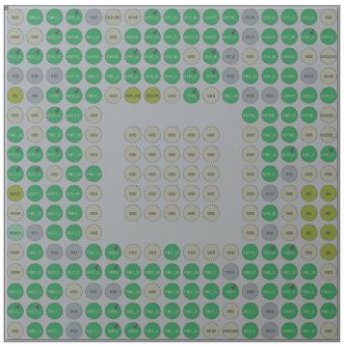

Figure 1: Pinout of the STM32H7

b) What functionality do GPIOs have and what are they used for?

Analog IO

Due to the digital nature of a microcontroller, reading and writing signals with voltage levels other than 3.3V or 0V requires a GPIO to be internally connected to a Digital-Analog-Converter (DAC) for outputs and an Analog-Digital-Converter (ADC) for inputs.

Timers

Timers are used for timing sensitive signals (e.g. PWM). A GPIO connected to a channel of a timer can be used to output this signal.

Connectivity

Certain GPIOs support various interfaces such as I2C, UART […], making it possible to connect peripherals like sensors (e.g. Gyroscope) or data storage (e.g. SD-Card) to these special GPIO-Pins.

Interrupts

GPIOs are capable of detecting edges and sending a flag to the interrupt-controller. The interrupt request can then be used to immediately execute a specified code.

c) What is a duty cycle? How do you calculate it?

The duty cycle is the percentage of a period, in which the signal level is high. To calculate the duty cycle, the formular can easily be derived from Figure 2:

$$DC = \frac{t_1}{T} \cdot 100$$

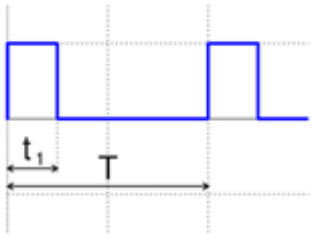

Figure 2: Duty Cycle of a Signal

Task A: Blink an LED

a) Calculations

Cycle Time	3000ms
Duty Cycle	66.6%
LED ON Time	2000ms
LED OFF Time	1000ms

Table 1: Task A calculations

b) Implementation

A function has been implemented to flash the decimal point of the segment display with a given period and a duty cycle. *HAL_GPIO_WritePin()* sets the GPIO output of a certain pin, in this case *SEGDP_Pin* which is defined in *main.h* generated with CubeMX. *HAL_Delay()* waits for a given time in milliseconds.

```
void USR_SEG_FlashDP(uint16_t period_ms, float dc)
{
  HAL_GPIO_WritePin(SEGDP_GPIO_Port, SEGDP_Pin, GPIO_PIN_SET);
  HAL_Delay(period_ms * dc);
  HAL_GPIO_WritePin(SEGDP_GPIO_Port, SEGDP_Pin, GPIO_PIN_RESET);
  HAL_Delay(period_ms * (1 - dc));
}
```

Codesegment 1: USR_SEG_FlashDP()

The Function is called indefinitely in the main while loop.

```
int main(void)
{
  const float dc = 0.666;
  const uint16_t period_ms = 3000;
  while (1)
  {
    USR_SEG_FlashDP(period_ms, dc);
  }
}
```

Codesegment 2: Relevant Code in main() for Task A

c) Results

As a result, the decimal point of the seven segment display flashes.

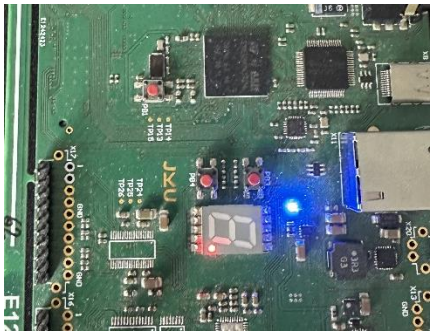

Figure 3: Blinking Decimal Point (on)

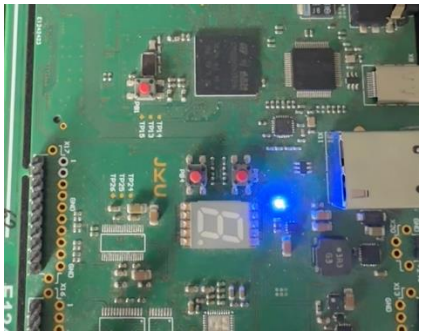

Figure 4: Blinking Decimal Point (off)

d) Discussion

Design Decisions

Following the "*Pascal_Snake*"-Casing of the HAL-Library code for a coherent code style and following the naming convention of *Library_Module_Function()*.

Example	CubeMX Code	*HAL_GPIO_Write(…)*
	Custom Code	*USR_SEG_FlashDP(…)*

Furthermore, easy to read abbreviations are used for compact function and variable names.

Code Reusability

The code has been consolidated into a function so it can be used in later tasks. If LEDs on pins other than the *SEGDP_Pin* are to be "Flashed", it would be clever to include GPIO Port and Pin into the function arguments.

The function could look like:

```
void USR_LED_FlashDC(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, uint16_t period_ms, float dc)
{
  HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_SET);
  HAL_Delay(period_ms * dc);
  HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_RESET);
  HAL_Delay(period_ms * (1.0f - dc));
}
```

Codesegment 3: Alternative Flash Function

Task B: Vary the Cycle Time

a) Calculations

	LOW	MID	HIGH	Generic
Cycle Time	1000	500	200	period_ms [ms]
Duty Cycle	66.6	66.6	66.6	dc [%]
LED ON Time	666	333	133	[ms]
LED OFF Time	333	167	66	[ms]

Table 2: Task B calculations

b) Implementation

Defines for the blinking intervals:

```
#define N_LEVELS 3
#define BLINK_COUNT 15
```

Codesegment 4: Defines for Task B

The outer for-loop is used to iterate through the levels and the inner for-loop repeats the function call of *USR_SEG_FlashDP()* 15 times. This time an array is used to store the cycle time of the individual levels.

```
int main(void)
{
  const float dc = 0.666;
  const uint16_t period_ms[] = {1000, 500, 200};
  while (1)
  {
    for (int i = 0; i < N_LEVELS; i++)
    {
      for (int j = 0; j < BLINK_COUNT; j++)
      {
        USR_SEG_FlashDP(period_ms[i], dc);
      }
    }
  }
}
```

Codesegment 5: Relevant code in main() for Task B

c) Discussion

Defines are used to reduce the amount of magic numbers in the code which makes it easier to read and understand. The *USR_SEG_FlashDP()* function can be adapted from Task A: Blink an LED.

Task C: Vary the Duty Cycle

a) Calculations

	Step 1	Step 2	Step 3	Generic
Cycle Time	2000	2000	2000	period_ms [ms]
Duty Cycle	70	50	30	dc [%]
LED ON Time	1400	1000	600	[ms]
LED OFF Time	600	1000	1400	[ms]

Table 3: Task C Calculations

b) Implementation

The outer for-loop is used to iterate through the levels and the inner for-loop repeats the function call of *USR_SEG_FlashDC()* 15 times. This time an array is used to store the duty cycle of the individual levels.

```
int main(void)
{
  const float dc[] = {.7, 0.5, 0.3};
  const uint16_t period_ms = 2000;
  while (1)
  {
    for (int i = 0; i < N_LEVELS; i++)
    {
      for (int j = 0; j < BLINK_COUNT; j++)
      {
        USR_SEG_FlashDP(period_ms, dc[i]);

      }
    }
  }
}
```

Codesegment 6: Relevant code in main() for Task C

c) Discussion

All defines, as well as the function *USR_SEG_FlashDP()* can be reused from Task B: Vary the Cycle Time.

Task D: Vary the Blinking Frequency with Push Buttons

a) Implementation

The values for defined constants are taken directly from the task sheet.

One goal is to keep the cycle time between 100 and 1000 milliseconds. The *LIMIT(x, min, max)* macro returns the input value *x* capped to the *min* or *max* value. The reason for a macro implementation instead of a function implementation is, that the macro takes all datatypes as opposed to a function.

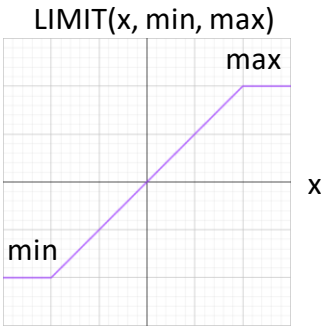

Figure 5: LIMIT() Macro-Output

Keep in mind that this macro can easily be exploited when directly calling a function within this macro (e.g. *LIMITS(Get_Next()*[1], …)) because *x* is directly replaced with the function call and would be called three times. The macro could be reinforced with an extra declaration for *x* within the macro itself but won't be implemented due to simplicity.

```
#define MIN_PERIOD_MS (100)
#define MAX_PERIOD_MS (1000)
#define BLINK_COUNT (15)
#define DC (0.5f)
#define STEP_SIZE_MS (100)

#define LIMIT(x, min, max) ((x) < (min) ? (min) : (x) > (max) ? (max) : (x))
```

Codesegment 7: Defines and Macros for Task D.

[1] This is an example function, sensitive to multiple consecutive calls.

The data we need to preserve to perform queries of a button is stored in a button-datatype with the fields shown in Codesegment 8.
To instance a button, a *Button_t* declaration can be passed into the initialization function (Codesegment 9) to set the individual fields.

```
typedef struct {
  GPIO_TypeDef* GPIOx;
  uint32_t GPIO_Pin;
  uint8_t prev;
  uint8_t has_changed;
} Button_t;
```

Codesegment 8: Button_t type

```
void USR_BTN_Init(Button_t* btn, GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  btn->GPIOx = GPIOx;
  btn->GPIO_Pin = GPIO_Pin;
  btn->prev = HAL_GPIO_ReadPin(GPIOx, GPIO_Pin);
  btn->has_changed = 0;
}
```

Codesegment 9: USR_BTN_Init()

A feature of the board is that the buttons are *active low*. This means that if a button is read while it is **not pressed**, it would return **high**[2]. Therefor we need to **invert** the condition of a logical falling edge to get a physical falling edge (releasing the button).

btn->prev	curr	Edge (logical)	btn->has_changed
0	0	No change	0
0	1	Rising edge	1
1	0	Falling edge	0
1	1	No change	0

Table 4: Truth Table for a physical falling edge

```
void USR_BTN_Check(Button_t* btn)
{
  GPIO_PinState curr = HAL_GPIO_ReadPin(btn->GPIOx, btn->GPIO_Pin);
  btn->has_changed = (curr && !btn->prev);
  btn->prev = curr;
}
```

Codesegment 10: USR_BTN_Check()

In the main program, two instances of a button are created and initiated. The Flashing function is called in the while loop as before. Additionally, each of the buttons are read and checked for any changes. Because the *has_changed* field of the Button can only be one or zero, we can determine if the current period in- or decreases by applying a sign to the *step_up* (+) and *step_dn* (-) buttons *has_changed* property.

[2] This is preferred due to current consumption of the GPIO driver.

By adding them together, we know if only one of the buttons has been pressed, otherwise *sgn* is 0.

step_up.has_changed	step_dn.has_changed	sgn (step_up.has_changed - step_dn.has_changed)
0	0	0
0	1	-1
1	0	1
1	1	0

Table 5: sgn Table

In theory the *if(sgn == 0)…* is not needed because the current cycle time would be incremented by 0, but it allows for an early return to save operations (more important in Task E: Display the Current Steps).

```
int main(void)
{
  Button_t step_up; // BTN1 (PB3 onboard button)
  Button_t step_dn; // BTN2 (PB4 onboard button)

  USR_BTN_Init(&step_up, BTN1_GPIO_Port, BTN1_Pin);
  USR_BTN_Init(&step_dn, BTN2_GPIO_Port, BTN2_Pin);

  int16_t curr_period_ms = MIN_PERIOD_MS;

  while (1)
  {
    USR_SEG_FlashDP(curr_period_ms, DC);
    USR_BTN_Check(&step_up);
    USR_BTN_Check(&step_dn);
    int8_t sgn = (step_up.has_changed - step_dn.has_changed);

    // sgn = 1 if step_up has changed,
    // -1 if step_dn has changed, 0 otherwise
    if (sgn == 0) continue;

    curr_period_ms += sgn * STEP_SIZE_MS;

    // Limit curr_period_ms to [MIN_PERIOD_MS, MAX_PERIOD_MS]
    curr_period_ms = LIMIT(curr_period_ms, MIN_PERIOD_MS, MAX_PERIOD_MS);
  }
}
```

Codesegment 11: Relevant code in main() for Task D

b) Discussion

Approach And Initial Thoughts

"How to check if a button state is changed?"

> To know if a button state is changed, the newly read button state has to be compared to the previous one. If they differ the button state changed.

"Considerations for rising and falling edge detections."

> Ignoring *how* the button changed would result in the cycle time being update twice. Once when pressed and once when released. The task sheet requires the cycle time to update when physically *releasing* the button.

"How should the button 'module' to look like?"

> Implementing the necessities above needs a custom data structure for a button to save the previous button state and if the button has changed among other things to reduce the number of parameters in function calls and provide readable code.

Limits

The issue with this implementation is, that a button is only checked after a blinking interval of the LED and does therefore not respond during a blinking cycle. With fast flashing the delay is barely noticeable but on slow delays you must hold down the button a considerable amount of time until the frequency updates.

To create a delay, *HAL_Delay()* does nothing more than wait a certain time before continuing. This causes the LED to flash, but also prevents the *USR_BTN_Check()* function to be called during this wait.

Possible Solutions

The ideal behavior would be to update the blinking frequency with a short push on the button. This can be achieved with either external interrupts on the buttons or handling the LED blinking by a timer.

Issues

A potential issue would have been not noticing that the .ioc File does not configure the buttons as inputs.

Task E: Display the Current Steps

a) Calculations

No	SEGA	SEGB	SEGC	SEGD	SEGE	SEGF	SEGG
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Table 6: Pin mask of the segment numbers

Calling *HAL_GPIO_WritePin()* for each LED on the display would be very cumbersome. Therefor a function is implemented, which writes directly to the output data register (ODR) of the segment display port. This simplification only works if all LEDs of the segment display are connected to the same GPIO port. In this case port J.

Using bit math, a map in form of an array is created which has also the convenience of returning the number mask of the index written in the brackets. In cases where there are less zeros than ones, the low pins are subtracted from the segment pin mask to shorten the expression.

```
const uint16_t segments[] = {
  /* 0 */ SEG_MASK & ~(SEGG_Pin),
  /* 1 */ SEGB_Pin | SEGC_Pin,
  /* 2 */ SEG_MASK & ~(SEGC_Pin | SEGF_Pin),
  /* 3 */ SEG_MASK & ~(SEGE_Pin | SEGF_Pin),
  /* 4 */ SEG_MASK & ~(SEGA_Pin | SEGD_Pin | SEGE_Pin),
  /* 5 */ SEG_MASK & ~(SEGB_Pin | SEGE_Pin),
  /* 6 */ SEG_MASK & ~(SEGB_Pin),
  /* 7 */ SEGA_Pin | SEGB_Pin | SEGC_Pin,
  /* 8 */ SEG_MASK,
  /* 9 */ SEG_MASK & ~(SEGE_Pin)
};
```

Codesegment 12: Pin Mask Map

c) Implementation

The defined constants are updated to match the task sheet requirements. Additional constants for the Segment display are defined as well.

```
#define MIN_PERIOD_MS (200)
#define MAX_PERIOD_MS (1900)
#define STEP_SIZE_MS (200)
#define DC (0.5f)

#define SEG_MAX_NUM (9)
#define SEG_MIN_NUM (0)
#define SEG_NUM_COUNT (SEG_MAX_NUM - SEG_MIN_NUM + 1)
#define SEG_MASK (SEGA_Pin | SEGB_Pin | SEGC_Pin | SEGD_Pin | SEGE_Pin |
SEGF_Pin | SEGG_Pin)
```

Codesegment 13: Defines for Task E

To pass information into the segment display related functions, another datatype is created.

All parameters are assigned initial values in the init function. To save one parameter in the function call, the mask of the number 8 can be used to set the *pin_mask* variable, as it lights up all the LEDs.

```
typedef struct {
  GPIO_TypeDef* GPIOx;
  uint32_t pin_mask;
  const uint16_t* segments;
  uint8_t curr_num;
  uint8_t max_num;
  uint8_t min_num;
} Segment_t;
```

Codesegment 14: Segment type

```
void USR_SEG_Init(Segment_t* seg, GPIO_TypeDef* GPIOx, const uint16_t*
segments, uint8_t min_num, uint8_t max_num)
{
  seg->GPIOx = GPIOx;
  seg->pin_mask = segments[8];
  seg->segments = segments;
  seg->min_num = min_num;
  seg->max_num = max_num;
}
```

Codesegment 15: USR_SEG_Init()

The write function for the seven segment limits the input number to 0 through 9 and modifies the register according to the predefined number mask array. This is also an example of reusability, as the LIMIT macro is used here as well.

```
void USR_SEG_Write(Segment_t* seg, int num)
{
  seg->curr_num = LIMIT(num, seg->min_num, seg->max_num);
  MODIFY_REG(seg->GPIOx->ODR,
      seg->pin_mask, seg->segments[seg->curr_num]);
}
```

Codesegment 16: USR_SEG_Write()

The only thing added to the main function is a *Segment_t* instance as well as the write function after a button press is detected. In the call of the write function, the number is incremented by the in Task D: Vary the Blinking Frequency with Push Buttons documented *sgn* variable. Checking if *sgn* is equal to zero is now more important because only new data should be written to the Output Data Register. Writing the same data over and over would be very redundant.

```
int main(void)
{
  Button_t step_up; // BTN1 (PB3 onboard button)
  Button_t step_dn; // BTN2 (PB4 onboard button)
  Segment_t seg;

  USR_SEG_Init(&seg, SEGA_GPIO_Port, segments, SEG_MIN_NUM,
SEG_MAX_NUM);
  USR_BTN_Init(&step_up, BTN1_GPIO_Port, BTN1_Pin);
  USR_BTN_Init(&step_dn, BTN2_GPIO_Port, BTN2_Pin);

  USR_SEG_Write(&seg, 0);

  int16_t curr_period_ms = MIN_PERIOD_MS;

  while (1)
  {
    USR_SEG_FlashDP(curr_period_ms, DC);
    USR_BTN_Check(&step_up);
    USR_BTN_Check(&step_dn);
    int8_t sgn = (step_up.has_changed - step_dn.has_changed);

    if (sgn == 0) continue;

    USR_SEG_Write(&seg, seg.curr_num + sgn);
    curr_period_ms += sgn * STEP_SIZE_MS;
    curr_period_ms = LIMIT(curr_period_ms, MIN_PERIOD_MS,
                           MAX_PERIOD_MS);
  }
}
```

Codesegment 17: Relevant code in main for Task E

b) Results

Following figures show various states of the program.

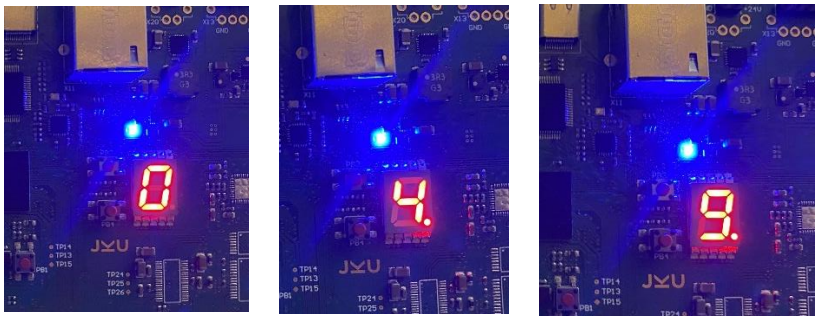

Figure 6: Results Task E

d) Discussion

As mentioned in Limits from Task E, updating the cycle requires one to hold down a button for a longer period of time.

Bit Math

Useful bit operations:

```
#define SET_BIT(REG, BIT)     ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)    ((REG) & (BIT))
#define CLEAR_REG(REG)        ((REG) = (0x0))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))
#define READ_REG(REG)         ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG),
(((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
```

Codesegment 18: Bitmath macros

Index

Figures

Figure 1: Pinout of the STM32H7 4
Figure 2: Duty Cycle of a Signal 4
Figure 3: Blinking Decimal Point (on) 6
Figure 4: Blinking Decimal Point (off) 6
Figure 5: LIMIT() Macro-Output 9
Figure 6: Results Task E 16

Code Segments

Codesegment 1: USR_SEG_FlashDP() 5
Codesegment 2: Relevant Code in main() for Task A 5
Codesegment 3: Alternative Flash Function 6
Codesegment 4: Defines for Task B 7
Codesegment 5: Relevant code in main() for Task B 7
Codesegment 6: Relevant code in main() for Task C 8
Codesegment 7: Defines and Macros for Task D. 9
Codesegment 8: Button_t type 10
Codesegment 9: USR_BTN_Init() 10
Codesegment 10: USR_BTN_Check() 10
Codesegment 11: Relevant code in main() for Task D 11
Codesegment 12: Pin Mask Map 13
Codesegment 13: Defines for Task E 14
Codesegment 14: Segment type 14
Codesegment 15: USR_SEG_Init() 14
Codesegment 16: USR_SEG_Write() 15
Codesegment 17: Relevant code in main for Task E 15
Codesegment 18: Bitmath macros 16

Tables

Table 1: Task A calculations 5
Table 2: Task B calculations 7
Table 3: Task C Calculations 8
Table 4: Truth Table for a physical falling edge 10
Table 5: sgn Table 11
Table 6: Pin mask of the segment numbers 13