
GEBZE TECHNICAL UNIVERSITY

ENGINEERING FACULTY

ELECTRONICS ENGINEERING

# ELEC 334

## MICROPROCESSORS

## HW4


| | |
|---|---|
| Name - Surname | Yağmur DERYA |
| Student ID | 171024011 |

**Problem 1 – Board Support Package**

A BSP is essential code for a given computer hardware device that will make that device work with the computer's operating system.

```c
/*
 * bsp.h
 *
 *      Author: Yagmur Derya
 */

#ifndef BSP_H_
#define BSP_H_

/* for on-board LED on PC6 */
void BSP_init_led(void);
void BSP_set_led(void);
void BSP_clear_led(void);
void BSP_toggle_led(void);

/* for button on PB3 */
void BSP_init_button(void);
int BSP_read_button(void);

#endif /* BSP_H_ */
```

```c
/*
 * bsp.c
 *
 *      Author: Yagmur Derya
 */

#include "bsp.h"
#include "stm32g0xx.h"

void BSP_init_led(void){

	/* Enable GPIOC clock */
	RCC->IOPENR |= (1U << 2); // 100

	/* Setup PC6 as output */
	GPIOC->MODER &= ~(3U << 2*6); // clear bits ~(11) = 00
	GPIOC->MODER |= (1U << 2*6); // set PC6 as output : 01

	/* clear LED */
	GPIOC->BRR |= (1U << 6);
}

void BSP_set_led(void){

	/* turn on */
	GPIOC->ODR |= (1U << 6);
}

void BSP_clear_led(void){

	/* turn off */
	GPIOC->BRR |= (1U << 6);
}
```

```c
void BSP_toggle_led(void){

	/* toggle */
	GPIOC->ODR ^= (1U << 6);
}

void BSP_init_button(void){

	/* Enable GPIOB clock */
	RCC->IOPENR |= (1U << 1); // 10

	/* Setup PB3 as input */
	GPIOB->MODER &= ~(3U << 2*3);
}

// returns 1 if button is pressed
int BSP_read_button(void){
	int b = ((GPIOB->IDR >> 3) & 0x1);
	return (b) ? 1 : 0;
}
```

**Problem 2 – Software Faults in the world**

1. On its mission to Mars in 1998, the Climate Orbiter spacecraft was lost in space. Although the failure confused the engineers for some time, it was revealed that a sub-contractor on the engineering team failed to make a simple conversion from English units to metric. These miscalculations had an impact on the flight path and in the end, the probe was destroyed because of friction with the Martian atmosphere.

2. In February 1991, an Iraqi missile hit the US base of Dhahran in Saudi Arabia and killed 28 American soldiers. The base's antiballistic system failed to launch because of a computer bug, which is the "Patriot missile battery", whose role is to detect and intercept enemy missiles by "crashing" against them in mid-air, had been running for 100 hours straight. After every hour, the internal clock drifted by milliseconds and that had a huge impact on the system which equals to 0.33 seconds after 100 hours. This micro-delay translates into a 600-meter error. The radar first identified an object in the sky but didn't manage to track it due to the error and didn't launch itself.

3. Blue screen of death occurred during a presentation of a Windows 98 beta given by Bill Gates on April 20, 1998. The demo PC crashed with BSOD when it is connected a scanner to demonstrate Windows 98's support for Plug and Play devices. https://www.youtube.com/watch?v=IW7Rqwwth84

4. Mariner 1 was the first spacecraft in the American Mariner program, and it was launched on July 22, 1962 on a mission to collect a variety of scientific data about Venus. Shortly after take-off the rocket responded improperly to the commands from the guidance systems on the ground, setting the stage for an apparent software guidance system failure. With the craft effectively uncontrolled, a range safety officer ordered its destructive abort. The destruct command was sent 6 seconds before separation, after which the launch vehicle could not have been destroyed. The radio transponder continued to transmit signals for 64 seconds after the destruct command had been sent.
   https://www.youtube.com/watch?v=fmJVPtVlCWk

**Problem 3 – Bouncing**

When a button is pushed, two metal parts come together. Inside the switch, there are moving parts and when the switch pushed, it initially contacts with the other metal part but just in a brief split of a microsecond. Then it contacts a little longer and then again, a little longer. In the end the switch is fully closed. The switch is bouncing between in-contact and not in-contact. Usually the hardware works faster than the bouncing which results in that the hardware things you are pressing the switch several times.

One of the hardware prevention methods is to connect a capacitor (analog filtering), 0.1µF etc., to filter bounces.

Another hardware prevention method is to use an SR flip-flop. The circuit consists of two NAND gates forming a SR F-F. Whenever the switch is moving between the contacts to create the bounce, the F-F maintains the output because the 0 is fed back from the output of the NAND gates.

One of the software prevention methods is using a counting algorithm. Most people use a simple approach that looks for n sequential stable readings of the switch, where n is a number ranging from 1 to seemingly infinity. Generally, the code detects a transition and then starts incrementing or decrementing a counter, each time rereading the input, till n reaches some presumable safe, bounce-free, count. If the state isn't stable, the counter resets to its initial value. A simple debouncer that is called every CHECK_MSEC by the timer interrupt:

```
#define CHECK_MSEC 5 // Read hardware every 5 msec
#define PRESS_MSEC 10 // Stable time before registering pressed
#define RELEASE_MSEC 100 // Stable time before registering released

// This function reads the key state from the hardware.
extern bool_t RawKeyPressed();

// This holds the debounced state of the key.
bool_t DebouncedKeyPress = false;

// Service routine called every CHECK_MSEC to
// debounce both edges
void DebounceSwitch1(bool_t *Key_changed, bool_t *Key_pressed)
{
	static uint8_t Count = RELEASE_MSEC / CHECK_MSEC;
	bool_t RawState;
	*Key_changed = false;
	*Key_pressed = DebouncedKeyPress;
	RawState = RawKeyPressed();
	if (RawState == DebouncedKeyPress) {
		// Set the timer which allows a change from current state.
		if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
		else   Count = PRESS_MSEC / CHECK_MSEC;
	} else {
		// Key has changed - wait for new state to become stable.
		if (--Count == 0) {
			// Timer expired - accept the change.
			DebouncedKeyPress = RawState;
			*Key_changed=true;
			*Key_pressed=DebouncedKeyPress;
			// And reset the timer.
			if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
			else Count = PRESS_MSEC / CHECK_MSEC;
		}
	}
}
```

The code doesn't wait for n stable states before declaring the debounce over. Instead it's all based on time and is therefore very portable and maintainable. `DebounceSwitch1()` returns two parameters. `Key_pressed` is the current debounced state of the switch and `Key_changed` signals the switch has changed from open to closed or the reverse.

For quick response and relatively low computational overhead, one to five milliseconds are ideal. Most switches seem to exhibit bounce rates under 10 ms. 50 ms response seems instantaneous, it's reasonable to pick a debounce period in the 20 to 50 ms range. A similar method is shift register method. The algorithm assumes unsigned 8-bit register value usually found in microcontrollers.

1. Initially set-up a shift register variable to 0xFF
2. Using a timer set-up, a sampling event with a period
3. On the sample event; shift the variable towards MSB
4. Set LSB to current switch value
5. If shift reg = 0 - set internal switch state to pressed
6. Else - set internal switch state to released

**Problem 4 – State Machines**

The BSP which is created for problem 1 is used in this code with a little change in initial button function;

```
void BSP_init_button(void){

	/* Enable GPIOB clock */
	RCC->IOPENR |= (1U << 1); // 10

	/* Setup PB3 as input */
	GPIOB->MODER &= ~(3U << 2*3);

	EXTI->RTSR1 |= (1U << 3);
	EXTI->EXTICR[0] |= (1U << 8*3);
	EXTI->IMR1 |= (1U << 3);

	NVIC_SetPriority(EXTI2_3_IRQn, 0);
	NVIC_EnableIRQ(EXTI2_3_IRQn);
}
```

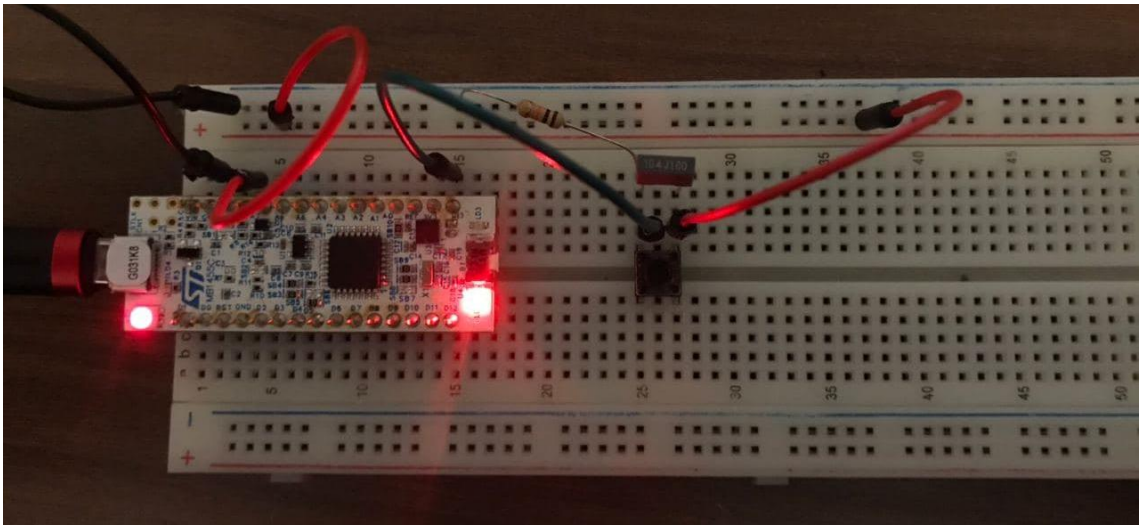

**Figure 1.** on-board circuit for problem 4.

I used 10kΩ impedance and 100nF capacitor to debounce push-button.

```c
/*
 *
 * Yagmur Derya
 *
 *
 */

#include "stm32g0xx.h"
#include "bsp.h"

int main(void);
void SysTick_Handler(void);
void delay_ms(uint32_t);
void StateMode(int);
void EXTI2_3_IRQHandler(void);

int count = 0;
int mode = 0;
volatile uint32_t TimeDelay;
```

```c
int main(void) {
	SystemCoreClockUpdate(); // to update processor clock speed
	SysTick_Config(SystemCoreClock/1000); // 1 ms interrupt interval
=16MHz/1ms = 16 000 = SystemCoreClock/1000

	BSP_init_led();
	BSP_init_button();

    while(1){
	  mode = count;
	  StateMode(mode % 5);
    }
    return 0;
}
```

```c
void SysTick_Handler(void){
	if (TimeDelay != 0)
		TimeDelay --;
}
```

```c
void delay_ms(uint32_t time){
	TimeDelay = time;
	while(TimeDelay != 0);
}
```

```c
void StateMode(int mode){
	if(mode == 0){	//LED is OFF
		BSP_clear_led();
	} else if(mode == 1){ // LED is toggling at 1 second intervals
		delay_ms(1000);
		BSP_toggle_led();
	} else if(mode == 2){ // LED is toggling at 0.5 second intervals
		delay_ms(500);
		BSP_toggle_led();
	} else if(mode == 3){ // LED is toggling at 0.1 second intervals
		delay_ms(100);
		BSP_toggle_led();
	} else if(mode == 4){ // LED is ON
```

```
            BSP_set_led();
      }
}
```

```
void EXTI2_3_IRQHandler(void){
      count ++;
      EXTI->RPR1 |= (1U << 3); // clear pending
}
```

**Problem 5 – Trends in Embedded Software Engineering**

Embedded system developers must consider resource limitations and environmental conditions. Because of that, developers often create a product-safe hardware platform.
The main difference between IT systems and embedded system is the need to create specialized hardware and software platforms. IT system users are used to wait for the system to react but in embedded system, there must be real time reacts, for example a car should stop when it is braked etc.
Also embedded system developers must have extensive domain knowledge such as physics etc.
Model-driven development is one of the promising approaches that have emerged over the last decade. In this approach, generators automatically create the code implementing the system functionalities. Researchers improved many approaches focusing on the specific aspects of MDD.
Developers can create a system functional design based on the system requirements. These designs cover an embedded system's functionalities without any technical implementation details but not all embedded software systems require this design.
An architecture definition consists of various views covering the architecture's different aspects. After defining the system architecture, developers refine the different components to obtain an executable and platform-independent system.
MDD is based on general purpose languages and code generators for different application domains.
Next generation languages and tools will provide extended extension and tailoring mechanisms, potentially combining the advantages of MDD's generality with those of specialized domain-specific modelling languages.
Ensuring the quality of a system's functional and extra-functional properties is crucial in embedded systems development.
To ensure software quality, developers can apply techniques such as static and formal verifications, but these techniques have different drawbacks. For example, applying formal techniques to a complete, complex software system is impossible and formally proven software might be still unsafe and safe software might not be completely correct.
Model-based safety and reliability analysis provides information at an early stage however it requires additional effort and time.
The measurement-based approach is only applicable after the system has been fully implemented and because measurements are never complete, producing dependable results requires statistical techniques.
However, many developers use dynamic testing. They can apply tests based on the design models in early design phases, enabling a continuous quality-assurance process and dynamic tests can be applied to complex systems.
Developers can implement testing in the actual operating environment such as those usually not detected through formal verification. Also, dynamic testing is scalable.
In the development of safety-critical systems, new functionalities are useless if you can't ensure their quality and safety.

**Problem 6 – Applying test driven development to embedded software**

The Test-Driven Development workflow has some steps;

- Create a new test
- Do a build, run all the tests and see the new one fail
- Write the code to make the test pass
- Do a build, run all the tests and see the new one pass
- Refactor to remove duplication
- Repeat

When it is decided to work on a specific module, that module has a set of responsibilities and behaviours that allow the module to provide what is needed to the embedded system. The models are typically used to partition responsibilities, while leaving the details in code.

TDD provides benefits to the developer and the development team such as improved predictability, which is improved because TDD breaks programming into a series of small verifiable tests that can be used as a measure of progress, repeatability, which is crucial because the software will continue to evolve throughout the product development cycle and potentially in future products, and reduced debugging time which is reduced by repeatable cause-and-effect testing.

Unit tests provide feedback to the developer about if the code written works as it is expected or not. Unit tests are behavioural tests and attempt to fully exercise the module under test. They also provide regression test and relieve the programmer from having to repeatedly manually test the same code.

Acceptance tests operate on integrated groups of modules to show that the software meets its requirements. Nonprogrammers write these tests in an application-specific test language.

TDD can be applied as unit tests and as acceptance tests but in this article is mainly concerned with unit testing.

The embedded TDD cycle is an extension of the core TDD cycle. The first stage is the traditional TDD cycle and is performed on the development system with the goal of developing clean code with very few defects. In the second stage, which concerns the risk of incompatible compiler feature, the target compiler is used to ensure that no compile and link problems with the target exist. In the third and fourth stages, the automated unit tests are run on the evaluation board and then the target. Finally, end-to-end manual tests are performed to ensure that the system is correctly wired. End-to-end testing is essential and embedded developers know that integrating early is one key success factor.

The ideal situation is to run all automated tests on the actual hardware but it is often not possible as a result of concurrent hardware and software development and constrained memory size on the target. The embedded TCC cycle helps to remove some of the roadblock associated with hardware scarcity.

Testing in the development system is an important accelerator for projects because the development system is a proven and more stable execution environment and it is the first place to run any hardware-independent test.

Although it limits the risk of doing all the testing at the end of the project, using a testing approach on a development system can introduce other risks. Stage 3 addresses these risks.

To make embedded software testable, the hardware dependencies must be isolated by designing the embedded software for this purpose. By telling the application what it wants done, not how to do it, code and tests become more intuitive and future hardware changes are made easier.

## References:

[1] https://raygun.com/blog/costly-software-errors-history/
[2] https://www.allaboutcircuits.com/technical-articles/switch-bounce-how-to-deal-with-it/
[3] http://www.heptapod.com/lib/statemachines/