Embedded System

Cortex
Intelligent Processors by ARM

Week 10 Lab

ISE Department
Prof. Mehdi Pirahandeh

Cortex
Intelligent Processors by ARM®

Embedded System with 32L476G DISCOVERY

Contents

인하대학교

Clock Configuration

STM32L476 Clock System

- According to the datasheets, STM32L4xx series can speed up to 80MHz clock speed.
- But when you reset the device, the clock speed is set to 4MHz by default.

That is painful, right? I mean, it is even 4 times SLOWER than the maximum clock speed of ATmega128, or default value of Arduino Uno.

So, to **maximize the performance** of your development board, you should understand the **clock system of STM32Lxx series**.

Code with Maximum Clock!

This time, you will use these techniques with the example code from before.

```c
#include <stm32l4xx.h>

void ClockInit(void);

int main(void)
{
    ClockInit();

    RCC->AHB2ENR = RCC_AHB2ENR_GPIOBEN
                 | RCC_AHB2ENR_GPIOEEN;
    GPIOB->MODER &= ~GPIO_MODER_MODE2_1;
    GPIOE->MODER &= ~GPIO_MODER_MODE8_1;

    while (1)
    {
        GPIOB->BSRR = GPIO_BSRR_BS2;
        GPIOE->BSRR = GPIO_BSRR_BS8;

        for (int i = 0; i < 1000000; i++);

        GPIOB->BSRR = GPIO_BSRR_BR2;
        GPIOE->BSRR = GPIO_BSRR_BR8;

        for (int i = 0; i < 1000000; i++);
    }
}
```

Code with Maximum Clock!

```c
void ClockInit(void)
{
    FLASH->ACR |= FLASH_ACR_LATENCY_4WS;

    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN
                 | (20 << RCC_PLLCFGR_PLLN_Pos)
                 | RCC_PLLCFGR_PLLM_0
                 | RCC_PLLCFGR_PLLSRC_HSI;
    RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;

    while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS)
          && (RCC->CR & RCC_CR_PLLRDY)
          && (RCC->CR & RCC_CR_HSIRDY)));

    RCC->CFGR = RCC_CFGR_SW_PLL;

    RCC->CR &= ~RCC_CR_MSION;
}
```

Make a new project and C source file.
Type this code to it, build and debug the project.

I HIGHLY RECOMMEND you type this by your own with comments, not to copy and paste it.

Please be sure to build(F7) first, before debug(Ctrl + F5).

If there is a debugger connection error, check if your debugger is set to ST-LINK. You can check it at **Project -> Options for Target…(Alt + F7) -> Debug**.
This icon is also in the toolbar.

Code with Maximum Clock!

If you are having trouble when building, especially you see this message:
`Error: L6320W: Ignoring --entry command. Cannot find argument 'Reset_Handler’.`

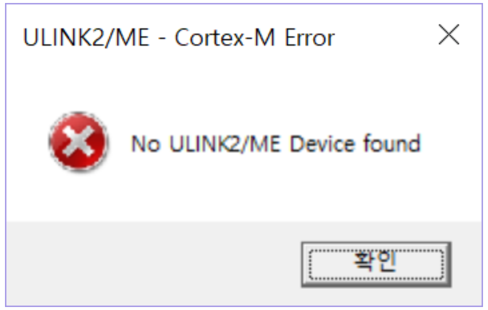
Then you might have forgot to include software packs.

That’s okay. They can be included after project creation.

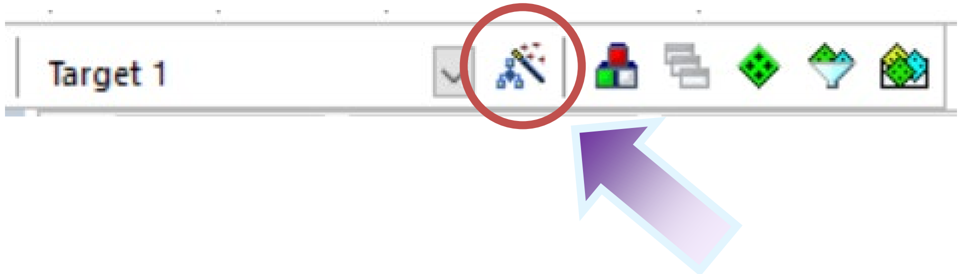
Push ‘Manage Run-Time Environment’ button and include **CMSIS -> CORE** and **Device -> Startup**.
Once you click ‘OK’, everything will work fine.

Code with Maximum Clock!

If you are having trouble when debugging, especially you see this message:



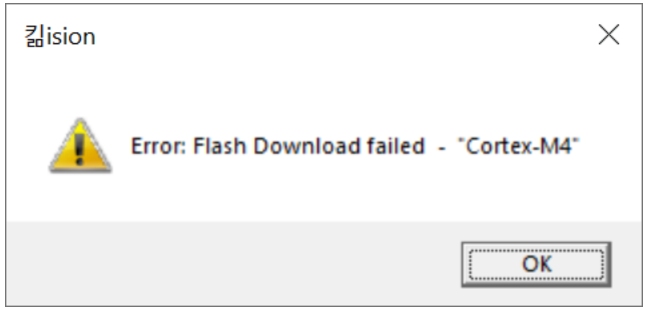
Then you might have forgot to set the debugger to ST-Link.



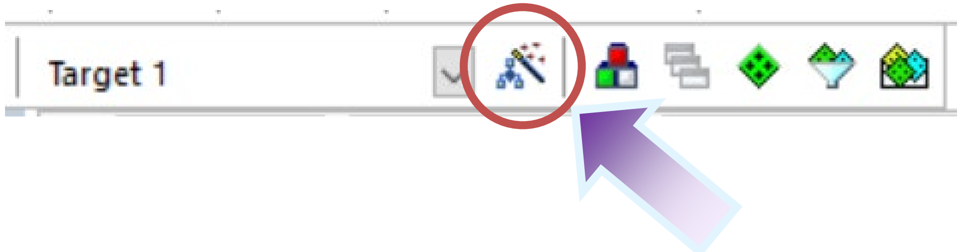
Go to **Project -> Options for Target…(Alt + F7, or the icon at the picture above) -> Debug** and set the debugger to '**ST-Link Debugger**'.
Make sure that the circle is selected next to "**Use:**".

Code with Maximum Clock!

If you are having trouble when debugging, especially you see this message:



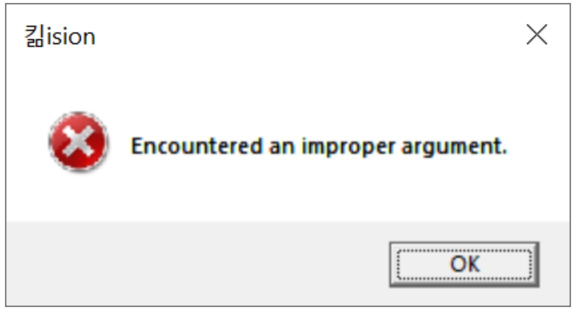
It is due to the instability of Keil.



Go to **Project -> Options for Target…(Alt + F7, or the icon at the picture above) -> Debug** and press '**Settings**' next to 'ST-Link Debugger'.
At 'Target Com', change the clock speed to any value you want, press 'OK', then come back to this window, change the value back to 4 and press 'OK' again.

Code with Maximum Clock!

If you see this message:

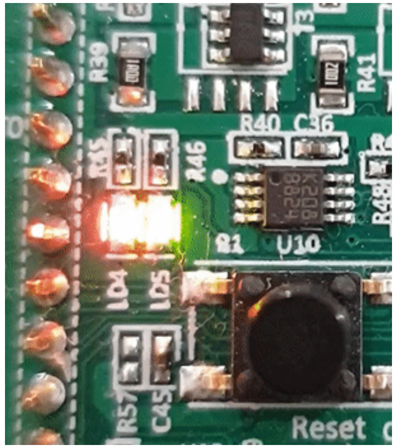


There is no way to correct it.
Open your task manager, exit the IDE and restart.

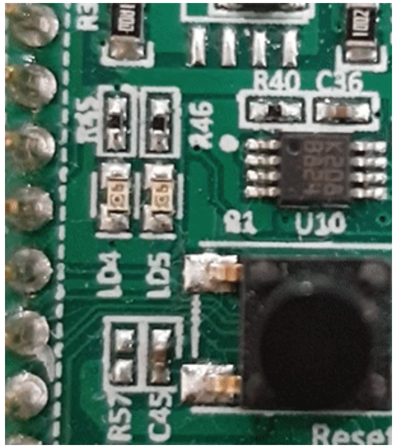
Code with Maximum Clock!

Press ‘**Run**’(F5) on the toolbar. If everything works fine, congratulations!
You may now be seeing the LEDs blinking at some crazy speed.

Indeed, that the manual stone-aged sentences to give some delay(for loop) are executed about 20 times faster than before.
This is the power of high-speed clock system.



4MHz



20MHz

But… that means is there not any way to give the accurate delay we want?
Nope, you can use the peripheral named timer, for time-related tasks.
Especially in this purpose, the Arm Cortex-M4 system timer called ‘SysTick’.

Maximum Clock Configuration

The default value of the system clock is 4MHz.
To maximize the performance to your project, you have to change the clock settings.
There are several ways to achieve it, but to our labs, I would recommend you my beautiful clock plan(???).

My beautiful plan is like this:

- **Use HSI**
  It is more accurate than MSI
- **Not use MSI**
  The system won't change its clock frequency during run-time
- **Use PLL**
  To boost up the clock rate to 80MHz from 16MHz HSI
- **Not prescale SYSCLK to AHB**
  The system core will use 80MHz

This is the simplest configuration that can also achieve the highest system clock.

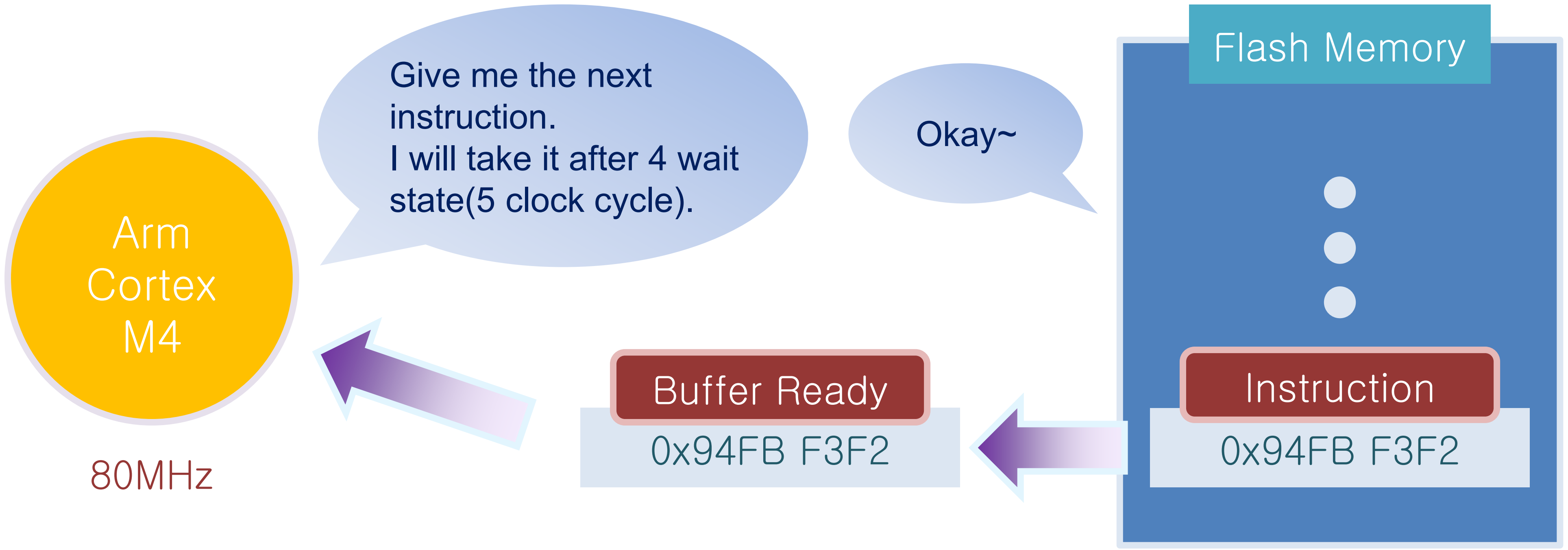
Maximum Clock Configuration

There is one problem to concern. The configuration uses PLL, so the system must ensure that the restrictions of PLL is not violated.

To be sure that the configuration obeys the restrictions, we will use the following PLL configuration:

- **Divide the input clock by 2**
  - To be sure that the input frequency to PLL is between 4MHz and 16MHz
  - Now the clock speed is **8MHz**
- **Multiply the clock by 20**
  - To be sure that the output frequency from PLL is between 64MHz and 344MHz
  - Now the clock speed is **160MHz**
- **Divide that output by 2**
  - To be sure that the system clock speed does not exceed 80MHz
  - Now the clock speed is **80MHz**

Maximum Clock Configuration



According to the datasheet, at 80MHz clock rate, the read latency must be over 4 WS. (STM32L4xx series datasheet, page 100)

RCC(Reset and Clock Controller)

3.7.1 Flash access control register (FLASH_ACR)

Address offset: 0x00

Reset value: 0x0000 0600

Access: no wait state, word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	SLEEP_PD	RUN_PD	DCRST	ICRST	DCEN	ICEN	PRFTEN	Res.	Res.	Res.	Res.	Res.	LATENCY[2:0]		
	rw	rw	rw	rw	rw	rw	rw						rw	rw	rw

Bits 2:0 **LATENCY[2:0]**: Latency

These bits represent the number of HCLK (AHB clock) period to the Flash access time.

000: Zero wait state

001: One wait state

010: Two wait states

011: Three wait states

100: Four wait states

others: Reserved

There is `LATENCY[2:0]` bits in the `ACR` register in `FLASH`.

We want 4 wait states, so you should set `LATENCY[2]`.

But thanks to <stm32l476xx.h>, there is a nice bit mask named '`FLASH_ACR_LATENCY_4WS`'.

We will use this bit mask macro constant.

```
FLASH->ACR |= FLASH_ACR_LATENCY_4WS;
```

Be sure you don't change other bits by using '|='. Otherwise, you will turn off the caches that enhance the execution speed.

RCC(Reset and Clock Controller)

6.4.4 PLL configuration register (RCC_PLLCFGR)

Address offset: 0x0C

Reset value: 0x0000 1000

Access: no wait state, word, half-word and byte access

This register is used to configure the PLL clock outputs according to the formulas:

- f(VCO clock) = f(PLL clock input) × (PLLN / PLLM)
- f(PLL_P) = f(VCO clock) / PLLP
- f(PLL_Q) = f(VCO clock) / PLLQ
- f(PLL_R) = f(VCO clock) / PLLR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PLLPDIV[4:0]					PLLR[1:0]		PLL REN	Res.	PLLQ[1:0]		PLL QEN	Res.	Res.	PLLP	PLL PEN
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	PLLN[7:0]							Res.	PLLM[2:0]			Res.	Res.	PLLSRC[1:0]	
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw

Bits 26:25 **PLLR[1:0]**: Main PLL division factor for PLLCLK (system clock)

Set and cleared by software to control the frequency of the main PLL output clock PLLCLK. This output can be selected as system clock. These bits can be written only if PLL is disabled.

PLLCLK output clock frequency = VCO frequency / PLLR with PLLR = 2, 4, 6, or 8

00: PLLR = 2
01: PLLR = 4
10: PLLR = 6
11: PLLR = 8

Caution: The software has to set these bits correctly not to exceed 80 MHz on this domain.

Bit 24 **PLLREN**: Main PLL PLLCLK output enable

Set and reset by software to enable the PLLCLK output of the main PLL (used as system clock).
This bit cannot be written when PLLCLK output of the PLL is used as System Clock.
In order to save power, when the PLLCLK output of the PLL is not used, the value of PLLREN should be 0.
0: PLLCLK output disable
1: PLLCLK output enable

Next, we will configure the PLL. Especially PLLR signal of the peripheral named 'PLL'(there are other PLLs, you remember?), because it is the only signal that can be the system clock.

First, the final division factor was 2, that divides 160MHz PLL output to 80MHz system clock. But that is the default value of `PLLR[1:0]`, so we only need to set `PLLREN`.

RCC(Reset and Clock Controller)

6.4.4 PLL configuration register (RCC_PLLCFGR)

Address offset: 0x0C

Reset value: 0x0000 1000

Access: no wait state, word, half-word and byte access

This register is used to configure the PLL clock outputs according to the formulas:

- f(VCO clock) = f(PLL clock input) × (PLLN / PLLM)
- f(PLL_P) = f(VCO clock) / PLLP
- f(PLL_Q) = f(VCO clock) / PLLQ
- f(PLL_R) = f(VCO clock) / PLLR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PLLPDIV[4:0]					PLLR[1:0]		PLL REN	Res.	PLLQ[1:0]		PLL QEN	Res.	Res.	PLLP	PLL PEN
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	PLLN[7:0]							Res.	PLLM[2:0]			Res.	Res.	PLLSRC[1:0]	
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw

Bits 14:8 **PLLN[6:0]**: Main PLL multiplication factor for VCO

Set and cleared by software to control the multiplication factor of the VCO. These bits can be written only when the PLL is disabled.

VCO output frequency = VCO input frequency x PLLN with 8 =< PLLN =< 86

0000000: PLLN = 0 wrong configuration

0000001: PLLN = 1 wrong configuration

...

0000111: PLLN = 7 wrong configuration

0001000: PLLN = 8

0001001: PLLN = 9

...

1010101: PLLN = 85

1010110: PLLN = 86

1010111: PLLN = 87 wrong configuration

...

1111111: PLLN = 127 wrong configuration

Caution: The software has to set correctly these bits to assure that the VCO output frequency is between 64 and 344 MHz.

The multiplication factor was 20, that boosts up the 8MHz signal to 160MHz.
It is hard to set each of the bits or to find that the required value is (`0b10100 << 8`).
Instead, you can use this phrase.

```
20 << RCC_PLLCFGR_PLLN
```

RCC(Reset and Clock Controller)

6.4.4 PLL configuration register (RCC_PLLCFGR)

Address offset: 0x0C

Reset value: 0x0000 1000

Access: no wait state, word, half-word and byte access

This register is used to configure the PLL clock outputs according to the formulas:

- f(VCO clock) = f(PLL clock input) × (PLLN / PLLM)
- f(PLL_P) = f(VCO clock) / PLLP
- f(PLL_Q) = f(VCO clock) / PLLQ
- f(PLL_R) = f(VCO clock) / PLLR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PLLPDIV[4:0]					PLLR[1:0]		PLL REN	Res.	PLLQ[1:0]		PLL QEN	Res.	Res.	PLLP	PLL PEN
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Res.	PLLN[7:0]							Res.	PLLM[2:0]			Res.	Res.	PLLSRC[1:0]	
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw

Bits 6:4 **PLLM:** Division factor for the main PLL and audio PLL (PLLSAI1 and PLLSAI2) input clock

Set and cleared by software to divide the PLL, PLLSAI1 and PLLSAI2 input clock before the VCO. These bits can be written only when all PLLs are disabled.

VCO input frequency = PLL input clock frequency / PLLM with 1 <= PLLM <= 8

000: PLLM = 1
001: PLLM = 2
010: PLLM = 3
011: PLLM = 4
100: PLLM = 5
101: PLLM = 6
110: PLLM = 7
111: PLLM = 8

Caution: The software has to set these bits correctly to ensure that the VCO input frequency ranges from 4 to 16 MHz.

The division factor of the HSI input was 2, that divides 16MHz clock frequency to 8MHz. This time, it seems that setting the bit with its number is easier, since the binary number of PLLM[2:0] doesn't match to the number of division factor.
So we will use '`RCC_PLLCFGR_PLLM_0`' macro instead.

RCC(Reset and Clock Controller)

6.4.4 PLL configuration register (RCC_PLLCFGR)

Address offset: 0x0C

Reset value: 0x0000 1000

Access: no wait state, word, half-word and byte access

This register is used to configure the PLL clock outputs according to the formulas:

- f(VCO clock) = f(PLL clock input) × (PLLN / PLLM)
- f(PLL_P) = f(VCO clock) / PLLP
- f(PLL_Q) = f(VCO clock) / PLLQ
- f(PLL_R) = f(VCO clock) / PLLR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PLLPDIV[4:0]					PLLR[1:0]		PLL REN	Res.	PLLQ[1:0]		PLL QEN	Res.	Res.	PLLP	PLL PEN
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	PLLN[7:0]							Res.	PLLM[2:0]			Res.	Res.	PLLSRC[1:0]	
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw	rw

Bits 1:0 **PLLSRC**: Main PLL, PLLSAI1 and PLLSAI2 entry clock source

Set and cleared by software to select PLL, PLLSAI1 and PLLSAI2 clock source. These bits can be written only when PLL, PLLSAI1 and PLLSAI2 are disabled.

In order to save power, when no PLL is used, the value of PLLSRC should be 00.

00: No clock sent to PLL, PLLSAI1 and PLLSAI2

01: MSI clock selected as PLL, PLLSAI1 and PLLSAI2 clock entry

10: HSI16 clock selected as PLL, PLLSAI1 and PLLSAI2 clock entry

11: HSE clock selected as PLL, PLLSAI1 and PLLSAI2 clock entry

Finally, we want the PLL to use the clock source from HSI.
You can set that bit separately, but there is a better macro constant.
We will use ‘`RCC_PLLCFGR_PLLSRC_HSI`’.

RCC(Reset and Clock Controller)

The final form of the code looks like this:

```
RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | (20 << RCC_PLLCFGR_PLLN_Pos)
             | RCC_PLLCFGR_PLLM_0 | RCC_PLLCFGR_PLLSRC_HSI;
```

Like this, if you want to turn several bits on at the same time, you can Bitwise OR the bit masks you want to turn on. In case of turning off, OR the masks first, and then invert it.

The other bits should remain 0, so it is okay to just use ‘=’, instead of ‘|=’.

We just finished setting the PLL, so let’s turn it on with HSI clock source.

RCC(Reset and Clock Controller)

6.4.1 Clock control register (RCC_CR)

Address offset: 0x00

Reset value: 0x0000 0063. HSEBYP is cleared upon power-on reset. It is not affected upon other types of reset.

Access: no wait state, word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	PLL SAI2 RDY	PLL SAI2 ON	PLL SAI1 RDY	PLL SAI1 ON	PLL RDY	PLLON	Res.	Res.	Res.	Res.	CSS ON	HSE BYP	HSE RDY	HSE ON
		r	rw	r	rw	r	rw					rs	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	HSI ASFS	HSI RDY	HSI KERON	HSION	MSIRANGE[3:0]				MSI RGSEL	MSI PLLEN	MSI RDY	MSION
				rw	r	rw	rw	rw	rw	rw	rw	rs	rw	r	rw

Bit 24 **PLLON**: Main PLL enable

Set and cleared by software to enable the main PLL.

Cleared by hardware when entering Stop, Standby or Shutdown mode. This bit cannot be reset if the PLL clock is used as the system clock.

0: PLL OFF

1: PLL ON

Bit 8 **HSION**: HSI16 clock enable

Set and cleared by software.

Cleared by hardware to stop the HSI16 oscillator when entering Stop, Standby or Shutdown mode.

Set by hardware to force the HSI16 oscillator ON when STOPWUCK=1 or HSIASFS = 1 when leaving Stop modes, or in case of failure of the HSE crystal oscillator.

This bit is set by hardware if the HSI16 is used directly or indirectly as system clock.

0: HSI16 oscillator OFF

1: HSI16 oscillator ON

In the CR, you can find the bits that turn on the clock sources.
Be careful! Until before you change your clock source to PLL, you MUST NOT TURN OFF MSI.
That is the reason we do '|=' again here.

```
RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;
```

Then you have to wait for the flash wait state be affected, and the oscillator circuits be stabilized.

RCC(Reset and Clock Controller)

You have to wait for the flash wait state to be affected, and oscillators to be stabilized.
There are some bits set by hardware, to show its state.
We can read that value by Bitwise AND to those bits.

In C, any non-zero value means ‘true’, so if you AND the value with the bit mask, it means ‘true’ when the bit was set, and ‘false’ when the bit was clear.

6.4.1 Clock control register (RCC_CR)

Address offset: 0x00

Reset value: 0x0000 0063. HSEBYP is cleared upon power-on reset. It is not affected upon other types of reset.

Access: no wait state, word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	PLL SAI2 RDY	PLL SAI2 ON	PLL SAI1 RDY	PLL SAI1 ON	PLL RDY	PLLON	Res.	Res.	Res.	Res.	CSS ON	HSE BYP	HSE RDY	HSE ON
		r	rw	r	rw	r	rw					rs	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	HSI ASFS	HSI RDY	HSI KERON	HSION	MSIRANGE[3:0]				MSI RGSEL	MSI PLLEN	MSI RDY	MSION
				rw	r	rw	rw	rw	rw	rw	rw	rs	rw	r	rw

The flash wait state can be known by read back `LATENCY[2:0]` bits.

Using empty while loop to wait for those bits are set, the sentence looks like this:

```
while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS)
      && (RCC->CR & RCC_CR_PLLRDY)
      && (RCC->CR & RCC_CR_HSIRDY)));
```

RCC(Reset and Clock Controller)

6.4.3 Clock configuration register (RCC_CFGR)

Address offset: 0x08

Reset value: 0x0000 0000

Access: 0 ≤ wait state ≤ 2, word, half-word and byte access

1 or 2 wait states inserted only if the access occurs during clock source switch.

From 0 to 15 wait states inserted if the access occurs when the APB or AHB prescalers values update is on going.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	MCOPRE[2:0]			MCOSEL[3:0]				Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
	rw	rw	rw	rw	rw	rw	rw								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STOP WUCK	Res.	PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw

Bits 1:0 **SW[1:0]**: System clock switch

Set and cleared by software to select system clock source (SYSCLK).

Configured by HW to force MSI oscillator selection when exiting Standby or Shutdown mode.

Configured by HW to force MSI or HSI16 oscillator selection when exiting Stop mode or in case of failure of the HSE oscillator, depending on STOPWUCK value.

00: MSI selected as system clock

01: HSI16 selected as system clock

10: HSE selected as system clock

11: PLL selected as system clock

Finally, it's time to switch the system clock to the clock from PLL.
That is done by `SW[1:0]` bits, and there is a wonderful macro constant named '`RCC_CFGR_SW_PLL`'.

```
RCC->CFGR = RCC_CFGR_SW_PLL;
```

After this, we can turn off MSI to save power, by clearing `MSION` bit in `CR`.
Remember when turning off, you can use Bitwise AND with Bitwise NOT.

```
RCC->CR &= ~RCC_CR_MSION;
```

Arm Cortex-M4
SysTick

Make your own Delay() function

The final code is below:

```
#include <stm32l4xx.h>

//System Milliseconds
unsigned int sysMillis = 0;

//Initialize the clock to 80MHz and SysTick
void ClockInit(void);
//Insert delay in milliseconds
void Delay(unsigned int duration);

int main(void)
{
    ClockInit();

    RCC->AHB2ENR = RCC_AHB2ENR_GPIOBEN
                 | RCC_AHB2ENR_GPIOEEN;
    GPIOB->MODER &= ~GPIO_MODER_MODE2_1;
    GPIOE->MODER &= ~GPIO_MODER_MODE8_1;

    while (1)
    {
        GPIOB->BSRR = GPIO_BSRR_BS2;
        GPIOE->BSRR = GPIO_BSRR_BS8;

        //for (int i = 0; i < 1000000; i++);
        Delay(1000);

        GPIOB->BSRR = GPIO_BSRR_BR2;
        GPIOE->BSRR = GPIO_BSRR_BR8;

        //for (int i = 0; i < 1000000; i++);
        Delay(1000);
    }
}
```

Make your own Delay() function

The final code is below:

```
#include <stm32l4xx.h>

//System Milliseconds
unsigned in

//Initializ
void ClockI
//Insert de
void Delay(

int main(void)
{
    ClockInit();

    RCC->AHB2ENR = RCC_AHB2ENR_GPIOBEN
                 | RCC_AHB2ENR_GPIOEEN;
    GPIOB->MODER &= ~GPIO_MODER_MODE2_1;
    GPIOE->MODER &= ~GPIO_MODER_MODE8_1;
```

```
    while (1)
    {
        GPIOB->BSRR = GPIO_BSRR_BS2;
        R_BS8;

        1000000; i++);

        R_BR2;
        R_BR8;

        //for (int i = 0; i < 1000000; i++);
        Delay(1000);
    }
}
```

- The comments for GPIOs are for the next topic of the lab.
- This is the last time I wrote the code to this slide directly with comments since this lab contains a lot to do.
- By the next lab, you should be familiar with the code style like this and be able to write the code and comments on your own.

Make your own Delay() function

```
void ClockInit(void)
{
    //Increase the delay by 4 wait states(5 clock cycles) to read the flash
    FLASH->ACR |= FLASH_ACR_LATENCY_4WS;

    //Enable PLLR that can be used as the system clock
    //Divide the 16MHz input clock by 2(to 8MHz), multiply by 20(to 160MHz),
    //divide by 2(to 80MHz)
    //Set PLL input source to HSI
    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | (20 << RCC_PLLCFGR_PLLN_Pos)
                 | RCC_PLLCFGR_PLLM_0 | RCC_PLLCFGR_PLLSRC_HSI;

    //Turn on HSI oscillator and PLL circuit.
    RCC->CR |= RCC_CR_PLLON | RCC_CR_HSION;

    //Be sure that the wait state of the flash changed,
    //PLL circuit is locked, and HSI is stabilized
    while (!((FLASH->ACR & FLASH_ACR_LATENCY_4WS)
          && (RCC->CR & RCC_CR_PLLRDY)
          && (RCC->CR & RCC_CR_HSIRDY)));
```

Make your own Delay() function

```
    //Set the system clock source from PLL
    RCC->CFGR = RCC_CFGR_SW_PLL;

    //Turn off MSI to reduce power consumption
    RCC->CR &= ~RCC_CR_MSION;

    //Read the calibrated value of 24-bit, and put it in the reload value register
    SysTick->LOAD = SysTick->CALIB & SysTick_LOAD_RELOAD_Msk;

    //Enable SysTick with the exception request
    SysTick->CTRL = SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;
}
```

Make your own Delay() function

```
void Delay(unsigned int duration)
{
    //Temporarily store 'sysMillis' and compare the difference between
    //the value and 'duration' until the value be equal to or bigger than 'duration’
    //This method is okay with overflow
    unsigned int prevMillis = sysMillis;
    while (sysMillis - prevMillis <= duration);
}

//Increase 'sysMillis' by 1 every 1ms
void SysTick_Handler(void) { sysMillis++; }
```

Make your own Delay() function

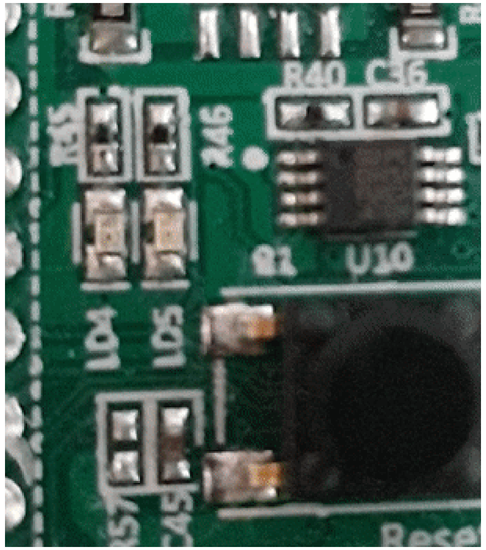
Now let's test your Delay() function.

At the example code we used for now, the ancient-version delays

```
for (int i = 0; i < 1000000; i++);
```

are replaced by

```
Delay(1000);
```



Build, Debug and Run.
See how the LEDs are blinking.

Pretty awesome than before, right?

SysTick Timer

Arm Cortex-M4 has the special counter inside the core, that can be used for generating system tick interrupt for operating system.
The name of that counter is SysTick.

Even though we do not use an operating system, it is very helpful to use it as a basic timer that counts the milliseconds elapsed after initialization.

SysTick timer counts down from the reload value, and if it reaches to 0, it returns to the reload value at the next timer clock.

SysTick Timer

By default, SysTick timer gets the clock source from AHB divided by 8.
So for our configuration, the clock speed to the timer is 10MHz.

If you want the time of one period of the counter becomes 1ms, you can set the reload value with the following formula:

Reload Value = (0.001 * (HCLK / 8)) – 1

where HCLK is the clock frequency of AHB.

For our case, the reload value is

0.001 * 10M – 1 = 9,999

SysTick Timer

According to the datasheet, the calibrated reload value might be in `TENMS[23:0]` bits in `CALIB` register.
This value is originally used for generating timer overflow event every 10ms, as the name of the bits implies. This value is stored by the hardware manufacturer.

But STMicroelectronics does not precisely calculated STM32L476VG SysTick, but instead they set this value to be 1ms when the clock speed is 80MHz in theory.

That means, the `TENMS` value is fixed to 9,999`(0x270F)`.

As we know what the value should be, we could set the reload value directly, but for the future compatibility, let's use `TENMS` value anyway.

SysTick Exception

4.5.2 SysTick reload value register (STK_LOAD)

Address offset: 0x04

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								RELOAD[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RELOAD[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

4.5.4 SysTick calibration value register (STK_CALIB)

Address offset: 0x0C

Reset value: 0x0000000

Required privilege: Privileged

The CALIB register indicates the SysTick calibration properties.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NO REF	SKEW	Reserved						TENMS[23:16]							
r	r							r	r	r	r	r	r	r	r
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TENMS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

You may read the value of `CALIB` register, Bitwise AND with ‘`SysTick_LOAD_RELOAD_Msk`’, that is equal to 0x00FFFFFF, to ignore the last two bits and assign to `LOAD` register.

```
SysTick->LOAD = SysTick->CALIB & SysTick_LOAD_RELOAD_Msk;
```

SysTick Exception

At the time when the counter value reaches from 1 to 0, `COUNTFLAG` bit in `CTRL` register is set. Moreover, if `TICKINT` bit in `CTRL` register is set, at that time SysTick exception is generated.

To deal with the exception, you must use:

```
void SysTick_Handler(void) { … }
```

If you are using C++, you should cover the exception handlers using extern “C” keyword.

```
extern “C” {
void SysTick_Handler(void) { … }
}
```

Now let’s turn on the counter and enable the exception request.
Please refer to the datasheet of Cortex-M4, since SysTick timer is a core peripheral.

SysTick Exception

4.5.1 SysTick control and status register (STK_CTRL)

Address offset: 0x00

Reset value: 0x0000 0000

Required privilege: Privileged

The SysTick CTRL register enables the SysTick features.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

Bit 1 **TICKINT**: SysTick exception request enable

0: Counting down to zero does not assert the SysTick exception request

1: Counting down to zero to asserts the SysTick exception request.

Note: Software can use COUNTFLAG to determine if SysTick has ever counted to zero.

Bit 0 **ENABLE**: Counter enable

Enables the counter. When ENABLE is set to 1, the counter loads the RELOAD value from the LOAD register and then counts down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

0: Counter disabled

1: Counter enabled

As we turn on these two bits, SysTick timer will be activated and generate the exception when it reaches to zero.

Be aware that there is no bit mask constants defined like 'SysTick_CTRL_ENABLE' for the core peripherals. Instead you should use '`SysTick_CTRL_ENABLE_Msk`'.

```
SysTick->CTRL = SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;
```

It may be good to put these sentences in `ClockInit()` from before.

SysTick Exception

4.5.1 SysTick control and status register (STK_CTRL)

Address offset: 0x00

Reset value: 0x0000 0000

Required privilege: Privileged

The SysTick CTRL register enables the SysTick features.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNTFLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSOURCE	TICKINT	ENABLE
													rw	rw	rw

Bit 1 **TICKINT**: SysTick exception request enable

0: Counting down to zero does not assert the SysTick exception request

1: Counting down to zero to asserts the SysTick exception request.

Note: Software can use COUNTFLAG to determine if SysTick has ever counted to zero.

Bit 0 **ENABLE**: Counter enable

Enables the counter. When ENABLE is set to 1, the counter loads the RELOAD value from the LOAD register and then counts down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

0: Counter disabled

1: Counter enabled

As we

reache

- **There is a C function that can be used as '`SysTick_Config(tick)`', but it is much slower than the registers approach.**

Be aware that there is no bit mask constants defined like 'SysTick_CTRL_ENABLE' for the core peripherals. Instead you should use '`SysTick_CTRL_ENABLE_Msk`'.

```
SysTick->CTRL = SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;
```

It may be good to put these sentences in `ClockInit()` from before.

Additional Code to Make Delay()

To make it work, you will need the global variable:

```
unsigned int sysMillis = 0;
```

The SysTick Exception Handler must increase this variable every exception.

```
void SysTick_Handler(void) { sysMillis++; }
```

Delay() gets the duration of the delay in milliseconds, temporarily store current '`sysMillis`' and compare it to the real '`sysMillis`' value until the difference be bigger than the duration.

```
void Delay(unsigned int duration) {
    unsigned int prevMillis = sysMillis;
    while (sysMillis - prevMillis <= duration);
}
```

THANK YOU

인하대학교