Lab 2 Pre-lab

This lab, like many others, will require you to look into technical documents and find information like memory address, register fields, and other processor-specific details that you must be aware of to write functioning code. The reference manual will be heavily relied on in this lab (and in later labs). Make sure that that document is downloaded and accessible.

The reference manual: rm0090-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf (found on canvas)

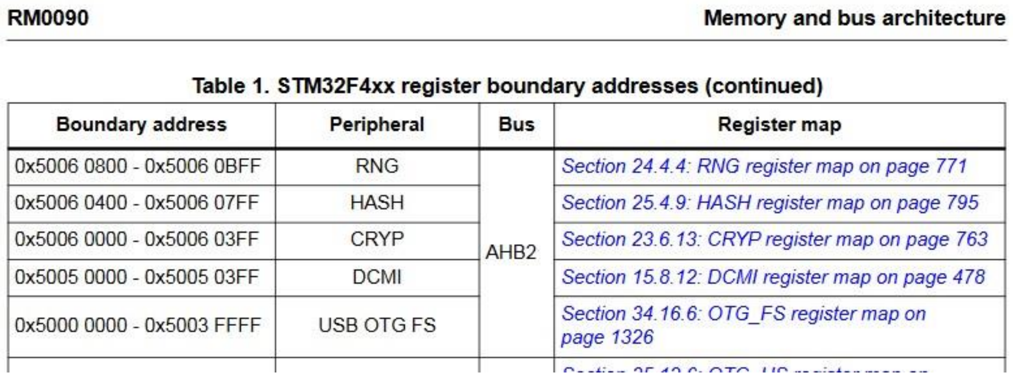
The electrical schematic document:
en.MB1075-F429I-D02_Schematic.pdf (found on canvas)

Much of the information pulled from the reference manual will inherently reside in STM32F429I.h (a file you will create and populate). There will NOT be a source file (as in a STM32F429I.c) with this, and this header file should only be included by driver codes directly.

Below are the key terms that **MAY** be useful for this lab and many others.

Key Terms

- RNG (Random Number Generator) - This peripheral will be an example for explaining and connecting other key terms.
- Bus - These can be looked at as predetermined "routes" that help the processor communicate with the peripherals. A lecture will contain a more in-depth explanation of the bus architecture.
- Memory Map - This will give a peripheral's boundary address (from the CPU's perspective) and which system bus it is on.



RM0090 **Memory and bus architecture**

Table 1. STM32F4xx register boundary addresses (continued)

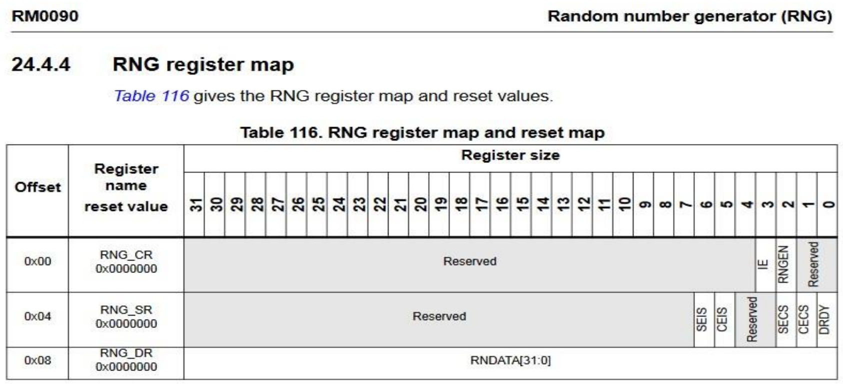
Boundary address	Peripheral	Bus	Register map
0x5006 0800 - 0x5006 0BFF	RNG	AHB2	*Section 24.4.4: RNG register map on page 771*
0x5006 0400 - 0x5006 07FF	HASH		*Section 25.4.9: HASH register map on page 795*
0x5006 0000 - 0x5006 03FF	CRYP		*Section 23.6.13: CRYP register map on page 763*
0x5005 0000 - 0x5005 03FF	DCMI		*Section 15.8.12: DCMI register map on page 478*
0x5000 0000 - 0x5003 FFFF	USB OTG FS		*Section 34.16.6: OTG_FS register map on page 1326*

  - For RNG, we can tell the memory address range is 0x50060800 - 0x50060BFF
    - This means the base address is 0x50060800
  - For RNG, we can also conclude it is on the AHB2 Bus
    - This bus information will be critical for Reset and Clock Control
  - Writing code to depict the memory address for this peripheral will consist of the following:

        - Knowing the base address of the AHB2 Bus
        - Knowing the offset of the RNG peripheral from the AHB2 base address
        - The code would look something like this:

```
#define AHB2_BASE_ADDR          0x50000000u
#define RNG_BASE_ADDR           (AHB2_BASE_ADDR + 0x60800)
```

- Register Map - This will give the register names and reset values of a given peripheral and the **offset** of that peripheral's **base address**



RM0090 Random number generator (RNG)

24.4.4 **RNG register map**

Table 116 gives the RNG register map and reset values.

Table 116. RNG register map and reset map

Offset	Register name reset value	Register size																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	RNG_CR 0x0000000	Reserved																												IE	RNGEN	Reserved	
0x04	RNG_SR 0x0000000	Reserved																									SEIS	CEIS	Reserved		SECS	CECS	DRDY
0x08	RNG_DR 0x0000000	RNDATA[31:0]																															

    - For RNG, it has three registers.
    - The data structure needed to express this register map in the code using C would look like the following:

```
typedef struct
{
   volatile uint32_t CR;     // Address Offset 0x00
   volatile uint32_t SR;     // Address Offset 0x04
   volatile uint32_t DR;     // Address Offset 0x08
}RNG_RegDef_t;
```

    - CR - Control Register, SR - Status Register, DR - Data Register
    - Understand the importance of the keyword volatile, this means the compiler will not optimize the value and will force all read and write accesses to go to the memory address where that value is stored
    - We use size uint32_t due to each register being 32 bits

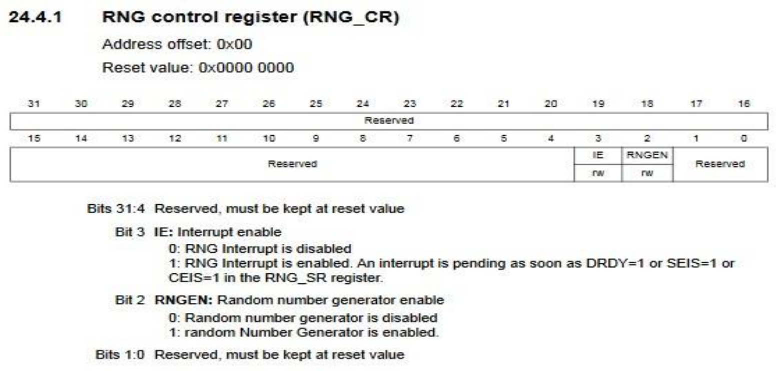
- Pointer Type Casting - We may want to cast a pointer of a particular type. In this case, we want the pointer to be casted to reflect the memory map.
    - For RNG, we would want to have a pointer to the memory address, and we will use members of the struct (the one we made above) to coordinate with what memory offset we want to access.
        - To do this, we need:

```
#define RNG      ((RNG_RegDef_t*) RNG_BASE_ADDR)
```

        - This will allow us to access the register set. Below is how we would access the Control Register:

```
RNG->CR
```

- Bit Mask - Sometimes, there may be two or more bits in a register that we want to manipulate.
    - If we had a 16-bit register and wanted to access the last three bits, we would use 0x7 as a bitmask.
        - This is due to 0x7 being 0b'111 (I removed the leading zeros, the compiler would add the leading zeros)
    - If we had a 32-bit register and wanted to access the MSB (The 31st bit), we would use C's shift operator to assist us with the bit mask.
        - This would be (1 << 31), which would get compiled as 0x80000000.
- Predefined Register Functionality - A certain value must be written to a register to obtain a specified result. Sometimes we want a macro or something to signify this so we don't have to write many "magic numbers". For example, consider the RNG Control register



24.4.1 RNG control register (RNG_CR)

Address offset: 0x00

Reset value: 0x0000 0000

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

Bits 31:4 Reserved, must be kept at reset value

Bit 3 **IE:** Interrupt enable
0: RNG Interrupt is disabled
1: RNG Interrupt is enabled. An interrupt is pending as soon as DRDY=1 or SEIS=1 or CEIS=1 in the RNG_SR register.

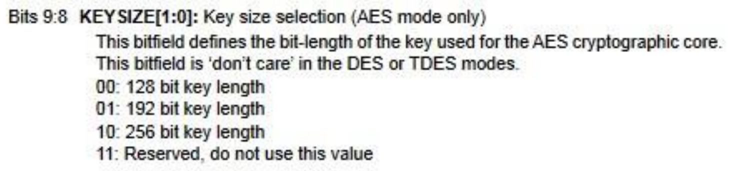
Bit 2 **RNGEN:** Random number generator enable
0: Random number generator is disabled
1: random Number Generator is enabled.

Bits 1:0 Reserved, must be kept at reset value

    - Notice how for each configurable bit in the register, they give us what each **bit** will do.
        - We then can take this information and put it in Macro form like :

```
#define RNG_INTERRUPT_DISABLE 0
#define RNG_INTERRUPT_ENABLE  1
```

    - Another example with a completely different register set (this is for the cryptographic processor) with more than 2 potential control options



Bits 9:8 **KEYSIZE[1:0]:** Key size selection (AES mode only)
This bitfield defines the bit-length of the key used for the AES cryptographic core.
This bitfield is 'don't care' in the DES or TDES modes.
00: 128 bit key length
01: 192 bit key length
10: 256 bit key length
11: Reserved, do not use this value

    - The macros in this case would look like:

```
#define KEY_SIZE_128_BIT            0
#define KEY_SIZE_192_BIT            1
#define KEY_SIZE_256_BIT            2
```

- Register Operations (known as bitwise operations) - These are operations we use to configure specific bit(s) in a register.
    - The legal operations that can be done are the following:
        - & - AND
        - | - OR
        - ~ - NOT
        - ^ - TOGGLES
    - If I want to do an operation on a register, I do the operation with the appropriate bit mask and then store the result back into the register.
        - Let's look at the RNG registers. f I wanted to write to CR register, more specifically, the bit that enables the RNG. I would need to do the following.

          ```
          RNG->CR = RNG->CR | (1 << 2);
          ```

            - I can also do the following:

              ```
              RNG->CR |= (1 << 2);
              ```

        - If I wanted to clear that bit, I would do the following:

          ```
          RNG->CR= RNG->CR & ~(1 << 2);
          ```

            - I can also do the following:

              ```
              RNG->CR &= ~(1 << 2);
              ```

        - If I wanted to read all bits in the status register, I would do the following:

          ```
          uint32_t statusRegRead = RNG->SR;
          ```

        - If I wanted to only store the bottom 5 bits (bits 4-0), I would do the following:

          ```
          uint32_t statusRegRead = RNG->SR & 0x1F;
          ```

- Resets and Clock Control (RCC) - This is responsible for enabling, disabling, and resetting clocks for peripherals and other components on the microcontroller.
    - You must enable the clock to the given peripheral for it to work.
        - This enabling of the clock should be done BEFORE you configure the peripheral (ie execute the initialization function or functionality)
    - RCC has a large register map containing clock control for all buses and other timer-related things like PLLs.
    - Since we know RNG is on the AHB2 bus, we would want to enable to clock using the AHB2 Enable Register (AHB2ENR). Below is a very oversimplification of the register map:

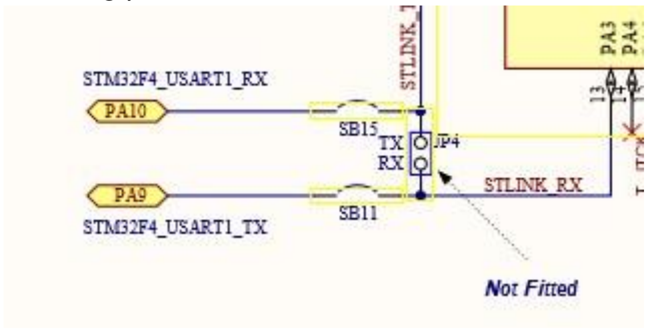
      ```
      typedef struct
      {
        volatile uint32_t STUFF;
        volatile uint32_t AHB2ENR;
      ```

```
        volatile uint32_t MORESTUFF[4];
     }RCC_RegDef_t;
```

        - Note, the concept of an array being a member of the struct is a valid concept
            - You SHOULD use this approach when dealing with consectutive reserved memory sections of a register map OR when a register's functionality spans across to registers and is continuous in memory.
    - Enabling the RNG clock enable bit in the RCC register look would look like the following:

```
RCC-> AHB2ENR |= (1 << 6);
```

- Ports and Pins -
    - Using the following document that shows the electric schematic, we can locate all ports and pin numbers that are supported by the processor. Consider the following picture:



    - If we wanted to know what port and pin to use for the USART1_RX line, we could look at the schematic and see PA10. This means port A pin 10.
        - This port is associated with GPIO.. so this would be GPIO port A pin 10
        - How to tell the processor what to do with this information will be explained in this lab!

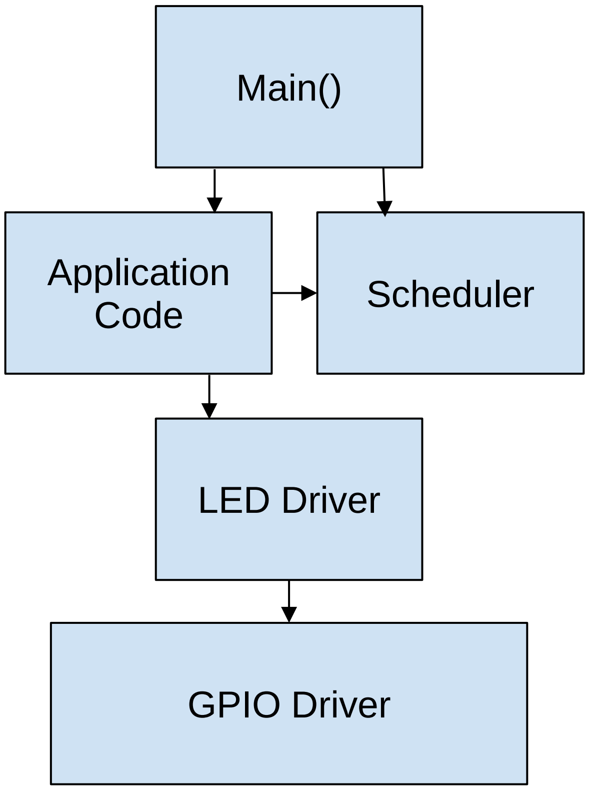
Things to Consider

- We will not dictate what you name your variables. We will dictate HOW the variables are named..
    - When naming variables, just make sure they make sense.
- Read the entire list of sub-tasks if things are not clear.
    - Remember some details may be left out intentionally. This is how we learn. Feel free to ask a classmate or an instructional staff member for a hint or to point you in the right direction. **Don't just ask for the solution.**
- Prototypes go in the header files (.h), these are the declaration of functions.
- Function definitions will go into the source files (.c).
- Data types like uint8_t, uint16_t, and uint32_t will be used more often than not.
    - Refresh on those data types if needed.

Coding Hierarchy for Lab 2

Remember, files can only directly include files they “point” to in the diagram.
For example, for the application code fileset (.c and .h file), in the applicationCode.h, there should be an include statement for “Led_Driver.h”.

The source files should only include their respective header file. So LED_Driver.c should only include LED_Driver.h while LED_Driver.h will include GPIO_Driver.h



Main() - The entry point for our program
Application Code - Will help us solve the task at hand
Scheduler - Will be responsible for which functions run and “when” this is very useful in big applications to verify that appropriate functions get ran in a certain order
LED Driver - Will abstract GPIO driver and use the GPIO driver to control LEDs
GPIO Driver- Will interface with the hardware and be responsible for register accesses