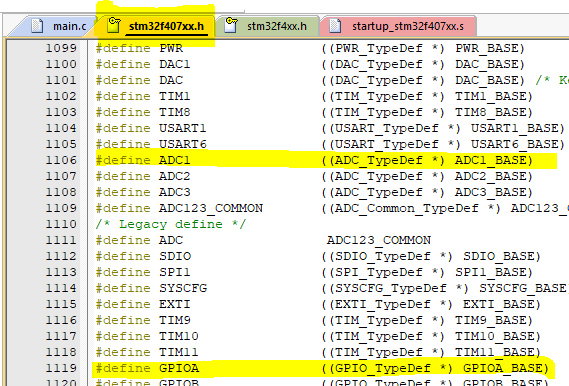
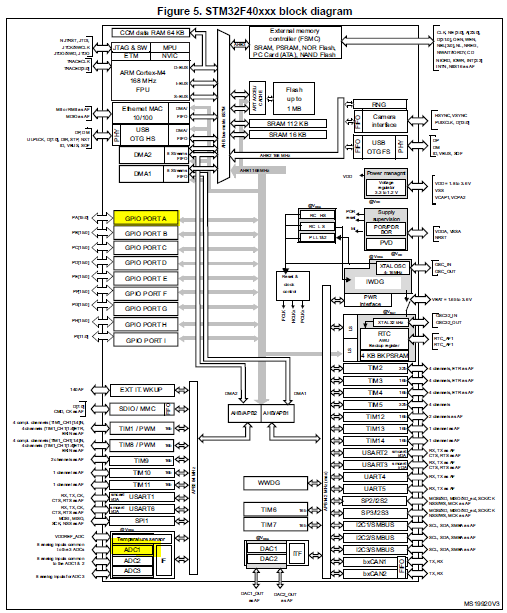
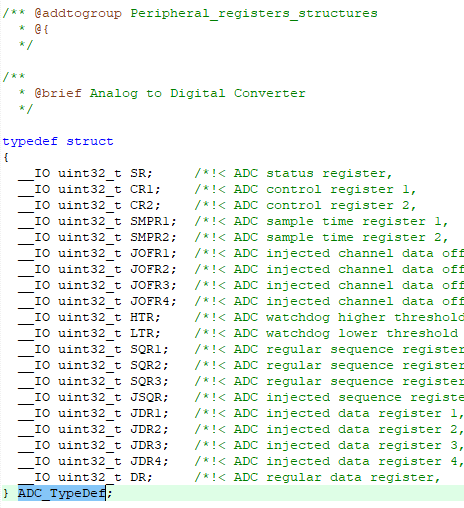
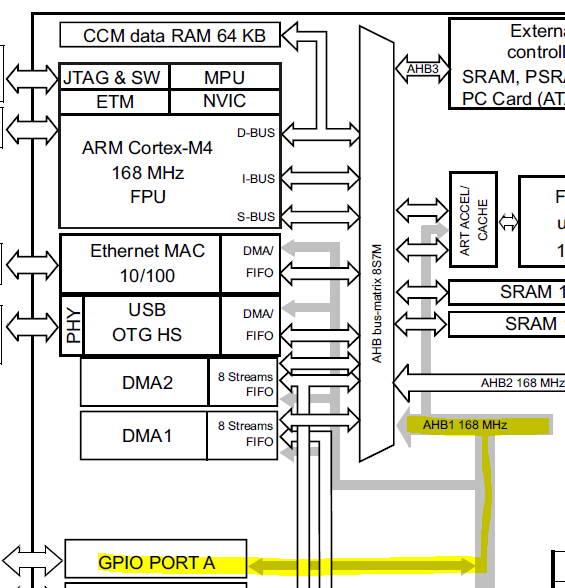
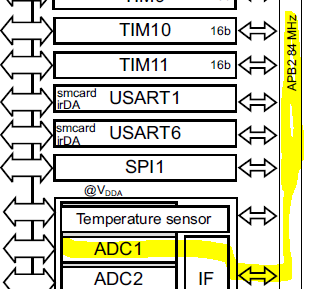
RCC Clocks and Peripheral Enables

1. BEFORE we write any code, we need some information – and we can track it through the setup files.
2. In order to use on-chip peripherals, first their corresponding clocks must be enabled.
3. Peripheral definitions are stored in the MCU header file: stm32f407xx.h Make sure it’s #include
4. Select peripherals. We choose ADC1 and GPIOA. We can find these in the block Diagram, and the Peripheral Definitions:

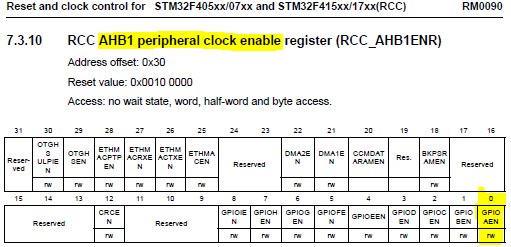
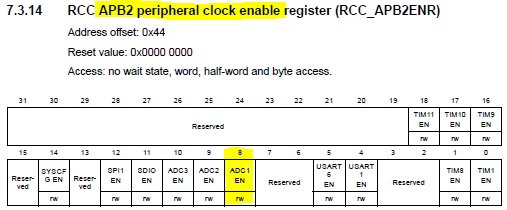
 

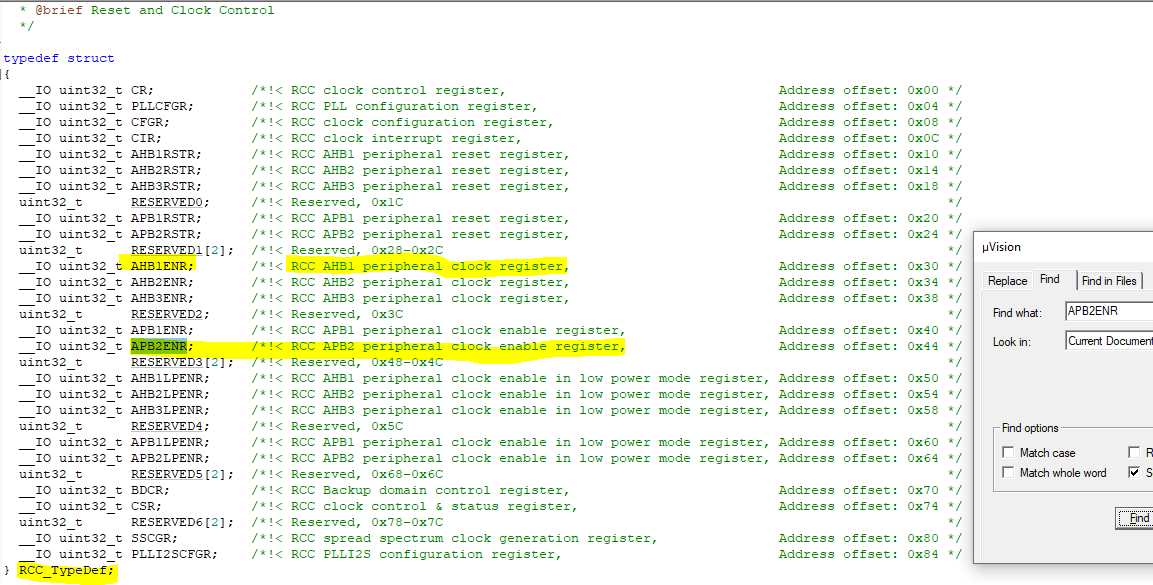
1. Just a note: The structures for these peripherals defined in the device header file are an ADDRESS eg:  
    ADC1\_BASE = (APB2PERIPH\_BASE + 0x2000UL)  
    APB2PERIPH\_BASE = (PERIPH\_BASE + 0x00010000UL)  
    PERIPH\_BASE = 0x40000000UL

With the address CAST as a POINTER of TYPE: STRUCT :  
So, I guess that means the STRUCT (ADC\_TypeDef) is now located at address ADC1\_BASE?  
  
 

1. Following the peripherals on the block diagram, we can see what clock busses they are attached to:  
      
   Therefore, clock AHB1 and clock APB2 must be enabled for peripherals GPIOA and ADC1, respectively.

|  |  |
| --- | --- |
| **To Do:** | |
|  | Use ADC 1 peripheral |
|  | 1. The peripheral definition is ADC1 |
|  | 1. Enable APB2 clock for ADC1 |
|  | 1. ? |
|  | Use GPIO Port A peripheral |
|  | 1. The peripheral definition is GPIOA |
|  | 1. Enable AHB1 clock for GPIOA |
|  | 1. ? |

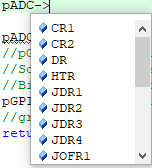
1. Tracing down how to enable the clocks:  
   According to the reference manual, searching “AHB1” we find:  
     
   Therefore, we need something to do with the RCC\_AHB1ENR register, and bit 0.   
   Similarly with ADC1:  
      
   So we are interested in APB2ENR, bit 8.
2. Ok, so now we need to figure out HOW to set those bits to 1 on those registers in order to enable our clocks. So let’s look for something referencing “\*APB2ENR\*” in the header file.

Found it, and AHB1ENR referenced in struct RCC\_TypeDef, with the comment ‘peripheral clock enable register.’ That sounds pretty good.  


|  |  |
| --- | --- |
| **To Do:** | |
|  | Use ADC 1 peripheral |
|  | 1. The peripheral definition is ADC1 of type ADC\_TypeDef |
|  | 1. Enable APB2 clock for ADC1 – **bit 0 on RCC\_APB2ENR** |
|  | 1. Use struct RCC\_TypeDef->APB2ENR to access the enable clock register and set the correct bit |
|  | Use GPIO Port A peripheral |
|  | 1. The peripheral definition is GPIOA of type GPIO\_TypeDef |
|  | 1. Enable AHB1 clock for GPIOA – **bit 8 on RCC\_AHB1ENR** |
|  | 1. Use struct RCC\_TypeDef->APB2ENR to access the enable clock register |

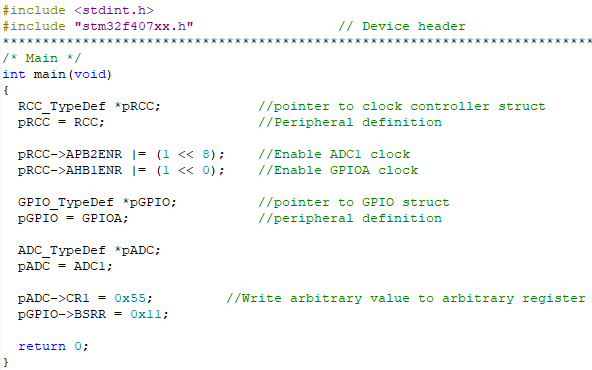
1. Writing some code  
   Let’s reorganize our priorities

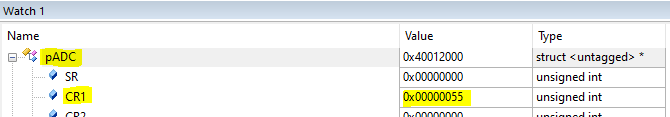
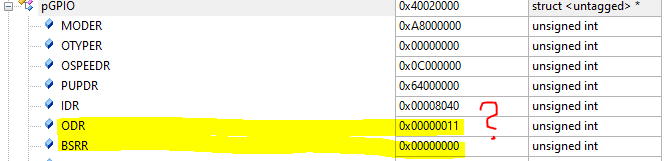
|  |  |
| --- | --- |
| **To Do:** | |
|  | Enable the peripheral Clocks |
|  | 1. Create a pointer to the clock controller structure (It’s bad form to access it directly) RCC\_TypeDef \*pRCC’ |
|  | 1. Populate the structure with the peripheral definition pRCC = RCC; |
|  | Peripherals Clock Enable: |
|  | 1. Enable the ADC1 Clock which is setting bit 8 to 1 in the APB2ENR Register. APB2ENR is part of the RCC\_TypeDef structure, assigned to peripheral RCC. We are accessing the RCC structure via our pointer pRCC. We will bit-shift in a ‘1’ into the 8-bit position of the register, OR it with the original APB2ENR register, and set the APB2ENR register = to that new value. pRCC->APB2ENR |= (1 << 8) |
|  | 1. Repeat for GPIOA: Set bit 0 in the AHB1ENR register to 1 pRCC->AHB1ENR |= (1 << 0) |
|  | 1. The peripheral clocks are now enabled. |
|  | ADC1 and GPIOA Access |
|  | 1. Create a pointer to the peripheral structures GPIO\_TypeDef \*pGPIO; ADC\_TypeDef \*pADC; |
|  | 1. Populate the structure with the peripheral definition pGPIO = GPIOA;   pADC = ADC1; |
|  | 1. Let’s write some arbitrary values to some arbitrary registers |

1. Writing Arbitrary values to arbitrary registers  
   We can access the options for registers from these structures via auto-complete or CTRL+SPACE  
     
   Lots of options. We can blindly write stuff, or we can go into the reference manual and discover a little more about these registers. Let’s start with blindly writing values to random registers.

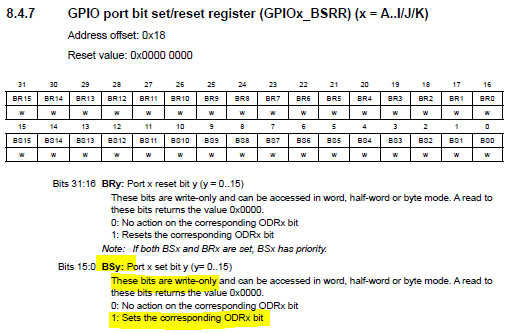
pADC ->CR1 = 0x55;

pGPIO->BSRR = 0x11;

1. Now we have complete code. Build, Load and Debug (F7, F8, Ctrl+F5)  
   

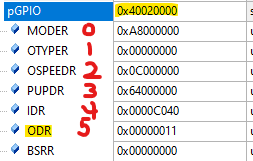
Add pADC and pGPIO to the WATCH window.   
Step through the code (F10)   
  


pADC looks good – 0x00000055 (Little Endian of 0x55)   
pGPIO looks odd… why is ODR populated with little endian 0x11, not BSRR like we said?   
TO THE REFERENCE MANUAL

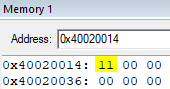


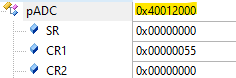
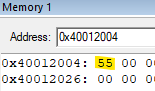
Oh. So, BSRR is write-only. And basically if we write a not-zero value to it, it will actually just set the corresponding ODR bits (in this case, the ODR register since we wrote to the BSRR register) to the value we wrote to it. So it’s working as intended.

Let’s look at the memory map to validate further

pGPIO base address is 0x40020000, and ODR is the 5th location (0, 1, 2, 3, 4, 5) 

The registers are 32-bit (4 bytes) so 5th location x 4 bytes per location = 20, so add 20 to the base address. Decimal 20 = Hex 0x14:

Address ODR = 0x40020014  


And for pADC? CR1 is the 1st location (0, 1). 1x4 = 4. 4 = 0x04 so address 0x40012004  
 

1. **Verified! Problem Solved!**