**STM32F4 교재**

2013-11500 박동훈

본 교재는 무단 배포를 금지합니다. 시그마 내에서만 Exclusive하게 사용되어야만 하는 교재입니다. 수정은 시그마 교육부장 및 교육 부장이 허락한 사람에게만 허용하며, 수정한 사람은 Revision Note에 이름, 전화번호, 이메일, 수정사항을 배포시에 기입해야 합니다.

서론

본 교재는 ‘로봇 제작을 위한 아두이노 실습’ 교재를 완료한 사람 수준에서 작성되었습니다. 아두이노 수준의 개발이 아직 어려운 분은 아두이노 개발을 조금 더 연습하고 오시기 바랍니다. 물론 시간이 넘치고 처음부터 제대로 된 개발을 원하는 분도 환영합니다. 본 교재를 읽어서 도움을 받을 수 있는 사람은 아두이노로 처리하기 힘든 고성능 제어 시스템 개발자 혹은 다중 통신 솔루션 개발을 목표로 하고 있는 사람입니다. 시작하기에 앞서 말하자면, 대부분의 아이디어는 아두이노로 구현이 가능합니다. 그러나 구현이 불가능하거나, 아두이노로 세련되게 구현할 수 없는 상황이 아두이노를 조금 다루어 본 여러분에게는 분명 있었을 것입니다. 그러한 상황을 타파하고자 하는 사람에게 이 교재의 내용은 충분히 매력적으로 다가갈 것입니다.

교재에서 다루는 내용은 크게 세 가지 부류로 나누어집니다. 첫 번째로, 공부하게 될 프로세서의 특징과 장점에 대해서 보고, 어디까지의 Application에 이 프로세서가 적당한지 이야기하며 프로세서를 개발하기 위한 주변환경 세팅을 합니다. 두 번째로, 프로세서의 기능에 대해 차근차근 공부해 나갑니다. 이 부분이 가장 긴 여정이 될 것입니다. 마지막으로, 프로세서에 같이 사용하면 좋은 라이브러리나, 추가사항들에 대해 심도 있게 설명합니다.

교재는 최대한 학습자의 하드웨어 프로그래밍 실력 증진을 위해 작성되었습니다. 따라서 최대한 개발용 보드 위에 있는 요소(스위치, 센서, 포트, LED)를 사용하여 학습할 수 있게 작성하였으나, 중간에 꼭 필요한 부분에서는 납땜을 하거나 빵판을 이용해야 하는 부분이 존재함을 미리 알립니다.

또한, 개발 툴에 대한 설명도 따로 지면을 할애하지 않았습니다. 교재를 읽어가면서, 필요한 부분에 개발 툴의 활용에 대한 이야기를 적어놓았습니다. 이러한 구성을 취한 이유는 개발 툴을 따로 공부하면 집중도가 매우 떨어지고, 어디에 써야 하는지 상당히 애매하다고 느낄 수 있기 때문입니다. 그러므로 이 교재는 기존에 ARM 경험이 있는 사람이라도, 차근차근 따라오면 디버깅이나, 소스 편집에 있어서 유용한 팁을 상당히 많이 가져갈 수 있을 것입니다.

2015년 12월 24일 크리스마스이브에

박동훈

# **목차**

1. **개발환경 설정 및 플랫폼**
   1. STM32
   2. IAR Compiler/Debugger
   3. Source Insight
   4. STM32CubeMX
   5. ST-Link
2. **STM32 개발 요소**
   1. Clock Configuration(RCC)
      1. Hardware External Clock 설계에 대하여
      2. Clock Setting with CubeMX
   2. GPIO
      1. Using Basic GPIO : Hello World!
      2. Pull-up/ Pull-down and Switch Design
   3. External Interrupt
      1. Switch Interrupt
      2. Rotary Encoder
   4. Timer
      1. Timer Interrupt
      2. PWM
         1. Control DC motor Speed with Rotary encoder
         2. Control Servo motor
   5. ADC (Analog to Digital Converter)
      1. Polling Reading
      2. Interrupt Reading
   6. Uart
      1. Conventional Uart
      2. Uart Interrupt
   7. I2C
      1. MPU6050 Gyroscope
   8. DAC
   9. FREERTOS
      1. What is task
      2. What is Queue
3. Additional Information
4. **개발환경 설정 및 플랫폼** 
   1. STM32

STM32는 32비트 마이크로 프로세서로, ARM 계열의 MCU이다. 저가형 라인업부터, 고가의 라인업까지 Cortex-M0~M7로 커버하기 때문에 제어용 프로세서로 제격이다. 아두이노와 비교하여 성능을 보면, 고성능 라인업의 경우 Floating Point 연산을 한다고 했을 때, 100배 이상의 속도를 보여준다. 또한, ARM 전용의 DSP 라이브러리가 제공되기 때문에, FFT, Kalman Filter 등 고성능이 요구되는 연산에 있어서 효과적인 솔루션이다.

이번에 이 교재에서 공부하게 될 보드는 STM32F407VG를 탑재한 Discovery Board이다. 이 보드 없이 교재를 공부하는 것은 불가능하니, 혹시나 주변에 보드가 없다면, 구매하기 바란다. 여타 EVM괴는 달리 상당히 저렴한 가격(3만원 안쪽)에 구매할 수 있다. ST사에서 직접 대량생산하여 사실상 부품값만 받고 팔고 있기 때문에, 금전적, 유통적 환경이 구하기 어렵지 않을 것이다.



* 1. IAR 컴파일러 / 디버거

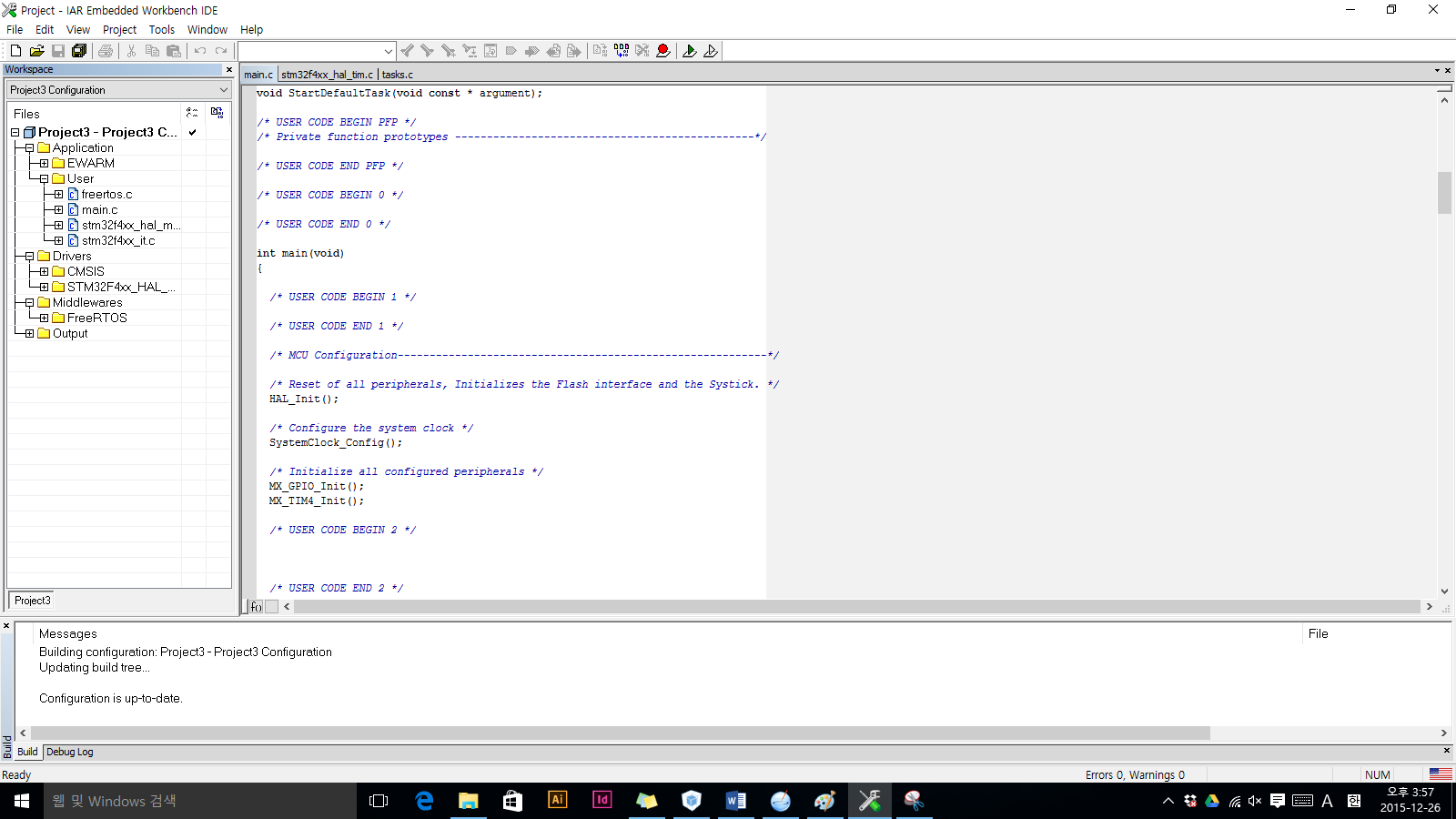
IAR은 모든 칩의 Embedded Software를 컴파일 할 수 있는 툴이라고 이야기해도 과언이 아니다. 필드 엔지니어들도 실제로 대부분 사용하는 툴이며, 강력한 디버깅 기능을 제공한다. IAR을 통해 할 수 있는 작업은, 실시간으로 변하는 변수의 모니터링, 런타임에서 변수의 조작, 원하는 지점에서의 break 등 다양하다. 하지만 매우 아쉬운 점은 인터페이스가 구시대적이고, Source를 편집하는 에디터가 메모장보다 나은 점이 그다지 없다는 점이다. 그래서 코드 에디터는 아래에 다른 것을 소개할 예정이다. 그리고 또 하나의 단점은 비싸다. 개인용 copy가 1000만원상당이다. 그러나 학구열에 불타는 여러분은 어떻게든 IAR을 준비했다고 가정하겠다.

[](https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwix-Ojh5PTJAhXCLaYKHf4mDiEQjRwIBw&url=http://kr.mouser.com/new/IAR/IARstartSTM32F103ZE/&psig=AFQjCNExChyJTmHCf2qJJsE0JUKlOINfqw&ust=1451056477452216)

IAR 실행 화면을 살펴보자.

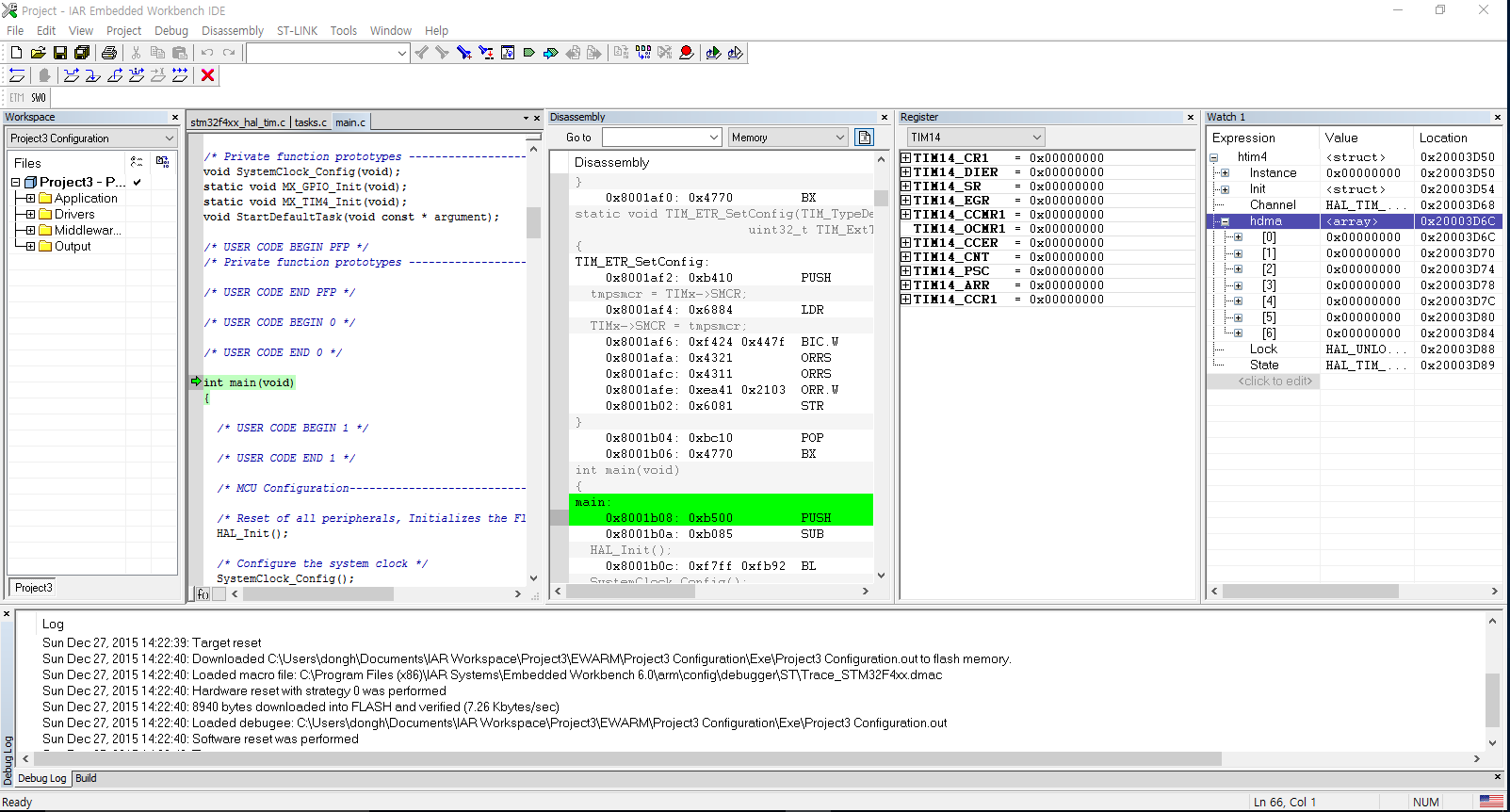


프로젝트 생성이나 라이브러리 링크는 뒤에 소개할 CubeMX가 자동으로 해주기 때문에, 큰 걱정하지 않아도 된다. 아래 내용부터는 CubeMX 프로젝트가 생성된 후의 이야기를 하겠다. D번 CubeMX 관련 내용을 읽고서 오기 바란다.



CubeMX로 생성한 프로젝트 구조이다. FreeRTOS를 include하지 않았다면 Middleware 폴더와 freertos 파일은 없는 것이 정상이다. 보드가 연결된 상태에서 상부의 초록색 삼각형 버튼을 누르면 프로그램을 업로드하고 디버그 모드로 진입한다. 디버그모드에서는 View 탭에서 어떤 것을 볼지 선택이 가능한데, 다음 페이지에 나오는 창과 같이 구성하여 사용하면 된다.

각 창의 기능을 살펴보자.



주로 사용하는 창들은 위 세 개이다. View 탭에서 눌러 추가할 수 있다.

Disassembly : 어셈블리어로 변한 소스코드를 보여준다. 실제로 기계 안에서 돌아가고 있는 코드는 어셈블리 코드이기 때문에, 디버깅할 때 꼭 같이 봐야 하는 부분이다.

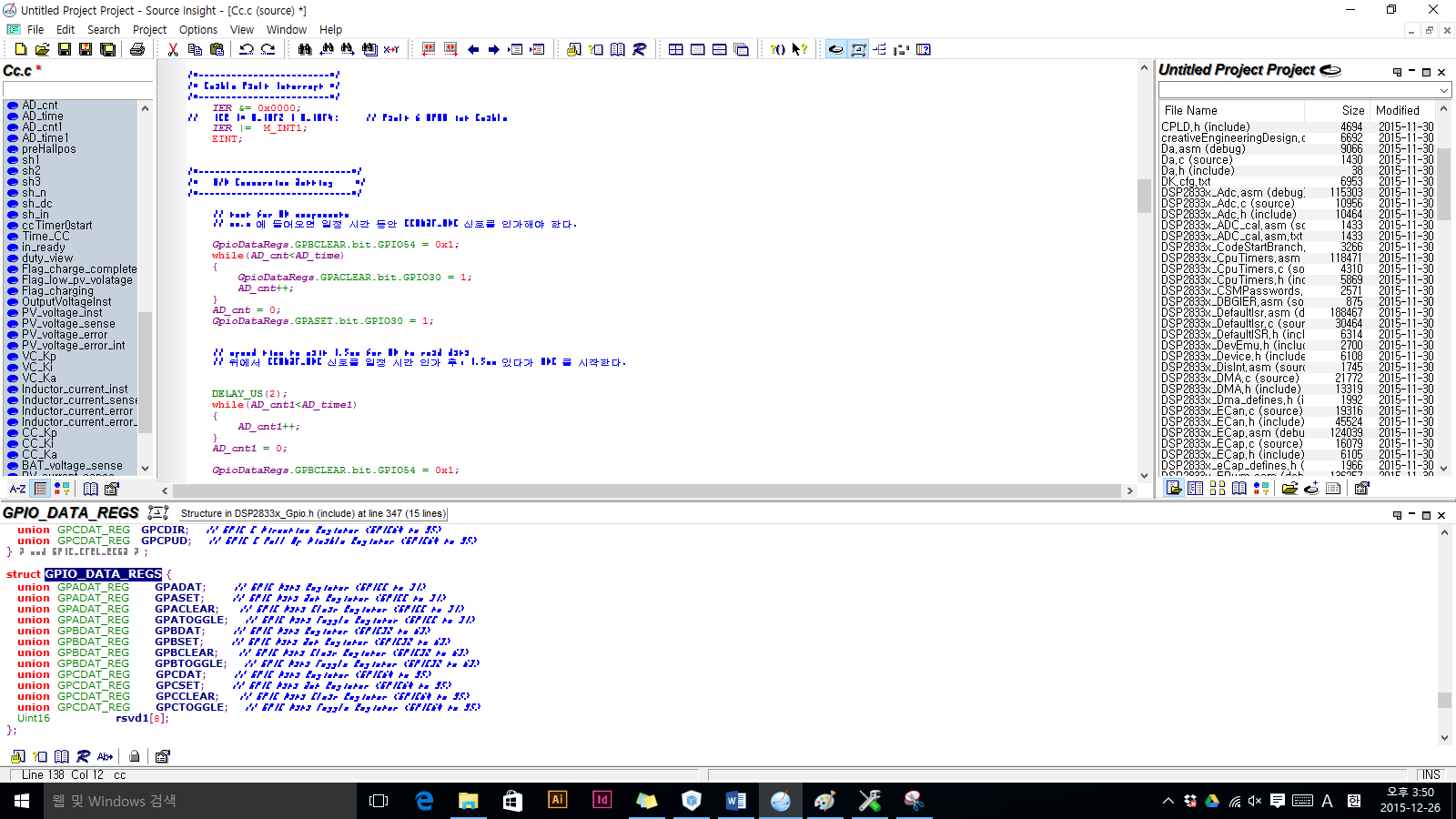
Register : CPU 내에 있는 특정 주소의 값으로, 이 값을 바꾸면 CPU의 특정 기능이 동작한다. CPU가 오동작할 때 어떤 상태인지 확인해야 하기 때문에 보고있어야 한다.

Watch : 변수들의 값을 CPU를 잠시 멈춘 상태에서 볼 수 있다.

그리고 Breakpoint기능도 제공한다. 디버그 모드에서 소스 좌측의 회색 부분을 더블클릭하면 만들어지며, 해당 지점에 갈 때마다 CPU는 동작을 멈추고 사용자의 입력을 기다린다.

* 1. Source Insight

소스인사이트는 자명하게 하드웨어 C 코드 편집기 중에서는 최강이다. 하드웨어 코드는 일반적인 코드와 다른 헤더와 구조를 많이 가지고 있는데, 여타 편집기와는 다르게 소스인사이트는 어떤 소스에 대해서든 완벽한 분석을 제공한다.



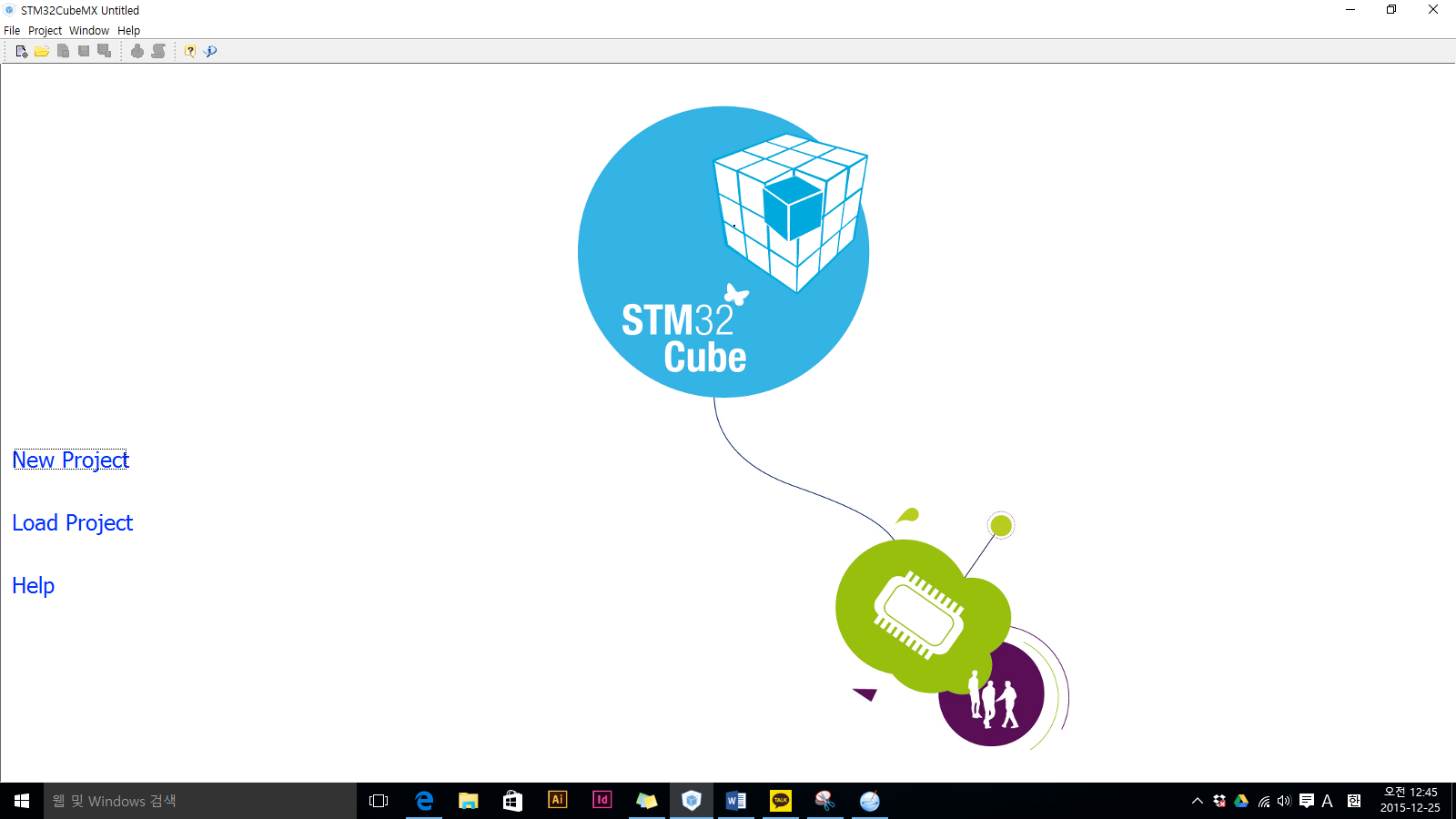
위 화면은 소스인사이트에서 하드웨어 코딩 프로젝트를 열었을 때 화면이다. 커서를 이동하면 해당 커서 위에 있는 요소의 정의를 실시간으로 찾아서 아래에 보여주는 기능을 소스인사이트는 자동으로 지원한다. (심지어 상당히 빠르다) 이는 하드웨어 프로그래머가 헤더를 들춰보는 시간을 최소화 할 수 있도록 해준다. 조금 아쉬운 점이라면 프로그램 업데이트가 수년간 거의 없다는 점과, 그로 인해 필연적으로 인터페이스가 구리다는 점이 있겠다. 소스인사이트 프로젝트 생성은 아래와 같다. CubeMX 프로젝트 생성을 가정하고 이야기하겠다.

* 1. STM32CubeMX
  2. ST-Link

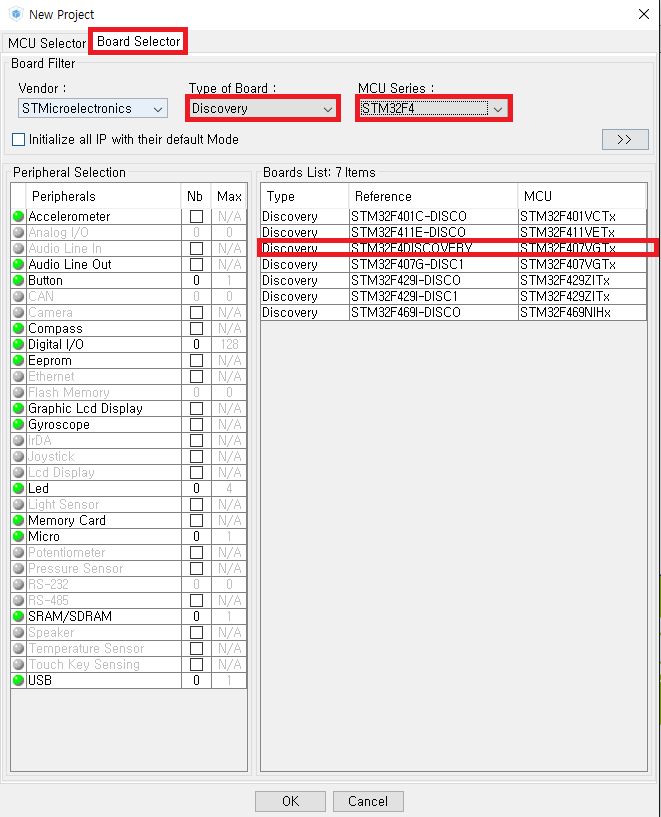
1. STM32 개발 요소
   1. Clock Configuration(RCC)

ARM 계열 Microcontroller는 아두이노와 다르게 자신의 작동 속도를 조절할 수 있다. STM32F407은 최대 168MHz까지 동작할 수 있는 MCU고, 매우 세부적으로 속도의 설정이 가능하다. 간단하게 생각하면 최대 작동 속도가 제일 좋은 것이 아닌가 할 수 있지만, 빠른 반응을 요하지 않고, 배터리가 연결된 솔루션이라면, 낮은 속도로 동작하는 편이 전력을 아끼기 때문에 더 효과적이다.

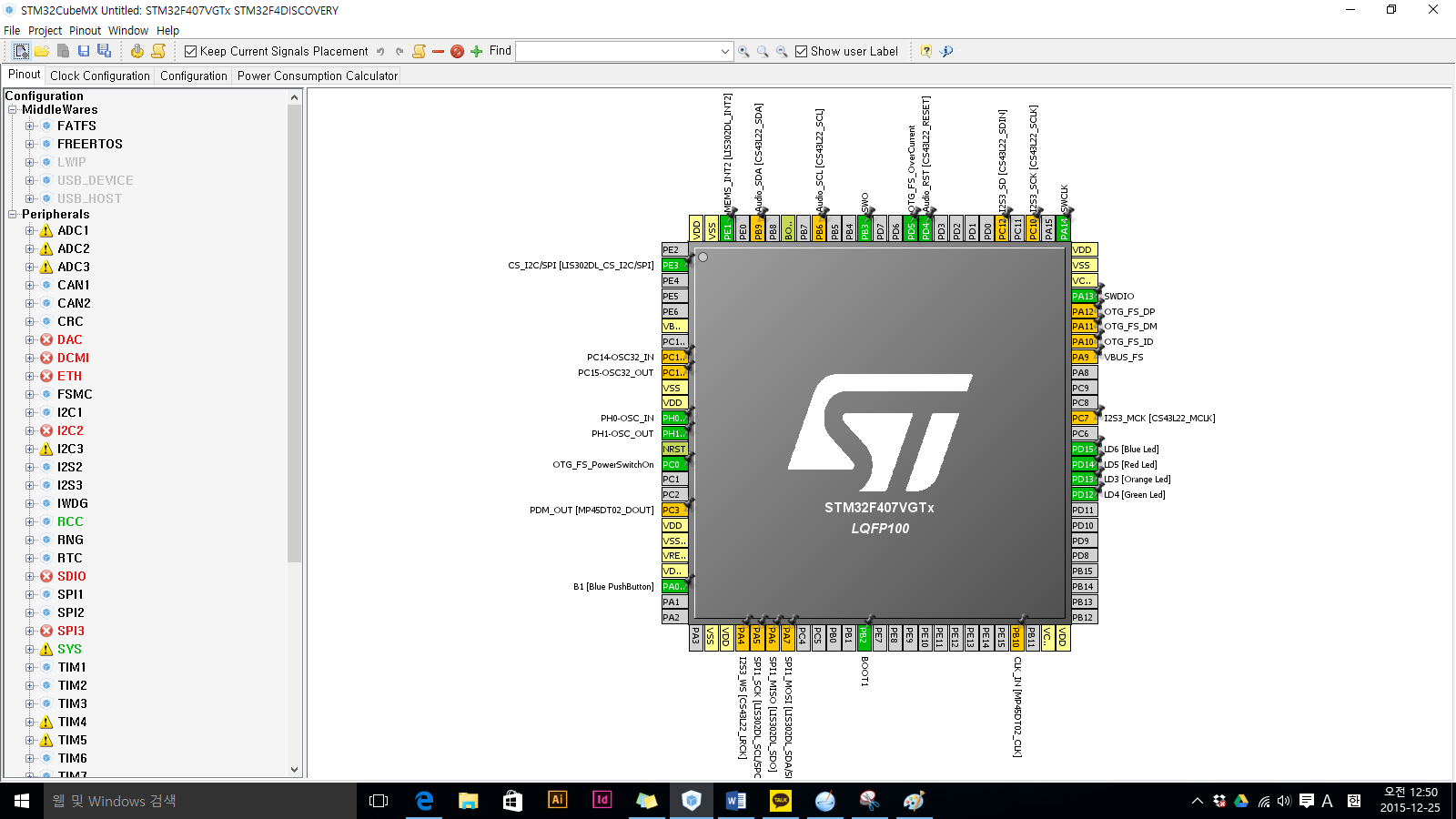
먼저, STM32CubeMX를 실행하여보자.



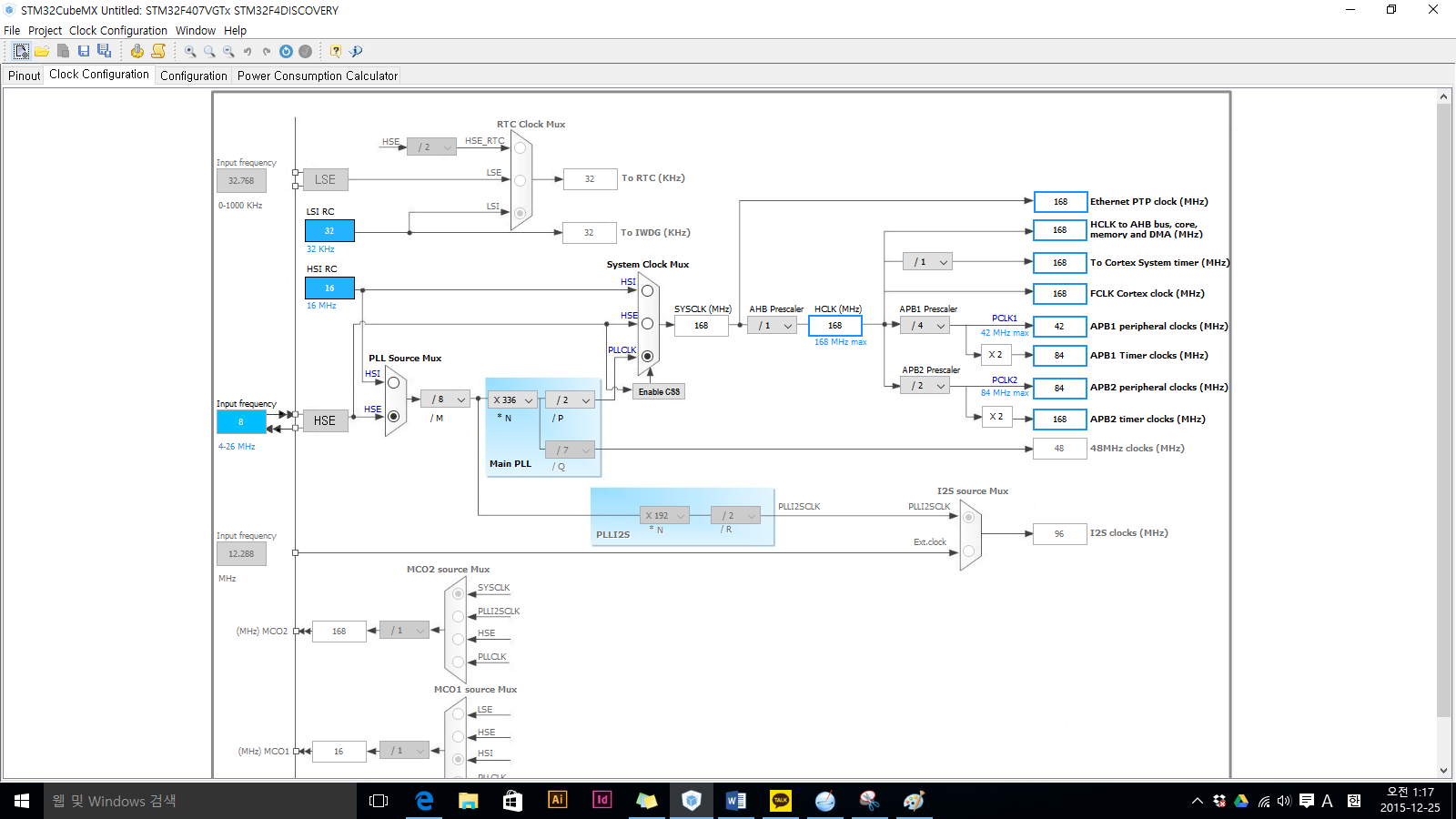
위 그림과 같은 창에서 New Project를 실행한다.



Board Selector를 클릭한다. (나중에 자신이 만든 커스텀 보드는 알맞은 MCU를 선택한 후 진행하면 된다.) STM32F4DISCOVERY라 나와있는 것을 선택하고, OK를 누르면, 아래와 같은 창으로 이동한다.



우측에는 MCU Package가 있고, 좌측에는 기능요소들이 정렬되어 있다. 상단에는 탭으로 Pinout, Clock Configuration, Configuration, Power Consumption Calculator가 있다. 여기서 Clock Configuration으로 들어가보자.



사실 보드를 선택한 순간 보드에 알맞은 세팅이 자동으로 이미 되었기에, 우리가 건드릴 만한 부분은 없다. 각 요소들에 대한 설명만 하고 넘어가겠다.

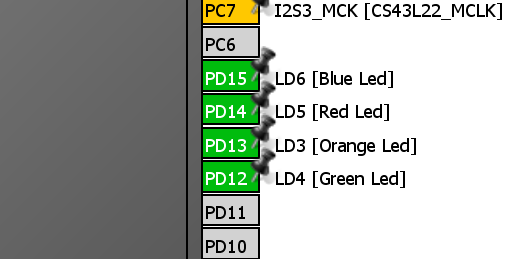
좌측 맨 끝에 파란색으로 표시된 것이 외부 Oscillator의 Clock이다. Discovery Board 위를 보면 은색으로 된 비교적 큰 소자가 위쪽의 작은 칩 주변에 하나, 아래쪽의 큰 칩 주변에 하나 총 두개 있는 것을 볼 수 있다. 그리고 그 소자 위에는 희미하게 8.000 이라는 숫자가 적혀 있다. 이 Oscillator가 8MHz로 동작함을 의미하는 숫자이다.

직접 만든 보드의 클럭을 설계한다면, 해당 보드에 탑재된 Oscillator의 Clock을 Input Frequency 칸에 적으면 된다. 중간에 배율 등의 값은 너무 크거나, 작으면 소프트웨어가 알아서 경고를 해주니 걱정하지 않아도 된다. 최종적으로 각 기능 요소에 들어가는 Clock 값은 제일 우측에 있는 값이다. APB1 타이머와 APB2 타이머의 클럭이 다름에 주의하자.

해당 타이머 클럭은 나중에 타이머 요소를 다룰 때 매우 중요하게 사용되기 때문에 기억해 두는 것이 좋다.

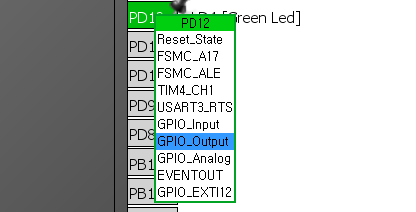
* 1. GPIO
     1. Using Basic GPIO : Hello World!

이제 제대로 된 첫 번째 코드를 돌려보자. 프로젝트 기본 설정 전체 과정도 같이 진행하고, 이후엔 다시 설명하지 않는다. 2-A에서 나온 것과 같이 세팅 후 다시 Pinout 탭으로 들어가자. 여기에서 LED에 연결되어있는 4개의 핀을 보자.

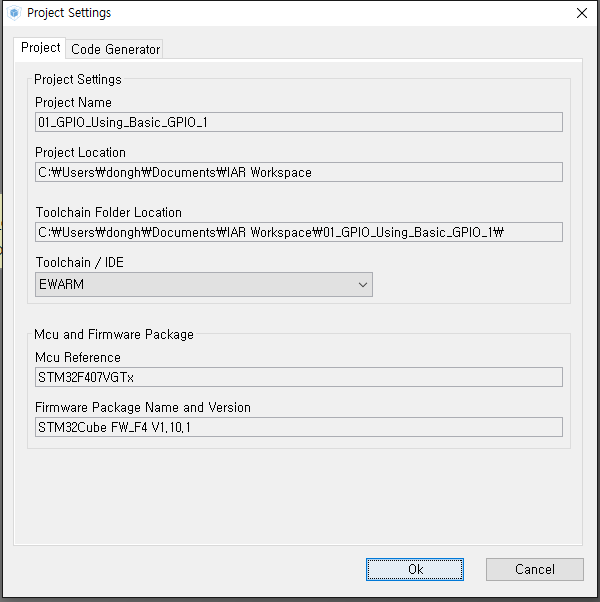


PD12,13,14,15번 핀이 각자 LED에 연결된 것을 확인할 수 있다.

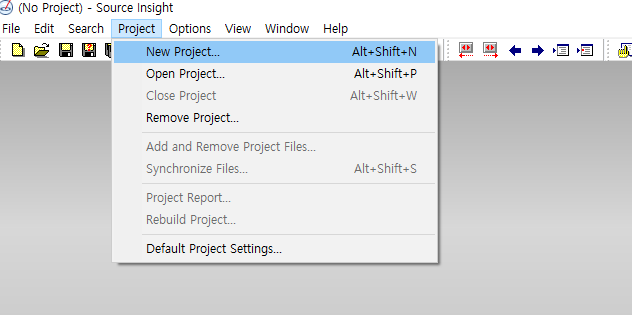
여기서 PD12번을 더블클릭해보자. 아래와 같이 기능을 선택할 수 있는 것을 확인할 수 있다. 여기서 GPIO\_Output으로 설정해주고 나면 해당 핀을 사용하기 위한 설정은 끝난다. (사실 보드를 선택하면 LED핀들의 기본 설정을 GPIO\_Output으로 해주기 때문에 여기서도 추가적인 설정이 필요하진 않다.)



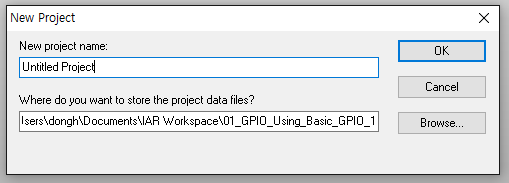
이제 톱니바퀴모양 버튼을 눌러 프로젝트를 생성해보자. IDE를 EWARM으로 하고, 프로젝트 경로에 한글이 없게 하는 것에 주의하자.



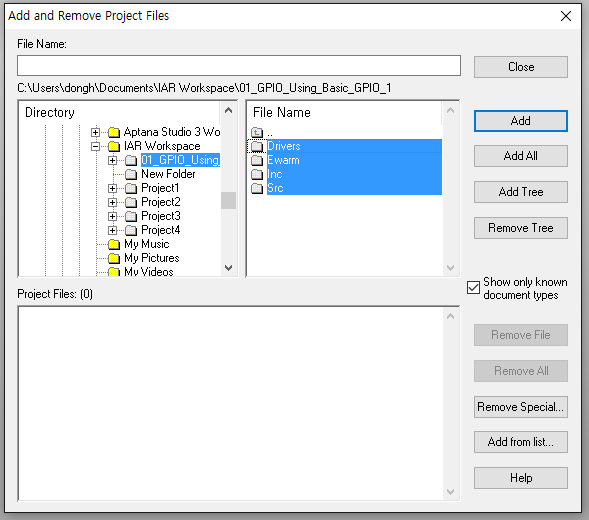
프로젝트를 생성하고, 소스인사이트 프로젝트를 생성하여 코드 작성을 위한 환경을 만들어주자.

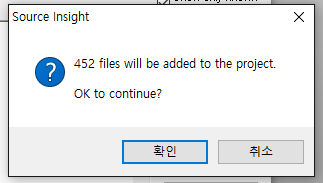


프로젝트 경로를 CubeMX로 만든 폴더로 해주는게 좋다.

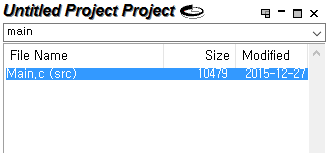
OK

OK

폴더 전체 선택후 Add Tree

확인

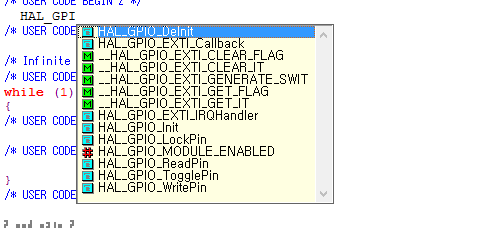
우측의 검색 창에서 main을 찾아서 두번클릭하면 열린다.



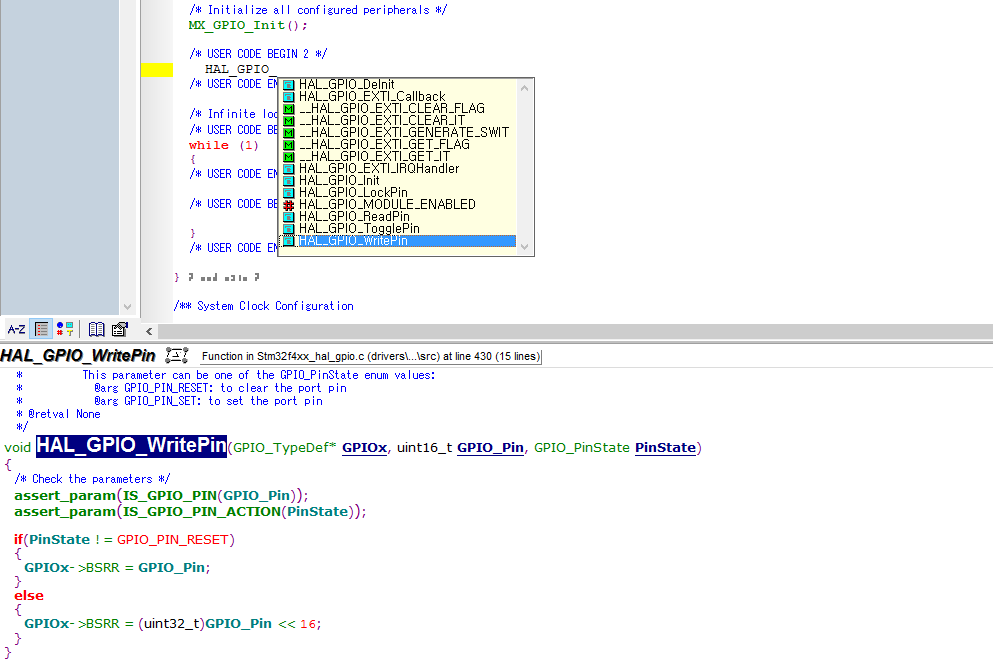
소스인사이트가 전체 프로젝트를 읽는데 시간이 약간 걸리므로 3분정도 쉰 후 작업을 진행하면 좋다. 이제 main 안쪽에 USER CODE BEGIN 2 라고 주석으로 써진 부분 안쪽에 GPIO를 컨트롤하는 코드를 짜보자.

HAL\_GPIO\_

를 소스인사이트에 적어보자. 그러면 아래와 같이 연관 코드들이 쭉 나온다. (참고로 HAL은 Hardware Abstraction Layer의 줄임 말이다.)



Initialization code는 이미 다 만들어져 있는 상태이기 때문에 사용하는 함수만 보면 되겠다. GPIO에 값을 주는 함수는 HAL\_GPIO\_WritePin으로 추정된다. 여기에 커서를 갖다 놓아보면, 아래에 이 함수에 대한 정보가 같이 나온다.



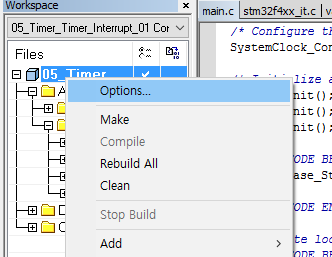
여기서 Enter를 누르면 코드가 자동으로 완성된다.

이제 각각의 Parameter를 채워야 하는데, PD12번을 켠다는 가정 하에 설명하겠다. PD12는 Port D 12 PIN의 약자로, GPIOD, GPIO\_PIN\_12를 사용한다. 그리고 GPIO\_PinState는 GPIO\_PIN\_SET과 GPIO\_PIN\_RESET으로 구분되는데, SET은 GPIO를 ON, RESET은 GPIO를 OFF한다. 따라서 PD12를 켜고 싶으면 아래와 같이 적으면 된다.

1. HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12 ,GPIO\_PIN\_SET);

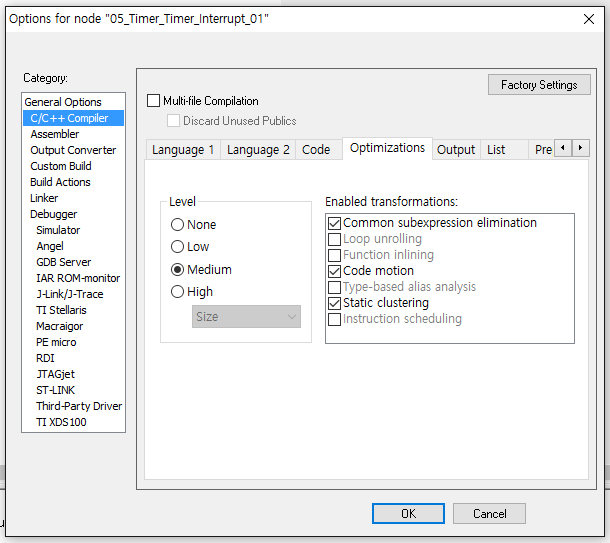
이 코드를 컴파일하고 올려보면, 보드 위의 초록색 LED가 켜지는 것을 확인할 수 있다.

여기서 컴파일이 잘 안 되는 경우가 있는데, 이는 IAR의 컴파일러가 최적화의 과정에서 HAL함수를 잘 읽지 못 하는 경우가 있기 때문이다. 이를 방지하기 위해 컴파일러의 최적화 정도를 낮춰준다.



옵션에 들어가서

다음 페이지에 계속 ->

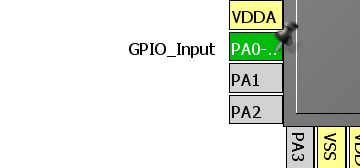
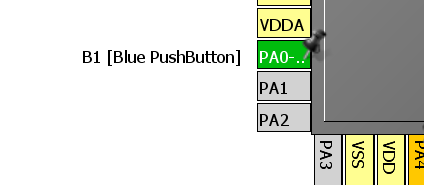


‘C/C ++ Compiler’ 탭에서 ‘Optimizations’ 항목에 있는 ‘Level’을 ‘High’에서 ‘Low’로 낮춰준다.

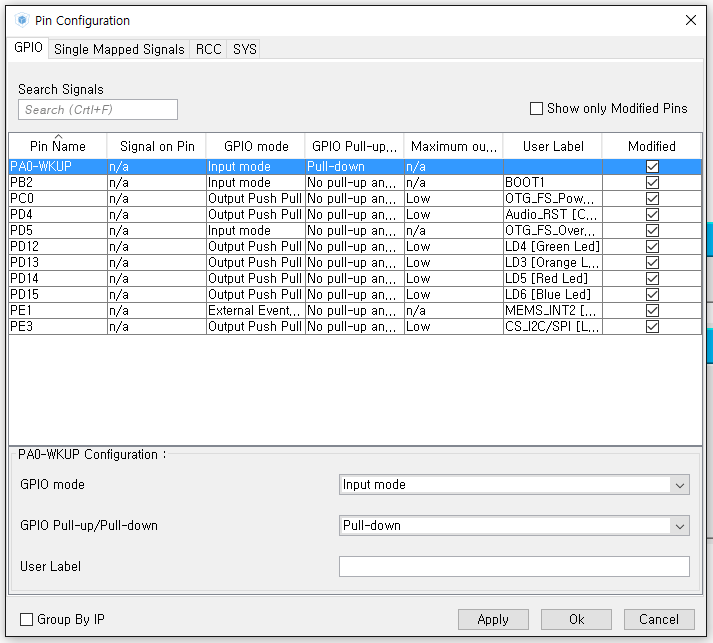
이제 아두이노에서 처음 했던 것 같이 보드 위의 LED를 1초 주기로 깜빡여보자. Delay함수는 HAL\_Delay(ms) 이다. 아래와 같이 적은 코드를 main 코드 안의 while 문 안쪽에 삽입하고 실행하면 1초 주기로 점멸하는 LED를 볼 수 있다.

1. while (1) { /\* USER CODE END WHILE \*/
2. HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, GPIO\_PIN\_SET);
3. HAL\_Delay(500);
4. HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, GPIO\_PIN\_RESET);
5. HAL\_Delay(500); /\* USER CODE BEGIN 3 \*/
6. }
   * 1. Pull-up/ Pull-down and Switch Design

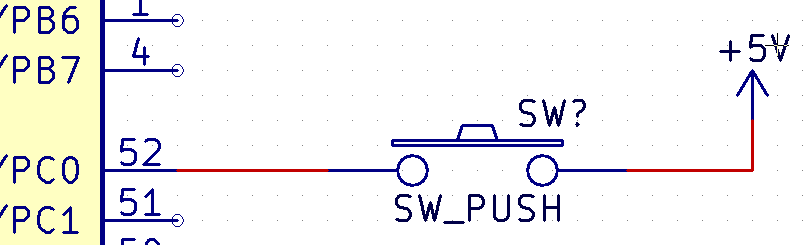
이제 GPIO INPUT을 실험해보자. 보드 위의 파란색 버튼은 PA0와 연결되어 있다. PA0는 기본적으로 Interrupt에 연결되어 있는데, GPIO\_INPUT으로 다시 세팅해보자.



그리고 configuration 탭에 들어가서 GPIO 세팅에서 PA0를 Pull-down 시켜주자.



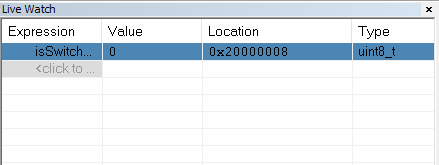
Pull-down이란, 아무것도 연결되지 않은 상태에서의 기본 GPIO 값을 0으로 세팅해주는 것이다. 왜 이러한 작업을 해야 되는지 이해하기 위해서 아래 회로를 보자.



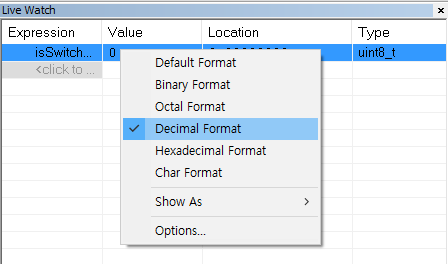
위 회로처럼 구성을 하면 스위치가 눌리지 않은 상태에서 PC0는 아무 곳에도 연결되지 않은 상태이기 때문에, 원래대로라면 입력 값이 이상한 값으로 나온다. 그러나 프로세서 내부에서 Pull-down을 시켜주면 기본값을 0으로 지정해주는 것이기 때문에, 스위치가 눌리지 않은 상태에서는 0 값이 들어온다. 아래와 같은 코드를 짜고, 돌려보자. 아래 코드는 계속해서 스위치를 감시하다가, 스위치가 눌리면, isSwitchPressed를 1로 만들어준다.

1. uint8\_t isSwitchPressed = 0;
2. int main(void) {
3. HAL\_Init(); /\* Configure the system clock \*/
4. SystemClock\_Config(); /\* Initialize all configured peripherals \*/
5. MX\_GPIO\_Init();
6. while (1) { /\* USER CODE END WHILE \*/
7. if (GPIO\_PIN\_SET == HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_0)) {
8. isSwitchPressed = 1;
9. } else {
10. isSwitchPressed = 0;
11. } /\* USER CODE BEGIN 3 \*/
12. } /\* USER CODE END 3

Live Watch 기능을 이용하여 스위치 변수가 변하는 것을 보자. 프로그램을 실행시킨 후 View 탭에서 Live Watch를 선택하여 창을 띄우자. 그리고 Expression 탭에 변수 이름을 입력하면 IAR이 자동적으로 변수를 추적한다. 아래는 정상적으로 실행되었을 때 화면이다.



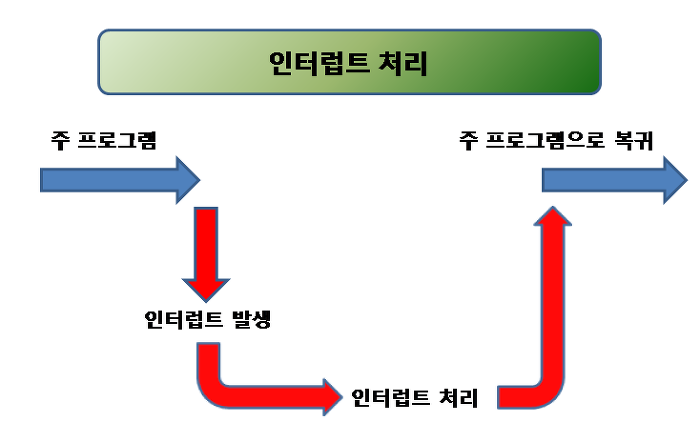
우클릭하여 옵션을 클릭하면, 변수 업데이트 주기 등을 설정할 수 있다. 그러나 너무 많은 변수가 Live Watch에 있는 경우 프로그램의 실행이 느려질 수 있으니 주의하기 바란다. 그리고 너무 짧은 주기로 변수가 업데이트되게 설정해 놓아도 프로그램 실행을 느리게 하는 요인이 될 수 있기 때문에 주의 바란다. 또한 우클릭하여 나오는 내용 중 Value를 표시하는 Format을 설정하는 부분도 있으니 각 변수를 적절한 형태로 볼 수 있다.



위 선택은 Decimal(10진수) 형태로 표시형식을 설정하는 것이다.

* 1. External Interrupt
     1. Switch Interrupt

이번엔 스위치로 EXTI 인터럽트를 받아보자. EXTI 인터럽트는 외부 스위치 전압 레벨 변화를 이용해 특정 상황에서 특정 작업을 하게 만들어주는 기능이다.

[](https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwiol-HV5PvJAhXFJqYKHVJPA2wQjRwIBw&url=http://www.hardcopyworld.com/ngine/aduino/index.php/archives/594&psig=AFQjCNGtPITpiaOnlP-0_OpFiTXQbOd47A&ust=1451296967203999)

인터럽트가 발생하면 원래 돌아가고 있던 프로그램은 잠시 실행을 중단하고, 인터럽트에 등록된 작업을 처리한 후 다시 복귀한다. 인터럽트가 발생할 수 있는 조건들은 상당히 많은데, 그 중 많이 사용하는 조건들은 아래와 같다.

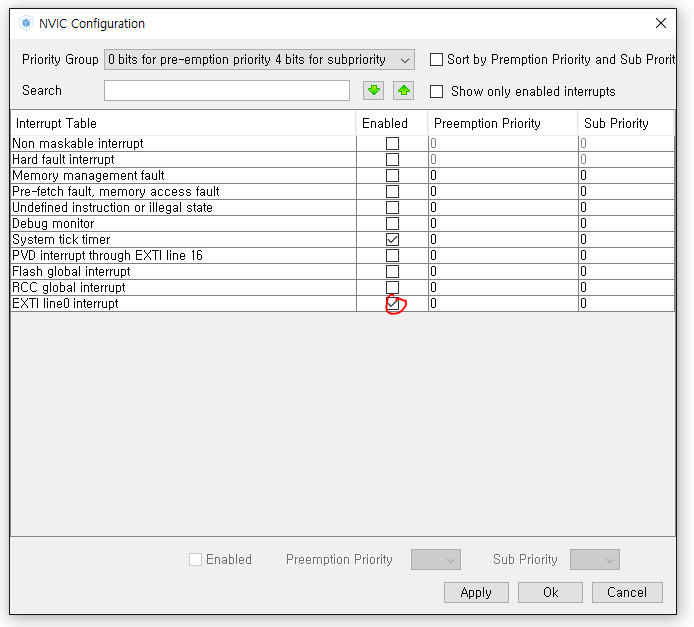
* 전압 레벨의 변화 : 어떤 핀의 전압이 0V 에서 3.3V 혹은 3.3V에서 0V 로 변할 때 그 변화 타이밍을 감지한다. 스위치, 엔코더 등은 이러한 인터럽트를 필수적으로 사용한다.
* 타이머 오버플로우 : 타이머 오버플로우를 이용하면 일정 주기로 어떤 작업을 수행하도록 등록할 수 있다. 예를 들면 2ms마다 한번씩 스위치의 값을 읽어와서 저장하는 등의 동작 수행이 가능하다.
* 타이머 Compare Match : 타이머 카운터가 특정 값에 도달하면 발생하는 인터럽트이다. Software PWM을 만들 때 오버플로우 인터럽트와 같이 많이 사용된다.
* 통신 RX : 통신 데이터가 들어온 순간 발생하는 인터럽트로, 데이터를 빠르게 읽어오기 위해서 필수적인 인터럽트이다.

이 외에도 자주 사용하는 인터럽트 종류는 많지만 그것들은 뒤에 소개하겠다. 이번 장에서 소개할 EXTI 인터럽트는 위의 유형 중에서 전압 레벨의 변화를 감지하는 인터럽트이다. 전압 레벨의 변화가 일어나면 인터럽트가 발생했을 때 하도록 등록한 일(ISR : Interrupt Service Routine)이 실행되는데, ISR을 어떻게 등록하는지 등에 대해 이번 장에서 이야기하도록 하겠다.

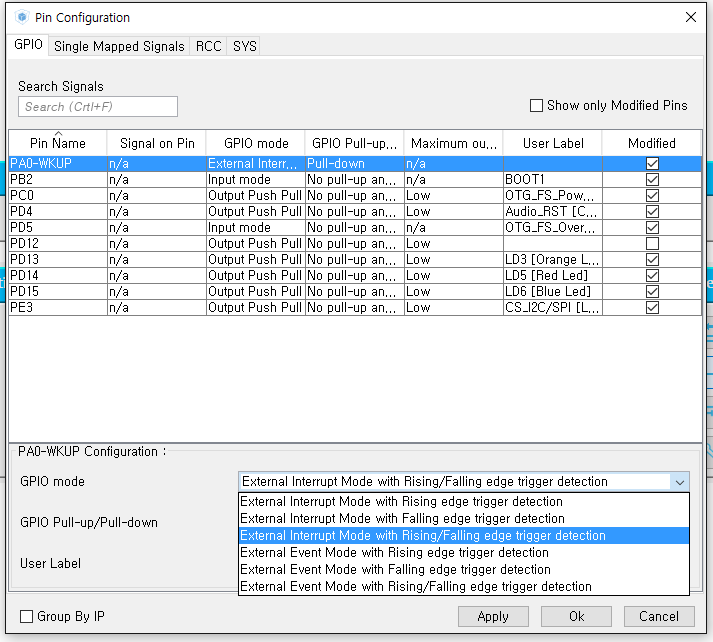
어떤 핀에 대해 EXTI 인터럽트를 등록하는 방법은 앞서 GPIO 설정을 했던 것과 동일하다.



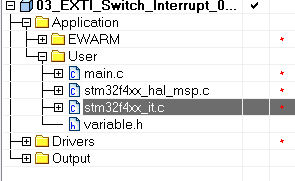
핀을 클릭하고, EXTI로 설정해주면 된다. 그리고, Configuration 탭으로 들어가서 NVIC를 눌러 들어간 후 EXTI0 인터럽트를 Enable한다.



그리고 GPIO를 눌러 어떤 상황에서 인터럽트를 발생시킬 것인지 선택한다.



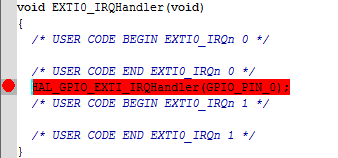
Rising Edge로 설정하면 전압 상승시, Falling Edge로 설정하면 전압 하강시에 인터럽트가 발생한다. Rising 상황에서 인터럽트가 발생하게 설정한 뒤 프로젝트를 생성하자. 그리고 프로젝트를 보면 아래와 같이 있는 stm32f4xx\_it.c file을 열어보자. 여기가 Interrupt Service Routine을 정의하는 부분이다.



열어보면 EXTI0 인터럽트가 정의되어 있다.

1. void EXTI0\_IRQHandler(void) {
2. /\* USER CODE BEGIN EXTI0\_IRQn 0 \*/
3. /\* USER CODE END EXTI0\_IRQn 0 \*/
4. HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_0);
5. }

이제 스위치가 눌리면 EXTI0\_IRQHandler 함수로 언제나 진입하도록 인터럽트가 설정된 것이다. 아래와 같이 Breakpoint를 잡아두고, 이 함수로 제대로 진입하는지 확인해보자.



제대로 세팅했다면, 스위치를 누를 때마다 인터럽트로 진입하여 breakpoint에 걸리는 것을 확인할 수 있을 것이다.

**(Tip)** 인터럽트 정의 파일과 메인 파일은 분리된 C code로 이루어져 있다. 보통 인터럽트 구문을 짤 때 인터럽트에서 조작하는 변수와, main 안에서 조작하는 변수를 공유해야 하는 상황이 많이 발생하는데, 처음 가장 많이 하는 실수 중 하나는 main 혹은 it 둘 중 하나에 변수를 선언하거나 Interrupt Service Routine 자체를 main 안에 통합해 버리는 것이다. 이런 방식으로 코드를 짜면 필연적으로 코드가 스파게티가 될 수 밖에 없다. 따라서 여기서는 **extern** 이라는 특별한 키워드를 이용하여 말괄량이 같은 전역 변수를 그나마 관리하는 방법에 대해 이야기해보고자 한다.

전역 변수를 어떤 헤더 파일 하나와, 어떤 C 파일 하나로 관리하고, 프로젝트의 어떤 파일에서, 헤더 파일만 가져오면 변수를 사용할 수 있는 것을 목표로 코드를 짜보자. 그렇다면 전역변수는 헤더 파일에 선언이 되어 있어야 다른 파일에서 사용할 수 있을 것이다. 그런데, 헤더 파일에 전역 변수를 선언해 놓으면 문제가 생긴다. 이 헤더 파일은 두 군데 이상의 소스에서 include 될 수 없다. 변수 이름이 중첩되기 때문이다. 이것을 피해가기 위한 키워드가 extern이다. 헤더 파일에 extern으로 선언된 변수는 그 이름이 컴파일러에게 알려지지만, 당장 그 자리가 할당되지는 않는다. 따라서 여러 파일에 include 되어도 전혀 문제가 없다.

그러면 이제 이 extern 키워드로 선언된 변수의 자리는 variable.c 에서 할당해주면 아무런 문제가 없다. 아래와 같이 두개의 파일을 만들고, 하나는 Inc 폴더에, 하나는 Src 폴더에 넣어준다.

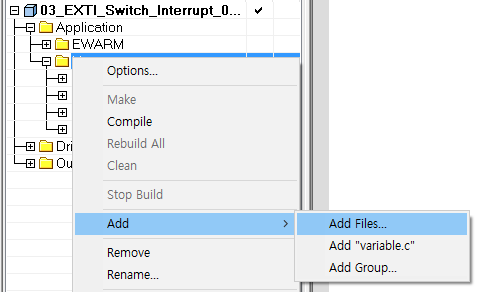
variable.h -> Inc

1. #ifndef VARIABLE\_H\_
2. #define VARIABLE\_H\_
3. extern int interruptOccur;
4. #endif

variable.c -> Src

1. #include "variable.h"
2. int interruptOccur = 0;

이렇게 한 뒤 IAR에서 variable.c file을 추가해준다.

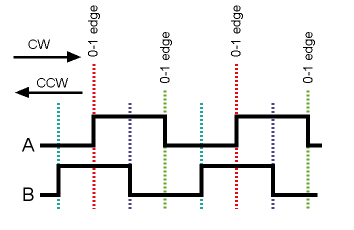


Add Files를 클릭하여 variable.c를 추가해주면 된다.

SourceInsight 에서도 Project->Add and remove project files 에서 variable.c 와 variable.h 를 추가해주면 된다. 이렇게 하면 interruptOccur이라는 변수를 모든 파일에서 variable.h 를 불러오면 사용할 수 있도록 만들어준 것이다.

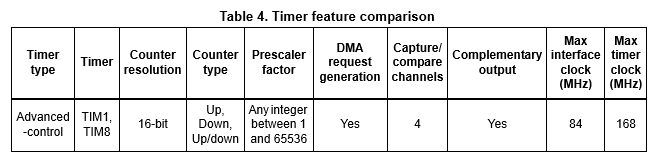
* + 1. Rotary Encoder

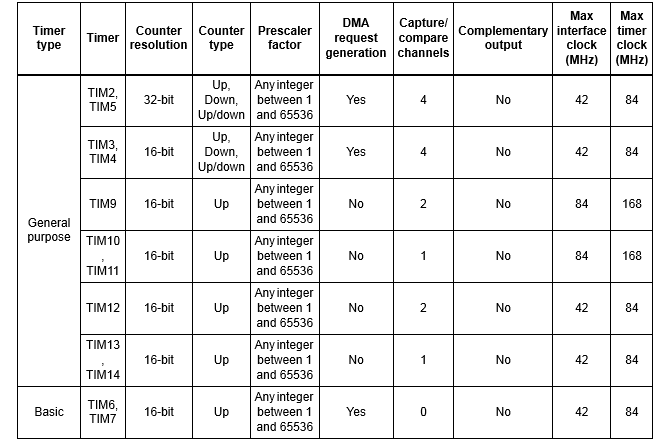
앞서 이야기한 EXTI 인터럽트, GPIO를 이용하면 Rotary Encoder Library를 제작할 수 있다. Rotary Encoder는 아래와 같은 동작을 보이는 센서이다. 각 Edge에서 인터럽트가 일어나고, 그 때 마다 엔코더 회전 값을 update 하도록 라이브러리를 작성하면 된다. 라이브러리에서 GPIO Hardware Abstraction Layer를 사용해야 하기 때문에 이러한 부분들을 include해서 사용하면 된다.

[](https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwj1iPDBh_zJAhVHJKYKHe22BSsQjRwIBw&url=http://playground.arduino.cc/Main/RotaryEncoders&psig=AFQjCNET4TKoZYQh0i8-4C4nNpgbP6mTJg&ust=1451306325853520)

* 1. Timer
     1. Timer Interrupt

타이머 인터럽트는 인터럽트 함수를 등록하여 특정 주기마다 어떤 일을 하도록 만드는 것이다. 타이머 인터럽트는 타이머 설정을 선행해야 하며 이 때 타이머에 인가되는 클럭부터 신경 쓸 필요가 있다. STM32F4 데이터시트를 참고하면, STM32F4에서 동작하는 타이머들은 아래와 같은 성질을 가지고 있다. 타이머에 들어가는 클럭은 기본 클럭 세팅에서 Max timer clock (MHz)로 맞추어져 있다. 예를 들어서 기본 클럭 세팅에서 TIM1은 168MHz로 동작한다. TIM12는 84MHz로 동작한다.



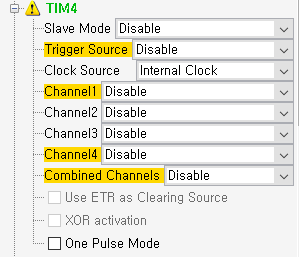


그러면 이제 Prescaler와 Max Counter를 선택할 필요가 있다. 타이머는 이 두가지 요소를 이해해야 제대로 사용할 수 있기에 예를 들어서 해당 변수가 무엇을 이야기하는지 설명하여 보겠다.

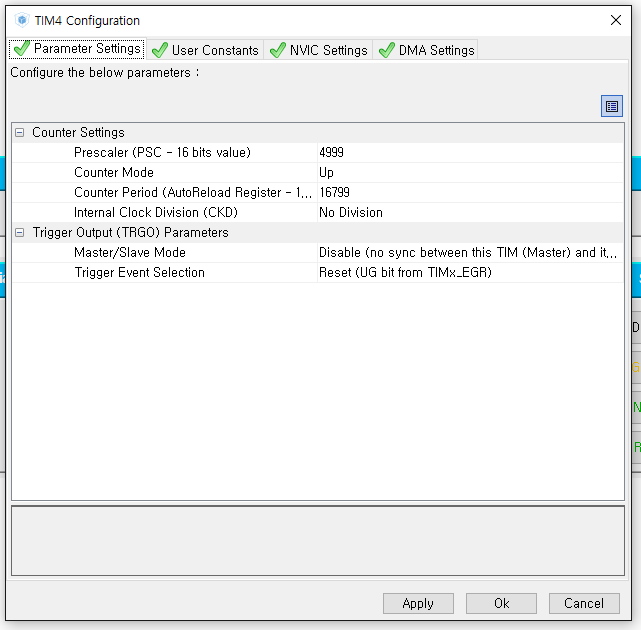
먼저, 타이머가 박수소리에 맞추어 움직이는 기계라고 생각을 해보자. 타이머는 박수소리가 들릴 때 마다 손가락으로 숫자를 하나씩 올려가며 센다. 예를 들어 박수가 1초에 1번 들릴 때, 타이머는 10초간 손가락을 하나씩 올린다. 그러다가 손가락이 더 이상 남지 않으면 타이머는 Overflow Interrupt를 발생시키고 다시 손가락을 처음부터 센다. 위와 같은 상황에서 타이머는 10초에 한번씩 인터럽트를 발생시킬 것이다. 여기서 손가락은 실제로는 Counter, 박수소리는 Clock이다.

그런데, 이렇게 적용시키면 문제가 생긴다. 만약 1초 주기의 인터럽트를 발생시키고 싶은 상황이라고 생각해보자. TIM1의 클럭은 168MHz이다. 1초 주기의 인터럽트를 발생시키고 싶다면 손가락이 168000000개 있어야 하는 상황이다. 그러나, TIM1은 16비트 타이머로, 2^16(=65536)개의 손가락 밖에 가지고 있지 않다. 이런 문제 때문에 고안해 낸 모듈이 prescaler이다. Prescaler는 바깥에서 들려오는 박수소리를 분할해준다. 예를 들어 prescaler에 밖에서 박수소리가 5000번 들릴 때 박수를 한번 치라는 명령을 주어 놓고, 타이머한테는 prescaler의 박수소리를 기준으로 동작하게 하면, 타이머는 원래보다 5000배 느린 박수소리를 듣게 된다. 따라서 1초 주기의 인터럽트를 발생시키고 싶을 때 필요한 손가락의 개수가 33600개로 줄어든다. 이제 65536개보다 적은 손가락이 필요하기 때문에 타이머 세팅을 제대로 할 수 있다.

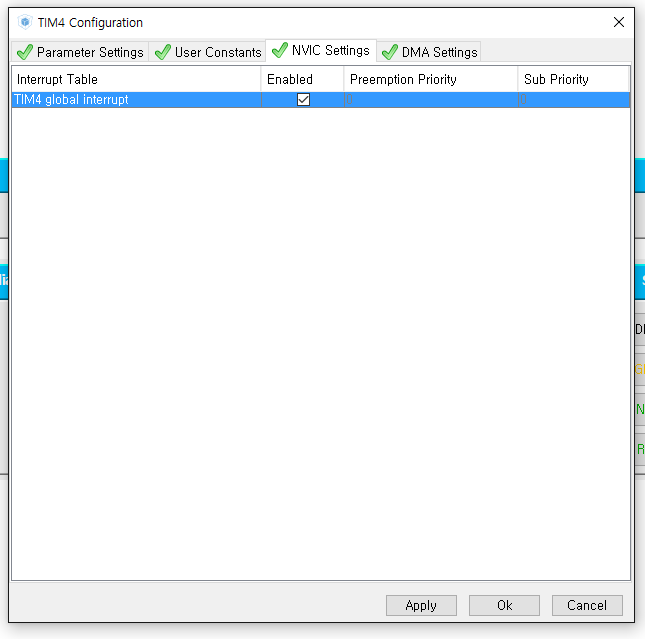
타이머4(TIM4)을 세팅해보자. 인터럽트를 1초 주기로 하는 것을 목표로 한다. 위 표를 참고하면 TIM4의 클럭은 84MHz이다. 그러면 prescaler 5000, counter 16800을 설정해주면 1초 마다 인터럽트가 발생할 것이라 예상가능하다.



사용하는 클럭을 내부 클럭으로 세팅한다. 그리고 configuration 탭에 TIM1설정에 들어가서 아래와 같이 세팅한다. 세팅 값을 주의해서 보기 바란다.



위에서 말한 값에서 1을 빼는 이유는 프로그래머의 숫자는 0부터 시작하기 때문이다. 이제 인터럽트를 세팅하기 위해 NVIC 세팅에 들어가서 TIM4 global interrupt를 enable하자.



앞서 설명한 방법처럼 variable.c, variable.h를 만들어 사용할 수 있는 global 변수를 만들자. 그리고, 해당 변수를 타이머 인터럽트 안에서 1씩 증가시키는 코드를 짜보자.

인터럽트를 사용하기 위해 작성해주어야 하는 코드는 아래와 같다.

Main.c

1. int main(void) {
2. HAL\_Init();
3. SystemClock\_Config();
4. MX\_GPIO\_Init();
5. MX\_TIM1\_Init();
6. MX\_TIM4\_Init();
7. HAL\_TIM\_Base\_Start\_IT( & htim4);
8. while (1) {
9. }
10. }

Line 7에 있는 HAL\_TIM\_Base\_Start\_IT 를 통해 인터럽트를 시작해주는 작업을 해주어야 하는 것에 주의하자.

Variable.h

1. #ifndef VARIABLE\_H\_
2. # define VARIABLE\_H\_
3. extern int timerTick;
4. #endif

Variable.c

1. #include "variable.h"
2. int timerTick = 0;

Stm32f4xx\_it.c

1. void TIM4\_IRQHandler(void) {
2. HAL\_TIM\_IRQHandler( & htim4); /\* USER CODE BEGIN TIM4\_IRQn 1 \*/
3. timerTick++; /\* USER CODE END TIM4\_IRQn 1 \*/
4. }

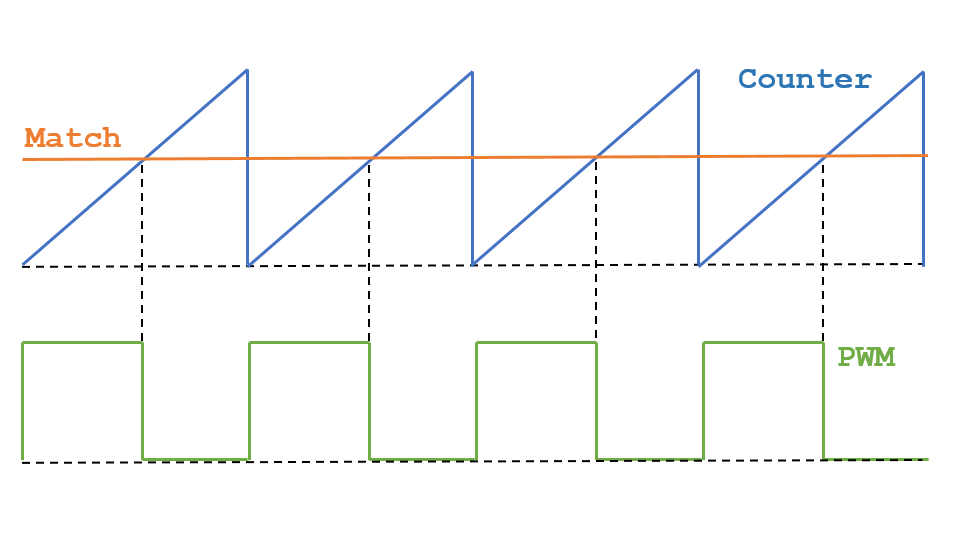
그리고 Live Watch를 이용하여 1초에 변수가 실제로 1씩 증가하는지 확인해보자.

타이머 인터럽트를 사용하면 주기적으로 매우 정확한 시간 간격 안쪽에 일어나야 하는 일들의 정확한 처리가 가능하다. 특히, 타이머 인터럽트를 이용하면 microsecond 안쪽에서 일어나는 미세한 전기적 제어가 가능하기 때문에, 자체 통신 프로토콜 작성 등을 위해서도 많이 사용된다.

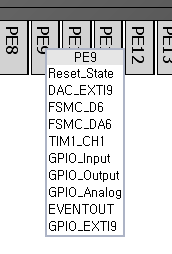
그러나, 1초에 한번 실행하고, 오차가 0.1초정도는 허용되는 경우는 타이머를 사용하지 않는 것이 바람직하다. 이러한 naïve한 활용에는 뒤에 나올 Real Time OS의 Task를 사용하는 것이 훨씬 효과적이기 때문이다.

* + 1. PWM

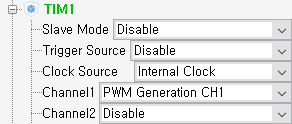
PWM은 Pulse Width Modulation의 약자로, 디지털 스위칭을 통해 아날로그 출력을 모방하는 방법이다. 작동 원래는 아래 그림과 같다. 타이머 카운터는 계속 증가하고, 특정 값이 되면 다시 0으로 돌아가는 작업을 타이머 인터럽트 때와 마찬가지로 반복한다. 그런데, 어떤 Match Value가 존재해서 카운터가 그 값까지 올라가는 도중에는 PWM 값을 1로 두게 하고, 그 값보다 커진 경우에는 PWM값을 0으로 둔다. 그래서 아래와 같은 상황에서는 Match Value가 작아지면 PWM은 1인 시간이 적어지고, Match Value가 커지면 PWM은 1인 시간이 길어진다.



PWM을 세팅하기 위해서는 PWM을 내보내고 싶은 핀이 어느 타이머에 연결되어 있는지를 먼저 파악해야 한다. 예를 들어 PE9에서 1kHz의 50% duty PWM을 내보내고 싶다고 하면, 먼저 아래와 같이 TIM1\_CH1이 하드웨어적으로 연결되어있는 것을 확인한다.



그리고 핀을 TIM1\_CH1으로 세팅하고, 좌측의 타이머 세팅을 아래와 같이 진행한다. 클럭을 인가하고 채널1을 PWM생성으로 설정한다.



그리고 1kHz를 생성하기 위한 Prescaler와 Counter 값을 계산한다.

1000 = 168,000,000 / ( 1000 \* 168 )

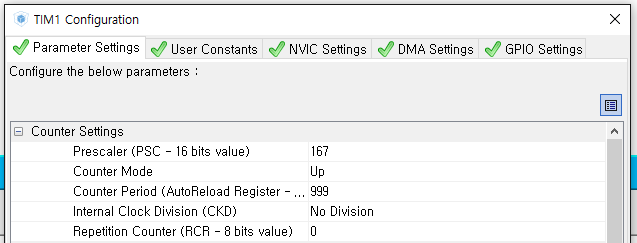
Prescaler = 168 – 1 : 167

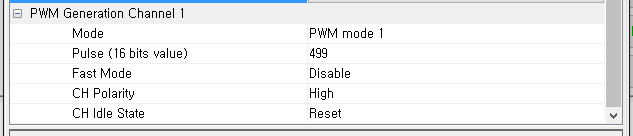
Counter = 1000 – 1 : 999

그리고 50% duty를 만들기 위한 Match 값을 계산한다.

(999 + 1)/2 -1 : 499

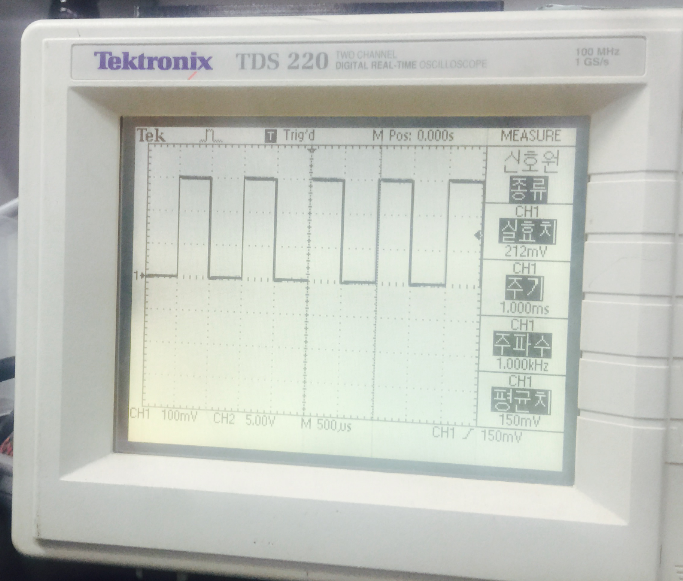
그리고 아래와 같이 세팅한다.





그리고 main 안쪽에 HAL\_TIM\_PWM\_Start(&htim1, TIM\_CHANNEL\_1)을 넣어서 PWM을 시작해주어야 한다.(Line 6) 또한 optimization level 조정도 잊지 말자.

1. int main(void) {
2. HAL\_Init(); /\* Configure the system clock \*/
3. SystemClock\_Config(); /\* Initialize all configured peripherals \*/
4. MX\_GPIO\_Init();
5. MX\_TIM1\_Init(); /\* USER CODE BEGIN 2 \*/ /\* USER CODE END 2 \*/
6. HAL\_TIM\_PWM\_Start( & htim1, TIM\_CHANNEL\_1);
7. while (1) {
8. }



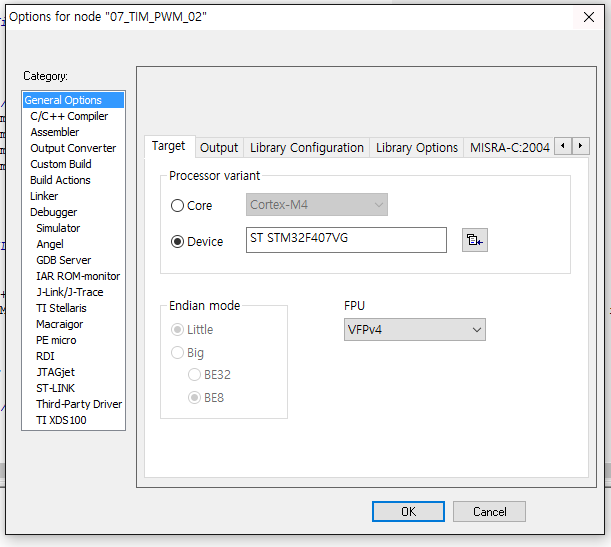
제대로 설정한 후 실행하면 위와 같은 깔끔한 PWM을 얻을 수 있다.

PWM의 duty (켜져 있는 비율)를 조정하여 주변기기의 출력을 조절하는데, 이러한 조절은 실시간으로 이루어져야 한다. 그래서 코드 중간에서 PWM을 조절하기 위한 함수가 HAL에 제공되는데 그 코드는 아래와 같다. (x에는 각각 알맞은 타이머 번호와 채널 번호를 넣어주면 된다.)

1. \_\_HAL\_TIM\_PWM\_SETCOMPARE(&htimx, TIM\_CHANNEL\_X, match\_value);

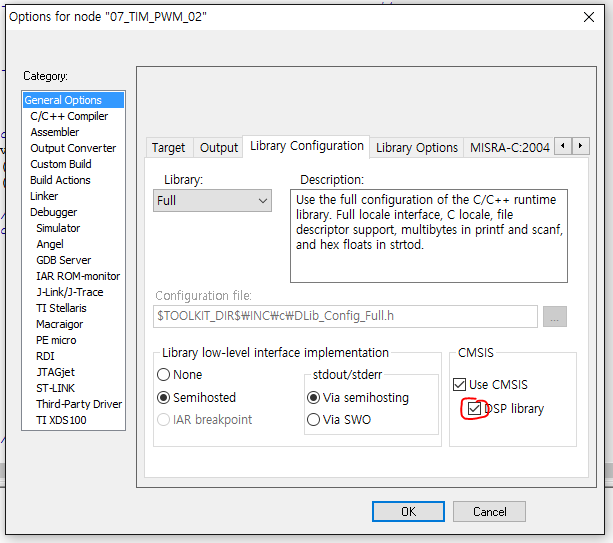
이것을 이용하여 첨부된 영상 video\_1의 동작을 하는 코드를 작성해보자. 예제 폴더 07번에 있는 코드를 실행시켜보아도 좋다. 해당 코드는 Math 라이브러리를 같이 사용하여 sinusoidal하게 밝기를 조절하는 코드이다.

**(TIP)** 아래 코드와 같이 math.h 라이브러리를 불러오면 일반적인 c 코드를 작성하듯이 math library를 사용할 수 있다. 그러나, math 표준 라이브러리는 컴퓨터 수준의 연산을 STM32 프로세서에 요구하기 때문에 다른 여러 기능을 사용하는 중에는 성능이 생각보다 잘 나오지 않는다. 가장 간단하게 성능을 개선하는 방법은 FPU를 켜는 것이다.



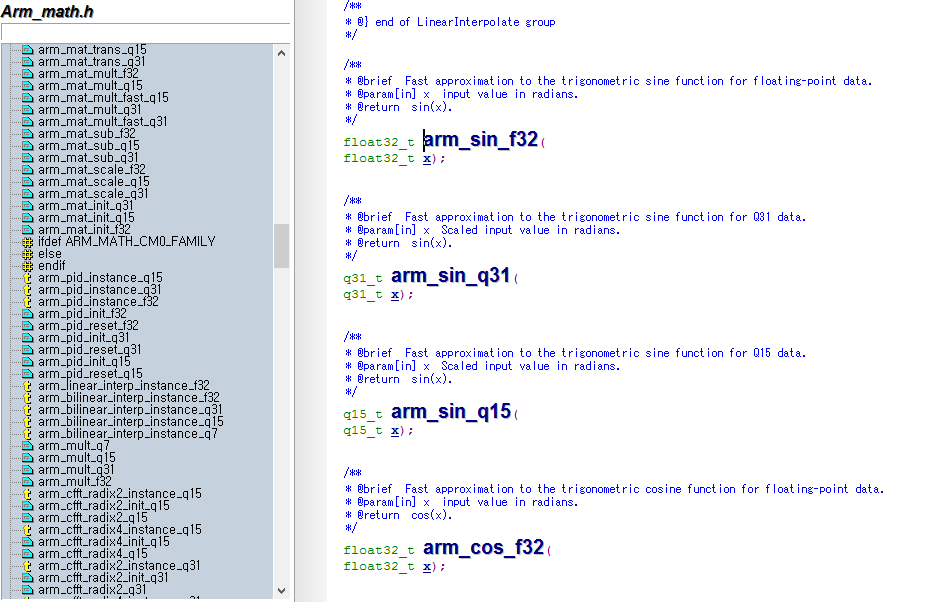
위와 같이 general option에서 FPU를 VFPv4로 넣어주면 소수점 연산 유닛을 하드웨어적으로 이용하여 연산하기 때문에 속도에서 확실한 증가를 볼 수 있다.

더 빠른 연산을 원한다면 table을 사용하면 된다. STM32F4는 DSP Library를 지원하는데, DSP 라이브러리 설정을 하고, 아래 세개의 헤더를 불러오면 CMSIS-DSP 라이브러리를 사용할 수 있다.



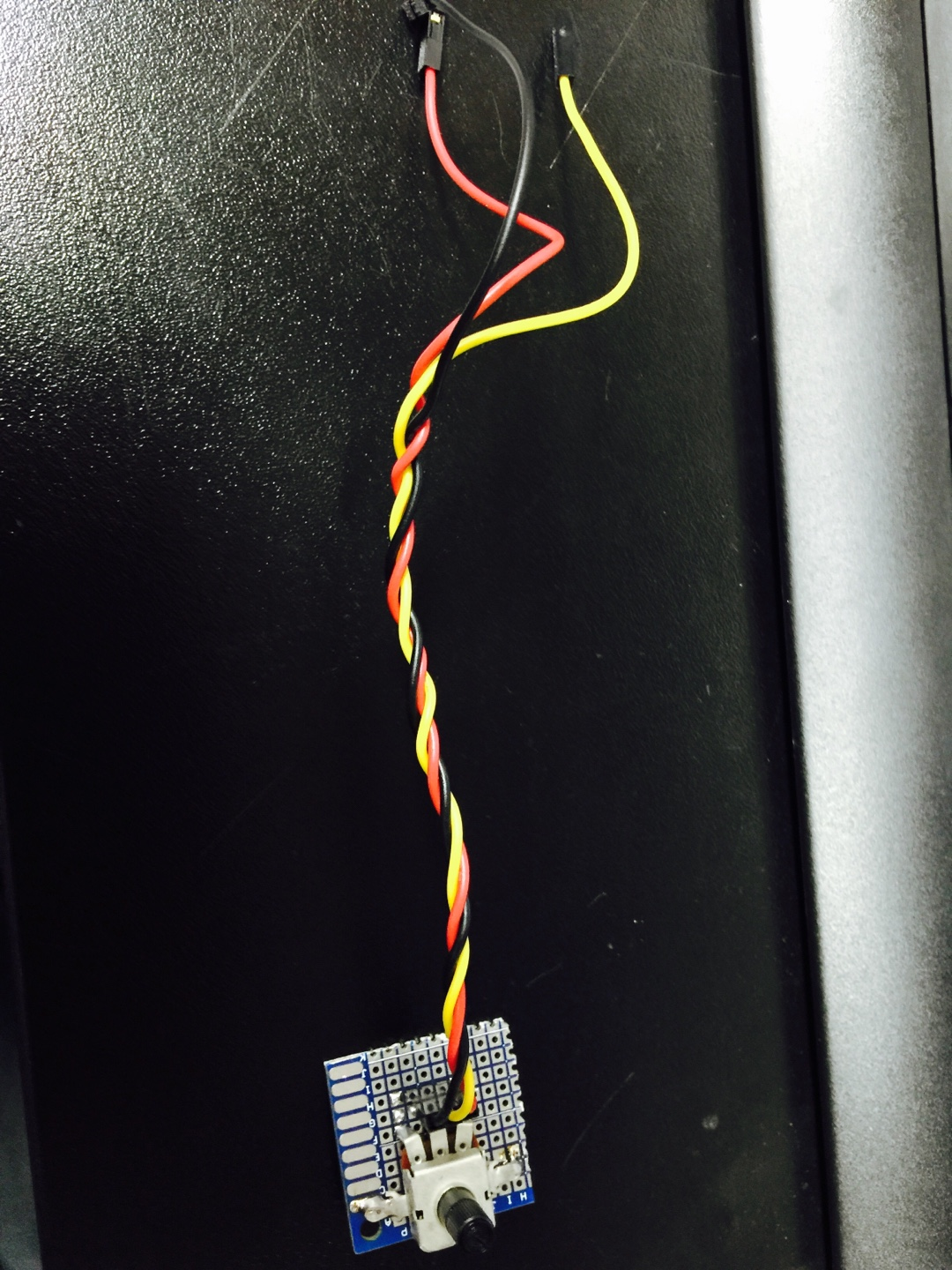
1. #include "arm\_math.h"
2. #include "arm\_const\_struct.h"
3. #include "arm\_common\_tables.h"

여기에서는 table 연산을 지원한다. sin, cos 값들이 전부 상수로 선언되어있기 때문에, 해당 상수를 바로 불러와서 사용할 수 있다. DSP 라이브러리에서는 FFT, PID 등 유용한 함수들이 추가적으로 제공되기 때문에, 해당 기능을 사용할 예정이라면 DSP 라이브러리를 사용하는게 최적화 면에서 엄청나게 도움이 되는 선택이다. 정확한 내용은 source insight로 해당 헤더 파일 내부를 잘 살펴보기 바란다. 아래와 같이 arm\_math.h 파일을 열고, 좌측의 함수들을 보면, 행렬 연산부터 시작해서 언젠가 제어공학개론, 공학 수학, 선형대수학에서 보았던 수학 내용들이 보인다. 이것들은 우리가 코딩할 필요 없이 가져다 사용하면 된다. Math.h 라이브러리를 사용하여 직접 작성하는 것보다 확실한 퍼포먼스를 보인다. (Math.h 라이브러리와 달리 source insight에서 자동완성 또한 지원된다. Arm\_ 을 키워드로 해서 불러오면 좋다.)



* 1. Analog to Digital Converter

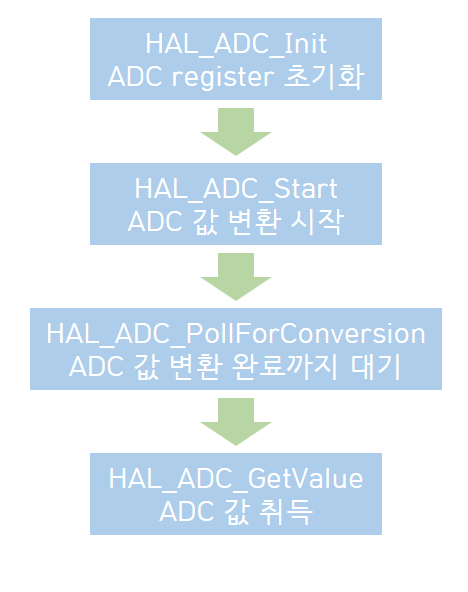
ADC는 Analog to Digital Converter의 약자로 아날로그 전압을 디지털 값으로 바꾸어주는 기능이다. 대부분의 아날로그 센서들에 쓰인다. 이번 단원을 시작하기 앞서 아날로그 값을 내보내 주는 장비를 하나 준비하자. 가장 기본적인 아날로그 센서로 가변 저항을 아래와 같이 연결하여 사용하자. 물론 다른 센서로 실험을 해보고 싶다면 다른 센서를 사용해도 좋다.



Analog Digital Converter는 핀에 걸린 아날로그 전압을 Reference voltage에 대한 상대적인 값으로 변환하여 프로그램상에서 읽어오는 장치를 말한다. STM32 Series의 기본 Reference voltage는 3V이고, ADC를 통해 컨버팅되어 나오는 전압 값은 12bit(max 4096)이다. 따라서 핀에 1V가 걸려있는 상황이라면, 그 핀을 통해 읽은 전압 값은 (1.0/3.0)\*2^12 = 1365 이다.

* + 1. Polling Reading

아두이노에서는 ADC를 analogRead(pin) 이라는 함수 하나로 구현해 놓았다. 이것이 Polling 방식의 아날로그 값 읽기인데, Polling 이란 값이 준비될 때까지 대기하고 있다가 값이 준비되면 읽어 들이는 방식의 읽기 동작이다. Polling 방식에서 ADC가 실제로 작동하는 순서를 그려보면 아래와 같다.



아두이노의 analogRead는 위 4개 과정을 하나의 함수로 묶어놓아 사용을 간편하게 만들었다. 하지만 느리다. 아두이노 기준 ADC에서 변환하여 받아올 수 있는 값은 프로그램을 잘 짠다고 해도 초당 4000개를 넘기 힘들다. 그러나 STM32F4의 경우 최적화 과정을 거치면 초당 1,000,000개까지 데이터를 받아올 수 있다.

위 4개의 과정 중 가장 오래 걸리는 것이 ADC 값 변환 완료까지 대기하는 과정이다. ADC 설정에 따라 다르지만 약 2us 정도가 ADC 변환에 필요한데, Polling 방식으로 프로그램을 작성하면 이 시간은 그대로 버려야 한다. 참고로 2us면 clock cycle이 168,000,000\*(2\*10^-6) = 336 cycle 돌 수 있는 시간이다. 어떻게 보면 짧은 시간이고, 어떻게 보면 긴 시간인데, 이 시간을 줄여야 하는지에 대한 판단은 어플리케이션마다 다르기 때문에 각자의 판단에 맡긴다.

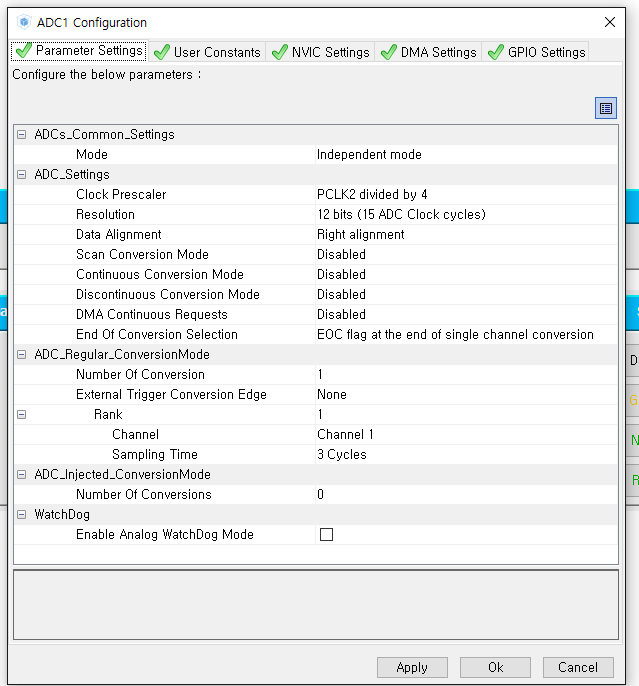
Polling 방식으로 코드를 작성했을 때 장점은 코드가 매우 간결하다. 코드가 간결한 것은 디버깅이 쉽다는 것이고, 총 개발시간을 단축할 수 있다는 것이다. 그리고, 다른 하드웨어 자원을 아낄 수 있다. Polling 방식은 ADC만 사용하기 때문에, DMA나 Timer를 연동해서 이용할 필요가 없기 때문이다.

이제 Polling 방식의 ADC 설정 과정을 살펴보자.





ADC1 IN1을 켜고, 핀이 제대로 세팅 되었는지 확인한다.



기본 설정을 그대로 사용해도 된다. Resolution을 낮추면 변환 속도가 빨라지니 참고하자. Continuous Conversion Mode를 Enable하면 HAL\_ADC\_Start를 한번만 불러와 주어도 된다.

프로젝트를 생성하고, main문 안쪽에 아래와 같이 코드를 작성하자. 참고로 hadc1은 main 위쪽에 선언되어 있는 adc 컨트롤 객체이다.

1. int value;
2. int main(void) {
3. HAL\_Init();
4. SystemClock\_Config();
5. MX\_GPIO\_Init();
6. MX\_ADC1\_Init();
7. while (1) {
8. HAL\_ADC\_Start( & hadc1);
9. HAL\_ADC\_PollForConversion( & hadc1, 10);
10. value = HAL\_ADC\_GetValue( & hadc1);
11. HAL\_Delay(10);
12. }
13. }

이 코드를 실행시키고 Live Watch로 변수가 실시간으로 변하는 것을 살펴보자.

이제 여러 개의 채널에서 ADC를 읽어와 보자. ADC1의 1,2,3번 채널에서 값을 읽는다.

먼저, ADC1 Configuration을 아래와 같이 설정한다.

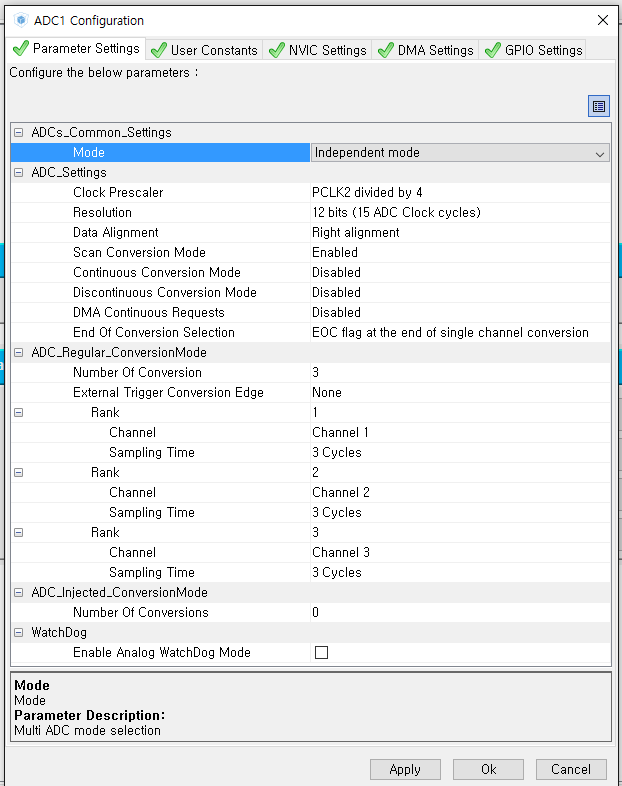
Number of Conversion: 3

Rank1 → Channel 1

Rank2 → Channel 2

Rank3 → Channel 3

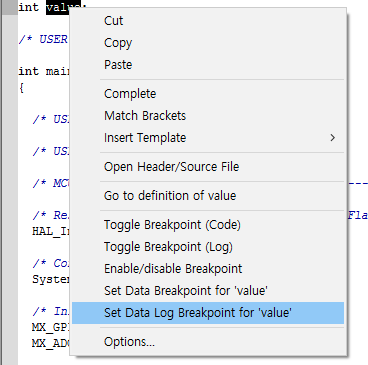
Scan Conversion Mode: Enable



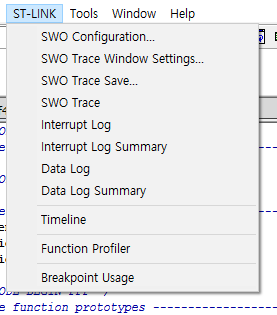
1. int value[3];
2. int main(void)
3. HAL\_Init();
4. SystemClock\_Config();
5. MX\_GPIO\_Init();
6. MX\_ADC1\_Init();
7. while (1) {
8. HAL\_ADC\_Start( & hadc1);
9. HAL\_ADC\_PollForConversion( & hadc1, 10);
10. value[0] = HAL\_ADC\_GetValue( & hadc1);
11. HAL\_ADC\_PollForConversion( & hadc1, 10);
12. value[1] = HAL\_ADC\_GetValue( & hadc1);
13. HAL\_ADC\_PollForConversion( & hadc1, 10);
14. value[2] = HAL\_ADC\_GetValue( & hadc1);
15. }
16. }

값을 읽어오는 코드는 위와 같이 순서대로 읽어오면 된다.

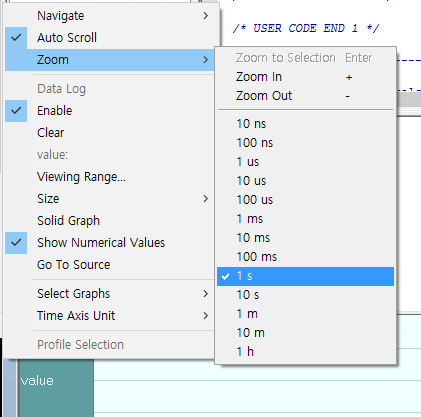
**(TIP)** Live Watch는 실시간으로 변수를 보여주지만 변수 변화 양상까지 보여주진 못한다. IAR은 내부에 Data Log 및 Timeline 이라는 기능을 통해 한가지 변수에 대해 그래프를 그려주는 기능을 포함하고 있다. 먼저 Debug 모드로 들어가서 추적할 변수를 우클릭한다. 그리고 Set Data Log Breakpoint for ‘변수 이름’ 을 클릭하면 이제 Data Log 창에서 해당 변수를 계속해서 추적한다.

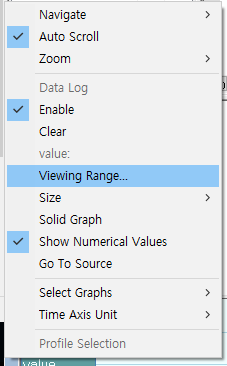
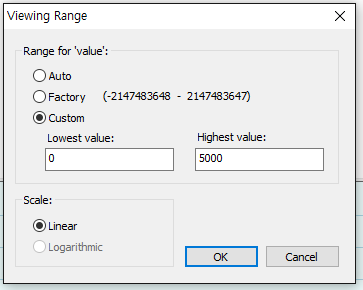


이제 ST-Link 탭에서 Data Log와 TimeLine을 켜자.

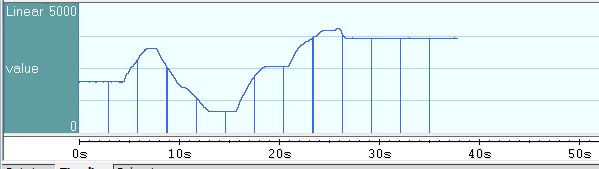


TimeLine에서 변수의 범위와, 볼 시간 축의 scale을 설정한다.

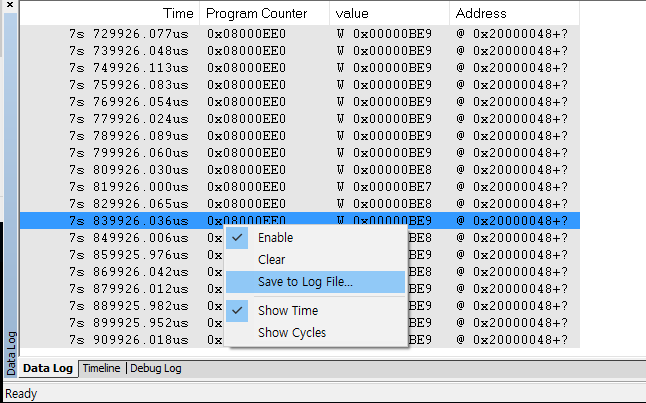


이제 Timeline에 실시간으로 변수의 값이 그래프로 출력된다.



실험 데이터를 얻어서 분석해야 하는 상황이라면 Data Log에 있는 값을 가져가면 된다.

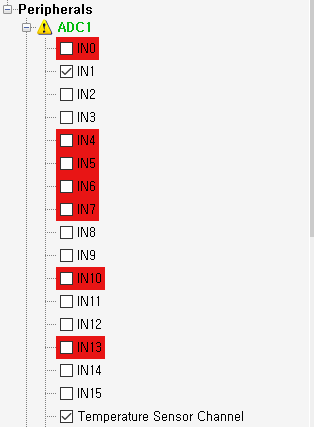


Text file로 저장되기 때문에 Matlab, Excel 등 원하는 툴에 적당히 가져가서 사용하면 된다.

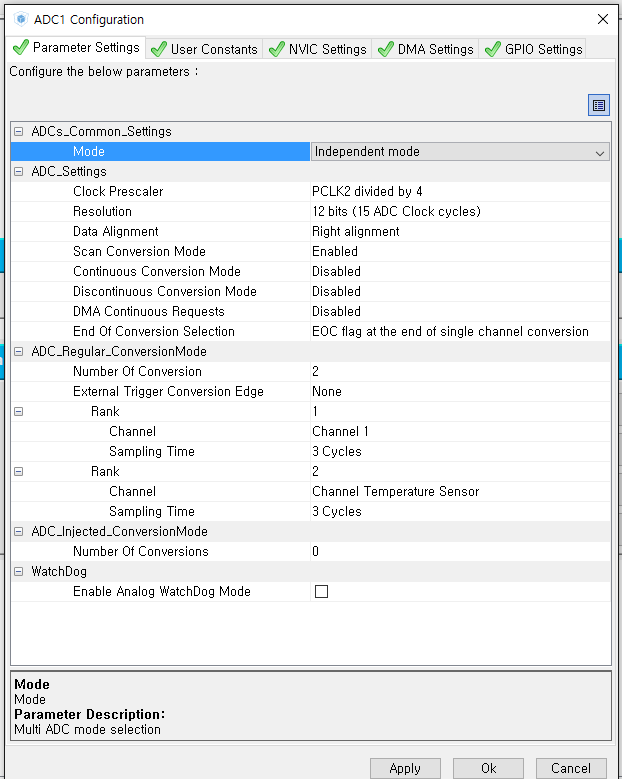
* + 1. Interrupt Reading

이제 인터럽트를 사용해서 ADC 변환 시간을 따로 기다리지 않고, ADC 변환이 끝났다는 인터럽트 신호를 이용해서 값을 읽어오는 코드를 작성해보자.

STM32 안쪽에 있는 1번채널을 사용하자.



아래와 같이 설정한다.



그리고 NVIC 설정에서 ADC 인터럽트를 enable한다.



Main과 Interrupt에서 모두 사용할 수 있는 global variable을 관리하기 위해 variable 파일을 만들어 앞서 이야기한 방법대로 세팅한다.

마지막으로, stm32f4xx\_it 에서 ADC 값을 읽어오는 코드를 작성합니다.

* 버그가 좀 있는듯 합니다. 사실상 ADC는 인터럽트 방식으로 사용하는 일이 드물기 때문에 뒤의 DMA 방식으로 넘어가겠습니다. 예제 코드도 참고하지 마시기 바랍니다.
  + 1. ADC DMA

Direct Memory Access는 메모리 관리 하드웨어 유닛으로, CPU의 일을 분산하여 처리하는데 그 목적이 있다. DMA가 무엇이고, 왜 쓸모 있는지에 대해서는 아래와 같은 예시를 들 수 있겠다.

신입사원 STM은 128잔의 커피를 끓여서 임원진에게 가져다 달라는 부탁을 받았다. 가스레인지는 하나이고, 주전자도 하나이다. 이러한 상황에서 대처하는 방식은 아래와 같을 수 있다.

Polling : 물을 넣고 주전자를 올린 후 (데이터를 요청한 후) 물이 끓을 때 까지 앞에서 지켜보고 있는다. 지켜보다가 물이 끓으면(데이터 요청이 완료되면) 끓는 물을 부어서 커피를 완성시킨다. (데이터를 프로그램 메모리로 가져온다.) 이 작업을 128번 반복한다.

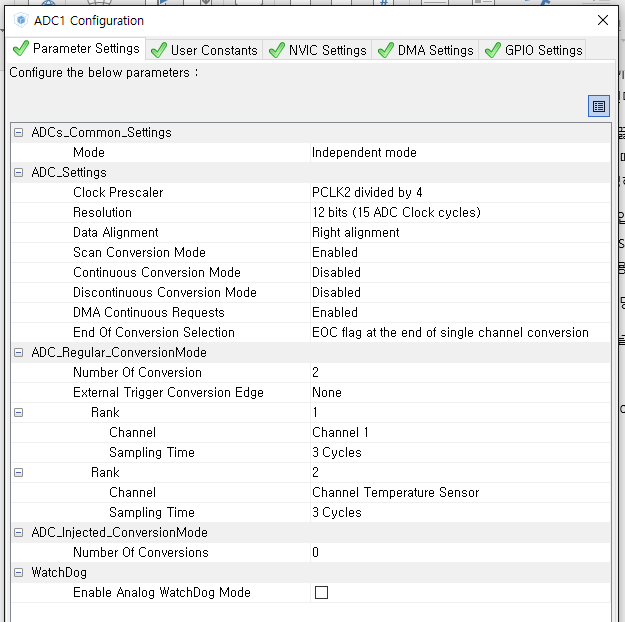
Interrupt : 물을 넣고 주전자를 올린 후 (데이터를 요청한 후) 물이 끓는다는 소리가 나기 전까지 어디선가 농땡이를 피운다. 물이 끓는 소리가 들리면(인터럽트가 발생하면) 끓는 물을 부어서 커피를 완성시킨다. (데이터를 프로그램 메모리로 가져온다.) 이 작업을 128번 반복한다.

DMA : 커피를 끓이는 로봇에게 128잔의 커피를 요청한다. (DMA에게 128개 데이터 요청) 모든 작업이 완료될 때까지 어디선가 농땡이를 피우다가 로봇이 다 완료되었다고 말하면 (DMA 완료 인터럽트가 발생하면) 완성된 128잔의 커피를 가져온다.

가장 영리한 신입사원은 DMA를 사용할 것이다. 그러나 DMA는 세팅이 조금 까다롭고, 다른 통신 모듈(UART, I2C, SPI)도 DMA를 사용하기 때문에, ADC의 경우 DMA의 우선순위에서 밀리면 Polling 방식을 많이 사용하기 도 한다.

DMA 세팅은 상당히 복잡하기 때문에 아래 나오는 설정 값들을 잘 따라오기 바란다.

먼저 두개의 채널을 사용하자. ADC1 IN1, Temp sensor 두개의 채널을 enable한다.



ADC를 위와 같이 세팅한다. Scan conversion mode로 한번 변환할 때 모든 채널을 스캔하며 변환하도록 하고, Continuous Mode는 지속적으로 ADC를 돌릴 것이면 Enable, 주기적으로 따로 ADC를 돌릴 것이면(Sampling 주기를 Timer를 이용해 결정하는 등) Disable한다. 처음에는 DMA의 동작을 보기 위해 Disable로 하자. DMA Continuous Request를 Enable한다. 채널 설정은 위에서 했던 것과 같다.

DMA를 아래와 같이 세팅한다.



CubeMX에서 프로젝트를 생성한 후,

ADC값이 들어갈 16비트 32개짜리 버퍼를 만든다. (Line 1)

main에서 ADC DMA를 시작하고(Line 9), 0.5초에 한번씩 ADC를 시작하는 코드를 짠다.(Line 11,12)

1. volatile uint16\_t ADC1ConvertedValues[32];
2. int main(void) {
3. HAL\_Init();
4. SystemClock\_Config();
5. MX\_GPIO\_Init();
6. MX\_DMA\_Init();
7. MX\_ADC1\_Init();
8. MX\_USART2\_UART\_Init();
9. HAL\_ADC\_Start\_DMA( & hadc1, (uint32\_t \* ) ADC1ConvertedValues, 32);
10. while (1) {
11. HAL\_ADC\_Start(&hadc1);
12. HAL\_Delay(500);
13. }

이제 이 버퍼가 어떻게 동작하는지 Live Watch를 통해 지켜보자. 0.5초에 하나씩 버퍼가 채워지는 것을 알 수 있을 것이다.

**(TIP)** 초당 10,240개의 샘플을 일정한 시간 간격으로 받아와야 하고, 1024개 단위로 0.1초마다 어떤 작업을 해야 하는 활용을 생각해보자. 그러면 아래와 같은 방식으로 시스템을 설계하면 된다.

* 1/10240 sec 주기로 동작하는 Timer Interrupt 구문 안에 ADC를 시작하는 코드를 넣는다.
  + TIM4 기준
  + Prescaler = (1-1) = 0
  + Counter = 8203.125 -1 ≒ 8202
* 1024길이를 갖는 DMA 버퍼를 만들고, 위에서 이야기한 것과 같이 DMA설정을 한다.
* Main 아래에 void HAL\_ADC\_ConvCpltCallback(ADC\_HandleTypeDef\* hadc){ } 를 정의하고, 해당 함수 안쪽에 어떤 작업에 대한 코드를 작성한다. ( 위 함수는 HAL에서 미리 weak로 선언되어 있는 prototype 함수로, DMA 변환이 한 주기 끝날 때 마다 자동적으로 불리게 되어있다.)
  1. UART

UART는 아두이노의 시리얼 통신에 해당하는 유닛으로, 칩 내부의 데이터를 외부에 전송할 때, 데이터 전송으로 주변기기를 컨트롤 할 때 사용한다. GPS 등의 센서는 센서 값을 UART로 주는 경우도 있기 때문에, 필수적으로 사용하는 주변기기 중에 하나이다.

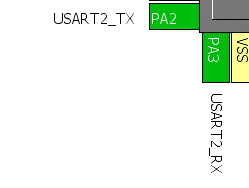
컴퓨터와의 통신을 가정하고 시작하자. 아래와 같은 USB to UART 장비가 필요하다.

 RX, TX, GND 핀을 사용한다.

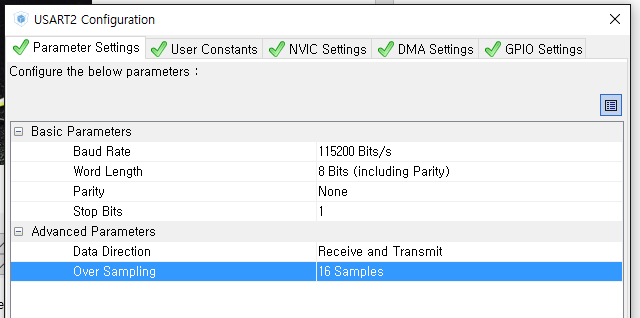
* + 1. Conventional UART (Polling TX, Interrupt RX)



USART2를 이용해 UART 통신을 한다. Mode는 Asynchronous로 맞추고, RS232는 disable 한다. UART2를 사용하기 때문에, PA2, PA3 핀이 활성화된다.

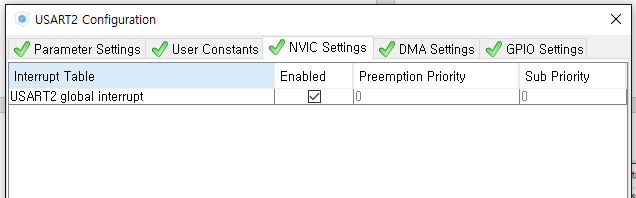


STM32의 TX핀은 USB to UART 기기의 RX핀에, STM32의 RX핀은 USB to UART 기기의 TX핀에 연결한다. 그리고 두 기기의 GND를 일치시키는 것을 잊지 말자.



Baud Rate (통신 속도)는 115200bps로 맞추고, 나머지 설정은 건드리지 않아도 된다.

UART는 데이터를 받는 동작에는 무조건 Interrupt를 사용하는 것이 좋다. UART 데이터는 언제 들어올지 모르기 때문에, Polling 방식을 사용한다면 CPU 낭비가 매우 심하기 때문이다.



여기까지 했으면 설정은 끝났다.

이제 RX 인터럽트를 main 안쪽에서 enable(Line 6)하고, 1초에 한번씩 Hello World를 보내는(Line 8,9) 프로그램을 작성해보자.

1. int main(void) {
2. HAL\_Init();
3. SystemClock\_Config();
4. MX\_GPIO\_Init();
5. MX\_USART2\_UART\_Init();
6. \_\_HAL\_UART\_ENABLE\_IT( & huart2, UART\_IT\_RXNE);
7. while (1) {
8. HAL\_UART\_Transmit( & huart2, "Hello World!\r\n", 14, 1000);
9. HAL\_Delay(1000);
10. } /
11. }

들어오는 데이터에 대한 확인은, Stm32f4xx\_it.c 안의 UART ISR에 breakpoint를 걸어놓고, huart2.instance->DR char 형태로 보면 들어오는 것을 볼 수 있다. 해당 데이터를 원하는 곳에 넣어서 해석하면 된다.

**(TIP)** UART는 다양한 인터럽트를 지원한다. 이러한 인터럽트를 구분해서 처리할 필요가 있는데, 인터럽트의 구분은 FLAG라는 데이터를 통해 이루어진다. 원래는 FLAG의 위치 등을 직접 데이터시트에서 찾아서 해야 하지만 HAL에서 간단히 검사할 수 있는 함수를 제공한다. UART함수의 경우에는

\_\_HAL\_UART\_GET\_IT\_SOURCE

라는 함수를 통해 검사하는데, 함수의 설명을 source insight를 통해 뜯어보면 아래와 같다.

**\* @brief Checks whether the specified UART interrupt has occurred or not.**

**\* @param \_\_HANDLE\_\_: specifies the UART Handle.**

**\* This parameter can be UARTx where x: 1, 2, 3, 4, 5, 6, 7 or 8 to select the USART or**

**\* UART peripheral.**

**\* @param \_\_IT\_\_: specifies the UART interrupt source to check.**

**\* This parameter can be one of the following values:**

**\* @arg UART\_IT\_CTS: CTS change interrupt (not available for UART4 and UART5)**

**\* @arg UART\_IT\_LBD: LIN Break detection interrupt**

**\* @arg UART\_IT\_TXE: Transmit Data Register empty interrupt**

**\* @arg UART\_IT\_TC: Transmission complete interrupt**

**\* @arg UART\_IT\_RXNE: Receive Data register not empty interrupt**

**\* @arg UART\_IT\_IDLE: Idle line detection interrupt**

**\* @arg USART\_IT\_ERR: Error interrupt**

**\* @retval The new state of \_\_IT\_\_ (TRUE or FALSE).**

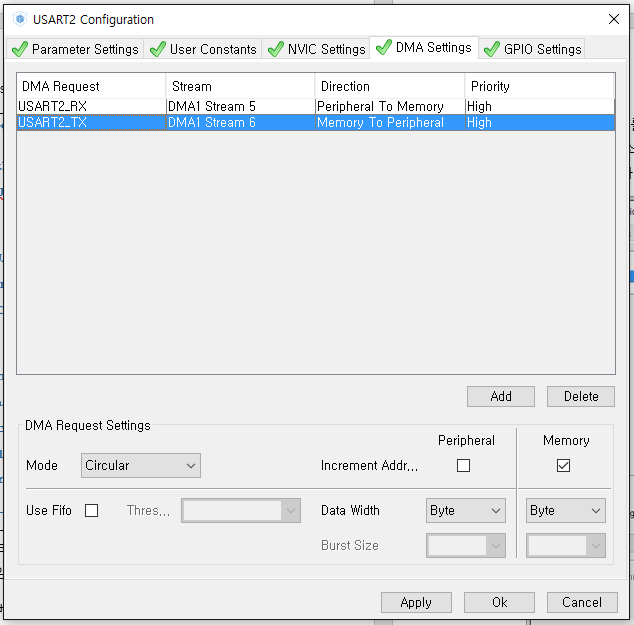
첫번째 파라미터로는 &huartx, 두번째 파라미터로는 UART\_IT\_TYPE을 넣으면 현재 인터럽트가 이것이 맞는지 true or false로 return 해주는 함수임을 알 수 있다.

따라서 안전하게 작성된 데이터를 받아와서 버퍼에 넣는 UART ISR은 아래 코드와 같다. 마지막에 다시 \_\_HAL\_UART\_ENABLE\_IT 를 불러와야 함에 유의하자.

1. void USART2\_IRQHandler(void) { /\* USER CODE BEGIN USART2\_IRQn 0 \*/
2. if (a = \_\_HAL\_UART\_GET\_IT\_SOURCE( & huart2, UART\_IT\_RXNE)) {
3. HAL\_UART\_Receive\_IT( & huart2, uartRxBuffer + uartRxBufferIndex, 1);
4. uartRxBufferIndex++;
5. } /\* USER CODE END USART2\_IRQn 0 \*/
6. HAL\_UART\_IRQHandler( & huart2); /\* USER CODE BEGIN USART2\_IRQn 1 \*/
7. \_\_HAL\_UART\_ENABLE\_IT( & huart2, UART\_IT\_RXNE);
8. }
   * 1. UART DMA

UART 역시 DMA를 지원한다. DMA 기능은 RX, TX 모두 지원한다. ADC의 DMA가 peripheral(기능을 가지고 있는 요소) 에서 메모리로의 접근이었다면, UART는 TX의 경우 메모리에서 peripheral로의 접근을 해야 한다. 각각을 어떻게 configure하는지 보자.





UART 세팅은 이전과 동일하다.

Main 코드는 아래와 같다.

1. uint8\_t uartRXBuffer[128];
2. uint8\_t uartTXBuffer[128];
3. int main(void) {
4. HAL\_Init();
5. SystemClock\_Config();
6. MX\_GPIO\_Init();
7. MX\_DMA\_Init();
8. MX\_USART2\_UART\_Init();
9. HAL\_UART\_Receive\_DMA( & huart2, uartRXBuffer, 128);
10. HAL\_UART\_Transmit\_DMA( & huart2, uartTXBuffer, 128);
11. while (1) {
12. HAL\_Delay(500);
13. }
14. }

Live Watch를 통해 uartRXBuffer를 감시하면 알아서 들어온 데이터가 RX Buffer에 쌓이는 것을 볼 수 있다. uartTXBuffer를 사용하는 방법은 조금 까다로운데, 일단 DMA는 uartTXBuffer에 데이터가 있으면 무조건 전송을 해버린다. 문제는 이 작업을 무한반복하기 때문에 한번 전송한 후에는 해당 데이터를 제어해주는 작업이 필요하다.

그렇기 때문에 일단 데이터를 전송하려면, uartTxBuffer에 데이터를 직접 넣어주어야 한다.

DMA를 통해 UART 데이터가 전송되고 나서는 void HAL\_UART\_TxCpltCallback(UART\_HandleTypeDef \* huart){ } 함수가 callback으로 자동호출되기 때문에, 이 함수에서 데이터를 지우는 작업을 해주면 된다.

그러나 사실 TX는 인터럽트 + Queue로 처리하는 것이 Scheduling할 때 조금 더 코드 상으로 깔끔하다. 따라서 UART에서 DMA는 RX Example만 확인하면 되겠다.

UART\_IT

PC

(COM)

STM

UART\_DMA

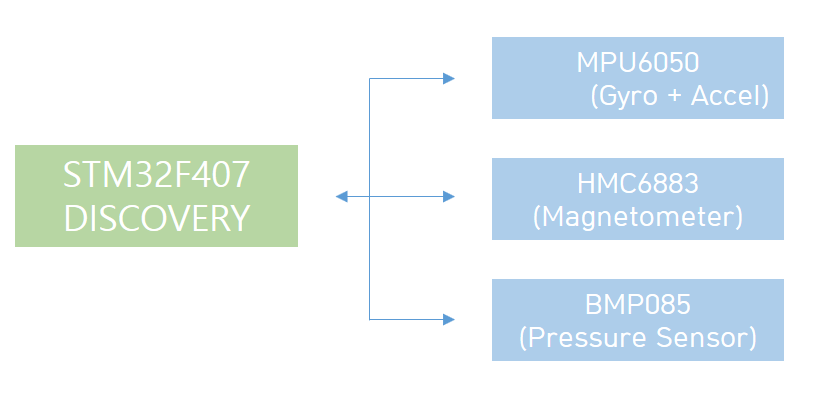
* RxBuffer의 Index값 확인하는 법

Buffer의 특성 상, 들어온 값이 차곡차곡 쌓입니다. 데이터가 차이면서 ‘데이터가 어디까지 차있다.’라는 것을 나타내 주는 변수가 Index입니다. 이 값은 UART2를 이용하면 ‘huart2.hdmarx->Instance->NDTR’라는 값을 참조하여 알 수 있습니다. ( 여기서 ‘.’과 ‘->’의 의미는 다들 아시겠지만 Structure내부의 Structure나 Pointer의 값을 참조하기 위함입니다.)

조금 주의할 점은 Index가 0에서 커지는 것이 아니라 Buffer의 값에서 줄어듭니다. ( 만약 Buffer 크기가 128이면 처음에는 128이고 3개의 문자가 들어오면 125로 줄어듭니다. )

* 1. I2C

I2C는 여러 기기를 병렬로 연결하여 사용할 수 있는 통신방식으로, 센서 네트워크를 구성할 때 많이 사용하는 통신방식이다. I2C 통신 방식의 장점으로는 아래와 같이 한 개의 MCU에 여러 센서를 동시에 물려 사용할 수 있다는 점이고,



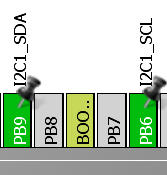
단점으로는 통신속도가 여타 통신 방식에 비해 현저하게 느리면서, 가변적으로 통신 속도를 조절하기 힘들다는 점이 있다. 물론 이것을 해결하기 위해 여러 해결책이 있지만, 기본 통신 방식에서 벗어나기 때문에 설정이 힘들고, 차후에 호환성 문제에 부딪힐 수 있다.

일반적으로 I2C 방식은 메모리를 읽어오고/ 쓰는 통신방식을 많이 사용한다. 예를 들어, MPU6050이라는 자이로스코프/가속도센서는 계속해서 내부에서 ADC를 돌려서 각속도/ 가속도를 업데이트 해놓는다. 그리고 그 값은 센서 내부의 특정 주소에 저장되도록 만들어져 있다. 이제 STM32F4에서는 특정 주소에 있는 값을 요청하면, 그 값이 읽히는 방식이다. 따라서 I2C 방식의 센서는 아래와 같이 센서의 주소지에 대한 정보를 제공해준다. (sensor name) + Register Map으로 검색하면 보통 나온다.



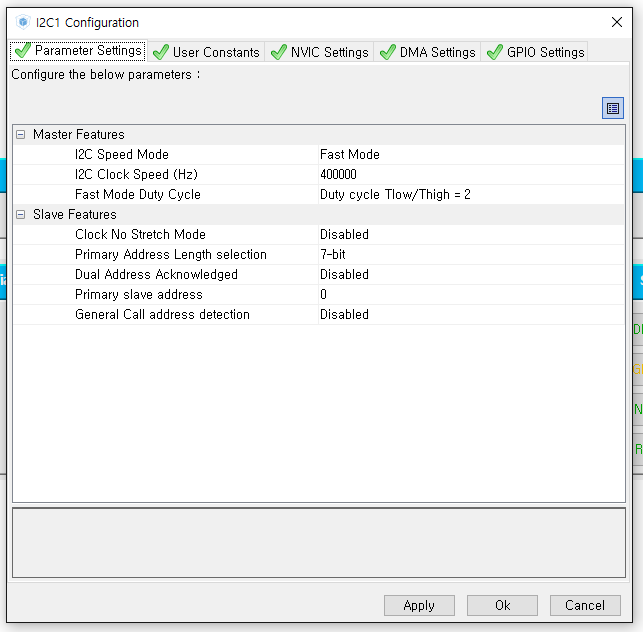
또한, 여러 센서와 동시에 통신하기 때문에 센서들은 자신의 고유 주소지를 따로 가지고 있어야 한다. 이것을 Slave Address라고 한다. 보통 센서 마다 가지고 있는 번지수가 다르기 때문에 큰 문제는 되지 않는다. MPU6050의 경우 기본 0x68을 Slave Address로 사용한다.

이제부터 그럼 MPU6050의 센서 값을 읽어보자. CubeMX에서 I2C1을 enable한다.



MPU6050의 SDA를 PB9에, SCL을 PB6에 연결한다. (추가로 센서를 연결하는 경우에도 SDA와 SCL을 맞추어 놓고 프로그램상에서 Slave Address만 변경하여 사용하면 된다.)

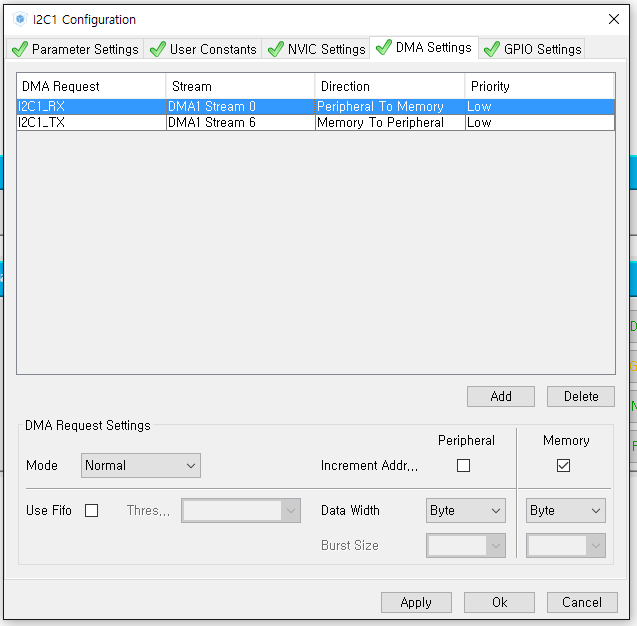
아래와 같이 I2C 세팅을 진행한다. Fast Mode로 하지 않아도 정상작동하나, 안 그래도 느린 통신 방식이니 Fast Mode를 가능하면 설정하는 것을 추천한다. (센서에 따라 구형은 Fast Mode를 지원하지 않는 것도 있으니 Datasheet를 잘 읽어보고 사용하기 바란다.)

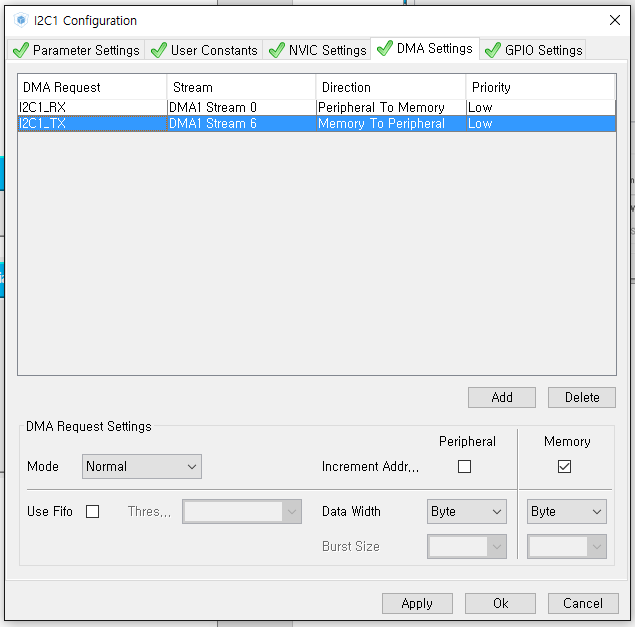


I2C를 DMA없이 사용하려면 위와 같이 설정하면 설정이 완료되었다. 그런데, I2C를 DMA 없이 사용하는 것은 상당히 위험한 행동인데, 그 이유는 I2C가 매우 느린 통신방식이기 때문이다. MPU6050의 경우 데이터를 한 번 요청하고 받는데 드는 시간이 Slave Address/ Memory Address 요청을 제외하고도 (8\*14)/400,000 = 0.00028(s) = 0.28ms 나 된다. MPU6050은 보통 초당 200회 가까이 데이터를 요청하기 때문에 통신으로 허비되는 시간만 1초에 56ms나 된다. 여기에 센서가 추가되거나, 센서에서 읽어와야 하는 데이터가 많아지면 상황은 더 심각해진다. 중간에 매우 복잡한 계산 (FFT, Filter) 등이 들어가면 STM32F4가 더 버티지 못하는 상황이 발생할 수 있다.

이런 데이터를 읽어오는 작업은 최대한 DMA에게 맡기는 것이 좋다. 따라서 I2C는 DMA 예제만 제공을 한다. 물론 데이터 업데이트 간격이 긴 경우에는 오히려 DMA로 짜는 것이 코드를 더 복잡하게 만들어주기도 한다. DMA 없이 사용하는 경우에는 아래 함수 뒤에 붙은 DMA를 제거하고, 함수 마지막에 timeout parameter를 넣어주면 된다.

CubeMX에서의 DMA 세팅은 아래와 같다.

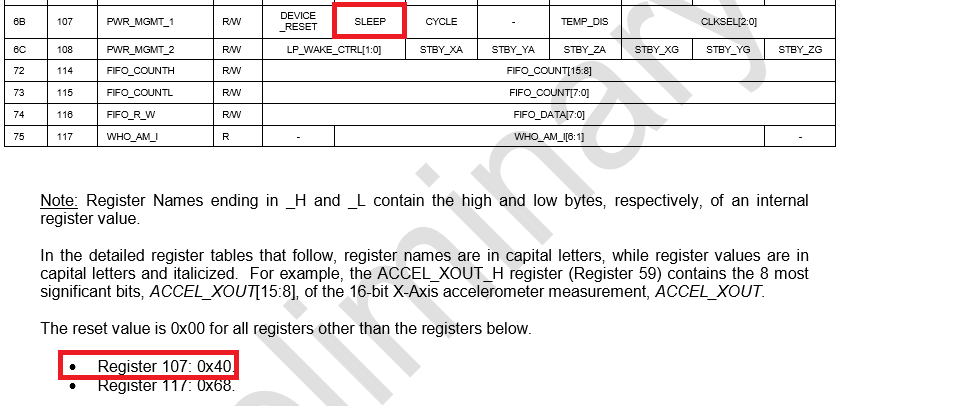




main 안쪽에서는 처음에 MPU6050의 Sleep mode를 disable하고, 데이터를 읽어와서 재정렬해 넣는다. 이것을 초당 약 100번 반복한다. 주의해야 하는 사항은 Slave Address를 넣을 때 우리가 아는 값에서 1bit shift해서 넣어주어야 한다는 사실이다. (함수가 조금 길어 문서에서 보기 힘드니 예제 폴더 14번을 열어 보는 것을 추천합니다.)

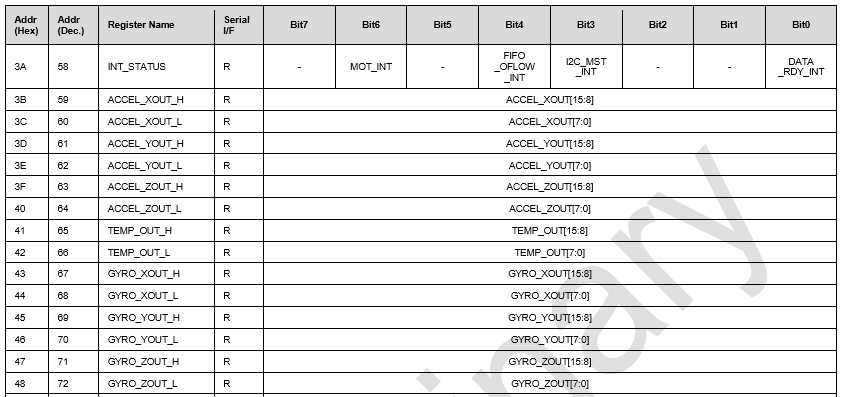
1. uint8\_t i2cReadBuffer[14];
2. int16\_t mpu\_value[7];
3. uint8\_t init\_mpu = 0x00;
4. int main(void) {
5. int i, ;
6. HAL\_Init();
7. SystemClock\_Config();
8. MX\_GPIO\_Init();
9. MX\_DMA\_Init();
10. MX\_I2C1\_Init();
11. HAL\_I2C\_Mem\_Write\_DMA( & hi2c1, **(0x68 << 1)**, 0x6B, I2C\_MEMADD\_SIZE\_8BIT, & (init\_mpu), 1); // mpu 6050 Wake up (mpu 6050 default is sleep mode)
13. while (1) {
14. HAL\_I2C\_Mem\_Read\_DMA( & hi2c1, **(0X68 << 1)**, 0x3B, I2C\_MEMADD\_SIZE\_8BIT, (uint8\_t \* ) i2cReadBuffer, 14);
15. for (i = 0; i < 7; i++) {
16. mpu\_value[i] = (((int16\_t) i2cReadBuffer[2 \* i]) << 8) | i2cReadBuffer[2 \* i + 1];
17. }
18. HAL\_Delay(10);
19. }
20. }

코드를 설명하면, 처음에 0x6B 번지에 0x00를 쓰는 것을 MPU6050을 Wakeup 시키는 것이다. 아래 MPU6050 Register Map을 참조하면, MPU6050의 0x6B초기값은 0x40 = 0b01000000 으로 Sleep bit가 켜져 있는 상태이다. 이것을 0으로 만들어주기 위해서 0b00000000 = 0x00을 써넣어준다.



이제 Sleep Mode에서 나왔으니, MPU6050은 각속도/가속도를 지정된 번지에 계속 업데이트 할 것이다.

해당 번지가 어디인지도 역시 MPU6050 Register Map에 제공된다.



0x3B부터 0x48까지가 MPU6050의 데이터 주소라는 것을 알 수 있다. 16비트 int 데이터이기 때문에 주소 두개가 데이터 하나를 가지고 있는 형태이다. 이것을 i2cReadBuffer에 읽어오면 아래와 같이 데이터가 Mapping된다.

|  |  |  |
| --- | --- | --- |
| **Array Location** | **Register Address** | **Register Name** |
| i2cReadBuffer[0] | 0x3B | ACCEL\_XOUT HIGH BITS |
| i2cReadBuffer[1] | 0x3C | ACCEL\_XOUT LOW BITS |
| i2cReadBuffer[2] | 0x3D | ACCEL\_YOUT HIGH BITS |
| i2cReadBuffer[3] | 0x3E | ACCEL\_YOUT LOW BITS |
| i2cReadBuffer[4] | 0x3F | ACCEL\_ZOUT HIGH BITS |
| i2cReadBuffer[5] | 0x40 | ACCEL\_ZOUT LOW BITS |
| i2cReadBuffer[6] | 0x41 | TEMP\_OUT HIGH BITS |
| i2cReadBuffer[7] | 0x42 | TEMP\_OUT LOW BITS |
| i2cReadBuffer[8] | 0x43 | GYRO\_XOUT HIGH BITS |
| i2cReadBuffer[9] | 0x44 | GYRO\_XOUT LOW BITS |
| i2cReadBuffer[10] | 0x45 | GYRO\_YOUT HIGH BITS |
| i2cReadBuffer[11] | 0x46 | GYRO\_YOUT LOW BITS |
| i2cReadBuffer[12] | 0x47 | GYRO\_ZOUT HIGH BITS |
| i2cReadBuffer[13] | 0x48 | GYRO\_ZOUT LOW BITS |

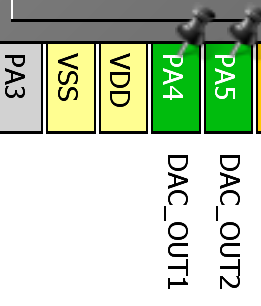
Array Location 상에서 상위비트묶음이 하위비트묶음보다 아래에 있으므로 이것을 다시 mpu\_value라는 array에 정렬하여 넣어주면 된다.

* 1. DAC

DAC는 ADC의 반대 역할을 수행하는 장치로, 프로세서 내부의 디지털 값을 아날로그 값으로 출력해준다. PWM과 다른 점이라면, PWM은 디지털 출력을 빠르게 켜고 끄는 작업을 통해 아날로그 출력을 ‘모방’ 한 것이고, DAC는 진짜 아날로그 출력을 내준다는 점이다.

DAC는 고속으로 값을 아날로그로 뽑을 수 있기 때문에, 매우 짧은 시간간격에서의 변수의 모니터링이 가능하다. 사람 눈으로 좇을 수 없는 변수를 오실로스코프로 모니터링 하는 데에 사용할 수 있기 때문이다.

그러나 구현하기 상당히 힘든 장치 중 하나이기 때문에, 고급 MCU에만 부착되어 있는 기능이며, MCU에도 몇 개 없는 기능이기 때문에, DAC를 많이 사용하지는 못한다. STM32F4에는 두개의 DAC가 달려있다. 두 개 전부 사용해보자. 아래와 같이 PA4, PA5를 DAC로 설정한다.

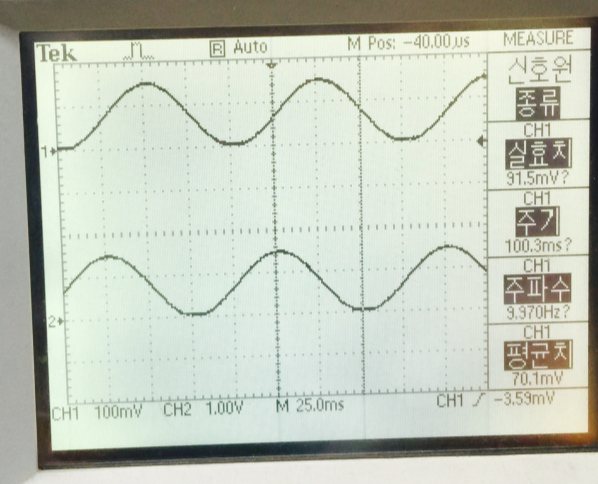


DAC를 통해 Sinusoidal 파형을 출력해보자.

먼저 DAC를 시작하고,(Line 7,8) DAC Value를 계속해서 세팅해주면 된다.(Line 10,11)

1. int time;
2. int main(void) {
3. HAL\_Init();
4. SystemClock\_Config();
5. MX\_GPIO\_Init();
6. MX\_DAC\_Init();
7. HAL\_DAC\_Start( & hdac, DAC\_CHANNEL\_1);
8. HAL\_DAC\_Start( & hdac, DAC\_CHANNEL\_2);
9. while (1) {
10. HAL\_DAC\_SetValue( & hdac, DAC\_CHANNEL\_1, DAC\_ALIGN\_12B\_R, (int16\_t)(1024 \* arm\_sin\_f32(PI \* 2 \* 0.01 f \* time) + 1024));
11. HAL\_DAC\_SetValue( & hdac, DAC\_CHANNEL\_2, DAC\_ALIGN\_12B\_R, (int16\_t)(1024 \* arm\_sin\_f32(PI \* 2 \* 0.01 f \* time + PI / 2) + 1024));
12. time++;
13. HAL\_Delay(1);
14. }
15. }

Arm\_sin\_f32 함수는 arm\_math.h 를 include하고, 설정에서 DSP Lib을 사용체크해야 하는 것을 잊지 맞자. Math.h를 include한 후에 sinf 함수를 사용해도 동일한 결과를 볼 수 있다.



위와 같은 파형이 나오면 성공이다. 혹시나 arm\_math 라이브러리 설정에서 막힌다면 예제 15번을 참고하거나, 28-29page의 TIP을 참고하기 바란다.

* 1. Real Time Operating System

CubeMX는 Middleware로 RealtimeOS 중 하나인 FreeRTOS를 지원한다. 임베디드 펌웨어를 조금 짜 본 사람도 RealtimeOS라는 말 자체가 생소할 수 있다. 일단 Realtime OS란 무엇인가에 대해서는 필자보다 훨씬 잘 정의해 놓은 사람들의 글이 있기 때문에 첨부한다.

**Real-Time OS**

A **real-time operating system** (**RTOS**) is an operating system (OS) intended to serve real-time application process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.

A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is *jitter*. A *hard* real-time operating system has less jitter than a *soft* real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or *generally* meet a *deadline* is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.[[2]](https://en.wikipedia.org/wiki/Real-time_operating_system#cite_note-2)

An RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.[[3]](https://en.wikipedia.org/wiki/Real-time_operating_system#cite_note-3)

간단히 요약하면, 우리가 원하는 타이밍에 원하는 작업을 할 수 있도록 도와주는, 다시 말하면 여러 작업을 동시에 MCU에게 시켰을 때 교통정리를 해주는 프로그램이라는 뜻이다.

이러한 작업을 task-scheduling 이라고 하는데, RTOS를 활용할 때 우리는 task라는 특별한 함수를 정의하게 될 것이다. 혹시 c/c++/java 등을 조금 공부해봤다면 Thread라는 개념에 대해서 들어봤을 것이다. Task는 Thread와 매우 유사한 개념이다. 이 task는 하나일 수도 있고 하나 이상일 수도 있다. 그러나 확실한 것은 이 task 들은 사용자 입장에서는 ‘동시에’ 작동하는 것처럼 보인다. 이것은 생각보다 사용자가 프로그램을 만들 때 큰 장점이다.

예를 들어 아래와 같은 작업을 동시에 처리하는 프로그램을 짠다고 생각해보자.

* 1초에 10번씩 UART2로 자신의 기울기를 전송한다.
* 1초에 200번씩 I2C로 센서에서 기울기를 읽어온다.
* 1초에 100번씩 ADC에서 가변 저항 값을 읽어와서 업데이트한다.
* 1초에 50번씩 GPIO에서 스위치가 눌렸는지 검사하고, 눌렸으면 LED를 켠다.

생각하기 가장 쉬운 방법은 타이머 4개를 돌리고 각각의 Interrupt Service Routine을 정의하는 것이다. 그러나 이 방법은 타이머의 낭비가 심하기 때문에 만약 PWM을 많이 사용해야 하는 상황이라면 적용하기 힘들 수도 있다.

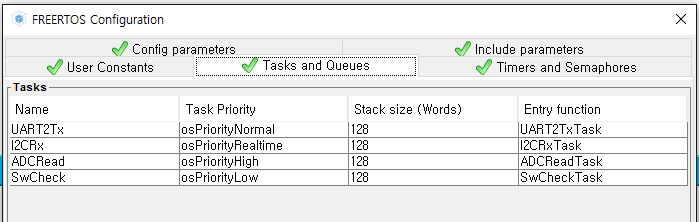
타이머 하나로 처리할 수 있는 방법도 있다. 타이머 하나를 200Hz로 정의하고, 인터럽트가 발생할 때마다 1씩 증가하는 변수를 둔다. 그리고 이 변수에 modular 연산을 해서 아래와 같이 코드를 작성한다.

1. int timerCnt = 0;
2. void timerInterrupt200Hz() {
3. if (timerCnt % 20 == 0) {
4. task10Hz();
5. }
6. if (timerCnt % 4 == 0) {
7. task50Hz();
8. }
9. if (timerCnt % 2 == 0) {
10. task100Hz();
11. }
12. if (timerCnt % 1 == 0) {
13. task200Hz();
14. }
15. timerCnt++;
16. }

그런데, 해당 방법은 구현하기도 손이 많이 가고, 무엇보다 어떤 Task의 주기가 다른 Task의 배수가 아닌 경우에는 사용하기 까다롭다. 그리고 Task의 순위를 사용자가 생각해야 하며, Task의 추가와 제거, 주기 변경 또한 어렵다. 또한 타이머를 한 개 무조건 잡아먹기 때문에, 타이머가 남는 경우가 아니라면 적용할 수 없다.

RTOS를 사용한다고 가정하면 위 작업은 매우 쉬워진다.

CubeMX에서 아래와 같이 설정하고, 아래와 같이 함수를 등록하면 된다.

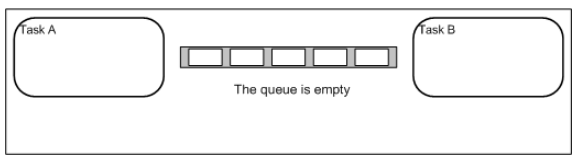


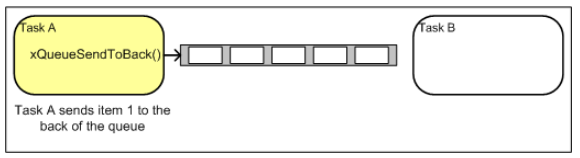
1. void UART2TxTask(void
2. const \* argument) { /\* USER CODE BEGIN 5 \*/ /\* Infinite loop \*/
3. for (;;) {
4. osDelay(100);
5. } /\* USER CODE END 5 \*/
6. } /\* I2CRxTask function \*/
7. void I2CRxTask(void
8. const \* argument) { /\* USER CODE BEGIN I2CRxTask \*/ /\* Infinite loop \*/
9. for (;;) {
10. osDelay(5);
11. } /\* USER CODE END I2CRxTask \*/
12. } /\* ADCReadTask function \*/
13. void ADCReadTask(void
14. const \* argument) { /\* USER CODE BEGIN ADCReadTask \*/ /\* Infinite loop \*/
15. for (;;) {
16. osDelay(10);
17. } /\* USER CODE END ADCReadTask \*/
18. } /\* SwCheckTask function \*/
19. void SwCheckTask(void
20. const \* argument) { /\* USER CODE BEGIN SwCheckTask \*/ /\* Infinite loop \*/
21. for (;;) {
22. osDelay(20);
23. } /\* USER CODE END SwCheckTask \*/
24. }

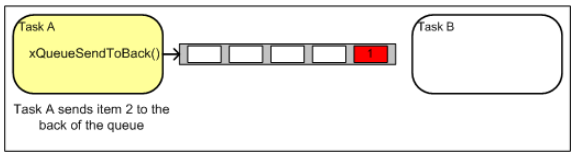
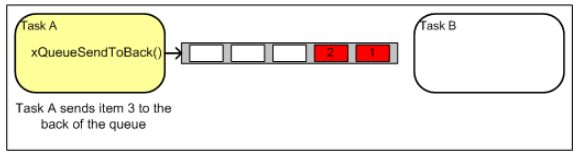
그러나 RTOS도 만능은 아니다. 1000Hz보다 작은 시간단위에서의 정밀한 작업은 Timer에 의존을 해야 한다. 예를 들면 모터 제어를 50kHz로 해야 하는 상황 등에서는 RTOS의 Task는 최소분할시간이 1ms이기 때문에 사용할 수 없다.

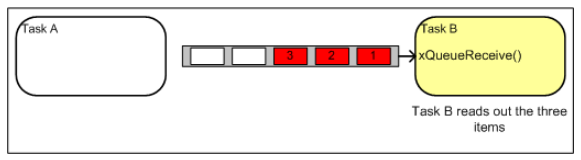
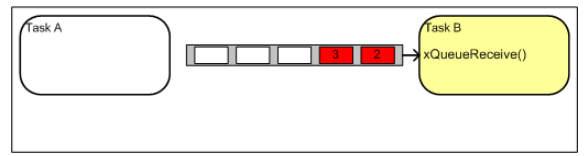
RTOS의 기능은 Task 이외에도 많지만, 그 중 많이 사용되는 기능을 한가지만 더 소개하겠다.

Queue는 Task간 데이터를 주고받을 때 많이 사용된다. 예를 들면 Task1이 Task2에게 어떤 정보를 보내고 싶다. 가장 간단한 방법은 전역 변수에 데이터를 넣어놓는 것인데, Task2는 Task1과 독립적으로 작동하기 때문에, 언제 Task1이 데이터를 넣는지 알 수 없다. 이럴 때 Queue를 사용하면 Queue에 데이터가 남아있는지 검사하는 것 만으로도 새로운 데이터가 Task1으로부터 왔는지 알 수 있다.



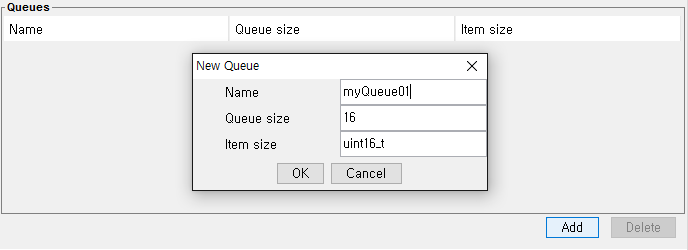


Queue에서 데이터 전송 과정은 위와 같다.

Queue의 생성 또한 CubeMX에서 할 수 있다. Queue를 만들 때 정의해야 하는 것은 Queue에 들어갈 변수의 Type, Queue 전체의 길이이다.

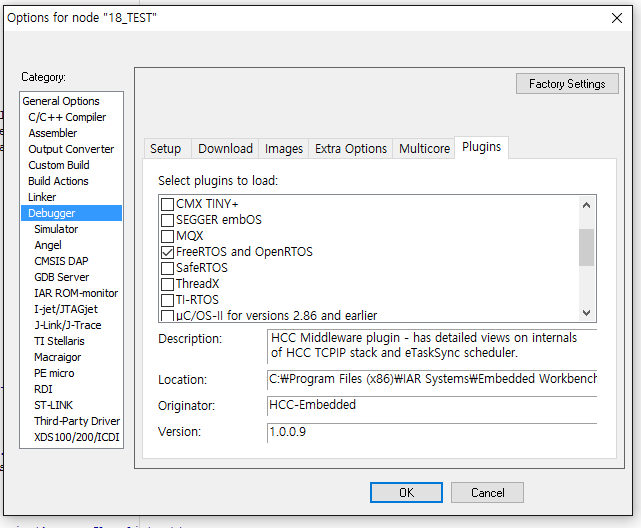


이렇게 생성한 Queue는 Queue 관련 함수를 통해 데이터를 넣고, 뺄 수 있다. 해당 기능은 TX Interrupt를 만들 때 상당히 유용하다.

해당 함수들은 <http://www.freertos.org/a00018.html> 에 나와 있다.

한가지 주의해야 하는 사항은 ISR(Interrupt Service Routine) 안에서 Queue에 내용물을 넣거나 뺄 때는 호출하는 함수가 다르다는 점이다.

그리고 FreeRTOS는 RTOS에 맞는 Debug Mode를 제공한다. 아래와 같이 Debugger Setting을 하면 breakpoint에서의 Queue, Task 상태를 모니터링 할 수 있다.



Task의 경우는 플러그인만 켜면 바로 추적이 가능하다. 그러나 Queue는 Debugger에 잡히도록 함수를 하나 호출해주어야 한다. 원래 Queue가 void\*로 선언되어있기 때문에 그 위치를 알려주어서 지정된 모니터링 포인터에 엮어주는 작업이 필요하기 때문이다.

myQueue01Handle = osMessageCreate(osMessageQ(myQueue01), NULL);

vQueueAddToRegistry(myQueue01Handle, "myqueue01");

아래 붉은 색 함수가 위에 선언된 Queue의 Debugger 모니터링을 가능하게 해주는 함수이다.

FREERTOS는 상당히 광범위한 기능들을 가지고 있기 때문에 짧게 소개하기보다는 아래 링크를 참조하여 어떤 기능이 있는지, 어떤 함수를 호출해야 작동하는지 등을 보기 바란다.

<http://www.freertos.org>

* 1. USB

STM32는 USB규격의 통신을 지원한다. 다시 말하면 STM32가 마우스나, 키보드와 같은 역할을 수행하도록 만들 수 있다는 것이다. STM32F4DISCOVERY 보드 아래에 있는 USB 단자가 해당 기능을 수행하기 위해 붙어있는 것이다. STM32 내부에 USB 서비스를 정의하고, 해당 USB 단자에 Micro 5pin 케이블을 연결하고, 컴퓨터 USB에 연결하면 된다.