001


```c
int main(void)
{
    printf("multp of a and b is : %I64x\n ",multiplicationOfNumber(0x0FFF1111,0x0FFF1111));
    }
```

In C , executable statements must be in scope of a function otherwise will not compile correctly.

Function provide mobilarity and modularity to code project

main() function returns the status of a program to the Parent process, showing the success or faileure of the program. 0 mean Success. Non zero ERROR.

Character group	Description
%c	Input a single character
%d	Input a decimal integer
%e	Input a floating point number
%f	Input a floating point number
%g	Input a floating point number
%h	Input a short integer number
%i	Input a decimal or hexadecimal or octal number
%o	Input an octal number
%p	Input a pointer
%s	Input a string
%u	Input an unsigned interger
%x	Input a hexadecimal number
%ld	Input a long signed integer
%lu	Input a long unsigned integer
%lf	Input a double integer

'\a'	Çan sesi; 7 numaralı *ASCII* karakteri.
'\b'	Geri boşluk (backspace); 8 numaralı *ASCII* karakteri.
'\f'	Sayfa ileri (form feed); 12 numaralı *ASCII* karakteri
'\n'	Aşağı satır (new line); 10 numaralı *ASCII* karakteri.
'\r'	Satır başı karakteri (carriage return); 13 numaralı *ASCII* karakteri.
'\t'	Tab karakteri (tab); 9 numaralı *ASCII* karakteri.
'\v'	Düşey tab karakteri (vertical tab); 11 numaralı *ASCII* karakteri.
'\\'	Ters bölü karakteri.
'\"'	Çift tırnak karakteri.
'\0'	0 numaralı ASCII karakteri; (NULL karakter)

I64x → mean 64 bit 8 byte size data type that represents long long int.

Here is an specific example of type casting in function

Here is our basic function

```
long long int multiplicationOfNumber(int a, int b)
{
    return a*b;
    }
```

Then we called the function in main() function in our program

```
int main(void)
{
    printf("multp of a and b is : %I64x\n ",multiplicationOfNumber(0x0FFF1111,0x0FFF1111));
    }
```

```
multp of a and b is : fffffffff014321
```

but there is something wrong about the output so if we try to calculate this hex numbers on calculator it will give us:

That means there is some data corruption on result. But why ? Because as we declare the multiplication function it take two int argument. Because of int is 4 bytes naturally output will show us only section that consist of 4 bytes. The other remains will not be shown. What do we do now? We should do type casting.

```
long long int multiplicationOfNumber(int a, int b)
{
  return (long long int)a*b;
  }
```

```
multp of a and b is : ffe222ff014321
```

Result is as we expected no data loss solid transaction.

Type casting

1. Implicit casting
2. Explicit casting

Data will be truncate if higher data type is casted by lower data type .

Implicit casting

Compiler does this.

Explicit Casting

Programmer does this.

Inc folder üzerinde projemizdeki header dosyalarını tutuyoruz. Src folder'da bulundurduğumuz bütün souce kodlarını bulunduruyoruz.

Hangi elektronik kart olursa olsun bütün elektronik kartların bir start kodu bulunmak zorunda.

stm kartlarının m3 m4 m7 çekirdeklerinde printf fonksiyonu çalışıyor ve bu printf fonksiyonu `SWO` pini üzerinden çalışıyor.

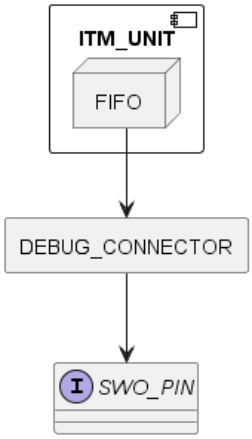
Via **ST Link** debug circutry stm32f4 card can communicate with computer.

Debug → eğer işlemci ile ilgili bir registerın bir hafızadaki konumunu okumak istiyorsan, eğer break point eklemek, işlemciyi durdurmak veya çalıştırmak istiyorsak ve bu tarzdan aktiviteleri gerçekleştirmek istiyorsak `debug` arayüzünü kullanmamız gerekiyor. Burad kullandığımız debug interface'i SWD (Serial Wire Debug) interface'idir.

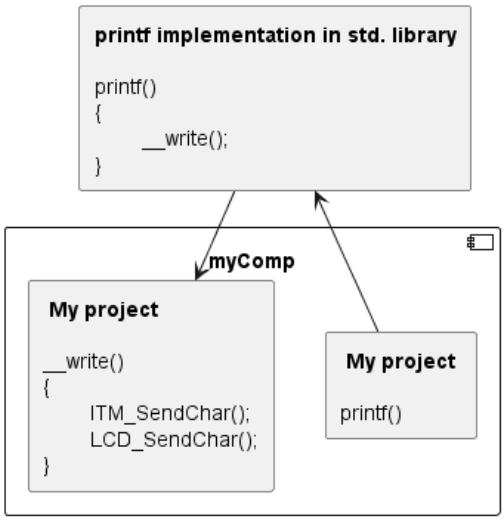
Mikroişlemci ile konuşmak için SWDIO ve SWDCLK pinlerini kontrol ederek konuşmak olayını gerçekleştirebiliriz.

Bir mikro işlemcinin itm unit içersinde fifo değidiğim yapılar vardır. Bunlara `buffer ya da registerlar` da denir. Buradaki amacımız yazdığımız karakter bu fifo yapısının içersine yazmamız gerekiyor. Bu fifo'dan veri okumamızın bir yolu var o da swd debug connector aracılığı ile bilgisayar üzerinde mikro işlemci ile bağlantı kurmak. Bu bağlantı **swo** pini üzerinden olacaktır.

Genel görüntü şeması



Peki burada stm32f4 kartımızın itm birimindeki fifo adrresine kendi bilgisayarımızdan printf() fonksiyonu ile nasıl veri gönderebilirm? Burada ise printf() fonksiyonunun stdlibrar'sindeki __write() fonksiyonundaki bir değişken yerine itm_write olarak oluşturduğumuz fonksiyon ile yer değiştiriyoruz. Böylelikler artık stdlibrarden kullandığımız printf() fonksiyonu herhangi bir pointer kullanarak ekrana çıktı bakmaktansa kullandığımız fonksiyon sayesinde stm32f4 kartın mikroişlemcisindeki fifo adresine veri yazma işlemini gerçekleştirmiş olacağız. Genel olarak sistemin işleyiş şekli görselde anlatılmıştır.



ITM_SendChar() fonskiyonumuzun içindeki kodlar:

printf() fonskiyonun içine implemente ettiğimiz bu kod parçacığı sayesinde itm birimimizdeki register'ın içersine printf() fonksiyonu içersindeki karakter versini st-link debugger üzerinden gönderebiliyoruz.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//          Implementation of printf like feature using ARM Cortex M3/M4/ ITM functionality
//          This function will not work for ARM Cortex M0/M0+
//          If you are using Cortex M0, then you can use semihosting feature of openOCD
////////////////////////////////////////////////////////////////////////////////////////////////////


//Debug Exception and Monitor Control Register base address
#define DEMCR             *((volatile uint32_t*) 0xE000EDFCU )

/* ITM register addresses */
#define ITM_STIMULUS_PORT0    *((volatile uint32_t*) 0xE0000000 )
#define ITM_TRACE_EN            *((volatile uint32_t*) 0xE0000E00 )

void ITM_SendChar(uint8_t ch)
{

  //Enable TRCENA
  DEMCR |= ( 1 << 24);

  //enable stimulus port 0
  ITM_TRACE_EN |= ( 1 << 0);

  // read FIFO status in bit [0]:
  while(!(ITM_STIMULUS_PORT0 & 1));

  //Write to ITM stimulus port0
  ITM_STIMULUS_PORT0 = ch;
}
```

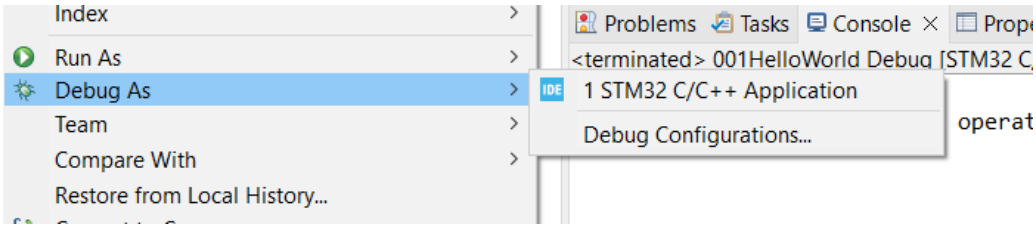
Bu kodu syscalls.c içindeki __write() adlı fonksiyonun içine yazdığımız zaman işlem tamamlanmış olacak.

Kodun kartta çalıştırılması:

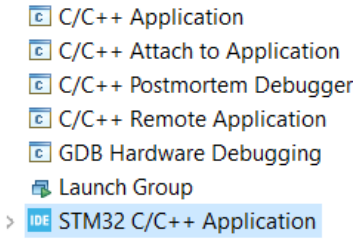
1. Yazılan main kodu itm birimindeki register'a yazılacak Hello World satırı

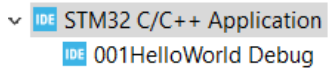
```
 */

#include <stdint.h>

int main(void)
{
    printf("Hello World\n");
    for(;;);
}
```

2. Debug configürasyon ayarları

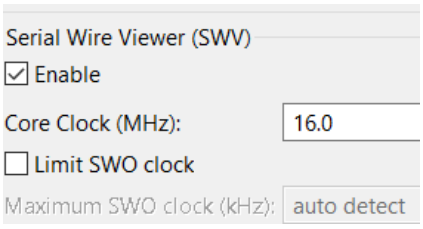


Burada stm32 c/c++ application çift tıklıyoruz



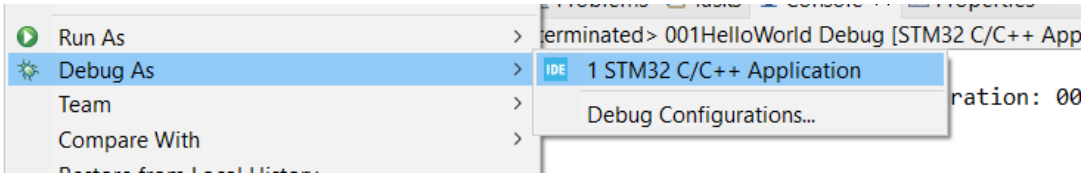


Serial Wire Viewer aktive edilecek

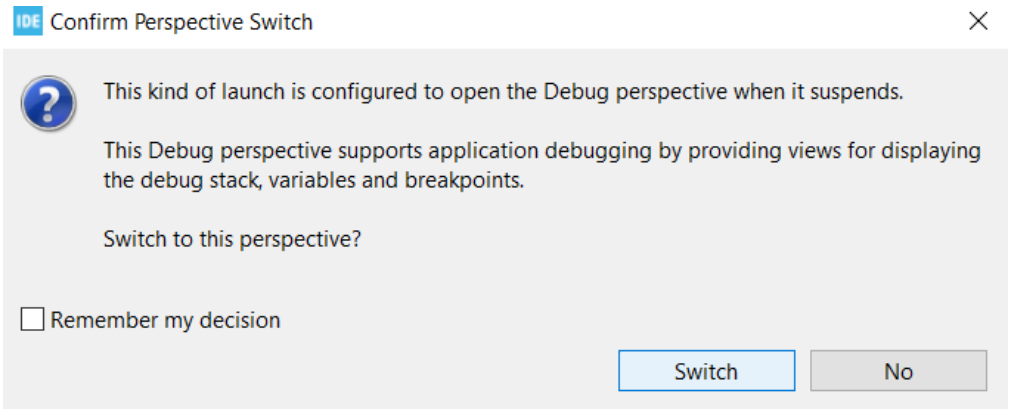


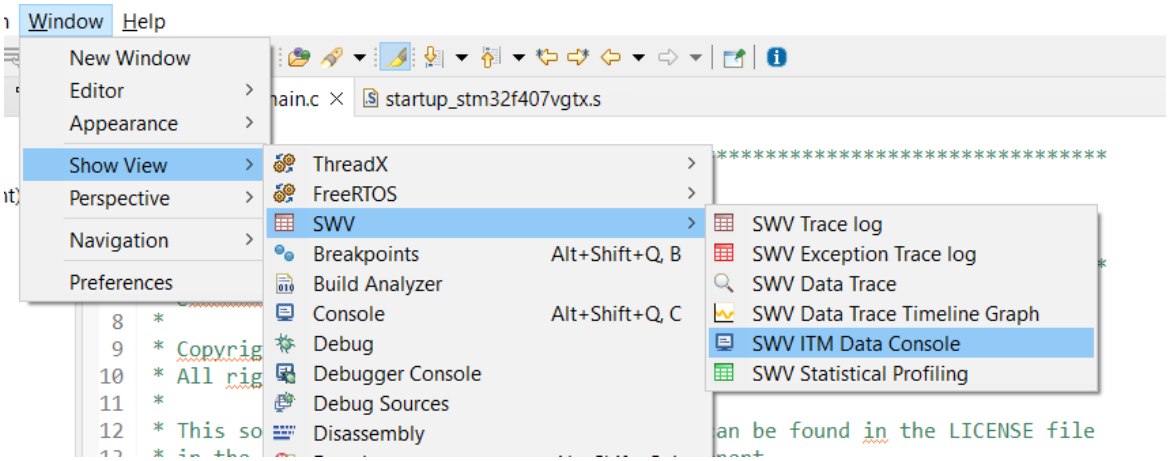
Ardından kapatın

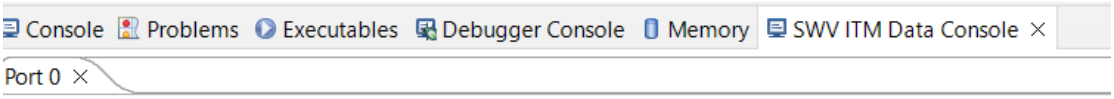
Debug yapın

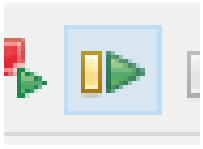


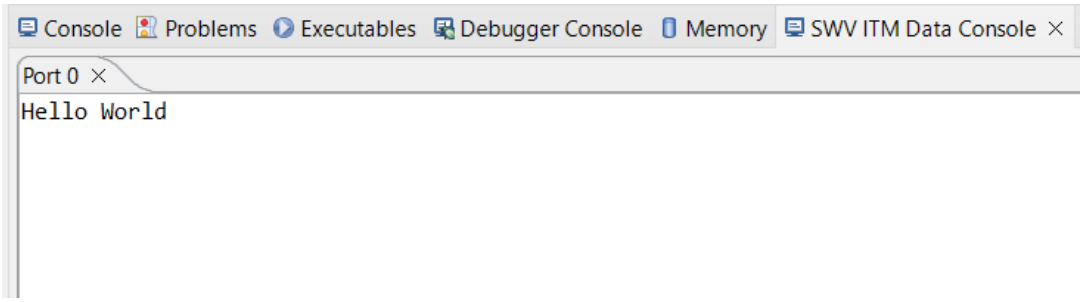
Switch'e tıkla











Bu şekilde stm32f4 cihazımızdaki ilk uygulamayı oluşturmuş olduk. ITM birim register'ına hello world yazdırarak swv ile pin üzerinde gelen hellow world verisini gözlemlemiş olduk

Fakat bunları doğru bir şekilde çalıştırabilmek için cross compilation işlemini gerçekleştirmemiz gerekiyor. Zaten stmcube ide ile birlikte hazır olarak cross compilation eklentisi de geliyor.

Cross Compilation Nedir?

Executable bir işlemin kullanmak istediğimiz bir cihazda oluşturmak istiyorsak ve ilk olarak bu execution kodları karşı taraftaki cihazımızda değil de kendi bilgisayarımızda derleniyorsa ve ardından derleme sonucu oluşan execute dosyaları karta gidiyorsa bu işlemin adında cross compilation işlemi gerçekleştire ise cross compiler denir. Burada executable dosyadan kastımız ".elf/.bin/.hex" gibi dosyalardır.

.elf → debug işlemi sırasında oluşan executable dosyasıdır.

.bin ve .hex → genellikle üretim sırasında ortaya çıkan saf binary dosyalarıdır.

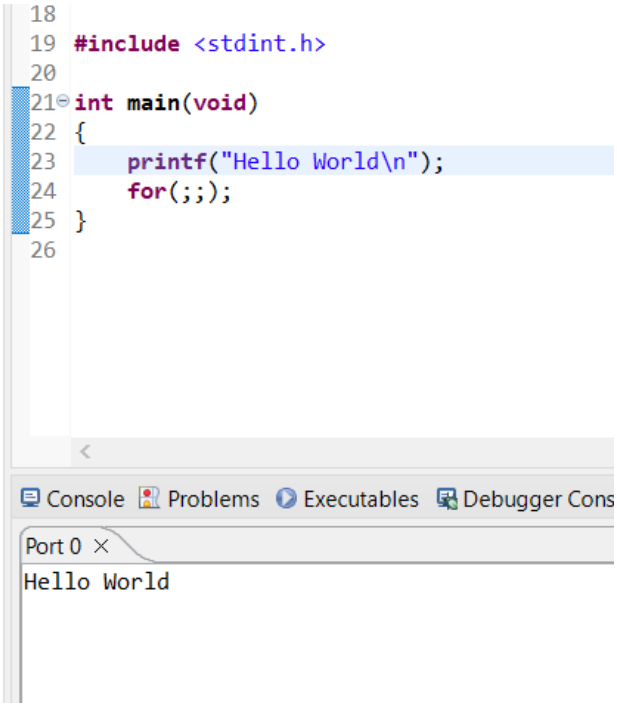
Örneğin müşterinize oluşturduğunu projenin executable dosyasını verecekseniz. Bu dosya formatını .bin ve .hex formatında vermeniz gerekir. Aksi takdirde .elf programınız hakkındaki bütün debug bilgilerini bulunduğu için ve neredeyse herkes bir .elf analizöre erişebilme durumu olduğu için bu analizörü kullanarak .elf dosyanızı okuduğu zaman projenizde kullandığınız bütün kod bilgilerine erişme fırsatı olur ve bilgileriniz çalınabilir.

Burada da görüldüğü üzere HelloWorld için oluşturduğumuz projenin .elf executable dosyası oluşmuş.

```
make -j8 all
arm-none-eabi-size   001HelloWorld.elf
   text    data     bss     dec     hex filename
   3792     108    1588    5488    1570 001HelloWorld.elf
Finished building: default.size.stdout


01:00:09 Build Finished. 0 errors, 0 warnings. (took 183ms)
```

Eğer bir program bilgisayarda derlenip bilgisayarda çalışıyorsa bu durumu **native compilation** deniyor.

Ön ayarlamalar



İşlemler sonucunda swo pini üzerinden hello world yazımızı almış olduk

Build process

1. Processing
2. Parsing
3. Producing object file(s)
4. Linking object file(s)
5. Producing final executable
6. Post processing of final executable

Yüksek seviyeli yazdığımız bir kodun çalışmasını sağlamak için bu aşamalardan geçilmesi lazım. Bütün bu süreç toplamda 2 aşamaya bölünebilir.

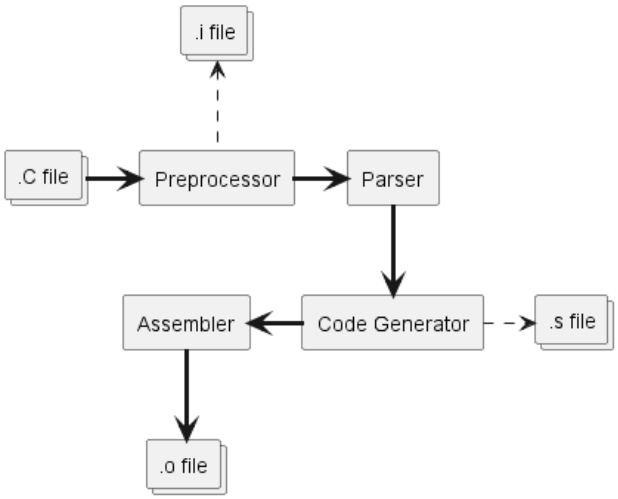
1. Compile aşaması
2. Linking aşaması

Compile aşaması

İlk olarak .c uzantılı kaynak kodumuz **preprocessing** aşamasından geçmekte. Bu ön işlem aşaması her .c uzantılı dosya için .i uzantılı dosyalar oluşturur. Bu aşama .h include uzantılı dosyaların ve C makrolarının çözümlenmesi sonucunda .i uzantılı dosyaların oluştuğu aşamadır. Sonraki aşama kontrol ise **parser** kontrolüdür. Bu kontrolde yazdığımız .c uzantılı kod satır satır belli standartlara göre parse edilerek kodların sytax çözümlemesi yapılıyor. Eğer programın bu aşamasında herhangi bir sytax hatası veya bir sıkıntı oluşursa bu kısımda hatalar programcıya rapor ediliyor. Eğer bu aşamada da hiç bir sıkıntı oluşmazsa oluşturduğumuz kodlar bir code generator yardımıyla .s uzantılı dosyaya dönüştürülüyor. Yani bu durumda .i uzantılı dosyamız code generator yazılım motoru yardımıyla .s uzantılı bir dosyaya dönüştürülüyor. Aslında burada kullandığımız yüksek seviyeli bir dil olan C dili ile yazdığımız program daha düşük ve bilgisayar diline daha yakın bir dil olan makine dilinine dönüştürülüyor.

`.i → .s`

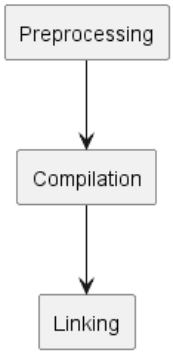
Son olarak makine dilinden oluşan .s uzantılı dosya assembler motoru ile makinenin tam olarak anlayabileceği dil olan 1 ve 0 lardan oluşan bir dosyaya dönüştürüyor ve oluşturduğumuz kodu bilgisayar anlamış oluyor.



Linking aşaması

Bu aşamada birden fazla yeri değiştirilebilir(relocateable) obje dosyasının (`.o` uzantılı dosya) ortak bir executable file formatına dönüştürüldüğü kısımdır. Örneğin 5 adet .c uzantılı dosya için 5 adet .o uzantılı dosyalar oluşturulacak her bir .o uzantılı dosyanın her birinin kendine ait bir makine kodu bulunacaktır. Burada **linker** yapımız bütün bu obje dosyalarını alıp hepsini ortak bir executable dosyaya dönüştürüyor. Burada executable dosyamızın formatı `.elf` debug formatında olacaktır. Son olarak object copy toolları kullanılarak `.elf` uzantılı dosya `.bin` ve `.hex` gibi dosyalara dönüştürülüyor. Bu son kısma **post processing** aşaması olarak adlandırılıyor.

> Sonuç olarak bilgisayarda yazdığımız .c uzantılı dosyanın executable file olana kadar geçen süreci tanımlamış olduk.



I2C haberleşme protokolü

Bir cihazdan 2 hat üzerinden diğer alt cihazlara bağlanabilecek. Amacı düşük hızlı çevre birimleri anakartlar, cep telefonları, gömülü sistemler gibi elektronik gibi elektronik cihazlara daha az kablo ile bağlanılabilmesine olanak sağlamaktadır. Cihaza giden birçok veri tek bir hat üzerinden bağlanmaktadır. I2C de sadece 2 hat bulunmaktadır. SDA (Serial Data Line) ve SCL (Serial Clock Line) hattı. Ayrıca bu iletişim hattı **pull-up** direncine ihtiyaç duyar. Direncin bir ucuna gerilim verilip diğer ucunun hatta bağlanması durumuna **pull-up** direnci denir. Bu I2C hattının çalışabilmesi için pull-up direncine ihtiyacı vardır. Bu direnç genellikle 5V veya 3.3V gerilim değeri verilip direnci **4k7** civarında olmaktadır.

- I2C düşük bant genişliğine sahiptir ve kısa mesafelerde çalışır.
- I2C hattı SDA hattının logic high seviyesinde logic low seviyesine düşmesi ile başlar. Aynı şekilde logic low seviyesinden logic high seviyesine çıkması ile sonlanır. SDA hattının haberleşmeyi başlatabilmesi için SCL hattı da high olmalıdır.
- İletişim kurulacak her bir cihazın kendine ait bir adresi var ve i2c ile master'dan slave cihazlara bu adres üzerinden iletişim sağlanıyor.

PLL (Phase Locked Loop)

Normalde dahili osilatörümüzün frekansı 8 MHz civarında. PLL sayesinde bu değeri 168 MHz'e kadar çıkarabiliyoruz.

PLL

- PLL - M
- PLL - N
- PLL - P

System Clock (Sysclk) =

$$SystemClock_{frequency} = \frac{((\frac{HSE}{PLL_M}) * PLL_N)}{PLL_P}$$

Örnek :

PLL_M = 4, PLL_N = 168, PLL_P = 4 olursa;

$$SystemClock_{frequency} = \frac{\frac{8MHz}{4} * 168}{4} = 84MHz$$

olarak buluruz

Register seviyesinde RCC ayarlarının yapılması

> 💡 **Register ne demek?**
> Kaydedici demektir. Bir mikrodenetleyiciye belli işlemler yaptırmak istiyorsak o denetleyiciye belli işlemler yapabilecek noktalarına belli ayarlamaların yapıldığı kısımlardır. Nasıl telefonlarımızı kullanmadan önce bir kurulum ayarı yapıyorsak, mikrodeneyleyiciye de kullanmadan önce bir kurulum ayarı yapmamız gerekiyor.

bazı kutucuklarda "r" bazılarında "rw" yazıyor. Eğer "r" ise sadece okunabilir "rw" ise hem okunup hem de yazılabilir anlamı olduğu ortaya çıkıyor.

PLL açılmadan önce configurasyon ayarlamalarının yapılması gerekiyor.
PLL ayarlamaları için RCC_Config() adında bir method oluşturuyoruz. Bu metod ile register düzeyinde pll ve system clock ayarlarını gerçekleştirmiş olucaz.

```
void RCC_Config(void){
```

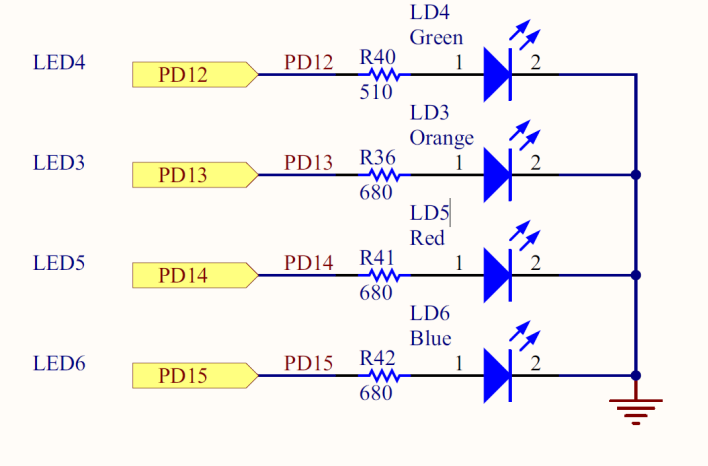
```
  RCC->CR &= ~(1<<0); // HSI OFF
  RCC->CR |= 1 << 16; // HSE ON
  while(!(RCC->CR & 1<<17)); // WAIT HSE ACTIVE
  RCC->CR |= 1 << 19; // CLOCK SECURITY ON
  // PLL ayarlamalarının yapılması ile uğraşıyoruz.
//  RCC->PLLCFGR &= ~(1<<0); // PLLM0 0 div : 4
//  RCC->PLLCFGR &= ~(1<<1); // PLLM1 0 div : 4
//  RCC->PLLCFGR |= ~(1<<2); // PLLM2 1 div : 4
//  RCC->PLLCFGR &= ~(1<<3); // PLLM3 0 div : 4
//  RCC->PLLCFGR &= ~(1<<4); // PLLM4 0 div : 4
//  RCC->PLLCFGR &= ~(1<<5); // PLLM5 0 div : 4
  // daha kolay bir yol ile pll_m değerini ayarlama
  //RCC->PLLCFGR &= ~(31<<1); // ilk 5 biti 1 yap sonra tersini alıp sıfırlama
  // bit kısımlarını teker teker sıfırlamaktansa bütün 32 bitlik kısmı sıfırlama işlemi yapılarak devam edilir
  RCC->PLLCFGR = 0x00000000; // buradaki 8 adet sıfır 32 bit'i temsil eder çünkü her basamak 4 bitlik alanı temsil ediyor
  RCC->PLLCFGR = (1<<22); // HSE SELECT
  RCC->PLLCFGR |= (4<<0); // PLLM 4
  RCC->PLLCFGR |= (168<<6); // PLLN 168

  // configuring PLLP 2
  RCC->PLLCFGR &= ~(1<<16);
  RCC->PLLCFGR &= ~(1<<17);

  RCC->CR |= (1<<24); // PLL ON

  while(!(RCC->CR & (1<<25))); // PLL READY

  // System clock is PLL
  RCC->CFGR &= ~(1<<0);
  RCC->CFGR |= (1<<1);

  while(!(RCC->CFGR & (1<<1))); // IS SYSTEM CLOCK SET TO PLL

}

int main(void)
{
//  systemClock = SystemCoreClock;
//  RCC_DeInit(); // HSIEN PLL OFF
//  SystemCoreClockUpdate(); // 16 000 000
//  systemClock = SystemCoreClock;

  RCC_Config();
  SystemCoreClockUpdate();
  systemClock = SystemCoreClock;

  while (1)
  {

  }
}
```

Burada temel amaç belli bitlere belli bit değerlerini atayarak sistem içersindeki pll ve clock ayarlarını yapmış oluyoruz.

GPIO işlemleri

Burada en önemli şey işlem yapabilmek için gpio’ları aktive etmemiz gerekmektedir. GPIO’lar clock hattında çalışırlar ve clock bilgisine ihtiyaç duyarlar. Yani bir gpio çalıştıracaksak onu çalıştırmadan önce o portun bağlı olduğu clock configurasyon ayarlarını yapıp, clock hattını aktive etmemiz gerekiyor.

stm32f4 kartımızın user manuel dökümanına baktığımız zaman hangi ledin hangi porta bağlı olduğu açıkça belli olmaktadır.



Görüldüğü gibi ledler PDXX olarak adlandırılmış pinlere bağlıdır. Buradaki ‘D’ harfi bu ledlerin D portunda AHB1 clock hattına bağlı olduğunu göstermektedir. Peki biz GPIOD portunun clock hattına clock bilgilerini nasıl göndereceğiz?

Gpiod pinlerimizin özelliklerin bulunduğu bir structer yapısı oluşturuyoruz. Bu yapının ismi GPIO_InitTypeDef GPIO_InitStruct; şeklinde olacaktır. GPIO configurasyonları artık GPIO_InitStructer nesnesi üzerinden yapabileceğiz.

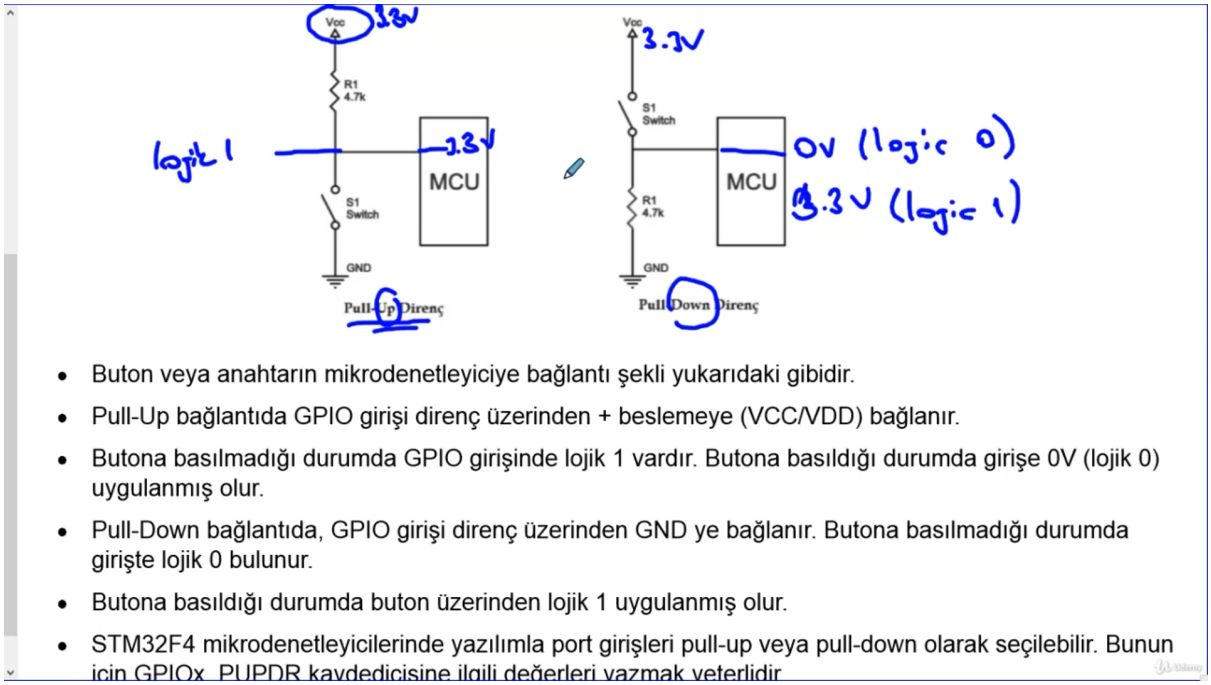
```
GPIO_InitTypeDef GPIO_InitStruct;

void GPIO_Config(void)
{
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD,ENABLE);
  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;

  GPIO_Init(GPIOD,&GPIO_InitStruct);
  }

int main(void)
{
  GPIO_Config();
  while (1)
  {
    GPIO_SetBits(GPIOD,GPIO_Pin_12);
    GPIO_SetBits(GPIOD,GPIO_Pin_13);
    GPIO_SetBits(GPIOD,GPIO_Pin_14);
    GPIO_SetBits(GPIOD,GPIO_Pin_15);
  }
}
```

Ledleri güzgün olarak çalıştırabilmek için GPIOD portumuzu yukarıdaki stucture içersinde tanımlayıp clock ayarlarını gerçekleştirmemiz gerekiyor. Bu ayarları tanımladıktan sonra GPIO_Config() isimli bir fonksiyon içersinde atıyoruz. Ardından main() programı çalıştırdığımız fonksiyonun içersinde led işlemlerimizi yapmadan önce bu fonksiyonu çağırıyoruz. While döngüsü içersinde ledlerimiz durmadan yanık bir şekilde kalmasını sağlıyoruz.

GPIO port moder kaydedicisi seçtiğimiz bir pini nasıl kullanacağımızı seçtiğimiz kısımdır. Yani bir pini input, output, analog ya da alternate function (spi,i2c vb) için kullanacağımızı burada belirleriz.

Buton kontrol

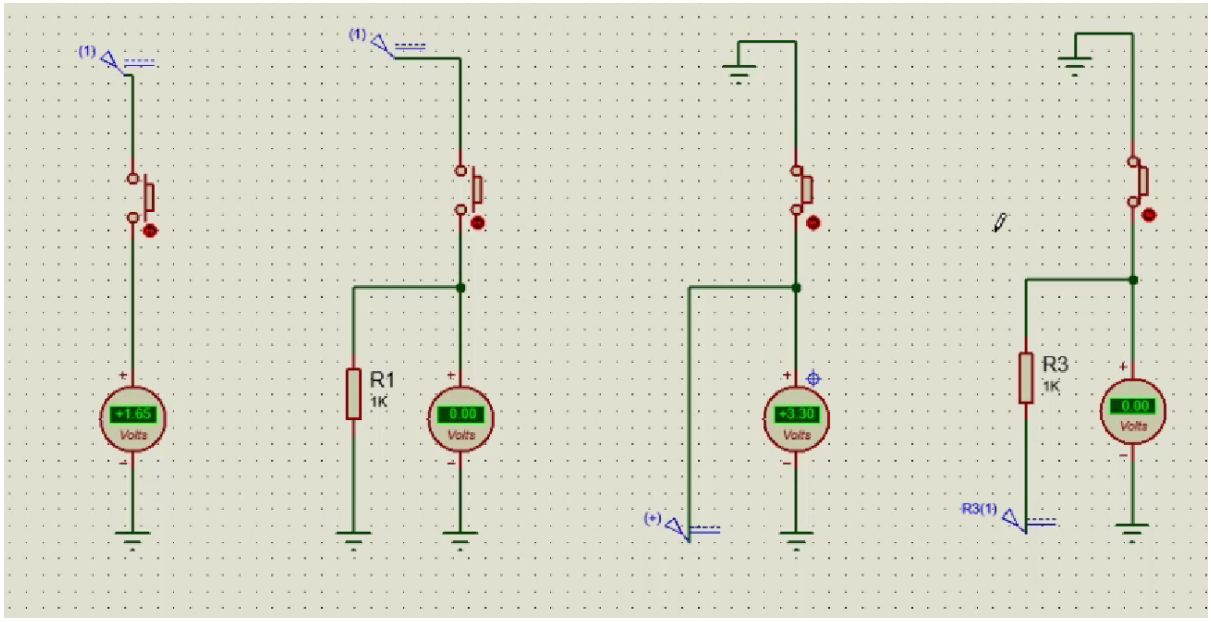
Pullup ve pulldown dirençleri



- Buton veya anahtarın mikrodenetleyiciye bağlantı şekli yukarıdaki gibidir.
- Pull-Up bağlantıda GPIO girişi direnç üzerinden + beslemeye (VCC/VDD) bağlanır.
- Butona basılmadığı durumda GPIO girişinde lojik 1 vardır. Butona basıldığı durumda girişe 0V (lojik 0) uygulanmış olur.
- Pull-Down bağlantıda, GPIO girişi direnç üzerinden GND ye bağlanır. Butona basılmadığı durumda girişte lojik 0 bulunur.
- Butona basıldığı durumda buton üzerinden lojik 1 uygulanmış olur.
- STM32F4 mikrodenetleyicilerinde yazılımla port girişleri pull-up veya pull-down olarak seçilebilir. Bunun için GPIOx_PUPDR kaydedicisine ilgili değerleri yazmak yeterlidir.

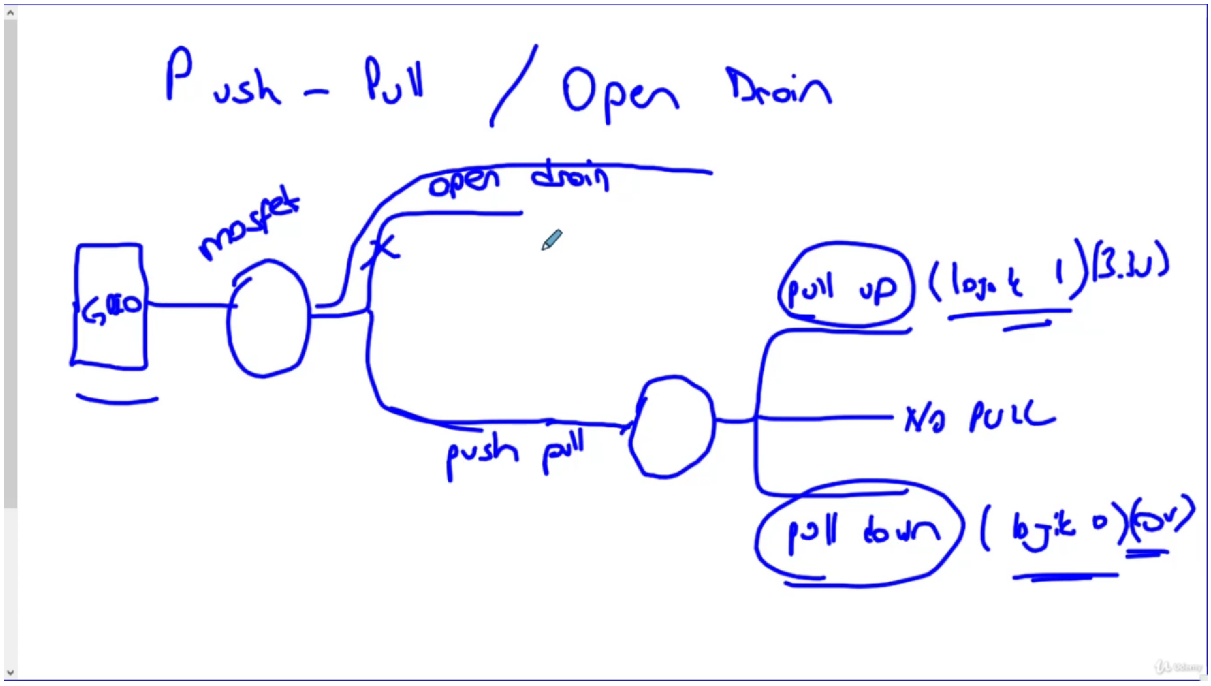
pull direncinde buton kapalıyken mcu pin girişinde logic 1 seviyesinde bir gerilim gözlenmektedir. Switcth kapalı duruma geldiği zamansa akım toprak yönünden akacağından gerilim logic 0 seviyesinde olacaktır. Pull down durumunda ise ilk olarak logic 0 seviyesi fakat anahtar açıldığı zaman logic 1 seviyesine çıkmaktadır.

Bu resimde pull-up pull-down dirençlerinin sistem üzerindeki etkisi anlatılmıştır.



Open drain ve push pull

Mikrokontrolcümüzün pininin çıkışında mosfet bulunmaktadır. Bu mosfet üzerinde geçen akım iki yol ayrılır biri open drain biri de push pull. Open drain tarafından giderse akım olduğu gibi direkt hedefine ulaşır. Fakat push pull yönünden giderse tekrardan bir mosfet ile karşılaşır ve bu mosfet 2 akıma 3 gidiş seçeneği daha sunar bunlar pull-up, pull-down ve no-pull seçenekleridir.

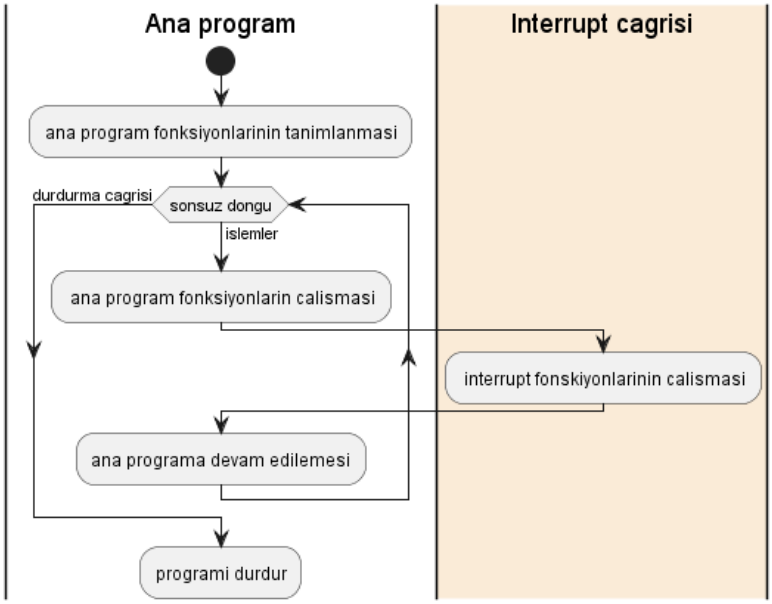


Interrupt nedir ve nasıl kullanılır?

Önceliği yüksek işlerin mikrodenetleyici tarafında ana programın akışını keserek yapılmasına interrput (kesme) denir. Eğer bir kesme kaynağından mikrodenetleyiciye bir uyarı gelirse mikrodenetleyici yapmakta olduğu işi bekletir, kesme alt programına gider ve oradaki işlemleri gerçekleştirir. Daha sonra da ana program işlemlerine kaldığı yerden devam eder.

Kesmeler genellikle çok hızlı yapılması gereken işlemlerde anlık tepki verilmesi gerekn yerlerde kullanırız.

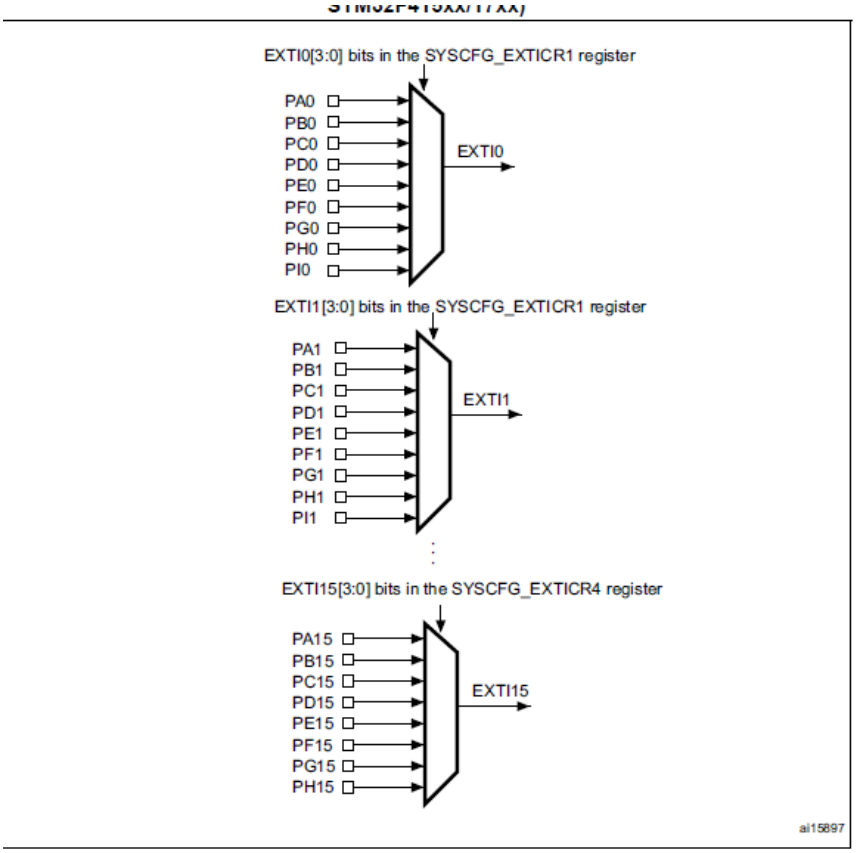
Interrupt sisteminin çalışma mantığı aşağıdaki gibidir;



Örnek vermek gerekirse araba sürerken anlık olarak frene basılması. Araba sürme işlemi düzenli bir döngü de yapılırken öncelikli olarak acil bir durumda frene basma işleminin gerçekleştirilmesi örnek verilebilir.

Harici bir kaynaktan oluşan olaylardan dolayı meydana gelen kesmelere, harici kesmeler denir. STM32F4 discovery kartı 23 adet harici kaynaktan kesme hkabul edebilmektedir. Harici kaynak olarak, dış ortamdan pinler vasıtasıyla gelecek kesme ve kendi içindeki donanımlardan gelen kesmeleri anlayabiliriz. (**EXTERNAL INTERRUPT**)

External interrupt haritası



PA0,PB0,PC0,…. gibi pinler EXTI0 hattına bağlı olduğu görülebiliyor. Bu da demek oluyor ki **A portunun 0. pini ile B portunun 0. pinini aynı anda kullanamam çünkü ikisi de EXTI0 hattına bağlı.**

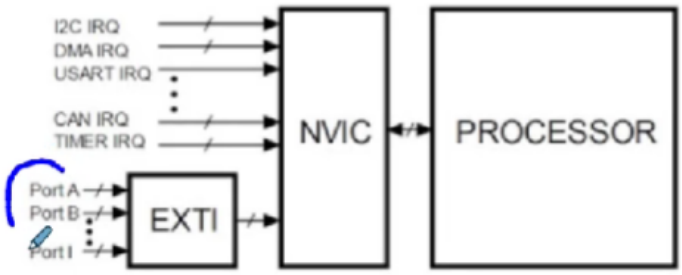
NVIC (Nested Vectored Interrupt Controller)

Karmaşık kesme işlemlerinin işlemciye sürekli yük getirmemesi içn işlemci içerisinde özel bir donanım boğu oluşturulmuştur. Bu donanıma kesme kontrolörü (interrupt controller) adı verilir.

Kesme kontrolörü haklı bir sebeple gelen kesme isteği neticesinde **düzgün işleyen programı askıya alarak kesme fonksiyonu (interrupt function) olarak adlandırılan özel kod parçasını işlemeye başlar.** Kesme fonksiyonun işletilmesinin biitiminde program kaldığı yerden çalışmaya devam eder.

NVIC kontrolör mikroişlemci içerisindeki önemli donanım kesmelerini (DMA istekleri, USART, CAN, I2C ve Timer gibi donanım kesmeleri) ve ayrıca External Interrupt/Event Controller (EXTI) adı verilen donanım vasıtasıyla portlardan gelen kesmeleri kontrol eder.

Kesme isteklerinin işletilmesi basit bir şekilde gösteren blok diyagramı:

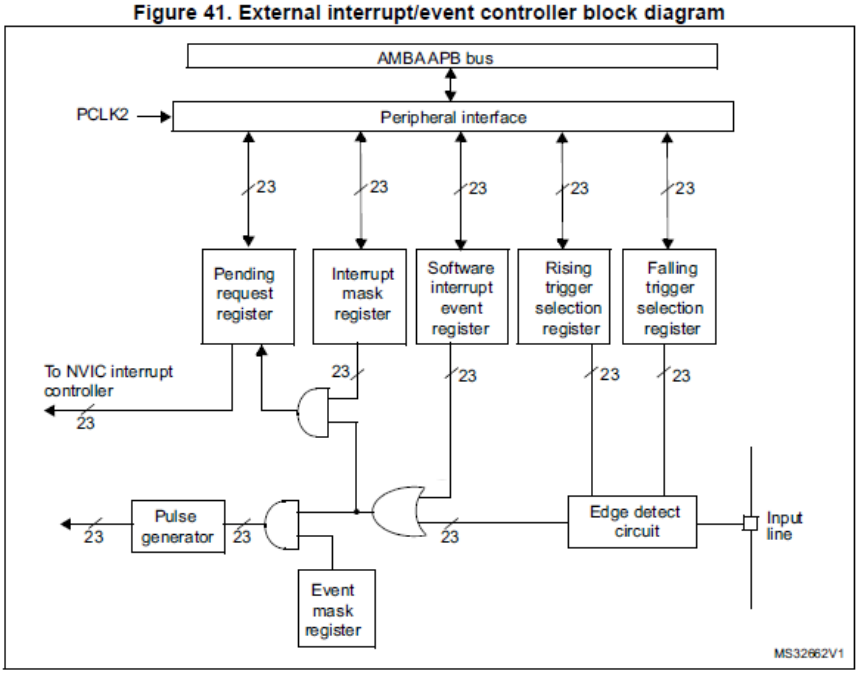


Aşağıdaki blok diyagarmı ile birlikte çeşitrli interrupt ayarlama bitlerinin çalışma sırası gösterilmiştir.

EXTI block diagram

Figure 41 shows the block diagram.

Figure 41. External interrupt/event controller block diagram



- SYSCFG_EXTICRx (Peripheral interface) registerleri ile interrupt olarak algılanacak port/pin (bit) seçilir.
- Kesmenin yükselen/düşen kenar veya her ikisinden gelen kesme seçilir. Durumda bir değişiklik olduğunda bunu kesme olarak algılanmasını sağlayan EXTI_IMR registerinin biti 1 yapılır.
- EXTI_PR kaydedicisi “1” yapılarak kesme isteğinin oluştuğu bildirilir.
- NVIC kontrollör ilgili interrupt rutinini yönlendirir.
- Rutin içersinde komutlar işlendikten sonra tekrar ana programam dönülür.
- **Event** olarak isimlendirilen kaydedici mikrodenetleyiciyi stanby(bekleme) durumundan çıkaran (uyandırma) işlemi de yapabilmektedir. EXTI_EMR kaydedicisi ile bu sağlanmaktadır.