Mühendisler İçin Arduino Kaynak Kodu İncelemesi

Gökhan Dökmetaş

Mikrodenetleyiciler dersi için yardımcı kaynak

Benimle iletişime geçmek için aşağıdaki e-posta adresini kullanabilirsiniz,

gokhandokmetas0@gmail.com

Ayrıca Facebook grubumuza üye olarak bütün çalışmalarımızdan haberdar olabilirsiniz.

https://www.facebook.com/groups/lojikprob/

Söz Başı

Ülkemizde mikrodenetleyici ve gömülü sistemler eğitiminin Arduino ile sınırlı kalması veya Arduino'dan pek ileri gidememesi bu alanda pek fazla ileri seviye kaynak olmamasından dolayıdır. www.lojikprob.com internet sayfamda yazdığım "Arduino Kaynak Kodu İncelemesi" adı altındaki yazı dizisini elektronik kitap haline getirerek meslek yüksekokulları ve mühendislik fakültelerinde kaynak doküman olarak kullanılmasını amaçlıyorum. Kendi firmamızda arkeojeofizik ve jeofizik ölçüm ve görüntüleme sistemlerinin argesini yapmaktayım. Aynı zamanda gömülü sistemler, yazılım ve elektronik eğitiminde kullanılmak üzere kar amacı gütmeden kaynak hazırlıyorum. Bu elektronik kitapçık da ücretsiz olup ders notu olarak veya okumak için basılabilir, çoğaltılabilir ve paylaşılabilir. Ödev, proje ve tezlerde kaynak göstermek zorunludur. Her ne sebeple olursa olsun belgenin bütünlüğü bozulamaz, değiştirilemez ve yazarın ismi belgeden çıkartılamaz. Böyle bir duruma şahit olanların yazarla iletişime geçmesi rica olunur.

Gökhan DÖKMETAŞ

-1- Wiring.h Dosyası

Günümüzde pek çok mühendis adayının hatta mezun mühendisin bile öğrenmeye çalıştığı (!) Arduino platformuna biz daha derinden bir göz gezdireceğiz. Böylelikle Arduino programlamayı değil Arduino'nun nasıl yapıldığını görme şansınız olacak. Kaynak kodunun tamamını incelemek için zamanım olmadığından sadece önemli gördüğüm ve AVR programlamada bizi ilgilendiren kısımlarından bahsedeceğim. Arduino programcısı olup AVR'ye geçen fakat Arduino fonksiyonlarına alışmış kişiler de bu fonksiyonların içyapısını öğrenince rahatça AVR üzerinde programlama yapabilecektir.

Bu yazı dizisini daha rahat anlayabilmek için bilmeyenlerin yazdığım Arduino Eğitim Kitabı ve C ile AVR Programlama yazı dizilerini okumasını rica ederim. Bu çalışma her ne kadar önemli olsa da giriş seviyesine hitap eder nitelikte değildir. Hiç bilmeden bu yazıları okuyup da Arduino'nun kaynak kodunu anlamanız beklenemez. Yine de yer yer derslerden ve kitaptan referans göstererek anlamanız için basitleştireceğim.

Arduino'nun anlatılmayan hikâyesi yazımızda Hernando Barragan'ın dilinden Arduino'nun nasıl ortaya çıktığını okuyabilirsiniz. Bu yazıları okumanız Arduino'ya olan bakışınızı değiştirecektir.

http://www.lojikprob.com/embedded/arduinonun-anlatilmayan-hikayesi-1-wiringin-dogusu/

http://www.lojikprob.com/embedded/arduinonun-anlatilmayan-hikayesi-2-arduinonun-dogusu/

Arduino kaynak kodunun pek çeşitli sürümleri mevcuttur. En güncel sürümü değil en eski ve sade sürümü incelemeyi hedeflediğim için elimde fazla seçenek olmasa da işin ortasını bulma gayretinde oldum. Eski sürümleri incelediğimde Arduino namına pek bir şey göremediğimi söylemem gerekir. Tamamı Wiring tabanlı olan sistemde kendi ürettikleri ucuz kartı çalıştırmaya yönelik bir kütüphane dosyasından başka Arduino'ya özgü bir şey göremiyoruz.

O yüzden Arduino Core adı verilen Arduino çekirdeğinin ilk sürümünü incelemekle işe başlayalım. Hem günümüz Arduino kaynak koduna benzerlik gösterecek hem de yeteri kadar sade bir kod olacak. Yıllar geçtikçe bu kodun ortalarına yapılan eklemeler ve çıkarmalar ile kod kalabalığı oldukça artmış olduğundan biraz da mecburiyetten ilk sürümleri tercih ettiğimizi söyleyelim. İşin içinden çıkamadıkça çalışmamız bir işe yaramaz.

Benim tercih ettiğim Arduino kaynak kodu şu bağlantıdadır.

https://github.com/arduino/ArduinoCore-avr/tree/master-older

Sizin bu kodu indirip Notepad++ gibi bir kelime işlem programında takip etmeniz anlamanız için daha faydalı olcaktır.

Arduino'nun Github sayfasını incelediğinizde diğer geliştiriciler tarafından söylenen bugları ve hataları okuyabilirsiniz. Bu hataları incelemek şimdilik bizim görevimiz değil. Şimdi kodları C ve AVR yönünden inceleyelim.

Arduino çekirdeğinin ilk dosyaları Wiring adıyla adlandırılmıştır. Aslında bu dosyalar Arduino'ya ait olmayıp Wiring'e aittir. Wiring'in kaynak dosyalarına göz atmanız mümkündür. Arada pek fark göreceğinizi sanmıyorum. Bu ilk dosyalardan biri olan Wiring.h dosyasını incelemekle işe başlayalım.

```
/*
  wiring.h - Partial implementation of the Wiring API for the ATmega8.
  Part of Arduino - http://www.arduino.cc/

  Copyright (c) 2005-2006 David A. Mellis

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General
  Public License along with this library; if not, write to the
  Free Software Foundation, Inc., 59 Temple Place, Suite 330,
  Boston, MA  02111-1307  USA

  $Id$
*/

#ifndef Wiring_h
#define Wiring_h

#include <avr/io.h>
#include "binary.h"

#ifdef __cplusplus
extern "C"{
#endif

#define HIGH 0x1
#define LOW  0x0

#define INPUT 0x0
#define OUTPUT 0x1

#define true 0x1
#define false 0x0

#define PI 3.14159265358979323846264338327950
#define HALF_PI 1.57079632679489661923132169163975
#define TWO_PI 6.28318530717958647692528676655900
#define DEG_TO_RAD 0.01745329251994329576923690768488612
#define RAD_TO_DEG 57.29577951308232087679815481410517
```

```
#define SERIAL  0x0
#define DISPLAY 0x1

#define LSBFIRST 0
#define MSBFIRST 1

#define CHANGE 1
#define FALLING 2
#define RISING 3

#define INTERNAL 3
#define DEFAULT 1
#define EXTERNAL 0

// undefine stdlib's abs if encountered
#ifdef abs
#undef abs
#endif

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))
#define abs(x) ((x)>0?(x):-(x))
#define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
#define round(x)     ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
#define radians(deg) ((deg)*DEG_TO_RAD)
#define degrees(rad) ((rad)*RAD_TO_DEG)
#define sq(x) ((x)*(x))

#define interrupts() sei()
#define noInterrupts() cli()

#define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
#define clockCyclesToMicroseconds(a) ( (a) / clockCyclesPerMicrosecond() )
#define microsecondsToClockCycles(a) ( (a) * clockCyclesPerMicrosecond() )

#define lowByte(w) ((uint8_t) ((w) & 0xff))
#define highByte(w) ((uint8_t) ((w) >> 8))

#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
#define bitWrite(value, bit, bitvalue) (bitvalue ? bitSet(value, bit) :
bitClear(value, bit))

typedef unsigned int word;

#define bit(b) (1UL << (b))

typedef uint8_t boolean;
typedef uint8_t byte;

void init(void);

void pinMode(uint8_t, uint8_t);
void digitalWrite(uint8_t, uint8_t);
int digitalRead(uint8_t);
int analogRead(uint8_t);
void analogReference(uint8_t mode);
void analogWrite(uint8_t, int);
```

```
void beginSerial(long);
void serialWrite(unsigned char);
int serialAvailable(void);
int serialRead(void);
void serialFlush(void);

unsigned long millis(void);
unsigned long micros(void);
void delay(unsigned long);
void delayMicroseconds(unsigned int us);
unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout);

void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, byte val);

void attachInterrupt(uint8_t, void (*)(void), int mode);
void detachInterrupt(uint8_t);

void setup(void);
void loop(void);

#ifdef __cplusplus
} // extern "C"
#endif

#endif
```

Bu kod bildiğimiz C/C++ kütüphanesinden başka bir şey değildir. Derleyici için bir şey söz konusu değildir. Eğer AVR kütüphane başlık dosyalarını inceleseydik makine dilinde yapılan bazı tanımları görebilirdik. Örneğin PORTA, DDRB gibi yazmaç adları aslında birer adres değerine tanımlanmış olurdu. Burada bizim karşılaştıracağımız ise AVR-GCC ve Arduino olacaktır. Şimdi kodları satır satır inceleyelim. En baştan itibaren baktığımızda iki adet kütüphane dosyasının eklendiğini görüyoruz.

```
#include <avr/io.h>
#include "binary.h"
```

Burada avr/io.h başlık dosyasını C ile AVR Programlama derslerimizden hatırlamış olmanız gereklidir. Bu başlık dosyası AVR-GCC derleyicisinde AVR mikrodenetleyiciler için bazı temel giriş ve çıkış fonksiyonlarını barındırıyordu. Aslında bu başlık dosyası kendi başına çalışmayıp eklendiğinde çeşitli başlık dosyalarını da beraberinde getirir. Bu tamamen AVR programlama ile alakalı olup temel giriş ve çıkış ve port işlemleri kullanacağımız zaman io.h başlık dosyasını muhakkak çağırmamız gereklidir. Bu başlık dosyasının kaynak kodu aşağıdaki bağlantıda mevcuttur. Fikir edinmek için inceleyebilirsiniz.
https://www.nongnu.org/avr-libc/user-manual/io_8h_source.html

Bu kütüphane dosyasının sizin kullandığınız her mikrodenetleyici için ayrı bir başlık dosyası olduğuna dikkat edin. Hangi mikrodenetleyiciyi kullanmayı seçtiyseniz ona göre ayrı bir dosya çağırıyor. Bu Arduino çekirdeğinde hala Atmega8 kullanıldığı için şimdilik Atmega8 başlık dosyasını sizlerle paylaşalım ve buradan bir ayrıntıyı sizlere açıklayalım.

https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom8.h

Atmega8 başlık dosyasının Atmega328'e göre biraz daha kısa yani basit olduğunu görebiliyoruz. Peki AVR-GCC derleyicisi için yazılmış bu başlık dosyasında neler var ? Şimdi bir kod kesitinden buna bakalım.

```
/* Port D */
#define PIND    _SFR_IO8(0x10)
#define DDRD    _SFR_IO8(0x11)
#define PORTD   _SFR_IO8(0x12)

/* Port C */
#define PINC    _SFR_IO8(0x13)
#define DDRC    _SFR_IO8(0x14)
#define PORTC   _SFR_IO8(0x15)

/* Port B */
#define PINB    _SFR_IO8(0x16)
#define DDRB    _SFR_IO8(0x17)
#define PORTB   _SFR_IO8(0x18)
```

Biz C dilini kullarak AVR mikrodenetleyicileri programladığımızda PINC, PORTA, DDRB diye bazı değişkenleri kullanıyorduk. Bunlar önceden tanımlanmış değişkenlerdi ve biz bunlardan değer okuyup ya da bunlara değer aktarabiliyorduk. Diğer yazmaçlar da bu şekildeydi yani hepsinin birer adı vardı. Bu adları datasheet'den öğrensek de bu adların arkasında ne olduğunu bilmiyorduk. İşte bu adların arkasında ne olduğu burada yazılıdır. PIND için 0x10 adres değeri, DDRD için 0x11 adres değeri,PORTD için 0x12 adres değeri… Bu değerlerin sağlamasını datasheet'den bakarak yapabilirsiniz. Biz PORTD'ye bir değer atadığımızda bu değer doğrudan bellekteki 0x12 adresine aktarılıyordu. PORTD sadece işi kolaylaştırmak için kullanılan bir aracı ad idi. Gerçekte PORTD diye bir şey yok 0x12 diye bir şey vardır. 0x12 değeri yazılımsal bir soyut değer değil donanımsal bir adres değeridir.

Assembly dilinin de buna benzer bir yöntemle ortaya çıktığını söyleyebiliriz. Assembly dili kısaca makine dilinin alfanümerik sembollerle ifade edilmesidir diyebiliriz. Assembly diline ait birkaç özellik de olsa da genel olarak biz makine koduna verilen kısa isimleri kullanarak programlama yapıyorduk. C dilini kullanan bir derleyici öncelikle kodu C dilinden Assembly diline çevirmek zorundadır. Assembly dilinden de makine diline kodun çevrilmesi çok kolay olmaktadır. Çünkü Assembly ile makine dili arasında pek yabancılık yoktur.

Yukarıda bahsettiğimiz mesele AVR-GCC derleyicisini ilgilendirse de burada da kullanıldığı için kısaca bahsedelim dedik. Dikkatimizi çeken bir başka şey ise binary.h adında bir başlık dosyası oluyor. Bu başlık dosyasının ne olduğuna baktığımızda ise sırayla yazılmış değerleri görüyoruz. Bu değerler çok uzun olduğu için küçük bir kısmını buraya koyalım.

```
#define B0 0
#define B00 0
#define B000 0
#define B0000 0
#define B00000 0
#define B000000 0
#define B0000000 0
#define B00000000 0
#define B1 1
#define B01 1
#define B001 1
```

```
#define B0001 1
#define B00001 1
#define B000001 1
#define B0000001 1
#define B00000001 1
#define B10 2
#define B010 2
#define B0010 2
#define B00010 2
#define B000010 2
#define B0000010 2
#define B00000010 2
#define B11 3
#define B011 3
#define B0011 3
#define B00011 3
#define B000011 3
#define B0000011 3
#define B00000011 3
#define B100 4
#define B0100 4
#define B00100 4
#define B000100 4
#define B0000100 4
#define B00000100 4
```

Bu kodun binary (ikilik) sayıları decimal (onluk) sayılara çevirdiğini görebiliriz. 255'e kadar böyle devam etmektedir. Dosya oldukça uzun olsa da basit bir dönüşüm tablosundan ibarettir. İleride bu kodları incelediğimizde böyle bir değer gördüğümüz zaman dönüşümün yapıldığını aklımızda tutmamız gerekir.

```
#define HIGH 0x1
#define LOW  0x0
```

digitalWrite(1, HIGH) ya da digitalWrite(1,LOW) dediğimizde bu HIGH ve LOW'un tam olarak ne olduğunu merak etmiş olabilirsiniz. İşte bu kullanılan HIGH ve LOW burada tanımlanmıştır. HIGH 1'e eşittir ve LOW 0'a eşittir. Aslında derleyici sizin HIGH yazdığınız yere sonra kendisi 1 yazmaktadır. Eğer HIGH yerine 1 yazarsanız yine programınız sıkıntısız çalışacaktır. Buradaki HIGH ve LOW anlaşılırlığı artırmak için koyulan iki tanımdır. Aşağıda INPUT ve OUTPUT, true ve false tanımlarında da bu görülmektedir.

```
#define INPUT 0x0
#define OUTPUT 0x1

#define true 0x1
#define false 0x0
```

Görüldüğü gibi bunların çok da özel bir niteliği yok. Sadece 1 sayısına ve 0 sayısına farklı adlar verilmiştir. Hiç bilmeyen biri perde arkasında çok karmaşık kodların çalıştığını düşünse de aslında böyle değildir. Gördüğünüz gibi programlarken çok derin anlamlar yüklediğimiz INPUT, OUTPUT, HIGH ve LOW değişkenleri aslında 1 ve 0'ın farklı adlarla temsil edilmesinden başka bir şey değildir. Bu bir (1) ve sıfır (0) değerlerinin fonksiyonlarda nasıl kullanıldığına ise ilerleyen bölümlerde değineceğiz.

```
#define PI 3.14159265358979323846264338327950
#define HALF_PI 1.57079632679489661923132169163975
#define TWO_PI 6.28318530717958647692528676655900
#define DEG_TO_RAD 0.01745329251994329576923690768488612
```

```
#define RAD_TO_DEG 57.295779513082320876798154814105
```
Burada matematik işlemleri için kullanılacak sabit değerleri görüyoruz. Bunlardan başlıcası Pi sayısıdır. Ayrıca çevirim işlerinde kullanılacak sabitler de burada tanımlanmıştır. Matematik işleminin olduğu yerde bu değerlerin ne olduğunu bu başlık dosyasına bakarak öğrenmiş oluyoruz.

```
#define LSBFIRST 0
#define MSBFIRST 1
```
Alt bitin önce mi gönderileceği yoksa baş bitin önce mi gönderileceğini kararlaştıran tanımlardır. Bunu SPI fonksiyonlarında kullanıyoruz. Bu değerleri LSBFIRST ya da MSBFIRST olarak yazsak da aslında 0 ve 1 değerlerinden ibarettir.

```
#define CHANGE 1
#define FALLING 2
#define RISING 3
```
Bu tanımlar kesmeler için kullanılmıştır. Kesmelerin ayak değişiminde, düşen kenarda ya da yükselen kenarda mı çalıştıracağı bu değerlere bağlıdır. Bunların da 1, 2 ve 3 değerlerinden ibaret olduğunu görüyoruz. Fonksiyona baktığımızda daha iyi anlayacağız.

```
#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))
#define abs(x) ((x)>0?(x):-(x))
#define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
#define round(x)     ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
#define radians(deg) ((deg)*DEG_TO_RAD)
#define degrees(rad) ((rad)*RAD_TO_DEG)
#define sq(x) ((x)*(x))
```
Pek çok matematik fonksiyonunun burada tanımlandığını görüyoruz. Bu fonksiyonlar aslında bizim bildiğimiz fonksiyon şeklinde çalışmaz. #define ile tanımladığımız bir değer ya da fonksiyona atadığımız değeri derleyici doğrudan kes-yapıştır yapar. Normalde C dilinde fonksiyonun bir adres değeri vardır fakat burada #define ile kullandığımız fonksiyonlar ne kadar fazla olursa kod kalabalığı da o kadar fazla olur. C dilindeki normal bir fonksiyonu çağırdığımızda program o adrese gider burada ise derleyici kodu kopyalayıp yapıştırır. O yüzden #define ile fonksiyon tanımlamanız pek sağlıklı değildir fakat bazı yönlerde size esneklik verecektir.

Burada min(), max(), abs() gibi fonksiyonların nasıl çalıştığını görüyoruz. Bunlar basit matematik işlemleri olduğu için çok fazla anlaşılmayacak bir nokta yok. Mesela yukarıda tanımlanan DEG_TO_RAD ve RAD_TO_DEG değerlerinin burada radyan ve derece çevriminde kullanıldığını görüyoruz. sq() denen kare alma işlemi ise x sayısını kendisiyle çarpmaktan ibaret bir işlem. Bu matematik işlemlerinin yazılımsal olarak yapıldığına dikkat edin. İşlemci makine dilinde bu işlemleri yapabilecek bir karmaşıklığa sahip değildir. İşlemciye toplama, çıkarma ve karşılaştırma gibi çok basit işlemleri yaparak nasıl karmaşık bir işlem yapacağını burada biz öğretiyoruz.

Konumuz C dili dersi olmasa da programcıların gözünden kaçan bir noktayı dile getirme adına min(a,b) fonksiyonunu inceleyelim. Bu fonksiyon (a)<(b)?(a):(b) işlemini yürütür. C dilinde ? operatörü if/else işlemini yapmaktan öte gitmez. Eğer a değeri b'den küçükse a'yı geri döndürür değilse b'yi geri döndürür. Bunu if/else ile yapabileceğimiz gibi kısaltma adına ? operatörü ile de yapabiliriz.

```
#define interrupts() sei()
#define noInterrupts() cli()
```
Bu fonksiyonlar kesmeleri açıyor ya da kapatıyordu. AVR derslerinde göreceğiniz üzere kesmeleri açmak için sei() ve kapamak için de cli() fonksiyonlarını kullanıyorduk. Bunlar da bu fonksiyonları almış ve farklı bir ad vererek yeni bir fonksiyon diye ortaya koymuş. Bu tanım sadece fonksiyonun adını daha anlaşılır yapmaktan öte gitmemektedir. Beğenen veya beğenmeyen muhakkak çıkacaktır. Bu sei() ve cli() fonksiyonları öyle bir şeydir ki AVR Assembly dilinde de aynı şekilde kullanılır. Yani makine kodu olarak da sei ve cli olarak bunu yazarız. Aslında bu komutların yaptığı iş SREG yazmacındaki kesme bayrak bitini birlemek ya da sıfırlamaktan öte bir şey değildir. SEI ve CLI komutlarının mikroişlemci komut kümesinde yer aldığını şu bağlantıdan görebilirsiniz.

https://people.ece.cornell.edu/land/courses/ece4760/AtmelStuff/AVRinstr2002.PDF

Mikroişlemci komut kümesini ileride AVR Assembly hakkında bir çalışmamız olduğunda inceleyeceğiz. Siz yine de fikir edinmek için bir göz gezdirebilirsiniz.

```
#define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
#define clockCyclesToMicroseconds(a) ( (a) / clockCyclesPerMicrosecond() )
#define microsecondsToClockCycles(a) ( (a) * clockCyclesPerMicrosecond() )
```
Bu fonksiyonlar zamanlama işlemlerinde kullanılmak üzere tanımlanmıştır. Normalde Arduino referansında görmemiz mümkün değildir. Demek ki kaynak kodda karşımızda çıkacaktır. Bu fonksiyonlar saat çevrimi ve zaman birimi arasında dönüşümleri yapar. F_CPU değeri işlemcinin kaç MHz'de çalıştığını belirleyen bir değerdi. AVR programlamada F_CPU'yu tanımlasak da aynı zamanda bu değer Makefile'da da tanımlanabilir.

```
#define lowByte(w) ((uint8_t) ((w) & 0xff))
#define highByte(w) ((uint8_t) ((w) >> 8))
```
Biz 8-bitten yukarı bir işlem yaptığımızda her zaman düşük ve yüksek bayt olmak üzere iki ayrı bayt değerinden bahsetmiştik. Örneğin 8 bitlik yazmaca 10 bitlik ADC okuma verisi sığdırılamıyordu. Bu durumda ise yüksek bayt ve düşük bayt olmak üzere iki ayrı yazmaca sırayla bu değer yazdırılıyordu. Okuma yine 8 bit üzerinden olacağı için okumanın bir sırası olması gerekliydi. İşte burada bu yüksek bayt ve düşük payt üzerinde yapılan işlemler görülmektedir. Kısacası yüksek bayt 8 adım sağa kaydırılırken düşük bayt ise değer sağlaması yapılarak olduğu gibi yazılmaktadır.

```
#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
```
Burada bit bazlı işlem yapılırken kullanılan fonksiyonlar tanımlanmıştır. AVR Assembly dilinde bile bit bazlı işlem yapmak için SBI (birleme) ve CBI (sıfırlama) komutları mevcuttur. Biz C dilinde ise biraz karışık bir iş yapıp bit bazlı operatörleri kullanıyorduk. Bu operatörler mantık işlemleri üzerinden aynı işlemi yapıyordu. Bu tanımlar ise bu mantık işlemlerini fonksiyon haline getirmiştir. sıfır yapmak için yine &= ~ operatörünü bir yapmak için yine |= operatörünü kullanmışlar. Anlaşılmayacak bir şey olduğunu sanmadığımdan bir sonraki tanımı sizlere açıklayayım.

```
#define bitWrite(value, bit, bitvalue) (bitvalue ? bitSet(value, bit) :
bitClear(value, bit))
```

Bu tanıma göre bitWrite fonksiyonu değer, bit ve bit değeri olarak üç ayrı argüman alır. Burada bitvalue 1 ya da 0 olup ? operatörü tarafından kullanılır. Eğer bir (1) ise bitSet() fonksiyonu tarafından birlenir (anlayamayanlar için sözlük bağlantısı yukarıda verilmiştir.) eğer sıfır ise de bitClear() fonksiyonu tarafından sıfırlanır.

```
typedef unsigned int word;

#define bit(b) (1UL << (b))

typedef uint8_t boolean;
typedef uint8_t byte;
```

Burada C diline pek aşina olmayanlar için yabancı gelecek typedef kelimesini görmekteyiz. typedef “tip tanımlama” diyeceğimiz bir dil özelliği olup değişken adlarını değiştirip kendi değişken adımızı oluşturmamızı sağlar. Burada aynı uint8_t (8-bit işaretsiz tam sayı) değişken adını boolean ve byte olarak tanımlandığını görüyoruz. Fakat bir dakika, boolean sadece bir (1) ve sıfır (0) değerlerini alabilen bir değişken değil miydi? Burada 0-255 değerini alabilen 8-bitlik bir değişken olarak tanımlandığını görüyoruz. Açıkcası “boolean” diye bizi kandırıyorlar. ☺ Aslında şu şekilde bir boolean yapısı kullanılsaymış amaca yönelik olabilirmiş.

```
typedef enum { false, true } bool;
```

Böyle bir tanım daha fazla “boolean” özelliği taşımaktadır. Sadece adda boolean olan bir tanımdan daha kaliteli olduğu ortadadır. Alternatif olarak C99’da gelen stdbool.h başlık dosyasını kullanmak daha standart olan yoldur.

Ben programlama yaparken doğrudan u_int8 değişkenini kullanma taraftarıyım. Bu değişken 0 olduğunda boolean olarak sıfır, sıfırdan farklı olduğunda ise boolean olarak bir sayılırsa hiç bool ile uğraşmadan aynı iş görülmüş olur.

Burada #define bit(b) (1UL << (b)) adında bir makroyu görmekteyiz. Bu bir yerden bize tanıdık gelmektedir. Sanki AVR-GCC’deki BV_() makrosu ile tıpatıp aynı duruyor. Makroyu incelediğimizde gerçekten de öyle olduğunu görüyoruz. BV_() makrosunun tanımı şu şekildedir.

```
#define _BV(bit)   (1 << (bit))
```

Şimdi geriye ise #define tanımları değil fonksiyon tanımları kalıyor. Bu fonksiyonların ne olduğunu Arduino programcısı olarak az çok bilmeniz gereklidir. Fonksiyonları listeleyelim ve hangi fonksiyonlarının ön tanımının burada yapıldığına bir bakalım.

```
void init(void);

void pinMode(uint8_t, uint8_t);
void digitalWrite(uint8_t, uint8_t);
int digitalRead(uint8_t);
int analogRead(uint8_t);
void analogReference(uint8_t mode);
void analogWrite(uint8_t, int);

void beginSerial(long);
void serialWrite(unsigned char);
int serialAvailable(void);
```

```
int serialRead(void);
void serialFlush(void);

unsigned long millis(void);
unsigned long micros(void);
void delay(unsigned long);
void delayMicroseconds(unsigned int us);
unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout);

void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, byte val);

void attachInterrupt(uint8_t, void (*)(void), int mode);
void detachInterrupt(uint8_t);

void setup(void);
void loop(void);
```

Baktığımızda **Wiring.h** başlık dosyasında Arduino kodlarının temellerinin olduğunu görüyoruz. Hatta setup() ve loop() fonksiyonları bile burada tanımlanmıştır. Bu fonksiyonların ne işe yaradığını bilseniz de argüman olarak hangi değeri aldığını ve hangi değeri değer olarak geri döndürdüğünü buradan görmeniz mümkündür. Wiring.h dosyasının tamamı bu şekildedir. Birkaç ufak yeri bilerek atladım ve konudan çıkmak istemedim.

Arduino bir kütüphane paketi olduğu için pek çok başlık dosyası birbiriyle bağlantılı olarak çalışmaktadır. Kaynak kodun tamamını anlamak için pek çok başlık dosyasını incelemek ve birbiri arasındaki bağlantıyı çözmek gereklidir. Bir dili ne kadar iyi bilirseniz bilin başkasının yazdığı bir kodu anlamanın zorluğu daima mevcut olduğu için hatalarımızın olması kaçınılmazdır. Bu hataları ise bilgili okuyucular yorumlarda “doğrusunu yazmak” kaydıyla belirtebilir. Gerekli düzeltmeleri her zaman yapıyoruz.

-2- Wiring_private.h ve Wiring.c Dosyaları

Arduino'nun açık kaynak olduğu ve geliştiricilerin her zaman kaynak kodunu okuyabileceği ifade edilir. Şu bir gerçek ki Arduino öğrenmekle bu kaynak kodlarını anlamanız mümkün değildir. C dilini ve AVR programlamayı öğrenmeniz gereklidir. Bunları bilen birisinin ise hala Arduino kullanması pek beklenemez. Burada bir çıkmaz meydana gelmektedir. C ve AVR bilen insanların kaynak kodlarını kullanarak değil genellikle C dilini ve AVR komutlarını kullanarak Arduino için kütüphane yazdığını görüyoruz. Yani bu platformda kütüphaneleri anlamak için başka bir platformda çalışmanız gerektiği gibi kütüphane yazmak için de yine başka bir platformu bilmeniz gereklidir. Arduino'ya Arduino kütüphanelerini kullanarak kütüphane yazmak pek sağlıklı bir iş değildir. Zaten performans ve kararlılık konusunda sıkıntılı olan bir platformda bir de bu kodlar üzerine kütüphane yazmak işi iyice çıkmaza götürebilir. Yine de bazı kütüphaneleri Arduino komutlarını kullanarak yazmak mümkündür.

Önceki yazıda Wiring.h dosyasını inceleyerek tanımlara bir göz atmıştık. Şimdi ise .h dosyasının devamı olan .c dosyasını inceleyerek yazımıza devam ediyoruz. Ayrıca bu dosyaların #include ile çağırdığı dosyalara da yer vermemiz gereklidir. Çünkü anlaşılmadık bir yerin kalmamasını ve yüzeysel bir inceleme yapmamayı hedefliyoruz. Şimdi incelemeniz için wiring.c kütüphane dosyasının tamamını verelim ve devamında her zaman olduğu gibi satır satır kodları açıklayalım.

```
/*
  wiring.c - Partial implementation of the Wiring API for the ATmega8.
  Part of Arduino - http://www.arduino.cc/

  Copyright (c) 2005-2006 David A. Mellis

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General
  Public License along with this library; if not, write to the
  Free Software Foundation, Inc., 59 Temple Place, Suite 330,
  Boston, MA  02111-1307  USA

  $Id$
*/

#include "wiring_private.h"

// the prescaler is set so that timer0 ticks every 64 clock cycles, and the
// the overflow handler is called every 256 ticks.
#define MICROSECONDS_PER_TIMER0_OVERFLOW (clockCyclesToMicroseconds(64 * 256))

// the whole number of milliseconds per timer0 overflow
#define MILLIS_INC (MICROSECONDS_PER_TIMER0_OVERFLOW / 1000)
```

```
// the fractional number of milliseconds per timer0 overflow. we shift right
// by three to fit these numbers into a byte. (for the clock speeds we care
// about - 8 and 16 MHz - this doesn't lose precision.)
#define FRACT_INC ((MICROSECONDS_PER_TIMER0_OVERFLOW % 1000) >> 3)
#define FRACT_MAX (1000 >> 3)

volatile unsigned long timer0_overflow_count = 0;
volatile unsigned long timer0_millis = 0;
static unsigned char timer0_fract = 0;

SIGNAL(TIMER0_OVF_vect)
{
	// copy these to local variables so they can be stored in registers
	// (volatile variables must be read from memory on every access)
	unsigned long m = timer0_millis;
	unsigned char f = timer0_fract;

	m += MILLIS_INC;
	f += FRACT_INC;
	if (f >= FRACT_MAX) {
		f -= FRACT_MAX;
		m += 1;
	}

	timer0_fract = f;
	timer0_millis = m;
	timer0_overflow_count++;
}

unsigned long millis()
{
	unsigned long m;
	uint8_t oldSREG = SREG;

	// disable interrupts while we read timer0_millis or we might get an
	// inconsistent value (e.g. in the middle of a write to timer0_millis)
	cli();
	m = timer0_millis;
	SREG = oldSREG;

	return m;
}

unsigned long micros() {
	unsigned long m, t;
	uint8_t oldSREG = SREG;

	cli();
	t = TCNT0;

#ifdef TIFR0
	if ((TIFR0 & _BV(TOV0)) && (t == 0))
		t = 256;
#else
	if ((TIFR & _BV(TOV0)) && (t == 0))
		t = 256;
#endif

	m = timer0_overflow_count;
	SREG = oldSREG;
```

```
	return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
}

void delay(unsigned long ms)
{
	unsigned long start = millis();

	while (millis() - start <= ms)
		;
}

/* Delay for the given number of microseconds.  Assumes a 8 or 16 MHz clock.
 * Disables interrupts, which will disrupt the millis() function if used
 * too frequently. */
void delayMicroseconds(unsigned int us)
{
	uint8_t oldSREG;

	// calling avrlib's delay_us() function with low values (e.g. 1 or
	// 2 microseconds) gives delays longer than desired.
	//delay_us(us);

#if F_CPU >= 16000000L
	// for the 16 MHz clock on most Arduino boards

	// for a one-microsecond delay, simply return.  the overhead
	// of the function call yields a delay of approximately 1 1/8 us.
	if (--us == 0)
		return;

	// the following loop takes a quarter of a microsecond (4 cycles)
	// per iteration, so execute it four times for each microsecond of
	// delay requested.
	us <<= 2;

	// account for the time taken in the preceeding commands.
	us -= 2;
#else
	// for the 8 MHz internal clock on the ATmega168

	// for a one- or two-microsecond delay, simply return.  the overhead of
	// the function calls takes more than two microseconds.  can't just
	// subtract two, since us is unsigned; we'd overflow.
	if (--us == 0)
		return;
	if (--us == 0)
		return;

	// the following loop takes half of a microsecond (4 cycles)
	// per iteration, so execute it twice for each microsecond of
	// delay requested.
	us <<= 1;

	// partially compensate for the time taken by the preceeding commands.
	// we can't subtract any more than this or we'd overflow w/ small delays.
	us--;
#endif

	// disable interrupts, otherwise the timer 0 overflow interrupt that
```

```
        // tracks milliseconds will make us delay longer than we want.
        oldSREG = SREG;
        cli();

        // busy wait
        __asm__ __volatile__ (
                "1: sbiw %0,1" "\n\t" // 2 cycles
                "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
        );

        // reenable interrupts.
        SREG = oldSREG;
}

void init()
{
        // this needs to be called before setup() or some functions won't
        // work there
        sei();

        // on the ATmega168, timer 0 is also used for fast hardware pwm
        // (using phase-correct PWM would mean that timer 0 overflowed half as
often
        // resulting in different millis() behavior on the ATmega8 and ATmega168)
#if !defined(__AVR_ATmega8__)
        sbi(TCCR0A, WGM01);
        sbi(TCCR0A, WGM00);
#endif
        // set timer 0 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR0, CS01);
        sbi(TCCR0, CS00);
#else
        sbi(TCCR0B, CS01);
        sbi(TCCR0B, CS00);
#endif
        // enable timer 0 overflow interrupt
#if defined(__AVR_ATmega8__)
        sbi(TIMSK, TOIE0);
#else
        sbi(TIMSK0, TOIE0);
#endif

        // timers 1 and 2 are used for phase-correct hardware pwm
        // this is better for motors as it ensures an even waveform
        // note, however, that fast pwm mode can achieve a frequency of up
        // 8 MHz (with a 16 MHz clock) at 50% duty cycle

        // set timer 1 prescale factor to 64
        sbi(TCCR1B, CS11);
        sbi(TCCR1B, CS10);
        // put timer 1 in 8-bit phase correct pwm mode
        sbi(TCCR1A, WGM10);

        // set timer 2 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, CS22);
#else
        sbi(TCCR2B, CS22);
#endif
```

```
        // configure timer 2 for phase correct pwm (8-bit)
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, WGM20);
#else
        sbi(TCCR2A, WGM20);
#endif

#if defined(__AVR_ATmega1280__)
        // set timer 3, 4, 5 prescale factor to 64
        sbi(TCCR3B, CS31);      sbi(TCCR3B, CS30);
        sbi(TCCR4B, CS41);      sbi(TCCR4B, CS40);
        sbi(TCCR5B, CS51);      sbi(TCCR5B, CS50);
        // put timer 3, 4, 5 in 8-bit phase correct pwm mode
        sbi(TCCR3A, WGM30);
        sbi(TCCR4A, WGM40);
        sbi(TCCR5A, WGM50);
#endif

        // set a2d prescale factor to 128
        // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
        // XXX: this will not work properly for other clock speeds, and
        // this code should use F_CPU to determine the prescale factor.
        sbi(ADCSRA, ADPS2);
        sbi(ADCSRA, ADPS1);
        sbi(ADCSRA, ADPS0);

        // enable a2d conversions
        sbi(ADCSRA, ADEN);

        // the bootloader connects pins 0 and 1 to the USART; disconnect them
        // here so they can be used as normal digital i/o; they will be
        // reconnected in Serial.begin()
#if defined(__AVR_ATmega8__)
        UCSRB = 0;
#else
        UCSR0B = 0;
#endif
}
```

Burada ilk koda baktığımızda wiring_private.h adında bir dosyanın eklendiğini görüyoruz. Bu dosyayı gözden kaçırmamak için öncelikle incelememiz gereklidir. Sonrasında ise wiring.c dosyasına geri dönüp incelemeye devam edeceğiz. wiring_private.h dosyasının kaynak kodu şu şekildedir.

```
/*
  wiring_private.h - Internal header file.
  Part of Arduino - http://www.arduino.cc/

  Copyright (c) 2005-2006 David A. Mellis

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.
```

```
  You should have received a copy of the GNU Lesser General
  Public License along with this library; if not, write to the
  Free Software Foundation, Inc., 59 Temple Place, Suite 330,
  Boston, MA  02111-1307  USA

  $Id: wiring.h 239 2007-01-12 17:58:39Z mellis $
*/

#ifndef WiringPrivate_h
#define WiringPrivate_h

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/delay.h>
#include <stdio.h>
#include <stdarg.h>

#include "wiring.h"

#ifdef __cplusplus
extern "C"{
#endif

#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

#define EXTERNAL_INT_0 0
#define EXTERNAL_INT_1 1
#define EXTERNAL_INT_2 2
#define EXTERNAL_INT_3 3
#define EXTERNAL_INT_4 4
#define EXTERNAL_INT_5 5
#define EXTERNAL_INT_6 6
#define EXTERNAL_INT_7 7

#if defined(__AVR_ATmega1280__)
#define EXTERNAL_NUM_INTERRUPTS 8
#else
#define EXTERNAL_NUM_INTERRUPTS 2
#endif

typedef void (*voidFuncPtr)(void);

#ifdef __cplusplus
} // extern "C"
#endif

#endif
```

Burada bazı kütüphane dosyalarının daha çağrıldığını görüyoruz. Bu dosyalar şu şekildedir.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/delay.h>
#include <stdio.h>
```

```
#include <stdarg.h>

#include "wiring.h"
```
Bu kaynak kodları incelemenin asıl zorluğu zaten bu dosyalar arasında bağlantıyı kurabilmekti. Şimdi wiring.c dosyasının çağırdığı wiring_private.h dosyasında yine bir sürü kütüphane dosyasının çağrıldığını görüyoruz. Burada bize tanıdık gelen bazı dosyalar mevcuttur. Öncelikle wiring.h dosyasını incelememizin faydasını burada görüyoruz. Burada wiring.h dosyası kullanıldığı için bu dosya hakkında bilgiyi önceki yazıdan edinebilirsiniz. Ayrıca avr/io.h, avr/interrupt.h ve avr/delay.h dosyalarını C ile AVR programlama derslerimizde anlatmıştık. O yüzden bunları anlama konusunda bir sıkıntımız yok. avr/interrupt.h dosyasını kesmeler için kullanırken avr/delay.h dosyasını ise bekleme komutları için kullanıyorduk.

Burada ilk defa karşımıza çıkan iki adet kütüphane yer alıyor. Bunlardan biri stdio.h dosyası diğeri ise stdarg.h dosyasıdır. stdio.h dosyası C dilini bilenler için yabancı gelmeyecektir. Standart giriş ve çıkış fonksiyonlarını kullanmak için bu dosyayı kullanıyorduk. stdarg.h dosyası ise pek tanıdık olacağınız bir dosya olmayabilir.

stdarg dosyası özet olarak C programlama dilinin standart kütüphanesinde yer alan bir kütüphane dosyası olup fonksiyonların sınırsız sayıda argüman alabilmesini sağlatır. Variadic fonksiyon adı verilen fonksiyonları şu şekilde tanımlamamız mümkündür.

```
int check(int a, double b, ...);
```
Bu C dilini ilgilendiren bir konu olduğu için bu kadarını anlatmakla yetinelim.

Bu kütüphane dosyasının çağırdığı başlık dosyalarını gördük. Şimdi ise tanımlamalara geçiyoruz.

```
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif
```
Burada sbi() ve cbi() makroları tanımlanmadıysa tanımlamayı sağlayan bir yapı var. AVR Assembly dilindeki sbi ve cbi komutları ile AVR-GCC'deki komutları birbirine karıştırmamak gerektiği buradan anlaşılıyor. Çünkü C dilinde sbi)() ve cbi() fonksiyonları çok farklı bir şekilde çalışıyor. Öncelikle örneğin PORTD adında bir değişken kullanıyorsak bu değişkenin adres değeri _SFR_BYTE() fonksiyonu ile bulunuyor ve sonrasında _BV makrosu ve bitwise operatörleri ile bu bit bir (1) veya sıfır (0) yapılıyor. Bu AVR konusu olduğu için AVR derslerinde bunun hakkında daha ayrıntılı bilgiye ulaşmanız mümkündür. Önceki derste yine aynı konuya değindiğimiz için tekrarlamaktan kaçınıyoruz.

```
#define EXTERNAL_INT_0 0
#define EXTERNAL_INT_1 1
#define EXTERNAL_INT_2 2
#define EXTERNAL_INT_3 3
#define EXTERNAL_INT_4 4
#define EXTERNAL_INT_5 5
#define EXTERNAL_INT_6 6
#define EXTERNAL_INT_7 7
```

```
#if defined(__AVR_ATmega1280__)
#define EXTERNAL_NUM_INTERRUPTS 8
#else
#define EXTERNAL_NUM_INTERRUPTS 2
#endif
```

Burada harici kesmelerle ilgili tanımlamalar yapılmış olup ayrı bir karar yapısında ek tanımlamalar yapılmıştır. Arduino donanım için yazılan bir kütüphane olduğu için taşınabilirliği oldukça düşüktür. O yüzden her donanıma uyumlu hale getirmek için kodların arasına böyle ek kodlar sokuşturulur. Bu ilerlediği zaman okunabilirliği olumsuz etkileyen bir durum haline gelmektedir. Biz de bundan kaçınmak için eski sürümü incelemeyi tercih ettik. Yeni sürümü bir göz gezdirdiğimde bu #if defined yapılarının iyice arttığını ve iyice kod yığını haline geldiğini gördüm. O yüzden kod incelememizde bu #if defined yapılarını atlamamız gerekebilir.

```
typedef void (*voidFuncPtr)(void);
```

Bu komut voidFuncPtr işaretçisini void'e atamaya yarar. Burada fonksiyon argüman almıyor ve void'i geri döndürmektedir.

Buraya kadar wiring_private.h dosyasını incelemeyi tamamladık. Şimdi wiring.c dosyasını incelemeye devam edebiliriz. Fakat bu dosya aşırı derecede uzun olduğu için açıklamaları bir sonraki yazıya almak durumundayım. Bir sonraki yazıda wiring.c dosyasını açıklamaya devam edeceğiz.

-3- Wiring.c Dosyası ve Temel Arduino Komutları

Arduino kaynak kodlarını incelemeye 3. yazımızda devam ediyoruz. Bu inceleme yazı dizisinin bu alanda yapılan ilk çalışma olduğunu da söylememiz gerekir. İngilizce olarak şöyle bir kaynak bulsam da burada kısmen ve yüzeysel olarak incelendiğinden dolayı pek kullanma ihtiyacı hissetmedim. Yine de bu alanda yazılmış başka bir kaynak olarak bunu zikretmemiz gerekir.

http://garretlab.web.fc2.com/en/arduino/inside/index.html

wiring.c dosyasını incelediğimizde iş biraz daha ciddileşiyor. Burada AVR kodlarını ve yazmaçları görmemiz mümkün. Öncesinde ise zamanlayıcılar için çeşitli #define tanımlarını görüyoruz. Bu #define tanımlarını açıklayarak yazımıza devam edelim.

```
#define MICROSECONDS_PER_TIMER0_OVERFLOW (clockCyclesToMicroseconds(64 * 256))
```

Burada önceki dosyada gördüğümüz clockCyclesToMicroseconds() fonksiyonu kullanılmış. Bunu anlattığımız için burada yapılan işlemi anlatalım. Burada TC0 zamanlayıcısının ön derecelendiricisi (prescaler) 64 olduğu için zamanlayıcının sayma ünitesi her 64 saat çevriminde birer birer artmaktadır. 256'da taşma gerçekleşeceği için bunu çevirim bazında hesaplamak için 64 ile 256 birbiriyle çarpılır. Böylelikle taşmanın gerçekleşeceği çevirim sayısı bulunmuş olur. Sırada fonksiyon ile bunu mikro saniyeye çevirmek vardır.

MICROSECONDS_PER_TIMER0_OVERFLOW yazan yerde anlayacağımız TC0 zamanlayıcısının taşması için gerekli mikro saniye değeridir.

```
#define MILLIS_INC (MICROSECONDS_PER_TIMER0_OVERFLOW / 1000)
```

Bu tanım ise yukarıdaki makroyu kullanır fakat bunu 1000'e bölerek yeni bir değer elde eder. Yukarıdaki mikro saniye değerini MILLIS_INC ile milisaniye olarak alıyoruz. Kodda MILLIS_INC gördüğümüz yerde TC0 zamanlayıcısının kaç milisaniyede taştığı verisi aklımıza gelmelidir.

```
#define FRACT_INC ((MICROSECONDS_PER_TIMER0_OVERFLOW % 1000) >> 3)
#define FRACT_MAX (1000 >> 3)
```

Burada ise yukarıdaki değerler kesirli olarak elde edilir. 1000 üzerinden mod alma işlemi uygulanmış ve sonrasında 3 bit sağa kaydırılmıştır. Buradaki tanımlamaların ardından zamanlayıcılar ile alakalı daha fazla koda rastlıyoruz.

```
volatile unsigned long timer0_overflow_count = 0;
volatile unsigned long timer0_millis = 0;
static unsigned char timer0_fract = 0;
```

Burada zamanlayıcılarla alakalı üç adet değişken tanımlanmıştır. Bu değişkenler tanımlanırken başta hepsi sıfır değerini almıştır.

```
SIGNAL(TIMER0_OVF_vect)
{
	// copy these to local variables so they can be stored in registers
	// (volatile variables must be read from memory on every access)
```

```
        unsigned long m = timer0_millis;
        unsigned char f = timer0_fract;

        m += MILLIS_INC;
        f += FRACT_INC;
        if (f >= FRACT_MAX) {
                f -= FRACT_MAX;
                m += 1;
        }

        timer0_fract = f;
        timer0_millis = m;
        timer0_overflow_count++;
}
```

Burada SIGNAL adında farklı bir fonksiyon görmekteyiz. Bu sizi şaşırtmasın. Bu fonksiyon avr/interrupt.h kütüphane dosyasının içerisinde mevcuttur ve ISR fonksiyonu ile aynı görevi görür. Yalnız eski sürümde olduğu için şimdi kullanmamız tavsiye edilmez. Burada SIGNAL (TIMER0_OVF_vect) aslında ISR(TIMER0_OVF_vect) anlamına gelir. Farklı bir özelliği yoktur. Burada kesme vektörlerinin anlamını yine C ile AVR Programlama dersimizde tablo şeklinde vermiştik. Hatırlamak amacıyla tekrar söyleyelim. TIMER0_OVF_vect, TC0 zamanlayıcısı taştığı zaman yürütülen kesmedir.

Burada unsigned long m, f olarak iki adet yerel değişken tanımlanmış ve volatile değişkenlerdeki değer bunlara aktarılmıştır. Bunun sebebi ise yerel değişkenler yazmaçlarda tutulabilirken volatile değişkenler her defasında hafıza ünitesinden okunması gerekir.

m değerine MILLIC_INC değeri ilave edilir. Aynı şekilde f değerine de FRACT_INC değeri ilave edilir. Bu değerlerin ne olduğunu ise yukarıda açıklamıştık. FRACT_INC için bir azami değer tanımlanmış olup kesirli kısımla eğer bir bütün elde edilebiliyorsa bir milisaniye olarak f değişkenine eklenir ve f değerinden bu azami değer çıkarılır.

Bu fonksiyon özet olarak TC0 zamanlayıcısı taştığı zaman MILLIS_INC ve FRACT_INC makrolarından elde edilen değerleri alıp düzenleyerek kaydeder. Sonrasında ise önceden program tarafından tanımlanmış değerler üzerinden yapılan hesaplama değerlerini volatile olan timer0_millis ve timer0_fract değerlerine kaydeder. Böylelikle sonrasında yine kalan değerler üzerinden hesaplama yapabilir. Fonksiyonun en sonunda ise timer0_overflow_count bir artırılmaktadır. Bu ++ operatörüyle olup her taşma gerçekleştiğinde bu değişken birer birer artar. Böylelikle toplamda kaç adet taşma gerçekleştiği görülebilir.

Bu fonksiyonun işleyişi bu kadardır. Bu fonksiyonda toplamda kaç milisaniye geçtiği kaydedilir, küsüratlı değer bir sonraki hesaplama için saklanır ve toplamda kaç taşma gerçekleştiği kaydedilir.

```
unsigned long millis()
{
        unsigned long m;
        uint8_t oldSREG = SREG;

        // disable interrupts while we read timer0_millis or we might get an
        // inconsistent value (e.g. in the middle of a write to timer0_millis)
```

```
        cli();
        m = timer0_millis;
        SREG = oldSREG;

        return m;
}
```

Bu fonksiyon hepimize tanıdık gelecektir. Meşhur millis() fonksiyonu işte bundan ibarettir. Fakat yukarıda verdiğimiz fonksiyon olmadan millis() fonksiyonunun bir işe yaramayacağını söyleyelim. timer0_millis değişkenine bütün değer atamasını TC0 taşma kesme fonksiyonu yapıyordu. Bu fonksiyon ise unsigned long cinsinde timer0_millis değişkeninin değerini geri döndürüyor. oldSREG = SREG şeklinde bir komut kullanıldığına dikkat ediniz. SREG yani mikrodenetleyicinin durum yazmacının değerini oluşturduğumuz bir bayt değişkenine atıyoruz. Böylelikle lazım olduğunda sonra kullanma imkanımız oluyor. Sonrasında ise cli() fonksiyonuyla kesmeleri kapatıyoruz. Hesaplama anında bir kesme yürürse hesaplama bozulacağı için bunu yapıyoruz. Sonrasında ise yukarıda unsigned long olarak tanımladığımız m değişkenine volatile olan timer0_millis değişkenini atıyoruz. Sonrasında ise SREG yazmacına eski değerini yükleyerek kesmeleri tekrar etkin hale getiriyoruz. Fonksiyon en sonra ise m değişkenini geri döndürecektir. İşte millis() fonksiyonu bundan ibarettir. Burada timer0_millis değeri çalıştıkça artmaya devam edecektir ve belli bir süre sonra sıfırlanacaktır. Bunu elle sıfırlamanın da sistemde kararsızlığa neden olacağını söyleyenler var. En iyisi hiç dokunmamak.

```
unsigned long micros() {
        unsigned long m, t;
        uint8_t oldSREG = SREG;

        cli();
        t = TCNT0;

#ifdef TIFR0
        if ((TIFR0 & _BV(TOV0)) && (t == 0))
                t = 256;
#else
        if ((TIFR & _BV(TOV0)) && (t == 0))
                t = 256;
#endif

        m = timer0_overflow_count;
        SREG = oldSREG;

        return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
}
```

Burada ise micros() fonksiyonunun iç yapısını görmekteyiz. micros() fonksiyonu millis() fonksiyonuna benzer bir özelliğe sahiptir. millis()'den farkı program başlangıcından itibaren kaç mili saniye geçtiğini değil kaç mikro saniye geçtiğini bize söylemesidir. Burada millis() fonksiyonuna benzer olarak m ve t adında değişkenlerin tanımlandığını ve kesmelerin kapatıldığını görüyoruz. Yine SREG yazmacının değerini bir değişkene kaydettiklerini görebiliriz. Burada yine #ifdef diye başlayan bir kontrol yapısı görmekteyiz. Arduino'nun taşınabilirliğinden söz etmiştik. Burada farklı bir donanım kullanıldığında yapılacak işlemden bahsediliyor. Aynı yazmaç bir mikrodenetleyicide TIFR adındayken ötekinde TIFR0 adında olabiliyor. Bu yazmaçların farklı adda olması taşınabilirliği olumsuz etkilese de yapabileceğimiz pek bir şey yoktur. Kodumuzu ona uygun yazmak durumundayız.

Atmel Studio'da AVR programlarken her aygıtın farklı bir başlık dosyasında farklı tanımlamalarının olduğunu görebiliriz. Yazmaç adları ve bazı değerler mikrodenetleyiciye göre değişmektedir. O yüzden gömülü sistemlerde Assembly ve C dilleri etkin olarak kullanılırken bir Framework veya üst seviye bir dil kullanılırken zorluk yaşanmaktadır. İster istemez sürekli farklı donanımlarda çalıştığımız için donanım bilgisine sahip olmamız gerekir ve bu donanım bilgisi gerektiren dilleri kullanmamız gerekir.

m değişkenine timer0_overflow_count değişkenindeki değer aktarılmaktadır. Böylelikle TC0 zamanlayıcısının kaç kere taştığı aşağıdaki komutta kullanılabilir.

Burada m değeri 8 bitlik zamanlayıcının taşma sayısını ifade ettiğinden ikilik olarak sekiz basamak üstte yazılmalıdır. Bu ise (m << 8) + t komutuyla sağlanır. t değerinin üstteki kodda 256 olduğuna dikkat edelim. Bu değer ise clockCyclesPerMicrosecond() fonksiyonun 64'e bölümüyle çarpılır ve mikrosaniye değeri elde edilmiş olur. return ile de bu değeri geri döndürüyoruz.

```
void delay(unsigned long ms)
{
        unsigned long start = millis();

        while (millis() - start <= ms)
                ;
}
```

Burada ise bizim belki binlerce defa kullandığımız delay() fonksiyonunun iç yapısını görüyoruz. delay() fonksiyonu bizim belirlediğimiz mili saniye değerini argüman olarak alıp o değer kadar mikrodenetleyiciyi beklemeye sokuyordu. Bu bekleme donanımsal bir bekleme değil yazılım olarak meşgul ederek suni bir bekleme ortaya koymak idi. Burada da millis() fonksiyonu kullanarak bir başlangıç değeri elde ediliyor. Bu başlangıç değeri millis() fonksiyonu taşmaya yakın alınırsa programı sonsuz döngüye sokup çalışmaz hale getireceği bir gerçektir. Belki on binde bir bir ihtimal olsa da böyle bir zayıflık burada mevcuttur. Fonksiyon argüman olarak aldığı değeri millis() fonksiyonunun güncel değeri ile karşılaştırır ve ms kadar zaman geçtikten sonra program sonsuz döngüden çıkar. İşlemci bu süreç içerisinde boş yere çalıştırılmış olur. Bu basit matematik ve mantık işlemini anladığınızı varsayıyoruz.

```
void delayMicroseconds(unsigned int us)
{
        uint8_t oldSREG;

        // calling avrlib's delay_us() function with low values (e.g. 1 or
        // 2 microseconds) gives delays longer than desired.
        //delay_us(us);

#if F_CPU >= 16000000L
        // for the 16 MHz clock on most Arduino boards

        // for a one-microsecond delay, simply return.  the overhead
        // of the function call yields a delay of approximately 1 1/8 us.
        if (--us == 0)
                return;

        // the following loop takes a quarter of a microsecond (4 cycles)
        // per iteration, so execute it four times for each microsecond of
```

```
        // delay requested.
        us <<= 2;

        // account for the time taken in the preceeding commands.
        us -= 2;
#else
        // for the 8 MHz internal clock on the ATmega168

        // for a one- or two-microsecond delay, simply return.  the overhead of
        // the function calls takes more than two microseconds.  can't just
        // subtract two, since us is unsigned; we'd overflow.
        if (--us == 0)
                return;
        if (--us == 0)
                return;

        // the following loop takes half of a microsecond (4 cycles)
        // per iteration, so execute it twice for each microsecond of
        // delay requested.
        us <<= 1;

        // partially compensate for the time taken by the preceeding commands.
        // we can't subtract any more than this or we'd overflow w/ small delays.
        us--;
#endif

        // disable interrupts, otherwise the timer 0 overflow interrupt that
        // tracks milliseconds will make us delay longer than we want.
        oldSREG = SREG;
        cli();

        // busy wait
        __asm__ __volatile__ (
                "1: sbiw %0,1" "\n\t" // 2 cycles
                "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
        );

        // reenable interrupts.
        SREG = oldSREG;
}
```

Burada ise delaymicroseconds() fonksiyonunu görmekteyiz. Bu fonksiyonu satır satır açıklama zahmetine girmeyip yine anladığımızı yazarak geçeceğiz. Burada yine oldSREG adında bir değişken tanımlanmıştır. Çünkü yine kesmeler kapatılacak ve sonrasında SREG yazmacının eski değeri yazmaca yüklenecektir. Bekleme fonksiyonlarında kesmeleri kapatmazsak bekleme esnasında mikrodenetleyici kesmeye gidecek ve programımız kararsız çalışabilecektir. Programın kararlılığına etkisi olan hassas işlemlerde kesmeleri kapatmamız gereklidir.

#if F_CPU >= 16000000L ile yine bir karar yapısı görmekteyiz. Burada farklı frekansta çalışan Arduino kartları için taşınabilirlik adına yapılan bir uyumluluk yapısı söz konusudur. Böylelikle her sistem için ayrı kütüphane yazmak yerine aynı kütüphaneyi farklı sistemlerde kullanabiliriz. C diline dair pek çok kitapta C dilinin taşınabilir olduğunu yazsalar da bu Gömülü C için pek geçerli değildir. Bunu da yine buradan görmekteyiz. Burada argüman olarak alınan us değerine dair çeşitli işlemler yapılmaktadır. Bu aşağıdaki Assembly koduna uyumlu hale getirmektedir.

```
        __asm__ __volatile__ (
                "1: sbiw %0,1" "\n\t" // 2 cycles
                "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
        );
```

Bu meşhur delay fonksiyonu AVR mikrodenetleyiciler için sıklıkla kullanılır. Assembly dilinde yazıldığı için ayrıntısını şu an açıklamamıza gerek yoktur. AVR için örnek kodları arattığınızda da bu fonksiyonu pek çok yerde görebilirsiniz.

```
void init()
{
        // this needs to be called before setup() or some functions won't
        // work there
        sei();

        // on the ATmega168, timer 0 is also used for fast hardware pwm
        // (using phase-correct PWM would mean that timer 0 overflowed half as
often
        // resulting in different millis() behavior on the ATmega8 and ATmega168)
#if !defined(__AVR_ATmega8__)
        sbi(TCCR0A, WGM01);
        sbi(TCCR0A, WGM00);
#endif
        // set timer 0 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR0, CS01);
        sbi(TCCR0, CS00);
#else
        sbi(TCCR0B, CS01);
        sbi(TCCR0B, CS00);
#endif
        // enable timer 0 overflow interrupt
#if defined(__AVR_ATmega8__)
        sbi(TIMSK, TOIE0);
#else
        sbi(TIMSK0, TOIE0);
#endif

        // timers 1 and 2 are used for phase-correct hardware pwm
        // this is better for motors as it ensures an even waveform
        // note, however, that fast pwm mode can achieve a frequency of up
        // 8 MHz (with a 16 MHz clock) at 50% duty cycle

        // set timer 1 prescale factor to 64
        sbi(TCCR1B, CS11);
        sbi(TCCR1B, CS10);
        // put timer 1 in 8-bit phase correct pwm mode
        sbi(TCCR1A, WGM10);

        // set timer 2 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, CS22);
#else
        sbi(TCCR2B, CS22);
#endif
        // configure timer 2 for phase correct pwm (8-bit)
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, WGM20);
#else
        sbi(TCCR2A, WGM20);
```

```
#endif

#if defined(__AVR_ATmega1280__)
        // set timer 3, 4, 5 prescale factor to 64
        sbi(TCCR3B, CS31);      sbi(TCCR3B, CS30);
        sbi(TCCR4B, CS41);      sbi(TCCR4B, CS40);
        sbi(TCCR5B, CS51);      sbi(TCCR5B, CS50);
        // put timer 3, 4, 5 in 8-bit phase correct pwm mode
        sbi(TCCR3A, WGM30);
        sbi(TCCR4A, WGM40);
        sbi(TCCR5A, WGM50);
#endif

        // set a2d prescale factor to 128
        // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
        // XXX: this will not work properly for other clock speeds, and
        // this code should use F_CPU to determine the prescale factor.
        sbi(ADCSRA, ADPS2);
        sbi(ADCSRA, ADPS1);
        sbi(ADCSRA, ADPS0);

        // enable a2d conversions
        sbi(ADCSRA, ADEN);

        // the bootloader connects pins 0 and 1 to the USART; disconnect them
        // here so they can be used as normal digital i/o; they will be
        // reconnected in Serial.begin()
#if defined(__AVR_ATmega8__)
        UCSRB = 0;
#else
        UCSR0B = 0;
#endif
}
```

Burada init() adında Arduino'da görmediğimiz bir fonksiyon görüyoruz. Bu fonksiyon içerisinde zamanlayıcılara ait kodları bulunduruyor. Bu fonksiyonun nerede ve ne zaman çalıştığı hakkında bir bilgimiz yok. Fakat bunun mikrodenetleyici ilk başladığında yürütülecek tanımlama ve hazırlama kodlarından oluştuğunu görmekteyiz. Bu fonksiyonun nerede kullanılacağına dair bir bilgiyi elde etmek için diğer dosyaları araştırdığımızda bu fonksiyonu main.cxx dosyasında görmekteyiz. Şimdi bu dosyanın içeriğine bakalım.

```
int main(void)
{
        init();

        setup();

        for (;;)
                loop();

        return 0;
}
```

Bu aslında bizim gizlenmiş ana programımızdan ibarettir. Biz Arduino'da kod yazarken setup() ve loop() fonksiyonlarından ibaret bir yapıyı görmekteydik. Aslında gerçek yapı bu olup önce init() fonksiyonu çalıştırılıp ardından setup() fonksiyonu çalıştırılmaktadır. En sonunda ise loop() fonksiyonu sonsuz döngü içerisinde çalışmaya devam etmektedir. loop() içerisine program kodunu ve setup() içerisine bir kere çalışmaya mahsus tanımlama ve başlatma kodlarını yazıyorduk. Burada gizli bir init() fonksiyonu olduğunu ise şimdi

görüyoruz. Arduino geliştiricilerinin bize bundan haber vermemesi oldukça normaldir çünkü bu fonksiyon Arduino programcısını alakadar etmez.

Şimdi bu gizli init() fonksiyonunda hangi kodların çalıştırıldığına bakalım.

```
void init()
{
        // this needs to be called before setup() or some functions won't
        // work there
        sei();

        // on the ATmega168, timer 0 is also used for fast hardware pwm
        // (using phase-correct PWM would mean that timer 0 overflowed half as
often
        // resulting in different millis() behavior on the ATmega8 and ATmega168)
#if !defined(__AVR_ATmega8__)
        sbi(TCCR0A, WGM01);
        sbi(TCCR0A, WGM00);
#endif
        // set timer 0 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR0, CS01);
        sbi(TCCR0, CS00);
#else
        sbi(TCCR0B, CS01);
        sbi(TCCR0B, CS00);
#endif
        // enable timer 0 overflow interrupt
#if defined(__AVR_ATmega8__)
        sbi(TIMSK, TOIE0);
#else
        sbi(TIMSK0, TOIE0);
#endif

        // timers 1 and 2 are used for phase-correct hardware pwm
        // this is better for motors as it ensures an even waveform
        // note, however, that fast pwm mode can achieve a frequency of up
        // 8 MHz (with a 16 MHz clock) at 50% duty cycle

        // set timer 1 prescale factor to 64
        sbi(TCCR1B, CS11);
        sbi(TCCR1B, CS10);
        // put timer 1 in 8-bit phase correct pwm mode
        sbi(TCCR1A, WGM10);

        // set timer 2 prescale factor to 64
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, CS22);
#else
        sbi(TCCR2B, CS22);
#endif
        // configure timer 2 for phase correct pwm (8-bit)
#if defined(__AVR_ATmega8__)
        sbi(TCCR2, WGM20);
#else
        sbi(TCCR2A, WGM20);
#endif
```

```
#if defined(__AVR_ATmega1280__)
        // set timer 3, 4, 5 prescale factor to 64
        sbi(TCCR3B, CS31);      sbi(TCCR3B, CS30);
        sbi(TCCR4B, CS41);      sbi(TCCR4B, CS40);
        sbi(TCCR5B, CS51);      sbi(TCCR5B, CS50);
        // put timer 3, 4, 5 in 8-bit phase correct pwm mode
        sbi(TCCR3A, WGM30);
        sbi(TCCR4A, WGM40);
        sbi(TCCR5A, WGM50);
#endif

        // set a2d prescale factor to 128
        // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
        // XXX: this will not work properly for other clock speeds, and
        // this code should use F_CPU to determine the prescale factor.
        sbi(ADCSRA, ADPS2);
        sbi(ADCSRA, ADPS1);
        sbi(ADCSRA, ADPS0);

        // enable a2d conversions
        sbi(ADCSRA, ADEN);

        // the bootloader connects pins 0 and 1 to the USART; disconnect them
        // here so they can be used as normal digital i/o; they will be
        // reconnected in Serial.begin()
#if defined(__AVR_ATmega8__)
        UCSRB = 0;
#else
        UCSR0B = 0;
#endif
}
```

Yine bir kontrol yapısıyla karşı karşıyayız. Bu kontrol yapısı derleyiciyi ilgilendiren bir yapı olup hangi kodun hangi şartta programa dahil edileceğini belirler. Örneğin Atmega8 kullanılmıyorsa bu komutlar programa eklenmez ve devamında belirlenen komutlar eklenir. Burada ise TCCR yazmaçlarının ad farklılığından dolayı ortaya çıkan bir farklılık söz konusudur.

Zamanlayıcı yazmaçları ve diğer bazı yazmaçların adı yukarıda belirttiğimiz gibi her AVR mikrodenetleyicisinde aynı değildir. Bunu mikrodenetleyicinin başlık dosyasından ve teknik veri kitapçığından okuyup öğrenmeniz gereklidir. C ile AVR programlama derslerini de ATmega328p mikrodenetleyicisine göre anlattık. ATmega32 kullanacağınız zaman zamanlayıcı yazmaçlarının adları değişmektedir ve bu adları programcının öğrenmesi gereklidir. Tabi ki her mikrodenetleyicide tamamen değişen adlar söz konusu değildir. Fakat belli başlı serilerde belli başlı adlar değişmektedir.

Buradaki kodları incelediğimizde ise ön derecelendirici ve zamanlayıcı ayarlarının yapıldığını görmekteyiz. Arduino'da zamanlayıcıları tam anlamıyla kullanamadığımızı biliyoruz. Bunun bir sebebi de Arduino fonksiyonlarının zamanlayıcıları kendine göre kullanmasıdır. Bu koddan bunu çok rahat anlamamız mümkündür. Çünkü tüm ayarlar bize sorulmadan önceden yapılmaktadır.

Burada TC0 zamanlayıcısının millis() gibi zamanlama fonksiyonlarını çalıştırmak için kullanıldığını ve buna göre ayarlandığını görebiliriz. TC1 ve TC2 zamanlayıcıları ise PWM sinyali üretmek için kullanılır ve kodda ona göre ayarlanmıştır.

Burada yazmaç ve bit adlarını anlamak için C ile AVR Programlama derslerimize bakabilirsiniz. Mikrodenetleyici değişse de birbirine benzemektedir ve buradan çıkarım yapmanız mümkündür. Burada tek tek anlatarak konuyu boş yere uzatıp tekrarlamak istemiyorum.

Wiring.c dosyamız init() fonksiyonu ile bitmektedir. Arduino'nun temel kodlarını burada görmemiz mümkündür.

-4- Arduino Ayaklarını Tanımlayan Dosyalar

Arduino çekirdeğini incelediğimizde digitalWrite(), digitalRead(), analogRead() gibi fonksiyonlar için ayak tanımlaması yapan dosyaların kullanıldığını görürüz. Bu ayak tanımlamaları biraz karışık olmaktadır. Ayak adı belirtme adına bu kadar programlama zahmetine girilmektedir. AVR programlamada PORT adı ve PD1, PD2 şeklinde ayak adı belirtebiliyorduk. Fakat bunların aslında hafıza adresi ve bit değerinden ibaret olduğunu söyleyebiliriz. AVR-GCC derleyicisinde iom8.h dosyasında Atmega8'in tanımlamaları yer alıyordu. Bu dosyayı ilk yazıda biraz incelesek de ayak kısmına hiç değinmemiştik. Şimdi bu dosyadan örnek bir parça alalım ve inceleyelim.

```
/* PORTB */
#define PB7    7
#define PB6    6
#define PB5    5
#define PB4    4
#define PB3    3
#define PB2    2
#define PB1    1
#define PB0    0
```

Görüldüğü gibi burada PB7 ile adlandırılan ayak adı aslında 7 rakamından başka bir şey değil. Yani (1<<PB7) yerine (1<<7) demiş oluyoruz. C ile AVR programlama derslerinde bitwise operatörlerle giriş ve çıkışı anlattığımda nasıl giriş ve çıkış yapılabildiğinden bahsetmiştim fakat PORTB7 veya PB7 gibi adların aslında ne olduğunu sanırım anlatmamıştım. Bunu da burada anlatmış olalım. Aslında bu konuyu gömülü sistemler için C programlama derslerinde anlatmayı düşünsem de Arduino'nun kaynak kodunu incelemekte yardımcı olacağından şimdi anlatalım. Gömülü sistemler için C programlamayı anlattığımızda ise daha ayrıntılı olarak ele alırız.

Arduino'nun ayakları ve portları ile alakalı bit bazlı tanımlama ve fonksiyonların yer aldığı üç adet ayrı dosya görmekteyiz. Bunlardan birisi atmega8 klasöründe yer alan pins_atmega8.c dosyasıdır. Bu dosya Arduino kodlarını atmega8'in ayakları ile uyumlu hale getirir. Ne yazık ki Arduino kodlarını tek bir dosya ile her AVR mikrodenetleyiciye uyumlu hale getirmemiz mümkün değildir. O yüzden Arduino kaynak kodu içerisinde bol bol eklemelerin olduğunu görüyoruz. Eğer Arduino tam kapsamlı bir derleyici olabilseydi diğer derleyicilerde olduğu gibi tek bir kütüphane dosyasında ayakları ve yazmaçları tanımlar işin içinden çıkardı.

Bu dosyaların aşırı derecede uzun olduğunu gördüğüm için bu dosyaları indirip veya Github üzerinden açıp ayrı bir sekmede görüntülemeniz daha verimli olacaktır. Buraya kod bloklarını kopyalasam da orjinal dosyayı şu bağlantıdan açmanız gereklidir.

https://github.com/arduino/ArduinoCore-avr/blob/master-older/cores/atmega8/pins_atmega8.c

Bu dosyayı açıp ekranın bir köşesine alalım. Şimdi yukarıdan itibaren dosyadaki kodları inceleyelim ve neler yapıldığına bir bakalım.

```
#include <avr/io.h>
#include "wiring.h"
```

Burada wiring.h dosyasının alındığını görmekteyiz. Wiring dosyalarını önceden incelemenin faydasını şimdi görüyoruz. Önceden incelediğimiz için bu dosyayı da anlamamız kolaylaşacak.

```
#define NUM_PINS 28
#define NUM_PORTS 4

#define PB 2
#define PC 3
#define PD 4
```

Burada NUM_PINS olarak belirtilen ayak numarası 28 olarak belirlenmiştir ve NUM_PORTS olarak belirlenen port sayısı dört olarak belirlenmiştir. Ayrıca PB, PC, PD olarak belirlenen ve portları temsil eden adlar 2, 3 ve 4 olarak belirlenmiştir. Burada ne maksatla kullanıldığını çözemeyiz o yüzden incelemeye devam edelim.

```
int port_to_mode[NUM_PORTS + 1] = {
        NOT_A_PORT,
        NOT_A_PORT,
        _SFR_IO_ADDR(DDRB),
        _SFR_IO_ADDR(DDRC),
        _SFR_IO_ADDR(DDRD),
};
```

Burada port_to_mode adında bir dizi değişkeni tanımlanmıştır. Burada DDR yazmaçları kullanıldığı için bu değişkenin ayak modunu tanımlamada portları hazır hale getirdiğini gözlemleyebiliriz. Yani _SFR_IO_ADDR(DDRB) değeri ile içinde DDRB yazmacının adresini bulundurur veya diğer değerler ile DDRC veya DDRD yazmaçlarının adreslerini bulundurur. NOT_A_PORT ise 0 değerine eşittir ve pins_arduino.h dosyasında tanımlanmıştır. Bu dosyaları inceleyeceğiz.

```
int port_to_output[NUM_PORTS + 1] = {
        NOT_A_PORT,
        NOT_A_PORT,
        _SFR_IO_ADDR(PORTB),
        _SFR_IO_ADDR(PORTC),
        _SFR_IO_ADDR(PORTD),
};
```

Burada ise yine dizi değişkenine PORT yazmaçlarının adresleri atılmaktadır. Bu adresler ileride giriş ve çıkış işlemlerinde kullanılacaktır.

_SFR ile başlayan makroların tanımlarını ise AVR LibC referans sayfasında bulabilirsiniz. Biz sadece ne işe yaradığından bahsediyoruz.

https://www.nongnu.org/avr-libc/user-manual/sfr__defs_8h_source.html

```
int port_to_input[NUM_PORTS + 1] = {
        NOT_A_PORT,
        NOT_A_PORT,
        _SFR_IO_ADDR(PINB),
        _SFR_IO_ADDR(PINC),
        _SFR_IO_ADDR(PIND),
};
```

Burada ise yine aynı değer tanım ve değer atama bu sefer dijital giriş için kullanılan PIN yazmacı için kullanılmıştır.

```
pin_t digital_pin_to_port_array[] = {
        { NOT_A_PIN, NOT_A_PIN },

        { PC, 6 },
        { PD, 0 },
        { PD, 1 },
        { PD, 2 },
        { PD, 3 },
        { PD, 4 },
        { NOT_A_PIN, NOT_A_PIN },
        { NOT_A_PIN, NOT_A_PIN },
        { PB, 6 },
        { PB, 7 },
        { PD, 5 },
        { PD, 6 },
        { PD, 7 },
        { PB, 0 },

        { PB, 1 },
        { PB, 2 },
        { PB, 3 },
        { PB, 4 },
        { PB, 5 },
        { NOT_A_PIN, NOT_A_PIN },
        { NOT_A_PIN, NOT_A_PIN },
        { NOT_A_PIN, NOT_A_PIN },
        { PC, 0 },
        { PC, 1 },
        { PC, 2 },
        { PC, 3 },
        { PC, 4 },
        { PC, 5 },
};

pin_t *digital_pin_to_port = digital_pin_to_port_array;
pin_t *analog_in_pin_to_port = digital_pin_to_port_array;
pin_t *analog_out_pin_to_port = digital_pin_to_port_array;
```

Burada pin_t değişkenini görebiliriz. Muhtemelen typedef ile tanımlanmış bir uint16_t dizisi olsa gerektir. Çünkü değer atarken her ikişer adet tamsayı değeri atanmaktadır. Burada bir dizi tanımlanmış ve ayaklara göre port adları ve bit adları belirtilmiştir. Bu belirtmelerin ATmega8'in ayaklarıyla uyuşması gerekir. pin_t değişkeninin nerede tanımlandığını ise tüm dosyaları ve derleyici dosyalarını aratsam da bulamadım. Belki ileride karşımıza çıkar diyoruz ve burayı geçiyoruz.

-5- pins_arduino.h ve pins_arduino.c

Arduino'da ayak işlemlerini anlayabilmek için öncelikle bu iki dosyayı incelememiz lazımdır. Fakat ayak işlemlerini anlamaktan ziyade çalışan AVR kodlarını ve fonksiyonları anlamamız daha önemlidir. Aslında bu iki dosyayı atlamak istesem de sonra karşıma çıkan digitalWrite() gibi fonksiyonlarda geri dönüp bu dosyalardan belli kısımları anlatmam gerekecekti. O yüzden anlatılacak pek bir şey olmasa da bu iki dosyayı gözden geçireceğiz. Öncelikle pins_arduino.h dosyasını açalım ve kodları incelemeye başlayalım. Dosyaya aşağıdaki bağlantıdan erişebilirsiniz.

https://github.com/arduino/ArduinoCore-avr/blob/master-older/cores/arduino/pins_arduino.h

```
#define NOT_A_PIN 0
#define NOT_A_PORT 0

#define NOT_ON_TIMER 0
#define TIMER0A 1
#define TIMER0B 2
#define TIMER1A 3
#define TIMER1B 4
#define TIMER2  5
#define TIMER2A 6
#define TIMER2B 7

#define TIMER3A 8
#define TIMER3B 9
#define TIMER3C 10
#define TIMER4A 11
#define TIMER4B 12
#define TIMER4C 13
#define TIMER5A 14
#define TIMER5B 15
#define TIMER5C 16
```

Burada NOT_A_PIN ve NOT_A_PORT tanımlarına sıfır (0) değeri verilmiştir. Bu koddan itibaren bu değerlerin kullanıldığı yerlerde sıfır (0) olduğunu anlamamız gerekir. Sonrasında ise zamanlayıcıları temsil eden adlara çeşitli numaralar verilmiştir. Arduino'da zamanlayıcılar iki yerde kullanılıyordu. Bunlardan biri millis() gibi zaman fonksiyonları ile alakalıydı ve diğeri ise ayaklardan alınan PWM çıkışında kullanılmaktaydı. Dijital giriş ve çıkışta PWM çıkış denetimi yapılacağı için ve PWM çıkışının da pek çok ayaktan yapıldığı için böyle çeşitli tanımlar oluşturulmuştur.

```
extern const uint16_t PROGMEM port_to_mode_PGM[];
extern const uint16_t PROGMEM port_to_input_PGM[];
extern const uint16_t PROGMEM port_to_output_PGM[];

extern const uint8_t PROGMEM digital_pin_to_port_PGM[];
// extern const uint8_t PROGMEM digital_pin_to_bit_PGM[];
extern const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[];
extern const uint8_t PROGMEM digital_pin_to_timer_PGM[];
```

Burada çeşitli değer dönüşüm ve maskeleme değerlerinin toplandığı dizi değişkenleri tanımlanmıştır. Bunların farklı bir yanı PROGMEM olarak program hafızasına kaydedilmeleridir. Dosyanın devamında göreceğimiz üzere bu değişkenlere pek çok değer atanıyor ve fazlaca yer tutuyordu. O yüzden program hafızasına kaydedilmiştir.

```
#define digitalPinToPort(P) ( pgm_read_byte( digital_pin_to_port_PGM + (P) ) )
#define digitalPinToBitMask(P) ( pgm_read_byte( digital_pin_to_bit_mask_PGM +
(P) ) )
#define digitalPinToTimer(P) ( pgm_read_byte( digital_pin_to_timer_PGM + (P) ) )
#define analogInPinToBit(P) (P)
#define portOutputRegister(P) ( (volatile uint8_t *)( pgm_read_word(
port_to_output_PGM + (P))) )
#define portInputRegister(P) ( (volatile uint8_t *)( pgm_read_word(
port_to_input_PGM + (P))) )
#define portModeRegister(P) ( (volatile uint8_t *)( pgm_read_word(
port_to_mode_PGM + (P))) )
```

Burada çeşitli makro tanımlarını görmekteyiz. pgmspace.h kütüphane dosyasında mevcut olan fonksiyonları kullanmaktadır. Çeşitli ayak ayarlarından oluşan bu makroları yeri geldiğinde daha ayrıntılı inceleme şansımız olacak.

Şimdi pins_arduino.c dosyasını açalım ve dosyanın devamına bakalım. pins_arduino.c dosyasını aşağıdaki bağlantıdan açabilirsiniz.

https://github.com/arduino/ArduinoCore-avr/blob/master-older/cores/arduino/pins_arduino.c

```
#define PA 1
#define PB 2
#define PC 3
#define PD 4
#define PE 5
#define PF 6
#define PG 7
#define PH 8
#define PJ 10
#define PK 11
#define PL 12

#define REPEAT8(x) x, x, x, x, x, x, x, x
#define BV0TO7 _BV(0), _BV(1), _BV(2), _BV(3), _BV(4), _BV(5), _BV(6), _BV(7)
#define BV7TO0 _BV(7), _BV(6), _BV(5), _BV(4), _BV(3), _BV(2), _BV(1), _BV(0)
```

Burada yine çeşitli tanımları görmekteyiz. PA, PB, PC adıyla sembolize edilen portlara çeşitli rakam değerleri verilmiştir. Ayrıca REPEAT8, BVOTO7 ve BV7TO0 adımda üç makro tanımlanmıştır. Bu makronun kullanılışına ilerideki incelemelerimizde değinebiliriz. Burada ise şimdi PORT, DDR ve PIN yazmaçlarının kullanımına dair tanımları görmekteyiz.

```
#if defined(__AVR_ATmega1280__)
```

Bu kontrol yapısı ATmega1280 mikrodenetleyicisine uyumluluk için yazılmıştır. Daha doğrusu Wiring dili orjinal haliyle ATmega1280 mikrodenetleyicisi için yazılmıştır. İlk Arduino ise ATmega8 için yapılmıştır. O yüzden ana kod bu olsa da biz Arduino kısmını inceleyeceğimiz için bu parçayı atlayalım.

```
const uint16_t PROGMEM port_to_mode_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        &DDRB,
        &DDRC,
        &DDRD,
};
```

Bu diziler port adlarını haritalandırır ve çeşitli fonksiyonlara uyumlu hale getirir. Burada da üç adet DDR yani veri yönü yazmacı görmekteyiz. Bunlar PORTB, C ve D'nin ayaklarının giriş ya da çıkış olacağını belirler. Bu veri yapısının pinMode() fonksiyonu için kullanılacağını kestirebiliriz.

```
const uint16_t PROGMEM port_to_output_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        &PORTB,
        &PORTC,
        &PORTD,
};

const uint16_t PROGMEM port_to_input_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        &PINB,
        &PINC,
        &PIND,
};
```

Burada ise yine aynı şekilde PIN ve PORT yazmaçlarını görüyoruz. Bu veri yapısı üzerinde PORT tabanlı işlemler yapılacaktır. Sadece PORT değerinin olduğuna dikkat ediniz.

```
const uint8_t PROGMEM digital_pin_to_port_PGM[] = {
        PD, /* 0 */
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PB, /* 8 */
        PB,
        PB,
        PB,
        PB,
        PB,
        PC, /* 14 */
        PC,
        PC,
        PC,
        PC,
        PC,
};
```

Bu yapı ise ayak numaralarını bulundurur. Bu ayaklar sıra ile D portuna sonra B portuna en son ise C portuna bağlıdır. Arduino'da 0-14 arası dijital ayak 14'den sonraki ayakların da analog ayak olduğunu hatırlayalım. Burada da o ayakların hangi portta olduğu belirtilmiştir.

```
const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[] = {
        _BV(0), /* 0, port D */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
        _BV(6),
        _BV(7),
        _BV(0), /* 8, port B */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
        _BV(0), /* 14, port C */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
};
```

Burada ise yukarıdaki veri yapısına benzer şekilde fakat bit değerlerini bulunduran bir yapı olduğunu görmekteyiz. Bu iki veri yapısındaki veriler Arduino ayağının nasıl kullanılacağını belirler. Biz AVR'de tek bir ayaktan çıkış almak için port ve bit değerini kullanıyorduk. Bu iki değere ihtiyacımız olduğu için bu iki değer burada sırayla belirtilmiştir.

```
const uint8_t PROGMEM digital_pin_to_timer_PGM[] = {
        NOT_ON_TIMER, /* 0 - port D */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        // on the ATmega168, digital pin 3 has hardware pwm
#if defined(__AVR_ATmega8__)
        NOT_ON_TIMER,
#else
        TIMER2B,
#endif
        NOT_ON_TIMER,
        // on the ATmega168, digital pins 5 and 6 have hardware pwm
#if defined(__AVR_ATmega8__)
        NOT_ON_TIMER,
        NOT_ON_TIMER,
#else
        TIMER0B,
        TIMER0A,
#endif
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 8 - port B */
        TIMER1A,
        TIMER1B,
#if defined(__AVR_ATmega8__)
        TIMER2,
#else
        TIMER2A,
```

```
#endif
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 14 - port C */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
};
#endif
```

Burada ise yine Arduino ayaklarının zamanlayıcıya bağlı olup olmadığı belirtilmiştir. Burada 0'dan başlayıp 19'a kadar gittiğine dikkat edin. Bu Arduino ayak numaralarını temsil etmektedir.

Buraya kadar ayaklar ile ilgili dosyalara kısa bir göz attık. Bundan sonra ise programlamada kullandığımız fonksiyonları inceleyeceğiz. Bu kısmı öğrenmenin bize bir getirisi olmasa da program fonksiyonlarını daha rahat anlamamızda faydası olacak. Biz AVR programlarken dijital giriş ve çıkış ayaklarına doğrudan bir erişim sağlıyorduk. Burada bu erişim dolaylandırılmıştır. Bu da ayak numarası kullanma adına yapılan bir işlemdir.

-6- DigitalWrite, DigitalRead ve PinMode

Şimdi sıkıcı kısmı sonunda atladık ve çok sevdiğim fonksiyon inceleme kısmına geldik. Siz de değer tanımlamalarını, makroları ve değişkenleri değil işleyen kodları görmek istemiş olsanız gerektir. Şimdi Arduino'da en sık kullanılan ve en temel fonksiyonlardan olan dijital giriş ve çıkış ile alakalı fonksiyonları inceleyeceğiz. Bunun için öncelikle bizim ile beraber wiring_digital.c dosyasını açıp bir taraftan dosyanın tamamını incelemeniz gereklidir.

Bu dosyayı şuradan açıp inceleyebilirsiniz.

https://github.com/arduino/ArduinoCore-avr/blob/master-older/cores/arduino/wiring_digital.c

Arduino'nun kaynak kodunu incelerken aynı zamanda Wiring'in kaynak kodunu da incelediğimizi belirtelim. Arduino'nun Wiring'in kodlarının kopyalanmasıyla meydana geldiğini biliyoruz. O yüzden bu ve buna benzer platformların tamamını incelediğimizi iddia edebiliriz. İçindeki kodlar donanıma göre farklılık gösterse de mantığı ve metodu benzerlik göstermektedir.

```
#include "wiring_private.h"
#include "pins_arduino.h"
```

Burada iki adet kütüphane dosyasının alındığını görüyoruz. Bu wiring_private.h dosyası içinde wiring.h dosyasını da çağırıyor. O yüzden kodu incelerken üç adet dosyaya daha göz atmamız gerekebilir.

```
void pinMode(uint8_t pin, uint8_t mode)
{
	uint8_t bit = digitalPinToBitMask(pin);
	uint8_t port = digitalPinToPort(pin);
	volatile uint8_t *reg;

	if (port == NOT_A_PIN) return;

	// JWS: can I let the optimizer do this?
	reg = portModeRegister(port);

	if (mode == INPUT) *reg &= ~bit;
	else *reg |= bit;
}
```

İşte pinMode() fonksiyonumuzun yapısı bu şekildedir. Bu fonksiyonu belki yüzlerce kere kullanırız fakat arka planda neler olduğundan haberimiz olmaz. AVR programlarken tek bir satırda bunu yapabilirken Ardunio'da onlarca gereksiz işlemin yürütüldüğünü görmekteyiz. Sadece pinMode değil digitalWrite ve digitalRead fonksiyonlarında da onlarca komut yürütülür ve fonksiyonların işleyişleri oldukça yavaşlar. Ben birkaç led yakayım, motor süreyim bana yeter diyorsanız sorun yok. Fakat ciddi bir uygulama yapmaya kalkayım diyorsanız sıkıntı çıkarabilir. Arduino'yu öğrenenlerin çoğu da oyuncak niyetine oynamak için öğrenmiyor açıkcası. Mühendislik öğrencisi Arduino'yu öğrenip kullansa dahi bunları bilerek kullanması gereklidir.

Bu fonksiyon uint8_t pin ve uint8_t mode olarak iki argüman alıyor. Pin adındaki değişkene ayak numarasını ve mode adındaki değişkene ise INPUT ya da OUTPUT olarak bir değişken koyduğumuzu hatırlatalım. Wiring.h dosyasını incelediğimizde INPUT'un sıfır (0), OUTPUT'un ise bir (1) olduğunu görürürüz. Yani 1 ve 0 değerlerinden başka bir şey değil. Fonksiyonun nasıl bir argüman aldığını tasavvur ettikten sonra bu argümanların nasıl kullanıldığına bakalım.

```
uint8_t bit = digitalPinToBitMask(pin);
```

Burada pins_arduino.h dosyasında yer alan bir makro kullanılmış. Bu makro digital_pin_to_bit_mask değişkeninden uygun değeri okuyup geri döndürür. Bu değişkeni ise pins_arduino.c dosyasında görüyoruz. Örneğin değerimiz sıfır ise _BV(0) değeri elde edilmiş oluyor. Geri kalanı ise PORT değerinin bitlerine göre sıralanmış. Örneğin 8. ayağımızın _BV() değeri de sıfır oluyor fakat PORTB kullanılmış oluyor. Burada hangi portun kullanıldığı değil hangi ayağa karşılık gelen bitin kullanıldığı önemli. Önceki yazıda yer versem de burada tekrar bahsedelim. Burada pin argümanı ile okuma yapılan ve sonrasında bit değişkenine aktarılan sabitin veri yapısı şu şekildedir.

```
const uint8_t PROGMEM digital_pin_to_bit_mask_PGM[] = {
        _BV(0), /* 0, port D */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
        _BV(6),
        _BV(7),
        _BV(0), /* 8, port B */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
        _BV(0), /* 14, port C */
        _BV(1),
        _BV(2),
        _BV(3),
        _BV(4),
        _BV(5),
};
```

Burada ayak sırasına göre _BV() değerleri alınır ve geliştirici tarafından tanımlanan sabit bir değerdir. Bu değerleri ne zaman değiştirebiliriz derseniz, Arduino platformunu kullanan kendimize ait bir geliştirme kartı üretmek istiyorsak bu bitleri donanıma göre değiştirebiliriz.

```
uint8_t port = digitalPinToPort(pin);
```

Dijital çıkış için pin yani portun bit değerini almayı başardık. Fakat hangi portun kullanılacağı hala meçhul. Bunun için yine böyle bir makro kullanıyoruz ve yine pin değişkenine tanımlı PORT adını buradan öğreniyoruz. digitalPinToPort() makrosunu yine pins_arduino.h dosyasında buluyoruz. Bu makro yine digital_pin_to_port sabit diziye ait değeri pins_arduino.c dosyasından okuyup geri döndürüyor. Şimdi bu sabit diziye bir bakalım.

```
const uint8_t PROGMEM digital_pin_to_port_PGM[] = {
        PD, /* 0 */
```

```
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PD,
        PB, /* 8 */
        PB,
        PB,
        PB,
        PB,
        PC, /* 14 */
        PC,
        PC,
        PC,
        PC,
        PC,
};
```

Görüldüğü gibi burada ayak numarasına göre port adları önceden belirtilmiş. Bu port adları kartın tasarımıyla uyuşmak zorundadır. Örneğin 0'dan 7'ye kadar olan dijital ayaklar PORTD'yi kullanır. Sonrası ise PORTB'yi kullanır. Analog ayaklar ise PORTC'yi kullanır ve 14'den itibaren başlar. Bunu kart üzerinde yazılı halde de görebilirsiniz. Burada kartın tasarımına bağlı olarak bir dizilim yapılmıştır. Örneğin 0 numaralı ayağı seçtiysek bu değer içinden PD değeri alınacaktır. Yukarıdaki yapı ile mantığı aynı olduğu için kolaylıkla anlayabilirsiniz.

```
volatile uint8_t *reg;
```

Burada reg adında bir işaretçi değişkeni tanımlanmıştır. Bunun kullanımına sonra geleceğiz.

```
if (port == NOT_A_PIN) return;
```

Programlamayı hiç bilmiyoruz ve bariz bir hata yaptık ve kafamıza göre bir numara yazdık. Bu durumda NOT_A_PIN değeri elde edilir ve port değişkenine eklenir. Bu yapı kullanıcı hatasını engelleme yönünde yapılmış bir kontrol yapısıdır. Eğer ayak numarası kısmına 500 yazıp bir şey olmasını beklerseniz hiçbir şey olmayacaktır. return ile fonksiyondan çıkıp program işlemeye devam eder.

```
reg = portModeRegister(port);
```

Bizim reg değişkenimize burada bir değer atanmaktadır. Bu makronun yine pins_arduino.h başlık dosyasında olduğunu görüyoruz. Bu ise port_to_mode sabit diziden verileri almaktadır. Bakalım argüman olarak kullandığımız port değerine göre hangi değerleri alabiliyoruz.

```
const uint16_t PROGMEM port_to_mode_PGM[] = {
        NOT_A_PORT,
        NOT_A_PORT,
        &DDRB,
        &DDRC,
        &DDRD,
};
```

Bu değerler doğrudan yazmaçların adlarını adresliyor. Döndük dolaştık yine yazmaçlara geldik. Fakat yazmaçlara gelmek için de o kadar prosedürle uğraşmak zorunda kaldık.

Birsürü değer, sabit ve dizi işte bu yazmaçlara ulaşmak içindi. Şimdi elimizde BV_() olarak bit değeri var ve PORT olarak yazmaç var. Artık AVR programlamada olduğu gibi işlemimizi yapabiliriz.

```
if (mode == INPUT) *reg &= ~bit;
```

mode adıyla aldığımız argümanı ancak şimdi kullanabiliyoruz. Bu mode INPUT ya da OUTPUT olarak iki ayrı değer alabiliyordu. Burada INPUT_PULLUP değerini göremiyorum nedense. Ya o dönem koda eklenmemiş ya da kodun başka bir yerinde karşımıza çıkacak. Fakat şimdiye kadar görmedik. Bunu yok sayalım.

Burada INPUT'un sıfır (0) olduğundan bahsetmiştik. Kısacası yukarıdaki kod şu işlemi yapmaktadır.

```
DDRx &=~ BV_(pin);
```

Bu bildiğimiz AVR kodudur, başka birşey değil. DDR yazmacındaki bir biti sıfır durumuna getirir. Bunu yapmak içinse yine en temel yol olan bitwise operatörlerini kullanır. Bu komutu çalıştırmak için nerelere gidip geldiğimizi görüyor musunuz ?

Şimdi giriş tanımlamasını yaptık fakat çıkış tanımlaması kaldı. Onu da şu komutla yapıyoruz.

```
else *reg |= bit;
```

Program burada bizim OUTPUT yazıp yazmadığımıza bakmıyor. INPUT yani 0'dan başka değer yazdıysak bunu çıkış olarak anlıyor ve INPUT değilse çıkış olarak tanımlıyor. OUTPUT yerine 50 de yazsanız aynı yola çıkacaktır. Bu komut ise aşağıdaki komut ile aynı işi görmektedir. Yine o sabitten gidip PD değerini alır sonra gider PD değeri ile yazmaç adresini alır ve *reg'e koyar ve karşımıza yine aynı DDR yazmacı çıkar.

```
DDRx |= BV_(bit);
```

Bir satır kod ancak bu kadar zor yazılabilirdi. Üç tane yazmaç öğrenmemek, iki operatör kullanmamak için ne kadar zahmete girildiğini görmüş olsanız gerek.

```
static inline void turnOffPWM(uint8_t timer)
{
	if (timer == TIMER1A) cbi(TCCR1A, COM1A1);
	if (timer == TIMER1B) cbi(TCCR1A, COM1B1);

#if defined(__AVR_ATmega8__)
	if (timer == TIMER2) cbi(TCCR2, COM21);
#else
	if (timer == TIMER0A) cbi(TCCR0A, COM0A1);
	if (timer == TIMER0B) cbi(TCCR0A, COM0B1);
	if (timer == TIMER2A) cbi(TCCR2A, COM2A1);
	if (timer == TIMER2B) cbi(TCCR2A, COM2B1);
#endif

#if defined(__AVR_ATmega1280__)
	if (timer == TIMER3A) cbi(TCCR3A, COM3A1);
	if (timer == TIMER3B) cbi(TCCR3A, COM3B1);
	if (timer == TIMER3C) cbi(TCCR3A, COM3C1);
	if (timer == TIMER4A) cbi(TCCR4A, COM4A1);
	if (timer == TIMER4B) cbi(TCCR4A, COM4B1);
```

```
        if (timer == TIMER4C) cbi(TCCR4A, COM4C1);
        if (timer == TIMER5A) cbi(TCCR5A, COM5A1);
        if (timer == TIMER5B) cbi(TCCR5A, COM5B1);
        if (timer == TIMER5C) cbi(TCCR5A, COM5C1);
#endif
}
```

Dosyanın devamında ise turnOffPWM fonksiyonunu görüyoruz. Bu fonksiyon oldukça basittir ve tek yaptığı iş zamanlayıcılara ait yazmaçları sıfırlar. Burada bir uyumluluk için karar yapısı mevcut olduğundan biraz kod kalabalığı var. Hem Atmega128 hem de ATmega8 için yazılmış. Fonksiyon aldığı argüman ile hangi zamanlayıcıyı kapatacağını belirliyor.

```
void digitalWrite(uint8_t pin, uint8_t val)
{
        uint8_t timer = digitalPinToTimer(pin);
        uint8_t bit = digitalPinToBitMask(pin);
        uint8_t port = digitalPinToPort(pin);
        volatile uint8_t *out;

        if (port == NOT_A_PIN) return;

        // If the pin that support PWM output, we need to turn it off
        // before doing a digital write.
        if (timer != NOT_ON_TIMER) turnOffPWM(timer);

        out = portOutputRegister(port);

        if (val == LOW) *out &= ~bit;
        else *out |= bit;
}
```

digitalWrite fonksiyonu da yukarıda anlattığımız pinMode fonksiyonundan çok farklı değildir. Arduino kodlarında aynı şeyler tekrar tekrar yapılsa da biz kendimizi yorup tekrar tekrar anlatmayacağız. Bu fonksiyonda farklı noktaları ele alıp pinMode ile aynı olan kısımları geçeceğiz.

```
uint8_t timer = digitalPinToTimer(pin);
```

Burada farklı bir makronun kullanıldığını görüyoruz. Bunun sebebi ise dijital ayaklardan PWM çıkışının da alınmasıdır. Bu PWM çıkışı alınan zamanlayıcıları tespit etmek ve durdurmak gereklidir. PWM çıkışı alınan bir ayaktan dijital çıkış alınması beklenemez. Bunu programcı programlarken dikkat edemeyebilir. Sonuçta Arduino mühendislere yönelik değil sanatçılara ve tasarımcılara veya yeni öğrencilere yönelik bir alet. Bunu programın hatırlayıp yapması gerekiyor ve bunun için öncelikle bu makroyu kullanıyoruz.

Bu makronun yine pins_arduino.h dosyasında olduğunu görüyoruz. Diğer makrolarla aynı işi yapıp digital_pin_to_timer adındaki dizi sabitinden değeri okuyup geri döndürüyor. Peki bu sabitte nasıl değerler yer alıyor şimdi ona pins_arduino.c dosyasından bakalım.

```
onst uint8_t PROGMEM digital_pin_to_timer_PGM[] = {
        NOT_ON_TIMER, /* 0 - port D */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        // on the ATmega168, digital pin 3 has hardware pwm
#if defined(__AVR_ATmega8__)
        NOT_ON_TIMER,
```

```
#else
        TIMER2B,
#endif
        NOT_ON_TIMER,
        // on the ATmega168, digital pins 5 and 6 have hardware pwm
#if defined(__AVR_ATmega8__)
        NOT_ON_TIMER,
        NOT_ON_TIMER,
#else
        TIMER0B,
        TIMER0A,
#endif
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 8 - port B */
        TIMER1A,
        TIMER1B,
#if defined(__AVR_ATmega8__)
        TIMER2,
#else
        TIMER2A,
#endif
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER, /* 14 - port C */
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
        NOT_ON_TIMER,
};
#endif
```

Burada basit olarak tüm ayakların hangi zamanlayıcıyı kullandığı verisi yer alıyor. Bu veriler ise pins_arduino.h başlık dosyasında görülebilir.

```
#define NOT_ON_TIMER 0
#define TIMER0A 1
#define TIMER0B 2
#define TIMER1A 3
#define TIMER1B 4
#define TIMER2  5
#define TIMER2A 6
#define TIMER2B 7
```

Her bir zamanlayıcıya farklı numaralar verilse de kontrol yapısında NOT_ON_TIMER yani sıfır(0)'ın önemi görülüyor. Burada sadece NOT_ON_TIMER'ın önemini görmekteyiz.

```
if (timer != NOT_ON_TIMER) turnOffPWM(timer);
```

Makroları kullanarak önceden tanımlanan sabit dizilerden değeri çağırdık ve değerimiz NOT_ON_TIMER'dan (0) farklı bir değer oldu. Bu bacağa bağlı bir zamanlayıcının olduğu anlamına gelir. TIMER0A, TIMER0B olarak sırayla adlandırdığımız numaraların önemini ise turnOffPWM fonksiyonunda görüyoruz. Eğer sıfırdan farklı bir değer elde ettiysek turnOffPWM fonksiyonu yürütülür.

Şimdi turnOffPWM fonksiyonunu daha iyi anlayabiliriz. Örneğin elimizde TIMER1B yani 4 değeri ile turnOffPWM fonksiyonuna gidiyoruz. Bu fonksiyon sıralı karar yapılarından oluşuyor ve bizim değerimize uygun kodu yürütüyor.

```
if (timer == TIMER1B) cbi(TCCR1A, COM1B1);
```
TIMER1B değerini yine ayaklara göre önceden tanımlanmış sabitlerden aldığımızı hatırlatalım. Bu değer ise burada TCCR1A'nın COM1B1 bitini sıfırlamak için kullanılmış. Normalde programcı bu biti eliyle sıfırlaması gerekirken burada program bizim yerimize yapıyor.

```
if (val == LOW) *out &= ~bit;
        else *out |= bit;
```
Bu fonksiyonda ise yukarıda anlattığımızın aynısı yapılıyor. Bu sefer PORT yazmaçları üzerinden yürütülüyor. O değeri de portOutputRegister makrosu ile elde ediyoruz. İşleyişini yukarıda anlattığım için tekrar tekrar anlatmayacağım.

```
int digitalRead(uint8_t pin)
{
        uint8_t timer = digitalPinToTimer(pin);
        uint8_t bit = digitalPinToBitMask(pin);
        uint8_t port = digitalPinToPort(pin);

        if (port == NOT_A_PIN) return LOW;

        // If the pin that support PWM output, we need to turn it off
        // before getting a digital reading.
        if (timer != NOT_ON_TIMER) turnOffPWM(timer);

        if (*portInputRegister(port) & bit) return HIGH;
        return LOW;
}
```
DigitalRead fonksiyonunun iç yapısını ise yukarıda görmekteyiz. Fonksiyon yapı olarak digitalWrite() ve pinMode() fonksiyonlarına çok benzemektedir. Aynı kodları çıkardığımızda geriye kalan yine AVR kodu kalmaktadır. Burada ise PIN yazmaçlarındaki bit değeri okunur ve HIGH ya da LOW olarak geri döndürülür. Mantığı aynı olduğu için tek tek açıklamamıza gerek yok.

Sonraki yazılarımızda geriye kalan dosyaları inceleyeceğiz. Şu ana kadar kodların temelini incelemiş olduk ve işin mantığını buradan anladık.

-7- AnalogRead() ve AnalogWrite() Fonksiyonları

Önceki yazımızda Arduino'nun digitalWrite(), digitalRead() gibi dijital giriş ve çıkış fonksiyonlarını incelemiştik. Şimdi ise sıra analog fonksiyonlara geldi. Analog fonksiyonlar mikrodenetleyicinin dijital bir sistem olmasından dolayı kısıtlıdır. AVR programlamada bile yapabileceğimiz analog işlem ADC ve PWM ile sınırlı kalmaktadır. Yani kare dalga üretebiliriz ya da gelen analog sinyali okuyabiliriz. Analog işlemler hakkında bilgi için lütfen Arduino Eğitim Kitabı'nın ve C ile AVR Programlama derslerinin ilgili yerlerine bakın. Burada sadece kodu incelemekle yetineceğiz.

Bizim analog işlemleri yaptırdığımız kodların yer aldığı dosyanın adı wiring_analog.c dosyasıdır. Bu dosyayı bu yazıyı okumadan önce açıp ekranın bir köşesine çekmenizi tavsiye ederiz. Dosyayı aşağıdan okuyabilirsiniz.

https://github.com/arduino/ArduinoCore-avr/blob/master-older/cores/arduino/wiring_digital.c

Bu dosya wiring_private.h ve pins_arduino.h adında iki kütüphane dosyasını çağırıyor. Bu iki dosyayı da daha önce inceledik.

```
void analogReference(uint8_t mode)
{
	// can't actually set the register here because the default setting
	// will connect AVCC and the AREF pin, which would cause a short if
	// there's something connected to AREF.
	analog_reference = mode;
}
```

analogReference fonksiyonu ile ADC referansının ne olacağını kararlaştırıyorduk. Burada ise bizim argüman olarak gönderdiğimiz değeri analog_reference değişkenine aktardığını görüyoruz. Burada analog referansına dair bir işlem yapılmamıştır.

```
int analogRead(uint8_t pin)
{
	uint8_t low, high;

	// set the analog reference (high two bits of ADMUX) and select the
	// channel (low 4 bits).  this also sets ADLAR (left-adjust result)
	// to 0 (the default).
	ADMUX = (analog_reference << 6) | (pin & 0x07);

#if defined(__AVR_ATmega1280__)
	// the MUX5 bit of ADCSRB selects whether we're reading from channels
	// 0 to 7 (MUX5 low) or 8 to 15 (MUX5 high).
	ADCSRB = (ADCSRB & ~(1 << MUX5)) | (((pin >> 3) & 0x01) << MUX5);
#endif

	// without a delay, we seem to read from the wrong channel
	//delay(1);

	// start the conversion
	sbi(ADCSRA, ADSC);

	// ADSC is cleared when the conversion finishes
	while (bit_is_set(ADCSRA, ADSC));
```

```
        // we have to read ADCL first; doing so locks both ADCL
        // and ADCH until ADCH is read.  reading ADCL second would
        // cause the results of each conversion to be discarded,
        // as ADCL and ADCH would be locked when it completed.
        low = ADCL;
        high = ADCH;

        // combine the two bytes
        return (high << 8) | low;
}
```

Burada ise analogRead adında ADC okuma fonksiyonunu görmekteyiz. Aslında bu fonksiyona çok benzer bir fonksiyonu C ile AVR programlama derslerinin ADC konularında görmüştük. Klasik ADC okuma fonksiyonunu buraya taşıdıklarını görüyoruz. Tabi ki uyumluluk açısından #if defined gibi eklemelerin var olduğunu görmekteyiz. analogRead fonksiyonu aynı AVR kodunda olduğu gibi bir ayak numarası almak zorundadır. Bu ayak numarası ise ADMUX yazmacına yazılır. Böylelikle hangi ayaktan okuma yapılacağı kararlaştırılır.

```
uint8_t low, high;
```

Analog okuma değerinin 10-bit olup 8 bitlik bir değişkene veya yazmaca sığamayacağını biliyoruz. Bu yüzden düşük ve yüksek bayt olarak iki bayt verisini ortaklaşa kullanarak okumamız gerekli. Sonrasında ise bunu 16 bit olarak geri döndürebiliriz. Burada ise düşük ve yüzsek yazmaçlardan veri almak için iki adet 8 bitlik değişken tanımlandığını görmekteyiz.

```
ADMUX = (analog_reference << 6) | (pin & 0x07);
```

Burada ADMUX yazmacı hem analog referans verisini hem de multiplexer verisini içeriyor. Burada iki değerin uygun bitlere yazıldığını görmekteyiz. Daha ayrıntılı bilgi için ADC yazmaçlarını konu aldığımız derslere bakabilirsiniz.

```
#if defined(__AVR_ATmega1280__)
        // the MUX5 bit of ADCSRB selects whether we're reading from channels
        // 0 to 7 (MUX5 low) or 8 to 15 (MUX5 high).
        ADCSRB = (ADCSRB & ~(1 << MUX5)) | (((pin >> 3) & 0x01) << MUX5);
#endif
```

Eğer ATmega1280 kullanılıyorsa farklı kodlar çalıştırılır. Bizim hedefimiz Arduino'nun kullandığı ATmega8 ve aynı serinin 16 ve 32 kodlu entegreleri olduğu için bu kısımları atlayacağız.

```
        sbi(ADCSRA, ADSC);

        // ADSC is cleared when the conversion finishes
        while (bit_is_set(ADCSRA, ADSC));
```

Burada ADC çevirim başlatma biti bir (1) yapılır ve sonrasında ise çevirim bitene kadar program sonsuz döngüye sokulur. Aslında başka bir şekilde de kodu yazmak pek mümkün değildir. Çünkü mikrodenetleyicinin kurallarına göre programı yazmadıkça doğru çalıştırmamız mümkün değildir.

```
        low = ADCL;
        high = ADCH;
```

Burada ADCL'yi önce okumak gereklidir aksi halde doğru sonuç alınmaz. Bunu derslerimizde de söylemiştik. ADCL yani ADC dönüşümünün verisini tutan alt yazmaç low değişkenine üst yazmaç ise high değişkenine aktarılır.

```
        return (high << 8) | low;
```

Bunu AVR derslerinde de anlatmıştım. Üst bitler sekiz adım sola kaydırılır ve alt baytın önüne yazılır. Böyellikle 16-bit bir veri için doğru dizilim elde edilmiş olur.

```
void analogWrite(uint8_t pin, int val)
{
        // We need to make sure the PWM output is enabled for those pins
        // that support it, as we turn it off when digitally reading or
        // writing with them.  Also, make sure the pin is in output mode
        // for consistenty with Wiring, which doesn't require a pinMode
        // call for the analog output pins.
        pinMode(pin, OUTPUT);

        if (digitalPinToTimer(pin) == TIMER1A) {
                // connect pwm to pin on timer 1, channel A
                sbi(TCCR1A, COM1A1);
                // set pwm duty
                OCR1A = val;
        } else if (digitalPinToTimer(pin) == TIMER1B) {
                // connect pwm to pin on timer 1, channel B
                sbi(TCCR1A, COM1B1);
                // set pwm duty
                OCR1B = val;
        } else if (digitalPinToTimer(pin) == TIMER2) {
                // connect pwm to pin on timer 2, channel B
                sbi(TCCR2, COM21);
                // set pwm duty
                OCR2 = val;
                } else if (val < 128)
                digitalWrite(pin, LOW);
        else
                digitalWrite(pin, HIGH);,
}
```

Bu analogWrite fonksiyonunu atmega8'e göre tekrar düzenleyip buraya koydum. Yoksa kolay kolay anlatmam mümkün değildi. Çünkü kodun ortasında atmega1280 komutu yer alıyor sonra yine atmega8 komutu yer alıyor ardından ise tekrar atmega1280 komutlarına geçiliyordu. Kafanızın karışmaması için atmega8 için işletilen komutları yukarıda sırasıyla görmeniz mümkündür. analogWrite fonksiyonu Arduino'da PWM üretmek için kullanılıyordu.

```
pinMode(pin, OUTPUT);
```

Bu fonksiyon pin ve value adında iki argüman alıyor. Bunlardan biri çıkış alınacak ayak numarası öteki ise çıkış değeridir. Bu çıkış değeri 0-255 arasında olup 8 bitlik bir değerdir. Aslında ATmega'dan 16 bitlik PWM sinyali almak mümkündür. Fakat yazılımda böyle bir kısıtlama vardır.

```
if (digitalPinToTimer(pin) == TIMER1A) {
```

Bu kontrol yapısında digitalPinToTimer makrosu çalıştırılır ve her ayak için tanımlanmış zamanlayıcılar bulunur ve ortaya çıkarılır. Bu makroyu ve sabit diziyi önceki yazıda incelediğimiz için bir önceki yazıya bakabilirsiniz. Burada bunu tekrar incelemeyeceğiz.

```
sbi(TCCR1A, COM1A1);
// set pwm duty
OCR1A = val;
```
OCR1A yazmacı karşılaştırma değerini içeren yazmaçtı. Karşılaştırmaya kadar yazmaç saymaya devam eder ve karşılaştırma değerine gelinde taşma gerçekleşir ve sıfırlanırdı. Burada da val değerinin karşılaştırma değerini ve pek tabii görev döngüsünü (duty cycle) belirlediğini görmemiz mümkün.

```
else if (val < 128)
                digitalWrite(pin, LOW);
        else
                digitalWrite(pin, HIGH);
}
```
En sonunda ise başka bir kontrol yapısını görmekteyiz. Ayaklar zamanlayıcılara isabet etmeyince bunlar değere göre sabit sinyal veriyor. Bu programlayıcı hatasından kaynaklanan bir durumu kararlı hale getirmek için yapılmış olsa gerektir.

wiring_analog.c dosyasında bizi heyecanlandıran pek bir şey göremedik. Oldukça basit ADC okuma ve PWM üretme fonksiyonlarına rastladık. digitalWrite() gibi fonksiyonları AVR'ye uyarlamak biraz uğraştırıcı olsa da bu fonksiyonları neredeyse olduğu gibi AVR'de kullanmamız mümkün. Kaynak kodları incelerken diğer dosyaları incelediğimde ise bazılarının yazılımsal yönünün ağır bastığını ve C++ ile yazılmış olduğunu görüyorum. Diğer dosyalar ise AVR donanımına yönelik ve bizim derslerde anlattığımız işlemlerden çok da farklı değil. O yüzden Arduino kaynak kodunu incelemeyi burada bitirelim.

Bu kitapçıktan anlamamız gereken en önemli şey birkaç Arduino fonksiyonu ile ne mühendislik ne de geliştiricilik yapılabildiğidir. İki arduino fonksiyonu kullanmakla, hazır kodları kopyalamakla kendine maker diyen insanların olduğu bir sektörde yabancı makaleleri çat pat İngilizce ile yarım yamalak tercüme edebilen "İçerik yazarı" oluyor. Bu böyle devam ettikçe bizim bu alanda ses getirecek bir iş ortaya koymamız mümkün değildir. O yüzden ne kadar zahmetli olsa da en ileri seviye bilgiyi sizlere açıklamaya çalıştım. Her ne kadar okuyucusu az olsa da bir gün birilerinin işine muhakkak yarayacaktır.