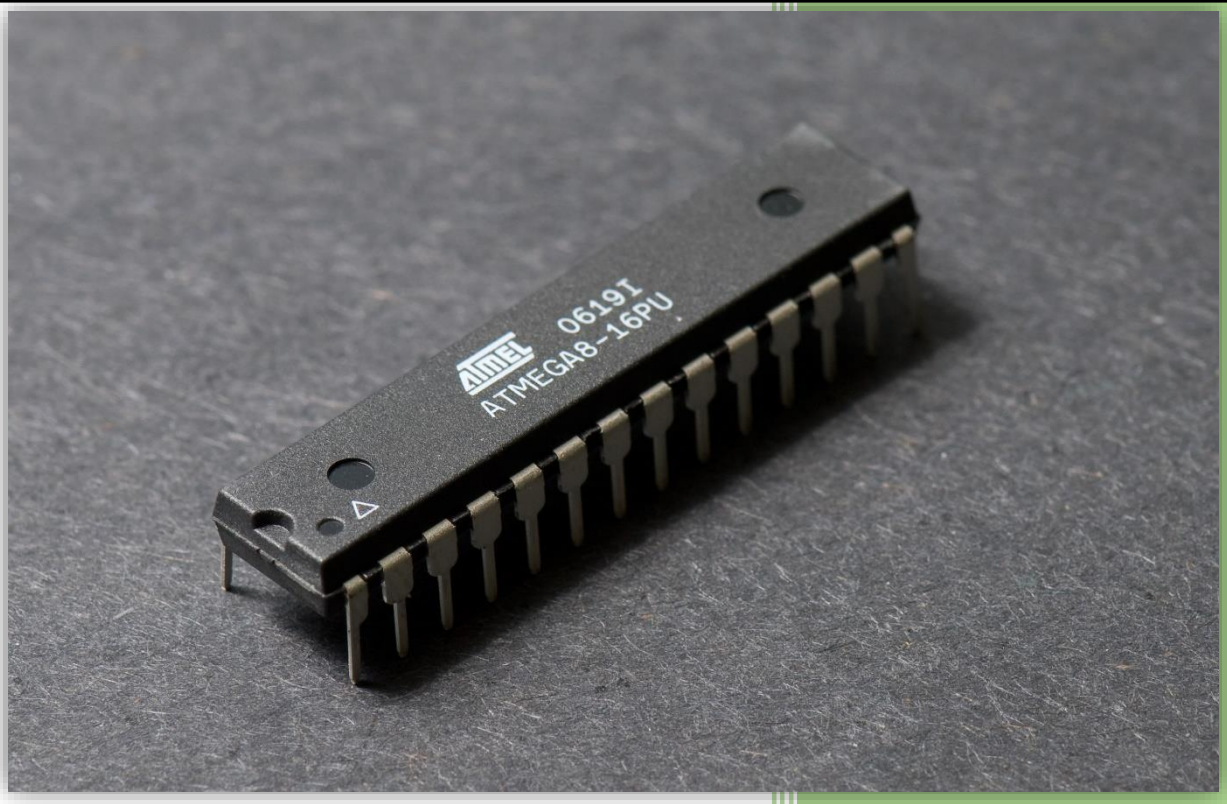
AVR ile Mikrodenetleyici Uygulamaları



Gökhan

DÖKMETAŞ

Gökhan DÖKMETAŞ Kimdir?

Bilgisayarla 1998 yılında henüz 4 yaşındayken tanışmıştım. 8-9 yaşımdan itibaren sürekli bilgisayar kullanmakta ve bilgisayarla alakalı teknolojilerle ilgilenmekteyim. Bu yüzden lisede Anadolu Meslek Lisesi Bilişim Teknolojileri bölümünü okudum. Lise yıllarında bilgisayarın yanında elektroniğe de meraklandım. Liseden mezun olduktan sonra bu alandan farklı bir fakülteye gitsem de amatör olarak gömülü sistemler başta olmak üzere elektronik ve bilgisayar bilimleri üzerinde çalışmaya devam ettim. Bu arada işin eğitiminde büyük bir açığın olduğunu, kaliteli Türkçe kaynakların bulunmadığını fark ettim. Bir yandan öğrenirken aynı zamanda da öğretme kararı aldım. İlk yazdığım kitap olan "Arduino Eğitim Kitabı" 2016'nın Nisan ayında yayınlandı. Sonrasında "Arduino ve Raspberry PI ile Nesnelerin İnterneti" adlı kitabı yazdım. Bu süreç içerisinde de aile firmamızda jeofizik görüntüleme cihazları ve metal detektörlerinin gömülü yazılımı ve dijital devre tasarımına yardımcı oldum ve olmaktayım. Yazarlık noktasında yazdığım kitaplardan olumlu bir netice elde edince lojikprob.com adlı sitemde teknik makaleler yazmaya devam ettim. Ayrıca bunun yanında Facebook grubumda da mentorluk ve bilgi paylaşımı yapmaktayım. Şu an web teknolojileri ve gömülü sistemlerde halen kendi başıma mesleği öğrenmeye devam ediyorum. Gazi Üniversitesi "Elektronik Teknolojisi" bölümü, Atatürk Üniversitesi Açıköğretim "Bilgisayar Programcılığı" ve Anadolu Üniversitesi Açıköğretim "Web tasarım ve kodlama" bölümlerinde öğrenciyim. Bir yandan meslek öğrenirken ve kendi işimi yaparken öteki yandan bu mesleğin eğitimini ve diplomasını almaya da önem veriyorum. Bana aşağıdaki iletişim adreslerinden ulaşabilirsiniz.

E-mail: gokhandokmetas0@gmail.com
Blog sayfam: www.lojikprob.com
Facebook profilim: https://www.facebook.com/lojikprob/
Facebook grubum: https://www.facebook.com/groups/lojikprob
LinkedIn profilim: https://www.linkedin.com/in/g%C3%B6khan-d%C3%B6kmeta%C5%9F-b9257a1b4/

Bu Kitap Kimler İçin Yazıldı?

"AVR ile Mikrodenetleyici Uygulamaları" kitabı gömülü sistemler üzerinde belli bir alt yapıya sahip kişilerin uygulama eksiğini gidermek amacıyla kaleme alınmıştır. Daha öncesinde LojikProb sitesinde yazdığım "C ile AVR Programlama" makalelerini okumanız ve AVR mikrodenetleyiciler hakkında teorik bilgiye sahip olmanız gereklidir. Orada yazdığım teorik bilgileri burada tekrar etmemeye özen göstereceğim. Bunun yanında dijital elektronik ve pratik elektronik bilgisi uygulamaları doğru anlayabilmek için gereklidir. İşin yazılım tarafında ise temel seviyede C programlama dilini bilmeniz gereklidir. Bunun için yine LojikProb sitesinde yer alan "Temel C Programlama" makalelerini okuyabilirsiniz. Bahsettiğim makalelerin PDF formatını aşağıdaki bağlantılardan indirebilirsiniz.

http://www.lojikprob.com/c-ile-avr-programlama-e-kitap-indir/
http://www.lojikprob.com/c/temel-c-programlama-gokhan-dokmetas-pdf-e-kitap/

Burada önemli olan uygulamayı bakarak yapmanız değil uygulama yaparak konuyu daha iyi anlamanız ve pratiğinizi geliştirmenizdir. Bunun için temel teorik bilgi şarttır. Bu kitap gömülü sistem geliştiriciliğini öğretmeyi hedef aldığı için mühendislik ve meslek yüksek okulu öğrencileri başta olmak üzere bu konu ile ilgilenen ve belli bir birikime sahip olan herkes bu kitaptan faydalanabilir.

Kitapta geçen uygulamaların proje dosyaları aşağıdaki bağlantıda yer almaktadır.

https://github.com/GDokmetas/AVRUygulamalari

Kitapta bir hata, yanlışlık ile karşılaştığınızda iletişim kanallarından lütfen bana bildiriniz. Aynı zamanda kitap hakkında görüş, öneri ve yorumlarınızı bekliyorum.

Kitabı Yazarken Kullandığım Kaynaklar

Gömülü sistemler hakkında Türkçe kaynak eksikliği hepimizin bildiği bir durum olsa da bu kaynak araştırmasını oldukça ciddiye alan biri olarak İngilizce kaynakların da yeterli olmadığını söyleyebilirim. İnternette yer alan basit tutorial makaleleri veya Arduino projeleri asla gözünüzü boyamasın. Bu konu hakkında yazılan İngilizce kitaplar belki teorik bilgi verme konusunda yeterli olsa da iş uygulamaya gelince bu kitapları okumak yeterli gelmiyor. Bu durumda Github başta olmak üzere internet kaynaklarını kullanarak kod ve proje örnekleri üzerinden öğrenmeye devam edebilseniz de bir noktadan sonra bunun da yeterli olmadığını kendi kod örneklerinizi ve kütüphanelerinizi kendinizin ortaya koymanız gerektiğini görüyorsunuz. Ben de uygulamaları yaparken pek çok çalışmayan örnek ve kütüphanelerle karşılaştığımdan çoğu zaman kendi örneklerimi kendi başıma yapmak zorunda kaldım. Bunun dışında örnek kod ve kütüphaneler için Github sitesinden faydalandım. Kullandığım mikrodenetleyici ile alakalı üst düzey bilgiyi üreticinin sağlamış olduğu datasheet ve uygulama notu gibi teknik dokümanlardan elde ettim. Aynı zamanda AVR derleyicisinin kullanma kılavuzu da yazılım noktasında yardımcı oldu. Bu kılavuza aşağıdaki bağlantıdan erişebilirsiniz.

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

Bunun dışında AVR öğrenirken oldukça faydalı bulduğum iki İngilizce kitaptan da bahsedebilirim.

AVR Microcontroller and Embedded Systems: Using Assembly and C – M. Ali Mazidi
Embedded C Programming and the Atmel AVR- Richard H. Barnett

Bunun dışında işin pratik tarafını öğrenirken karşılaştığım sorunların çözümünü AVRfreaks forumu başta olmak üzere Stackoverlow ve Quora sitelerinde aradım. Bazen de cevabını bulamadığım sorunları kendi başıma sorunu çözmek zorunda kaldım.

Uygulamalara Başlamadan Önce

Bu kitapta yer alan uygulamaların hepsi Arduino UNO kartı üzerindeki AVR ATmega328p mikrodenetleyicisi, AVR-GCC derleyicisi ve Atmel Studio 7.0 programı kullanılarak gerçekleştirilmiştir. Bu durumda sizin de bir Arduino UNO kartı edinmeniz ve bilgisayarınıza Atmel Studio 7.0 geliştirme stüdyosunu yüklemeniz gereklidir. Eğer klon Arduino kartı kullanıyorsanız kartın üzerindeki USB-TTL çevirici çipi öğrenip bunun sürücüsünü ayrıca yüklemeniz gereklidir. Genellikle klonlarda CH341H çipi kullanıldığından bu sürücüyü indirme bağlantısını aşağıda vereceğim.

http://www.wch.cn/downloads/CH341SER_ZIP.html

Atmel Studio'yu Microchip'in sitesinden indirip kolayca kurabilirsiniz.

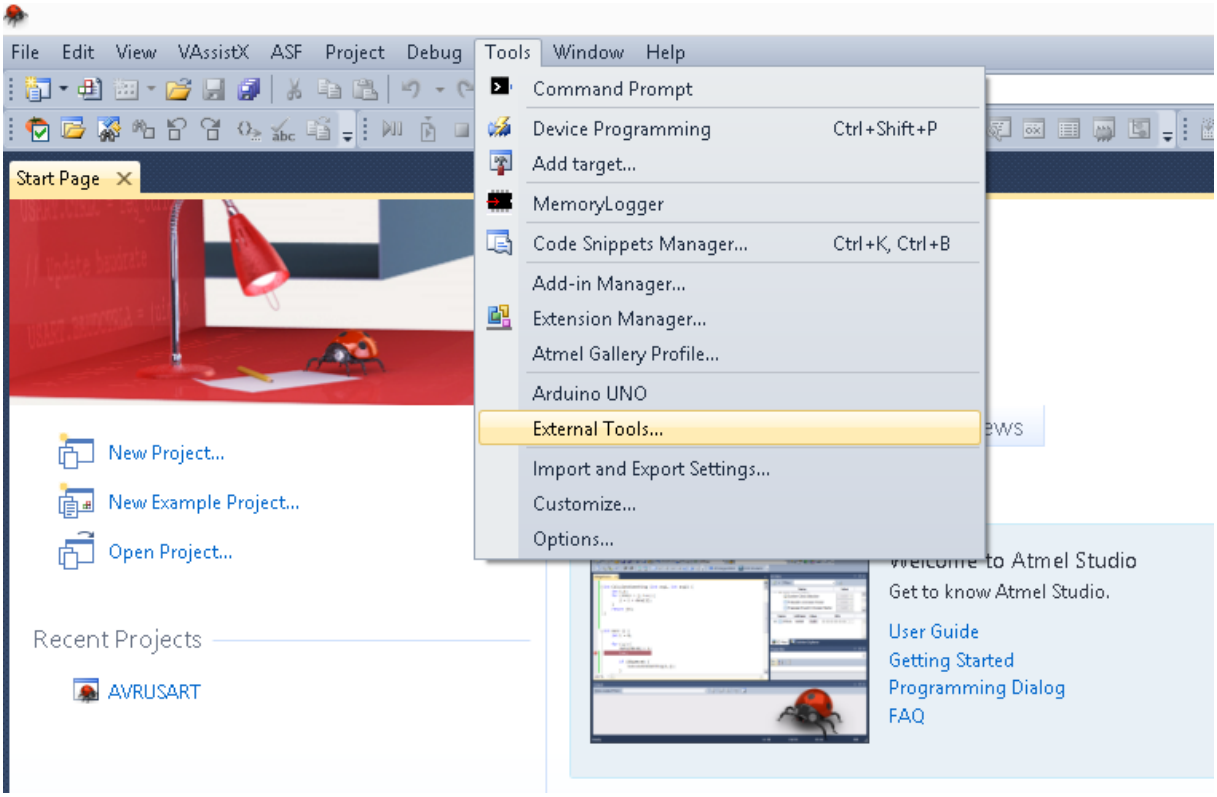
https://www.microchip.com/mplab/avr-support/atmel-studio-7

Bütün bu kurulumların ardından Arduino UNO'ya bir tık ile program atabilmeniz için Atmel Studio'da araç olarak tanımlamanız gereklidir. Bunun nasıl yapılacağını daha önceden yazdığım için burada yazdığım makaleyi doğrudan buraya kopyalıyorum.

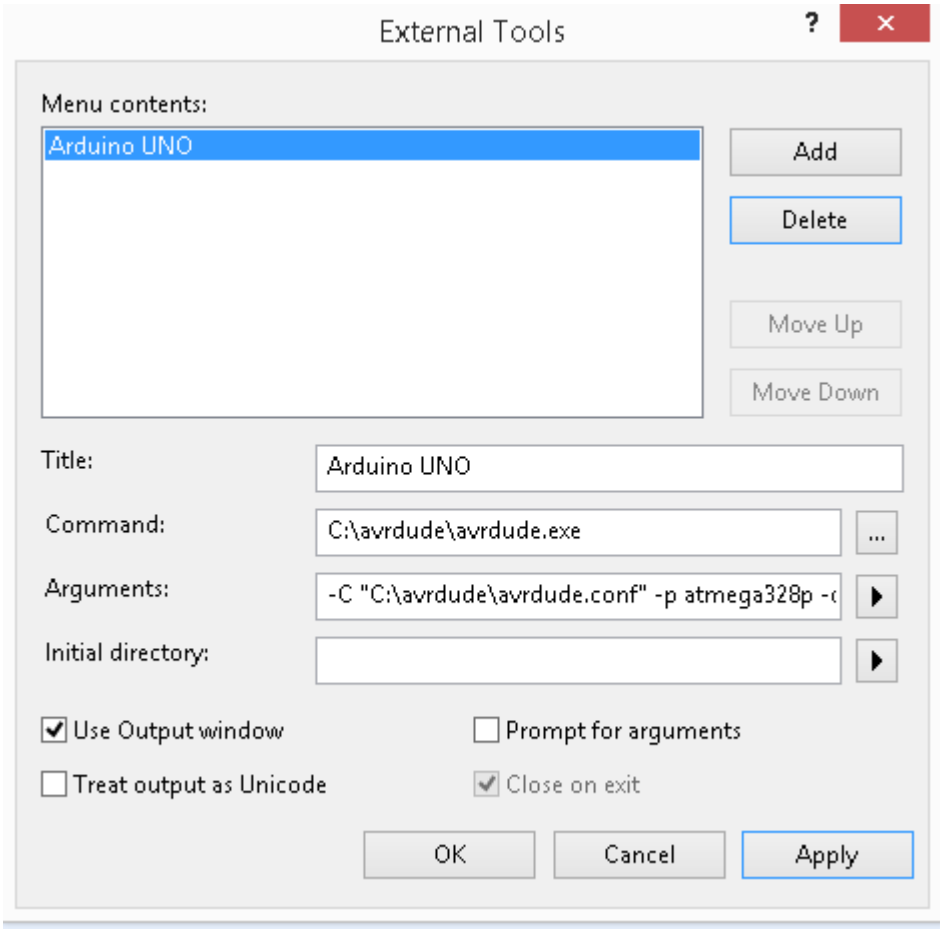
Arduino UNO kartı Atmel Studio ile doğrudan kullanılamaz. Bunun için Atmel Studio'nun bir özelliği olan **Tool** (Alet) olarak tanımlanması gerekir. Fakat ondan da önce AVRDUDE programını yüklememiz gerekir. Bundan öncesinde AVRDUDE programından bahsetmiştik. Programı aşağıdaki bağlantıdan indiriyoruz.

http://download.savannah.gnu.org/releases/avrdude/avrdude-5.11-Patch7610-win32.zip

C sürücüsünde AVRDUDE adında bir klasör açıp indirdiğimiz arşiv dosyasını oraya çıkardıktan sonra Atmel Studio'yu açıyoruz. **Tools** kısmından **External Tools** sekmesini seçiyoruz.



Burada açılan pencerede ADD düğmesine tıklayıp yeni bir alet oluşturuyoruz. Bilgiler resimdeki gibi girilmelidir.



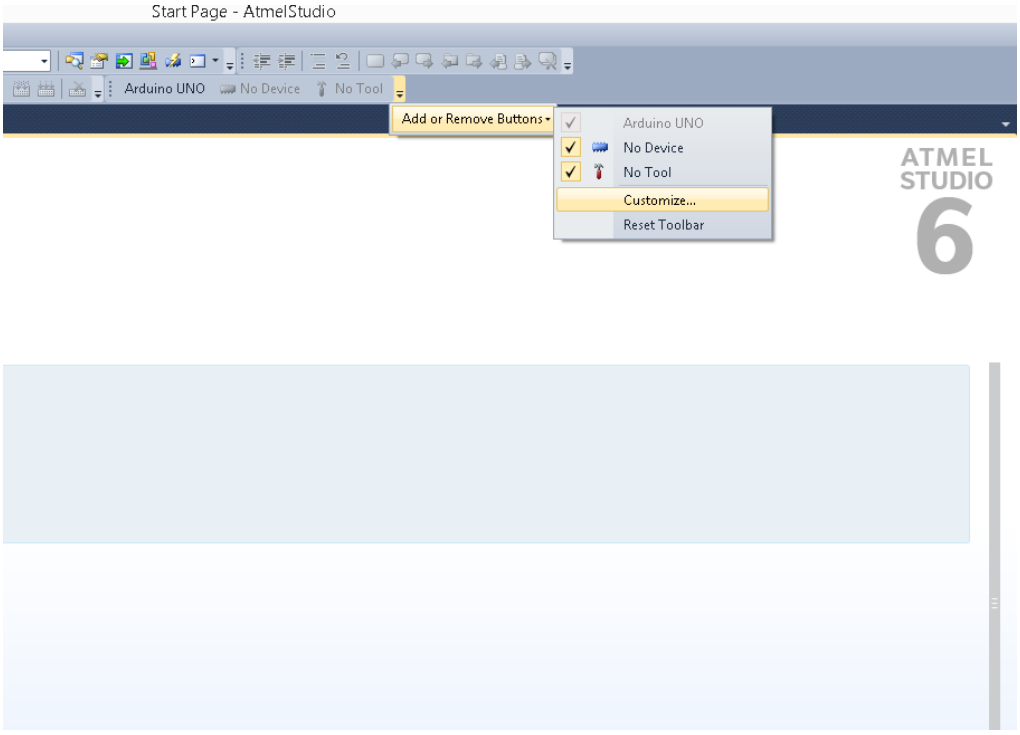
Title	Arduino UNO (veya istediğiniz bir başlık)
Command	C:\avrdude\avrdude.exe
Arguments	-C "C:\avrdude\avrdude.conf" -p atmega328p -c arduino -P COM3 -b 115200 -U flash:w:"$(ProjectDir)Debug\$(ItemFileName).hex":i

Use Output Window kutusunu işaretli değilse işaretliyoruz.
COM9 yerine Aygıt Yöneticisinden baktığımız COM değerini yazıyoruz.
Atmel Studio 7'de ItemFileName yerine TargetName yazınız.

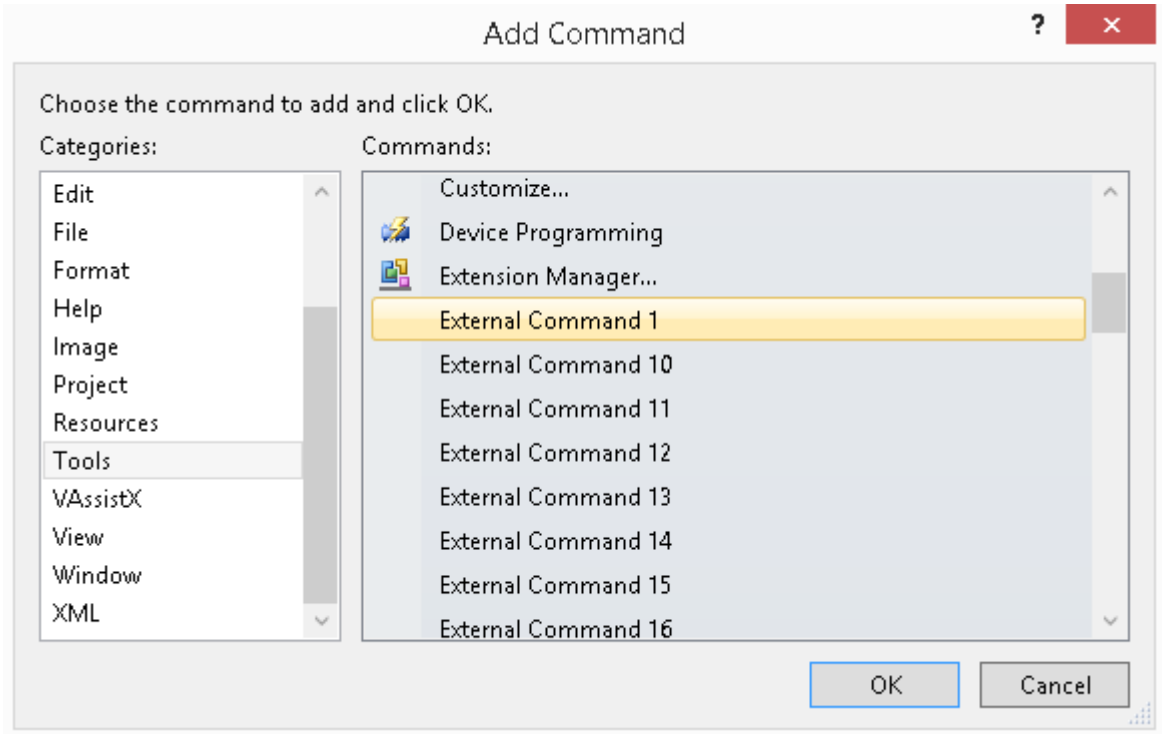
Doğrudan Copy-Paste yapmayın. " işaretleri düzgün çıkmayabiliyor o yüzden hata verebilir. Elle yazmayı deneyin.

Özellikle Arguments kısmındaki komutun doğru olarak yazıldığından emin olun. AVRDUDE programına gönderilecek bilgiler burada yer alır ve Arduino UNO'ya göre düzenlenmiştir. Eğer farklı bir kart ya da entegre kullanmayı istiyorsanız argümanların değerlerini değiştirerek programı işletebilirsiniz. Burada Atmel Studio AVRDUDE programı üzerinden Arduino kartına debug klasöründeki .hex dosyasını atacaktır.

Bu işlem ile beraber Atmel Studioda derleyeceğimiz program doğrudan Arduino UNO kartına atılacaktır. Ama bunun kısayolunu oluşturup ekrandaki bir düğmeye basarak işi kolaylaştıralım. Öncelikle şekildeki gibi bir düğme oluşturmak için sağ taraftaki oka tıklayıp Add or Remove Buttons sekmesinden Customize düğmesine tıklıyoruz.

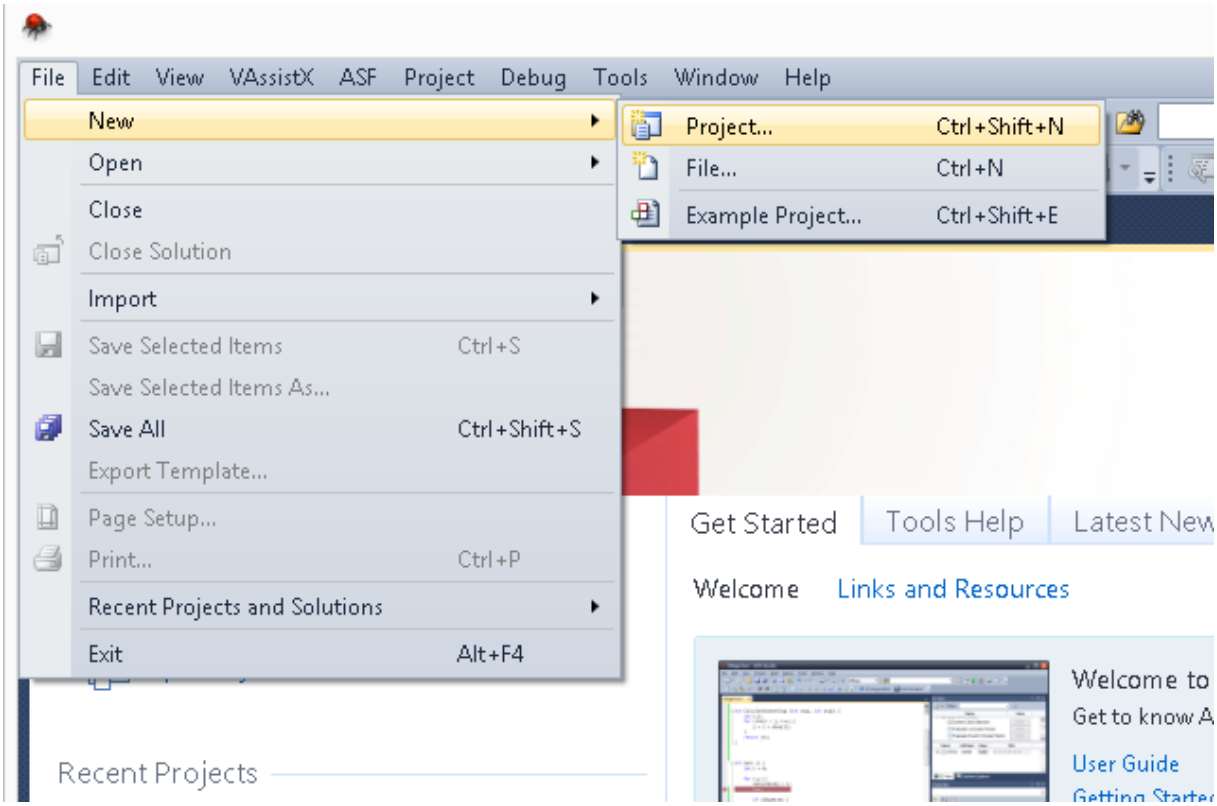


Açılan pencerede **Add Command…** düğmesine tıklıyoruz ve Tools sekmesinden External Command 1'i seçiyoruz.

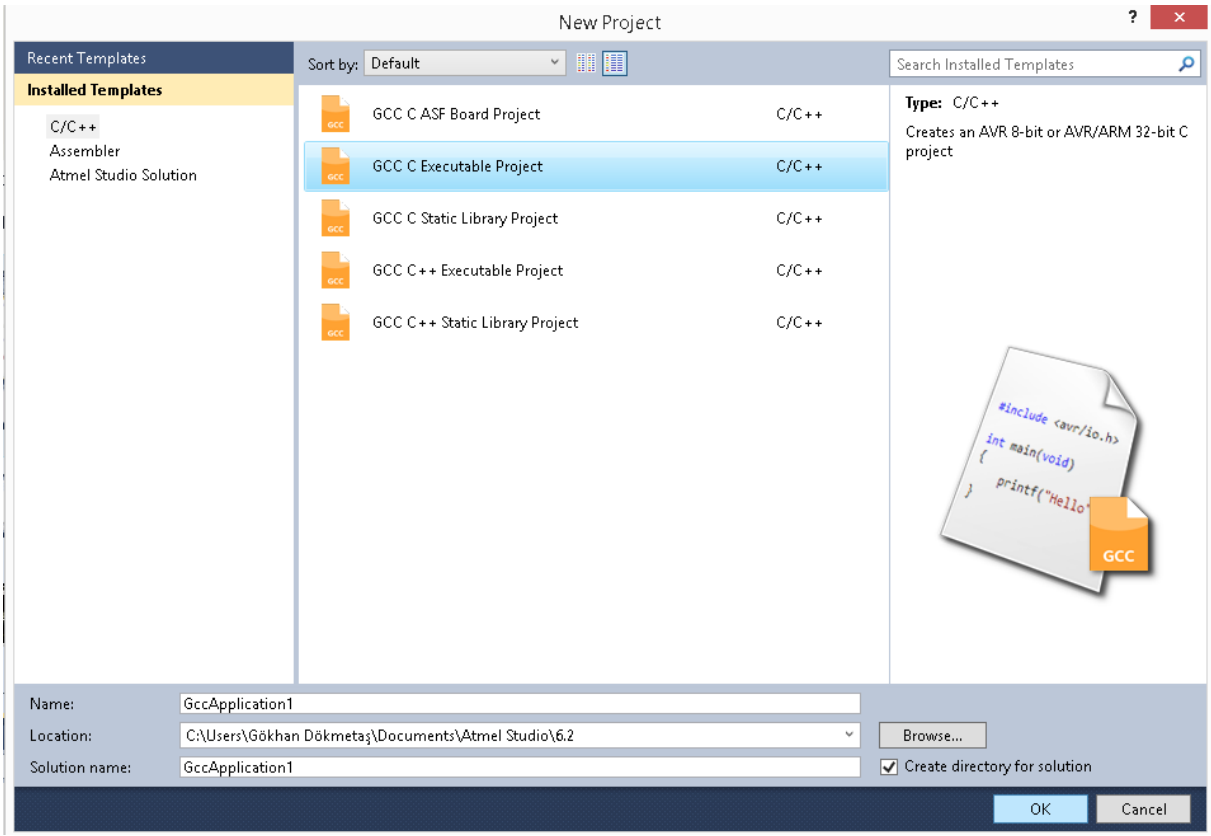


Böylelikle şimdilik Atmel Studio'da yapacağımız ek bir şey kalmamış oluyor. Şimdi yeni proje açalım ve ilk programı yazıp çalıştıralım.

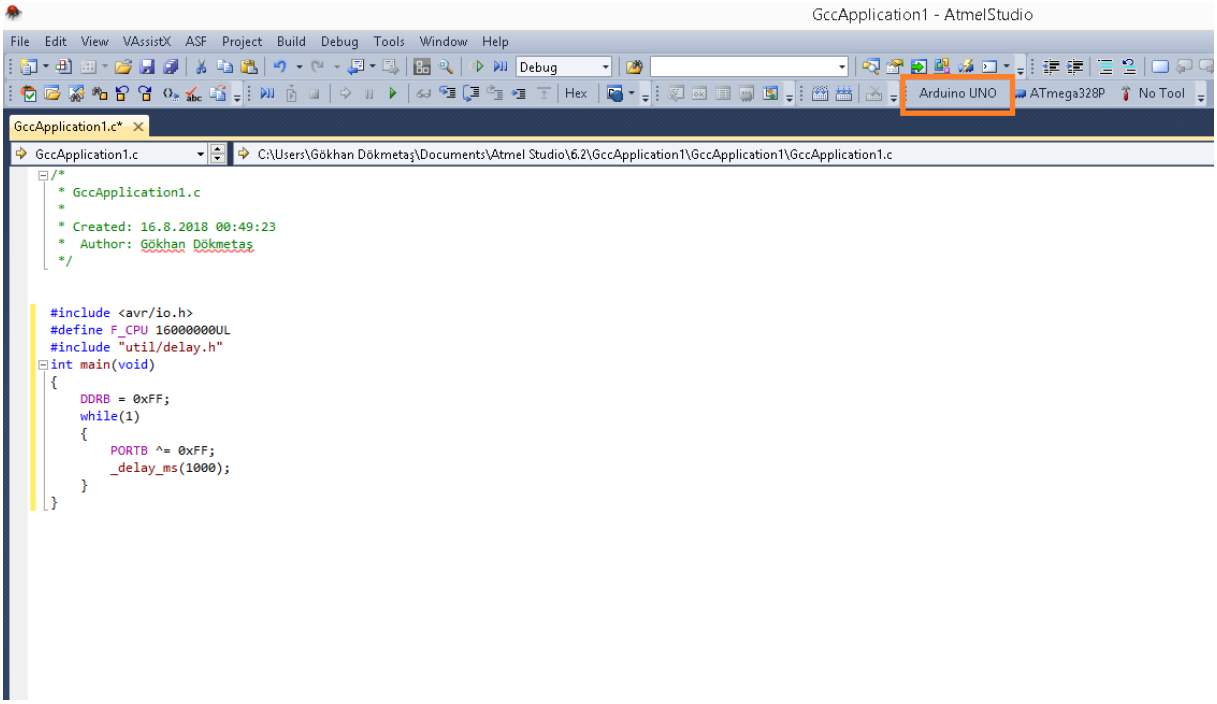
Öncelikle File/New/ New Project diyoruz.



Karşımıza iki dil ve o dillerdeki proje tipleri çıkıyor. Assembly ve C/C++ dilleri ayrı olduğu gibi C ve C++ projeleri de ayrı olarak açılmaktadır. GCC C Executable Project seçeneğini seçiyoruz ve projemize isim verdikten sonra projeyi açıyoruz.



Proje ekranından sonra karşımıza aygıt seçme ekranı geliyor. Atmega328P kullanacağımız için arama kısmına Atmega328P yazıyoruz. Ayrıca aygıtı geçtikten sonra teknik veri sayfası (datasheet) ve desteklenen araç listesi çıkıyor. Buranın ekran görüntüsünü verme gereği duymadım çünkü yukarıdaki ekran görüntüleri yeteri kadar fazla oldu. Şimdi karşımıza programı yazacağımız .c uzantılı dosya geliyor ve programı yüklemek için yukarıdaki Arduino UNO düğmesine tıklıyoruz.



1. Uygulama : LED Yakma

İster gömülü sistemlere sıfırdan başlayın isterseniz de yeni bir donanıma geçin ilk uygulama olarak giriş ve çıkış portuna bir LED bağlayıp yakmanız gereklidir. 😊 Gerek 8-bit basit denetleyicileri öğrenirken gerekse 32-bit gelişmiş denetleyicilerde uygulama yaparken önce bir LED yakma uygulamasını başarı ile gerçekleştirip sonrasında bunun üzerine ilave ede ede pratiği öğrenirsiniz. Kodu inceleyerek uygulamaya devam edelim.

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

int main(void)
{
     DDRD = 0xFF; // D portunun bütün ayakları çıkış
      PORTD = 0xFF; // PORTD AÇIK

    while (1)
    {

    }
}
```

Burada mikrodenetleyicinin D portunun herhangi bir ayağına bağlayacağınız LED yanacaktır. D portu ise Arduino UNO kartının 0'dan 7'ye kadar olan dijital ayaklarına karşılık gelmektedir. Hangi kartın hangi ayağa karşılık geldiğini bulmak için şu şemayı kullanmanızı tavsiye ederim. Bunu yazıcıdan çıkarıp göreceğiniz bir yere asmanız iyi olacaktır.

https://forum.arduino.cc/index.php?action=dlattach;topic=146315.0;attach=90365

Bu uygulamayı yapmak değil anlamak daha önemli olduğu için kodları satır satır inceleyip açıklayalım. Diğer uygulamalarda da kodları satır satır anlayıp açıklayabilir duruma gelmeden bir sonraki uygulamaya geçmemek gerekli. Şimdi bu çok kısa olan programı açıklayalım.

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

Burada derleyicide bulunan kütüphane dosyalarından *io.h* adlı dosyayı programa dahil ediyoruz. Bu dosya aslında stüdyoda seçtiğimiz mikrodenetleyiciye göre ilgili mikrodenetleyici başlık dosyasını programa dahil etmekte. Bu başlık dosyasında

ise ilgili mikrodenetleyicinin yazmaçlarının adresleri belli tanımlamalarla ilişkilendirilmiştir. Mesela programda DDRD yazdığımızda aslında başlık dosyasında belirtilen özel fonksiyon yazmacının adresini bu sayede yazmış oluyoruz. Bunu *iom328p.h* dosyasında açıkça görebilirsiniz. Örneğin DDRD tanımlaması şu şekilde yer almaktadır.

```
#define DDRD _SFR_IO8(0x0A)
```

Yalnız burada DDRD yazmacının adresinin mikrodenetleyici hafızasındaki 0x0A adresi olduğunu düşünmeyin. Ne kadar özel fonksiyon yazmaçlarının adresi böyle belirtilse de datasheette de göreceğiniz üzere bunun üzerine 0x20 değeri eklenmelidir. Burada yer alan *_SFR_IO8()* makrosunun görevi de aslında bu offset değerini eklemekten ibarettir. Bu makroyu tanımlandığı *sfr_defs.h* dosyasında şu şekilde görmekteyiz.

```
#define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)
```

Görüldüğü gibi yukarıdaki makro *__SFR_OFFSET* adlı değeri adres değerinin üzerine ilave etmekten başka bir iş yapmıyor. Bu offset değerini ve bunun mimari farkı ve buna uyum için kullanıldığını yine ilgili dosyanın şu kısmından anlayabiliriz.

```
#  if __AVR_ARCH__ >= 100
#    define __SFR_OFFSET 0x00
#  else
#    define __SFR_OFFSET 0x20
#  endif
#endif
```

Uygulama esnasında derleyicinin kütüphane dosyalarını incelemek konuyu anlama noktasında oldukça fayda sağlayacaktır. Kendi uygulamalarınızı yaparken de datasheet okumanın yanında derleyici kılavuzunu okumak ve bu kütüphane dosyalarını kullanmanız gerekecektir.

```
DDRD = 0xFF;
```

Burada DDRD (*PORT D Data Direction Register*) yazmacına onaltılık tabanda 0xFF yani ikilik tabanda 0B11111111 değerini atıyoruz. Bunu neden atadığımızın cevabını bulmak için datasheete bakmamız gerekecektir. Datasheette yazan bilgiye göre temel giriş ve çıkış portlarına ait PORTx, PINx ve DDRx olmak üzere (x burada portun adı) üç yazmaç var ve DDRx yazmacı burada portun ilgili ayaklarının giriş mi çıkış mı olduğunu tanımlamamızı sağlamakta.

Mikrodenetleyici RESET aşamasından sonra bu portların değerleri 0 olmakta. DDRx portunda ilgili ayağa karşılık gelen bit 0 ise giriş, 1 ise çıkış olarak tanımlanıyor. Bizim portun ayaklarından dijital olarak 1 veya 0 çıkışı almak için DDRx portunun değerini 0xFF yapmamız gerekecektir. Resetten itibaren ayakları giriş olarak kullanmak istediğimizde DDRx portuna herhangi bir değer atamak gerekmez. Bunu programda ayrıca yapmanız tercih meselesidir.

```
PORTD = 0xFF;
```

DDRD'nin değerini 0xFF yaparak bütün ayakları çıkış olarak tanımlasak da yukarıda dediğimiz gibi bu portların değeri açılışta sıfır olduğu için PORTD'nin bütün bitleri sıfır olacaktır. Bu durumda dijital 0 olarak akım çeken bir konumda (*Sink*) olacaktır. Bu ayakların üzerinden LED'lere akımın akması için dijital olarak HIGH yani bir (1) yapılmalıdır. Bütün ayakları HIGH yapmak için de PORTD yazmacına 0xFF değeri verilmiştir. Bu uygulama gibi diğer uygulamaları iyi anlayabilmek için bir yandan da datasheet üzerinden ilgili birimleri ve yazmaç açıklamalarını takip etmeniz iyi olur. Aşağıdaki bağlantıdan datasheeti indirebilirsiniz.

https://www.microchip.com/wwwproducts/en/ATmega328p#datasheet-toggle

2. Uygulama : LED Yakıp Söndürme

Bu uygulamada basit bir LED yakıp söndürme (*Blink*) uygulaması yapacağız. Bu uygulamanın önemli noktası "*Delay*" yani geciktirme fonksiyonlarının kullanımıdır. LED yakma uygulamasında olduğu gibi D portunun bütün ayaklarına bağlı LED'ler sırayla yanıp sönecektir. Bu yüzden bir önceki uygulamayı yaparken kurduğunuz devreye hiç dokunmadan bu uygulamayı gerçekleştirebilirsiniz.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
int main(void)
{
    DDRD = 0xFF; // PORTD ÇIKIŞ
    while (1)
    {
            PORTD = 0xFF;
            _delay_ms(1000);
            PORTD = 0x00;
            _delay_ms(1000);
```

```
    }
}
```

Burada "Delay" kütüphanesi ve onunla alakalı bazı fonksiyon ve tanımları inceleyelim. Delay işlemi işin aslında mikrodenetleyiciyi gereksiz kodları belli bir süre çalıştırmak suretiyle meşgul etmekten ibarettir. Mikrodenetleyici "*Dur*" demekle durmamaktadır ve uyku modu haricinde sürekli çalışmaya devam etmektedir. Biz belli bir süre bir sonraki kodu işlemesini istemediğimiz zaman mikrodenetleyiciyi durdurmak yerine belli bir işle meşgul ederiz.

```
#define F_CPU 16000000UL
```

Burada öncelikle F_CPU adındaki tanımın değerini belirlemekteyiz. F_CPU tanımlaması AVR mikrodenetleyicinin işlemci frekansının değerini belirtmek için kullanılır. Uygulamada kaç MHz'lik kristal kullanıyorsanız veya saat kaynağı ile besliyorsanız o saat değerini elle belirlemeniz gereklidir. Sadece *delay* kütüphanesi değil, pek çok harici kütüphane de F_CPU değerini referans almaktadır. Bazı kütüphanelerde F_CPU değerini belirlemezseniz standart bir değer kullanılmaktadır. Bu pek çok zaman uygulamada kullandığınız denetleyicinin saat frekansı ile ölçüşmez ve aksaklıklara sebep olur. Bu yüzden her uygulamada F_CPU değerini belirleme alışkanlığı edinmekle işinizi sağlama almış olursunuz. Bu uygulamada ise *delay* kütüphanesi F_CPU değeri belirlenmezse derleme esnasında uyarı verdirecektir.

```
#include <util/delay.h>
```

Burada derleyicinin yerleşik kütüphane dosyalarından biri olan *delay.h* dosyasını programa dahil ediyoruz. Bu dosyayı açtığımızda *_delay_ms()* ve *_delay_us()* olarak iki fonksiyon tanımını görmekteyiz. Bunlar milisaniye ve mikro saniye bazında bekleme yapan fonksiyonlardır. Bu fonksiyonlar ise F_CPU değerini esas alarak *delay_basic.h* dosyasında yer alan AVR Assembly ile yazılmış bekleme döngülerini çalıştırmaktadır. Örnek bir bekleme döngüsünü aşağıda görebilirsiniz.

```
void
_delay_loop_1(uint8_t __count)
{
	__asm__ volatile (
		"1: dec %0" "\n\t"
		"brne 1b"
		: "=r" (__count)
		: "0" (__count)
	);
}
```

Inline Assembly adı verilen teknikle siz de C dosyaları içerisine Assembly kodlarını sıkıştırabilirsiniz. Yukarıdaki örnekte bu tekniği görebilirsiniz. Bu derleyicide yerleşik olan *delay* kütüphanesinin oldukça isabetli çalıştığını söyleyebilirim. Yalnız *delay* fonksiyonları içerisine herhangi bir değişken yazmanız mümkün değildir. Sadece sabit değer kabul ettiğinden değişken değerlerde beklemeyi kendi *delay* fonksiyonunuzu yazarak veya bu fonksiyonları döngü içerisinde kullanarak yapabilirsiniz. Örneğin 300 milisaniye bekleme için *_delay_ms(1)* fonksiyonunu 300 kere çalıştıran bir fonksiyon yazabilirsiniz. Yalnız burada başka kodlar da işin içine girdiği için doğruluk noktasında dezavantaj yaşayabilirsiniz.

```
_delay_ms(1000);
```

Burada PORTD yazmacına 0xFF değeri verildikten sonra mikrodenetleyici 1 saniye meşgul edilmektedir. Bu fonksiyonlar *double* tipinde değer aldığı için parantezler arasında *double* tipinde bir değer yazabilirsiniz.
Delay kullanımı çok fazla tavsiye edilmese de bazı noktalarda *delay* fonksiyonlarını kullanmanız zorunlu olabilir. Mesela DHT serisi algılayıcılarla iletişime geçerken bir veriyi yolladıktan sonra belli bir süre beklemek ve o süreden sonra okuma yapmak gereklidir. Bu durumda başka bir kodun çalışmasına izin vermeden beklemek gereklidir. Çünkü zamanlamada az bir hata bile iletişimin sağlığını olumsuz etkilemektedir. Bunun dışında ana program akışına bol bol *delay* yerleştirmek belli bir süre sonra aşırı derecede yavaşlığa sebep olacaktır. Biz burada sadece LED yakıp söndürmekten ibaret bir cihaz yaptığımız için *delay* kullanmakta bir sakınca görmedik.

3. Uygulama : Trafik Işığı

Bu uygulama basit bir LED yakıp söndürme uygulaması olmasından ziyade mikrodenetleyicinin ayaklarını nasıl teker teker nasıl bir (1) ve sıfır (0) yapacağımızı öğretmeye yöneliktir. Bunun gibi ileride göreceğiniz bazı uygulamalar da birbirine benzer gibi görünse de her biri farklı bir noktayı öğretmeyi amaçlamaktadır. Açıklamalarda da bu farklı noktalara odaklanacağım. Bu devrede D portunun 2, 3 ve 4 numaralı ayaklarına bağlanan kırmızı, sarı ve yeşil LED'ler ile bir trafik ışığı benzetimi yapılmıştır. Devreyi kurarken Arduino UNO kartının dijital 2, 3 ve 4 ayaklarını kullanacaksınız.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

int main(void)
{
```

```
    DDRD = 0xFF;
    while (1)
    {
      PORTD |= (1<<2); // Birinci Led (Kırmızı)
      _delay_ms(1000);
      PORTD &=~(1<<2); // Led söndür
      PORTD |= (1<<3); // İkinci led
      _delay_ms(600);
      PORTD &=~(1<<3);
      PORTD |=(1<<4);
      _delay_ms(2000);
      PORTD &=~(1<<4);
    }
}
```

Burada artık PORTD yazmacına doğrudan değer atanmadığını ve onun yerine C dilindeki *bitwise* operatörler kullanılarak tek bit üzerinde işlem yapıldığını görüyoruz. Aslında öğrendikten sonra belli başlı işler için belli başlı operatörlerin sırayla kullanıldığını görürsünüz. Yani her komutta tek tek bu sembolleri anlamakla uğraşmazsınız. Mesela |= koyulan yerde HIGH yapılmıştır veya & koyulan yerde bit denetlenmiştir diyebilir ya da &=~işaretini gördüğünüz zaman o bitin sıfıra çekildiğini hemen anlayabilirsiniz. Mesela (1<<4) ifadesini de 4. Bit olarak okumanız gereklidir. Bu semboller üzerine kafa yorarak meseleyi anlamanız gerekse de anladıktan sonra artık sadece göz ila takip ederek ne olduğunu bilebilirsiniz. *Bitwise* operatörlerin çalışma mantığını hem C ile AVR programlama yazılarımda hem de daha ayrıntılı olarak Temel C Programlama yazılarımda yazdım.

4. Uygulama : Düğmeye Basınca Yanan LED

Bu uygulama artık çıkış olarak kullanımını öğrendiğimiz portların nasıl giriş olarak kullanılacağını göstermektedir. Portları DDRx, PORTx ve PINx yazmaçları üzerinden kullanacağınızı unutmayın. İşte burada PINx yazmacı giriş modunda porta uygulanan dijital sinyali okumamızı sağlamaktadır. Çıkış modunda ise oldukça garip olarak "*Toggle*" özelliğine sahiptir. Yani bir bit 0 ise bunu 1 yapar, 1 ise 0 yapar. Doğrudan PINx yazmacına bu değeri yazarak *toggle* yani aç kapa yapmamız performans açısından büyük bir kazanımdır. Ama biz yine C dilinde yine *bitwise* operatörlerini kullanarak bunu yaparız. Derleyici bunu anlattığımız şekilde optimize ederek Assembly diline çevirir.

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

```
int main(void)
{
      DDRD |= (1<<4); // PD4 ÇIKIŞ
      DDRD &=~(1<<3); // PD3 GİRİŞ
      PORTD |= (1<<3); //PD3 PULL UP
    while (1)
    {
        if(!(PIND & (1<<3)))
        PORTD |= (1<<4);
        else
        PORTD &=~(1<<4);
    }
}
```

Burada sadece dijital port okumayı değil dahili *pull-up* dirençlerini kullanmayı da görebilirsiniz.

```
PORTD |= (1<<3);
```

Bu komut normalde dijital 1 çıkışı almamıza yaramaktadır. Ama DDR yazmacı üzerinden ayak ayarı giriş olarak tanımlanırsa işler değişmektedir. Kullandığımız AVR mikrodenetleyicinin bütün portlarında dahili *pull-up* direnci bulunmakta ve giriş olarak kullanıldığı zamanlar bunlar PORTx yazmacının ilgili bitlerine yazılan 1 değeri ile etkinleştirilmektedir. Düğme, anahtar gibi giriş aygıtı bağlarken pek çok zaman *pull-up* direncine ihtiyaç duyacağımız için bu işimizi oldukça kolaylaştıracaktır. Aynı zamanda I2C gibi *pull-up*'ın neredeyse zorunlu olduğu protokollerde bu dirençleri kullanabiliriz.

```
if(!(PIND & (1<<3)))
```

Bu karar yapısında "*PIND yazmacının 3. Biti 1 değil ise*" şartını koşmuş olduk. Normalde *bitwise* DEĞİL operatörü olmadan da bu karar yapısını tersine yazabilsek de ben bu tarz yazmayı alışkanlık edindim. Bir iş gerçekleştiğinde if yapısının ilk kısmının işletildiğini düşünerek kodu okuyup yazmaktayım.

5. Uygulama : Düğmeler ile Yürüyen Işık

Bu uygulamada B portunun 0 ve 1 numaralı ayaklarına iki *tactile* düğme bağladım ve D portunun bütün ayaklarına birer kırmızı LED koydum. Yani toplamda 8 LED'i sıra ile bağlamamız gerekiyor. Uygulama başladığı zaman her düğmeye basışımızda bir sağdaki veya soldaki LED yanıp önceki LED sönerek birer adım atlanmış oluyor. Yalnız en kenarlara geldiğimizde bir daha basarsak artık o bit taşmış ve ortadan kaybolmuş oluyor. Bu uygulamayı geliştirmek için bunu önleyen bir karar yapısı koyabilirsiniz. Ben buna gerek görmedim.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{

	DDRD = 0xFF;
	DDRB &=~ ((1<<0) | (1<<1));
	PORTB |= ((1<<0) | (1<<1));
	PORTD |= (1<<PD0);

	while (1)
	{
		if(bit_is_clear(PINB,0))
		{
			PORTD<<=1;
			_delay_ms(250);
		}


		if(bit_is_clear(PINB,1))
		{
			PORTD>>=1;
			_delay_ms(250);
		}
	}
}
```

Burada bit kaydırma operatörlerinin uygulaması yanı sıra derleyici içerisinde yer alan *bit_is_clear()* makrosunun kullanımını görmekteyiz. Evet, isterseniz *bitwise* operatörlerini kullanmadan böyle makrolarla giriş ve çıkış işlemi yapabilirsiniz. Ben şahsen operatörleri kullanmaya alıştığım için makrolar veya hazır fonksiyonlar kadar okunaklı gelmekte. Bazıları Arduino'nun *digitalRead()* veya *digitalWrite()* gibi fonksiyonlarına alıştığı için operatörleri kullanırken zorluk çekebilir. Bu makroları kullanmak tercih meselesi olduğu için gösterme adına

uygulamaya ekledim. Bu makroların tamamını ve iç yapısını *sfr_defs.h* dosyasından görmekteyiz. Burada kaynak kodu ile beraber açıklamasını yapalım.

```
#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
```

Bu makro PORTx, PINx gibi özel fonksiyon yazmaçlarında yer alan bir bitin bir olup olmadığını denetler. Örneğin *bit_is_set(PIND, 0);* dediğimizde PIND yazmacının 0 numaralı biti bir ise 1 değerini geri döndürür.

```
#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
```

Bu makro ise özel fonksiyon yazmaçlarındaki bir bitin sıfır olup olmadığını denetler. Eğer sıfır ise bir değerini geri döndürür. Yani burada *bitwise* operatörlerini yazarak aynı işi yapmak yerine hazır fonksiyon kullanır gibi makroları kullanıyoruz. Arka planda yine bu iş *bitwise* operatörleri ile yapılmakta. Makronun açıklamasındaki & operatörünü fark ettiniz umarım. Bir de makro içinde *_BV()* makrosu yer almakta. Bu makronun ne işe yaradığını da yine aynı dosyadan öğreniyoruz.

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

Bu da bizim bitwise operatörleriyle yaptığımız (örneğin (1<<5)) işi makro ile yapmamızı sağlamakta. İsterseniz *_BV()* makrosu dahil olmak üzere bunları değer okumakta kullanabilirsiniz. Bunun yanında bir biti 1'e veya 0'a çekmek için de makrolar mevcuttur.

```
#define bitset(byte,nbit)   ((byte) |=  (1<<(nbit)))
#define bitclear(byte,nbit) ((byte) &= ~(1<<(nbit)))
#define bitflip(byte,nbit)  ((byte) ^=  (1<<(nbit)))
```

İlk makro istenilen baytın (burada yazmacın) belirtilen bitini bir yapar, ikinci makro ise 0 yapar, üçüncüsü ise *toggle* yani aç/kapa işlemi yapmaktadır. Mesela D portunun 0 numaralı ayağını bir (1) yapmak istediğimizde *bitset(PORTD,0);* şeklinde komut yazabiliriz. Bu makrolar performansta herhangi bir kayıp yaşatmaz.

sfr_defs.h dosyasında iki makro daha gözüme çarptı. Açıkçası bunlara daha önce dikkat etmemiştim ama uygulamada bazen bir bitin 1 veya 0 olmasını beklememiz gerekebiliyor. Bu iki makro böyle durumlarda kullanılabilir.

```
#define loop_until_bit_is_set(sfr, bit) do { } while
(bit_is_clear(sfr, bit))
```

Burada bir bit bir seviyesine gelene dek işlemci sonsuz döngüye sokulmakta. Örneğin ADC çevirimi tamamlandığında ilgili yazmacın biti 1 olmakta. Bu kontrolü de sonsuz döngüye sokup sürekli biti denetleyerek yapmaktayız.

```
#define loop_until_bit_is_clear(sfr, bit) do { } while
(bit_is_set(sfr, bit))
```

Bu makro da ilgili yazmacın biti 0 olana dek programı sonsuz döngüye sokmaktadır. Bu saydığım makroları kullanmak için herhangi bir kütüphaneye ihtiyacınız olmasa da *bitset(), bitclear()* ve *bitflip()* makrolarını kendiniz başta tanımlamanız gereklidir.

6. Uygulama : Karaşimşek

Amatör elektronik kitapları başta olmak üzere uzun yıllar boyunca pek çok elektronik kitabında yer alan bu devre ismini eski bir dizi olan Knight Rider'ın Türkçe karşılığından almıştır. Elektronik eğitiminde bir klasik olduğu için ben de her donanımı anlatırken bu uygulamayı yapmaya özen göstermekteyim.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{

DDRD = 0xFF;
while (1) {

      for (PORTD = 0x01; PORTD != 0; PORTD <<= 1)
      _delay_ms(50);

      for (PORTD = 0x80; PORTD != 0; PORTD >>= 1)
      _delay_ms(50);
}
}
```

Bu kod Arduino ile yapılan karaşimşek uygulamalarına göre biraz zor göründü değil mi? Gerçekten de okunabilirliği düşük ama verimliliği yüksek olan bir karaşimşek uygulamasıdır. D portuna bağlı olan 8 adet LED sırayla yakılırken bütün yakıp söndürme işlemleri for döngüsünün içerisinde gerçekleştirilmektedir.

for döngüsünde döngü değişkeni normalde "i" adında sadece döngüde kullanılan bir değişken olarak tanımlansa da burada PORTD'yi döngü değişkeni olarak kullandık. PORTD'nin döngü değişkeni olması bile belki kafanızı karıştırmaya yetecektir. Ama bunun nasıl güzel bir kod olduğunu açıklayalım.

```
for (PORTD = 0x01; PORTD != 0; PORTD <<= 1)
      _delay_ms(50);
```

Burada öncelikle PORTD değişkenine (yani yazmacına) 0x01 değeri veriliyor. Bu durumda PORTD'nin 0 numaralı biti (Her zaman ayaklar 0'dan başlar.) 1 yapılmış ve ilk LED'imiz yanmış oluyor. Sonrasında PORTD != 0 diye döngü şartını görmekteyiz. Yani PORTD sıfır olmadığı sürece bunu yapmaya devam et demektedir. Daha öncesinde düğmeler ile yürüyen ışık uygulamasında ışığı kenarlardan taşırdığımızda kaybolduğunu fark etmişsinizdir. İşte ışık bir sonraki kenara ulaşıp taştığında yazmacın değeri sıfır olmaktadır. Bu durumda da döngü bitecek ve bir sonraki döngü yani sağdan sola yürüyen ışık döngüsü çalışacaktır.

PORTD <<= 1 kısmında ise sağ kısımda (*Less significant bit* ya da LSB) bulunan 1 değeri birer birer sola kaydırılmaktadır ve bu kaydırılma sürecinde *_delay_ms(50)* fonksiyonu ile gecikme sağlanmakta ve bu sayede insan gözü fark edebilmektedir.

Görüldüğü gibi oldukça kompakt bir kod karşımıza çıkmakta. Bir sonraki döngü ise aynı şekilde ama ters yönde bu işlemi gerçekleştirmekte ve ana program döngüsü bu şekilde devam etmektedir. Bu kadar kompakt bir kodun karşısında size Arduino ile yapılmış bir karaşimşek uygulamasını göstereyim. Böylelikle ikisi arasında nasıl bir fark olduğunu rahatça görebilirsiniz.

```
digitalWrite(pin2, HIGH);
delay(timer);
digitalWrite(pin2, LOW);
delay(timer);
digitalWrite(pin3, HIGH);
delay(timer);
digitalWrite(pin3, LOW);
delay(timer);
digitalWrite(pin4, HIGH);
delay(timer);
digitalWrite(pin4, LOW);
delay(timer);
digitalWrite(pin5, HIGH);
delay(timer);
digitalWrite(pin5, LOW);
delay(timer);
```

```
digitalWrite(pin6, HIGH);
delay(timer);
digitalWrite(pin6, LOW);
delay(timer);
digitalWrite(pin7, HIGH);
delay(timer);
```

Kaynak: https://www.arduino.cc/en/Tutorial/KnightRider

Bu kod parçasını Arduino'nun resmî sitesinde yer alan KnightRider uygulamasından aldım. Kodun tamamını kalabalık olmasın diye almıyorum. Bu kod parçası anlamanız için yeterlidir. Görüldüğü gibi bizim iki satırda hallettiğimiz işi burada tek tek bitleri bir ve sıfır yaparak ve gecikmeyi tekrar tekrar yazarak yapmaktalar. Bir tarafta iki satır kodla yapılan iş varken öteki tarafta aynı işi satırlarca kod yazarak yapıyorsunuz.

7. Uygulama : Kolay Karaşimşek

Yukarıdaki kodu yeni öğrenmeye başlayan birinin yazacağını beklemeyin. Karaşimşek uygulamasının yeni öğrenenlerin rahatça anlayacağı ve yazacağı bir sürümü de şu şekilde olabilir. Burada performans yerine okunabilirliğe önem verilmiştir.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{

	DDRD = 0xFF;
	int bekleme = 150;
	while (1)
	{

		for(int i = 0; i < 7; i++)
		{
			PORTD |=_BV(i);
			_delay_ms(bekleme);
			PORTD &=~_BV(i);
		}

		for(int i = 7; i > 0; i--)
		{
```

```
                PORTD |=_BV(i);
                _delay_ms(bekleme);
                PORTD &=~_BV(i);
            }
        }
}
```

Her ne kadar yukarıdaki kod gibi kompakt olmasa da C dilini ve AVR programlamayı yeni öğrenmeye başlayan birisi bu kodu okuyup rahatça çözebilir. Burada *_BV()* makrosunun nasıl kullanıldığını göstermek adına makroyu özellikle kullandım. Yukarıdaki for döngüsü yerine her öğrencinin bildiği ve kitaplardaki kullanılan şekliyle bir i değişkeni tanımlayıp döngü içerisine sadece döngüye özel değişkeni dahil ettik. Programda ise teker teker bitler yakılıp, beklenip sonra da söndürülmektedir. Bu arada mümkün mertebe açıklama (//) satırı kullanmamaya dikkat ediyorum. Çünkü yeni öğrenen bir öğrenci kodu okumak yerine açıklama satırını okumayı yeğliyor. Bizim kodu okuyup anlayabilmemiz gerek, kimse açıklama satırı ile bunu izah etmek zorunda değil. O yüzden uygulama kodlarını açıklamasız okuyup anlayabilmeniz sizi çok ileriye götürecektir.

8. Uygulama : Binary Sayıcı

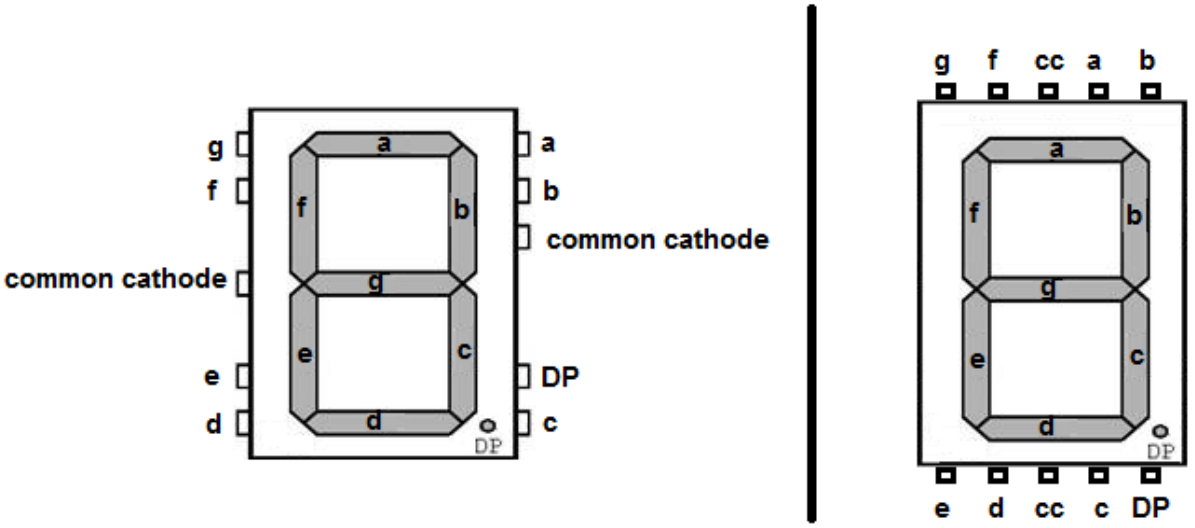
Bu uygulamayı yaparken eski elektronik kitaplarında yer alan "Binary saat" uygulamasından da biraz ilham aldım. Bir ve sıfırları gözünüzle görmeniz ve işlerin nasıl döndüğünü anlamanız açısından bu tarz uygulamalar güzel oluyor. Burada da PORT değişkeni birer birer artırılırken bu porta bağlı LED'ler üzerinden değerleri gerçek bir ve sıfırlar olarak görebilirsiniz.

```
#include <avr/io.h>
#define  F_CPU 16000000UL
#include <util/delay.h>

int main(void)
{
    DDRD = 0xFF;
    PORTD = 0x00;
    while (1)
    {
        PORTD++;
        _delay_ms(100);
    }
}
```

Yukarıda yaptığınız devreyi hiç değiştirmeden yani D portuna sıra ile toplamda 8 LED bağlayarak bu uygulamayı yapabilirsiniz. D portu 100'er milisaniyelik aralıklarla birer birer artmaktadır. En sonunda ise taşar ve tekrar 0 değerine geri dönüp artmaya devam eder.

9. Uygulama : 7 Segman Gösterge

7 segman göstergeler on yıllardır kullanılan ve halen de kullanılmaya devam eden düşük maliyetli ve etkili LED göstergelerdir. Daha öncesinde bunun VFD gibi farklı teknolojiler ile yapılanları olsa da günümüzde LED teknolojisi kullanılmaktadır. Diğer göstergelere baktığınızda gerçekten oldukça uygun fiyatlı olduklarını rahatça görebilirsiniz. Bu uygulamada da basit bir ortak katotlu tekli 7 segman gösterge D portuna bağlanmıştır. Göstergenin ayak haritası şu şekildedir.



Burada sırayla A – 0, B – 1 diye D portuna bağlamak gereklidir. Ortak katot ayağı ise şaseye (GND) bağlanmalıdır. Yine eğitimde sıkça kullanılan bir uygulama olduğu için başlangıçta buna yer verdim. Şahsen bu tarz göstergeleri kullanmak istediğimde sürücü entegreleri olan modülleri kullanmayı tercih ederim. Mikrodenetleyicinin ayaklarıyla tek tek sürmek pek verimli bir seçenek değildir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
```

```
int main() {
	char seg_code[]={0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0xff,0x6F};

	DDRD = 0xff;

	while (1)
	{
		for (int cnt = 0x00; cnt < 9; cnt++)
		{

				PORTD = seg_code[cnt];
				_delay_ms(1000);

		}
	}
}
```

Burada *seg_code[]* adında bir dizi oluşturduk ve her bir rakamı yakacak kodları buraya yerleştirdik. Elbette 0x06'nın 1'e karşılık geldiğini burada programcının ilk bakışta anlaması mümkün değil. Bu değeri yazarken de öncelikle 7 segman göstergenin şemasına bakıp 1 rakamını yakmak için yakmamız gereken segmanları belirliyoruz. Baktığımızda b ve c segmanlarını yakmamız gerektiği anlaşılıyor. Sonrasında devreye baktığımızda b ve c segmanlarına karşılık gelen ayakların D portunun 1. ve 2. Ayakları olduğunu görüyoruz. Bu durumda 0b00000110 değerini vermemiz gerekli. Bunu 1 ve 0'lar halinde yazmak da zahmetli olacağı için on altılık tabana çeviriyoruz ve 0x06 değerini elde ediyoruz. Yukarıda 0x5B, 0x66 gibi değerlerin hepsi işte böyle tek tek elde edilmiştir. Görüldüğü gibi donanımı bilen biri için bu tarz değerler, tanımlamalar, fonksiyonlar hiç anlaşılmaz değildir.

10. Uygulama : RGB LED ile Animasyon

Burada ortak katotlu bir RGB LED kullandım ve bunu D portunun 0, 1 ve 2 numaralı bacaklarına bağladım. Biraz basit kaçan bir uygulama olsa da düşük maliyetli ve verdiği görsel etki ile yeni başlayanları bu mesleği sevdirmede katkıda bulunacaktır. Şahsen ben lisede bu tarz LED yakıp söndürerek bu işi sevmeye başlamıştım. Hatta ilk yaptığım yanıp sönen LED'i bir gün boyunca izlemiştim. 😊

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
int main(void)
{
```

```
    DDRD |= ( (1<<0) | (1<<1) | (1<<2) );

  while(1)
  {
  PORTD |= (1<<0); // KIRMIZI
    _delay_ms(500);
    PORTD &=~ (1<<0); // KIRMIZI
    PORTD |= (1<<1); // YEŞİL
    _delay_ms(500);
    PORTD &=~ (1<<1); // KIRMIZI
    PORTD |= (1<<2); // MAVİ
    _delay_ms(500);
    PORTD &=~ (1<<2);
    PORTD = 0;

    PORTD |= (1<<2);
    PORTD |= (1<<1);
    _delay_ms(1000);
    PORTD = 0;
    PORTD |= (1<<2);
    PORTD |= (1<<0);
    _delay_ms(1000);
    PORTD = 0;
    PORTD |= (1<<2);
    PORTD |= (1<<0);
    PORTD |= (1<<1);
    _delay_ms(2000);
    PORTD = 0;
  }
}
```

11.Uygulama : ADC ile değişken yanıp sönen LED uygulaması

Artık dijital giriş ve çıkış uygulamalarını bitirdik ve mikrodenetleyicinin diğer çevre birimlerini kullanacağız. Sadece LED yakıp söndürme bilgisiyle en fazla röle aç-kapa uygulamaları yapabilirsiniz. Daha nitelikli işleri yapabilmek için öncelikle mikrodenetleyicinin çevre birimlerini etkin bir şekilde kullanabilmek lazımdır. Daha sonrasında işinize göre yüzlerce entegre, sensör ve modül karşınıza çıkacaktır.

Bu uygulamada C portunun 0 numaralı yani Arduino'daki A0 ayağına bir potansiyometreyi gerilim bölücü direnç olarak bağladım. Yani bir ayağı VCC öteki GND orta ayak ise analog girişe bağlı. Sonrasında ise D portunun 2 numaralı

ayağına bir LED bağladım. Bu LED yanıp sönse de yanıp sönme hızı doğrudan okunan analog değere bağlı olmaktadır.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
void adc_init();
unsigned int read_adc(unsigned char channel);
void __delay_ms(int n);
int main(void)
{
	adc_init();
	DDRD |= (1<<PD2);
	while (1)
	{
		unsigned int bekleme = 0;
		int adc = read_adc(0);
		bekleme = adc;
		PORTD |= (1<<PD2);
		__delay_ms(bekleme);
		PORTD &=~(1<<PD2);
		__delay_ms(bekleme);

	}
}
void adc_init(void)
{
	ADCSRA |= ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
	ADMUX |= (1<<REFS0);
	ADCSRA |= (1<<ADEN);
	ADCSRA |= (1<<ADSC);
}

unsigned int read_adc(unsigned char channel)
{
	ADMUX &= 0xF0;
	ADMUX |= channel;
	ADCSRA |= (1<<ADSC);
	while (ADCSRA & (1<<ADSC));
	return ADCW;
}

void __delay_ms(int n) {
	while(n--) {
		_delay_ms(1);
	}
}
```

Biraz uzun bir program bizleri karşılamakta. Öncelikle burada *adc_init(), read_adc()* fonksiyonlarını incelemek gerekir. Bu fonksiyonları doğrudan datasheetten kopyaladım. Zaten farklı bir ADC okuma kodunu da pek göremeyiz çünkü ADC birimini kullanmak sıkı kurallara bağlıdır. Bu kurallar datasheette yazmakta ve bizim bunu kullanabilmemiz için bunlara harfiyen uymamız gerekmektedir. Datasheet okumadan ancak hazır kütüphanelerle bunu kullanmamız mümkündür.
Öncelikle *adc_init()* fonksiyonunda yer alan kodları açıklamakla devam edelim.

```
ADCSRA |= ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
```

Burada ADCSRA, ADPS2, ADPS1 gibi ifadelerin ne olduğunu merak edebilirsiniz. Daha öncesinde bahsettiğim gibi ADCSRA aynı DDRD veya PORTD gibi bir özel fonksiyon yazmacının adıdır ve hafızada belli bir adreste yer almaktadır. Bu yazmacın bitleri ile oynayarak analog-dijital çevirici birimi kontrol edip ayarlamalarını yaparız. *iom328p.h* dosyasına baktığımızda bu tanımlamayı görebiliriz.

```
#define ADCSRA _SFR_MEM8(0x7A)
```

Bunun yanında ADPS2, ADPS1 gibi tanımlamaların nereden geldiğini merak edebilirsiniz. Bu adlar datasheette yazmaktadır. Yani datasheeti okuduğunuzda bu bitleri bu adlarla öğrenirsiniz. Yalnız program yazarken (1<<4), (1<<3) gibi ifadelerle yazmak zorunda kalmayasınız diye bu tanımlamalar ilgili aygıtın dosyasında yapılmıştır.

```
#define ADPS0 0
#define ADPS1 1
#define ADPS2 2
#define ADIE 3
#define ADIF 4
#define ADATE 5
#define ADSC 6
#define ADEN 7
```

Gördüğünüz gibi aslında bunlar sayılardan ibarettir. Yani yukarıda (1<<ADPS2) yazdığınızda aslında (1<<2) yazmış oluyorsunuz. Bu tarz tanımları kullanmak bit konumlarını tek tek ezberlemekten sizi kurtarmaktadır.
Bu noktayı açıkladığımıza göre şimdi kodların ne işe yaradığını açıklayalım. Bahsettiğimiz satırda ADPS2, ADPS1 ve ADPS0 bitleri bir yapılarak ADC ön bölücüsünün bölme değeri 128'e ayarlanmaktadır. AVR denetleyicilerde 32-bit

gelişmiş denetleyiciler gibi ayrı ayrı saat kaynakları olmadığı için bütün çevre birimleri harici olmadığı ve WDT gibi istisnalar hariç CPU saatine yani yazılımda ifade ettiğimiz F_CPU değerine bağlıdır ve buradan beslenir. ADC'nin sağlıklı çalışması için de belli bir saat sınırının altında olması gereklidir. İşte yüksek işlemci hızında bu uygun bölme değerini 128'e bölerek elde etmekteyiz. Eğer ADC biriminin bölme değeri doğru ayarlanmaz ve yüksek frekansla beslenirse kararsızlıklar ve yanlış ölçümlerle karşılaşırız. Bunu 32-bit STM32 mikrodenetleyiciler üzerinde çalışırken de müşahede etmiştim. Mesela bölme değerini ADC'yi hızlı kullanma adına biraz düşürüp ADC saat hızını yükseltince ADC kanalları birbirini etkilemeye ve kararsız çalışmaya başlamıştı. Oysa ki doğru saat hızında oldukça gürültüsüz ve hassas çalışan bir ADC birimine sahipti. Yazılımda sırf bu değeri düzgün ayarlamadınız diye uygulamada çok başınız ağrıyabilir!

```
ADMUX |= (1<<REFS0);
```

Burada ADC'nin referans gerilimi seçilmektedir. AVCC yani besleme gerilimi diyeceğimiz 5 volt gerilim bu kod ile referans gerilimi olmaktadır. Besleme geriliminin doğru olmasının bu durumda doğru okumada nasıl etkili olduğunu görebilirsiniz. Yine de işi garantiye almak için mikrodenetleyicinin içinde 1.1V referans gerilim kaynağı da mevcuttur. Bu durumda 5 volt ile çalışan denetleyiciyi 3.3V ile çalıştırsanız dahi 1.1V referans alınacağı için okunacak değerde bir değişme olmayacaktır. Elbette AREF ayağından da harici bir referans gerilimi uygulayabilirsiniz.

```
ADCSRA |= (1<<ADEN);
```

Burada başlangıçta kapalı olan ADC birimi etkinleştirilmekte ve çalışmaya başlamaktadır. Diğer birimlerde de göreceğiniz üzere gereksiz yere güç tüketmemesi için mikrodenetleyici başladığında kapalı olan birimleri bizim elle açmamız gereklidir.

```
ADCSRA |= (1<<ADSC);
```

Burada da ADC birimi hazır hale gelmesi için ilk ölçüm yapılmaktadır.
Şimdi *read_adc()* fonksiyonunu inceleyelim ve ADC okumasının nasıl yapıldığına bakalım. Buraları okurken bir yandan da datasheetten takip etmeniz çok önemlidir.

```
ADMUX &= 0xF0;
ADMUX |= channel;
```

read_adc() fonksiyonundaki bu kodlar ile ADC kanal seçimi yapılmaktadır. ADC birimi C portuna bağlıdır ve C portunun ayak numaraları ile kanal numaraları aynıdır. Yani Analog 0 kanalı C portunun 0 numaralı ayağına denk gelmektedir. Bunu dijital olarak kullanmamız mümkün olsa da analog olarak kullanmak istediğimizde giriş olarak, *pull-up* dirençleri olmadan yani yüksek empedans (*Hi-Z*) moduna ayarlamak gereklidir. Daha önce dediğimiz gibi port ayarlarına hiç dokunmazsak başlangıçta bu şekilde ayarlanmaktadır.
İlk satırda ADMUX yazmacının ilgili kısmı (son 4 biti) temizlenmekte ve sonrasında ise *channel* argümanına yazdığımız değer ADMUX yazmacına eklenmektedir. Yalnız burada 255 gibi uçuk bir değer yazmakla referans ve hizalama değerlerini de bozacağınızı bilmeniz gerekir. Bu fonksiyonu kullanan programcı bunun farkında olduğu için böyle bir denetimi burada göremeyiz. Bu tarz denetimleri genellikle acemilerin kullandığı Arduino kütüphanelerinde sıkça görmekteyiz.

```
ADCSRA |= (1<<ADSC);
```

Kanal ayarlarını yaptık. Şimdi ise çevirimi başlatmalı ve ADC'nin analog sinyali okumasını beklemeliyiz. Çevrimi başlatsak da ADC bunu anında yapmamakta ve bizi biraz bekletmektedir. Biz ADC'yi beklemek için şöyle bir döngü kullanıyoruz.

```
while (ADCSRA & (1<<ADSC));
```

Yukarıdaki kodda bu biri 1 yaparak çevrimi başlatmıştık. Şimdi ise bu biti okuyarak çevrimin bitip bitmediğini öğreniyoruz. Bu süreçte ise program sonsuz döngüye sokulmakta. Mikrodenetleyici bu şekilde meşgul edilmek istenmezse ADC kesmesi de kullanılabilir. Kullandığımız AVR denetleyicide çevrim bittiğinde yürütülecek bir kesme vektörü yer almaktadır. Bunun uygulamasını ayrıca göstermeyeceğim. Yapacağınız uygulamaların %99'unda verdiğim bilgiler yeterli olacaktır.

```
return ADCW;
```

Burada ise normalde iki parçadan oluşan (10-bit) ADC veri yazmacını geri döndürüyoruz. Normalde 8-bit mimari üzerinde çalıştığımız için 8-bitten büyük veriler iki ayrı yazmaca bölünüp üzerinde çalışmak biraz zahmetli olsa da burada C derleyicisi otomatik olarak bunları yapmaktadır. *AVR Assembly* kullanmadıkça bunu dert etmeniz gerek yok.

Buraya kadar ADC'yi ayarlamayı ve bundan veri okumayı öğrendik. Ama bu okunan değer, değişken bir değer olduğu için *delay* fonksiyonu ile kullanılamayacak. Daha önceden dediğim gibi *delay* içerisine sabit değerleri koymadıkça derleme bile yapılmamakta. Bu durumda kendi *delay* fonksiyonumu yine *delay* kullanarak yazdım. Bu uygulama için gayet de yeterli bir sonuç elde ettim.

```
void __delay_ms(int n) {
      while(n--) {
            _delay_ms(1);
      }
```

Burada okunan değere göre *_delay_ms(1)* fonksiyonunun kaç defa çalıştırılacağı belirleniyor. N değeri ise ADC değeri olduğu için 0-1023 mili saniye aralıklarla bu program çalışacaktır.

12.Uygulama : AVR Port Performansı

Bu uygulamada mikrodenetleyicinin sınırlarını zorlayıp ondan alabileceğim en yüksek değerleri almayı hedefledim. Bunun için PORT değişkenlerini herhangi bir bekleme koymadan ardı ardına açıp kapattım. Yalnız döngü içerisinde olduğu için atlama komutları da kullanıldığından bu kullanılan komutlar iki çevirim (*cycle*) de olsa bir gecikme vermekte. Yani 16MHz'de çalışan bir denetleyicide 8 yerine 4MHz'lik bir sinyal almaktayım. Bu sinyali gözlemleyebilmek için iyi bir frekansmetre veya daha iyisi osiloskopa ihtiyacınız olacaktır.

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

int main(void)
{
      DDRD = 0xFF;
    while(1)
    {
      PORTD = 0xFF;
      PORTD = 0x00;
    }
}
```

Aslında kaç MHz'lik sinyal alacağımızı bilmek için osiloskopa ihtiyacımız yok. Programı yazarken bile bunu anlayabiliriz. Yalnız C diline bağlı kalırsak bunu anlamamızın imkânı olmaz. Çünkü burada komutların birbiri ardında işletildiğini

ama gerçekte kaç komut işletildiğini ve kaç çevirim harcandığını bilmiyoruz. Bu durumda daha derine inmek için derlenmiş dosyaya yani Assembly komutlarına bakmamız gereklidir. Proje içerisinde yer alan *.lss* uzantılı dosyayı açarak Assembly komutlarına çevrilen C kodlarını görebiliriz.

```
    while(1)
    {
    PORTD = 0xFF;
  84: 8b b9           out   0x0b, r24  ; 11
     PORTD = 0x00;
  86: 1b b8           out   0x0b, r1   ; 11
  88: fd cf           rjmp  .-6        ; 0x84 <main+0x4>
```

Burada hem makine kodlarını hem C kodlarını hem de Assembly dilinde makine kodlarını görebiliriz. Görüldüğü gibi bu döngüde iki adet OUT ve bir adet RJMP komutu kullanılmakta. OUT komutu ile özel fonksiyon yazmaçlarına değer atanmakta ve RJMP komutu ile de programda istenilen noktaya atlanmakta. Bu komutları "*Instruction Set Manual*" adlı kılavuzda bütün ayrıntısıyla görebiliriz. İncelemeniz için kılavuzun bağlantısını aşağıya bırakıyorum.

http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

Burada bizi ilgilendiren nokta şimdilik bütün bu kodların toplamda kaç çevirimde (*cycle*) işletildiğidir. OUT komutu bir çevirimde işletilmekte ve RJMP komutu ise iki çevirim harcamaktadır. Yani toplamda 4 çevirim ile 1 ve ardından 0 sinyalini elde etmekteyiz. İşlemci saati olan 16MHz'i bu değer ile böldüğümüzde sonuç olarak 4MHz'i elde ederiz. İlerleyen konularda zamanlayıcılar vasıtasıyla işlemci hiç meşgul edilmeden bunun iki katı daha hızlı sinyali nasıl elde edeceğimizi göstereceğim.

13.Uygulama : Karakter LCD

AVR denetleyicilerde karakter LCD kullanmak için programcıların pek çoğu Peter Fluery'in karakter LCD kütüphanesini kullanmaktadır. Burada da bu kütüphaneyi kullanarak ufak bir deneme uygulaması yaptım. Öncelikle kütüphaneyi şu bağlantıdan indirmelisiniz.

http://www.peterfleury.epizy.com/?i=1

Kütüphane *lcd.h* ve *lcd.c* dosyalarından meydana gelmekte. *lcd_definitions.h* dosyasını burada kullanmayacağız. *lcd.h* dosyasında ise belli başlı konfigürasyonları yapmak gerekli. Kurduğunuz devre ve kullandığınız LCD modüle göre bunu her projede ayarlamalısınız.

```
#define LCD_IO_MODE      1            /**< 0: memory mapped mode, 1:
IO port mode */

#if LCD_IO_MODE

#ifndef LCD_PORT
#define LCD_PORT         PORTD        /**< port for the LCD lines
*/
#endif
#ifndef LCD_DATA0_PORT
#define LCD_DATA0_PORT   LCD_PORT     /**< port for 4bit data bit 0
*/
#endif
#ifndef LCD_DATA1_PORT
#define LCD_DATA1_PORT   LCD_PORT     /**< port for 4bit data bit 1
*/
#endif
#ifndef LCD_DATA2_PORT
#define LCD_DATA2_PORT   LCD_PORT     /**< port for 4bit data bit 2
*/
#endif
#ifndef LCD_DATA3_PORT
#define LCD_DATA3_PORT   PORTB    /**< port for 4bit data bit 3 */
#endif
#ifndef LCD_DATA0_PIN
#define LCD_DATA0_PIN    5            /**< pin for 4bit data bit 0
*/
#endif
```

```
#ifndef LCD_DATA1_PIN
#define LCD_DATA1_PIN    6            /**< pin for 4bit data bit 1
*/
#endif
#ifndef LCD_DATA2_PIN
#define LCD_DATA2_PIN    7            /**< pin for 4bit data bit 2
*/
#endif
#ifndef LCD_DATA3_PIN
#define LCD_DATA3_PIN    0            /**< pin for 4bit data bit 3
*/
#endif
#ifndef LCD_RS_PORT
#define LCD_RS_PORT      LCD_PORT     /**< port for RS line
*/
#endif
#ifndef LCD_RS_PIN
#define LCD_RS_PIN       2            /**< pin  for RS line
*/
#endif
#ifndef LCD_RW_PORT
#define LCD_RW_PORT      LCD_PORT     /**< port for RW line
*/
#endif
#ifndef LCD_RW_PIN
#define LCD_RW_PIN       3            /**< pin  for RW line
*/
#endif
#ifndef LCD_E_PORT
#define LCD_E_PORT       LCD_PORT     /**< port for Enable line
*/
#endif
#ifndef LCD_E_PIN
#define LCD_E_PIN        4           /**< pin  for Enable line
*/
#endif
```

Yukarıdaki konfigürasyona göre Arduino-LCD bağlantıları şu şekilde olmalıdır
.

Arduino UNO	Karakter LCD
D2	RS
D3	R/W
D4	E
D5	D0
D6	D1
D7	D2
D8	D3

Github'a yüklediğim projede kütüphane dosyası bu konfigürasyonla beraber gelmektedir. O yüzden sadece yukarıdaki tablodaki bağlantılara göre devreyi kurmanız yeterlidir.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include "lcd.h"

int main (void)
{
	lcd_init(LCD_DISP_ON);
	lcd_clrscr();
	lcd_home();
	char str [16];
	int pi = 30;
	sprintf(str, "Sayi = %i", pi);
	lcd_puts(str);
	while(1)
	{
	}
}
```

Kütüphane fonksiyonlarının daha kapsamlı açıklaması için C ile AVR programlama yazılarına bakabilirsiniz. Burada sadece kullanılan fonksiyonları kısaca açıklayıp ardından önemli bir konuya değineceğim.

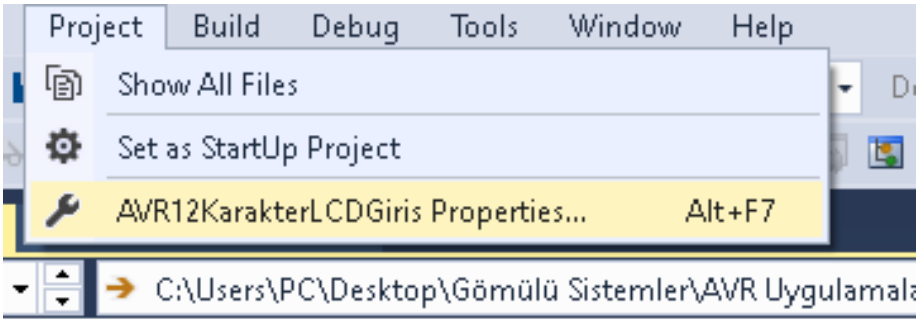
```
	lcd_init(LCD_DISP_ON);
	lcd_clrscr();
	lcd_home();
```

Burada *lcd_init()* fonksiyonu ile LCD'yi tanımlayıp başlattıktan sonra *lcd_clrscr()* fonksiyonu ile ekranı temizliyoruz ve ardından *lcd_home()* fonksiyonu ile imleci en başa götürüyoruz. LCD ekranda bir yazı yazdırmak için *lcd_puts()* fonksiyonunu kullanmamız gerekli. Yalnız bu fonksiyon sadece karakter dizisi tipinde değer kabul etmekte. Yani *integer, float* gibi değerleri yazdırabilmek için bunları karakter dizisine çevirmemiz gerekli. *Integer* için C kütüphanesinde *itoa()* gibi fonksiyonlar olsa da bu işler için en beğendiğim fonksiyon *sprintf()* fonksiyonudur. Aynı *printf()* fonksiyonu gibi istediğimiz her değeri karakter dizisi içerisinde istediğimiz biçimde yazdırma imkanımız var. Sadece boş bir karakter dizisi oluşturup bunun içerisine yazdırıyoruz. Kullanım bakımından *printf()* ile aynı olan fonksiyon işlev olarak giriş çıkış akışına değil bizim tanımladığımız boş diziye yazdırma işlevi yapmakta. Sonrasında o diziyi de fonksiyon argümanı olarak kullanmaktayız.

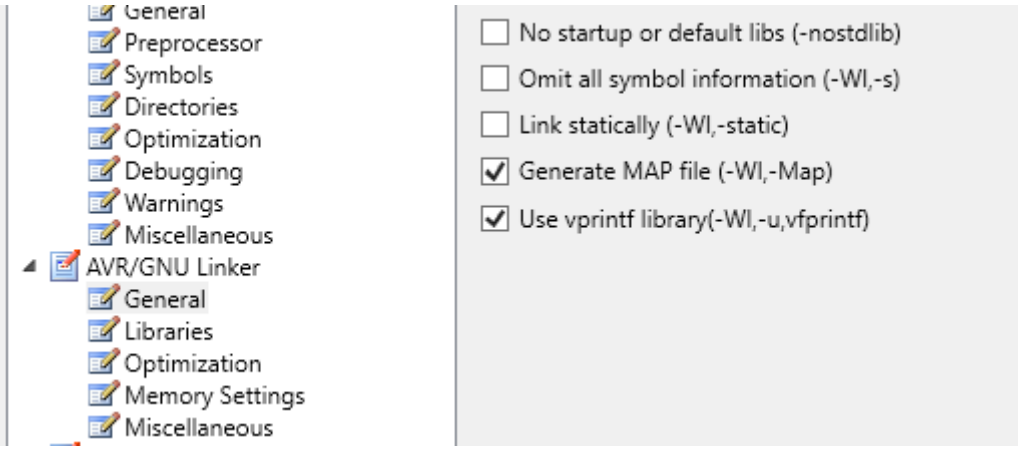
```
sprintf(str, "Sayi = %i", pi);
```

Burada görüldüğü gibi %i format belirleyicisi ile pi değişkenini karakter dizisi içerisine yerleştirip kolayca ekranda yazdırabildik. Ama *float* tipindeki değişkenlerde bunu denemek istediğimizde ekranda gerçek anlamda bir soru işareti (?) çıkacaktır. Bu soru işaretinin neden çıktığını ne C kitaplarında ne de AVR'ın datasheetinde bulabilirsiniz. Aslında çıkmaması gerekli, fakat bağlayıcı (*linker*) performans kaygılarından dolayı böyle bir kısıtlamaya gitmiş ve siz ek bir ayar yapmadığınız sürece *sprintf()* fonksiyonu *float* tipindeki değişkenleri desteklememekte. Bunun çözümü ise şöyle olmaktadır.

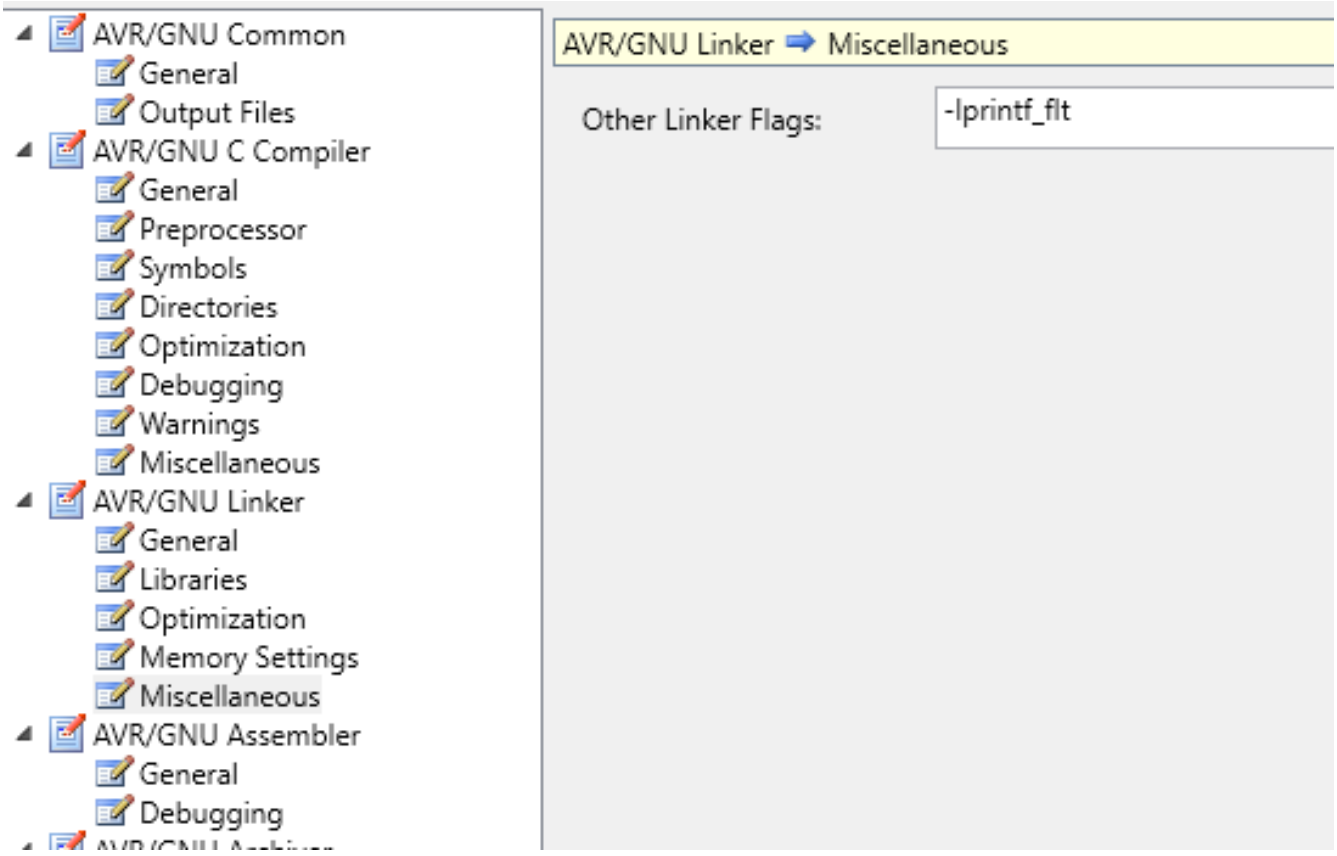
Öncelikle yukarıdaki Project adlı sekmeden ayarlar kısmına geliyoruz.



Sonrasında AVR/GNU Linker kısmından General sekmesini seçiyoruz ve "*Use vprintf library*" kutucuğunu işaretliyoruz.



Sonrasında "Miscellaneous" kısmındaki metin kutusuna -lprintf_flt parametresini yazıyoruz.



Artık LCD ekrana float değerlerini de yazdırabileceğiz. Dezavantaj olarak ise program hafızada biraz daha fazla yer kaplayacak. Görüldüğü gibi sırf C dilini ve donanımı da bilmek yeterli değil. Aynı zamanda kullandığınız derleyici, stüdyo ve diğer araçları da kullanmayı bilmeniz gerekiyor.

14. Uygulama : ADC ve LCD Uygulaması

Bu uygulama ile artık ADC'den okuduğumuz değerleri LCD ekranda göstereceğiz ve gerçek anlamda projeleri ortaya koymada büyük bir adım atmış olacağız. Bu uygulamayı esas alarak bir dijital termometre, alarm sistemi veya meteoroloji istasyonu yapabilirsiniz. Pek çok projede analog değer okuma ve LCD ekrana veya bir göstergeye yazdırma işlemlerini görebilirsiniz. Sizin de kendi projelerinizi yapmanız için bu uygulamada işi en temelden göstermek istedim.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include "lcd.h"
// Fonksiyon Prototipleri
void adc_init(void);
unsigned int read_adc(unsigned char channel);
long map(long x, long in_min, long in_max, long out_min, long out_max);
int main(void)
{
      lcd_init(LCD_DISP_ON);
      adc_init();
    while(1)
    {
    lcd_home();
      lcd_puts("HAM:");
      unsigned int adc_deger = read_adc(0);
      char lcd_ch[10]="";
      sprintf(lcd_ch, "%u", adc_deger);
      lcd_puts(lcd_ch);
      lcd_puts("   ");

      // ALT SATIR
      lcd_gotoxy(0,1);
      lcd_puts("YUZDE:");
      lcd_ch[0] = '\0'; // String sıfırlandı
    long deger = map(adc_deger, 0, 1023, 0, 100);
      sprintf(lcd_ch, "%u", (int)deger);
```

```
        lcd_puts(lcd_ch);
        lcd_puts("%   ");


    }
}

void adc_init(void)
{
        ADCSRA |= ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
        ADMUX |= (1<<REFS0);
        ADCSRA |= (1<<ADEN);
        ADCSRA |= (1<<ADSC);
}

unsigned int read_adc(unsigned char channel)
{
        ADMUX &= 0xF0;
        ADMUX |= channel;
        ADCSRA |= (1<<ADSC);
        while (ADCSRA & (1<<ADSC));
        return ADCW;
}

long map(long x, long in_min, long in_max, long out_min, long out_max)
{
        return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min;
}
```

Burada daha önceden anlattığım ADC fonksiyonlarını tekrar anlatmayacağım. Öncelikle LCD ekrana okunan ham ADC değerini yazdırıyoruz. Bu işlemi yapan kod blokuna daha yakından bakalım.

```
        lcd_puts("HAM:");
        unsigned int adc_deger = read_adc(0);
        char lcd_ch[10]="";
        sprintf(lcd_ch, "%u", adc_deger);
        lcd_puts(lcd_ch);
        lcd_puts("   ");
```

Daha öncesinde *lcd_home()* diyerek ilk satıra geçtik. Sonrasında bir değişken tanımlayıp buna 0 numaralı kanaldan okunan ADC değerini atadık. Sonrasında

daha önceden gösterdiğim gibi *sprintf()* ile bu sayı değerini önce karakter dizisine çevirdik sonrasında ise ekrana yazdırdık. Sonrasında ise belli bir miktarda boşluk yani " " ifadesini koyduğuma dikkat edin. Bu boşluğu koymamdaki sebep değer 1000 gibi dört haneli iken 10 gibi iki haneli bir sayıya düştüğünde arta kalan kısmı temizlemesi içindir. Bu karakter LCD ekranlarda ekran temizleme komutu olsa da bunu hiç kullanma ihtiyacınız yoktur. Silmek istediğiniz kısma boşluk karakterleri ekleyin. Eğer silme komutunu sürekli kullanırsanız ekran pırpır edecek ve gözü rahatsız edecektir. Ben şahsen acemi iken ekran silme komutunu sıkça kullanıyordum ve bunun zararını tecrübe ettim.
Burada ekranın alt satırına geçmek için aşağıdaki komutu kullanmaktayız.

```
lcd_gotoxy(0,1);
```

İlk argüman x yani yatay konum değeri, ikinci argüman ise y yani düşey konum değeridir. Oldukça basit bir kullanıma sahip bir fonksiyondur.

Sonrasında ise analog okuma ve bu veriyi işlemede oldukça mühim olan *map()* fonksiyonunu görmekteyiz. Bu map fonksiyonu hakkında bir uygulama örneği olarak batarya şarj yüzdesini gösteren bir uygulamadan bahsedebiliriz. Örneğin 12V'luk bir batarya ile sistemi besliyoruz ve bu batarya 9 volta düştüğünde bitmiş durma gelmesi gerekiyor ve şarj gerektiriyor. Bu durumda bizim 12 voltta %100, 9 voltta ise %0 gösterecek bir göstergeye ihtiyacımız var diyelim. Biz map fonksiyonu ile ilk 1024 ve 1024-255 değeri arasında okunan değeri 0 ve 100'e haritalandırmamız lazım. 255 dememin sebebi 12 volt ölçülürken 0-3 volt arası sayısal 255 değerine tekabül etmektedir. Yani analog okuma değeri 768 olduğu zaman 9 voltu okumuş oluyoruz. Bunu teorik bakımdan söylesem de pratik olarak 12 voltluk bataryaların daha üst seviye gerilimlere sahip olabildiğini ve bunun için pay bırakılması gerektiğini bilmeniz gereklidir. Ayrıca yüksek gerilimleri gerilim bölücü direnç ile analog girişlere bağlamanız gereklidir.

```
long map(long x, long in_min, long in_max, long out_min, long
out_max);
```

Bu *map()* fonksiyonunda bir x değeri, bu değerin aldığı asgari değer, bu değerin aldığı azami değer, çıkıştaki asgari değer ve çıkıştaki azami değer olarak argümanları görmekteyiz. Şimdi bunun nasıl çalıştığını uygulamada görelim.

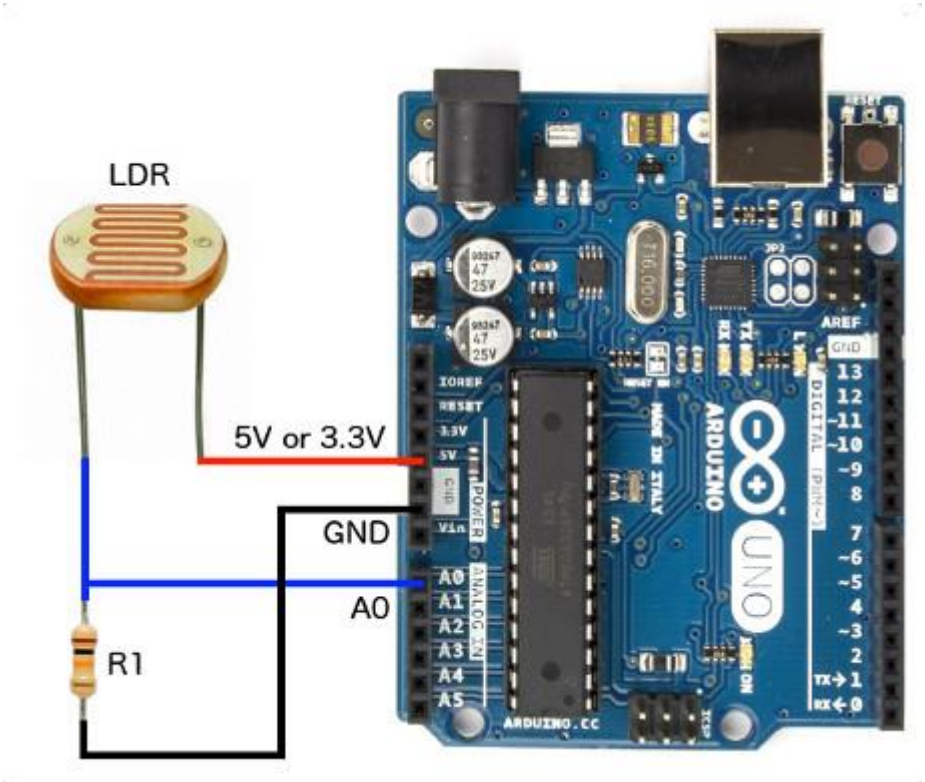
```
long deger = map(adc_deger, 0, 1023, 0, 100);
```

Öncelikle *adc_deger* değişkenine okunan ham ADC değeri daha önceden aktarılmaktadır. Bu okunan ADC değeri 0 ile 1023 arasında (10-bit) olacağı için bir sonraki argümana 0, diğerine ise 1023 değeri yazılmıştır. Bunu doğru yazmak

önemlidir. Sonrasında ise elde etmek istediğimiz değer aralığı 0 ve 100 olarak yani yüzde şeklinde yazılmaktadır. İstediğiniz sayı aralığını girebilirsiniz. Artık okunan ADC değeri işlemden geçecek ve 0 ve 100 arasında bir değer olacaktır.
Bu devreyi kurmak için A0 ayağına bir potansiyometre bağlamak ve yukarıda verdiğim bağlantıda LCD ekranı bağlamak yeterlidir.

15. Uygulama : LDR ve LCD Uygulaması

Bu uygulama yukarıdaki uygulamaya benzerlik gösterse de bir algılayıcıdan okunan analog değerin nasıl yorumlanabileceğini gösterme adına güzel bir uygulamadır. Öncelikle LDR'ı resimde gördüğünüz gibi Arduino kartına bağlamanız gereklidir.



Yukarıdaki LCD uygulamasının devresini hiç bozmadan sadece bu modifikasyonla uygulamayı çalıştırabilirsiniz. Şimdi kodlara bakalım.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include "lcd.h"
// Fonksiyon Prototipleri
```

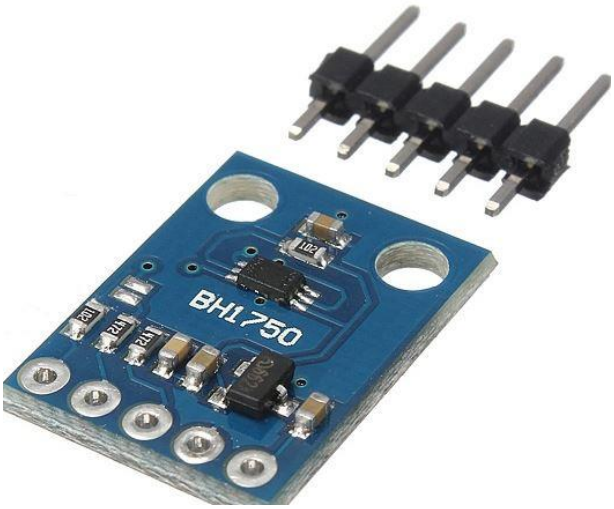
```
void adc_init(void);
unsigned int read_adc(unsigned char channel);
long map(long x, long in_min, long in_max, long out_min, long out_max);
int main(void)
{
      lcd_init(LCD_DISP_ON);
      adc_init();
    while(1)
    {
    lcd_home();
      unsigned int adc_deger = read_adc(0);
    long deger = map(adc_deger, 0, 1023, 0, 100);
      if (deger>80)
            {
                  lcd_puts("AYDINLIK");
            }
    else if (deger > 50)
            {
                  lcd_puts("GOLGE");
            }
      else
            {
                  lcd_puts("KARANLIK");
            }
      lcd_puts("     "); // boşluk yazdırma

    }
}

void adc_init(void)
{
      ADCSRA |= ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
      ADMUX |= (1<<REFS0);
      ADCSRA |= (1<<ADEN);
      ADCSRA |= (1<<ADSC);
}

unsigned int read_adc(unsigned char channel)
{
      ADMUX &= 0xF0;
      ADMUX |= channel;
      ADCSRA |= (1<<ADSC);
      while (ADCSRA & (1<<ADSC));
      return ADCW;
}

long map(long x, long in_min, long in_max, long out_min, long out_max)
```

```
{
      return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min;
}
```

Burada farklı olan kısma dikkat edersek bir if yapısıyla ortamın karanlık mı gölge mi yoksa aydınlık mı olduğu belirlenmektedir.

```
if (deger>80)
            {
                  lcd_puts("AYDINLIK");
            }
    else if (deger > 50)
            {
                  lcd_puts("GOLGE");
            }
      else
            {
                  lcd_puts("KARANLIK");
            }
```

Burada değer 80'den büyük ise aydınlık, 50'den büyük ise gölge ve 50 ve aşağısında bir değerde ise karanlık yazdırılmaktadır. Bu değerleri öğrenmek için biraz da deney yapmak lazımdır açıkçası! Öncelikle gözümüzle gördüğümüz ve bütün insanların gölge olarak kabul ettiği ışık aralığında LDR'dan hangi değerlerin okunduğunu tespit edip not alıyoruz. Bunu aydınlık ve karanlık diyeceğimiz eşik değerleri belirlemek için de aynı şekilde uygulamaktayız. Yani kendi gözlemimizle elde ettiğimiz bilgiyi programa aktarıyoruz ve program bizim beynimiz yerine, LDR ise gözümüz yerine iş yapıyor. Sonuçta LDR ilkel ve toleranslı bir algılayıcı olduğu için bu şekilde çalışmamız gerekebilir. Ama biraz daha araştırdığınızda tamamen dijital iletişim protokolleri ile çalışan luxmetre modüllerinin bile olduğunu görebilirsiniz.



Bahsettiğim dijital luxmetre modüllerinden BH1750 oldukça yaygındır. Yukarıda da bu modülün resmini görebilirsiniz.

Bütün sensörlere ve modüllere baktığımızda artık analog ve ilkel modüllerin yerini özellikle son yıllarda artan bir hızda dijital modüllerin aldığını görmekteyiz. Artık kimse analog değerle, formüllerle, kalibrasyonla uğraşmak istemiyor. Fabrika çıkışında kalibre edilmiş bir dijital sensör kendisine güç verildiği zaman okuduğunu sayısal değer olarak bize vermekte. Üstelik bu dönüşümleri, hesaplamaları da kendi içerisinde yapmakta. Biz çok elektronik ve fizik bilmeden sadece yazılım ile bu tarz aygıtları kullanabilir ve projelerimize dahil edebiliriz. Her zaman bu alanda yazılımın öneminin giderek arttığını söylemekteyim. İsteyenler Analog Devices'ın yazılım ile programlanabilir sinyal üreteçlerini veya filtrelerini inceleyebilir. Artık ileri seviye analog elektronik bilgisi gerektiren devreleri kurmak yerine bu tarz çipleri kullanıp yazılım ile çok daha kolay bir şekilde aynı işi, hem de daha iyi bir şekilde yapabiliyoruz.

16. Uygulama : LM35, LDR ve LCD Uygulaması

Yukarıda yaptığımız uygulamaya LM35 sıcaklık algılayıcısını da bağlayıp hep ortam ışığını hem de sıcaklığını ölçen bir cihaz yapacağız. Bunun için LM35'in çıkışı A1 ayağına bağlanmalıdır. Geri kalan hesaplamanın nasıl yapıldığını kod üzerinden anlatacağım.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include "lcd.h"
// Fonksiyon Prototipleri
void adc_init(void);
unsigned int read_adc(unsigned char channel);
long map(long x, long in_min, long in_max, long out_min, long out_max);
int main(void)
{
      lcd_init(LCD_DISP_ON);
      adc_init();
    while(1)
    {
    lcd_home();
      unsigned int adc_deger = read_adc(0);
```

```
    long deger = map(adc_deger, 0, 1023, 0, 100);
      if (deger>80)
            {
                  lcd_puts("AYDINLIK");
            }
    else if (deger > 50)
            {
                  lcd_puts("GOLGE");
            }
      else
            {
                  lcd_puts("KARANLIK");
            }
      lcd_puts("     "); // boşluk yazdırma

      //LM35 ve sıcaklık yazdırma
      unsigned int lm35_ham = read_adc(1);
      float mv = ( lm35_ham/1024.0)*5000;
      float cel = mv/10;
      char lcd_str[5];
      sprintf(lcd_str,"%i",(int)cel);
      lcd_gotoxy(0,1);
      lcd_puts("SICAKLIK:");

      lcd_puts(lcd_str);
      lcd_puts("    ");
      _delay_ms(200);
    }
}

void adc_init(void)
{
      ADCSRA |= ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
      ADMUX |= (1<<REFS0);
      ADCSRA |= (1<<ADEN);
      ADCSRA |= (1<<ADSC);
}

unsigned int read_adc(unsigned char channel)
{
      ADMUX &= 0xF0;
      ADMUX |= channel;
      ADCSRA |= (1<<ADSC);
      while (ADCSRA & (1<<ADSC));
      return ADCW;
}
```

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
      return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min;
}
```

Burada LM35'den analog değeri okuyup bunu ekrana yazdırma kısmını inceleyeceğim. Geri kalanı bir önceki örnek ile aynı kodlardır.

```
      unsigned int lm35_ham = read_adc(1);
      float mv = ( lm35_ham/1024.0)*5000;
      float cel = mv/10;
```

Öncelikle *lm35_ham* adında bir değişken oluşturup okunan ham değeri kaydediyoruz. Sonrasında ise bir formülü görmekteyiz. Bu okunan milivolt değerini bulmamıza yardımcı olmaktadır. Biz ADC'den ham değer olarak 0-1023 arası değeri okumaktayız ve burada referans gerilimi de 5000 milivolt yani 5 volt olmakta. Bunu 1024 ile böldüğümüzde elimizde 0-1 arası bir ondalıklı değer kalacaktır. Referans gerilimi 5V olduğu için bunu 5000 ile çarpıyoruz ve gerilim değerini böylelikle buluyoruz.

Elde edilen gerilim değeri ile gerilim ölçer yapsak da bizim hedefimiz bir sıcaklık ölçer yapmak. LM35'in datasheetinde her bir santigrat derece için 10mV gerilim verdiği yazmakta. O halde bulunan milivolt değeriin 10'a böldüğümüzde santigrat derece olarak değeri buluyoruz.

Burada değerlerin *float* cinsinden olduğunu unutmayın. Çünkü bu işlemde pek çok zaman ondalıklı değer elde edeceksiniz.

```
sprintf(lcd_str,"%i",(int)cel);
```

Burada ben biraz kolaya kaçıp *float* olan *cel* değerini (int) tip dönüştürücü ile tam sayıya çevirip öyle yazdırdım. Daha önceden dediğim gibi *sprintf()* fonksiyonu bu derleyicide *float* değerleri doğrudan desteklemiyor ve soru işareti olarak bize geri dönüyor. Bu uygulamayı geliştirmek adına yukarıda bahsettiğim *linker* ayarlarını yaparak *float* olarak yazdırabilirsiniz.

17. Uygulama : Dahili Sıcaklık Algılayıcısı

AVR denetleyicilerde bir de dahili sıcaklık algılayıcısı yer almaktadır. ADC üzerinden gerekli ayak ayarları yapıldığı zaman içeriden bu algılayıcıya bağlanmakta ve sıcaklık ölçülmektedir. Bu sıcaklık algılayıcısının kalibrasyonu uğraştırdığı ve doğruluğu yüksek olmadığı için aşırı ısınma durumlarında alarm olarak kullanılabilir. Unutulmaması gereken bir nokta da ortam sıcaklığı değil denetleyicinin iç sıcaklığının ölçüldüğüdür. Bilgisayar işlemcileri gibi bu denetleyiciler de çalıştığı zaman ısınmaktadır. Özellikle portlara çok yüklenirse bu sıcaklık oldukça artabilir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <stdlib.h>
#include "lcd.h"
unsigned int Ctemp;
unsigned int Ftemp;
int main(void)
{

      lcd_init(LCD_DISP_ON);
    /* Setup ADC to use int 1.1V reference
    and select temp sensor channel */
    ADMUX = (1<<REFS1) | (1<<REFS0) | (0<<ADLAR) | (1<<MUX3) | (0<<MUX2) |
(0<<MUX1) | (0<<MUX0);

    /* Set conversion time to
    112usec = [(1/(8Mhz / 64)) * (14 ADC clocks  per conversion)]
     and enable the ADC*/
    ADCSRA = (1<<ADPS2) | (1<<ADPS1) | (1<<ADEN);

    /* Perform Dummy Conversion to complete ADC init */
    ADCSRA |= (1<<ADSC);

    /* wait for conversion to complete */
    while ((ADCSRA & (1<<ADSC)) != 0);

    while(1)
    {
     /* start a new conversion on channel 8 */
     ADCSRA |= (1<<ADSC);

     /* wait for conversion to complete */
     while ((ADCSRA & (1<<ADSC)) != 0);
```

```
      /* Calculate the temperature in C */
      Ctemp = (ADC - 247)/1.22;
      Ftemp = (Ctemp * 1.8) + 32;

        char lcd_ch[10]="";
        itoa((int)Ctemp, lcd_ch, 10);
        lcd_home();
        lcd_puts(lcd_ch);
        lcd_puts("C     ");
    }
}
```

Bu kodu mikrodenetleciyi üreten firma verdiği için olduğu gibi kopyalayıp uygulamada kullandım. Önceki LCD uygulamasına hiç dokunmadan bunu çalıştırabilirsiniz. Çok üzerinde durmaya gerek olmasa da fikir edinme açısından deneyebilirsiniz. Daha ilerisi için şu uygulama notunu okumak faydalı olabilir.

https://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en591573
https://microchipdeveloper.com/8avr:avrtemp
https://microchipdeveloper.com/8avr:avradc

18. Uygulama : AVR ADC Kütüphanesi

Önceki uygulamalarda karakter LCD ekran kullanmak için hazır kütüphane kullanmış ve bunun için kütüphane yazmaya girişmemiştim. Halen de kütüphane yazma ihtiyacı hissetmiyorum çünkü hali hazırda yazılmış güzel bir kütüphane mevcut. Fakat AVR konularının hepsi için bu geçerli değil. Mesela ADC konusunda bir kütüphane aradığım zaman hepsi yarım yamalak yazılmış ve gerçek bir sürücü niteliğinde olmayan kütüphanelerle karşılaştım. Bu durumda benim eksiksiz bir kütüphane yazmam gerekti ve işe koyuldum. Kütüphane yazmak basit kütüphanelerde bile oldukça zahmetli olsa da sonrasında işiniz oldukça kolaylaşmakta ve aynı kodları tekrar tekrar yazmak durumunda kalmamaktasınız. Önce kodları vereyim ve sonrasında kütüphaneyi nasıl yazdığımı ve nasıl kullanacağınızı anlatayım.

ADC.H dosyası

```
#ifndef ADC_H
#define ADC_H
/*
    AVR Atmega328P ADC Library
    Written By Gökhan Dökmetaş 2020
    www.lojikprob.com
*/
// INCLUDES
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
// USER CONFIGURATIONS
//**********************BEGINNING OF USER CONFIGURATIONS
***************************//
#define ADC_REFERENCE ADC_REF_VCC

/*  Define Your ADC Reference
    ADC_REF_EXTERNAL : AREF Pin
    ADC_REF_VCC      : Power Supply
    ADC_REF_INTERNAL : Internal 1.1V Reference
*/
#define ADC_PRESCALER ADC_DIV128
/*
   Define Prescaler Value
   ADC_DIV2 /2
   ADC_DIV3 /3
   ADC_DIV4 /4
   ADC_DIV8 /8
   ADC_DIV16 /16
   ADC_DIV32 /32
   ADC_DIV64 /64
   ADC_DIV128 /128
*/
#define TRIGGER_SOURCE FREE_RUN
/*
    Define Trigger Source
    FREE_RUN
    ANALOG_C  // Analog Comparator
    EXINT_0  // External Interrupt Request 0
    TC0_COMP_A // Timer/Counter0 Compare Match A
    TC0_FLOW // Timer/Counter0 Overflow
    TC1_COMP_B // Timer/Counter1 Compare Match B
    TC1_FLOW  // Timer/Counter1 Overflow
    TC1_CAPTURE // Timer/Counter1 Capture Event
*/
```

```
// USE adc_autotrig(uint8_t mode) function to enable trigger sources

// ADC INTERRUPTS
// adc_interrupt(ENABLE) to Enable ADC Interrupt
// adc_interrupt(DISABLE) to Disable ADC Interrupt

/*
    DIGITAL PIN DISABLE
    USE adc_disable_digital(int pins); function to disable digital pins for
    reduce power consumption.
    DISABLE_NONE 0 // ENABLE Digital Pins
    DISABLE_0   // Disable Pin 0
    DISABLE_1   // Disable Pin 1
    DISABLE_2   // Disable Pin 2
    DISABLE_3   // Disable Pin 3
    DISABLE_4   // Disable Pin 4
    DISABLE_5   // Disable Pin 5
    DISABLE_ALL // Disable All Pins
//****************** END OF USER CONFIGURATIONS
***********************************/
// Definitions and Macros
//
#define adc_set_channel(channel) (ADMUX |= channel)

// ADC reference values
#define ADC_REF_EXTERNAL 0
#define ADC_REF_VCC      1
#define ADC_REF_INTERNAL 3  // 0b11

// ADC Prescaler Values
#define ADC_DIV2     0
//#define ADC_DIV2     1
#define ADC_DIV4     2
#define ADC_DIV8     3
#define ADC_DIV16    4
#define ADC_DIV32    5
#define ADC_DIV64    6
#define ADC_DIV128   7

// ADC Auto Trigger Source Values
#define FREE_RUN    0
#define ANALOG_C    1
#define EXINT_0     2
#define TC0_COMP_A  3
#define TC0_FLOW    4
#define TC1_COMP_B  5
#define TC1_FLOW    6
```

```
#define TC1_CAPTURE 7
// STATUS
#define DISABLE 0
#define ENABLE  1
// Disable Digital Pins
#define DISABLE_NONE 0
#define DISABLE_0    1
#define DISABLE_1    2
#define DISABLE_2    4
#define DISABLE_3    8
#define DISABLE_4    16
#define DISABLE_5    32
#define DISABLE_ALL  63

// Functions

// init adc
extern void adc_init(void);
// deinit adc
extern void adc_deinit();
// Start converstaion and return data (single & first of free-run)
extern uint16_t adc_read(uint8_t channel);
// Return only ADC data, use at free-run and other trigger sources
extern uint16_t adc_read_data();
// Enable and select autotrigger sources
extern void adc_autotrig(uint8_t mode);
//Enable or disable ADC interrupt
extern void adc_interrupt(uint8_t status);
// Disable digital inputs for reduce power consumption
extern void adc_disable_digital(uint8_t pins);
// ADC Left Adjust
extern void adc_left_adjust();
// Read ADC with reduced noise
extern uint16_t adc_read_smooth(uint8_t channel);
// ADC Read + Map Function
extern long adc_read_map(uint8_t channel, long out_min, long out_max);
#endif // ADC_H
```

ADC.C Dosyası

```
#include "adc.h"

void adc_init(void){
      ADCSRA |= (ADC_PRESCALER);
      ADMUX |= ((ADC_REFERENCE) << 6);
      ADCSRA |= (1<<ADEN); // Enable ADC
      ADCSRA |= (1<<ADSC); // Make first conversation and ready adc
}
```

```
void adc_deinit(){
	ADCSRA &= ~(1<<ADEN); // Shut-down ADC
}

uint16_t adc_read(uint8_t channel)
{
	ADMUX &= 0xF0;
	ADMUX |= channel;
	ADCSRA |=(1<<ADSC);
	while(ADCSRA & (1<<ADSC));
	return ADCW;
}

uint16_t adc_read_smooth(uint8_t channel)
{
	uint16_t adtotal = 0;
	for (int i=0; i<10; i++)
	{
		ADMUX &= 0xF0;
		ADMUX |= channel;
		ADCSRA |=(1<<ADSC);
		while(ADCSRA & (1<<ADSC));
		adtotal += ADCW;
	}
	return adtotal / 10;
}

long adc_read_map(uint8_t channel, long out_min, long out_max)
{
	ADMUX &= 0xF0;
	ADMUX |= channel;
	ADCSRA |=(1<<ADSC);
	while(ADCSRA & (1<<ADSC));
	long x = ADCW;
	x = (x - 0) * (out_max - out_min) / (1023 - 0) + out_min;
	return x;
}
uint16_t adc_read_data()
{
	return ADCW;
}

extern void adc_autotrig(uint8_t mode)
{
	ADCSRB |= (mode); // last 3 bits
	ADCSRA |= (1<<ADATE);
```

```
	if (mode == FREE_RUN)
	ADCSRA |= (1<<ADSC); // Start in free running.
	// If you dont want to auto start, comment this and use adc_read()
}

void adc_interrupt(uint8_t status)
{
	if (status == 1) // ENABLE INTERRUPTS
	{
		ADCSRA |= (1<<ADIE);
	}
	else
	{
		ADCSRA &= ~(1<<ADIE);
	}

}

extern void adc_disable_digital(uint8_t pins)
{
	DIDR0 = pins;
}

extern void adc_left_adjust()
{
	ADMUX |= (1<<ADLAR);
}
```

Bu kütüphaneyi Github'da yayınladığım için açıklamaları ve değişken adlarını İngilizce yazdım. Anlamakta zorlanmayacağınızı ümit ediyorum.

ADC.H dosyasına baktığımızda en başta "*User Configurations*" yani kullanıcı ayarları kısmının olduğunu görmekteyiz. Burada ADC biriminin nasıl çalışacağı noktasında çeşitli ayarlar mevcut ve bunları açıklamalar kısmında tek tek belirttim. Diğer pek çok kütüphanede olduğu gibi kullanıcının .h uzantılı başlık dosyasına girip çeşitli ayarları yapması gerekecek. Bunu kütüphanelerin çoğunda görsem de bazı gelişmiş kütüphanelerde yapı tabanlı tanımlamalar yapılabiliyor. Arduino kütüphanelerine baktığımızda ise C++'nın sınıf özelliği ile bunun tamamen ortadan kalktığını görüyoruz. Yani bütün parametre ve ayarlar yeni bir sınıf oluşturularak ana programda belirlenmekte. Biz şimdilik bu şekilde devam edelim.

Kullanıcı parametrelerini geçtikten sonra tanımlamaların olduğu kısmı görüyoruz. Buradaki değerleri yazarken tek tek datasheetteki tablolara baktım ve böyle isimlendirip değerleri girdim. Bu yönüyle biraz zahmetli olmakta.

```
// ADC reference values
#define ADC_REF_EXTERNAL 0
#define ADC_REF_VCC      1
#define ADC_REF_INTERNAL 3  // 0b11
```

Mesela yukarıdaki tanımlamaları yazarken aslında aşağıdaki tabloyu birebir aktarmış oldum.

Table 24-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Bu tanımlamalarda en ufak hataya yer yoktur. O yüzden çok dikkatli olunmalı. Mesela şurada da aynı şekilde ön bölücü değerlerini belirledim.

```
// ADC Prescaler Values
#define ADC_DIV2     0
//#define ADC_DIV2     1
#define ADC_DIV4     2
#define ADC_DIV8     3
#define ADC_DIV16    4
#define ADC_DIV32    5
#define ADC_DIV64    6
#define ADC_DIV128   7
```

Bu değerlerin hepsi yukarıda yer alan başka bir tanımlamada konfigürasyon parametresi olarak kullanılıyor. Daha sonrasında ise fonksiyon tanımlamaları yaptığımı görebilirsiniz. Bu fonksiyonların ne işe yaradığını açıklamakla devam edelim.

```
extern void adc_init(void);
```

Bu fonksiyon ile ADC birimi tanımlanır ve başlatılır.

```
extern void adc_deinit();
```

Bu fonksiyon başlatılmış ADC birimini devre dışı bırakır. Güç tasarrufu için kullanılabilir.

```
extern uint16_t adc_read(uint8_t channel);
```

Bu fonksiyonla bir çevirim başlatılır ve bu çevirimden elde edilen okuma değeri geri döndürülür. Serbest çalışma modunda ise ilk değer döndürülür.

```
extern uint16_t adc_read_data();
```

Bu fonksiyon ile sadece ADC veri yazmacındaki değer döndürülür. ADC kendi haline bırakılır. Bu fonksiyonu yazma sebebim serbest çalışma modundayken veya farklı bir tetikleyici kullanılırken elde edilen değeri okumaktır.

```
extern void adc_autotrig(uint8_t mode);
```

Bu fonksiyonda otomatik tetikleyici etkinleştirilir ve tanımlamalarda yer alan modlardan biri seçilir.

```
extern void adc_interrupt(uint8_t status);
```

Bu fonksiyon ile ADC kesmesi etkinleştirilir veya devre dışı bırakılır. Kesme fonksiyonunu ana programda *ISR(ADC_vect)* diye yazarak programa dahil edebilirsiniz. Aynı zamanda *avr/interrupt.h* dosyası da programa dahil edilmelidir.

```
extern void adc_disable_digital(uint8_t pins);
```

Bu fonksiyon güç tasarrufu için analog olarak kullanılan ayakların dijital port özelliğini devre dışı bırakmaktadır.

```
extern void adc_left_adjust();
```

Bu fonksiyonla hesaplanan ADC verisi sola hizalanır ve böyle okunur. Bu şekilde okuma yaparsanız 0-69535 arası bir değer elde edersiniz. Eğer üst yazmacı okursanız 8 bitlik bir çözünürlük elde edersiniz.

```
extern uint16_t adc_read_smooth(uint8_t channel);
```

Bu fonksiyon gürültüyü azaltma ve yumuşak bir okuma yapma adına 10 adet ölçüm alıp bunların ortalamasını bize geri döndürmektedir. Arduino uygulamalarında da sıkça kullanılan bu yöntemi kullanışlı olduğu için kütüphaneye dahil ettim.

```
extern long adc_read_map(uint8_t channel, long out_min, long
out_max);
```

map() fonksiyonunu ADC ile sıkça kullandığımız için hem ADC okuma fonksiyonuna *map()* fonksiyonunu entegre ettim. Bu sayede tek fonksiyonda haritalandırılmış değeri elde edebiliriz.

Şimdi C dosyasını inceleyelim ve bu tanımlamaların nasıl kullanıldığını görelim.

```
void adc_init(void){
	ADCSRA |= (ADC_PRESCALER);
	ADMUX |= ((ADC_REFERENCE) << 6);
	ADCSRA |= (1<<ADEN); // Enable ADC
	ADCSRA |= (1<<ADSC); // Make first conversation and ready adc
}
```

Burada *ADC_PRESCALER* ve *ADC_REFERENCE* tanımlamasının değeri ile adc_init() fonksiyonunun işletildiğini görebilirsiniz. Geri kalanı ile daha önce gördüğünüz adc_init() fonksiyonu ile aynıdır.

```
uint16_t adc_read_smooth(uint8_t channel)
{
	uint16_t adtotal = 0;
	for (int i=0; i<10; i++)
	{
		ADMUX &= 0xF0;
		ADMUX |= channel;
		ADCSRA |=(1<<ADSC);
		while(ADCSRA & (1<<ADSC));
		adtotal += ADCW;
	}
	return adtotal / 10;
}
```

Burada gördüğünüz fonksiyon da *adc_read()* fonksiyonunun biraz geliştirilmiş halidir. Gördüğünüz gibi biraz datasheet okumakla ve birkaç fonksiyon üzerine ilave ederek bir kütüphaneyi kolayca yazmak mümkün. Pek çok hazır kütüphanenin istediğiniz gibi çalışmadığını veya kullandığınız donanıma uyumsuz olduğunu görebilirsiniz. Bazen kendi kütüphanenizi yazmanız, bazen de var olan bir kütüphane üzerinde ciddi değişiklik yapmanız gereklidir. O yüzden hazır kütüphane kullansanız da yeri geldiğinde o kütüphaneyi baştan yazabilecek bir düzeyde olmak gereklidir.

19. Uygulama : Temel UART Uygulaması

Bu uygulama ile mikrodenetleyicinin üç temel iletişim biriminden biri olan USART protokolünü kullanmaya başlıyoruz. Diğer protokoller ise SPI ve I2C protokolleridir. Bu üç protokol ile hem diğer bilgisayarlarla hem de diğer modüllerle iletişim sağlanmaktadır. Günümüzde bu iletişimin önemi daha da artmıştır çünkü mikrodenetleyiciye bağladığımız pek çok entegre ve modül bu iletişim protokollerinden birini kullanmakta ve yazılım vasıtasıyla bunlarla haberleşmekteyiz. Artık LM35'den analog değer okuyup bunu santigrat dereceye çevirmek yerine örneğin MCP9808 sıcaklık algılayıcısından I2C protokolü ile sayısal veri okumaktayız. Hem algılayıcı LM35'e göre daha yüksek doğrulukta hem de hesaplama ve kalibrasyon ile uğraşmadan doğrudan veriyi almamız mümkün. Bu tarz aygıtların pahalı olduğunu da hiç düşünmeyin. Mesela MCP9808 entegresini 0,77 dolara alabiliyorsunuz.

Şimdi uygulamanın kodlarını inceleyerek devam edelim.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#define BAUD 9600
// Baud Oranını Tanımlıyoruz
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)            //UBRR için baud
verisini ayarlıyoruz
void uart_basla(uint32_t baud);
void uart_gonder(uint8_t data);
uint8_t uart_oku();
int main(void)
{
    uart_basla(9600);
    while (1)
    {
            uart_gonder('a');
    }
}



void uart_basla(uint32_t baud){
      uint16_t baudRate=F_CPU/baud/16-1;
      UBRR0H=(baudRate>>8);
      UBRR0L=baudRate;
      UCSR0B|=(1<<RXEN0)|(1<<TXEN0);
      UCSR0C|=(1<<UCSZ01)|(1<<UCSZ00);
}
```

```
void uart_gonder(uint8_t uData){
	while(!(UCSR0A & (1<<UDRE0)));
	UDR0=uData;
}

uint8_t uart_oku(){
	while(!(UCSR0A & (1<<RXC0)));
	return UDR0;
}
```

Burada UART ile alakalı *uart_basla(), uart_gonder()* ve *uart_oku()* fonksiyonları yer almaktadır. Bu fonksiyonlarla çok temel de olsa veri alma ve veri gönderme işlemlerini gerçekleştirebilirsiniz. USART biriminde senkron veri alıp gönderme de mümkün olsa da biz UART olarak yani asenkron olarak kullanmaktayız. Bu birim diğer birimlerde olduğu gibi işlemci saatinden beslendiği için F_CPU değerini baz almaktadır. Eğer F_CPU değeri doğru girilmezse baud oranı yanlış hesaplanacak ve iletişim sağlıksız olacaktır. UART protokolünün kullanılması için gereken birinci şart iki cihazın baud oranlarının aynı olmasıdır.

```
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)
```

Burada datasheette de yer alan baud oranı hesaplama formülünün makro olarak yazıldığını görmekteyiz. Bu formülü datasheette ise şu şekilde görüyoruz.

Table 20-1. Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRRn Value
Asynchronous Normal mode (U2Xn = 0)	$BAUD = \frac{f_{OSC}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{OSC}}{16BAUD} - 1$

Bu hesaplanan baud oranını *uart_basla()* fonksiyonunda ilgili UART birimi yazmaçlarına yüklemekteyiz. Yani UART birimini başlatırken öncelikle saniyede kaç bit gönderildiğini belirliyoruz. Bu işi yapan kodlar aşağıdadır.

```
uint16_t baudRate=F_CPU/baud/16-1;
	UBRR0H=(baudRate>>8);
```

```
    UBRR0L=baudRate;
```

Burada 16 bitlik bir değişkenin 8 bitlik yazmaçlara nasıl yüklendiğini de görebilirsiniz. UBRR0H ve UBRR0L olmak üzere iki ayrı 8 bitlik yazmaca 16 bitlik veri yüklenirken verinin 8 bitlik üst kısmı >> operatörüyle 8 adım sağa kaydırılıp böyle yüklenmektedir. 8-bit mimari üzerinde çalışırken yaşadığımız en büyük eksiklik aslında oldukça dar bir veri alanı ile çalışmaktır. Tüm hafıza hücreleri 8-bit olduğu için özel fonksiyon yazmaçları da 8 bittir. Ayrıca aritmetik işlemler hep 8 bitlik veriler üzerinden yapılmaktadır. Neyse ki C dili kullandığımızdan 16 veya 32 bitlik değişkenler üzerinden işlem yapabilmekteyiz. Derleyici arka planda gerekli işleri bizim yerimize yapmaktadır.

Baud oranı kısmından sonra *uart_basla()* fonksiyonunda diğer ayarlar yapılmakta ve UART birimi (Datasheette USART0 olarak geçer.) etkinleştirilmektedir. Bunun için şu kodlar kullanılır.

```
    UCSR0B|=(1<<RXEN0)|(1<<TXEN0);
    UCSR0C|=(1<<UCSZ01)|(1<<UCSZ00);
```

Burada RXEN0 ve TXEN0 bitleri bir yapılarak UART biriminin hem gönderici hem de alıcı birimi etkinleştirilmektedir. Dilerseniz bazı cihazlarda sadece gönderici, bazılarında ise alıcı olarak kullanabilirsiniz. Çift yönlü iletişim için ikisinin de bir (1) yapılması gereklidir. UCSZ01 ve UCSZ00 bitleri ile de karakter boyutu 8-bit yapılmaktadır.

Bilmeniz gereken bir nokta da UART protokolünde veri bir çerçeve (*frame*) halinde gönderilmektedir ve bunun tek bir standardı yoktur. Uygulamaya göre karakter boyutu 7, 8 veya 9 bit gibi bir boyutta olabilir. Stop biti 1 veya 2 bit olabilir ve bunun yanında bir de eşlik (*parity*) biti olabilir. Bütün bu ayarların iletişimin gerçekleştirildiği iki cihazda da aynı olması gereklidir. Aksi halde bu iki cihaz birbirini anlayamaz. Bu yüzden kullandığınız cihazın bu parametrelerini iyi bilip bu ayarları yapmanız gereklidir. Genellikle 9600 baud oranı, 1 stop biti ve eşlik biti olmadan iletişim gerçekleştirilir. Ama bunun her yerde geçerli olmayacağını biliniz.

Şimdi geriye kalan *uart_gonder()* ve *uart_oku()* fonksiyonlarına bakalım.

```
void uart_gonder(uint8_t uData){
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0=uData;
}
```

Burada UDRE0 bitinin 1 olması beklenmektedir. Bu bit bir bayrak biti olarak UART veri yazmacının boş olup olmadığını bize bildirmektedir. Eğer bu yazmaç doluysa hali hazırda bir iletişim var demektir ve bunun üzerine veri yazmak istemeyiz. Eğer bu yazmaç boş ise UART birimi kullanılabilir durumdadır. Biz bu yazmaca veriyi yüklediğimiz zaman otomatik olarak gönderim işlemi başlayacaktır.

```
uint8_t uart_oku(){
      while(!(UCSR0A & (1<<RXC0)));
      return UDR0;
}
```

Aynı şekilde *uart_oku()* fonksiyonunda da RXC0 bitinin 1 olması beklenmektedir. Bu bit veri alımının tamamlandığını ve verinin UDR0 yani UART veri yazmacından okunabileceğinin haberini verir. Veri gelene kadar programın sonsuz döngüye girdiğini unutmayın. Bunu önlemek için genellikle UART kesmesi kullanılır. Biz en basit haliyle göstersek de bu yöntem verimli değildir.
Programı yüklediğinizde Arduino kartı USB-Seri çevirici vasıtasıyla bilgisayara sürekli olarak 'a' harfini göndermeye başlayacaktır. Bunu okumak için Arduino derleyicisinin "*Serial monitör*" özelliğini kullanmak en pratiği olacaktır. Bunun yanında **Putty** programı da bu tarz işlemler için kullanılabilir.

20. Uygulama : UART ve Düğme Uygulaması

Bu uygulamada dijital giriş ve çıkış portları ile UART protokolünü beraber kullanacağız. Bu uygulama oldukça basittir. Kullanıcı belli düğmelere bastığında hangi düğmeye basıldığı anlaşılır bir biçimde bilgisayara mesaj olarak yollanmakta ve bu ekranda gösterilmektedir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#define BAUD 9600
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)
void uart_basla(uint32_t baud);
void uart_gonder(uint8_t data);
void uart_string(const char *s );
uint8_t uart_oku();
int main(void)
{

uart_basla(9600);
```

```
DDRD &=~ ( (1<<PD2) | (1<<PD3) | (1<<PD4) );
//PORTD |= ((1<<PD2) | (1<<PD3) | (1<<PD4));
PORTD = 0b11100;
_delay_ms(100);
while (1)
{
if (!(PIND & (1<<PD2)))
uart_string("1. Dugme Basili \n");
if (!(PIND & (1<<PD3)))
uart_string("2. Dugme Basili \n");
if (!(PIND & (1<<PD4)))
uart_string("3. Dugme Basili \n");

}
}



void uart_basla(uint32_t baud){
uint16_t baudRate=F_CPU/baud/16-1;
UBRR0H=(baudRate>>8);
UBRR0L=baudRate;
UCSR0B|=(1<<RXEN0)|(1<<TXEN0);
UCSR0C|=(1<<UCSZ01)|(1<<UCSZ00);
}

void uart_gonder(uint8_t uData){
while(!(UCSR0A & (1<<UDRE0)));
UDR0=uData;
}

uint8_t uart_oku(){
while(!(UCSR0A & (1<<RXC0)));
return UDR0;
}

void uart_string(const char *s )
{
while (*s)
uart_gonder(*s++);

}
```

Bu uygulamayı yaparken yaşadığım bir sorundan bahsedeyim. Ana program döngüsünün hemen öncesinde *_delay_ms(100)* fonksiyonunu görüp bunun işlevinin ne olduğunu merak etmiş olmalısınız. Bu beklemeyi koymadan önce

karta güç verince hemen ana program çalışmaya başlıyor ama dijital okuma kısmında kararsızlıklar yaşanıyordu. Yani 1. Düğmeye basılmadığı halde program bunu basılmış gibi algılıyordu. Program tamamen doğru olsa da beslemeden kaynaklı bu tür kararsızlıkları daha öncesinde de yaşamıştım. O yüzden mikrodenetleyici açıldıktan sonra ana programa geçmeden önce 100 mili saniyelik bir *delay* fonksiyonu koydum ve program düzeldi. Normalde bu durumu kitaplarda, datasheette görmüş değilim. Sorunu çözdükten sonra çok araştırma gereği duymadım, bu ilginç sorunun kaynağını siz araştırabilir veya sizde de bu sorunun olup olmadığını *delay* fonksiyonunu kaldırarak deneyebilirsiniz.
Burada farklı bir fonksiyon olarak karakter dizisi gönderme fonksiyonunu görmekteyiz. Bunu biraz acemice olarak for döngüsü içinde ve '\0' karakterini denetleyecek şekilde yapabilir ve indeks erişim operatörüyle ([ve]) karakterleri tek tek yazdırabilirdik. Ama burada C'nin en kuvvetli özelliklerinden biri olan işaretçilerle çok daha etkili gerçekleştiriyoruz. Şimdi fonksiyonu derinlemesine inceleyelim.

```
void uart_string(const char *s )
 {
 while (*s)
 uart_gonder(*s++);

 }
```

Burada const char *s diyerek karakter dizisi sabitini argüman olarak alıyoruz. Sonrasında while döngüsü içerisinde *s diyerek s karakter dizisinin ilk karakterini denetliyoruz. Bu denetlemeyle karakterin sıfır olup olmadığını denetlemiş oluyoruz. Sıfırdan farklı bir değilse TRUE ifade edecektir ve döngü çalışmaya devam edecektir. Sonrasında ise *uart_gonder(*s…* diyerek s dizisinin işaret ettiği karakter değerini göndermekteyiz. Sonrasında ise *s++ yazdığı için ++ operatörü çalışacak ve karakter dizisinin işaret ettiği adres değerini bir artıracaktır. Yani bir sonraki değeri işaret eder olacaktır. Burada ++ operatörünün önde veya sonda olmasının büyük önemi vardır. Sonra olan ++ operatörü en son olarak çalışmaktadır. Yani fonksiyona değer gider, fonksiyon çalıştırılır ve en sona s dizisinin değeri (yani adres değeri) bir artırılır. Dizilerin aslında birer işaretçi olduğunu ve dizi adının dizinin ilk elemanının adresini verdiğini unutmayın. Bunu "*Temel C Programlama*" çalışmamda güzelce anlatmıştım.

21. Uygulama : Kesmeler ile UART Kullanımı

Bu uygulama ile UART kesmelerinin nasıl çalıştığını basit bir şekilde göstereceğim. Uygulamada D portunun 5. ve 6. Ayaklarına birer LED bağlı olup 5. Ayak veri alındığında, 6. Ayak ise veri geldiğinde yanıp sönecektir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>
#define BAUD 9600
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)
void uart_basla(uint32_t baud);
void uart_gonder(uint8_t data);
void uart_string(const char *s );
uint8_t uart_oku();
int main(void)
{

	uart_basla(9600);
	UCSR0B |= (1<<RXCIE0);
	UCSR0B |= (1<<TXCIE0);
	sei();
	DDRD &=~ ( (1<<PD2) | (1<<PD3) | (1<<PD4) );
	// PORTD |= ((1<<PD2) | (1<<PD3) | (1<<PD4));
	PORTD = 0b11100;
	DDRD |= (1<<PD5);
	DDRD |= (1<<PD6);
	_delay_ms(100); // MCU Kendine Gelsin
	while (1)
	{
		if (!(PIND & (1<<PD2)))
		uart_string("1. Dugme Basili \n");
		if (!(PIND & (1<<PD3)))
		uart_string("2. Dugme Basili \n");
		if (!(PIND & (1<<PD4)))
		uart_string("3. Dugme Basili \n");

	}
}



void uart_basla(uint32_t baud){
	uint16_t baudRate=F_CPU/baud/16-1;
	UBRR0H=(baudRate>>8);
```

```
	UBRR0L=baudRate;
	UCSR0B|=(1<<RXEN0)|(1<<TXEN0);
	UCSR0C|=(1<<UCSZ01)|(1<<UCSZ00);
}

void uart_gonder(uint8_t uData){
	while(!(UCSR0A & (1<<UDRE0)));
	UDR0=uData;
}

uint8_t uart_oku(){
	while(!(UCSR0A & (1<<RXC0)));
	return UDR0;
}

void uart_string(const char *s )
{
	while (*s)
	uart_gonder(*s++);

}

ISR (USART_RX_vect)
	{
		PORTD ^= (1<<PD5);
	}

ISR (USART_TX_vect)
	{
		PORTD ^= (1<<PD6);
	}
```

Diğer uygulamalardan farklı olarak ISR adında fonksiyonların olduğunu ve bu fonksiyonların parametrelerinde USART_RX_vect gibi değerlerin yer aldığını görüyoruz. AVR'de kesme kullanmak bu kadar basittir. Öncelikle *#include <avr/interrupt.h>* diyerek kesme başlık dosyasını programa dahil ediyoruz. İlk açılışta kesmeler kapalı olduğu için *sei()* fonksiyonu ile kesmeleri etkinleştiriyoruz. Ardından ilgili birimin kesmesini yine ilgili birimin yazmacından etkinleştiriyoruz. Burada iki kesme söz konusudur. Birincisi veri alımı tamamlandığında yürütülecek kesme, öteki ise veri gönderimi tamamlandığında yürütülecek kesmedir. İkisini etkinleştirmek için şu kodlar yazılmıştır.

```
	UCSR0B |= (1<<RXCIE0);
	UCSR0B |= (1<<TXCIE0);
```

Sonrasında ise ISR adında bir fonksiyon tanımlayıp parantez içerisine kullanacağımız kesmenin adını yazıyoruz. Bu kesme vektör adlarını en kolay kullandığınız denetleyicinin başlık dosyasında bulabilirsiniz. *AVRLibC* kılavuzunda da yer alsa da proje üzerinde çalışırken bu şekilde bulabilirsiniz. Mesela kullandığımız ATmega328p denetleyicisinde *iom328p.h* dosyasına giriyoruz ve ilgili yerde vektör adlarını ve bunların hangi kesmeler olduğunu görebiliyoruz.

```
#define USART_RX_vect_num 18
#define USART_RX_vect     _VECTOR(18)  /* USART Rx Complete */

#define USART_UDRE_vect_num   19
#define USART_UDRE_vect   _VECTOR(19)  /* USART, Data Register Empty */

#define USART_TX_vect_num 20
#define USART_TX_vect     _VECTOR(20)  /* USART Tx Complete */

#define ADC_vect_num      21
#define ADC_vect          _VECTOR(21)  /* ADC Conversion Complete */
```

Şimdi bir kesme fonksiyonunu inceleyelim.

```
ISR (USART_RX_vect)
     {
          PORTD ^= (1<<PD5);
     }
```

USART_RX_vect diyerek UART biriminde veri alımı tamamlandığında gerçekleşecek kesmeyi belirtiyoruz. Bu kesme her gerçekleştiğinde süslü parantez içindeki kodlar yürütülecek, yani D portunun 5. Ayağına bağlı olan LED yanıp sönecektir. Arduino'nun seri port ekranını açıp buradan uzun bir metin yolladığınızda çok hızlı da olsa her karakterde yanıp sönme gerçekleşecektir. Kesme fonksiyonları içerisine asla bekleme fonksiyonu koyulmamalı ve bu fonksiyonlar oldukça kısa olup mikrodenetleyiciyi meşgul etmemelidir. Eğer kesme fonksiyonu içinde bir kesme gerçekleşirse mikrodenetleyici bunu kaçırabilir. Kesme fonksiyonuna geçildiği zaman donanım otomatik olarak kesmeleri kapatmaktadır. İç içe kümelenmiş kesmeler gelişmiş denetleyicilerde söz konusu olsa da burada kesmelerin teker teker yürütülmesi esastır.

22. Uygulama : Python ile Seri İletişim

Bu uygulamanın bilgisayar tarafında *Python* dili ve bu dil ile kullanabileceğimiz kullanıcı arayüzü kütüphaneleri ve *PySerial* kütüphanesi ile mikrodenetleyici ile etkileşime geçeceğiz. Mikrodenetleyici tarafında ise önceki uygulamada edindiğimiz tecrübeler ile UART birimini kullanacağız. Gömülü sistemlerde pek çok zaman bir bilgisayar ile iletişime geçmemiz gerekebilir. Bu tarz pek çok uygulamayı görebilirsiniz. Genelde bu önceleri Visual Basic, günümüzde de *Visual C#* ile yapılsa da ben burada *Python* ile arayüz uygulaması yapmayı tercih ettim. Bana göre kolayca öğrenebileceğiniz ve pek çok kütüphanesi ve kullanım alanı olan *Python* dilini gömülü sistem geliştiricisi olarak öğrenmeniz oldukça iyi olacaktır. Öncelikle mikrodenetleyici programını inceleyelim.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>
#define BAUD 9600
#define BAUDRATE ((F_CPU)/(BAUD*16UL)-1)
void uart_basla(uint32_t baud);
void uart_gonder(uint8_t data);
void uart_string(const char *s );
volatile uint8_t a;
int main(void)
{

	uart_basla(9600);

	sei();
	DDRD &=~ ( (1<<PD2) | (1<<PD3) | (1<<PD4) );
	// PORTD |= ((1<<PD2) | (1<<PD3) | (1<<PD4));
	PORTD = 0b11100;
	DDRD |= (1<<PD5);
	DDRD |= (1<<PD6);
	_delay_ms(100); // MCU Kendine Gelsin
	while (1)
	{
	if (a=='1')
	PORTD |= (1<<5);
	if (a=='2')
	PORTD &= ~(1<<5);
	if (a=='3')
	PORTD |= (1<<6);
	if (a=='4')
```

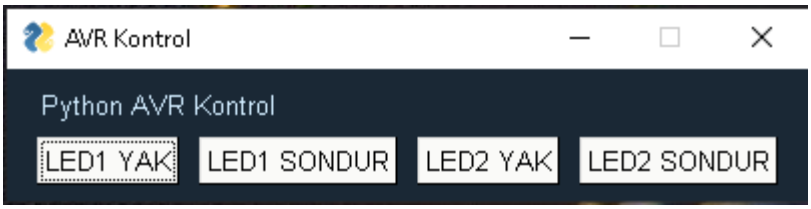
```
        PORTD &= ~(1<<6);
        }
}



void uart_basla(uint32_t baud){
        uint16_t baudRate=F_CPU/baud/16-1;
        UBRR0H=(baudRate>>8);
        UBRR0L=baudRate;
        UCSR0B|=(1<<RXEN0)|(1<<TXEN0);
        UCSR0C|=(1<<UCSZ01)|(1<<UCSZ00);
        UCSR0B |= (1<<RXCIE0);
}



ISR (USART_RX_vect)
{
        while(!(UCSR0A & (1<<RXC0)));
        a = UDR0;
}
```

Burada artık veri alımını daha önceki uygulamada öğrendiğimiz kesmeler ile yapmaktayız. Böylelikle mikrodenetleyici boş yere meşgul olmayacaktır. Veri alımı tamamlandığında a = UDR0; diyerek global a değişkenine okunan değeri atıyoruz. Sonrasında ise ana program akışında sırayla kontrol yapıları oluşturarak '1', '2', '3' ve '4' değerini alıp almadığına göre çeşitli yakıp söndürme işlemlerini yapmaktayız. Önceki uygulamanın devresi üzerine yaptığımız için bir değişiklik yapmanıza gerek yok. Yine D portunun 5 ve 6 numaralı ayaklarına birer LED bağlanacak.
Buradan gördüğümüz kadarıyla mikrodenetleyicinin anladığı dil '1', '2', '3' ve '4' karakterleridir. Bilgisayar mikrodenetleyici ile konuşması gerekiyorsa bu "kelimeleri" kullanmak zorundadır. O yüzden bilgisayar programı yazarken protokol parametreleri hariç bize gerekli olan tek bilgi budur, bunu bir kenara not ediyoruz. Şimdi *Python* kodlarını inceleyelim.

```
import PySimpleGUI as sg
import serial

ser = serial.Serial('COM4', 9600, timeout=0, parity=serial.PARITY_NO
NE, stopbits = serial.STOPBITS_ONE , bytesize = serial.EIGHTBITS, rt
scts=0)

sg.theme('DarkBlue')

layout = [  [sg.Text('Python AVR Kontrol')],
            [sg.Button('LED1 YAK', key=1), sg.Button('LED1 SONDUR',
key=2), sg.Button('LED2 YAK', key=3), sg.Button('LED2 SONDUR', key=4
) ] ]

window = sg.Window('AVR Kontrol', layout)
while True:
    a = window.read()
    okunan = a[0]
    okunan = str(okunan)
    print(okunan)
    ser.write(okunan.encode('Ascii'))
window.close()
```

Burada *import* anahtar kelimesi ile *PySimpleGUI* ve *PySerial* kütüphanelerini programa dahil ettik. *PySimpleGUI* kütüphanesi çok basit bir şekilde ara yüz uygulamaları tasarlamamızı sağlayacaktır. *PySerial* kütüphanesi ise seri port üzerinden Arduino ile iletişime geçmemizi sağlayacaktır. Her ne kadar USB üzerinden bağlansa da bilgisayar bunu seri port olarak görmektedir. USB uygulamaları içinse *PyUSB* kütüphanesi mevcuttur. Bu iki kütüphane gömülü sistemler üzerinde uygulama yapacaklar için çok önemlidir.
Sonrasında ise *serial.Serial(…* diyerek bir nesne oluşturuyor ve buraya seri port parametrelerini yazıyoruz. Aygıt yöneticisinden öğrendiğimiz COM numarasını buraya COM4 olarak yazdık. Sizde farklı ise bunu değiştirmeniz gereklidir. Sonrasında eşlik biti, stop biti sayısı ve kaç bitlik veri olduğunu yazıyoruz. Bizim AVR tarafında datasheete bakıp yazdığımız parametreleri aynı olacak şekilde buraya da yazmaktayız. İkisi aynı olmazsa uygulama çalışmayacaktır.

Sonrasında *ser.write()* fonksiyonu ile ASCII kodlamasını yaptığımız veriyi seri port üzerinden göndermekteyiz. Arayüz programının ekranı şu şekilde olmaktadır. *Python* yükledikten sonra pip install pysimplegui ve pip install pyserial komutları ile iki kütüphaneyi yüklemeyi unutmayınız.



Eğer *Python* öğrenmeye karar verdiyseniz Facebook grubumdaki paylaşımları ve yaptığım kaynak toplama çalışmasını anlattığım makaleyi okumakla başlayabilirsiniz.

https://www.facebook.com/groups/1233336523490761/post_tags/?post_tag_id=1615172005307209&ref=rhc_tag

https://medium.com/@gokhandokmetas/python-%C3%B6%C4%9Frenme-kaynaklar%C4%B1-toplama-%C3%A7al%C4%B1%C5%9Fmas%C4%B1-b17e0f7f3d49

23. Uygulama : UART ile Karakter Dizisi Okuma

Daha öncesinde karakter dizisi göndermeyi, tek karakter okumayı ve göndermeyi göstermiştim. Şimdi ise içlerinde en zor işlem olan karakter dizisi okumayı anlatacağım. Çoğu zaman UART'dan gönderilen ve alınan veriler tek bir karakter halinde olmak yerine karakter dizisi halinde olmaktadır. Bu karakter dizisi üzerinde işlem yapabilmek için de öncelikle bunu sağlıklı bir şekilde kaydetmek gereklidir. Bunun için karakter LCD kullanırken olduğu gibi Peter Fluery'in UART kütüphanesini kullanacağım. Kütüphane kullanmamın sebebi öncelikle bu karakter dizisini okumak için bir tempon bellek oluşturmak gereklidir. Her karakter geldiğinde tampon belleğe bu veri yazılacak ve biz de bu tampon bellek üzerinden sırayla karakterleri okuyacağız. Bunu yapmak biraz zahmetli olduğu için sıfırdan yazmak yerine hazır kütüphane kullandım. Hazır kütüphanede ise tampon belleğin boş olup olmadığını kontrol eden bir fonksiyon yoktu, bunun için Arduino'daki *Serial.available()* fonksiyonunu bu kütüphaneye uyarladım ve yerleştirdim. Orijinal kütüphanede bulamayacağınız için proje ile beraber gelen kütüphane dosyalarını kullanmanız gerekecek.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include "uart.h"
#include <string.h>
```

```
int main(void)
{
    sei();
    uart_init(UART_BAUD_SELECT(9600,F_CPU));
    _delay_ms(100);
    uart_puts("UART STRING OKUMA \n");

    while (1)
    {
      if (uart_available() > 0)
            {
            int i = 0;
          char buf [20];
            while(1)
                  {

                  buf[i] = uart_getc();
                  if (buf[i] == '\n')
                  break;
                  if (buf[i] == '\0')
                  break;
                  if (buf[i] == '\r')
                  break;
                  i++;
                  if (i>19)
                  i = 0;
                  }
            uart_puts(buf);
            }

      }

}
```

Burada if karar yapısının içinde uart_available() > 0 diyerek tampon bellekte veri olup olmadığı kontrol edilmektedir. Tampon bellek boşsa boşuna okuma yapmaya gerek yoktur. Program akışında bu denetlenir ve veri olduğu zaman bir döngüye girilir ve sırayla okunan karakterler buf adı verilen boş karakter dizisinin içerisine yerleştirilir. Newline (\n), Return(\r) ve karakter dizisi bitimi (\0) karakterleri denetlenerek verinin bitip bitmediği anlaşılır. *Uart_puts(buf)* fonksiyonu ile de okunan veri tekrar seri port ekranına yazdırılacaktır. Bu verinin okunup okunmadığını öğrenmek için kullanılmıştır. Program çalıştığında yazdığınız karakter dizisi ekranda bir daha çıkıyorsa program doğru çalışıyor demektir. Bu kodları siz istediğiniz projede lazım olunca kullanabilirsiniz. Proje yaparken muhakkak gerekli olduğu bir yer çıkacaktır.

24. Uygulama : UART Kütüphanesi Örneği

Yukarıda karakter dizisi okuması yaptığımız UART kütüphanesini pek çok uygulamada rahatça kullanabilirsiniz. AVR'de UART protokolünü kullanmak için en sorunsuz ve yaygın kütüphane budur. Ben yine bu projede kütüphaneye büyük işlev kattığı için *uart_available()* fonksiyonunu ekledim. Kütüphane dosyalarını projede bulabilirsiniz.

```
#include <avr/io.h>
#define  F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>
#include "uart.h"
int main(void)
{
      sei();
      uart_init(UART_BAUD_SELECT(9600,F_CPU));
      _delay_ms(100);
      uart_puts("UART DENEME PROGRAMI \n");
    while (1)
    {
            if (uart_available() > 0)
            uart_putc(uart_getc());
    }
}
```

25. Uygulama: T0 Zamanlayıcısı Uygulaması

Zamanlayıcı/Sayıcılar (*Timer/Counter*) hemen hemen bütün mikrodenetleyicilerde gördüğümüz, en gerekli çevre birimlerinden biridir. Giriş seviyesi mikrodenetleyicilere bile baktığımızda ADC, USART, SPI gibi birimlerden önce en azından 8 bitlik bir adet zamanlayıcının yer aldığını görürüz. Bu zamanlayıcılar zamana bağlı uygulamalarda kullanıldığı gibi pals genişliği (*pulse width*) okuma, frekans okuma, değişken frekans üretme ve PWM sinyali üretme gibi çok mühim uygulamalarda kullanılmaktadır. Özellikle analog kısımla ilgilenenler için pek çok devrede sinyal üretmek ve okumamız gerekecektir. Bunu yazılım ile hassas bir şekilde yapabilmek için zamanlayıcıları kullanmayı iyi bilmemiz gereklidir.

Zamanlayıcılar temelde basit olsa da fazlaca özellik barındırmalarından dolayı mikrodenetleyicilerdeki en karmaşık çevre birimlerinden biridir. Herhangi bir

denetleyicinin datasheetini açtığınızda yaklaşık %30-40'lık bir kısımda bu çevre biriminin anlatıldığını görebilirsiniz. Sadece AVR denetleyicilerde değil 32 bitlik STM32 denetleyicilerde de 1000 sayfayı geçen datasheetlerde yukarıda verdiğim oranda bu çevre birimleri anlatılmaktadır. AVR mikrodenetleyicilerde zamanlayıcılar 32 bitlik mikrodenetleyicilerdeki kadar gelişmiş olmasa da pek çok işte yeterli olacak niteliktedir. Zamanlayıcı uygulamalarını eksiksiz yapmaya ve bütün özelliklerini uygulamada göstermeye gayret ettim. Sizden de bu uygulamaları büyük bir dikkatle takip etmenizi rica ediyorum.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>

int main(void)
{
      DDRC |= (1<<PC5);
     TCCR0B |= (1<<CS00);
      TIMSK0 |= (1<<TOIE0);
      sei();
    while (1)
    {
    }
}

ISR (TIMER0_OVF_vect)
{
      PORTC ^= (1<<PC5);
}
```

Burada C portunun 5 numaralı ayağını osiloskopa veya frekansmetreye bağlayarak ölçüm yapmamız gereklidir. Çünkü 8 bitlik TC0 zamanlayıcısı her ne kadar kendisini besleyen saat frekansını (*Fcpu*) ön bölücüler ile bölsek de yine de oldukça kısa bir zamanda dolacaktır. Özellikle zamanlayıcılar ile çalışırken osiloskop üzerinden sinyali görmeniz gerekecektir. Şimdi zamanlayıcıyı nasıl çalıştırdığımıza bakalım.

```
TCCR0B |= (1<<CS00);
```

Burada TCCR0B yazmacındaki CS00 ayağını bir (1) yaparak zamanlayıcıyı başlatmış oluyoruz. Ön bölücü değerine herhangi bir değer yazdığınız anda zamanlayıcı çalışmaya başlamış oluyor. Yani hem ön bölücü ayarı hem de etkinleştirmeyi buradan yapıyorsunuz. Hepsi sıfır (0) olduğu zaman ise zamanlayıcı durmuş oluyor. Program akışında ayarlarda bazı değişiklikleri yapmanız gerektiğinde veya güç tasarrufu için zamanlayıcıyı durdurmanız

gerekebilir. Sadece CS00 biti bir (1) yapıldığı için zamanlayıcı saat kaynağı olarak işlemci saatini doğrudan alacaktır. 8 bitlik bir sayaca sahip olduğu için işlemci saatinin her çeviriminde birer birer artacaktır. 255 değeri ise üst sınır olduğu için bu değere ulaştığı zaman tekrar sıfırlanacaktır. 16MHz'de 255 adımda bir sıfırlandığı için 16000000/255 yani 62.745 kere taşma oluşacaktır.

```
TIMSK0 |= (1<<TOIE0);
```

Bu komutta ise taşma kesmesini etkinleştirdim. Bu sayede her taşma gerçekleştiğinde program akışı kesme vektörüne gidecektir.

```
ISR (TIMER0_OVF_vect)
{
      PORTC ^= (1<<PC5);
}
```

sei() fonksiyonu ile genel kesmeleri etkinleştirdikten sonra ISR fonksiyonunun içine TIMER0_OVF_vect yazarak T0 zamanlayıcısının taşma kesmesini seçmiş oldum. Bundan sonrasında ise C portunun 5 numaralı ayağını her taşmada *toggle* yani yakıp söndürme işini yapmaktayım. Bir zamanlayıcının mümkün olabilecek en basit kullanımı bu şekildedir.

Not: Eğer elinizde hiç osiloskop yok ve bütçeniz kısıtlı ise Çin sitelerinden alacağınız 5 dolarlık lojik analizör ile de bu sinyalleri gözlemleme imkânı elde edebilirsiniz. Sonuçta bunlar dijital sinyal olduğu için osiloskop ekranında gördüğünüz görüntü de lojik analizördeki görüntü gibi olacaktır.

26. Uygulama : Ön Bölücüler ile Zamanlayıcı Kullanımı

AVR mikrodenetleyicilerin bütün zamanlayıcılarında ön bölücüler mevcuttur. Bu ön bölücüler (*prescaler*) zamanlayıcıyı besleyen saat sinyalini bölüp zamanlayıcının istenilen hızda çalışmasını sağlamaktadır. Pek çok uygulama için zamanlayıcılar aşırı hızlı olabilir, bu durumda yavaşlatılması gereklidir. Bu uygulamada da yukarıda en basit haliyle kullandığımız zamanlayıcıyı ön bölücü ile kullanacağız.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>

int main(void)
{
      DDRC |= (1<<PC5);
    TCCR0B |= (1<<CS00);
      TCCR0B |= (1<<CS02);
      TIMSK0 |= (1<<TOIE0);
      sei();
      // osiloskop ve hesap makinesinden ölçülecek.
    while (1)
    {
    }
}

ISR (TIMER0_OVF_vect)
{
      PORTC ^= (1<<PC5);
}
```

Yukarıdaki uygulamadan farklı olarak burada bir de CS02 bitinin bir (1) yapıldığını görebilirsiniz. Bu sayede datasheette yer alan ön bölücü tablosuna göre saat sinyali 1024'e bölünecektir.

Table 15-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Bu tabloda ön bölücü değerlerini seçtiğimiz gibi saat kaynağını harici bir saat kaynağı olarak da seçme imkânımız vardır. Bu sayede T0 ayağına uygulanan saat sinyalleri ile zamanlayıcıları sayıcı olarak kullanabilmekteyiz. İki uygulamayı çalıştırıp frekansını ölçtüğünüzde ciddi bir değişikliğin olduğunu görebilirsiniz. Bu uygulama ile TC0 zamanlayıcısını en yavaş haliyle çalıştırdık.

Lütfen uygulamaları yaparken "C ile AVR Programlama" yazılarımı ve datasheeti takip ediniz.

27. Uygulama : Karşılaştırma Değeri ile Zamanlayıcıları Kullanmak

Yukarıda zamanlayıcıyı kullandığımız uygulamalarda zamanlayıcı 0'dan başlayıp 8 bitlik alanın son değeri olan 255'e kadar sayıp sonra kesmeye gidip, kendini sıfırlıyordu. Biz kesin bir sürede çalışmasını istediğimiz zaman frekansın kaça bölüneceğini ve zamanlayıcının kaça kadar sayacağını belirlememiz gerekecek. Yani bir yerde 120'ye kadar say, öteki yerde 50'ye kadar say diyeceğiz. Kaça kadar sayması gerektiğini de formül ile hesaplıyoruz. Bu formülün nasıl kullanıldığını "*C ile AVR Programlama*" yazılarımda ve datasheette bulabilirsiniz. Bu hesaplama ile elde ettiğimiz değeri ise OCR yazmaçlarına (*Output compare register*) yazmamız gereklidir. Yalnız 8 bitlik zamanlayıcıların belli bir sınırı olduğunu ve belli bir aralıkta ve çözünürlükte istenilen beklemeyi sağlayacağını unutmayınız. Ön bölücülerle bu aralığı ileri geri götürme imkânımız olsa da 8 bitlik bir zamanlayıcı oldukça sınırlı bir aralık vermektedir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>
```

```
int main(void)
{
    DDRC |= (1<<PC5);
    TCCR0B |= (1<<CS00);
      TCCR0B |= (1<<CS02);
      TCCR0A |= (1<<WGM01); // CTC MOD
      OCR0A = 155;

      TIMSK0 |= (1<<OCIE0A);
      sei();
    while (1)
    {
    }
}

ISR (TIMER0_COMPA_vect)
{
      PORTC ^= (1<<PC5);
}
```

Burada TCCR0A yazmacındaki WGM01 bitini bir (1) yaparak zamanlayıcıyı CTC (*Clear Timer on Compare Match*) moduna aldım. Bu modda zamanlayıcı 0'dan itibaren ön bölücüyle bölünen saat sinyalinden beslenerek saymaya başlar ve OCR0A yazmacında belirlediğimiz değere (Burada 155) kadar saymaya devam eder. Değere ulaştığında ise zamanlayıcı sıfırlanır ve kesmeye gidilir. Zamanlayıcı tekrar sıfırdan itibaren saymaya başlayacak ve bizim belirlediğimiz değerde tekrar kesmeye gidecektir.

28. Uygulama – TC0 Zamanlayıcısı ile Frekans Üretme Uygulaması

Daha öncesinde PORT yazmaçları üzerinden dijital çıkış ile bir şekilde frekans üretiyorduk. Aslında bu şekilde frekans üretmenin doğru olmadığını belirtmek gerekir. Zamanlayıcının yanı sıra işlemciyi meşgul ederek zamanlamada ufak da olsa bir hata elde etmekteyiz. Aynı zamanda zamanlayıcıda işlemciyi hiç meşgul etmeden frekans üretme özelliği dururken bizim kesmeler ve portlar üzerinden bunu yapmamız anlamsızdır. Bu uygulamada işlemci hiç meşgul edilmeden tamamen zamanlayıcılar kullanılarak bu işlemin nasıl yapıldığını göstereceğim.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>
#include <util/delay.h>
int main(void)
{
    DDRD |= (1<<PD6);
    TCCR0B |= (1<<CS00);
      TCCR0B |= (1<<CS02);
      TCCR0A |= (1<<WGM01); // CTC MOD
      TCCR0A |= (1<<COM0A0);
    while (1)
    {
      for (int i=0; i<256; i++)
      {
      OCR0A = i;
      _delay_ms(20);
      }

    }
}
```

Burada işin içine biraz görsellik katmak için for döngüsü içerisinde belli bir bekleme ile üretilen frekansın değerini değiştirdim. Siz osiloskopta bunu incelerken frekanstaki değişmeyi fark edeceksiniz. Yukarıdaki uygulama ile aynı olsa da burada COM0A0 biti bir (1) yapılmaktadır. Arduino UNO ayak haritasına baktığımızda da göreceğimiz üzere COM0A0 ayağı D portunun 6. ayağına denk gelmektedir. O yüzden en başta D portunun 6. Ayağını çıkış olarak ayarlıyoruz. AVR mikrodenetleyicilerde zamanlayıcılardan çıkış almak için bunu yapmamız şarttır. Neden COM0A0 bitini bir (1) yaptığımızı şu tablodan anlayabiliriz.

Table 15-5. Compare Output Mode, non-PWM Mode

COM0B1	COM0B0	Description
0	0	Normal port operation, OC0B disconnected.
0	1	Toggle OC0B on Compare Match
1	0	Clear OC0B on Compare Match
1	1	Set OC0B on Compare Match

Burada bizim kare dalga elde etmemiz için aç/kapa işlemi yapmamız gereklidir. *Set* veya *Clear* işinde bu bir defaya mahsus olacağı için herhangi bir frekans elde edemeyiz.

Siz bu uygulamada CS00, CS01 ve CS02 bitlerini değiştirerek ön bölücü değeriyle oynayıp farklı frekans aralıklarında sinyaller elde edebilirsiniz. 8 bit zamanlayıcı olduğundan kısıtlı bir aralıkta sinyal elde ettiğinizi siz de gözlemleyebilirsiniz.

29. Uygulama – PWM Sinyali Üretme Uygulaması

8 bitlik zamanlayıcılar her ne kadar frekans üretme ve zamanlama işlemlerinde kısıtlı olsa da PWM sinyali üretme noktasında oldukça yeterli olmaktadır. 8 bitlik zamanlayıcılar ile 8 bit çözünürlükte PWM sinyali elde edebiliriz. Bu sinyal de servo motor sürme gibi uygulamalarda yeterli olmaktadır. İsterseniz bu uygulama ile bir servo motoru sürebilirsiniz. Ama burada osiloskopa bağlayıp sinyali incelemek yeterli olacaktır.

```
#include <avr/io.h>
#define  F_CPU 16000000UL
#include <util/delay.h>

int main(void)
{
      DDRD |= (1 << PD6);
     OCR0A = 200;
      TCCR0A |= (1 << COM0A1);
      TCCR0A |= (1 << WGM00);
      TCCR0B |= (1 << CS01);
      while (1)
      {

      }

}
```

Görev döngüsünü ayarlamak için OCR0A yazmacına 0-255 arası değer girmek gereklidir. Bu değeri programda başlangıçta sabit bir değer olarak girdim. Eğer program akışında değiştirilmek isteniyorsa öncelikle zamanlayıcı durdurulmalı, kontrol yazmaçları sıfırlanmalı ve sonrasında değer güncellenip eski ayarlarla zamanlayıcı tekrar başlatılmalıdır. PWM çıkışı alırken OCR0A yazmacının değerini güncellemek alınan sinyal üzerinde etkili olmayacaktır.

Burada WGM00 bitini ayarlayarak zamanlayıcıyı PWM moduna sokuyoruz. Bu bit ayarlarına göre zamanlayıcı çeşitli modlarda çalışmaktadır. Bu modları yine datasheetteki tablodan görebiliriz.

Table 15-8. Waveform Generation Mode Bit Description

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on (1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

8 bit zamanlayıcı olduğu için çok fazla bir konfigürasyon göremiyoruz. 16 bit zamanlayıcıyı incelediğimizde bundan çok daha fazla ayarın olduğunu görebiliriz.

30. Uygulama – Ayarlanabilir PWM Uygulaması

Bu uygulamayı bir önceki uygulamanın üzerine yaptım. Dediğim gibi OCR değerini güncelleyerek görev döngüsünü ayarlamak mümkündür. Bunu zamanlayıcı yazmaçlarını sıfırlayıp sonrasında değeri güncelleyip ardından zamanlayıcıyı tekrar başlayarak yapmak gereklidir. Bu programı yükleyip osiloskopta D portunun 6. Ayağını ölçtüğünüzde görev döngüsünün sıfırdan %100'e artarak ilerlediğini göreceksiniz.

```
#include <avr/io.h>
#define  F_CPU 16000000UL
#include <util/delay.h>

int main(void)
{
      DDRD |= (1 << PD6);
    OCR0A = 200;
      TCCR0A |= (1 << COM0A1);
```

```
      TCCR0A |= (1 << WGM00);
      TCCR0B |= (1 << CS01);
      while (1)
      {
    for (int i = 0 ; i < 256 ; i++)
      {
      TCCR0A = 0;
      TCCR0B = 0;
      OCR0A = i;
      TCCR0A |= (1 << COM0A1);
      TCCR0A |= (1 << WGM00);
      TCCR0B |= (1 << CS01);
      _delay_ms(20);

      }
      }

}
```

Bu uygulamalar üzerinde gerekmedikçe çok fazla açıklama yapmak istemiyorum. Zaten "*C ile AVR Programlama*" yazılarımda bunu teorik tarafıyla anlattım. Bu uygulamalar sadece eğitici nitelikte değil aynı zamanda "iş yaptıracak" niteliktedir. Bunları alıp, birleştirip, değiştirip ve geliştirip projeler yapacaksınız. İş yaptırmayacak, oyalayacak uygulamaya yer vermiyorum. Şahsen bunları ben de kendi projelerimde kullanmak için yazdım.

31. Uygulama – Zamanlayıcıları Sayıcı Olarak Kullanmak

Şu ana kadar zamanlayıcılarla mikrodenetleyici içinde zamanlama işlemleri ve sinyal çıkış işlemi yaptık. Ama zamanlayıcılar aynı zamanda dış dünyadan da belli sinyalleri alıp buna göre çalışabilir. Buna "*Input Capture*" adı verilmektedir. Zamanlayıcı sayacı artırmak için işlemci saatini kullanmak yerine giriş ayağını kullanmakta ve bu giriş ayağına uygulanan sinyaldeki değişime göre sayaç artmakta ve bunu işlemci okumakta, yeri geldiğinde de kesmeye gitmektedir. Bu uygulamada da C portunun 5. Ayağına bağlı bir LED ve D portunun 4. Ayağında ise *pull-up* direnci ile bağlanmış bir düğme vardır. Düğmeye toplamda 5 kere basıldığında kesmeye gidilecek ve LED açılıp kapatılacaktır. Bu uygulama biraz tafsilatlı olduğu için kodları dikkatle incelemek gerekecek.

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

```
#define F_CPU 16000000UL
#include <avr/interrupt.h>

int main(void)
{
	DDRC |= (1<<PC5);
	DDRD &= ~(1<<PORTD4);
	TCCR0B |= (1<<CS01);
	TCCR0B |= (1<<CS02);
	TCCR0A |= (1<<WGM01); // CTC MOD
	OCR0A = 5;

	TIMSK0 |= (1<<OCIE0A);
	sei();
	while (1)
	{
	}
}

ISR (TIMER0_COMPA_vect)
{
	PORTC ^= (1<<PC5);
}
```

Şimdi programda önemli kısımları inceleyerek devam edelim.

```
DDRD &= ~(1<<PORTD4);
```

Burada zamanlayıcının giriş ayağını yani D portunun 4 numaralı ayağını giriş olarak ayarlıyorum. Normalde açılışta bu giriş olarak tanımlı olsa da ne yapıldığını kod üzerinde görmek için bunu yazmayı alışkanlık edindim. Zamanlayıcı girişinde de port ayarı yapmayı unutmayınız.

```
	TCCR0B |= (1<<CS01);
	TCCR0B |= (1<<CS02);
```

Burada CS01 ve CS02 bitlerini bir (1) yaptım. Datasheette yer alan tabloda bu modda zamanlayıcı dışarıdan sinyal alıyor ve düşen kenarda artırım gerçekleşiyor. Bunun yerine yükselen kenarı da tercih edebilirsiniz. Ben *pull-up* direnci bağladığım için düşen kenarı tercih ettim. Düğmeye basınca sıfır (0) sinyali elde edilmekte.

```
	TCCR0A |= (1<<WGM01);
```

Burada zamalayıcıyı CTC modunda çalıştıracağım. Çünkü bir sonraki kodda kaça kadar sayılması gerektiğini elimle belirleyeceğim.

```
OCR0A = 5;
```

Burada zamanlayıcının 5'e kadar saymasını belirledim. Düğmeye 5 kere bastığımda kesmeye gidecek ve bağlı LED'i yakacak veya söndürecek. Yalnız düğme arkını hesaba kattığımızda bu sayıda bir değişiklik yaşanabilir. Bunu *delay* yöntemi ile engellemenin imkânı olmadığı için donanımsal olarak ek devreler ile engellemek gerekir. Şimdilik bununla uğraşmaya gerek yok.

```
TIMSK0 |= (1<<OCIE0A);
```

Burada da OCR0A yazmacının kesmesini etkinleştiriyorum. İsterseniz ek bir karşılaştırma değerine ihtiyaç duyduğunuzda OCR0B yazmacını ve bunun kesmesini de kullanabilirsiniz. Yani belli bir sınıra aştığında bir fonksiyon, öteki sınırı aştığında ise farklı bir fonksiyon çalıştırılabilir.

En son olarak zamanlayıcının karşılaştırma kesmesinde de (taşma değil) C portunun 5. Ayağına *toggle* işlemi yapılmaktadır. Daha öncesinde taşma kesmesini kullansak da taşma kesmesi zamanlayıcının gerçek değerini referans almaktadır. Biz burada elimizle bir değer atadık.

Burada en basit haliyle gösterdiğimiz sinyal okuma yönteminin üzerinde çalıştığımızda hassas bir frekansmetre cihazı bile ortaya koyabiliriz. Birkaç uygulama sonrasında bunu göreceğiz.

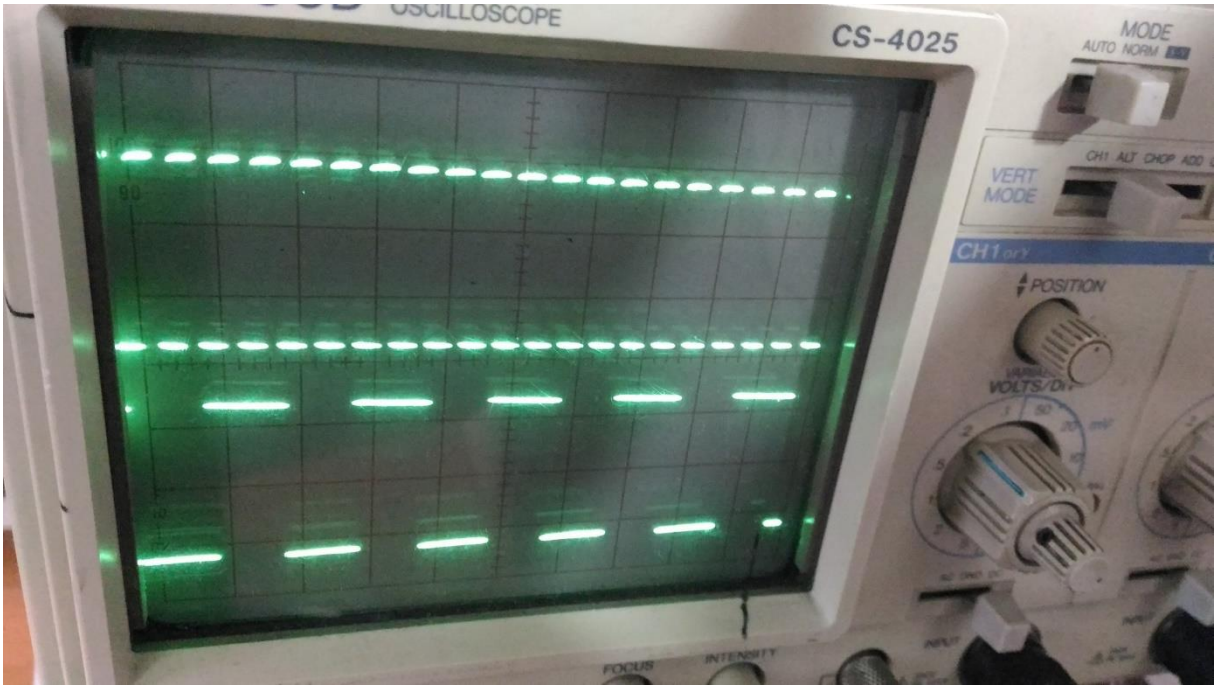
32. Uygulama – Zamanlayıcılar ile Frekans Bölücü Uygulaması

Normalde zamanlayıcılar ile zamanlama, frekans üretme, frekans okuma pals genişliği (*pulse width*) okuma işlemleri yapılsa da bu birimin temelinde bir sayaç olduğu ve entegre olarak sayaçların genelde frekans bölmede kullanıldığını bildiğim için "Neden mikrodenetleyicideki bu birim bu iş için kullanılmasın?" dedim ve ortaya böyle bir uygulama çıktı. Yalnız bu frekansı bölmek için bir frekans girişine de ihtiyaç vardı ve bu frekansı da yine başka bir zamanlayıcı kullanarak aynı mikrodenetleyici içerisinde ürettim. Bu frekans bölücünün diğerleri gibi olmadığını, yazılım ile ince ayar yapılıp güncellenebildiğini unutmayın.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>

int main(void)
{
      DDRC |= (1<<PC5);
      DDRD |= (1<<PORTD6);
      DDRD &= ~(1<<PORTD4);
      TCCR0A |= (1<<WGM01); // CTC MOD
      TCCR0A |= (1<<COM0A0);
      TCCR0B |= (1<<CS01);
      TCCR0B |= (1<<CS02);
      OCR0A = 1;

       unsigned long frekans = 10000UL; // değiştirilebilir
    DDRB |= ((1<<PB1) | (1<<PB2));
    TCCR1A = 0;
    TCCR1B = 0;
    OCR1B = 0;

    int ocr_deger = 1.0/frekans / (2.0/16000000.0)-1;
    OCR1A = ocr_deger;

    TCCR1A |= ((1<<COM1B0) | (1<<COM1A0));
    TCCR1B |= (1<<WGM12); // CTC modu
    TCCR1B |= (1<<CS10); // 8x prescaler
      while (1)
      {
```

```
        }
}
```

Devreyi kurmak için öncelikle T1 zamanlayıcısının frekans çıkışını T0 zamanlayıcısının girişine bağlamanız gereklidir. Burada B portunun 1 numaralı ayağını jumper kablo ile D portunun 4. Ayağına bağlıyoruz. Frekans bölücünün çıkışı ise D portunun 6. Ayağıdır. Osiloskopa hem giriş hem de çıkış sinyalini ayrı kanallardan bağlamamız gereklidir. Ben bu kurulumu yaptıktan sonra osiloskopta şöyle bir sinyal aldım.



Burada işlemcinin yaptığı hiçbir şey yoktur. Bütün işi donanım halletmektedir. Bu biraz deneysel bir uygulama olsa da pek çok yerde frekans bölücülerin kullanıldığını gördüm ve böyle bir ayarlanabilir frekans bölücünün bir yerde işinize yarayacağına eminim.

Frekans bölme oranı hem T0 zamanlayıcısının CS bitleri ayarlanarak yapılabildiği gibi hem de OCR yazmacının değeri değiştirilerek yapılabilir. Bu durumda bölme oranı oldukça esnek olmakta, diğer frekans bölücü olarak kullanılan sayaçlar gibi 2'nin katlarıyla bile sınırlı olmamaktadır. 16-bit yazmaçla frekans üretme uygulamasını sonraki uygulamalarda anlatacağım.

33. Uygulama : Aşırı Yavaş Zamanlayıcı

Artık zamanlayıcıların sınırlarını zorlama vakti geldi. İlk yaptığımız uygulamada T0 zamanlayıcısının ne kadar uğraşılsa uğraşılsın bir türlü yavaş çalışamadığını görmüşsünüzdür. İnsan gözüyle görülecek yavaşlıkta bir LED yakıp söndürme işini bile yapamamakta. Pek çok zaman biz zamanlama işleminde saniyeleri, dakikaları ve hatta saatleri kullanırız. Bu kadar uzun süre bekleme yapamayan zamanlayıcılar çok kullanışlı olmayacaktır. Ben bu uygulamada bu eksikliği giderme adına mikrodenetleyiciyi hiç meşgul etmeden, yazılım tarafıyla uğraşmadan sırf donanımla bunu yapmaya çalıştım. Program başladığı zaman ilk ayarı yapıyoruz ve sonrasında unutuyoruz. Başka yöntemlerle de uzun zamanlama işlemleri yapılabilse de bu uygulamanın güzel yanı buradadır.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <avr/interrupt.h>

int main(void)
{
	DDRD &=~(1<<PORTD5);
	DDRD |=(1<<PORTD6);
	DDRB |=(1<<PORTB1);

	TCCR0B |= (1<<CS00);
	TCCR0B |= (1<<CS02);
	TCCR0A |= (1<<WGM01); // CTC MOD
	TCCR0A |= (1<<COM0A0);
	OCR0A = 255;

	TCCR1A |= (1<<COM1A0);
	TCCR1B |= (1<<WGM12); //CTC
	TCCR1B |= ((1<<CS10) | (1<<CS11) |(1<<CS12) ); //external input
	OCR1A = 1;


	while (1)
	{
	}
}
```

Burada T0 ve T1 zamanlayıcıları beraber kullanılmıştır. T0 zamanlayıcısı en yavaş haliyle çalışmakta ve ürettiği sinyal ile T1 zamanlayıcısını beslemektedir. T1 zamanlayıcısı ise karşılaştırma değerine göre çok daha yavaşlatılmış bir sinyali dış dünyaya çıkış olarak vermektedir. Burada sinyali yavaşlatmak için T0 ön

bölücü, T0 karşılaştırma değeri, T1 ön bölücü ve T1 karşılaştırma değeri üzerinde oynama yapabiliriz. En yavaş haliyle birkaç saate varan bir bekleme süresi vermektedir.
Devreyi kurmak için öncelikle D portunun 6 numaralı ayağı D portunun 5 numaralı ayağına bağlanmalı. B portunun 1 numaralı ayağına da bir LED veya sinyali gözlemleyeceğimiz bir aygıt bağlanmalıdır. Her ne kadar aynı çip içerisinde olsa da bu zamanlayıcıların birbirini beslemesi için dışarıdan bağlantı yapmak gereklidir. Gelişmiş denetleyicilerde birlikte çalışan zamanlayıcıları görebiliriz.

Bu uygulamayı geliştirmek ve donanımı en son sınırında kullanmak için T2 zamanlayıcısını da işin içine dahil edebiliriz. Bu durumda saatlerce değil günlerce, haftalarca bekleme elde etme şansımız olur. Bunun hesabını "*C ile AVR Programlama*" yazılarımdaki formüllere göre yapıp istediğiniz beklemeyi elde edebilirsiniz.

Bu bekleme kristal osilatör ile ve mikrodenetleyici tabanlı olduğu için diğer zamanlayıcı devrelerin verdiği zamanlamaya göre daha hassas olacaktır. Mesela aynı işi 555 ile yapsanız kondansatör ve dirençlerin toleransı ve ortamdan etkilenmesi işin içine girecektir.

34. Uygulama – 16-bit PWM

Bu uygulamada T1 zamanlayıcısı ile 16-bit çözünürlükte PWM sinyali elde edeceğiz. 8-bit PWM sinyali LED uygulamalarında yumuşak bir animasyon elde etmemiz için yetersiz kalmakta. Özellikle bu uygulamayı LED'ler ile denediğinizde 8 bit ile 16 bit farkını rahatça anlayabilirsiniz. Bu uygulamada T1 zamanlayıcısında yer alan ICR yazmacı referans alındığından uygulamada doğrudan OCR değeriyle oynayarak görev döngüsünü değiştirebildim. Aynı yöntem 8 bit zamanlayıcılarda sıkıntı çıkarmıştı.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

int main()
{

	DDRB|=(1<<PB1); // PB1 yani OC1A PWM çıkışı
	TCCR1A|=((1<<COM1A1) ); //Yukarı sayımda eşleşme olunca OC1A/OC1B'yi
sıfırla
	TCCR1B|=(1<<WGM13); // Faz ve frekans düzeltmeli PWM etkin.
	TCCR1B|=(1<<CS10); // Ön derecelendirici katsayısı 1 (F_CPU)
```

```
	ICR1=65535; // TOP değerini ayarla. (Çözünürlük)
	OCR1A=0; // Karşılaştırma değerini sıfırla

	while (1)
	{
    OCR1A++;  // Karşılaştırma değerini her döngüde bir artır
	_delay_us(50);
	}
}
```

Bu konu hakkında yazdığım ayrıntılı bir makaleyi okumanızı şiddetle tavsiye ederim.
http://www.lojikprob.com/avr/avr-ile-pwm-kullanimi-ve-16-bit-pwm-cikisi-almak/

35. Uygulama – Ayarlanabilir Hassas Frekans Üretici

Bu uygulamayı frekans üretme uygulamalarında rahatlıkla kullanabilirsiniz. Belli bir değer aralığında istediğimiz frekansı oldukça hassas bir şekilde üretmektedir. Bunun için AD9833 gibi dijital entegreler kullanabilsek de bunların fiyatı mikrodenetleyicinin birkaç katı olabilmektedir. Aynı zamanda analog çözümlerde ise frekans ortamdan etkilendiği için sürekli kaymaktadır. Burada kristal kullanan denetleyici gayet tutarlı bir sonuç vermektedir. Eğer mikrodenetleyiciye daha iyi bir saat beslemesi yapılırsa çok daha isabetli bir sonuç elde edilecektir.

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


int main(void)
{
	double frekans = 100000; // de?i?tirilebilir
	DDRB |= ((1<<PB1) | (1<<PB2));
	TCCR1A = 0;
	TCCR1B = 0;
	OCR1B = 0;
	uint8_t cs_deger;
	uint16_t ocr_deger;
	if(frekans>260)
	{
		ocr_deger = 1.0/frekans / (2.0/16000000.0)-1;
```

```
            cs_deger = 1;
        }
        else if (frekans>40)
        {
            ocr_deger = 1.0/frekans / (16.0/16000000.0)-1;
            cs_deger = 2;
        }
        else if (frekans>4)
        {
            ocr_deger = 1.0/frekans / (128.0/16000000.0)-1;
            cs_deger = 3;
        }
        else if (frekans>1)
        {
            ocr_deger = 1.0/frekans / (512.0/16000000.0)-1;
            cs_deger = 4;
        }
        else
        {
            ocr_deger = 1.0/frekans / (2048.0/16000000.0)-1;
            cs_deger = 5;
        }
        OCR1A = ocr_deger;

        TCCR1A |= ((1<<COM1B0) | (1<<COM1A0));
        TCCR1B |= (1<<WGM12); // CTC modu
        TCCR1B |= cs_deger;
        while (1)
        {
        }
}
```

Burada T1 zamanlayıcısını kullandım, 16 bit zamanlayıcı ile yeterli seviyede frekans aralığında istediğimiz değeri elde edebiliyoruz. 8 bit ile 16 bit arasında dağlar kadar fark olduğunu söyleyebilirim. Yalnız çok düşük frekanslarda yine zamanlayıcı yeterli kalmadığından istenilen değer alınamıyordu. Burada ön bölücü olmadan 260Hz ve 8MHz arasında sinyali 16MHz işlemcide alabiliyordum. 260Hz'den aşağı inmek istediğimizde sinyal bozuluyordu. Neyse ki ön bölücüler sayesinde bu frekans aralığını biraz genişletme fırsatı buldum ve böylelikle daha düşük frekanslardaki sinyali üretebildim.

Bunun için gerekli hesaplamaları yaptıktan sonra belli değer aralıkları için doğru ön bölücü değerini tespit ettim ve 2048'e adar bölme değerini artırdım. Bu sayede 1Hz'in altındaki sinyallerde bile başarılı sonuç elde edildi. Yanlış hatırlamıyorsam 0.3Hz'e kadar doğru bir sinyal almıştım.

En alt sınırı böyle güzel belirlesem de üst sınırda elimizi kolumuzu bağlayan şey CPU frekansı olmakta. Ne yaparsak yapalım CPU frekansının yarısından fazla frekansı elde edemiyoruz. Yani bu 16MHz'de 8MHz, 20MHz'de ise 10MHz olmakta. Üstelik bu üst değerlerde yine çözünürlük düşmekte ve frekans kademeleri arasında ciddi fark olmakta. Mesela 8MHz'den bir düşük frekans 4MHz olmakta. Kesinlikle 5-6MHz veya bu ikisinin arasında herhangi bir değeri alamıyoruz. Bu tüm zamanlayıcılar için geçerli olan oldukça normal bir durumdur. Çünkü biz 8MHz sinyal elde etmek istediğimizde karşılaştırma değerini 1 yapmamız gereklidir, 1'den bir sonraki değerle daha yavaş bir sinyal elde ederiz bu da 2 değeri olmaktadır. Bu 2 değeri ile 5333333Hz değeri alırız. Bu durumda 3 olduğu zaman 4MHz sinyali alırız. Değerler artmaya başladıkça bölünen parçalar küçüldüğünden daha hassas değer alabiliriz. Mesela 1KHz-1MHz arasında çalışacaksanız bu program hassas bir sinyal jeneratörü olabilir.

Bir proje fikri olarak söylemem gerekirse buna birkaç düğme ve LCD ekran ekleyerek öğrencilere yönelik bir sinyal üreteci yapabilirsiniz.
Burada aşağıdaki OCR değerini bulmak için yapılan bölme işlemi, datasheette verilen formülün uygulamasıdır.

Not: Bu uygulamayı ufak bir hoparlör ile denediğinizde oldukça hoş sonuçlar elde edebilirsiniz. Frekans çıkışına bir hoparlör bağlamanız yeterlidir.

36. Uygulama – Frekans Ölçer Uygulaması

Bu uygulama da yukarıdaki frekans üreteci gibi elektronik laboratuvarında sıkça kullanılan cihazların yaptığı görevi yapmaktadır. Siz öğrenci olarak pahalı ekipmanlara para vermek yerine işinizi görecek hassasiyette olan bu cihazları yapıp kullanabilirsiniz. Bu uygulama mikrodenetleyicinin sınırları çerçevesinde olan frekansları hassas bir şekilde ölçmektedir. Daha yüksek frekanslarda frekans bölücü kullanmanız gerekecektir.

```
#include <avr/io.h>
#include "freq.h"
#include "uart.h"
#define  F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>
int main(void)
{
```

```
    sei();
      uart_init(UART_BAUD_SELECT(9600,F_CPU));
      _delay_ms(10);
      uart_puts("AVR FREKANS OKUYUCU \n \n");

    while (1)
    {
      f_comp = 10;
      char buf [20];
      freq_start(1000);
      while (f_ready == 0)  {}

      uint32_t frq = f_freq;
      sprintf(buf,"%lu \n",frq);
      uart_puts(buf);
    }
}
```

Burada UART üzerinden okunan frekans değerini seri port ekranına yazdırdım. Hz hassasiyetinde okumak istiyorsanız *freq_start(1000)* şeklinde 1000ms bekleme süresi ayarlamanız gereklidir. Eğer bu kadar sabrınız yoksa burada 100ms yazıp 10Hz çözünürlükte okuma yapabilirsiniz. Frekans okumayı elektronik ve gömülü sistemlerde sıkça yapmak durumunda kalıyoruz. Bazı algılayıcılar analog çıkış vermek yerine frekans olarak çıkış vermekte. Bu durumda bu frekansı ölçmeden bu algılayıcıyı kullanmak imkânsız oluyor. Frekans ne kadar hassas ölçülürse o kadar doğru bir ölçüm yapılmış oluyor. Böyle frekans ölçüm uygulamalarında yukarıdaki örneği rahatlıkla kullanabilirsiniz.

Frekans ölçüm işlemi için güvendiğim bir kütüphaneyi kullanma kararı aldım. Arduino için olan aşağıdaki kütüphaneyi AVR'ye uyarladım.
http://interface.khm.de/index.php/lab/interfaces-advanced/arduino-frequency-counter-library/index.html

37. Uygulama – Pals Genişliği (Pulse Width) Ölçümü

Bu uygulama da hiç oyuncak olarak göremeyeceğiniz ve karşınıza çıktığında hayat kurtarıcı nitelikte olan bir uygulamadır. Örneğin mikrodenetleyicinin ayağına 1 veya 0 konumunda bir sinyal uygulanıyor diyelim. Bu sinyalin frekansını değil, uzunluğunu bulmanız gerekiyor. Bunu nasıl yapacaksınız? Yani bir sinyal kaç milisaniye veya mikrosaniye süreyle mikrodenetleyicinin ayağına uygulandı sorusunun cevabını arıyoruz. Bunun için pals genişliği okumamız gerekecektir.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define  F_CPU 16000000UL
#include <util/delay.h>
#include "uart.h"
#include <stdio.h>
volatile unsigned char ov_counter;
volatile unsigned int starting_edge, ending_edge;
volatile unsigned long clocks;
int main(void)
{
      DDRB &= ~(1<<PORTB0);
    sei();
      uart_init(UART_BAUD_SELECT(9600,F_CPU));
      _delay_ms(100);
      uart_puts("INPUT CAPTURE DENEME PROGRAMI \n");
      TCCR1B &= ~(1<<ICES1);
      TCCR1B |= (1<<CS10);
      TCCR1B |= (1<<CS01);
      TIMSK1 |= ((1<<ICIE1) | (1<<TOIE1));
    while (1)
    {
      char buffer[10];
      sprintf(buffer, "%lu \n",clocks);
      uart_puts(buffer);
    }
}

ISR (TIMER1_OVF_vect)
{
      ++ov_counter;
}

ISR (TIMER1_CAPT_vect)
{
```

```
        ending_edge = 256*ICR1H + ICR1L;
        clocks = (unsigned long)ending_edge + ((unsigned long)ov_counter *
65536) - (unsigned long)starting_edge;
        ov_counter = 0;
        starting_edge = ending_edge;
}
```

Bu uygulamayı yaparken kitabın başında bahsettiğim Barnett'in kitabının oldukça katkısı oldu. Çünkü T1 zamanlayıcısındaki Input Capture (*Giriş Yakalama*) özelliğini kullanmaya çalışmak oldukça zahmetli oldu ve internette buna dair elle tutulur bir örnek de bulamadım. Oldukça önemli, ama bir yandan da spesifik bir uygulama olduğu için diğer bu tarz uygulamalarda olduğu gibi aradığımı bulmakta zorlandım. Ama en sonunda imdadıma bu kitap yetişti.

Burada B portunun 0'ıncı ayağına bağlı bir *pull-up* direnciyle desteklenen bir düğme yer almakta. Biz düğmeye bastığımızda düşen kenardan itibaren sayaç saymakta ve ne kadar süre geçtiğini bize söylemekte. Burada sadece sayaç değeri görünse de gerekli hesaplamaları yaptıktan sonra istenilen değer ortaya çıkacaktır. Bu uygulamanın da diğer zamanlayıcı uygulamaları gibi bir gün size büyük yardımı olacaktır.

38. Uygulama – Analog Karşılaştırıcı Uygulaması

Atmega328P mikrodenetleyicide bir adet analog karşılaştırıcı bulunmaktadır. Normalde bu karşılaştırma devreleri işlemsel yükselteçler (OPAMP) ile yapılmaktadır. Ama burada iç çevre birimi olmasından dolayı ek bir elemana ihtiyaç duymadan ve dijital olarak kontrol ederek yapabiliyoruz. Bu denetleyicide sadece bir adet yer alsa da gelişmiş analog ağırlıklı denetleyicilerde 10-20 arası karşılaştırıcının olduğunu ve bunların daha özellikli ve hızlı olduğunu görmekteyiz. Mesela benim analog yönüyle çok sevdiğim STM32F3 serisi böyledir. Şimdi basit bir uygulama yaparak bunu görelim. Zaten datasheette de birkaç sayfa yer ayrılmıştır. Anlaması ve kullanması oldukça kolay bir birimdir.

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


int main(void)
{
    DDRC |= (1<<PC5);
```

```
    while (1)
    {
            if(ACSR & (1<<ACO))
            PORTC |= (1<<PC5);
            else
            PORTC &=~(1<<PC5);
    }
}
```

Burada tek yapılan işlemin ACO bitinin denetlenmesi olduğunu ve buna göre C portunun 5 ayağının yakılıp yakılmadığı belirlenmektedir. Devrede C portunun 5. Ayağına bir LED bağlanmakta, AIN0 pozitif giriş, AIN1 ise negatif giriş olmaktadır. Negatif giriş aslında eşik değeri olarak çalışmaktadır. Pozitif giriş negatifin üstündeyse ACO bitinde bir (1), altındaysa aynı bitten sıfır (0) değerini okumaktayız. En basit haliyle bir bitlik ADC diyebiliriz. Analog karşılaştırıcı genellikle analog değerin sayısal olarak ifade edilmesi yerine bir eşiği geçip geçmediğinin kontrol edildiği kurumlarda kullanılır. Örneğin bir termemoetrenin 50 dereceyi geçtiğinde uyarı vermesini veya bir bataryanın 9 voltun altına düştüğünde bir şekilde uyarı vermesini istiyorsak karşılaştırıcıları kullanırız.

Yukarıda ve diğer uygulamalarda bahsettiğim ayak adlarını kitabın en başında verdiğim şemada rahatça bulabilirsiniz.

39. Uygulama – Analog Karşılaştırıcı Kütüphanesi

Github'da AVR için analog karşılaştırıcı kütüphanesi aradığımda bulamadım. Kütüphane olmadığı için kütüphaneyi kendim yazayım dedim. Basit bir birim olduğu için kısa sürede kütüphane bitti. Şimdi bu kütüphaneyi inceleyelim ve örnek kod üzerinden kullanımını görelim.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "AnalogComparator.h"
int main(void)
{
    ac_int(ENABLE, INT_TOGGLE);
      sei();
      DDRC |= (1<<5);
    while (1)
```

```
    {
    }
}

ISR (ANALOG_COMP_vect)
{
      PORTC ^= (1<<PC5);
}
```

Burada analog karşılaştırıcı kesmesini kullanarak C portunun 5 numaralı ayağına aç/kapa işlemi uyguluyoruz. Böylelikle eşik değeri aşıldığı zaman LED yanacak, sonra tekrar gerilim eşik değerinden düşünce sönecektir. AIN0 ayağına pozitif, AIN1 ayağına ise negatif bağlantı yapmayı unutmayınız. Genellikle AIN1 ayağına referans gerilimi uygulanır. Örneğin potansiyometre ile bunu 2.5V seviyelerine getirebiliriz. Sonrasında ise AIN0 ayağına bağlı potansiyometre ile oynayarak bu deneyi yapabiliriz.

```
ac_int(ENABLE, INT_TOGGLE);
```

Bu komut ile eğer devre dışı bırakıldıysa analog karşılaştırıcı etkinleştirilecek ve hangi modda çalışacağı belirtilecektir. Normalde etkinleştirme diye bir şey olmasa da devre dışı bırakılmış karşılaştırıcıyı burada etkinleştirmekteyiz. Yine de kullanıcı analog karşılaştırıcıyı kullanıma hazırladığından emin olması için bunu böyle belirledim.

Şimdi *AnalogComparator.h* dosyasına bakarak kütüphaneyi inceleyelim.

```
#ifndef ANALOGCOMPARATOR_H_
#define ANALOGCOMPARATOR_H_

// Written By Gökhan Dökmetaş
// www.lojikprob.com

#define INT_TOGGLE 0
#define INT_FALLING 1
#define INT_RISING 2

#define BANDGAP_FIX 0
#define BANDGAP_AIN 1

#define DISABLE 0  // STATUS
#define ENABLE 1

void ac_status(int status);
```

```
void ac_bandgap_select(int bandgap);

int ac_read();  // return ACO

int ac_int_flag(); // Return ACI

void ac_int(int status, int mode); // Interrupt Enable

void ac_capture(int status); // Capture Enable



#endif /* ANALOGCOMPARATOR_H_ */
```

Burada datasheetten gerekli parametreleri okudum ve bunların tanımını en başta INT_TOGGLE, INT_FALLING gibi tanımlamalarla belirledim.

```
void ac_status(int status);
```

Bu fonksiyonda AC (*Analog Comparator*) birini kesme olmadan etkinleştirilmekte veya devre dışı bırakılmaktadır. Başlangıçta etkin olduğu için sonradan etkinleştirmeye gerek yoktur.

```
void ac_bandgap_select(int bandgap);
```

Bu fonksiyonda bant genişliği seçilmektedir. Bandgap değeri olarak BANDGAP_FIX ve BANDGAP_AIN değerleri tanımlıdır. Yani istersek FIX değer olarak iç referans gerilimini pozitif taraf için seçebiliriz. Ya da AIN diyerek AIN0 ayağına uyguladığımız pozitif değer seçilir.

```
int ac_read();
```

Burada ACO biti geri döndürülmektedir. Yani analog karşılaştırıcı birimini okumak için bu fonksiyonu kullanıyoruz.

```
int ac_int_flag();
```

Bu fonksiyon analog karşılaştırıcı kesme bayrağını geri döndürmektedir. Kesmenin gerçekleşip gerçekleşmediğini buradan öğreniyoruz. Uygulamada genellikle pek kullanılmasa da özellik olarak bunu ekledim.

```
void ac_int(int status, int mode);
```

Burada analog karşılaştırıcı ilgili kesme modu ile beraber başlatılacaktır. Analog karşılaştırıcıyı kullanmanın en verimli yolu kesmelerle beraber kullanmaktır.

```
void ac_capture(int status);
```

Bu özellikle analog karşılaştırıcının çıkışı TC1 zamanlayıcısının girişine (*Input Capture)* bağlanmış olur. Bu sayede analog karşılaştırıcıda oluşan her değişimde zamanlayıcı tetiklenecektir. Yeri geldiğinde faydalı bir özellik olarak kullanılabilir.

Bunun dışında analog karşılaştırıcı ayakları diğer ADC ayakları üzerinden de çalışsa da o özelliği ekleme gereği duymadım. Bizim burada kullanacağımız başlıca özellik kesmeler üzerinden bu birimi kullanmaktır.

Buraya kadar .h dosyasını incelemiş oldum. *.c* dosyası oldukça basit olduğu için inceleme gereği duymuyorum ve datasheet üzerinden sizin takip edip incelemenizi istiyorum. Bu kitabı okuyacak seviyeye gelip bir de bu noktaya geldiyseniz artık datasheet okuma, yazılı kodu inceleme, kütüphane inceleme gibi konularda kendinizi iyice geliştirmeniz gereklidir. Bu da bireysel çalışma ile mümkündür.

40. Uygulama – Dış Kesme Kullanımı

Buraya kadar kesmeleri etkin bir biçimde kullandık ve kesmeler sayesinde mikrodenetleyici pek çok işe koşar, bir işe yarar hale geldi. Kesmeler olmadan bunları yapmak istesek 4-5 mikrodenetleyiciyi bile beraber kullanmamız gereken yerler çıkacaktı. Çünkü kesme olmadığı zaman mikrodenetleyicinin bir işle her daim meşgul olması gerekiyordu. Bu meşguliyet ise sürekli birkaç biri denetleyip if karar yapısını buna göre çalıştırmaktan ibarettir. Kesmeler bizi bu zahmetten kurtarmaktadır fakat buraya kadar hep ADC, UART, AC gibi iç birimlerin kesmelerini kullandık. Mikrodenetleyiciye bir düğme veya bir termostat bağladığımızı düşünelim. Bu düğmenin veya termostatın durumunu program akışı boyunca her zaman kontrol etmemiz gereklidir. Eğer zamanında kontrol etmezsek uygulamada ciddi sorunları yaşayabiliriz. Bir düğmeye bastığında anında tepki verecek, hemen mevcut işi bırakıp gerekli komutları işletecek bir denetleyici oldukça işimize yarayacaktır. Bunun için kullandığımız ATmega328P mikrodenetleyicisinde iki adet dış kesme vektörü ve dış kesme ayağı bulunmaktadır. Böylelikle dış dünyada gerçekleşen bir olay ile mikrodenetleyici kemseye gitmekte ve bizim belirlediğimiz kodları işletmektedir. Bu dış kesmeleri kullanmak oldukça kolaydır. Aynı dijital girişte olduğu gibi belli bit ayarlarını yaptıktan sonra INT0 ve INT1 ayaklarına istediğimiz elemanı ve devreyi

bağlayabiliriz. INT0 ayağı şemadan göreceğiniz üzere D portunun 2 numaralı ayağına, INT1 ayağı ise D portunun 3 numaralı ayağına denk gelmektedir. Şimdi o iki ayağa *pull-up* direnci ile iki düğme bağlayıp, C portunun 4 ve 5 numaralı ayaklarına da birer LED bağlayıp uygulamayı deneyelim.

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

int main(void)
{
    EIMSK |= ((1<<INT1)|(1<<INT0));
      DDRC  |= ((1<<PC5)|(1<<PC4));
      sei();
    while (1)
    {

    }
}

ISR (INT0_vect)
{
      PORTC |= (1<<PC5);
}

ISR (INT1_vect)
{
      PORTC |= (1<<PC4);
}
```

Kodların oldukça sade olduğunu fark edebilirsiniz. Siz kendi projelerinizde sadece INT1_vect veya INT0_vect fonksiyonunun içindeki kodları değiştirerek aynen kullanabilirsiniz. Dış kesmelerin bir avantajı da sadece düşen kenarda değil, yükselen kenar, ayak durumunda değişme ve sıfır durumunda sürekli yürütme gibi özelliklerinin de olmasıdır. Datasheette ilgili bit ayarları ve özellikle şu şekilde yer almaktadır.

Table 13-1. Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Her iki kesme için ayrı ayak, ayrı mod ve ayrı vektör vardır. Yani bunları birbirinden tamamen bağımsız şekilde kullanabilirsiniz.

41. Uygulama – PCINT Kullanımı

Dış kesmeler sadece INT0 ve INT1 ayakları ve vektörlerinden ibaret değildir. Bunun yanında dış kesme olarak istediğiniz dijital ayağa atayacağınız (*Pin Change Interrupt*) adı verilen kesmeler vardır. Bu kesmeler istenilen dijital ayakta kullanılabilen ayrı vektörlü kesmeler olduğu için dış kesmelerin yanında veya onların yerine kullanılabilir.

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

int main(void)
{
    PCICR |= ((1<<PCIE2)|(1<<PCIE1));
      PCMSK2 |= (1<<PCINT20);
      PCMSK1 |= (1<<PCINT8);
      PCMSK0 |= (1<<PCINT2);
      DDRC |= ((1<<PC5) | (1<<PC4));
      sei();
    while (1)
    {

    }
}

ISR (PCINT1_vect)
{
      PORTC ^= (1<<PC5);
}

ISR (PCINT2_vect)
{
      PORTC ^= (1<<PC4);
}
```

Burada öncelikle PCICR yani ayak değişmesi kesmelerini kontrol eden yazmaçta PCIE2 ve PCIE1 kesmelerini etkinleştiriyorum. Toplamda 3 kesme olsa da PCIE0 kesmesini bunlarla beraber bir türlü çalıştıramadım. Sorunun nereden kaynaklandığını bulmayı sizlere bırakıyorum. Sonrasında PCMSK2, PCMSK1 ve PCMSK0 yazmaçlarından ilgili kesmelere ayak ataması yaptım. Burada her

kesmenin her ayağa atanabileceğini göremiyoruz. Mesela PCINT0 kesmesi PCINT0 ve PCINT7 ayakları içine atanabiliyor. PCINT1 kesmesi ise PCINT8 ve PCINT15 arasına atanabiliyor. Bu şekilde 3 kesmenin atanabileceği ayaklar 3 ayrı bölgeye ayrılmış oluyor. Bu ayaklar UNO kartının şemasında da yer almaktadır. Mesela burada PCINT2 kesmesi için D portunun 4. Ayağı, PCINT1 kesmesi için de C portunun 0 numaralı ayağı seçilmiştir. Yukarıdaki uygulamada olduğu gibi C portunun 4. Ve 5. Ayaklarına da birer LED bağlıdır.

Siz uygulamada hangi ayağı kullanmak istiyorsanız PCMSKx yazmaçlarının değerlerini değiştirmeniz gereklidir.

42. Uygulama – EEPROM ile Datalogger

Veri günlüğü kaydedici (Datalogger) uygulamaları pek çok projede karşınıza çıkacaktır. Bir değeri okumak veya bunu ekranda göstermek her zaman yeterli olmamaktadır. Mesela bir gün içindeki sıcaklık değişimi grafiğini oluşturmak için öncelikle saat ve sıcaklık bilgisinin belli periyotlarla bir belleğe kaydedilmesi gereklidir. Ya cihazın başında 24 saat bekleyip kâğıt kalemle bunları kayıt altına alacağız ya da yazdığımız program ile bunu cihaz günler boyu bizim yerimize yapacak. Böylelikle okuduğunuz verileri en basitinden Excel gibi programda görselleştirebilirsiniz. Datalog özelliği ticari anlamda ürünler ortaya koyarken çok işinize yarayacaktır.

```
#include <avr/io.h>
#include "adc.h"
#define  F_CPU 16000000UL
#include <util/delay.h>
void datalog(void);
uint16_t address = 0;
int main(void)
{
    adc_init();
      DDRC |=(1<<PC4);
    while (1)
    {
            if(!(PINC & (1<<PC3)))
            datalog();
    }
}

void datalog()
```

```
{
      PORTC |= (1<<PC4);
      while (PINC & (1<<PC3));
    _delay_ms(50);
      uint8_t veri = adc_read_map(0, 1, 255);
      while(EECR & (1<<EEPE))
      ;
      /* Adres ve Veri yazmaçlarını ayarla*/
      EEAR = address;
      EEDR = veri;
      /*EEMPE bitini bir yap */
      EECR |= (1<<EEMPE);
      /* EEPE bitini bir yaparak yazmaya başlat.*/
      EECR |= (1<<EEPE);
      address++;
      while(EECR & (1<<EEPE))
      ;
      /* Adres ve Veri yazmaçlarını ayarla*/
      EEAR = address;
      EEDR = 0;
      /*EEMPE bitini bir yap */
      EECR |= (1<<EEMPE);
      /* EEPE bitini bir yaparak yazmaya başlat.*/
      EECR |= (1<<EEPE);
      PORTC &= ~(1<<PC4);
}
```

Burada EEPROM komutları için datasheette verilen örnek kodları kullandım. AVR derleyicisinin EEPROM kütüphanesi de bulunmakta. Kullanımı kolay bir kütüphane olsa da şimdilik işin temelinde bunu kullanmaya gerek yok. Burada sadece veri yazma fonksiyonunu kullandım. Önce veri yazılmakta ve bir sonraki adrese ise sıfır (0) değeri yazılmakta. Bu sıfır değeri aslında verinin sonunu belirten bir değer olarak okumada yardımcı olacaktır. Aynı C dilindeki '\0' işareti gibi EEPROM hafızasında kaydın sonunu bu şekilde anlıyoruz. Kaydedilen veri ise bu yüzden 1 ve 255 arasında olmakta.

Burada C portunun 4. Ayağına bir *pull-up* direnciyle bir düğme bağlamaktayız. Bu kayıt düğmesi olacak ve her düğmeye bastığımızda A0 kanalından okunan analog değer kaydedilecek. Bunun yanında C portunun 4. Ayağına bir buzzer veya LED bağlıyoruz ve bizim kayıt işlemini yaptığımızı bize bildiriyor. Her kayıtta adres değerini bir artırarak EEPROM'un kapasitesi boyunca bu kaydı yapıyoruz. Şimdi bu kaydedilen değerlerin nasıl okunduğuna bakalım.

43. Uygulama – EEPROM ile Sıralı Veri Okuma

Bu uygulamada datasheette yer alan EEPROM okuma fonksiyonu ile beraber daha öncesinde bahsettiğim UART kütüphanesini kullandım. EEPROM'dan okunan veriler sıra ile seri port ekranına yazdırılmaktadır. 0 değeri okunduğu zaman dizinin sonuna gelindiği anlaşılmakta ve okuma durmaktadır. Aksi halde kaydedilen verinin yanı sıra pek çok "çöp" veri de ekranda yazdırılacak ve her şey birbirine karışacaktır.

```
#include <avr/io.h>
#include <stdio.h>
#include <avr/interrupt.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include "uart.h"
unsigned char EEPROM_read(unsigned int uiAddress);

int main(void)
{
     sei();
     uart_init(UART_BAUD_SELECT(9600,F_CPU));
     _delay_ms(100);
     uart_puts("EEPROM OKUMA \n");
       unsigned int address = 0;
       unsigned char veri = 0;
       char buf[20];
      while(1)
      {
      veri = EEPROM_read(address);
      if (veri == 0)
      break;
      sprintf(buf,"Adres:%i    Veri:%i \n", address, veri);
      uart_puts(buf);
      address++;
      }
    while (1)
    {

    }
}

unsigned char EEPROM_read(unsigned int uiAddress)
{
      /* Önceki okuma ve yazmanın bitmesini bekle */
      while(EECR & (1<<EEPE))
```

```
        ;
        /* Adres değerini güncelle */
        EEAR = uiAddress;
        /* Okuma işlemini başlat. */
        EECR |= (1<<EERE);
        /* Veri yazmacını değer olarak döndür */
        return EEDR;
}
```

Normalde datalog işlemlerinde genellikle dosya üzerinde çalışılmakta ve ASCII karakterler kullanılmaktadır. Bunu size basit bir şekilde göstermek adına bu şekilde yaptım. Bu konuda farklı yöntemler de kullanabilirsiniz.

44. Uygulama – SPI Ana Aygıt Uygulaması

Daha öncesinde SPI ve I2C protokollerinin öneminden bahsetmiştim. Gerçekten kayda değer işler yapmak istiyorsanız UART, SPI ve I2C iletişim protokollerini öğrenip etkin bir şekilde kullanmanız gerekecektir. SPI birimi bu üç birim içerisinde kullanımı en kolay olanı ve en hızlı olanıdır. Benim gözlemlediğim kadarıyla bu protokol en çok SD kart uygulamalarında ve diğer hafıza entegrelerinde kullanılmaktadır. Bunun yanında pek çok gelişmiş entegre SPI ile kullanılmaktadır. Şimdi basit bir gönderici uygulamasıyla bu birimin nasıl kullanıldığına bakalım.

```
#include <avr/io.h>
void spi_transmit(unsigned char data);
void spi_init(void);
int main(void)
{
      spi_init();
    while (1)
    {
    spi_transmit(0xFF);
      spi_transmit(0x00);
```

```
      }
}

void spi_init(void)
{
          /* MOSI ve SCK Çıkış, Diğerleri Giriş*/
          DDRB |= (1<<DDB3)|(1<<DDB5);
          DDRB |= (1<<DDB2); // SS ÇIKIŞ ÖNEMLİ
          /*SPI Enable, Master Mod, Fcpu/16*/
          SPCR |= ((1<<SPE)|(1<<MSTR)|(1<<SPR0));
}

void spi_transmit(unsigned char data)
{
                  /* Veriyi Yolla*/
                  SPDR = data;
                  /* Bitmesini Bekle */
                  while(!(SPSR & (1<<SPIF)))
                  ;

}
```

Bu programda *spi_init()* ve *spi_transmit()* olmak üzere iki SPI fonksiyonu tanımlanmıştır. Bu fonksiyonlardan biri adından anlayacağınız üzere SPI biriminin gerekli ayarlarını yapmakta ve bunu ilk kullanıma hazırlamaktadır. Öteki fonksiyon ise ana (*master*) modda veri yollamaktadır.

```
/* MOSI ve SCK Çıkış, Diğerleri Giriş*/
          DDRB |= (1<<DDB3)|(1<<DDB5);
```

SPI birimini kullanırken dikkat etmeniz gereken en büyük noktalardan biri de SCI, MOSI ve SS ayaklarını çıkış olarak ayarlamanız gerektiğidir. Ben SS ayağını çıkış olarak ayarlamayı unutunca sırf bu yüzden program çalışmamıştı. Datasheette yazan bu şekilde ince noktaları gözden kaçırınca veya unutunca sorunu çözmek oldukça zaman alıcı olabiliyor.

```
SPCR |= ((1<<SPE)|(1<<MSTR)|(1<<SPR0));
```

Burada SPCR adındaki SPI biriminin kontrol yazmacında bazı bit ayarlarını yapmaktayız. SPE biti (*SPI Enable*) SPI birimini etkinleştirmekte ve kullanıma hazır hale getirmektedir MSTR bitini bir (1) yaparak ana aygıt modunda çalıştırmaktayız. SPR0 biti ile de SPI biriminin saat hızını *Fcpu/16* olarak bölmekteyiz. SPI birimi günümüz teknolojisine göre yavaş sayılan bu denetleyicide bile şaşırtıcı derecede hızlı sonuçlar vermektedir.

Şimdi SPI gönderme fonksiyonunu inceleyelim.

```
SPDR = data;
```

Aynı UART biriminde olduğu gibi SPI veri yazmacına veriyi yazar yazmaz gönderim otomatik olarak başlamaktadır. Veri okurken de UART biriminde olduğu gibi aynı yazmacı kullanacağız.

```
while(!(SPSR & (1<<SPIF)))
            ;
```

Sonrasında ise programı sonsuz döngüye sokarak gönderim işleminin bitmesini bekliyoruz. Gönderim işlemi bitmeden tekrar bir gönderim yapmakla veriler üst üste binecek ve çakışma olacaktır.

Burada *while* döngüsünün altında bir noktalı virgül (;) görebilirsiniz. Eğer noktalı virgülü aynı satıra koysaydım gözden kolaylıkla kaçabilirdi. Tek satırda döngüleri ifade ederken bu karakteri alta almakta fayda var. Bu şekilde hemen göze çarpabiliyor ve hata yapmanın önüne geçiliyor.

45. Uygulama – SPI ile Veri Alışverişi

SPI protokolünde bir adet ana aygıt bulunmakta ve bu ana aygıt da eğer başka bir bilgisayar sistemine bağlanmıyorsak kullandığımız mikrodenetleyici olmaktadır. Kullandığımız mikrodenetleyici bağlı olan istediği aygıta istediği bilgiyi istediği zamanda rahatça gönderebilir. Ama bu aygıtlar mikrodenetleyiciye bir bilgi göndermek istediği zaman mikrodenetleyiciden "Şimdi söyle!" mesajıyla izin almak zorundadır. Yani öncelikle uydu aygıta bu mesaj verisi gönderilmekte ve sonrasında ise bu aygıt dinlenilmektedir.

```
#include <avr/io.h>
unsigned char spi_send_recieve(unsigned char data);
void spi_init(void);
int main(void)
{
      spi_init();
      while (1)
      {
            unsigned char data = spi_send_recieve();
      }
}
```

```
void spi_init(void)
{
      /* MOSI ve SCK Çıkış, Diğerleri Giriş*/
      DDRB |= (1<<DDB3)|(1<<DDB5);
      DDRB |= (1<<DDB2); // SS ÇIKIŞ ÖNEMLİ
      /*SPI Enable, Master Mod, Fcpu/16*/
      SPCR |= ((1<<SPE)|(1<<MSTR)|(1<<SPR0));
}

unsigned char spi_send_recieve(unsigned char data)
{
      /* Veriyi Yolla*/
      SPDR = data;
      /* Bitmesini Bekle */
      while(!(SPSR & (1<<SPIF)))
      ;
      return SPDR;
}
```

Veri okumanın veri gönderme ve dinleme şeklinde yapılmasına "*transceive*" yani veri alışverişi adı verilmektedir. Bizim alışık olduğumuz UART protokolü gibi protokollerden en büyük farkı burada görebiliriz.

46. Uygulama – SPI Birimini Uydu Modunda Kullanma

Burada mikroişlemciyi uydu modda kullanırken programda ise sadece veri almaktayız. Yani ana aygıt olarak kullandığımız karttan gelen veriyi okuyup bunun üzerinden işlem yapıyoruz.

```
#include <avr/io.h>
void spi_slaveinit(void);
unsigned char spi_read(void);
int main(void)
{
    DDRD |= ((1<<PORTD2) | (1<<PORTD4));
      spi_slaveinit();
    while (1)
    {
            unsigned char data = spi_read();
          if (data == '1')
            PORTD ^= (1<<PORTD2);
            if (data == '2')
            PORTD ^= (1<<PORTD4);
```

```
	}
}


void spi_slaveinit(void)
{
	DDRB |= (1<<DDD4);
	SPCR = (1<<SPE);
}
unsigned char spi_read(void)
{
	while(!(SPSR & (1<<SPIF)))
	;
	return SPDR;
}
```

Burada MISO ayağı olan D portunun 4. Ayağını çıkış yapıyoruz. Bu çıkış yapılan ayakla ana aygıta veri gönderebiliriz. Diğer bütün ayaklar ise ana aygıttan gelen sinyalleri okuyacağı için giriş olarak bırakılmaktadır. Burada D portunun 2 ve 4 numaralı ayaklarına birer adet LED bağlanmıştır ve ana aygıttan gelecek '1' ve '2' karakterlerine göre bunlar yakılıp söndürülmektedir. Ana aygıtın kodunu ise bir sonraki uygulamada göreceğiz. Bu uygulamayı yapabilmek için iki ayrı karta ihtiyacınız olacaktır.

47. Uygulama – Ana Aygıttan Uydu Aygıta Veri Gönderme

Bu uygulamada gönderdiğimiz verilerle bizim yazdığımız programı çalıştıran uydu aygıtı kontrol etmekteyiz. Uydu aygıtta ayarladığımız şekilde '1' ve '2' karakterlerini yollamaktayız.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
unsigned char spi_send_recieve(unsigned char data);
void spi_init(void);
int main(void)
{
	spi_init();
	while (1)
	{
		spi_send_recieve('1');
```

```
            _delay_ms(500);
            spi_send_recieve('2');
            _delay_ms(500);
            spi_send_recieve('1');
            _delay_ms(500);
            spi_send_recieve('2');
            _delay_ms(500);
      }
}

void spi_init(void)
{

      DDRB |= (1<<DDB3)|(1<<DDB5);
      DDRB |= (1<<DDB2);
      /*SPI Enable, Master Mod, Fcpu/16*/
      SPCR |= ((1<<SPE)|(1<<MSTR)|(1<<SPR0));
}

unsigned char spi_send_recieve(unsigned char data)
{
      /* Veriyi Yolla*/
      SPDR = data;
      /* Bitmesini Bekle */
      while(!(SPSR & (1<<SPIF)))
      ;
      return SPDR;
}
```

48. Uygulama – SPI ile Shift Register Kontrol Etme

Shift Register (*Ötemeli yazmaç*) özellikle çıkış portlarını çoğaltmak, yüzlerce LED sürmek, LED pano kullanmak gibi işlerde bize oldukça yardımcı olmaktadır. Üstelik bunu veri ve saat sinyali gibi çok az ayak kullanarak yapmaktayız. Burada 75HC595 shift register entegresini kullandım. Burada SCK ayağı saat ayağı olan SH_CP ayağına, MOSI ayağı veri ayağı olan DS ayağına ve ek olarak ST_CP olan *latch* ayağı da B portunun 2. Ayağına bağlanmıştır.

```
#include <avr/io.h>
void spi_transmit(unsigned char data);
void spi_init(void);
#define  F_CPU 16000000UL
```

```
#include <util/delay.h>
int main(void)
{
	spi_init();
	while (1)
	{
		/*
		PORTB &= ~(1<<PORTB2);
		spi_transmit(0xFF);
		PORTB |= (1<<PORTB2);
		_delay_ms(500);
		PORTB &= ~(1<<PORTB2);
		spi_transmit(0x00);
		PORTB |= (1<<PORTB2);
		_delay_ms(500);
		*/
		for (int i = 0; i<255; i++)
		{
			PORTB &= ~(1<<PORTB2);
			spi_transmit(i);
			PORTB |= (1<<PORTB2);
			_delay_ms(80);
		}
	}
}

void spi_init(void)
{
	/* MOSI ve SCK Çıkış, Diğerleri Giriş*/
	DDRB |= (1<<DDB3)|(1<<DDB5);
	DDRB |= (1<<DDB2); // SS ÇIKIŞ ÖNEMLİ
	/*SPI Enable, Master Mod, Fcpu/16*/
	SPCR |= ((1<<SPE)|(1<<MSTR)|(1<<SPR0));
}

void spi_transmit(unsigned char data)
{
	/* Veriyi Yolla*/
	SPDR = data;
	/* Bitmesini Bekle */
	while(!(SPSR & (1<<SPIF)))
	;

}
```

Yukarıda yorum haline aldığım kod blokunda LED yakıp söndürme programı yer almaktadır. Burada ise binary sayıcı olarak Q çıkışlarına bağlı LED'ler binary

değer ifade edecek şekilde yanmaktadır. Görüldüğü gibi kullanımı oldukça basittir. Sadece 3 ayak bağlayıp veri göndermekten ibarettir.

49. Uygulama – SPI Kütüphanesi

Github'da SPI birimini kullanan örnek kodları incelediğimde AVR için güzel bir SPI kütüphanesi olmadığını fark ettim. Hem bu eksikliği gidermek adına hem de ileri yapacağım projelerde kullanmak için bir SPI kütüphanesi yazma kararı aldım. SPI birimi oldukça basit olduğu için bu kütüphaneyi yazmak hiç uzun sürmedi. Öncelikle kütüphane ile yapılmış örnek bir uygulamaya bakalım.

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

int main(void)
{
    spi_init(MASTER);
    while (1)
    {
            spi_transceive(0x01);
    }
}
```

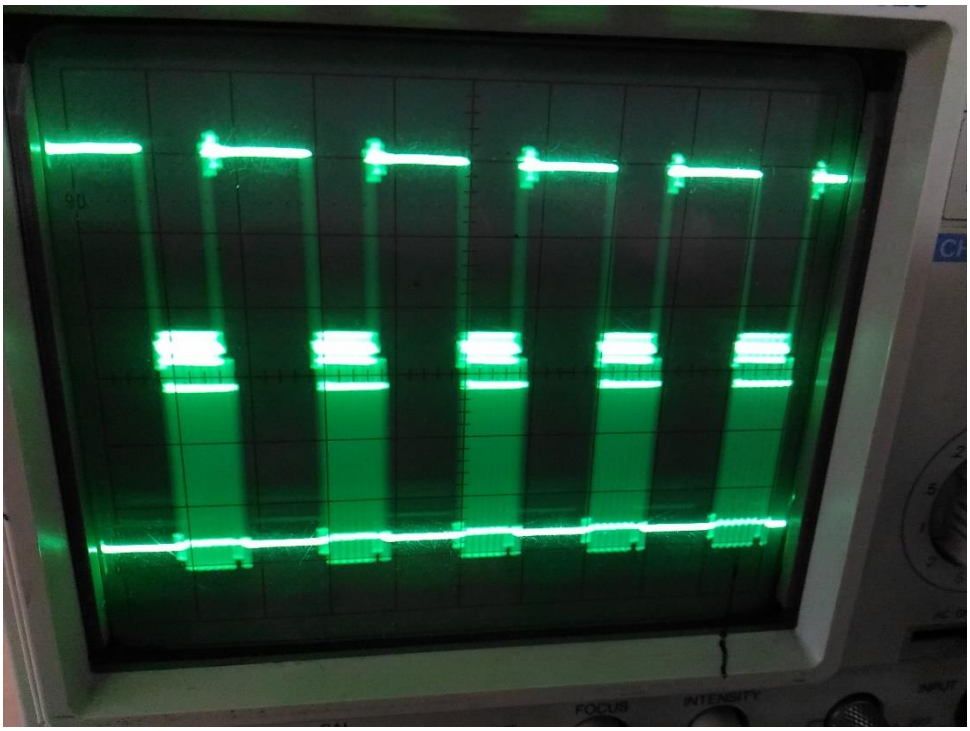
Görüldüğü gibi iki fonksiyonla hiç datasheet okumadan SPI birimini kullanabiliyoruz. Sanıldığı gibi her zaman bir tarafta datasheeti açıp tek tek bitlere bakarak yazmaçların bitleriyle oynayarak program yazmanız şart değildir. Bir entegre veya çevre birimini kullanmadan önce bunun sürücüsünü yazmak oldukça zaman kazandıracaktır. Defalarca kullanacağınız bir birim olduğu zaman her defasında zaman kaybetmenizin önüne geçer. Gömülü sistemler dünyasında en gerekli yazılımlar olan bu sürücülerin bile genellikle şahsa, aygıta göre yazıldığını ve çoğu zaman farklı aygıt ve uygulamalarda sıkıntı çıkardığını görmekteyim. Github'dan arama yapmayı ihmal etmememe rağmen kimsenin kütüphanesine kolay kolay güvenip kullanamıyorum.

```
#ifndef SPI_H_
#define SPI_H_
// USER CONFIGURATIONS
```

```
#define SPI_DATA_ORDER MSB_FIRST
#define SPI_DEFAULT_MODE SPI_MODE_0
#define SPI_CLOCK_RATE SPI_CLOCKDIV_4
#define SPI_SPEED SPI_DOUBLESPEED

// Macros and Functions
#define spi_deinit() SPCR &= ~(1<<SPE) /* deinit SPI, if you need */
void spi_init(uint8_t mode);
uint8_t spi_transceive(uint8_t data);
void spi_interrupt(uint8_t status);
void spi_mode(uint8_t mode);
unsigned char spi_read(void);




// Definitions
#define SLAVE 0
#define MASTER 1

#define ENABLE 1
#define DISABLE 0

#define MSB_FIRST 1
#define LSB_FIRST 0

#define SPI_MODE_0 0
#define SPI_MODE_1 1
#define SPI_MODE_2 2
#define SPI_MODE_3 3

#define SPI_CLOCKDIV_4 0b00
#define SPI_CLOCKDIV_16 0b01
#define SPI_CLOCKDIV_64 0b10
#define SPI_CLOCKDIV_128 0b11

#define SPI_DOUBLESPEED 1
#define SPI_REGULARSPEED 0




#endif /* SPİ_H_ */
```

SPI.H dosyasına baktığımızda yine en başta kullanıcı konfigürasyonlarını görmekteyiz. SPI birimini kullanmadan önce buradan ayar yapmanız gereklidir.

Datasheette bulunan bütün ayarlara yer verdim ve SPI donanımının sınırlarını zorlayarak kullanmanız mümkün. Ben en son hızda kullanmak isteyince saniyede yaklaşık 5 megabaytlık bir hız elde ettim. Bunu osiloskop ekranında ise şöyle gözlemledim.



Burada bir mikrosaniyelik zaman diliminde gerçekten 5 baytın gönderildiğini görmekteyiz.

Kütüphanede ise ayarlar standart kabul edilen değerlerde gelmektedir. İhtiyaç olduğunda bunları değiştirmeniz ve kullanacağınız aygıt ile uyumlu hale getirmeniz gereklidir. Kütüphane ayarlarından göreceğiniz üzere tek bir SPI standardı olmayıp toplamda 4 ayrı mod yer almaktadır. Kütüphane fonksiyonlarını açıklayarak devam edelim.

```
void spi_init(uint8_t mode);
```

Bu fonksiyonda SPI birimi istenilen modda (Master ya da Slave) başlatılmaktadır.

```
uint8_t spi_transceive(uint8_t data);
```

Bu fonksiyon yukarıda anlattığımız gibi ana aygıt için gönderme ve okuma işlemini beraber gerçekleştirmektedir.

```
void spi_interrupt(uint8_t status);
```

Bu fonksiyon ile SPI kesmesini etkinleştirebilir veya devre dışı bırakabiliriz.

```
void spi_mode(uint8_t mode);
```

Bu fonksiyonla SPI modu belirlenebilir.

```
unsigned char spi_read(void);
```

Bu fonksiyonla ise SPI protokolü üzerinden okuma işlemi gerçekleştirilmektedir.

Bu kütüphane sürücü olarak donanımın bütün özelliklerini kullanmaya yönelik olduğu için bu sürücüyü kullanarak daha gelişmiş kütüphaneler yazabilirsiniz. SPI.C dosyasını kalabalık olmasın diye buraya koymayacağım. İlgili proje dosyasında bulup inceleyebilirsiniz.

50. Uygulama – I2C Veri Gönderme Uygulaması

SPI protokolünün ardından I2C protokolüne geçiş yapıyoruz. Yalnız I2C protokolünün hiç de kolay olmadığını söylemem gerekir. SPI protokolü gibi birkaç ayar yapıp basit bir şekilde veri gönderemiyoruz. Biraz karmaşık olsa da I2C protokolünü öğrenip kullanmamız büyük önem arz ediyor.

Not: Hatta diyebilirim ki mikrodenetleyiciye dair bir şey öğrenmeyip sadece I2C ve SPI protokollerini öğrenseniz bile geliştiricilik yapabilmeniz mümkündür. Bunun sebebine gelirsek mikrodenetleyicinin içinde yer alan bütün birimler ve çok daha fazlası zaten SPI ve I2C uyumlu entegreler halinde satılmaktadır. Üstelik zamanlayıcı aradığınız zaman öyle 8 ya da 16 bitlik değil 64 bitlik zamanlayıcılar karşınıza çıkmakta, bir ADC kullanmak istediğiniz zaman 10-bitlik değil 16, hatta 24 bitlik bile ADC'ler karşınıza çıkmaktadır. Bunun gibi mikrodenetleyici içerisinde hiç göremediğiniz veya o kadar ileri seviye olmayan bütün entegreleri SPI ve I2C protokolleri ile kullanma imkânınız vardır. Tek dezavantajı maliyet, dahili birime göre bazen yavaş kalma ve muhtemel performans sorunlarıdır. Bu misali bu prokollerin önemine dikkat çekmek için veriyorum, yoksa bütün konuları öğrenmeniz gereklidir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
```

```
void i2c_init();
void i2c_start();
void i2c_write(unsigned char data);
void i2c_stop();
int main(void)
{
      i2c_init();
      i2c_start();
      i2c_write(0x3F << 1);
      i2c_write(0x00);
      i2c_stop();
      while(1)
      {

      }
}


void i2c_init()
{
      TWSR = 0x00;
      TWBR = 0x72;
      TWCR = 0x04;
}

void i2c_start()
{
      TWCR = ((1<<TWINT) | (1<<TWSTA) | (1 << TWEN));
      while ((TWCR & (1<<TWINT)) == 0);
}

void i2c_stop()
{
      TWCR = ((1<<TWINT) | (1<<TWEN) | (1<<TWSTO));
}

void i2c_write (unsigned char data)
{
      TWDR = data;
      TWCR = ((1<<TWINT) | (1<<TWEN));
      while ((TWCR & (1<<TWINT)) == 0);
}
```

Burada I2C biriminin en basit haliyle kullanımını görüyoruz. Yalnız bu birimin özellikleri bundan ibaret değildir. Uydu modunda kullanma veya bilgi mesajları gibi özellikler de vardır. Burada pek çok özelliği bir kenara bırakıp basit bir gönderme işi yapılacaktır

Bunun için öncelikle *i2c_init()* fonksiyonu ile bu birim başlatılmıştır. Şimdi bu fonksiyonun içine bakalım.

```
TWSR = 0x00;
TWBR = 0x72;
TWCR = 0x04;
```

Öncelikle datasheeti açtığımızda I2C diye bir birimden bahsedilmediğinin farkına varacaksınız. Bazı yerde TWI bazı yerde de *2-wire* olarak ifade edilen bu protokol aslında Philips'in patentli protokolünün isim haklarından dolayı farklı şekilde ifade edilebilmektedir. Aynı pratikte *Javascript* dediğimiz dilin *Java* ile isim benzerliğinden dolayı resmiyette *ECMAScript* olarak ifade edilmesi ve spesifikasyonunun da bu başlıkta olması gibidir. İşlev bakımından ise I2C ile bir farkı yoktur.

Öncelikle TWSR yazmacının değeri 0 yapılarak I2C ön bölücü değeri 1 yapılmaktadır. I2C protokolü bu bölücüler vasıtasıyla çeşitli hızlarda çalıştırılabilmektedir. Bazen hızı yavaşlatmanız gerekebilir.

TWBR yazmacına ise datasheette yer alan formüle göre hesaplanan değer bit oranını belirtmek için yazılmaktadır. Daha önceden ayarladığımız ön bölücü değeri de bu formüle dahil edilmelidir. Bu formül datasheette şu şekilde yer alır.

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot (PrescalerValue)}$$

En son olarak TWCR yazmacındaki TWEN biti bir (1) yapılarak I2C birimi etkinleştirilmektedir. Bundan sonra *i2c_start()* fonksiyonunun içine bakalım.

```
TWCR = ((1<<TWINT) | (1<<TWSTA) | (1 << TWEN));
while ((TWCR & (1<<TWINT)) == 0);
```

Burada TWINT bayrak biti üzerine bir (1) yazılarak sıfırlanmaktadır. Bu iletişimin başlaması için gereklidir. Sonrasında TWEN biti bir yapılırken bir yandan da START durumunun başlatılması için TWSTA biri bir yapılmaktadır. START durumu başladıktan sonra veri aktarımını yapabiliriz. Şimdi sırada *i2c_write()* fonksiyonu var.

```
TWDR = data;
TWCR = ((1<<TWINT) | (1<<TWEN));
while ((TWCR & (1<<TWINT)) == 0);
```

Sonrasında I2C veri yazmacı olan TWDR yazmacına veri yazılmakta ve sonrasında kesme tabanlı çalıştığı için TWEN biti bir yapılırken bu kesme bayrak biti de sıfırlanmaktadır. Hatta sonrasında döngü içerisinde bu kesme bayrak bitinin sıfırlanması beklenmektedir. İletişimi bitirmek için ise *i2c_stop()* fonksiyonunu kullanırız.

```
TWCR = ((1<<TWINT) | (1<<TWEN) | (1<<TWSTO));
```

Burada TWSTO biti bir (1) yapılarak STOP durumuna geçilmektedir. Şimdi iletişim boyunca hangi verilerin gönderildiğine bir bakalım.

```
i2c_init();
i2c_start();
i2c_write(0x3F << 1);
i2c_write(0x00);
i2c_stop();
```

Öncelikle 0x3F << 1 diyerek adres verisini gönderiyoruz. Adres değeri 7-bit ve sola hizalı olduğu için bunu bir bit sola kaydırmak gereklidir. Kullandığınız aygıtın adres değerini kolayca bulmak için Arduino için yazılmış *i2cscanner* uygulamasını kullanabilirsiniz. Aşağıdaki bağlantıdan bu programa ulaşabilirsiniz.

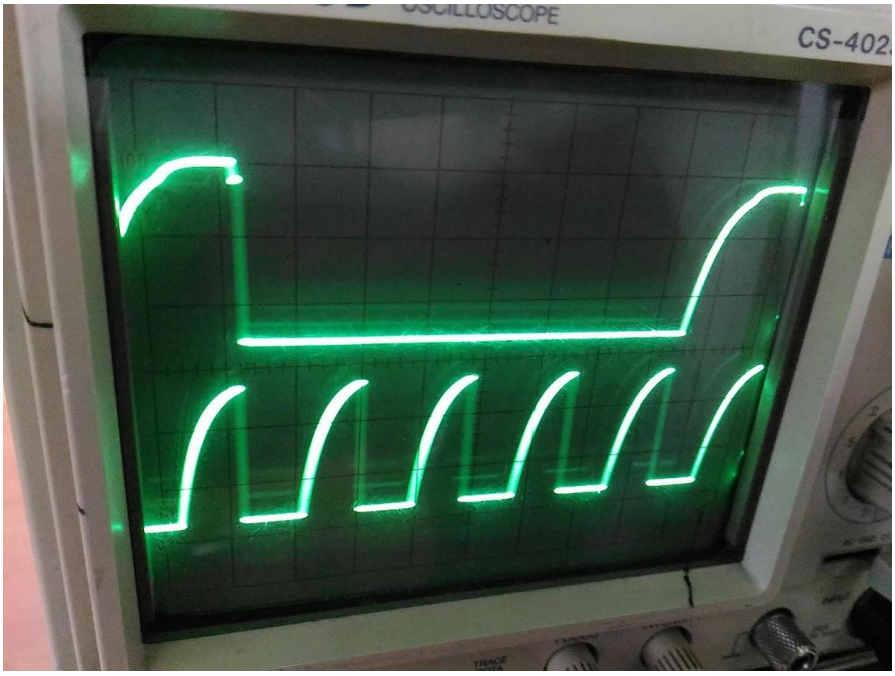
https://playground.arduino.cc/Main/I2cScanner/

Ben burada LCD ekranlar için kullanılan PCF8574 entegresini kullandığım için bu entegre 0x3F adresine sahipti. Burada önemli nokta *i2cscanner* programı ile öğrendiğiniz adres değerini bir adım sola kaydırmanız gerektiğidir.

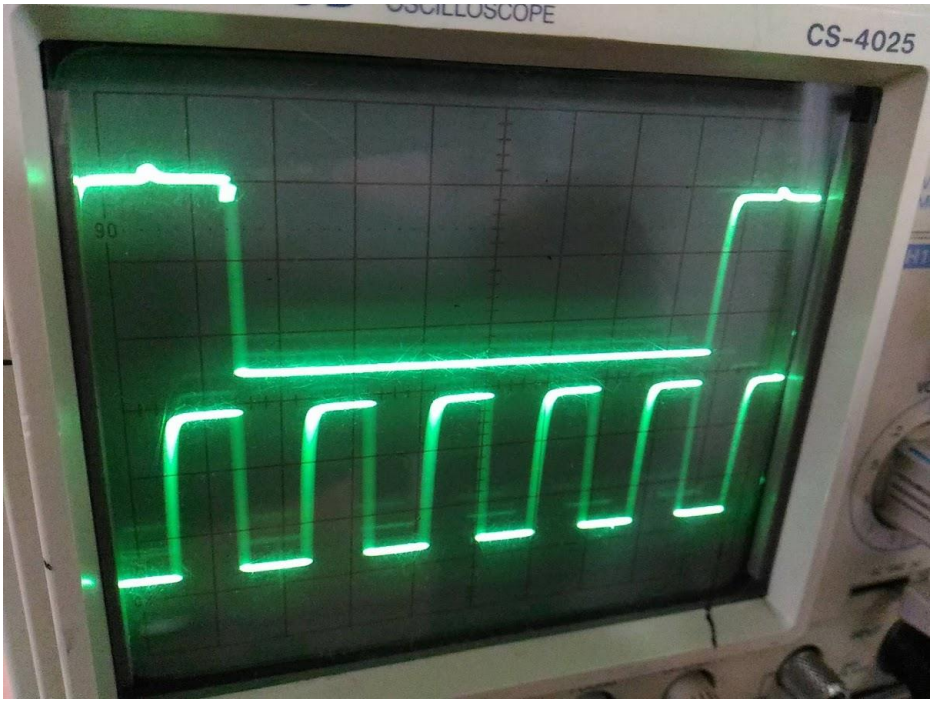
Önce adres değerini yazdık, sonrasında ise veri değerini yazmamız gerekli. Ben burada 0x00 değerini yazarak bütün ayakları 0 konumuna getiriyorum. Bu değeri

değiştirerek, mesela 0xFF yaparak bütün ayakları yapabilirsiniz. Bu uygulamayı bu değeri değiştirerek deneyebilirsiniz. I2C bağlantısının Arduino'nun A4 ve A5 ayakları üzerinden yapıldığını ve bu ayaklara 4.7K'lık bir *pull-up* direnci koymanız gerektiğini unutmayın.

Neden *pull-up* direncinin gerekli olduğunu da osiloskop ekranından göstereyim.



Bu resimde *pull-up* direnci kullanmadan elde ettiğim sinyali görmektesiniz. Görüldüğü gibi kare dalga olmaktan çok uzaktadır.



Bu resimde ise 4.7K direnç ile *pull-up* yaptığımda elde ettiğim sinyali görmektesiniz. *Pull-up* direncinin değeri düştükçe sinyal iyice düzelmektedir. Eğer fazlaca aygıt bağlıyorsanız daha düşük değerde *pull-up* dirençleri kullanın.

51. Uygulama – I2C Kütüphanesi ile Veri Gönderme

Bu sefer daha öncesinde karakter LCD ve UART kütüphanelerini kullandığımız Peter Fluery'in I2C kütüphanesini kullanacağız. Bu sayede bir kütüphane ile işler daha da kolaylaşacak.

```
#include <avr/io.h>
#include "i2cmaster.h"
#define F_CPU 16000000UL
#include <util/delay.h>
int main(void)
{
    i2c_init();

    while (1)
    {
            i2c_start(0x3F << 1);
            i2c_write(0xFF);
            _delay_ms(500);
            i2c_write(0x00);
            _delay_ms(500);
            i2c_stop();

    }
}
```

Burada yine *i2c_start()* fonksiyonu ile yazdığımız adres değeri ile START durumuna geçiyoruz ve ardından veri yazmaya başlıyoruz. İletişimi bitirmek için de *i2c_stop()* fonksiyonu kullanılıyor. Adres belirtirken yine bir bit sola kaydırmayı unutmayın.

52. Uygulama – Atmel'in I2C kütüphanesi ile Veri Gönderme

Bu sefer farklı bir kütüphane kullanma adına Atmel'in uygulama notu ile beraber verdiği TWI Master kütüphanesini kullandım. Bu kütüphane de alternatif olarak kullanılabilir. Uygulama notuna şu bağlantıdan erişebilirsiniz.

https://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en591794

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "TWI_Master.h"
#define F_CPU 16000000UL
#include <util/delay.h>
int main(void)
{
    TWI_Master_Initialise();
      sei();
    while (1)
    {
      unsigned char buf[2];
      buf[0] = 0x3F << 1;
      buf[1] = 0xFF;
      TWI_Start_Transceiver_With_Data(buf, 2);
      _delay_ms(500);
      buf[1] = 0x00;
      TWI_Start_Transceiver_With_Data(buf, 2);
      _delay_ms(500);
    }
}
```

Bu uygulamada bir dizi oluşturup dizinin ilk elemanına adres değerini ekledikten sonra sırayla göndereceğimiz verileri diziye yerleştiriyoruz. Ardından bu diziyi fonksiyona argüman olarak atıyoruz. Bu uygulama da önceki uygulamada olduğu gibi PCF8574 entegresini kullanmıştır. Bu entegrenin çıkışına bağlı LED'ler 500 milisaniye aralıkla yanıp sönecektir.

53. Uygulama – I2C LCD Kütüphanesi

Elimdeki karakter LCD'leri I2C modülleriyle kullanmak için bir kütüphane arayışına girdim ve en sonunda çalışan bir kütüphane bulabildim. Projenin içinde yer alan kütüphane dosyalarıyla ufak bir örnek yazdım. LCD modülünün adresi her zaman 0x3F olmadığı için *lcd1602.h* dosyasından bu adresi değiştirmeniz gerekebilir.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include "lcd1602.h"

int main(void)
{
    lcd1602_init();
      lcd1602_send_string("Merhaba Dunya");
    while (1)
    {

    }
}
```

54. Uygulama – I2C Uydu Aygıt (Slave) Kütüphanesi

I2C protokolünü uydu aygıt olarak kullanmak için bu projede yer alan kütüphaneyi kullanabilirsiniz. Bu kütüphane sorunsuz çalışmaktadır. Aşağıda bu kütüphane fonksiyonları ile yazdığım basit bir örneği görebilirsiniz.

```
#include "I2CSlave.h"

#define I2C_ADDR 0x2F

volatile uint8_t data;

void I2C_received(uint8_t received_data)
{
      data = received_data;
}

void I2C_requested()
{
      I2C_transmitByte(data);
```

```
}

void setup()
{
	// set received/requested callbacks
	I2C_setCallbacks(I2C_received, I2C_requested);

	// init I2C
	I2C_init(I2C_ADDR);
}

int main()
{
	DDRD = 0xFF;

	setup();
	while(1)
	{
		PORTD = data;
	}
}
```

Burada okunan veri PORTD'den çıkış olarak alınmaktadır. Önceki örnekleri de kullanarak iki Arduino'yu birbirine bağlayabilir ve PORTD'ye bağlayacağınız LED'ler ile bir uygulama yapabilirsiniz. I2C protokolü üzerinde çalışırken çok zaman kaybetmemek için hazır kütüphane aramayı ve kullanmayı tercih ettim. İleri zamanlarda daha derinlemesine inceleyeceğim.

EK: AVR Sigorta Ayarları

AVR denetleyicileri kullanırken programdan hariç bu mikrodenetleyicilerin sigortaları olduğunu ve sigorta bitleriyle önemli ayarların yapıldığını unutmayın. Bu sigorta bitlerini programlayıcı vasıtasıyla çeşitli arayüz programlarından değiştirebilirsiniz. Eğer AVRISP mkII kullanıyorsanız Atmel Studio'daki ara yüzden bu ayarları yapabilirsiniz. Eğer *usbasp* gibi bir programlayıcınız varsa en iyisi AVRDUDESS programını kullanmaktır. Bu iki programın yanında sigorta hesaplayıcı yazılımları da kullanmanız gerekir. Buradan seçenekleri seçtiğinizde yazdırmanız gereken sigorta değeri karşınıza çıkmaktadır. Benim kullandığım hesaplayıcıya aşağıdan erişebilirsiniz.

https://eleccelerator.com/fusecalc/fusecalc.php?chip=atmega328p

Bu sigorta ayarları sayesinde kod korumayı açabilir, Brown-out ayarını yapabilir ve en önemlisi olan saat ayarlarını kontrol edebilirsiniz. Eğer saat ayarları düzgün yapılmazsa saate bağlı bütün uygulamalar yanlış çalışacak, mikrodenetleyiciden istenilen hız alınmayacaktır. Mesela "*Divide clock by 8 internally*" seçeneği kazara işaretlendiği zaman mikrodenetleyici 8 kat daha yavaş çalışacaktır. O yüzden projelerde sigorta ayarlarını ihmal etmemek gereklidir. Arduino UNO kartındaki denetleyici Arduino'ya göre sigorta ayarlarıyla geldiği için kod koruma haricinde bir değişiklik yapmaya gerek yoktur. Kod koruma içinse C0 değerini yazdırmanız yeterlidir.

EK 2: Malzeme Listesi

- **2 adet tactile düğme (mümkünse iki bacaklı olanlar)**
- **1 adet breadboard ya da UNO uyumlu breadboard tahtası**
- **2 adet Arduino UNO (klon da olabilir)**
- **2 adet pull-up direnci (4.7k/10k)**
- **8 adet kırmızı LED**
- **8 adet 220/330 Ohm direnç**
- **1 adet ortak katot RGB LED**
- **1 adet ortak katot 7 segman LED gösterge**
- **1 adet potansiyometre**
- **1 adet 16x2 karakter LCD**
- **1 adet I2C LCD modül**
- **1 adet LDR**
- **1 adet LM32**
- **1 adet hoparlör**
- **1 adet 75HC595**
- **1 adet PCF8574**

Son Söz

Aslında Udemy'de yayınlayacağım AVR eğitimi için yaptığım uygulamaları yeterli zamanım olmamasından dolayı kitaplaştırma kararı almıştım ve 3 günün sonunda böyle bir kitap ortaya çıktı. Bazı uygulamaların eksik kaldığını, eksiksiz bir kitap olmadığını bilsem de en önemli kısımları anlattığımı düşünüyorum. Burada yer alan kod örnekleri ve "C ile AVR Programlama" yazılarımdan istifade ederek profesyonel anlamda projeler yapabileceğinize inanıyorum. Bu kitabın ikinci cildi diyebileceğim ve AVR ile yaptığım projeleri anlattığım bir kitabı ilerisi için düşünmekteyim. O zamana kadar bu uygulamalardan faydalanmanız için bu kitabı yazdım.

Kitapta bir hata, yanlışlık ile karşılaştığınızda iletişim kanallarından lütfen bana bildiriniz. Aynı zamanda kitap hakkında görüş, öneri ve yorumlarınızı bekliyorum.

Kitapta geçen uygulamaların proje dosyaları aşağıdaki bağlantıda yer almaktadır.

https://github.com/GDokmetas/AVRUygulamalari