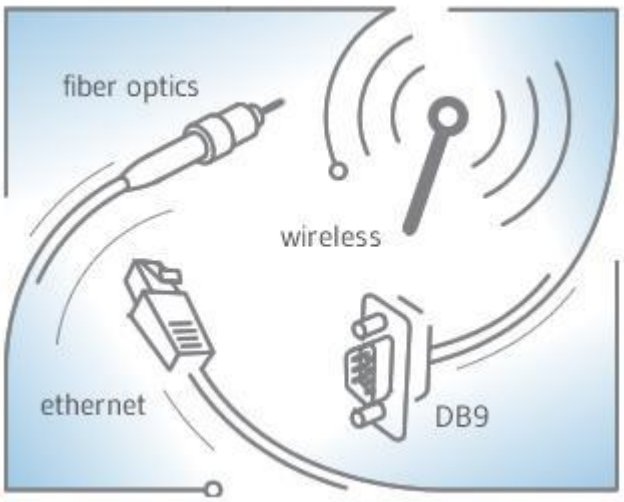


Gömülü Sistemlerde Cihazlar Arası Haberleşme: Kendi İletişim Protokolümüzü Geliştiriyoruz

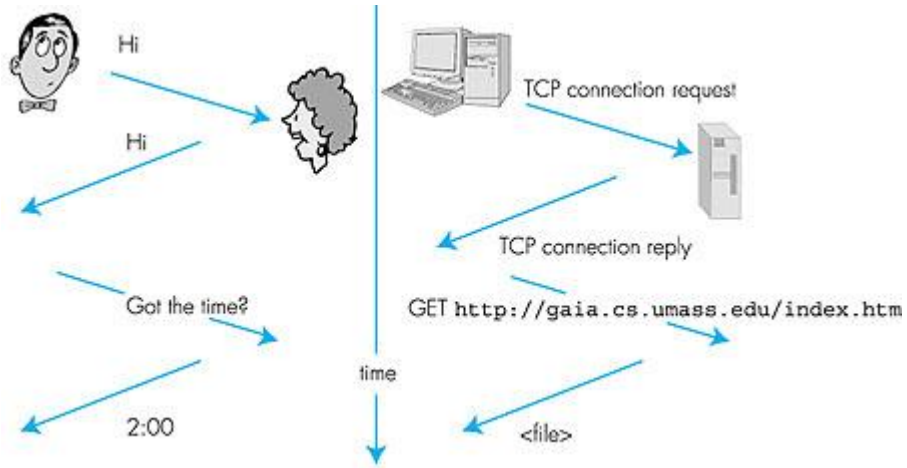
2017

Fatih Özcan
ozcanfatih.wordpress.com

Kendi İletişim Protokolümüzü Geliştiriyoruz 1: Protokol Yapısı

Bu uygulama notunda gömülü sistemlerde önemli bir sorun olan, bir verinin bir mikrodenetleyiciden bir diğerine veya başka bir platforma aktarılması konusunu irdeleyeceğiz. Özellikle mikrodenetleyici programlaya yeni başlayanlar için büyük bir soru işareti olan seri iletişim ile bilgi transferi konusuna detaylıca değineceğim. Bunun için kendi protokol yapımız oluşturacağız. Elbette anlatılanlar sadece seri iletişimi değil tüm fiziksel iletişim katmanlarında kullanılabilir olacaktır.

Dijital dünyada cihazlar arası iletişim oldukça önemli bir konudur. Pek çok uygulamada iki cihaz arasında veri transferi ihtiyacı vardır. Veri bir uçtan diğer uca aktarılırken, alıcı tarafta gönderilen verinin alınması ve bu verinin işlenip anlamlandırılması gerekmektedir. Bu noktada cihazların (makinelerin) birbirleriyle anlaşabilmeleri yani konuşabilmeleri gerekmektedir. Nasıl ki biz insanlar diller vasıtasıyla konuşup anlaşıyorsak makineler içinde aynısı geçerlidir. Bizler için İngilizce, Almanca, Türkçe vb. diller varken makineler içinde insanlar tarafından ModBus, M-Bus, Profibus, DLMS, MQTT, TCP, FTP, HTTP gibi diller (haberleşme protokolleri) geliştirilmiştir. İki farklı cihaz birbiri ile bu protokoller sayesinde konuşabilmektedir.



Tabi burada önemli olan nokta iki tarafında aynı dili konuşabiliyor olması yani aynı protokolü desteklemesidir. Geçmişteki ihtiyaç ve daha önceki tecrübeler ile mühendisler makineler için yukarıdaki gibi çeşitli protokoller geliştirmişlerdir. Bu protokollerin hepsi yeni bir ihtiyaç ve gereksinimden dolayı ortaya çıkmıştır. Örneğin Modbus protokolü oldukça basitken bir TCP paketi daha karmaşık yapıdadır. Gömülü sistemlerde bizlerde hemen hemen her projede bir fiziksel iletişim katmanı kullanarak(USB, Seri İletişim v.b) iki cihaz arasında veri alışverişi yapmak isteriz. Buda iki cihaz arasında bir protokol kullanarak yapılabilir. Mevcut protokoller bu durumda kullanılabilir olsa da bazen bu protokol yapıları amacımız için gereksiz kontrol yapılarına, uzun veri paketlerine ihtiyaç duyduğundan bunları kullanmak istemeyiz. Bu durumda kendi protokol yapımızı geliştirme ihtiyacı doğar. Aşağıda endüstride kabul görmüş bir protokol yapısı bulunmaktadır.

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Şekil: Modbus ASCII paket yapısı (frame format)

Yukarıda çok eski ve halen popüler olan Modbus ASCII protkolünün “frame” yapısı görülmektedir. Paket oldukça basit bir yapıya sahiptir. Paket; bir byte başlangıç verisi ardından iki byte adres bilgisi, bu paket hakkında ayrıntılı bilgi veren iki byte fonksiyon bilgisi, n byte bilgi verisi, iki byte hata kontrol verisi ve

son olarak da iki byte bitiş verisinden oluşmaktadır. Bu veriler sıralı şekilde hatta gönderilmektedir. Modbus ASCII standardında başlangıç için ':' karakteri, bitiş içinde "CR" ve "LF" karakterleri gönderilmektedir. Hata kontrolü içinse CRC-16 (Cyclic Redundancy Check) hata bulma mekanizması kullanılmaktadır. Adres bilgisi alıcının adresini içerirken, fonksiyon bilgisi de yine modbus standartları ile tanımlanmış özel anlamlar ihtiva etmektedir.

Şimdi bizde kendi iletişim protokolümüzü oluşturmaya başlayalım. Temel bir protokol için en azından başlangıç ve bitiş karakterimizin olması, aslında çok basitte olsa bir protokol için yeterlidir. Yukarıdaki örnek ışığında kendi frame yapımızı belirleyelim. Aşağıda böyle bir yapı görülmektedir.

Başlangıç Verisi 0x01	N x byte Veri Paketi	Paket Bitiş Verisi 0x02

Eğer yukarıdaki gibi bir yapıda haberleşirsek iletişim hattında meydana gelen gürültüler sebebiyle veriler bozulursa hatalı veri alımı sorunu ortaya çıkmaktadır. Gelen veri bir sıcaklık verisi ise ve biz sınır değerlere göre işlemler yapıyorsak, gelen verinin bozulması, yanlış bir alarm veya alarm durumda sistemin çalışmaya devam etmesi gibi sorunları ortaya çıkartmaktadır. Bu durumda bizimde bir hata kontrol mekanizmasına ihtiyacımız vardır. Bu konuda en basit yöntemlerden birisi BCC (Block Check Character) dir. Temel olarak bu üç veri bir protokol oluşturmak için yeterlidir. Daha fazla ayrıntıya girmeden protokolümüze bu yapıya göre şekillendirelim.

Başlangıç Verisi 0x01	N x byte Veri Paketi	Hata Kontrol Verisi BCC	Paket Bitiş Verisi 0x04

Şekil: Paket yapısı

Başlangıç Verisi (Start of Header, **SOH**)	**0x01**
Hata Kontrol Algoritması	**BCC**
Paket Bitiş Verisi (End of Transmission, **EOT**)	**0x04**

Tablo: Paket verileri

Görüldüğü gibi paket yapımız oldukça sadedir. Hata kontrolü olarak BCC günü kurtarsa da daha iyi bir çözüm olarak CRC kullanılabilir Şimdi temel bir gönderim fonksiyonu yazalım.

```
#define BUFFER_SIZE  128
#define FRAME_START 0x01
#define FRAME_END    0x04

void send_package(const char *buff, uint8_t size)
{
   uint8_t bcc = 0;

   if(size > BUFFER_SIZE)
      return;


   serial_send_data(FRAME_START); // Paket başlıyor
```

```
    for (uint8_t i = 0; i < size; i++) //Paket verileri
    {
        bcc ^= buff[i];
        serial_send_data(buff[i]);
    }

    serial_send_data(bcc); //BCC
    serial_send_data(FRAME_END); //Paket sonu
}
```

Yukarıdaki fonksiyon için elimizde data hattına gönderilmek üzere hazırlanmış bir buffer'ın olduğu varsayılmıştır. Bu buffer ve uzunluğu fonksiyona gönderildiğinde, fonksiyon protokol için gerekli olan bilgileri hesaplayıp göndermektedir. Biz örneğimizde seri haberleşme kullanacağımızdan bu veriler byte byte UART donanımına "serial_send_data" fonksiyonu ile gönderilmektedir. Gönderim kısmı oldukça basit. Şimdi bizim yapımıza uygun olarak gelen paketleri alıp çözen bir fonksiyon yazalım.

```
typedef enum
{
   COM_STATE_IDLE,
   COM_STATE_COLLECT_DATA,
   COM_STATE_FRAME_ESC,
   COM_STATE_PACKAGE_READY
} com_state_t;

typedef struct
{
   uint8_t buffer[BUFFER_SIZE];
   uint8_t index;
   uint8_t bcc;
   com_state_t com_state;
} serial_t;

serial_t serial;

void reset_serial(void)
{
   memset(serial.buffer, 0x00, serial.index);
   serial.index = 0;
   serial.com_state = COM_STATE_IDLE;
}

void serial_interrupt_handler(uint8_t data)
{

   switch (serial.com_state)
   {
   case COM_STATE_IDLE:

      if (data == FRAME_START)                    /* Paket başlangıcı alindi. Gelen verileri topla */
```

```
            serial.com_state = COM_STATE_COLLECT_DATA;
        break;

    case COM_STATE_COLLECT_DATA:

      if (data == FRAME_END)                    /* Paket sonu */
            serial.com_state = COM_STATE_PACKAGE_READY;
        else
            serial.buffer[serial.index++] = data;       /* Paket datası */

        if (serial.index >= BUFFER_SIZE)            /* Buffer   dolarsa iletişimi resetle */
        {
            reset_serial();
            return;
        }
        break;

    case COM_STATE_PACKAGE_READY:
        //Paket hazırsa paket islenene kadar yeni veri alma.
        break;

    default:
        break;
    }
}

uint8_t is_package_ready(void)
{
    if (serial.com_state != COM_STATE_PACKAGE_READY)
        return 0;
    return 1;
}
```

Gelen paketi çözme işlemi biraz karışık gibi görünse de aslında yapılan iş basit. “serial_interrupt_handler” fonksiyonu kesme içerisinden her bir byte alındığında çağrılıyor. Bu fonksiyon içerisinde alınan verinin paketin hangi kısmı olduğunu anlamak ve ona göre işlem yapmak için bir “state machine” çalışıyor. Başlangıç verisi (SOH) gelince paketin asıl bilgi kısmını almak için state değişiyor ve bitiş verisi gelinceye kadar burada gelen veriler toplanıyor. Ardından paket hazır konumuna geçiliyor ve paketin işlenmesi için bekleniyor. Bir paket doğru şekilde alındıktan sonra yeni paket alınmıyor. Dolayısıyla gelen paket olup olmadığı ”is_package_ready” fonksiyonu ile sık sık kontrol edilmeli, paket alınmışsa işlenip yeni paket alınması için state machine resetlenmeli ve buffer temizlenmelidir. Elbette paket hazır olduktan sonra BCC kontrolü yapılıp alınan veride bozulma olup olmadığı kontrol edilmelidir.

Yukarıdaki haliyle aslında kendi protokolümüzü geliştirmiş olduk. Protokol ve yazılım katmanı bu haliyle kullanılabilir durumda. Fakat bu yapıda önemli eksiklikler bulunmaktadır. Eğer veri paketi içerisinde “paket bitiş verisi” bulunursa daha paket bitmeden veri alımını durdurmuş oluruz. Bu durumla karşılaşmamak için veri paketi içerisinde paket başlangıç ve bitiş verileri olmamalıdır. Bu noktada bir alternatifte sunmak istiyorum. Eğer BCC verisini paket sonu verisinden sonra gönderirsek ve veri paketi içerisinde de paket başı ve sonu verisinin geçmediğini garanti altına alırsak bu sorunla karşılaşmayız.

Bir diğer sorun ise veri iletiminin tam ortasında veri gönderen cihazdan veri gönderimi durursa, yani paket tam olarak alınmaz ise state machine bir sonraki paketi yanlış alacaktır. Bu sorun içinde bir "timeout" yapısına ihtiyaç vardır. Paket alımı başladıktan belirli bir süre sonra paket alımı bitmemişse yazılım tarafında alım kısmını resetlememiz gerekmektedir.

Kendi İletişim Protokolümüzü Geliştiriyoruz 2: Byte Stuffing

Bir önceki bölümde karşılaştığımız büyük bir sorun olan veri paketi içerisinde "frame" yapımızda kullandığımız özel verilerin(paket başlangıcı ve bitişi) olması durumunda hatalı paket alımı yaptığını görmüştük. Bu bölümde bu gibi sorunlara karşı geliştirilmiş olan "Byte Stuffing" algoritmasının çalışma mantığına bakıp kendi protokolümüze ekleyeceğiz.

Byte stuffing oldukça basit bir algoritma. Algoritmanın temeli; eğer paket içerisinde kullanılan özel bir veri varsa, bu veriden önce başka bir tanımlayıcı veri gönderip ardından asıl veriyi göndermeye dayanıyor. Yani kabaca; tanımlayıcı veriyi alan cihaza, "bak kardeşim bundan sonra gelen veri; paket başlangıcı, bitişi, hata kontrol verisi v.b ne olursa olsun değerlendirmeden buffer'a al ve normal akışına kaldığın yerden devam et" diyoruz. Dolayısı ile burada bir tanımlayıcı veriye ihtiyacımız vardır. Bir örnek üzerinde inceleyelim. Örneğimiz için tanımlayıcı verimiz (DLE: Data Link Escape) "0x10" olsun.

Orijinal Paket:

Byte Stuff İşleminden Sonraki Hali:

Görüldüğü gibi veri paketi içerisinde geçen "0x04" verisi önüne "0x10" eklenmiştir. Alıcı taraf bu veriyi değerlendirip ona göre işlem yapmalıdır. Dikkat edilirse BCC değişkeninin değeri de 0x01, 0x04 veya 0x10 de olabilir. Bu durumu değerlendirilip benzer şekilde BCC'den önce tanımlayıcı veri gönderilmelidir.

Eğer veri paketi içerisinde DLE verisi varsa yine öncelikle tanımlayıcı olan DLE, ardından paket bilgisi olan DLE gönderilir. Son bir örnek daha yapalım.

Orijinal Paket:

Byte Stuff İşleminden Sonraki Hali:

Kendi İletişim Protokolümüzü Geliştiriyoruz 3: İlk Haberleşme

Yukarıda ele aldığımız önlemeler de göz önüne alınarak geliştirilen kütüphane ve bununla yapılmış bir örnek proje ektedir.

Örneğimizde U2 mikrodenetleyicisinden gönderilen “LED_ON” ve “LED_OFF” komutları ile U1’ e bağlı D2 LED’i kontrol edilmiş ve aynı zamanda “GET_INFO” komutuyla da U1’den veri çekilmiştir. “GET_INFO” sorgusuna cevap alındığında ise U2’ye bağlı D4 LED’i blink yapmaktadır. Aşağıda simülasyon görüntüsü yer almaktadır. Örnek uygulamayı buradan indirebilirsiniz.

Alıcı kısımda, hattan gelen verinin nasıl işlendiği incelenirse çalışma mantığı daha iyi kavranır. Bu yapı aslında pek çok yazılım katmanında benzer şekilde kullanılmaktadır. Bir yerden bilinmeyen uzunlukta ve birden fazla mesaj geliyor ve gelen mesaja bağlı olarak işlemler yapılması isteniyorsa gelen mesajı tek tek kontrol etmek uygun bir çözüm olmayabilir. Protokol çok fazla komut ihtiva ediyorsa bu durumda tek tek kontrol yerine “Hash Tablosu”, “Binary Search” gibi çeşitli yöntemler kullanılmaktadır.

Aşağıda “byte stufing” destekli alım fonksiyonu ve gönderim fonksiyonu yer almaktadır.

Alım Fonksiyonu:

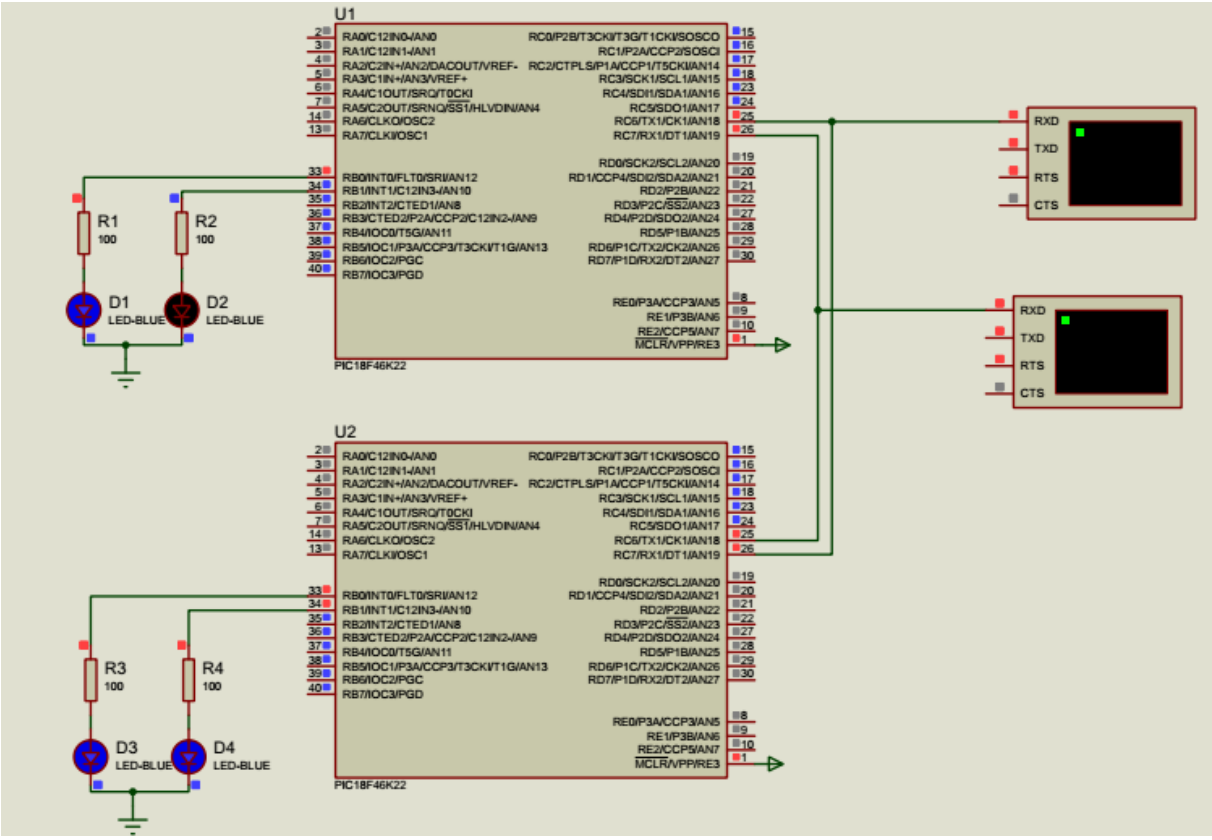
```
void serial_interrupt_handler(uint8_t data)
{
   switch (serial.com_state)
   {
   case COM_STATE_IDLE:

      if (data == FRAME_START) /* Paket başlangıcı alindi. Gelen verileri topla */
         serial.com_state = COM_STATE_COLLECT_DATA;
      break;

   case COM_STATE_FRAME_ESC:

      serial.buffer[serial.index++] = data; /* Frame ESC verisinden sonra pakette bulunan gerçek data gelir */
      serial.com_state = COM_STATE_COLLECT_DATA; /* Bir sonraki veri için tekrar ilgili case e git */
      if (serial.index >= BUFFER_SIZE) /* Buffer   dolarsa iletişimi resetle */
      {
         reset_serial_com ();
         return;
      }
      break;

   case COM_STATE_COLLECT_DATA:

      if (data == FRAME_ESC) /* Paket içinde başlangıç, bitiş verisi var !!! */
      {
         serial.com_state = COM_STATE_FRAME_ESC;
         return;
      }
      else if (data == FRAME_END) /* Paket sonu */
      {
         serial.com_state = COM_STATE_PACKAGE_READY;
         return;
      }
      else if (data == FRAME_START) /* Paket başlangıcı. Bu paket yarıda kaldı. Yeni paket alınacak */
```

```
        {
            reset_serial_com ();
            serial.com_state = COM_STATE_COLLECT_DATA; /* Yeni pakete başlanıyor,*/
            return;
        }
        else
            serial.buffer[serial.index++] = data; /* Paket datası */


        if (serial.index >= BUFFER_SIZE) /* Buffer dolarsa iletişimi resetle */
        {
            reset_serial_com ();
            return;
        }
        break;

    case COM_STATE_PACKAGE_READY:
        //Paket hazırsa paket islenene veya timeout zamanı dolana kadar yeni veri alma.
        break;

    default:
        break;
    }
}
```

Gönderim Fonksiyonu:

```
void serial_send_package(const char *buff, uint8_t size)
{
    uint8_t bcc = 0;

    if (size > BUFFER_SIZE)
        return;

    serial_send_data(FRAME_START); // Paket basliyor

    for (uint8_t i = 0; i < size; i++) //Paket verileri
    {
        if (buff[i] == FRAME_START || buff[i] == FRAME_END || buff[i] == FRAME_ESC)
        {
            bcc ^= FRAME_ESC;
            serial_send_data(FRAME_ESC);
        }

        bcc ^= buff[i];
        serial_send_data(buff[i]);
    }

    if (bcc == FRAME_START || bcc == FRAME_END || bcc == FRAME_ESC) /* BCC degeri herhangi bir
deger olabilir !*/
    {
        bcc ^= FRAME_ESC;
        serial_send_data(FRAME_ESC);
    }
```

```
    serial_send_data(bcc); //BCC
    serial_send_data(FRAME_END); //Paket sonu
}
```



Şekil: Simülasyon ekran görüntüsü

Ayrıntılı bir network oluşturmak için paket içerisine hedef cihazın ID'si (kimlik numarası), mesajın kimden geldiğinin anlaşılması için göndericinin ID'si eklenebilir. Elbette bu değişkenler veri paketi içerisine de eklenebilir. Fakat burada önemli nokta, bu bilgilerin yerinin/sıralamasının "frame" içerisinde değişmemesidir. Veri güvenliğinin önemli olduğu durumlarda veri paketi şifrelenmelidir. Yaygın ve güçlü bir şifreleme olan "AES" şifreleme bu iş için uygun olabilir.

Son olarak üç bölüm boyunca ele aldığımız "gömülü sistemlerde haberleşme" konusunun hedefinin seri iletişim olmadığını hatırlatmak isterim. Bu yapıyı her türlü fiziksel iletişim katmanının da kullanabiliriz.

Faydalı olması dileğiyle.