MODBUS Notlarım

MODBUS haberleşme protokolü üzerine aldığım notlar

Alper Yazar <ayazar@alperyazar.com>

Önsöz

Bu doküman, ihtiyaç gereği MODBUS protokolü (MODBUS RTU) ile çalışmam gereken zamanlarda ağırlıklı olarak resmi dokümanları temel alarak derlediğim notlarımı içermektedir. MODBUS oldukça eski ve yaygın kullanılan bir protokol olduğu için hem Türkçe hem İngilizce, yazı ve video şeklinde birçok kaynak bulmak mümkün. Bu dokümanı da ihtiyacınız varsa bulduğunuz diğer kaynaklara *katık edebilirsiniz* diye paylaşıyorum.

Görüş ve önerilerinizi ayazar@alperyazar.com e-posta adresinden bana ulaştırabilirsiniz. Şimdiden teşekkürler.

Giriş

MODBUS, özünde OSI modelinde en tepede yani application layer'da duran bir protokoldür.

1979 yılından beri hayatımızdadır.

MODBUS şu açıdan ilginç bir protokoldür: Hem TCP/IP, hem de RS232/RS422 gibi asenkron seri kanal protokolleri üzerinde çalışabilir. **MODBUS TCP/IP**, adı üstünde TCP/IP üzerinde de çalışmaktadır. **MODBUS RTU** ve **MODBUS ASCII** ise RS-485 gibi seri kanal protokolü üzerinde çalışmaktadır. RTU, **R**emote **T**erminal **U**nit demektir. RTU'da iletişim binary olurken, ASCII olanda ASCII karakterler üzerinden olmaktadır. RTU'da `0xAB` göndereceksek bir byte olarak `0xAB` gönderiyoruz, ASCII olanda iki byte gidiyor, `'A'` ve `'B'` karakterlerinin ASCII kodları. ASCII olanı daha verimsiz bir protokol olsa da doğrudan "yazı" olarak gönderilip, monitör edilmesi bazı işleri kolaylaştıracaktır. MODBUS'ta pek bir verim, yüksek *throughput* ihtiyacı zaten yoktur.

MODBUS TCP/IP, varsayılan olarak TCP port 502'de çalışmaktadır.

MODBUS, **Server/client**, **request/reply**, **master/slave** şeklinde tasarlanmıştır.

Bir de MODBUS Plus protokolü vardır ve bu dokümanda yer almamaktadır.

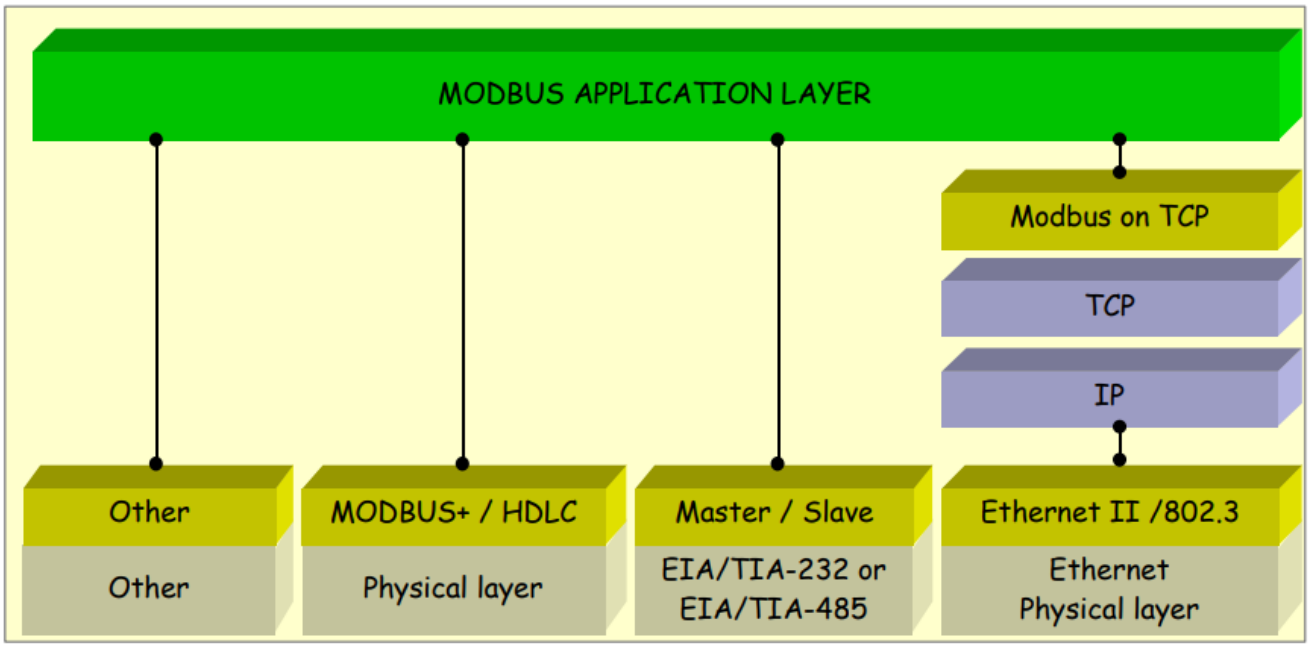


Figure 1: MODBUS communication stack

MODBUS protokolü birçok farklı katman üzerinde çalışabilmektedir. Görsel alıntıdır. [modbusappproto]

Protokol

Bir MODBUS paketi tipik olarak şu şekildedir:

```
| ---------- ADU: Application Data Unit -------- |

| Adres | Function Code |   Data   | Error Check |

        | PDU: Protocol Data Unit  |
```

`PDU`, altta kullanılan iletişim yönteminden (seri kanal, ethernet vs) yani alt katmanlardan bağımsız olarak tanımlanmaktadır. `PDU` içerisinde, `Function Code` ve `Data` alanları vardır. MODBUS paketlerinin gönderileceği fiziksel katmana göre `Adres` ve `Error Check` gibi ek alanlar `PDU` ya eklenerek `ADU` paketi oluşturulur.

İletişim client tarafından başlatılır, `ADU` paketini client oluşturur. RS-485 üzerinde çalışan MODBUS üzerinde ise RS-485 master node (hattı elektriksel olarak ilk süren) client olmakta ve diğer nodelar ise RS485 slave ve server olmaktadır. Yani ilk paket client tarafından hatta konur, o esnada o node RS-485 master olmaktadır.

`Function Code` ile yapılması istenen iş söylenir, 1 byte genişliktedir. `1-255` arası değerler alabilir. `128-255` arası değerler exception cevaplar için ayrılmıştır ve rezervedir. `0` geçerli değildir. Özetle `1-127` arası (sınırlar dahil) `Function Code` değerleri geçerlidir.

`Sub-function Code` lar fonksiyonun ne yapacağını detaylandırmak için eklenebilir. Bazı `Function`

Code lar, yapılacak eylemi detaylandırmak için Data kısmında Sub-function Code barındırır.

Data kısmı olmayabilir, 0 uzunlukta olabilir. PDU sadece Function Code dan oluşabilir.

Eğer her şey yolunda ise cevap paketinde Function Code aynen geri konur. Eğer hata durumu ile karşılaşıldı ise 0x80 ile **OR**lanır yani Function Code un MSB biti set edilir. Bu yüzden geçerli Function Code lar 127 ye kadar (dahil) dir.

İlk tasarlan RS485 temelli MODBUS protokolünde ADU maksimum 256 byte olarak tanımlanmıştır. Bu protokolde, 1 byte Adres ve 2 byte CRC vardır. Bu yüzden MODBUS'taki maksimum PDU, 256-1-2 = 253'tür. 253'e TCP MODBUS için ilgili ek alanları koyarsak, bunlar da 7 byte etmektedir, TCP MODBUS için ADU nun maksimum uzunluğu 260 byte olarak bulunur.

MODBUS PDU

MODBUS, 3 farklı PDU tipi tanımlamaktadır:

- Request PDU, mb_req_pdu
- Response PDU, mb_rsp_pdu
- Exception Response PDU, mb_excep_rsp_pdu

Request PDU

Bu PDU da 1 byte Function Code ve n byte Data vardır. Data kısmının anlamı Function Code a bağlıdır.

Response PDU

Bu da request PDU ile aynı yapıdadır.

Exception Response PDU

2 byte büyüklüğünde bir pakettir. 1 byte'lık Function Code, request PDU'da bulunan Function Code un, 0x80 ile **OR**lanması ile oluşur. 1 byte'lık data kısmında ise MODBUS Exception Code vardır. Bununla ilgili açıklamalar MODBUS dokümanının 7 MODBUS Exception Responses kısmında anlatılmıştır, 9 adet kod tanımlanmıştır. [modbusappproto] Exception paketler, bir şeyler yolunda gitmediği zaman hata durumlarını iletmek için kullanılan paketlerdir.

Data Encoding

Bilgisayarlarımızda olduğu gibi MODBUS protokolünde de en küçük veri birimi/boyutu byte yani 8-bittir. Peki göndereceğimiz veri bir byte'tan büyükse bu veriyi nasıl böleceğiz? MODBUS'ta Big Endian kullanılmaktadır. 1 byte'ı aşan büyüklükler, 1 byte'lık parçalara bölünür. Kabloya (örneğin RS-485 hatta) ilk olarak MSB konur. Örneğin 16-bit yani 2 adet 8-bit yer kaplayan 0x1234 değerini göndereceksek önce 0x12 sonra 0x34 gönderilir.

UYARI	RS-485 gibi protokollerde tipik olarak 1 byte içerisinde 8-bit verinin önce en düşük biti, LSB, gönderilir. Yani MODBUS protokolü önce en yüksek anlamlı byte'ın, MSB, gönderilmesini söylese de bu byte'ın gönderimi sırasında RS-485 üzerinde çalışıyorsak tipik olarak önce bu byte'ın en düşük anlamlı bitini, LSB, görürüz. Bu iki kavramı karıştırmayalım.

Data Model

MODBUS, endüstriyel uygulamalar için tasarlanmıştır, o yüzden dokümantasyonundaki bazı terimler garip gelebilmektedir.

MODBUS dokümanlarında, protokol ile cihazın belleğindeki çeşitli değerlere erişildiği (okuma veya yazma) düşünülmüştür. 4 farklı temel blok tanımlanmıştır. Bu 4 adet blok adeta cihazın belleğinde durmaktadır ve bizler MODBUS üzerinden bunlara erişmekteyiz.

Elbette MODBUS gerçekleştiren cihazların "içi" böyle olmak zorunda değildir, bu bir modeldir. *Memory-mapped I/O* modeline benzemektedir.

Blok	Nesne Tipi	R/W
Discrete Input	1-bit	R
Coils	1-bit	R/W
Input Registers	16-bit word	R
Holding Registers	16-bit word	R/W

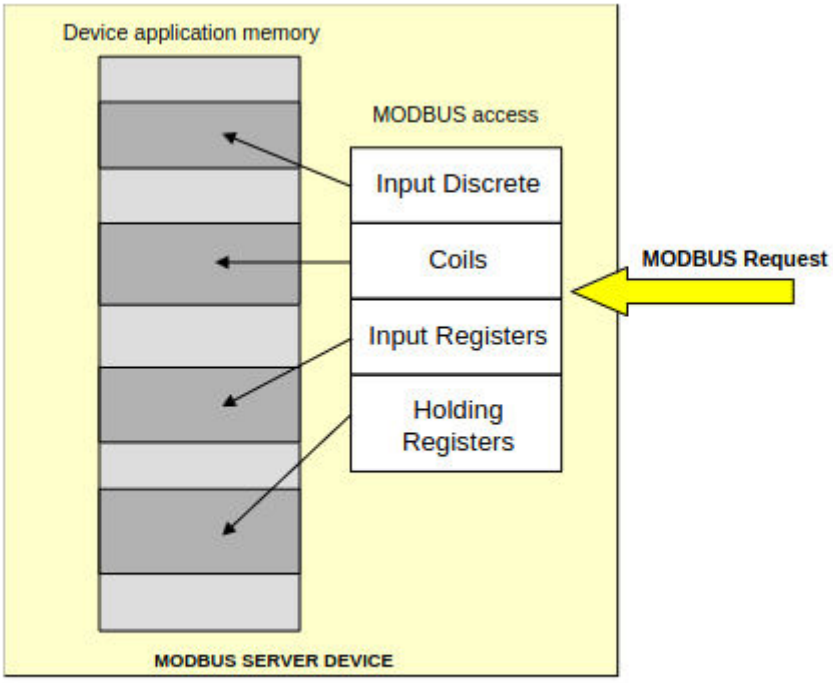


Figure 6 MODBUS Data Model with separate block

Bellekte sanki bir bölgede Discrete Input, bir yerde Coils, bir yerde Input Registers varmış gibi düşünebiliriz. Görsel alıntıdır. [modbusappproto]

Elbette bu kısımlar overlap edebilir. Yani 16-bit genişliğindeki bir *register*ı hem `Input Registers` ile *word* (16-bit) genişliğinde hem de `Coils` ile *bitwise* görebiliriz. Yani bellekteki bazı lokasyonlar birbirinin **alias** ı olabilir.

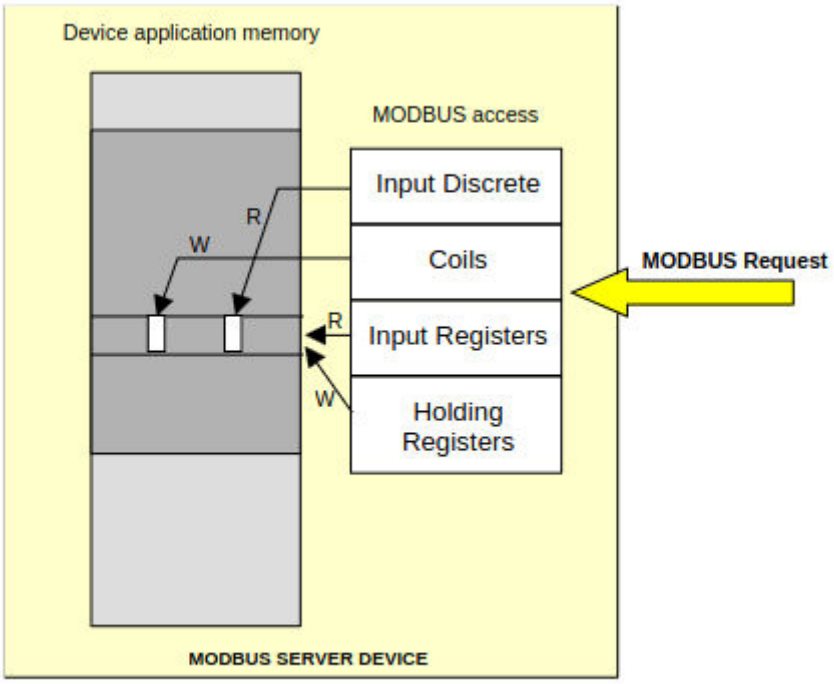


Figure 7 MODBUS Data Model with only 1 block

Burada da tüm 4 blok alias durumdadır. Görsel alıntıdır. [modbusappproto]

Bellekteki bu alanların değerlerinin ne ifade edeceği tamamen **vendor (üretici) bağımlıdır.** MODBUS'un sunduğu bu veri modelinin, uygulamaya nasıl bağlanacağı IEC-61131 gibi *application model* ler ile belirlenir.

MODBUS protokollerinde bu 4 farklı türdeki hafıza, bir numara ile adreslenmektedir. MODBUS paketlerinde, `PDU` içerisinde, bu adresler `0-65535` arasında olmaktadır. Fakat *MODBUS data model* i denen bir modelde adres `1` den başlamaktadır. Örneğin 3 nolu `Discrete Input` türünden bir biti okumak için gerekli olan MODBUS paketindeki adres değeri 2 olmalıdır, kafa karıştırıcı değil mi...

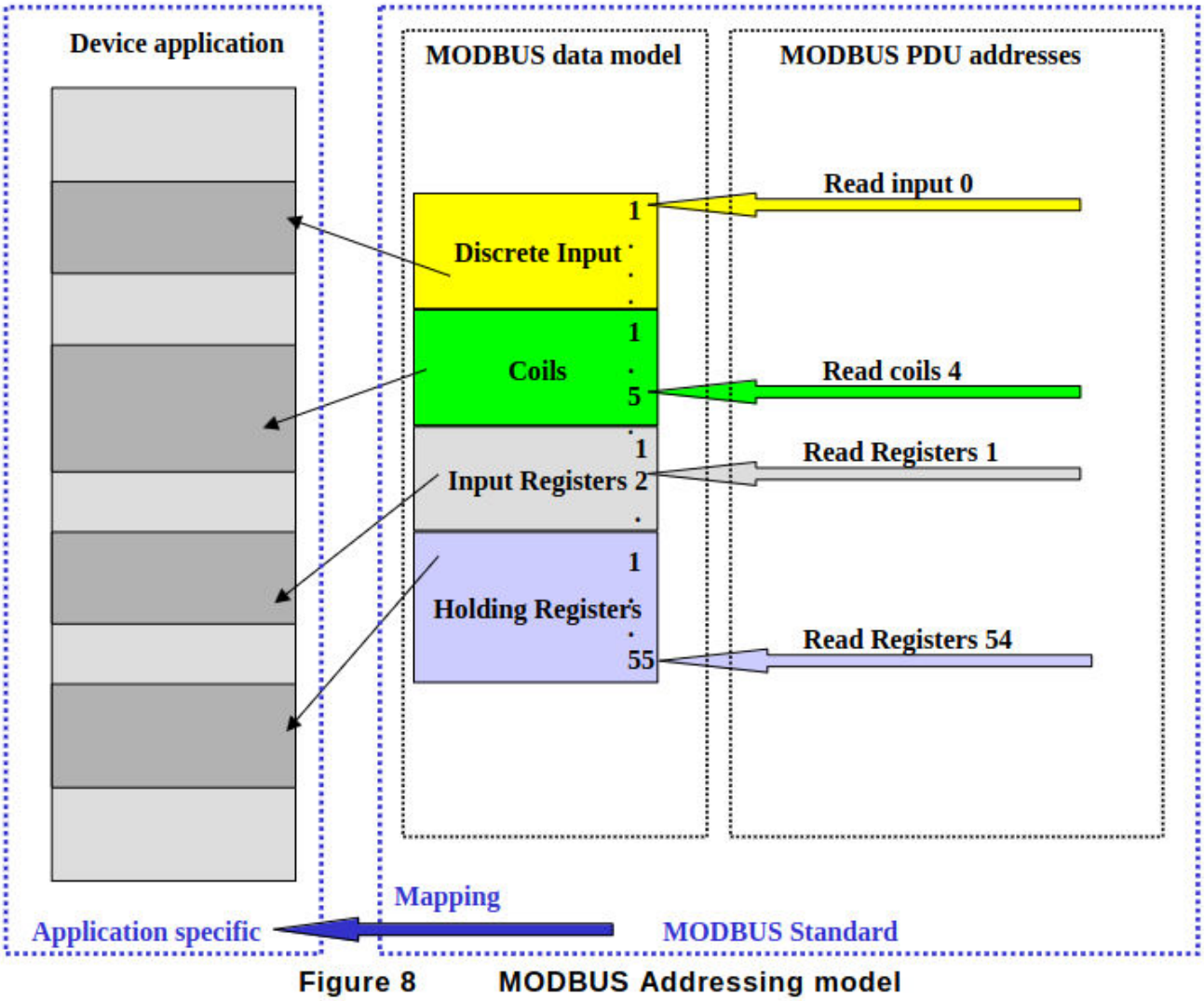


Figure 8 MODBUS Addressing model

Paket'te `n` yazıyorsa hafızada `n+1` e erişiyoruz. Görsel alıntıdır. [modbusappproto]

Function Codes

Yukarıda `1-127` (sınırlar dahil) arasındaki `Function Code` ların geçerli olduğundan bahsetmiştik.

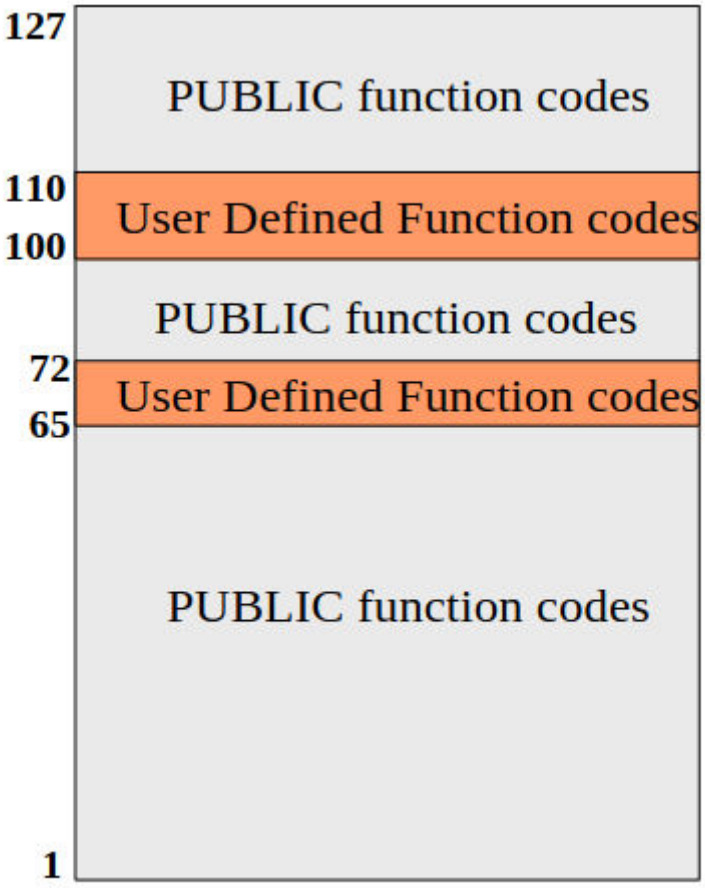


Figure 10 MODBUS Function Code Categories

`Function Code` ların bir kısmı rezerve, bir kısmına da önden anlamlar yüklenmiştir. Görsel alıntıdır. [modbusappproto]

Bu aralıktaki kodların bir kısmı MODBUS tarafından `PUBLIC` olarak işaretlenmiş ve genel amaçlıdır. Bir kısmı ise `User Defined` olarak bırakılmıştır.

MODBUS RTU

MODBUS bir seri kanal üzerinde, RS-232 veya RS-485 üzerinde implement edildiği zaman ve veri binary olarak taşındığı zaman **MODBUS RTU** adını almaktadır. Binary değil de text yani metin yani *görünen/yazdırılabilir karakterler* olarak taşınıyorsa da **MODBUS ASCII** olmaktadır. RTU'yu anlamak için MODBUS dokümanını temel alacağız. [modbusserial]

RTU'da hattaki master node (hattı ilk süren) MODBUS client, slave node'lar (hattı ilk dinleyen) MODBUS server olmaktadır.

Seri bir haberleşme hattı üzerinde implement edilen MODBUS RTU'da sadece 1 adet master olmaktadır ve maksimum 247 adet slave olabilir.

- İletişim her zaman master tarafından başlatılır.
- Master'dan bir şey sorulmadan slave kendi başına gaza gelip bir mesaj atmaz.
- Slave'ler kendi arasında konuşamaz.
- Master aynı anda sadece 1 adet transaction başlatabilir.

İki tip *mode* tanımlıdır. **Unicast mode** da master bir adet slave'e bir şey der, slave de bunun cevabını verir. Burada transaction dediğimiz şey bu iki paketten oluşur. Her slave'in `[1-247]` arası (sınırlar dahil) adresi vardır.

Broadcast mode da ise master bir isteği tüm slave'lere yollar. Burada slave'ler tarafından bir cevap verilmez yani transaction dediğimiz şey tam da tanımlı değildir ama tanımlamak istersek tek mesajdan oluşuyor gibi düşünebiliriz. **Adres `0` gönderilen paketler broadcast paket olmaktadır ve tüm slave'ler bu paketi almalıdır.**

Adres alanı 1 byte'tan oluşmaktadır:

Adres	Fonksiyon
`0`	Broadcast adresi
`[1-247]`	Slave adres
`[248 - 255]`	Rezerve

Bir adres iki adet slave'e atanamaz. Master mode'un bir adresi olmak zorunda değildir, sadece slave node'ların adresi olması zorunludur. Slave'ler cevap verirken kendi adreslerini belirtirler. Yani unicast giden istek ve cevaplardaki adresler aynı olmaktadır.

RTU'da paket formatı (en başta `ADU` diye belirttiğimiz) şöyledir:

```
| Adres (1 byte) | Function code (1 byte) | Data (0-252 byte) | CRC (2 byte) |
```

MODBUS RTU dokümanında master ve slave node'lar için state diyagramlar verilmiştir.

Master State Diagram

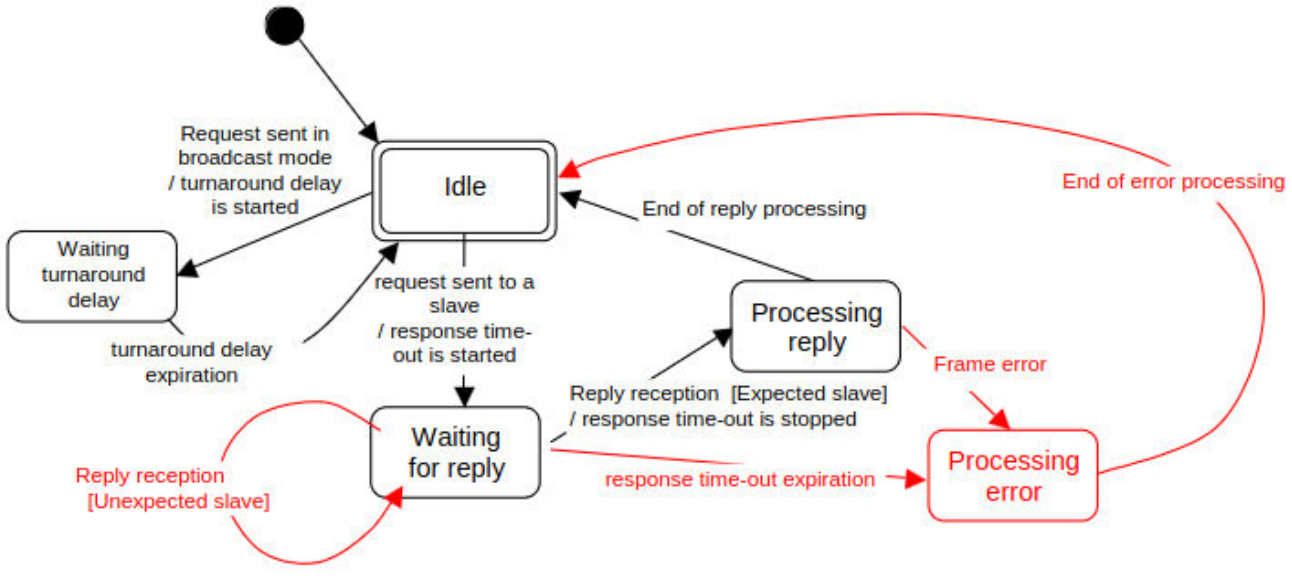


Figure 7: Master state diagram

Görsel alıntıdır. [modbusserial]

- Master açılınca `Idle` state'inde duruyor. **Bu state'te değilse request atamıyor.** Yani bir request attıktan sonra Idle'a gelene kadar başka request atılamaz.
- Unicast mode'da bir slave adreslenerek mesaj atıldıysa master cevap beklemeye geçer. Bir yandan da bir time-out süresi saymaya başlar. Time-out süresi *implementation defined* olarak bırakılmıştır.
- Diyelim ki mesaj geldi. Başka bir slave cevap verdiyse ya da gelen pakette CRC hatası vs varsa ya da time-out olduysa hata durumuna gidilir. Master isterse retry yapabilir.
- Broadcast mesajlarda slave'ler bir cevap dönmez. Ama master slave'lerin bunu işlemesi için **trunaround delay** kadar beklemelidir. Bu bekleme sırasında yeni mesaj atamaz.
- 9600 bps için turnaround delay tipik olarak 100-200 ms yani ~96 - ~192 byte arası, time-out süresi ise saniyeler mertebesindedir.

Slave State Diagram

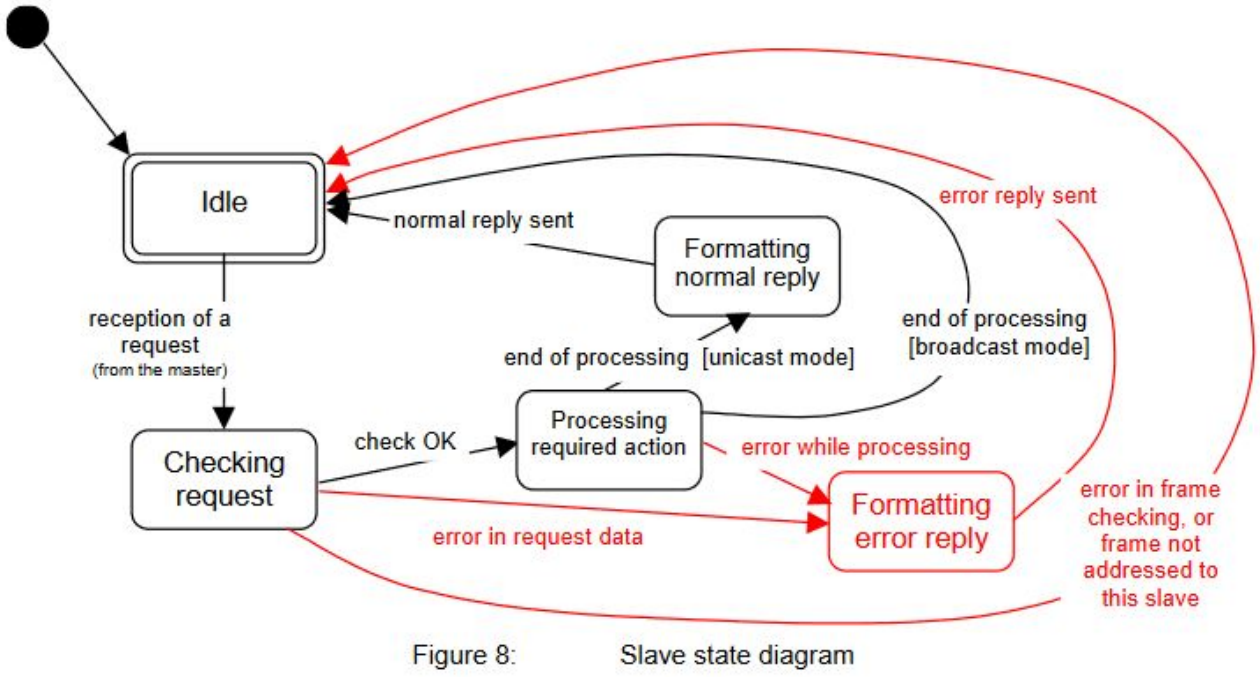


Figure 8: Slave state diagram

Görsel alıntıdır. [modbusserial]

- Slave açılınca `Idle` state'inde duruyor.
- Bir paket geldiği zaman eğer gelen pakette CRC hatası gibi hatalar varsa veya master'ın attığı paket ile ilgili slave adreslenmediyse, paket slave tarafından discard/drop edilebilir. Bu durumda slave'in bir cevap vermesine gerek yoktur.
- Eğer pakette slave'in yapamayacağı bir şey isteniyorsa ya da paketin içeriği hatalı ise master'a cevap dönülmelidir.
- Her şey yolunda ise master'ın istediği yapıldıktan sonra ve paket **unicast ise** master'a cevap dönülmelidir.

State diyagramında ve açıklamada net olmayan bence şöyle bir kısım var: Broadcast paketinin içeriğinde hata varsa slave yine cevap dönmeli mi? Bence broadcast mesajında böyle bir akış olmamalı. Birden fazla node cevap dönmeye çalışırsa ne olacak hata durumunda? Bunun cevabını MODBUS dokümanı içerisinde yakalayabiliyoruz: [modbusserial]

> It comprises also the error detected in broadcast messages even if an exception message is not returned in this case.

Yani diyor ki broadcast durumunda hata oluşursa cevap dönülmez. Yine de açıkça state diagramda belirtilirse daha iyi olurmuş.

Örnek Akış

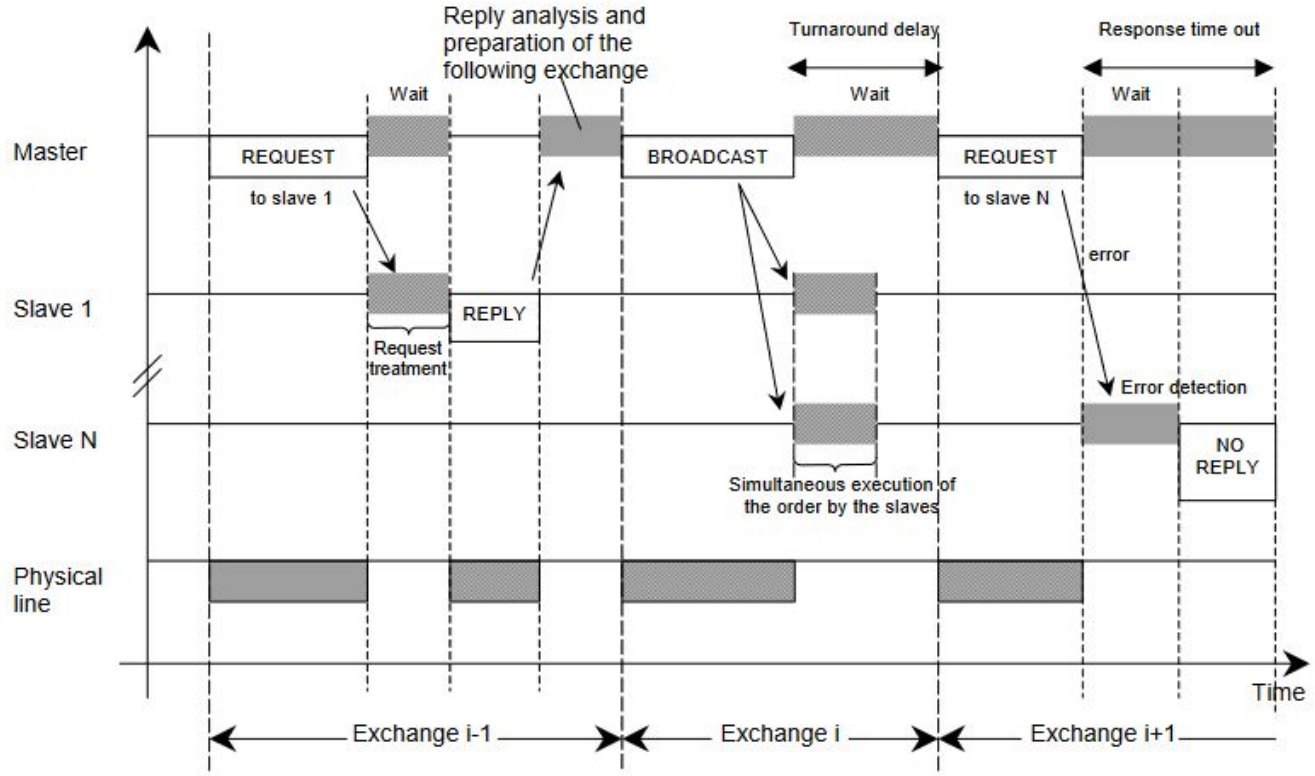


Figure 9: Master / Slave scenario time diagram

Görsel alıntıdır. [modbusserial]

Yukarıdaki görselde 3 adet transaction verilmiştir.

İlk olanda, yani `i-1` olanda, master bir istek atmakta, arkasından da slave cevap vermektedir. Burada slave'in paketi aldıktan sonra işlemesi bir müddet vakit almaktadır: `Request treatment`. Daha sonra slave cevap atmakta ve master'da bir süre gelen cevabı incelemektedir.

Bir sonrakinde, `i` olanda, master bir broadcast mesajı atmakta ve broadcast mesajlarında slave'ler cevap vermemektedir. Fakat master `Turnaround delay` kadar bir süre *open loop* şekilde beklemektedir. Bu süre, slave'lerin mesajı işlemesi için master'ın beklediği bir süredir.

Son `i+1` isimli transaction'da slave paketi hatalı almakta, CRC hatası örneğin, ve paketi drop etmektedir. Bu durumda slave bir cevap oluşturmaz. Master time out'a girer ve bir sonraki

transaction'a geçer.

`REQUEST`, `REPLY`, `BROADCAST` gibi süreler paket uzunluğuna ve baud rate'e bağlıdır. Onun dışındaki bekleme süreleri ise slave'lere göre seçilmelidir. Ne kadar sürede mesaj işlenebiliyor vs.

Transmission Modes

Seri kanal üzerinde implement edilen MODBUS protokolü için iki adet *transmission mode* tanımlanmıştır: **RTU** ve **ASCII**. RTU modu tüm cihazlar tarafından implement edilmelidir, ASCII opsiyoneldir. İki modu implement eden cihazlarda default mode RTU olmalıdır. Mode, verinin mesaj içerisine nasıl kodlanacağını (encoding) belirler. ASCII, text modu RTU ise binary mode olarak düşünülebilir. Bu dokümanda sadece RTU'ya odaklanıyorum.

RTU

"Bildiğimiz" seri kanal mesajlarından oluşur.

```
| 1 bit start | 8 bit data (LSB önce) | 1 bit parity | 1 bit stop |
```

Data gönderilirken önce LSB biti hatta konur. Default olarak **even parity** kullanılır. Opsiyonel olarak odd parity veya no parity de desteklenebilir. MODBUS dokümanları geniş bir destek aralığı için no parity'nin de desteklenmesini önermektedir. Eğer parity kullanılmazsa yerine stop biti konulmalıdır, yani 2 stop bit gönderilmelidir. Özetle 8-bit faydalı veri gönderimi için her zaman 11 bit veri gönderimi yapılmalıdır.

CRC

RTU formatını hatırlayalım:

```
| Adres (1 byte) | Function code (1 byte) | Data (0-252 byte) | CRC (2 byte) |
```

Burada CRC'nin önce LSB byte'ı sonra MSB byte'ı gönderilmelidir:

```
CRC Low (1 byte) | CRC High (1 byte)
```

şeklinde.

ÖNEMLI	**MODBUS data order'ı big endian iken CRC order'ı little endian olmaktadır.**

Her 8-bit, byte, veride `parity` nin olup olmamasından bağımsız olarak paket sonunda `CRC` olmak zorundadır.

CRC gözüktüğü gibi 16-bit genişliğindedir.

CRC, tüm mesaj üzerinden hesaplanır yani `Adres`, `Function code` ve `Data` hesaplaması CRC'ye dahil

edilir.

MODBUS dokümanlarında CRC'nin hesaplanması detaylı olarak anlatılmaktadır. [modbusserial] Pratikte aşağıdaki sitelerden CRC hesaplaması yapılabilir:

- https://crccalc.com/ `CRC-16/MODBUS` kullanılmalıdır.
- https://www.lammertbies.nl/comm/info/crc-calculation `CRC-16 (MODBUS)` kulanılmalıdır.

Yukarıdaki sitelere veri girerken `ASCII` değil `HEX` girdiğinizden emin olun. MODBUS dokümanlarından devam edecek olursak elimizdeki frame içeriği `0207` den oluşuyorsa, yani adres `0x02` ve function code `0x07` ise yani data yoksa CRC, `0x1241` olarak bulunmalıdır. Fakat bu CRC hatta konulduğu zaman hatta `0x41` ve `0x12` görülmelidir. Çünkü MODBUS protokolünde CRC'nin önce düşük byte'ı, LSB, sonra yüksek byte'ı, MSB, gönderilmelidir. Günün sonunda hat üzerinde `02074112` görülmelidir.

Framing

RTU mesajlaşmada **frame'ler (yani paketler) arasında en az 3.5 karakter boşluk bulunmalıdır. Peki karakter süresi ne kadardır?** Bir karakter 8 bit olarak mı yoksa 11 bit (1 bit start + 8 bit data + 1 bit parity + 1 bit stop) olarak mı alınmalıdır? Ben MODBUS dokümanında net bir tanım göremedim. 8 bit, 11 bit, hatta 10 bit alan var (bence en alakasızı bu, en azında RTU için ASCII modda 10 bit almak doğru olacaktır). [sortuchartime] 8 bit bence yanlış çünkü dokümantasyonda bir yerde

> Only the eight bits of data in each character...

diye bir laf geçiyor. Demek ki character 8 bit'ten fazla. 10 bit doğru değil bence, parity var. O yüzden 11 bit olarak düşünmek mantıklı geliyor.

MODBUS RTU için 8-bit anlamlı veri içeren fakat start, stop ve parity bit ile toplam boyutu 11-bit olan veri birimine karakter (character) diyebiliriz.

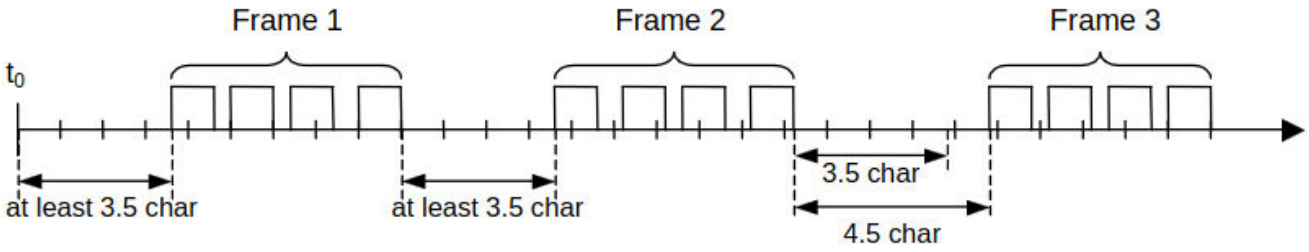
Bir frame içinde de karakterler arasında **1.5 karakter süresinden fazla boşluk olmamalı.** Böyle bir durumda o frame eksik kabul edilir ve alıcı tarafından ihmal edilmelidir.

3.5 karakterlik süre `t3.5`, 1.5 karakterlik süre de `t1.5` olarak geçmektedir. Driver implementasyonuna bağlı olarak timer kullanımı CPU interrupt yükünü arttırmaktadır. Baud rate arttıkça timer interrupt sıklığını arttırmamak için MODBUS dokümanı baud rate'ten bağımsız sabit değerler kullanılmasını önermektedir. Buna göre:

```
if baud rate > 19200
  t3.5 = 1750 us
  t1.5 =  750 us
else
  t3.5 = (3.5 * 11) / baud rate
  t1.5 = (1.5 * 11) / baud rate
```

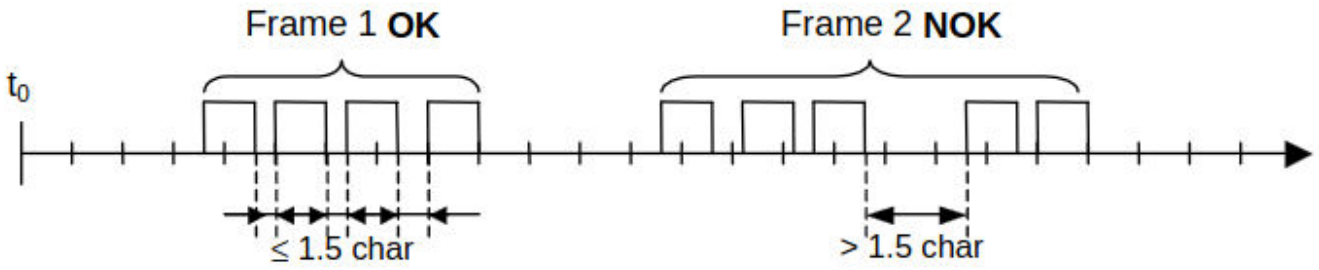
Karakter süresinin kaç bit olacağı dediğim gibi net değildir. 19200 bps değerindeki bu sayıları

tutturmak için 11 değil 10 almak daha uygun oluyor ama 19200 bps değerinde bu sayılar tam örtüşecek diye bir şey de yok. Örneğin bu kütüphanede de benim gibi 11 bit almışlar, benzer şekilde Wikipedia'da da 11 bit olarak alınmış, devam edelim...



Görüldüğü gibi frame'ler arası en az 3.5 karakter süresi olmalıdır. Elbette 4.5 gibi daha büyük bir sayı da olabilir. Görsel alıntıdır. [modbusserial]

Benzer şekilde:



Bir frame içerisinde iki karakter arasında 1.5 karakter süresinden daha fazla boşluk bulunamaz. Bu durumda alan kişi mesajı discard etmelidir. Görsel alıntıdır. [modbusserial]

Hem master hem de slave açısından şöyle bir state diagram çizilebilir:

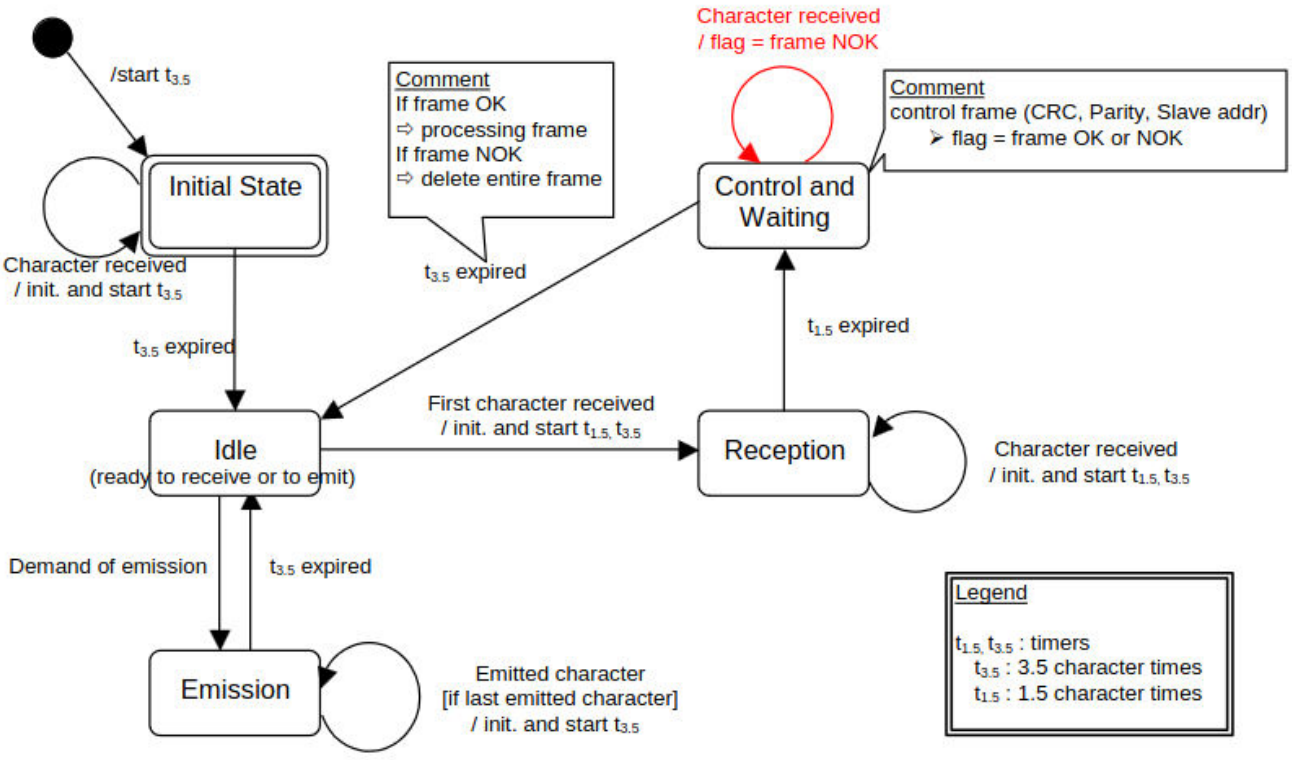


Figure 14: RTU transmission mode state diagram

Görsel alıntıdır. [modbusserial]

- En az 3.5 karakter süresi boyunca, `t3.5`, bir iletim olmazsa hat **idle** olarak kabul edilir.
- Alan taraf `t3.5` bittikten sonra frame'in bittiğini düşünüp işler.
- Alan taraf isterse işlem kolaylığı açısından CRC'yi beklemeden adres alanını işleyip, eğer kendisi adreslenmediyse frame'in bitişini beklemeye başlayabilir. Devamını almasa da olur.

MODBUS dokümanının devamında RS-485/422 protokolü ile ilgili daha genel bilgiler, elektriksel özellikler, konnektör ve LED önerileri bulunmaktadır. Bu bilgileri buraya tekrar almıyorum, çoğu da zaten MODBUS'tan bağımsız bilgiler. Yine de dokümandan takip edilebilir. [modbusserial]

MODBUS Function Codes

Bu kısımda, MODBUS standartında belirtilen `Function Code` lara bakacağız.

`1-127` arası function code'ların geçerli olduğundan bahsetmiştik. Bir hatırlayalım:

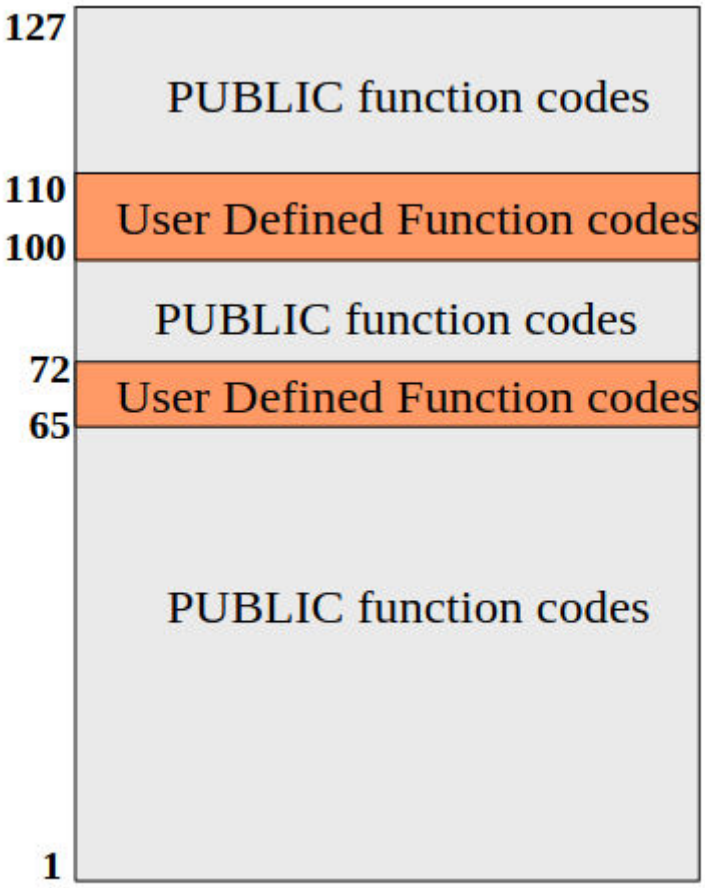


Figure 10 MODBUS Function Code Categories

Function code'ların bir kısmı rezerve, bir kısmına da önden anlamlar yüklenmiştir. Görsel alıntıdır. [modbusappproto]

Bu aralıktaki kodların bir kısmı MODBUS tarafından PUBLIC olarak işaretlenmiş ve genel amaçlıdır. Bir kısmı ise User Defined olarak bırakılmıştır.

				Function Codes			
				code	Sub code	(hex)	Section
Data Access	Bit access	Physical Discrete Inputs	Read Discrete Inputs	02		02	6.2
		Internal Bits Or Physical coils	Read Coils	01		01	6.1
			Write Single Coil	05		05	6.5
			Write Multiple Coils	15		0F	6.11
	16 bits access	Physical Input Registers	Read Input Register	04		04	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	03		03	6.3
			Write Single Register	06		06	6.6
			Write Multiple Registers	16		10	6.12
			Read/Write Multiple Registers	23		17	6.17
			Mask Write Register	22		16	6.16
			Read FIFO queue	24		18	6.18
	File record access		Read File record	20		14	6.14
			Write File record	21		15	6.15
Diagnostics			Read Exception status	07		07	6.7
			Diagnostic	08	00-18,20	08	6.8
			Get Com event counter	11		OB	6.9
			Get Com Event Log	12		0C	6.10
			Report Server ID	17		11	6.13
			Read device Identification	43	14	2B	6.21
Other			Encapsulated Interface Transport	43	13,14	2B	6.19
			CANopen General Reference	43	13	2B	6.20

Yukarıdaki görsel MODBUS dokümanından alınmıştır. Bazı satırların neden sarı ile highlight edildiğini bilmiyorum. Görsel alıntıdır. [modbusappproto]

Her bir function code'un anlamı ve ne yapılacağı MODBUS dokümanında anlatılmıştır. [modbusappproto] Bunları tekrar yazmak pek anlamlı değildir. Temel kavramları aktarmaya çalışacağım. RTU ile ilgili bilgileri de MODBUS dokümanından alacağım. [modbusserial]

Read Coils, `0x01`

MODBUS protokolünde **coil** 1-bit genişliğinde yazma ve okuma yapılabilen bir alan olarak düşünülebilir. Bu komut server'dan (RTU'da slave) bu register'ları okumak için kullanılır. MODBUS'a göre 65536 adet coil olabilmektedir. Bu komut ile okunmak istenen coillerin base adresi ve range'i verilir. Ardışıl olan coiller tek bir komut ile okunabilir. Bir komut ile maksimum 2000 adet coil okunabilir. Cevap paketinde de her coil 1 bit ile ifade edilir.

İstek:

- Function code: `0x01`
- Data, ilk 2 byte: `0x0000-0xFFFF`, coil başlangıç adresi yani okunacak en düşük adresli coil adresi.
- Data, sonraki 2 byte: `1-2000`, kaç adet coil'in okunacağı.

İstek paketi toplam 5 byte'tan oluşmaktadır. MODBUS bellek modelinde paketlerdeki adresleme ile bellek adresleri arasında 1 fark olduğundan bahsetmiştik. Yani cihaz belleğinde coil adresleri `1-65536` arasında iken haberleşme sırasında coiller `0-65535` ile adreslenir. Bu tüm veri modelleri için

geçerlidir.

Cevap:

- Function code: `0x01`
- Data, ilk 1 byte: Arkada kaç byte'lık data olacağı, `N` diyelim
- Data, sonraki `N` byte: Coil status bilgileri

Eğer 16 adet coil status okunmak istendiyse `N` 2 olacaktır fakat 17 adet istendiyse 3 olacaktır. Sayının 8'e bölümünün üste yuvarlanması ile `N` sayısı elde edilir.

Diyelim ki `20-39` adreslerindeki coilleri okumak istiyoruz. MODBUS RTU üzerinde aşağıdaki gibi olacaktır. MODBUS'ın byte order'nın big endian, RTU'daki CRC'nin ise little endian olduğunu hatırlayalım.

İstek:

```
| Adres | 0x01 | 0x00 | 0x13 | 0x00 | 0x14| CRC-low | CRC-high |
```

Başlangıç adresimiz 20 fakat haberleşmede 1 eksiğini alıyoruz, 19 yani 0x13 oluyor. Okumak istediğimiz aralık ise `20 = 39 - 20 + 1` adet coil içeriyor, yani 0x14.

20/8 = 2.5, bunu 3'e yuvarlıyoruz yani 3 byte cevap vereceğiz.

Cevap:

```
| Adres | 0x01 | 0x03 | Coil 27-20 | Coil 35-28 | Coil 39-36 | CRC-l | CRC-h |
```

olacaktır.

Coil status kısmı, base adres ile başlar yani en düşük adresli coil. İlk byte'ın LSB biti en düşük adresli coil'in durumunu içermelidir. `1`, ON; `0`, OFF demektir. Örneğin slave bu byte'ın değerini `1010 1101` olarak gönderirse:

```
Bit:   1  0  1  0  1  1  0  1
Coil: 27 26 25 24 23 22 21 20
```

olarak anlamdırırız. Fakat unutmayın ki MODBUS RTU'da data'nın önce LSB biti konur. Yani osiloskop ile bakıyorsak hatta aslında `1011 0101` görürüz.

Adresleme bu şekilde artarak devam eder. En son byte'ta boşluklar olabilir, örneğin bu örnekte 4 bit padding yapılması gerekmektedir. Bunlar 0 ile pad edilmelidir. Son byte:

```
Bit:   0  0  0  0  1  1  1  1
Coil: -- -- -- -- 39 38 37 36
```

gibi...

Bir de **hata durumlarına** bakalım. Dokümanda gayet güzel gösterilmiş:

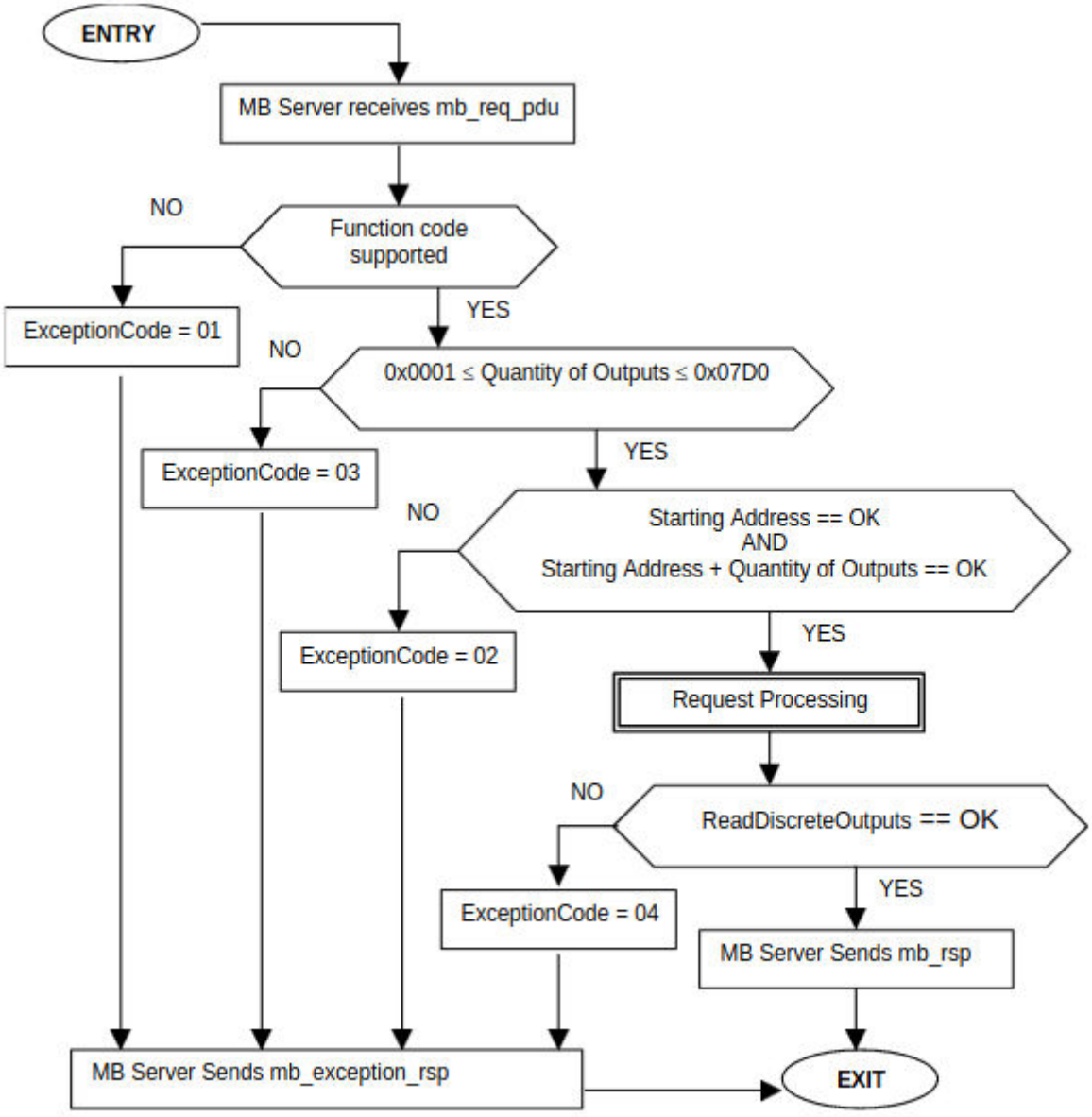


Figure 11: Read Coils state diagram

Görsel alıntıdır. [modbusappproto]

Görebileceğimiz gibi farklı hata durumlarında farklı hatalar dönmelidir. Hata durumlarında 1 byte `function code`, istek `function code` un `0x80` ile ORlanması ile oluşturulur. Daha sonra 1 byte data olarak `ExceptionCode` konur.

Diyelim ki master, olmayan bir adresteki coil'i okumak istedi. Bu durumda da `02` nolu exception dönüyorsak

Cevap:

```
| Adres | 0x81 | 0x02 | CRC-l | CRC-h |
```

olacaktır.

Yazının geri kanalındaki komutların çalışma biçimi bu komuta benzediği için daha yüzeysel anlatacağım.

Read Discrete Inputs, `0x02`

Read coils, `0x01`, ile aynı çalışmaktadır.

Read Holding Registers, `0x03`

Coil ve discrete input okumak ile benzerdir özünde. `Holding Registers`, MODBUS standartında 16-bit olarak tanımlanmıştır. Buna göre protokolde değişiklikler olmaktadır. Farklı olarak tek seferde en fazla 125 adet register okunabilir, her biri 16-bit. Bir register 2 byte şeklinde gönderilir. MODBUS'a uygun olacak şekilde ilk olarak MSB byte hatta konur, yani big endian order kullanılır.

Read Input Registers, `0x04`

Read Holding Registers, `0x03`, ile aynı çalışmaktadır.

Write Single Coil, `0x05`

Tek 1-bit lik coil register'ın değerini değiştirmek için kullanılır.

RTU İstek:

```
| Adres | 0x05 | 0x0000 - 0xFFFF | 0x0000 or 0xFF00 | CRC-l | CRC-h |
```

CRC ve adres hariç 5 byte'lık bir pakettir. `Function code` sonrası ilk 2 byte yazma yapılacak regsiter adresini belirtir. Önceki paketlerde olduğu gibi `1` offset olayı burada da vardır. İlgili register'ı `ON` yapmak için `0xFF00`, `OFF` yapmak için `0x0000` yazılır. Diğer değerler geçersizdir.

Write Single Register, `0x06`

Tek bir holding register'a (16-bit) yazmak için kullanılır. Write Single Coil, `0x05`, e oldukça benzer. Data kısmı sabit iki adet 16-bit veri yerine yazılması istenen 16-bit veridir.

Read Exception Status, `0x07`

Sadece seri kanal implementasyonlarında vardır. 8-bit bir değer okunuyor ama ne işe yarıyor anlamadım.

Diagnostics, 0x08

Sadece seri kanal implementasyonlarında vardır. Slave cihazdan (server) çeşitli sorgular ve test yapmaya yarar. 2 byte'lık sub-function code'lar ile istek şekillendirilir. Detayları MODBUS dokümanında vardır.

- Loopback test
- Bir slave cihazı susturmak
- İletişimi resetlemek
- Çeşitli istatistik register'larını okumak için

kullanılır.

Get Comm Event Counter, 0x0B ve Get Comm Event Log, 0x0C

Sadece seri kanal implementasyonlarında vardır. Detayları dokümanlardan okunabilir. Özünde seri kanal ile çeşitli istatistikleri döner.

Write Multiple Coils, 0x0F

Birden fazla 1-bit genişliğinde olan coil'e yazmaya yarar. Write Single Coil, 0x05 den farkı birden fazla, ardışıl 1-bit register'a yazma imkanı sunmasıdır. Tek seferde en fazla 1968 adet register'a yazma yapılabilir.

Write Multiple Registers, 0x10

Ardışıl 16-bit genişliğindeki holding register'lara yazma yapmayı sağlar. Write Single Register, 0x06 komutundan farklı olarak birden fazla register'a tek seferde yazmaya yarar. Bu komut ile tek seferde 123 adet register'a kadar yazma yapılabilir.

Report Server ID, 0x11

Sadece seri kanal implementasyonlarında vardır. Cihaza özgü bir cevabı vardır.

Read File Record, 0x14

MODBUS'ta **file** denen bir kavram vardır. Fiziksel olarak tam neye karşılık geldiği bence net değil, üreticiye bırakılmış. Örneğin data logger gibi bir cihazda file, capture edilen bir waveform'u gösterebilir. Ya da uzaktan yazılımını güncelleyeceğimiz bir cihazın flash'ında duran yazılımı bir file gibi kurgulayabiliriz. Bu kısım, slave cihaz üreticisine bağlı.

Kurgusual olarak her bir dosyanın bir numarası vardır, bu numara aralığı 1-65535 arasındadır. Bu numara, dosya ismi gibidir. Her bir dosya **record** adı verilen 2-byte genişliğindeki parçalara bölünmüştür. Her bir dosya da en fazla (?), 10000 adet record bulunabilir. **Böylece bir dosya en**

fazla `20000 = 2 * 10000 byte, ~19.53 KB` boyutunda olabilir.

Bu komut ile istenirse birden fazla file'dan ardışıl olmak üzere birden fazla uzunlukta **record** okunabilir. Detayları dokümanında vardır. Protokol açısında maksimum paket boyutunu geçmememiz lazım.

Write File Record, `0x15`

Okuma komutuna benzer, paket formatı olarak da. Bu da yazma yapmak için kullanılır.

Mask Write Register, `0x16`

Bir adet 16-bit genişliğindeki holding register'a doğrudan veri yazmadan, bit bit bazı bitleri AND ve OR işlemine tutarak, yani maskeleyerek, set veya reset etmeye yarar. Yani diyelim ki 10.bit'i set etmek, aynı anda 4.bit'i reset etmek için bunu kullanabiliriz.

Read/Write Multiple registers, `0x17`

Tek bir transaction ile birden fazla 16-bit holding register'ı, ardışıl olmak şartı ile, okuma yapmaya ve yazma yapmaya (aynı adres aralığında olmak zorunda değil okuma ile yazma fakat ikisi de kendi içinde ardışıl olmalı) yarar. MODBUS standardına göre önce yazma sonra okuma yapılır.

Read FIFO Queue, `0x18`

Anladığım kadarıyla MODBUS'ta genişliği 16-bit yani bir holding register genişliğinde olan bir FIFO data modelinden bahsediliyor. Bu FIFO teorik olarak `65535 + 31` derinliğinde. Bu komut ile FIFO'nu herhangi bir offsetine gidip maksimum `31` adet veri okuyabiliyoruz. Bu komut ile FIFO'dan okuma yapıldığı zaman FIFO içeriği silinmemektedir.

Encapsulated Interface Transport, `0x2B`

Bu komut ile MODBUS'a *tünelleme* yaptırılabilir. SSH Tünel gibi MODBUS üzerinden başka bir protokolün taşınması sağlanabilir. MODBUS dokümanlarında CANOpen mesajlarının taşınması anlatılmıştır. Tünelleme ihtiyacı olunca dönüp bakılabilir.

MODBUS Exception Responses

MODBUS dokümanlarında çeşitli exception'lar tanımlanmıştır. Bunlar protokollerin içinde de anlatılmaktadır. Diyelim ki desteklenmeyen komut attık, yanlış parametre attık, bu durumda **exception response** gelecektir. Detayları dokümanda anlatılmıştır. Bu durumda daha önceden bahsettiğimiz gibi **function code**, `0x80` ile ORlanmaktadır.

Alıntıla

```
@misc{yazar2024blogmodbusv1,
  title = {MODBUS Notlarım},
  author = {Yazar, Alper},
  year = {2024},
  url = {https://www.alperyazar.com/dow/modbus-v1.pdf},
}
```

NOT	Bu dokümana https://www.alperyazar.com/dow/modbus-v1.pdf adresinden erişebilirsiniz. Fakat dokümanın daha güncel hali yayınlanmış olabilir. Bunun için lütfen https://www.alperyazar.com/dow/modbus.pdf adresini ziyaret ediniz. Eğer bu doküman güncel ise bu dokümanı, değilse daha güncel bir sürümünü bulabilirsiniz. Bu dokümanın sürümü **v1** olup **2024** tarihinde yayınlanmıştır.

Teşekkürler

Bu içeriğin hazırlanmasında katkılarından dolayı Hamza Murat Yılmaz'a teşekkür ederim.

Kaynaklar

- [modbusappproto] MODBUS Application Protocol Sepcification V1.1b3 Link1 Link2
- [modbusserial] MODBUS over Serial Line Specification & Implementation Guide V1.02 Link1 Link2
- MODBUS (Wikipedia)
- [sortuchartime] Calculating modbus RTU 3.5 character time