**T. C.**
**GEBZE YÜKSEK TEKNOLOJİ ENSTİTÜSÜ**
**MÜHENDİSLİK VE FEN BİLİMLERİ ENSTİTÜSÜ**

# CONTROLLER AREA NETWORK VE UYGULAMASI

**Uğur COŞKUN**

**YÜKSEK LİSANS TEZİ**
**ELEKTRONİK MÜHENDİSLİĞİ**

**GEBZE**
**2008**

**T. C.**
**GEBZE YÜKSEK TEKNOLOJİ ENSTİTÜSÜ**
**MÜHENDİSLİK VE FEN BİLİMLERİ ENSTİTÜSÜ**

# CONTROLLER AREA NETWORK VE UYGULAMASI

**Uğur COŞKUN**

**Danışmanı**
**Yrd. Doç. Dr. Serdar S. ERDEM**

**YÜKSEK LİSANS TEZİ**
**ELEKTRONİK MÜHENDİSLİĞİ**

**GEBZE**
**2008**

| GEBZE YÜKSEK TEKNOLOJİ ENSTİTÜSÜ | **MÜHENDİSLİK VE FEN BİLİMLERİ ENSTİTÜSÜ<br>JÜRİ ONAY FORMU** |
|---|---|

## JÜRİ

ÜYE (BAŞKAN) : Yrd.Doç.Dr.Serdar Süer ERDEM

ÜYE : Yrd.Doç.Dr.Abdülkadir BALIKÇI

ÜYE : Yrd.Doç.Dr.Tuğrul YANIK

Gebze Yüksek Teknoloji Enstitüsü Mühendislik ve Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 28/07/2008 tarih ve 2008/25 sayılı kararı ile yukarıdaki öğretim elemanlarından oluşmuş jüri tarafından düzenlenen 21/10/2008 tarihli Tez Savunma Tutanağı neticesinde Yüksek Lisans öğrencisi Uğur COŞKUN'un çalışması GYTE Mühendislik ve Fen Bilimleri Yönetim Kurulu'nun ...../...../..... tarih ve ...../..... sayılı kararıyla Elektronik Mühendisliği Anabilim Dalında Yüksek Lisans tezi olarak onaylanmıştır.

İMZA/MÜHÜR

# ÖZET

**TEZİN BAŞLIĞI:** Controller Area Network ve Uygulaması

**YAZAR ADI :** Uğur COŞKUN

CAN Bus 1986 yılında Robert Bosch tarafından otomobillerdeki çok sayıda sensör ve mikro denetleyiciyi bir kablo yumağı ile bağlamak yerine bunlar arasındaki veri transferini yazılım kontrollü tek bir hattan sağlamak amacıyla geliştirilmiştir. CAN, otomotiv endüstrisindeki en bilinen haberleşme sistemidir. Her ne kadar başlangıçta yalnızca otomotiv uygulamaları için tasarlanmış olsa da yüksek performansı güvenirliliğinden dolayı birçok dağıtık (distributed) endüstriyel kontrol uygulamalarında yaygın olarak kullanılmaktadır. Güvenliğin çok önemli olduğu gerçek zamanlı uygulamalarda da kullanılır. Öyle ki istatistiksel olasılık hesapları sonucunda bir asırda bir tane tespit edilemeyen mesaj hatası yapabileceği tespit edilmiştir.

Uygulama alanı yüksek hızlı ağlardan düşük maliyetli çoklu kablolamalı sistemlere kadar geniştir. CAN Bus, otomobil elektroniği, akıllı motor kontrolü, robot kontrolü, akıllı sensörler, asansörler, makine kontrol birimleri, kaymayı engelleyici sistemler, trafik sinyalizasyon sistemleri, akıllı binalar ve laboratuar otomasyonu gibi uygulama alanlarında maksimum 1Mbit/sn'lik bir hızda veri iletişimi sağlar.

Bu tezde CAN 2.0B standardı incelenmiştir. ARM7 çekirdeği üzerine kurulu STR752FR0 mikro denetleyicisinin donanım özellikleri ve yazılım kütüphanesi açıklanmıştır. Bu mikro denetleyici bir CAN uygulaması yapılmıştır.

# SUMMARY

**THESIS TITLE:** Controller Area Network and Its Application

**AUTHOR :** Ugur COSKUN

CAN Bus standard was developed by Robert Bosch GmbH for the communication of the sensors and microcontrollers in cars with each other through a single bus. Because the ever increasing use of these components in cars results in a high level of wiring burden and communication problems, CAN Bus soon becomes the most popular communication standard in automotive industry. The high performance and robustness of the protocol has expanded its use to many distributed automation and industrial applications. It is also used in the real time applications where reliability is the most important. Statistical probability calculations show that a CAN system can miss only one faulty message in a century.

CAN bus can be used for a wide range of applications, even the ones requiring very high-speeds. It supports the data rates up to 1 Mbit/s for the applications like automotive electronics, intelligent engine control, robot control, intelligent sensors, lift systems, machinery control systems, anti-skid systems, traffic signalization systems, intelligent home and laboratory automation systems.

CAN Bus Specification 2.0B is investigated in this thesis. Hardware configuration and software library of STR752FR0 microcontroller is analyzed. A CAN application is implemented using this microcontroller.

# TEŞEKKÜR

Bu tez çalışması sırasında beni yönlendiren ve yol gösteren sayın hocam Yrd. Doç. Dr. Serdar Süer ERDEM'e, Ortem Elektronik Ltd. Şirketi'ne, eğitim hayatım boyunca beni her konuda destekleyen aileme, tezimin hazırlanması sırasında yardımlarını esirgemeyen arkadaşlarıma teşekkür ederim.

Uğur COŞKUN

# İÇİNDEKİLER DİZİNİ

# SİMGELER VE KISALTMALAR DİZİNİ

| | |
|---|---|
| CAN | : Controller Area Network |
| ID | : Identifier |
| SOF | : Start of Frame |
| CRC | : Cyclic Redundancy Check |
| RTR | : Remote Transfer Request |
| DLC | : Data Length Code |
| RTR | : Remote Transmission Request |
| IDE | : Identifier Extension Bit |
| EOF | : End of Frame |
| TEC | : Transmit Error Counter |
| REC | : Receive Error Counter |
| RISC | : Reduced Instruction Set Computer |
| ARM | : Advanced RISC Machine |
| RTC | : Real Time Clock |
| PWM | : Pulse Width Modulation |
| ADC | : Analog to Digital Converter |
| $I^2C$ | : Inter Integrated Circuit |
| UART | : Universal Asynchronous Receiver Transmitter |
| USB | : Universal Serial Bus |
| GPIO | : General Purpose Input Output |
| GSM | : Global System for Mobile Communications |
| GPRS | : General Packet Radio Service |
| GPS | : Global Positioning System |
| SJW | : Synchronization Jump Width |

# ŞEKİLLER DİZİNİ

# TABLOLAR DİZİNİ

# 1. GİRİŞ

Bu çalışmanın amacı günümüzde otomotiv, endüstriyel otomasyon ve yüksek güvenilirliğe ihtiyaç duyulan birçok alanda kullanılan CAN Bus protokolünü tanıtmak ve bu protokol kullanılarak bir uygulama gerçekleştirmektir.

CAN çok yüksek güvenilirlik seviyesine sahip günümüzün en popüler seri haberleşme protokollerindendir.

2. Bölüm'de CAN Bus standardı detaylı şekilde incelenmiş ve bu protokolü kullanmanın neden doğru seçim olduğu açıklanmıştır. Ayrıca protokolün hata önleme mekanizmaları açıklanarak yüksek güvenilirliğe sahip olmasının nedenleri üzerinde durulmuştur.

3. Bölüm'de, bu tezde gerçekleştirilen uygulamada kullanılan mikro denetleyici, donanım özellikleri ve yazılım kütüphanesi olarak ayrıntılı olarak incelenmiştir.

4. Bölüm uygulamanın gerçekleştirildiği bölümdür. Bu bölümde uygulamaya ait kaynak kodları sunulmuştur ve haberleşmenin performansını önemli ölçüde etkileyen CAN bit zamanlaması açıklanmış ve hesapları yapılmıştır.

Sonuç bölümü olan 5. Bölüm'de uygulama ile elde edilen sonuçlar ve bu uygulamanın gelişmeye açık yönleri anlatılmıştır.

# 2. CAN BUS (Controller Area Network)

## 2.1 Giriş

CAN (Controller Area Network) istasyona değil mesaja dayalı, asenkron seri CSMA/CD (Carrier Sense Multiple Access/Collision Detection), mikro denetleyiciler arası bir haberleşme protokolüdür [1]. Protokol, Bosch Corporation tarafından geleceğin otomobillerindeki birçok elektronik cihaz arasındaki veri aktarımı sorununu çözmek için 1980'lerde geliştirilmiştir. Daha sonra geniş bir kullanım alanı bulmuştur ve günümüzde endüstriyel otomasyon, otomotiv ve mobil cihaz sektöründe yaygın olarak kullanılmaktadır. CAN Bus, zekâyı, hata toleransını ve yüksek derecede güvenilirliği [2] birleştiren, gerçek zamanlı dağıtılmış kontrolü [3] destekleyen bir protokoldür. Genel olarak, CAN zeki sistemlerin haberleşmesine ihtiyaç duyulan her yerde kullanılabilir [4].

CAN'de merkezi bir birim yoktur ve iki veya daha fazla birim arasındaki mesaj iletimi doğrudan, yönetici birim aracılığı olmadan yapılır. CAN, tek-gönder çok-al modunda çalışabilir ve bu mod onu otomotiv uygulamalarında kullanmak için çok avantajlı duruma getirir [5]. CAN kullanmanın diğer avantajları da sistem düzeyinde kolay yapılandırılması ve merkezi bir tanı koyma (diagnose) imkânının bulunmasıdır [6].

CAN protokolü çift yönlü seri veri haberleşmesi için bir ISO standardıdır. ISO 11519 Standardı: Düşük hızlı uygulamalar içindir (Basic CAN ve Standard CAN olarak da bilinir). ISO 11898 Standardı: Yüksek hızlı uygulamalar içindir (Full CAN veya Extended Frame CAN olarak da bilinir).

CAN protokolünün otomotiv, endüstriyel otomasyon ve mühendisliğin çeşitli alanlarında kabul edilmiş olmasının bazı sebepleri şunlardır: Multi-master yeteneği; multicast özelliği; hataların bulunması ve belirtilmesi; gürültüye karşı önemli ölçüde bağışıklık; gecikmelerin en aza indirilmesi; mesaj önceliği; öncelikteki karar mekanizmasının zeki olması (zaman ve bilgi kaybı olmaması); sürekli ve geçici

hataların ayırt edilmesi; başarısız (sürekli hatalı mesaj gönderip hattı gereksiz yere meşgul eden) birimlerin kapatılması; yüksek iletim hızı (1Mbit/s); iletim hızının ayarlanabilmesi; fiziksel iletim için sadece iki tel kullanılması; esneklik; güvenilirlik; düşük maliyet; ISO standardı olması; standart donanımının bulunması [7].

Tablo 2.1'de CAN standardının gelişim aşamaları gösterilmiştir.

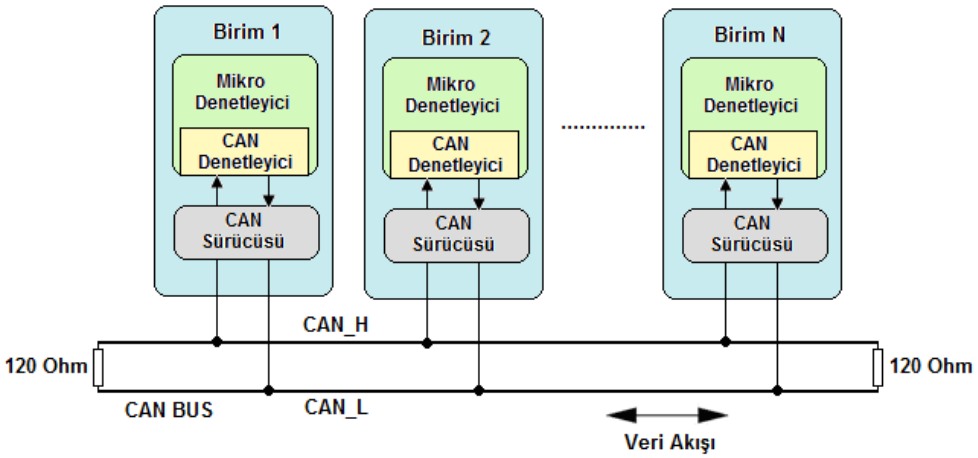
**Tablo 2.1 - CAN Standardı Gelişimi [8]**

| **Versiyon** | **Yıl** | **Değişiklikler** |
|---|---|---|
| 1.0 | 1985 | |
| 1.1 | 1987 | Bit zamanlama ihtiyaçları yeniden tanımlandı |
| 1.2 | 1990 | Osilatör toleransı artırıldı |
| 2.0 | 1991 | Part A-Versiyon 1.2 ile aynı |
| | | Part B – Data Frame ve Remote Frame için ikinci bir format olan "Extended Frame" tanıtıldı |

CAN protokolünün doğru seçim olmasının birçok sebebi vardır [9]:

- Gelişmiş bir standarttır: 20 yıldan uzun bir süredir (1986'dan beri) kullanılmaktadır. Günümüzde piyasada birçok CAN ürünü bulunmaktadır.
- Protokolün donanım uygulaması: CAN protokolü silikon üzerine kurulmuştur. Bu, CAN'e hata bulma ve düzeltme avantajlarını getirir.
- Basit bir iletim ortamı gerektirir.
- Mükemmel hata düzeltme: Bu, CAN'in güçlü yönlerinden biridir. Hata düzeltme algoritmaları geliştirilmiştir. Hata bulma ve hatalı mesajın tekrar gönderilmesi CAN donanımı tarafından otomatik olarak yapılır.
- Hatalı birimi kapatma: CAN sürekli hatalı iletim yapan birimi kapatarak sistemi tıkamasını engeller.

Şekil 2.1'de standart CAN Bus bağlantısı gösterilmiştir.




**Şekil 2.1 – ISO 11898 Standart CAN Bus Bağlantı Şeması**

## 2.2 Temel Kavramlar

CAN Bus şu özelliklere sahiptir:

- Mesaj önceliği
- Konfigürasyon esnekliği
- Veri kararlılığı
- Hata bulma ve bildirme
- Multimaster
- Çoğa gönderim (Multicast)
- Bozulmuş mesajları otomatik olarak yeniden iletme
- Geçici hataları sürekli hata ve eksikliklerden ayırt etme ve sürekli hatalı gönderim yapan cihazları kapatma.

Tasarım saydamlığı ve uygulama esnekliği amacıyla CAN iki farklı alt bölüme ayrılmıştır [7].

1. Veri Bağlantı Katmanı (Data Link Layer)
    a. Lojik Bağlantı Kontrolü (Logical Link Control (LLC))
    b. Ortam Geçiş Kontrolü (Medium Access Control (MAC))
2. Fiziksel Katman (Physical Layer)

Fiziksel katman sinyallerin gerçekte nasıl iletildiği ile ilgilenir. Bit zamanlama, bit paketleme ve senkronizasyon bu katmanda yapılır.

Ortam Geçiş Kontrolü (MAC) katmanı CAN protokolünün çekirdeğini temsil eder. LLC alt katmanından gelen mesajları alır ve LLC alt katmanına gönderilecek olan mesajları kabul eder. MAC alt katmanı mesaj paketleme, karar mekanizması, onaylama, hata bulma ve belirtmeden sorumludur.

LLC alt katmanı mesaj filtreleme, aşırı yüklenme uyarısı ve geri alma işlemi ile ilgilenir.

### 2.2.1 Mesajlar

Veriler farklı fakat sınırlı uzunluktaki sabit biçimli mesajlar şeklinde gönderilir. Hat boş olduğunda hatta bağlı herhangi bir birim yeni mesaj gönderebilir.

CAN Bus'da mesaj iletimi, veri ve zaman kaybı olmadan gerçekleştirilir. Ethernet'de bu mümkün değildir, iki veya daha fazla birim aynı anda hatta mesaj göndermeye başlarsa ikisi de bir bozulma olduğunu anlayıp mesajı geri çeker ve rastgele bir süre sonra mesajı tekrar yayınlar, bu ise zaman kaybına sebep olur. CAN'de ise bu problem öncelik (arbitration) ile çözülmüştür. İki mesaj aynı anda gönderilmeye başlanırsa daha yüksek önceliğe sahip mesaj öncelik hakkını elde eder, zaman ve veri kaybı olmadan iletilir. Önceliği yüksek mesaj gönderilirken daha düşük öncelikli mesaj gönderici birim tarafından anında geri çekilir ve hattaki mesajı dinler. Yüksek öncelikli mesajın iletimi tamamlandığı anda diğer mesajlar gönderilebilir.

### 2.2.2 Veri Yönlendirme

CAN sistemlerinde bir CAN biriminin sistem yapılandırılması (birim adresleri gibi) hakkında herhangi bir bilgiye ihtiyacı yoktur. Bunun bazı önemli sonuçları şunlardır:

- *Sistem Esnekliği:* CAN hattına yeni birimler diğer birimlerde hiçbir yazılım, donanım değişikliği yapılmadan ve uygulama katmanı değiştirilmeden eklenebilir.

- *Mesaj Yönlendirme:* Mesaj içeriği Identifier ile belirtilir. Identifier mesajın hedefini belirtmez, verinin anlamını belirtir, böylece hatta bağlı bütün birimler Mesaj Filtreleme ile mesajın kendilerine ait olup olmadığına karar verirler.

- *Çoğa Gönderim (Multicast):* Mesaj Filtreleme kavramının sonucu olarak herhangi bir sayıdaki birim aynı mesajı aynı anda alıp değerlendirebilir.

- *Veri Kararlılığı:* CAN hattında mesajın eş zamanlı olarak bütün birimler tarafından kabul edildiği veya hiçbir birim tarafından kabul edilmediği garanti edilmiştir. Böylece sistemin veri kararlılığı çoğa gönderim ve hata bulma kavramları tarafından gerçekleştirilir.

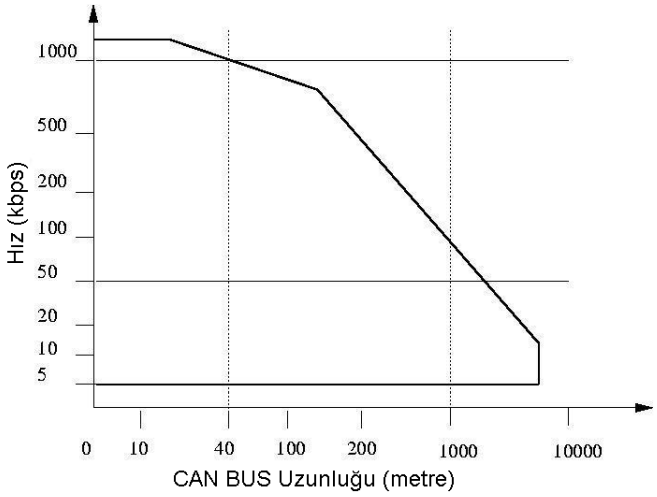
### 2.2.3 CAN Bus Hızı

CAN hattının hızı farklı sistemlerde farklı olabilir. Ancak bir sistem içindeki hız (bit-rate) sabittir. Tablo 2.2'de seçilen bazı hat uzunluklarına karşılık gelen iletim hızları verilmiştir.

**Tablo 2.1 - Hat Uzunluğuna Bağlı Hız Değişimi [10]**

| Hat Uzunluğu (metre) | Maksimum Hız |
|---|---|
| 40 | 1 Mbit/s |
| 100 | 500 kbit/s |
| 200 | 250 kbit/s |
| 500 | 125 kbit/s |
| 6000 | 10 kbit/s |

Şekil 2.2 hat uzunluğuna bağlı olarak iletim hızı değişim eğrisini göstermektedir.




**Şekil 2.2 - CAN Bus Hat Uzunluğu - Hız Değişim Grafiği**

### 2.2.4 Öncelikler

Identifier hattaki iletimde mesajın statik önceliğini tanımlar. Nümerik olarak daha küçük öncelik alanına (arbitration field) sahip olan mesaj önceliğe sahip olur.

### 2.2.5 Uzaktan Veri Talebi (Remote Data Request)

Belirli bir veriye ihtiyaç duyan birim Remote Frame göndererek diğer bir birimden ihtiyaç duyduğu veriyi göndermesini talep edebilir. Data Frame ve buna karşılık gelen Remote Frame aynı Identifier ile adlandırılır.

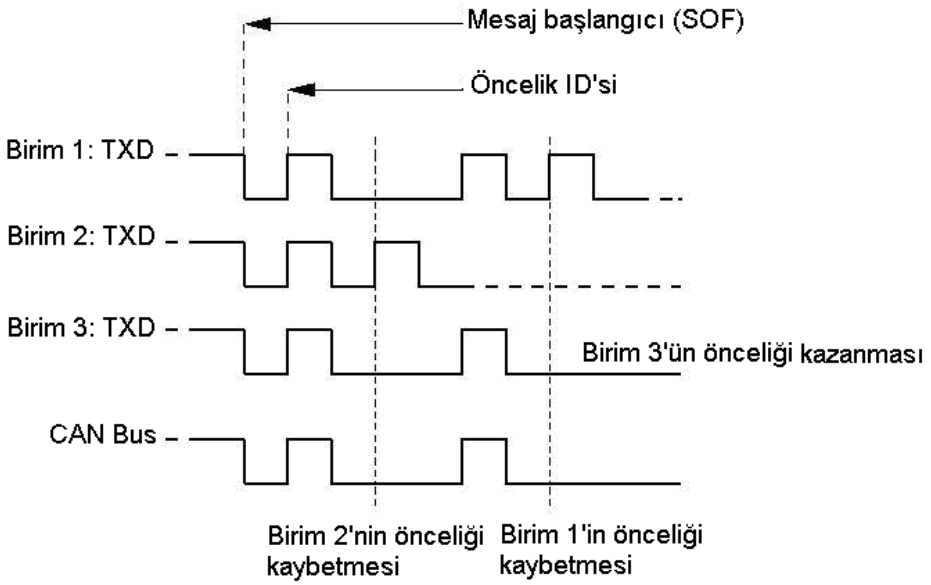
### 2.2.6 Multimaster

Hat boş olduğunda herhangi bir birim mesaj göndermeye başlayabilir. Daha yüksek öncelikli bir mesaj gönderen birim hat geçiş hakkını kazanır.

### 2.2.7 Karar Mekanizması (Arbitration)

Hat boş olduğu anda herhangi bir birim mesaj göndermeye başlayabilir. Eğer iki veya daha fazla birim aynı anda mesaj göndermeye başlarsa, ID üstündeki öncelik bitleri (arbitration field) kullanılarak karar mekanizması işletilir. Bu karar mekanizması veri ve zaman kaybı olmamasını garanti eder. Eğer aynı ID'ye sahip bir Data Frame ve Remote Frame aynı anda iletime başlarsa, Data Frame önceliklidir. Karar verme sürecinde her verici (transmitter) gönderilen bit seviyesi ile hatta gözlenen hat seviyesini karşılaştırır. Eğer seviyeler eşit ise göndermeye devam eder. 'Çekinik' (recessive) bir seviye gönderilip hatta 'baskın' (dominant) bir seviye gözlendiği anda gönderme hakkını kaybetmiş olur ve bir bit daha göndermeden mesajı geri çeker [11].

Şekil 2.3'de karar mekanizmasının işletilmesi gösterilmiştir.




**Şekil 2.3 - CAN Mesaj Gönderme Karar Mekanizması**

### 2.2.8 Güvenilirlik

Veri iletim güvenilirliğini en üst seviyede tutmak için, hatalı mesajı belirlemede güçlü ölçümler, hata belirtme ve kendi kendini test (self-checking) işlemleri bütün birimler tarafından gerçekleştirilir.

- *Hata Belirleme*

Hataları belirlemek için aşağıdaki ölçümler gerçekleştirilir:

  - Gözleme (Vericiler gönderilecek bit seviyeleri ile hattaki bit seviyesini karşılaştırır)
  - CRC (Cyclic Redundancy Check)
  - Bit Doldurma (Bit stuffing)
  - Mesaj Bütünlük Kontrolü (Message Frame Check)

- *Hata Belirleme Performansı*

Hata belirleme mekanizması aşağıdaki özelliklere sahiptir:

  - Bütün global hatalar belirlenir.
  - Vericilerdeki bütün yerel hatalar belirlenir.
  - Mesajdaki 5 adede kadar olan ve rastgele dağılmış hatalar belirlenir.
  - Mesajdaki uzunluğu 15'ten daha az olan burst hataları belirlenir.
  - Mesajdaki herhangi bir tekil sayı hatası belirlenir.

CAN Bus'da fark edilmeden kalan bozulmuş mesaj iletilme ihtimali $4.7x10^{-11}$ den daha azdır.

### 2.2.9 Hata Belirtme ve Düzeltme Zamanı

Bozulmuş mesajlar hatayı belirleyen herhangi bir birim tarafından işaretlenir. Böyle mesajlar iptal edilir ve otomatik olarak yeninden gönderilir. Hatayı belirleme ile sonraki mesajı gönderme arasında geçen süre -eğer daha fazla hata yoksa- en fazla 31 bit zamanıdır.

### 2.2.10 Hata Sınırlama

CAN birimleri geçici ve sürekli hataları ayırt eder. Sürekli hatalı mesaj gönderen birimler kapatılır.

### 2.2.11 Bağlantılar

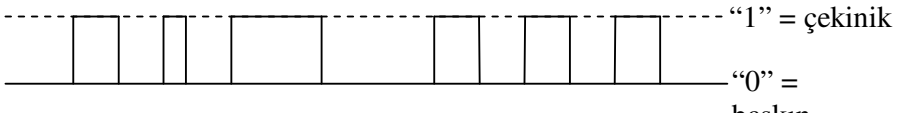
CAN seri iletişim hattı birçok birimin bağlanabileceği bir şebekedir. Bağlanabilecek birim adedinin teorik bir limiti yoktur. Pratikte ise bağlanabilecek toplam birim sayısı gecikme zamanları ve/veya hattaki elektriksek yük ile sınırlıdır.

### 2.2.12 Tekli Fiziksel İletim Ortamı

Hat, bitleri taşıyan tek bir kablodan oluşur. Buradan veriyi yeniden eş zamanlı hale getirme (resynchronization) bilgisi türetilebilir.

### 2.2.13 Hat Değerleri

Hat birbirinin tersi olan iki lojik değeri alabilir: Baskın (dominant) veya çekinik (recessive). Baskın ve çekinik bitlerin eş zamanlı iletimlerinde hattın seviyesi baskın olacaktır. Örnek olarak kablolu-AND implementasyonunda dominant seviye lojik 0, çekinik seviye ise lojik 1 olacaktır. Lojik seviyeler Şekil 2.4'te gösterilmiştir.




**Şekil 2.4 - CAN Bus'daki Lojik Seviyeler**

Hatta birden fazla birimin mesaj göndermesi durumunda hattın hangi seviyede olacağını Tablo 2.3 açıklamaktadır.

Tablo 2.2 - CAN Bus Doğruluk Tablosu (D: Dominant, R: Resesif)

| BİRİM A | BİRİM B | BİRİM C | BUS |
|---|---|---|---|
| D | D | D | **D** |
| D | D | R | **D** |
| D | R | D | **D** |
| D | R | R | **D** |
| R | D | D | **D** |
| R | D | R | **D** |
| R | R | D | **D** |
| R | R | R | **R** |

### 2.2.14 Onay Mekanizması (Acknowledgment)

Bütün alıcılar alınan mesajın tutarlılığını kontrol eder ve mesajın tutarlı ya da tutarsız olduğunu belirtir.

### 2.2.15 Uyku Modu / Uyanma

Sistemin güç tüketimini azaltmak için, bir CAN cihazı uyku moduna ayarlanabilir.

### 2.2.16 Osilatör Toleransı

Bit zamanlama ihtiyaçları 125kbit/s hızlarına kadar seramik resonatör kullanılmasına müsaade etmiştir. CAN standardındaki maksimum hat hızına ulaşmak için quartz osilatör kullanılmalıdır.

## 2.3 Can Mesajları ve Mesaj Çeşitleri

CAN, istasyona değil mesaja dayalı bir protokoldür. Bu, bütün birimlerin bütün iletimleri fark edebileceği anlamına gelir. Sadece özel bir birime mesaj gönderilemez, bütün birimler hattaki trafiğin tamamını kontrol eder. CAN donanımında yerel bir filtreleme yapılır, bu şekilde bütün birimler sadece ilgilendikleri mesajları kabul ederler.

CAN 2.0 tanımlamasına göre iki çeşit mesaj paketi vardır: standart ve genişletilmiş (extended) biçim. Aralarındaki fark identifier bitlerinin sayısıdır. Standart biçimde identifier 11 bit, genişletilmiş biçimde ise 29 bittir. Bu iki formatın da kendine göre avantajlı tarafları vardır. Genişletilmiş formatta çok daha fazla (500 milyondan fazla) çeşitlilikte farklı mesajlar üretilebilir. Standart formatta hat geçiş süresi daha kısadır ve bu format için üretilen donanım da daha çoktur. Standart formatta network iletişimi 2032 farklı mesaj çeşidi ile yönetilir.

1 Mbit/s transfer hızında Standart Frame'in geçiş süresi 110/134 µs (doldurma bitsiz/doldurma bitli) iken Extended Frame'de bu süre130/159 µs kadardır.

CAN kısa mesajlar kullanır. Mesajlarda adres bilgisi bulunmaz, bunun yerine "içerik adresli" olduklarını söylenebilir.

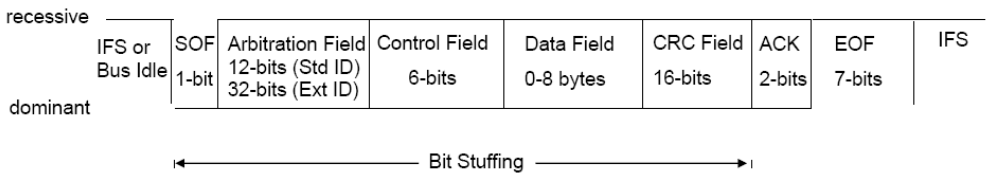
Dört farklı mesaj çeşidi vardır:

1. Data Frame
2. Remote Frame
3. Error Frame
4. Overload Frame

Data Frame ve Remote Frame, Standart Mesaj formatında veya Extended Mesaj formatında olabilir. Bunlar önceki mesajlardan Interframe Space ile ayrılır.

## 2.3.1 Data Frame

En yaygın mesaj tipidir ve hatta veri göndermek için kullanılır. Şekil 2.5 Data Frame ve içeriğindeki bölümlerin bit uzunluklarını göstermektedir.



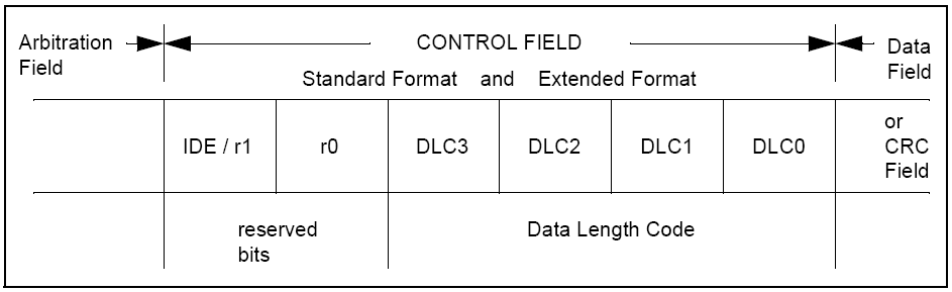

**Şekil 2.5 - CAN Data Frame ve Bit Uzunlukları [8]**

SOF (Start Of Frame): Data Frame ve Remote Frame'in başlangıcını belirtir. Tek bir dominant bitten oluşur.

Arbitration Field: Mesaj önceliğini belirler. (CAN 2.0A'da 11 bit, CAN 2.0B'de 29 bit ID içerir)

RTR (Remote Transfer Request): Data frame'de dominant, remote frame'de resesiftir.

Control Field: Mesajda kaç adet veri baytı bulunduğunu belirten 4 adet veri uzunluk biti içerir. Şekil 2.6'da Control Field gösterilmiştir. Tablo 2.4'te ise 4 bit veri uzunluk bitinin durumlarına göre mesajın kaç bayt veri içerdiği gösterilmiştir.




**Şekil 2.6 – Control Field [8]**

**Tablo 2.3 - Veri Baytları Kodlaması (d: dominant, r: resesif)**

| Veri Baytları Sayısı | Veri Uzunluğu Kodu (DLC) | | | |
|---|---|---|---|---|
| | DLC3 | DLC2 | DLC1 | DLC0 |
| 0 | d | d | d | d |
| 1 | d | d | d | r |
| 2 | d | d | r | d |
| 3 | d | d | r | r |
| 4 | d | r | d | d |
| 5 | d | r | d | r |
| 6 | d | r | r | d |
| 7 | d | r | r | r |
| 8 | r | d | d | d |

SRR: Standart formattaki RTR yerine kullanılmıştır.

IDE (Identifier Extension Bit): 11 bit standart ve 29 bit genişletilmiş yapıyı ayırır. Standart yapıda dominanttır.

Data Field: 0-8 byte arasında veri içerir.

CRC Field: 15-bit checksum bilgisi içerir. Hata belirleme için kullanılır.

ACK (Acknowledgement) Slot: Onay bitidir. Gönderici bunu çekinik olarak gönderir, alıcı mesajı doğru olarak alırsa baskın bir bit gönderir, gönderici bu baskın bitten mesajın doğru olarak teslim edildiğini anlar. Eğer verici baskın bit almazsa mesajı yeniden gönderir.

EOF (End Of Frame): Bütün Data Frame ve Remote Frame'ler 7 adet resesif bit içeren mesaj sonu bitleri ile sonlandırılır.

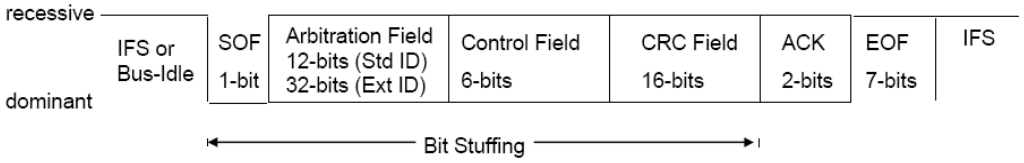
Eğer sistem içinde birkaç birim aynı anda aynı ID'ye sahip mesaj gönderirlerse geçiş önceliği için şu kurallar uygulanır. Data frame remote frame'e karşı önceliğe sahiptir, standart frame extended frame'e karşı önceliklidir.

### 2.3.2 Remote Frame

Data Frame ile benzerdir. Fakat iki tane önemli farklılık vardır.

- RTR biti çekiniktir.
- Data field yoktur.

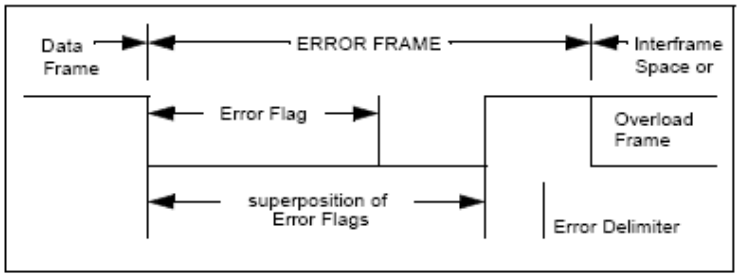
Remote Frame'in amacı veri transfer isteğinde bulunmaktır. CAN birimleri remote framelere otomatik olarak cevap vermek için veya sadece CPU'yu haberdar etmek için kullanılabilir. Remote Frame ve içeriğindeki bölümlerin bit uzunlukları Şekil 2.7'de gösterilmiştir.




**Şekil 2 - Remote Frame [8]**

## 2.3.3 Error Frame

CAN'in mesaj kurallarını ihlal eden mesajlardır. Hatta bağlı bir birim hata belirlediğinde error frame iletir ve böylece diğer birimler de hatadan haberdar olur ve error frame iletir. Bu durumda mesajı ileten birim otomatik olarak mesajı yeniden gönderir. CAN'deki kontrol algoritması bir birimin sürekli hatalı mesajlar ileterek hattı meşgul etmesini engeller. Şekil 2.8 Error Frame'i göstermektedir.



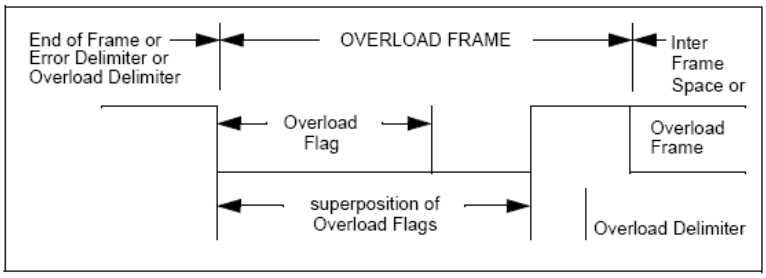

**Şekil 2.8 - Error Frame [8]**

Error frame, error flag ve error delimiter içerir. Error flag aynı değere sahip ardışık 6 bit içerir. Bu, bit doldurma kurallarını ihlal eder. Error delimiter 8 çekinik bitten oluşur ve diğer birimlerin error flagi ilk belirlediklerin de onların da error flag göndermesi için bir aralık sağlar.

## 2.3.4 Overload Frame

Çok meşgul olan bir birim tarafından sonraki data veya remote frame'in gecikmesini istemek için gönderilir. Fakat günümüzdeki CAN denetleyicileri overload frame kullanılmasını gerektirmeyecek kadar hızlı ve zekidir.

Bir sonraki remote veya data frame'i geciktirmek için art arda en fazla iki overload frame gönderilebilir. Overload Frame'nin yapısı Şekil 2.9'da gösterilmiştir.




**Şekil 2.9 - Overload Frame [8]**

## 2.4 Mesaj Filtreleme

Mesaj filtreleme işlemi Identifier'a bağlı olarak yapılır. İsteğe bağlı maske yazmaçları filtreleme yapılmaması yönünde ayarlanabileceği gibi istenen bir grup Identifier'ın alınan mesaj tamponuna bağlanmasını ve sadece o mesajların kabul edilmesi yönünde de kullanılabilir.

Maske yazmaçları mesaj filtrelemek için aktif ve pasif yapılabilir. Maske yazmaçlarının uzunluğu Indentifier'ın bir kısmını kaplayabileceği gibi tamamını da kaplayabilir.

## 2.5 Mesaj Geçerliliği

Mesajın geçerli olduğuna karar verilen nokta, verici (transmitter) ve alıcı (receiver) için farklıdır.

*Vericiler*: Verici için mesajın geçerli olması EOF (End Of Frame) sonuna kadar hata olmaması durumundadır. Eğer mesaj bozulmuşsa, mesajın otomatik olarak yeniden gönderimi öncelik kurallarına göre yapılır.

*Alıcılar*: Alıcılar için mesaj EOF'in sondan önceki bir bitine kadar hata olmaması durumunda geçerlidir. EOF'in son biti önemsiz (don't care) olarak değerlendirilir ve baskın bir değer Form Error'a sebep olmaz.

## 2.6 Mesaj Kodlama

Bit Dizisi Kodlaması: Mesaj frame bölümleri olan Start Of Frame, Arbitration Field, Control Field, Data Field ve CRC sırası bit doldurma (bit stuffing) metodu ile kodlanır. Verici gönderilecek mesajda beş adet aynı değere sahip ardışık bite rastladığı anda otomatik olarak araya ters değerde bir bit ekler.

Data Frame ve Remote Frame'in geriye kalan diğer bölümleri (CRC Delimiter, Ack Field ve End Of Frame) sabittir ve bunlara bit doldurma uygulanmaz. Error Frame ve Overload Frame'de sabit formlara sahiptir ve bit doldurma metodu bu mesajlar için de kullanılmaz.

## 2.7 Hata Yönetimi

### 2.7.1 Hata Belirleme

5 farklı hata önleme ve kontrol yöntemi vardır:

1. Bit gözleme (Bit monitoring)
2. Bit doldurma (Bit stuffing)
3. Frame kontrolü (Frame check)
4. Onay kontrolü (Acknowledgement check)
5. CRC (Periyodik Fazlalık Kontrolü) (15 bit)

### 2.7.2 Hata Belirtme

CAN protokolünde kullanılan hata kontrolü ve düzeltmesinin CAN sisteminin performansı için büyük bir önemi vardır. Hata kontrolü hattaki hatalı mesajların bulunmasını ve bunların yeniden iletilmesini amaçlar. Hatta bağlı bütün CAN kontrolörleri mesajdaki hataları belirleyebilir. Eğer bir hata bulunursa, bunu bulan birim hatta bir error flag gönderir, bu şekilde hattaki trafik akışı değişir. Diğer birimler de error flag tarafından sebep olan hatayı belirler (eğer hala orijinal hatayı belirlememişlerse) ve hatalı mesajı kabul etmezler.

Her birim iki sayıcının durumunu değiştirebilir. Bunlar Gönderme Hatası Sayıcısı (Transmit Error Counter) ve Alım Hatası Sayıcısıdır (Receive Error Counter). Bu sayıcılar belli bir değeri geçtikten sonra hat Error Passive ve Bus Off gibi konumlara geçebilir.

## 2.8 Hata Sınırlama Mekanizması

Bütün CAN kontrolörleri hattaki mesajlardaki hataları bulmaya çalışır. Eğer bir hata bulunursa, bunu bulan birim bir error flag iletir ve hattaki mesaj trafiği bozulur. Diğer birimler de eğer daha önce hatalı mesajdan kaynaklanan hatayı belirlememişlerse bu error flagi fark eder ve hatalı mesajı reddeder.

Her bir birim iki hata sayıcısının değerini değiştirebilir. Bunlar gönderme hata sayıcısı ve alıcı hata sayıcısıdır. Bu sayaçların değerlerinin nasıl artırılıp azaltılacağına dair çeşitli kurallar vardır. Kısaca, hata belirleyen bir verici kendi gönderme hata sayacını bu mesajı dinleyen birimler alıcı hata sayaçlarını artırmadan daha önce artırır.

Bütün birimler Error Active modda çalışmaya başlar. Sayaçlardan herhangi biri 127'yi geçtiği zaman birim Error Passive moda geçer. Gönderme hata sayıcısı 255'i geçtiğinde ise birim Bus Off durumuna geçer. Bu durum Tablo 2.5'te gösterilmiştir.

**Tablo 2.4 - Hata Modları ve Şartları (TEC: Transmit Error Counter, REC:ReceiveError Counter) [8]**

| **CAN Birimi Durumu** | **Gerekli Şart** |
|---|---|
| Normal Mode | TEC = REC = 0 |
| Error Active Mode | TEC < 128 ve REC < 128 |
| Error Passive Mode | TEC >= 128 veya REC >= 128 |
| Bus Off Mode | TEC >255 |

- Error active durumundaki birim hata belirlediğinden active error flag iletir.
- Error passive birim hata belirlediğinde passive error flag iletir.
- Bus Off durumundaki birim hatta hiçbir mesaj göndermez.

Hata sayaçlarının değerlerinin artırılıp azaltılmasına ait kurallar karmaşıktır ancak prensip basittir. Gönderme hataları 8 hata puanı, alım hataları ise 1 hata puanıdır. Doğru iletilen ve/veya alınan mesajlar da hata sayaçlarının değerinin azaltılmasına sebep olur.

## 2.9 CAN Bit Zamanlaması

CAN Bus bit zamanlaması Bölüm 3.3'de açıklanmıştır. Ayrıca uygulamaya ait bit zamanı hesaplaması da aynı bölümde yapılmıştır.

# 3. STR750 MİKRO DENETLEYİCİSİ

## 3.1 Giriş

Uygulamada ST Microelectronics tarafından üretilen STR752FR0 mikro denetleyicisi kullanılmıştır. Bu, ARM7TDMI-S çekirdeği üzerine kurulu, 32-bit, RISC mimarisinde bir mikro denetleyicidir.

Kaynak kod C Programlama Dili'nde oluşturulmuştur ve ST Microelectronics tarafından sağlanan yazılım kütüphanesi kullanılmıştır.

## 3.2 STR750 Donanım Özellikleri

STR750 ailesi, 32-bit RISC (Reduced Instuction Set Computer) çekirdeğine sahip ARM7TDMI ile yüksek performans, çok düşük güç tüketimi, yoğun kod özelliği, zengin çevresel birimler ve ST Microelectronics'in en gelişmiş 0.18µ Flash teknolojisini birleştirmektedir. STR750 ailesi, genelde birçok işlemcide bulunan çevresel birimleri içerdiği gibi USB, CAN, gelişmiş motor kontrol zamanlayıcısı ve saat darbesi hatası belirleme gibi bazı gelişmiş özellikleri de içerir. Giriş gerilimi olarak 3.3V veya 5V kullanılabilir ve genişletilmiş sıcaklık sahasında (-40 - +105°C) kullanılabilir. Bütün bu özellikler bu aileyi geniş bir uygulama sahasında kullanılabilecek orijinal bir genel amaçlı mikro denetleyici ailesi yapar[12]:

- CAN uygulamaları
- USB çevresel birimleri, UPS, alarm sistemleri
- Fırçasız motor sürücüleri
- Programlanabilir lojik denetleyiciler (PLC), devre kesiciler, inverterler
- Tıbbi ve portatif gereçler

STR750F ailesi 64 pin ve 100 pin paketler içerir. Uygulamamızda kullanılan STR752FR0 64 pine sahiptir. Her iki paket tipi de aşağıdaki ortak özellikleri içermektedir:

### 3.2.1 CAN Çevresel Birimi

STR752FR0'da CAN çevresel birimi CAN Çekirdeği (CAN Core), Mesaj Hafızası (Message RAM), Mesaj İşleyici (Message Handler), Kontrol Yazmaçları (Control Registers) ve Modül Arayüzü'nden (Module Interface) oluşur. Fiziksel katmana bağlantı için ek bir donanım gereklidir ve bizim uygulamamızda MCP2551 kullanılmıştır.

CAN hattına bağlantı için ayrı Mesaj Nesneleri yapılandırılır. Mesaj Nesneleri ve kabul filtrelemesi için kullanılan Tanıtıcı Maskeleri (Indentifier Masks) mesaj hafızasında depolanır.

Mesajların işlenmesi ile ilgili bütün fonksiyonlar Mesaj İşleyici'de gerçekleştirilir. Bu fonksiyonlar kabul filtrelemesi, mesajların CAN Çekirdeği ve Mesaj Hafızası arasındaki iletimi, iletim isteklerinin işlenmesi ve modül kesmelerinin üretilmesini içerir.

CPU, CAN çevresel biriminin yazmaç grubuna modül arayüzü vasıtası ile doğrudan erişebilir. Bu yazmaçlar CAN Çekirdeği ve Mesaj işleyicinin denetimi/yapılandırılması için ve Mesaj Hafızası'na erişim için kullanılır.

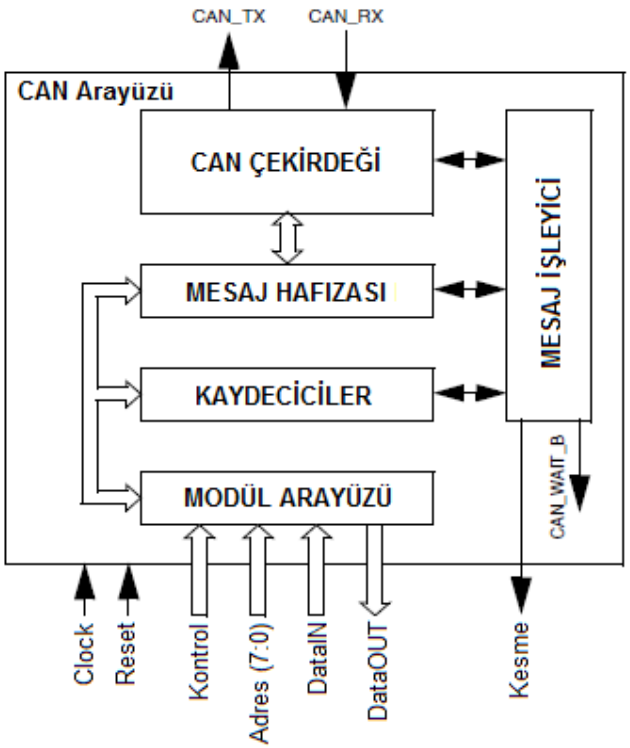
#### 3.2.1.1 Başlıca Özellikler

- CAN protokol versiyon 2.0 A ve B'yi destekler.
- 1 MBit/s hat hızını destekler.
- 32 Mesaj Nesnesi vardır.
- Her bir Mesaj Nesnesi'nin kendi ayrı tanıtıcı maskesi vardır.
- Programlanabilir FIFO modu vardır.
- Maskelenebilir kesmeler bulunur.

- Zaman tetiklemeli CAN uygulamaları için otomatik yeniden göndermeyi iptal edebilme modu vardır.
- Kendi-kendini test için programlanabilir geri döngü modu (Loop-back mode) bulunur.
- APB (Advanced Peripheral Bus) hattı için iki adet 16-bit modül arayüzü vardır.

### 3.2.1.2 Blok Diyagram

CAN çevresel birimine ait blok diyagram Şekil 3.1'de gösterilmiştir. CAN Çekirdeği, CAN protokolü denetleyicisidir ve Rx/Tx kaydırma yazmaçlarını (shift register) içerir. Mesaj Hafızası, Mesaj Nesnelerini ve Tanıtıcı Maskeleri'ni depolar. Yazmaçlar, CAN arayüzünü denetlemek ve yapılandırmak için bütün yazmaçları içerir. Mesaj İşleyici, CAN Çekirdeği'ndeki Rx/Tx Kaydırma Yazmaçları ile Mesaj Hafızası arasındaki veri transferini denetler ve Kontrol ve Yapılandırma Yazmaçları'nda programlandığı şekilde kesmeleri üretir. Modül Arayüzü, APB 16-bit bus ile CAN çevresel birimi yazmaçları arasında bir arayüz vazifesi görür.




**Şekil 3.1 - CAN Arayüzü Blok Diyagramı**

### 3.2.1.3 Fonksiyonel Özellikler

Alınan mesajlar eğer Mesaj İşleyici'nin kabul filtresini geçerlerse uygun Mesaj Nesneleri'nde saklanır. Mesajın bütün bölümleri Mesaj Nesnesi'nde depolanır. Aynı anda birçok Mesaj Nesnesi için transfer isteği olabilir ve Mesaj Nesneleri kendi dâhili önceliklerine göre gönderilir. Mesaj Nesnesi'nin yapılandırılmasına bağlı olarak talep edilen bir mesajın iletimi otomatik olarak yapılabilir.

CAN çevresel birimi iletim sırasında önceliği kaybeden veya bozulan mesajların yeniden iletimini destekler. Bu özellik Zaman Tetiklemeli CAN (TTCAN, ISO11898-1) ortamı için devre dışı bırakılabilir.

CAN çevresel biriminde bazı test amaçlı modlar vardır. Sessiz Mod'da sadece CAN hattı dinlenir ve geçerli mesajlar alınır, hatta hiçbir mesaj gönderilmez. Bu modda hattı etkilememek için dominant bit gönderilmez ve CAN hattını analiz etmek için kullanılabilir. Geri Döngü Modu kendi kendini test için kullanılır ve kendi gönderdiği mesajları hattan alınan mesajlar gibi depolar. Temel Mod'da CAN çevresel birimi Mesaj Hafızası olmadan çalışır.

### 3.2.1.4 Yazmaçlar (Registers)

256 baytlık adres alanı 16 bitlik yazmaçlar şeklinde düzenlenmiştir. CAN protokolü ile ilişkili yazmaçlar; çalışma modlarının, CAN bit zamanlamasının yapılandırılmasını kontrol eder ve durum bilgisi sağlar.

İki yazmaç grubu (IF1 ve IF2) CPU'nun Mesaj Hafızası'na erişimini kontrol eder ve bu şekilde CPU erişimi ile mesaj alımı/gönderimi arasındaki çakışmayı önler. Bu yazmaçlar transfer edilecek mesajlar için bir arabellek vazifesi görerek bu işi başarır. Tek bir transferde, Mesaj Hafızası ile IFn Mesaj Arabellek Yazmaçları arasında tam bir Mesaj Nesnesi transfer edilebileceği gibi Mesaj Nesnesi'nin bir bölümü de transfer edilebilir. Mesaj Arabellek Yazmaçları, Mesaj Hafızası'ndaki Mesaj Nesneleri'nin tam bir kopyasını içerir.

Mesaj Hafızasında 32 Mesaj Nesnesi vardır. CPU erişimi ile mesaj alımı/gönderimi arasındaki çakışmadan kaçınmak için CPU bu Mesaj Nesneleri'ne doğrudan erişemez. Bu erişimler IFn Yazmaçları üzerinden gerçekleştirilir.

Mesaj İşleyici Yazmaçları; salt okunur yazmaçlardır. Bu yazmaçlar Mesaj Hafızası'ndaki bütün Mesaj Nesneleri için Yeni Veri (NewDat), Bekleyen Kesme (IntPnd), Mesaj Geçerli (MsgVal) bilgilerini içerir ve CPU bu yazmaçları okuyarak hangi Mesaj Nesnesi'nin veri baytlarında güncelleme olup olmadığını, bekleyen kesme isteği olup olmadığını ve Mesaj Nesnesi'nin geçerli olup olmadığını kontrol edebilir.

#### 3.2.1.5 CAN Haberleşmesi

Bir Mesaj Nesnesi'nin yapılandırılması iki Arayüz Yazmaçları'ndan birinin Maske, Öncelik, Kontrol ve Veri alanlarını istenen değerlere programlamakla yapılır. Karşılık gelen IFn Komut İstek Yazmacı'na yazmakla IFn Mesaj Arabellek Yazmaçları, Mesaj Hafızası'ndaki adreslenen Mesaj Nesnesi'ne yüklenir.

STR752FR0'da IFn Komut Maske Yazmacına yazılan değer Mesaj Nesnesi'nin bir kısmının veya tamamının transfer edileceğini belirler. Öncelikle, Mesaj Nesnesi'nin değiştirilmeyecek kısmı Mesaj Hafızası'ndan okunur ve sonra Mesaj Arabellek Yazmaçları'nın tam içeriği Mesaj Nesnesi'ne yazılır.

Mesaj Nesneleri, gönderilecek veya alınacak mesajların bölümlerinin bir araya getirilerek anlamlı bir bütün oluşturduğu nesnelerdir. Mesaj Hafızası'nda yer alırlar. Bir sonraki iletimde sadece öncekine göre değişen kısımlar değiştirilerek verimli bir iş yapılır ve bu iş için gerekli süre kısaltılmış olur.

Eğer bekleyen çeşitli kesmeler varsa, CAN Kesme Yazmacı, kronolojik sıraya bakmadan en yüksek öncelikli kesmeyi işaret edecektir.

CAN bit zamanlamasının yapılandırılmasındaki küçük hatalar hemen bir hata ile sonuçlanmasa da, CAN hattının performansını önemli ölçüde azaltır. Çoğu durumda, CAN bit senkronizasyonu hatalı bit zamanlamasını düzeltecektir ve

nadiren bir hata mesajı üretilecektir. Ancak iki birim mesajı aynı anda göndermeye başlayıp öncelik alma durumu söz konusu olduğunda mesajın yanlış noktada örneklenmesi göndericilerin Error Passive olmasına sebep olacaktır. Bu durum devam ederse de birimler Bus Off olarak CAN hattından izole edilirler. Bu yüzden CAN bit zamanının yapılandırılmasına dikkat edilmelidir.

### 3.2.2 Gömülü Flash ve RAM içeren ARM7TDMI-S Çekirdeği

STR750 ailesi gömülü ARM çekirdeğine sahiptir ve böylece bütün ARM araçlarıyla ve yazılımlarıyla uyumludur. Yüksek performanslı ARM7TDMI-S CPU ile zengin çevresel birimleri ve gelişmiş Giriş/Çıkış (I/O) yeteneklerini birleştirir. Bu ailenin bütün üyeleri gömülü olarak yüksek hızlı tek voltaj gerektiren FLASH ve yüksek hızlı RAM'e sahiptir.

### 3.2.3 Gömülü Flash Hafızası

Bank 0, programları ve verileri depolamak için 256KB gömülü Flash içerir. Ekstra Bank 1, 16KB RWW (Read While Write – Yazarken Okunabilen) hafıza içerir ki bu anında silme/programlama yeteneği kazandırır. Bu iki bölüme ayırma özelliği uygulama parametrelerini depolamak için idealdir.

- Burst modda yapılandırıldığında Flash hafızaya erişim CPU saat hızında gerçekleştirilir. Bu modda ardışık adreslere erişimlerde hiç gecikme olmaz, rastgele adreslere yapılan erişimlerde ise 1 saat darbesi kadar gecikme vardır (maksimum 60 MHz).
- Burst modda yapılandırılmadığında Flash hafızaya erişim CPU saat hızında gerçekleştirilir ve hiç gecikme olmaz (maksimum 32 MHz).

### 3.2.4 Gömülü SRAM

16 KB gömülü SRAM'a hiç gecikme olmaksızın CPU hızında erişilir (okuma/yazma).

### 3.2.5 Gelişmiş Kesme Yöneticisi (Enhanced Interrupt Controller )

Standart ARM kesme yöneticisine ilave olarak, STR750F 32 kesme vektörü olan ve 16 öncelik seviyesine sahip iç içe geçmiş kesme yöneticisi içerir. Eklenen bu donanım bloğu, minimum gecikmeyle esnek kesme yönetimi özelliği sağlar.

### 3.2.6 Seri Hafıza Arayüzü (Serial Memory Interface – SMI)

Bu seri hafıza ara yüzü 4 adede kadar seri Flash aygıtına doğrudan bağlanabilir. Bu ara yüz veriye ulaşmak, doğrudan kod çalıştırmak veya uygulamayı dışarıdaki birimden başlatmak (boot) için kullanılabilir. Her biri 16 MB genişliğine kadar olan 4 farklı hafıza bölgesini adresleyebilmektedir.

### 3.2.7 Saat Darbeleri ve Başlama (Clocks and Start-Up)

Yeniden başladıktan sonra veya Düşük Güç Modu'ndan çıkarken, CPU saat darbesi 5 MHz'lik dâhili RC osilatör (FREEOSC) tarafından anında sağlanır, böylece uygulama kodu herhangi bir gecikme olmadan çalışmaya başlamış olur. Bu işlemle paralel olarak, 4/8 MHz'lik harici osilatör etkinleştirilir ve bu işe adanmış bir sayıcı tarafından stabilizasyon zamanı gözlenir [13].

Bir osilatör arızası algılandığında, XT1 pini üzerinde saat darbesi kaybolduğunda, otomatik olarak dâhili FREEOSC osilatörüne geçilir ve bir kesme üretilir.

Çalışma modunda, AHB ve ABP saat darbesi hızları PLL ve önbölücü (prescaler) kullanılarak çok çeşitli frekanslara ayarlanabilir: Flash'dan çalışırken AHB için 60 MHz'e kadar ve ABP için 32MHz'e kadardır (SRAM'den çalışırken bu hızlar 64 MHz ve 32MHz'dir).

Yavaş çalışma modunda (Slow Mode) güç tüketimini azaltmak için AHB saat darbesi hızı belirgin olarak düşürülebilir.

Yerleşik saat darbesi yöneticisi (Clock Controller) aynı zamanda fazladan osilatöre veya PLL'e ihtiyaç duymadan 48 MHz USB saat darbesini de sağlar. Örnek olarak, 4 MHz'lik bir kristal osilatör ile paralel olarak 60 MHz'lik AHB saat darbesi, USB için 48 MHz saat darbesi ve ABP çevresel birimleri için 30 MHz'lik saat darbesi elde etmek mümkündür [13].

### 3.2.8 Başlama Modları (Boot Modes)

Başlama sırasında, dört başlama seçeneğinden birini seçmek için başlama pinleri kullanılır [12].

- Dâhili Flash'dan başlama
- Harici seri Flash hafızdan başlama
- Dâhili başlangıç yükleyicisinden (boot loader) başlama
- Dâhili SRAM'den başlama

SMI hafızadan başlama, seri Flash'dan başlatmaya izin verir. Alternatif olarak, STR750F dâhili başlangıç yükleyicisinden başlayabilir bu da UART'dan başlatmayı gerçekleştirir.

### 3.2.9 Güç Kaynağı Modları

Uygulamaya bağlı olarak aşağıdaki güç modlarından herhangi biri kullanılabilir.

- **Güç Modu 1: Tek bir harici 3.3 V güç kaynağı.** Bu düzende dâhili lojik tarafından ihtiyaç duyulan $V_{CORE}$ gerilimi dâhili olarak ana voltaj regülatörü tarafından üretilir ve $V_{BACKUP}$ kaynağı ise dâhili düşük güç voltaj regülatörü

tarafından üretilir. Bu düzenin avantajı tek bir 3.3 V güç kaynağına ihtiyaç duymasıdır.

- **Güç Modu 2: 3.3 V ve 1.8 V'luk çift harici güç kaynağı.** Bu düzende VREG_DIS pininin yüksek seviyeye gelmesi zorlanarak dâhili voltaj regülatörleri kapatılır. $V_{CORE}$, harici olarak $V_{18}$ and $V_{18REG}$ güç pinlerinden, $V_{BACKUP}$ ise $V_{18\_BKP}$ pininden sağlanır. Bu güç düzeni hâlihazırda 1.8 V güç regülatörü olan uygulamalarda güç tüketimini azaltmayı amaçlamaktadır.
- **Güç Modu 3: Tek bir harici 5.0 V güç kaynağı.** Bu düzende dâhili lojik tarafından ihtiyaç duyulan $V_{CORE}$ gerilimi dâhili olarak ana voltaj regülatörü tarafından üretilir ve $V_{BACKUP}$ kaynağı ise dâhili düşük güç voltaj regülatörü tarafından üretilir. Bu düzenin avantajı tek bir 5.0 V güç kaynağına ihtiyaç duymasıdır.
- **Güç Modu 4: 5.0 V ve 1.8 V'luk çift harici güç kaynağı.** Bu düzende VREG_DIS pininin yüksek seviyeye gelmesi zorlanarak dâhili voltaj regülatörleri kapatılır. $V_{CORE}$, harici olarak $V_{18}$ and $V_{18REG}$ güç pinlerinden, $V_{BACKUP}$ ise $V_{18\_BKP}$ pininden sağlanır. Bu güç düzeni 5V seviyesinde giriş/çıkış sağlamayı amaçlamaktadır. 5.0V ile güç verildiğinde USB çevre birimi çalışmamaktadır.

### 3.2.10 Düşük Güç Modları

STR750F 5 adet düşük güç modunu destekler: SLOW, PCG, WFI, STOP ve STANDBY [13].

- SLOW Modu: System saat darbesi hızı düşürülür. Alternatif olarak, PLL ve ana osilatör durdurulabilir ve sistem düşük güçlü saat darbesi ($f_{RTC}$) tarafından sürülebilir. Bu 32.768 kHz'lik harici osilatör olabileceği gibi dâhili düşük güçlü RC osilatör de olabilir.
- PCG Modu (Peripheral Clock Gating Mode): Çevresel birimler kullanılmadığında, onların APB saat darbeleri ayrılarak güç tüketimi optimize edilir.

- WFI Modu (Wait For Interrupts – Kesmeyi Bekle): Sadece CPU saat darbesi durdurulur, bütün çevresel birimler çalışmaya devam eder ve kesme oluştuğunda CPU uyanabilir.
- STOP Modu: Bütün saat darbeleri ve çevresel birimler durdurulur. Aynı zamanda osilatörleri ve Ana Voltaj Regülatörü'nü durdurmak da mümkündür ($V_{CORE}$, tamamen $V_{18\_BKP}$ tarafından beslendiğinde). Bu mod ile SRAM ve yazmaç içerikleri tutularak en düşük güç tüketiminin başarılması amaçlanmıştır. Sistem, harici kesmeler (external interrupts) / uyanma hatları (wake-up lines) veya isteğe bağlı olarak çalışır durumda bırakılabilen RTC zamanlayıcısı tarafından uyandırılabilir. RTC'nin saat darbesi sağlayıcısı 32.768 kHz'lik harici kristal veya Düşük Güçlü RC osilatör olabilir.

  Alternatif olarak STOP mod, main osilatör veya Flash ya da Ana Voltaj Regülatörü'nün açık bırakılarak uyandıktan sonra hızlı bir başlangıç yapılabilmesi esnekliğini de sunar (fakat bu fazladan güç tüketimine sebep olacaktır).
- STANDBY Modu: Bu mod sadece tek kaynaklı güç modlarında uygulanabilir ve yükselen sıcaklıklarda bile en düşük güç tüketimini başarmayı amaçlamaktadır. Dijital güç kaynağı ($V_{CORE}$) tamamen kaldırılır (yüksek ortam sıcaklıklarında bile sızıntı olmaz). SRAM ve bütün yazmaç içerikleri kaybedilir. Sadece $V_{18\_BKP}$ tarafından beslenen RTC'nin gücü kesilmez. STR750F, STANDBY'dan tekrar RUN moduna WKP_STDBY pininin tetiklenmesiyle veya RTC sayıcısındaki alarm zaman aşımı ile geçebilir.

$V_{DD\_IO}$ güç kaynağının gücü, STANDBY modu dâhil bütün düşük güç modlarında kesinlikle kesilmemelidir.

### 3.2.11 Diğer Çevresel Birimler

STR750 ailesi zengin ve gelişmiş kesme kabiliyetleri olan çevresel birimlere sahiptir [12].

#### 3.2.11.1 DMA

Esnek, 4 kanal genel amaçlı DMA, hafızadan hafızaya, çevresel birimden hafızaya ve hafızadan çevresel birime transfer yapılmasını yönetebilir. DMA denetleyicisi dairesel tampon hafıza yönetimini destekler ve denetleyici tampon hafızanın sonuna ulaştığında kesme üretilmesini engeller.

DMA ana çevresel birimler ile kullanılabilir: UART0, SSP0, Motor kontrol PWM zamanlayıcısı, standart TIM0 zamanlayıcısı ve ADC.

#### 3.2.11.2 RTC (Real-Time Clock)

RTC, uygun bir yazılım ile saat ve takvim fonksiyonlarının gerçekleştirilebileceği sürekli çalışan sayıcılar sağlar ve alarm kesmesi ve periyodik kesme üretir. Saat darbesi kaynağı olarak harici 32.768 kHz osilatör veya dâhili düşük güçlü RC osilatör kullanılabilir. RC'nin tipik frekansı 300 kHz'dir ve ayarlanabilir.

#### 3.2.11.3 WDG (Watchdog Timer)

Watchdog sayıcısı 16 bit aşağı doğru sayan bir sayıcıya ve 8 bir ön bölücüye (prescaler) dayalı olarak çalışır. Bir problem oluştuğunda cihazı yeniden başlatmak için veya zaman aşımı amaçlarıyla serbest çalışan bir zamanlayıcı olarak kullanılabilir.

#### 3.2.11.4 Timebase Timer (TB)

Bu zamanlayıcı 16 bit otomatik olarak yeniden yüklenen bir sayıcıya bağlıdır, giriş/çıkış pinlerine bağlanmamıştır. Yazılım tetikleyicisi olarak veya gerçek zamanlı işletim sisteminin (RTOS) zamanlama tablosunu gerçekleştirmek için kullanılabilir.

#### 3.2.11.5 Senkronize Edilebilir Standart Zamanlayıcılar (TIM2:0)

Bu üç standart zamanlayıcı 16 bit otomatik olarak yeniden yüklenen bir sayıcıya bağlıdır ve 2 giriş yakalama (input capture), 2 de çıkış karşılaştırma (output

compare) özelliğine sahiptir. Senkronizasyon ve durum değiştirme için zamanlayıcı bağlama özelliği kullanılarak PWM zamanlayıcısı ile birlikte çalışabilirler. Standart zamanlayıcılardan herhangi biri PWM çıkışı üretmek için kullanılabilir. Bir zamanlayıcı (TIM0) DMA kanalına eşleştirilebilir.

#### 3.2.11.6 Motor Kontrol PWM Zamanlayıcısı

Motor Kontrol PWM Zamanlayıcısı (PWM), 6 kanala çoklanmış şekilde üç fazlı PWM şeklinde görülebilir. 16 bit PWM üreteci tam modülasyon yeteneğine sahiptir (0 - %100), kenara veya merkeze hizalanmış desenler oluşturulabilir ve ölü zaman (dead time) eklemeyi de destekler. Standart TIM zamanlayıcıları ile ortak birçok özelliği vardır ve senkronizasyon veya durum değiştirme amacı ile zamanlayıcı bağlama özelliği sayesinde standart TIM zamanlayıcıları ile birlikte çalışabilir. PWM zamanlayıcısı bir DMA kanalına eşleştirilmiştir.

#### 3.2.11.7 $I^2C$ Bus

$I^2C$ Bus ara yüzü multi-master veya slave modda çalışabilir. Standart ve hızlı modları destekler (400 kHz'e kadar).

#### 3.2.11.8 Yüksek Hızlı Üniversal Asenkron Alıcı Verici (UART)

3 adet UART ara yüzü 2 Mbit/s hızında haberleşebilmektedir. CTS ve RTS sinyallerinin donanım olarak yönetimini sağlar ve LIN Master yeteneğine sahiptir. İşlemci ve çevresel birim arasındaki iletimi optimize etmek için her biri 16 byte'lık iki FIFO (alma/gönderme) gerçekleştirilmiştir. Bir UART (UART0), DMA denetleyicisi tarafından desteklenebilir.

#### 3.2.11.9 Senkron Seri Çevresel Birim (SSP)

İki SSP, standart full duplex 4 pin ara yüz modunda, master olarak 8 Mbit/s (SSP1) veya 16 Mbit/s (SSP0), slave olarak ise 2.66 Mbit/s hızında haberleşebilir. İşlemci ve çevresel birim arasındaki iletimi optimize etmek için her biri 8x16 bit word'luk iki FIFO (alma/gönderme) gerçekleştirilmiştir. SSP'ler, Motorala SPI veya

TI SSI protokollerini destekler. Bir SSP (SSP0), DMA denetleyicisi tarafından desteklenebilir.

#### 3.2.11.10 Universal Seri Bus (USB)

STR750F, USB Full speed 12 Mbit/s ile uyumlu USB çevresel birimi içerir. Yazılımla yapılandırılabilen son nokta (endpoint) ayarları ve askıya alma/devam etme (suspend/resume) desteği vardır. USB için ayrılmış 48 MHz saat darbesi, dâhili ana PLL'den üretilir. USB'nin çalışması için $V_{DD}$ 3.3V ± %10 seviyesinde olmalıdır.

#### 3.2.11.11 ADC (Analog Dijital Dönüştürücü)

10 bit analog dijital dönüştürücü, 16 adede kadar harici kanalı tek bakış (single-shot) veya tarama modunda dönüştürür. Tarama modunda, seçilmiş bir grup analog giriş üzerinde sürekli olarak dönüşüm gerçekleştirilir. Örnekleme zamanı dâhil asgari dönüşüm zamanı 3.75µs'dir.

Analog watchdog özelliği, en fazla dört kanala kadar çevrilen voltaj seviyelerinin çok hassas olarak gözlenmesini sağlar. Dönüştürülen voltaj seviyesi programlanan eşik seviyelerinin dışında olduğu anda bir kesme üretilir.

Uygulamada A/D dönüştürücüleri ve zamanlayıcıları senkronize etmek için TIM0, TIM2 ve PWM zamanlayıcıları dâhili olarak ADC başlama tetiği olarak bağlanabilir.

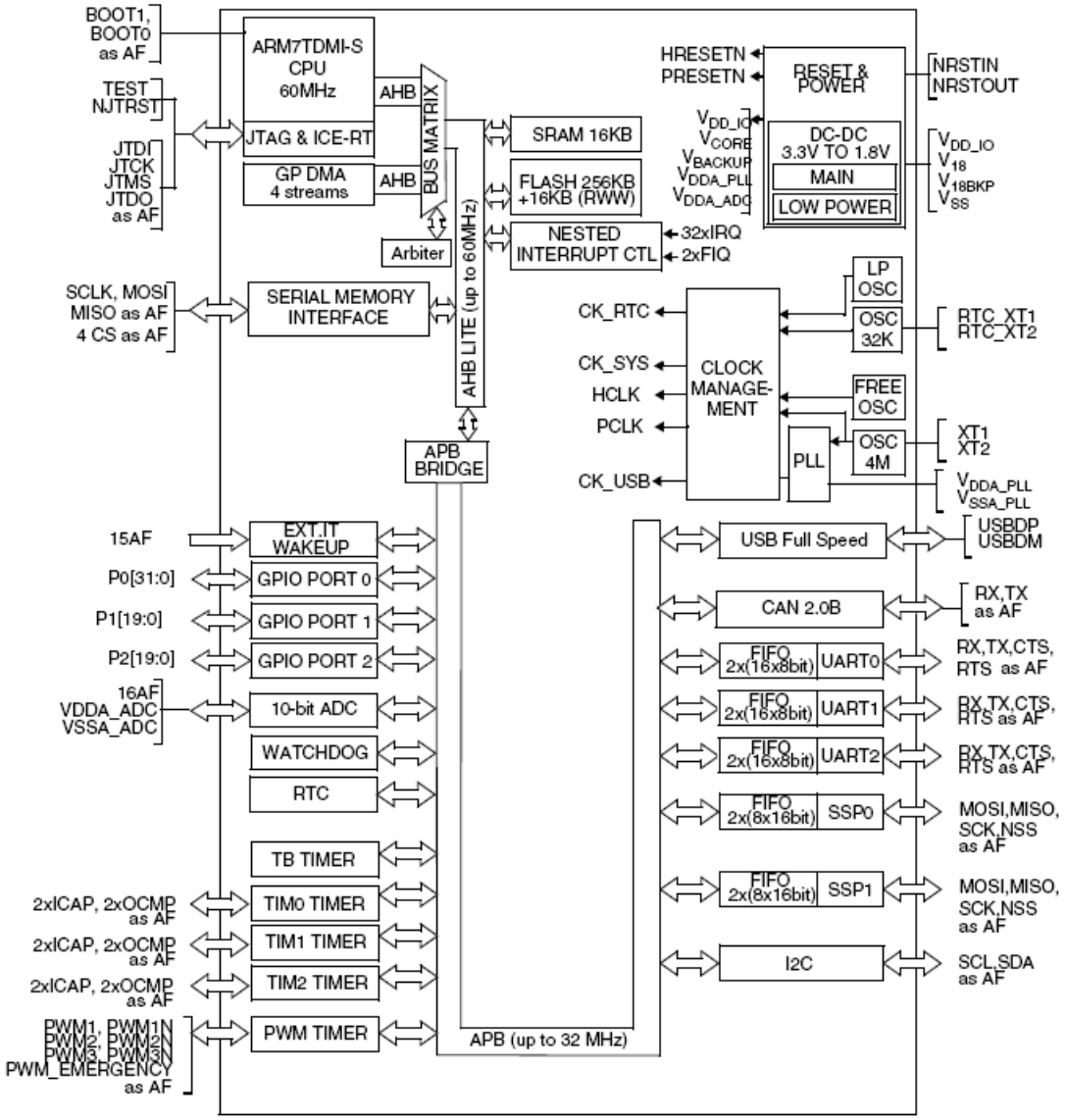
#### 3.2.11.12 GPIO (Genel Amaçlı Giriş Çıkış)

72 adet GPIO'nun her biri yazılım ile çıkış (push-pull veya open-drain) olarak, giriş (pull-up veya pull-down olabilir ya da olmayabilir) olarak veya çevresel birime göre değişimli fonksiyonda yapılandırılabilir. GPIO'ların birçoğu dijital veya analog değişimli fonksiyonlarla paylaşılmıştır.

### 3.2.12 Blok Diyagram

Şekil 3.2'de STR752F mikro denetleyicisinin blok diyagramı gösterilmiştir.

Blok diyagramdaki bazı pinlerde yer alan AF (Alternate Function) belirteci, Giriş/çıkış pini üzerinde değişimli fonksiyon tanımlı olduğunu göstermektedir.




**Şekil 3.2 - STR752F Blok Diyagramı [12]**

## 3.3 STR750 Yazılım Kütüphanesi

STR750 yazılım kütüphanesi, bütün standart STR750 çevresel birimleri için cihaz sürücüleri içeren bir yazılım paketidir. Bu kütüphane sayesinde bütün çevresel birimler hakkında derinlemesine bir çalışma yapmadan uygulamalarda STR750 ailesini kullanmak mümkündür. Sonuç olarak, bu kütüphaneyi kullanmak bize oldukça fazla zaman kazandıracak, böylece kodlama için aşırı vakit ayırmanın önüne geçilecek ve uygulama geliştirme maliyetleri düşecektir.

Her bir cihaz sürücüsü, çevresel birimin özelliklerini kapsayan bir takım fonksiyonlar içerir. STR750 çevresel birimlerinin ve bu birimlere ait yazmaçların (register) hafıza-haritalı (memory-mapped) olmasından dolayı, bir çevresel birim 'C' kodlarıyla kolaylıkla yönetilebilir. 'C' dilinde geliştirilen kaynak kod tamamen dokümante edilmiştir. Bu kütüphaneyi kullanabilmek için 'C' programlama bilgisi gereklidir. Kütüphanedeki tüm kodlar 'C' dilinde olduğu için ARM uyumlu herhangi bir 'C' derleyicisi ile birlikte kullanılabilir.

CAN çevresel biriminde tamamen yapılandırılabilen 32 Mesaj Nesnesi vardır. Yazılım kütüphanesindeki bir dizi fonksiyon bu çevresel birimin ve Mesaj Nesneleri'nin yapılandırılması için kullanılır. Bu fonksiyonlar sayesinde kullanıcının CAN çevresel birimine ait yazmaçları yapılandırmak için bit düzeyinde işlem yapmasına gerek kalmaz.

Tablo 3.1'de CAN kütüphanesinde kullanılan çeşitli fonksiyonlar listelenmiştir

**Tablo 3.1 - CAN Kütüphanesi Fonksiyonları**

| **Fonksiyon Adı** | **Açıklama** |
|---|---|
| CAN_DeInit | CAN çevresel birimi yazmaçlarını varsayılan değerlerine ilkler. |
| CAN_Init | CAN hücresini ilkler ve bit hızını ayarlar. |
| CAN_StructInit | Her bir CAN_StructInit üyesini varsayılan değeri ile doldurur. |
| CAN_EnterInitMode | CAN ilkleme moduna girilir. |
| CAN_LeaveInitMode | İlkleme modundan çıkılır (normal moda geçilir). |
| CAN_EnterTestMode | CAN test moduna girilir. |
| CAN_LeaveTestMode | Test modundan çıkılır (normal moda geçilir). |
| CAN_SetBitrate | Standart CAN hızını ayarlar. |
| CAN_SetTiming | CAN zamanlamasını belirli parametrelerle ayarlar. |
| CAN_SetUnusedMsgObj | Mesaj nesnesini kullanılmamış şeklinde yapılandırır. |
| CAN_SetTxMsgObj | Mesaj nesnesini TX (gönderme) şeklinde yapılandırır. |
| CAN_SetRxMsgObj | Mesaj nesnesini RX (alma) şeklinde yapılandırır. |
| CAN_InvalidateAllMsgObj | Bütün mesaj nesnelerini kullanılmamış olarak yapılandırır. |
| CAN_ReleaseMessage | Mesaj nesnesini serbest bırakır. |
| CAN_ReleaseTxMessage | Gönderme mesaj nesnesini serbest bırakır. |
| CAN_ReleaseRxMessage | Alma mesaj nesnesini serbest bırakır. |
| CAN_SendMessage | Mesaj gönderme işlemini başlatır. |
| CAN_ReceiveMessage | Eğer ulaşmış bir mesaj varsa, onu alır. |
| CAN_WaitEndOfTx | Güncel iletim bitene kadar bekler. |
| CAN_BasicSendMessage | BASIC moda mesaj iletimine başlar. |
| CAN_BasicReceiveMessage | Eğer ulaşmış bir mesaj varsa, BASIC modda onu alır. |
| CAN_IsMessageWaiting | Alınan mesajın bekleme durumunu test eder. |
| CAN_IsTransmitRequested | Mesaj gönderme isteği olup olmadığını test eder. |
| CAN_IsInterruptPending | Mesaj nesnesinin kesme durumunu test eder. |
| CAN_IsObjectValid | Mesaj nesnesinin geçerliliğini (kullanıma hazırlık) test eder. |

CAN_Init fonksiyonu, CAN çevresel birimini CAN_InitStruct'da belirtilen parametrelere göre ilkler. Bu fonksiyon içerisinde CAN_EnterInitMode(), CAN_SetBitrate(), CAN_LeaveInitMode(), CAN_LeaveTestMode() fonksiyonları çağrılır.

Bu ve bunun gibi diğer fonksiyonda kullanılan CAN yapılandırma parametreleri 75x_can.h dosyasında tanımlanmıştır. Bu dosya içinde farklı tanımlamalar yaparak fonksiyonlara farklı parametreler geçirmek mümkündür. Bunun pratikte faydası yokmuş gibi görünse de özel uygulamalar için (örneğin standart hızlar dışında farklı bir CAN bit hızı kullanılması) faydalı olabilirler.

Yazılım kütüphanesinde birbirleri ile bağlantılı fonksiyonlar mevcuttur. Bu fonksiyonlardan birini kullanmak diğerini de kullanmayı gerektirir. Örnek olarak kesmeleri etkin hale getirmek için kullanılan CAN_EnterInitMode(CAN_CR_IE) fonksiyonunu müteakip CAN_LeaveInitMode fonksiyonu kesin olarak kullanılmalıdır.

Mesaj Nesneleri, CAN kütüphanesinde üzerinde yoğunlaşılan ve yapılandırılan temel öğelerdir. Bu fonksiyonların birçoğu IF0 ve IF1 yazmaçlarını okuyup bu değerlere göre Mesaj Nesnelerini yapılandırmak için kullanılır.

Gönderme ve alma işlemi yapılırken hangi Mesaj Nesnesi numarasının kullanılacağı belirtilirken nesnelerin öncelik seviyeleri hesaba katılmalıdır. Nesne tipine bakılmaksızın, daha düşük numaralı nesne (0) en büyük önceliğe, daha büyük numaralı nesne ise (31) en küçük önceliğe sahiptir. En iyi performans için, nesne listesinde nesneler arasında boşluklar olmamalıdır.

Mesaj gönderimi sırasındaki öncelik için kullanılan CAN ID, idLow ve idHigh olarak iki parçadan oluşmuştur. Bu ID aralığı seçimine dikkat edilmelidir. ID'ler donanım tarafından filtrelenmektedir, bu yüzden idLow ve idHigh bölümlerinin bütün kombinasyonları her zaman beklenen sonucu üretmeyebilir. Bu ölçüt alınan bir mesaja şu şekilde uygulanır: Alınan ID ve Mask ID bit düzeyinde VE operatörü işleminden geçirilerek Öncelik ID'si elde edilir. Sonuç olarak idLow için bazı LSB bitleri sıfırlanmış bir değer seçmek ve idHigh için ise mantıksal olarak

idLow'u içeren ve bazı LSB bitleri bir olan değerler seçmek daha iyidir. Örnek olarak 0x100-0x3FF çalışacaktır fakat 0x100-0x2FF çalışmayacaktır, çünkü 0x100 mantıksal olarak 0x2FF içinde yoktur (0x100 & 0x2FF = 0).

Fonksiyonlarda Mesaj Nesneleri'ne çağrı yapmak için Komut İstek Yazmacı'na Mesaj Nesnesi numarası değil 1+MesajNesnesi değeri yazılmalıdır.

## 3.3.1 Yazılım Kütüphanesi İçeriği

Yazılım kütüphanesi dizin yapısı Şekil 3.3'de gösterilmiştir.

### 3.3.1.1 Örnekler (Examples)

Bu dizin her bir çevresel birim için ayrı bir alt dizin içerir. Bir çevresel birimin nasıl kullanılacağına ait tipik bir örneği çalıştırmak için gerekli minimum dosyalar:

- ***Readme.txt:*** Örneği tanımlayan ve nasıl çalışır hale getirileceğini açıklayan kısa bir metin dosyasıdır.
- ***75x_conf.h:*** Kullanılan çevresel birimleri yapılandırmak ve çeşitli tanımlamaları yapmak için başlık dosyasıdır.
- ***75x_it.c:*** Kesme sonrasında icra edilen fonksiyonları içeren kaynak dosyasıdır (kullanılmayan fonksiyon gövdeleri boştur).
- ***main.c:*** Örnek program dosyasıdır.

Bütün örnekler kullanılan geliştirme ortamından bağımsızdır.

### 3.3.1.2 Kütüphane (Library)

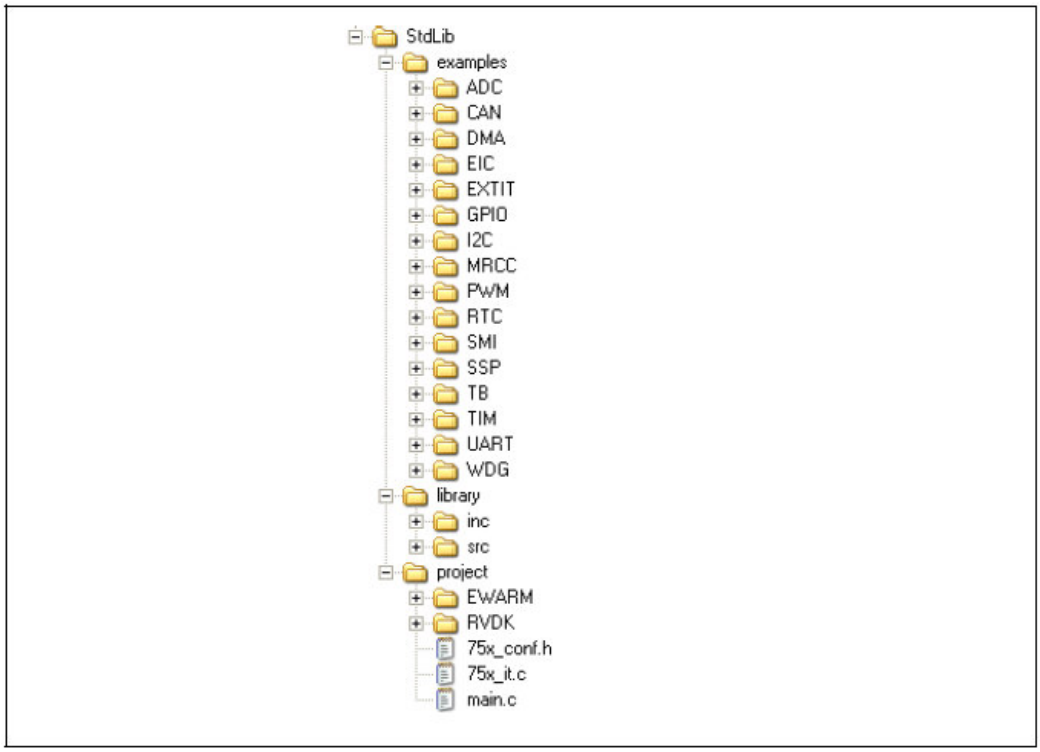
Bu dizin kütüphanenin çekirdeğini oluşturan bütün alt dizinleri ve dosyaları içerir.

- ***inc*** alt dizini, kullanıcı tarafından değiştirilmemesi gereken yazılım kütüphane başlık dosyalarını içerir:

- ***75x_type.h:*** Diğer bütün dosyalarda kullanılan ortak veri tiplerini ve listelemeleri içerir.
- ***75_map.h:*** Çevresel birimlerin hafıza haritalamalarını ve yazmaçların veri yapılarını içerir.
- ***75_lib.h:*** Diğer bütün başlık dosyalarını içeren ana başlık dosyasıdır.
- ***7x_ppp.h*** (her çevresel birim için bir başlık dosyası): Fonksiyon prototiplerini, veri yapılarını ve listelemeleri içerir.

- ***src*** alt dizini, kullanıcı tarafından değiştirilmemesi gereken yazılım kaynak dosyalarını içerir:
  - ***7x_ppp.c*** (her çevresel birim için bir kaynak dosyası): Her bir çevresel birimin fonksiyon gövdelerini içerir.

Bütün kütüphane dosyaları *ANSI-C* olarak kodlanmıştır ve geliştirme ortamından bağımsızdır.




**Şekil 3.3 - Yazılım Kütüphanesi Dizin Yapısı [14]**

#### 3.3.1.3 Proje (Project)

Bu dizin bütün kütüphane dosyalarını derleyen bir standart proje şablonu içerir. Aynı zamanda yeni bir proje oluşturmak için kullanıcı tarafından değiştirilebilecek dosyalar da bu dizindedir.

- ***75x_conf.h:*** Varsayılan olarak tanımlanmış bütün çevresel birimler için yapılandırma ayarlarını içeren başlık dosyasıdır.
- ***75x_it.c:*** Kesme fonksiyonlarını içeren kaynak dosyasıdır (bu şablon içinde fonksiyon gövdeleri boştur).
- ***main.c:*** Ana program gövdesidir.

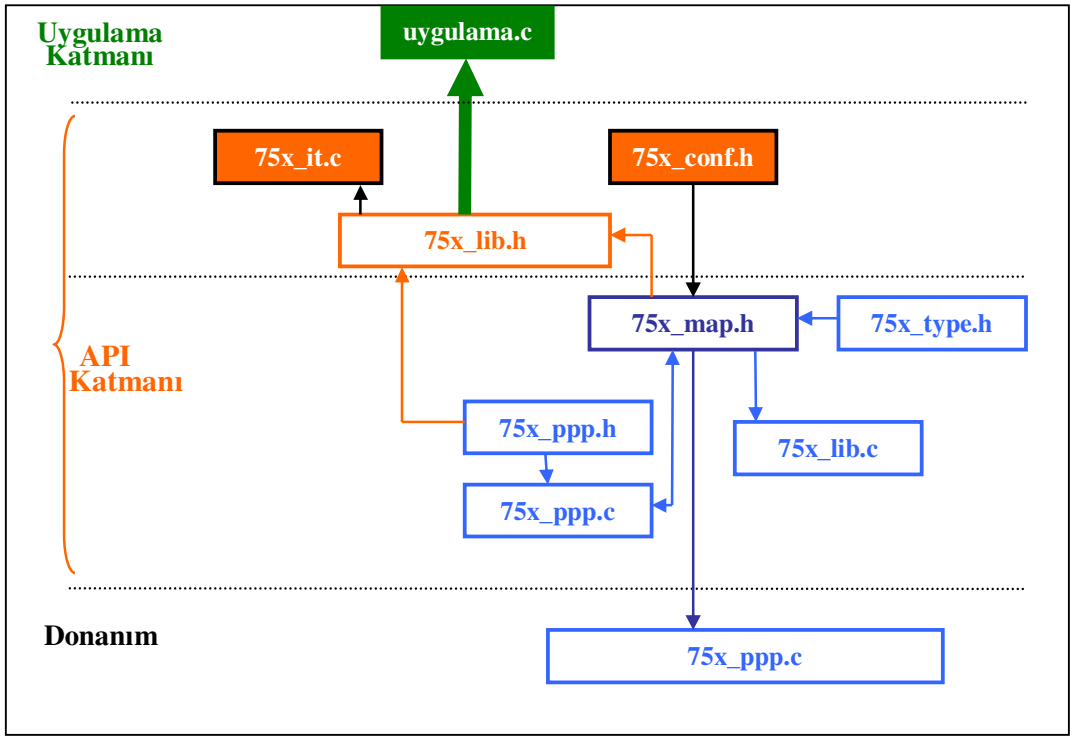
### 3.3.2 Dosya Tanımları

Yazılım kütüphanesinde çeşitli dosyalar kullanılmıştır. Tablo 3.2 kütüphanede kullanılan farklı dosyaları listelemekte ve tanıtmaktadır.

**Tablo 3.2 - Yazılım Kütüphanesi Dosya Tanımları [14]**

| Dosya Adı | Tanım |
|---|---|
| 75x_conf.h | Parametre yapılandırma dosyasıdır. Herhangi bir uygulama çalıştırılmadan önce kütüphane ile ilişkilendirmek için çeşitli parametreleri belirtmek için kullanıcı tarafından değiştirilmelidir. Eğer şablon kullanılıyorsa çevresel birimler aktif/pasif hale getirilebilir ve harici osilatörün değeri de değiştirilebilir. |
| main.c | Ana örnek program dosyasıdır. |
| 75x_it.c | Çevresel birim kesme fonksiyonları dosyasıdır. Uygulamada kullanılan kesme fonksiyonlarına ait kodlar kullanıcı tarafından değiştirilebilir. Aynı kesme vektörüne ait birden çok kesme isteği olması durumunda, fonksiyon kesme bayraklarını bakarak kesmenin gerçek kaynağını belirler. Bu fonksiyonların isimleri yazılım kütüphanesinde tanımlanmıştır. |
| 75x_lib.h | Bütün çevresel birimlere ait başlık dosyalarını içeren başlık dosyasıdır. Kullanıcı uygulamasının kütüphane ile ilişkisini kurmak için dâhil edilmesi gereken tek dosyadır. |
| 75x_lib.c | Hata ayıklama modu başlangıç dosyasıdır. Her birinin belirli bir çevre biriminin ilk adresini işaret ettiği değişken işaretçilerinin tanımlarını içerir. Aynı zamanda hata ayaklama moduna girildiğinde çağrılan fonksiyonların tanımlarını da içerir. Bu fonksiyon tanımlanan işaretçilerinin başlangıç ayarlarını yapar. |
| 75x_map.h | Bu dosya hem geliştirme hem de hata ayaklama amaçlarıyla hafıza haritalamasını gerçekleştirir ve fiziksel yazmaç adreslerinin tanımlarını içerir. Bu dosya bütün çevre birimleri için sağlanmıştır. |
| 75x_type.h | Ortak tanımlar dosyasıdır. Bütün çevre birimleri tarafından kullanılan ortak tipleri ve sabitleri içerir. |
| 75x_ppp.c | C dilinde yazılan PPP çevresel birimine ait sürücü kaynak kodudur. |
| 75x_ppp.h | PPP çevresel birimi için başlık dosyasıdır. PPP çevresel birimine ait fonksiyon tanımlarını ve bu fonksiyonlarda kullanılan değişkenleri içerir. |

Yazılım kütüphanesi mimarisi ve dosya içerme ilişkisi Şekil 3.4'de gösterilmiştir.




**Şekil 3.4 - Yazılım Kütüphanesi Dosya Mimarisi [14]**

Her çevre birimin *75x_ppp.c* şeklinde kaynak kod dosyası ve *75x_ppp.h* şeklinde başlık dosyası vardır. *75x_ppp.c* dosyası ilgili çevresel birimi kullanabilmek için gerekli olan bütün yazılım fonksiyonlarını içerir. Bütün çevresel birimler için *75x_map.h* adında tek bir hafıza haritalama dosyası bulunmaktadır. Bu dosya, hem geliştirme hem de hata ayıklama için gerekli olan bütün yazmaç tanımlamalarını içerir.

*75x_lib.h* başlık dosyası bütün çevresel birimlerin başlık dosyalarını içerir. Bu, kullanıcı uygulamasının kütüphane ile ilişkilendirilmesi için içerilmesi gereken tek dosyadır [14].

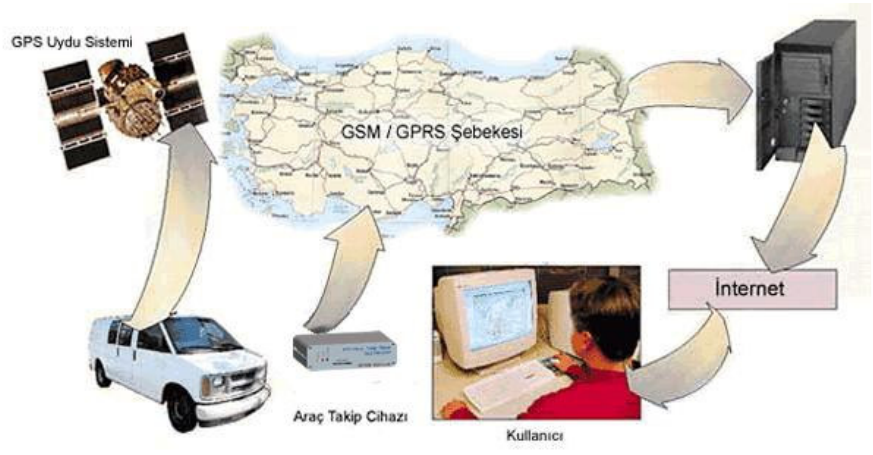
# 4. CAN UYGULAMASI

## 4.1 Giriş

Bu bölümde donanım ve yazılım olarak gerçekleştirilen bir CAN Bus uygulaması anlatılacaktır. Uygulamanın detayları, CAN Bus performansı için gerekli olan bit zamanı hesaplamaları ve uygulamaya ait açıklamalı kaynak kodlar bu bölümde yer almaktadır.

## 4.2 Uygulama Tanımı

Bu uygulama bir araç takip cihazında kullanılmak üzere gerçekleştirilmiştir. Araç takip cihazı araç üzerine monte edilir ve GPS uydularından aldığı konum, hız, yön gibi bilgiler ile araçtan toplanan yakıt seviyesi, sıcaklık, motor devri, ses, görüntü vb. gibi uygulamanın gerektirdiği daha birçok bilgi ile birleştirip GSM alt yapısını kullanarak bir merkeze gönderir. Kullanıcılar internet üzerinden bu bilgilere ulaşabilirler. Şekil 4.1'de araç takip sisteminin çalışma prensibi gösterilmiştir.

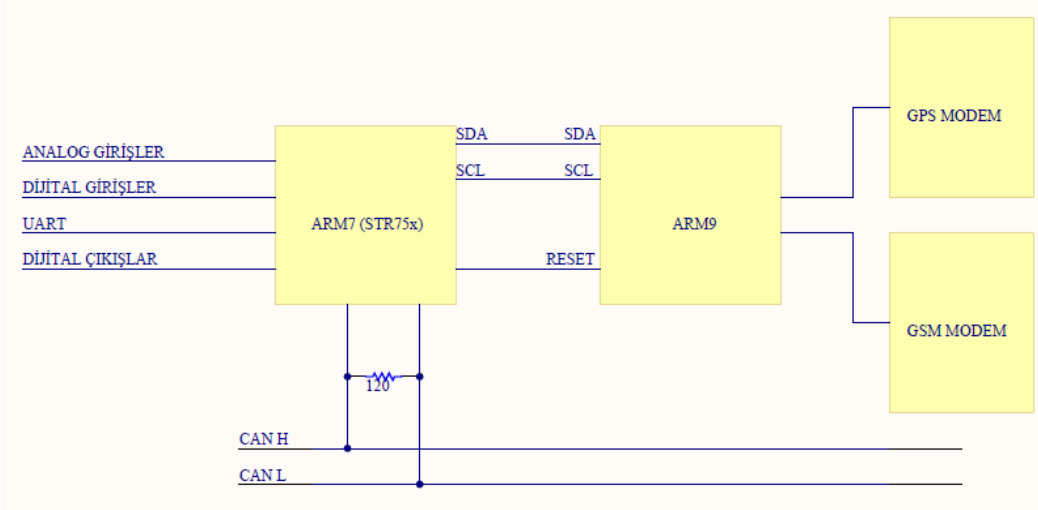



**Şekil 4.1 – Araç Takip Sistemi Çalışma Prensibi [15]**

Araç takip sistemleri kullanılarak araçlardan toplanan bütün bilgiler internet üzerinden geçmişe dönük olarak da rapor edilebilmektedir. Araca ait bilgilerin düzenli periyotlarla kaydedilmesi, arıza ve benzeri durumların izlenebilmesi ve tüm

bunların raporlanması önemlidir. Bu veriler motorlu araç üreticilerine arıza yapan sistemlerin düzeltilmesinde ve mevcut sistemlerin iyileştirilmesinde yol gösterir. Ayrıca ilerde bu sistem kullanılarak araca uzaktan müdahale (CAN Bus'a bağlı bir birime yapılandırma mesajı göndererek farklı çalışmasını sağlamak) etmek ve böylece servis maliyetlerini düşürmek mümkün olacaktır.

Bizim uygulamamızın içinde yer alacağı sistem; ARM7 ve ARM9 mimarisinde iki adet mikro denetleyici, GPS Modem, GSM Modem ve bu birimlerin çevresel elemanlarından oluşur. Sistemin blok şeması Şekil 4.2'de gösterilmiştir. İki mikro denetleyici aralarında $I^2C$ protokolünü kullanarak haberleşir. ARM7 işlemci araç üzerindeki analog girişleri, dijital girişleri ve CAN Bus hattı üzerindeki veriyi okuyup değerlendirir ve bunları $I^2C$ hattı üzerinden ARM9 işlemciye iletir. ARM9 işlemci bu verileri GPS konum bilgileri ile birleştirerek istenen bir mobil telefona SMS olarak veya GPRS Modemi kullanarak sabit bir IP adresine gönderir.




**Şekil 4.2 – Araç Takip Cihazı Blok Şeması**

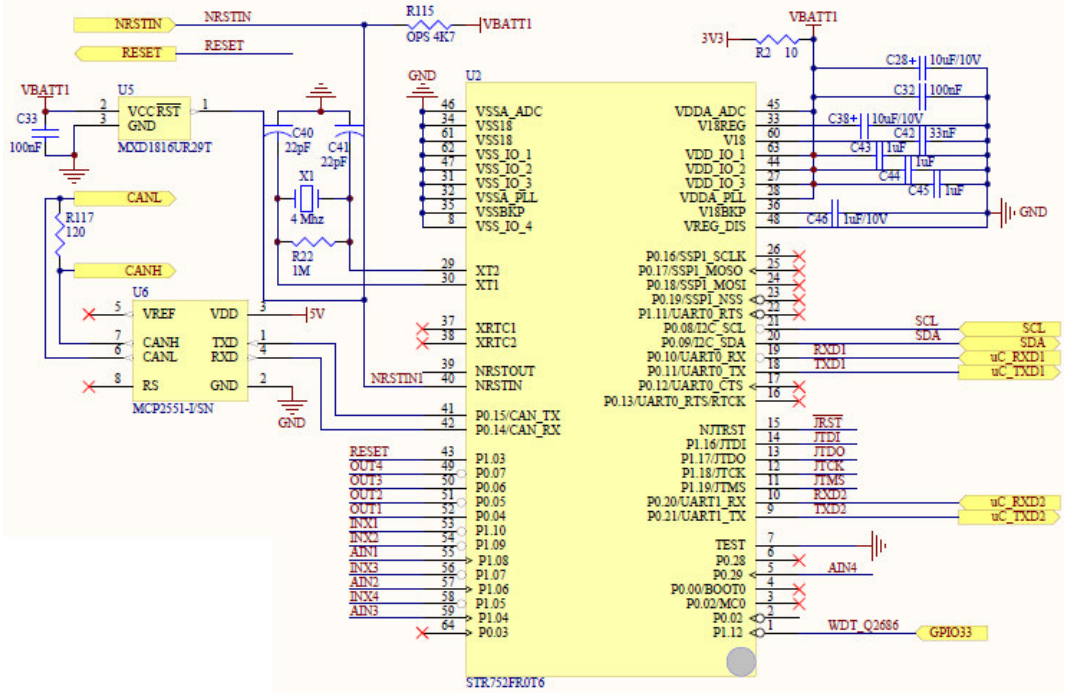
## 4.3 ARM7 ile Araç İçi Veri Toplama ve Araç Kontrolü

Bizim uygulamamız araç takip sisteminin ARM7 mikro denetleyicisi ile gerçekleştirilen kısmını kapsamaktadır.

ARM7 işlemci sistemde şu görevleri üslenmiştir:

- CAN Bus hattındaki ihtiyaç duyulan bilgileri okuyup değerlendirerek anlamlı hale getirip ARM9 işlemciye iletmek.
- Frekansı azami 10 kHz olan 4 adet kare dalga işareti okuyup Hz cinsinden ARM9 işlemciye iletmek (Harici kesmeler kullanılarak).
- 4 kanal analog girişi 10 bit analog dijital dönüştürücü (ADC) kullanarak okuyup ARM9 işlemciye iletmek.
- 2 adet UART (Universal Asynchronous Receiver-Transmitter) kullanılarak diğer çevresel birimlerle veya PC ile haberleşmek.
- 4 adet çıkışı ARM9 işlemciden gelen komut ile aktif ya da pasif yapmak (Bu komut kullanıcı tarafından araca müdahale şeklinde internet üzerinden veya SMS ile verilebilir).
- Kilitlenmesi durumunda ARM9 işlemciyi yeniden başlatmak.
- RTC (Real Time Clock) ile gerçek zamanı tutmak.
- $I^2C$ protokolünü kullanarak ARM9 işlemci ile haberleşmek.

Şekil 4.3'de ARM7 işlemciye ve bazı çevresel birimlerine ait devre şeması gösterilmiştir.



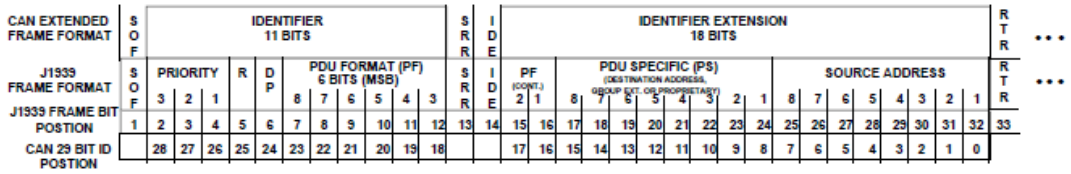

**Şekil 4.3 - ARM7 Devre Şeması [16]**

ARM7 işlemci 4 MHz'lik harici osilatör ile çalışmaktadır. Bu osilatör frekansı mikro denetleyici içindeki dâhili PLL (Phase Locked Loop) ile 64 MHz seviyesine yükseltilerek ana sistem saat darbesi elde edilmiş olur. Bu sistem frekansından sistemin farklı birimleri (çekirdek, çevresel birimler, RTC, vb.) için frekans bölücüler vasıtasıyla farklı frekanslar elde edilir [12].

STR752 mikro denetleyicisinin CAN hattına bağlantısını sağlamak için 2.0B protokolünü destekleyen CAN alıcı/vericisi olan MCP2551 tercih edilmiştir. MCP2551'nin mikro denetleyici ve CAN Bus arasındaki bağlantısı Şekil 4.3'de gösterilmiştir. Ayrıca bu sürücü SPI TM seri ara yüzü içerir, bu yüzden herhangi bir mikro denetleyicinin CAN Bus'a bağlantısı için kullanılabilir [17]. CAN hattına bağlanan fiziksel yol 120 Ohm sonlandırma direnci ile sonlandırılmıştır.

Araç üzerindeki CAN mesajları SAE J1939 standardına göre yapılandırılmıştır. SAE (Society of Automotive Engineers) J1939, motorlu araç

bileşenleri arasındaki haberleşme ve sistem kontrolü için kullanılan bir standarttır. Araç üzerine fiziksel olarak dağıtılmış elektronik kontrol ünitelerinin gerçek zamanlı kapalı çevrim kontrol fonksiyonlarını icra etmek üzere tasarlanmıştır. J1939 standardına göre 29 bit identifier Şekil 4.4'te gösterilmiştir [18].




**Şekil 4.4 – SAE J1939 29 Bit Identifier [18]**

Aracın haberleşme sistemi olan CAN Bus üzerinde araca ait bütün bilgiler J1939 standardında bulunmaktadır [19]. Araç üzerindeki CAN Bus hattından aşağıdaki mesajlar alınarak değerlendirilip anlamlı bir şekilde sunucuya gönderebilmesi için ARM9 işlemciye iletilmektedir.

1. Araç Hızı
2. Yakıt Seviyesi
3. Motor Devri
4. Motor Harareti (Sıcaklığı)
5. Dış Ortam Hava Sıcaklığı

## 4.4 CAN Bit Zamanlaması

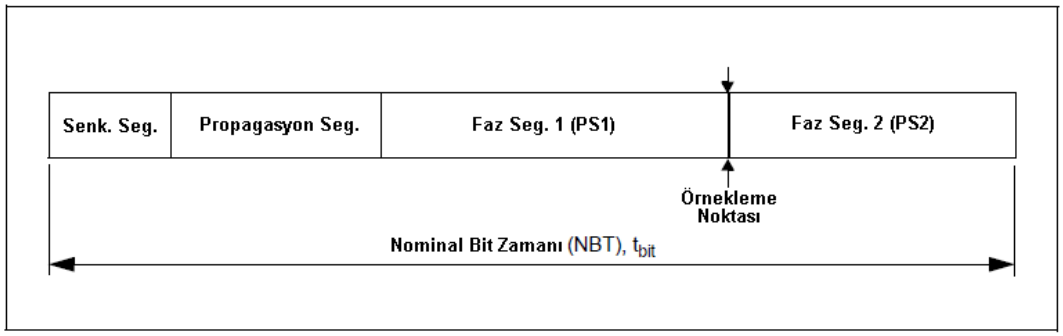
CAN Bus performansı ve birimlerin hatalı duruma düşmelerini engellemek için bit zamanlaması yapılmalıdır. Burada hesaplama için gerekli temel kavramlar verilmiş ve bizim uygulamamız için hesaplama yapılarak mikro denetleyicinin CAN zamanlama yazmacına yazılacak değer belirlenmiştir.

### 4.4.1 CAN Bit Zamanının Yapılandırması

CAN bit zamanı üst üste binmeyen bölümlerden oluşmuştur. Bu bölümlerden her biri Zaman Parçası (Time Quanta – $t_q$) adı verilen tam sayı birimlerden oluşur.

Nominal Bit Hızı (NBR) hiçbir senkronizasyona ihtiyaç duyulmadan ideal bir verici tarafından saniyede gönderilen bit sayısıdır ve Eşitlik 4.1 ile tanımlanır:

$$NBR = f_{bit} = \frac{1}{t_{bit}} \qquad (4.1)$$




**Şekil 4.5 - CAN Bit Zamanı Bölümleri [20]**

### 4.4.1.1 Nominal Bit Zamanı

Nominal Bit Zamanı (NBT) veya $t_{bit}$, Şekil 4.5'de gösterilen ve üst üste binmeyen parçalardan oluşur. Sonuç olarak NBT Eşitlik 4.2'deki bölümlerin toplamından oluşur.

$$t_{bit} = t_{SnkSeg} + t_{Prop} + t_{PS1} + t_{PS2} \qquad (4.2)$$

### 4.4.1.2 Senkronizasyon Segmenti

Nominal Bit Zamanı içindeki ilk bölümdür ve hatta bağlı birimleri senkronize etmek için kullanılır. Bite ait yükselen ve düşen kenarların bu bölüm içinde oluşması beklenir. 1 $t_q$ genişliğinde sabittir.

### 4.4.1.3 Propagasyon Segmenti

Hatta bağlı birimler arasındaki fiziksel gecikmeleri telafi içindir. Propagasyon gecikmesi fiziksel hat üzerindeki sinyal yayılım zamanlarının toplamının iki katı

olarak tanımlanmıştır ve CAN sürücüsünden kaynaklanan gecikmeyi de içerir. Bu bölüm 1 – 8 $t_q$ arasında programlanabilir.

#### 4.4.1.4 Faz Segmenti 1 ve Faz Segmenti 2

Bu iki bölüm kenar faz hatalarını telafi etmek için bulunmaktadır. Yeniden eşleme (resyncronization) ile PS1 uzatılabilir ve PS2 kısaltılabilir. PS1 1- 8 $t_q$ arası uzunlukta, PS2 ise 2- 8 $t_q$ arası uzunlukta programlanabilir.

#### 4.4.1.4 Örnekleme Noktası

Lojik seviyesinin okunup değerlendirildiği noktadır. Örnekleme noktası Faz Segmenti 1'den sonra yerleştirilmiştir.

#### 4.4.1.5 Bilgi İşleme Zamanı (Information Processing Time – IPT)

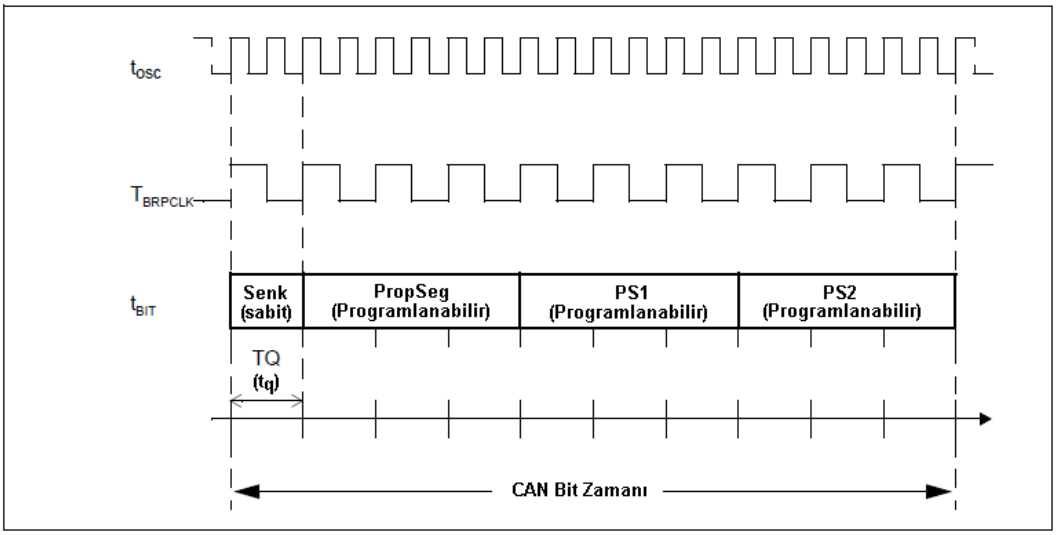
Örneklenen bitin seviyesini (lojik 1/0) belirlemek için ihtiyaç duyulan süredir.

#### 4.4.1.6 Senkronizasyon Atlama Genişliği (SJW)

Gönderilen mesajla senkonizasyonun sağlanabilmesi için bit saat darbesini 1 – 4 $t_q$ arasında (yapılandırılan şekilde) ayarlar.

#### 4.4.1.7 Zaman Parçası (Time Quanta – $t_q$)

Bit zamanını oluşturan her bir bölüm tam sayı katı olarak ifade edilen zaman bölümlerinden (Time Quanta – $t_q$) oluşur. Her bir zaman parçasının uzunluğu osilatör periyoduna ($t_{OSC}$) bağlıdır. Ana zaman parçacığı yapılandırılmaya bağlı olarak osilatör periyodunun katlarına eşittir. Şekil 4.6 bit periyodunun osilatör periyodundan elde edilmesini göstermektedir.




**Şekil 4.6 - $t_q$ ve Bit Periyodu [21]**

$t_q$ uzunluğu bir $t_q$ saat darbesi uzunluğuna ($t_{BRPCLK}$) eşittir ve programlanabilir bir ön bölümleyici olan Baud Rate Prescaler (BRP) kullanılarak programlanabilir. Bu Eşitlik 4.3'de gösterilmiştir:

$$t_q = 2 \cdot BRP \cdot T_{OSC} = \frac{2 \cdot BRP}{f_{OSC}} \qquad (4.3)$$

## 4.4.2 Bit Zamanının Senkronize Edilmesi

CAN Bus üzerindeki bütün birimler aynı nominal bit hızına sahip olmalıdır. Gürültü, faz kaymaları ve osilatör dalgalanmaları, nominal bit hızının sistemdeki gerçek bit hızına eşit olmaması durumlarını yaratır. Bu nedenle, hattaki mesajlarla senkronize olabilen bir metoda ihtiyaç vardır.

CAN spesifikasyonu olabilecek en kötü osilatör toleransının % 1.58 olduğunu ve bu toleransın da yavaş iletimler (125 kb/s ve altı) için uygun olduğunu belirtir.

CAN protokolü, mesaj önceliği için kullanılan zararsız karar vermek mekanizmasını gerçekleştirmek için çekinik (lojik 1) ve baskın (lojik 0) durumlar

tanımlamıştır. Bu mekanizma en fazla yayılma gecikmelerinden etkilenmektedir. Her bir birim öncelik mekanizmasını işletebilmek için aynı bit zamanı için her bir bit seviyesini örneklemek zorundadır. Örnek olarak farklı uçlardaki iki birim aynı anda mesajlarını göndermeye başladığında öncelik kontrol mekanizmasını işletmelidir. Öncelik mekanizmasının işletilmesi de her iki birimin de aynı bit zamanı süresince örnekleme yapabilmesi durumunda etkilidir. Bu gerçek, belirlenen veri iletim hızları için CAN iletim hattının sınırlı olduğunu belirtir. Bir CAN sisteminin propagasyon gecikmesi, fiziksel hat üzerinde sinyalin iletilmesi ($t_{bus}$), çıkış sürücüsü gecikmesi ($t_{drv}$) ve giriş karşılaştırıcısının ($t_{cmp}$) toplamları olarak Eşitlik 4.4'deki gibi hesaplanır:

$$t_{\mathrm{Pr}op} = 2 \cdot (t_{bus} + t_{cmp} + t_{drv}) \qquad (4.4)$$

#### 4.4.2.1 Senkronizasyon

Mesajların doğru olarak değerlendirilebilmesi için alıcılar gönderilen mesaja kendilerini senkronize etmek zorundadırlar. Senkronizasyonu gerçekleştirmek için iki metot vardır.

- **Hard Senkronizasyon:** Bu senkronizasyon türü sadece hat seviyesinin çekinikten baskına değişimi sırasında olur (mesaj başlangıcında). Bit sayıcısının Senkronizasyon Segmenti ile yeniden başlatılmasını sağlayarak senkronizasyonu sağlar. Bu noktada bütün alıcılar gönderici ile senkron duruma gelir. Hard senkronizasyon bir mesaj boyunca sadece bir kez yapılır.

- **Yeniden Senkronizasyon:** Bu senkronizasyonun amacı hard senkronizasyon ile yapılan düzenin korunmasıdır. Bu senkronizasyon olmadan alıcı birimler osilatörlerindeki dalgalanmalardan dolayı senkronizasyon dışı kalabilirler.

### 4.4.3 Bit Zamanlama Parametrelerinin Hesaplanması

Zamanlama yapılandırılması için önce istenen hat hızı belirlenmelidir. Buradan da minimum bit zamanı ortaya çıkar. Bit zamanı 4-25 arası zaman parçacığı

($t_q$) içerebilir. $t_q$'nun uzunluğu bit hızı ön bölümleyicisi (BRP) tarafından $t_q$ = (BRP) $f_{sys}$ eşitliği ile tanımlanır. Burada $f_{sys}$ sistem saat darbesi frekansıdır.

Bit zamanı, uzunlukları $t_q$'nun katı olan segmentlerden oluşur. Bunlardan ilki PropSeg uzunluğudur. Bu uzunluk hat uzunluğuyla doğru orantılı olarak değişir. Sonuçta elde edilen PropSeg, zaman parçacığı ($t_q$) cinsinden bulunur. SenkSeg sabit olup 1 $t_q$ uzunluğundadır. PropSeg ve SenkSeg'den arta kalan süre (Bit Süresi – PropSeg – 1) faz segmentleri tarafından kullanılır. Eğer arta kalan bu süre çift sayıda $t_q$ içeriyorsa FazSeg1 = FazSeg2 olur, tek sayıda $t_q$ içeriyorsa FazSeg2 = FazSeg1 +1 eşitliği sağlanır. FazSeg2 uzunluğunun [0..2] $t_q$ olan bilgi işleme zamanından (IPT) daha kısa olamayacağı göz önünde bulundurulmalıdır. SJW (Syncronization Jump Width) uzunluğu maksimum değerine ayarlanır. Bu da 4 veya FazSeg1'den kısa olanına eşittir.

Yapılandırmada osilatör toleransı, propagasyon gecikmesi ve benzeri durumlar hesaba katılmalıdır. Eğer birden fazla yapılandırma mümkünse en yüksek osilatör toleransına müsaade eden yapılandırma seçilmelidir. Çünkü CAN sisteminin osilatör toleransı sistemdeki en düşük toleransa sahip biriminki ile sınırlıdır. CAN Bus üzerindeki propagasyon zamanı hesaplaması, en fazla gecikmeye sahip birimler hesaba katılarak yapılmalıdır. Aynı bit hızını elde etmek için farklı sistem saat darbelerine sahip birimler farklı yapılandırmalar gerektirir.

CAN bit zamanlamasının protokole uygun olarak yapılandırması için, hesaplama sonucu hat uzunluğunun veya iletim hızının düşürülmesini, osilatör frekansının artırılmasını işaret edebilir.

Eşitlik 4.5 ile oluşturulan yapılandırma, Bit Zamanlama Yazmacı'na (Bit Timing Register) yazılır:

$$(FazSeg2 - 1)\ \&\ (FazSeg1 + \Pr opSeg - 1)\ \&\ (SJW - 1)\ \&\ (\ddot{O}nB\ddot{o}l\ddot{u}mleyici - 1) \quad (4.5)$$

Bizim uygulamamızda çevresel birimler için kullanılan saat darbesi frekansı (APB_CLK) 8 MHz, bit hızı ön bölücüsü (BRP) 0 ve iletim hızı 250 kbit/s olarak seçilmiştir. Bu değerlere göre bit zamanı hesaplandığında [13]:

| | | | |
|---|---|---|---|
| $t_q$ | 125 | ns | $= t_{APB\_CLK}$ |
| CAN Bus sürücü gecikmesi | 50 | ns | |
| Alıcı devre gecikmesi | 30 | ns | |
| Hat uzunluğu gecikmesi (40m) | 220 | ns | |
| $t_{Prop}$ | 750 | ns | $= 6 \cdot t_q$ |
| $t_{SJW}$ | 125 | ns | $= 1 \cdot t_q$ |
| $t_{TSeg1}$ | 875 | ns | $= t_{Prop} + t_{SJW}$ |
| $t_{TSeg2}$ $1 \cdot t_q$ | 250 | ns | = Bilgi işleme zamanı (IPT) + |
| $t_{SenkSeg}$ | 125 | ns | $= 1 \cdot t_q$ |
| Bit zamanı ($t_{bit}$) | 1250 | ns | $= t_{SenkSeg} + t_{TSeg1} + t_{TSeg2}$ |
| APB_CLK toleransı | 0.78 | % | $= \frac{\min(PB1, PB2)}{2 \times (13 \times bit\_zamani - PB2)}$ $= \frac{0.25\mu s}{2 \times (13 \times 1.25\mu s - 0.25\mu s)}$ |

Bu hesaplamalar sonucunda Bit Zamanlama Yazmacı'na programlanacak değer:

$(FazSeg2 - 1)$ & $(FazSeg1 + \Pr opSeg - 1)$ & $(SJW - 1)$ & $(\ddot{O}nB\ddot{o}l\ddot{u}mleyici - 1)$ (Bkz. 4.5)

ile uygun olarak Eşitlik 4.6 ile:

$$(2 -1)_1 \ \& \ (16 - 1)_{15} \ \& \ (1 - 1)_0 \ \& \ (1 - 1)_0 = 0x1F00 \qquad (4.6)$$

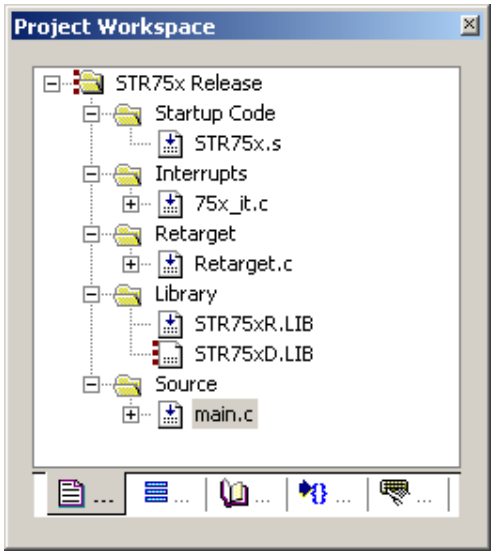
olarak bulunur.

## 4.5 Uygulama Kaynak Kodu

Uygulama geliştirilirken C Programlama Dili kullanılmıştır. Uygulama geliştirme ortamı olarak Keil µVision3 v3.53 tercih edilmiştir.

Projeye ait kaynak kodu dosyalarını ve kullanılan kütüphane dosyalarını gösteren proje çalışma sayfası Şekil 4.7'de görülmektedir.

Ana program main.c dosyası içinde yer almaktadır. Oluşan kesmeler (CAN, I2C, RTC, harici seviye değişimi, ADC çevrim tamamlanması) sonucunda kesme vektörlerinin dallanıp icra ettiği fonksiyonlar ise 75x_it.c dosyasında bulunmaktadır.




**Şekil 4.7 - Proje Çalışma Sayfası**

### 4.5.1 Main.c

Main.c dosyası içinde yer alan kaynak kodları aşağıda verilmiştir. Bu fonksiyonda öncelikle farklı sistemlerin saat darbelerini ve çevresel birimleri yapılandıran alt fonksiyonlar çağrılmaktadır. Daha sonra watchdog zamanlayıcısı kurularak sonsuz bir döngü içerisine girilmektedir. Bu döngü içerisinde watchdog zamanlayıcısı sürekli olarak sıfırlanarak taşıp sistemi yeninden başlatılması önlenmektedir. CAN Bus üzerinden alınan mesajlar da burada değerlendirilip uygun değişkenlere yazılmaktadır. Harici kesme ile okunan frekans değerleri değişkenlere atanmakta ve temizlenerek bir sonraki saniyede okunmaya hazır hale gelmektedir. Ayrıca bizim sistemimize bağlı olan ARM9 işlemcinin kilitlenmesi durumunda yeniden başlatılması da main fonksiyonu içinde yapılmaktadır.

#### 4.5.1.1 Main

```
void main()
{
      /* Sistem clock yapilandirilmasi */
      MRCC_Configuration();

      /* Giris cikis portlari yapilandirilmasi */
      GPIO_Configuration();

      /* CAN yapilandirilmasi */
      CAN_Configuration();

      /* I2C yapilandirilmasi */
      I2C_Configuration();

      /* RTC yapilandirilmasi */
      RTC_Configuration();

      /* ADC yapilandirilmasi */
      ADC_Configuration();

      /* UART yapilandirilmasi */
      UART_Configuration();

      /* EXTIT yapilandirilmasi */
      EXTIT_Configuration();

      /* Harici kesme yapilandirilmasi */
      EIC_Configuration();

      /* Watchdog zamanlayicisini 4.2s icin kur */
      WDG_InitStructure.WDG_Mode = WDG_Mode_WDG;
      WDG_InitStructure.WDG_Preload = 0xFFFF;
      WDG_InitStructure.WDG_Prescaler = 0xFF;
      WDG_Init(&WDG_InitStructure);
      WDG_Cmd(ENABLE);

  while(1)
  {
      /* Watchdog zamanlayicisini yeniden baslat */
      WDG->KR = 0xA55A;
      WDG->KR = 0x5AA5;

      /* CAN Bus'dan okunan mesajlari degerlendir */
      if (CAN_Data_Received) {

        switch (RxCanMsg.Id)
        {
          case 0x0CFEF17A: /* Arac hizi */
                  SpeedL = RxCanMsg.Data[1];
                  SpeedH = RxCanMsg.Data[2];
                  SpeedH <<= 8;
                  Speed =  (SpeedH | SpeedL) / 256;
                  break;
          case 0x1CFEFCB1: /* Yakit seviyesi */
                  FuelLevel = RxCanMsg.Data[1] * 4 / 10;
                  break;
          case 0x0CF004EE: /* Motor devri */
                  RpmL = RxCanMsg.Data[3];
```

```
                RpmH = RxCanMsg.Data[4];
                Rpm <<= 8;
                Rpm =  (RpmH | RpmL) / 8;
                break;
          case 0x0CFEEEEE: /* Motor sicakligi */
                EngTemp =   RxCanMsg.Data[0] - 40;
                break;
          case 0x1CFEF5AC: /* Ortam hava sicakligi */
                AmbTempL = RxCanMsg.Data[3];
                AmbTempRpmH = RxCanMsg.Data[4];
                AmbTemp <<= 8;
                AmbTemp =  (AmbTempH | AmbTempL) - 273;
                break;
            CAN_Data_Received = TRUE;
              break;

            default:
              break;

            CAN_Data_Received = FALSE;
            }
      }

    if(TimeDisplay == TRUE)   /* Saniyede bir kez icra edilecek */
    {
              f1 = freq1_value;
              f2 = freq2_value;
              f3 = freq3_value;
              f4 = freq4_value;

              freq1_value = freq2_value = freq3_value = freq4_value=
            0;

            if ((GPIO_ReadBit(GPIO1, GPIO_Pin_3) == Bit_RESET)&&
               (ResetTimer >= 60)) {
                  GPIO_WriteBit(GPIO1, GPIO_Pin_3, Bit_SET);
                  printf("Yeniden baslatilma sona erdi!\n\r");
            }

            ResetTimer++;

            if (ResetTimer >= 900) {
                  GPIO_WriteBit(GPIO1, GPIO_Pin_3, Bit_RESET);
                  printf("ARM9 yeniden baslatildi!\n\r");
                  ResetTimer = 0;
            }

      TimeDisplay = FALSE;
    }
  }
}
```

#### 4.5.1.2 MRCC_Configuration

Farklı sistemlerin saat darbeleri bu fonksiyon içinde yapılandırılmaktadır.

```
void MRCC_Configuration(void)
{
```

```
  /* 4MHz'lik osilatorun baslamasini bekle */
  OSC4MStartUpStatus = MRCC_WaitForOSC4MStartUp();

  if(OSC4MStartUpStatus == SUCCESS)
  {
    /* HCLK 32MHz */
    MRCC_HCLKConfig(MRCC_CKSYS_Div2);

    /* CKTIM 16MHz */
    MRCC_CKTIMConfig(MRCC_HCLK_Div2);

    /* PCLK 8MHz */
    MRCC_PCLKConfig(MRCC_CKTIM_Div2);

    /* CKSYS 64MHz */
    MRCC_CKSYSConfig(MRCC_CKSYS_OSC4MPLL, MRCC_PLL_Mul_16);
  }

  /* Giris-cikis pinlerini 3.3V icin yapilandir */
  MRCC_IOVoltageRangeConfig(MRCC_IOVoltageRange_3V3);

  /* Kullanilan cevresel birimlerin saat darbelerini aktif et */
  MRCC_PeripheralClockConfig(MRCC_Peripheral_I2C |
MRCC_Peripheral_ADC | MRCC_Peripheral_GPIO | MRCC_Peripheral_UART0 |
MRCC_Peripheral_EXTIT | MRCC_Peripheral_CAN, ENABLE);

  /* 4MHz'lik osilatoru 128'e bolerek RTC saat darbesini olustur */
  MRCC_CKRTCConfig(MRCC_CKRTC_OSC4M_Div128);

  /* RTC'nin hazir olmasini bekle */
  while(MRCC_GetFlagStatus(MRCC_FLAG_CKRTCOK) == RESET);
}
```

#### 4.5.1.3 GPIO_Configuration

Giriş çıkış pinleri kullanımlarına uygun olarak bu fonksiyon içinde yapılandırılır.

```
void GPIO_Configuration(void)
{
  GPIO_InitTypeDef  GPIO_InitStructure;

  /* CAN TX ve CAN RX pinlerini yapilandir */
  GPIO_StructInit(&GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_WriteBit(GPIO0, GPIO_Pin_15, Bit_SET);
  GPIO_Init(GPIO0, &GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIO0, &GPIO_InitStructure);

  /* I2C SCL ve SDA pinlerini yapilandir   */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 ;
  GPIO_Init(GPIO0, &GPIO_InitStructure);
```

```
  /* UART0_Rx pinini yapilandir */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_10;
  GPIO_Init(GPIO0, &GPIO_InitStructure);

  /* UART0_Tx pinini yapilandir */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_11;
  GPIO_Init(GPIO0, &GPIO_InitStructure);

  /* P0.04, P0.05, P0.06, P0.07 pinlerini Open-Drain cikis yap */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6
|
  GPIO_Pin_7;
  GPIO_Init(GPIO0, &GPIO_InitStructure);

  /* ADC kanal 9, 10 ve 11 pinlerini yapilandir */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_6 | GPIO_Pin_8
;
  GPIO_Init(GPIO1, &GPIO_InitStructure);

  /* ADC kanal 8 pinini yapilandir */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_29;
  GPIO_Init(GPIO0, &GPIO_InitStructure);

  /* Frekans girisi pinlerini yapilandir */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_7 | GPIO_Pin_9
| GPIO_Pin_10 | GPIO_Pin_12;
  GPIO_Init(GPIO1, &GPIO_InitStructure);

  /* ARM9'u resetleyen cikis pinini yapilandir*/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
  GPIO_Init(GPIO1, &GPIO_InitStructure);
}
```

#### 4.5.1.4 CAN_Configuration

CAN çevresel biriminin yapılandırıldığı fonksiyondur.

```
void CAN_Configuration(void)
{
  /* CAN Buz hizini ayarla, kesmeleri etkinlestir */
  CAN_Struct.CAN_ConfigParameters=CAN_CR_IE;
  CAN->BTR = 0x1F00;
  CAN_Struct.CAN_Bitrate=CAN_BITRATE_250K;
  CAN_Init(&CAN_Struct);

  /* Mesaj nesnelerini yapilandir */
  CAN_InvalidateAllMsgObj();
  CAN_SetTxMsgObj(CAN_TX_MSGOBJ, CAN_EXT_ID);
  CAN_SetRxMsgObj(CAN_RX_MSGOBJ, CAN_EXT_ID, 1, CAN_LAST_EXT_ID,
TRUE);
}
```

#### 4.5.1.5 EIC_Configuration

Çevresel birimlere ait kesmelerin yapılandırıldığı ve önceliklerinin atandığı fonksiyondur.

```
void EIC_Configuration(void)
{
  EIC_IRQInitTypeDef EIC_IRQInitStructure;

  /* CAN kesmesini yapilandir ve onceligini ata */
  EIC_IRQInitStructure.EIC_IRQChannel=CAN_IRQChannel;
  EIC_IRQInitStructure.EIC_IRQChannelPriority=1;

  /* I2C kesmesini yapilandir ve onceligini ata */
  EIC_IRQInitStructure.EIC_IRQChannel = I2C_IRQChannel;
  EIC_IRQInitStructure.EIC_IRQChannelPriority = 2;

  /* Harici kesmeyi yapilandir ve onceligini ata */
  EIC_IRQInitStructure.EIC_IRQChannel = EXTIT_IRQChannel;
  EIC_IRQInitStructure.EIC_IRQChannelPriority = 3;

  /* RTC kesmesini yapilandir ve onceligini ata */
  EIC_IRQInitStructure.EIC_IRQChannel = RTC_IRQChannel;
  EIC_IRQInitStructure.EIC_IRQChannelPriority = 4;

  /* ADC kesmesini yapilandir ve onceligini ata */
  EIC_IRQInitStructure.EIC_IRQChannel = ADC_IRQChannel;
  EIC_IRQInitStructure.EIC_IRQChannelPriority = 5;

  EIC_IRQInitStructure.EIC_IRQChannelCmd = ENABLE;
  EIC_IRQInit(&EIC_IRQInitStructure);
  EIC_IRQCmd(ENABLE);
}
```

#### 4.5.1.6 RTC_Configuration

RTC (gerçek zamanlı saat) ve saniye değişimlerinde kesme üretilmesi bu fonksiyon içinde yapılandırılır.

```
void RTC_Configuration(void)
{
      /* RTC'yi etkinlestir */
      MRCC_PeripheralClockConfig(MRCC_Peripheral_RTC, ENABLE);

      /* RTC yazmaclarinin senkronize olmasini bekle */
      RTC_WaitForSynchro();

      /* RTC yazmaclarina yazma isleminin bitmesini bekle */
      RTC_WaitForLastTask();

      /* RTC Saniye kesmesini etkinlestir */
      RTC_ITConfig(RTC_IT_Second, ENABLE);
```

```
      /* RTC yazmaclarina yazma isleminin bitmesini bekle */
      RTC_WaitForLastTask();

      /* RTC periyodunu 1 saniyeye ayarla */
      RTC_SetPrescaler(31249);

      /* Saati ayarla */
      THH = 00;

      /* Dakikayi ayarla */
      TMM = 00;

      /* Saniyeyi ayarla */
      TSS = 00;

      /* Sayici degerini hesapla */
      Tmp = THH * 3600 + TMM * 60 + TSS;

      /* RTC yazmaclarina yazma isleminin bitmesini bekle */
      RTC_WaitForLastTask();

      /* Sayici degerini ayarla */
      RTC_SetCounter(Tmp);
}
```

#### 4.5.1.7 ADC_Configuration

Analog dijital çevrimin nasıl yapılacağı ve hangi durumda kesme üreteceği bu fonksiyon içinde yapılandırılır.

```
void ADC_Configuration(void)
{
  ADC_InitTypeDef  ADC_InitStructure;
  ADC_InitStructure.ADC_ConversionMode = ADC_ConversionMode_Scan;
  ADC_InitStructure.ADC_ExtTrigger = ADC_ExtTrigger_Disable;
  ADC_InitStructure.ADC_AutoClockOff = ADC_AutoClockOff_Disable;
  ADC_InitStructure.ADC_SamplingPrescaler = 5;
  ADC_InitStructure.ADC_ConversionPrescaler = 1;
  ADC_InitStructure.ADC_FirstChannel = ADC_CHANNEL8;
  ADC_InitStructure.ADC_ChannelNumber = 4;
  ADC_Init(&ADC_InitStructure);

  /* Secilen kanallarin tamami cevrildikten sonra kesme
olusturulmasini
     etkin kil */
  ADC_ITConfig(ADC_IT_ECH, ENABLE);

  /* ADC'yi etkinlestir */
  ADC_Cmd(ENABLE);

  /* Kalibrasyonu baslat */
  ADC_StartCalibration(ADC_CalibAverage_Enable);

  /* Cevrim islemine basla */
  ADC_ConversionCmd(ADC_Conversion_Start);
}
```

#### 4.5.1.8 UART_Configuration

Bu fonksiyonda UART belirtilen ayarlar ile yapılandırılmıştır.

```
void UART_Configuration(void)
{
  UART_InitTypeDef UART_InitStructure;

/* UART0 yapilandirilmasi */
  /* UART0 su sekilde yapilandirilmistir:
        - Veri bitleri = 8 bit
        - Dur bitleri = 1 bit
        - Eslik = tek
        - Saniyedeki bit sayisi = 115200
        - Akis denetimi = yok
        - Gonderme ve alma etkin
  */
  UART_InitStructure.UART_WordLength = UART_WordLength_8D;
  UART_InitStructure.UART_StopBits = UART_StopBits_1;
  UART_InitStructure.UART_Parity = UART_Parity_Odd ;
  UART_InitStructure.UART_BaudRate = 115200;
  UART_InitStructure.UART_HardwareFlowControl=
  UART_HardwareFlowControl_None;
  UART_InitStructure.UART_Mode = UART_Mode_Tx_Rx;
  UART_InitStructure.UART_FIFO = UART_FIFO_Disable;

  UART_Init(UART0, &UART_InitStructure);

  /* UART0 etkinlestir */
  UART_Cmd(UART0, ENABLE);
}
```

#### 4.5.1.9 fputc

Bu fonksiyon ile C kütüphanesindeki printf fonksiyonu UART0'a bağlanmaktadır. Yani printf fonksiyonu ile yazdırılan karakterler UART0 üzerine gönderilmektedir. Bizim uygulamamızda bu hata ayıklama mesajları bastırmak için kullanılmıştır.

```
int fputc(int ch, FILE *f)
{
    UART_SendData(UART0, (u8) ch);
    while(UART_GetFlagStatus(UART0, UART_FLAG_TxFIFOFull) != RESET);

    return ch;
}
```

#### 4.5.1.10 EXTIT_Configuration

Bu fonkiyonda harici kesme kanalları kare dalganın düşen kenarında kesme üretmek üzere yapılandırılmaktadır.

```
void EXTIT_Configuration(void)
{
      EXTIT_InitTypeDef EXTIT_InitStructure;

      /* 7,8,9,10 harici kesme kanallarindaki bekleyen biti temizle
*/
      EXTIT_ClearITPendingBit(EXTIT_ITLine7 | EXTIT_ITLine8 |
      EXTIT_ITLine9 | EXTIT_ITLine10 | EXTIT_ITLine12);

      /* 7,8,9,10 harici kesme kanallarinin dusen kenarlarinda kesme
uretecek sekilde yapilandir */
      EXTIT_InitStructure.EXTIT_ITLine = EXTIT_ITLine7 |
EXTIT_ITLine8|
      EXTIT_ITLine9 | EXTIT_ITLine10 | EXTIT_ITLine12;
      EXTIT_InitStructure.EXTIT_ITTrigger = EXTIT_ITTrigger_Falling;
      EXTIT_InitStructure.EXTIT_ITLineCmd = ENABLE;
      EXTIT_Init(&EXTIT_InitStructure);
}
```

#### 4.5.1.11 I2C_ReceivedDataEvaluate

Bu fonksiyon I2C kesmesi oluştuğu anda 75x_it.c dosyası içindeki I2C_IRQHandler fonksiyonu tarafından çağrılmaktadır. ARM9 işlemciden gelen veri isteği değerlendirilip uygun karşılık bir tampon hafızaya yazılmaktadır. ARM9 işlemcinin bu bilginin kendisine gönderilmesini istemesi durumunda ise tampon hafızadaki mesaj yine 75x_it.c dosyası içindeki I2C_IRQHandler fonksiyon tarafından gönderilmektedir.

```
void I2C_ReceivedDataEvaluate(void)
{
      if ((Rx_Buffer[0]) != ('#')) {
            sprintf(Tx_Buffer,"#HATALI MESAJ ALINDI!");
      } else if ((Rx_Buffer[1]) == ('C')) {    /* CAN Mesajlari*/
            if ((Rx_Buffer[2]) == ('1')) {
                  sprintf(Tx_Buffer,"#C1%04X!",Speed);
            } else if ((Rx_Buffer[2]) == ('2')) {
                  sprintf(Tx_Buffer,"#C2%04X!",FuelLevel);
            } else if ((Rx_Buffer[2]) == ('3')) {
                  sprintf(Tx_Buffer,"#C3%04X!",Rpm);
            } else if ((Rx_Buffer[2]) == ('4')) {
                  sprintf(Tx_Buffer,"#C4%04X!",EngTemp);
            } else if ((Rx_Buffer[2]) == ('5')) {
                  sprintf(Tx_Buffer,"#C5%04X!",AmbTemp);
            }
      } else if ((Rx_Buffer[1]) == ('A')) {    /* Analog degerler*/
            if ((Rx_Buffer[2]) == ('1')) {
                  sprintf(Tx_Buffer,"#A1%02X!",adc1_value);
```

```
            } else if ((Rx_Buffer[2]) == ('2')) {
                  sprintf(Tx_Buffer,"#A2%02X!",adc2_value);
            } else if ((Rx_Buffer[2]) == ('3')) {
                  sprintf(Tx_Buffer,"#A3%02X!",adc3_value);
            } else if ((Rx_Buffer[2]) == ('4')) {
                  sprintf(Tx_Buffer,"#A4%02X!",adc4_value);
            }
      } else if ((Rx_Buffer[1]) == ('F')) { /* Frekans degerleri */
            if ((Rx_Buffer[2]) == ('1')) {
                  sprintf(Tx_Buffer,"#F1%02X!",freq1_value);
            } else if ((Rx_Buffer[2]) == ('2')) {
                  sprintf(Tx_Buffer,"#F2%02X!",freq2_value);
            } else if ((Rx_Buffer[2]) == ('3')) {
                  sprintf(Tx_Buffer,"#F3%02X!",freq3_value);
            } else if ((Rx_Buffer[2]) == ('4')) {
                  sprintf(Tx_Buffer,"#F4%02X!",freq4_value);
            }
      }

      printf("%s\n\r",Rx_Buffer);
      Rx_Buffer[Rx_Idx - 1] = 0;
}
```

### 4.5.2 75x_it.c

75x_it.c dosyası kesme (interrupt) üretilmesi sonucunda icra edilen fonksiyonları içermektedir. main.c dosyası içindeki EIC_Configuration fonksiyonunda kesme üretmek için yapılandırılan çevresel birimlerin (CAN, $I^2C$, harici kenar değişimi kesmesi, RTC, ADC) ürettiği kesmeler bu dosya içindedir.

#### 4.5.2.1 CAN_IRQHandler

CAN Bus üzerinden mesaj alındığında bir kesme üretilmektedir. CAN kesme vektörü bu kesme fonksiyonuna dallanmaktadır. Burada kesme oluşmasına sebep olan durum belirlenmekte ve bir değişken true yapılmaktadır. main.c dosyası içindeki ana döngü içindeki kod tarafından ise alınan mesaj işlenerek anlamlı hale getirilmektedir.

```
void CAN_IRQHandler(void)
{
  u32 msgobj = 0;

  if(CAN->IDR == 0x8000) {    /* durum kesmesi */
    (void)CAN->SR;      /* yazmaci temizlemek icin oku*/
  } else if(CAN->IDR >= 1 && CAN->IDR <= 32) {
    /* kesme olusmasina sebep olan mesaj nesnesi numarasini al */
```

```
    switch(msgobj = CAN->IDR - 1)
    {
      case  0 /* CAN_TX_MSGOBJ - Gonderme kesmesi */:
        CAN_ReleaseTxMessage(msgobj);
            break;

      case 1 /* CAN_RX_MSGOBJ  - Alma kesmesi */:
        CAN_ReceiveMessage(msgobj, TRUE, &RxCanMsg);
            switch (RxCanMsg.Id)
            {
              case 0x0CFEF17A: /* Arac hizi */
              case 0x1CFEFCB1: /* Yakit seviyesi */
              case 0x0CF004EE: /* Motor devri */
              case 0x0CFEEEEE: /* Motor sicakligi */
              case 0x1CFEF5AC: /* Ortam hava sicakligi */
                  CAN_Data_Received = TRUE;
                  break;

              default:
                  break;
            }

            CAN_ReleaseRxMessage(msgobj);
            break;

      default:
            CAN_ReleaseMessage(msgobj);
        break;
    }
  }
}
```

#### 4.5.2.2 I2C_IRQHandler

$I^2C$ haberleşmesi gereksiz kod işletilmesini önlemek için sürekli gözleme yerine kesme kontrolünde yapılmaktadır. $I^2C$ ile alınan mesajlar main.c içindeki I2C_ReceivedDataEvaluate fonkiyonunda değerlendirilerek gerekli yanıt mesajı bir tampon hafızaya yazılmaktadır. Tampon hafızadaki bu mesaj yine kesme kontrolünde alıcıya iletilmektedir.

```
void I2C_IRQHandler(void)
{
  switch (I2C_GetLastEvent())
  {
    case I2C_EVENT_SLAVE_ADDRESS_MATCHED:
       Rx_Idx = Tx_Idx = 0;
      break;

    case I2C_EVENT_SLAVE_BYTE_RECEIVED:
      Rx_Buffer[Rx_Idx++] = I2C_ReceiveData();
        if (Rx_Buffer[Rx_Idx - 1] == '!') {
            I2C_ReceivedDataEvaluate();
        }
      break;
```

```
    case I2C_EVENT_SLAVE_BYTE_TRANSMITTED:
      I2C_SendData(Tx_Buffer[Tx_Idx++]);
      break;

    case I2C_EVENT_SLAVE_ACK_FAILURE:
      I2C_SendData(0xFF);
      break;

    case I2C_EVENT_SLAVE_STOP_DETECTED:
        Rx_Idx = Tx_Idx = 0;
      break;

    default:
      break;
  }
}
```

#### 4.5.2.3 EXTIT_IRQHandler

Harici kesme ile 4 adet 10kHz seviyesine kadar frekans girişi okunmakta ve aynı zamanda ARM9 işlemcinin gönderdiği kare dalga işaretler okunarak bu işaret kesildiğinde ARM9 işlemci resetlenmektedir. Bu işlemi yanlışsız ve en etkin şekilde yapmanın yolu bu girişlerde yükselen veya düşen bir kenar görüldüğünde kesme oluşturmaktadır. Bu uygulama düşen kenarlarda kesme üretilmesi için yapılandırılmıştır. Kesmeye sebep olan düşen kenarın hangi kanal üzerinde oluştuğu da EXTIT_GetITStatus fonksiyonu ile belirlenmektedir.

```
void EXTIT_IRQHandler(void)
{
      if(EXTIT_GetITStatus(EXTIT_ITLine7) != RESET)
      {
            freq4_value++;
      /*Harici kesme kanali 7'nin kesme bekleme bitini temizle*/
            EXTIT_ClearITPendingBit(EXTIT_ITLine7);
      } else if(EXTIT_GetITStatus(EXTIT_ITLine8) != RESET)
      {
            freq3_value++;
       /* Harici kesme kanali 8'in kesme bekleme bitini temizle */
            EXTIT_ClearITPendingBit(EXTIT_ITLine8);
      } else if(EXTIT_GetITStatus(EXTIT_ITLine9) != RESET)
      {
            freq2_value++;
      /* Harici kesme kanali 9'un kesme bekleme bitini temizle */
            EXTIT_ClearITPendingBit(EXTIT_ITLine9);
      } else if(EXTIT_GetITStatus(EXTIT_ITLine10) != RESET)
      {
            freq1_value++;
      /*Harici kesme kanali 10'un kesme bekleme bitini temizle*/
            EXTIT_ClearITPendingBit(EXTIT_ITLine10);
      }

      /*ARM9 islemci 12 nolu harici kesme kanalinin seviyesini
```

```
      degistirirse ARM9'un resetlenmesini onlemek icin sayiciyi
      sifirla */
      if(EXTIT_GetITStatus(EXTIT_ITLine12) != RESET)
      {
            ResetTimer = 0;
      /*Harici kesme kanali 12'nin kesme bekleme bitini temizle*/
            EXTIT_ClearITPendingBit(EXTIT_ITLine12);
      }
}
```

#### 4.5.2.4 RTC_IRQHandler

Saniyede bir kez RTC kesmesi oluşmakta ve bu fonksiyon icra edilerek zaman güncellenmektedir.

```
void RTC_IRQHandler(void)
{
  if(RTC_GetITStatus(RTC_IT_Second) != RESET)
  {
    /* RTC saniye kesmesi bekleme bitini temizle */
    RTC_ClearITPendingBit(RTC_IT_Second);

    /* RTC yazmaclarina yazma isleminin bitmesini bekle */
    RTC_WaitForLastTask();
  }

  /* Zamani guncelle */
  TimeDisplay = TRUE;
}
```

#### 4.5.2.5 ADC_IRQHandler

Bütün kanallardaki analog dijital çevrim tamamlandıktan sonra bir kesme oluşturularak bu fonksiyon icra edilir ve çevrim sonuçları gerekli değişkenlere atanır.

```
void ADC_IRQHandler(void)
{
      /* Analog dijital cevrim sonuclarini al */
      adc1_value = ADC_GetConversionValue(ADC_CHANNEL11);
      adc2_value = ADC_GetConversionValue(ADC_CHANNEL10);
      adc3_value = ADC_GetConversionValue(ADC_CHANNEL9);
      adc4_value = ADC_GetConversionValue(ADC_CHANNEL8);

      /* Cevrim sonunda kesme bekleme bitini temizle */
      ADC_ClearITPendingBit(ADC_IT_ECH);
}
```

# 5. SONUÇLAR

Bu çalışmada CAN Bus çevresel birimine sahip ARM7 mimarisindeki STR752FR0 mikro denetleyicisi kullanılarak bir uygulama geliştirilmiştir.

Geliştirilen uygulama Araç Takip Cihazı'nın bir parçasıdır. Araç takip cihazı, aracın internet üzerinden izlenmesi, araca ait çeşitli bilgilerin uzaktan görülebilmesi, gerekli durumlarda aracın bir sistemini çalıştırmak için komut gönderilmesi gibi amaçlarla kullanılmaktadır. Bu cihaz günümüzde araçlarda standart hale gelen CAN Bus haberleşme sistemine bağlanarak sistemden ihtiyaç duyulan bilgileri alıp değerlendirmekte ve bir merkeze kablosuz olarak göndermektedir. Uygulamanın bütünlük arz etmesi amacıyla CAN Bus'dan alınan bilgiler haricinde araç üzerindeki frekans bilgileri, analog bilgiler gibi olabilecek diğer bilgiler de alınarak değerlendirilmektedir.

CAN Bus sistemine bağlı cihazların hatasız ve yüksek performansta çalışmaları için bit zamanlamaları yapılmalıdır. Bu uygulamada bit zamanlama hesaplamaları açıklamalı olarak yapılmış ve mikro denetleyici hesaplanan bu değere göre yapılandırılmıştır.

CAN verileri gönderildikleri merkezde araç sahipleri tarafından incelenebileceği gibi araç üreticileri tarafından da incelenebilir. Bu veriler araçtaki arızalı parçaları, aracın kullanım istatistiklerini, hız, devir, hararet gibi daha birçok bilgiyi içerebilir. Araç CAN haberleşme sisteminde bu ve bunun gibi araca ait tüm bilgiler mevcuttur. Araç üreticilerinin bu bilgileri incelemeleri sonucunda arıza oluşan parçalarda geliştirme ve performans ölçümü yapılabilir.

Bu uygulamanın bir ileri safhası okunan bu verilerin incelenmesi ve örnek olarak araçta CAN Bus'a bağlı bir birimin merkezden özel bir mesaj gönderilmesi yoluyla tekrar çalışır hale gelmesi olabilir. Bu, servis maliyetlerini önemli ölçüde azaltacaktır ve araç üreticilerinin ihtiyaç duyduğu bir özelliktir.

# KAYNAKLAR DİZİNİ

[1] Syed Misbahuddin, Nizar Al-Holou, 'Efficient Data Communication Techniques for Controller Area Network (CAN) Protocol', ACS/IEEE International Conference on Computer Systems and Applications, Tunis, pp.6-11, 2003.

[2] Hans A, Hansson Thomas Nolte and Christer Norstrom, 'Integrating Reliability and Timing Analysis of CAN-based Systems', IEEE Transactions on Industrial Electronics, Vol.49, no.6, pp.1240-1250, 2002.

[3] E. Gil-Dolcet, J. M. Fuertes, 'A New Communication Protocol for Automotive Real-time Applications', The 26th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Poland, pp.147-152, 2003.

[4] Introduction to Controller Area Network (CAN), Microchip Technology Incorporated, Web Seminar, 2004

[5] Jin Hui, Zhang Hong-kun and Ge An-lin, 'The Application of CAN Bus to the Vehicle Intelligent Shift System', Journal of Highway and Transportation Research and Development, Vol.21, no.3, pp.114-116, 2004.

[6] Esd Gmbh Hannover, "Controller Area Network, A Serial Bus System- Not Just For Vehicles", www.esd-electronics.com/pdffile/CAN/English/intro-e.pdf; pp. 1-8.

[7] 12th International CAN Conference, Conference Handouts, Barcelona, 2008

[8] CAN Specification Version 2.0, Robert Bosch GmbH, Stuttgart, 1991

[9] P. Göhner, Universität Stuttgart, Institute of Industrial Automation and Software Engineering, CAN Bus Theory, Stuttgart, 2006

[10] Murphy, N., A Short Trip On The CAN Bus, Embedded Systems Design, http://www.embedded.com/columns/murphyslaw/13000304?_requestid=639577, Erişim Tarihi Mayıs 2008

[11] CAN in Automation, Physical Layer Standards, http://www.can-cia.de/index.php?id=88#c2029, Erişim Tarihi Nisan 2008

[12] STR750F ARM7TDMI-S 32-bit MCU with Flash, SMI, 3 Std 16-bit Timers, PWM Timer, Fast 10-bit ADC, I2C, UART, SSP, USB and CAN User Manual Rev. 2.0, ST Microelectronics, France, 2006

[13] STR750 ARM7TDMI-S-Based Microcontroller Family Reference Manual Rev. 3.0, ST Microelectronics, France, 2007

[14] STR75X Software Library Rev. 1.0, ST Microelectronics, France, 2006

[15] ATAK Araç Takip Sistemi Kullanım Kılavuzu Rev. 2.1, Ortem Elektronik Sanayi ve Ticaret Limited Şirketi, Gebze, 2007

[16] ATAK Araç Takip Sistemi Devre Şeması Rev. 4.53, Ortem Elektronik Sanayi ve Ticaret Limited Şirketi, Gebze, 2007

[17] Wang Xin-jie, Tang Xiao-qi and Yang Bo, 'Computer Monitor Network of Car Based on CAN Bus', Machinery and Electronics, Vol.2, pp.21-23, 2002.

[18] Recommended Practice for a Serial Control and Communications Vehicle Network Rev. 2005-01, SAE International, 2005

[19] Surface Vehicle Recommended Practice Vehicle Application Layer Rev. 2005-01, SAE J1939-71, SAE International, 2005

[20] CAN in Automation, http://www.can-cia.de/index.php?id=88, Erişim Tarihi Nisan 2008

[21] Richards, P., Understanding Microchip's CAN Module Bit Timing, Microchip Technology Incorporated, Application Note, 2001

# ÖZ GEÇMİŞ


**Adı-Soyadı** : Uğur Coşkun

**Doğum Tarihi/Yeri** : 22.01.1981 / Konya

**Eğitim**

**İlköğretim** : Lalebahçe İlköğretim Okulu, Konya, 1993

**Ortaöğretim** : Konya Lisesi, Konya, 2000

**Lisans** : Erciyes Üniversitesi, Elektronik Mühendisliği Bölümü, Kayseri, 2005

**Yüksek Lisans** : Gebze Yüksek Teknoloji Enstitüsü, Elektronik Mühendisliği Bölümü, Gebze, 2008

**Sürekli Adres** : Gülbahçe Mh. Yeniçay Sk. No: 10 Meram / Konya

**Telefon** : 0 535 5608835

**E-Posta** : ucoskun@gyte.edu.tr