Baştan Sona Gömülü Yazılım Projesi Yapımı

Kaynak	Tarih	Versiyon	İşlem
Zafer SATILMIŞ	2021-01-16	0.1	Döküman oluşturuldu

1. Amaç

Bu belgenin amacı Bare-Metal kodlama ile bir gömülü yazılım projesi oluşturmaktır. Projeyi oluştururken donanım tasarımı yapılmayacaktır. Donanım olmadan kodlama yapılacak ve bu sayede donanım bağımlı kod yazmanın önüne geçmiş olunacaktır. Proje tamamlandığında çok büyük oranla farklı platform ve işlemciler üzerinde çalışabilecek bir proje yapılmış olunacaktır.

Projenin amacı 10 maddelik yazılım gereksinimlerini kapsamak olacaktır. Fakat yazılım tasarımımız amacı bu 10 maddelik gereksinimlerin genişletilmesine karşılık verecek şekilde olacaktır. Bu nedenle genişletilebilir kodlama yapılmaya çalışılacak.

2. Yazılım Gereksinimleri

Gömülü sistem projesinde yazılım gereksinimlerini belirleyebilmek için doğal olarak ilk önce proje gereksinimleri çıkarılmış olması lazım. Proje gereksinimlerinden donanım gereksinimleri, yazılım gereksinimleri ve mekanik gereksinimler gibi gereksinimler çıkarılır. Tüm bu birimlerin gereksinim sonuçları projenin/ürünün gereksinimlerini karşılamış olmalıdır.

Burada örnek olması amacıyla sadece 10 maddelik yazılım gereksinimleri belirlendi. Normalde yazılım gereksinimlerinden yazılım tasarım belgesi oluşturulur ve belge sonrasında akış diyagramları, modülleri belirleme, kodlama-test, test-kodlama sırasında proje tamamlanır.

1. Sıcaklık Değerini 70 derecenin üzerine çıkarsa fan çalıştırılacaktır.
2. Sistemin RTC değeri uarttan gelen istek mesajına karşılık gönderilecek.
3. Sıcaklık değeri uart2 üzerinden gelen sıcaklık istek mesajına karşılık gönderilecektir.
4. Sıcaklık ve ADC değerleri canbus üzerinden 500 ms periyotta gönderilecektir.
5. 250000 bit/rate hızında Canbus üzerinden istenilen 8 led kontrol edilecektir..
6. 8 push button ile sistemde bulunan sekis led kontrol edecektir. Butonlar basılı iken ledler yanacak, çekili iken ledler sönecektir.
7. Keypad bilgileri uart üzerinden gönderilecektir.
8. Sistem durumunu bir saniyelik periyot ile Can üzerinden gönderilecektir.
9. Sitemin zaman bilgisi Canbus üzerinden 1sn periyot ile gönderilecek.
10. 10- Sistemin zaman bilgisi hem uart hem de can üzerinden güncellenebilecek.

Görüldüğü üzere çok kaba biçimde yazılım ile ne yapacağımızı yukarıda tanımladık. Sistem açılma anı, haberleşme hızları, mesaj biçimi, hata durumları, sistem modu gibi birçok maddeye değinmedik.

3. Donanım Bilgisi ve Gerekliliği

Gömülü yazılım projesinde yazılım, doğası gereği tasarlanmış bir donanım üzerinde koşar. Yazılım ile donanım aslında uyum içinde ve görev paylaşımını yapacak şekilde kurgulanır. Yazılımcının bu kurguya uyarak kod yazabilmesi için yazılıma başlamadan önce donanımın

genel yapısını bilmelidir. Hatta devre şemasını okuyup yorumlama kabiliyeti kazanan gömülü yazılımcılar, tüm sistemi donanım olmadan sadece devre şemasına bakarak kodlayabilir.

Örneğin ADC girişine bağlanan sensörün çıkış değerleri ve adc girişinde gürültü/dalgalanma önlemlerinin alınıp alınmaması yazılımcının kodlamasını etkileyecektir. Bu yüzden ADC kısmını kodlamadan önce donanım yapısının incelenmesi gerekir. Daha da etkili örnek ise dijital girişlerin okunması olabilir. Ayrık sinyal(discrete) olarak gelen bilginin okunması için kurulan yapıda yazılımcı bu veriyi sürekli sorgu yöntemi ile(poll) okuyabileceği gibi bu hat için kesme ayarlaması(interrupt) yaparak da okuyabilir. Kimi zaman bu seçenekler donanım yada sistemin yapısından dolayı yazılımcının tercihine kalmaz. Ayrık sinyal okumada yazılımcının sürekli sorgu(poll) yöntemini kullandığını düşünelim. Eğer sinyalin gelme hızı sorgu hızından fazla ise yazılımcı bu durumda kesme yöntemini kullanmak zorunda kalacaktır. Kesme yöntemini kullanabilmesi için ise donanımda sinyal hattının pull-up yada pull-down yapılıp yapılmadığı bilgisine ihtiyaç duyacaktır.

Yukarıdaki örnekler daha da çoğaltılabilir. Hatta daha da detaylı bakınca donanım okumasını bilmeden gömülü yazılım yazmanın neredeyse imkansız olacağı bilgisine de ulaşabiliriz. Fakat ters düşünce ile bakarak yani donanım bilgisi olmadan gömülü yazılım alanında kodlama yapılabileceğini düşündüğümüzde de haklı çıkabiliriz. Burada birbirine ters düşen iki düşünce olmuş olsada iki düşünce de doğrudur. Eğer proje yapısında donanım soyutlayan bir yapı kurulmuş ise projedeki her yazılımcının donanım ile içli dışlı olmasına gerek kalmaz. İşte iki zıt düşünceyi de haklı çıkartan anahtar, “Donanım Soyutlama” kavramıdır.

Donanım Soyutlama yapısının birçok faydası vardır. Yazılımcıların donanımla içli dışlı olmamasını sağladığı gibi birde projenin farklı donanımlar üzerinde koşmasını sağlayacak kabiliyeti kazandırır. Bu özelliği projemize kazandırdığımızda artık donanımın(MCU) ne olduğu çok da önemli olmayacaktır. Projenin kodlanmasında da donanım bağlı fonksiyonlar artık ortadan kaldırılmış olur. Fakat ne kadar iyi, hatta süper ötesi bir donanımı soyutlama yapılmış olunsada işin içinde yine de donanım bilgisi kullanılan yerler olacaktır. Örneğin gömülü linux sistemlerinde bile donanım bilgisi ile iş yapmak zorunda kaldığımız yerler vardır. En bilindik örneği kernel device driver ve device tree katmanlarıdır.

Özetle donanımı soyutlayan bir proje oluşturabilmek için donanım okuma ve yorumlama bilgisi gerekir. Donanım soyutlanmış bir projede ise donanımla içli dışlı olmadan çalışılabilinir.

4. Donanım Birimlerinin Çıkarılması

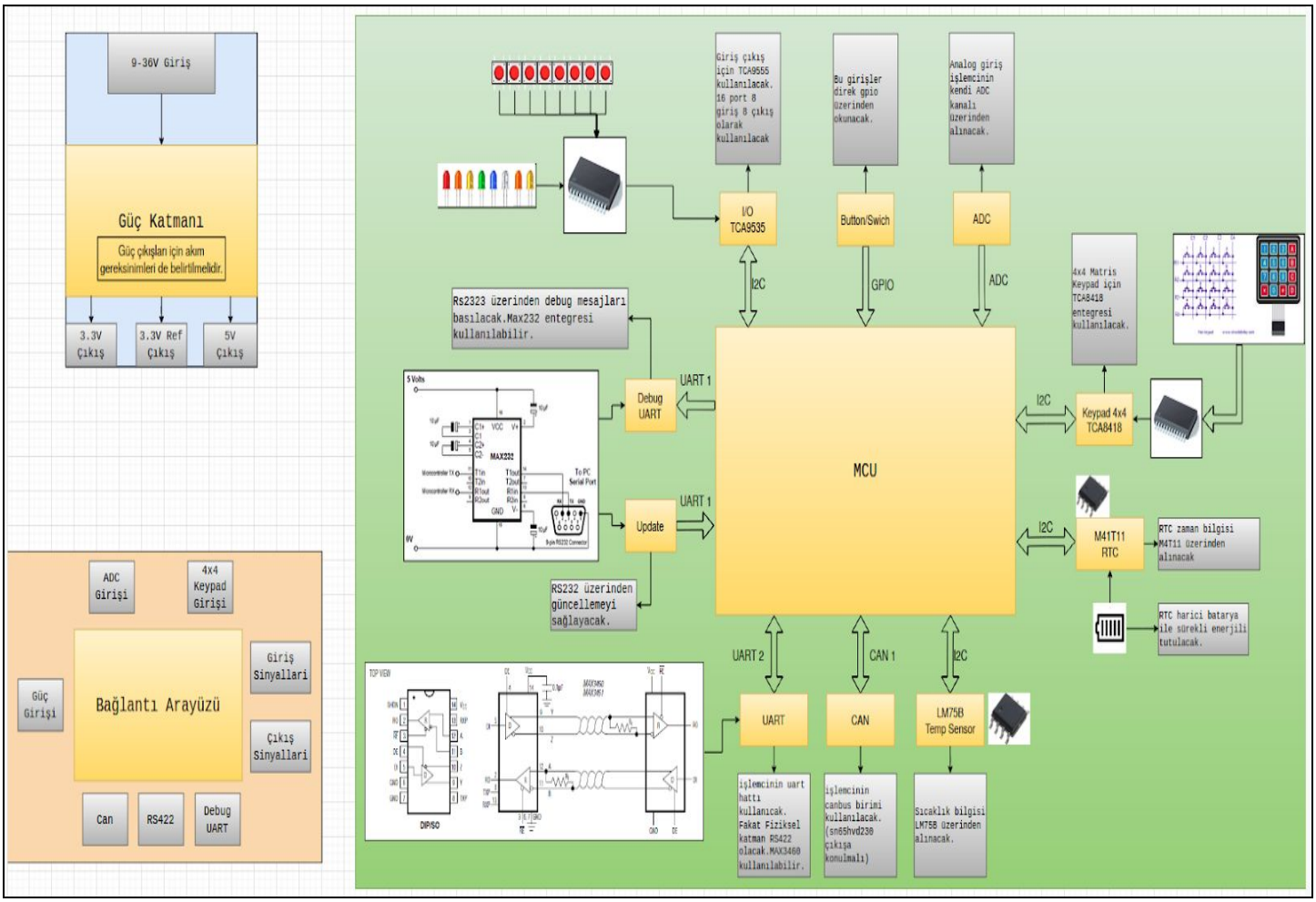
Yukarıdaki Donanım Bilgisi ve Gerekliliği başlığı altında donanım soyutlayan proje oluşturmanın avantajlarını incelemiş olduk. Projeye bu özelliği kazandırabilmek için ilk olarak donanım yapısı birimlere ayrılır ve sonrasında yazılımda da bu birimleri yöneten ilgili birimler oluşturulur. Bu kavramları ilk okunduğunda anlaması zor gelebilir, olayları örneklere dökmeye çalışarak anlamayı kolaylaştıralım. Örneğin bir giriş/çıkış çoklayıcı (I/O Expender) entegresi üzerinden bir hattı aktif yapmak isteyelim.

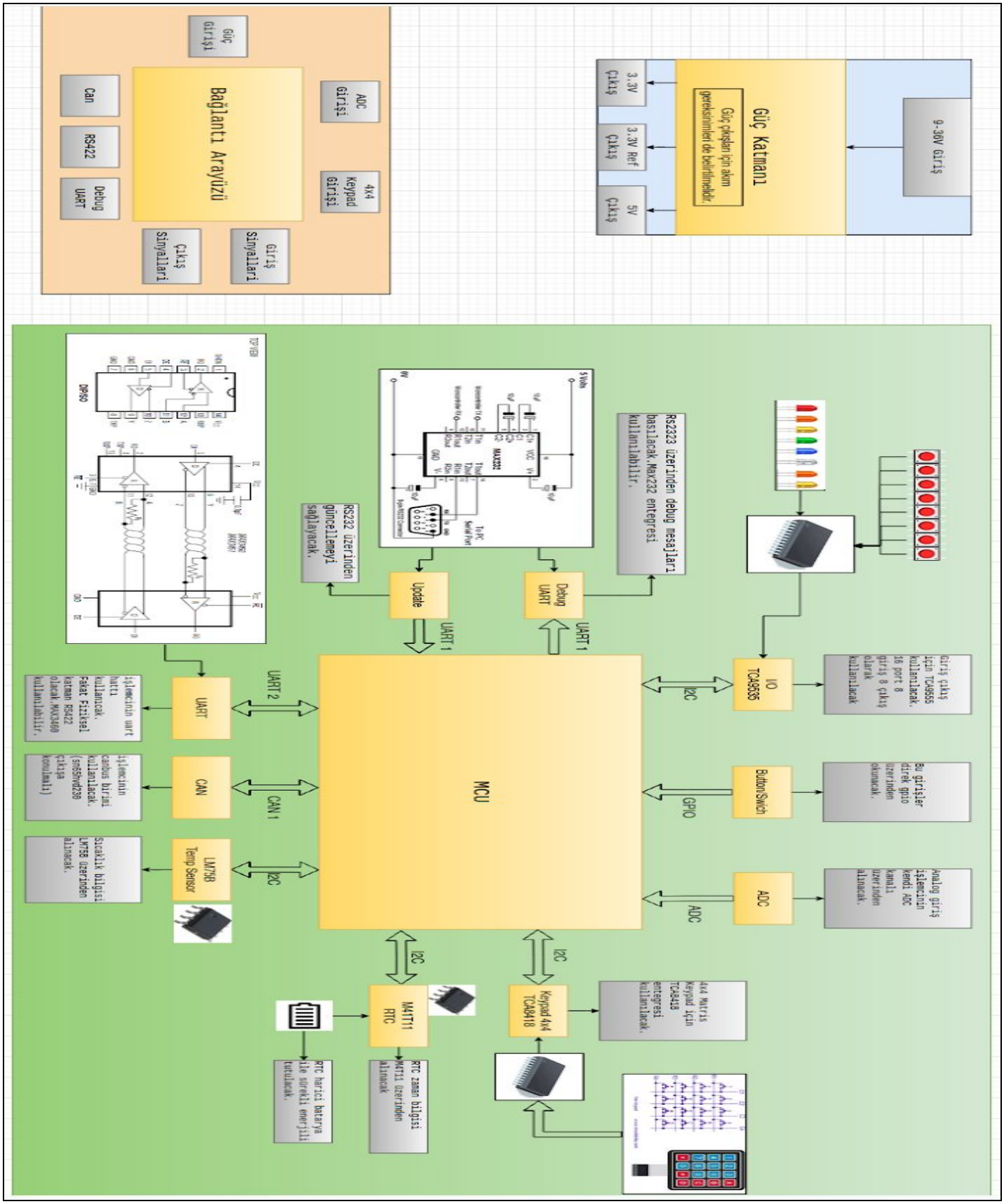
i2c_write(0x48, 0x04, 0x01); ==> 0x48 adresindeki entegrenin 0x04 adresindeki register alanına 0x01 yazarek ilk bini 1 yaptık

middIOController(EN_OUTPUT_1, ENABLE); ==> donanım soyutlama özelliği olan projede istenilen çıkışın bir yapılması için gerekli fonksiyon çağrımı. EN_OUTPUT_1 birinci çıkışı temsil eder ve ENABLE değeri ile çıkışın aktif yapılması sağlanır.

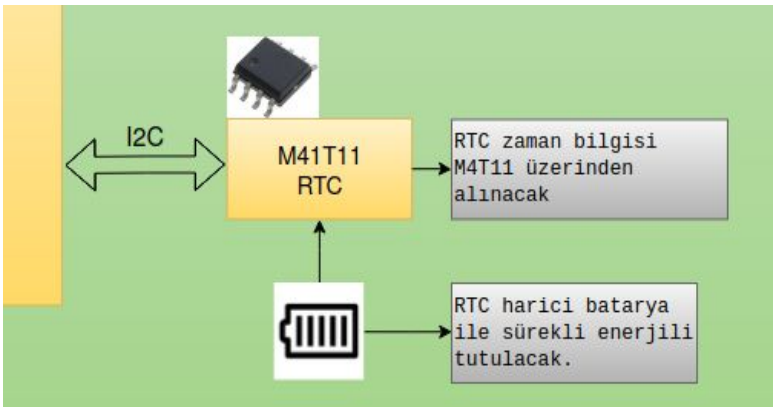
Bir çıkışın yönetilmesi için yapılan iki farklı kodlamayı görüyorsunuz. İlk çağrım donanımla direk bağlı iken ikinci çağrımın donanım bağımlılığı yoktur.

Donanım birimlerinin oluşturulması büyük oranla donanım ekibinin işidir ve yazılımcıların bu birimleri oluşturulmasındaki katkı azdır. Yazılım ekibi proje isteklerini karşılamak için kimi zaman gözden kaçan kimi zaman da akla sonradan gelen durumları belirtir. Aşağıda hem "Yazılım Gereksinimleri" başlığında verdiğimiz istekleri hem de projenin diğer isteklerini karşılayan donanım birimleri görülmektedir.




9-36V Giriş
Güç Katmanı
Güç çıkışları için akım gereksinimleri de belirtilmelidir.
3.3V Çıkış
3.3V Ref Çıkış
5V Çıkış
Bağlantı Arayüzü
Güç Girişi
ADC Girişi
4x4 Keypad Girişi
Can
RS422
Debug UART
Giriş Sinyalleri
Çıkış Sinyalleri
MCU
I/O TCA9535
I2C
Giriş çıkış için TCA9555 kullanılacak. 16 port 8 giriş 8 çıkış olarak kullanılacak
ButtonSwitch
GPIO
Bu girişler direk gpio üzerinden okunacak.
ADC
Analog giriş işlemcinin kendi ADC kanalı üzerinden alınacak.
Keypad 4x4 TCA8418
4x4 Matris Keypad için TCA8418 entegresi kullanılacak.
M41T11 RTC
RTC zaman bilgisi M41T11 üzerinden alınacak
RTC harici batarya ile sürekli enerjili tutulacak.
LM75B Temp Sensor
Sıcaklık bilgisi LM75B üzerinden alınacak.
UART
işlemcinin uart hattı kullanılacak. Fakat Fiziksel katman RS422 olacak.MAX3490 kullanılabilir.
CAN
işlemcinin canbus birimi kullanılacak. (sn65hvd230 ciksa konulmalı)
Debug UART
RS2323 üzerinden debug mesajları basılacak.Max232 entegresi kullanılabilir.
Update
RS232 üzerinden güncellemeyi sağlayacak.
UART 1
UART 2
CAN 1
MAX3232
DP9B
5 Volts
To PC Serial Port

Donanım birimlerini oluşturmak projeyi hem donanım ekibinin hem de yazılım ekibinin daha net anlamasını sağlar. Yukarıdaki resme bakınca hangi donanım birimlerinin kullanılacağını, görevlerede hangi entegrelerin kullanılacağı ve ve bunların işlemci ile nasıl bağlanacağını görmekteyiz.

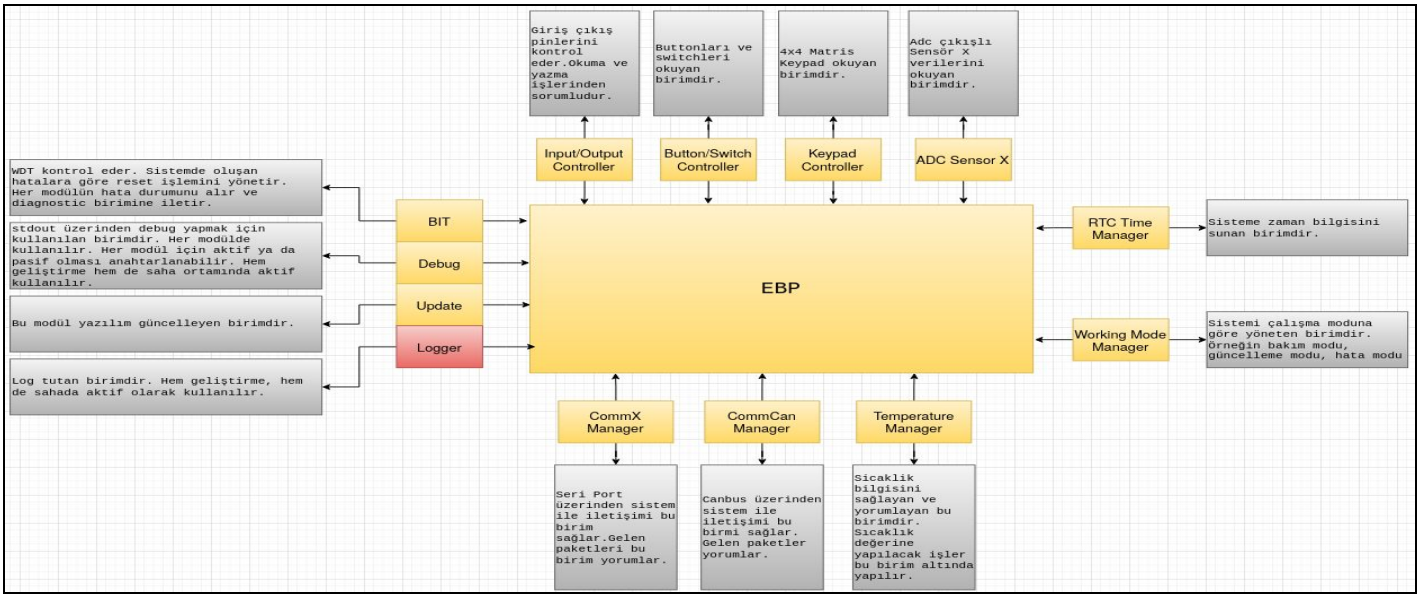


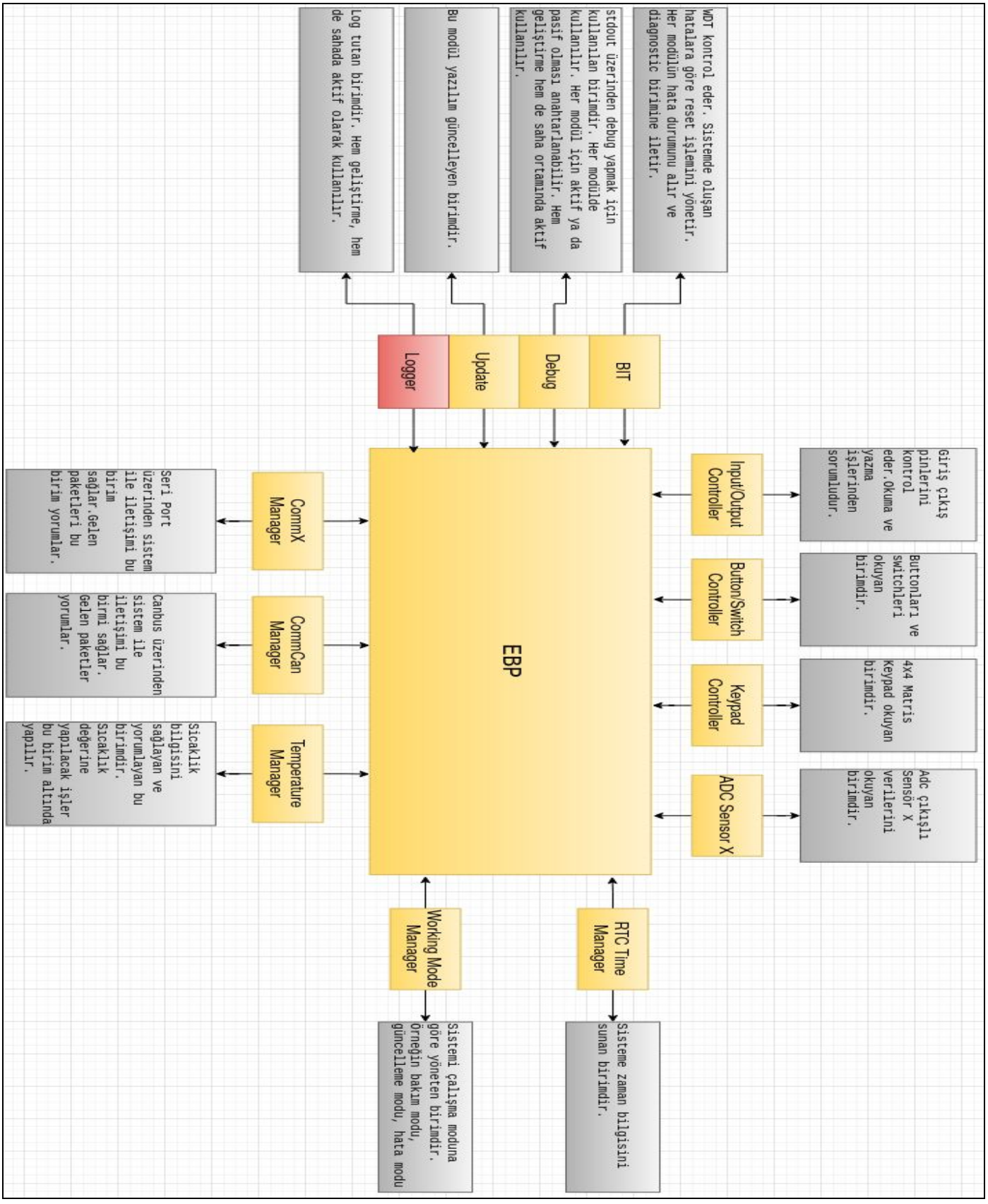
Rtc birimine baktığımızda sistemde zamanın nasıl sağlandığını görmekteyiz. Yazılım ekibi rtc biriminin olması nedeniyle yazılıma zaman biriminin eklenmesi gerektiğini anlar. Daha sonrasında rtc entegresinin zaman/kayması, rtc pil takibi, zaman okuma, zaman güncelleme gibi konuları kendi rtc birimi içinde kodlar.

Yazılım isterlerinin ve donanım yapısının belirlenmesinden sonra yazılım için ana eksikler tamamlanmış olur. Bundan sonra yazılım ekibi kendi içinde çalışmaya başlayabilir.

5. Yazılım Birimlerinin Çıkarılması

Bu başlığa kadar yazılım gereksinimleri ve donanım birimleri belirlenmiş oldu. Sıra yazılım birimlerini çıkarmaya geldi. Yazılım birimleri hem hem yazılım isterlerini hem de donanım birimlerini karşılayacak şekilde olmalıdır.




BIT
Debug
Update
Logger
EBP
Input/Output Controller
Button/Switch Controller
Keypad Controller
ADC Sensor X
CommX Manager
CommCan Manager
Temperature Manager
Working Mode Manager
RTC Time Manager
WDT kontrol eder. Sistemde oluşan hatalara göre reset işlemini yönetir. Her modülün hata durumunu alır ve diagnostic birimine iletir.
stdout üzerinden debug yapmak için kullanılan birimdir. Her modülde kullanılır. Her modül için aktif ya da pasif olması anahtarlanabilir. Hem geliştirme hem de saha ortamında aktif kullanılır.
Bu modül yazılım güncelleyen birimdir.
Log tutan birimdir. Hem geliştirme, hem de sahada aktif olarak kullanılır.
Giriş çıkış pinlerini kontrol eder.Okuma ve yazma işlerinden sorumludur.
Buttonları ve switchleri okuyan birimdir.
4x4 Matris Keypad okuyan birimdir.
Adc çıkışlı Sensör X verilerini okuyan birimdir.
Seri Port üzerinden sistem ile iletişimi bu birim sağlar.Gelen paketleri bu birim yorumlar.
Canbus üzerinden sistem ile iletişimi bu birmi sağlar. Gelen paketler yorumlar.
Sıcaklık bilgisini sağlayan ve yorumlayan bu birimdir. Sıcaklık değerine yapılacak işler bu birim altında yapılır.
Sistemi çalışma moduna göre yöneten birimdir. Örneğin bakım modu, güncelleme modu, hata modu
Sisteme zaman bilgisini sunan birimdir.

Yazılım birimleri projeyi toplu olarak görmemizi sağlar. Birimlerin birbirine olan bağlılığı, ortak yapılar gibi birçok bilgiyi kodlamaya başlamadan önce belirlenmesine olanak sağlar. Ayrıca hatalı kodlama mekanizmalarının kurulmaması için çok iyi referans olur. Ayrıca kodlama sırasında nerelerde ne yapıldığını görüp proje içinde kaybolunmasını engeller.

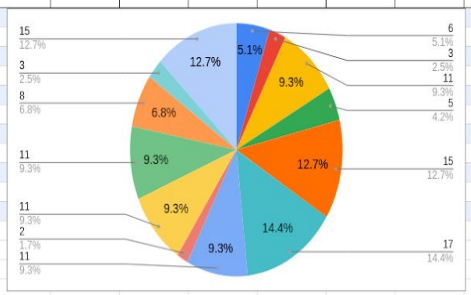
Yazılım birimleri yazılım tasarımı yaparken referans olarak kullanılabilir. Hatta uygulama akışının nasıl bir yapıda olması yada olmaması gerektiğini az buçuk ekibe anlatır. Örneğin event bazlı bir yapının bu projeye uyumlu olup olmadığını yazılım birimlerini inceleyerek karar verebiliriz.

Fakat şöyle bir beklentiye düşmemek gerekir; yazılım birimlerine bakınca direk yazılım gereksinimleri anlaşılmaz. Çünkü birimler, iş parçacıklarını ya direk yapar yada yapılması için ön koşul hazırlar. Örneğin "Input/Output Controller" birimi giriş/çıkış işlerinden sorumludur. Bu birim başka bir karar vericinin isteğini yerine getirmek için çalışır. Dolayısı ile yazılım gereksinimlerinde bu amaç ile bağdaşan bir madde göremeyiz. Ama gereksinimdeki "Sıcaklık Değerini 70 derecenin üzerine çıkarsa fan çalıştırılacaktır." maddesinin gerçekleşmesi için fan ünitesini açan çıkış ünitesini yönetir.

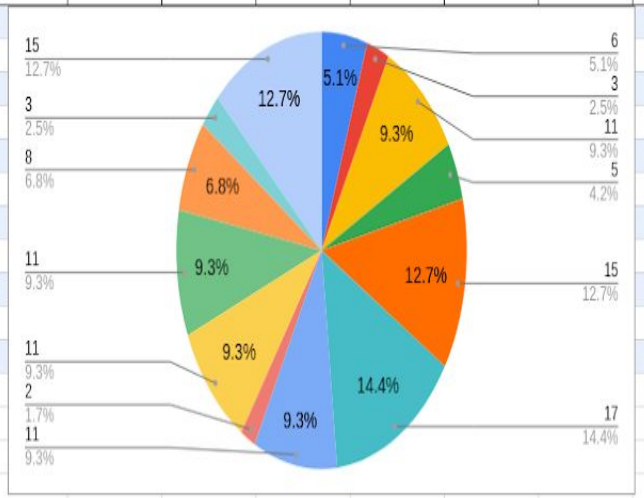
Yazılım birimleri ayrıca katmanlı yapıda tasarım için ön çalışmayı sağlar Hangi katmanda hangi birimlerin olacağı ne işler yapacağı ve üst katmanda kimlerle iletişim içinde olacağını buradan görebiliriz. Ayrıca kodlamada birimlerin ilişkisi ve bağlılığını da gösterir. Yanlış kodlama yapılarının oluşmasını engeller.

Yazılım birimleri ayrıca proje yönetimine de bilgi sunar. Proje ilerledikçe tamamlanan birimleri projenin tamamlanma oranını verir. Tabi her birimi birbiri ile aynı oranda tutmamak gerekir. Örneğin debug birimi ile keypad okuma biriminin bitirilmesi projeye aynı oranda katkı sağlamaz.

	Projeye Katkı Çarpanı													Gereksinim Karşılama	
Birimler	1	1	1	2	2	2	3	5	3	4	2	2	2	Değer	Oran
Debug	YG-1	YG-2				YG-6						YG-11		6	5.08
BIT	YG-1											YG-11		3	2.54
Update					YG-5	YG-1		YG-8				YG-11		11	9.32
Working Mode Manager									YG-9				YG-12	5	4.24
EBP	YG-1	YG-2	YG-3	YG-4				YG-8	YG-9			YG-11		15	12.71
I/O Controller			YG-3	YG-4		YG-6	YG-7		YG-9	YG-10	YG-11			17	14.41
Button/Switch Controller	YG-1	YG-2			YG-5			YG-8					YG-12	11	9.32
Keypad Controller												YG-11		2	4.00
Can Comm Manager	YG-1	YG-2			YG-5			YG-8				YG-11		11	9.32
UART Comm Manager	YG-1	YG-2			YG-5			YG-8				YG-11		11	9.32
Temperature Manager						YG-6				YG-10			YG-12	8	6.78
RTC Time Manager	YG-1					YG-6								3	2.54
ADC Sensor X		YG-2			YG-5		YG-7	YG-8				YG-11	YG-12	15	12.71



	Birimler	Projeye Katkı Çarpanı													Gereksinim Karşılama	
		1	1	1	2	2	2	3	5	3	4	2	2	2	Değer	Oran
3	Debug	YG-1	YG-2				YG-6						YG-11		6	5.08
4	BIT	YG-1											YG-11		3	2.54
5	Update					YG-5	YG-1		YG-8				YG-11		11	9.32
6	Working Mode Manager									YG-9				YG-12	5	4.24
7	EBP	YG-1	YG-2	YG-3	YG-4				YG-8	YG-9			YG-11		15	12.71
8	I/O Controller			YG-3	YG-4		YG-6	YG-7		YG-9	YG-10	YG-11			17	14.41
9	Button/Switch Controller	YG-1	YG-2			YG-5			YG-8					YG-12	11	9.32
10	Keypad Controller												YG-11		2	4.00
11	Can Comm Manager	YG-1	YG-2			YG-5			YG-8				YG-11		11	9.32
12	UART Comm Manager	YG-1	YG-2			YG-5			YG-8				YG-11		11	9.32
13	Temperature Manager						YG-6				YG-10			YG-12	8	6.78
14	RTC Time Manager	YG-1					YG-6								3	2.54
15	ADC Sensor X		YG-2			YG-5		YG-7	YG-8				YG-11	YG-12	15	12.71
16																



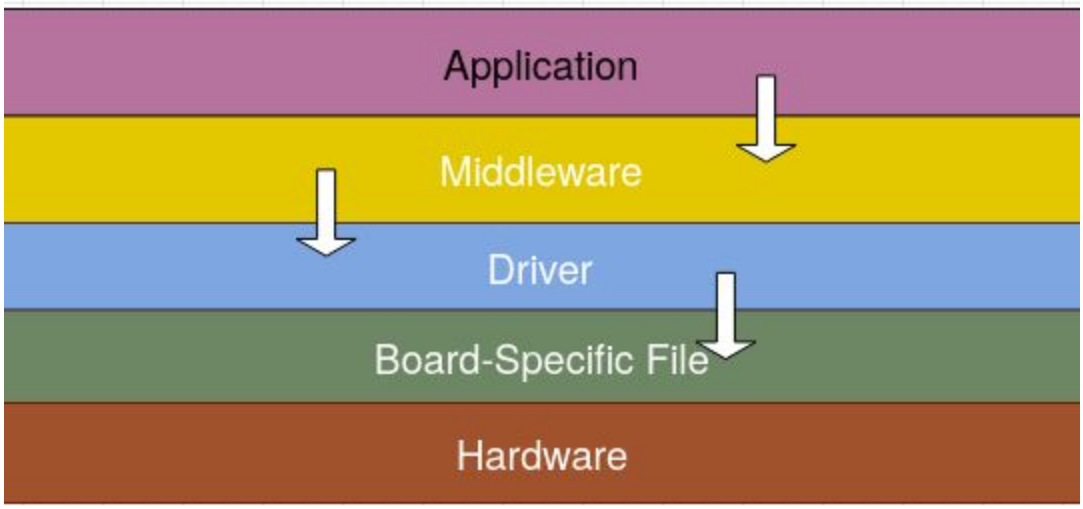
Yukarıdaki tabloda hangi yazılım gereksinimleri hangi birim ile karşılandığı görülmektedir. (Bu liste tamamen gerçek dışı olduğu için alakasız eşleşmeler olabilir.) Örneğin YG-4 gereksinimi EBP ve I/O Controller birimleri tarafından karşılanıyor. Bu tablo sonunda tüm yazılım gereksinimlerin karşılanıp karşılanmadığı, gözden kaçan isteğin olup olmadı görülür. Gereksinim karşılama değeri ile birimlerin proje içindeki önemi de görülmüş olur. Bu veri ile hangi birime ne kadar zaman ve kişi ayrılması gerektiği de kabaca çıkmış olur.

Fakat şunu da belirtmek gerekir ki yukarıdaki tablo ile proje yönetimi amaçlanmamıştır. Yazılım ekibinin kendi ilerleme durumunu görmesi ve zaman yönetimini sağlaması için hazırlanmıştır. Zaten proje yönetimi sadece yazılımı değil diğer birçok çıktıyı gözlemleyerek projeyi yönetir. Yazılım, donanım, kablaj, …

6. Yazılım Katmanları

Donanımdan bağımsız ve taşınabilir kod yazmak için yazılımı katmanlara ayırmak ilk şarttır. Sistem yapısına göre yazılım katmanları değişse de genel olarak ortak katmanları çoktur. Örneğin gömülü linux olan sistemde kernel katmanını ve işletim sistemi katmanını da ele almak gerekir. FreeRTOS olan yapı da aynı şekilde farklı katmanlar içerir.

Ister FreeRTOS, ister linux isterse de bare metal yapı olsun her katmanın amacı üst katmana hizmet etmek ve altındaki katmanların varlığını üst katmana yansıtmamaktır. Biz bu projemizde Bare-Metal kodlama yapacağımız için yazılım katmanımız aşağıdaki gibi olacaktır.



Yazılımı katmanlara ayırdıktan sonra her katmanın görevini ve sınırlarını iyi bilmek ve belirlemek gerekir. Genelde kodlama yapılırken katman sınırları aşılarak katmanların delinme hatası yapılıyor. Örneğin application katmanında direk driver katmanındaki bir fonksiyonu çağırdığımızda application katmanı sınırlarının dışına çıkmış olur. Bu tür hatalar yazılım tasarımını bozmaya başlar ve ilerleyen zamanlarda olumsuz etkisi ortaya çıkmaya başlar. Örneğin sıcaklık verisinin alındığı sensör değiştiğinde değişiklik hem application hem de driver

katmanında yapılması gerekiyor. Bu olumsuzlukları yaşamamak için her bir katman bir altındaki katman ile çalışmalıdır. Application -> middleware, middleware -> driver şeklinde olmalıdır.

Yukarıda belirlediğimiz katman yapısında “Board Specific File” katmanı kodun taşınabilirliği ve donanım bağımsız kodlama için oldukça önemli olsa da tam manasıyla bir katman oluşturmamaktadır. Ama yinede biz “Board Specific File” bir katman gözüyle bakıp katman kurallarını burası için de uygulayacağız. Hatta bu katmanı ilerde özel bir başlık altında inceleyeceğiz.

Application katmanı uygulama seviyesinde işlerin yapıldığı yerdir. Yazılım gereksinimlerinin çoğu bu katmanda karşılanır Örneğin can hattından gelen paketlerin yorumlanması(parse), keypad basımı sonrası ne yapılacağı, sıcaklığın belli derecelerinde neler yapılacağı bu katmanda yapılır. Ayrıca katmanın isminden de anlaşılacağı gibi donanımla hiçbir bağlantısı yoktur. Bu yüzden bu katmanda hiçbir driver yada board özgü fonksiyon, değişken kullanımı görülmez. Bu katman sadece middleware katmanını kullanır.

Middleware katmanı driver ile application katmanı arasındadır ve ana amacı application katmanına hizmet etmektir. Driver katmanından aldığı verilerin application katmanı kullanmadan önce işlenmesi gerekiyor ise bu katmanda yapılır. Örneğin adc okuması sonrasında oluşan dijital değerin anlamlı değere(sıcaklık, volt, akım) çevrilmesi bu katmanda oluşur. Yada haberleşme sırasında şifreli paket gidip geliyor ise paketleri şifrelemek yine bu katmanın işidir. Başka bir örnek ise zamanlayıcı(timer) kurma olarak verilebilir. Application katmanında birden fazla birim zamanlayıcı(timer) ihtiyacı olabileceğinden bu middleware katmanı birçok application katmanına hizmet etmiş olabilir.

Driver katmanı ise tahmin edileceği üzere kullanılan entegreleri süren kodların olduğu katmandır. Örneğin TCA8418 entegresi ile keypad okumak için bu katmanda tca8418 için driver olası gerekir.

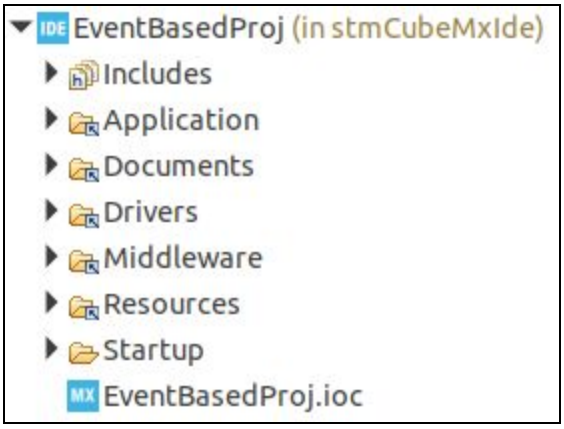
Board Specific File katmanı tamamen MCU etkisini ortadan kaldırmak üzere kullanılır. Bu katmanı ilerde daha detaylı inceleyeceğiz.

Katmanlı yapının birçok artısının yanında birkaç eksisi/dezavantaj de bulunmaktadır. Kod boyunun(code size) artmasına neden olabilir, gerçi bu eksi artılarının yanında hiç de önemli değildir. Donanım kesmelerinin yönetimini zorlaştırabilir.(Gerçi bu durum birazda kod kalitesine bağlıdır.) Kural olarak yukarıdan aşağıya doğru katman kullanımı olduğu için alt seviyeden yukarı haber vermek direk olarak yapılamaz. Örneğin oluşan bir donanım kesmesi sonrasında driver katmanından ne middleware ne de application katmanı fonksiyonu çağrılabilir. Bu problemi aşmak için ise callback yapısı kullanılır. Bu sayede alttan yukarı bildirim katman delinmesi olmadan tamamlanmış olur.

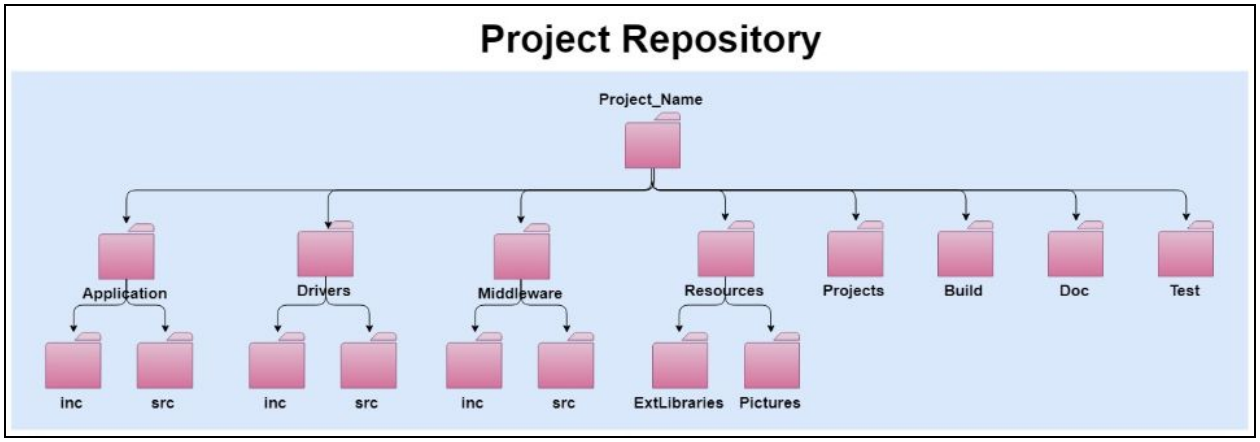
7. Klasör Yapısı

Yazılım projesinde önemli adımlardan biri de projenin klasör yapısını belirlemektir. Klasör yapısı aslında belirlenecek yazılımın katmanları ile bağlıdır. Örneğin driver katmanına ait kodların, klasör yapısında Driver isimli klasör altında bulunur. Tabi kodun derlenmesi için bu şart değildir ama projede neyin nerde olduğunu anlamak ve hangi kaynak dosyasının hangi seviyeye ait olduğunu anlamakta oldukça faydalıdır. Projemizde kullanacağımız klasör yapısı aşağıdaki gibi olacaktır.

Bu klasör yapısının aynısını kod geliştirme ortamımız olan Eclipse üzerinde de göreceğiz.



Proje klasör yapısının etki ettiği bir diğer nokta ise projenin versiyonlama sistemidir.(git, svn…) Çünkü oluşturulan tüm klasörler depoda(repository) oluşacağı için klasör yapısındaki düzen bize git deposunda da artı sağlayacaktır.



Yapılan değişikliğin hangi dizin altında olduğu kolayca görülebilir ve takip edilebilir olur. Klasör yapısı ile ilgili daha fazla bilgiye daha önce yazdığım bu yazımdan ulaşabilirsiniz.

- ➔ **Project_Name/Application:** Bu dizin altında sadece uygulama seviyesinde kullanılan kaynak(.c) ve başlık(.h) dosyaları bulunur. Yani uygulama katmanı kodlarını burada aramamız gerekir.
- ➔ **Project_Name/Drivers:** Projede kullanılan tüm driver dosyalarını içerir. Örneğin kullanılan bir LCD' nin driver burada bulunması gerekir.
- ➔ **Project_Name/Middleware:** Middleware tanımı, farklı iki uygulamayı ya da katmanı birbiriyle ilişkilendiren ara yazılım demektir. Bizim tarafımızda application seviyesindeki kodlar ile driver seviyesindeki kodları birbirine bağlayan kodları içerir. Application seviyesinde direk driver kodlarını çağırmayı engeller.
- ➔ **Project_Name/Projects:** Projenin oluşturulma dizini bu dizin altındadır. Örneğin Eclipse ile proje oluşturulacak ise bu dizin altında oluşturulmalıdır. Bu dizin altında kod dosyaları bulunmaz. Örneğin Eclipse projesini açmak için gerekli olan .project uzantılı dosya bulunur. Bu dosya açılarak tüm proje açılmış olur.
- ➔ **Project_Name/Documents:** Proje geliştirilmesinde kullanılan ya da oluşturulan tüm belgeler bu dizin altında toplanır. Örneğin uygulamanın akış diyagramı, proje gereksinimleri, donanım şeması …

Ayrıca belirlenen bir proje klasör yapısı mümkün olduğunca tüm projelerde kullanılmaya çalışılmalıdır. Bir projeden başka bir projeye geçildiğinde aynı yapı olmasından dolayı çok çabuk uyum sağlanır. Bir diğer artısı da projeden projeye kaynak kod aktarımında kolaylık sağlamasıdır.

8. Borda Özgü Başlık Dosyası (Board-Specific Header File)

Board Specific File katmanı kullanılan MCU fonksiyonlarını içererek üst birimlerin donanım bağımlılığını ortadan kaldırır. Örneğin I2C kullanan birim direk işlemcinin i2c fonksiyonunu kullanmak yerine onu function like makro yöntemi ile sarmalayan makroyu kullanması ile artık MCU bağımlı çağrım yapılmamış olur.

Board-Specific Header File içinde MCU çevre birimlerini çevreleyen makrolar, donanımsal bilgiler, I2C hattındaki entegrelerin adres değerleri, hafıza birimlerinin(eeprom, fram) boyutları, gpio isimleri gibi bordu tanımlayan makrolar bulunur. Git hesabında bulunan örnek projede Board klasörü altında farklı boardlar için gerekli dosyalar bulunmaktadır. Örneğin EBP uygulamasını stm32f407 işlemcisinin üzerinde koşturmak için "BoardConfig_STM_010101.h" dosyası kullanılır. Aynı uygulamayı STM32L476RGTX işlemcisi üzerinde koşturmak istenildiğinde ise "BoardConfig_STM_LP_010101.h" dosyası kullanılır.

Borda özgü başlık dosyası ile donanımsal değişiklikleri kolayca tek bir noktadan gerçekleştirmiş oluruz. Örneğin I2C hattında olan bir entegrenin adresi değiştiğinde bu dosya içindeki ilgili makroya yeni değer atanır. Örneğin projemizde I2C hattında bulunan entegrelerin adresleri aşağıdaki gibi ayarlanmıştır.

```
#define TCA9555_I2C_ADDR                    (0x42)
#define TCA8418_I2C_ADDR                    (0x68)
#define LM75B_I2C_ADDR                      (0x66)
#define M41T11_I2C_ADDR                     (0x35)
```

```
/******************** BOARD I2C CONTROL *****************/
#define _I2C_LINE_1                                     (hi2c1)
#define _I2C_LINE_3                                     (hi2c3)

#define _I2C1_INIT()                                    MX_I2C1_Init()
#define _I2C2_INIT()                                    MX_I2C2_Init()

#define _I2C_IS_DEVICE_READY(line, deviceAdr, try, timeout)     HAL_I2C_IsDeviceReady(&line, (uint16_t)deviceAdr, try, timeout)

#define _I2C_SEND(line, deviceAdr, buff, leng)          HAL_I2C_Master_Transmit(&line, (uint16_t)deviceAdr, \

#define _I2C_RECIVE(line, deviceAdr, buff, leng)        HAL_I2C_Master_Receive(&line, (uint16_t)deviceAdr,\
                                                                (uint8_t *)buff, leng, 300)
#define _I2C_WRITE(line, deviceAdr, addr, buff, leng)   HAL_I2C_Mem_Write(&line, (uint16_t)deviceAdr, addr, \
                                                            I2C_MEMADD_SIZE_8BIT, (uint8_t *)buff, leng, 100)

#define _I2C_READ(line, deviceAdr, addr, buff, leng)    HAL_I2C_Mem_Read(&line, (uint16_t)deviceAdr, addr,\
                                                           I2C_MEMADD_SIZE_8BIT, (uint8_t *)buff, leng, 100)
```

Borda özgü başlık dosyası yönetimi ile projelerimiz farklı platformlar için hızlıca hazır hale getirebiliriz. Önişlemci komutları ile bu anahtarlamalar aşağıdaki örnek kullanımdaki gibi yapılabilir.

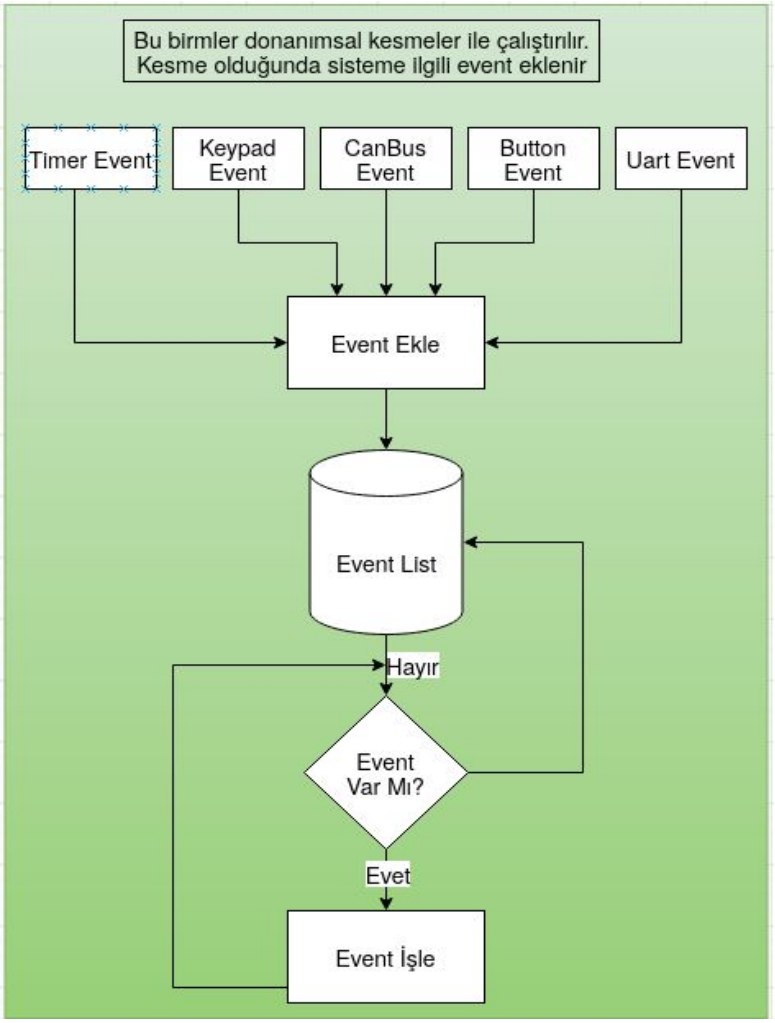
```
#define BOARD_LINUX_PC        (1)
#define BOARD_STM_010101      (2)
#define BOARD_STM_LP_010101   (3)
/* add new board here*/

#ifdef __linux
      #define CURRENT_BOARD   (BOARD_LINUX_PC)
#else
      #define CURRENT_BOARD   (BOARD_STM_LP_010101)
#endif

#if (CURRENT_BOARD == BOARD_STM_010101)
      #include "BoardConfig_STM_010101.h"
#elif (CURRENT_BOARD == BOARD_LINUX_PC)
      #include "BoardConfig_LinuxPC.h"
#elif (CURRENT_BOARD == BOARD_STM_LP_010101)
      #include "BoardConfig_STM_LP_010101.h"
#else
      #error "!!! Current board is undefined. Check GeneralBoardConfig.h file !!!"
#endif
```

9. Uygun Yazılım Yapısı

Projelerin farklı davranış biçimleri vardır. Kimi projede girişlere karşı işlem yapılırken kimi projelerde belirlenen işler sürekli yada periyodik olarak yapılır. Örneğin bir sayaçta ölçüm yapmak cihazın sürekli yaptığı iştir. Fakat televizyonun kumandadan gelen sinyali ele elması ise bir girdiye karşılık yapılacak iştir.

Projede bu ayrımları görmek yapılacak mimarinin ana iskeletini oluşturur. Çünkü bu durumlara karşın donanımın özellikleri de kullanılabilir. Örneğin bare metal kodlamada girişleri takip etmek için donanım kesmeleri kullanmak avantaj sağlar. Gömülü linux ortamında ise donanım kesmelerini direk kullanılmadığından thread, sinyal ve posix fonksiyonlarını kullanmak avantaj sağlar.

Projemizdeki yazılım gereksinimlerine baktığımızda hem periyodik olarak yapılacak işlerin olduğunu hem de dışarıdan gelen sinyallere karşı yapılacak işlerin olduğunu görüyoruz. Örneğin can yada uart hattından gelen mesaja karşı bir şeylerin yapılması, mesaja karşılık yapılan iştir. Sıcaklığın aralıklarla olarak ölçülüp, sıcaklık değerine göre işlem yapmak ise periyodik iştir. Bu yüzden event bazlı proje yönetimini bu projede uygulamaya karar verdik.



Yapı kısaca şu şekilde çalışacaktır: donanım birimlerinden gelen her istek sisteme bir event olarak girilecektir. Event sisteme girilirken önceliği ve taşıdığı bilgi gibi veriler ile girilecektir. Uygulamamızın ana döngüsü ise event listesinden önceliği en yüksek event alarak sırasıyla eventleri işler. Sistemde event olmadığı zaman bir iş yapılmaz.

Eventler ayrıca gruplayarak da istenildiğinde sadece belli bir gruptaki eventleri işleme fırsatı yaratmış oluruz. Örneğin bakım modunda keypad dijital giriş/çıkış gelen eventleri işlememek istenebilir. Çünkü bakım modunda cihazın fonksiyonel olarak çalışması değil de can, uart yada keypad üzerinden yeni ayarların yapılması gibi işler yapılır.

Aşağıda sisteme kaydedilecek her bir event için kullanılan yapıyı görmekteyiz. Oluşan event hangi kaynaktan oluştuğu, önceliği, taşıdığı veriyi görebiliriz.

```
typedef struct _EventStr
{
	U32				event		:10;	// 10 bit
	EVENT_SOURCE	source		:10;	// 10 bit
	U32				leng		:10;	// 10 bit
	EVENT_PRIORITY	priority	:2;		// 2 bit

	U32				value;
	void*			param;

}EventStr;
```

```
/** @brief Event List */
typedef enum _EVENTS
{
	EN_EVENT_NO_EVENT,
	EN_EVENT_BIT,
	EN_EVENT_KEYPAD,
	EN_EVENT_KEYS,
	EN_EVENT_CAN_COMM_MSG, //can line message received
	EN_EVENT_SERIAL_COMM,
	EN_EVENT_SERIAL_COMM_TIMEOUT,
	EN_EVENT_CAN_COMM_TIMEOUT,
	EN_EVENT_CHECK_TEMP

}EVENT;
```

```
/** @brief Event Priority */
typedef enum _EVENT_PRIORITY
{
	EN_PRIORITY_LOW,
	EN_PRIORITY_MED,
	EN_PRIORITY_HIG,
	EN_PRIORITY_EMG
}EVENT_PRIORITY;

/** \brief Event Source */
typedef enum _EVENT_SOURCE
{
	EN_SOURCE_PER_TIMER     = 0001, //periodic timer
	EN_SOURCE_ONE_TIMER     = 0x02, //one-shot timer
	EN_SOURCE_INTERNAL      = 0x04,
	EN_SOURCE_COMM_LINE     = 0x08,
	EN_SOURCE_CAN_COMM_LINE = 0x10,
	EN_SOURCE_USER_INPUT    = 0x20,
	EN_SOURCE_DIGITAL_INPUT = 0x40,

	EN_SOURCE_ALL = 0X3FF //max 10 bit

}EVENT_SOURCE;
```

Örnek olarak keypad basıldığında sisteme nasıl event yüklendiğine bakalım.

```
appEventThrowEvent(EN_EVENT_KEYPAD, EN_SOURCE_USER_INPUT, EN_PRIORITY_MED,
NULL, 0, key);
```

appEventThrowEvent() fonksiyonu ile event listeye eklenmiş olur. Fonksiyona gönderilen argümanları anlamak için fonksiyonun bildirimine bakmamız yeterlidir.

```
RETURN_STATUS appEventThrowEvent(EVENT event, EVENT_SOURCE source,
EVENT_PRIORITY priority, void *param, U32 leng, U32 value)
```

Event listesinden event almak için ise appEventGet() fonksiyonu kullanılır.

```
EventStr* appEventGet(U32 timeoutMs, U32 eventSource);
```

TimeoutMs belirlenen sürede event beklemek için kullanılır. Eğer o süre içinde event listesine bir event düşmez ise fonksiyon NULL döner. eventSource parametresine ise istenillen event

kaynakları yazılabilir. Örneğin sadece can hattı için oluşan eventleri istemek istediğimizde bu parametreye `EN_SOURCE_CAN_COMM_LINE = 0x10`, değeri geçilir. Eğer birden fazla event kaynaklarına bakmak istiyorsak event kaynakları OR operatörü ile birleştirilerek gönderilir. Örneğin sadece UART ve keypad gelen eventleri istiyorsak aşağıdaki gibi bir çağrım yapmalıyız.

```
EventStr *event;

event = appEventGet(0, EN_SOURCE_COMM_LINE | EN_SOURCE_USER_INPUT);
if (NULL != event)
{
    handleEvent(event);
    appEventClearEvent(event);
}
```

Yapının nasıl çalıştığı git deposunda bulunan örnek kod üzerinden daha net görülebilir. Ayrıca tüm belgeler git deposunda Documents dizini altında bulunuyor.

Event listesindeki event aldıktan sonra sıra o event işlenmesine gelir. Eventleri ele alan birim de cihazın durumuna göre farklı şekilde yorumlaması gerekebilir. Örneğin cihaz belli bir donanımının çalışmaması fark ederek kendini hata moduna sokabilir ve bundan sonra oluşacak her event hata modunun varlığına göre ele alınmalıdır. Mesela hata modunda can ve uarttan gelen mesajlara karşılık vermeyip sadece can üzerinden periyodik hata durumunu yayınlayan mesaj gönderebilir. Projemizde bu yaklaşımı karşılayan fonksiyonlar aşağıda gösterilmiştir.

```
/**
 * @brief handle event in working mode
 * @param event pointer
 * @return if everything is OK, return SUCCESS
 *          otherwise return FAILURE
 */
RETURN_STATUS appEvntHandWorkingMode(const EventStr *event);

/**
 * @brief handle event when device closed
 * @param event pointer
 * @return if everything is OK, return SUCCESS
 *          otherwise return FAILURE
 */
RETURN_STATUS appEvntHandClosedMode(const EventStr *event);
```

```
/**
 * @brief handle event in maintenance mode
 * @param event pointer
 * @return if everything is OK, return SUCCESS
 *          otherwise return FAILURE
 */
RETURN_STATUS appEvntHandMaintenanceMode(const EventStr *event);

/**
 * @brief handle event in failure mode
 * @param event pointer
 * @return if everything is OK, return SUCCESS
 *          otherwise return FAILURE
 */
RETURN_STATUS appEvntHandFailureMode(const EventStr *event);
```

10. Kötü Tasarıma Neden Olabilecek Üç Ana Unsuru Ortadan Kaldırmak

Kodlamaya başlamadan önce yazılım tasarımı yapılmalıdır. Bu cümleye itiraz edecek kimse yoktur ve neredeyse herkes biz zaten kodlamadan önce tasarımı oluşturuyoruz der. Ama gerçek hayatta kodlamaya başlamadan önce tasarım yapan firma/yazılımcı sayısı oldukça azdır.

Tasarımlar genelde ihtiyaçları karşılamak üzere yapılan çalışma olarak görülüyor. Fakat ayrıntılara yer verilmediği zaman kodlama esnasında birçok eksiklik olduğu görülür. Tasarımlar yazılım gereksinimlerini kapsamasının yanında yazılım prensiplerini de içermelidir. Aksi takdirde birden fazla kişinin yazdığı kodların uyum içinde çalışması zorlaşır. Ayrıca kodun taşınabilirliğine, genişletilebilirliğine ve okunabilirliğine tasarımda yer verilmez ise bu özellikler hiç bir zaman yazılımcıların bireysel çabaları ile sağlanamaz.

Yazılım birimleri, yazılım katmanları ve bord özgü başlık dosyası çalışması ile aslında iyi tasarım için bazı şartları sağlamış olduk. Örneğin katmanlı yapı, Rigidity(Esnemezlik) problemini aşmamızı sağlar. Rigidity(Esnemezlik): Kullanılan tasarımın esnek olmadığını gösterir. Yani kullanılan tasarımın geliştirmeye, yeni özellikler eklemeye uygun olmadığını gösterir.

Aşağıda yazılım tasarımında aşağıdaki üç ana unsurdan kaçarak tasarım yapılmalıdır.

- **Rigidity (Esnemezlik):** Kullanılan tasarımın esnek olmadığını gösterir. Yani kullanılan tasarımın geliştirmeye ve plug-in mimarisine uygun olmadığını gösterir.
- **Fragility (Kırılganlık):** Sistemin bir yerinde yaptığınız bir değişikliğin, sistemin bir başka yerinde sorun çıkarmasıdır.

- **Immobility (Sabitlik):** Geliştirdiğiniz bir modülün tekrar kullanılabilir olmadığını gösterir.

Yazılım prensipleri için daha fazla bilgiye buradaki yazımdan ulaşabilirsiniz.

11. Debug Mesajlarını Önemsemek

Debug yazılımcılar arasında genelde hatayı bulmak ve çözmek olarak algılanmakta. Aslında yazılımcı kodlama yaparken uygun gördüğü noktalara debug mesajları koyarak hatanın kendiliğinden önüne serilmesini sağlayabilir.

Debug mesajları için birçok yöntem vardır. Kimi debug mesajları uyarı durumunda, kimi debug mesajları hata durumunda kimi ise bilgi amacıyla konsola/çıkışa düşer. Bu yapının kurulması hem geliştirme hem de sahada cihazın çalışması esnasında çok fazla yardımcı olur.

Debug mesajlarını genelde application katmanında kullanmayı tercih ediyorum. Bazı noktalarda da middleware katmanında da kullanmak faydalı oluyor. Örneğin middleware katmanında uart kanalından alınan her paketi yazdırarak gelen paketlerin hatalı olup olmadığını görebiliriz. Application katmanında başarısız işlemlerin sonrasında debug mesajı basmak faydalı olur. Hangi dosyadaki hangi fonksiyon hatalı dönüyor kolayca anlaşılmış olur.

```
-> initMcuCore():55:  ->[I] initMcuCore return: 0
-> appSystemSetup():120:
##--- > Board File: BoardConfig_STM_010101.h - Board Name: EBP_STM_010101 - Board Version: V1.0 Board MCU: STM32F407VGT6-100-LQFP-CORTEX-M4

-> appSystemSetup():121: ##--- > SW Version 1.0.0
```

Yukarıdaki debug mesajından işlemci çevre birimlerinin kurulumunun başarılı oldğunu anlıyoruz. Yazılımın kullandığı borda özgü başlık dosyasının **Board File: BoardConfig_STM_010101.h** olduğunu görüyoruz. Kullanılan donanım ile donanım dosyasının uyumlu olup olmadığını buradan sınayabiliriz. **appSystemSetup():121: ##->SW Version 1.0.0** ile de yazılım versiyonu takibi yapabiliriz. Bu özellikler özellikle proje ile için birden fazla bord ve yazılım versiyonu oluşmaya başladığında oldukça yardımcı olur.

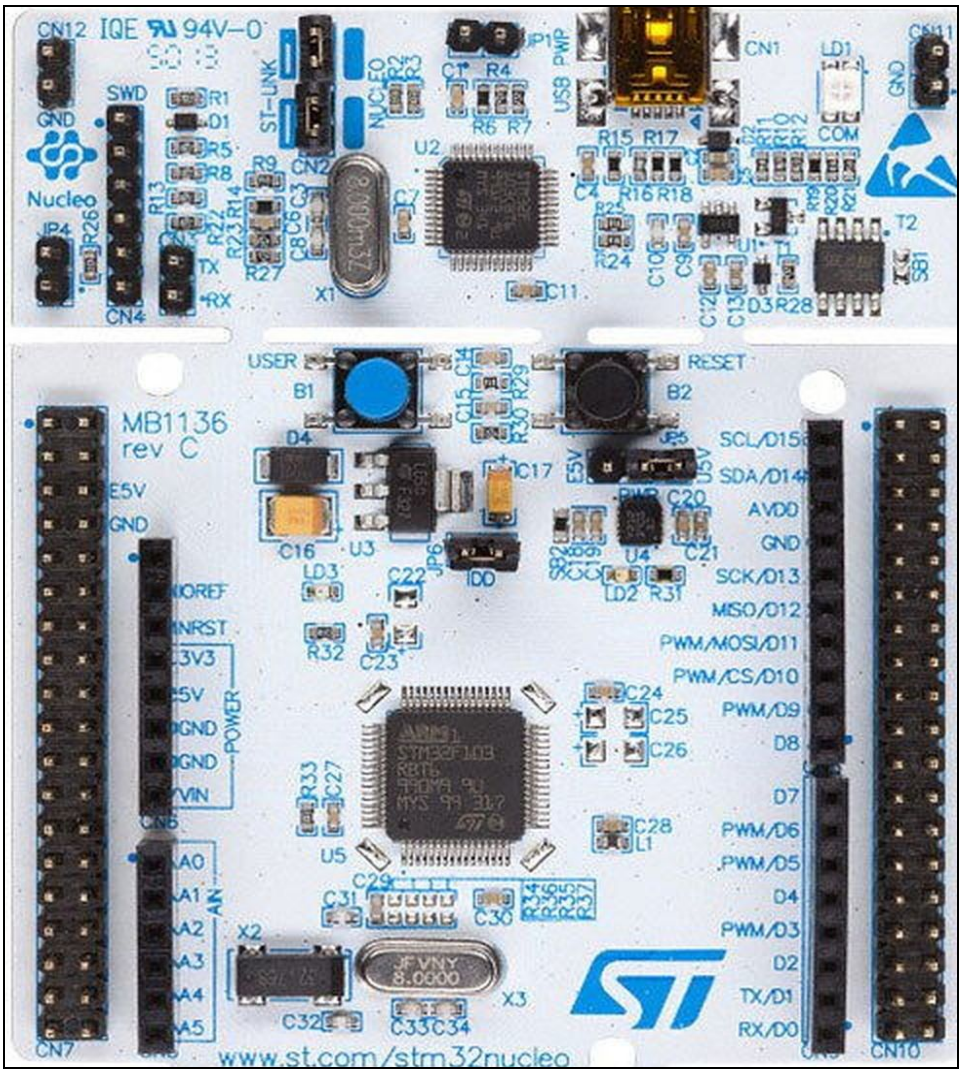
```
-> appSystemSetup():121: ##--- > SW Version 1.0.0

-> initDeviceDrivers():96:  ->[I] Device Driver Starting
-> initDeviceDrivers():103:  ->[I] Device Driver completed, result 0
-> middSysTimerInit():79: -> [I]middSysTimerInit return 0
-> middEventTimerInit():88: -> [I]middEventTimerInit return: 0
-> middIOInit():70: ->[I] midd IO Init retVal: 0
-> middKeypadInit():78: ->[I] midd keypad Init retVal: -1
-> middKeysInit():224: ->[I] midd keys Init retVal: -1
-> middLedInit():105: ->[I] midd Key_Led Init retVal: -1
-> appSysVarInit():36: ->[I] Starting Sys Value List:
->[I] Sys Mode: 0

-> appEventDeleteEvents():93: ->[I] All events deleted !!!
-> initSWRequirement():86:  ->[I] initSWRequirement return value: 0
-> appSystemSetup():142:  ->[E] !! CANBUS LINE 2 Could not be started !!!!!!!


-> appSystemSetup():148:  ->[E] !! CANBUS LINE 1 Could not be started !!!!!!!
```

Debug mesajlarının devamına baktığımızda işlerin rast gitmediğini bazı hataların oluştuğunu görüyoruz. Keypad ve giriş/çıkış çoklayıcı entegrelerinin kurulumunda hata yaşandığını görüyoruz. Alınan bir diğer hata ise canbus hatlarının kurulumu ile ilgili olduğunu görüyoruz. İşte buna benzer şekilde çalışma zamanında oluşan hataları debug mesajları üzerinden takip edebiliriz.

Proje kodlarını stmCubeIDE ile derleyip stm32F Discovery kart üzerinde çalıştırabilirsiniz. Keypad ve diğer harici entegreler olmadığı için bu şekilde hata mesajlarını inceleyebilirsiniz.

12. Projeyi Başka Bir İşlemci Üzerinde Çalıştırmak

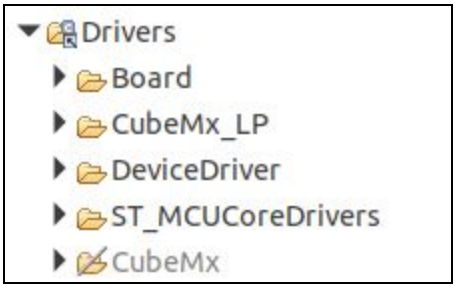
Belgenin başından beri taşınabilir ve donanım bağımsız kodlamadan bahsedip de bu özelliği test etmeden olmazdı. Projenin başka bir donanım üzerinde koşmasını sağlamak için yeni donanıma uygun borda özgü başlık dosyasının oluşturulması yeterli.



Projemizi üzerinde koşturacağımız ikinci bordumuz yukarıda resmi olan NUCLEO-L476RG - STM32 Nucleo-64 STM32L476RG geliştirme kartı. (Elimde fazladan bu kart olduğu için bu kart üzerinde deneme yaptım). Bu kart için BoardConfig_STM_LP_010101.h dosyasını oluşturdum ve kullanılacak bord dosyasını aşağıdaki gibi ayarladıktan sonra işlem tamamlanmış oldu.

```
#define BOARD_LINUX_PC      (1)
#define BOARD_STM_010101    (2)
#define BOARD_STM_LP_010101 (3)
/* add new board, dont change queue*/

#ifdef __linux
    #define CURRENT_BOARD   (BOARD_LINUX_PC)
#else
    #define CURRENT_BOARD   (BOARD_STM_LP_010101)
#endif

#if (CURRENT_BOARD == BOARD_STM_010101)
    #include "BoardConfig_STM_010101.h"
#elif (CURRENT_BOARD == BOARD_LINUX_PC)
    #include "BoardConfig_LinuxPC.h"
#elif (CURRENT_BOARD == BOARD_STM_LP_010101)
    #include "BoardConfig_STM_LP_010101.h"
#else
    #error "!!! Current board is undefined. Check GeneralBoardConfig.h file !!!"
#endif
```

Projenin derlenmesi için yapılması gereken son işlem ise stm32f407 işlemcisine ait olan cubeMx kodlarını projenin dışında tutmak gerekli. Bunun için Driver dizini altında bulunan cubeMx dosyasının üzerinde sağ tıklayıp "Resource Configuration->Exclude from build" adımlarını yapmamız yeterli olacaktır. İşlem sonunda Driver dizini aşağıdaki gibi olacaktır.



İki projede git reposunda bulunmaktadır. Hangi donanıma özgü derleme yapılacak ise cubeMxIDE ilgili proje inport edilebilir.

13. Son

Bu belgede özetle proje geliştirirken izlenmesi gereken adımlar, taşınabilirlik ve donanım bağımsız kodlama yapısı üzerinde durulmuştur. Herkesin bildiği üzere bu konular çok daha

detaylı ve anlatımı uzun olan konulardır. Burada okuyucuya sadece özet bilgi verilmek istenmiştir.

Ayrıca kodlama yapmak için donanım ortamının bulunması mecburi değildir. Kendi kendinize donanım birimlerini çıkardıktan sonra projenin kodlayabilirsiniz. Hatta donanımı yazılımınızda ne kadar çok soyutlarsanız o kadar iyi kodlama ve sistem kurmuş olacaksınızdır.

Zafer Satılmış - 24.02.2021