Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Szabó Attila

ESZKÖZMEGHAJTÓ PROGRAM FEJLESZTÉSE AUTÓIPARI SZABVÁNY ALAPJÁN

KONZULENS

Szikszay László

(Thyssenkrupp Components Technology Hungary Kft.)

Dr. Sujbert László, docens

BUDAPEST, 2019

Tartalomjegyzék

Összefoglaló ... **7**
Abstract ... **8**
1 Bevezetés ... **9**
1.1 Feladat értelmezése ... 9
2 SPI kommunikációs interfész ... **11**
2.1 Bevezetés ... 11
2.2 Az interfész felépítése ... 11
2.3 Buszelrendezések ... 12
2.3.1 Single slave konfiguráció ... 12
2.3.2 Párhuzamos konfiguráció ... 12
2.3.3 Daisy-chain konfiguráció ... 12
2.4 Adatátvitel leírása ... 13
3 Az AUTOSAR szabvány ... **15**
3.1 Bevezetés ... 15
3.2 Szoftver architektúra ... 15
3.2.1 Alapvető rétegek ... 15
3.2.2 Basic Software (BSW) réteg ... 17
3.2.3 Funkcionális felosztás ... 18
3.3 SPI Handler Driver ... 18
3.3.1 Általános működés ... 19
3.3.2 Egyszerű szinkron SPI Handler/Driver ... 20
3.3.3 Egyszerű aszinkron SPI Handler/Driver ... 20
3.3.4 Fejlesztett (szinkron/aszinkron) SPI Handler/ Driver ... 21
4 A mikrokontroller SPI modul működése ... **22**
4.1 Felépítés ... 22
4.2 Inicializálás ... 23
4.3 Adatküldés ... 24
4.4 Megszakítások ... 24
5 SPI eszközmeghajtó program felépítése, működése ... **25**
5.1 A program tervezése ... 25

5.2 Adattípusok 26
5.2.1 Szabványos adattípusok és szimbólumok 26
5.2.2 Egyéb típusok és szimbólumok: 29
5.3 Adat struktúrák 30
5.3.1 Inicializáló struktúra 30
5.3.2 Háttérstruktúrák 32
5.3.3 A Queue struktúrái 33
5.4 Felhasználó függvények 34
5.4.1 inicializálás és deinicializálás 34
5.4.2 Adatkezelő függvények 35
5.4.3 Státusz lekérdező függvények 36
5.4.4 Küldést megvalósító függvények 38
5.5 Privát függvények 39
5.5.1 Kiírással kapcsolatos függvények 39
5.5.2 Queue-val kapcsolatos függvények 42
5.5.3 Hardverrel kapcsolatos függvények 44
5.6 Küldés állapotdiagramja 45
5.6.1 Szinkron működés 45
5.6.2 Aszinkron működés 47
5.7 Megvalósított funkciók 48
5.7.1 Sequence megszakíthatóságának engedélyezése 48
5.7.2 DET követelmények 48
6 A feladat összegzése, tanulságai 52
Irodalomjegyzék 53
Függelék 54

SZAKDOLGOZAT-FELADAT

Szabó Attila (RMERDD)

szigorló villamosmérnök hallgató részére

Eszközmeghajtó program fejlesztése autóipari szabvány alapján

A modern gépjármûvek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzõinek javításában stb. egyre jelentõsebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel 150 elektronikus vezérlõegységbõl (ECU) és számos fedélzeti kommunikációs sínbõl kell kialakítani egy megbízhatóan mûködõ elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Az így adódó komplexitás uralására alakultak ki különféle szabványok, pl. a megbízható kommunikáció biztosítására a CAN és FlexRay sínek, a valós idejû feladatok futtatására az OSEK operációs rendszer, stb. A vezetõ autógyártók által 2002-ben életre hívott AUTOSAR konzorcium célja az, hogy ezen szakterületi szabványokra építve specifikáljon egy (i) alapvetõ szolgáltatásstruktúrát, amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) egy modellezési nyelvet az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti transzparens kommunikációját lehetõvé tevõ elosztott runtime szolgáltatást (RTE):

- A base software stack magában foglalja az alacsony szintû eszközmeghajtókat (pl. digitális IO, SPI, EEPROM driver, stb.), az ezeket eltakaró absztrakciós rétegeket (pl. memória absztrakciós felület) és az ezekre ültetett magas szintû funkciókat (pl. perzisztens adattárolás).
- A modellezési nyelv lehetõvé teszi, hogy precízen specifikáljuk az adattípusokat, illetve az alkalmazást alkotó komponensek interface-eit és belsõ felépítését.
- Az RTE egy generált glue kód réteg, amely eltakarja az alkalmazáskomponensek elõl, hogy az általuk fogadott vagy küldött információ pontosan hogyan jut el a forrástól a célig, potenciálisan ECU-k közötti kommunikációs buszok igénybevételével. A konzorcium jelentõs hangsúlyt fektet az API-k szabványosítására, de kifejezetten támogatja a versengést az egyes szolgáltatások megvalósításában („Cooperate on standards, compete on implementation"). Az AUTOSAR egy élõ, aktívan fejlesztett szabvány, amelynek a közelmúltban jelent meg a 4.3-as verziója.
- A jelölt feladata egy AUTOSAR Base Software modul megvalósítása egy általa választott mikrovezérlõre a szabvány 4.3 verziójának megfelelõen az alábbiak szerint:

- A szabvány kapcsolódó részeinek megismerése: ismertesse az AUTOSAR rétegzett BSW struktúráján belül a kommunikációért felelõs modulok szerepét, különös tekintettel az SPI protokoll megvalósítására. A témavezetõvel egyeztetve válassza ki az AUTOSAR által specifikált SPI funkciók egy olyan részhalmazát, amely releváns egy autóipari elektronikus vezérlõegység elsõdleges mikrovezérlõje szempontjából.
- Szoftvertervezés és megvalósítás: ismerje meg a választott mikrovezérlõ SPI perifériájának programozási modelljét, majd a szabvány által definiált interface betartásamellett tervezze és valósítsa meg az SPI Handler Driver modult (a fent azonosított funkció részhalmazra korlátozva).
- A szabvány a modulok megvalósítását egy statikus (kézzel írt, minden konfigurációban azonos) és egy dinamikus (konfigurációtól függõ, tipikusan generált) részre bontással javasolja és megadja a konfigurációs adatok modelljét egy osztálydiagrammal. A dinamikus kódrészletek elõállítása a szabványos adatmodellbõl az aktuális megvalósítás (tehát a szakdolgozat feladat) része úgy, hogy az illeszkedjen a statikus részhez és megfeleljen az alkalmazási terület erõforrás-használatra vonatkozó követelményeinek (pl. minimális ROM használat). A megvalósítás tesztelése: az Ön által fejlesztett modul helyességének vizsgálatához

(i) készítsen integrációs környezetet, mely a céleszközön vizsgálja meg a modul mûködését, (ii) futtassa a teszteket, és gyõzõdjön meg arról, hogy megvalósítása megfelel a szabvány által elvártaknak, illetve (iii) szükség esetén javítsa a megvalósítást.

Tanszéki konzulens: Sujbert László docens

Külső konzulensek: Szikszay László (Thyssenkrupp Components Technology Hungary Kft.)

Budapest, 2019. október 13.

……………………

Dr. Dabóczi Tamás

tanszékvezető

HALLGATÓI NYILATKOZAT

Alulírott **Szabó Attila**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzé tegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2019. 12. 19.

...…………………………………………….

Szabó Attila

Összefoglaló

Napjainkban egy személyautó tervezése során sok elektronikus vezérlőegység (ECU) és kommunikációs sín együttműködését kell biztosítani. Egy ilyen rendszer megtervezése egy komoly tervezési kihívást jelent, melynek a kezelésére különböző szabványok alakultak ki, mint a CAN és FlaxRay sínek. Ennek a kihívásnak a megkönnyítésére egyfajta megoldást jelent egy olyan szoftverréteg létrehozása, ami támogatja az alkalmazási szoftver hordozhatóságát, egy modellezési nyelvet biztosít az ECU-kon futó alkalmazási szoftver szabványos leírására, valamint az egyes szoftverkomponensek közötti transzparens kommunikációt. Ezen feladatoknak a specifikációja volt a célja az AUTOSAR konzorciumnak, amely létrehozta az AUTOSAR szabványt.

Dolgozatom célja volt egy olyan SPI handler/driver modul elkészítése volt egy általam választott mikrokontrollerre, ami eleget tesz az AUTOSAR architektúra által támasztott követelményeknek, és képes működni egy ilyen rendszerben.

Jelen szakdolgozatomban először ismertetem az SPI interfész működésével kapcsolatos fogalmakat, és az interfész működését. Ezután az AUTOSAR alapú rendszer általános felépítését mutatom be, melynek a részletezésében kitérek az SPI handler/driver működésére is.

A dolgozatom maradék részében egyrészt bemutatom az általam választott mikrokontroller programozási modelljét. Másrészt az általam elkészített handler/driver felépítése alapján mutatom be a megvalósított funkciókat.

Abstract

Nowadays, in the development of a car, many electronic control units (ECUs) and communication buses have to be interoperable. Designing such a system is a serious design challenge that has been addressed by various standards such as CAN and FlaxRay rails. One solution for this challenge is to create a software layer that supports application software portability, provides a modelling language for standard description of application software running on ECUs, and provides transparent communication between individual software components. The specification of these tasks was the purpose of the AUTOSAR consortium, which created the AUTOSAR standard.

The purpose of my thesis was to create an SPI handler/driver module that meets the requirements of the AUTOSAR architecture and can function in such a system.

In my thesis, I will first describe the concepts of the operation of the SPI interface and the functioning of the interface. I then present the general structure of the AUTOSAR-based system, which includes the SPI handler/driver moduls in detail.

In the rest of my paper, first I present the programming model of the microcontroller of my choice. After that, I present the implemented features based on the structure of the handler/driver I have created.

1 Bevezetés

Dolgozatom témája egy AUTOSAR architektúrán alapuló SPI Handler/Driver megvalósítása, ami az általam kiválasztott mikrokontrolleren lefut, sikeresen továbbítva adatokat az SPI porton keresztül.

Napjainkban a korszerű autókban egyre több elektronikus berendezést találunk. A vevők egyre szélesebb körű igényei, a mind szigorúbb biztonsági és környezetvédelmi követelmények arra ösztönzik az autógyártókat, hogy újabb és újabb megoldásokkal jelenjenek meg a piacon. Az ilyen fejlesztések nagyfokú bonyolultsággal járnak. Ennek a megoldására hozták létre az AUTOSAR architektúrát.

Ez az architektúra az egyes kommunikációs szabványokra épülve (CAN, LIN, FlexRay, SPI) egy olyan rendszert hoz létre, amiben a fejlesztők számára szabványos felépítést határoznak meg az interfészek számára, ezzel egyrészt megkönnyítve az egyes modulok fejlesztését, valamint az autóipari környezetben megírt alkalmazások újrahasznosíthatóságát oldották meg.

Dolgozatom célja, hogy megvalósítsak egy általam választott mikrokontrollerre tervezzek egy SPI Handler/Driver-t, ami megfelel az AUTOSAR szabvány 4.3 verziójának. A programnak egy központi részből áll, amelyhez különböző funkciókat lehet preprocesszor időben engedélyezni.

1.1 Feladat értelmezése

A diplomaterv célja egy AUTOSAR Base Software modul megvalósítása, ami képes egy AUTOSAR. szabvány 4.3 verziójának megfelelő rendszerben működni

A megvalósítására egy STM32F429 Discovery board-ot választottam. A fejlesztése során az SPI interfészhez való hozzáférést az STM32F4 HAL könyvtárával oldottam meg, és a fejlesztéshez az Atollic TrueSTUDIO® programot használtam.

A feladatban említett központi program részlet tervezésének során a modul olyan működését tűztem ki célul, ami képes a következő feladatok elvégzésére:

- Aszinkron és szinkron kommunikációt valósít meg polling és interrupt módban is.
- Az egyes adatelemek állapota lekérdezhető

- hardver és adategységek konfigurálhatósága
- Képes belső és külső tárolók használatára

Az általam választott a programhoz csatolható funkciók:

- Sequence küldés megszakításának engedélyezése
- Default Error Tracer (DET) modul követelményeinek implementálása

2 SPI kommunikációs interfész

2.1 Bevezetés

A Serial peripheral interface (SPI) az egyik legelterjedtebb kommunikációs interfész, amit beágyazott rendszerekben alkalmaznak. Az interfészt a Motorola fejlesztette ki az 1980-as évek közepén, és napjainkra egy, a beágyazott rendszerekben gyakran használt rendszerré vált, melyet főként mikrovezérlők és a kisebb perifériák, például a shift regiszterek, érzékelők, SD-kártyák közötti adatok küldésére alkalmaznak.

SPI egy rövid távú kommunikációra használt szinkron soros kommunikációs interfész, aminek a segítségével az eszközök full-duplex módban kommunikálhatnak. Minden esetben van egy master egység, amely a kommunikáció irányításáért felelős. A master határozza meg a buszon folyó kommunikáció sebességét, adatkeret hosszát, valamint felelős a megfelelő slave egység kiválasztásáért.

2.2 Az interfész felépítése

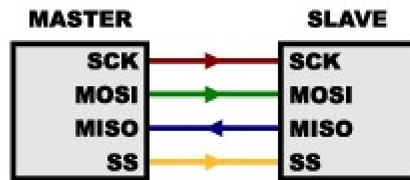
Az SPI egységnek 4 vezetékre van szüksége, melyek a következők:

- Órajel (SCK)
- Slave select (SS)
- Master Output Slave Input (MOSI)
- Master Input Slave Output (MISO)

Az órajel előállítása a master feladata. A két irányú kommunikációhoz 2 darab adatvezeték tartozik, mivel mind a két vezetéket nem lehet adatvezetéknek hívni, ezért a két vezetéknek a MISO, MOSI nevet vezették be, ahol a megnevezés az adott csatlakozási pont kimenetként vagy bemenetként való használatáról ad felvilágosítást az eszköz felhasználása szempontjából. Ennél fogva a MISO kivezetés a master-nél bemenetként, a slave-nél kimenetként működik. Az egyes egységek kiválasztásához szükség van még egy engedélyező jelre, amit slave select-nek (SS) hívnak. Az SS-vonal értéke általában logikai magas értéket vesz fel nyugalmi állapotban, melyet a kiválasztás során logikai alacsony értékre változtatja a master.

2.3 Buszelrendezések

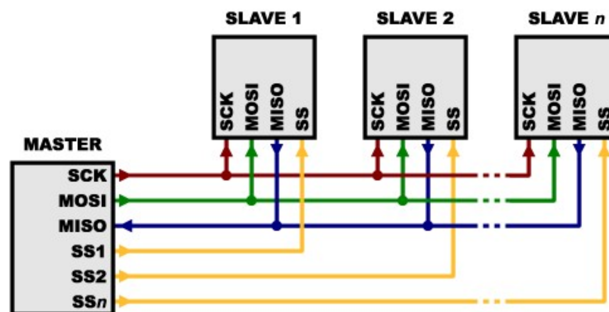
2.3.1 Single slave konfiguráció



2.1 ábra single slave konfiguráció [1]

Egyetlen slave egység során a 2.1 ábrán látható elrendezést valósítjuk meg. Mivel a kommunikációban nem vesz részt több egység, ezért a címzésért felelős SS vezeték bekötése elhagyható.

2.3.2 Párhuzamos konfiguráció



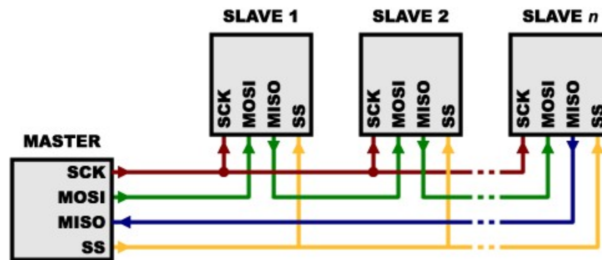
2.2 ábra párhuzamosan kapcsolt slave egységek [1]

A leggyakrabban használt elrendezés során az egyes slave egységek ugyanazt az órajelet kapják, és közös adatvezetéken keresztül kommunikálnak. A különböző slave egységek kiválasztásához a hozzá tartozó engedélyező bemenetet kell használni, ami minden egységhez külön tartozik, amint a 2.2 ábrán látható.

2.3.3 Daisy-chain konfiguráció

Ebben az elrendezésben a slave egységek össze vannak fűzve a 2.3 ábrán látható módon. A láncba fűzés az adatvezetékeken keresztül történik olyan módon, hogy az egyes slave egység MISO kimenete a következő egység MOSI bemenetére csatlakozik, és a master az így kialakult láncot zárja le. Az így kialakult láncban az adatok végig mennek az egységeken, így egyetlen egység címzéséhez nincs szükség külön

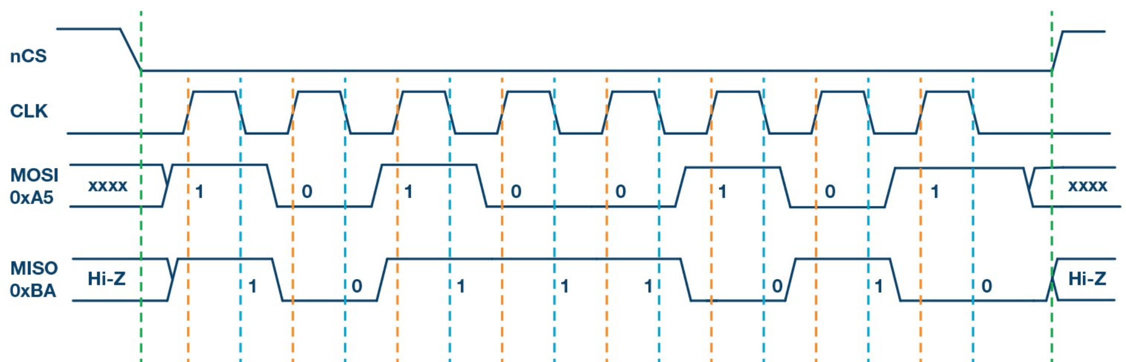
engedélyező jelre, hanem ebben az esetben a késleltetéssel állítjuk be az címzett eszközt. Az adatok megfelelő továbbításáért az egységek közös órajelről működnek.



2.3 Daisy-chain kapcsolás [1]

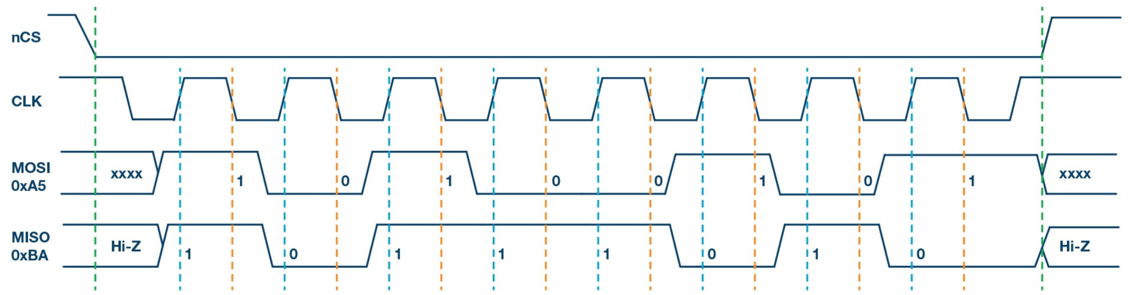
2.4 Adatátvitel leírása

A 2.4 és 2.5 ábrán egy-egy adatátviteli ciklus látható. Az üzenet elküldés az engedélyező jel lefutó élére kezdődik. Az üzenet mindig az elküldendő adat legmagasabb helyiértékű bitjével kezdődik.



2.4 ábra egy adatátviteli ciklus CPOL = 0, NCPHA = 0 beállításokkal [2]

Az adatátvitel során a lehetőségünk van beállítani az órajel nyugalmi állapotban lévő polaritását (CPOL), valamint a mintavételezés fázisát is (NCPHA). A 2.4 ábrán az órajel nyugalmi állapotban logikai 0 állapotban van és a mintavételezés felfutó élre történik, míg a 2.5 ábrán logikai 1 állapotban nyugszik az órajel és lefutó élen mintavételez. Az üzenet megfelelő átviteléhez a két egység beállításainak meg kell egyeznie.



2.5 ábra egy adatátviteli ciklus CPOL = 1, NCPHA = 1 beállításokkal [2]

3 Az AUTOSAR szabvány

3.1 Bevezetés

Az *AUTomotive Open System ARchitecture* (AUTOSAR) egy szabadon elérhető, az autóipari szoftverfejlesztés architektúráját meghatározó szabvány. Vezető autógyártó cégek által 2002-ben kezdődött el a szabvány fejlesztése, és jelenleg is fejlesztik, frissítik, és a classic platformja jelenleg a 4.4-es verziónál tart.

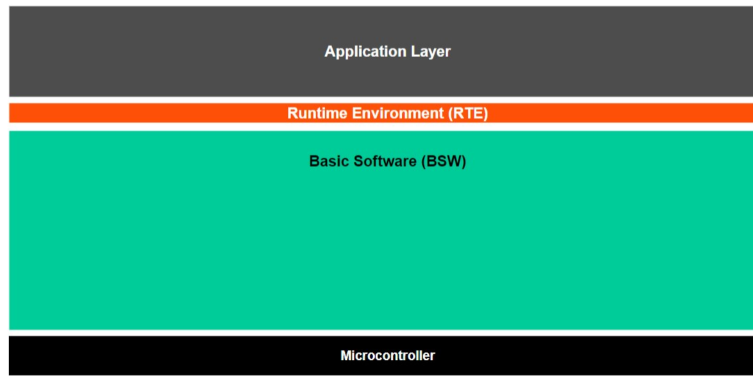
Manapság a személyautókban nagy mennyiségben hódítottak teret az elektronikai megoldások, ennek köszönhetően egy autó gyártása során számos elektronikus vezérlőegység (ECU) és kommunikációs sín együttműködését kell biztosítani. Az egyes eszközök kommunikációjának egységesítésére (CAN, LIN, FlexRay), valamint a komplexitás uralására (OSEK operációs rendszer) különféle szabványok jelentek meg. Az AUTOSAR konzorcium célja volt, hogy ezen szakterületi szabványokra alapulva egy olyan rendszert specifikáljon, melyben az alkalmazások hordozhatóak rendszerek között, egy modellezési nyelvet biztosít az egyes ECU-kon futó szoftver szabványos leírására, valamint ezek mellett biztosítsa az egyes szoftverkomponensek közötti transzparens kommunikációt.

3.2 Szoftver architektúra

3.2.1 Alapvető rétegek

Az AUTOSAR szabvány által meghatározott szoftver architektúra 3 fő részre bontható:

- Application Layer
- Runtime Environment (RTE)
- Basic Software (BSW)



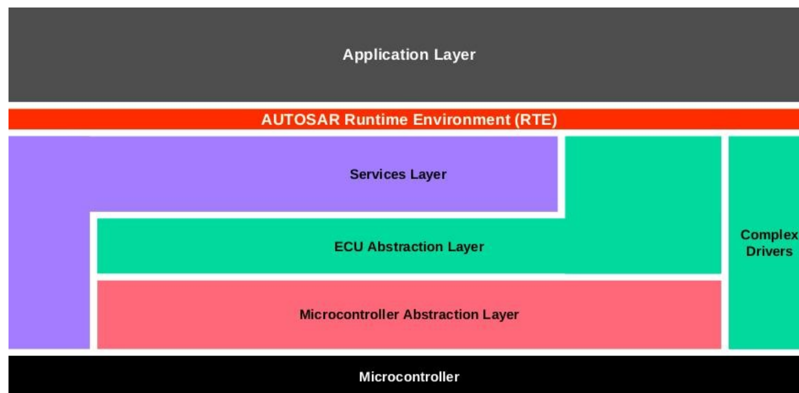
3.1 ábra AUTOSAR szoftver architektúra felépítése [3]

Az *alkalmazási réteg* (Application Layer) tartalmazza az ECU-n futó, az AUTOSAR elvek szerint megalkotott szoftver komponenseket (SWC). A komponensek kommunikációja szabványos portokon keresztül történik, amiknek az összeköttetése egy virtuális hálózaton történik, ami *Virtual Function Bus*-nak (VFB) neveztek el. Ennek a busznak a feladata, hogy az egyes komponensek számára a kommunikáció tényleges megvalósulása rejtve maradjon, így hardverfüggetlen kapcsolatot hozva létre a szoftverek között. A VFB-nek köszönhetően az így megírt alkalmazások nem függenek az ECU tulajdonságaitól.

A futásidejű környezet, a *Runtime Environment* (RTE) a komponensek szemszögéből látott, generált VFB. Az RTE biztosítja a kapcsolatot a szoftver és hardverréteg között, egyrészt elrejti az alkalmazási réteg felől a hardverréteg moduljait, így biztosítva az alkalmazások hordozhatóságát. Az RTE minden ECU-ra külön kerül regenerálásra, mivel minden egység másfajta tulajdonságokkal rendelkezik. Az RTE dönt arról is, hogy mely adatcsomagoknak van szüksége egy másik ECU-val történő kommunikációra.

A Basic Software (BSW) egy szabványosított szoftvercsomag, ami biztosítja a működési feltételeket az alkalmazások futtathatóságához. Ez a réteg további modulokra bonthatóak, melyek között jól definiált interfészeken zajlik a kommunikáció. A BSW további rétegekből tevődik össze.

3.2.2 Basic Software (BSW) réteg



3.2 ábra BSW rétegzett felosztása [3]

A BSW a 3.2 ábrán lévő felosztás szerint további rétegekre lehet bontani, amiknek a feladatai jó elkülöníthetőek.

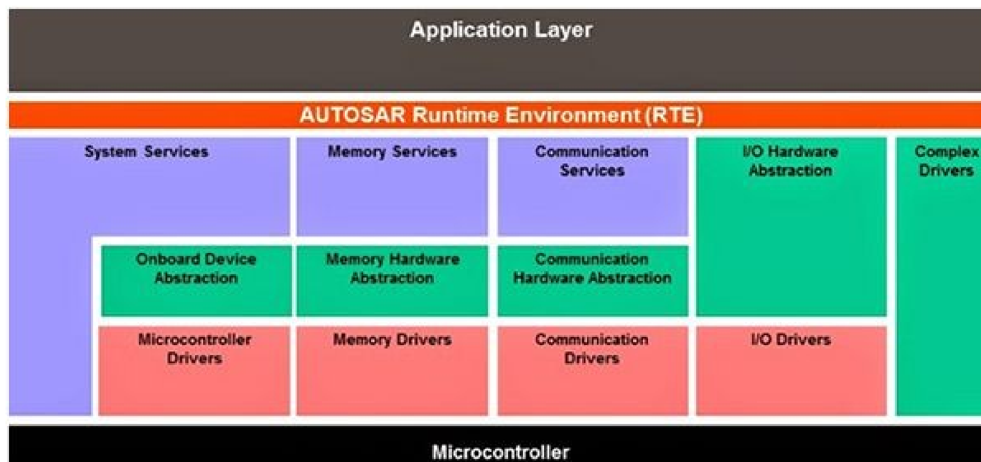
A szolgáltatási réteg (Services Layer), a BSW legfölső rétegeként alapvető szolgáltatásokat nyújt az alkalmazások számára az RTE-n keresztül. Ennek a rétegnek a szolgáltatásai között szerepel a hálózati kommunikáció lebonyolítása és menedzsmentje, memóriakezelés, diagnosztikai szolgáltatások és az ECU állapotmenedzsmentje. Ebben a rétegben helyezkedik el az operációs rendszer is.

Az ECU absztrakciós réteg (ECU Abstraction Layer) biztosítja a fölsőbb rétegek számára a hardverfüggetlen viselkedést. Ez a réteg elérést biztosít a perifériákhoz, a csatlakozási módjuktól függetlenül (külső / belső eszközök).

A mikrokontroller absztrakciós réteg (Microcontroller Abstraction Layer) a BSW legalsó rétege. Ennek a rétegnek hozzáférése van a belső perifériákhoz, és a célja, hogy a felsőbb rétegek számára biztosítja a mikrokontrollerek tulajdonságaitól független működést.

A komplex eszközmeghajtók (complex device drivers) olyan speciális modulok, melyek közvetlenül elérik a hardvert, másrészről közvetlenül kapcsolódnak az alkalmazáshoz is. Ezek a modulok olyan időkritikus, egyedi funkcionalitást valósítanak meg, ami nem érhető el a szabványok meghajtókon keresztül, vagy az időzítések miatt nincs lehetőség a szabványos, rétegezett implementációra.

3.2.3 Funkcionális felosztás



3.3 ábra BSW réteg funkcionális felosztása [3]

A 3.3 ábrán látható módon az egyes rétegek további modulokra bontható az egyes modulok által nyújtott szolgáltatások jellege alapján. A szakdolgozatom alapjául szolgáló SPI Handler Driver a Communication Drivers modul csoporton belül található egyéb kommunikációt megvalósító driverekkel együtt. Ezen driverek a mikrokontroller több kivezetését használják, amiket más modulok nem használhatnak.

3.3 SPI Handler Driver

Ezen meghajtó program feladata, hogy a szabványban megszabtak szerinti funkciókat ellássa és biztosítsa a megfelelő interfészeket a felsőbb rétegek felé. Ez a program felelős az SPI buszhoz szükséges kimenetek kezeléséért. A buszhoz tartozó összes kimenetet a drivernek kell kezelnie, így a kommunikációhoz nélkülözhetetlen engedélyező bemeneteket is a program fogja kezelni. A működéséhez szükséges órajel előállítása nem tartozik a feladatai közé, ezt az MCU modulnak kell elvégeznie.

A szabvány az alábbi 3 szintet definiálja a meghajtóprogram funkcionalitásában a működésük szerint:

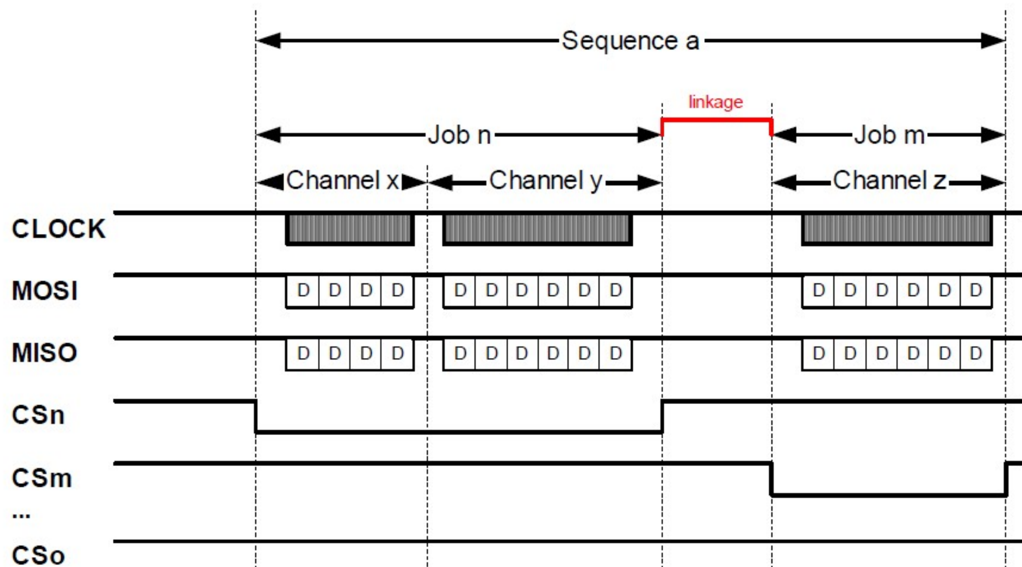
- **LEVEL 0, Egyszerű szinkron SPI Handler/Driver**
- **LEVEL 1, Egyszerű aszinkron SPI Handler/Driver**
- **LEVEL 2, Fejlesztett (szinkron/aszinkron) SPI Handler/ Driver**

3.3.1 Általános működés

Ebben a fejezetben az egyes működési szintektől független működését mutatom be a meghajtóprogramnak.

3.3.1.1 Adategységek

A program buszra kiadott adatokat rendszerezni kell, ennek felosztása Csatornákra (Channel), Munkákra (Job), Szekvenciákra (Sequence) történik. Minden elemnek egyedi azonosítóval kell rendelkeznie.



3.4 ábra példa SPI Sequence küldésére [4]

A Channel a legkisebb egység és egy kiküldött adategységnek felel meg. A definiálásakor a tartalmazott adatok számát, valamint ezen adatoknak forrását és érkezési helyét kell megadni.

A Job egy vagy több Channel-ből épül fel. Ezen adatok küldésekor a rendszer nem szakíthatja meg a küldést egy másik Job küldésével, mivel ez egy elemi adategységnek tekintet elem. Egy Job megadásakor a használt busz tulajdonságait kell megadni, illetve a használt Channel-ek azonosítóit.

A Sequence a szabvány által meghatározott legnagyobb adategység, és egy vagy több Job-ból épül fel. A Sequence-ek küldésénél engedélyezni lehet a konfiguráció során, hogy az adott Sequence kiküldésénél a küldés meg lehet-e szakítani nagyobb prioritású Jobokkal.

3.3.1.2 Bufferek

A mikrokontroller képességeinek jobb kihasználása érdekében kisebb tárolókat ún. buffereket kell a programhoz adnunk, amelyek segítségével kiegyenlíthető a rendszer és az SPI periféria közötti sebességkülönbség. A szabvány szerint a tárolóknak a Channelek-hez kell tartoznia, és tárolás helyétől függően 2 típust különböztet meg:

- **Belső buffer (Internally Buffered Channels, IB)**
- **Külső buffer (Externally Buffered Channels, EB)**

Belső tároló (buffer) során a tárolót a program inicializálása során hozzuk létre és a programon belül kezeljük, a többi program csak meghatározott felületeken keresztül írhatnak bele adatot, vagy kérdezhetik le a megérkezett üzeneteket.

Külső tároló (buffer) esetén a felhasználó adja meg az elküldendő adatok helyzetét. Ebben az esetben az inicializálás után, de még a küldés előtt a felhasználónak meg kell adnia a program számára egy memóriaterületet, ahol a szükséges adatok elhelyezkednek.

3.3.2 Egyszerű szinkron SPI Handler/Driver

Ezen a szinten a program csak szinkron küldéshez kapcsolódó funkciókat valósít meg. Ezen a szinten a csak egy Sequence lehet aktív, így a felhasználó egy küldés elindítása után nem férhet hozzá további Sequence-ek indításához, illetve nem módosíthatja az adatokat.

3.3.3 Egyszerű aszinkron SPI Handler/Driver

Ezen a szinten a program csak aszinkron küldéshez kapcsolódó funkciókat valósít meg. Az aszinkron kommunikáció azt jelenti, hogy a küldési szolgáltatások nincsenek letiltva éppen folyó küldés alatt. Az egyes Job-ok a hozzájuk rendelt prioritás szintjétől függően kerülnek sorra. Az inicializálás során be lehet állítani az egyes Sequence-re, hogy meg lehet-e szakítani azokat, ezek arra vonatkoznak, hogy az egyes Job-ok elvégzése után a program áttérhet-e másik Sequence-re, melyhez tartozó Job-nak nagyobb a prioritása. Az aszinkron működést meg lehet valósítani interruptokkal, vagy pollinggal is.

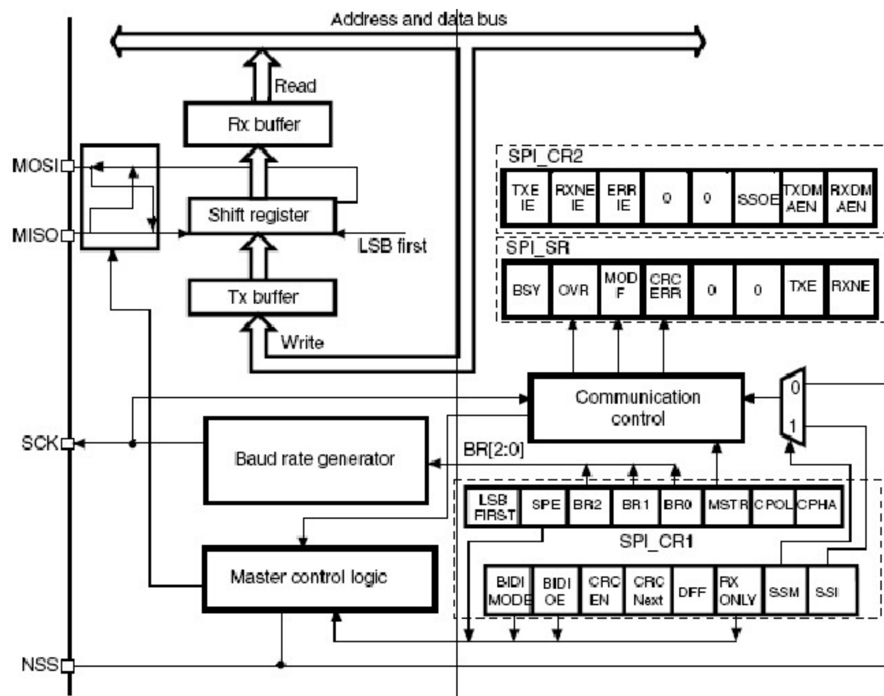
3.3.4 Fejlesztett (szinkron/aszinkron) SPI Handler/ Driver

Ezen a szinten a program kezelhet szinkron és aszinkron küldéshez kapcsolódó funkciókat is. Ez azt jelenti, hogy egy program képes több busznak az egyszeri kezelésére is, egy buszon viszont nem lehet többféle kommunikáció, tehát aszinkron kommunikációt nem lehet megszakítani szinkron küldéssel.

4 A mikrokontroller SPI modul működése

Ebben a fejezetben az általam megvalósított STM32F429 Discovery board SPI perifériájának működését fogom bemutatni. A működés bemutatására a board-hoz tartozó STM32F4 HAL könyvtárat használom. A program megvalósítása során az SPI1-es egységet használtam, mivel ennek a modulnak a használatát nem befolyásolja más a modul megvalósítása során használt hardverelem.

4.1 Felépítés



4.1 ábra stm32f4 típusú mikrokontroller spi perifériája

A 4.1 ábrán látható egy a feladatom során használt mikrokontroller felépítése. Az ábrából látható, hogy az egység 4 kivezetéssel rendelkezik: a két adatvezeték (MISO, MOSI), az órajel kivezetés (SCK) és az engedélyező jel (NSS). A busz működési paramétereit 2 konfigurációs regiszter segítségével állíthatjuk. Az adatvezetékek egy shift regiszterre csatlakoznak, amik egy-egy címezhető bufferen keresztül csatlakoznak az adatvezetékekre. Az SCK által kiadott órajel sebessége, polaritása és a mintavételezés időzítése állítható a CR1 regiszter segítségével. A NSS egy egyszerű engedélyező port, aminek csak a kimenetként, vagy bemenetként való használhatóságát lehet állítani. További engedélyező jelek hozzáadása a rendszerhez

megoldható callback függvényeken keresztül. A CR1 regiszterben a periféria fizikai paramétereit lehet állítani, míg a CR2 az interruptokkal és a DMA-val kapcsolatos beállításokat tartalmazza.

4.2 Inicializálás

Az SPI periféria működéséhez szükséges, hogy a működéséhez szükséges regisztereket beállítsuk. Az inicializálást a HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi) függvény végzi, ami a paraméterként megkapott hspi struktúra Init elemében megadott paraméterek alapján beállítja az SPI konfigurációs regisztereit, a CR1 és CR2 regisztereket. Az Init struktúrában a programom során használt beállítható paraméterek:

- Mode: az egység Master-ként vagy Slave egységként funkcionáljon
- Direction: az egység full-duplex, módban működjön, vagy csak a küldést/fogadást lehessen megvalósítani
- Datasize: az egység képes 8 és 16 bites adatok kiküldésére is az SPI portra
- NSS: Az engedélyező jel kimenetként vagy bemenetként működjön.
- FirstBit: A buszra először az MSB vagy az LSB kerüljön ki
- CLKPolarity: Az órajel nyugalmi állapotát lehet beállítani
- CLKPhase: Felfutó vagy lefutó órajelre történjen a mintavételezés
- BaudratePrescaler: A kiadott órajel generálásához szükséges órajel sebességét lehet beállítani

A fenti beállítások közül a BaudratePrescaler, CLKPhase és CLKPolarity érétkeket futási időben lehet megadni. A Mode Master lesz, mivel az általam írt egység irányítja a kommunikációt, a direction az architektúra által előírt tehát full-duplex. Mivel az adatok megadása 8 bites konténerekben történik, ezért a Datasize-nak 8 bites beállítást használtam. Az NSS pedig kimenetként került inicializálásra. A FirstBit beállítása során MSB-t állítottam be az első kiadott bitnek, mivel ennek konfigurálhatóságát nem építettem bele a program működésébe.

4.3 Adatküldés

Az adatok kiküldésére több függvényt is lehet használni. A program során mivel az SPI egység full-duplex módban működik, ezért a programom során olyan függvényt használtam, ami egy időben végzi el az adatok küldését és fogadását is. Ezenkívül egy nem blokkoló változata is van a függvénynek, amiben az adatok kiírása az adott egységhez tartozó IRQHandler-ben történik. Mivel a küldés során fontos a kiszámítható működés, és nem szeretnénk várni a küldés végrehajtása során, ezért a blokkoló változatát használtam a függvénynek. A küldésre használt függvény így néz ki:

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t
*pTxData, uint8_t *pRxData, uint16_t Size, uint32_t Timeout)
```

A függvény a hspi paraméter által definiált hardver egységen keresztül küldi ki a paraméterben megkapott tömbben lévő adatokat (`pTxData, Size`) és ezzel egy időben képes adatokat is fogadni egy másik tömbbe (`pRxData, Size`). Az egyes kiküldött adatok közötti késleltetést is be lehet állítani a Timeout beállítással, ennek az értékét a használatnál 0-ra állítottam. A küldés indítása után a függvény az egység állapotregiszterét nézi, hogy van- e már fogadott adat, és ekkor végzi el a kiolvasást, és írást a megfelelő regiszterekbe. A küldés során bekövetkező hibát a visszatérési értékében jelzi.

A programom során ezt a függvényt használtam, azonban a modul rendelkezik olyan beállítható funkciókkal, amelyek megvalósításához módosítások szükségesek, ezeket a módosításokat csak az általam készített modul bővítése esetén szükségesek.

4.4 Megszakítások

Az SPI kommunikáció egyes állapotaiban az egység megszakítás kérést küld, amivel jelzi az adott állapot bekövetkeztét. a könyvtár ezen állapotokra való reagálásra callback függvények segítségével ad lehetőséget. Ilyen hozzáférési pontok közé tartozik az egyes küldés vagy fogadás végén meghívódó callback függvények, vagy a hibák észlelése esetén bekövetkező callback. A programom során az aszinkron kommunikáció interrupttal való megvalósítás során használtam callbacket. Az általam használt függvény a `HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)`, ami a full duplex kommunikáció befejeződése után kerül meghívásra.

5 SPI eszközmeghajtó program felépítése, működése

A következő fejezetekben a programom működést fogom bemutatni, a megavalósított függvényeken keresztül. Először a program tervezése során szem előtt tartott szempontokat ismertetem, majd a program által használt adattípusokat, szimbólumokat, struktúrákat fogom bemutatni, ezután az architektúra által előírt standard interfészekhez kapcsolódó függvények működését mutatom be. Végül az általam megvalósított kommunikáció algoritmusát mutatom be, valamint a. Az egyes elemek leírása során ismertetem az adott elemmel kapcsolatos elvárásokat, illetve indoklom, hogy miért ezen megvalósítás mellett döntöttem.

5.1 A program tervezése

A programom tervezése során egy olyan általános modul megalkotása volt az elképzelésem, ami képes az aszinkron és szinkron kommunikációra, képes a belső és a külső tárolók használatára, emellett a szabványos interfészek működőképesek. Ennek elérésére egyrészt definiálnom kellett a szabvány által használt adattípusokat, struktúrákat. Ezek során létre kellett hoznom a konfiguráció során használható struktúra rendszert, amiben a program futása során használható beállítások találhatóak. A szabványos típusokon kívül meg kellett terveznem egy olyan tároló rendszert, ami a futás során szükséges adatok tárolását végzi. Ennek a rendszernek a tervezése során az volt a fő szempont, hogy az egyes függvények lefutását a lehető legjobban segítse, de emellett a lehető legkisebb területet foglalja el a memóriából.

A programomhoz preprocesszor időben lehetőség van olyan programrészletek engedélyezésére, amik a futás alatt a program lassítása mellett valamilyen funkciót adnak a működéshez. Az általam megvalósított funkciók a Sequence küldés megszakításának engedélyezése, illetve a paraméterek beállítását segítő DET modul számára küldött hibajelentések. A program futását segítendő ilyen elemként lehetne megadni a modul működési szintjét, valamint a használható tárolókat is lehetne ilyen módon korlátozni, ezeknek a megvalósítását nem készítettem el.

5.2 Adattípusok

A program működéséhez szükség van bizonyos típusok megadására. Ezek a típusok szolgálhatnak a hordozhatóság elősegítésére, a standard interface által előírt típusok, esetleg a program működését segíthetik.

5.2.1 Szabványos adattípusok és szimbólumok

5.2.1.1 Platform típusok és szimbólumok

Itt a hordozhatóság elősegítésére szolgáló típusokat írom le, amely típusok deklarációja a Platform_Types.h fájlban találhatóak, és a definiálásukra a stdint.h-ban található típusokat használtam.[4]

Itt definiált típusok:

- `uintx`
- `sintx`
- `uintx_least`
- `sintx_least`
- `float32`
- `float64`

Ahol x lehet 8, 16, 32, 64 is.

`bool`: A `bool` változó a logikai értékek leírására szolgál. Ezt a változót a rendszer által máshol is definiálásra kerülhet, ebben az esetben a rendszernek azt a definíciót kell használnia. értéke TRUE vagy FLASE érték lehet, ahol a TRUE értéke 1-nek míg a FALSE értéke 0-nak kell lennie.

5.2.1.2 Standard Típusok és szimbólumok

Itt a program működéséhez elengedhetetlen típusokat és szimbólumokat írom le, amik az Std_Types.h-ban találhatóak. Ezeket a típusokat más modulegységek is használhatják, és a megfelelő működés érdekében külön fájlba kerültek. [4]

- `Std_ReturnType`: Uint8 típusú változó, az általánosan használt visszatérési érték. Az értékével jelezhetjük, hogy történt-e hiba a függvény futása során. További hibák jelzésére a fennmaradó értékek felhasználhatóak, ebben a megvalósításban nem adtam hozzá. Értéke lehet:

E_OK: 0x00 értékű, azt jelezzük vele, hogy minden rendben lement.
E_NOT_OK: 0x01 értékű, azt jelezzük vele, hogy a függvény futása során hiba történt

- `Std_VersionInfoType`: Az adott modul verziójáról ebben a típusban kerül kiadásra a felhasználó számára. Az értékének megadása `STD_GET_VERSION_INFO(_vi,_module)` makróval történik.

Itt kerülnek még definiálásra a különböző állapotok leírására szolgáló szimbólumok. Ezek a szimbólumok a következők:

STD_HIGH: Fizikai 5V vagy 3,3V értékű állapot leírására szolgál, értéke 0x01u
STD_LOW: Fizikai 0V értékű állapot leírására szolgál, értéke 0x00u
STD_ACTIVE: Logikai active állapot leírására szolgál, értéke 0x01u
STD_IDLE: Logikai idle állapot leírására szolgál, értéke 0x00u
STD_ON: Bekapcsolt állapotot jelzésére szolgál, értéke 0x01u
STD_OFF: Kikapcsolt állapotot jelzésére szolgál, értéke 0x01u

5.2.1.3 Spi Típusok és szimbólumok

Ezeket a típusokat és szimbólumokat csak a Spi modul használja, és az spi.h fájlban található. Ezeket a típusokat egyrészt a standard interfészek használják, másrészt a program futása során az egyes elemek állapotának leírására szolgálnak. [5]

`Spi_StatusType`: Az SPI driver állapotának leírására szolgál.

SPI_UNINIT: A driver nincs inicializálva
SPI_IDLE: A driver nem küld el Job-ot
SPI_BUSY: A driver éppen egy Job-ot küld el

`Spi_JobResultType`: Egy Job állapotának leírására használható. Értéke lehet:

SPI_JOB_OK: A Job utolsó küldése sikeresen lezárult
SPI_JOB_PENDING: A driver éppen ezt a Jobot küldi
SPI_JOB_FAILED: A Job utolsó küldése meghiúsult
SPI_JOB_QUEUED: A Job el van fogadva, de még nem került feldolgozásra

- `Spi_SeqResultType`: Egy Sequence állapotának leírására szolgál. Értéke lehet:

SPI_SEQ_OK: A Sequence utolsó küldése sikeresen lezárult
SPI_SEQ_PENDING: A driver éppen ezt a Sequence-t küldi
SPI_SEQ_FAILED: A Sequence utolsó küldése meghiúsult
SPI_SEQ_CANCELED: A Sequence utolsó küldése meg lett szakítva

- `Spi_DataBufferType`: A modulban az adatok megadására, mozgatására és tárolására ezt az adattípust használjuk

- `Spi_NumberOfDataType:` Az átadott adatok számáról ebben a típusban adhatunk át információt.
- `Spi_ChannelType:` Egyedi azonosító egy Channel számára
- `Spi_JobType:` Egyedi azonosító egy Job számára
- `Spi_SequenceType:` Egyedi azonosító egy Sequence számára
- `Spi_HWUnitType:` Egyedi azonosító egy hardver egység számára
- `Spi_AsyncModeType:` Az aszinkron küldés során a küldés mechanizmusának leírására szolgál. Értéke lehet:

  SPI_POLLING_MODE: Az aszinkron küldés pollinggal valósul meg
  SPI_INTERRUPT_MODE: Az aszinkron küldés interruptal valósul meg

  Szimbólumok:

Services ID-k: Ezek az egyes interfészek azonosítására szolgál a szimbólumokat a Default Error Tracer (DET) modul által megszabott követelmények megvalósítása során használjuk, mivel hiba esetén jelezni kell a probléma helyét. Értékük a 5.1 táblázatban található.

SPI_INIT_SERVICE_ID	0x00
SPI_DEINIT_SERVICE_ID	0x01
SPI_WRITEIB_SERVICE_ID	0x02
SPI_ASYNCTRANSMIT_SERVICE_ID	0x03
SPI_READIB_SERVICE_ID	0x04
SPI_SETUPEB_SERVICE_ID	0x05
SPI_GETSTATUS_SERVICE_ID	0x06
SPI_GETJOBRESULT_SERVICE_ID	0x07
SPI_GETSEQUENCERESULT_SERVICE_ID	0x08
SPI_GETVERSIONINFO_SERVICE_ID	0x09
SPI_SYNCTRANSMIT_SERVICE_ID	0x0A
SPI_GETHWUNITSTATUS_SERVICE_ID	0x0B
SPI_CANCEL_SERVICE_ID	0x0C
SPI_SETASYNCMODE_SERVICE_ID	0x0D

5.1 táblázat SPI Driver Service ID-k

Hiba kódok: A hiba jelentéséhez meg kell adnunk a hiba típusát is, ezeknek az értékét a szabvány határozza meg, amik a 5.2 táblázatban olvashatóak.

SPI_E_PARAM_CHANNEL	0x0A
SPI_E_PARAM_JOB	0x0B
SPI_E_PARAM_SEQ	0x0C
SPI_E_PARAM_LENGTH	0x0D
SPI_E_PARAM_UNIT	0x0E
SPI_E_UNINIT	0x1A
SPI_E_SEQ_PENDING	0x2A
SPI_E_SEQ_IN_PROCESS	0x3A
SPI_E_ALREADY_INITIALIZED	0x4A

5.2 táblázat SPI Driver hiba kódjai

5.2.2 Egyéb típusok és szimbólumok:

A szabványosított típusokon kívül az spi.h fájlban lehetőségem volt olyan típusok létrehozására, amik segítségével a hibás adatok megadása elkerülhető. Ezeknek a létrehozására Enumeration típusokat használtam, mivel ennek a típusnak, ha rossz adatot adunk meg, akkor a fordító jelez. Ezek a típusok az alábbiak:

- `Spi_BufferType`: Egy Channel kiírása során a szükséges adatok tárolásának módjának leírására szolgál. Értéke felveheti:

  SPI_EB: Az adott Channel csak külső memóriát használ
  SPI_IB: Az adott Channel csak belső memóriát használ
  SPI_IBEB: Az adott Channel használja a külső és belső memóriát is

- `Spi_EdgeType`: A hardveregység számára ebben a paraméterben adható meg, hogy az adatok kiküldése a kiküldött órajel felfutó vagy lefutó élével legyenek szinkronban. Értéke felveheti:

  SPI_EDGE_LEADING: a küldés az órajel felfutó élére történik
  SPI_EDGE_TRAILING: a küldés az órajel lefutó élére történik

  Szimbólumok:

Egyéb szimbólumokkal adjuk meg az inicializálás és a futás alatt használt tömbök maximális méretét is. Ilyen szimbólumok közé tartozik a belső buffer mérete (`SPI_MAX_IB_SIZE`), Channel-ek, Job-ok, Sequence-ek maximális száma (`SPI_MAX_CHANNEL, SPI_MAX_JOB, SPI_MAX_SEQUENCE`). Itt definiáltam még azon szimbólumokat is, mellyel jelezhető, hogy az adott adategység nem használható. ezek a `CH_NOT_VALID, JOB_NOT_VALID, SEQ_NOT_VALID` értékek, amiknek az értékeik az adott elem Id-jének a legnagyobb értékű eleme.

5.3 Adat struktúrák

5.3.1 Inicializáló struktúra

A rendszer felállítására szolgáló inicializáló struktúrák felépítését a szabvány nem írja elő, de minden a kommunikáció működéséhez szükséges adatot meg kell adni ebben a struktúrában. A struktúra felépítése során az volt az elsődleges szempontom, hogy az egyes adategységhez tartozó konfigurációs struktúrákhoz könnyen hozzá lehessen férni, ezért egy központistruktúrát hoztam létre, és ebben tárolom az összes többi struktúrát, illetve hogy az egyes típusokból hány darabot adtunk a struktúrához. Minden további elem konfigurációjához tartozó struktúra tartalmaz egy azonosító értéket (Id), ezeknek a függvények által címzésre használt értékeket használtam. Ezenkívül fontosnak láttam, hogy csak a megvalósított funkciókat implementáljam ezekbe a struktúrákba.

`Spi_ConfigType`

Ebben a struktúrával inicializáljuk az `Spi_Init()` függvény segítségével. Ebben a struktúrában adjuk meg a további adategységek működéséhez szükséges struktúrákat. Az egyes elemeket a megfelelő adattípusokhoz tartozó tömbökben tárolom (`SpiJobs`, `SpiChannels`, `SpiSequences`) és mellette az adott tömbben tárolt elemek számát (`SpiMaxChannel`, `SpiMaxJob`, `SpiMaxSequence`). Megoldható lehetne úgy is, hogy egy pointert tárolnánk és a létrehozás után hozzáadni a struktúrákat tároló tömböket. Utóbbi esetben a struktúra összesített mérete kisebb lenne, viszont a memóriában nem feltétlen egymás után kerülnének tárolásra. Az én megoldásom akkor előnyösebb, ha a tömb minden elemét feltöltjük, mivel ilyenkor a két megoldás mérete azonos és a felbontatlan állapot előnyt jelent.

`Spi_DeviceConfigType`

Ebben a struktúrában adjuk meg a driver által használt SPI egység beállításait. Mivel a program egyetlen hardveregységet használ, ezért az Id lehagyható, de a későbbi bővíthetőség miatt én meghagytam. Ebben a struktúrában a Job független beállításokat tárolom ebben a struktúrában, amiket a hardveregység támogat. Ezek a beállítások a sebesség és a kiadott órajellel kapcsolatos beállítások. Az egység sebességét a `BaudratePrescaler` paraméterrel állíthatjuk, melynek értéke 2, 4, 8, 16, 32, 64, 128 és 256 lehet. Az órajellel kapcsolatban be lehet állítani a nyugalmi állapotát (`ClockIdleLevel`) és a mintavételezés fázisát (`Edge`). Előbbi értéke a `STD_HIGH`, `STD_LOW`

értékekkel adható meg, míg az utóbbi egy `Spi_EdgeType` változó, tehát `SPI_EDGE_LEADING` vagy `SPI_EDGE_TRAILING` értéke lehet.

`Spi_ChannelConfigType`

Ilyen struktúrákban tárolódnak a Channel-ek beállításai. Azonosítására a `channelID` szolgál. A további tárolt adatok a futás során a kiküldendő adatok tárolásával kapcsolatosak. A `BuffType` a tárolás helyét adja meg, a `DataWidth` a kiküldendő adat méretét adja meg. A `DefaultData-nál` meg az alapesetben kiküldendő adatot adhatjuk meg. A jelenleg kiküldésre használt függvény nem alkalmas bitszintű küldésre, viszont a későbbi megvalósítás esetére meghagytam, hogy a `DataWidth` paraméterben bitszinten lehet megadni a kiküldendő adat hosszát.

`Spi_JobConfigType`

Ilyen struktúrákban tárolódnak a Job-ok beállításai. Azonosítására a `jobID` szolgál. Eltárolom még a hardveregység azonosítóját (`HwUnit`), amit a jelenlegi program nem használ, de későbbi bővíthetőségre meghagytam, a Job prioritását (`JobPriority`), aminek az értéke 0-3 között lehet, valamint a kiküldendő Channel-ek listáját. Emellett még a `SpiJobEndNotification` callback függvény segítségével egy olyan függvényt adhatunk meg, ami a Job elküldése után fog lefutni. A Channel-ek tárolására egy tömböt használok, és a tárolt elemek mennyiségét a `channelNum` paraméterben tárolom. Egy másik felmerült ötletem volt, hogy egy invalid értékkel zárom le a tömböt, ez viszont annál jobban növeli a futási időt minél több elemet tárolunk, ami a tömbösített tárolás előnye.

`Spi_SeqConfigType`

Ilyen struktúrákban tárolódnak a Sequence-ek beállításai. Azonosítására a `SequenceID` szolgál. A struktúrában `interruptEn` segítségével lehet engedélyezni, hogy megszakítható a Squence küldése egy másikkal. A struktúrában tárolásra kerül még a kiküldendő Jobok listája, ami a `Spi_JobConfigType` Channel-ek tárolásához hasonlóan működik. A Jobok tárolása a `jobs` tömben történik és a `jobNum` adja meg a tárolt elemek számát. Emellett még a `SpiSeqEndNotification` callback függvény segítségével egy olyan függvényt adhatunk meg, ami a Sequence elküldése után fog lefutni.

5.3.2 Háttérstruktúrák

A futás során szükségünk van olyan tárolókra, amelyekben egyrészt a futás során használandó adatokat tároljuk, valamint az adott elem állapotát is nyomon tudjuk követni. Ezen struktúrákat a programom megvalósítása során globális változókként hoztam létre, mivel az értékeikre több függvénynek is szüksége van. A következő struktúrák megtervezése során ezeket a szempontokat vettem figyelembe, valamint, hogy az egyes interfészek megvalósítása során a lehető legkevesebb időre legyen szükség.

`Spi_GlobalType`

Ebben a tárolóban egyrészt az inicializáló struktúrára tárolok el egy hivatkozást ügyelve arra, hogy a program ne tudja változtatni az értékeit, úgy, hogy konstans területre mutató pointerként hozom létre. A tárolóban kap még helyet az aszinkron módot jelző `asyncMode`, valamint a modul állapotát jelző `modulStatus`. A `modulStatus` egy `Spi_StatusType` típusú változó, tehát annak értékeit veheti fel. A `interruptEn` azt mondja meg, hogy az éppen kiküldés alatt álló Sequence küldését meg lehet-e szakítani egy nagyobb prioritással rendelkező Job kiküldésével. FALSE értéke esetén a modul nem dolgoz fel újabb küldést, amíg ki nem küldjük a megszakíthatatlan Sequence-t. Ilyen típusú változó az `Spi_Global`,

`Spi_ChannelUnitType`

Ebben a típusban egy Channel kiküldéséhez szükséges adatokat tárolom. `dataLenght` a küldendő adat hosszát adja meg, a tárolásához szükséges `Spi_DataBufferType` -ok száma, ami a konfigurációs struktúrában átadott `DataWidth` paraméterből határozható meg. A kiküldendő adatot tároló IB-t és EB-t megvalósító struktúrákhoz a hozzáférést is itt tárolom. Ilyen típusú változó a `Spi_ChannelUnit` tömb.

`Spi_EbType`

Ebben a típusban tárolom az egyes Channelek-hez tartozó külső tároló paramétereit. A `srcPtr` és a `destPtr` a külső tároló helyére mutató pointerek helyei, amik a `Spi_SetupEB()` függvény hívása során állítódnak be, aminek a során a tároló méretétmegadó `length` paraméter is beállításra kerül. Az `active` paraméter segítségével meg tudjuk mondani, hogy az adott egység beállítása megtörtént-e. A `currentPos` érték a következő a tárolandóból kiolvasandó, illetve írandó érték helyét mondja meg, és az értéke mindig a hozzá kapcsolódó ChannelUnit egység `DataWidth` paraméterével nő.

`Spi_IbType`

Ebben a típusban tárolom az egyes Channelek-hez tartozó belső tárolóban tárolt adatokat. A típus két tömbből áll, amelyeknek a mérete a belső tároló maximális méretével egyezik meg. Mivel a tárolandó adat mérete uint32 és uint8 közötti érték lehet, ezért használhatnánk union típust is a tárolásra, mivel azonban az SPI egység 8 bites tárolóval rendelkezik, ezért célszerűbb egy 4 darab Spi_DataBufferType típusú 8 bites tömb-ben tárolni az adatot, amiből 1,2 vagy 4 helyet foglalunk a tárolás során. Későbbiekben a belső tároló méretét lehet növelni úgy, hogy láncolt listaként hozzuk létre.

`Spi_JobUnitType`

Ebben a típusban egy Job kiküldéséhez szükséges adatokat tárolom. `jobCfgPtr` egy referenciát tárolok az adott elemet konfiguráló struktúrára. Itt tárolom a Job prioritását (jobPriority), valamint a kiküldendő Channel-ek listájára is tárolok hivatkozást (ChannelsPtr, channelNum). Az adott Job státuszáról a `jobResult` paraméter ad információt, ami a `Spi_JobResultType` értékeit veheti fel. Ilyen típusú változó a `Spi_JobUnit` tömb.

`Spi_SeqUnitType`

Ebben a típusban egy Sequence kiküldéséhez szükséges adatokat tárolom. `seqCfgPtr` egy referenciát tárolok az adott elemet konfiguráló struktúrára. A `jobLeft` a még kiküldendő Jobok számát tartalmazza. Ennek az értékét a küldés elfogadásakor állítjuk be és minden egyes a Sequence-hez tartozó Job kiküldésekor csökkentjük az értékét, ha eléri a 0 értéket, akkor a Sequence küldése befejeződött. A Sequence küldés megszakítását az `interruptEn` értéken keresztül engedélyezhető. Ilyen típusú változó a `Spi_SeqUnit` tömb.

5.3.3 A Queue struktúrái

Ezen struktúrák segítségével tárolom a már elfogadott, de még ki nem küldött Jobokat. A `Spi_queueDataType` a konténer, amiben az adatok tárolódnak, míg a `Spi_queueType` a struktúra, amiben az egyes konténereket prioritás szempontjából sorba rendezve tároljuk. A Queue-ban a kiküldendő Job-ok Id-jét és a hozzá tartozó Sequence Id-jét és a Job prioritását tároljuk el.

`Spi_queueDataType`

Ebben a típusban az elküldendő Job Id-jét, és a hozzá tartozó Sequence Id-jét tárolom el. A Job prioritását külön tárolom, mivel az adatok feldolgozásánál arra nincs szükség.

`Spi_queueType`

Ebben a típusban egy tömbben prioritás szerint kerül eltárolására az adat. Mivel nem szeretnék a program megírása során dinamikus memóriakezelést használni, ezért egy fix méretű tömbben tárolom az adatot. A tömb méretének meghatározásához a lehető legrosszabb állapotot vettem figyelembe, minden Job kiküldésre kerül. Tehát a Jobok maximális számával kell megegyeznie. A `maxJob` az aktuálisan tárolt utolsó Job helyét jelzi. Az adatok gyorsabb kiolvasása érdekében a `minJob` értékben az első adat helyét tárolom

`Spi_seqFifo`

Ebben a struktúrában tárolódnak a nem megszakítható Sequence futása alatt érkezett Sequence-ek. A Sequence lefutása után újrafeldolgozásra kerülnek és megfelelően fognak tárolódni az `spi_queue` egységben.

5.4 Felhasználó függvények

Ebben a fejezetben a driverben elsősorban a szabvány által meghatározott interfészek felépítését és működését írom le. Ezen függvények intreface-nek, és feladatát írja le az architektúra. A program tárolóit úgy alakítottam ki, hogy ezeknek a függvényeknek a sebességét a lehető legjobban növeljem. Ezért ezen függvények felépítése nem összetett.[5]

5.4.1 inicializálás és deinicializálás

Az adatok küldéséhez szükség van a tárolók alaphelyzetbe állítására, illetve ha szeretnénk újabb adategységeket, hozzáadni, vagy módosítani szeretnénk futás közben, akkor először fel kell szabadítanunk a használt erőforrásokat. Ezeket végzik el a következő függvények.

```
void Spi_Init(
const Spi_ConfigType* ConfigPtr)
```

Ez a függvény végzi el a tárolók alaphelyzetbe állítását. Paraméterként a már leírt konfigurációs struktúrát kapja meg, és a futás közben használt tárolók adatainak

beállítását végzi el a kapott struktúra alapján. A futás során az `Spi_Global` típusban eltárolja az paraméterként megkapott struktúrát, `asyncMode` -nak alapesetben pollingot állít be, és a helyes beállítás után a modul állapotát SPI_IDLE -re változtatja. A függvény végigmegy az egyes adattípusokon, először elvégzi a Sequence-ek, majd a Job-ok, végül a Channel unit egységek beállítását. A Sequence-ek beállítása után elvégzi a szükséges hardveregységek inicializálását is. A megfelelő egységeket az adott konfigurációs struktúra Id-je alapján találja meg. A nem használt egységek értékeire 0-t állít be. Sequence-ek és Job-ok során az adatok beállítása mellett elvégzi az SPI_JOB_OK és a SPI_SEQ_OK állapotok beállítását, míg Channelek-nél az belső tárolóba feltölti a `DefaultData`-t, illetve elvégzi a `DataWidth`, `dataLenght` konverziót is, a következő makró segítségével:

```
#define SPI_GET_BUFFERSIZE(_width)  ((_width / 8 )+( _width % 8))
```

```
Std_ReturnType Spi_DeInit(
void)
```

A deinicializáló függvény nem kap paramétereket a bemenetére, és a feladata a modul alaphelyzetbe állítása. A megfelelő futás után E_OK-kal kell visszatérnie, ha a módosítás nem végezhető el E_NOT_OK-kal kell visszatérnie. A függvény egyrészt az SPI egység deinicializálását végzi, valamint a `Spi_Global initRun paraméterét FALSE`-ra állítja és a `configPtr_`pointerben lévő hivatkozást a struktúrára törli. Ha az SPI egység még küldést végez, a művelet nem hajtható végre.

5.4.2 Adatkezelő függvények

Az adatok kiküldéséhez szükség van a regiszterekhez való hozzáféréshez, mivel a külső tároló esetén a felhasználó adja át a hozzáférést, ezért ennek az írásáról a programnak nem kell gondoskodni, az belső tároló esetén viszont a programnak megfelelő interfészt kell biztosítania kiküldendő adatok megadásához, illetve a megérkezett adatokhoz is hozzáférést kell biztosítani. Ezeknek a megoldását biztosítja a következő függvények.

```
Std_ReturnType Spi_WriteIB(
 Spi_ChannelType Channel,
 const Spi_DataBufferType* DataBufferPtr)
```

Ez a függvény a megadott Channel-hez (Channel) tartozó ChannelUnit egységben található belső tárolóba írja a DataBufferPtr segítségével megkapott adatot.

A megfelelő futás után E_OK-kal kell visszatérnie, ha a módosítás nem került elfogadásra E_NOT_OK-kal kell visszatérnie. Az interfésznek nem gondoskodik az tárolóban lévő adat felülírásáról a függvény hívása során.

```
Std_ReturnType Spi_ReadIB(
 Spi_ChannelType Channel,
 Spi_DataBufferType* DataBufferPointer)
```

Ez a függvény a megadott Channel-hez (`Channel`) tartozó ChannelUnit egységben található belső tárolóból olvassa ki a `DataBufferPointer` által a megkapott adatot. A megfelelő futás után E_OK-kal kell visszatérnie, ha a módosítás nem került elfogadásra E_NOT_OK-kal kell visszatérnie.

```
Std_ReturnType Spi_SetupEB(
 Spi_ChannelType Channel,
 const Spi_DataBufferType* SrcDataBufferPtr,
 Spi_DataBufferType* DesDataBufferPtr,
 Spi_NumberOfDataType Length)
```

Ez a függvény a megadott Channel-hez (Channel) tartozó ChannelUnit egységben található külső tároló beállítását végzi el a kapott paraméterek segítségével. A megfelelő futás után E_OK-kal kell visszatérnie, ha a módosítás nem került elfogadásra E_NOT_OK-kal kell visszatérnie. A működése során az interfész megkeresi a megfelelő egységet és a megfelelő ChannelUnit egység Eb struktúrájában állítja be a megfelelő paramétereket. `SrcDataBufferPtr-t`, ami a kiküldendő adatok forrására hivatkozik, a megfelelő `Eb srcPtr` paraméterjébe tárolja, `DesDataBufferPtr-t`, ami az érkező adatok helyére mutat, a megfelelő `Eb destPtr` paraméterébe tárolja, és a tárolóban lévő adatok számát megadó `length` paraméter, a megfelelő `Eb length` paraméterének adja meg az értéket. A paraméterek helyes beállítása után aktiválja a tárolót az Eb `active` paraméter TRUE értékűvé változtatásával.

5.4.3 Státusz lekérdező függvények

Mivel az adatok küldése nem minden esetben engedélyezett, vagy a felhasználó szeretné megtudni, hogy az adott adat továbbításra került-e, ezért a szabvány definiál erre használható interfészeket is. Ezen interfészek a megfelelő egység állapotáról ad felvilágosítást a hozzá társított típuson keresztül.

```
Spi_StatusType Spi_GetStatus(
 void)
```

A függvény a modul státuszáról ad információt. Az `Spi_Global` változó `modulStatus` paraméterével egyezik meg a visszatérési értéke.

```
Spi_JobResultType Spi_GetJobResult(
 Spi_JobType Job)
```

A paraméterként megkapott Job állapotával tér vissza, amit a `Spi_JobUnit` megfelelő elemének `jobResult` paraméterével tér vissza a függvény.

```
Spi_SeqResultType Spi_GetSequenceResult(
 Spi_SequenceType Sequence)
```

A paraméterként megkapott Sequence állapotával tér vissza, amit a `Spi_SeqUnit` megfelelő elemének `seqResult` paraméterével tér vissza a függvény.

```
Spi_StatusType Spi_GetHWUnitStatus(
 Spi_HWUnitType HWUnit)
```

Mivel a modul nem használ egynél több SPI egységet, ezért a hardveregység állapota megegyezik a modul állapotával, tehát az `Spi_Global` változó `modulStatus` paraméterével egyezik meg a visszatérési értéke.

```
void Spi_GetVersionInfo(
 Std_VersionInfoType* versioninfo)
```

Ez a függvény a paraméterként megkapott `versioninfo` pointer által meghatározott struktúrába adja ki vissza a modul ilyen jellegű információit. Ennek a megoldásához a `versioninfo` leírásánál említett makrót használja, ami a következő képpen működik:

```
#define STD_GET_VERSION_INFO(_vi,_module) \
	if(_vi != NULL) {\
		((_vi)->VersionID =  _module ## _VERSION_ID);\
		((_vi)->ModuleID = _module ## _MODULE_ID);\
		((_vi)->sw_major_version = _module ## _SW_MAJOR_VERSION);\
		((_vi)->sw_minor_version =  _module ## _SW_MINOR_VERSION);\
		((_vi)->sw_patch_version =  _module ## _SW_PATCH_VERSION);\
	}
```

A `_vi` paraméterében egy `Std_VersionInfoType` típusú változót adunk meg a _modulban pedig az adott modul nevét, esetünkben SPI. A fordítás során a preprocesszor összeilleszti és behelyettesíti az értékeket az egyes helyeken.

5.4.4 Küldést megvalósító függvények

A modul képes szinkron és aszinkron küldésre is, valamint az aszinkron küldés megtörténhet polling vagy interruptok segítségével is, ezeknek a szolgáltatásokhoz tartozik egy-egy interfész, amiket a fejezet ezen részén ismertetek.

```
Std_ReturnType Spi_SyncTransmit(
 Spi_SequenceType Sequence)
```

Ez az függvény `Sequence` paraméterben megkapott azonosítójú egység kiírásának az elindítását végzi szinkron módban. A függvény a `spi_writeSequence(Sequence, SPI_SYNCRONUS_TRANSMIT)` hívásával indítja el a küldést Ha elfogadásra került a küldés, akkor E_OK, ha valamilyen oknál fogva a küldését nem lehet végrehajtani, akkor E_NOT_OK értékkel fog visszatérni a függvény.

```
Std_ReturnType Spi_AsyncTransmit(
 Spi_SequenceType Sequence)
```

Ez az függvény `Sequence` paraméterben megkapott azonosítójú egység kiírásának az elindítását végzi aszinkron módban A függvény a `spi_writeSequence(Sequence, SPI_ASYNCRONUS_TRANSMIT)` hívásával indítja el a küldést Ha elfogadásra került a küldés, akkor E_OK, ha valamilyen oknál fogva a küldését nem lehet végrehajtani, akkor E_NOT_OK értékkel fog visszatérni a függvény.

```
Std_ReturnType Spi_SetAsyncMode(
 Spi_AsyncModeType Mode)
```

Ez az függvény a modul aszinkron küldésének a módját változtatja meg, és a kapcsolódó interruptok kezelésével is foglalkozik. Ha a modul küldést végez, akkor a szükséges módosításokat nem tudja elvégezni. Interrupt módba lépés esetén a kapcsolódó interruptokat engedélyezi, illetve tiltja a megfelelő parancsokkal. Ha a módosítás sikeresen végbement, akkor E_OK, ha valamilyen oknál fogva módosítást nem lehet végrehajtani, akkor E_NOT_OK értékkel fog visszatérni a függvény.

```
void Spi_Cancel(
 Spi_SequenceType Sequence)
```

Ez az függvény `Sequence` paraméterben megkapott azonosítójú egység küldésének felszámolását végzi el, és nincs visszatérési értéke. A megvalósítása során a `queue_removeSequence(Sequence)` függvény hívásával eltávolítja az `spi_queue`-ban tárolt összes az adott Sequence-hez köthető Job-ot, mivel az éppen folyó Job kiküldését nem lehet megszakítani, ezért annak állapotát nem figyelem. A sikeres művelet elvégzése után a Sequence állapotát SPI_SEQ_CANCELLED-re állítja.

5.5 Privát függvények

Ezeknek a függvényeknek a hívása más modulok számára nem elérhető, a felépítésüket és a működésüket én terveztem.

5.5.1 Kiírással kapcsolatos függvények

```
static void spi_writeSequence(Spi_SequenceType seqIndex,
                              Spi_TrasmitTypeType transmit)
```

Ez a függvény a `Spi_SyncTransmit()` és `Spi_AsyncTransmit()` függvényekből hívódik meg. A paraméterként megadott Sequence küldésének a feldolgozását végzi. A függvény először elhelyezi a Sequence elvégzéséhez szükséges Jobok-at a Queue-ban, majd amennyiben szinkron kommunikáció történik, akkor a küldés végrehajtását is elvégzi. Amennyiben a megszakítást tiltjuk, a `suspendFifo`-ban tároljuk, de a megfelelő állapotokat változtatja. A Jobok Queue-ba rakása az alábbi kódrészlet alapján történik:

```
SPI_SET_SEQRESULT(seqIndex, SPI_SEQ_PENDING);
// The Sequence sending is accepted

uint8 globalPriotity = 0;
bool interruptEn = Spi_SeqUnit[seqIndex].interruptEn

if(interruptEn == FALSE || transmit == SPI_SYNCRONUS_TRANSMIT)
{
      Spi_Global.interruptEn = FALSE;
      Spi_JobType index = Spi_SeqUnit[seqIndex].jobsPtr[i];
      globalPriotity = Spi_JobUnit[index].jobPriority;
}

for(uint8 i = 0; i < Spi_SeqUnit[seqIndex].jobNum; i++)
{
      Spi_JobType index = Spi_SeqUnit[seqIndex].jobsPtr[i];

      uint8 queuePriority = globalPriotity;

      if(interruptEn == TRUE)
      {
            queuePriority = Spi_JobUnit[index].jobPriority;
      }

      queue_addJob(index, queuePriority, seqIndex);

      SET_JOBRESULT(index, SPI_JOB_QUEUED);
}

Spi_SeqUnit[seqIndex].jobLeft = Spi_SeqUnit[seqIndex].jobNum;
```

A kódrészletben először a Sequence állapotát állítja SPI_SEQ_PENDING-re, majd egy for ciklus segítségével végigmegy a `seqIndex` paraméterben megkapott

azonosítójú Sequence-hez tartozó kiküldendő Jobok listáján, aminek során eltárolja a Queue-ban az adott Job-ot a hozzá tartozó prioritással és a seqIndex-xel. Az eltárolás után a Job állapotát SPI_JOB_QUEUED állapotúra változtatja. Miután eltárolta a Jobok-at beállítja a jobLeft paramétert. Amennyiben a Sequence nem engedélyezi a megszakítást vagy szinkron módban vagyunk a Spi_Global.interruptEn paraméternek FALSE értéket ad. Ebben az esetben az eltárolt Joboknak ugyan azt a prioritás értéket adja, ami az első elem prioritása.

Szinkron esetben azért itt végzem el a küldést, mivel a feldolgozásnak és a kiküldésnek szinkronban kell lennie a végrehajtás során. A küldés folyamán annyiszor hívja meg az spi_sendSequence() függvényt, amíg ki nem küldi az összes tárolt Jobot, vagyis amíg a Sequence állapota nem lesz SPI_SEQ_OK, aminek az állapotát a meghívott függvény állít. Aszinkron esetben a küldés feldolgozása után indítja el a timer egységet, aminek a segítségével az adatok kiküldése megtörténik. amennyiben a függvény hívásakor a Spi_Global.interruptEn paraméter értéke FALSE, vagyis a küldés alatt álló Sequence-t küldését nem lehet megszakítani, akkor a Sequence-t a suspendFifo egységben tárolja, ami mellett a Sequence és a Job állapotát állítja.

```
static void spi_sendSequence()
```

Ez a függvény végzi el a Queueban tárolt adatok kiküldését. A függvény nem kap paramétert a bemenetére, hanem a spi_queue változóból olvassa ki a következő Jobot a queue_getJob() függvény segítségével. Ezután a spi_writeJob() hívásával elvégzi a Job kiírását a hardveregységen keresztül. A küldés sikeres elvégzése után eggyel csökkenti a Sequenceunit egység jobLeft paraméterét, így ciklusszervezést végezve a Sequence küldésén. Amennyiben kiküldtük a Sequence-hez tartozó összes Jobot, akkor egyrészt SPI_SEQ_OK állapotot állít be a Sequence-nek, valamint amennyiben rendelkezik SpiSeqEndNotification-nel, akkor végrehajtja azt a függvényt. Amennyiben a küldött függvény feldolgozását nem lehetett megszakítani, akkor a futás alatt összegyűlt Sequencek újrafeldolgozását is elvégzi a spi_writeSequence() segítségével, és a Spi_Global.interruptEn-t TRUE érétkűre változtatja. Ennek a függvénynek a létrehozásakor a célom volt egy olyan függvény létrehozása, aminek a hívását minden küldési módban meg lehet tenni, így egységesítve a küldés folyamamtát.

```
static Spi_JobResultType spi_writeJob( Spi_queueDataType queueData)
```

Ez a függvény végzi el az adott Job kiadását az SPI portra. A függvény először a `queueData` paraméterben megkapott azonosítójú Sequence-hez tartozó kiküldendő Jobok listáján megy végig és küldi ki a `HAL_SPI_TransmitReceive()` függvény segítségével. A küldés egy ciklusa alatt a küldés mellett a küldendő adat kinyerését végzi el a megfelelő tárolóegységből, valamint az esetlegesen beérkezett adatokat is eltárolja a megfelelő helyre. Ib és Eb esetén a ChannelUnit egység megfelelő egységéből olvassa ki, tárolja el az adatot. Amennyiben mindkét tárolóegységet használjuk, akkor olvasás az Ib-ből olvassa ki az adatot és az Eb-ből átküldi az IB-be a következnek kiadandó elemet, írás során az utolsó adat kivételével az Eb-ben tárolja A Job kiküldése után amennyiben tartozik hozzá `SpiJobEndNotification`, végrehajtja azt a függvényt. A függvény a futása alatt a Job állapotát SPI_JOB_PENDING-re, a modul állapotát SPI_BUSY-re változtatja. A küldés során bekövetkező hiba esetén a Job állapotát SPI_JOB_FAILED-re állítja, amit a visszatérési értékén keresztül jelez az őt meghívó modulnak.

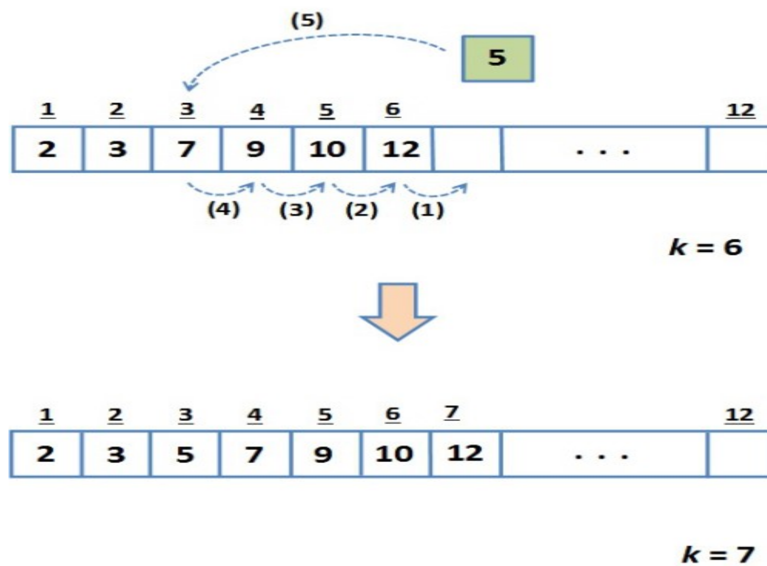
```
HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)
```

Ez a függvény a `HAL_SPI_IRQHandler()`-ből hívódik meg az SPI egység küldésének a végén, és a `HWready` paraméter értékét állítja be TRUE értékűre, amennyiben van még a Queue-ban elem. Ennek a paraméternek a TRUE értékre állításával lehet megmondani a rendszernek, hogy el lehet indítani a következő küldést, melyet a következő callback függvény végez el.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

Ez a függvény a `HAL_TIM_IRQHandler`()-ből hívódik meg és egyrész polling módban megnézi, hogy egy Job kiírás alatt van-e, tehát van-e SPI_JOB_PENDING állapotú Job, ebben az esetben a `HWready` TRUE értékű lesz. Másrészt `HWready` TRUE értéke esetén elindítja a küldést a `spi_sendSequence()` hívásával és `HWready`-t FALSE értékűre változtatja. A küldés végrehajtása alatt letiltom a timer által generált interruptokat, és amennyiben van még kiküldendő Job újraindítom a timer interruptokat.

5.5.2 Queue-val kapcsolatos függvények



5.3 ábra Prioritásos sor működése

A `queue` egy `Spi_queueType` típusú globális változó és egyfajta prioritásos sorként működik, vagyis az adatok berakása és kivétele nem feltétlenül ugyanabban a sorrendben történik, hanem egy tárolt paraméter szerint, az ún. prioritás szerint. A prioritásos sor működése a 4.3 ábrán látható, az adatok egy megadott érték, a prioritás, szerint tárolódnak, és kérésre a megszabott érték, az ábrán a legkisebb érték, kerül kiadásra. Az általam készített típusban `Spi_queueDataType_`típusban történik az adatok tárolása, erre azért volt szükség, mivel az adatok kiadásakor mind a két benne tárolt értékre szükség van, és így a tárolt adatok könnyebben visszaadhatók. A tárolt adatok az adott Job-hoz `JobId`-ből és a hozzá tartozó `SequenceId`-ből áll. A `SequenceId-t` azért tárolom, hogy a Sequence törlése esetén a művelet gyorsan elvégezhető legyen. A Queue kezelésére a következő függvényeket készítettem el. A kiolvasás gyorsabb végrehajtása érdekében olvasás során nem törlök elemeket, hanem újabb adatok hozzáadásánál teszem ezt meg

```
static bool queue_getJob(Spi_queueDataType * queueDataPtr)
```

Ez a függvény a paraméterként megkapott `queueDataPtr`-ben adja vissza az `spi_queue` soron következő elemét. A függvény nem törli ki a kiadott adatot az `spi_queue`-ból, a gyors végrehajtás miatt. A paramétert azért pointer segítségével adjuk vissza mivel, ha visszatérési értékben adnánk vissza, akkor az fordítófüggő viselkedéshez vezetne, ami nem felelne meg az használt architektúra előírásainak. A

függvény sikeres olvasás esetén TRUE-val, ha nincs mivel visszatérni, akkor FALSE-al tér vissza.

```
static void queue_addJob(Spi_JobType jobData,
                         uint8 PriorityData,
                         Spi_SequenceType SequenceId)
```

Ez a függvény a paraméterként megkapott adatokat hozzáadja az `spi_queue`-hoz. A függvény úgy adja hozzá az adatot, hogy megkeresi azt a helyet, ahol a tárolt Job prioritása már kisebb, mint a tárolandóé, és arra a helyre, ha nem talál ilyen akkor az tárolt adatokat követő helyre illeszti be az adatot. Sikeres tárolás esetén TRUE-val, ha már tároljuk az adott Job-ot, akkor FALSE-al tér vissza. Amennyiben a függvény kezdetén a minJob nem 0, akkor a addig mozgatja az elemeket, amíg ez nem teljesül

```
static uint8 queueSize()
```

Ezt a függvényt azért hoztam létre, hogy meg lehessen nézni az `spi_queue` tartalmaz-e még elemet. A visszatérési értéke a `spi_queue` `maxJob` értéke.

```
static void queue_removeSequence(Spi_SequenceType seqData)
```

Ez a függvény kitörli az `spi_queue`-ból az összes a `seqData` paraméterként kapott SequeceId-hoz tartozó Jobot. Ezt a függvényt a `Spi_Cancel()` függvény hívja meg.

Amennyiben az éppen küldendő Sequence küldése nem megszakítható állapotban történik, akkor a végrehajtás alatt beérkező Sequence-ek tárolásra kerülnek a `suspendFifo` változóban, így megakadályozva azt, hogy megszakítsák az éppen folyó küldést. Ez azonban még nem kezd semmit az éppen a Queue-ban lévő Jobok-kal, ennek a megoldását még nem oldottam meg. A benne tárolt adatok a Sequence kiküldése után kerülnek feldolgozásra. A változóval az alábbi függvények segítségével interaktálhatunk:

```
static void SeqFifo_addSeq(Spi_SequenceType SeqData)
```

Ez a függvény adja hozzá a fifo-hoz a paraméterként megkapott `SeqData`-t.

```
static Spi_SequenceType SeqFifo_getSeq()
```

Ez a függvény adja vissza a fifo első elemét, valamint a tárolóban lévő elemeket előrelépteti egyel. Mivel a `Spi_SequenceType` lényegében uint8 típusú, ezért nyugodtan lehet a függvény visszatérési értéke.

```
static uint8 suspendFifoSize()
```

Ezt a függvényt azért hoztam létre, hogy meg lehessen nézni a fifo tartalmaz-e még elemet. A visszatérési értéke a fifo `maxSeq` értéke.

5.5.3 Hardverrel kapcsolatos függvények

```
static void spiDevice_Init(Spi_DeviceConfigType device)
```

Ez a függvény végzi el az SPI, valamint a pollinghoz használt Timer egység inicializációját. Ez a függvény a paraméterként megkapott config struktúra alapján állítja be az SPI egység regisztereit. A beállítások elvégzéséhez egy `SPI_HandleTypeDef` illetve `TIM_HandleTypeDef` típusú egységeket használtam. Ezek a típusok a HAL könyvtár része A struktúrák beállítására egyrészt a device paramétereit használom, másrészt az architektúra által megszabottak szerint masternek állítom be az egységet és full-duplex módban működik. A timer egység beállítása során ugyanazt a prescalert használom, mint az SPI esetében, illetve egy periódusnak az SPI üzenet kiküldésének idejét állítottam be, 16 értékűre, mivel ennél gyakrabban nincs szükség az adatok olvasására. Az adatok beállítása után `HAL_SPI_Init(&hspi1)` függvény segítségével beállítom az SPI egység megfelelő regisztereit. illetve a `HAL_TIM_Base_Init(&htim2)`, `HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig)`, `HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig)` függvények segítségével a timernél is elvégzem azokat.

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t *pRxData, uint16_t Size, uint32_t Timeout)
```

Ez a függvény végzi az adatok kiírását az SPI portra. Ez a függvény a HAL könyvtár része. A függvény a paraméterként megkapott tömbben lévő adatokat küldi ki és ezzel egy időben az adatok fogadására is képes, ha nem sikerült az adatok küldése, akkor a visszatérési értékben jelzi.

```
__HAL_SPI_ENABLE_IT(&hspi1, (SPI_IT_TXE | SPI_IT_RXNE));
__HAL_SPI_DISABLE_IT(&hspi1, (SPI_IT_TXE | SPI_IT_RXNE));
```

Ennek a két makró segítségével lehet tiltani, illetve engedélyezni az SPI egységhez tartozó interruptokat, amik segítségével interrupt módban jelzem a port rendelkezésre állását. A `Spi_SetAsyncMode()` függvényben használom őket.

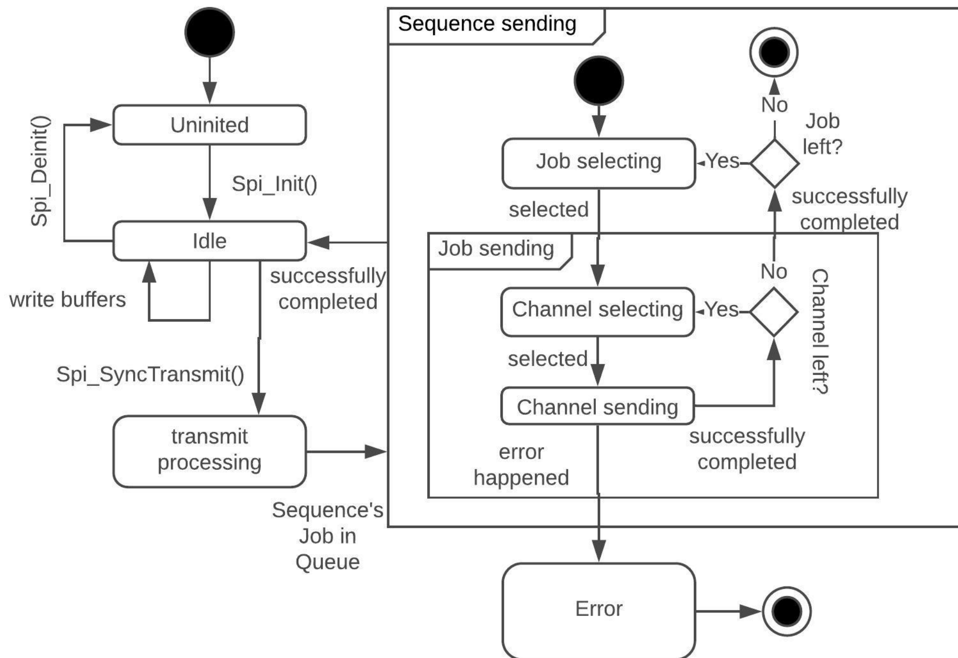
```
HAL_TIM_Base_Start_IT(&htim2); HAL_TIM_Base_Stop_IT(&htim2);
```

Ezek a függvények segítségével indítom el, illetve állítom le a timer által generált interruptokat. Az indítást a `Spi_writeSequence()` függvény végzi, míg a

leállítást a `HAL_TIM_PeriodElapsedCallback()` callback függvény alatt történik. A küldés végrehajtása alatt letiltom az órajel interruptjait.

5.6 Küldés állapotdiagramja

5.6.1 Szinkron működés



5.4 ábra Szinkron küldés működési diagramja

A 5.4 ábrán látható diagram alapján mutatom be a szinkron működést. A modul az indítás után Uninited állapotba kerül, amiből csak az Spi_Init() függvény hívható, és aminek a hatására Idle állapotba kerül, ahonnan az Spi_Deinit() hívásával kerülhet vissza. Az előbbi állapotban a modul státusza SPI_UNINIT, míg az utóbbi esetében SPI_IDLE, és az egyes adategységek alap állapotba kerülnek, tehát a sikeres küldés állapotát veszik fel (SPI_JOB_OK, SPI_SEQ_OK). Ebben az állapotban van lehetősége a felhasználónak, hogy beállítsa a kiküldendő adatokat az egyes Channelekhez. Erre a feladatra az Ib-t és Eb-t beállító függvények használhatóak. `Spi_SyncTransmit()` függvény hívásával a felhasználó elindíthatja a szinkron küldést, aminek a hatására a transmit processing küldése állapotba kerül, amiből akkor kerül át a következő állapotba, amikor a Sequence-hez tartozó összes Job-ot elhelyezte a Queue-ban, azaz elvégezte a Sequence feldolgozását.

5.6.1.1 Sequence feldolgozása

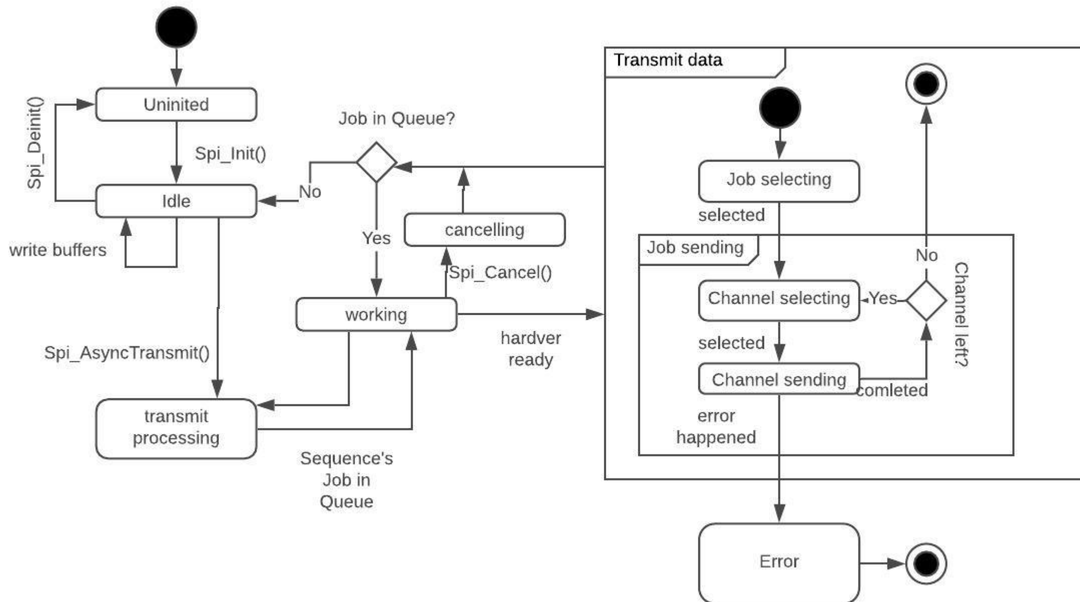
A Sequence feldolgozását a `spi_writeSequence()` függvény végzi el. A feldolgozás során először kiolvassa a megfelelő a `Spi_SeqUnit` egységből a kiküldendő Jobok listáját, majd a Jobokat egyesével hozzáadja a `spi_queue` elemhez. A hozzáadás során mivel a tárolóegység prioritás szerint rendezi a benne tárolt elemeket, így beállítva a későbbi kiolvasás sorrendjét. A műveletek elvégzése során állítja az egyes egységek állapotát is, így a Sequence állapotát SPI_SEQ_PENDING-re a Jobok feldolgozása után az állapotukat SPI_JOB_QUEUED-re változtatja.

5.6.1.2 Sequence küldése

A kiküldendő Job-ok feldolgozása után már csak ki kell küldenünk a hardveregység számára a megfelelő Channelek adatait. A kiküldés megkezdése a `spi_sendSequence()` hívásával kezdődik, ami a `spi_writeSequence()`, függvényből történik. A függvény először kivesz egy Job-ot az `spi_queue` elemből, majd elkezdi a kapott Job kiküldését az SPI port felé az `spi_writeJob()` függvény segítségével. Ennek a függvénynek a futása alatt az adott Job állapota SPI_JOB_PENDING, míg a modul állapota SPI_BUSY lesz. Sikeres küldés esetén a Job állapotát SPI_JOB_OK-ra, míg meghibásodás esetén SPI_JOB_FAILED-re változtatja a függvény.

Az adatok kiküldése a függvény leírásánál ismertetettek szerint megy végbe, aminek a során az adatok a `HAL_SPI_TransmitReceive()` függvény által kerülnek kiküldésre. Visszatérve az `spi_sendSequence()` függvénybe először, amennyiben nem sikerült a Job kiküldése, akkor a Sequence állapotát SPI_SEQ_FAILED-re változtatjuk, majd ezt jelezzük a Diagnostic Event Manager (DEM) modul felé, aminek a működését még nem implementáltam. Sikeres küldés esetén a függvény ismertetésénél leírtak szerint, először elvégzi a ciklusszervezést, majd amennyiben az adott Sequence-hez tartozó utolsó Job-ot küldjük ki, akkor a Sequence állapotát SPI_SEQ_OK-ra változtatja. A küldés elvégzése után az `Spi_SyncTransmit()` E_OK-kal tér vissza.

5.6.2 Aszinkron működés



5.5 ábra Aszinkron küldés működési diagramja

A 5.5 ábrán látható diagram alapján mutatom be egy küldés lebonyolítását. Egy Sequence kiküldése a hardveregységen keresztül itt is két lépésben történik, a szinkron működéshez hasonlóan. Ebben az esetben azonban a feldolgozás végző függvény nem hívja meg a kiírást végző függvényt. A kiírás feldolgozását a `Spi_AsyncTransmit()` függvény hívásával indítható el, ami a kiírás elvégzése alatt is meghívható. A kiírás végrehajtásának időzítése történhet pollinggal, vagy interrupttal.

Interrupttal történő jelzés esetén a hardveregység, az `HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)` függvény meghívásán keresztül jelzi, hogy az SPI egység kiküldte az adatokat. A callback függvény a helyes meghívás esetén a `HWready` érték TRUE értékre állításával jelzi, hogy el lehet végezni a következő küldést. Polling estében a timer egység a beállított időegységeként létrejövő `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` callback függvény nézi meg, hogy az egység állapota SPI_IDLE vagy SPI_BUSY, és SPI_IDLE állapotban a `HWready` érétken keresztül fog jelezni.

Amennyiben el lehet kezdeni a küldést, akkor az előbb említett callback függvényben kerül meghívásra a `sendSequence()`, a szinkron működésnél leírtak szerint kiküldve az adatokat. A szinkron működéssel ellentétben a küldés elvégzése alatt további Sequencek kiírását lehet elindítani. ezek amennyiben a kiküldendő Sequence

megszakítható, akkor a Jobjai az `Spi_queue`-ban tárolódnak, ha ez nem teljesül, akkor az `Spi_suspendFifo` egységben. Az utóbbi esetben a Sequence kiküldése után hozzáadódnak az `Spi_queue`-hoz. A `Spi_queue`-ban lévő összes Job kiküldése esetén a timer egység leállításra kerül, és csak újabb küldés feldolgozásánál indul újra.

5.7 Megvalósított funkciók

5.7.1 Sequence megszakíthatóságának engedélyezése

Az spi.h-ban található INTERRUPTIBLE_SEQ_ALLOWED szimbólumnak STD_ON érték adásával lehet engedélyezni. A modul kikapcsolása esetén az egyes Sequencek küldését nem lehet más Sequence küldésével megszakítani, ezért az összes Job-ot a `spi_writeSequence()` függvény során az ott meghatározott `globalPriority` értékkel tárolja el, így a Queue működését leegyszerűsíti és lényegében egy FIFO ként működik. Amennyiben engedélyezzük a funkciót a program során adhatunk meg olyan Sequence-t, ami megszakítható. Ebben az esetben a tárolás során a hozzá tartozó Job-ok a prioritásuk szerint kerülnek eltárolásra.

5.7.2 DET követelmények

Default Error Tracer egy modul, ami a rendszer beállítása során, a hibátlan működés beállításának érdekében, a hibákat jelenteni kell. A hibák jelzésére ez a függvények szolgálnak:

`Std_ReturnType Det_ReportError( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )`

`Std_ReturnType Det_ReportRuntimeError( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )`

A két megegyező paraméterekkel rendelkeznek, azonban másfajta hibák jelzésére szolgálnak. A függvények a paraméterekként kapott adatok alapján lejelenti a hibát, az első 3 paraméter a hiba helyét mondja meg a következők szerint:

- `ModuleId:` az SPI modul azonosítására szolgál a SPI_MODULE_ID tartalmazza
- `InstanceId: Ez` a felhasználó azonosítására szolgál, mivel a programom nem alkalmas több felhasználó azonosítására, ezért az értéke 0 lesz.

- `ApiId: Megmondja,` hogy a hiba melyik felhasználói függvényben történt, ez az egyes függvények ServicesID-val egyezik. Ezekről a szimbólumoknál már írtam.
- `ErrorId:` Ennek az érétkével lehet jelezni a hiba típusát.

5.7.2.1 Error ID

Ezekkel az értékekkel lehet a hiba típusát jelezni. Az ehhez szükséges szimbólumokat már definiáltam, de most leírom, hogy melyik szimbólum milyen hiba jelzésére szolgál:

- SPI_E_PARAM_CHANNEL: A Channel adatainak rossz megadásának jelzésére megadására szolgál, egyrészt az inicializáció során, ha rosszul adjuk meg a ChannelConfig struktúrában az adatokat kell ezt az Id-t kiküldeni, valamint amikor egy függvény számára rossz Id-t adunk át, ilyen az adatkezelő függvényeknél lehetséges.
- SPI_E_PARAM_JOB: A Job adatainak rossz megadásának jelzésére szolgál, egyrészt az inicializáció során, ha rosszul adjuk meg a JobConfig struktúrában az adatokat kell ezt az Id-t kiküldeni, valamint amikor egy függvény számára rossz Id-t adunk át, ilyen a `Spi_GetJobResult()`-nál fordulhat elő.
- SPI_E_PARAM_SEQ: A Sequence adatainak rossz megadásának jelzésére szolgál, egyrészt az inicializáció során, ha rosszul adjuk meg a SequenceConfig struktúrában az adatokat kell ezt az Id-t kiküldeni, valamint amikor egy függvény számára rossz Id-t adunk át, ilyen a küldést megvalósító függvényeknél lehetséges, illetve a `Spi_GetSequenceResult()` függvény hívásakor fordulhat elő
- SPI_E_PARAM_LENGTH: az inicializáció során az adat hosszának rossz megadása során kerül kiküldésre
- SPI_E_PARAM_UNIT: A HWUnit Id-jének rossz megadása esetén kerül kiküldésre.
- SPI_E_UNINIT: Ha a modul többi függvényét úgy próbáljuk meghívni, hogy nem inicializáltuk a modult, akkor ezt az Id-t kell küldeni.
- SPI_E_SEQ_PENDING: Aszinkron küldés feldolgozásánál, ha a kiküldendő Sequence már kiküldés alatt áll, vagy a hozzá tartozó egyik Job egy olya

Sequence-hez tartozik, ami kiküldés alatt áll, akkor ezt a runtime errort kell jelenteni.

- SPI_E_SEQ_IN_PROCESS: Szinkron küldés feldolgozásánál, ha a kiküldendő Sequence már kiküldés alatt áll, akkor ezt a runtime errort kell jelenteni.
- SPI_E_ALREADY_INITIALIZED: Ha az inicializáló függvény akkor szeretnénk meghívni, amikor a modul már inicializált állapotban van, akkor ezt az Id-t kell küldeni.

A követelmények implementálása során az ellenőrzéseket a SPI_DEV_ERROR_DETECT szimbólumnak STD_ON értéket adva preprocesszor időben ki lehet kapcsolni, hogy a rendszer szerű működés során a működést gyorsítani lehesse, mivel ebben az esetben az egyes beállításokból nem fog hiba adódni.

5.7.2.2 Megvalósítás

A funkció működése az spi.h-ban található SPI_DEV_ERROR_DETECT szimbólum STD_ON érték adásával kapcsolható be. A megvalósítás során az egyszerűbb implementálás érdekében preprocesszor makrókat használtam. Ennek működéséhez egy makrót definiáltam.

```
#define SPI_VALIDATE(_exp,_api,_err , _returnData, _returnerr) \
        if( !(_exp) ) { \
          Det_ReportError(SPI_MODULE_ID,0,_api,_err); \
          _returnData = _returnerr;\
        }
```

Ez a makró a development hibák felderítésére és jelentésére használható. A makró az `_exp` paraméterben megkapott kifejezést kiértékeli, és amennyiben nem teljesül egyrészt jelenti a hibát a DET modulnak, másrészt a függvény visszatérési értékét is módosítja. Annak érdekében, hogy a függvény csak egy helyen térjen vissza az egyes függvényeknél létrehoztam egy returnData változót, aminek alapértelmezetten a helyes futás esetén kiadandó értéket adtam. Ennek az érétkét változtatja meg a makró, amit a `_returnData` paraméterben adok át, a hibát jelző értékkel együtt (`_returnerr`). Az `_api` az apiId, míg a `_err` az errorId, amiknek a segítségével a hiba jelenthető.

A futás során keletkező hibák jelzését nem makróval oldottam meg, mivel ezek csak az aszinkron, szinkron küldést indító függvényekben találhatóak. Szinkron küldésnél csak azt nézi, hogy az adott Sequence küldés alatt áll-e, míg aszinkron küldésnél azt is nézi, hogy a Sequencehez tartozó valamelyik Job fel van-e már

dolgozva. Amennyiben hibát észlel a `Det_ReportRuntimeError()` függvény segítségével jelenti és a függvénynek is jelzi a hibát.

6 A feladat összegzése, tanulságai

Az általam megírt modul elkészülése után a programot tesztelésnek vetettem alá, aminek megfelelt. A tesztelés során egy konfigurációs struktúrát állítottam be, amiben több Sequence, Job, Channel is megtalálható volt. Az így beállított rendszerben ezután kiküldtem a modul egyes módjaiban küldtem ki adatokat, amit egy másik mikrokontroller segítségével ellenőriztem. Ennek az ellenőrzésnek megfelelt, így sikeresnek tartom az elkészített munkámat.

A félévi munka során rengeteg tapasztalatot szereztem az autóiparban használt AUTOSAR architektúráról, valamint az SPI egység működéséről is nagyban elősegítette szakmai fejlődésemet. A program tervezése során további ismereteket szereztem az autóiparban használható programtervezési, felépítési elvekkel kapcsolatban. Ennek a szoftvernek a tervezése során megszerzett ismereteim nagyban megkönnyítik egy hasonló jellegű feladat elvégzését.

A program elvárásaimnak megfelelően működik, azonban adódhat olyan helyzet, amiben ez a program nem elég hatékony. Több felhasználó kezelésére a megoldásom nem alkalmas. Későbbiek folyamán elsősorban optimalizálnám a működését, illetve maradtak olyan az architektúrában lévő funkciók, amiket a programom nem lát el, pl.: DMA kezelés.

Irodalomjegyzék

[1] Sparkfun*: Serial Peripheral Interface (SPI)*
https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all
Letöltés ideje: 2019.10.25

[2] Analoge Dialogue: *Introduction to SPI Interface*
https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html
Letöltés ideje: 2019.10.25

[3] AUTOSAR Consortium: *Layered Software Architecture, 4.3.0*
https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
Letöltés ideje: 2019.10.25

[4] AUTOSAR Consortium: *Specification of Platform Types, 4.3.0*
https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_PlatformTypes.pdf
Letöltés ideje: 2019.10.25

[5] AUTOSAR Consortium: *Specification of Standard Types, 4.3.0*
https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_StandardTypes.pdf
Letöltés ideje: 2019.10.25

[6] AUTOSAR Consortium: *Specification of SPI Handler / Driver, 4.3.0*
https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SPIHandlerDriver.pdf
Letöltés ideje: 2019.10.25

[7] STMicroelectronics*: STM32f4 Discovery board user manual*
https://www.st.com/content/ccc/resource/technical/document/user_manual/6b/25/05/23/a9/45/4d/6a/DM00093903.pdf/files/DM00093903.pdf/jcr:content/translations/en.DM00093903.pdf
Letöltés ideje: 2019.10.25

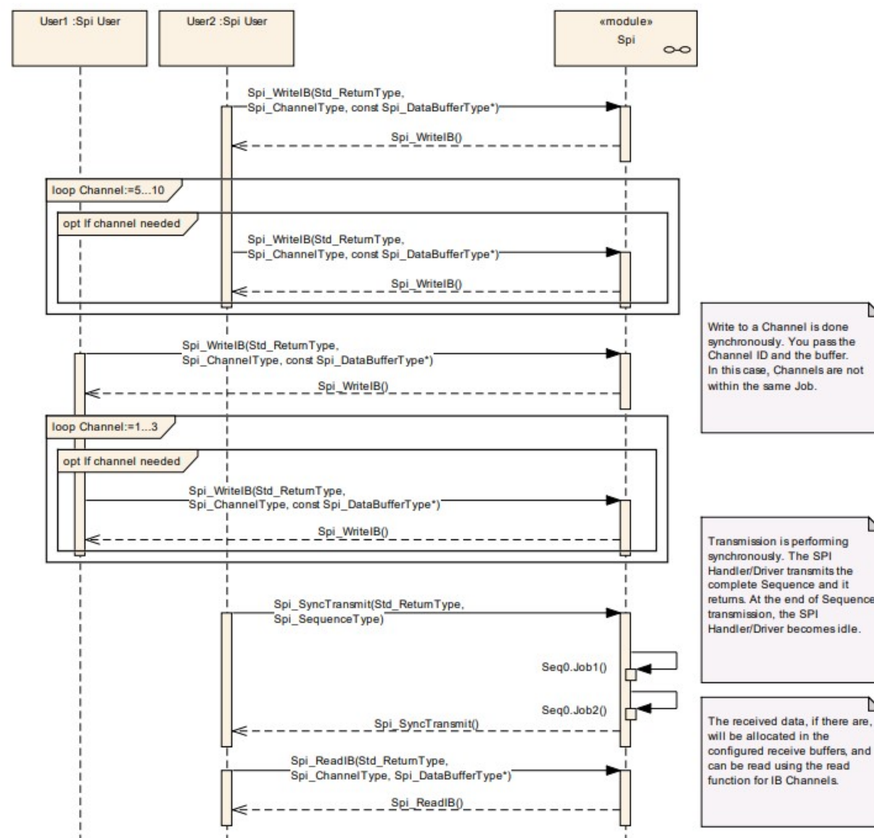
[8] Wikipédia: *Priority queue ismertető*
https://en.wikipedia.org/wiki/Priority_queue
Letöltés ideje: 2019.10.25

[9] STMicroelectronics: *Description of STM32F4 HAL and LL drivers*
https://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf
Letöltés ideje: 2019.10.25

Függelék

1. Függelék: Szinkron kommunikációt leíró diagram 1 Sequence, 2 Job, sok Channel belső tárolókkal

Sequence	Job		Channel
	Name	Priority	
ID0	ID1	High	ID0…ID3
	ID2	Low	ID4…ID10



2. függelék: Aszinkron kommunikációt leíró diagram 1 Sequence, 1 Job, 2 Channel külső tárolókkal

Sequence	***Job***	***Channel***
ID0	ID1	ID2
		ID3