Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

Kővári Balázs

GRAFIKUS FÜGGVÉNYKÖNYVTÁR FEJLESZTÉSE BEÁGYAZOTT RENDSZEREN

Konzulens:

Molnár Károly, Balogh László

Budapest, 2011

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Kővári Balázs**, 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(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. 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, 2011. május 11.

…….

Kővári Balázs

Összefoglaló

Napjainkban a folyadékkristályos kijelzővel és érintőképernyővel rendelkező beágyazott rendszerek egyre elterjedtebbek. A hagyományos parancssori interfészektől azonban jelentősen eltér a grafikus felhasználói interfészek kezelése. Indokolt tehát egy könnyen portolható, egyszerű elemekből álló fejlesztői felület, azaz egy grafikus függvénykönyvtár létrehozása, amely illeszkedik a beágyazott rendszerek sajátosságaihoz. A szakdolgozatom során ennek a feladatnak a tervezésével, megvalósításával és tesztelésével foglalkoztam. Áttanulmányoztam a szoftvertervezés folyamatát és általános módszereit, és ez alapján kiválasztottam egy jól illeszkedő fejlesztési modellt. A modell lépésein keresztül megterveztem a grafikus függvénykönyvtár funkcióit, interfészeit és belső struktúráját, majd implementáltam a kiírt feladatot. A szoftverfolyamat fontos része az elkészült platformfüggetlen szoftver működésének ellenőrzése, ezért két különböző tesztkörnyezetet valósítottam meg: egy PC-n futó alkalmazást, amellyel a függvénykönyvtár belső viselkedését vizsgáltam, és egy digitális jelfeldolgozó processzorral rendelkező beágyazott rendszert. Az utóbbi környezetet az Analog Devices ADSP-BF527 EZ-KIT Lite fejlesztői kártyáján építettem ki. A DSP-s kártya használatához megvalósítottam a rendszerhez tartozó hardverfüggő részeket vezérlő alacsonyszintű szoftvert is. Az LCD modul és az érintőképernyő eszközillesztőjét a Visual DSP Kernel valósidejű operációs rendszer segítségével készítettem el, majd egy iránymérő alkalmazást illesztettem rendszerhez. A jelfeldolgozási algoritmus paramétereinek beállításához, és az eredmények kijelzéséhez a fejlesztői kártya érintőképernyős LCD paneljén kialakított felhasználói felületet készítettem el a grafikus függvénykönyvtár segítségével.

Abstract

Embedded systems with liquid crystal displays and touchscreens are becoming more and more frequent nowadays. Using graphical user interface is significantly different from the traditional command line interface. Therefore it is reasonable to create an easily portable and modular developing interface: a graphical library which is adapted to the specific attributes of embedded systems. Planning and testing a solution for this task is the main subject of this thesis. Based on the study of software design's general processes and methods I chose an appropriate model for this development. Features, interfaces and the internal operation of the graphical library were designed and implemented following the steps of the model. Testing a software is one of the most important part of the development. For this purpose two different test environments were built: an application running on a PC which is capable of monitoring internal behavior of the software and an embedded system realized on a digital signal processor. The latter environment was created on Analog Devices' ADSP-BF527 EZ-KIT Lite evaluation board. Low-level software components of the evaluation kit were also implemeneted in order to test the graphical library. Device drivers of the LCD module and the touchscreen controller were developed and then added to the software enviroment running on the Visual DSP Kernel real-time operating system. As an example a user interface was designed for direction finding applications. This graphical interface handles input actions of the user to change parameters of the data processing by touching the screen and results of the algorithm are also displayed.

Bevezetés

Az 1970-es évek elején jelentek meg az első 4, majd 8 bites processzorok, amelyek a mai számítógépek és mikrovezérlők közös őseinek tekinthetők. A mikrovezérlők fejlődése azóta is töretlen. A processzorok számítási kapacitásának és memóriájuk méretének növekedése mellett perifériakészletük is nagy változásokon ment keresztül napjainkig. A beágyazott rendszerek hagyományos interfészei mellett az ember-gép kapcsolat egyik legfontosabb eszközeinek, a kijelzők alkalmazásának igénye is egyre növekszik. A grafikus felhasználói interfészeknek számos fajtája van: kezdetben leginkább LED-ek segítségével jelenítettek meg információkat, később elterjedtek a különböző típusú szegmenses kijelzők. A folyamatosan fejlődő gyártástechnológiák lehetővé teszik olcsó és nagy sebességű processzorok előállítását, valamint egyre bonyolultabb grafikus interfészek használatát. Mára általánossá vált a színes, folyadékkristályos (LCD = *Liquid Crystal Display*) kijelzőpanelek, és az azokhoz illesztett érintőképernyő alkalmazása. A kijelzők használatát jelentősen megkönnyítheti fejlesztői szempontból egy grafikus függvénykönyvtár, amely segítségével egyszerűen vezérelhetjük a megjelenítést. A piacon több ilyen megoldás is létezik, de a legtöbb esetben nem ingyenesen használható szoftverekről van szó [1] [2]. Indokolt tehát egy szabadon felhasználható, egyszerű elemekből álló és könnyen portolható grafikus függvénykönyvtár kifejlesztése, illeszkedve a beágyazott rendszerek speciális igényeihez.

A *Diplomatervezés* című tantárgy során ennek megoldásával foglalkoztam. Első lépésben áttekintettem a szoftverfejlesztés folyamatának legfontosabb szempontjait és a rendszertervezés általános modelljeit. A módszerek vizsgálata után a feladathoz választott modell alapján megterveztem és implementáltam a grafikus függvénykönyvtárt. A megoldás működésének ellenőrzésére tesztkörnyezeteket építettem ki, melyek segítségével megvizsgáltam a szoftver viselkedését különböző platformokon, a tervezés során megfogalmazott tesztesetek alapján. A tesztelés eredményeinek értékelése után az elkészült grafikus függvénykönyvtárat egy rádió iránymérést megvalósító alkalmazáshoz illesztettem, és az így kialakított rendszeren további, magasabb szintű teszteket valósítottam meg. Utolsó lépésként összefoglaltam a grafikus függvénykönyvtár kifejlesztése előtt megfogalmazott feladatok eredményeit.

1. Szoftvertervezési módszerek

A szoftvertervezése a szoftvertermékek minden aspektusát érinti a rendszer-specifikáció korai szakaszaitól kezdve a rendszerkarbantartáson át egészen a rendszer bevezetéséig és karbantartásáig. Egy szoftvertervezési módszer nem más, mint a szoftverfejlesztés strukturált megközelítése, amelynek célja, hogy elősegítse jó minőségű szoftverek költséghatékony előállítását. Számos módszert dolgoztak ki az 1970-es évektől kezdődően, a különféle módszerek különféle területeken alkalmazhatóak, minden folyamatra ideális módszer nem létezik.

Érdekes kérdés, hogy miként definiálhatjuk a „jó szoftver" kifejezést. A szoftvereknek az általuk megvalósított szolgáltatásokon túl számos egyéb kapcsolódó tulajdonságaik is vannak, amelyek ugyan nincsenek közvetlen kapcsolatban a szoftver által elvégzett tevékenységekkel, de tükrözik a termék minőségét. Ezek a gyakran nemfunkcionális tulajdonságoknak nevezett jellemzők sokkal inkább a szoftver működés alatti viselkedésére, a szerkezetére, a forráskód szervezettségére és a kapcsolódó dokumentációra vonatkoznak. Ezek alapján a következő fő tulajdonságokról beszélhetünk, melyek meghatározzák egy szoftver minőségét:

- **Karbantarthatóság**

  A szoftvert oly módon kell elkészíteni, hogy fokozatosan követni tudja a megváltozott igényeket, könnyen bővíthetőnek kell lennie, hogy megfeleljen az új elvárásoknak

- **Üzembiztonság**

  Valójában számos tulajdonság összessége, mely magába foglalja a megbízhatóságot, a biztonságosságot és a védelmet. Az üzembiztos szoftver rendszerösszeomlás esetén sem okozhat fizikai vagy gazdasági károkat.

- **Hatékonyság**

  A szoftver nem pazarolja a rendszer erőforrásait, például a processzoridőt vagy a memóriát. A hatékonyság magába foglalja az algoritmusok számítási idejét, a memória kihasználtságát.

- **Használhatóság**

  A szoftvernek megfelelő felhasználói interfésszel és ehhez kapcsolódó dokumentációval kell rendelkeznie.

A szoftver tervezésének folyamata a tevékenységek és a kapcsolódó eredmények halmaza, mely végeredményben egy szoftverterméket állít elő. Általánosságban a következő alapvető tevékenységeket különböztethetjük meg, amelyek minden szoftverfolyamatban közösek, és ezekkel a teljes folyamat lefedhető:

- **Szoftverspecifikáció**

  A szoftver működése és az erre vonatkozó megszorítások, követelmények definiálása.
- **Szoftverfejlesztés**

  A specifikáció szerint a szoftver megtervezése és implementációja.
- **Szoftvervalidáció**

  A követelmények alapján az elkészült szoftver validációja, a felhasználók által elvárt működés biztosítása.
- **Szoftverevolúció**

  A szoftver karbantartása és a felhasználók igényeinek megváltozása esetén a termék továbbfejlesztése.

A feladatok sorrendje első ránézésre egyértelműnek tűnik, azonban a legtöbb esetben nem ilyen egyszerű a helyzet. Megfontolandó kérdés, hogy lehet-e az egyes munkafázisokat párhuzamosan végezni vagy más sorrendben ütemezni. A felmerülő kérdésekre a fejlesztési folyamatának modellezése adhat válaszokat, mely a szoftverfolyamat egy bizonyos nézőpontból adódó egyszerűsített leírása.

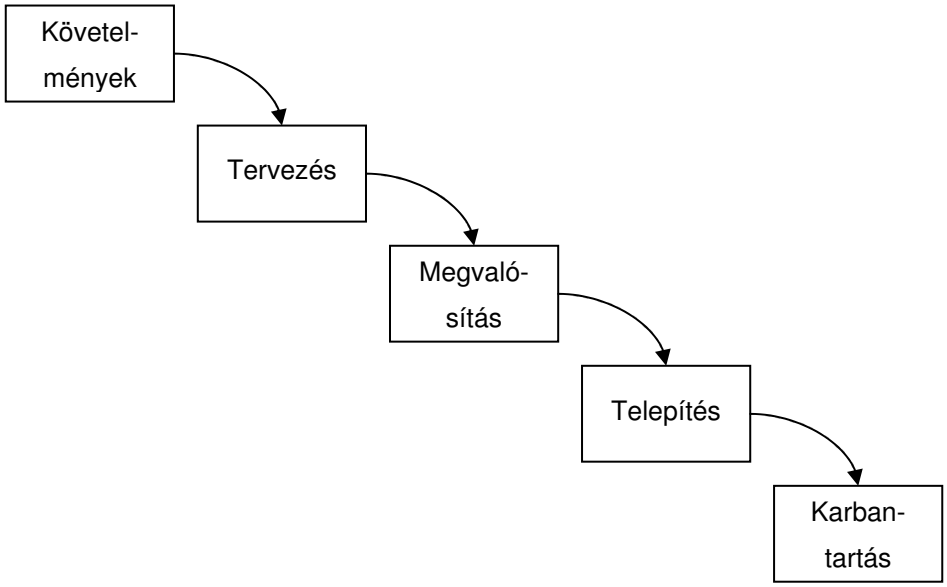
1.1. Szoftvertervezési modellek

Ahogy ezt az előző fejezetben említettük, a szoftverfolyamat modellje a folyamat absztrakt reprezentációja. Minden egyes folyamatmodell különböző, speciális perspektívából mutat be egy folyamatot, tehát csak részleges információkat közöl magáról a folyamatról. A következő alfejezetekben néhány általános modellt mutatunk be a szakirodalom alapján, architekturális nézőpontból szemlélve őket [3] [4]. A

választott szoftvertervezési módszert ezután külön fejezetben, részletesebben ismertetjük.

1.1.1. Vízesés modell

A szoftverfejlesztés folyamatának első publikált modellje. Az egyik fázis szekvenciálisan követi a másikat, és mivel az egyes fázisok lépcsősen kapcsolódnak egymáshoz, a modell a vízesés modellként vált ismerté (1.1. ábra).



1.1. ábra – Vízesés modell

A modell szakaszai a következő fejlesztési tevékenységekre képezhetők le:

- **Követelmények elemzése és meghatározása**
  A termék szolgáltatásainak, megszorításainak és céljainak definiálása felhasználói konzultációk alapján. A követelmények részletes kifejtése, a rendszerspecifikáció elkészítése.
- **Rendszer- és szoftvertervezés**
  Követelmények szétválasztása hardver- illetve szoftverkövetelményekre. A rendszer átfogó architektúrájának, a logikai rendszerterv kialakítása. Alapvető szoftverrendszer-absztrakciók és a közöttük lévő kapcsolatok leírása.

- **Implementáció és modulteszt**

  Szoftverterv realizálása, programok, illetve programegységek halmazának megalkotása. A létrejött egységek tesztelése a követelmények alapján.
- **Integráció és rendszerteszt**

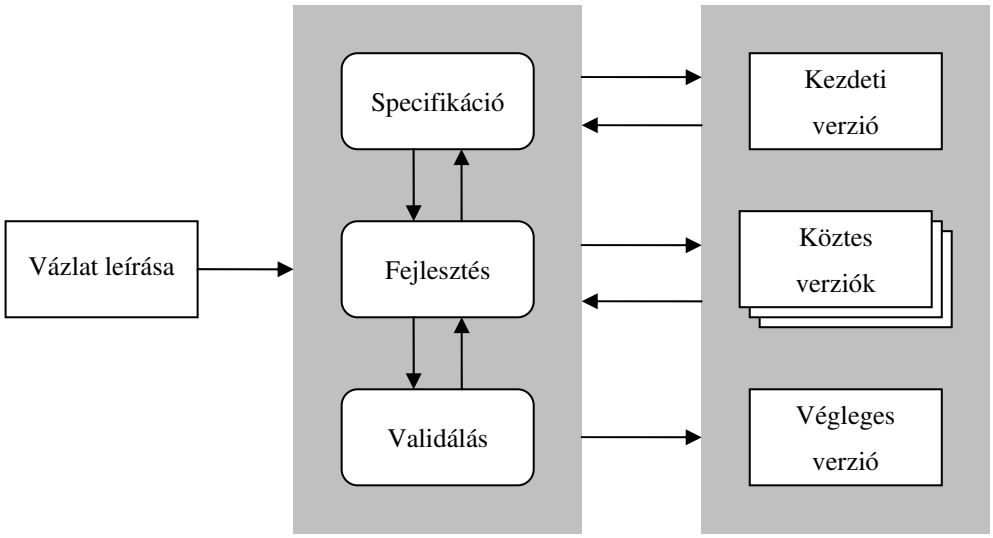
  Különálló szoftveregységek integrációja, és teljes rendszerként való tesztelése. Az elkészült és javított rendszer átadása a felhasználónak.
- **Működtetés és karbantartás**

  A rendszer telepítése és gyakorlati használatbavétele után a termék karbantartása. Karbantartás, azaz az eddig felderítetlen hibák javítása. A rendszer implementációjának és szolgáltatásainak továbbfejlesztése a fellépő új igények alapján.

A vízesés modell problémáját a folyamat szakaszainak különálló részekké történő, nemflexibilis particionálása okozza. A modell nagy hibája, hogy nem írja le jól a valós folyamatokat, továbbá kizárja a párhuzamos végrehajtás lehetőségét. A fejlesztés elindulása után csak nehezen lehet változtatni a folyamaton, mivel a modell korai szakaszaiban állást kell foglalnunk a további lépésekre vonatkozóan. Összefoglalásként a modell akkor használható hatékonyan, ha már előre jól ismerjük a követelményeket, vagy ha kezdeti prototípus előállítási folyamatában alkalmazzuk.

1.1.2. Evolúciós modell

Az evolúciós fejlesztés a következő alapötletet követi: fejlesszünk ki egy kezdeti implementációt, adjuk oda a felhasználónak véleményezésre, majd ezek alapján több verzión keresztül finomítsuk a rendszert, amíg el nem érjük a megfelelő szintet. A vízesésmodell szétválasztott specifikációs, fejlesztési és validációs tevékenységeihez képest az a modell jobban lehetővé teszi a tevékenységek közötti párhuzamosságot és a gyors visszacsatolást (1.2. ábra).

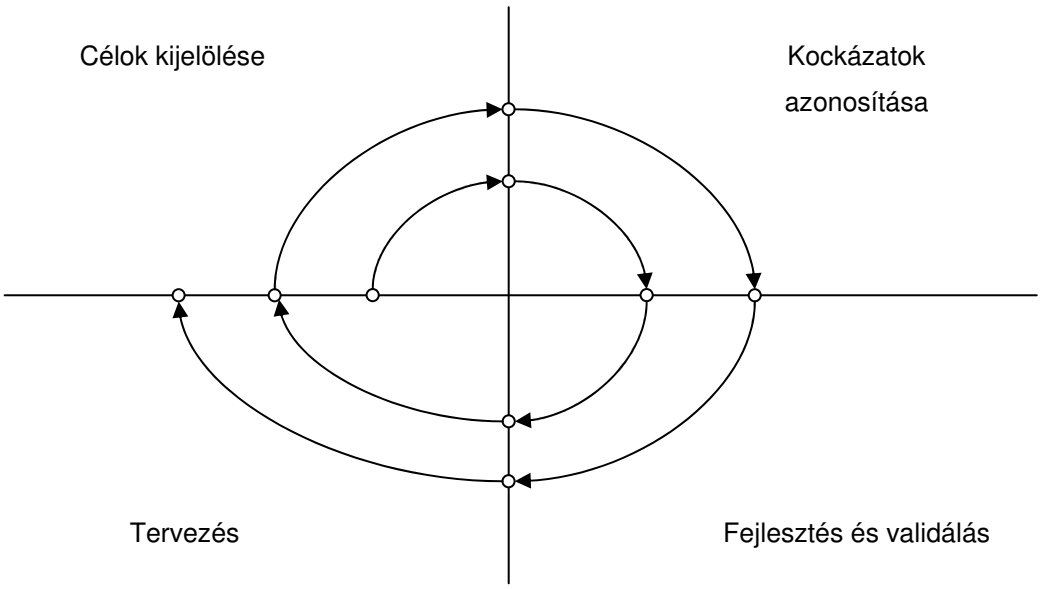


1.2. ábra – Evolúciós fejlesztés

Alapvetően két különböző típusú evolúciós fejlesztésről beszélhetünk. Feltáró fejlesztés esetén a folyamat célja, hogy a felhasználóval együtt feltárja a követelményeket és kialakítsa a végleges rendszert úgy, hogy egyre több, a megrendelő által kért tulajdonságot adunk hozzá a már meglévőkhöz. A másik típusnál eldobható prototípusok készítésén keresztül érjük el a célt. Ez a módszer arra koncentrál, hogy a lehető legjobban megértsük a felhasználó elvárásait, a prototípus segítségével pedig a kevésbé érthető követelmények pontosítására van lehetőség.

Az evolúciós megközelítés hatékonyabb a vízesésmodellnél, ha olyan rendszer kifejlesztése a cél, amely közvetlenül megfelel a felhasználó elvárásainak. További előnye, hogy a rendszer-specifikáció inkrementálisan bővíthető, illetve lehetőséget biztosít a párhuzamos fejlesztésre is. Gyakran alkalmazzák olyan területeken is, ahol nagy megbízhatóságú rendszer létrehozása a cél. Problémát okoz azonban, hogy a folyamatos változtatások, finomítások lerontják a szoftver strukturáltságát, a változtatások összevonása pedig egyre nehezebbé és költségesebbé válik a fejlesztés előrehaladásával. Hátrányként megemlíthető még az is, hogy a fejlesztési folyamat nehezen „mérhető”, hiszen előre nehéz megjósolni, hogy hány köztes verzióra lesz szükség a végleges termék elérése előtt.

Az evolúciós módszerhez szorosan kapcsolódik a folyamatiteráció spirális modellje. Ez a szoftverfolyamatot nem tevékenységek és közöttük található esetleges visszalépések folyamataként tekinti, hanem spirálként reprezentálja (1.3. ábra).



1.3. ábra – A spirál modell

- **Célok kijelölése**

  A projekt adott fázis által kitűzött célok, megszorítások meghatározása, illetve ezek alapján alternatívák kidolgozása.
- **Kockázatok azonosítása**

  Kockázatok részletes elemzése, lépések kidolgozása a kockázati tényezők csökkentése érdekében.
- **Fejlesztés és validálás**

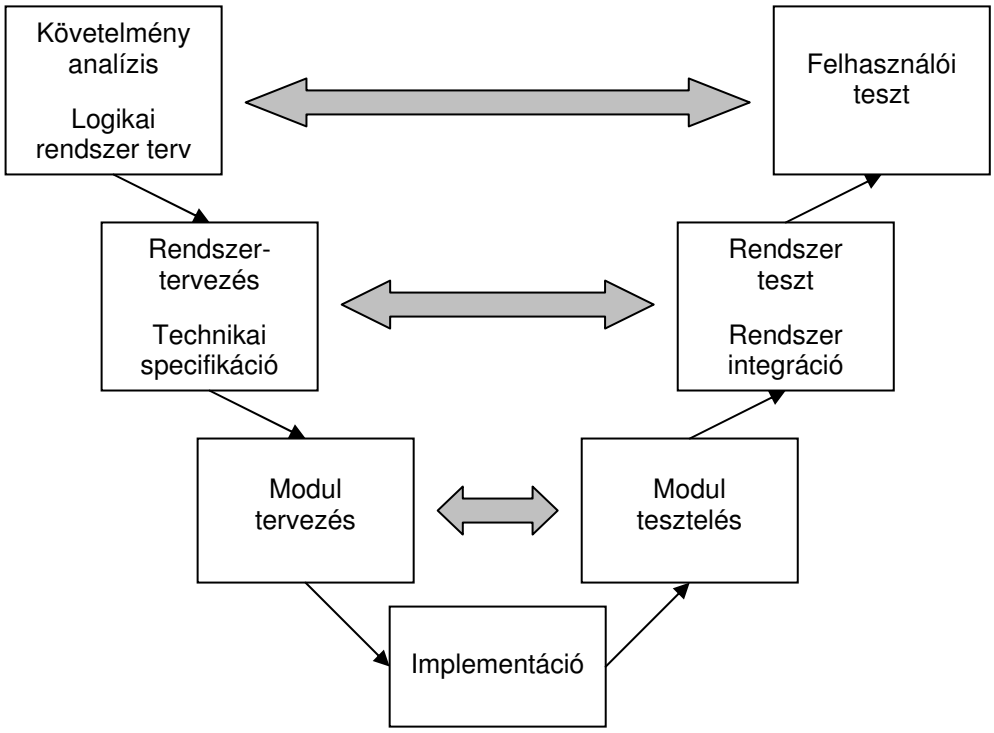
  A feltárt kockázatok alapján a legjobb alternatíva kiválasztása.
- **Tervezés**

  A kifejlesztett rendszer vizsgálata alapján a következő spirál és fázis ütemezésének megtervezése, ha erre szükség van.

A spirális folyamatiteráció egyik nagy előnye, hogy a modell részeként számol a kockázati tényezőkkel. Ezek problémákat okozhatnak a projektben, elsősorban határidő- és költségtúllépéseket, melyek elkerülése nagyon fontos projektmenedzselési feladat. Határozott hátránya azonban, hogy nagyon tapasztalt vezetők szükséges az eredményességhez, illetve nem bánik gazdaságosan az erőforrásokkal.

1.2. A V-modell

A V-modell 1997-ben jelent meg, gyakorlatilag a vízesés modell továbbfejlesztésével: a legfontosabb eltérés a tesztelési ág visszahajtása. Ezzel a megoldással azonos hierarchia szintre emelkedtek a tervezési és a hozzájuk tartozó tesztelési lépések. A modell már jobban leírhatja a valós folyamatokat, továbbá lehetőséget ad a lépések párhuzamos végrehajtására a fejlesztési absztrakciós szintek bevezetésével, és az azonos szintekhez tartozó tesztek és fejlesztési lépések összekapcsolásával. A hierarchikus felbontás mellett rendszer, alrendszer és modul absztrakciós szintekről beszélhetünk. A konkrét tervezési feladathoz igazított modellünk (1.4. ábra) az alrendszer absztrakciós szinthez tartozó lépéseket kihagyja, tehát nem foglalkozik az alrendszer szintű tervezéssel és teszteléssel. Tekintsük át a módosított modell lépéseit részletesen [5].

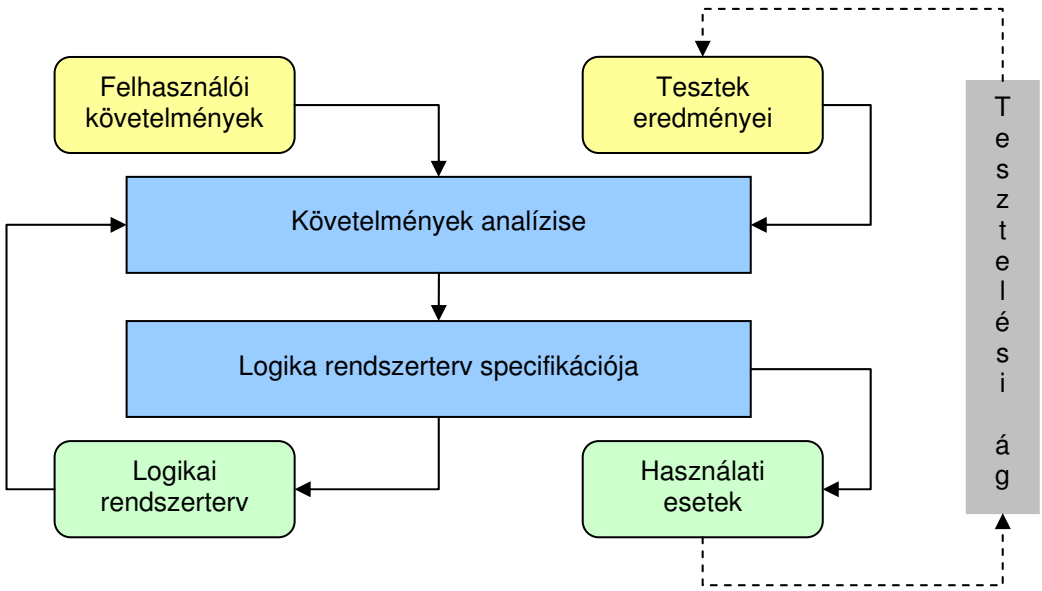


1.4. ábra – Módosított V-modell

1.2.1. Követelmények analízise és a logika rendszerterv elkészítése

A modell első lépésének első feladata a felhasználói követelmények felmérése és elemzése. A követelmények a legtöbb esetben a felhasználó nyelvén íródnak, tehát először le kell azokat fordítanunk a műszaki nyelvre, illetve ki kell szűrnünk az ellentmondásokat és következetlenségeket. A második feladat a megismert felhasználói igények alapján elkészíteni a rendszer logika tervét: a rendszert alkotó logikai egységek funkcióinak, interfészeinek leírását. A megvalósítás részletkérdéseivel ebben a lépésben a modell nem foglalkozik. Röviden összefoglalva a logikai rendszerarchitektúra a következő dolgokat tartalmazza:

- Tulajdonságok és funkciók, amivel a rendszernek rendelkeznie kell.
- Tulajdonságok és funkciók, amivel a rendszernek nem kell rendelkeznie.

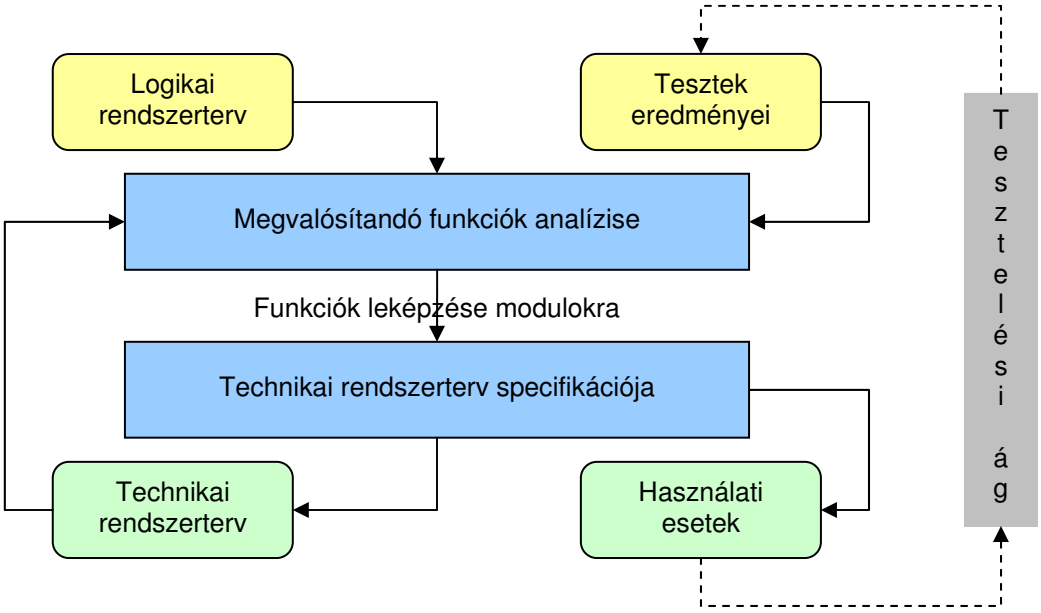


1.5. ábra – V-modell első lépésének feladatai

A szöveges leírás mellett különféle modellezési eszközök is segítséget nyújthatnak a logikai rendszerterv elkészítéséhez Elsősorban az UML alapú leírások használhatók erre a célra. A rendszerarchitektúra specifikációja mellett a tesztelési ággal való kapcsolatot is figyelembe kell vennünk a modell első lépésében: használati példákat (*use case*) kell definiálunk, melyek a rendszer és végfelhasználói tesztek alapjaiul fognak szolgálni.

1.2.2. A logikai rendszerterv elemzése és a technikai architektúra

Az előző lépés során elkészült logikai rendszerterv alapján megalkotott technikai rendszerarchitektúra adja meg a rendszer technikai vázát. A modellünk következő szakaszának feladata, hogy a meghatározott funkciókat elossza a definiált modulok között (1.6. ábra). Ebben a lépésben fogjuk továbbá a szoftver és hardverfolyamatokat különválasztani. A logikai architektúra leképzése a technikaira sokszor nem egyszerű hozzárendelést jelent, általában egy modul több logikai funkciót valósít meg, éppen ebből adódik, hogy ez a feladata külön lépésként szerepel modellünkben.

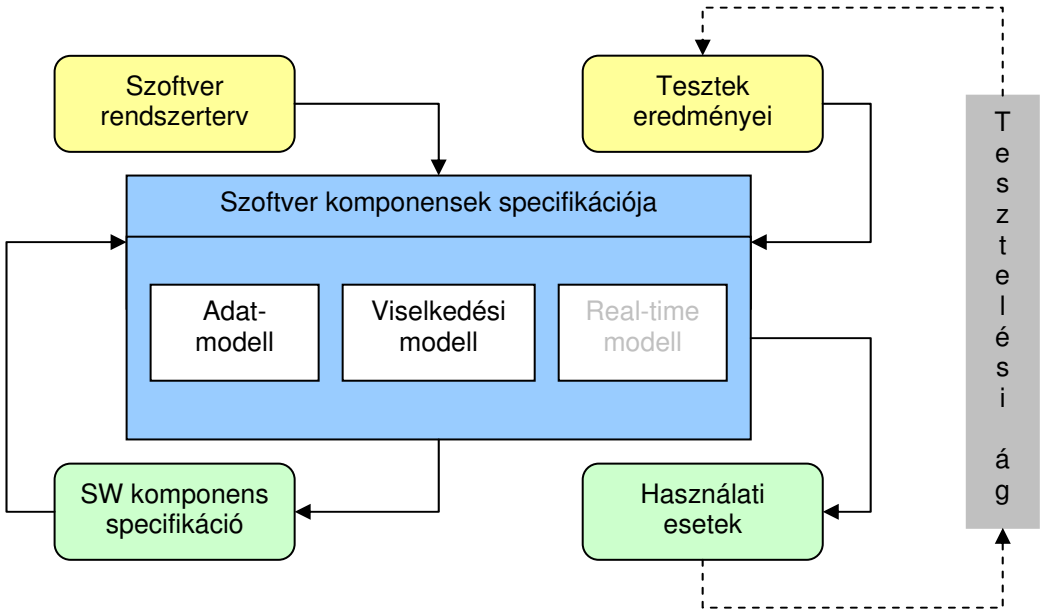


1.6. ábra – A tervezési folyamat második lépése

1.2.3. Szoftverkomponensek specifikálása

A modell előző lépésében a szoftver architektúra megtervezése során pontosan definiáltuk, hogy a rendszer milyen komponensekből áll. Ebben a lépésben kidolgozzuk az egyes komponensek részletes viselkedését, amely alapvetően a következő részekből áll: adatmodell és viselkedési modell (1.7. ábra). Ha a valós idejű végrehajtás is tervezési szempont, ezek mellett egy ún. *real-time* modellt is össze kell állítanunk. Első lépésben az adatmodell megadása a feladatunk a szoftverarchitektúra alapján. Az adatmodell tartalmazza, hogy egy adott komponensnek milyen bemenetei és kimenetei

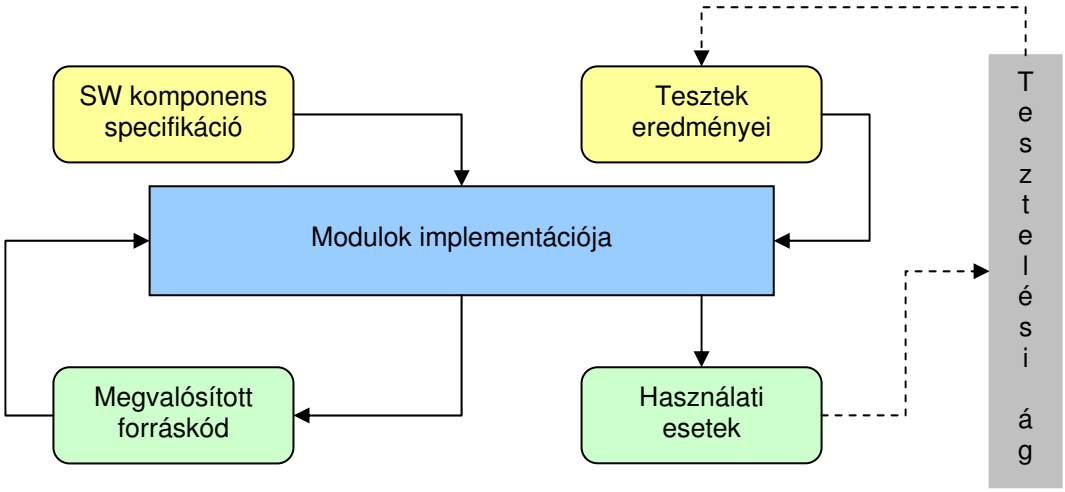
vannak. Második lépésben definiáljuk a viselkedési modellt, mely leírja az adott szoftverkomponens belső működését.



1.7. ábra – Szoftverkomponensek specifikálása

1.2.4. Modulok implementációja

Az implementációs folyamat során (1.8. ábra) az előző fejezetben megtervezett adat és viselkedési modellt valósítjuk meg. Fontos szempont ebben a lépésben a hardveres korlátok felderítése, valamint a megfelelő implementációs környezet kiválasztása is.



1.8. ábra – A modulok implementációjának lépései

A vízesésmodellhez hasonlóan, a folyamat szakaszainak különálló részekké történő bontása okozza a modell fő problémáit, mivel a fejlesztés elindulása után csak nehezen lehet változtatni a folyamaton, ugyanis a modell korai szakaszaiban már állást kell foglalnunk a további lépésekre vonatkozóan. Ezért erre a modellre is érvényes az a megállapítás, hogy használata során figyelembe kell vennünk, a valóságban elsőre sohasem készül tökéletes rendszer, tehát valamilyen folyamatiteráció mindenképpen szükségszerű a gyakorlati végrehajtás során.

A fejezetben áttekintettük és összehasonlítottuk a különböző szoftvertervezési modelleket. A hátrányok ellenére számos beágyazott rendszereket fejlesztő cég, főleg az autóipar piacán ezt a megoldást alkalmazza, emiatt esetünkben is a fejezetben bemutatott, egyszerűsített V-modellt követjük a szoftver fejlesztése során.

2. A grafikus függvénykönyvtár megtervezése

Miután megismertük a fejlesztési folyamatok modellezésének, azaz a szoftverfolyamat egyszerűsített leírásának módszereit, és áttekintettük a választott modell lépéseit, vizsgáljuk meg ezeken keresztül az adott feladat megoldásának folyamatát. Az alábbi fejezet a bevezetőben részletesen definiált feladat (grafikus függvénykönyvtár fejlesztése beágyazott rendszeren) megoldásának megtervezésével foglalkozik, a korábban ismertetett szoftverfejlesztési modell segítségével.

2.1. Követelmények analízise és a logika rendszerterv

A modell első lépésének első részfeladata a felhasználói követelmények felmérése és elemzése. A második feladat a megismert felhasználói igények alapján elkészíteni a rendszer logika tervét, azaz a rendszerarchitektúrát. Emellett természetesen a tesztelési ággal való kapcsolatot is figyelembe kell vennünk a használati példák segítségével, melyekkel részletesen a következő fejezetben foglalkozunk. A legfontosabb felhasználói követelmények a grafikus függvénykönyvtárra vonatkozóan a következők:

- Általános célú megoldás
- Egyszerű portolhatóság és bővíthetőség
- Könnyen kezelhető felhasználói felület
- Beágyazott rendszerekhez optimalizált megvalósítás
- Kijelző típusától és jellemzőitől való függetlenség
- Érintőképernyő és jellemzőitől való függetlenség
- Egyszerű rajzelemek kezelése

A felsorolásból látható, hogy a követelmények nem túl szigorúak és konkrétak, így a logikai rendszerterv elkészítéséhez viszonylag szabad mozgásteret adnak. Azonban a laza követelmények mellett sem tekinthetünk el a gondos tervezéstől, ugyanis a rendszerarchitektúra megalkotása kulcslépése a szoftverfejlesztésnek, a

tervezés e szakaszában felmerült hibákat a későbbiekben csak nagy nehézségek árán küszöbölhetjük ki.

2.1.1. A logikai rendszerterv

A logikai rendszerarchitektúra ismertetése előtt tekintsük át a felhasználói követelményeket részletesebben. Első lépésben vizsgáljuk meg, hogy milyen megfontolások segítségével tehetjük a megldást általános célúvá és portolhatóvá. Az elkészült szoftver működése során különböző platformokon futhat: számos típusú és felépítésű processzor létezik a piacon, mindegyik rendelkezik egyedi tulajdonságokkal bizonyos szempontok alapján, a hardverből fakadó eltérések miatt. Így a tervezéskor figyelembe kell venni, hogy ahol erre van lehetőség, a hardvertől független megoldásban, magasabb absztrakciós szinten gondolkodjunk. A széles körben használt implementációs nyelv használata is szerepet játszik ebben, részleteivel azonban csak a tervezés utolsó lépésében foglalkozunk, de a rendszertervbe ezt is rögzítenünk kell. A bővíthetőség szempontját legegyszerűbben a megoldás moduláris felépítésének segítségével teljesíthetjük: a megvalósítandó funkciókat csoportosítanunk kell, majd a funkciócsoportokhoz hozzá kell rendelnünk a megvalósított követelményt is. Ennek segítségével a könnyű kezelhetőséget, mint felhasználói elvárást is teljesíthetjük egyszerre, mely emellett függ a használt programozási nyelvtől is. A moduláris felépítést nem csak a funkciók csoportosítására, hanem a rajzelemek kezelésére vonatkozóan is alkalmazhatjuk, így a későbbiekben definiált egyszerű rajzelemekből további, bonyolultabb elemeket építhetünk fel. A beágyazott rendszereknél általában az erőforrások optimális kihasználása a legfőbb szempont. Kritikus erőforrásnak legtöbbször a processzor és a memória számít az ilyen rendszerekben. Ezek alapján tehát olyan szoftvert kell készítenünk, amely ügyel a processzor számítási kapacitásának korlátaira, emellett a megvalósítandó funkciókat a lehető legtömörebb kódban implementálja. Továbbá foglalkoznunk kell az adatok tárolásának módjával is az optimális memóriakihasználtság miatt. Az előbbiekben ismertetett szempontok alapján, a felhasználó elvárásai alapján csoportosítva a következő műszaki követelményekre bonthatjuk le a feladatot:

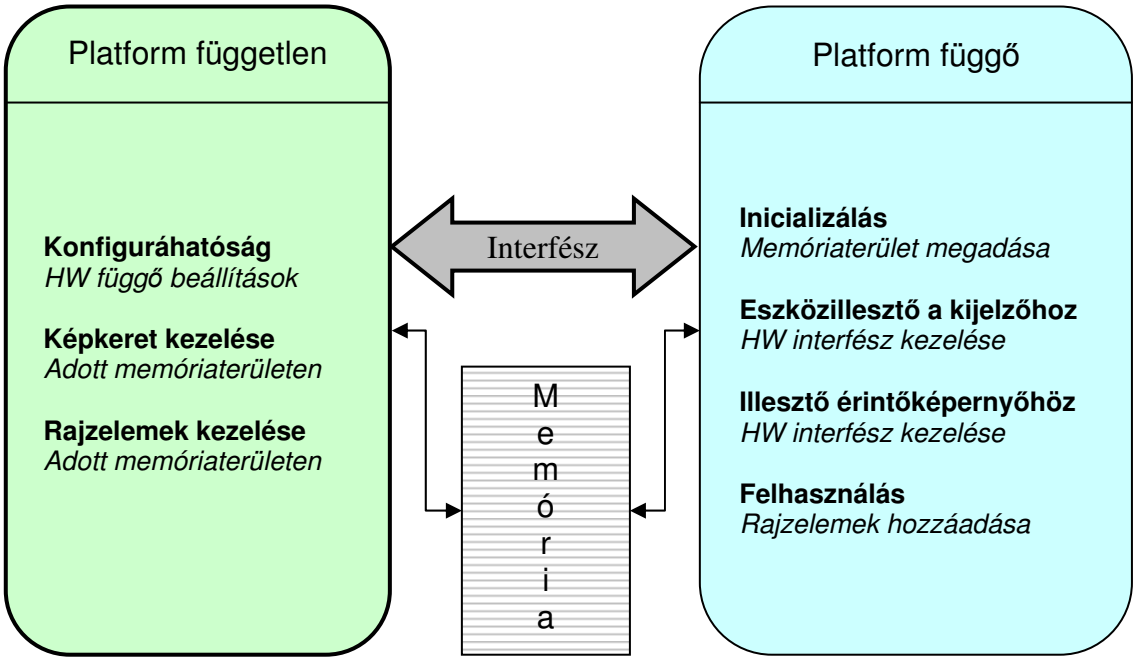
- **Általános célú megoldás, egyszerű portolhatóság**
  - Hardvertől független, magasabb absztrakciós szint
  - Széles körben elterjedt implementációs nyelv
- **Bővíthetőség, könnyen kezelhető felhasználói felület, rajzelemek kezelése**
  - Moduláris felépítés a funkcióknak megfelelően
  - Moduláris felépítés a rajzelemek kezeléséhez
- **Beágyazott rendszerekhez optimalizált megoldás**
  - Egységes memóriakezelés, memóriahasználat minimalizálása
  - Programozási nyelv megválasztása
- **Kijelző típusától és jellemzőitől való függetlenség**
  - Paraméterek (képméret, ábrázolási formátum) beállíthatósága
  - Képkeret egységes kezelése (bittérkép)
- **Érintőképernyő típusától és jellemzőitől való függetlenség**
  - Kezelőfüggvények beállíthatósága

A műszaki követelmények alapján a feladatunk két részre osztható alapján: platform független és platformfüggő részekre. A platform független modul megfeleltethető az általános célú grafikus függvénykönyvtárnak, amely megvalósítása a diplomatervezés feladata, míg a másik modul megfeleltethető a konkrét beágyazott rendszernek, amin az előbbit használjuk. A két modul közötti kapcsolatot a memória, mint közös erőforrás biztosítja. A hardverfüggő rész elsősorban a tesztelés, illetve a konkrát alkalmazáshoz való illesztés során kerül előtérbe, részleteit a következő fejezetben tárgyaljuk. A továbbiakban a platform független rész - melybe beleértendő a két egység közötti interfész – tervezésével foglalkozunk. A műszaki nyelvre „lefordított” követelmények alapján határozzuk meg, milyen funkciókat kell megvalósítania a megoldásnak:

- **Incializáló és konfigurációs funkciók**
  - a függvénykönyvtár inicializálása
    - a legmagasabb szintű modul, az almodulok, és ezek kapcsolatát biztosító interfész létrehozása
    - felhasználói interfész létrehozása

- képkeret (*frame*) kezeléséhez tartozó hardverfüggő beállítások megadása
  - frame méreteinek, ábrázolási formátumának megadása
  - a képkerethez tartozó memóriaterület megadása
- rajzelemek kezelésének beállításai
  - rajzelemek tárolásának memóriaterület biztosítása, a tárolás módja

- **Képkeretet kezelő funkciók**
  - frame frissítése, a rajzelemek újrarajzolása
  - a képkerethez tartozó memóriaterület váltása
  - háttérszín beállítása
  - teljes tartalom törlése
- **Rajzelemeket kezelő funkciók**
  - rajzelem hozzáadás és törlése
    - ábrázolás paramétereinek megadása
    - érintőképernyő megérintéséhez tartozó kezelőfüggvények megadása
  - rajzelemek rejtése és megjelenítése
  - rajzelemek módosítása



2.1. ábra – A logikai rendszerterv

A bemutatott rendszerarchitektúrát (2.1. ábra) a felhasználói elvárásoknak megfelelően terveztük meg, azokat mind teljesíti a definiált funkciókon keresztül. Ezután áttérhetünk az elkészült rendszerterv alapján a technikai rendszerterv megalkotására, azaz a V-modellünk következő lépésére.

2.2. A technikai rendszerarchitektúra

A korábbi fejezet során elkészült logikai rendszerterv alapján megalkotott technikai rendszerarchitektúra adja meg a rendszer technikai vázát. A modellünk e lépésének feladata, hogy a meghatározott funkciókat elossza a definiált modulok között. Mint láthattuk, a megvalósítandó funkciókat három csoportba sorolhatjuk, így a legegyszerűbben ezeket három különálló szoftvermodulra képezhetjük le, majd ezután a három kisebb modult egy magasabb szintű modul segítségével foglalhatjuk össze:

- **Konfigurációs modul** ← Inicializáló és konfigurációs funkciók
- **Frame kezelő modul** ← Képkeretet kezelő funkciók
- **Rajzelem kezelő modul** ← Rajzelemeket kezelő funkciók
- **Függvénykönyvtár modul** ← Funkciók összessége

A megvalósítandó funkciók szempontjából a legfontosabb a konfigurációs modul, mely a következő funkciókat biztosítja a felhasználó számára:

- Legmagasabb szintű modul, az almodulok, és kapcsolataik létrehozása
  - hardverfüggetlen és hardverfüggő rész közötti interfész
- Felhasználói interfész létrehozása a többi modul használatára
- A képkeret tulajdonságainak beállíthatósága
  - felhasználói interfészen keresztül
  - méret és ábrázolási formátum
  - memóriaterület mérete és formátuma
- A rajzelemek tárolásának konfigurálhatósága
  - felhasználói interfészen keresztül
  - memóriaterület és mérete
  - tárolás formátuma

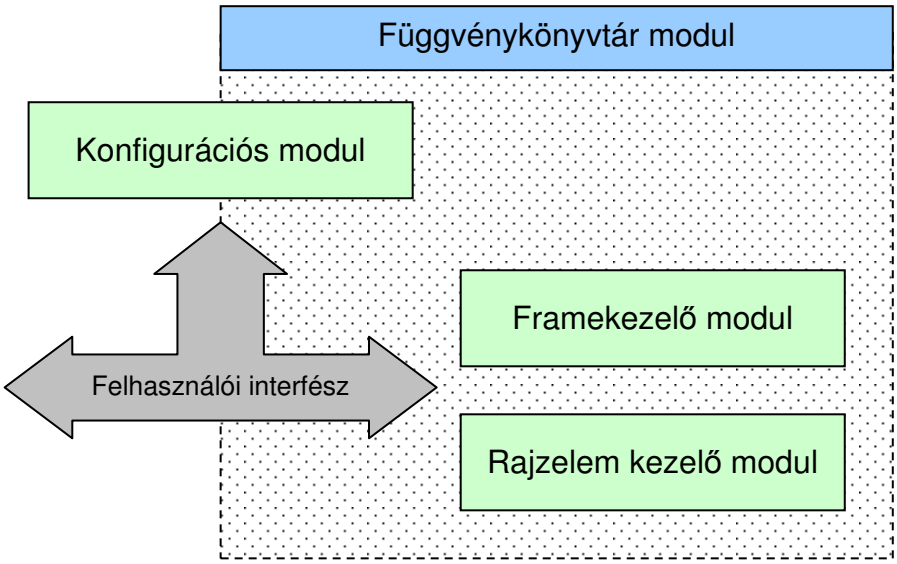
A frame kezelő modul a képkeretet kezelő funkciókat valósítja meg. A modult a konfigurációs modul funkcióinak segítségével hozhatjuk létre, illetve paramétereit is ezen keresztül állíthatjuk be a létrehozáskor. Ez a komponens a következő funkciókat valósítja meg:

- Képkeret frissítése, újrarajzolása
- Memóriaterület cseréje
  - új memóriaterület
- Háttérszín megadása, megváltoztatása
- Képkeret tartalmának törlése

A harmadik, rajzelem kezelő modul, a rajzelemeket kezelő funckiókat biztosítja. A komponenst a frame kezelő modulhoz hasonlóan a konfigurációs funkciók segítségével hozhatjuk létre, illetve állíthatjuk be a tárolás paramétereit. A következő lehetőségeket biztosítja a felhasználó számára:

- Rajzelemek létrehozása, felvétele a nyilvántartásba
  - rajzelem típusa, módosíthatósága
  - ábrázolás paraméterei (a típushoz tartozóan)
  - érintést kezelő függvény (típustól függően)
- Rajzelem törlése a nyilvántartásból, azonosító alapján
- Rajzelemek csoportjának megjelenítése és elrejtése
  - azonosítók tömbje
- Rajzelem módosítása
  - rajzelem típusa
  - új ábrázolási paraméterek
- Érintőképernyő eseményeinek kezelése
  - megérintett pont paraméterei

A technikai rendszerterv szöveges leírása mellett felépítését ábrán is szemléltethetjük (2.2. ábra).



2.2. ábra – A technikai rendszerterv

2.3. Szoftverkomponensek specifikálása

A szoftver architektúra megtervezése során pontosan definiáltuk, hogy a rendszer milyen komponensekből áll, illetve az egyes komponensek milyen funkciókat valósítanak meg. A modell következő lépésében kidolgozzuk az egyes komponensek részletes működését, amely alapvetően a következő részekből áll: adatmodell és viselkedési modell. Első lépésben az adatmodell megadása a feladatunk a szoftverarchitektúra alapján. Az adatmodell tartalmazza, hogy egy adott egységnek milyen bemenetei és kimenetei vannak. Második lépésben definiáljuk a viselkedési modellt, mely leírja az adott szoftverkomponens belső működését.

A szoftverarchitektúra megtervezésekor három modulra képeztük le a megvalósítandó funkciókat (konfigurációs modul, framekezelő modul, rajzelem kezelő modul), illetve egy, magasabb szintű modul segítségével kapcsoltuk össze ezeket. Definiáljuk az említett modulokat részletesebben és határozzuk meg az adat és viselkedési modelleket. A felhasználónak első lépésben mindig a konfigurációs modul funkcióin keresztül kell az alapvető beállításokat megadnia, a további modulok által biztosított funkciókat csak ezután érheti el. A modulokat a következő almodulokra oszthatjuk:

- **Konfigurációs modul**
  - Függvénykönyvtár létrehozása

    Bemenet

    - maximális rajzelemek száma

    Kimenet

    - a felhasználói interfész

    Viselkedés

    - felhasználói interfész létrehozása a platform független részhez
    - a magasabb szintű modul belső struktúrájának létrehozása
    - modulok kapcsolatainak beállítása
  - Képkeret tulajdonságainak konfigurálása

    Bemenet

    - alapértelmezett memóriaterület a frame számára
    - frame vízszintes pixeleinek száma
    - frame függőleges pixeleinek száma
    - megjelenítési formátum
    - frame mérete
    - frame hátterének alapértelmezett színe

    Viselkedés

    - képkerethez tartozó hardverfüggő beállítások tárolása
    - kezdeti paraméterek, memória beállítása
  - Rajzelemek tárolásának konfigurálás

    Bemenet

    - memóriaterület a rajzelemek tárolására számára
    - megadott memóriaterület mérete
    - tárolás lépésköze

    Viselkedés

    - rajzelemtárolás beállításainak tárolása
    - rajzelemek nyilvántartásának létrehozása
- **Képkeretet kezelő modul**
  - Képkeret frissítése

    Viselkedés

    - nyilvántartás megjelenítendő rajzelemei és háttérszín alapján az aktuális memóriatartományon a bittérkép előállítása

- Képkeret cseréje

  Bemenet

  - új memóriatartomány képkeret tárolására

  Viselkedés

  - képkerethez tartozó memóriatartomány cseréje
  - háttérszín alapján az új bittérkép inicializálása

- Képkeret törlése

  Viselkedés

  - összes rajzelem elrejtése
  - háttérszín alapján a bittérkép újrainicializálása

- Képkeret háttérszínének megadása

  Bemenet

  - új háttérszín

  Viselkedés

  - háttérszín alapján a bittérkép újrainicializálása

- **Rajzelemeket kezelő modul**
  - Új rajzelem hozzáadása

    Bemenet

    - új rajzelem típusa
    - a típushoz tartozó paraméterek

    Kimenet

    - új elem azonosítója

    Viselkedés

    - adott típusú rajzelem hozzáadása a nyilvántartáshoz
    - paraméterek tárolása a memóriában

  - Rajzelem törlése

    Bemenet

    - rajzelem azonosítója

    Viselkedés

    - megadott azonosítójú rajzelem törlése a nyilvántartásból
    - paramétereket tartalmazó memóriaterület felszabadítása

  - Rajzelem módosítása

    Bemenet

    - rajzelem azonosítója
    - a rajzelem típusa
    - a típushoz tartozó új paraméterek

    Viselkedés

    - megadott azonosítójú rajzelem paramétereinek módosítása

- Rajzelemek elrejtése

  Bemenet

  - rajzelemek azonosítói
  - elrejtendő rajzelemek száma

  Viselkedés

  - adott azonosítójú és számú rajzelemek rejtetté állítása
- Rajzelemek megjelenítése

  Bemenet

  - rajzelemek azonosítói
  - megjelenítendő rajzelemek száma

  Viselkedés

  - adott azonosítójú és számú rajzelemek megjelenítetté állítása
- Érintőképernyő érintésének kezelése

  Bemenet

  - megérintett pont koordinátái

  Viselkedés

  - koordináta és rajzelemek paraméterei alapján az érintési ponthoz tartozó, megfelelő típusú rajzelem kezelőfüggvényének hívása

Minden modulhoz tartozik egy kimenet, amely a végrehajtás sikerességét adja meg. Ezeket a kimeneteket a fenti felsorolásban külön nem tüntettük fel. A tervezési modellünk e lépésében előállítottuk a szoftverkomponensek részletes specifikációját, megadtuk azok adat- illetve viselkedési modelljét. Ezután elkezdhetünk foglalkozni a definiált almodulok implementációs kérdéseivel.

2.4. Modulok implementációja

Az implementációs folyamat során az előző fejezetben megtervezett adat és viselkedési modellt valósítjuk meg. Fontos szempont ebben a lépésben a hardveres korlátok felderítése, esetünkben azonban általános, platform független megoldást valósítunk meg, így lényegesebb szempont a portolhatóság. További fontos megkötést jelenthetnek a memóriaterület korlátai. Mivel korábban definiáltuk, hogy beágyazott rendszerekre optimalizált megoldást készítünk, már a tervezési szakasz elejétől kezdve kiemelt szerepet kapott ez a szempont: az elkészült szoftverkomponens specifikációban is rugalmasan konfigurálható memóriakezelést terveztünk meg. Azonban az adatmodell

implementációjánál is figyelembe kell vennünk ezt, és a megtervezett változók és paraméterek optimális méretéről és azok tárolásáról gondoskodnunk kell. A modell feladataihoz hozzátartozik a megfelelő implementációs környezet kiválasztása is. A memóriakezeléshez hasonlóan ezt a szempontot is figyelembe vettük a rendszerterv elkészítésekor, elsősorban a hordozhatóság követelményének teljesítése okán. Mivel a beágyazott rendszereknél a mai napig a C programozási nyelv a legelterjedtebb, és várhatóan az elkövetkező években ez továbbra is így lesz, az implementáció során mi is ezt a nyelvet használtuk.

Vizsgáljuk meg részletesebben az adatmodellek implementációját, azaz a memóriakezelést a definiált modulokra vonatkozóan. A memóriaterületek megadása elsősorban a konfigurációs modulon keresztül tehető meg, alapvetően a következő struktúrák tárolását kell megvalósítanunk:

- Függvénykönyvtár belső paramétereinek tárolása
- Képkeret tárolása
- Rajzelemek tárolása és nyilvántartása

Mint korábban említettük a felhasználónak mindig a konfigurációs modul funkcióin keresztül kell az alapvető beállításokat megadnia, és ide tartozik a memóriakezelés paramétereinek beállítása is. Első lépésben létre kell hoznunk a belső paraméterek tárolásához szükséges struktúrát, ami a függvénykönyvtár példányosításakor, dinamikus memóriafoglalással jön létre és a következő adatokat tartalmazza:

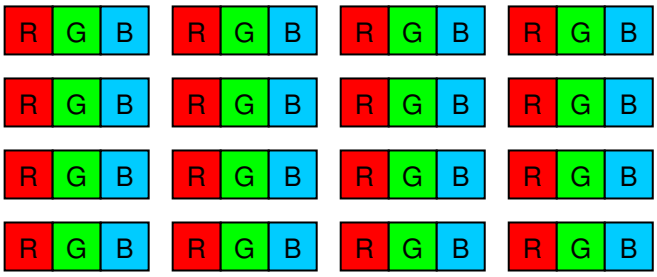
- Felhasználói interfészt biztosító függvénystruktúra
- Frame paramétereit tároló struktúra
  - mutató az aktuális bittérképre
  - szélesség, magasság pixelekben
  - formátum, méret és aktuális háttérszín
- Rajzelemek-tárolás paramétereihez tartozó struktúra
  - mutató a tároláshoz használt memóriaterületre
  - méret és lépésköz

- Rajzelemek nyilvántartásához szükséges struktúra
  - elemek maximális és aktuális száma
  - rajzelem azonosítók tömbje és az utolsó érvényes index
- Belső állapotváltozó

A képkeret tulajdonságait illetve a rajzelemek tárolásának tulajdonságait magába foglaló struktúra a már korábban ismertetett, a konfigurációs modulba tartozó funkciókkal állíthatók be. Mivel dinamikus memóriafoglalást használunk az egész struktúra létrehozásához, fontos kiemelni, hogy mindig a futó alkalmazás kezdeti fázisában tegyük ezt meg. Tekintsük át ezután a memória kezelésének részleteit.

2.4.1. A képkeret tárolása

Első lépésben vizsgáljuk meg a framekezelő függvényekhez tartozó memóriatartományt: a bittérképet. A használt bittérkép mérete, szélessége, magassága illetve formátuma az aktuális platformtól függ, felépítése azonban hardvertől függetlenül mindig hasonló. Példaként tekintsünk egy négyszer négy pixelből álló, RGB888 formátumú bittérképet (2.3. ábra).

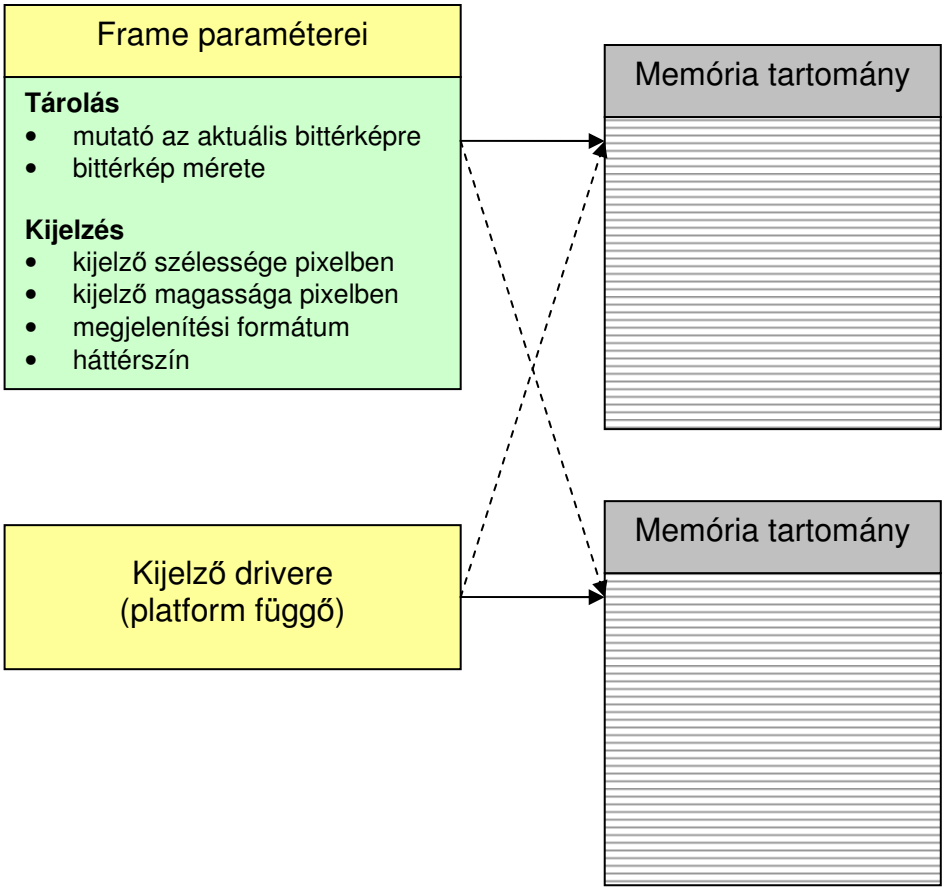


2.3. ábra – 4x4-es bittérkép

A pixelre vonatkozó RGB888 formátum annyit jelent, hogy egy pixel három színösszetevőből áll, vörösből, zöldből és kékből, és minden egyes szín intenzitását nyolc biten tároljuk. Ezek alapján az ábrán egy négyzet jelent egy bájtot, illetve egy téglalap egy pixelt.

A képkeret tárolásával kapcsolatban fontos még kiemelni, hogy a frame tárolására használt memóriaterület mérete nem minden esetben egyezik meg a

képpontok számának és a az egy képponthoz tartozó bitek számának szorzatával. Ennek oka abban keresendő, hogy általában a biztonságos működés érdekében nagyobb memóriaterületet biztosítunk a képkeret tárolásához, ugyanis egy esetleges alul- vagy túlindexelés rossz megjelenítést eredményezhet, de az egész rendszer működését is leállíthatja egy érvénytelen memória-hozzáférés. Ezen felül figyelembe kell azt is vennünk, hogy a legtöbb esetben azt a területet, amit éppen kijelzünk, tehát amit a kijelzőhöz tartozó eszközvezérlő szoftveregység használ, nem módosíthatjuk a megjelenítés közben. Emiatt célszerű legalább két memóriatartományt definiálni: az egyik területet az aktuális frame kijelzésére, a másikat az új frame szerkesztéséhez használjuk. A memóriaterületekre mutató hivatkozások egyszerű cseréjével így könnyen teljesíthetjük ezt a szempontot (2.4. ábra).



2.4. ábra – Bittérképek tárolása

2.4.2. Rajzelemek típusai és ábrázolásuk

A szoftver tervezés eddigi lépései során általánosságban már beszéltünk az alapvető rajzelemekről. Nézzük meg, hogy konkrétan melyek implementációja szükséges a legtöbb alkalmazáshoz. Fontos megjegyezni, hogy a kijelzőn csak olyan pontokat jeleníthetünk meg, melynek mindkét koordinátája egész szám, sőt valójában nem is pontok, hanem „kis négyzetek" a megjeleníthető legkisebb egységek. Első lépésben vizsgáljuk meg a talán legkézenfekvőbb rajzelemet: az egyenes vonalat, amelyet a két végpont (P_1 és P_2) megadásával definiálhatunk legegyszerűbben. Matematikailag a koordinátageometriából jól ismert, alábbi képlet segítségével írhatjuk le az egyenes szakaszokat:

$$P_1 = (x_1, y_1) \quad P_2 = (x_2, y_2)$$
$$y = mx + b \quad x \in [x_1, x_2] \tag{2.1}$$
$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad b = y_1 - mx_1 \tag{2.2}$$

Első megközelítésben használjuk a (2.1) és (2.2) képletet az ábrázoláshoz. Jelölje a `round(...)` kifejezés az egész számra való kerekítést, a `drawpoint(...)` kifejezés pedig a megfelelő pixel megjelenítését, ekkor az algoritmusunk a következő:

- x = `round`(x_1)
- m = ($y_2 - y_1$) / ($x_2 - x_1$)
- y = y_1
- **`while`** (x <= x2)
  - `drawpoint`(x, `round`(y))
  - x = x + 1
  - y = y + m

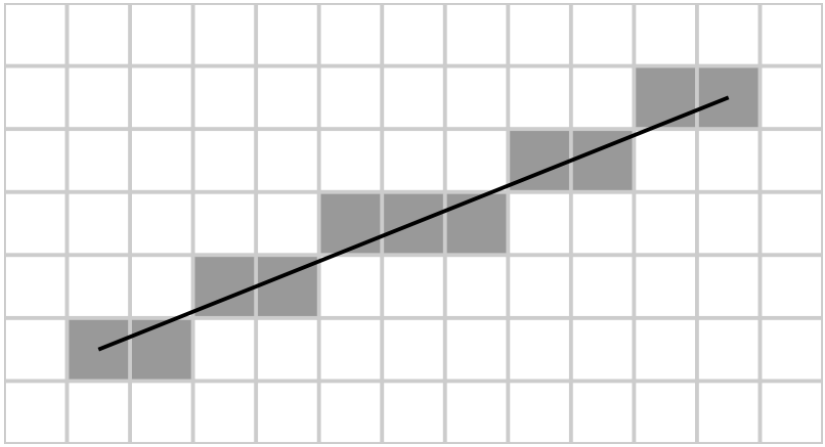
Már első ránézésre is látható, hogy ez a módszer számos hibával rendelkezik. Nem tudjuk például megjeleníteni az $x_1 = x_2$ esetet, illetve ezen felül az algoritmus kritikus része m, azaz a meredekség meghatározása. Belátható, hogy csak $|m| < 1$ esetén működik helyesen az algoritmus, illetve m legkisebb pontatlansága mellett is y inkrementálása miatt a hiba folyamatosan nő, végeredményben így a kijelzett és a megadott végpontok nem fognak egybeesni.

A felsorolt hibák elkerülésé érdekében az egyenes szakaszok ábrázolásához a *Bresenham* algoritmus [6] egyik speciális esetét használjuk. Első lépésben kiszámítjuk a `steep` változó értékét, mely segítségével biztosítjuk, hogy az |m| <= 1 feltétel minden esetben teljesüljön. Ha a kezdeti paraméterek mellett ez nem áll fent, ezt úgy érhetjük el, hogy a pontok koordinátáit felcseréljük, azaz tükrözzük a pontokat az x = y egyenesre, majd a konkrét ábrázolás során visszacseréljük őket. Ez nem okoz különbséget a végeredményben, de a számítást jelentősen megkönnyíti. A `dx` és `dy` változókban tároljuk a végpontok x és y koordinátáinak távolságát, ezekre az `error` változó kiszámítására van szükségünk, ami a megjeleníthető és a „valós" pont közötti eltérést, a hibatagot reprezentálja. A hibatag kiszámításához a kezdeti értékét tekintsük nullának. Az `x` változó minden inkrementálásakor `y (dy/dx)` értékkel változna, viszont a legkisebb lépésköz értéke 1, így ez `-(dy/dx)` hibát okoz (mivel `(dy/dx)` kisebb, mint egy). A hibatagot tehát minden inkrementáláskor ezzel az értékkel csökkentenünk kell. Ha a hiba kisebb lesz mint –½, akkor a lépésközzel, azaz eggyel inkrementálhatjuk vagy dekrementálhatjuk `y` értékét is, a meredekség előjelétől függően. Ekkor a hibatag is növekszik eggyel, így mindig –½ és ½ között lesz. Láthatóan ez a gondolatmenet a hibatag értékének tárolásához, illetve a hibához kapcsolódó logikai műveletekhez nem egész értékeket használ, ez pedig jelentősen lassítja az algoritmust. Ha az `error` változót a (hibatag + ½) értékként definiáljuk, akkor a logikai műveleteknél már kiküszöböltük a nem egészek használatát, továbbá ha ezt az értéket `dx`-szel megszorozzuk az `error` változó is mindig egész lesz. Ez alapján az algoritmus működését a következő, C programozási nyelvhez hasonló pszeudokód segítségével írhatjuk le, melyben az `abs(...)` kifejezés az abszolútérték képzést, a `swap(...)` két változó kicserélését, a `plot(...)` kifejezés pedig a megfelelő pixel megjelenítését jelenti, továbbá minden változó egész értékű:

- **`if`** (`abs`$(y_2 - y_1)$ > `abs`$(x_2 - x_1)$)
  - steep = 1 **`else`** steep = 0
- **`if`** (steep = 1)
  - `swap`(x_1, y_1) `swap`(x_2, y_2)
- **`if`** ($x_1 > x_2$)
  - `swap`(x_1, x_2) `swap`(y_1, y_2)
- dx = *abs* $(x_2 - x_1)$ dy = *abs* $(y_2 - y_1)$
- error = dx / 2

- **if** ($y_1 < y_2$)
  - ystep = 1 **else** ystep = -1
- $y = y_1$ $x = x_1$
- **while** ($x <= x_2$)
  - **if** (steep = 1)
    - `plot`(x, y) `else plot`(y,x)
  - error = error – dy
  - x = x + 1
  - **if** (error < 0)
    - y = y + ystep error = error + dx

Az algoritmus végeredményeként ábrázolt, illetve a „valós” vonalat a következő módon szemléltethetjük (2.5. ábra):

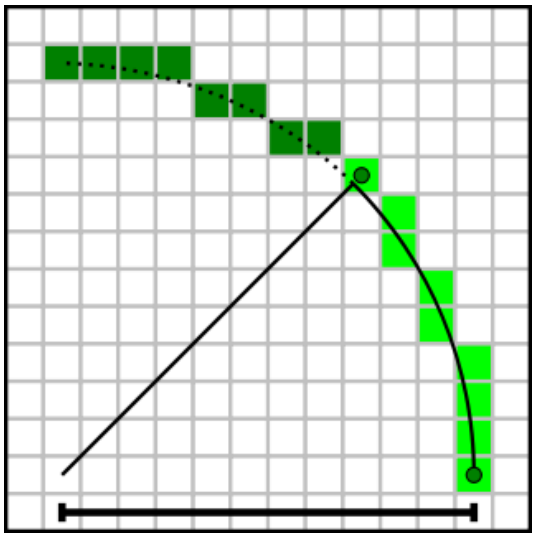


2.5. ábra – Bresenham vonalrajzoló algoritmus

A vonalhoz hasonló alapvető rajzelem a körvonal, melyet például a középpont (C) és a sugár (r) megadásával definiálhatunk. Tekintsük ismét koordinátageometriai egyenletet, és induljunk ki ebből a kör ábrázolásánál, figyelembe véve az egyenes szakasznál már ismertetett megkötést a kijelezhető koordináták vonatkozóan:

$$C = (x_1, y_1)$$
$$(x - x_1)^2 + (y - y_1)^2 = r^2 \qquad (2.3)$$

A körvonal rajzolásakor az egyenes szakaszhoz hasonlóan csak nehezen alkalmazható közvetlenül a (2.3) képlet, ezért ábrázolásukhoz használjuk az úgynevezett középpontos körrajzoló algoritmust (*Midpoint circle algorithm*) [7].



2.6. ábra – Midpoint circle algoritmus

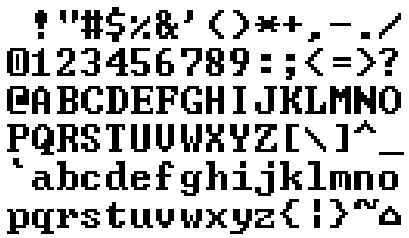
Az algoritmus valójában csak egy nyolcad kör koordinátáinak kiszámításával foglalkozik (2.6. ábra). A teljes körvonal kirajzolásához tükrözi ezt, majd további egyszerű koordináta transzformációkkal az így kapott negyed körökből állítja elő a teljes körvonalat.

Szintén alapvető elemnek tekinthető a téglalap is, amelyre vonatkozóan - az egyszerűség miatt, és a használat gyakoriságából fakadóan - alkalmazzuk a következő megkötést: oldalainak vízszintesnek illetve függőlegesnek kell lenniük. Ekkor a legtömörebben az egyik átló végpontjainak megadásával definiálhatjuk a rajzelemet. Emellett definiáljunk még egy paramétert, amely segítségével azt adhatjuk meg, hogy az ábrázolt téglalap „üres” vagy „teli” legyen. Teli téglalap esetén annak összes pontját, míg üres esetén csak az oldalinak pontjait rajzoljuk ki. A megkötés miatt itt nem kell foglalkoznunk a vonalnál és a körnél figyelembe vett ábrázolási nehézségekkel, a következő egyszerű algoritmusokkal ábrázolhatjuk az üres téglalapokat:

- **if** ($x_1 > x_2$)
  - `swap`(x_1, x_2)
- **if** ($y_1 > y_2$)
  - `swap`(y_1, y_2)
- $y = y_1$ $x = x_1$

- **while** ($x <= x_2$)
  - `plot`(x, y_1) `plot`(x, y_2)
- **while** ($y <= y_2$)
  - `plot`(x_1, y) `plot`(x_2, y)

A legtöbb alkalmazásban szöveg megjelenítésére is szükség van, ezért ennek is alapvető rajzelemnek kell lennie. A szövegeket az alkotó karaktereiknek sorozatával, és annak hosszával adhatjuk meg legegyszerűbben. A különböző karakterek ábrázolásához célszerű egy előre letárolt karaktertáblát létrehoznunk, amely tartalmazza az összes támogatott karakter bittérképét. A karaktertáblát úgy használhatjuk, hogy a megfelelő karakterhez tartozó bittérképet megkeressük, majd a teljes bittérképre a megfelelő sor és oszlop megadásával beillesszük. A szöveg, mint alapvető rajzelem önmagában nem rendelkezik elegendő információval annak kirajzolásához, ezért a grafikus függvénykönyvtár megkötése, hogy a téglalap rajzelemmel együtt létezhet csak. Az így létrehozott szövegdobozok már hordoznak magukban annyi adatot, hogy a megfelelően pozícionálás után kijelezhetjük azokat. A szövegdobozok ebből fakadóan már magasabb szintű rajzelemeknek számítanak. A grafikus függvénykönyvtár része egy 8x12 bites karaktertábla, mely számos különböző karakter kirajzolását biztosítja (2.7. ábra).



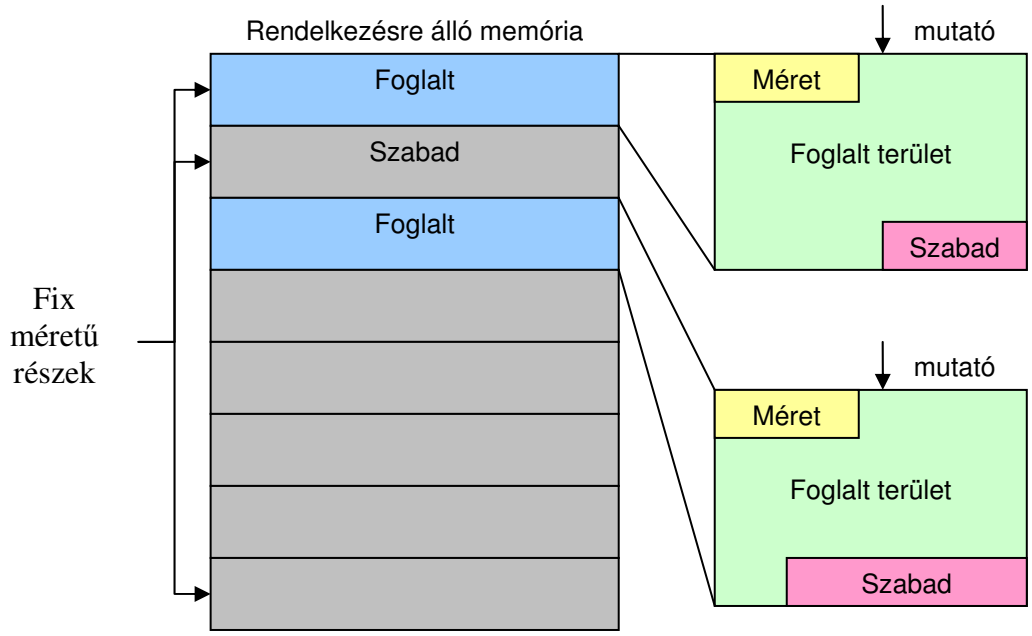
2.7. ábra – A támogatott karakterek

A bemutatott rajzelemekből a moduláris felépítés miatt további rajzelemeket építhetünk ki. Példaként tekintsük az érintőképernyő miatt gyakran használt, ezért a függvénykönyvtár által támogatott, menüelem elnevezésű rajzelemet. Ez egy szövegdobozból, és a hozzá rendelt kezelőfüggvényből áll. Kirajzolása megegyezik a szövegdobozéval, azonban ha a függvénykönyvtár felületén jelezzük, hogy a felhasználó megérintette a kijelzőt, a rendszer ellenőrzi, hogy a megadott koordináták a szövegdoboz téglalapja által létrehozott terület belsejében találhatók-e. Amennyiben ez teljesül, akkor meghívódik a menüelemhez tartozó, az érintést kezelő függvény.

Emellett számos további rajzelem képzelhető el. Az összes jelenleg támogatott rajzelem, és azok paramétereinek leírása a függelék 1. számú mellékletében található.

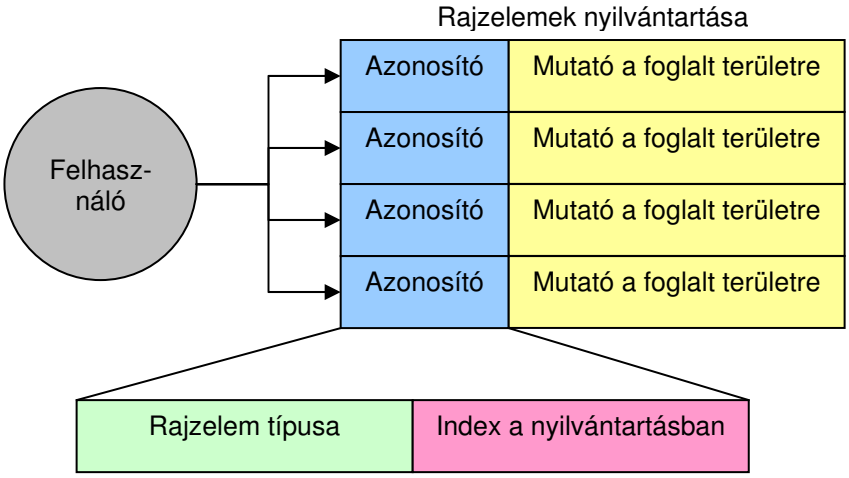
2.4.3. A rajzelemek nyilvántartása

Utolsó lépésként specifikáljuk a rajzelemek tárolásához és nyilvántartásához használt memóriaterületek. A korábban definiált konfigurációs funkció leírásakor bemutattuk, hogy a felhasználó rugalmas állíthatja be igényeinek megfelelően a rajzelemek tárolásához használt memóriatartományt. Erre azért van szükség, mert beágyazott rendszerek esetén kerülendő a futás közben történő dinamikus memóriafoglalás, ehelyett inkább egy előre adott tartományt definiálunk, és ezzel a területtel szabadon gazdálkodunk, így lehetőséget biztosítva a memóriafoglalásra illetve felszabadításra. Fontos tervezési szempont a foglalás és felszabadítás algoritmusa. Esetünkben abból a megfontolásból indulunk ki, hogy az alapvető rajzelemek paraméterei közel azonos méretű területen tárolhatók a típustól függetlenül, így a rendelkezésre álló tartományt fix méretű részekre osztva egyszerű használatot és kis töredezettséget érhetünk el. Könnyen belátható, hogy az alapvető rajzelemekből, mint modulokból felépített magasabb szintű rajzelemek tárolására is teljesül ekkora ez a feltétel. A megoldást a 2.8. ábra segítségével szemléltethetjük legjobban.



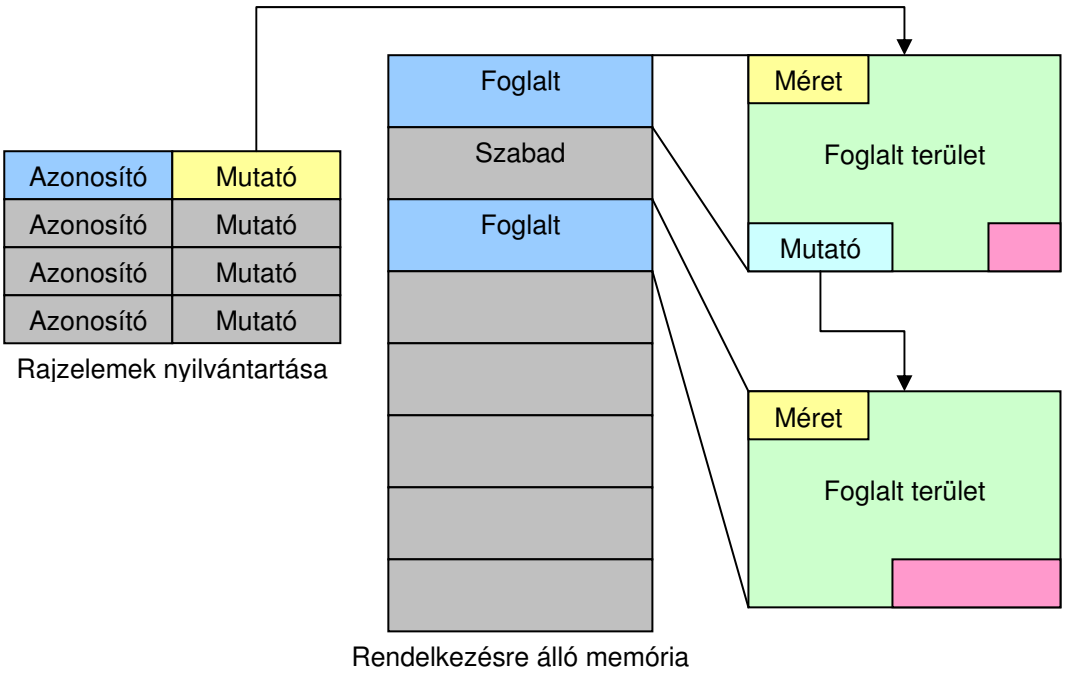
2.8. ábra – Rajzelemek tárolása

Tekintsük például egy memóriafoglalást és felszabadítást. Első lépésként az algoritmus megkeresi az első szabad területet a fix részre osztott memóriatartományon. Szabadnak számít az a terület, melynek az első bájtjának értéke nulla. Ha nem talál ilyet vagy a foglalni kívánt méret nagyobb a maximálisnál, a foglalás sikertelen. Ha talál, akkor a terület első bájtjára bejegyzi a terület méretét és visszaadja a következő bájtra mutató pointert. A felszabadítás jóval egyszerűbben zajlik: a felszabadítandó területre mutató pointertől egy bájtot visszalépünk a méretet megadó területre, és kinullázzuk azt. A felhasználó számára ugyanakkor ezt a belső működést szeretnénk elrejteni, ezért a rajzelemek tárolására fenntartott terület mellett egy a maximális rajzelemek számával egyenlő hosszúságú nyilvántartási tömböt is létrehozunk már az inicializálás során. A tömb egy eleme tartalmaz egy 16 bites azonosítót és egy mutatót a lefoglalt területre. Így a felhasználó az azonosítón keresztül tudja kezelni a rajzelemeket, számára a hozzátartozó mutató nem látható (2.9. ábra).



2.9. ábra – Rajzelemek nyilvántartása

A kezelés egyszerűsítésének érdekében az azonosító is tartalmaz többletinformációkat: magában hordozza a rajzelem típusát és az azonosító indexét is a nyilvántartásban, ezzel egyszerűsítve a keresést. A nyilvántartás megvalósításának része még egy belső paraméter, mely az utolsó érvényes indexet tartalmazza, szintén a kezelés egyszerűsítése érdekében. Magasabb szintű rajzelemek használatakor több memóriaterületre van szükségünk, azonban ilyenkor is az adott elemhez egyértelműen hozzá kell rendelnünk egy azonosítót.



2.10. ábra – Magasabb szintű rajzelemek tárolása és nyilvántartása

Ezt legegyszerűbben úgy tehetjük meg, hogy a fix méretű területekből lefoglalt memóriatartományokat egy mutató segítségével összekapcsoljuk, majd a nyilvántartásban azt a területet rendeljük az azonosítóhoz, amely ezt a mutatót tartalmazza (2.10. ábra).

A szoftver tervezésének utolsó lépésében tehát a szoftverkomponensek specifikációja alapján implementáltuk a megvalósítandó feladatot. A függelék 2. számú mellékletben megtalálható a megírt forráskód a hozzá tartozó angol nyelvű útmutatóval együtt, amely részletesen tartalmazza a felhasználói függvények paramétereit és viselkedését. Az elkészült szoftvert működését természetesen ellenőrizni kell, ez azonban szoftverfejlesztési modellünk másik ágát jelenti, melyet a következő fejezetekben tárgyalunk részletesen.

3. Tesztkörnyezetek

A szoftvertesztelés célja olyan programok, szekvenciák futtatása, melyek segítségével a szoftver hibái a fejlesztők számára láthatóvá válnak. A gyakorlatban a kimerítő tesztelés legtöbbször kivitelezhetetlen, reprezentatív tesztesetekkel azonban a hibák nagy része felderíthető. Fontos mindig figyelembe venni, hogy a tesztelés elsősorban a hibák jelenlétét és nem a hibamentességet tudja megmutatni. Általánosságban három tesztelési metódust különböztethetünk meg: fehérdoboz teszteket, feketedoboz teszteket és a két módszer határán elhelyezkedő szürkedoboz teszteket. A fehérdoboz tesztek esetén ismert a rendszer belső struktúrája, a belső működés végigkövetése a cél. Feketedoboz tesztek esetén csak a külső viselkedés ismert, a specifikált funkciók megléte és hiánya a tesztelés alapja. A szürkedoboz tesztek során mind a belső működés, mind a megvalósított funkciókat vizsgáljuk.

A megvalósított rendszer működésének tesztelésére célszerű tesztkörnyezeteket kiépíteni. A tesztkörnyezet felállítása, karbantartása természetesen pluszfejlesztést, így időt és költséget eredményez, de számos előnnyel jár. Ezek közül talán a legfontosabb, hogy a tesztelési és a fejlesztési feladatok jól elkülöníthetővé válnak. Ha a fejlesztett szoftver egy moduljának implementációja olyan fázisba kerül, hogy érdemes az általa megvalósított funkciókat, illetve a belső működést ellenőrizni, az adott verzió átadható a tesztelőknek. A tesztelés közben a fejlesztők tovább dolgozhatnak más modulokon vagy funkciókon, majd a tesztek lefutása után elemezhetik a korábban kiadott verzióhoz tartozó eredményeket, és javíthatják a felmerült hibákat. A szoftver tervezésének és implementálásának bemutatása után áttérhetünk a feladat igényeinek alapján módosított V-modell tesztelési ágára, a fenti megfontolások segítségével. A fejlesztési folyamat során ennek megfelelően a tervezés és a tesztelés párhuzamosan zajlott, a dokumentum átláthatósága érdekében azonban a két ág külön fejezetben kerül tárgyalásra. A következő alfejezetekben a megvalósított tesztkörnyzeteket mutatjuk be.

3.1. A PC-s tesztkörnyezet működése

A szoftverkomponensek tesztje tipikusan fehérdoboz tesztekkel történik, tehát elsősorban azok belső működésének ellenőrzése a cél (ilyen teszt lehet például a vezérlési gráf analízise a viselkedési modell alapján). A komponensek teszteléséhez célszerű egy megbízható és egyszerűen használható tesztkörnyezet kiépítése, ezért egy PC-n futó alkalmazást valósítottunk meg a működés ellenőrzésére. Ebben a környezetben számos olyan módszer is rendelkezésre áll, amely jelentősen segíti a tesztelést, ugyanakkor beágyazott rendszeren nem, vagy csak nehezen megvalósítható: például az egyes modulok belső tulajdonságainak megjelenítése parancssori interfészen. Előnyt jelent még a tesztelés idejének csökkenése is, mivel a számítógépeken gyorsabban futtatható a megvalósított program, mint a beágyazott rendszeren. Természetesen a számítógépen futó programmal számos, elsősorban magasabb szintű teszt nem végezhető el, illetve a két rendszer különbözőségéből adódóan nem garantált, hogy ami a tesztkörnyezeten működik, az a fejlesztői környezeten is fog (például a fordítók különbsége miatt). A szoftverkomponensek működésének ellenőrzésére az említett hátrányok mellett is alkalmas a környezet, ezért tekintsük át, hogy ezek tesztelését miképp tehetjük meg a PC-n futó alkalmazás segítségével.

Mivel a grafikus függvénykönyvtárunk legfontosabb kimenete az előző fejezetbe ismertetett bittérkép, ezt az információt kell szemléletesen megjelenítenünk a számítógépen futó tesztkörnyezetben. Ezt a legegyszerűbben a bittérkép BMP (*bitmap image file*) fájlformátumba történő tárolásával tehetjük meg [8]. A BMP formátumú fájl a bittérkép mellett egy általános fejlécből és egy információs fejlécből áll (3.1. ábra).

A BMP fájlba való importálás többek között a frame kezelésének tesztelésére használható. A PC-n futó alkalmazás csak RGB888 formátumú bittérképet képes kezelni, és a következő bemenetekkel rendelkezik:

- mutató a megjelenítendő bittérképre
- kép szélessége és magassága
- a BMP fájl neve és elérési útja, amelyben tárolni szeretnénk a bittérképet

BMP fejléc		
Mező név	**Méret**	**Leírás**
Típus	16 bit	Egyedi azonosító
Méret	32 bit	Fájl mérete bájtokban
Foglalt	16 bit	-
Foglalt	16 bit	-
Ofszet	32 bit	Bittérkép ofszete bájtokban
Összesen	*14 bájt*	
Információs fejléc		
Méret	32 bit	Fejléc mérete bájtokban
Szélesség	32 bit	Kép szélessége pixelben
Magasság	32 bit	Kép magassága pixelben
Síkok	16 bit	Síkok száma
Bitek	16 bit	Bitek száma pixelenként
Tömörítés	32 bit	Tömörítési típus
Képméret	32 bit	Kép mérete bájtokban
x felbontás	32 bit	Méterenkénti pixelek száma
y felbontás	32 bit	Méterenkénti pixelek száma
Színek száma	32 bit	Színek száma
Fontos színek	32 bit	Fontos színek
Összesen	*40 bájt*	
Bittérkép		

3.1. ábra – BMP fájl felépítése

Az információs fejlécbe a következő adatok kerülnek:

- méret = információs fejléc mérete
- szélesség = szélesség bemeneti paraméter
- magasság = magasság bemeneti paraméter
- síkok = 1
- bitek = 24
- tömörítés = 0 (nincs tömörítés)
- képméret = szélesség * magasság * 3

- x felbontás = 2835 (adott érték)
- y felbontás = 2835 (adott érték)
- színek = 0 (nem használt)
- fontos színek = 0 (nem használt)

Az ily módon felépített BMP fejlécet és információs fejlécet a bittérképpel együtt a paraméterként megadott elérési út és fájlnév alapján elmenti a program, melyet ezután egy egyszerű képnézegető program segítségével megnyithatunk.

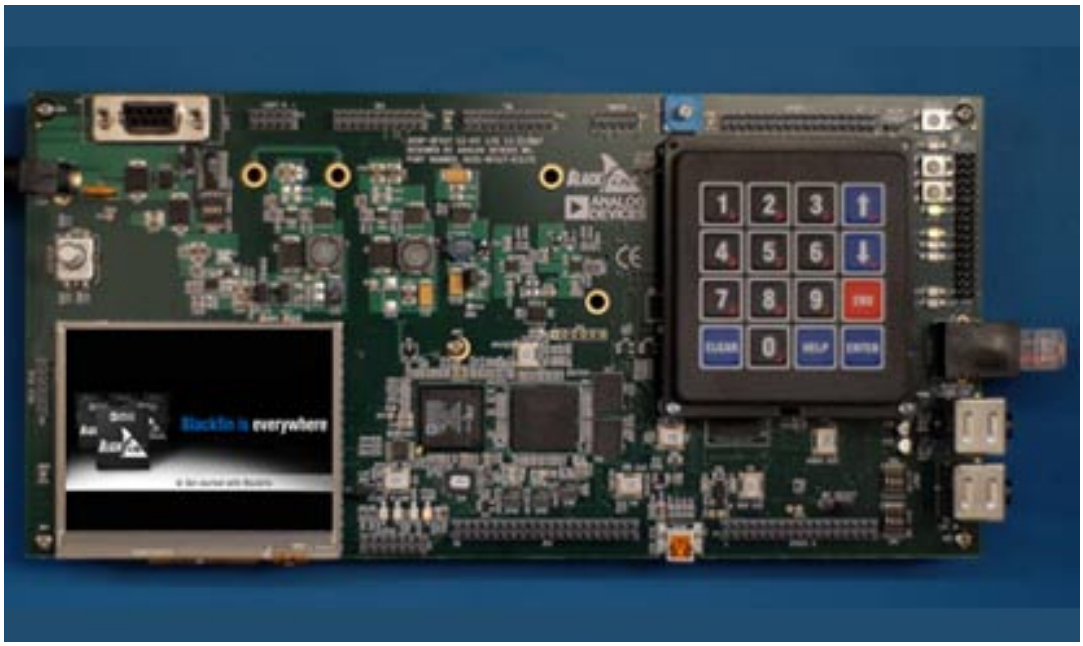
A frame kezelésének tesztelése mellett a rajzelemek nyilvántartásának és tárolásának vizsgálatára is lehetőséget biztosít a tesztkörnyezetünk a rajzelemeket kezelő modul memóriaterületének monitorozásának segítségével. Emellett a belső viselkedés szempontjából lényeges helyekre szöveges megjelenítésre alkalmas segédfunkciókat illesztettünk be, így a konfigurációs modul állapotváltozásai is nyomon követhetőek. Természetesen az érintőképernyő kezelésére vonatkozó teszteket ilyen egyszerűen nem tudunk elvégezni, de összegezve a PC-s tesztkörnyezet segítségével a fejlesztés alatt álló szoftver legtöbb modulját ellenőrizni tudjuk. Az elvégzett teszteket és eredményeiket a következő fejezetben mutatjuk be.

3.2. A fejlesztői kártyán megvalósított környezet

A magasabb szintű tesztesetek végrehajtásához ki kell kiépítenünk a teljes, beágyazott rendszeren futó környezetet. Ez a rendszerintegrációt jelenti, amely legfontosabb feladata, hogy a szoftveres és hardveres elemeket egyesítse. A szürke- és feketedoboz tesztek már a kész rendszeren futnak, a rendszer külvilággal való kapcsolatát tesztelik. A tesztelésnek e szakasza már többé-kevésbé leválasztható a grafikus függvénykönyvtár, azaz a szoftver belső felépítésétől, csak arra vagyunk kíváncsiak, hogy a rendszer úgy viselkedik-e, ahogy elterveztük. Ehhez természetesen szükséges egy másik tesztkörnyezet létrehozása, azaz egy konkrét, beágyazott rendszeren futó alkalmazás. A környezetet a rendelkezésre álló, az *Analog Devices* cég *ADSP BF-527 EZ-KIT Lite* elnevezésű fejlesztői kártyáján valósítottuk meg [9]. A következő alfejezetek mind hardveres, mind szoftveres szemszögből bemutatják a rendszer működését.

3.2.1. Az ADSP-BF527 EZ-KIT Lite fejlesztői kártya

Az Analog Devices ADSP-BF527 EZ-KIT Lite fejlesztői kártyáján egy ADSP-BF527 elnevezésű, fixpontos jelfeldogozó processzor található. A *Blackfin* processzor-család elsősorban audió, videó és kommunikációs alkalmazások megvalósítására használható, igazodva azok fogyasztásbeli, illetve számítási kapacitásra vonatkozó elvárásaihoz. Ezek a processzorok úgynevezett MISC (*media instruction set computing*) architektúrával rendelkeznek: a csökkentett utasításkészletű architektúra (RISC) mellett a digitális jelfeldolgozásra, valamint a multimédiás funkciókra vonatkozó utasításokat is támogatnak.



3.2. ábra - ADSP-BF527 EZ-KIT Lite

A fejlesztői kártya a processzor mellett számos egyéb hardverelemet és interfészt is tartalmaz az általános célú felhasználhatóság érdekében. A legtöbbjük a feladat szempontjából nem lényeges, ezért nem is használjuk őket (a megfelelő panelkapcsolók beállításával kikapcsolhatóak). A fejlesztői kártya processzorának, illetve a későbbiekben használt hardverelemeinek alapvető tulajdonságait a 3.1. táblázat foglalja össze.

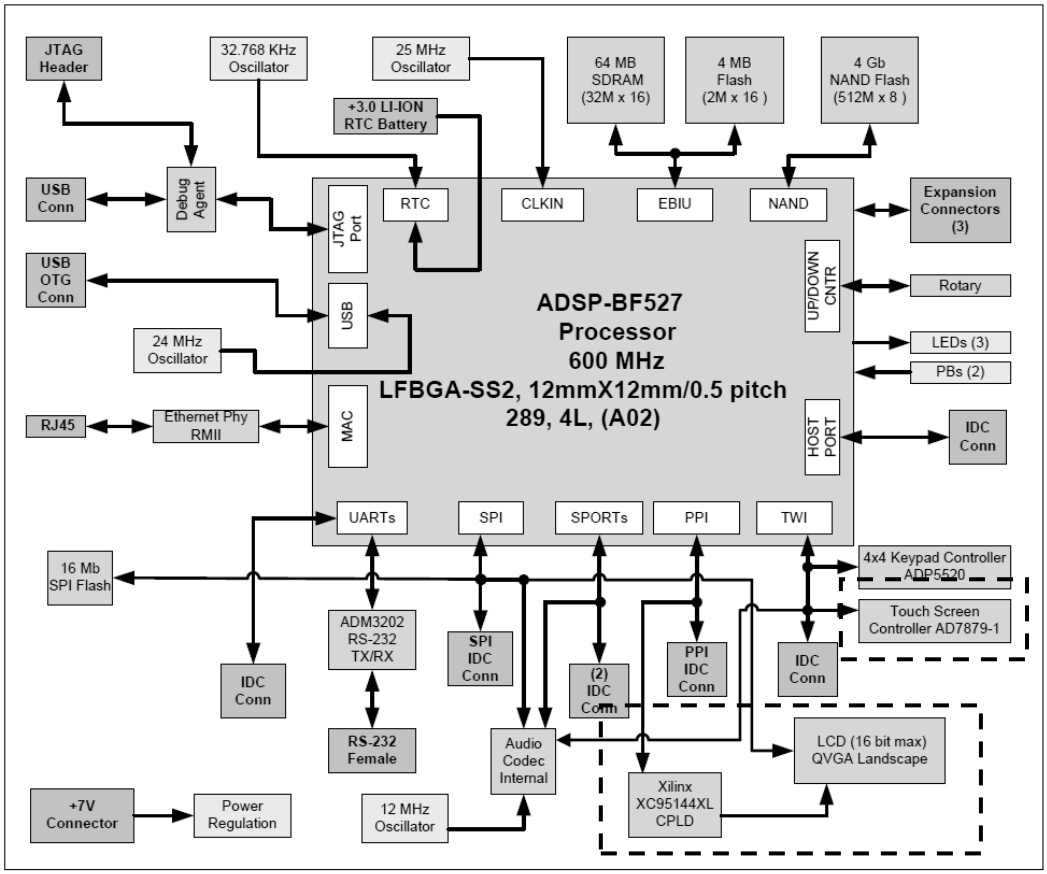
ADSP-BF527 Blackfin processzor	Maximálisan 600 MHz-es *core* frekvencia 133 MHz sebességű periféria busz 289 pin, mini-BGA tokozás 25 MHz-es oszcillátor 132 kB belső memória (L1 SRAM)
SDRAM	64 MB (8M * 16 bit * 4 bank)
TFT LCD kijelző érintőképernyővel	Sharp LQ035Q1DH02 típus Méret: 3,5 hüvelyk Felbontás: 320 x 240 képpont Színmélység: 16 bit AD7879-1 típusú érintőképernyő vezérlő
JTAG interfész	USB csatlakozón keresztül Fejlesztői környezetbe integrált kezelés Futattható kód feltöltése Debuggolási lehetőségek

3.1. táblázat

Futtatható kódot a JTAG interfész segítségével tölthetünk fel a kártyára. Az utóbbi interfész debuggolási lehetőséget is biztosít számunkra, ezzel bővebben a szoftverfejlesztői környezet bemutatásakor foglalkozunk.

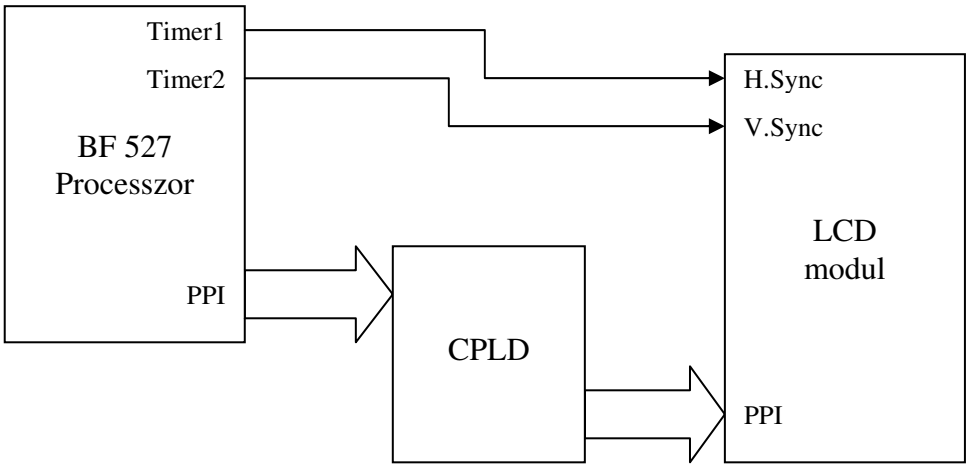
A kijelző adatai alapján kiszámolhatjuk, hogy egyetlen bittérkép tárolásának mekkora a memóriaigénye. Az adott felbontással és színmélységgel számolva és annak eredményét (150 kB) összehasonlítva a belső memória méretével láthatjuk, hogy nagyobb területre lesz szükségünk. A rendszer tartalmaz szinkron, dinamikus RAM-ot (*random access memory*) is, mely sebességéből és méretéből adódóan alkalmas erre a feladatra, így a megvalósításkor ezt a memóriát fogjuk használni a bittérképek tárolására. A tárolás módját a szoftver működését bemutató fejezetben mutatjuk be bővebben.

A rendszer hardveres architektúrája a fejlesztői kártya összes hardverelemét, illetve a processzorral való kapcsolatukat mutatja be (3.3. ábra). A szaggatott vonallal bekeretezett részek alapján látható, hogy az érintőképernyős LCD kijelző esetén valójában két különálló egységről beszélhetünk: a kijelzőről és az érintőképernyő vezérlőjéről.



3.3. ábra – A fejlesztői kártya architektúrája

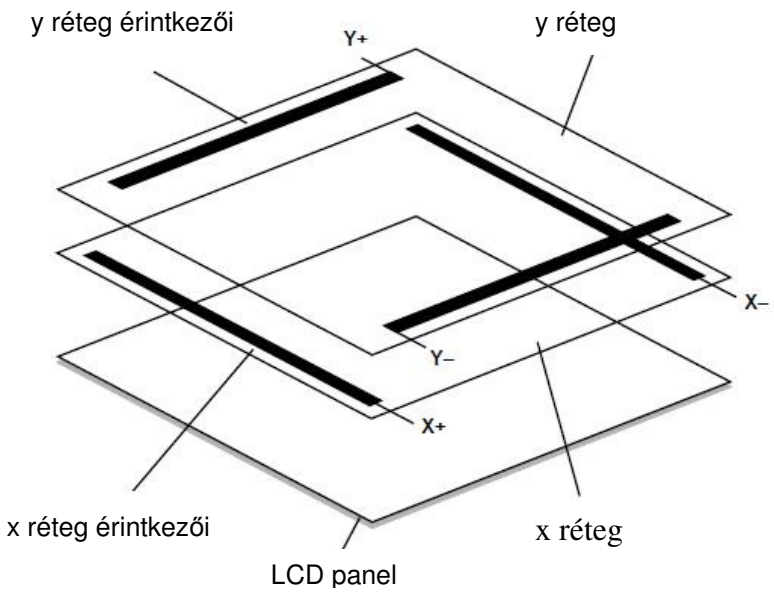
A fejlesztői kártya a Sharp LQ035Q1DH02 típusú érintőképernyős TFT LCD modulját használja. A kijelző 320 pixel széles, magassága 240 pixel és 16 bites színmélységgel rendelkezik. Alapértelmezetten RGB-888 formátumot támogató, soros-párhuzamos interfészen keresztül vezérelhető, tehát először 8 bitnyi piros, zöld, majd kék színinformációt kell kiadnunk az úgynevezett PPI (*parallel peripheral interface*) buszon. A későbbi fejezetben bemutatott Visual DSP++ fejlesztői környezet tartalmaz minden, a kijelzőhöz tartozó konfigurációs adatot. A processzor PPI buszáról érkező adatok valójában egy Xilinx CPLD (*complex programmable logic device*) bemenetei, és ez az eszköz hajtja meg közvetlenül az LCD modult (3.4. ábra). A kijelző további két fontos bemenettel rendelkezik, melyek a képkeret kijelzéséhez biztosítják a szinkronizációt. E két jel segítségével a kijelző vezérlője azonosítja, hogy a kapott adatokat a melyik sorhoz (*horizontal sync*) illetve oszlophoz (*vertical sync*) tartoznak.



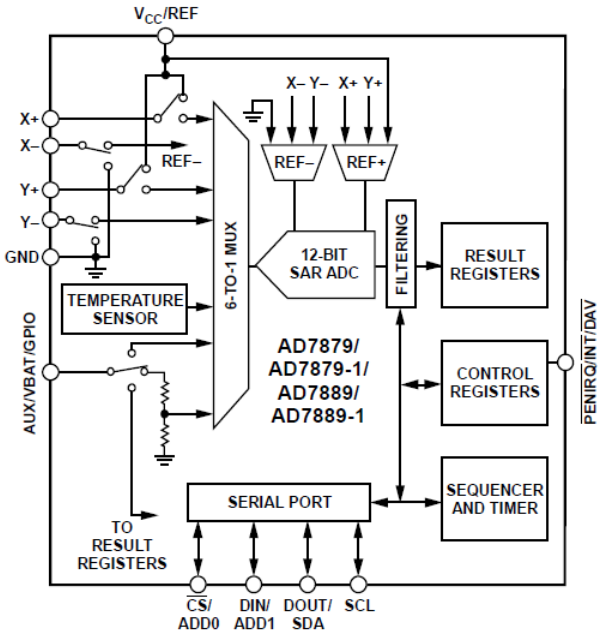
3.4. ábra – Az LCD modul működése

Az érintőképernyő vezérléséről az Analog Devices AD7879-1 típusú kontrollere [10] gondoskodik, amely az úgynevezett TWI (*two-wire interface*) interfészen keresztül kommunikál a processzorral, ez gyakorlatilag I^2C buszon keresztüli összeköttetést jelent. Emellett vezérlőnek van még egy megszakítást kérő, úgynevezett PENIRQ (*pen interrupt request*) kimenete is, amin keresztül értesíti a processzort, ha külső esemény történt. A vezérlő bemeneteit az LCD modul, pontosabban annak nyomásszenzorra biztosítja.

A szenzort két rugalmas és átlátszó vezető lemez valósítja meg, melyek között vékony, levegővel kitöltött réteg biztosítja a szigetelést. Az egyik réteg szolgál az x koordináta, a másik az y koordináta meghatározására, innen eredően x illetve y réteg az elnevezésük. Az egyes rétegek két szemközti oldalán vezető érintkezők vannak a meghatározandó koordinátának megfelelően (3.5. ábra). Ha a kijelzőt megérintjük és x réteget gerjesztjük, az egymással érintkező rétegek feszültségosztást hoznak létre, így y réteg érintkezőin fellépő feszültség alapján meghatározhatjuk a megérintett pont vízszintes koordinátáját. A gerjesztett és a mért érintkezők felcserélésével ezután a függőleges koordináta is meghatározható. Emellett a mért adatokból a nyomás erőssége is meghatározható. Természetesen a mérés csak akkor lehet pontos, ha a két réteg ellenállásának eloszlása a felületek mentén homogén.



3.5. ábra – Az érintőképernyő felépítése



3.6. ábra – Érintőképernyő vezérlője

Az érintőképernyő vezérlőjének blokkvázlata (3.6. ábra) alapján tekintsük át annak működését. A szenzor felépítéséből adódóan az érintőképernyő vezérlőjének a koordináták meghatározására vonatkozóan négy, analóg bemenete van, így a vezérlőnek kell az analóg-digitális átalakítást elvégeznie. Ezt a feladatot egy 12 bites szukcesszív approximációs AD átalakító végzi, melyhez a négy bemenet az LCD modul felől

multiplexereken keresztül csatlakozik. A mérés szempontjából befolyásoló tényező a hőmérséklet, ezért a vezérlő tartalmaz egy ilyen szenzort is. A mért, majd átalakított adatokon a vezérlő lehetőséget biztosít átlagolás és medián szűrés megvalósítására. A vezérlő beállításai, valamint a szűrés és átlagolás paraméterei az I^2C buszon keresztül konfigurálhatók, illetve ezen az interfészen keresztül olvashatók ki a mért eredmények is.

3.2.2. A Visual DSP++ és a VDK operációs rendszer

Az Analog Devices DSP-s fejlesztői kártyái mellé a cég saját, Visual DSP++ szoftverfejlesztő környezetét biztosítja az alkalmazásokat készítők számára [11]. A Visual DSP++ integrált fejlesztői környezet fő része a projektszerkesztő, mely segítségével a rendszeren futó szoftver forrásfájljait módosíthatjuk. A környezet több programozási nyelvet támogat, így a források készülhetnek Assembly, C vagy C++ nyelven is. Az egymáshoz rendelt forrásfájlok egy projektet alkotnak, amelyet a Visual DSP++ fordít le, linkel, majd ezután a processzorra letölthető fájlt generál.

A Visual DSP Kernel (VDK) az Analog Devices saját fejlesztésű, zárt, valósidejű operációs rendszere [12], amely a fejlesztői környezetbe szervesen integrálva van. Ez alatt azt értjük, hogy egyrészt a fejlesztői környezet megvásárlása után a VDK használata már ingyenes, továbbá a fejlesztői környezet lehetőséget biztosít VDK projektek létrehozására, és automatikusan generálja számunkra a szükséges fájlok vázait. A szoros integráció előnye, hogy a során a Visual DSP++ számos külön funkciót biztosít a VDK-t használó fejlesztők számára.

A VDK támogatja mindazon funkciókat, amelyeket minden valósidejű operációs rendszertől elvárhatunk. Létrehozhatunk végrehajtási szálakat (*thread*), melyek általában egy összefüggő feladatot végeznek el. Az egyes szálak függetlenek, egymással elsősorban az operációs rendszer eszközein keresztül kommunikálhatnak. Az objektum orientált programozáshoz hasonlóan szál típusokat definiálhatunk, majd futás során tetszőleges számú threadet hozhatunk létre egy adott típusból. A szálak létrehozhatók induláskor (*boot*), vagy akár futás közben is, de legalább egy szálat már bootoláskor létre kell hozni. A szálak rendelkeznek saját azonosítóval, úgynevezett *ThreadID*-val, melyek segítségével megkülönböztethetjük őket. A példányosított szálak saját *stack*-kel rendelkeznek, így az egyes szálak lokális változói nem keveredhetnek össze. A

megszakításokat kiszolgáló *interrupt* rutinok a VDK-ban nem jelentenek külön szálat, és nem is részei a szálaknak, azoktól külön kezelendők. Minden thread rendelkezik négy alapvető (tag)függvénnyel:

- **InitFunction**

  Az adott típusú szál létrehozásakor, egyszer fut le. Legtöbbször beállítások inicializálására, változók alapértékeinek megadására, továbbá dinamikus memóriafoglalásra használják.

- **DestroyFunction**

  Az adott típusú szál megszűnése előtt közvetlenül fut le. Elsősorban alapértelmezett beállítások visszaállítására, valamint dinamikusan foglalt memóriaterületek felszabadítására használható.

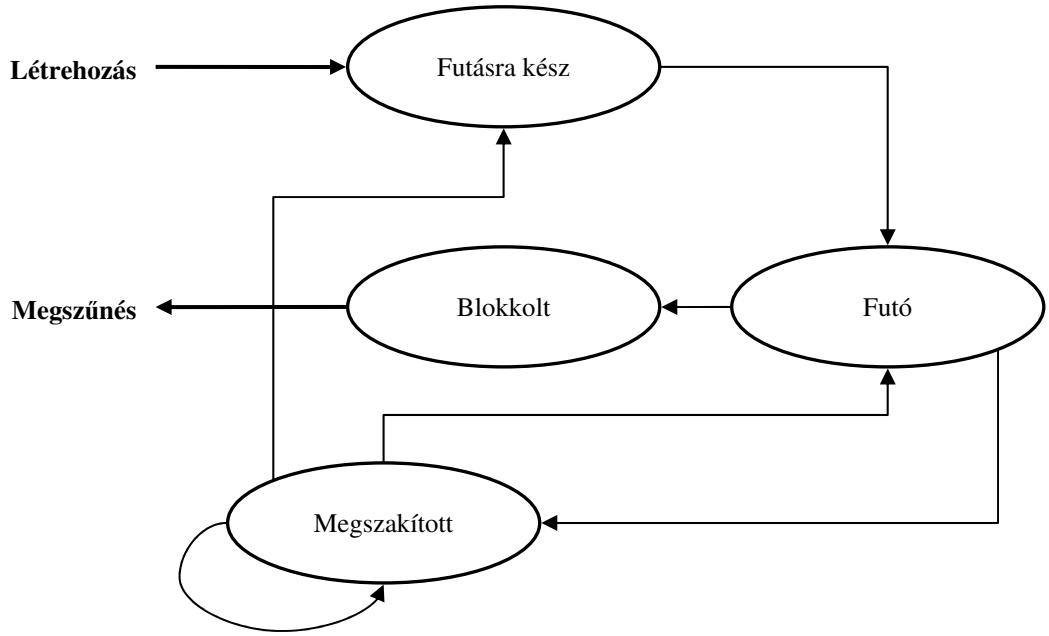
- **RunFunction**

  A szál futási idejének nagy részében ez a függvény fut, amely tartalma rendszerint egy végtelen cikluson belül van. Ha egy szál e függvénye véget ér, akkor a szál is törlődik.

- **ErrorFunction**

  Ha a *RunFunction* függvény futása során hiba lép fel, a VDK ezt a függvény hívja meg, mely a hiba lekezelésére használható.

Megfelelő operációs rendszer hívások segítségével lehetőség van a vezérlés átadására is más szálaknak. Minden VDK projekt létrehozása során automatikusan generálódik egy alapértelmezett, *Idle* elnevezésű szál is, mely akkor fut, ha más szál éppen nincs futásra kész állapotban. A szálakhoz 31 különböző prioritási szintet rendelhetünk hozzá, továbbá lehetnek azonos prioritású szálaink is. A VDK ütemezőjének működése (3.7. ábra) prioritás alapú és preemptív, azaz mindig az a szál kapja meg a processzort, amelyik a futásra kész szálak közül a legmagasabb prioritású, tehát a magasabb prioritású szálak meg szakíthatják az alacsonyabb prioritásúak futását. Amennyiben több azonos prioritású szál is futásra kész állapotban van, a VDK két üzemezési módszert biztosít: kooperatív ütemezést vagy időszelet alapú körforgó ütemezést.



3.7. ábra – Szálak állapotai a VDK-ban

Beágyazott rendszerekről lévén szó, minden alkalmazása fejlesztése során felmerül annak igénye, hogy bizonyos összetartozó műveleteket nem megszakíthatóvá kell tenni. A VDK két megoldást kínál ennek megvalósítására: a nem ütemezett régiót (*unscheduled region*), amely során az ütemező letiltásra kerül, illetve a kritikus régiót (*critical region*), amely során az ütemező letiltásán túl a megszakítások sem kerülnek kiszolgálásra. A szálak közötti szinkronizációra és kommunikációra a VDK több lehetőséget is biztosít. A rendelkezésre álló eszközök az egyéb valósidejű operációs rendszerekben megszokott megoldásokhoz hasonlóak, így a VDK támogatja a szemaforok (*semaphore*), az üzenetek (*message*), a *mutex*ek, az események (*event*) és az eszköz flagek (*device flag*) használatát.

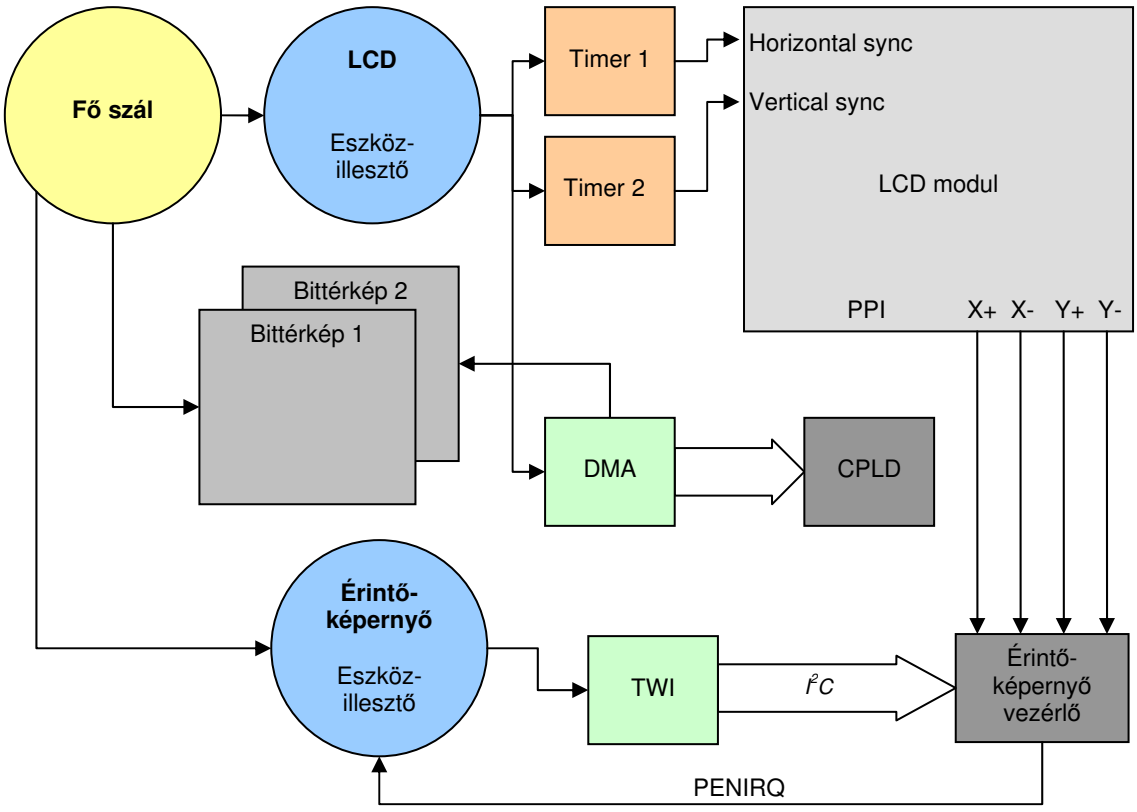
Fontos további, a VDK által biztosított funkció az eszközmeghajtók (*device driver*) fejlesztésének lehetősége, mely a szálak és a „külvilág”, azaz a különböző I/O interfészek közötti kommunikációra szolgál. Az eszközmeghajtók segítségével a hardver típusához szorosan kapcsolódó információkat és a megvalósítás részleteit elrejthetjük más fejlesztők elől, így egy magasabb absztrakciós szintet létrehozva. Az így elkészített eszközmeghajtókat az adott hardveren más projektekhez is felhasználhatjuk, illetve más platformokra portolhatjuk azokat, azonos felhasználói interfésszel.

A Visual DSP++ fejlesztői környezet része a *debugger* is, amely lehetőséget biztosít a megírt alkalmazások tesztelésére futás közben. Segítségével részletesen

vizsgálhatjuk a programok futását például töréspontok elhelyezésével, vagy léptetett utasítás-végrehajtással, továbbá a memória valamint a processzor regisztereinek tartalma is folyamatosan monitorozható. A korábban említett szoros integráció eredményeként a debugger képes a VDK operációs rendszert használó szoftver további vizsgálatára is: áttekinthetjük például a szálak ütemezését, és a különböző események (megszakítások, szemaforok) hatásait arra, az eltelt idő függvényében. Emellett az egyes szálak belső paramétereit (stack használat, állapot, hibaüzenet, stb) is megfigyelhetjük.

3.2.3. Beágyazott szoftver működése

A tesztkörnyezet, azaz a beágyazott szoftver működését az előző fejezetekben bemutatott hardveres és szoftveres megfontolások alapján foglalhatjuk össze. A belső felépítését a 3.8. ábra segítségével mutathatjuk be: a téglalapok a különböző hardverekhez kapcsolódó egységeknek feleltethetőek meg, a körök a VDK segítségével létrehozott szoftveres elemek.



3.8. ábra – Beágyazott szoftver működése

- **Fő szál**

  A fő programszálba használjuk a grafikus függvénykönyvtárt. Itt kell felkonfigurálnunk a megfelelő paraméterekkel és itt kell biztosítanunk számára a memóriaterületeket is. Összesen két bittérképünk van, mindkettőt az SDRAM-ban tároljuk. Az egyiken a függvénykönyvtár segítségével műveleteket végezhetünk: rajzelemeket adhatunk hozzá és törölhetünk, a másikat az eszközillesztő használja a kijelzéshez. Ezzel a megoldással biztosítani tudjuk a kölcsönös kizárást, ugyanis amint az új bittérképet szeretnénk kijelezni, a két memóriaterület megcserélődik: az kerül megjelenítésre az LCD panelen, amelyen eddig dolgoztunk, az eddig megjelenített területen pedig a függvénykönytvár segítségével a fő programszálból szerkeszthetjük. Minden egyes ciklusban ellenőrizzük, hogy a felhasználó megérintette-e a képernyőt, és ha igen, meghívjuk a függvénykönyvtár megfelelő kezelőfüggvényét. További feladata az eszközillesztők létrehozása és incializálása.

- **LCD eszközillesztő**

  Az eszközillesztőhöz három hardveres egység csatlakozik: két időzítő, mellyekkel a kijelzéshez szükséges szinkronizációs jeleket biztosítjuk, illetve egy DMA vezérlő, mely segítségével az aktuálisan megjelenítendő bittérképet PPI interfészen keresztül az LCD modulnak elküldjük. Ebbe a kapcsolatba egy programozható komplex logika is beékelődik a korábban említetteknek megfelelően, mely a különböző típusú formátumok közötti átalakítást végzi. A függvénykönyvtár ugyanis RGB-888 formátumú bittérképet hoz létre, ezt továbbítja a CPLD felé, a kijelző viszont a 16 bites színmélységű RGB-565 formátumot támogatja. A DMA vezérlő úgy van felkonfigurálva, hogy az LCD képfrissítési frekvenciájának megfelelően periodikusan küldje az adatokat a PPI buszon keresztül (tehát valójában ehhez az eszközillesztőhöz is tartozik megszakítás minden egyes DMA ciklus végén).

- **Érintőképernyő eszközillesztő**

  Az eszközillesztőhöz egy hardveres egység csatlakozik, a processzor TWI vezérlője, mely segítségével az érintőképernyő vezérlőjével kommunikálhatunk I^2C buszon keresztül. Inicializáláskor, azaz amikor a fő szál példányosítja az eszközillesztőt, történik meg az érintőképernyő beállításainak megadása: a

megszakítások engedélyezése előtt beállítjuk a szűrési és átlagolási paramétereket, továbbá töröljük az adatregisztereket. Ezután engedélyezzük a megszakítást, mely akkor keletkezik, amikor a felhasználó az érintés után elengedi az érintőképernyőt. Ezután az eszközillesztőhöz tartozó megszakításkezelő rutin megvizsgálja, hogy milyen értékű nyomást jelzett a kontroller. Amennyiben ez az érték meghalad egy bizonyos határt kiolvassuk a koordinátákat, majd azokat a megfelelő tartományba konvertáljuk. A minimális kvantálási hiba miatt ugyanis a vezérlő a teljes 12 bites tartományt kihasználja, azaz a $2^{12} - 1$ érték (4095) jelenti x koordináta esetén a 320. vízszintes pontot, y esetén pedig a 240. függőleges pontot. A transzformáció után a megszakításkezelő szemafor segítségével jelzi az eszközillesztőn keresztül a fő szálnak, hogy érintés történt.

4. A szoftver tesztelése és illesztése konkrét alkalmazáshoz

A feladathoz igazított és módosított V-modellünk tesztelési ágával foglalkozik a következő fejezet. A szoftvertesztelés során olyan programokat futtatunk a rendszeren, melyek segítségével a szoftver hibái láthatóvá válnak számunkra. A tesztek sikeres lefutása nem garantálja a hibamentességet, a tesztelés célja kizárólag a hibák felderítése.

A különböző absztrakciós szinteken különböző tesztelési módszereket használunk: alacsony szinten fehérdoboz teszteket, magasabb szinteken többségében szürke-, illetve feketedoboz teszteket. A megvalósított rendszer működésének tesztelésére tesztkörnyezeteket építettünk ki. A tesztelési ág első lépésében, azaz a komponensek tesztjénél tipikusan fehérdoboz teszteket kell végrehajtanunk, mivel elsősorban azok belső működésének ellenőrzése a cél. A komponensek teszteléséhez egy PC-n futó alkalmazást valósítottunk meg, mely egy megbízható és egyszerűen használható tesztkörnyezetet biztosított a működés ellenőrzésére. A számítógépen futó programmal a magasabb szintű tesztek legtöbbször nem végezhetőek el, ezért szükségünk van egy beágyazott rendszeren futó környezetre is. Ezzel a lépéssel a rendszerintegráció feladatát végeztük el, mely a modulok tesztelése utáni következő lépését jelenti a tesztelési modellünknek. Legfontosabb feladata az, hogy a szoftveres és hardveres elemeket egyesítse, valamint szürke- és feketedoboz tesztekkel a kész rendszer működését és külvilággal való kapcsolatát ellenőrizzék.

4.1. Szoftverkomponensek tesztelése és integrációja

A szoftver komponensek tesztelésekor elsősorban azok belső működésének ellenőrzése a cél. A megvalósított grafikus függvénykönyvtár felépítése moduláris, ezért a különböző szoftvermodulokat külön tesztelhetjük. Az egyes modulokra vonatkozóan teszteseteket fogalmazhatunk meg. A fejezetben elsősorban az elképzelhető tesztesetek és nem a konkrét eredményeiknek bemutatása a cél. A tesztek célja a hibák felderítése, ezért azokra az esetekre kell koncentrálnunk, amelyek ilyen szempontból kritikusak. Első lépésben a rajzelemek tárolását és nyilvántartását mutatjuk be egy példán

keresztül. A rajzelemeket kezelő modult a következő bemeneti paraméterekkel inicializáljuk: 40 byte méretű buffer, 20 byte méretű memóriaegységek (lépésköz). Ezután a grafikus függvénykönyvtár segítségével a következő lépéseket végezzük el:

1. Inicializálás a fenti paraméterekkel
2. Első megfelelő rajzelem hozzáadása
3. Túl nagy méretű rajzelem hozzáadása
4. Második megfelelő rajzelem hozzáadása
5. Első hozzáadott rajzelem törlése
6. Második hozzáadott rajzelem rejtése
7. Harmadik megfelelő rajzelem hozzáadása

A bemutatott szekvencia végrehajtásával a rajzelemek tárolására és nyilvántartására vonatkozó belső működés legtöbb esetét és elágazását lefedhetjük. Az egyes lépések eredményei a következők:

Inicializálás

```
Display object created.
Display object initialized.
Pool size:          40 bytes
Pool stride:        20 bytes
```

Első megfelelő rajzelem hozzáadása után

```
Result = 0          ID#0: 0x0200
1. element - State: Shown  - ID: 0x0200 - Pointer: 0x001F7B11
-- Pool structure --
Block # 0:
10 01 00 01 00 1E 00 1E
00 00 00 00 00 00 00 00
00 00 00 00
Block # 1:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
```

Nem megfelelő méretű rajzelem hozzáadása után

```
Result = 3          ID#1: 0xFFFF
```

Második megfelelő rajzelem hozzáadása után

```
Result = 0         ID#2: 0x0101
1. element - State: Shown  - ID: 0x0200 - Pointer: 0x001F7B11
2. element - State: Shown  - ID: 0x0101 - Pointer: 0x001F7B25
-- Pool structure --
Block # 0:
10 01 00 01 00 1E 00 1E
00 00 00 00 00 00 00 00
00 00 00 00
Block # 1:
0C 01 00 01 00 1E 00 1E
00 00 00 00 00 00 00 00
00 00 00 00
```

Első hozzáadott rajzelem törlése után:

```
Result = 0         ID#0: 0x0200
1. element - State: Free
2. element - State: Shown  - ID: 0x0101 - Pointer: 0x001F7B25
```

Második hozzáadott rajzelem rejtése

```
Result = 0         ID#2: 0x0101
1. element - State: Free
2. element - State: Hidden - ID: 0x0101 - Pointer: 0x001F7B25
```

Harmadik megfelelő rajzelem hozzáadása

```
Result = 0         ID#4: 0x0300
1. element - State: Shown  - ID: 0x0300 - Pointer: 0x001F7B11
2. element - State: Hidden - ID: 0x0101 - Pointer: 0x001F7B25
-- Pool structure --
Block # 0:
0C 64 00 64 00 1E 00 00
00 00 00 00 00 00 00 00
00 00 00 00
Block # 1:
0C 01 00 01 00 1E 00 1E
00 00 00 00 00 00 00 00
00 00 00 00
```

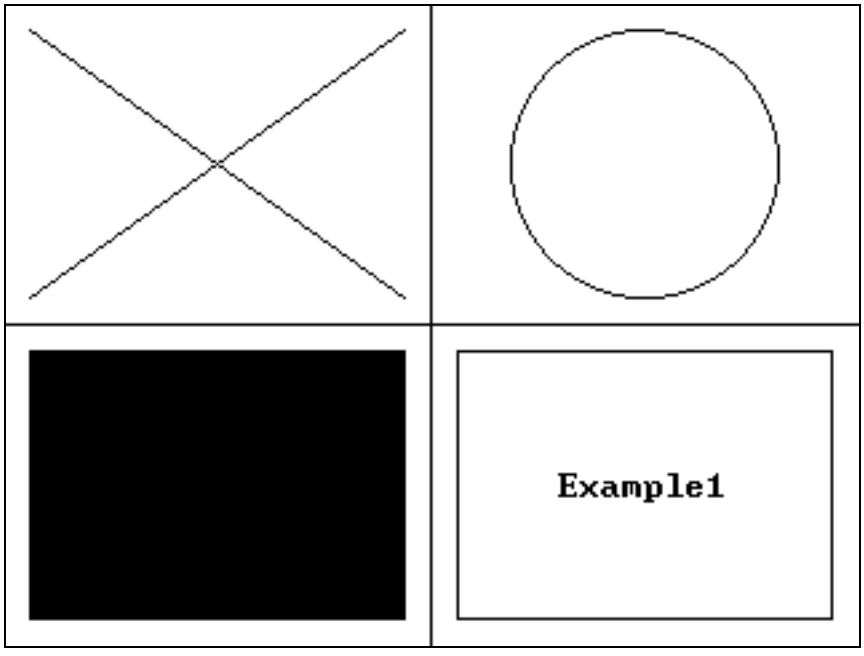
A PC-s tesztkörnyezet által biztosított, a modul belső működését szemléltető információk segítségével látható, hogy a szekvencia sikeresen lezajlott. Magasabb szinten a rajzelemeket kezelő modul használata során a további eseteket merülhetnek fel, amelyek lekezelését ellenőriznünk kellett:

- **Rajzelemek hozzáadása**
  - megfelelő rajzelem
  - inicializálás előtti hozzáadás
  - nem létező típus
  - nem megfelelő méretű rajzelem
  - túl sok rajzelem hozzáadása
  - rossz paraméterekkel rendelkező rajzelem
  - rosszul felszabadított területen tárolás
- **Rajzelemek törlése**
  - létező rajzelem
  - inicializálás előtti törlés
  - nem létező típus
  - nem hozzáadott rajzelem
  - rosszul tárolt rajzelem
- **Rajzelemek módosítása**
  - megfelelő rajzelem
  - inicializálás előtti módosítás
  - nem létező típus
  - nem hozzáadott rajzelem
  - nem változtatható rajzelem
  - rossz paraméterekkel rendelkező új rajzelem
- **Rajzelemek rejtése és felfedése**
  - megfelelő rajzelemek
  - nem hozzáadott rajzelemek

A képkeretet kezelő modul viselkedését a bittérkép BMP fájlformátumba történő tárolásával ellenőrizzük. Példaként a következő tesztesetet vizsgáljuk meg: a konfigurációs modell segítségével beállítjuk a megfelelő paramétereket, hozzáadunk különböző rajzelemeket a nyilvántartásához, majd megjelenítjük azokat a „kijelzőn”. A

frame mérete 320x240 képpont, formátuma RGB888, háttérszíne fehér, a hozzáadott rajzelemek fekete színűek:

1. Inicializálás a megadott paraméterekkel
2. Vonalak hozzáadása, a végpontok:
   - (1,120) és (320,120)
   - (160,1) és (160,240)
   - (10,10) és (150. 110)
   - (10,110) és (150,10)
3. Kör hozzáadása, paraméterei:
   - középpont (240, 60)
   - sugár 50 képpont
4. Teli téglalap hozzáadása, átlók végpontjai:
   - (10,130) és (150,230)
5. Szövegdoboz hozzáadása, paraméterei:
   - téglalap (170,130) és (310,230)
   - szöveg Example1



4.1. ábra – Frame kezelésének tesztje

A lépések végrehajtásának eredménye alapján látható, hogy a framekezelő modul megfelelő módon működik (4.1. ábra). A bemutatott példa mellett számos

további teszteset képzelhető el: a framekezelő modult csak úgy tesztelhetjük részletesen, hogy az összes rendelkezésre álló rajzelemet, a lehető legtöbb paraméter-kombinációval hozzáadjuk a bittérképhez, majd megvizsgáljuk a kapott eredményeket.

Az inicializáló modul tesztelése a többi modulhoz képest egyszerűen végrehajtható, hiszen ennek belső működése egyszerű: a bemenő paraméterek helyességét ellenőrizzük, és ezek alapján a modul állapotváltozóját módosítjuk. Az előbbiekben részletesen vizsgált modulok tesztelése során el is végeztük ezt a feladatot, ugyanis a fenti tesztesetek csak akkor futnak le megfelelően, ha az inicializáló modul is a definiáltak szerint működik. Az érintőképernyőt kezelő modul tesztelése a többi modultól eltérő, a PC-s környezet erre nem ad lehetőséget, a beágyazott rendszeren kell ezt megvizsgálnunk.

4.2. A konkrét alkalmazás bemutatása

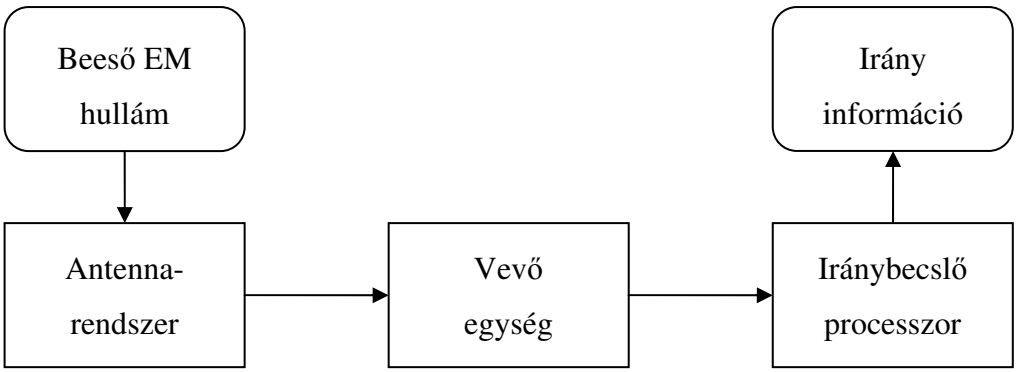
A szoftverkomponensek részletes tesztelése és a felmerült hibák javítása után áttérünk a módosított V-modellünk következő lépésére. A tesztelési folyamat e szakaszában a rendszerintegrációval és a rendszer tesztelésével foglalkozunk. A rendszerintegráció legfontosabb feladata, hogy a szoftveres és hardveres elemeket egyesítse, a tipikusan szürke- és feketedoboz tesztek már a kész rendszeren futnak, és a rendszer külvilággal való kapcsolatát tesztelik. Ebben a lépésben már nem vizsgáljuk a grafikus függvénykönyvtár belső felépítését, a tesztelés magasabb absztrakciós szinten folytatódik. A tesztelési feladat megvalósítására egy konkrét beágyazott rendszeren futó környezet szükséges, ennek szerepét a korábban már bemutatott DSP-s fejlesztői kártya tölti be. Mivel a grafikus függvénykönyvtár független a szoftver többi elemétől, egy konkrét alkalmazást implementáltunk a környezetben. Ennek segítségével megvizsgálhatjuk az elkészült szoftver működését. A választott alkalmazás esetünkben egy egycsatornás iránymérő rendszer.

4.2.1. Iránymérő rendszerek

Az iránymérő egy olyan eszköz, amely képes az antennarendszerére beérkező elektromágneses hullám beesési szögének meghatározására. Az iránymérő rendszerek három alapvető részegységből állnak (4.2. ábra):

- Antennarendszer
- Vevő egység
- Iránybecslő processzor

Az iránymérő antenna felépítését tekintve lehet egy antennából álló, vagy több antennából álló antennarendszer is. Az antennarendszer feladata, hogy a beeső elektromágneses hullámot olyan vezetett hullámmá alakítsa, amely a beesés irányára vonatkozó információt is tartalmaz.



4.2. ábra –Iránymérő rendszerek felépítése

A vevőegység erősíti az antennarendszer kimeneti jelét és a megfelelő frekvenciatartományba keveri az iránybecslő processzor bemenetének megfelelően. A bemenet lehet egy-, vagy többcsatornás is, esetünkben elsősorban az egycsatornás megoldásokat ismertetjük. Az iránybecslő processzor a bemenetére adott jelből megbecsüli a beeső elektromágneses hullám beesési irányát. Ez történhet analóg úton is, de korunkban a digitális megoldások az elterjedtebbek, elsősorban a kedvező ár-érték arányuk és a pontosságuk miatt. Az iránymérő berendezéseket a megvalósított mérési elvek alapján csoportosíthatjuk. Az antennarendszerre beeső hullám az egyes antennaelemekben feszültséget indukál, és a kimeneti feszültség jellemzőinek alapján

következtetünk a beesés irányára. Ezek alapján a következő iránymérési elvekről beszélhetünk:

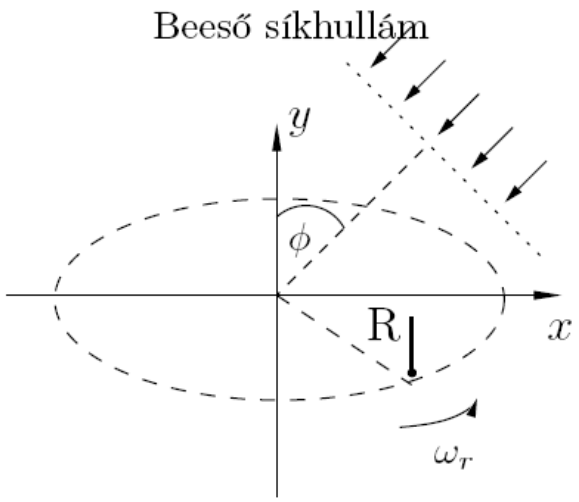
- amplitúdó mérésen alapuló,
- fáziskésleltetésen alapuló.

Az előbbi csoportosítás alapján a Watson-Watt eljárás az összehasonlító amplitúdó mérésen alapuló iránymérési elvek kategóriájába sorolható. Az eljárásnál legalább kettő, egymásra merőleges, nyolcas alakú iránykarakterisztikával rendelkező antennát használnak. Az antennák kimenetei jeleinek amplitúdóját mérik, amely alapján meghatározzák az elektromágneses hullám beesési irányára. Elterjedt megoldás a Watson-Watt iránymérő eljárás egycsatornás változata, amely esetén az egyes antennák kimeneti jeleit modulálják, és a modulált jelet vezetik be az egycsatornás vevő egységbe. Az eljáráshoz használatos nyolcas alakú iránykarakterisztikával a hurok, és az ún. Adcock antenna is rendelkezik, de az utóbbi típus gyakrabban használatos kedvezőbb paraméterei miatt.

Az iránymérő rendszerek számos egyéb fajtája létezik, közülük az úgynevezett pszeudo-Doppler módszert valósítottuk meg [13] [14]. Az eljárás részleteivel és megvalósítási módjával a következő fejezet foglalkozik.

4.2.2. A pszeudo-Doppler eljárás

A Doppler iránymérő elv alapja a Doppler-hatás, amely szerint a megfigyelőhöz képest mozgó hullámforrás által sugárzott hullám frekvenciája megváltozik a mozgás irányától függően: ha a megfigyelő közeledik a hullámforráshoz a mért frekvencia nő, ha távolodik a mért frekvencia csökken. Ennek alapján az iránymérő alapötlete a következő: forgassunk egy antennát R sugarú körpályán, ekkor ugyanis az antenna kapcsain megjelenő jel pillanatnyi frekvenciája periodikusan változik a forgatási sebesség és az elektromágneses hullám beesési szögétől függően (4.3. ábra).

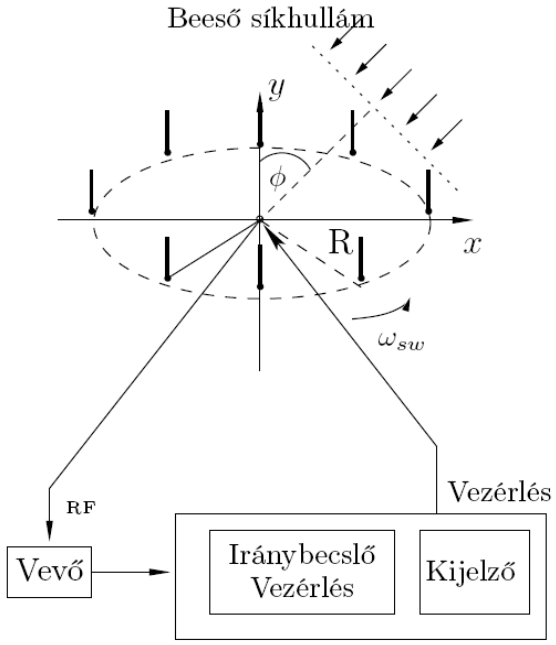


4.3. ábra – Doppler iránymérő alapelve

Maximális frekvencia eltolódást akkor figyelhetünk meg, ha a hullám beesési szögének vektora, és a pillanatnyi kerületi sebesség vektora párhuzamos egymással. A pillanatnyi frekvencia eltolódása ily módon f_0, azaz a beeső EM hullám frekvenciája körül ingadozik, vagyis az antenna kimeneti jele egy frekvenciamodulált jel. A maximális frekvencialöketet a körpálya sugara (R), a beeső elektromágneses hullám frekvenciája és a forgatási frekvencia határozza meg. Megfelelő frekvencialöket elérését a forgatási frekvenciával befolyásolhatjuk. Mivel a mechanikus forgatás ilyen sebességeknél már nagyon nehezen megvalósítható, a forgatást több, körpályán egyenletesen elhelyezett antenna periodikus kapcsolgatásával szimulálják. Innen ered a mérési rendszer elnevezése is: a pszeudo-Doppler eljárás. Az elektronikusan forgatott antenna kimenetén megjelenő jel időfüggvénye tehát a következő:

$$s(t) \;=\; a(t)\cos\left(\omega_0 t + \frac{2\pi R}{\lambda_0}\cos\left(\omega_r t - \phi\right) + \varphi\right) \tag{4.1}$$

A (4.1) képletben φ jelenti a jelben a forrástól az antennarendszerig bekövetkező fázistolást, a(t) a modulált jel amplitúdója, ω_0 és λ_0 a beeső EM hullám körfrekvenciája és hullámhossza, ω_r a „forgatás” körfrekvenciája, R pedig a körpálya sugara. A pszeudo-Doppler elven működő iránymérő berendezés felépítése a következő a 4.4. ábra látható.



4.4. ábra – Pszeudo-Doppler iránymérő felépítése

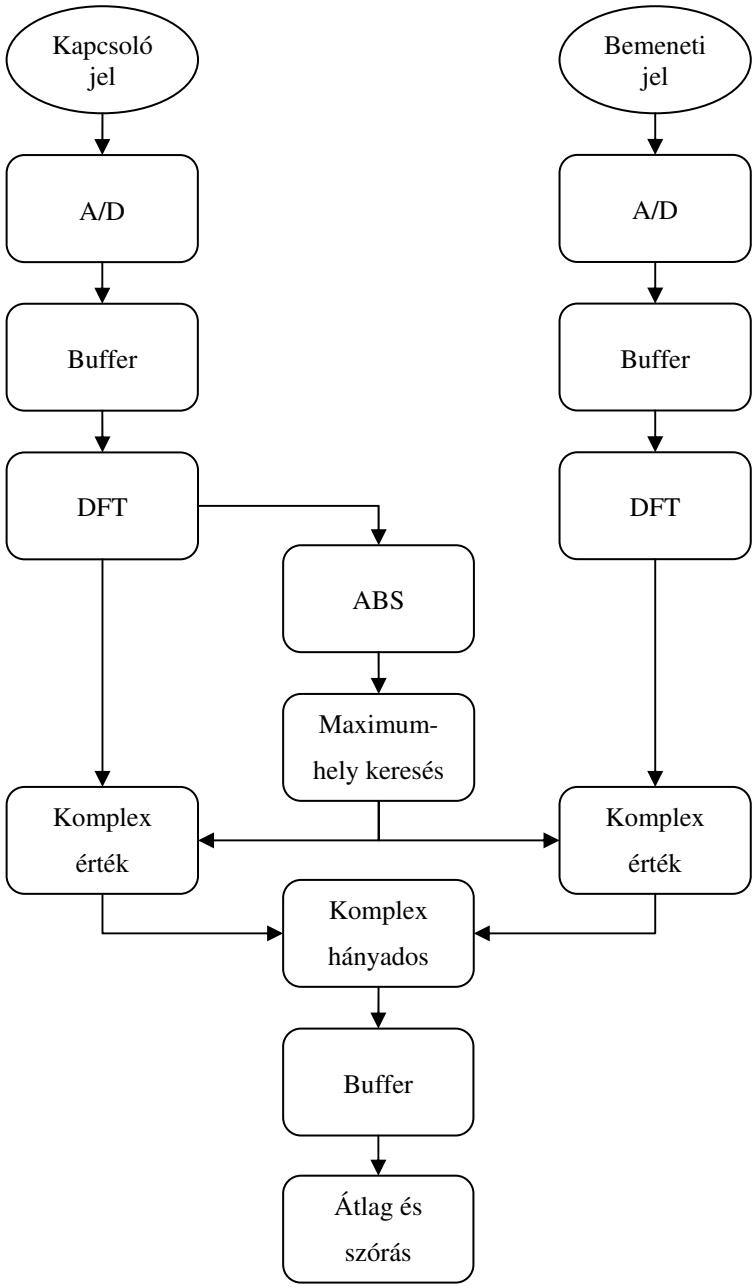
Az antennarendszer több antennából, és az antennák egymás utáni kapcsolgatását megvalósító áramkörből áll. A kapcsoló jel segítségével egyszerre csak egy antenna jelét vezetjük a vevő egység bementére. Az egyértelmű irányméréshez legalább három antennaelem szükséges, de gyakran négy, nyolc, tizenhat vagy akár harminckét antennaelemből álló rendszert is használnak. Az egyes antennaelemek kimeneti jelei azonos frekvenciájúak, csak a fázisuk változik. További feltétel az antennarendszerre vonatkozóan, hogy a szomszédos antennák távolságának kisebbnek kell lennie, mint a maximális frekvenciához tartozó hullámhossz fele.

Az iránymérő vevőberendezés a frekvencia megfelelő tartományba való transzformálását megvalósító keverő áramkör mellett egy szűrőt, valamint egy FM demodulátort is tartalmaz.. A vevő egységnek két fontos követelményt kell teljesítenie: sávszélességének nagyobbnak kell lennie, mint az FM moduláció frekvencialökete, illetve az áteresztő sávhoz tartozó fáziskarakterisztikájának lineárisnak kell lennie. A vevőegység kimenete ezek után felírható a (4.1) képlet alapján:

$$s_D(t) = \frac{2\pi R}{\lambda_0}\omega_r \sin(\omega_r t - \phi) \tag{4.2}$$

Az iránybecslő processzor feladata vezérelni az antennarendszer kapcsolgatását és megbecsülni az EM hullám beesési szögét. A beesési irány a kapcsoló jel és a vevő

berendezés kimeneti jele közötti fáziskülönbség mérésével határozható meg. Fontos megjegyezni, hogy az algoritmus bonyolultságát az antennaelemek száma nem befolyásolja, csak az algoritmus pontosságára van hatással. A jelfeldolgozás lépései a következő ábrán szemléltethetők:

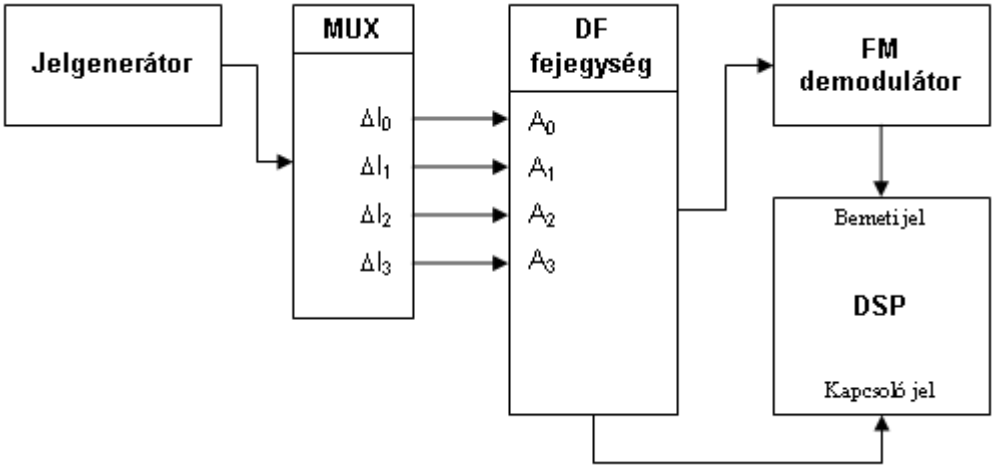


4.5. ábra – A megvalósított iránymérő algoritmus

Első lépésben meg kell határoznunk a kapcsoló jel alapharmonikusának frekvenciáját, mely a frekvenciatartománybeli amplitúdóspektrumon történő maximum-kereséssel valósítunk meg. A maximum értékhez tartozó maximumhely indexét a diszkrét frekvenciatengelyen ez alapján már meghatározhatjuk. A maximumhelyhez

tartozó komplex értékek hányadosát képezve a két jel fáziskülönbségére vonatkozó információt hordozó komplex számot kapunk, melyet ezután normálunk, ugyanis számunkra csak a komplex vektor szöge a hasznos információ. A feldolgozás hosszától függően ennek eredményeként normált, komplex vektorokat kapunk. Az így kapott komplex számok mindegyike egy-egy mérési eredményt jelent, melyekből átlagot képzünk, majd a mérés hatékonyságát jellemző információt számítunk.

Az algoritmus tesztelésére szükségünk van konkrét mérési adatokra is, amelyeket a 4.6. ábra látható mérési összeállítással kaptunk meg.



4.6. ábra – A mérőrendszer felépítése

A jelgenerátor feladata a multiplexer segítségével az antennarendszer kimenetének szimulálása. Ezt különböző késleltetések beiktatásával biztosítja (Δl_n). A megoldással a 4.1. táblázatban látható mérési eredmények álltak rendelkezésünkre.

Jelszint	Beesési szög	Jelszint	Beesési szög
-40 dBm	0°	-40 dBm	315°
-50 dBm	0°	-50 dBm	315°
-60 dBm	0°	-60 dBm	315°
-70 dBm	0°	-70 dBm	315°

4.1. táblázat – Az antennarendszer szimulált kimenetei

A szimulált antennarendszer kimenetét az iránymérő fejegységének (*Sagax SDF-3000*) bemeneteire kötöttünk (A_n). A mérési adatokat rögzítő DSP egyik bemenete a fejegység kapcsolójele, a másik az FM demodulátoron keresztül lekevert jel volt.

A jelek frekvenciatartománybeli előállítását diszkét Fourier transzformáció (DFT) segítségével kaphatjuk meg. A transzformáció legfontosabb paramétere a pontok száma, amely befolyásolja az algoritmus számításigényét, ezáltal pedig a végrehajtási időt. Ez a paraméter a bufferünk hosszával egyezik meg, és az előző megfontolások miatt célszerű az algoritmus konfigurálható paraméterévé tenni. Ha a Fourier transzformáció pontszáma N, és N*m minta érkezett az algoritmus bemenetére, akkor összesen m számú mérési eredményt kapunk. Ebből következően a másik konfigurálható paraméterünk a mérési eredmények száma lesz, amely segítségével az átlag és szórás értékét számíthatjuk ki. Összefoglalva tehát a következő képletekkel adhatjuk meg az egyes lépéseket:

1. **N hosszúságú bufferek alapján diszkrét Fourier transzformáltak kiszámítása a kapcsolójelre és a bemeneti jelre**

$$x(n) = Mintavételezett\ kapcsolójel$$
$$X(k) = \sum_{k=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi}{N}kn} \tag{4.3}$$
$$y(n) = Mintavételezett\ bemeneti\ jel$$
$$Y(k) = \sum_{k=0}^{N-1} y(n) \cdot e^{-j\frac{2\pi}{N}kn} \tag{4.4}$$

2. **Maximum keresése a kapcsoló jel transzformáltjának abszolutértékeinek sorozatán**

$$Max\left(|X(k)|\right) \qquad k = 0, \ldots, N-1 \tag{4.5}$$

3. **A maximumhely indexéhez tartozó komplex vektorok elosztása és normálása**

$$i = Maximumhely\ indexe$$
$$\frac{X(i)}{Y(i)} \cdot \left|\frac{X(i)}{Y(i)}\right|^{-1} \tag{4.6}$$

4. **Az első három lépés ismétlése a jelfeldolgozás mért értékek darabszámára vonatkozó paramétere alapján (C)**

$$R(m) = \frac{X_m(i)}{Y_m(i)} \cdot \left| \frac{X_m(i)}{Y_m(i)} \right|^{-1} \qquad m = 0, \ldots, C-1 \tag{4.7}$$

5. **A mérési eredmények, azaz a komplex vektorok átlagolása és a „szórás" meghatározása**

$$Átlag = \frac{1}{C} \cdot \sum_{m=0}^{C-1} R(m) \tag{4.8}$$

$$Beesési\ szög = arg(Átlag) \tag{4.9}$$

$$Szórás = \left| 1 - \left| Átlag \right| \right| \tag{4.10}$$

Természetesen a „szórás" alatt itt nem a matematikai értelembe vett szórásról van szó, de a szakirodalom nem túl szerencsésen ezt az elnevezést részesíti előnyben.

A konkrét alkalmazás elméleti hátterének, és az alkalmazás megvalósításához választott módszer bemutatása után már hozzáilleszthetjük a DSP-s fejlesztői kártyán kiépített tesztkörnyezethez az ismertetett jelfeldolgozási algoritmust, így a függvénykönyvtár segítségével annak eredményét szemléltethetjük a kijelző segítségével. Az érintőképernyő ezen felül lehetőséget biztosít, hogy az algoritmus bemeneti paramétereit, azaz a Fourier transzformáció pontszámát (N), illetve a mérendő értékek számát (C) a felhasználó egy egyszerű kezelői felületen keresztül megadhassa. A megvalósított grafikus függvénykönyvtár és a konkrét alkalmazás illesztését a következő fejezetben mutatjuk be.

4.3. Az alkalmazás illesztése a DSP-s tesztkörnyezethez

A tesztkörnyezetben egy fő végrehajtási szálat valósítottunk meg, amely inicializálja a grafikus függvénykönyvtárat illetve az eszközillesztőket, és végtelen ciklusban megvizsgálja, hogy megérintette-e a felhasználó az érintőképernyőt. Az illesztés első lépésében implementálnunk kell az iránymérés algoritmusát. A szimulált

tesztadatokat az egyszerű megvalósítás érdekében a DSP memóriájába töltjük, így a jelfeldolgozás analóg-digitális átalakítással és mintavételezéssel kapcsolatos lépéseivel nem kell foglalkoznunk. A tárolt adatokat ezután a kapcsoló-, illetve a bemeneti jelhez tartozó bufferekbe töltjük. Következő lépésben a processzor segítségével el kell végeznünk a a konkrét jelfeldolgozási feladatot, melyhez kihasználjuk a DSP korábbi fejezetben részletesen bemutatott tulajdonságait és sajátosságait:

1. **Twiddle tábla előállítása**

   A Fourier transzformáció kiszámítása során az (4.3) és (4.4) képletekből jól látható, hogy a következő faktorok függetlenek a bemeneti értékektől:

   $$t(n) = e^{-j\frac{2\pi}{N}kn} \tag{4.10}$$

   A fenti értékeket nevezzük *twiddle* faktoroknak. A bemeneti minták számától függetlenül, csak egyszer kell ezeket meghatároznunk, ezért nevezzük ezt az implementált algoritmus 0. lépésének. Célszerű ezt a feladat az alkalmazás inicializásakor elvégeznünk. Az (4.10) képlet alapján tehát így módosul a Fourier transzformáció:

   $$X(k) = \sum_{k=0}^{N-1} x(n) \cdot t(n) \tag{4.11}$$

   $$Y(k) = \sum_{k=0}^{N-1} y(n) \cdot t(n) \tag{4.12}$$

   N az algoritmus konfigurálható paramétere, így a rendszer futása közben változhat, emiatt érdemes meghatároznunk egy maximális értékét, ennek alapján pedig t(n) értékekkel fel kell töltenünk a twiddle táblát.

2. **N hosszúságú bemeneti bufferek feltöltése a memóriában tárolt adatokkal**

   A processzor memóriájából ezután betölthetjük az algoritmus aktuális lépésének megfelelően N darab értéket. Tehát az első ciklus során a kapcsoló- illetve bemeneti jeleket tartalmazó memóriaterület első N darab elemét, a másodikban a második N elemet, stb.

3. **A Fourier transzformációk végrehajtása**

   A twiddle tábla és a bemeneti adatok alapján a DSP saját függvénykönyvtára lehetőséget biztosít a transzformáltak kiszámítására. A használt, adott típusú DSP-re optimalizált függvény esetén az N paraméternek kettő hatványának kell lennie.

4. **A maximumhely meghatározása és a mérési eredmény kiszámítása**

   Ebben a lépésben (4.5) képlet alapján meg kell határoznunk a kapcsolójel Fourier transzformáltjai között a maximális abszolutértékű elem indexét. Ezt például iteratív, összehasonlításos kereséssel tehetjük meg: feltételezzük, hogy az első elem abszolutértéke a legnagyobb, majd hasonlítsuk össze a következő elem abszolutértékével. Ha ez nagyobb, cseréljük le az aktuális maximumhely indexét ennek az indexével. Csak valós bemeneti értékeink vannak, emiatt a Fourier transzformáció rendelkezik a következő tulajdonsággal:

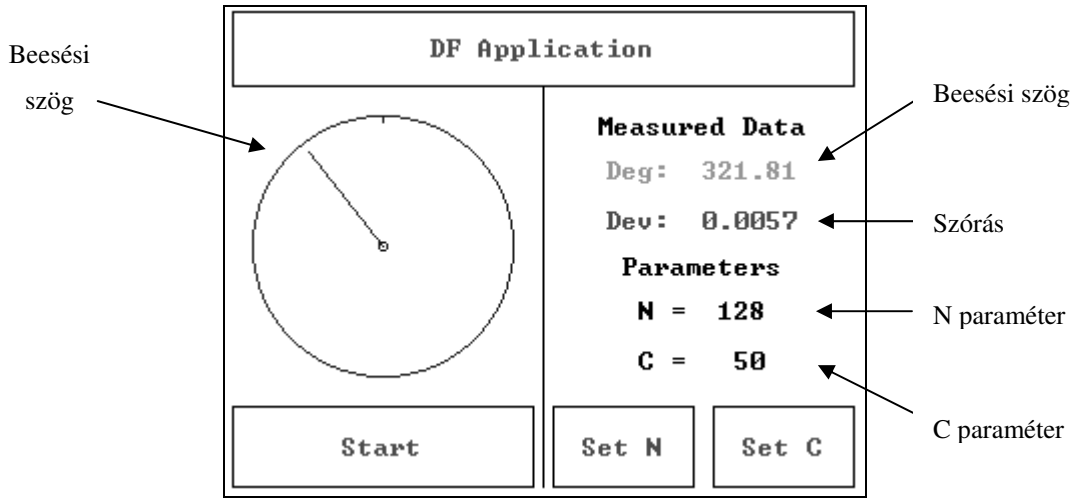
$$X_k = X^*_{N-k}$$

   Mivel komplex konjugáltak abszolútértéke azonos, elég csak az összes elem feléig vizsgálnunk az abszolútértékeket. A maximumhely meghatározása után (4.6) képletet alkalmazva előállíthatjuk az adott bemeneti bufferekhez tartozó mérési eredményt. Az abszolútérték képzés, a komplex számok osztása és a normálás műveletek - a Fourier transzformált kiszámításához hasonlóan - a processzor saját függvénykönyvtára segítségével elvégezhetőek.

5. **A mért értékek alapján átlag és „szórás” számítása**

   A beesési szög és a szórás kiszámításához C számú mért érték előállítása szükséges, ahol C az algoritmus mérési eredmények számára vonatkozó paraméterre. Az adatokat az 2-3. lépések C darabszámú ismétlésével kaphatjuk meg, ennek eredményét az (4.7) képlet segítségével adhatjuk meg. Ezután előbb (4.8) képlet segítségével kiszámítjuk az átlagot egy ciklus segítségével, majd meghatározzuk az így kapott komplex vektor szögét (4.9) képlet alapján. Ez az érték az elektromágneses hullám mért beesési szöge. Végül az (4.10) képlet segítségével kiszámítjuk a hatékonyságra vonatkozó szórás értéket.

A fenti lépések elvégzése után megkaptuk az algoritmus végeredményeit, azaz a beeső hullám szögét és a szórást. Vizsgáljuk meg, hogy milyen módon lehetne a mérési eredményeket a felhasználó számára szemléltetni, illetve biztosítani a paraméterek beállításának lehetőségét a megvalósított grafikus függvénykönyvtár segítségével. Ennek érdekében a függvénykönyvtár használata előtt célszerű megtervezni, hogy milyen grafikus felületet szeretnénk kialakítani.



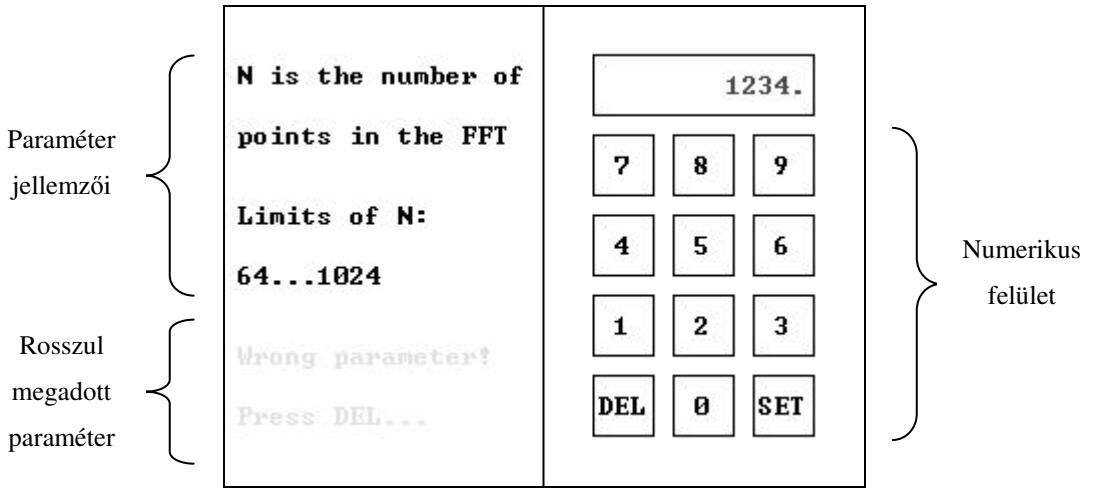
4.7. ábra – A fő felhasználói felület terve

A fő felhasználói felület (4.7. ábra) négy, jól elkülöníthető részből áll. A képernyő felső részében található az alkalmazás címe. A középső felületet egy függőleges vonal segítségével kettéosztottuk. A bal oldalára egy iránytűt ábrázoló rajzelemet helyeztünk, amely segítségével jól szemléltethetjük a beesési szöget. A jobb oldalon a mérési eredmény és szórása látható szöveges formába, emellett a paraméterek aktuális értékeit is itt találhatjuk. A képernyő alsó részében három menüelem látható. A `Start` gomb segítségével indíthatjuk el a mérést, azaz a gomb megnyomására végrehajtjuk a részletesen bemutatott jelfeldolgozási algoritmust. A jelfeldolgozás közben a képernyő jelzi, hogy éppen egy mérés van folyamatban, majd miután a processzor végzett a számítással, a grafikus függvénykönyvtár segítségével frissíti az iránytű által mutatott szöget és a szöveges felületen kijelzi a mért adatokat. Miután elindítottunk egy mérést az érintőképernyő inaktívvá válik, tehát meg kell várnunk az algoritmus teljes lefutását. A további két menüelem segítségével az N és C paraméterek beállítására van lehetőségünk. A `Set N` és `Set C` gombok megnyomása után egy új felhasználói felületet kapunk, amely egy numerikus billentyűzethez hasonlító,

érintőképernyős beviteli felület biztosít a paraméterek módosítására. Összegezve tehát a következő rajzelemeink vannak a fő felületen:

- szövegdoboz az alkalmazás címével,
- elválasztó vonal,
- iránytű a beesési szög szemléltetésére,
- szövegdobozok a mért adatokkal és a beállított paraméterekkel,
- mérést elindító menüelem (`Start`),
- N paramétert beállító menüelem (`Set N`),
- C paramétert beállító menüelem (`Set C`).

A paramétereket beállító felület (4.8. ábra) segítségével a felhasználó módosíthatja a Fourier transzformáció pontszámát, illetve a mérési eredmények számát. A felület mindkét esetben két részből áll. Bal oldalán az aktuálisan beállítandó paraméterre vonatkozó információkat és megkötéseket jelez ki szövegesen. A jobb oldalon látható a numerikus billentyűzet, amely segítségével a számológépekhez hasonlóan adhatjuk meg az új paramétert. A `SET` gomb megnyomása után a program ellenőrzi, hogy a megadott értékek teljesítik-e a paraméterekre vonatkozó feltételeket. Ha valamelyik megkötés nem teljesül a felület bal oldalán figyelmeztetjük erre a felhasználót, ekkor a `DEL` gomb megnyomásával törölnie kell a beírt értéket. A gomb segítségével természetesen a `SET` megnyomása előtt is lehetőségünk van törölni az eddig beírt adatot.



4.8. ábra – Paramétereket beállító felület terve

Amennyiben a megadott paraméter eleget tesz a megkötéseknek visszatérünk a fő felhasználói felületre, és frissítjük az új értékkel a jelfeldolgozás aktuális paramétereit kijelző szövegdobozokat. Ezek alapján a következő rajzelemeket kell megvalósítanunk a paramétereket módosító felületen:

- szövegdobozok a paraméter funkciójáról és megkötéseiről,
- elválasztó vonal,
- az adat megadására szolgáló, számozott gombok (`0, 1, ..., 9`),
- paramétert beállító gomb (`SET`),
- paramétert törlő gomb (`DEL`),
- figyelmeztető szövegdobozok a rosszul megadott paraméterekről.

4.4. Rendszer- és felhasználói tesztek

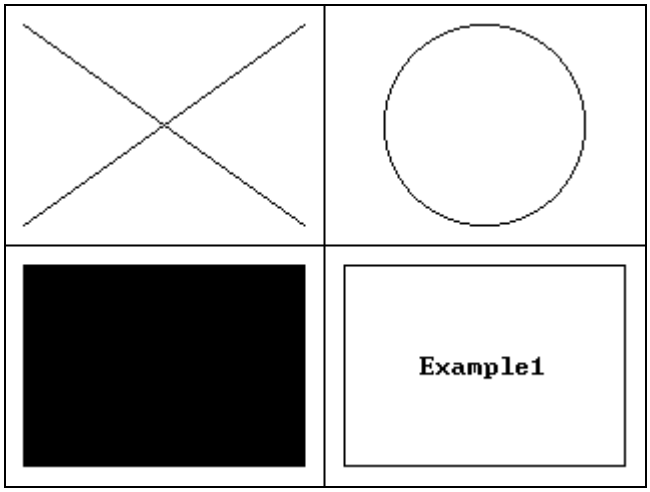
Miután implementáltuk a jelfeldolgozási algoritmust, továbbá megterveztük a felhasználói felületeket és a függvénykönyvtár segítségével meg is valósítottuk azokat, megvizsgálhatjuk az integrált, konkrét alkalmazáshoz illesztett rendszer működését. Az ellenőrzés két szakaszát, azaz a rendszer- és felhasználói teszteket összevonhatjuk. A fejezetben egy-egy példán keresztül mutatjuk be az elképzelhető teszteseteket.

A rendszer működésének vizsgálata során az LCD panelen kijelzett bittérkép és az érintőképernyő kezelését vizsgáljuk. Ellenőriznünk kell első lépésben, hogy a kijelzőn ugyanaz jelenik-e meg, mint ami PC-s tesztkörnyezet alapján elvárunk. Ehhez használhatjuk a korábbi fejezetben ismertetett, a framekezelés működését ellenőrző tesztesetet. Tehát adjuk hozzá a következő rajzelemeket a nyilvántartáshoz az alábbi paraméterekkel:

- Négy vonal, végpontjaik:
  - (1,120) és (320,120)
  - (160,1) és (160,240)
  - (10,10) és (150. 110)
  - (10,110) és (150,10)
- Teli téglalap, átlóinak végpontjai:
  - (10,130) és (150,230)

- Kör a következő paraméterekkel:
  - középpont (240, 60)
  - sugár 50 képpont
- Egy szövegdoboz a következő paraméterekkel:
  - téglalap (170,130) és (310,230)
  - szöveg Example1

Az LCD modul eszközillesztőjének segítségével ábrázoljuk is a rajzelemeket a kijelzőn (4.9. ábra):



4.9. ábra – Az LCD modulon megjelenő kép

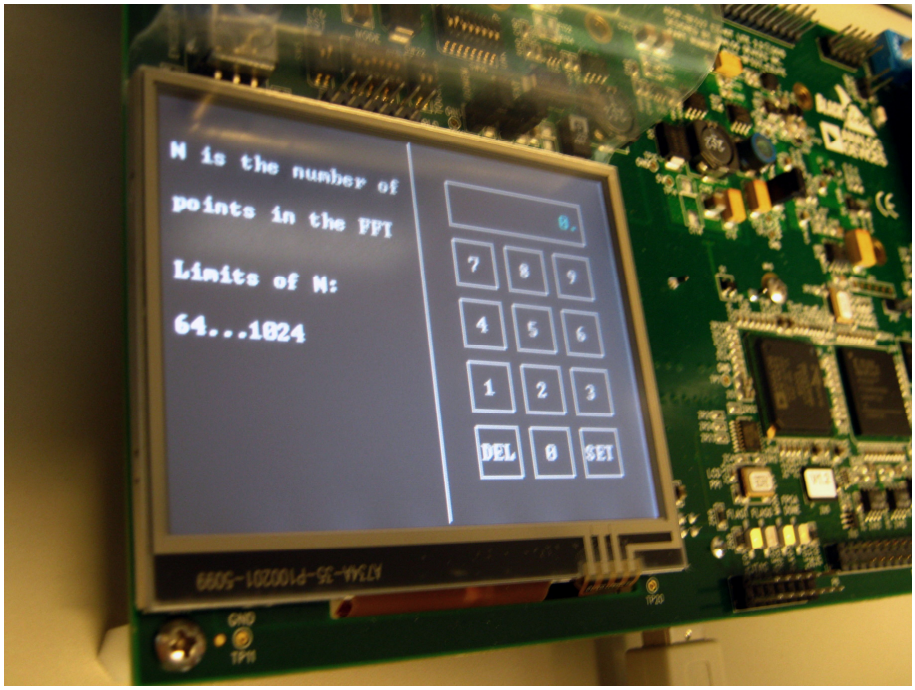
A megjelenített kép alapján megállapíthatjuk, hogy a két eredmény, tehát a PC-s tesztkörnyezet által generált, BMP formátumú fájl tartalma (4.1. ábra) és a kijelzőn látható ábra megegyeznek. Természetesen az összes rajzelemre vonatkozóan, külön-külön teszteltük a működést, és javítottuk a felmerült hibákat.

Az eszközillesztő az LCD modulhoz tartozó érintőképernyő vezérlőjétől kapott információk alapján jelzi a felhasználó számára a megérintett pont koordinátáját. A koordinátákkal ezután meghívhatjuk a függvénykönyvtárhoz tartozó, a képernyő megérintését lekezelő függvényt, amely ellenőrzi, hogy van-e olyan rajzelem, ami tartalmazza a megérintett pontot. Amennyiben talál ilyen rajzelemet, meghívja a hozzá tartozó, korábban letárolt kezelőfüggvényt. Ennek ellenőrzésére az alábbi tesztesetet használjuk: adjunk hozzá egy menüelemet a nyilvántartáshoz, amely egy szövegdobozból és az érintéskor meghívandó függvényből áll. Az utóbbi függvényt definiáljuk úgy, hogy lefutása során változtassa meg a szövegdobozhoz tartozó

karaktersorozatot. Szemléletes eredményt kapunk, ha szövegdobozban a megérintések számát jelöljük. A futassuk a rendszeren a tesztet, és érintsük meg a kijelzőt különböző pontokban, majd ellenőrizzük az eredményt. A kijelzéshez hasonlóan a szekvenciával nem fedtük le az elképzelhető összes lehetőséget, de miután a teszteset sikeresen lefutott (az eredmény nehezen ábrázolható), majd kiterjedten vizsgáltuk több kombinációval a működést, áttérhetünk a magasabb szintű, felhasználói tesztekre.

A felhasználói tesztek alatt esetünkben a konkrét alkalmazáshoz illesztett grafikus függvénykönyvtár segítségével összeállított szoftver működésének ellenőrzését értjük. Megvizsgáljuk, hogy a megtervezett, és a beágyazott rendszeren futó program úgy viselkedik, ahogy azt elvárnák. Első tesztünk során a jelfeldolgozás paraméterei a következők:

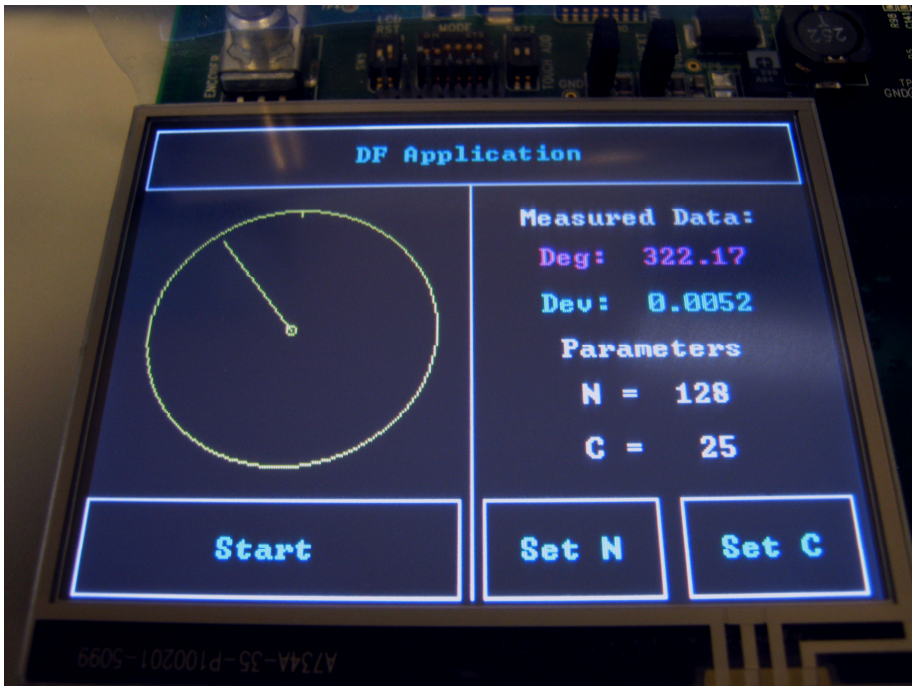
- Fourier transzformáció pontszáma: 128
- Mérési eredmények száma: 25
- Ismert beesési szög: 315°



4.10. ábra – A paramétereket beállító felület

Először beállítjuk a megadott paramétereket a felhasználói felület segítségével (4.10. ábra). Ezután elindítjuk a mérést, majd leolvassuk az iránymérő algoritmus eredményeit a kijelzőről (4.11. ábra):

- Mért beesési szög: 322,17°
- Számított „szórás”: 0,0052 radián



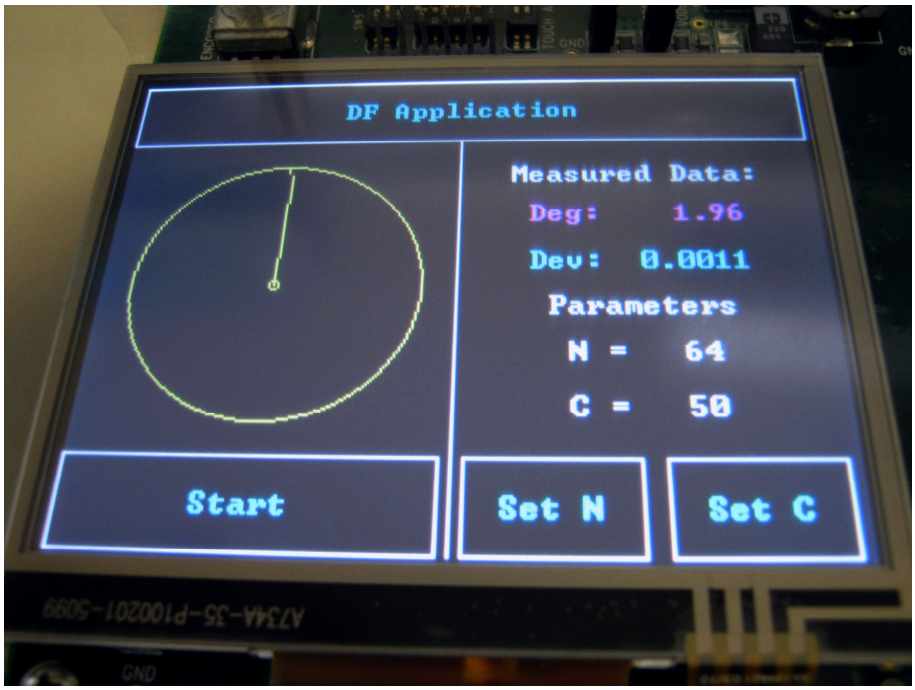
4.11. ábra – Első mérés eredményei a kijelzőn

Az eredményeket összevetve az elvártakkal megállapíthatjuk, hogy a mérés sikeres volt, a szöveges és grafikusan is kijelzett érték megfelelt az elvárásainknak. Természetesen számos paraméter-kombináció képzelhető el, ezeket szintén megvizsgáltuk a felhasználói tesztek során. Példaként egy további tesztet is ismertetünk a következő jelfeldolgozás paraméterekkel:

- Fourier transzformáció pontszáma: 64
- Mérési eredmények száma: 50
- Ismert beesési szög: 0°

A paramétereket az előző esethez hasonlóan először beállítjuk a megfelelő felhasználói felületen az érintőképernyő segítségével, majd elindítjuk a mérést. A kapott eredményeket ezután leolvashatjuk a kijelzőről (4.12. ábra):

- Mért beesési szög: 1,96°
- Számított „szórás”: 0,0011 radián



4.12. ábra – Második mérés eredményei a kijelzőn

A felhasználói teszteknek nem volt célja a megírt algoritmus részletes vizsgálata és értékelése, kizárólag az eredmények kijelzésének vizsgálatával foglalkoztunk. Ellenőriztük a megtervezett rendszer felhasználói felületeit és működésüket. Az eredmények alapján megállapítható, hogy az iránymérő alkalmazáshoz illesztett grafikus függvénykönyvtár, és az így kialakított rendszer a terveinknek megfelelően viselkedett, tehát a fejezetben ismertetett rendszer- és felhasználói tesztek végrehajtása sikeres volt.

Eredmények és kitekintés

A diplomatervezés során egy könnyen portolható, egyszerű elemekből álló, platformfüggetlen grafikus függvénykönyvtár kifejlesztését valósítottam meg. A feladat célja egy egységes felhasználói felület biztosítása az alkalmazásfejlesztők számára a folyadékkristályos kijelzők és érintőképernyők kezeléséhez, figyelembe véve a beágyazott rendszerek sajátosságait. Első lépésben a szakirodalom alapján áttanulmányoztam a legáltalánosabb szoftverfejlesztési szempontokat és módszereket, majd részletesen bemutattam a fejlesztéshez kiválasztott modell szakaszait. A feladathoz igazított modell lépésein keresztül megterveztem a rendszer felépítését: a követelmények alapján meghatároztam a megvalósítandó funkciókat, majd a funkciókat szoftvermodulokhoz rendeltem. A megtervezett modulok adat- és viselkedési modelljét definiáltam, és a szoftver felépítésének terve alapján implementáltam a grafikus függvénykönyvtárat. A tervezés során teszteseteket határoztam meg, különböző absztrakciós szinteken.

Az elkészült szoftver funkcióit három csoportba sorolhatjuk: a hardverfüggő paraméterek beállítása, képkeret kezelése és a rajzelemek kezelése. A használat során a felhasználónak először a platformhoz kapcsolódó konfigurációkat kell megadnia a függvénykönyvtárnak (például milyen méretű és felbontású a kijelző, vagy mekkora memóriaterület áll rendelkezésre a rajzelemek tárolására). Ezután különböző, a függvénykönyvtár által támogatott rajzelemeket adhat hozzá a kijelezni kívánt képhez, például vonalakat, köröket, téglalapokat vagy szövegdobozokat, azok ábrázolási paramétereinek megadásával. A függvénykönyvtár a nyilvántartásába felvett rajzelemek és a kijelző paraméterei alapján előállítja a megfelelő bittérképet, és a már közvetlenül ábrázolható formátumú képkeretet visszaadja a felhasználónak. A bittérkép megjelenítése a konkrét LCD modulon az aktuális platformtól függ, ezért nem része az elkészült függvénykönyvtárnak.

A bemutatott szoftver működésének ellenőrzésére különböző tesztkörnyezeteket valósítottam meg. A PC-n futó kód segítségével megvizsgáltam a grafikus függvénykönyvtár belső működését a megtervezett modulok leírása alapján. A szoftver által generált képkeretet BMP formátumú fájlba tároltam, így ellenőrizhetőek voltak a rajzoló-algoritmusok. A PC-s környezeten futtatott tesztek eredményei későbbi összehasonlításra is alkalmasak. A számítógépen megvalósított alkalmazás mellett egy

konkrét beágyazott rendszeren kialakított tesztkörnyezetet is megvalósítottam. Ehhez az Analog Devices *ADSP BF-527 EZ-KIT Lite* fejlesztői kártyáját használtam. A jelfeldolgozó processzoros kártyán lévő, a feladat szempontjából fontos hardverelemek vizsgálata alapján elkészítettem az LCD modult és az érintőképernyőt vezérlő alacsonyszintű szoftvert. A megvalósításhoz a Visual DSP++ fejlesztői környezetet és a Visual DSP Kernel valósidejű operációs rendszert használtam. Az eszközillesztők ellenőrzését a már részben letesztelt grafikus függvénykönyvtár által előállított bittérképek segítségével végeztem el.

Utolsó lépésként a fejlesztői kártyán futó tesztkörnyezethez egy iránymérő alkalmazást illesztettem. A jelfeldolgozási algoritmus beállításaihoz, illetve az eredményeinek szemléltetésére egy grafikus felületet terveztem a függvénykönyvtár segítségével. A felület lehetőséget biztosít a felhasználó számára a mérési paraméterek megadására és a mérés elindítására az érintőképernyő segítségével. Emellett a mérési eredményeket mind szövegesen, mind „grafikusan” szemlélteti. Az így összeállított rendszeren magasabb szintű teszteket végeztem el és ellenőrizem az eredményeket.

A grafikus függvénykönyvtár a moduláris felépítéséből fakadóan egyszerűen továbbfejleszthető. Az elkészült szoftver a legegyszerűbb rajzelemek mellett a konkrét alkalmazáshoz kapcsolódó rajzelemeket is támogatja, de egyéb rajzelemeket is hozzáadhatunk a rendszerhez (például grafikon). Ehhez definiálnunk kell az új elem paramétereit és ábrázolási algoritmusukat. A függvénykönyvtár aktuális verziója csak RGB888 formátumú bittérkép előállítására képes, de lehetőséget biztosít további formátumok támogatására is.

Irodalomjegyzék

[1] *easyGUI*, Graphical User Interface Tool for Embedded Systems
IBIS Solutions ApS
http://www.easygui.com/

[2] *BF2DGL-OCV*, Blackfin 2D Graphics Library
Analog Devices, Blackfin Software Modules
http://www.analog.com

[3] Ian Sommerville. *Szoftverrendszerek fejlesztése*
Panem Könyvkiadó Kft., 2007
ISBN 9789635454785

[4] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*
McGraw-Hill Higher Education, 2001
ISBN: 0073655783

[5] Jörg Schäuffele, Thomas Zurawka. *Automotive Software Engineering: Principles, Processes, Methods, and Tools*
SAE International, 2005
ISBN 0768014905

[6] Jack E. Bresenham. *Algorithm for Computer Control of a Digital Plotter*
IBM Systems Journal, Vol. 4, No.1, January 1965, pp. 25–30

[7] Michael L. V. Pitteway. *Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter*
Computer Journal, Vol. 10(3) November 1967, pp. 282-289

[8] John Miano. *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*
Addison-Wesley, 1999
ISBN 0201604434

[9] ADSP-BF527 EZ-KIT Lite Evaluation System Manual
Analog Devices, Revision 1.4, August 2009
http://www.analog.com

[10] AD7879/AD7889: Low Voltage Controller for Touch Screens
Analog Devices, Revision C, November 2010
http://www.analog.com

[11] VisualDSP++ 5.0 Compiler and Library Manual for Blackfin Processors
Analog Devices, Revision 5.4, January 2011
http://www.analog.com

[12] VisualDSP++ 5.0 Kernel (VDK) Users Guide
Analog Devices, Revision 3.5, January 2011
http://www.analog.com

[13] Nathan M. Harter. "Development of a Single-Channel Direction Finder Algorithm"
Thesis (MSc), Virginia Polytechnic Institute and State University, 2007

[14] D. Peavey, T. Ogumfunmi. *The single channel interferometer using a pseudo-Doppler direction finding system*
Acoustics, Speech, and Signal Processing, IEEE International Conference, 1997, pp. 4129–4132

Függelék

1. számú melléklet – Megvalósított rajzelemekhez tartozó header fájl

```
typedef unsigned char   U8;
typedef unsigned short  U16;
typedef unsigned int    U32;

typedef void (*touchCallBackFnT)(void*);

typedef enum
{
    INVALID_ELEM = 0,
    ELEM_LINE = 0x01,       /* lineParamsT      */
    ELEM_RECT,              /* rectParamsT      */
    ELEM_CIRCLE,            /* circleParamsT    */
    ELEM_TEXTBOX,           /* textBoxParamsT   */
    ELEM_TOUCHBOX,          /* touchBoxParamsT  */
    ELEM_COMPASS,           /* compassParamsT   */

    ELEM_MODLINE = 0x81,    /* lineParamsT      */
    ELEM_MODRECT,           /* rectParamsT      */
    ELEM_MODCIRCLE,         /* circleParamsT    */
    ELEM_MODTEXTBOX,        /* textBoxParamsT   */
    ELEM_MODTOUCHBOX,       /* touchBoxParamsT  */
    ELEM_MODCOMPASS,        /* compassParamsT   */
} elemT;

typedef struct lineParamsT
{
    U16 x1, y1;
    U16 x2, y2;
    U32 color;
} lineParamsT;

typedef struct rectParamsT
{
    U16 x1, y1;
    U16 x2, y2;
    U8  full;
    U32 color;
} rectParamsT;

typedef struct circleParamsT
{
    U16 x1, y1;
    U16 rad;
    U32 color;
} circleParamsT;

typedef struct pointSetParamsT
{
    U16 *buffer;
    U16 size;
    U32 color;
} pointSetParamsT;


#define MAX_STRLEN      31
#define SMALL_TEXT      1
#define LARGE_TEXT      2

typedef struct textParamsT
{
    U8   length;
    U8   size;
    char string[MAX_STRLEN+1];
    U32  color;
} textParamsT;
```

```
typedef struct textBoxParamsT
{
    rectParamsT rect;
    textParamsT *text;
} textBoxParamsT;

typedef struct touchBoxParamsT
{
    U8                active;
    touchCallBackFnT  touchCallbackFn;
    textBoxParamsT    *textBox;
} touchBoxParamsT;

typedef struct compassParamsT
{
    U16             degree;
    circleParamsT   circle;
} compassParamsT;

typedef union elemParamsT
{
    lineParamsT     line;
    rectParamsT     rect;
    circleParamsT   circle;
    pointSetParamsT pointset;
    textBoxParamsT  textBox;
    touchBoxParamsT touchBox;
    compassParamsT  compass;
} elemParamsT;
```

2. számú melléklet – A felhasználói függvényekhez tartozó header fájl

```
typedef enum
{
    DISPLAY_OK = 0,
    DISPLAY_NOCHANGE,
    DISPLAY_WARNING,
    DISPLAY_ERROR
} displayObjResT;

typedef struct displayObjT
{
    /*
     * ------------------------------------------------------------------------
     * initFrame            - Initialization function
     *
     * Description:
     *      Initializes the frame buffer
     * Parameters:
     *      display         - Display Object
     *      frameBuf        - the initial frame buffer
     *      frameWidth      - width of the frame in pixels
     *      frameHeight     - height of the frame in pixels
     *      frameFormat     - format of the frame --> defines at displayElems.h
     *      frameSize       - size of frame
     *                        use 0 if size = width * height * byte_per_pixel
     *      frameBgColor    - background color
     * Returns:
     *      displayObjResT  - see result codes at definition
     * ------------------------------------------------------------------------
     */
    displayObjResT (*initFrame)(struct displayObjT* display,
                                U8* frameBuf,
                                U16 frameWidth,
                                U16 frameHeight,
                                U8  frameFormat,
                                U32 frameSize,
                                U32 frameBgColor);
    /*
     * ------------------------------------------------------------------------
     * initElemPool         - Initialization function
     *
     * Description:
     *      Initializes the element pool
     * Parameters:
     *      display         - Display Object
     *      elemPoolBuf     - the buffer of the element pool
     *      elemPoolSize    - size of the pool in bytes
     *      elemPoolStride  - size of stride in bytes
     * Returns:
     *      displayObjResT  - see result codes at definition
     * ------------------------------------------------------------------------
     */
    displayObjResT (*initElemPool)(struct displayObjT* display,
                                   U8* elemPoolBuf,
                                   U16 elemPoolSize,
                                   U8  elemPoolStride);
    /*
     * ------------------------------------------------------------------------
     * updateFrame         - Framehandling function
     *
     * Description:
     *      Updates the current frame buffer according to the list of elements
     * Parameters:
     *      display         - Display Object
     * Returns:
     *      displayObjResT  - see result codes at definition
     * ------------------------------------------------------------------------
     */
    displayObjResT (*updateFrame)(struct displayObjT* display);
```

```
/*
 * ---------------------------------------------------------------------------
 * swapFrame            - Framehandling function
 *
 * Description:
 *      Swaps the current frame buffer to a new frame buffer
 * Parameters:
 *      display         - Display Object
 *      frameBuf        - the new frame buffer
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ---------------------------------------------------------------------------
 */
displayObjResT (*swapFrame)(struct displayObjT* display,
                            U8* newFrameBuf);

/*
 * ---------------------------------------------------------------------------
 * clearFrame           - Framehandling function
 *
 * Description:
 *      Clears the current frame and hides all elements
 * Parameters:
 *      display         - Display Object
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ---------------------------------------------------------------------------
 */
displayObjResT (*clearFrame)(struct displayObjT* display);

/*
 * ---------------------------------------------------------------------------
 * setBgColor           - Framehandling function
 *
 * Description:
 *      Sets the background color of the frame
 * Parameters:
 *      display         - Display Object
 *      newBgColor      - new background color
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ---------------------------------------------------------------------------
 */
displayObjResT (*setBgColor)(struct displayObjT* display,
                             U32 newBgColor);


/*
 * ---------------------------------------------------------------------------
 * addElem              - Element handling function
 *
 * Description:
 *      Adds a new display element to the element Pool
 * Parameters:
 *      display         - Display Object
 *      elemType        - type of the elements
 *                        see defines at displayElems.h
 *      elemParams      - pointer to the parameter structure of the element
 *                        see defines at displayElems.h
 *      elemId          - reference to the ID of the newly added element
 *                        value is ILLEGAL_ELEM_ID if result is not OK
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ---------------------------------------------------------------------------
 */
displayObjResT (*addElem)(struct displayObjT*
                          display,
                          elemT elemType,
                          elemParamsT elemParams,
                          U16* elemId);
```

```
/*
 * ------------------------------------------------------------------------
 * removeElem           - Element handling function
 *
 * Description:
 *      Removes a display element from the element Pool
 * Parameters:
 *      display         - Display Object
 *      elemId          - the ID of the element to remove
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ------------------------------------------------------------------------
 */
 displayObjResT (*removeElem)(struct displayObjT* display, U16 elemId);

/*
 * ------------------------------------------------------------------------
 * showElems            - Element handling function
 *
 * Description:
 *      Sets a list of elements to be shown
 * Parameters:
 *      display         - Display Object
 *      count           - number of elements in the list
 *      idList          - pointer to the ID list
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ------------------------------------------------------------------------
 */
 displayObjResT (*showElems)(struct displayObjT* display,
                             U8 count,
                             U16* idList);

/*
 * ------------------------------------------------------------------------
 * hideElems            - Element handling function
 *
 * Description:
 *      Sets a list of elements to be hidden
 * Parameters:
 *      display         - Display Object
 *      count           - number of elements in the list
 *      idList          - pointer to the ID list
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ------------------------------------------------------------------------
 */
displayObjResT (*hideElems)(struct displayObjT* display,
                            U8 count,
                            U16* idList);

/*
 * ------------------------------------------------------------------------
 * modElem              - Element handling function
 *
 * Description:
 *      Modifies an existing display element
 * Parameters:
 *      display         - Display Object
 *      elemType        - type of the elements
 *                        see defines at displayElems.h
 *      elemParams      - pointer to the parameter structure of the element
 *                        see defines at displayElems.h
 * Returns:
 *      displayObjResT  - see result codes at definition
 * ------------------------------------------------------------------------
 */
displayObjResT (*modElem)(struct displayObjT* display,
                          U16 elemId,
                          elemT elemType,
                          elemParamsT newElemParams);
```

```
    /*
     * ------------------------------------------------------------------------
     * handleTouch          - Element handling function
     *
     * Description:
     *      This function should be called if the touchscreen is touched
     * Parameters:
     *      display         - Display Object
     *      pressedX        - x coordinate of the point touched
     *      pressedY        - y coordinate of the point touched
     * Returns:
     *      displayObjResT  - see result codes at definition
     * ------------------------------------------------------------------------
     */
    displayObjResT (*handleTouch)(struct displayObjT* display,
                                  U16 pressedX,
                                  U16 pressedY);

} displayObjT;

/*
 * ----------------------------------------------------------------------------
 * createDisplayObj     - Constructor Function
 *
 * Description:
 *      Creates the Display Object
 * Parameters:
 *      maxElemCount    - maximum number of display elememts
 * Return:
 *      displayObjT     - the Display Object's API
 *                        see details at definition
 * ----------------------------------------------------------------------------
 */
displayObjT* createDisplayObj(U8 maxElemCount);

/* ----------------------------------------------------------------------------
 * destroyDisplayObj    - Destructor Function
 *
 * Description:
 *      Destroys the Display Object
 * Parameters:
 *      display         - the Display Object to destroy
 * Return:
 *      displayObjResT  - see result codes at definition
 * ----------------------------------------------------------------------------
 */
displayObjResT destroyDisplayObj(struct displayObjT* display);
```