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

Tusori Tibor

BEÁGYAZOTT LINUX ÚJRAKONFIGURÁLHATÓ ESZKÖZÖKÖN

KONZULENS

Wacha Gábor József

BUDAPEST, 2016

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Tusori Tibor**, 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 hitelesített 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, 2016. 12. 09.

...…….
Tusori Tibor

Összefoglaló

Az újrakonfigurálható eszközök, ezek közül is az FPGA alapú rendszerek a rugalmasságuknak és testreszabhatóságuknak köszönhetően mindig is nagy népszerűségnek örvendtek a nagy teljesítményt igénylő alkalmazások területén. Az ilyen jellegű felhasználásoknál sokszor hasznos, ha a rendszerünk operációs rendszert futtató komponenst is tartalmaz, ami magasabb absztrakciós szinten történő fejlesztést tesz lehetővé, továbbá leegyszerűsíti a többi, ugyancsak operációs rendszert használó számítógépekkel való kommunikációt. Az operációs rendszerek közül kiemelendő a Linux, ami nyílt forráskódjának, jól skálázhatóságának és rugalmasságának köszönhetően előszeretettel kerül felhasználásra a nagyobb teljesítményű és alapvetően nem real-time rendszerekben. A szakdolgozatom során a Linux újrakonfigurálható eszközökön való használatát vizsgáltam, különös tekintettel a hardveres környezet futás közbeni megváltozására, ami könnyen megtörténhet az újrakonfigurálható eszközökön való használat esetén, az FPGA chipekben megvalósított perifériák ugyanis nem maradandóak, azok működés közben egy más konfigurációval tetszőlegesen felülírhatók. Konkrét platformként egy a Xilinx Zynq-7000-t tartalmazó kártyát használtam (ZedBoard), ami az FPGA rész mellett nagyteljesítményű, fizikailag megvalósított processzormagokat tartalmaz, és ezáltal jól használható a fent vázolt probléma vizsgálatára. A szakdolgozatom célja, hogy megismerkedjek a Linux operációs rendszer Zynq-7000-es környezetben történő felhasználásával, és egy egyszerű mintarendszert alakítsak ki.

Abstract

The reconfigurable devices - and mainly the FPGA based systems - have been always popular in performance demanding applications, since they are flexible and highly customisable. In such environment it can be quite useful to have a component running an operating system, because it makes possible to develop on a higher abstraction layer, and to communicate with other computers using complex protocols more easily. Linux, which is an open source, well scalable and flexible operating system, is one of the most used in high performance and non-real time applications. In my thesis I have examined how the change of the hardware environment occurring during the normal operation time can be registered in the kernel. This is a typical feature of the reconfigurable devices, as the peripherals in an FPGA can be easily changed any time with a new configuration. The target platform I have chosen is a ZedBoard, which has a Xilinx Zynq-7000 on it. It consists of an FPGA and 2 high performance ARM processor, and this makes it an excellent device for the purpose. The aim of my thesis is to setup a Linux kernel on the Zynq platform and to create a sample project consisting of custom peripherals, drivers and some test applications.

1 Bevezetés

Napjainkban az újrakonfigurálható logikai eszközök, azok között is az FPGA-k egyre nagyobb népszerűségnek örvendenek, hiszen a bennük megvalósítható, konkrétan az adott feladatra optimalizált logikai hálózatok sokszor nagyságrendekkel nagyobb teljesítményt képesek nyújtani a hagyományos processzoros rendszerekhez képest. Az FPGA-k használata során azonban gyakran adódnak olyan feladatok, amiket egy processzorral sokkal egyszerűbben és rugalmasabban oldhatnánk meg, mint egy saját állapotgéppel, így a két technológia egyre gyakrabban együtt, egymást kiegészítve jelenik meg. Megnyilvánulhat ez egyrészt olyan módon, hogy a processzort logikai hálózatként közvetlenül az FPGA-ban implementálják (soft-core), vagy úgy, hogy a CPU magokat fizikailag a Si chipen valósítják meg (hard core). Az első megoldás előnye, hogy mind a processzor teljesítménye (órajel, pipe-line, cache, single core / multi core, regiszterkészlet), mind a rendelkezésre álló perifériahalmaz (saját, esetleg ritkán használt perifériák, speciális hardveres gyorsítók) az aktuális alkalmazáshoz skálázható, hátránya azonban, hogy sebesség, ár és fogyasztás terén rosszabbul teljesít a szilíciumon megvalósított hard-core processzorokhoz képest. Soft core processzorra példák a Xilinx Microblaze, Picoblaze és az Altera Nios processzorok, hard core-ra pedig az ARM és Intel CPU-k.

A fent említett két típus mellett léteznek olyan megoldások is, amik mindkét oldal előnyös tulajdonságait igyekszenek biztosítani. Ilyen a Xilinx Zynq-7000[1] eszköz család, ami ARM Cortex-A9-es magok mellett FPGA részt is tartalmaz, ahol egyedi perifériákat – akár egy soft core processzort – hozhatunk létre. Emiatt ezek a chipek kiválóan alkalmasak nagy teljesítményt igénylő feladatok – mint például a videó feldolgozás vagy a következő generációs vezeték nélküli kommunikáció - ellátására.

A technológia fejlődésével az így elkészített processzormagok teljesítménye - főleg a hard core magokat tartalmazóak esetén - olyan szintre növekedett, ahol már egyre inkább képesek lettek különböző operációs rendszerek futtatására. Ezek közül gyakran használt a Linux, ami nyílt forráskódjának köszönhetően sok különböző platformra portolható.

A szakdolgozat során megismerkedtem a Zynq-7000-es eszközzel, a Linux kernellel, a kernel modul fejlesztéssel, saját perifériákat, illetve hozzájuk tartozó

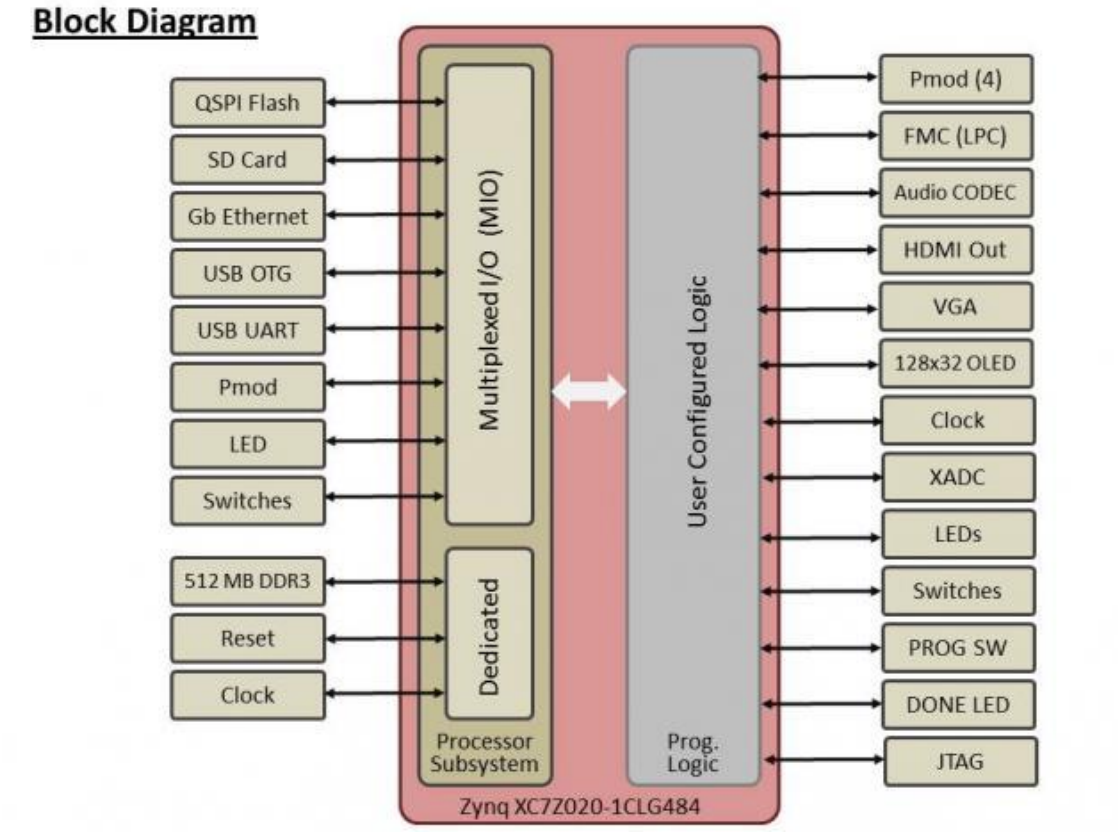
meghajtókat készítettem, továbbá megoldottam az új eszköz rekonfiguráció utáni automatikus regisztrációját és a releváns meghajtó betöltését.

1.1 A szakdolgozat felépítése

A szakdolgozat felépítését tekintve 6 hasznos fejezetből áll, melyek tartalom alapján két csoportra oszthatóak. Az elsőbe az 1-4 fejezetek tartoznak, amikben a feladat megértéséhez, illetve annak megoldásához szükséges elméleti ismereteket mutatom be, melyeket az elvégzett irodalomkutatás során gyűjtöttem. Ezekben fokozatosan haladunk a rendszer egyre magasabb szintjei felé: első lépésként a kiválasztott hardver platform jellemzői, illetve az arra történő fejlesztés módja kerülnek ismertetésre, amit követően rátérek a Linux operációs rendszer, illetve azon belül a kernel felépítésére. Megvizsgálom a feladat szempontjából fontos eszközfa koncepciót és az eszközmodellt, röviden bemutatom a kernel modul fejlesztés sajátosságait, majd végül kitérek a boot folyamatra és az operációs rendszer összeállításának módjára. Az elméleti összefoglalót a megvalósítás dokumentálása követi, ami tartalmazza a kész alkalmazás felépítését és a fejlesztés során meghozott döntések magyarázatát. Ebben a forráskód részletes ismertetésétől eltekintek, az nagyvonalakban kommentek formájában a kódban megtalálható. Végül bemutatom a kész alkalmazás tesztelésére használt programokat, illetve a teszt eredményeit, továbbá pár gondolat erejéig kitérek az esetleges továbbfejlesztési lehetőségekre.

2 Hardver platform

A felhasznált hardver egy AVNet Zedboard, aminek felépítése a 2-1 ábrán látható.



2-1. ábra AVNet Zedboard felépítése [2]

A kártya központi eleme egy Xilinx által gyártott Zynq-7000-es SoC, ehhez kapcsolódnak a különböző kiegészítő perifériák. A Zynq eszközcsalád tagjainak legfontosabb és legszembetűnőbb tulajdonsága az - mint ahogyan ez már a bevezetőben is szerepelt - hogy egy IC-re integrálva tartalmaz egy FPGA-s (Programmable Logic, továbbiakban PL) és egy processzoros (Processing System, továbbiakban PS) részt, melyek között az adatcserére különböző bonyolultságú és teljesítményű interfészek használhatóak.

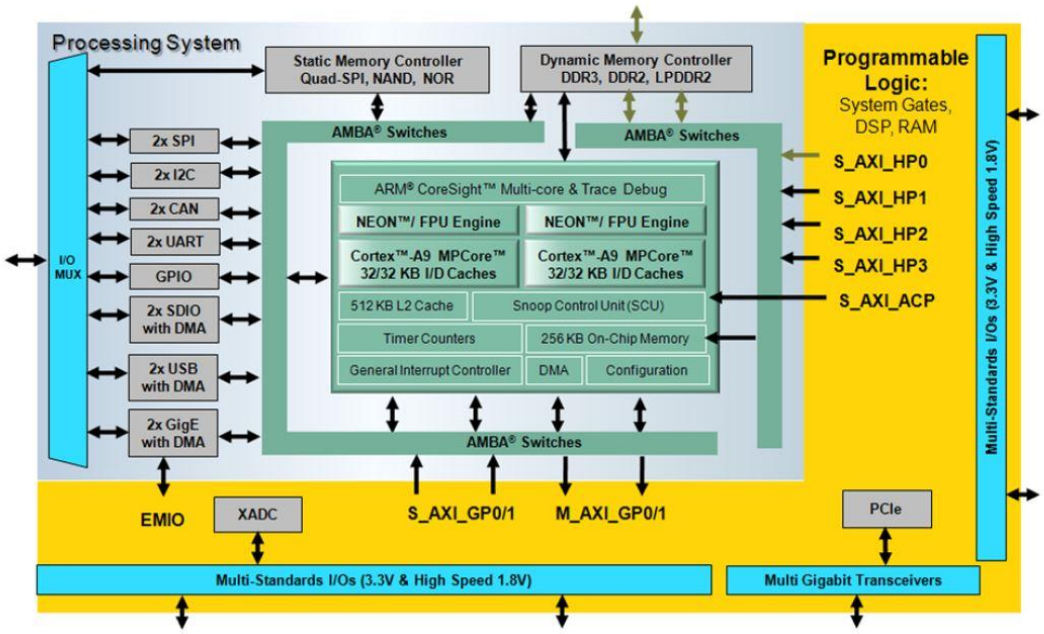
Mind a PS, mind a PL részhez saját perifériák tartoznak, melyekről az adatlap tartalmaz részletesebb információkat. Ezek közül számukra a következők fontosak:

- Nem felejtő memóriák: QSPI flash, SD kártya csatlakozó. Ezek segítségével konfigurálódik fel az eszköz indulás során. Tartalmazzák a PL számára szükséges bitfolyamot, illetve a processzorok által futtatandó programkódot.

- JTAG: A memóriák mellett ezen keresztül is fel lehet konfigurálni az eszközt (azonban lassúsága miatt nem minden esetben célszerű alkalmazni, pl. kernel memóriába töltésére), továbbá ezen történik a programkód debug-ja.
- Felejtő memória: Külső 512 MB DDR3 memória.
- USB 2.0 – UART Bridge
- 8 kapcsoló, 8 led (PL-ből érhetők el)

2.1 Zynq-7000 AP SoC

A Zynq-7000 blokkvázlatát mutatja a 2-2. ábra



2-2. ábra Zynq-7000 blokkvázlat [3]

Itt is jól látszik az előzőekben már említett PS és PL részek elkülönülése, továbbá megjelennek a kettő közötti kapcsolatot biztosító ABMA AXI buszok.

2.1.1 Processing System

Ebben található meg a 2 ARM Cortex-A9 processzor mag, továbbá a hozzájuk kapcsolódó perifériák.

A CPU magok maximális frekvenciája 1Ghz, Harvard architektúrájúak, támogatják a Thumb-2 utasításkészletet, illetve a Java byte kód hardveres gyorsítását

(Jazelle). Két szintű cache segíti a gyors programvégrehajtást, az első szintű magonként 32KB, a második 512KB, közösen használva.

A processzormagok mellett közvetlenül elérhető perifériák kaptak helyet, melyek egy IO MUX-on keresztül kapcsolódnak a lábakhoz, aminek segítségével a konkrét kiosztás az alkalmazáshoz szabható. Emellett van lehetőség a periféria jeleknek PL-be vezetésére is (EMIO).

Az indulás utáni legelső végrehajtott kódot az On-Chip Boot memory tartalmazza, ami a boot folyamat lezárulta után már nem érhető el. Az ebben található bootloader feladata a hardver inicializálása, továbbá az első szintű bootloader (FSBL) betöltése valamelyik nem felejtő memóriából (QSPI/SD kártya). A ROM mellett tartalmazza még a program végrehajtásához szükséges RAM memóriát is (256KB), azonban ennek kis mérete, továbbá a memória menedzser modultól való függetlensége sok alkalmazás számára alkalmatlanná teszi. Helyette a DDR kontrolleren keresztül külső memóriát illeszthetünk a rendszerhez.

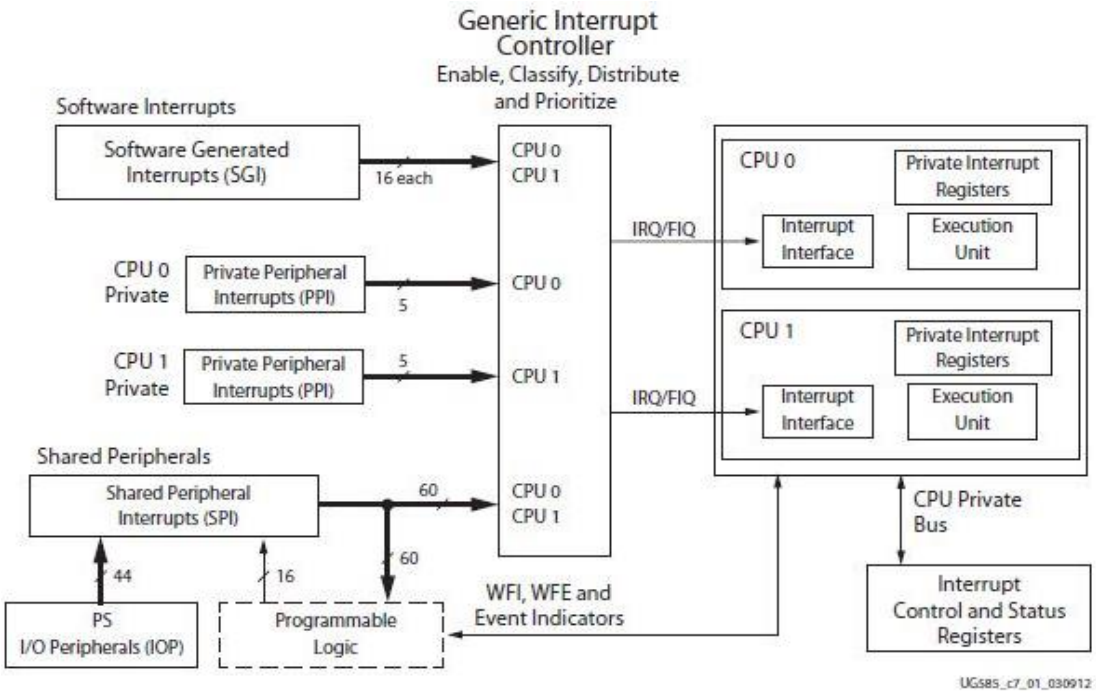
2.1.1.1 Megszakítás kezelés

A Zynq-7000 SoC rendszerben a megszakítások menedzselését a GIC (Generic Interrupt Controller) megszakítás kezelő végzi, ami az interrupt források és a CPU-k között helyezkedik el. Feladata az egyes vonalak engedélyezése, tiltása (maszkolás) a CPU-k által megadott beállításoknak megfelelően, a bejövő megszakítások prioritás szerinti sorba állítása, majd a megfelelő processzorok értesítése. A megszakítás kezelő alrendszer vázlatos felépítése a 2.3-as ábrán látható.

A megszakításoknak alapvetően 3 fajtája különíthető el: szoftveres megszakítás (SGI – software generated interrupt), megosztott megszakítás (SPH – shared peripheral interrupt) és privát megszakítás (PPI – private peripheral interrupt).

Szoftveres megszakításokat az egyes processzormagok kérhetnek úgy, hogy a GIC megfelelő regisztereibe a kért megszakítás számát írják. Ezzel a módszerrel egy CPU saját magának, a másik CPU-nak, vagy akár mindkettőjüknek képes jelezni.

A privát megszakítások jellemzője, hogy csupán az egyik processzorra hatnak. Mindkét CPU 5-5 ilyen saját vonallal rendelkezik, amiket jellemzően időzítők használnak (global timer, private timer, watchdog timer), de kettő-kettő a PL-ből is elérhető.

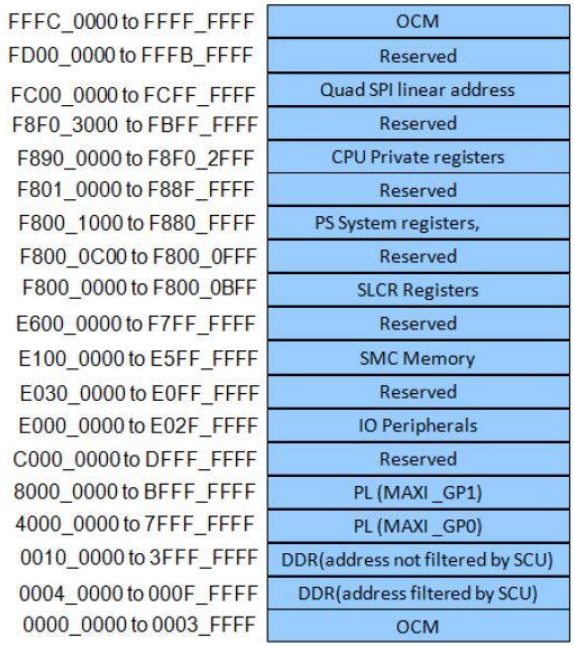


2-3. ábra Zynq-7000 megszakításkezelés [4]

Az utolsó, de legnagyobb halmazt a megosztott megszakítások jelentik, ezek azok, amiket a PS és a PL-beli perifériák jellemzően használnak. Ezek beállítástól függően vagy az egyik, vagy mindkét CPU-t megszakítják. Az utóbbi esetben a kérés kiszolgálását csak az egyik processzor végzi el, a másik egy nem létező (1023-as) interrupt ID-t kap, amikor a GIC-ből megpróbálja kiolvasni az aktív interrupt számát, és ennek hatására felfüggeszti a programvégrehajtást. (Emiatt az interrupt kezelő rutinban nem kell lock-ot alkalmazni.) A rendelkezésre álló 60 megosztott megszakítási vonal nem csak a GIC-be fut be, hanem a PL-ből is elérhető, így az ott megvalósított modulok értesülhetnek a PS alrendszer perifériáinak jelzéseiről is.

2.1.1.2 Memory Map

A Zynq-7000 processzorok által használt címkiosztást a 2.4-es ábra tartalmazza. Látható, hogy a perifériák és az egyéb konfigurációs regiszterek egy címtérben helyezkednek el a memóriákkal. (Ez a Memory mapped IO struktúra.) Ennek ellentéte az, amikor a perifériák külön címtérben helyezkednek el, és speciális utasításokkal érhetők el. (Az ilyen perifériákat Port Mapped IO-nak nevezzük.) Az első megoldás előnye, hogy ugyanolyan címzési módok használhatók a perifériák kezelésére, mint a memória esetében, továbbá így a CPU-ban nem kell külön erre a célra logikai áramköröket létrehozni.



2-4. ábra Zynq-7000 címkiosztása [3]

A későbbiek szempontjából a legjelentősebb, hogy a két master AXI_GP busz a 0x40000000 és a 0x80000000 címektől kezdődően érhető el. (Az AXI buszról a 2.2 fejezet szól.)

2.1.2 Programmable Logic

A Zynq eszközök processing system melletti másik nagy része a programmable logic, ami felépítésében és használhatóságában megegyezik egy FPGA-val. Típusszámtól függően vagy az Artix-7, vagy Kintex-7 FPGA logikán alapulnak.

Egy FPGA (field-programmable gate array) egy olyan integrált áramkör, amiben tetszőleges kombinációs vagy sorrendi hálózat valósítható meg a céláramkörbe való beültetés után, és a konfigurációja bármikor megváltoztatható. Felépítésüket tekintve egyforma logikai blokkokból (configurable logic block) állnak, melyek be és kimenetei egy nagy kiterjedésű összeköttetés hálózat segítségével kapcsolhatók össze. A logikai blokkok szeletekből (slice) épülnek fel, amikről általánosságban elmondható, hogy két fő elemük egy LUT és egy flip-flop. A LUT egy többnyire 4-6 bemenetű és 1 kimenetű igazságtábla, aminek segítségével egyszerű kombinációs hálózatokat lehet megvalósítani, sőt speciális esetekben lehetőség van a LUT RAM-ként való használatára is. A flip-flop-ok jellemzően D típusúak, amik bemenete megegyezik a LUT kimenetével. Ezek mellett még sok kiegészítő logika található meg (például carry logika, egy bites szorzó AND kapu, különböző multiplexerek a szomszédos logikai blokkok összekapcsolására), amik tovább bővítik a blokkok használhatóságát.

Az FPGA-k programozása hardver leíró nyelven (VHDL, Verilog) történik. Ezen nyelvek segítségével a kívánt működést írjuk le, a konkrét eszközre való implementálás a fordító feladata. A tervezés ún. regiszter-transzfer szinten történik, ami azt jelenti, hogy a létrehozni kívánt hálózatot regiszterekből és azokat összekötő kombinációs hálózatokból építjük fel. Látható, hogy ez a szemléletmód jól illeszkedik az FPGA-k felépítéséhez.

Egy hardver leíró nyelven megírt hálózatmodell a megvalósítás mellett szimulációra is lehetőséget ad. Ennek során a megtervezett hálózatra célszerűen választott bementet vezetünk, amire a szimulátor meghatározza a rendszer válaszát, azaz az egyes jelek és regiszterek értékeit az idő szerint. Az így kapott hullámformákat összevetve az elvárt viselkedéssel megállapítható a rendszer jósága.

A logikai blokkok és a külvilág közötti kapcsolat megteremtésére IO cellák szolgának. Ezek segítségével állíthatóak be egy IO láb következő paraméterei:

- kimenet / bemenet
- felhúzó / lehúzó ellenállások

- meghajtás erőssége
- feszültség szintek, IO standardok (pl.: LVCMOS33)
- kimenet sebessége, a szintváltás meredeksége (slew rate)

FPGA-ra való fejlesztés során a kimeneti jeleket és a kimenetek paramétereit ún. constraint fájlokban adjuk meg, melyek minden IO lábhoz hozzárendelik a fent leírt paramétereket. Általában minden FPGA-s fejlesztőkártyához tartozik egy, a gyártó által előre megírt constraint file, ami tartalmazza, hogy az adott lábhoz a kártyán milyen periféria kapcsolódik, megkímélve ezzel a fejlesztőt az adatlapok részletes vizsgálatától.

A logikai és IO cellákon kívül az FPGA-k még számos egyéb kiegészítő elemet tartalmaznak, mint például a blokk RAM-ok, DSP szeletek, órajel menedzser modulok.

A blokk RAM-ok a logikai cellák LUT-jainál nagyobb kapacitású memóriák, jellemzően több portosak, adatszélességük állítható, akár beépített FIFO-t is tartalmazhatnak.

A DSP szeletek a jelfeldolgozási feladatok hatékonyabb megvalósítását segítik azáltal, hogy a MAC (multiply and accumulate) műveletek hely és sebesség hatékony elvégzésére képesek. Ennek megfelelően két fő komponensük a hardveres szorzó (Zynq-7000 esetén 25*18 bites) és egy akkumulátor. Ezeket egészítik ki még további elemek, mint például a szimmetrikus szűrők esetén használható előösszegzők, vagy a párhuzamosításhoz szükséges pipeline regiszterek.

Az órajel menedzser modulok feladata az FPGA többi részét működtető belső órajelek előállítása a külső referencia órajelből. Ennek során képesek a bejövő jelet jitter mentesíteni, frekvenciát osztani vagy szorozni, állítható mértékű fázistolást beiktatni.

2.2 PS-PL interfész

A Zynq eszközök teljesítménye szempontjából fontos tényező az őket felépítő két modul, a processing system és a programmable logic hatékony összekapcsolhatósága. Az interfész az ARM által létrehozott AMBA 3.0[5] specifikáción alapszik, ami alapvetően SoC (System on Chip) környezetben előforduló blokkok összeköttetésére ad megoldásokat. Jól dokumentáltsága és szerzői jog mentessége miatt mára az ipar egyik de facto szabványává nőtte ki magát.

Az AMBA több protokollt is definiál, ezek közül az AXI (Advanced Extensible Interface) került felhasználásra a Zynq eszközökben. Az AXI busz emellett az FPGA chipekben megvalósított soft core mikroprocesszorok kedvelt busza, például a Xilinx cég MicroBlaze processzoraihoz ezen keresztül kapcsolhatunk perifériákat.

Jelen eszközünkben a PL és PS összekapcsolására a következő AXI portok használhatók[4]:

- 4 AXI HP (high performance) nagy sebességű port, amin keresztül a PL masterként közvetlenül a rendszer OCM (on-chip memory) vagy DDR memóriáját érheti el.
- 4 AXI általános célú port (AXI_GP), melyek közül 2 slave és 2 master. A processzorok ezeken keresztül érhetik el a PL-ben megvalósított AXI slave perifériákat.
- AXI ACP cache koherens port, amin keresztül a PL-ben elhelyezkedő masterek a két processzor által közösen használt L2 cache-t érhetik el ugyanazon az ún. SCU egységen keresztül, amin ezt a processzorok is teszik.

2.2.1 AXI busz

A buszok célja, hogy master és slave modulok között teremtsenek kommunikációs kapcsolatot. Mastereknek nevezzük azokat az egységeket, amik írási vagy olvasási műveletek kezdeményezésére képesek a buszon keresztül. Ezzel szemben a slave egységek csak a master kérésére vehetnek részt a kommunikációban. Egyszerű buszokon jellemzően a több slave mellett csupán egyetlen master van, és a buszon zajló forgalmat ilyenkor ez határozza meg. Abban az esetben viszont, ha több master csatlakozik a buszra, valamilyen úgynevezett arbitrálási módszerrel el kell tudniuk dönteni, hogy egy adott pillanatban ki irányítja az adatforgalmat.

Az AXI busz a második csoporthoz tartozik, azaz megengedi párhuzamosan több master használatát. A busz központi eleme az ún. AXI interconnect modul, ehhez kapcsolódik az összes többi elem. Szerepét tekintve a masterek számára slave-ként, a slavek számára masterként jelenik meg, és feladata a buszon zajló kommunikáció lehetővé tétele. Ennek során a következő feladatokat látja el:

- A buszra kapcsolódó master modulok között elosztja a busz használati jogát, azaz busz arbiterként működik.
- A kapcsolódó egységek által használt adatbusz szélesség különböző lehet, az ezek közötti átalakítást biztosítja.
- A kapcsolódó egységek az AXI szabvány különböző verzióit használhatják, így közöttük protokoll konverziót kell végezni. (Például a Zynq-7000 PS része AXI3-t használ, míg a PL-ben AXI4-Lite perifériát is létrehozhatunk)
- Ellátja a felfűzött egységek eltérő órajeléből eredő szinkronizációs feladatokat.
- Opcionális FIFO biztosításával a különböző gyorsaságú egységek között sebességkiegyenlítést végez.

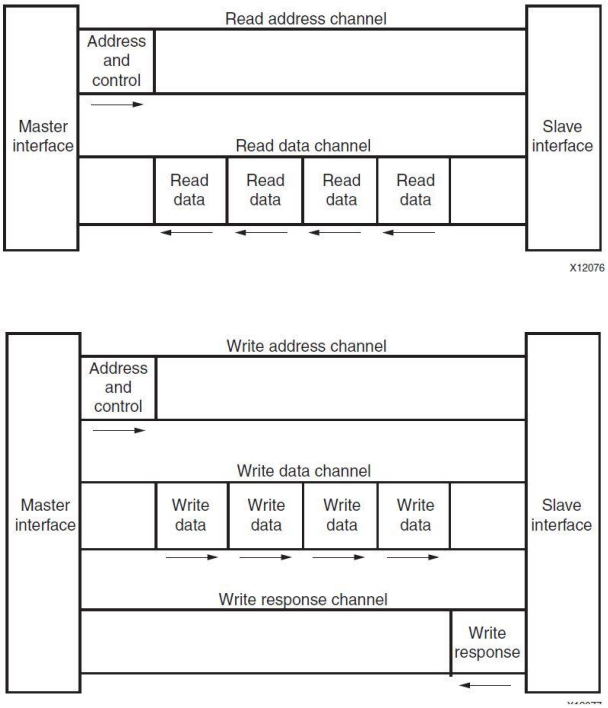
Az AXI buszon minden egyes slave-hez saját, egymással át nem lapolódó címtartományt rendelünk. Bejövő adat esetén az interconnect a célcím alapján választja ki, hogy melyik slave számára továbbítja azt. A kirendelt címtartományok jellemzően nagyobbak, mint amennyit a periféria ténylegesen kihasznál, a fennmaradó területekre történő írás vagy olvasás hatása megvalósítás függő. Jellemző megoldás, hogy a

célregiszter kiválasztása során csak az alsó címbiteket figyeli az egység, hiszen eleve csak olyan adatot kap meg, amit neki szántak. Ilyenkor a nem használt címeken a megvalósított regiszterek ciklikusan újra megjelennek. A címtartományok kiosztásánál ügyelni kell arra, hogy azok mindig blokkhatáron kell, hogy kezdődjenek, azaz például egy 4kB címtartományú periféria kezdőcíme *0x******000* alakú.

Az interconnect funkcióinak felsorolásában megjelent az AXI4-Lite megnevezés. Ez az AXI 4. verziójában jelent meg, és a teljes protokoll egyszerűsített változatának tekinthető. Létrehozásához a motivációt az adta, hogy sok esetben az AXI buszon keresztül csak konfigurációs regiszterek állítása történik (például egy Timer esetén, de ide sorolhatóak a lassabb kommunikációs perifériák – úgy, mint az UART, I2C – is), ami nem igényel nagy sávszélességet, és jó lenne, ha ilyenkor a buszra minél kevesebb logika felhasználásával tudnánk kapcsolódni. A legszembetűnőbb újdonsága a protokollnak, hogy nem implementálja a burst módot, azaz minden egyes írási vagy olvasási ciklusban csupán egyszer történik adatátvitel.

A szakdolgozat egyik célja a Zynq PL-részében perifériák implementálása és ezek a CPU-k számára elérhetővé tétele. Tekintve, hogy ezek közül egyik sem igényel nagy átviteli sebességet, a fentieknek megfelelően az AXI4-Lite protokollt választottam.

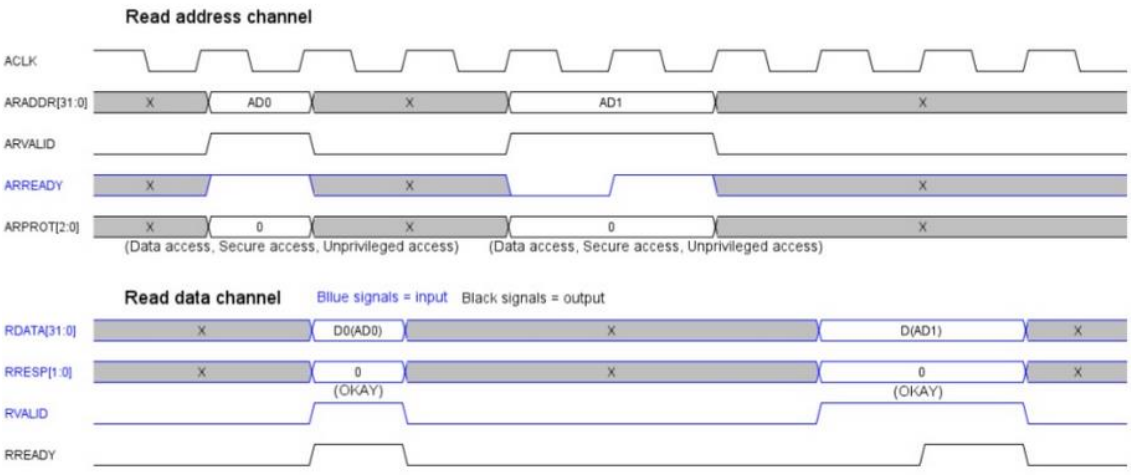
Az AXI4-Lite[6] interfész 5 csatornából épül fel, amikből kettő az olvasáshoz, és három az íráshoz használt. Minden csatornán kézfogásos adatátvitel történik, amit *ready* és *valid* jelek felhasználásával valósítanak meg. A *valid* vonalat mindig az információ küldője állítja, ezzel jelzi, hogy az adatvonalak érvényes állapotban vannak, ezért a másik fél mintavételezheti azokat. A *ready* vonalat ezzel szemben a fogadó oldal vezérli, ennek segítségével értesíti az adót, hogy készen áll egy újabb átvitelre. Az adatok mintavételezése tehát az órajel azon felfutó élénél történik, amikor mind a valid, mind a ready értéke magas. A felfutó él után mindekét fél alacsony szintre állítja a handshake jeleit, és felkészül a következő ciklusra.



2-5. ábra AXI busz

2.2.1.1 Olvasás

Olvasás során a Read address channel és a Read data channel használatos. Az elsőn küldi el a master a kért adat címét a slave felé, amire válaszképpen a másodikon érkezik az adat.



2-6. ábra AXI-Lite olvasás[8]

Read address channel jelei:

- ARADDR [31:0] : A master ezen keresztül küldi el az olvasás címét.
- ARVALID: Magas értéke az ARADDR vonalon érkező adatok érvényességét jelzi. Master vezérli.
- ARREADY: High értéke jelzi, hogy a slave egység készen áll az olvasás címének fogadására. Slave vezérli.
- ARPROT[2:0]: Az olvasás védelmi szintjét jelzi.

Az ARVALID és ARREADY együttes magas értéke esetén történik meg az olvasási cím átvitele, ezek után mind a master, mind a slave 0-ba állítja a handshake jeleit.

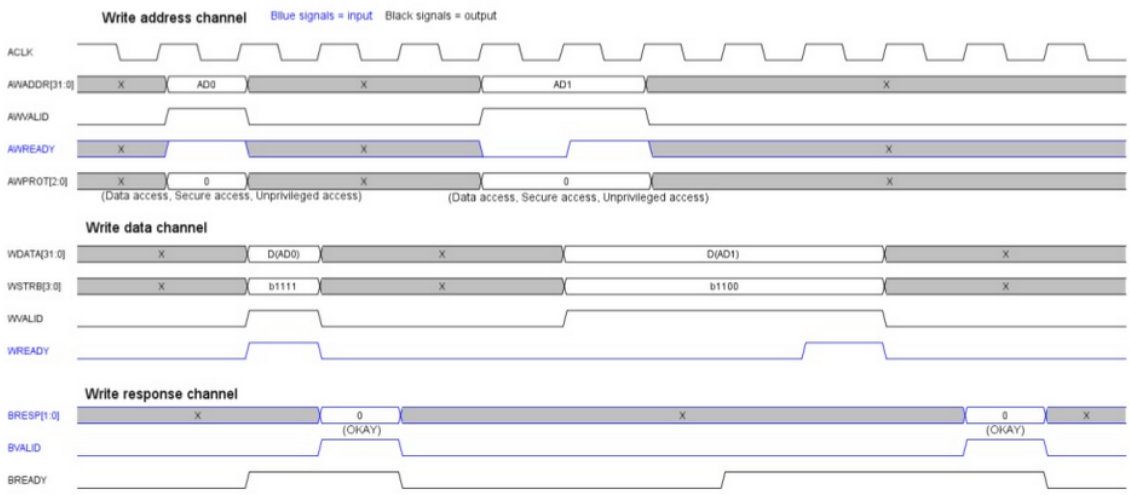
Read data channel jelei:

- RDATA [31:0]: Olvasott adat. Slave állítja.
- RVALID: Magas értéke esetén az RDATA vonalak érvényesek, a másik oldal mintavételezheti őket. Slave állítja.
- RREADY : Ha magas, a master oldal kész az adat beolvasására.
- RRESP[1:0]: Az olvasás sikerességéről értesíti a master egységet. Lehetséges értékei: OKAY(sikeres adatelérés), EXOKEY(sikeres kizárólagos adatelérés), SLVERR(slave hiba), DECERR(nem létező címről való olvasás. Funkciója hasonló a Write response csatornáéhoz.

Az adatvonalak tényleges mintavételezése az RVALID és RREADY magas értéke esetén következik be, ami után mindkét fél alacsony szintre állítja a handshake jeleket.

2.2.1.2 Írás

Az írási művelet három lépésből épül fel, amik külön-külön csatornákon történnek meg. Először a Write address channel-en keresztül a master elküldi a slave számára a célcímet, aminek visszaigazolása után kezdődik meg az adatok átvitele (Write data channel). Ennek végeztével a slave a Write response channel jeleivel visszaigazolja a művelet sikerességét.



2-7. ábra AXI-Lite írás[8]

Write address channel:

- AWADDR: Az írás művelet célcíme. Master küldi.
- AWVALID: Magas értéke az AWADDR vonalakon érkező adatok érvényes állapotát jelzi. Master küldi.
- AWREADY: Magas értéke esetén a slave egység készen áll az írási cím vételére. Slave állítja.
- AWPROT[2:0]: Az adatátvitel védelmi szintjét jelzi, továbbá megadja, hogy az adat vagy utasítás átvitel történik-e.

Az olvasási csatornákhoz hasonlóan az adatátvitel itt is a két handshake jel magas állapotában következik be, ami után azokat a felek alacsony szintre állítják.

Write data channel:

- WDATA[31:0]: Az írandó adatok átvitelére szolgál. Master állítja.
- WVALID: Magas értéke a WDATA vonalak érvényes állapotát jelzik. Master állítja.
- WREADY: Magas értéke esetén a slave készen áll az írni kívánt adatok fogadására. Slave állítja.
- WSTRB[3:0]: Bár a WDATA vonal 32 bit széles, lehetőség van a cél duplaszó csupán meghatározott bájtjait felülírni. A 4 WSTRB jel az átvitt 4 bájt közül azokat jelöli ki, amiket a slave egységnek fel kell dolgoznia.

Write response channel:

- BRESP[1:0]: Ezen keresztül a slave egység állapot információt küld a master számára az írás művelet sikerességéről. Slave állítja. Lehetséges értékek: OKAY (sikeres művelet), EXOKAY (sikeres kizárólagos hozzáférés), SLVERR(slave error), DECERR(decode error, nem létező címre történő írás).
- BVALID, BREADY: A többi csatornához hasonló handshake jelek.

Az AXI buszról bővebb információt a [6] és [8] forrás tartalmaz, itt a szakdolgozat szempontjából releváns, a megírt perifériákban felhasznált elemek bemutatására szorítkoztam.

A fentebb leírtak alapján látszik, hogy az AXI busz nem támogatja a rá kapcsolódó eszközök működés közbeni felderítését. Ez végső soron nem meglepő, hiszen alapjában véve statikus, időben nem változó komponenshalmaz összekapcsolására szolgál. Előfordulhat azonban olyan szituáció, mint például egy FPGA részleges, vagy egy Zynq PL részének teljes rekonfigurációja, amikor az AXI-ra felfűzött eszközök módosulnak. A szakdolgozatomnak egyik célja, hogy egy ilyen esetben használható azonosítási módszert hozzak létre.

2.3 Zynq-7000 boot folyamata [4]

A Zynq-7000 SoC indulása (vagy újraindulása) után a két processzor közül a 0-s indexű kezdi meg ténylegesen a működését, míg a másik WaitForEvent utasítás végrehajtásával várakozik. A legelőször végrehajtott programkódot az OCM (On Chip Memory) BootROM-ja tartalmazza, aminek a hardver minimális inicializálásán túl az a feladata, hogy külső forrásból betöltse a további futtatandó kódot. Az eszköz összesen a következő 5 féle forrásból bootolhat:

- JTAG
- NOR flash
- NAND flash
- Quad-SPI
- SD kártya

Ezek közül a BOOT_MODE lábak megfelelő feszültségszintre kötésével választhatunk, amihez a fejlesztőkártyákon általában jumpereket helyeznek el. Fontos

azonban, hogy az eszköz a BOOT_MODE lábakat csak ún. Power On Reset esetén mintavételezi, ami tényleges induláskor, vagy az eszköz PS_POR_B lábára impulzus ráadása esetén történik, ilyenkor ugyanis minden belső regiszter az alapértelmezett értékét veszi fel, azaz az eszköz nem emlékszik a korábbi beállításokra.

JTAG-on keresztül történő kódletöltés esetén a BootROM-ra minimális munka hárul, feladata mindössze a JTAG interface inicializálása, ami után slave szerepet betöltve fogadja az adatokat. A JTAG interface egyik hátránya, hogy aránylag lassú, így nagyobb kód letöltése - mint például egy Linux kernel – sok időt vehet igénybe. Az ilyen esetekben célszerű SD kártyáról bootolni, vagy egy bootloader segítségével Etherneten keresztül letölteni a kernel képet.

A többi mód esetén (NOR,NAND,QSPI,SD) a BootROM masterként olvasni kezdi a memóriákat ún. Boot Header blokkot keresve. Ez a tényleges programkód előtt helyezkedik el, és rá vonatkozó információkat tartalmaz, mint például hogy a kód milyen hosszú vagy le van-e titkosítva. Alapesetben az érvényes header utáni kódot a belső OCM-be (On-chip memory, RAM) másolja, majd átadja neki a vezérlést. NOR és QSPI módokban azonban lehetőség van a program közvetlenül flash memóriából való futtatására (execute-in-place), amikor is a processzor egyesével olvassa be az utasításokat végrehajtás közben. Ennek előnye, hogy ilyenkor a programkód mérete nem limitált 192 kB-ban, ami a belső OCM memória mérete.

A bootolási folyamat végeztével a BootROM kikerül a CPU-k címteréből, így az a későbbiekben már nem lesz elérhető. Az utasításvégrehajtás átkerül az OCM-be töltött programkódra, amit FSBL-nek (First Stage Boot Loader) nevez az irodalom.

Az FSBL inicializálja a hardver elemeket a fejlesztő által megadott módon, például beállítja a DDR memória kontroller és a PS-beli perifériák paramétereit, továbbá az opcionálisan mellékelt bitfile felhasználásával felkonfigurálja a PL részt is.

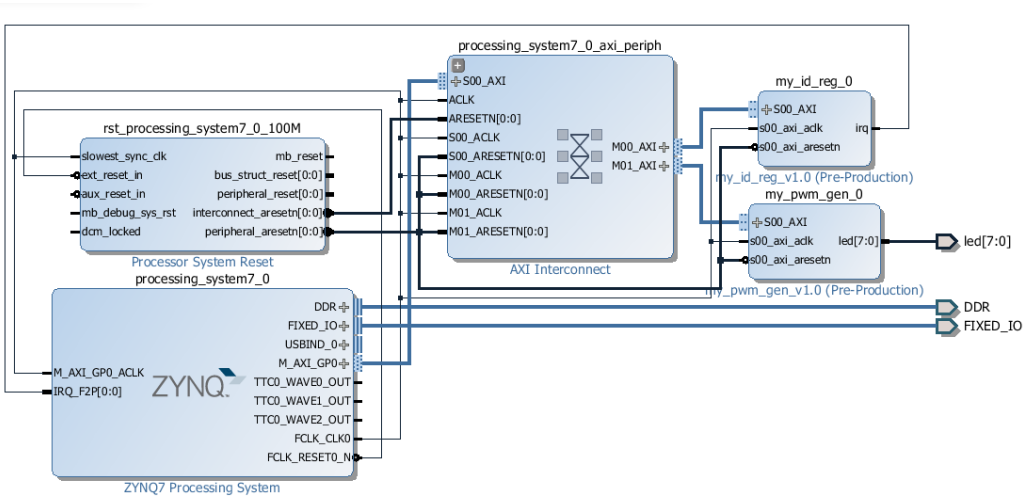
Mindezek után az eszköz készen áll a megírt programkód futtatására, így az FSBL utolsó műveletként betölti azt a DDR memóriába, és elkezdi annak végrehajtását.

3 Fejlesztés a hardver platformra

A Xilinx által készített eszközök konfigurálására és programozására a cég saját fejlesztőkörnyezetet adott ki. Ez a Vivado, amivel alapvetően FPGA-ra készíthetünk alkalmazásokat VHDL vagy Verilog nyelven. Emellett rengeteg kész, előre megírt és az adott platformra fordítható modult, ún. IP Core-t is tartalmaz, például a MicroBlaze processzor és a hozzá AXI-n keresztül kapcsolódó perifériák. Ezek segítségével a rendszerünk egy blokkvázlattal megadható, amiből később a program legenerálja a modulokat hardver leíró nyelven példányosító kódot, ami szükség esetén saját verilog kóddal bővíthető.

Zynq platformra történő fejlesztés esetén is hasonlóan kell eljárni: az ip core-ok között található egy PS-t szimbolizáló modul, aminek beillesztésével jelezzük a rendszer számára, hogy azt fel szeretnénk használni. (Természetesen ez a blokk verilog kódot nem tartalmaz, hiszen a processzorok fizikailag vannak megvalósítva.) A beillesztett blokk több porttal is rendelkezik, amiken keresztül más egységeket képes elérni. Ilyen például a már említett általános célú AXI interface, amire kész perifériákat csatlakoztathatunk. A blokkok felhelyezését követően a fejlesztőrendszer automatikus összeköttetést generáló varázslója létrehozza a szükséges kapcsolatokat, megkönnyítve ezzel a fejlesztés folyamatát. Néhány beállítást azonban kézzel kell megtenni: ilyen lehet például a lábkiosztás, a PS-PL interrupt vonalak és az órajel engedélyezése, stb.

Egy kész blokkvázlatra mutat példát a 3.1 ábra, amin a szakdolgozat során elkészített saját my_id_reg és my_pwm_gen perifériák szerepelnek, melyek ismertetéséről 5.2.1 a fejezetben lesz szó.



3.1. ábra Hardver blokkvázlat

A 3.1 blokkvázlatból a következő lépésben egy összefogó modult (HDL wrapper) generálunk, ami Verilog nyelven tartalmazza az előforduló almodulok példányosítását, továbbá a be és kimeneti jeleiknek megfelelő összekötését.

Ebből végül előállítható az FPGA konfigurációjához szükséges *.bit file.

Ahhoz, hogy az elkészített rendszerre a későbbiekben programot tudjunk fejleszteni, a fejlesztőkörnyezeteknek általában szükségük van a mögöttes hardver tulajdonságaira. Ilyen célú felhasználásra a Vivado egy speciális *.hdf fájlt tud generálni, ami többek között tartalmazza a processzorok típusát, paramétereit, a memóriákat, azok méreteit, továbbá az egyes perifériákat.

Operációs rendszer nélküli alkalmazások elkészítésére a Xilinx SDK használható, ami egy Eclipse alapú, egyedi kiterjesztéseket tartalmazó fejlesztőkörnyezet. Ez a hardver leíró fájl alapján egy egyedi driver csomagot (board support package) állít össze, ami magasabb szinten történő programfejlesztést tesz lehetővé.

Operációs rendszer, mint például a Linux, használata esetén vagy saját magunk konfiguráljuk fel és fordítjuk le a kernelt, vagy automatikusan generáltatjuk azt egy speciális programcsomaggal. Ilyen eszköz például a PetaLinux, ami a .hdf fájl alapján elvégzi helyettünk a szükséges beállításokat, és a fordítást. Ennek használatáról részletesebben az 5.2.2 fejezet szól.

4 Linux kernel

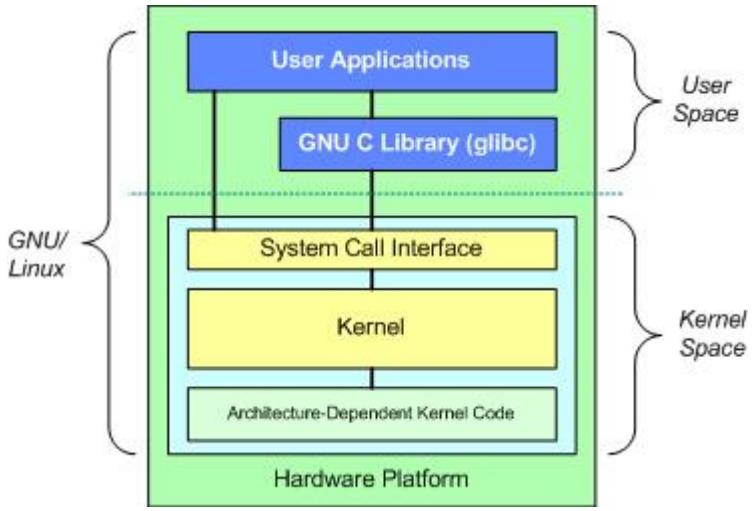
4.1 A Linux operációs rendszer története[9]

A Linux operációs rendszer története az 1990-es években kezdődött. Ekkorra jellemző, hogy a számítógépekhez kapott operációs rendszerek alapvetően zárt forráskódúak voltak (IBM MS-DOS, Mac OS, Unix), így azok pontos működését csak egy szűk fejlesztői kör ismerhette. Ez motiválta a holland származású Andrew S. Tanenbaum professzort egy MINIX nevű egyszerű, oktatási célú Unix szerű operációs rendszer megírására, amit erről a témáról szóló könyvéhez floppy formában mellékelt is. Ez volt az első alkalom, hogy egy operációs rendszer forráskódját szélesebb közönség megismerhette. Ennek köszönhetően sorra alakultak a MINIX működéséről szóló levelezési listák, sok programozó és fiatal egyetemista kezdett a témával részletesebben foglalkozni. Egy ilyen hallgató volt Linus Torvalds is, aki a Helsinki egyetemen tanult. Mivel a MINIX a könnyű megérthetőség miatt egyszerű felépítésű volt, egy másik, nagyobb teljesítményű operációs rendszer fejlesztésébe fogott bele. A kezdeményezést sokan támogatták, hiszen nem volt a piacon ilyen ingyenes szoftver, a GNU projekt kernelje pedig még nem volt kész (ezt a mai napig sem fejezték be, mivel helyette a Linux használható). A későbbiekben ebből nőtte ki magát a ma használt Linux. Az operációs rendszert GPL licenccel látták el, ami szerint bárki szabadon módosíthatja a forráskódot, és akár pénzért is árulhatja azzal a feltétellel, hogy a módosított forráskódot is nyilvánossá teszi, és az új verzió vagy program is GPL licenc alá esik.

Szigorúan nézve a Linux kifejezés csak a kernelre vonatkozik, de a hétköznapi szóhasználatban a Linux disztribúciókra is ezt az elnevezést használjuk. Egy disztribúció alatt egy teljes operációs rendszer csomagot értünk, ami a Linux kernel mellett összeválogatott, jellemzően GPL licenc alá eső nyílt forráskódú felhasználói programokat is tartalmaz. Egy asztali számítógépre készített disztribúcióban például általában találunk grafikus felhasználói felületet biztosító X szervert, ablakkezelőt, fordító programokat, fejlesztői könyvtárakat, emellett új programok telepítését megkönnyítő csomagkezelőket, irodai programokat, web böngészőt. Mivel az elemek nagyrészt egymástól függetlenek, így igen sok különböző disztribúció készíthető.

4.2 Linux operációs rendszer felépítése[10][11]

A GNU/Linux operációs rendszer vázlatos felépítése a 4.1 ábrán látható:



4.1. ábra GNU/Linux operációs rendszer felépítése

Egy operációs rendszer alapvető feladata, hogy a fizikai réteg részleteit, protokolljait elfedve egy szabványos, általános felületet biztosítson a felhasználói alkalmazások számára. Ennek a megközelítésnek az előnye, hogy a programjainknak nem kell tudniuk, pontosan milyen fizikai rendszeren futnak, mivel a hardver vezérlését a kernel, illetve az abba beépülő eszközmeghajtók végzik. Ezeknek kell ismerniük, hogy az adott rendszerben milyen buszok vannak, azokon milyen eszközök találhatók, továbbá hogy az egyes eszközökkel milyen módon lehet kommunikálni. Ez a réteg alkotja a kernel architektúrafüggő részét. Bár a rendszerhez utólag is illeszthetők eszközmeghajtók, a kernel egy részét kimondottan a cél platformra kell fordítanunk ahhoz, hogy az egyáltalán el tudjon indulni. Ide tartoznak azok a kódrészletek, amik például a processzor belső vezérlő regisztereit, az interrupt rendszert, a cache-ket vagy az MMU-t kezelik. Mindez jól megfigyelhető a Linux Cross Reference ([12]) segítségével. Ez egy olyan weboldal, ahol a különböző verziójú kernel forráskódokat tekinthetjük meg, illetve kereshetünk bennük. A hardver specifikus kódrészletek a */arch* könyvtárban helyezkednek el, ami alapján megállapítható, hogy a Linux-ot eddig milyen platformokra portolták. Néhány ismertebb példa: AVR32, ARM, Microblaze, NiosII, x86.

A hardverfüggő kód külön rétegbe helyezésével a kernel többi része architektúra függetlenné válik, mivel csak az előbbi által biztosított felületet használja. Ezen a szinten történik az operációs rendszer szolgáltatásainak implementálása, ami alapvetően a folyamat- ,állomány- , periféria- és hálózatkezelést takarja.

A folyamatkezelő alrendszer teszi lehetővé, hogy egyidejűleg több felhasználói program futhasson párhuzamosan. Alapvető célkitűzés, hogy mindezt úgy valósítsa meg, hogy a futó folyamatok számára úgy tűnjön, a CPU-t és a memóriát teljes egészében ők használhatják.

Az állománykezelő modul külső adattároló perifériákon fájlrendszerek létrehozását és kezelését teszi lehetővé. A fájlrendszerek állományokból épülnek fel, ezek segítségével tárolhatjuk adatainkat, vagy érhetjük el a rendszerhez csatlakozó eszközöket. Az állományok könyvtárak felhasználásával fa struktúrába szervezhetők, amiben név alapján kereshetünk.

A hálózati réteg feladata a számítógép hálózaton keresztüli kommunikációjának megvalósítása. Sokféle különböző protokollt implementál (IPv4, IPv6, TCP, UDP, stb.), és a felhasználói programok felé socketeket kínál.

A perifériakezelő a rendszer hardver elemeinek nyilvántartását, vezérlését és elérhetővé tételét látja el. Az eszközöket működtető driverek vagy a kernelbe vannak befordítva, vagy utólag modulként illeszthetők a rendszerbe.

A fenti funkciók ellátására a kernelek több különböző struktúrája alakult ki, melyek közül a Linux alapvetően az ún. monolitikus megközelítést használja. A monolitikus kernel felépítésére jellemző, hogy minden operációs rendszer szolgáltatás ugyanabban a kernel számára fenntartott memóriatartományban fut magas privilégium szinten. Az egyes funkciókat megvalósító modulok elválasztatlanul egymás mellett futnak, a kernel összes adatára rálátnak. A struktúra előnye, hogy mivel nincsenek a kernelen belül további rétegek, az operációs rendszer gyorsan, hatékonyan tud működni. Hátránya viszont, hogy egy rosszul megírt modul az egész rendszert működésképtelenné tudja tenni, amin csak az újraindítás segíthet, továbbá hogy új modul beillesztése csak a kernel újrafordításával tehető meg. Bár a Linux alapvetően monolitikus struktúrájú, fejlődése folyamán a hátrányok mérséklése érdekében lehetővé tette a kernel modulok futás közbeni betöltését, ami jelentősen egyszerűbbé teszi azok írását, telepítését.

A 4.1-es ábra következő egysége a System Call interface. Ezt magyarul rendszerhívási felületnek nevezhetjük, ez választja el a felhasználói programokat és az operációs rendszer kerneljét. Rendszerhívásnak nevezzük azt a folyamatot, amikor egy felhasználói program az operációs rendszert egy általa nyújtott szolgáltatás végrehajtására kéri. Alapvetően egy program biztonsági okokból csak a saját virtuális

címterét éri el, nem látja sem a többi programot, sem a kernelt, sem a rendszerben megtalálható hardver elemeket. Minden olyam művelet esetén, ami nem tehető meg a saját címterében, a programnak egy rendszerhívással a kernelhez kell fordulnia. Ennek megvalósítása architektúrafüggő, de általában a rendszerhívás kódját és az ahhoz szükséges paramétereket a processzor regisztereiben, vagy a stack-en adják át az operációs rendszernek, majd egy speciális utasítással megszakítják a program futását. Ennek hatására a processzor felhasználói módból kernel módba vált, ahol eléri a memóriában elhelyezkedő kernel függvényeit és adatait. A kért feladat elvégzése után a program visszatér felhasználói módba, és folytatja a végrehajtást.

Bár a felhasználói programokból elvileg közvetlenül is lehetséges lenne rendszerhívások használata, azonban ez architektúrafüggő, és akár assembly utasítások használatát is igényelné, megszüntetve ezzel a C nyelven írt kód portolhatóságát. Ehelyett a szabványos libc könyvtárak használata célszerű, amik elfedve a kernel rendszerhívási felületét egy egységes programozói interfészt nyújtanak.

4.3 Az eszközfa[14]

Számítógépes rendszerek esetén fontos kérdés, hogy az operációs rendszer hogyan szerezzen információt a rendelkezésre álló hardver komponensekről, perifériákról. Asztali számítógépek esetén egyrészt erre a célra szolgál az ACPI standard, másrészt pedig az ott használt buszok (PCI,USB) eleve lehetővé teszik a rájuk kapcsolódó perifériák felfedezését, illetve azok paramétereinek lekérdezését.

Beágyazott rendszerekben azonban ezek a megoldások nem alkalmazhatóak. Egyrészt az ACPI-t jellemzően nem használják, másrészt pedig az ott előforduló buszok (AMBA, AXI) nem nyújtanak az asztali rendszerekben lévő társaikhoz hasonló detektálási szolgáltatásokat. A problémára a legkézenfekvőbb, de nem túlságosan szerencsés megoldás az volt, hogy a hardver információkat, paramétereket, illetve a szükséges drivereket mind beépítették a kernel forráskódjába. A Linux-ot használni képes SoC kártyák számával azonban így a kernel variánsok száma is megnövekedett, ami átláthatatlanságot és nehezen kezelhetőséget eredményezett.

Jobb megoldás lenne, ha a hardver leírás kikerülne a kernelből, így annak egy lefordított példánya több, alapvetően azonos architektúrájú (ARM, x86), de eltérő perifériákkal rendelkező rendszeren is működőképes lenne. Így egy új kártyára való

portolásnál a kernel nem, csupán a hardver leíró adatok módosítandók, illetve természetesen mellékelni kell a megfelelő drivereket.

Ennek a megoldási módnak egy lehetséges realizációja az Open Firmware-ből származtatott eszközfa (device tree) használata. Ez alapvetően egy fa struktúrájú adatszerkezet, amiben minden busz vagy eszköz egy-egy csomópontot alkot, és ezek a csomópontok kapcsolódnak egymáshoz. A relatív viszony mellett megtalálhatóak a fában az eszközök paraméterei, mint például a használt memória terület címe, az órajelforrás neve, vagy éppen az, hogy milyen driverekkel kompatibilis az adott eszköz.

Az eszközfa forrásai szöveges fájlok, amikben olvasható formában vannak az adatok feltüntetve. Ennek kiterjesztése „dts” vagy „dtsi”, ez utóbbiak a C nyelvben használt header-ekhez hasonlóan más fájlokba illeszthetőek be egy include direktíva segítségével.

A device tree szintaxisára a következő kód mutat egy egyszerű példát:

```
/include/ „my_dev_tree_extension.dtsi”

/{
      prop1 = „value1”;
      node1@FF000000{
                  node_prop1 = „node_val”;
                  int_prop = <2 3 5>;
            };
      label: node2@FFFF0000{
            };
};
```

Az első sorban egy .dtsi fájl beszúrására szerepel, aminek hatására fordítás előtt a megnevezett fájl tartalma bemásolódik az adott sor helyére. Általában a .dtsi fájlok a SoC szintű, a .dts fájlok pedig a kártya szintű adatokat tárolják. Beszúráskor előfordulhat, hogy két fájl is tartalmaz a fa struktúra ugyanazon pontján tartózkodó, ugyanolyan nevű elemet. Ilyen esetben az ugyanazon csomópontra vonatkozó különböző információk egyesítésre kerülnek: a csomópont minden megadott tulajdonsággal rendelkezni fog. Amennyiben egy tulajdonságra két különböző értékadás is létezik, akkor a beszúrt fájl tartalma felülírja az eredetit.

Az eszközfában a csomópontokat kapcsos zárójelek között adjuk meg, melyek előtt szerepel az adott elem neve, ami célszerűen az ellátott funkcióra utal. A név egy „@” jel után kiegészíthető az eszköz elsődleges címével, előtte pedig megadhatók címkék

is „címke-név: „ alakban, amikkel később a megjelölt elemre hivatkozhatunk. A gyökér szint neve kötelezően „/”, és nem követheti cím.

Az eszközfa minden csomópontja tartalmazhat további alárendelt csomópontokat, illetve tulajdonságokat, melyek a következő típusúak lehetnek:

- string:
  ```
  string-prop = „string-val”;
  ```
- integer lista/cella:
  ```
  int-prop = <0xafcd0022 325 0x565>;
  ```
- bináris:
  ```
  binary-prop = [0x34 0x25 0x64];
  ```
- vegyes:
  ```
  mix-prop = „string”, <1 2 3>, [0x00 0x01 0x02];
  ```
- string lista:
  ```
  str-list = „string1”, „string2”, „string3”;
  ```

Gyakran előforduló szabványos tulajdonságok:

Általános tulajdonságok:

- *compatible*: Sztring lista tulajdonság, amit az operációs rendszer az eszközt működtető driver kiválasztásához használ. Ebben minden elem *„<gyártó>,<típus>”* formában van megadva, amik közül a legelső sztring az eszköz saját gyártóját és típusát adja meg, a többi csupán azokat jelöli, amikkel meghajtó szempontjából kompatibilis.

Címzéssel kapcsolatos tulajdonságok:

- *reg*: Az adott eszközhöz tartozó címtartományokat tartalmazza listában felsorolva. < *base_address1 length1 base_address2 length2 …*> alakban, ahol minden *address* és *length* mező *#address-cells* és *#size-cells* számú 32 bites értékek listáját takarja. Fontos, hogy amennyiben az eszköz rendelkezik címtartománnyal, akkor a név után a *reg* bejegyzés első elemét kell megadnunk.
- *#address-cells*: Meghatározza, hogy a közvetlen gyermek csomópontokban hány 32 bites érték reprezentálja a reg tulajdonságok báziscímeit. 32-bites rendszerekben ez jellemzően 1, 64 bitesnél pedig 2.

- *#size-cells*: Az address-cells párja, megadja, hogy hány 32 bites érték alkotja a reg tulajdonságok hossz mezőit a közvetlen gyermek csomópontokban. Alapértelmezésben 1 értékű.
- *ranges*: Megadja a saját címtartomány (amiben a gyermek csomópontok címzése érvényes) és a szülő címtartomány közötti összerendelést. Alakja: *< own-base1 parent-base1 length1 own-base2 parent-base2 length2…>* . Ennek jelentése az, hogy a saját buszon értelmezett, own-base kezdőcímű és length hosszú memóriatartomány kívülről a parent-base címtől érhető el. Az *own-base* és *length* mezők hosszára a saját *#address-cells és #size-cells* mezők, míg a *parent-base*-re a szülőé érvényesek. Amennyiben a címleképezés 1:1, azaz a gyermekek kívülről közvetlenül elérhetőek, akkor ezt egy érték nélküli „ranges;” sorral jelezzük.

  Megszakításkezeléssel kapcsolatos tulajdonságok:

- interrupt-controller: Üres tulajdonság, azt jelzi, hogy az adott eszköz megszakításkezelő.
- interrupt-parent: Az eszközhöz tartozó megszakításkezelőt jelöli ki. Ez a tulajdonság örökölhető, így a gyermek csomópontokra is vonatkozik, ha azok nem definiálják felül.
- interrupts: A használt interrupt vonalat azonosító integer lista.
- #interrupt-cells: Megadja, hogy az interrupt mezőben hány 32 bites érték reprezentál egy használt vonalat. Például „#interrupt-cells=2” esetén „interrupts=< 3 1>” alakú lehet. Az egy vonalhoz tartozó számok jelentése a megszakításkezelőtől függ, például az első a megszakítás számát, a második pedig az érzékenységet jelölheti.

  Hivatkozáshoz szükséges tulajdonságok:

- phandle: Az adott csomópontot azonosító 32 bites érték, ami az eszközfában egyedi. Az azonosítót később olyan helyeken használjuk, ahol másik csomópontra szeretnénk hivatkozni, mint például az interrupt-parent tulajdonság esetén. Lehetőség van a phandle manuális megadására, de ennek hiányában a dtc fordító automatikusan létrehozza, amennyiben hivatkozás miatt szükséges.

Gyakran fordul elő, hogy egy tulajdonságban másik csomópontra kell hivatkoznunk. Mint említésre került, alapvetően erre a phandle-k szolgának. Sokkal átláthatóbb kódot kapunk azonban, ha ehelyett címkéket vagy elérési utat használunk.

Elérési útvonal megadása esetén „/”-el elválasztva felsoroljuk, hogy egy adott csomóponthoz hogyan lehet eljutni. Hivatkozásként való használathoz még ki kell egészíteni egy &-jellel és kapcsos zárójelekkel a következő módon: interrupt-parent=<&{/amba/irq_chip@10000000}>. Fordítás során a compiler ezt úgy oldja fel, hogy a cél csomópontban egy phandle-t definiál, és annak értékét adja meg az útvonal helyett.

Az elérési útvonal alternatívájaként használhatóak a címkék, melyeket az elnevezni kívánt egység elé „:”-al elválasztva írhatunk (például lásd a csomópontok elnevezését). Hivatkozásokban a következő formában használandók: interrupt-parent=<&címke-név>;. A feloldás itt is phandle definiálásával, és a címke annak értékére való lecserélésével történik, így maga a címke nem jelenik meg a lefordított kimenetben.

Végül még megemlítendő az aliases és a chosen csomópont, amik közvetlenül a gyökér alatt helyezkednek el. A /aliases elemben „álnév”=”elérési út” alakban neveket rendelhetünk létező eszközökhöz, és ez az elnevezés a címkékkel ellentétben megjelenik a lefordított eszközfában is. A /chosen alatt pedig lehetőség van kernel paraméterek beállítására, illetve megadható a rendszer által alapértelmezetten használt be és kimenet csomópontja.

Az eszközfa forrásának összeállítása után azt egy, a számítógép által könnyebben értelmezhető bináris formára fordítjuk, melynek kiterjesztése „.dtb”. A fordításhoz használt device tree compiler forráskódja megtalálható a kernelforrás */scripts/dtc* mappájában, amiből a kernel fordítása során előáll a használható fordító. Megemlítendő, hogy a lefordított fájl az „fdtdump” programmal visszafejthető olvasható formára, amiben jól látható például a hivatkozások phandle-lel való helyettesítése.

A rendszer indulásakor a lefordított eszközfát a kernellel együtt a memóriába kell tölteni, amit legtöbbször a bootloader végez el. A kernel futásának már korai szakaszában elkezdi feldolgozni az eszközfa tartalmát: megkeresi a gyökér könyvtárban található compatible tulajdonságot, illetve a chosen csomópontot, és az azokban található paramétereknek megfelelően kezdi felkonfigurálni a rendszert. A boot folyamat egy

későbbi szakaszában ismét felkeresi az eszközfát, amikor is az egészet átalakítja egy hatékonyabban használható formára. (ezt az angol terminológia „unflattening”-nek nevezi) Az átkonvertált eszközfát ezután végigolvassa, majd a benne található információkat megpróbálja beilleszteni a 4.4-es fejezetben részletezett driver modellbe.

4.3.1 Az eszközfa dinamikus kiegészítése – overlay rendszer [15][16][17]

Az eddig bemutatott eszközfa koncepció alapvetően statikus rendszerekre lett kialakítva, ahol már fordítási időben pontosan lehet tudni, hogy milyen hardver elemek állnak rendelkezésre. Vannak azonban olyan beágyazott alkalmazások, ahol futási idő közben is megváltozhat a processzor körüli hardverkonfiguráció. Egyszerű példaként gondolhatunk a prototípusok készítésére előszeretettel alkalmazott egykártyás számítógépekre, mind például a BeagleBone vagy a RaspberryPi: ezekhez működés közben is csatlakoztathatunk kiegészítő kártyákat, és kívánatos lenne, ha a rajtuk található perifériák a rendszer újraindítása nélkül beregisztrálhatóak lennének. De ide tartoznak még a különböző FPGA chipekben soft-core-ként, vagy azok mellett fizikailag megvalósított processzorokon futó Linux-os rendszerek (ilyen a Zynq-7000-es SoC) is, amik esetén a hardver részleges vagy teljes rekonfiguráció után megváltozhat.

A fentiekhez hasonló eszközök támogatására hozták létre az overlay mechanizmust, amivel futás közben módosítható a rendszer eszközfája: csomópontokat vagy tulajdonságokat adhatunk hozzá, esetleg az egyes paraméterek értékeit változtathatjuk meg. Ennek alkalmazásához első lépésként el kell készítenünk azokat az eszközfa kiegészítéseket (overlay) tartalmazó fájlokat, amik az általunk bevezetni kívánt új információkat tartalmazzák. Ennek szintaktikájára a következő kódrészlet mutat példát, ami a szakdolgozat során elkészített kapcsoló periféria leírása egy demonstrációs célokat szolgáló „test” csomóponttal kiegészítve:

```
/dts-v1/;
/plugin/;

/{
     compatible =  "xlnx,zynq-7000";

     fragment@0{
          target=<&amba_pl>;
          __overlay__{
               sw: sw@43c10000 {
                    compatible="xlnx,my-axi-sw-1.0";
                    reg=<0x43c10000 0x10000>;

               };/*sw*/
```

```
                my_test: test {
                     selected-switch = <sw>;
                }; /*test*/
           }; /*overlay*/
      }; /*fragment*/
};
```

A legelső újítás a /plugin/ sor hozzáadása, amivel a fordító számára jelezzük, hogy overlay-t készítünk. (Valójában ennek köszönhetően nem dob hibát az ismeretlen csomópontokra való hivatkozáskor, és készíti el a *__fixups__* csomópontot, részleteket lásd később.)

Ezt követően megadjuk a cél eszközre vonatkozó *compatible* mezőt, majd létrehozunk egy fragmentet (töredék), ami az eszközfában alkalmazni kívánt változásokat foglalja magában. Ebben egyrészt kijelöljük, hogy melyik csomópontot akarjuk módosítani (target=<&label-to-target-node>), majd egy *__overlay__* bejegyzésben felsoroljuk mindazt, amit a cél elemhez hozzá kívánunk fűzni. Konkrétan a fenti esetben a gyökér alatt elhelyezkedő amba_pl buszt egészítjük ki egy új gyermek csomóponttal, ami a hozzáadott kapcsoló periféria adatait tartalmazza.

A forrásfájl elkészítése után a hagyományos eszközfához hasonlóan itt is a fordítás következik. Ehhez azonban a megszokott eszközfa fordító nem használható, hiszen olyan címkékre hivatkozunk, amik nincsenek definiálva a fájlban. Emiatt az overlay-ek készítéséhez egy patchelt verzióra van szükségünk, ami a [18] forrás alatt található script segítségével telepíthető.

A patchelt verzió újdonsága a „-@” parancssori kapcsoló, aminek hatására a fordító a következő csomópontokat adja a kész fához közvetlenül a gyökér alá:

- __symbols__: Ebben a fordító felsorolja az összes definiált címke nevét, és a cél elérési útvonalát, így a címkék fordítás után is elérhetőek maradnak. Lényegében hasonló célt szolgál, mind a /aliases csomópont.
- __fixups__: Itt találhatóak azok a címkék, amiket a fordító nem tudott feloldani, mivel azok nincsenek definiálva az adott fájlban. A címkék neve után egyenlőségjellel megadja azokat a tulajdonságokat, amik a hivatkozást tartalmazzák.
- __local_fixups__: Azon tulajdonságokat jelzi, amikben fájlon belüli hivatkozás található.

Mindezekre példaként tekintsük a kapcsoló perifériát leíró fájl lefordított változatát *fdtdump*-pal visszafejtve:

```
/ {
    compatible = "xlnx,zynq-7000";
    fragment@0 {
        target = <0xdeadbeef>;
        #address-cells = <0x00000001>;
        #size-cells = <0x00000001>;
        __overlay__ {
            sw@43c10000 {
                compatible = "xlnx,my-axi-sw-1.0";
                reg = <0x43c10000 0x00000031>;
                linux,phandle = <0x00000001>;
                phandle = <0x00000001>;
            };
            test {
                selected-switch = <0x00000001>;
                linux,phandle = <0x00000002>;
                phandle = <0x00000002>;
            };
        };
    };
    __symbols__ {
        sw = "/fragment@0/__overlay__/sw@43c10000";
        my_test = "/fragment@0/__overlay__/test";
    };
    __fixups__ {
        amba_pl = "/fragment@0:target:0";
    };
    __local_fixups__ {
        fixup = "/fragment@0/__overlay__/test:selected-switch:0";
    };
};
```

A legelső észrevétel a kóddal kapcsolatban a fragment csomópont target paraméter értéke lehet. A forrásban <&amba_pl> szerepel, de mivel ezt a fordító nem tudta feloldani, ezért ide egy helykitöltő értéket írt, majd megadta a fixups elemben, hogy amennyiben az amba_pl phandler értéke ismertté válik, akkor azt a fragment0 target paraméterének legelső helyére szükséges beírni.

A továbbiakban definiálásra került két új csomópont egy-egy címkével együtt. A címkéket a symbols elemben találjuk, a hivatkozott node elérési útvonalával.

Végül a local_fixups alatt a fájlon belüli hivatkozások szerepelnek. Ezek felsorolása azért fontos, mert az overlay beszúrása során módosulnak az abban található phandle értékek, így ezeket a hivatkozásokat is korrigálni kell.

Az így megkapott lefordított device tree blob fájl készen áll arra, hogy a futó rendszerhez hozzáadják. Egy overlay beszúrása során a következő lépéseket hajtja végre a kernel:

1. Meghatározza az aktuális eszközfa legnagyobb phandle értékét.
2. Az így kapott számmal megnöveli az overlay-ben található phandle azonosítókat, hiszen azoknak az egész fában egyedinek kell lenniük.
3. A local_fixups mezőben felsorolt belső hivatkozások értékét megnöveli, hogy kövessék a 2. lépés hatását.
4. Végignézi a fixups csomópontban felsorolt címkéket, mindegyiket megkeresi az aktuális eszközfában, majd a phandle értéküket beírja azokba a tulajdonságokba, amik a címke neve után szerepelnek.

A létrehozott rendszer további kedvező tulajdonsága, hogy a beszúrt overlay-eket nem olvasztja be az eszközfába, hanem azokat egy külön ráhelyezett rétegként kezeli, lehetőséget adva azok eltávolítására is. Természetesen abban az esetben, ha több overlay kerül egymásra, akkor mindig csak a legfelső távolítható el.

4.4 Linux kernel eszközmodellje [19]

A Linux által mai napig használt eszközmodell a 2.5-ös kernel verzióban jelent meg először. Ennek célja, hogy egységesen kezelni és reprezentálni tudja a rendszerben előforduló eszközöket és az azokat működtető drivereket az addigi egyedi megoldásokkal szemben. Az eszközmodell általános struktúrákat definiál, amiket a kernel modulok kibővítve használhatnak fel, biztosítva így a bizonyos szintű egységességet és az egyedi igényekhez való alkalmazkodást. A megoldás további jó tulajdonsága, hogy a rendszer felépítését egy virtuális fájlrendszeren keresztül az user space-ben is megjeleníti, ahonnan a paraméterek így könnyedén elérhetőek.

4.4.1 sysfs fájlrendszer

A fent említett megjelenítésre használt fájlrendszer a sysfs, ami általában a */sys* könyvtárba kerül felcsatolásra. Célja szempontjából hasonló a tipikusan */proc* alatt elérhető procfs-hez, mivel mindkettő az operációs rendszer működéséről kívánja tájékoztatni a felhasználót. A kernel korábbi verzióiban a hardveres információ megjelenítésére is a procfs-t használták, aminek következtében az egyre inkább

átláthatatlanná kezdett válni, tovább motiválva a sysfs megalkotását. Mára a kettő között egyfajta munkamegosztás alakult ki: a procfs tárolja a folyamatokkal, a sysfs a rendszert alkotó eszközökkel és meghajtókkal kapcsolatos információkat.

A */sys*-be belépve több, egy-egy alrendszert reprezentáló könyvtárat találhatunk.

Néhány fontosabb ezek közül:

- */sys/bus*: Ez tartalmazza a rendszerben fellelhető buszok listáját, illetve azok adatait. Mindegyikhez egy-egy alkönyvtár tartozik, amikben felsorolásra kerülnek a rá csatlakoztatott eszközök *(/sys/bus/<name>/devices)* és a buszra írt meghajtóprogramok *(/sys/bus/<name>/drivers)*. A kernel e két könyvtár tartalma alapján próbálja összepárosítani az eszközöket a velük kompatibilis driverekkel.
- */sys/devices*: A rendszerben megtalálható eszközök hierarchikus listája.
- */sys/class*: Az eszközöket jeleníti meg funkció szerint csoportosítva (pl. input, block, tty). Ezen keresztül a devices alkönyvtárhoz képest egy sokkal átláthatóbb képet kapunk a rendszerben megtalálható perifériákról, ezért elsősorban ebben érdemes keresni.
- */sys/modules*: A betöltött modulokról tartalmaz információkat, például azok paraméterei, attribútumai itt jelennek meg.
- */sys/firmware*: A Zynq-7000-es rendszeren ez alatt megtalálható a rendszer aktuális eszközfája és annak minden paramétere.

Észrevehető tehát hogy az eszközeink három aspektusból is vizsgálhatóak a *bus*, a *devices* és a *class* könyvtárakon keresztül. Valójában a tényleges bejegyzések a *devices* mappában helyezkednek el, a többiben csupán szimbolikus linkekkel utalnak rájuk.

Alacsony szinten a sysfs fájlrendszer mögött egy kobject struktúrákból felállított hierarchia található. A kobject-ek olyan struktúrák, amik segítségével a kernelben egyszerű objektum orientáltság valósítható meg. Ezek többek között tartalmaznak egy referencia számlálót, ami azt mutatja meg, hogy a kernelben hány hivatkozás létezik az adott objektumra. Amennyiben ennek értéke 0-ra csökken, akkor a foglalt memóriaterület felszabadítható. Ebből következik, hogy kobjectet tartalmazó struktúrákat soha nem szabadítunk fel explicit módon, hiszen ez a kernel feladata, helyette speciális függvények állnak rendelkezésre, amikkel ezt a referenciaszámlálót csökkenthetjük. Arról, hogy a

kobject ténylegesen milyen objektumot reprezentál egy ktype struktúrából nyerhetünk információt, amire minden kobject egy-egy pointerrel mutat. Ez meghatározza többek között, hogy felszabadítás során milyen műveleteket kell elvégeznünk, vagy, hogy milyen attribútumok tartoznak az adott objektumhoz. A kobject-ek másik fontos tulajdonsága a hierarchikus rendszerbe való szervezhetőségük, amit két féle módon is megtehetünk. Egyrészt minden struktúrában található egy másik, a szülő kobject-re mutató parent pointer, másrészt úgynevezett kset-eket is létrehozhatunk, amik azonos típusú objektumok listában történő tárolását teszik lehetővé. Az így előálló rendszer az alapja a sysfs-ben történő reprezentációnak. Ebben minden kobject-hez egy-egy könyvtár tartozik, ami a szülő objektum könyvtára alatt helyezkedik el. Ezen belül a gyermek kobjekteken kívül megjelennek az attribútumokat képviselő írható, olvasható állományok is, amik az adott eszközzel, busszal vagy meghajtóval (lást később) való kommunikációt teszik lehetővé. Ez a rendszer felelős az úgynevezett hotplug üzenetek generálásáért is, amikkel a kernel értesíteni tudja a user space-t a bekövetkezett eseményekről. Végül még megjegyzendő, hogy kobject-ek szinte soha nem léteznek egymagukban, majdnem mindig más struktúrákba ágyazódnak be. A kernelben található Linux driver model core is ilyen csomagoló struktúrákkal alakítja ki azt a programozói felületet, amin keresztül új elemek adhatóak a rendszerhez. Ebből következően meghajtóprogramok írása során a fentebb említett kobject mechanizmussal a fejlesztő nem is találkozik közvetlenül.

A Linux eszköz modell a rendszert buszokból, azokra kapcsolódó eszközökből, illetve azokat működtető meghajtóprogramokból álló hierarchikusan felépített halmazként kezeli. Minden elemhez definiál általános, kobject-eken alapuló struktúrákat, amiket kitöltve és beregisztrálva hozható létre új komponens. A továbbiakban ezt modellt tekintjük át.

4.4.2 Buszok

A buszok olyan elemek, amikhez eszközök kapcsolódhatnak, illetve amik maguk is további buszra csatlakozhatnak. A buszt reprezentáló struktúra:

```
struct bus_type {
char *name;
struct bus_attribute *bus_attrs;
struct device_attribute *dev_attrs;
struct driver_attribute *drv_attrs;

int (*match)(struct device *dev, struct device_driver *drv);
int (*probe)(struct device *dev);
int (*remove)(struct device *dev);
```

```
/*…*/
struct subsys_private *p;
};
```

Minden buszhoz a rendszer nyilvántartja a rá csatlakoztatott eszközöket, illetve az arra megírt meghajtóprogramokat, és megpróbálja a kompatibilis elemeket összerendelni. Egy új eszköz beillesztésekor megvizsgálja az összes adott buszon ismert drivert, hátha valamelyik kompatibilis azzal. Hasonlóképpen jár el, ha új driver kerül a rendszerbe, ilyenkor a meghajtóval még nem rendelkező eszközökhöz próbálja azt hozzárendelni. Azt, hogy egy meghajtó és egy eszköz kompatibilis-e, a buszhoz beregisztrált match függvény meghívásával próbálja kideríteni a kernel driver core része, ami egy egyszerű logikai igen/nem (1/0) értékkel tér vissza. Találat esetén bejegyzi az eszköz-t a meghajtó-, illetve a meghajtót az eszköz struktúrájába (lásd később), ellenkező esetben pedig tovább próbálkozik a párosítással.

Igény esetén lehetőség van a buszhoz egyedi attribútumok hozzáadására is, amik a sysfs-ben állományként lesznek elérhetők (*bus_attrs*). Emellett megadhatók olyan alapértelmezett attribútumok is, amik a buszhoz kapcsolódó eszközökhöz illetve meghajtókhoz automatikusan hozzárendelésre kerülnek (*dev_attrs, drv_attrs*).

Saját busz implementálásakor általában szokás saját eszköz és meghajtó struktúrákat is definiálni hozzájuk tartozó regisztráló függvényekkel együtt. Ezek tartalmazzák a későbbiekben részletezett *device* illetve *device_driver* struktúrákat, amiket egyedi, a buszra jellemző paraméterekkel egészítenek ki. A saját regisztráló függvények feladata, hogy ezen bennfoglalt struktúrák buszra vonatkozó mezőit kitöltse, illetve meghívja rájuk az általános regisztráló, felszabadító függvényeket. Egy ilyen megoldásra látunk majd példát a fejezet végén, ahol a platform busz kerül röviden bemutatásra.

A busz beregisztrálása után egy új, ahhoz tartozó könyvtár jön létre a */sys/bus* mappában, azon belül pedig a *devices* és *drivers* alkönyvtárban a kapcsolódó eszközök és meghajtók lesznek majd elérhetőek.

4.4.3 Eszközök

A Linux eszköz modellben szereplő minden eszközhöz tartozik egy általános leíró struktúra, aminek kivonatos definíciója a következő:

```
struct device {
     struct device        *parent;
     struct device_private     *p;
     struct kobject kobj;
     struct bus_type *bus;     /* type of bus device is on */
     struct device_driver *driver;  /* which driver has allocated this
                            device */

     void                 *driver_data;  /* Driver data, set and get with
                             dev_set/get_drvdata */
     struct device_node   *of_node; /* associated device tree node */
     struct class         *class;
     /*…*/
};
```

Az egyes adattagok szerepe:

- *parent*: szülő eszközre mutató pointer. Általában megegyezik a bennfoglalt kobject szülőjével.
- *p*: A kernel driver core része által kezelt privát adatok.
- *kobj*: a device-hoz tartozó kobject, amin keresztül megjelenik a sysfs-ben.
- *bus*: Az a busz, amire az eszköz kapcsolódik.
- *driver*: Az eszközt vezérlő meghajtó. Sikeres párosítás esetén ide történik a meghajtó bejegyzése.
- *driver_data*: Szabadon felhasználható adattag, ami az adott eszközhöz tartozó egyedi adatokat tartalmazó struktúrára mutathat. Beállítására és lekérdezésére a direkt mód helyett a *set_drvdata* és *get_dvdata* függvények használhatók, de ezek is csupán a mezőnek adnak értéket. Elsősorban a meghajtó program adott eszközzel kapcsolatos adatainak tárolására használatos.
- *of_node*: Amennyiben az adott struktúra az eszközfa alapján lett létrehozva, akkor ez a releváns csomópontra mutat. Egy sor segédfüggvény áll rendelkezésre, amikkel a csomópontban tárolt adatok kinyerhetők.
- *class*: Az eszköz osztályára mutató pointer. Az osztályok leírását lásd később.

Új eszköz hozzáadásához egy ilyen struktúra létrehozása, megfelelő kitöltése, és regisztrálása szükséges. Erre közvetlenül ritkán van szükség, mivel szinte minden busz definiál saját eszközleíró struktúrát, ami ezt az általánosat is tartalmazza, továbbá biztosít saját regisztráló függvényt is, ami a busz specifikus mezőket kitölti helyettünk.

A buszokhoz hasonlóan az eszközökhöz is rendelhetünk egyedi attribútumokat, amennyiben szükséges.

A buszokhoz visszatérve még megjegyzendő, hogy a buszvezérlő maga is egy eszköznek tekinthető, így ezt is regisztrálni szokás a rendszerben. Ennek következtében a busz a */sys/devices* alrendszerben is megjelenik, így a rá kapcsolódó eszközök parent pointere erre állítható.

4.4.4 Eszközmeghajtók

A rendszerben használt eszközmeghajtók általános jellemzésére a következő struktúra használatos:

```
struct device_driver {
      const char      *name;
      struct bus_type      *bus;

      const struct of_device_id *of_match_table;

      int (*probe) (struct device *dev);
      int (*remove) (struct device *dev);
      /*…*/
      struct driver_private *p;
};
```

Az egyes adattagok funkciója:

- *name*: A meghajtó neve, ez fog szerepelni a sysfs fájlrendszerben.
- *bus*: Az a busz, amire kapcsolódó eszközöket a meghajtó kezelni képes.
- *of_match_table*: Itt adhatóak meg azok az eszközfában tárolt tulajdonságok, amik alapján a meghajtóval kompatibilis eszközök meghatározhatók. Általában ebben a *compatible* mező értékére szűrünk rá, mivel ezzel adják meg az eszközök, hogy milyen driverrel tudnak együttműködni.
- *probe*: Amennyiben a busz az eszköz-meghajtó párosítás során kompatibilis párt talál, akkor meghívja a meghajtó *probe* függvényét. Ebben a driver további ellenőrzéseket végezhet, hogy a kapott eszközt

valóban képes-e kezelni. Amennyiben igen, akkor elvégzi a szükséges inicializáló műveleteket, ellenkező esetben pedig visszatérési értékben jelezheti a Linux driver core-nak hogy folytassa a keresést.

- *remove*: A használt eszköz eltávolításakor meghívott függvény, itt végzi el a meghajtó a szükséges felszabadításokat.
- *p*: A Linux driver core adott meghajtóhoz kapcsolódó adatai. Többek között itt tárolódik a driver által kezelt eszközök listája.

Az eszközökhöz és buszokhoz hasonlóan a driverekre is igaz, hogy saját attribútumokkal ruházhatjuk fel őket, emellett általában nem közvetlenül, hanem buszok által definiált más struktúrákba ágyazva fordulnak elő, így valósítva meg az egyedi tulajdonságokat. Erre példát majd a platform_drivernél látunk.

4.4.5 Osztályok

A Linux device modellben az osztályok létrehozásának célja az volt, hogy segítségükkel a rendszerben található eszközök funkcionalitás alapján csoportba sorolhatóak legyenek, elfedve az alacsony szintű, konkrét felépítésre vonatkozó információkat. A rendszerben regisztrált osztályok a */sys/class* könyvtárban találhatóak meg, amikben szimbolikus linkek mutatnak az oda tartozó eszközökre.

Új osztályt a következő makróval hozhatunk létre:

```
struct class* class_create(owner, name);
```

Ebben meg kell adnunk a hívó modul címét (THIS_MODULE pointer használható), illetve az osztály nevét.

4.4.6 Az udev daemon[19]

Daemonoknak olyan felhasználói módú programokat nevezünk, amik a felhasználóval való közvetlen kapcsolat nélkül a háttérben futnak és valamilyen szolgáltatásszerű tevékenységet végeznek. Az udev is egy ilyen program, aminek feladata a kernel által generált hotplug események megfelelő kezelése. Ezeket a sysfs fájlrendszer generálja minden olyan esetben, amikor új kobject jön létre, vagy amikor megszűnik egy, azaz például ha egy új eszközt csatlakoztatunk a rendszerhez. Számunkra az udev azért fontos, mivel ő felel a */dev* könyvtár tartalmának kezeléséért, azaz minden új eszközhöz létrehozza a megfelelő major és minor számmal rendelkező állományt, amihez a

szükséges információkat a */sys/class* alatt található könyvtárakból szerzi. Ahhoz hogy a */dev*-ben dinamikusan létrehozott saját eszközállományunk legyen, új eszközt kell regisztrálnunk a következő függvénnyel:

```
struct device *device_create(struct class *class, struct device *parent,
dev_t devt, void *drvdata, const char *fmt, ...);
```

Ahol:

- *class*: Saját, vagy egy már létező osztályra mutató pointer.
- *parent*: Szülő, a sysfs-ben ez alá kerül be az új eszköz. Lehet NULL is.
- *devt*: Az eszköz azonosítója, erről a 4.5.3.1 fejezetben lesz bővebben szó.
- *drvdata*: Saját adatok, amik a *device_driver data* mezőjébe kerülnek.
- *fmt*: Az eszköz neve, az utána következő tetszőleges számú paraméter segítségével *printk* formában adható meg.

Az így létrehozott eszközhöz automatikusan létrejön az osztályon belüli hivatkozás, továbbá az a dev nevű fájl, ami tartalmazza a minor és major számot, és ami alapján az udev a */dev*-ben állományt tud létrehozni.

Udev helyett egyszerűbb környezetekben (ilyen a PetaLinux is) inkább egy kevésbé erőforrás igényes alternatívája használatos, az *mdev*, ami a fent leírt feladatot ugyanilyen módon látja el.

4.4.7 Konkrét busz bemutatása: a platform busz

A platfrom busz a Linux kernelben egy virtuális, fizikailag általában nem létező busz, melynek feladata az olyan perifériák összefogása, amiket a processzor közvetlenül meg tud címezni, el tud érni. Számunkra a relevanciáját az adja, hogy az eszközfában megadott, PL-ben implementált perifériákhoz a kernel erre a buszra csatlakozó eszközöket készít, így azokhoz platform meghajtókat kell írnunk.

A következőekben áttekintjük a platform busz-t leíró struktúrákat és azok kezelését a */include/linux/platform_device.h* és a */drivers/base/platform.c*-ben szereplő forráskód alapján.

A *platform_bus_type* struktúra:

```
struct bus_type platform_bus_type = {
.name           = "platform",
.dev_groups     = platform_dev_groups,
.match          = platform_match,
.uevent         = platform_uevent,
.pm             = &platform_dev_pm_ops,
};
```

A *name* mezőből látszik, hogy a buszra csatlakoztatott eszközök */sys/bus/platform* könyvtárban lesznek majd elérhetőek. Érdemes még megnézni a *platform_match* függvény definícióját is, amiből kiderül, hogy a busz eszköz-meghajtó párosítás esetén a következő szempontok alapján dönt:

1. Amennyiben a *driver_override* mező beállított, akkor az abban konkrétan meghatározott meghajtót használja.
2. Ellenkező esetben megpróbálja a driverek *of_match_table* mezőit használni, amiben a meghajtók felsorolják, hogy milyen device tree paraméterekkel rendelkező eszközökkel kompatibilisek.
3. Ezt követi az ACPI típusú párosítás.
4. Megvizsgálja a driverben megadott *id_table*-t, amiben a meghajtó a kompatibilis platform eszközök neveit sorolja fel. Egyezés esetén párosít.
5. Végül pusztán az eszköz és meghajtó nevét hasonlítja össze, amik egyezése estén azok kompatibilisnek tekinthetőek.

A *bus_type* mellett még definiálásra kerül egy *struct device* is, ami a busz eszközeinek *parent* mezőjében szerepel majd:

```
struct device platform_bus = {
.init_name      = "platform",
 };
```

A busz létrehoz egy saját burkoló struktúrát is, ami magában foglalja az általános *struct device*-t:

```
struct platform_device {
     const char *name;
     struct device  dev;
/*…*/
    };
```

Ennek beregisztrálásához saját függvényt kínál, ami a belső *device* struktúra buszra vonatkozó mezőit automatikusan kitölti:

```
int platform_device_register(struct platform_device *pdev);
```

Természetesen létezik az ehhez kapcsolódó saját meghajtó struktúra is:

```
struct platform_driver {
     int (*probe)(struct platform_device *);
     int (*remove)(struct platform_device *);
     struct device_driver driver;
     const struct platform_device_id *id_table;
     /*…*/
     };
```

Az ehhez tartozó regisztráló függvény (valójában makró egy belső függvényre):

```
int platform_driver_register(platform_driver *drv);
```

Ez egyrészt elvégzi a bennfoglalt *device_driver* struktúra inicializálását, aminek során a *probe*, *remove* és *shutdown* pointerekhez saját belső metódusokat rendel, amik megfelelő előkészületek után a *platform_driver* saját, külső függvényeit hívják meg.

Mint fentebb már említésre került, a device tree-ben megadott eszközök a platform buszra kerülnek felfűzésre, így ezek a */sys/bus/platform/devices* és a */sys/devices/platform* könyvtárak alatt jelennek meg.

Platform meghajtó készítésekor meg kell írnunk a saját *probe* és *remove* függvényeinket, amiket egy *platform_driver* struktúrában rögzítünk. Ezt követi annak kiválasztása, hogy milyen módon történjen a device-driver párosítás. Tekintve hogy az eszközök a device tree-ből jönnek létre, így célszerű a bennfoglalt *device_driver* struktúra *of_match_table* mezőjének kitöltése. A meghajtó regisztrációja után új bejegyzés jelenik meg a */sys/bus/platform/drivers* könyvtárban, emellett elkezdődik a driver core által irányított párosítás, aminek folyamán minden még meghajtóval nem rendelkező eszközzel és az új driverrel meghívódik a busz *match* függvénye. Találat esetén a meghajtó a *device* struktúra *device_driver* mezőjébe, a device pedig a meghajtó belső listájába kerül bejegyzésre, majd meghívódik a meghajtó *probe* függvénye. Ekkor a driver még visszautasíthatja a párosítást, aminek következtében tovább folytatódik a keresés. Ellenkező esetben az összerendelés véget ér.

Eltávolítás során (akár a device, akár a driver kerül ki a rendszerből) a meghajtó *remove* függvénye elvégzi a szükséges felszabadításokat (a platform buszon ez meghívja

a külső, *platform_driver*-ben megadott *remove* függvényt is), majd az adott komponens kivezetésre kerül a sysfs nyilvántartásából.

4.5 Kernel modul fejlesztés[11]

4.5.1 Bevezetés

Az operációs rendszer feladatai közé tartozik, hogy a felhasználói programok felé egy egységes programozási felületet biztosítson, elrejtse a futtató gép fizikai kiépítésének részleteit. Ennek következtében a megírt programunk eltérő hardver elemekből felépített rendszereken is egyaránt működőképes lesz. Nem kell például foglalkoznunk azzal, hogy az általunk megnyitott fájl milyen típusú merevlemezen, netán USB memórián helyezkedik el, vagy éppen hogy a megnyitott TCP/UDP socket-en átküldött adat milyen fizikai átviteli rétegen halad keresztül. Mindezek kezelését az operációs rendszer látja el, amihez úgynevezett eszközmeghajtókra, driverekre van szüksége. Ezek feladata, hogy az egyedi eszköz paramétereit, címkiosztását, protokollját ismerve megvalósítsanak egy, az operációs rendszer által meghatározott interfészt, amin keresztül az eszközt a rendszer használni tudja. A Unix-szerű operációs rendszerek, így a Linux is, az eszközöket fájlrendszerbeli állományokként reprezentálják, így a drivernek fájlműveletekre kell visszavezetnie az eszközzel való kommunikációt. Ilyenek például a jól ismert *open(), close(), read(), write()* függvények, illetve az általános vezérlésre szolgáló *ioctl()*.

A kernelben történő fejlesztés sok szempontból különbözik a felhasználói módban futó alkalmazások készítésétől, és ezt programozás során végig figyelembe kell venni. Ilyen például a verziófüggőség. Míg az felhasználói programok által használt libc függvénykönyvtárban a szabványos függvények alapvetően nem változnak az idő múlásával, a kernel folyamatos fejlesztés alatt áll, módosulnak a függvényei, bővülnek az általa ellátott funkciók. Ez a folyamat aránylag gyors, amit még a dokumentáció sem nagyon tud követni. Ez megnyilvánul abban is, hogy nem könnyű a témában aktuális szakirodalmat találni. Például az általam sokat használt [11] és [19] forrás is a 2.6.39, illetve 2.6.10-es kernelhez készült, holott a szakdolgozathoz a Xilinx által biztosított legfrissebb 4.4-es verziót használtam. (A legújabb Linux kernel verzió jelenleg a 4.8 (2016 december). Mindazonáltal gyökeres eltérést sehol nem tapasztaltam, az esetleges különbségek a kernel forráskódja alapján könnyen felderíthetőek és kezelhetőek.

Programfejlesztés során a saját magunk által megírt kódok mellett már kész, függvénykönyvtárakban található függvényeket is felhasználunk. Ezek egyrészt a rendszerhívási felülethez biztosítanak hozzáférést (ilyen például a *printf(), fprintf(), fopen()*), másrészt pedig megkímélik a programozót a gyakran szükséges algoritmusok, eljárások saját kezű implementálásától (*qsort(), strlen(), strcpy(), strtok(), strstr(), atoi(),* stb.). Felhasználói program fejlesztésekor ezeket a függvényeket a standard c függvénykönyvtár tartalmazza. Mivel ez user módban fut, kernel kódban nem használhatjuk, helyette a kernel megfelelő header állományaiban (/include/linux) található függvényeket hívhatjuk csak meg. Általánosságban elmondható, hogy a rendszerhívással nem járó libc függvények megtalálhatóak a kernelben is, így nem kell őket újraimplementálni.

Fontos még megemlíteni a felhasználói és kernel mód memóriakezelésének eltérését. Minden, a rendszerben megtalálható felhasználói módú program alapvetően egy saját virtuális címtérben fut, megvalósítva ezzel a processzek egymástól való elszigetelését. 32 bites architektúra esetén ez a virtuális címtér – függetlenül a fizikailag rendelkezésre álló memória méretétől – 4GB méretű. A Linux operációs rendszerben ez két részre osztódik: az alsó 3 GB-ot használhatja ténylegesen az adott program, a felső 1GB területen minden processz esetén a kernel kódja illetve adatstruktúrái jelennek meg. Mivel azonban a program user módban fut, ezt a felső területet nem tudja elérni. Ezt az elválasztást a processzorok általában architektúrálisan támogatják ún. privilégiumszintek bevezetésével, amik meghatározzák, hogy milyen utasításokat, memóriaterületeket használhat az éppen futó programkód. Bár a CPU-k jellemzően több (>2) ilyen szintet képesek kezelni (x86 esetén 4), mind a Windows, mind a Linux ebből csak 2-t használ, amiket user illetve system módnak neveznek. Értelemszerűen a normál programok user, míg a kernel system módban fut, és ez utóbbi szükséges a felső 1 GB-os terület használatához. A két mód közötti átjárást a rendszerhívások teszik lehetővé, amik során a felhasználói program végrehajtása megszakad, és helyette nagyobb privilégium szinten a kívánt funkciót ellátó kernel kód kezd futni.

További kellemetlenség, hogy a kernelhez írt programkódok nehezen debugolhatók, és az azokban levő hibák az egész rendszer stabilitását veszélyeztethetik. A kernel készítői azonban igyekeztek segíteni a fejlesztést, így hiba esetén a kernel - amennyiben az nem túlságosan súlyos - hibaüzenetet generál. Ennek egyik leggyakoribb formája az Oops üzenet, ami után a rendszer általában még működőképes marad, lehetővé

téve például a rendszernapló olvasását. Célszerű azonban ilyenkor is minél hamarabb újraindítani az operációs rendszert, mivel az kiszámíthatatlan állapotba kerülhetett. Az Oops üzenet röviden közli a hiba okát (például hibás pointer hivatkozás), kiírja a hibát okozó processzor számát, az utasításszámláló értékét, a stack tartalmát, a meghívott függvényeket. Ezek alapján a forráskód ismeretében meghatározható a hiba helye.

Abban az esetben, ha a hiba olyan súlyos, hogy abból a kernel nem tud visszatérni, úgynevezett kernel panic üzenetet kapunk. Ez az Oops üzenethez hasonlóan kiírja a rendszer pillanatnyi állapotát, azonban ez után nem folytatja a működését, így ekkor elkerülhetetlen az újraindítás.

A program működésének követésére legegyszerűbben a *printk* függvény használható, amivel a rendszernaplóba írhatunk üzeneteket. Amennyiben nem grafikus felhasználói felületet használunk, akkor ezek a konzolban is megjelennek. A *printk* üzenetekhez 0-7-közötti prioritás szint is rendelhető, amik közül a 0-s KERN_EMERG a legsúlyosabb hibákról értesít, míg a 7-es KERN_DEBUG csupán debuggolási információt jelez. A rendszernaplóba az összes üzenet bekerül, a konzolra való kiíratást viszont a */proc/sys/kernel* könyvtárban található *printk* állománnyal szabályozhatjuk. Ebben megadható, hogy milyen szintnél alacsonyabb prioritásindexű üzenetek jelenjenek csak meg. Például a „*echo 7 > /proc/sys/kernel/printk*” parancs hatására a KERN_DEBUG üzeneteken kívül minden üzenetet megkapunk. Célszerű fejlesztés alatt ezt 8-ra állítani, hogy minden információ kijusson a konzolra, ugyanis kernel pánik esetén nincs módunk a rendszernapló megtekintésére a dmesg segítségével,

További kifinomultabb hibakeresési technikák is léteznek, azonban egyszerű modulok fejlesztése során a fent említett módszerek elegendőek a hibák felismeréséhez.

4.5.2 Modulok

Mint ahogyan már az előző fejezetekben említésre került, a Linux monolitikus, azaz a kernel egésze egy nagy egységet alkot, amik között nincsenek belső interface-k. A tisztán ilyen rendszermagok esetén problémás a kernel funkcionalitásának bővítése, hiszen például minden új eszközmeghajtó beillesztéséhez az egész kernel újrafordítása lenne szükséges. A Linux kernel 1.2-es verzió óta tartalmaz erre egy kényelmesebb megoldást, amivel megkerülhető az újrafordítás. Ez pedig a modulok bevezetése.

A modulok olyan függvénykönyvtárak, amik futás közben a kernelhez linkelhetők, illetve arról leválaszthatók. Meghívhatják a kernel vagy más modul

függvényeit, illetve elérhetővé tehetik sajátjaikat. A modulok betöltése az „*insmod* <*modul-név*>” paranccsal lehetséges, ami végrehajtja a linkelést, majd meghívja a modul inicializáló függvényét. Opcionálisan paraméterek megadására is van lehetőség „*insmod* <*modul-név*> <*paraméter-név*>=<*érték*>” formában. Az *insmod* mellett használható még a *modprobe* parancs is, ami automatikusan betölti a függőségeket is. Ehhez azonban az új modul elérhetőségét be kell regisztrálni a rendszerbe, emiatt kicsit körülményesebb a használata. Az *insmod* párja az „*rmmod* <*modul-név*>, ami a lebontó függvénye meghívása után eltávolítja a megadott modult a memóriából.

Modulok írásához szükségünk van a használt kernel forrására, vagy legalábbis a header fájljaira, illetve egy gcc fordítóra. A létrehozott .c forrásfájl mellett szükséges még egy Makefile is, aminek lefuttatásával kapjuk meg a lefordított modult. A Makefile beállítása megtalálható a [11]–es forrásban.

Egy modul minimális váza a következőképpen néz ki:

```
#include <linux/module.h>

static int __init my_init_func(void)
{
     // do initialization
}
static void __exit my_exit_func(void)
{
     // do cleaning up
}
module_init(my_init_func);
module_exit(my_exit_func);
MODULE_LICENSE(„GPL”);
```

Az include sorban megadott header file a kódban lévő makrókat tartalmazza. Ezt követik az inicializáló és lebontó függvények, amiket a *module_init* illetve *module_exit* makrókkal jelölünk ki. Értelemszerűen ezek hívódnak meg a modul beillesztésekor, eltávolításakor. A *static* előtaggal azt fejezzük ki, hogy ezeket a függvényeket csak a jelenlegi forrásfájlban akarjuk használni. Az *__init* és *__exit* makró alapvetően csak kernelbe fordítás esetén hordoz információt, ekkor az *__init* jelzésű függvényeket a kernel az inicializálás után eltávolítja a memóriából, az *__exit* jelzésűeket pedig eleve be sem tölti, hiszen egy kernelbe fordított modul nem távolítható el. Az esetleges befordítást támogatandóan célszerű azonban ezek használata betölthető modulok készítése esetén is.

4.5.3 Eszközök és eszközvezérlők[11][19]

A Linux driverek a kernel számára alapvetően háromféleképpen reprezentálhatják az általuk kezelt létező vagy virtuális eszközöket: karakteres eszköz, blokkos eszköz esetleg hálózati interfész, melyek között a kommunikációs felületükben van különbség, azaz más típusú függvényeket implementálnak és regisztrálnak be az operációs rendszer felé. Mint már említésre került, a felhasználó szempontjából minden eszköz egy fájlrendszerbeli állományként jelenik meg, és a fájlba történő írással, olvasással, esetleg speciális fájlműveletek végrehajtásával lehet az eszközzel kommunikálni.

A legegyszerűbbek a karakteres eszközök. Ezek nagyvonalakban egyszerű fileként képzelhetőek el: karaktereket írhatunk beléjük illetve olvashatunk ki belőlük. A minimálisan megvalósítandó függvények: *open, close, read, write*. A hagyományos fájloktól abban különböznek, hogy általában nem támogatják a tartalmon belüli mozgást, azaz nincs implementálva a *seek* illetve *mmap* függvény.

Egy fokkal bonyolultabbak a blokkos eszközök. Ezek az előzőekhez hasonlóan állományokon keresztül érhetőek el, azonban nem karakterenkénti, hanem blokkos adatátvitelre képesek. Emellett fontos tulajdonságuk, hogy fájlrendszert tartalmazhatnak, így mögöttük általában valamilyen fizikai adattároló (pl. merevlemez) található.

A fentiektől különbözőek a hálózati interfészek, amik nem rendelkeznek fájlrendszerbeli állomány reprezentációval. Ehelyett a kernel egyedi neveket rendel hozzájuk, amikkel később hivatkozni lehet rájuk (pl. wlan, eth0). Az adatátvitel módja is eltérő, read / write utasítások helyett csomagküldő függvényeket kell a drivernek a kernel felé biztosítania.

A továbbiakban kizárólag a karakteres eszközökkel foglalkozok, mivel a szakdolgozat során megvalósított egyszerű perifériákhoz ezek illeszkednek a legjobban.

4.5.3.1 Karakteres eszközvezérlők

A rendszerben található eszközöket reprezentáló állományok különböznek az adatot tartalmazó fájloktól. Létrehozásuk is eltér: erre a speciális „*mknod <állománynév> <eszköz típus> <major szám> <minor szám>*” parancs szolgál. A paraméterlistában 2 szám jellegű azonosító is megjelenik, a major és a minor szám. Ezek közül a major szám adja meg a kernelnek, hogy melyik eszközmeghajtót kell meghívnia, a minor pedig csupán a meghajtó számára szolgál információval. A Linux operációs rendszerben az

eszközállományok jellemzően a */dev* mappában találhatóak, amikről bővebb információt az „*ls –l*” utasítással kaphatunk. Példaként tekintsük a következő sort:

```
crw--w----  1 root tty       4,   0 okt   17 10:59 tty0
```

Számunkra az 5. és 6. oszlop érdekes, ezek jelentik rendre a major és minor számokat. A major szám alapján meghatározható, hogy a rendszerben milyen driver kezeli a fenti állományt, ehhez a */proc/devices* fájlban kell megkeresnünk a releváns összerendelést. Ennek alapján esetünkben a tty nevű driver biztosítja az állományműveletek megvalósítását.

Amennyiben tehát saját karakteres eszközt akarunk létrehozni, első lépésben major számot kell foglalnunk a rendszerben. Itt kétféleképpen járhatunk el: vagy egy fix azonosítót próbálunk elkérni, vagy rábízzuk a kernelre, hogy válasszon nekünk egy nem használtat. Mivel az első esetben tudnunk kellene, hogy már milyen azonosítók lettek regisztrálva, ezért a második módszer az ajánlottabb, amire a következő függvény szolgál:

```
int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *);
```

Ennek első paramétere a függvény kimenete, ebben kapjuk meg a major és első minor számból álló *dev_t* változót, amiből a *MAJOR(), MINOR()* makrókkal fejthetjük vissza a komponenseit. A második paraméter az első minor számot jelenti, ez általában 0, a harmadik pedig, hogy hány minor számot szeretnénk lefoglalni. Végül még meg kell adnunk, hogy milyen név kerüljön a */proc/devices* bejegyzésbe.

Amennyiben rendelkezünk major azonosítóval, a driverben implementálnunk kell a fájlműveleteket megvalósító függvényeket, amiket a következő definícióval rendelkező *struct file_operations* struktúrában kell összefognunk:

```
struct file_operations {
     struct module *owner;
     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
     ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
     int (*open) (struct inode *, struct file *);
     int (*release) (struct inode *, struct file *);
     /*…*/
};
```

Ebben a fájllal kapcsolatos összes eseményhez rendelhetünk saját kezelő függvényt a struktúra megfelelő pointerének beállításával. Amennyiben bizonyos funkciókat nem akarunk implementálni, akkor ezt a pointerek NULL-ra állításával jelezhetjük a kernel számára. Legegyszerűbb esetben elég csupán 4 függvény, az *open,*

release, read és *write* megírása, amik segítségével egy írható és olvasható csatornát alakíthatunk ki a user és a kernel space között.

Utolsó lépésként már csak a major szám és a fájlműveletek összerendelését kell megtennünk, amire szintén egy struktúra, a *struct cdev* szolgál. Statikus példányosítás esetén az inicializálás a következő függvénnyel történik:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Ebben meg kell adnunk a *cdev* példány, továbbá a fent említett fájlműveleteket összefogó struktúra címét. Az inicializálás után még célszerű az *owner* mezőnek THIS_MODULE értéket adni (ez egy az aktuális modulra mutató makró), majd következhet a kernelbe való regisztráció:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

A paraméterek között a *dev* a *cdev* struktúránk címét, a *num* a major és legelső minor számból *MKDEV(major,minor)* makró segítségével képzett *dev_t* számot, a *count* pedig a használt minor azonosítók számát adja meg.

Mindezek után minden a megadott major számmal rendelkező állományra érkező rendszerhívást a driverünk szolgál ki. Ilyen állományokat user space-ből hozhatunk létre vagy saját kezűleg, vagy az udev(/mdev) démon segítségével.

A driver eltávolításánál természetesen szükség van a fenti műveletek visszavonására. Az ehhez használt függvények a megírt forráskódban megtalálhatóak, itt külön nem részletezem.

A fentebb leírtak elvégzésével tehát létrehoztuk a driverünk user space oldali felületét, amin keresztül a felhasználói programok igénybe vehetik a meghajtó által nyújtott szolgáltatásokat.

4.5.3.2 IO kezelés

Az eszközmeghajtóknak a felhasználó által elérhető felület mellett többnyire szüksége van még a fizikai eszközzel való kapcsolatra is. Természetesen léteznek kivételek, gondolhatunk például a különböző virtuális eszközöket létrehozó meghajtókra, amikhez nem szükséges tényleges hardver elem megléte.

A CPU a rendszerben lévő perifériákat alapvetően két féle módon érheti el. A kevésbé gyakori megoldás az IO portok használata, amikor speciális *in* és *out* utasítások szolgálnak a memóriától különböző címtartományban elhelyezkedő perifériák elérésére.

Ehelyett elterjedtebb kialakítás a memóriába ágyazott IO. Ilyenkor a perifériákban található regiszterek és a memória egy közös címtartományban vannak, a CPU mindkettőt ugyanolyan író és olvasó utasításokkal éri el. Mint korábban már láttuk, a Zynq-7000-es SoC az ilyen periféria illesztést alkalmazza, így bár a Linux mindkét módot támogatja, a továbbiakban csak ezzel foglalkozunk.

4.5.3.3 I/O memória

Amennyiben egy meghatározott I/O memória tartományt használni akarunk, akkor első lépésben le kell foglalnunk azt, elkerülve ezzel, hogy több driver használja ugyanazt a perifériát. A foglaló függvény a következő:

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
char *name);
```

Ezzel elkérjük az operációs rendszertől a *start* címtől kezdődő, *len* hosszúságú fizikai memóriatartományt. A név paraméter a megadott címtartomány mögött található perifériára utal. A már beregisztrált tartományok és nevek a */proc/iomem* fájlban tekinthetőek meg, amik között a függvény lefutása után a sajátunk is megjelenik.

A fizikai memória lefoglalása önmagában még nem elég ahhoz, hogy az a kernel címtérben a driver számára elérhetővé váljon. Ehhez még egy leképezés végrehajtása szükséges, ami a következő függvénnyel kérhető:

```
void *ioremap(unsigned long phys_addr, unsigned long size);
```

Ennek meghívása után visszatérési értékként megkapjuk arra a kernel címre mutató pointert, ahonnan kezdve a kért I/O memória terület látható.

Fontos még megemlíteni, hogy a megkapott memóriacímet portolhatósági okok miatt nem szokás közvetlenül használni, helyette az írást és olvasást is speciális függvények segítségével tesszük meg. Ilyenek az *ioread8, ioread16, ioread32, iowrite8, iowrite16, iowrite32* függvények, amik a megadott címtől kezdve 1/2/4 byte-ot írnak/olvasnak.

A I/O műveletek befejeztével a lefoglalt tartományt fel kell szabadítani, hogy más programok is hozzáférhessenek. Az ehhez szükséges függvények a forráskódban szerepelnek.

4.5.3.4 Megszakítások

A perifériák működéséhez sokszor elengedhetetlen, hogy a CPU-t bizonyos bekövetkezett eseményekről minél gyorsabban értesíteni lehessen. Legegyszerűbb példa erre a rendszeridőzítő, aminek célja, hogy periodikusan lefuttassa az operációs rendszer ütemezőjét.

A megszakítások részletes lefutása architektúránként eltérő, azonban általánosságban elmondató, hogy a megszakítás bekövetkeztekor első lépésben a kernel speciális megszakítás kezelő része kezd futni, ami elvégzi az ilyenkor szükséges beállításokat, beolvassa a megszakításvezérlőből a megszakítás számát, megvizsgálja, hogy az adott vonalra van-e beregisztrált kezelő függvény, és ha igen, meghívja azt. Így a driverben ennek a függvénynek a megírása és beregisztrálása a feladatunk.

A megszakítás kezelő függvények prototípusa a következő:

```
irqreturn_t handler(int irq, void *devid);
```

Mivel az ilyen függvények a processzor speciális állapotában kerülnek végrehajtásra, bizonyos megkötések vonatkoznak rájuk. A legfontosabb, hogy nem végezhetünk semmilyen olyan műveletet, ami alvó állapotba kerüléssel és újraütemezéssel járna. Emiatt többek között a dinamikus memóriaallokációt végző *kmalloc()* csak GFP_ATOMIC flag-el használható, nem mozgathatunk adatokat a kernel-space és user space között, továbbá nem írhatunk a rendszernaplóba. Emellett nem használhatunk mutexeket és szemaforokat sem. A fentiek elkerülése mellett célszerű még törekedni a gyorsaságra, hiszen minél tovább tart a kezelő rutin, annál nagyobb késleltetéssel tud csak a rendszer a többi megszakításra válaszolni.

Miután e kitételeknek megfelelően megírtuk a kezelő függvényt, be kell regisztrálni azt a kernelbe. Erre a következő függvény szolgál:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
flags, const char *name, void *dev)
```

Ebben meg kell adnunk a használni kívánt megszakítási vonal számát, a kezelő függvényt, az esetleges jelző flageket (általános esetben lehet 0), a megszakítás nevét, illetve egy egyedi, megosztott megszakítások esetén használt azonosítót (nem megosztott megszakításoknál ez lehet NULL).

A megszakítás kezelő sikeres regisztrációja ellenőrizhető a */proc/interrupts* fájlban, ahol a rendszerben használt összes megszakítási vonal fel van sorolva.

A fent említett szigorú megkötések gyakorlatilag lehetetlenné teszik komolyabb feladatok végrehajtását a kezelő függvényben. Ha erre lenne szükség, akkor a következő eszközök állnak rendelkezésre:

- Tasklet (kisfeladat)
- Munkasor
- Kernelszál

Az összes megoldás lényege, hogy a megszakítás kezelő a legszükségesebb teendők elvégzése után értesíti a kernelt, hogy egy adott feladatot el kell végezni, majd a tényleges végrehajtást az ütemezőre bízza. Ezt a módszert BH-mechanizmusnak (Bottom Half) nevezzük, ami a teljes elvégzendő feladatot Top half-ra és Bottom half-ra bontja, ahol a Top half jelenti a tényleges megszakításkezelőt, míg a Bottom half a fent említett három lehetőség egyikét.

A taskletek használatakor a beütemezett műveleteket szoftver megszakítás kontextusban végezzük el, ami hasonló a normál megszakításhoz, csupán nem külső esemény váltja ki őket. A taskletek használatához meg kell írnunk a feladatot ellátó függvényt, majd kérnünk kell annak lefuttatását, ami leghamarabb a hardver megszakítást kezelő lefutása után esedékes. Általánosságban elmondható, hogy kis késleltetéssel kerülnek ütemezésre, azonban mivel megszakítás kontextusban futnak, így rájuk is vonatkozik, hogy bennük nem lehet alvással vagy újraütemezéssel járó műveletet végrehajtani.

Amennyiben a fenti megszorítások nem megengedhetőek, akkor célszerű egy munkasor (workqueue) használata. Ez a kisfeladatokhoz képest abban különbözik, hogy nem megszakítási környezetben, hanem kernelszálban fut, ahol lehetséges az átütemezés, és így megengedett a várakozás. A végrehajtott függvénynek a következő prototípussal kell rendelkeznie:

```
void work_func(struct work_struct *)
```

Ebből egy *work_struct*-ot kell létrehozni a következő makró segítségével:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Itt a name lesz a work_struct neve.

Ezek mellett szükség van egy s*truct work_queue* struktúrára, amit a következő függvénnyel készíthetünk:

```
struct workqueue_struct *create_workqueue(const char *name);
```

Végül a feladat végrehajtását a munkasorba való behelyezésével kérhetjük:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
```

A fenti két megoldáson kívül lehetőség van a feladatok külön kernelszálban való futtatására is. Ezek olyan folyamatok, amiknek nincs felhasználói címtartományuk, azaz csupán a kernel memóriáját használják. Kernelszál létrehozására és elindítására a következő függvény szolgál:

```
struct task_struct* kthread_run(int (func)(void *data), void *data, const
char name[],…);
```

Itt a *func* tartalmazza a folyamat által futtatott függvényt, a *data* a függvénynek átadott paramétert, a *name* pedig a kernelszál nevét, amit printf-szerűen lehet megadni.

4.6 Linux boot folyamata

Egy Linuxot futtató Zynq eszköz boot folyamata kezdetben megegyezik a 2.3 bekezdésben foglaltakkal. Eszerint tehát a BootROM-ban található kód bemásolja az első szintű rendszerbetöltőt (FSBL) a belső memóriába (OCM), majd átadja neki a vezérlést. Az FSBL felkonfigurálja a PL részt, a Vivado-ban megadott paraméterekkel inicializálja a PS-t, végül betölti a másodszintű rendszerbetöltőt (second stage bootloader) vagy a felhasználói programot a DDR memóriába, és lefuttatja azt.

Linux esetén a kernel kép betöltésére nem az FSBL-t használjuk, hanem azt egy második bootloaderre, az U-Boot-ra[20] bízzuk, ami egy nyílt forráskódú, beágyazott rendszerekben használt rendszerbetöltő. Indulása után soros porton keresztül konzolt nyit, így a boot folyamatot közvetlenül vezérelhetjük, de alapértelmezett beállítást is tartalmaz, így külső beavatkozás nélkül is el tudja indítani a rendszert. Az U-Boot a kernelt SD kártyától, NAND / NOR flash-ből, USB-n, SATA-n, vagy akár Etherneten keresztül is képes betölteni, melyekre az FSBL nem volna képes.

Miután az U-Boot betöltötte a kernel képet a kezdeti fájlrendszert tároló ún. initramfs-el, elkezdi lefuttatni azt. A bemásolt kernel kép nem közvetlenül a futtatható kódot, hanem annak többnyire zImage formátumú tömörített változatát tartalmazza. Emiatt a kép elejére egy hardver inicializálást és kitömörítést megvalósító program kerül, ez az, amit az U-Boot futása végén elindít. A kitömörített kernel a memória felső felébe íródik, erre ugrik a végrehajtás a kitömörítés után, és ezzel elindul a kernel boot-ja.

A kernel a futás kezdeti szakaszában a bootloader által memóriába másolt RAM fájlrendszert használja, így nincs szükség a háttértároló elérésére. Ez magában egy teljes értékű kisméretű fájlrendszer, ami alapvetően a perifériák működtetéséhez szükséges modulokat tartalmazza. Ezek közül csak azokat tölti be a rendszer, amikre ténylegesen szükség van, megspórolva a szükségtelen kódok memóriában tárolását. PC esetén a boot folyamat végén ezt a fájlrendszert felváltja a háttértárolón található, felhasználó által ismert fájlrendszer. Beágyazott környezetben azonban ezt a lépést jellemzően kihagyják, és a rendszer végig RAM disk-ből fut.

4.7 Linux rendszer konfigurálása

Ahhoz, hogy egy eszközön Linux operációs rendszert futtathassunk, a következő lépéseket kell végrehajtani:

4.7.1 A kernel lefordítása

A Linux kernel forráskódja a www.kernel.org oldalon bárki által szabadon elérhető és felhasználható. Emellett az eszközgyártók is gyakran elkészítik a Linux kernel egy, az általuk gyártott eszközökre optimalizált változatát. Így járt el a Xilinx is, az általa optimalizált kernel a [21] weboldalon érhető el. Miután beszereztük a kiválasztott verziójú kernel forrását, meg kell tennünk a szükséges beállításokat a környezetnek és a céljainknak megfelelően, amihez amellékelt parancssoros eszközöket használhatjuk. A szükséges beállítások után a fordítás következik a „make” parancs kiadásával. Ehhez, mivel nem a cél rendszeren történik a fordítás, egy megfelelő verziójú kereszt-platform gcc fordítóra van szükségünk. A kereszt-platform fordítók olyan programok, amik segítségével a fordítást végzőtől eltérő rendszerre generálhatunk futtatható kódot. A make a fordítás mellett a kernel tömörítését is elvégzi, az így kapott becsomagolt file-t kell a bootloader számára elérhető háttértárolóra másolni.

4.7.2 Root file system generálása

Az operációs rendszer működéséhez szükség van egy gyökér fájlrendszerre, ami rendszerbeállításokat, működéshez szükséges drivereket, függvénykönyvtárakat (például glibc) és felhasználói programokat tartalmaz. Beágyazott rendszerekben gyakori, hogy a futás során végig a kezdeti, RAM-ban található fájlrendszert használjuk, mivel a költséghatékonyság és fogyasztás szempontok miatt igyekszünk minél kevesebb kiegészítő elemet (SD kártya, Ethernet) felhasználni. Ha adatokat maradandóan

szeretnénk lementeni, akkor a tárolón kívül szükségünk van még annak driverjére is, amit vagy a kezdeti ram fájlrendszer tartalmaz, vagy más forrásból kell futás közben betöltenünk. A rootfs létrehozására több módszer is létezik. Megkísérelhetjük kézzel összeállítani, azonban ez munkaigényes, és sok hibalehetőséget rejt magában. Ehelyett célszerűbb választás vagy egy összeállított fájlrendszer letöltése, vagy egy generáló program használata. Kész fájlrendszert a Xilinx is biztosít az ARM és MicroBlaze eszközeihez, ami szabadon letölthető a cég honlapjáról. A generáló eszközök közül megemlítendő a Buildroot, amivel testre szabhatjuk, hogy milyen programok kerüljenek bele a fájlrendszerünkbe.

4.7.3 Eszközfa generálása

Az eszközfa források .dts és .dtsi fájlokban állnak rendelkezésre, amik ember számára is olvasható formátumúak. Ezekből egy speciális fordítóval a kernel számára értelmezhető bináris fájlt kell generálnunk, amit majd a bootloader által betöltendő állományba ágyazunk be.

Az eszközfa fordítójának kódja a kernelforrás */scripts/dtc* mappájában található meg, és a kernellel együtt lefordul. A kész fordító paraméterezésére tekintsük az alábbi példát:

```
dtc -I dts -O dtb -o system.dtb system-top.dts
```

Ebben a -I kapcsolóval adjuk meg a forrás, a -O-val pedig a cél állomány típusát, -o után következik a kimeneti fájl neve, kapcsoló nélkül pedig a forrásfájlok szerepelnek. A .dtsi fájlokat nem kell felsorolnunk. Amennyiben a futás során overlayeket is fel akarunk használni, akkor szükséges a „-@” kapcsoló megadása.

Az eredményként kapott fájl ezt követően felhasználható a kernel kép összeállításához.

4.7.4 Linux összeállításának automatizálása PetaLinux segítségével

A fentiek alapján látható, hogy egy Linuxos környezet összeállítása összetett feladat. Emiatt a Xilinx cég, hogy megkönnyítse az eszközeire való fejlesztést, egy parancssoros konfiguráló programcsomagot tett elérhetővé. Ez a PetaLinux SDK, ami a következő feladatokat képes végrehajtani:

- Létrehozza a rendszer bootolásához szükséges *BOOT.BIN* fájlt, ami a FSBL és a PL részt konfiguráló .bit fájl mellett tartalmazza az U-Boot bootloadert.
- Lefordítja a Xilinx által optimalizált kernelt a felhasználó beállításaival.
- A Vivado-ból exportált hardver adatok alapján legenerálja és lefordítja az eszközfát.
- Összeállítja a root fájlrendszert, ami a kész periféria meghajtók mellett tartalmazza a felhasználó által megírt modulokat, programokat.
- A kernelből, eszközfából és root fájlrendszerből elkészít egy U-Boot fejléccel ellátott *image.ub* nevű fájlt.
- A rendszer elindításához csak a kapott *BOOT.BIN* és *image.ub* fájlokat kell az SD kártyára másolni.

Az SDK emellett lehetőséget ad az eszköz JTAG-en keresztüli felprogramozására, vagy a rendszer emulátoron való tesztelésére. Erre szolgál a nyílt forráskódú QEMU program, amivel PC-n is ki tudjuk próbálni az összeállított operációs rendszert.

Mivel a szakdolgozatom célja nem a Linux konfigurációs folyamatának vizsgálata, ezért az operációs rendszer összeállítására a PetaLinux SDK-t használtam.

5 Saját alkalmazás készítése

5.1 Célkitűzés

A szakdolgozat célkitűzése egy Zynq-7000-es SoC-on olyan Linux alapú rendszer összeállítása, amihez a PL részben saját készítésű AXI-ra csatlakozó perifériák kapcsolódnak. Az ehhez szükséges lépések:

- PL részbe betölthető, AXI buszra kapcsolódó perifériák megírása.
- SD kártyáról futó Linux operációs rendszer összeállítása.
- A perifériák paramétereit tartalmazó eszközfa overlayek elkészítése és lefordítása.
- Az új periféria betöltését érzékelő mechanizmus létrehozása, ami beilleszti azt az operációs rendszer eszközfájába.
- A perifériákat működtető Linux driverek megírása, amik segítségével azok user space-ből használhatóvá válnak.

5.2 Megvalósítás

5.2.1 PL részbe betölthető perifériák

A Zynq programozható logikai részébe a Xilinx Vivado programmal készítettem el a perifériákat. Mivel ezek alapvetően demonstrációs célokat szolgálnak, így ahol szükséges volt, ki és bemenetek gyanánt a ZedBoard kártyán meglévő LED-eket illetve kapcsolókat használtam. Összesen 4 féle konfigurációt állítottam össze, melyek képesek:

- A kártyán található 8 LED fényerejét PWM-el vezérelni, a fényesség AXI buszon keresztül állítható.
- A kártyára ültetett 8 kapcsoló állását 1 byte-ban visszaadni.
- Időzítő segítségével periodikus megszakítást generálni a PS felé.
- Visszacsatolt shift regiszteres struktúra segítségével pszeudo-véletlen számokat generálni.

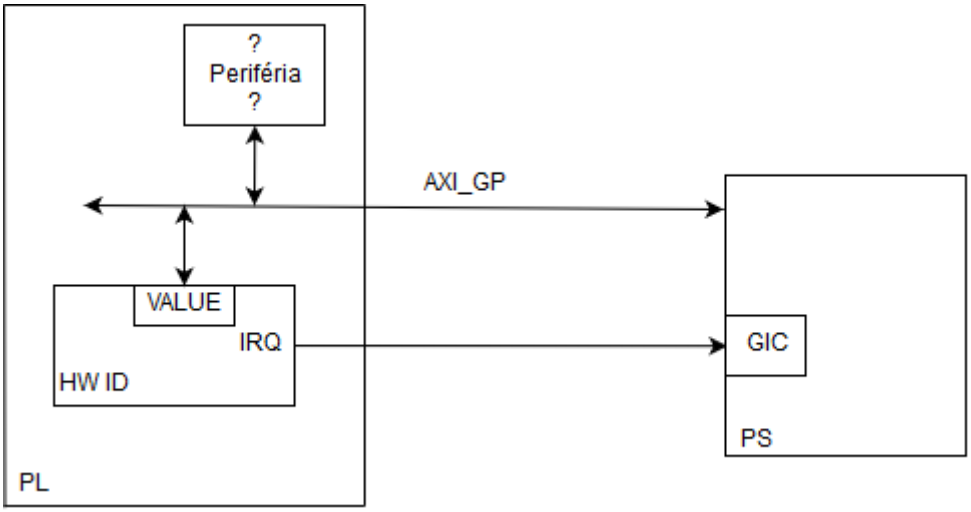
Minden esetben a feladatot egy-egy saját AXI-ra csatlakozó IP blokkal oldottam meg, majd a teljes konfigurációt a Vivado blokk diagram tervezője segítségével állítottam össze.

5.2.1.1 HW ID

Új konfiguráció PL részbe történő betöltése után kérdéses, hogy hogyan azonosítsuk a benne megvalósított perifériákat, hiszen önmagában az AXI busz ezt nem támogatja. A probléma megoldásához szükség van egy kiegészítő PL-beli elemre, ami az AXI-ra csatlakozva megvalósítja a szükséges funkcionalitást. Nevezzük ezt HW ID perifériának. Feladata egyrészt, hogy módot adjon a hardver felismerésre, amit például egy egyedi azonosítóval valósíthatunk meg, továbbá lehetővé kell tennie a rekonfiguráció detektálását. Mivel ez a processzor egy adott pillanatban való értesítését jelenti, ezért kézenfekvő megoldás egy megszakítás kérése, amit a periféria a betöltődés után rögtön jelezni kezd a CPU felé.

Tekintve, hogy az automatikus érzékelést ez a periféria valósítja meg, ezért ezt minden elkészített PL konfigurációban el kell helyeznünk. Ennek során fontos, hogy mindig ugyanazt az előre meghatározott báziscímet és interrupt vonalat kapja, hiszen a megírt kernel modulnak előre tudnia kell, hogy melyik megszakítási vonalat figyelje, illetve hogy milyen címről tudja majd az azonosítót lekérdezni.

Ezen megfontolások alapján a konfigurációk vázlatos blokkdiagramja:



5-1. ábra PL konfigurációk vázlatos felépítése

5.2.1.2 Megvalósítás

A Vivado fejlesztőkörnyezetben beépített varázsló segítségével készíthetünk AXI buszra csatlakozó saját perifériákat. Ehhez meg kell adnunk a blokk nevét, az AXI interface típusát (AXI Lite a jelen esetben), a busz adatszélességét (32 bit), illetve a szükséges regiszterek számát.

Mindezek után a varázsló elkészít egy minta projektet, ami implementálja a teljes AXI Lite protokollt, illetve tartalmazza a megadott számú regisztert. Ezt a kiinduló kódot kell igényeinknek megfelelően módosítanunk. A modul alapvetően több always blokkból áll, melyek közül mind 1-1 AXI jel előállításáért felel. Számunkra csupán 2 fontos: amelyik olvasáskor az adatot a buszra helyezi, és amelyik a bejövő adatot eltárolja, melyeket a kódhoz mellékelt komment alapján azonosíthatunk. A továbbiakban bemutatom az egyes perifériák esetén a mintán elvégzett módosításokat:

Annak érdekében, hogy a LED-eket impulzus szélesség modulációval szabályozó modulban minden egyes LED fényességét külön-külön címre történő írással lehessen beállítani, a varázslóval 4 helyett 8 regisztert generáltattam le. Magát az AXI interfészt a megvalósítás során nem szükséges módosítanunk, hiszen csupán a regiszterekben tárolt értékeket akarjuk felhasználni. A ténylegesen hozzáadott kód mindössze egyetlen számlálóból és 8 darab komparátorból áll. A számláló 0-99999 között számlál, így a PWM frekvenciája a bejövő órajel frekvenciájának 100000-ed részére, azaz esetünkben 1kHz-re adódik. Mivel az emberi szem fúziós frekvenciája ennél jóval kisebb (kb. 50Hz), a villogás nem lesz látható. Az egyes LED-eket meghajtó vonalak logikai értékét a számláló és a fényesség adat komparálásával állíthatjuk elő. A LED-eknek akkor kell világítania, amikor a számláló értéke kisebb a beállítottnál, mivel így érjük el, hogy a regiszter értékének növelésével a fényesség is nőjön. Ezek mellett a LED-eket meghajtó 8 vonalat még ki kell vezetnünk a modulokból, hogy a blokkdiagramon beköthetőek legyenek. Ezt a portlistába való output beszúrásával tehetjük meg.

A kapcsolók állását beolvasó modulnál a fentiekhez képest egyszerűbb dolgunk van. Először itt is létre kell hoznunk egy új bejövő portot a modulban, amire majd a kapcsolókat kötni tudjuk. Ennek input_signal nevet adtam. Ezután pedig mindössze az olvasott adatot előállító always blokkban kell a *slv_reg0*-t *input_signal*-ra átírni. Ennek hatására minden nulladik regiszterre érkező olvasás esetén annak értéke helyett a kapcsolók állását kapjuk vissza. Íráskor az adat ugyan eltárolódik, de azt visszaolvasni nem tudjuk. Mivel a Vivado varázslójában a legkisebb regiszterszám 4, így a fentien kívül

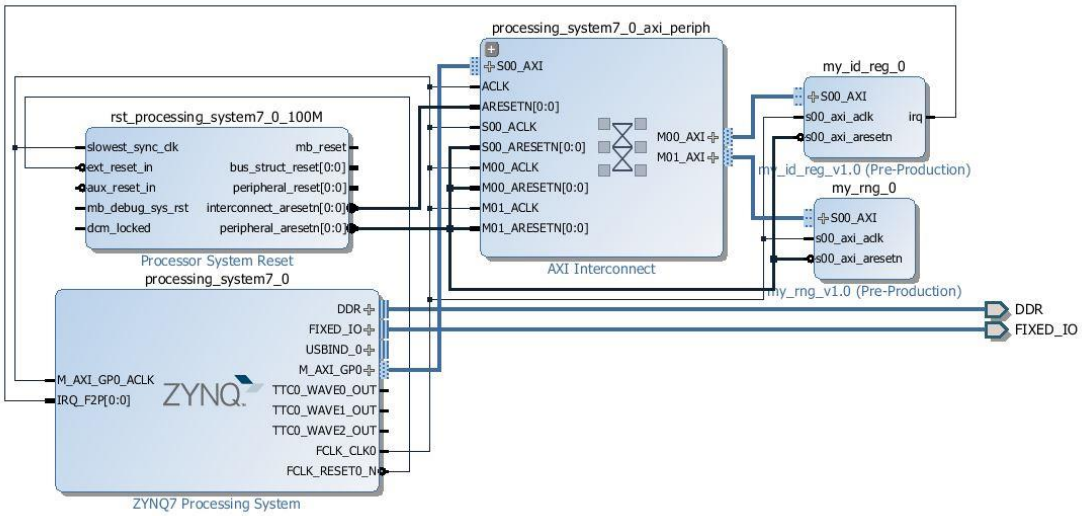
létezik még 3 írható és olvasható regiszter is a rendszerben, amik a periféria működését nem befolyásolják.

A visszacsatolt shift regiszterrel véletlen számokat generáló modul esetén az elvárt viselkedés az, hogy azt tetszőleges értékkel inicializálni tudjuk, majd minden kiolvasás után a generátor következő állapotát kapjuk meg. Legyen a generátor regiszter a periféria első (báziscímű) regisztere. A fenti specifikációból látszik, hogy a regisztert a reset jelen kívül mind olvasás, mind írás esetén szeretnénk módosítani. A Verilog szabályai szerint egy regiszter értékét csak 1 always blokkban szabad írni, azonban a már kész kódba nehezen illeszthető be az olvasás utáni módosítás. Ezért az írást egy saját külön always blokkban oldottam meg, a legenerált kódban pedig kikommenteztem a nem szükséges részletet. A visszacsatolt shift regiszterrel egy 16 bites Fibonacci struktúrát hoztam létre, amiben a 16., 14., 13. és 11. helyi értékből állítódik elő a következő bit. A generátor felső 16 bitjében az eltolás miatt megjelennek a korábbi kimeneti bitek, azonban ezek nem járulnak hozzá a működéshez.

A periodikusan megszakítást adó modulhoz saját IP blokkot nem készítettem, mivel ehhez a rendelkezésre álló AXI Timer teljesen megfelel.

A HW ID periféria megvalósításához a varázsló által legenerált kódhoz egy interrupt generátort és 2 paramétert adtam hozzá. Az első „VALUE” paraméter azt határozza meg, hogy a nulladik, azonosítót tartalmazó regiszter olvasásakor milyen érték jelenjen meg a buszon, a második „INTR_ENABLED” pedig azt, hogy a modul adjon-e interruptot induláskor (arra az esetre, ha nem akarjuk az automatikus kezelést). A megszakítás a reset jel felfutó élére keletkezik (alacsony aktív logika), és egészen addig magas állapotban marad, amíg nem érkezik egy írási művelet a periféria címtartományába.

Az így elkészített IP blokkok segítségével összeállíthatóak a PL részbe betölthető konfigurációk. Példaképpen az 5.2 ábrán látható a véletlenszám-generátor blokkdiagramja.



5-2. ábra Véletlenszám-generátor blokkdiagramja

Az összeállított blokkdiagramokból végül a Vivado segítségével előállíthatjuk a PL rész konfigurációjához szükséges .bit kiterjesztésű fájlokat, továbbá kiexportálhatjuk a hardvert leíró .hdf fájlt.

5.2.2 Linux operációs rendszer összeállítása

A Linux operációs rendszer összeállításához a Xilinx által készített PetaLinux SDK-t használtam fel, ami az ehhez szükséges lépések automatizálására használható parancssoros programokat tartalmaz. A gyártó részletes dokumentációkat tett elérhetővé, amik például leírják az egyes programok parancssori argumentumait [22], vagy példákon keresztül bemutatják a keretrendszer használatát[23]. Mindezeket felhasználva a következő lépéseket hajtottam végre:

Először egy új projektet kell létrehoznunk, ami a saját alkalmazásainkat, moduljainkat, beállításainkat, illetve a lefordított állományokat fogja tartalmazni:

```
$ petalinux-create –type project –template zynq –name my_project
```

Ezt követően meg kell adnunk a célplatformot leíró, Vivadoból exportált hdf kiterjesztésű fájlt:

```
$ cd <hw description dir>
$ petalinux-config --get-hw-description -p <petalinux-projekt-gyökér
könyvtár>
```

Ennek felhasználásával a kernel platformra vonatkozó paramétereit a rendszer automatikusan beállítja, továbbá összeválogatja a használt perifériákhoz szükséges gyári meghajtókat.

A parancsok lefuttatása után egy konfigurációs menü nyílik meg, amiből változtatás nélkül kiléphetünk.

Mivel az eszközfa overlayeket alapértelmezés szerint nem használ a kernel, ezért ezeket manuálisan kell engedélyeznünk:

```
$ cd <project-root>
$ petalinux-config -c kernel
```

A megjelenő menüben keressük meg a *Device Drivers → Device Tree and Open Firmware support → Device Tree overlays* bejegyzést és jelöljük be azt. Ennek köszönhetően fognak az overlay-ek kezeléséhez szükséges függvények befordulni a kernelbe.

A szükséges beállítások megtétele után elindíthatjuk a fordítást, ami számítógéptől függően kb. 5-10 percet is igénybe vehet:

```
$ cd <project-root>
$ petalinux-build
```

Ennek során a PetaLinux az eszközfa lefordítására egy hagyományos fordítót használ, ami nem állítja elő az overlayek kezeléséhez szükséges külön csomópontokat. Emiatt az eszközfa forrásokat a 4.3.1 fejezetben említett patchelt fordítóval külön manuálisan le kel fordítanunk. Ehhez menjünk a *<project-root>/subsystems/linux/configs/device-tree* mappába, ahol a forráskódok találhatóak. Adjuk ki innen a következő parancsot:

```
$ dtc -I dts -O dtb -o system.dtb -@ system-top.dts
```

Az így kapott „*system.dtb*” fájllal írjuk felül a *<project-root>/build/linux/device-tree* mappában található azonos nevű állományt, ez az ugyanis, amit a PetaLinux a kernel image elkészítésekor felhasznál. Ezután futtassuk újra a kernel kép becsomagolását:

```
$ petalinux-build -x package
```

Ez a parancs újragenerálja a *<petalinux-root>/images/image.ub* fájlt, amiben immár az általunk elkészített új bináris eszközfa szerepel.

Az induláshoz a kernel kép mellett szükségünk van még a BootROM által használt *BOOT.BIN* fájlra, ami az FSBL-t, az opcionális kezdeti PL konfigurációt, és az U-Boot-ot tartalmazza. Létrehozása a következőképpen történik:

```
petalinux-package --boot --format BIN --fsbl <petalinux-
root>/images/zynq_fsbl.elf --fpga <.bit fájl helye> --u-boot
```

A kimeneti állomány a *<petalinux-root>/images/* mappában található, másoljuk át innen az *image.ub* fájllal együtt az SD kártyára, és indítsuk el a rendszert, amibe root/root felhasználónév és jelszó párosítással léphetünk be.

5.2.3 Eszközfa overlayek megírása

A PL konfigurációk elkészítése után meg kell írnunk azokat az eszközfa overlayeket, amik az új perifériák paramétereit kernel által értelmezhető formátumban tartalmazzák.

Egy ilyen overlay általános felépítése:

```
/dts-v1/;
/plugin/;
/{
     compatible =  "xlnx,zynq-7000";
     fragment@0{
          target=<&amba_pl>;
          #address-cells = <1>;
          #size-cells = <1>;
          __overlay__{

               name: name@base_addr {
                    /* parameters */
               };

          }; /*overlay*/
     }; /*fragment*/
};
```

Ebben a *name* az aktuális periféria nevére, a *base_addr* pedig a báziscímére cserélendő le. Paraméterként mindenféleképpen meg kell adnunk egy *compatible* mezőt, ami alapján a kernel az eszközünkhöz meghajtót rendelhet, és egy *reg* bejegyzést, ami a periféria által használt memóriatartományt jelöli ki. Megszakítást használó perifériák esetében ezekhez hozzájön még az *interrupts* és az *interrupt-parent* mező. (Az időzítő esetén a PetaLinux által generált leírást használtam, ami további speciális, azonban a megírt meghajtóban ki nem használt adatokat is tartalmaz.)

Az így elkészített forrásfájlokat a patchelt device tree compilerrel a következő utasítással fordíthatjuk le:

```
dtc -I dts -O dtb -o <kimeneti fájl neve> -@ <forrásfájl>
```

5.2.4 Hardver detektáló mechanizmus megírása

A hardver és a Linux rendszer összeállítása után következik a kívánt feladatot ellátó kernel modulok elkészítése. Először olyan modul megírása a cél, ami detektálni képes a PL konfiguráció megváltozását, majd regisztrálja azt a rendszerben. Ehhez a kernel által nyújtott device tree overlay mechanizmust használom ki, azaz futási időben egészítem ki az eszközfát az aktuális környezetnek megfelelően. Ehhez alapvetően 3 bejövő információ szükséges:

- Értesülnünk kell a PL konfiguráció megtörténtéről. Lehetőség szerint ez legyen automatikus, ne igényelje a felhasználó beavatkozását.
- Ki kell tudnunk deríteni, hogy milyen új perifériák kerültek a rendszerbe.
- Szükségünk van a perifériákat tartalmazó overlayekre, amik fragmentek formájában tartalmazzák azok adatait.

Az elkészítendő modul ezek birtokában kiegészíti az eszközfát, majd a driverek betöltését a kernelre hagyja.

Az új konfiguráció betöltődéséről, illetve annak tartalmáról a már említett HW ID egység fogja informálni a modult. Emiatt szükség van egy interrupt kezelő rutinra, ami a megszakítás után kiolvassa az azonosítót, majd betölti az ennek megfelelő overlay-t. Kérdés, hogy honnan tudjuk a megszakítási vonal számát és az ID regiszter címét. Első és legkézenfekvőbb megoldás ezen adatok forráskódba való beépítése lenne, ez azonban rugalmatlan rendszert eredményezne, szerencsésebb volna ezeket az adatokat is beolvasni. Erre alkalmazhatnánk például kernel paramétereket, vagy akár még egy további eszközfa overlayt. A kernel paraméterekkel való megvalósítás hátránya egyrészt, hogy az alapértelmezettől való eltérés esetén minden egyes betöltéskor meg kell adni a helyes értéket, másrészt pedig problémásabb a megszakítás számának meghatározása. A Linux kernel ugyanis nem a GIC megszakításkezelő periféria számozását használja, hanem abból leképezéssel egy Linux megszakítás azonosítót generál. Mivel a Zynq adatlapja alapján csupán a hardveres (GIC számozású) számot ismerjük, el kell végezni a hw_num → linux_num konverziót. Ez az eszközfából olvasó *irq_of_parse_and_map* függvényben már implementálva van, így a második esetben nem kell azt kézzel elvégezni. Mindezek miatt úgy döntöttem, hogy a HW ID perifériához is készítek egy különálló overlay-t, ami tartalmazza a báziscímet és a megszakítás számát. Ha tehát

esetleg a PL konfigurációkban más címre szeretnénk áthelyezni a HW ID egységet, csupán ezt az overlay-t kell újrafordítanunk.

További kérdés, hogy a kernel modul hogyan jusson hozzá az eszközfa overlay-hez. A kódba való befordítás itt sem célszerű, ehelyett jobb lenne a fájlrendszerből beolvasni, amit a kernel firmware alrendszere támogat a request_firmware függvénnyel:

```
int request_firmware(const struct firmware **firmware_p,
const char *name, struct device *device);
```

A paraméterek:

- *firmware_p*: *struct firmware*-re mutató pointer címét várja, ezt állítja be a létrehozott *struct firmware* címére.
- *name*: A keresett file neve.
- *device*: Az az eszköz, amihez a firmware-t keressük. Lehet NULL pointer is.

Ennek meghívása után első lépésben a kernel megpróbálja a */lib/firmware* mappában megkeresni a kért nevű fájlt. Ha megtalálta, akkor annak tartalmát visszaadja egy *firmware* struktúrába ágyazva. Ellenkező esetben, amennyiben engedélyezett, a kernel hotplug eseményt generál, aminek hatására az udev megpróbálja a sysfs-en keresztül átadni a fájlt. A PetaLinuxban alapértelmezés szerint ez az opció nincsen bekapcsolva, és tapasztalatom szerint nem is szükséges.

A fentien kívül létezik még egy *request_firmware_direct()* függvény is, ugyanilyen paraméterezéssel. Ez annyiban különbözik az előzőtől, hogy soha nem próbálja bevonni az udev-et a beolvasásba. Fontos, hogy egy modul inicializáló függvényében lehetőleg ezt a verziót használjuk, ugyanis felhasználásakor az udev addig nem szolgálja ki a kérést, amíg az inicializáló függvény véget nem ér. Emiatt ha a *request_firmware*-t hívnánk meg, és engedélyezett az udev használata, akkor lehet, hogy a modul soha nem indulna el.

Ezen megfontolások figyelembe vételével a modul inicializálása a következőképpen működik:

1. Lekéri a HW ID egységhez tartozó device tree overlay-t. A keresett név: „*axi_id_reg.dtbo*”. Az adatfolyamot átmásolja egy lefoglalt memóriablokkba, majd elengedi a firmware-t.

2. A .dtb tartalmából *device_node* struktúrát készít.
3. Feloldja a *device_node* struktúrában található hivatkozásokat. Emiatt kellett az eszközfát speciális fordítóval lefordítani.
4. Felcsatolja az overlayt.
5. Megkeresi a kiegészített eszközfában a HW ID csomópontot. Ennek segítségével lefoglalja a használt báziscímet, és megszakítási vonalat.
6. Inicializálja a bottom-half-ként használt munkasort, és beregisztrálja a kezelőfüggvényt.
7. Amennyiben a modul betöltésekor a „*startup_check*” paraméternek nem nulla értéket adtunk, akkor megpróbálja azonosítani a már betöltött PL konfigurációt.

Mivel kernel modult írunk, ezért fokozottan kell ügyelnünk a lefoglalt erőforrások felszabadítására, hiszen a felhasználói módban futó programokkal ellentétben itt nem zárja le az operációs rendszer azokat, amikről véletlenül elfeledkeztünk. Ez azonban a program megírásakor sokszor nehézséget jelenthet. Jó példa erre az előbbi inicializáló függvény: a benne meghívott függvények közül szinte bármelyik sikertelen lehet, és ilyenkor az összes addig jól lefutott függvény hatását vissza kell vonnunk. Erre a problémára a Linux kernelben elterjedt megoldás a goto használata. Ilyenkor a függvényünk végén felsoroljuk a normális függvényekhez tartozó lebontó metódusokat az előbbiekhez képest fordított sorrendben, majd goto címkéket szúrunk be a sorok közé. Így hiba esetén csupán a megfelelő sorra kell ugranunk, és az összes szükséges lebontás végrehajtódik.

A bejövő megszakítás lekezelése két részből áll: a rögtön lefutó top-halfból és az utána beütemezett, workqueue-ban végrehajtott bottom-halfból. A felosztás azért szükséges, mivel olyan függvényeket fogunk meghívni, amik kontextus váltást idézhetnek elő (például a *printk*, vagy az *of_overlay_create*).

A top half feladata csupán az, hogy a HW ID címterületére való írással leállítsa a megszakítást, és kérje a bottom half végrehajtását.

A bottom half által elvégzett műveletsor:

1. Amennyiben már töltöttünk be overlay-t akkor azt eltávolítja.

2. Beolvassa a HW ID regiszterből a PL konfigurációt azonosító számot.
3. A firmware rendszeren keresztül bekéri az azonosítónak megfelelő overlay-t.
4. Ha megkapta, akkor az inicializációs részben ismertetett módon jár el: *device_node* struktúrát készít belőle, feloldja a hivatkozásokat, majd beszúrja az eszközfába.

A hibakezelés itt is a goto felhasználásával van kialakítva.

Végül a modul lezáró függvénye eltávolítja a betöltött overlayeket és elengedi a megszerzett erőforrásokat: a memóriaterületeket, a munkasort, a lefoglalt I/O memória tartományt és megszakításvonalat.

Az eddig megírt modul tehát elvégzi a PL-be betöltött konfigurációk azonosítását, és ennek megfelelően egészíti ki az eszközfát. A kód helyes működése könnyedén tesztelhető, mivel a *„/sys/firmware/devicetree/base”* könyvtárban megjelennek az eszközfa elemei. A betöltött új csomópontok ezen belül az „amba_pl” alatt szerepelnek, ahol az összes megadott paraméterük olvasható állományként elérhető.

5.2.5 Perifériákhoz tartozó driverek készítése

5.2.5.1 Felépítés

Mivel az új eszközöket a device tree-n keresztül regisztráltuk be a rendszerbe, azokat a kernel automatikusan a platform buszra helyezi *platform_device*-ként. Emiatt a meghajtót is erre a buszra kell megírnunk, aminek eszközhöz rendelését a kernel automatikusan elvégzi.

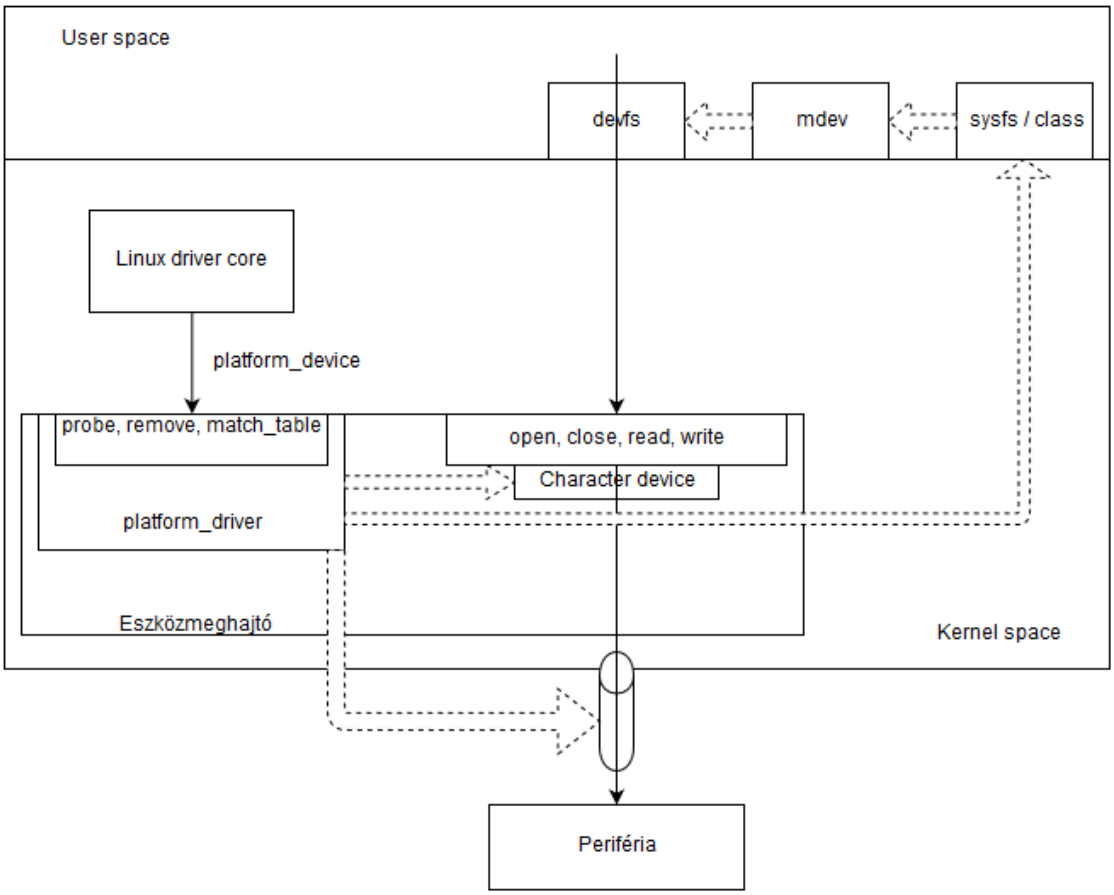
A *platform_driver*-ben egy egyszerű, pár függvényből álló interfészt kell létrehoznunk, amin keresztül a Linux driver core új eszközök megjelenése esetén értesíteni tudja a modulunkat. Így tehát ezekben a függvényekben kell az összes inicializáló és lebontó teendőt elvégeznünk.

A célunk az, hogy a felhasználói tér felé egy egyszerű, */dev* könyvtárban megjelenő karakteres eszközt mutassunk, ami lehetővé teszi a perifériával történő kommunikációt. Ehhez a következő lépéseket kel végrehajtanunk:

1. Meg kell teremtenünk a kapcsolatot a kernel tér és a periféria között az erőforrások megfelelő lefoglalásával.

2. Saját karakteres eszközt kell létrehoznunk, amely fájlműveletekre vezeti vissza a perifériával való kommunikációt.
3. Saját osztályt és eszközt kell regisztrálnunk a sysfs-ben, ami alapján az mdev dinamikusan létre tudja hozni a devfs-beli eszközállományt.

A fenti struktúrát és lépéseket az 5-3. ábra illusztrálja.



5-3. ábra Periféria driverek struktúrája

5.2.5.2 Megvalósítás

A meghajtókat egyetlen különálló modulban valósítottam meg, így a gyakran, ugyan úgy felhasznált kódrészleteket külön függvénybe ki tudtam emelni, megelőzve ezzel a másolást.

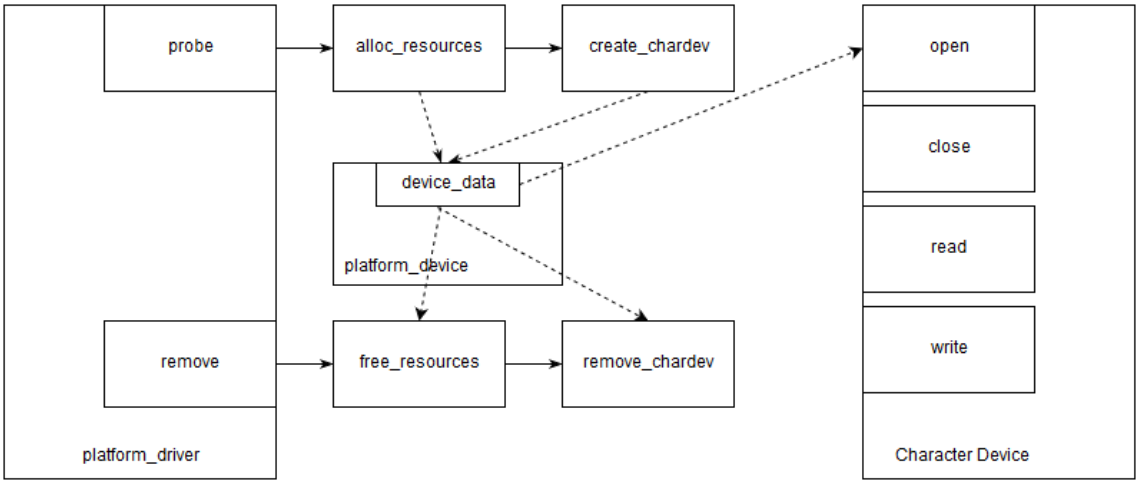
A megírt meghajtóprogramok azonos sémát követnek, felépítésük hasonló. Minden platform driverhez implementálni kell egy „*probe*” és egy „*remove*” függvényt, amik az eszközzel való párosításkor, vagy annak eltávolításakor hívódnak meg. Ezután létre kell hoznunk egy *of_device_id* struktúrát, amiben megadjuk azokat az eszközfában fellelhető paramétereket, amik alapján a kompatibilis eszközök felismerhetők. Ilyen célra

legáltalánosabb a *compatible* mező használata. Végül ezek felhasználásával összeállítható a meghajtót reprezentáló *platform_driver* struktúra, amit a modul inicializáló függvényében a *platform_driver_register*-nek átadva a driver regisztrálható a platform buszra.

A probe függvényekben végrehajtandó feladatok:

1. Le kell foglalnia az eszközhöz tartozó erőforrásokat, jelen esetben ez egy memória címtartomány és esetleg egy megszakítás vonal, továbbá létre kell hoznia a kívánt számú karakteres eszközt.
2. Amennyiben használt, akkor a megszakítás vonalra kezelő függvényt kell regisztrálnia.

Az 1. pontban végrehajtott műveletek minden meghajtó esetén azonosak, így erre külön függvényeket hoztam létre. Ezek az *alloc_resources*, és *create_chardev* metódusok, amik az 5-4. ábra szerint kerülnek meghívásra.



5-4. ábra Platform_driver működése

Az *alloc_resources* prototípusa:

```
static int alloc_resources(struct platform_device *pdev,const char*
name_base, int num, struct file_operations *fops)
```

A függvénynek paraméterként meg kell adni egy az aktuális *platform_device*-ra mutató pointert (pdev), a karakteres eszköz nevét, darabszámát, illetve az ahhoz kapcsolódó fájlműveleteket.

A függvény működése:

1. Létrehozunk egy *device_data* struktúrát, amiben a lefoglalt erőforrásokat tároljuk. Ennek definíciója:

```
struct device_data{
  int irq_num;
  void __iomem *base;
  struct resource res;
  struct chardev_data_type chardev_data;
};
```

Ahol:

   a. *irq_num*: A használt megszakítás vonal számát tartalmazza, annak hiányában értéke 0

   b. *base*: A periféria memóriatartományának kernel térben található kezdőcíme.

   c. *res*: A periféria memóriatartományát leíró struktúra.

   d. *chardev_data*: A létrehozandó karakteres eszközhöz kapcsolódó változók.

2. Az *of_address_to_resource* függvénnyel lekérjük a használt memóriatartományt. Ezt egy resource struktúra tartalmazza, amiben a *start* adattag jelöli a címtartomány kezdetét az *end* pedig a végét.

3. A *request_mem_region* függvénnyel lefoglaljuk a megkapott címtartományt. Ennek sikeres lefutása után a */proc/iomem* fájlban megjelenik a bejegyzésünk.

4. Az *ioremap*-al leképezzük a lefoglalt fizikai címet a kernel térbe. A kezdőcímet lementjük a *device_data* struktúra *base* adattagjába.

5. Az *irq_parse_and_map* függvénnyel lekérjük a használt megszakítási vonal számát. Ez elvégzi el az 5.2.4 részben említett hardver → Linux megszakítási szám konverziót.

6. A kitöltött *device_data* struktúra címét elhelyezzük a *platform_device*-ba befoglalt *device* struktúra szabadon felhasználható *driver_data* mezőjébe a *platform_set_drvdata* függvény segítségével, hogy onnan majd az open függvények elérjék.

7. Végül meghívjuk a *create_chardev* függvényt, ami létrehozza a szükséges számú karakteres eszközt.

A függvényünk visszatérési értéke hibátlan működés esetén 0, ellenkező esetben pedig a hibára utaló azonosítót ad vissza.

Az *alloc_resources* párja a *free_resources*, ami felszabadítja a platform_device-ba lementett *device_data* struktúrában található lefoglalt erőforrásokat.

A *create_chardev* függvény prototípusa:

```
int create_chardev(struct chardev_data_type *chardev_data,const char*
name_base, int num, struct file_operations *fops)
```

A paraméterek értelmezése:

A *chardev_data* az *alloc_resources* által létrehozott *device_data* azonos nevű adattagja, ami egy következő definíciójú struktúrára mutat:

```
struct chardev_data_type{
       struct cdev char_dev;
       struct class *myclass;
       struct device *interface_device[8];
       int Major;
       int minor_num;
};
```

Ebben:

- *char_dev*: A kernelbe való regisztráláshoz szükséges cdev struktúra.
- *myclass*: Saját osztály a karakteres eszközökhöz. Az mdev ennek segítségével hozza létre az eszközállományokat a */dev* könyvtárban.
- *interface_device*: A *device_create* függvénnyel létrehozott eszközök struktúrái. Mivel legfeljebb 8 darabot hozunk létre belőlük (a LED-es perifériánál), így statikusan történik a lefoglalásuk, megspórolva ezzel a *kmalloc*-ot és a hozzá szükséges hibakezelést.
- *Major*: A lefoglalt Major azonosító.
- *minor_num*: A használt minor azonosítók száma. A *device_create*-tel ennyi karakteres eszközt hozunk létre, így az *interface_device* tömbnek ennyi hasznos eleme van.

A függvény feladata, hogy egy karakteres eszköz interfészt hozzon létre a felhasználói tér felé. Ennek során major számot foglal, amihez hozzárendeli az fops-ban megkapott, az eszközfájlon végrehajtott műveletek megvalósításáért felelő függvényeket (alapesetben elég csak az *open*, *close*, *read*, *write* implementálása). Végül saját osztályt készít, amibe *num* darab karakteres eszközt regisztrál, amikhez nevet a *name_base*-ből

egy szám hozzáadásával képez. Például name_base=”led” és num=2 értékek esetén led0 és led1 nevű eszközök jönnek létre. Amennyiben a *num* értéke 0, akkor a név csak a *name_base*-ből áll.

A függvény működése:

1. Leellenőrzi, hogy a *num* értéke az elfogadható 1..8 tartományba esik-e. Ha igen, lementi az értéket a *chardev_data→minor_num* változóba.
2. Major számot kér a kerneltől az *alloc_chrdev_region* függvénnyel, *name_base* néven. Ennek sikeres lefutása után a */proc/devices* fájlban megjelenik a foglalást jelző bejegyzés. A megkapott Major számot a *chardev_data→Major* változóba mentjük.
3. Ezt követi a major szám és a kezelő függvények összekapcsolása a *cdev* struktúrával. Először a *chardev_data→char_dev* adattagot inicializáljuk a megadott *fops* struktúrával, majd kitöltjük a további mezőit is. Végül a karakteres eszköz kernelbe való regisztrációja a *cdev_add* meghívásával történik.
4. Saját osztályt készítünk a *class_create* függvénnyel.
5. Utolsó lépésként a saját osztállyal létrehozzuk a kívánt mennyiségű virtuális eszközt a *device_create* meghívásával. Ez legenerálja azokat a *sysfs*-beli bejegyzéseket, amik alapján az mdev a devfs-ben elő tudja állítani az eszközállományokat.

Amennyiben a létrehozott karakteres eszközökre nincs tovább szükségünk, a *remove_chardev*-vel távolíthatjuk el őket a rendszerből.

Végül térjünk ki a karakteres eszközöket megvalósító fájlműveletekre. Mint már sokszor említésre került, egyszerűbb esetekben elegendő csupán az *open*, *close*, *read* és *write* függvények implementálása, amik a nevükből adódó helyzetekben hívódnak meg. Ezek közül az *open* és *close* külön kezelhetők olyan szempontból, hogy az általuk végrehajtott műveletek függetlenek a konkrét perifériától.

Az *open* feladata, hogy megtegye az állomány használatához szükséges előkészítő lépéseket. Ennek során elsőként megnöveli a modul használat számlálóját a *try_module_get* függvénnyel. A használatszámláló lényege, hogy amíg értéke nagyobb, mint 0, a kernel nem engedi eltávolítani az adott modult, hiszen a rendszer többi része

még használja annak szolgáltatásait. Ezen kívül az *open*-re hárul az is, hogy a read és write függvények számára elérhetővé tegye a használt periféria báziscímét, amit azok egyébként nem tudnának megszerezni (csak globális változókon keresztül, azonban ezek használatát igyekeztem, ahol csak lehet, elkerülni). Az *open* ugyanis paraméterként megkapja az eszközállományhoz tartozó *struct file* és *struct inode* címeit is, míg a *read* és *write* csupán az előbbit, holott az aktuális karakteres eszközhöz tartozó *cdev* struktúra csak az *inode*-ból érhető el. Így az *inode→i_cdev*-ből meghatározzuk a *chardev_data_type* struktúrát, ami tartalmazza a keresett báziscímet, és lementjük azt a *file private_data* mezőjébe. A konverzióra a container_of(address, target_type, member) makró használható, ami visszaadja a *tartget_type* típussal rendelkező burkoló struktúrát egy *member* típusú adattag *address* címéből.

A *close* függvény ehhez képest egyszerűbb, csupán a használatszámlálót kell csökkentenie a *module_put()* függvénnyel.

A read és write prototípusai:

```
ssize_t (*read) (struct file *pfile, char __user *buffer, size_t count,
loff_t *ppos);
ssize_t (*write) (struct file *pfile, const char __user *buffer, size_t
count, loff_t *ppos);
```

Ezekben a *pfile* pointer a fájlhoz tartozó file struktúrára mutat, innen tudjuk az *open* függvényben előzetesen kimentett periféria báziscímet lekérdezni. A *buffer* arra a felhasználói térben lévő memóriaterületre mutat, ahonnan az adatokat beolvashatjuk, vagy ahova azokat ki kell írnunk. Fontos, hogy az általa mutatott memóriatartományt nem használhatjuk közvetlenül, hanem csak a *copy_to_user* és *copy_from_user* függvények segítségével, amik leellenőrzik, hogy a megkapott címek hozzáférhetőek-e és szükség esetén behozzák a kívánt lapo(ka)t a háttértárolóról. Az átvinni kívánt adatmennyiséget a *count* tartalmazza, ennyi bájtnyi adat áll rendelkezésre a bufferben, vagy ennyit írhatunk ki maximálisan. Végül következik a *ppos, ami a fájlon belüli aktuális pozíciót jelzi. A függvények visszatérési értékében arról informáljuk a hívót, hogy hány bájtot olvastunk be, vagy hányat írtunk ki a bufferbe.

Karakteres eszközöket úgy valósítottam meg, hogy a véletlenszám-generátoron kívül mindegyik olvasható legyen a beépített *cat* programmal, amihez az szükséges, hogy a *read* függvény visszatérési értékében összes adat kiírása után 0-val térjünk vissza, mivel különben a végtelenségig folytatódna a kiolvasás. Emiatt a *ppos* paraméter felhasználásától nem tekinthettem el.

Végül következzenek az egyes meghajtók által megvalósított funkciók:

- Switch driver:
  - *read*: Beolvassa a perifériából a kapcsolók állapotát reprezentáló duplaszót, majd szöveges formátumban (’1’/’0’ karakterekkel) kiírja az alsó bájtot a kimeneti bufferbe.
  - *write*: Kapcsolók esetén nincs értelme írásnak, így mindössze egy hibaüzenetet írunk a rendszernaplóba, majd visszatérünk.
- Véletlenszám-generátor:
  - *read*: Amennyiben van elegendő hely a kimeneti bufferben (2 byte), akkor beleírja a véletlenszám-generátor értékét bináris formában.
  - *write*: A bemeneti bufferből maximum 2 byte-ot bemásol, majd ezt 16 bites egész számként értelmezve beállítja a véletlenszám-generátor értékének.
- AXI Timer:
  - *read*: Beolvassa az időzítő periódusát, majd szöveggé alakítva visszaadja.
  - *write*: A megkapott szöveget számmá alakítja, majd ezt álltja be periódusnak. Az álalakítás során maximum az első 10 karaktert értelmezi, mivel 32 biten a legnagyobb ábrázolható szám a 10 jegyű 4.294.967.295. Visszatérési értékben a megadott count értékét adja vissza, hogy a hívó ne próbálkozzon többször egy hosszabb string beírásával. Mivel az időzítő megszakítást kér a rendszertől, így célszerű ezek gyakoriságát maximalizálni, hogy elkerüljük a rendszer túlterhelését. Emiatt ha periódusnak kevesebb, mint 100000-t adunk meg, hibaüzenet kíséretében leállítja az időzítőt. Mivel az időzítő frekvenciája 100Mhz, így a megszakítások közötti minimális időtartam 1ms. A Timer periféria használatáról annak adatlapja nyújt információt. Ebben megtalálható, hogy a legelső regiszter kontroll funkciót lát el, a második pedig a periódust tárolja. A kódban használt, az időzítő megállításához, újratöltéséhez, elindításához használt értékek ez alapján értelmezhetőek.
  - Megszakítás kezelő: Az időzítő által generált megszakítás kezelése itt is két lépésben történik. Kezdetben a top half függvény fut le, melynek az

interrupt jel megszüntetése a feladata. Ehhez a kontroll regiszter interrupt flag helyiértékére 1-es bitet kell írnia úgy, hogy közben a többit nem változtatja meg. Emiatt első lépésben kiolvassuk a teljes regisztert 1-es interrupt flaggel, majd ezt az értéket írjuk vissza. Ezután beütemezi a bootom half függvényt a létrehozott workqueue-ba, ami a megszakítás tényét majd bejegyzi a rendszernaplóba.

- PWM-es Led vezérlő:
  - *read*: Meghatározza a használt állomány minor számát, majd az ennek megfelelő LED-hez tarozó fényesség regisztert olvassa be, aminek értékét szöveggé alakítva kiírja a felhasználói bufferbe.
  - *write*: A szövegként megkapott számot 32 bites előjel nélküli integerré alakítja, majd kiírja a megnyitott állomány minor számának megfelelő LED-hez tartozó regiszterbe.

5.3 Tesztelés

A program magírását követően annak helyességéről minél kiterjedtebb tesztessel bizonyosodhatunk meg. Ennek során a rendszerünket különböző hatásokkal „gerjesztjük”, majd megfigyeljük, hogy az az elvárt módon reagál-e. Esetünkben a tesztelés célszerűen különböző FPGA konfigurációs fájlok betöltését, majd a megjelenő eszközök használatát jelenti. Ez természetesen végezhető kézzel („cat” és „echo” parancsokkal), azonban ez lassú és munkaigényes, emiatt ez a mód elsősorban a fejlesztés során használható a működőképesség gyors ellenőrzésére. Ehelyett célszerű az amúgy kézzel végzett munkát valamilyen felhasználói térben futó program megírásával automatizálni. Ez történhetne a kernel modul fejlesztéséhez hasonlóan C nyelven, azonban tekintve, hogy itt a teljesítmény nem kritikus, felmerül valamilyen interpretált nyelv használata. Ez egyrészt megkíméli a fejlesztőt a programkód lefordításától, másrészt pedig lehetőséget ad a forrás célrendszeren való módosítására valamilyen egyszerű szövegszerkesztő segítségével, mint amilyen például a PetaLinuxxal létrehozott rendszereken alapértelmezetten megtalálható vi. Az általam választott nyelv a Python lett, mivel ez a root fájlrendszerhez egy konfigurációs opció bekapcsolásával egyszerűen hozzáadható.

Összesen két tesztprogramot készítettem, amik a kapcsolókat és LED-eket, továbbá a véletlenszám-generátort működtetik. Az időzítőhöz külön alkalmazás nem készült, mivel a meghajtó a konfiguráció betöltésével egyszerűen ellenőrözhető.

A konfigurációk cseréjére a PetaLinux xdevcfg eszköze használható, ami a */dev/xdevcfg* útvonalon érhető el. A rekonfigurálás megindításához mindössze be kell írnunk a kívánt .bit fájl-t ebbe az állományba. Ez kézzel az

```
cat my_axi_timer.bit > /dev/xdevcfg
```

utasítással tehető meg.

Pythonból vagy direkt megnyitjuk a fájlt és beleírjuk a kívánt konfigurációt, vagy felhasználhatjuk az *os* csomag *os.system* függvényét a fenti utasítással, ami ugyanolyan, mintha a kézi módszert alkalmaztuk volna:

```
os.system(„cat /sd/bit/my_axi_timer.bit > /dev/xdevcfg”)
```

Az első program ciklikusan betölti a kapcsoló perifériát, beolvassa a fizikai kapcsolók állapotát, betölti a LED perifériát, majd kigyújtja az aktív kapcsolókhoz tartozó LED-eket. A rendszernaplóba írt üzeneteken keresztül az első periódus nyomon követhető:

```
root@my_axi_proj:/sd# python thesis.py
Removing character device.
device class 'myrandom': unregistering
class 'myrandom': release.
class_create_release called for myrandom
Device resources are deallocated.
Previous overlay deleted
Firmware found
Inserting the new overlay.
Probing switch driver.
Allocating platform device resources.
Physical address start: 0x43c10000.
Remapped base address: 0xe0996000.
Major number allocation succeeded: 245.
Creating class for udev.
device class 'sw': registering
Creating character devices.
Chardev creation succeeded.
Switch driver loaded.
AXI switch device detected.
Removing character device.
device class 'sw': unregistering
class 'sw': release.
class_create_release called for sw
Device resources are deallocated.
Previous overlay deleted
Firmware found
Inserting the new overlay.
Probing led_pwm driver.
Allocating platform device resources.
Physical address start: 0x43c10000.
Remapped base address: 0xe099a000.
Major number allocation succeeded: 245.
Creating class for udev.
device class 'led_pwm': registering
Creating character devices.
Chardev creation succeeded.
PWM led driver loaded.
AXI pwm device detected.
```

5-5. ábra LED és kapcsoló perifériák tesztelése

A második kizárólag a véletlenszám-generátort használja: bináris formában kiírja a visszacsatolt shiftregiszter értékét:

```
root@my_axi_proj:/sd# python thesis_random_tester.py
New random number generator seed: 29478.
Random number generator inicialized.
0110010011001110
0011001001100111
1001100100110011
0100110010011001
0010011001001100
0001001100100110
0000100110010011
1000010011001001
0100001001100100
0010000100110010
1001000010011001
0100100001001100
0010010000100110
0001001000010011
1000100100001001
0100010010000100
1010001001000010
0101000100100001
0010100010010000
0001010001001000
1000101000100100
0100010100010010
0010001010001001
0001000101000100
1000100010100010
1100010001010001
1110001000101000
0111000100010100
1011100010001010
1101110001000101
```

5-6. ábra Véletlenszám-generátor tesztelése

Elmondható, hogy a teszt segítségével sikerült pár hibára fényt deríteni, amik kijavítása után a kernelmodulok stabilan működtek.

6 Értékelés

Összességében megállapítható, hogy az elkészített rendszer teljesíti a feladatkiírásban szereplő célokat, tartalmaz különböző típusú, FPGA-ban implementált, AXI-ra csatlakozó perifériákat, ezekhez elkészültek a működtető meghajtóprogramok, amik egyszerű karakteres eszközként teszik őket elérhetővé a felhasználó számára. Végül sikerült megoldani a hardver betöltés utáni felismerését, amihez a megfelelő beállításokat követően (overlay fájlok */lib/firmware*-be másolása) nem szükséges a felhasználó beavatkozása. Jól látszik ez a tesztelésre készített programokban is: ezek saját maguk tudták a különböző perifériákat kicserélni és használni.

Az elkészült rendszer alapvetően tesztelési, bemutatási célokat szolgál, így nincsen felkészítve a különböző ritkán előforduló, vagy párhuzamos használatból eredő problémákra. Egyszerű környezetben azonban a működés megfelel a kívánalmaknak.

6.1 További fejlesztési lehetőségek

Bár az elkészített kernelmodulok a kitűzött célokat teljesítik, további funkciók hozzáadásával kényelmesebben és hatékonyabban lenne használható. Erre néhány példa:

- Az elkészített FPGA konfiguráció blokk RAM cellákban tartalmazhatná a benne megvalósított perifériákat leíró lefordított eszközfa állományt, így nem kellene azt külön bemásolni a /lib/firmware mappába.
- A rendszert biztonságosabbá lehetne tenni kölcsönös kizárást biztosító eszközökkel (szemaforok, mutexek).
- Az FPGA-rész lehetővé teszi a parciális rekonfigurációt is, aminek során annak csupán egy részét változtatjuk meg. Ez a funkció hasznos lehet például olyankor, ha a terv több olyan jól elkülönülő részt tartalmaz, amik közül egyszerre csak egyet használunk, hiszen ilyenkor helyet spórolhatunk azzal, ha csak az éppen szükséges példányt töltjük be. Érdemes lenne megvizsgálni, hogy az eszközfelismerő modullal hogyan lehetne megoldani a parciális betöltés utáni hardver detektálást.

7 Függelék

7.1 Irodalomjegyzék

[1] *Xilinx – Zynq:*
http://en.wikipedia.org/wiki/Xilinx#Zynq

[2] *ZedBoard*
http://zedboard.org/product/zedboard

[3] *Xilinx Zynq-7000 Architecture*
http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf

[4] *Zynq-7000 All Programmable SoC – Technical Reference Manual*
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[5] *Wikipedia – Advanced Microcontroller Bus Architecture*
https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture

[6] *Xilinx – AXI Reference Guide*
http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf

[7] *AXI4-Lite waveforms*
http://www.slideshare.net/marsee101/7-axi4-axi4ip2-44567631

[8] *AXI-Lite*
arco.esi.uclm.es/public/doc/tutoriales/Xilinx/old_training/Embbeded%20Designs/3-axi-configurations.ppt

[9] *Wikipedia – Linux*
https://hu.wikipedia.org/wiki/Linux

[10] *Anatomy of the Linux kernel*
https://www.ibm.com/developerworks/library/l-linux-kernel/

[11] *Asztalos Márk, Bányász Gábor, Levendovszky Tihamér : Linux programozás, második átdolgozott kiadás, ISBN 978-963-9863-29-3, Szak kiadó, 2012*

[12] *Linux Cross Reference*
http://lxr.free-electrons.com/

[13] *Wikipedia – System Call*
https://en.wikipedia.org/wiki/System_call

[14] eLinux - Device Tree
http://elinux.org/Device_Tree_Usage

[15] Transactional Device Tree & Overlays
http://events.linuxfoundation.org/sites/events/files/slides/dynamic-dt-elce14.pdf

[16] Device tree overlays
https://lwn.net/Articles/616859/

[17] Device trees, overlays and parameters
https://www.raspberrypi.org/documentation/configuration/device-tree.md

[18] Device Tree Compiler - Patched version, install script
https://raw.github.com/RobertCNelson/tools/master/pkgs/dtc.sh

[19] *Linux Device Drivers – Third Edition*
https://lwn.net/Kernel/LDD3/

[20] *Zynq-7000 All Programmable SoC Software Developers Guide*
http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf

[21] *The official Linux kernel from Xilinx*
https://github.com/Xilinx/linux-xlnx

[22] *PetaLinux Tools Documentation: PetaLinux Command Line Reference*
https://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_4/ug1157-petalinux-tools-command-line-guide.pdf

[23] *PetaLinux Tools Documentation: Workflow Tutorial*
http://www.xilinx.com/support/documentation/sw_manuals/2014_4/ug1156-petalinux-tools-workflow-tutorial.pdf

7.2 Használati útmutató a CD melléklethez

A szakdolgozathoz mellékelt CD-n megtalálhatóak az elkészített rendszer forrásfájljai, illetve a futtatáshoz szükséges lefordított állományok. Az egyes mappák tartalma:

- peripherals: Ez tartalmazza az elkészített Vivado projekteket, továbbá a saját perifériákat tartalmazó ip_repo-t.
- overlays:
    - az egyes perifériák adatait tartalmazó eszközfa overlay források
    - fordításhoz használható bash szkript.
- device_attacher:
    - a hardver megváltozását érzékelő modul forráskódja és a lefordított, betölthető kernelmodul.
- device_drivers:
    - a perifériák meghajtóit tartalmazó modul forrása és annak lefordított változata.
- SD_image:
    - /bit: az egyes perifériákat tartalmazó FPGA konfigurációs fájlok.
    - /ov: lefordított eszközfa overlayek
    - device_attacker.ko: a hardver megváltozását érzékelő modul lefordított állománya.
    - device_drivers.ko: a perifériák meghajtóit tartalmazó modul lefordított változata.
    - init_scripts.sh: Inicializáló szkript, lefuttatásakor bemásolja a lefordított overlayeket a /lib/firmware mappába, majd betölti a megírt két modult
    - thesis.py, thesis_random_tester.py: tesztelő Python kódok.

A rendszer kipróbálásához másoljuk az SD_image mappa tartalmát az SD kártyára, majd indítsuk el az operációs rendszert, amibe root/root felhasználói névvel és jelszóval tudunk belépni. Ezek után adjuk ki a

```
# source /bin/dw.sh”
```

parancsot, ami felcsatolja az SD kártyát a */sd* útvonalra, és végrehajtja az abban található init_script.sh-t. A tesztek lefuttatása a

```
# python /sd/thesis.py
```

, és a

```
# python /sd/thesis_random_tester.py
```

parancsokkal történhet.