Universidad del País Vasco
Euskal Herriko Unibertsitatea

Informatika Fakultatea

Informatika Ingeniaritzako Gradua

▪ Gradu Amaierako Proiektua ▪

Konputagailuen Ingeniaritza

KONPUTAGAILUEN EGITURA IKASGAIKO PRAKTIKAK RASPBERRY PI MAKINARA EGOKITZEA

Jon Viedma Castillo

2017 - ekaina

informatika fakultatea
facultad de informática

Laburpena

Gradu amaierako proiektuaren helburu nagusia **Raspberry Pi Fundazioak** orain dela bost urte merkaturatutako Raspberry Pi txartelaren bitartez, konputagailuen arkitektura saileko *Konputagailuen Egitura* irakasgaian moldaketa bat ezartzea da. Raspberry Pi kreditu txartelen tamaina duen eta zirkuitu-plaka bakarreko konputagailu pertsonalen serie bat da, gaur egungo elektronikako proiektu askotan erabiltzen dena.

Orain arte, *Konputagailuen Egitura* irakasgaia bi zatitan banatua izan da: Lehenengo atalean ikasleek behe mailako programazioa ikasten dute (ARM mihiztadura lengoaia). Bigarren zatian aldiz, Sarrera / Irteerako sinkronizazio kontzeptuak (etenak, inkestak) ikasten dira.

Proiektu honetan Raspberry Pi txartela orain arte erabilitako Nintendo etxeko DS kontsola ordezkatzeko gai den ala ez aztertuko da, irakasgaian irakasten diren kontzeptu teorikoak praktikara eramateko. Horretarako, Raspberry Pi-ak dituen periferiko zein memoria egituran sakonduko da, behe mailako programazioaren ikuspuntu batetik.

Gaien Aurkibidea

Irudi eta Taulen zerrenda

IRUDIAK

TAULAK

KODEAK

1

Proiektuaren helburuen dokumentua

Kapitulu honetan garatu den proiektuaren azalpena emango da. Hasieran proiektuaren aurkezpen bat egingo da, proiektuaren helburuak eta hauek lortzeko erabiliko diren baliabideak azalduz. Behin helburuak definituta daudela, hauek lortzeko landu beharreko atazak eta proiektua bukatzeko beharrezko denboraren estimazioa, egutegia eta irismena definituko dira. Kapituluarekin amaitzeko, proiektuaren jarraipen eta kontrolaren laburpen bat egingo da.

1.1. Sarrera

Helburuen dokumentu honen barruan proiektuaren aurredefinizio bat egingo da ondorengoak identifikatuz:

- Zein da proiektuaren helburua.
- Zeintzuk dira proiektu hau jorratzeko ditudan baliabideak eta atazak.
- Zein izango da hartuko dudan lan metodologia.
- Lanaren planifikazioa: orduen estimazioa, arrisku posibleak…

Proiektu guztietan gertatzen den bezala, denboran zehar desbiderapenak egoteko arriskuak daude, bai onerako, bai txarrerako. Desbiderapen guztien nondik-norakoak identifikatzea, hauek ondo dokumentatzea eta kontrol egoki bat eramatea izango dira nire betebeharrak lana ondo bideratzeko.

1.2. Helburua

Proiektu honen helburu nagusia *Konputagailuen Egitura* irakasgaian orain arte erabilia izan den Nintendo DS kontsola ordeztu dezakeen gailu bat aurkitzea da. Ordezketa posible den ala ez jakiteko, Raspberry Pi txartelarekin probak egingo ditut.

Raspberry Pi-a gaur egun elektronikako proiektu askotan erabilia den txartela da. Kostu baxuko txartel honen aurre-analisi baten ondoren, gure helburuak betetzeko lagungarria izan daitekeela uste da, helburuak ondorengoak izanik:

- ARM mihiztaduran idatzitako programak Raspberry Pi-ra moldatzea.
- Sarrera / Irteerako sinkronizazio moduak landu ahal izatea Raspberry Pi-a erabiliz.
- Sarrera / Irteerako periferikoen erabilpena ikasi ahal izatea Raspberry Pi-a erabiliz.

Helburuak betetzeko, proiektua bi zatitan banatuko da. Lehenengo atalean (Memoria honetako 3. kapituluan) ARM mihiztadura lengoaian idatzitako programak nola moldatu daitezkeen ikusiko da, programazio ingurune baten laguntzaz. Bigarren atalean Sarrera / Irteerako kontzeptuak landuko dira modu praktiko batean.

1.3. Baliabideak

Lehen aipatu bezala, proiektua aurrera eramateko beharko diren baliabideak aurredefinitzea ezinbestekoa da, nahiz eta denboran zehar aldatzeko beharra izan dezakedan.

Hardwarearekin hasiz, Raspberry Pi txartela erabiltzea ezinbestekoa zait, honen inguruan egindako proiektua baita. Txartelarekin konektatzeko gailu hauek erabiliko dira: Pultsadoreak, *display*-ak, Ledak... Osagai hauen ezaugarriak proiektuan zehar azalduko dira.

Software aldetik, Raspberry Pi txartelaren memoria eta erregistroak ikustea beharrezkoa da. Horretarako, programazio ingurune baten laguntza beharko da, atal hauek maneiatzen lagunduko duena. Honen adibidea *Codeblocks* ingurunea da, zeinek ARM mihiztadura lengoaian programatzeko aukera ematen duen. ARM motakoa ez den makina batetik kodea konpilatu nahi bada, Codeblocks inguruneaz aparte beharrezkoa zait zehar-konpiladore bat erabiltzea. Proiektuko lehenengo atalean ARM makina batean lan egingo denez, zehar konpiladorearen beharrik ez dago. Bigarren atalean aldiz, **GNU proiektuko** ARM gcc konpiladorearen 4.7 bertsioa erabili beharko da *Bare Metal* programazioa betetzeko.

1.4. Lan metodologia

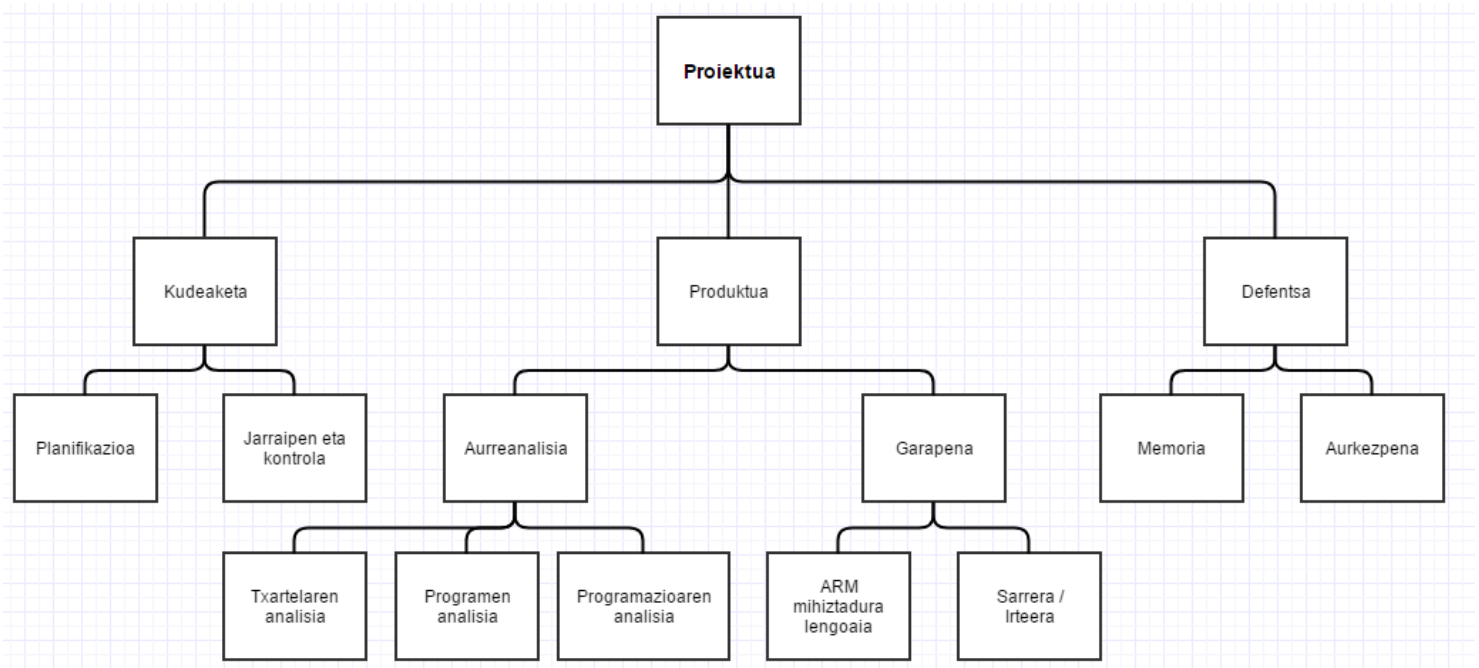
Lan metodologia hainbat fasetan banatzen da proiektuaren helburuak betetzeko.

Hasieran, ezer egin baino lehen, plangintza orokor bat prestatu da, lana egiten hasteko oinarri bezala. Plangintza honek bi helburu nagusi ditu, alde batetik lana egiten hasteko oinarri bat izatea, eta bestetik, proiektua antolatzeko lehenengo pausuak izatea.

Ondoren produktu zein memoriaren jarraipen eta kontrol fasea etorri da. Produktua garatzen den bitartean, honen kontrolaren dokumentazio bat egitea ezinbestekoa baita. Honen ondorioz, aurredefinitutako helburutik gertu nagoen ala ez jakin izan dut uneoro. Jarraipen eta kontroleko fase honetan beharrezko segurtasun kopiak egin ditut, lanaren bideragarritasuna bermatze aldera. Proiektuan zehar sartutako orduen kontrola izateko Google Form plataformaz lagundu naiz, desbiderapen konkretuak izateko.

Bukatzeko, memoriaren errepasoa egin da, proiektuaren defentsako atal bezala, eta honen inguruko aurkezpen bat prestatu da.

1.5. Atazen definizioa



1.1.Irudia: LDE-a (Lan deskonposaketaren egitura)

1.1 Irudian ikus dezakegun moduan, hauek dira jorratu beharreko atazak:

- Kudeaketa:
    - Planifikazioa: Proiektua aurrera eramateko ezarriko ditudan oinarri-puntuak.
    - Jarraipena eta kontrola: Proiektuaren nondik norakoaren berri jakiteko, planifikazioarekin egondako desbiderapenak ikusteko eta proiektuaren eguneroko kontrola izateko ataza.

- Produktua
    - Aurre-analisia: Produktua lantzen hasi baino lehen, zein tresna erabil daitezkeen jakiteko informazio bilketa hartzen duen ataza.
        - Txartelaren analisia: Zein gailurekin lan egingo den jakin. Beste hitzekin esanda, txartelaren datu orrialdea ikustea eta ikastea.
        - Programen analisia: Erabiliko den softwaren aukeraketa eta aukeratutakoen ulermena.
        - Programazioaren analisia: helburuak betetzeko beharrezko programazio metodologia ezartzea, adibidez liburutegirik gabe programatzea.

    - Garapena: Proiektuaren garapena bi zatitan banatzen da, *Konputagailuen Egitura* irakasgaia bezala. Atal bakoitzean software zein hardware hainbat mota beharko dira, memorian zehar zehaztuak izango direnak.
        - ARM mihiztadura lengoaia: Irakasgaiaren lehenengo atala. Memoria honetan kapitulu berezi bat eskainiko diot (3. kapitulua).
        - Sarrera / Irteera: Irakasgaiaren bigarren atala. Memoria honetan kapitulu berezi bat eskainiko diot (4. kapitulua).

- Proiektuaren defentsa: Proiektua amaitu ostean egin beharreko atazak.
    - Memoria: Proiektuan garatu den produktuaren dokumentazioa.
    - Aurkezpena: Proiektuan garatutako ideien azalpena.

1.6. Planifikazioa

1.6.1. Estimazioa

Proiektua aurrera eramateko ataza bakoitzari gutxienez eskaini beharreko denboraren estimazioa jasotzen da 1.1. Taulan:

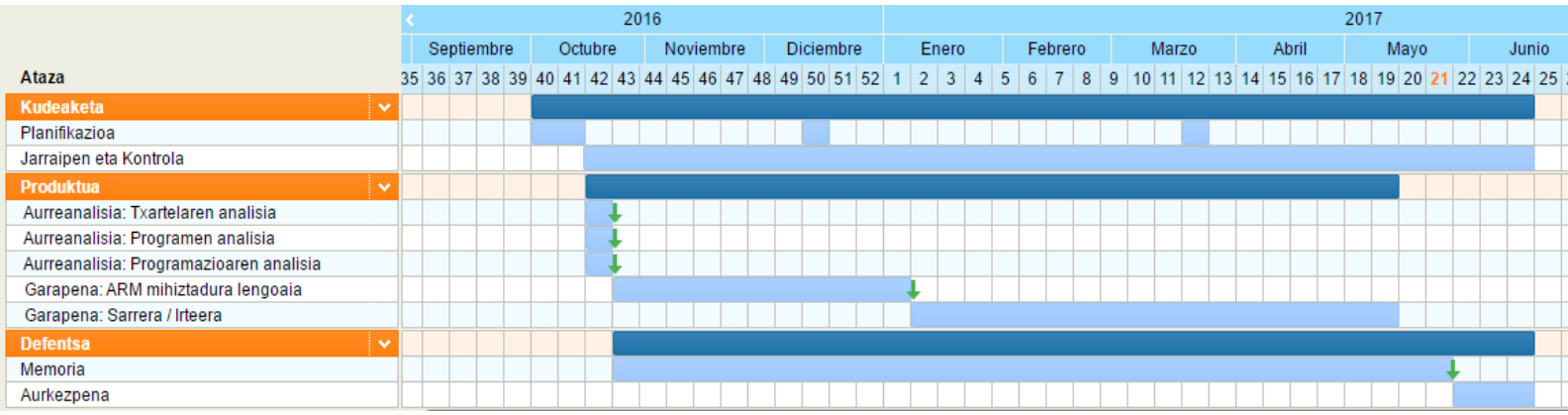
Ataza	estimatutako orduak
Kudeaketa	30
Planifikazioa	3
Jarraipen eta kontrola	27
Produktua	165
Aurreanalisia: Txartelaren analisia	5
Aurreanalisia: Programen analisia	10
Aurreanalisia: Programazioaren analisia	20
Garapena: ARM mihiztadura lengoaia	50
Garapena: Sarrera / Irteera	80
Defentsa	120
Memoria	110
Aurkezpena	10
Guztira	315

1.1. Taula: Atazen orduen estimazioa

1.6.2. Gantt diagrama

1.2 Irudiko diagraman atazak denboran zehar nola banatuko ditudan ikusten da:

- Egutegiko periodo jakin bat hartzen duten laukizuzenak (urdin kolorekoak) ataza bakoitza burutzeko estimatu diren denbora tarteak dira.
- Behera begira dagoen gezia (berde kolorekoak) atazen arteko dependentziak dira, hau da, ataza bat hasteko beste ataza bat amaituta egon behar denaren seinale.



1.2.Irudia: Gantt diagrama.

1.6.3. Arriskuak

Proiektuan zehar aurreikusten diren arriskuak ondorengoak dira:

- Datuen galera: Probabilitate oso txikia aurreikusten da, beharrezko segurtasun kopiak eraikiko direlako, ondorengoak izango direnak:
    - Kode osoaren biltegiratze lokala bi sistematan.
    - Kode osoaren hodeiko biltegiratzea *Google Drive* plataforman.

- Helburuak betearazten lagunduko didaten tresnak ez funtzionatzea: Probabilitate ertain batekin gerta daitezke, erabiliko den softwarea zein hardwarea niretzat berriak direlako. Arazo hauek gertatzerakoan:
    - Arazoa hardwarean badago, beste txartel batekin probatu, proiektu honetan bi txartel erabiltzeko aukera ikusten baita, lehenengoa, Raspberry Pi B modeloa, unibertsitateak utzita eta bigarrena, Raspberry 3 B+ modeloa, ikasleak erositakoa.
    - Arazoa programa batean badago, egin beharreko lehenengo gauza arazo hau konpontzen saiatzea izango da. Arazoari aurre ezin bazaio egin, beste antzeko programak aztertuko dira.
    - Arazoa kodean badago, kodeak zergatik ez duen funtzionatzen aztertu (aurreko bi arrazoiengatik izan daiteke, edo beste edozerengatik...) eta soluzio bat bilatu.

Hala eta guztiz ere eta arriskuak aztertuta, proiektua entregatzeko epea zabala denez, hau bideragarria ez izateko probabilitatea oso txikia dela aurreikusten da.

1.6.4. Irismena

Atal honetan proiektuaren oinarrizko helburua definituko dut, hau da, behin proiektua amaituta dagoenean zein produktu aurkeztu nahiko dudan definitzea. Helburua bi fasetan banatu dudanez, atal honetan bi produktu nagusi definituko dira.

Hasteko, produktuaren lehenengo atala ARM mihiztadura lengoaia irakasteko software baten funtzionamendua ikastean datza. Programaren sekuentzia aurrera doan heinean, Raspberry Pi txartelak dituen memoria helbideak eta erregistroak nola atzitzen diren ikusiko da. Atal honetan software hori erabiltzeko aukerak aztertu beharko dira.

Jarraitzeko, produktuaren bigarren atalean sinkronizazio moduak Raspberry txartelean nola erabili aztertu beharko da, praktiken bitartez hauek programatzeko. Atal honetan Raspberry Pi txartelak eskaintzen dizkidan periferikoak aztertzeko aukera izango dut, *timerra* eta eten kudeatzailea esaterako.

Beraz bi helburu hauek betetzen ditudan arte proiektua ez da amaitutzat emango. Aldiz, helburuak bete baditut eta oraindik proiektua hobetzeko aukera badago, hau hobetzera joko dut, betiere aurreikusitako denboraren barruan banago.

1.7. Jarraipen eta kontrola

1.7.1. Desbiderapenak

Proiektuan aurreikusitako eta benetan sartutako orduen arteko konparazioa jasotzen da 1.2. Taulan:

Ataza	Aurreikusitako orduak	Sartutako orduak
Kudeaketa	30	35
Planifikazioa	3	5
Jarraipen eta kontrola	27	30
Produktua	165	187
Aurreanalisia: Txartelaren analisia	5	2
Aurreanalisia: Programen analisia	10	6
Aurreanalisia: Programazioaren analisia	20	5
Garapena: ARM mihiztadura lengoaia	50	70
Garapena: Sarrera / Irteera	80	104
Defentsa	120	114
Memoria	110	104
Aurkezpena	10	10
Guztira	315	336

1.2. Taula: Atazen orduen desbiderapenak.

1.3. Taulan azalduko dira proiektuan zehar edukitako desbiderapenen arrazoiak. Kontuan izanda aurkezpena oraindik ez dela egin, aurkezpenaren denbora estimatutakoaren berdina jarri da.

Ataza	Estimatutako orduak	Arrazoia
Kudeaketa	+5	
Planifikazioa	+2	Proiektuan zehar irismena aldatzeko beharra izan da, batez ere sarrera / irteera ataleko produktua definitzeko.
Jarraipen eta kontrola	+3	Planifikazioa behin aldatuta eta proiektuan zehar hainbat aldaketa aplikatuta, jarraipena eta kontrol luzeago bat behar izan da.
Produktua	+22	
Aurre-analisia: Txartelaren analisia	-3	Analisian espero izandakoa baino denbora gutxiago behar izan da nahiko informazio dagoelako gai honen inguruan.
Aurre-analisia: Programen analisia	-4	Analisian espero izandakoa baino denbora gutxiago behar izan da nahiko informazio dagoelako gai honen inguruan.
Aurre-analisia: Programazioaren analisia	-15	Analisian espero izandakoa baino denbora gutxiago behar izan da nahiko informazio dagoelako gai honen inguruan.
Garapena: ARM mihiztadura lengoaia	+20	Produktua garatzeko informazio eskasa.
Garapena: Sarrera / Irteera	+24	Produktua garatzeko informazio eskasa.
Defentsa	-6	
Memoria	-6	Aurreikusitakoa baina gutxiago izan da proiektua aurrera joan den bitartean dokumentatu dudalako.
Aurkezpena	0	-
Guztira	315	

1.3. Taula: Atazen orduen desbiderapenaren arrazoiak.

1.7.2. Egondako intzidentziak

Atal honetan proiektuan zehar aplikatu behar izan ditudan aldaketa nabarmenak dokumentatuko ditut:

- Kudeaketa
  - Planifikazioa
    - 2016/11/4: Behin ikusita aurreko urteetan erabilitako laborategiak zuzenean aplikatu ezin direla, irismena aldatzen da laborategi hauek moldatzeko.
    - 2017/4/10: *Bare metal* programazioa ikasten hasi.

- Produktua
  - Garapena: ARM mihiztadura Lengoaia:
    - 2016/12/4: Behin ikusita aurreko urteetan erabilitako laborategiak zuzenean aplikatu ezin direla, hauek erabili ahal izateko moldaketa fasea hasten da. Fase hau Abenduan amaitzen da emaitza positiboekin.
  - Garapena: Sarrera / Irteera:
    - 2017/1/23: Ataza honetako lehenengo bilera batean egon eta gero, helburua betetzeko liburutegien laguntza ezin dela erabili erabakitzen da, beraz maila baxuan lan egin beharko dut. Fase honetan Raspbian sistema eragilearekin lanean jarraitzen da.
    - 2017/4/10: Raspbian sistema eragilea periferiko batzuentzat erabili ezin dudala ikusita, *bare metal* programazioa ikasten hasi.

- Defentsa
  - Memoria: aurreko intzidentziak direla eta, uneoro aldaketak aplikatzen egon behar izan naiz memorian.

2

Nintendo DS eta Raspberry Pi-a

Kapitulu honetan orain arte erabilitako nintendo DS-aren eta Raspberry Pi txartelaren arteko konparaketa bat azaltzen da, irakasgaiaren moldaketarako azkeneko hau baliagarria den ala ez ikusteko.

2.1. Raspberry Pi txartelaren prozesadorea

Raspberry Pi sistemaren bihotzean aurkitzen den prozesadorea Broadcom etxeko BCM28XX modeloa da. Prozesadore honek ARM CPU-a erabiltzen du, 32 biteko RISC (Reduced Instruction Set Computer) arkitekturan oinarritzen dena. Raspberry Pi-ko azken modeloa 64 biteko arkitektura erabiltzen hasi da. Hala eta guztiz ere, arkitektura hau *bare metal* proiektu sinpleek erabiltzen dute bakarrik, oraindik ez delako 64 biteko Raspbian sistema edo antzekorik merkaturatu.

2.1. Taulan Raspberry Pi hainbat modeloren ezaugarri batzuk ikusiko dira:

	Raspberry Pi 1	**Raspberry Pi 2**	**Raspberry Pi 3**
Prozesadorea	Broadcom BCM2835	Broadcom BCM2836	Broadcom BCM2837
CPU	ARM11	ARM cortex A-7	ARM cortex A-53
Arkitektua	ARMv6 (32 bit)	ARMv7 (32 bit)	ARMv8 (32/64 bit)

2.1.Taula: Raspberry Pi hainbat modeloen ezaugarriak

Hauek dira gure txartelak erabiltzen duen arkitekturaren hainbat ezaugarri:

- Aginduak: 32 bit.
- Helbideratze-unitatea : Byte-a.
- Datuak: 8 (byte), 16 (half-word) edo 32 (word) bitekoak izan daitezke.
- Memoria helbideak: 32 bitekoak dira, beraz, guztira 2^{32} helbide izango dira [0..2^{32}-1] helbideen artean.

- 16 erregistro ditu r0-tik r15-era:
    - r0 - r12 bitartekoak erregistro orokorrak dira.
    - r13-ak *Stack Pointer* erregistroa adierazten du. Pilako tontorreko helbidea gordetzen du.
    - r14-ak *Link Register* erregistroa adierazten du. Itzulera helbidea gordetzen du azpirutina bat exekutatzen denean.
    - r15-ek exekutatu behar den hurrengo aginduaren helbidea gordetzen du, *program counter* (*PC*) izenekoa.
    - *CPSR* erregistroa: programaren uneko egoera gordetzen duen erregistroa, baldintzak adierazten dituzten 4 bitekin (4 flag: Negatibo, Zero, Carry eta oVerflow) eta prozesadorearen exekuzio egoera adierazten duten lau eremu (etenak gaitu eta desgaitzeko eremua kasu).

2.2. Konparaketa

Lehen aipatu dudan bezala, *Konputagailuen Egitura* irakasgaian orain arte Nindendo DS-ak duen **ARM9 CPU**-arekin lan egin da. Raspberry Pi-ko modeloek prozesadore berriago batzuekin egiten dute lan eta honen ondorioz konparaketa bat egin beharko da, zein nolako antzekotasunak eta diferentziak dituzten jakiteko.

Hasteko, ARM9-a eta Raspberry Pi 1-eko prozesadoreak konparatuz ia berdinak direla esan daiteke eta gure intereserako datuak ez direla batere aldatu. Diferentzia aipagarrienak optimizazio kontuetan ageri dira, arkitektura bera errespetatuz.

Diferentzia handiagoak aurkituko ditugu aldiz Raspberry Pi 3-ko CPUa eta ARM9 bat konparatuz gero, 2.2. Taulan ikus daitekeen bezala.

	Nintendo DS	**Raspberry Pi 3**
Prozesadorea	ARM946E-S	ARM Cortex A-53
Sorrera urtea	2004	2015
CPU kopurua	1	4 (QuadCore)
Aginduak	32 bit	32 bit
Memoria helbideak	32 bit	64 bit
Erregistro orokorrak	16 erregistro (r0-r15)	31 erregistro (r0-r30) 64 bitekoak

2.2.Taula: ARM9 eta ARM cortex A-53 CPU-ren konparaketa

Ondorioz, errendimendua hobetzeko Raspberry Pi-ko azkeneko prozesadorean aldaketa batzuk sartu dira. Azken finean, gaitasun handiagoa duen makina bat da Raspberry Pi-a. 64 biteko arkitektura programatzerako orduan konplexuagoa denez (CPU core guztien sinkronizazioa beharko da, helbide berriak…), 32 biteko arkitekturako apustua egingo da.

Beste alde batetik, kontuan izan behar da Raspberry Pi txartela librea dela. Txartel honen inguruko informazioa aurkitzea errazagoa izango da helburu komertziala duen gailu batekin konparatuta (Nintendo DS-a kasu). Nintendo DS-a programatzeko liburutegia irekia bada ere dokumentazio eskasa du eta horrengan egiten diren aldaketak traba handiak eragiten diete programatzaileei. Aldiz, Nintendo DS-a grafikoak zuzenean erakusteko prestatua dago. Honek proiektua definitzeko orduan asko laguntzen du, praktikak egiteko aukera gehiago baitaude.

3

ARM mihiztadura lengoaia Raspberry Pi txartelaren barruan

Orain arte erabiliko den txartelaren inguruko informazioa barneratu da. Kapitulu honetan Raspberry Pi txartelean makina lengoaian nola programatu daitekeen ikusiko da, txartel honek informazioa nola prozesatzen duen eta memoria nola erabiltzen duen ikusteko.

3.1. Mihiztadura lengoaian programatzeko aukerak

Raspberry Pi txartelean ARM mihiztadura lengoaian programatzeko hiru aukera probatu ditut.

- QEMU debian emuladorea: **Raspberry Pi fundazioak** gomendatutako sistema eragilearen makina birtual bat da. *Debianen* bertsio zahar bat emulatzen duenez, hasiera hasieratik eguneraketak aplikatu behar zaizkio *Codeblocks* programazio ingurunea arazorik gabe instalatu ahal izateko, nahiz eta zaila ez izan instalatzen. Emuladore honekin izan dudan arazo nagusietako bat teklatuaren erabileran oinarritzen da, nahiz eta konfigurazio fitxategietan aldaketak ezarri, ez duelako Alt eta Alt Gr konbinazio teklak aplikatzen uzten. Arazo honen ondorioz, programazio kontuetan zein sistema beran ibiltzeko guztiz deserosoa iruditu zait.

- Debian sistema Raspberry Pi batean (Raspbian): ARM mihiztadura lengoaian programatzeko beste aukera bat, kasu honetan Raspberry Pi txartel fisikoa erabilita. Hasiera hasieratik emuladorea erabiltzearekiko diferentzia nabaritu nuen bai abiadura aldetik zein erosotasun aldetik. Txartel honekin helburuak betetzeko zailtasunak murrizten zirela ikusi nuen lehenengo momentutik.

- Ubuntu edo Windows sistema batean: GCC zehar-konpiladorearen laguntzaz, ARM mihiztadura lengoaian programatzeko aukera izan daiteke ARM arkitektura ez duen makina batean. Aukera hau proiektuaren **bigarren atalean** erabiliko da soilik, aurreko bi aukerek honek eskaintzen dizkidan ezaugarri berdinak eskaintzen dizkidatelako inongo zehar-konpiladorerik erabili gabe.

Atal honetarako zehar-konpilazio teknika **baztertuta**, QEMU emuladorea eta makina fisikoa konparatuko dira, biak **Raspbian sistemaren barruan**.

Ezaugarria	QEMU raspbian	Raspberry Pi makina
Kostua	0€	35€
Abiadura (sistema martxan, programak ireki...)	x	✓
Eraginkortasuna	x	✓
Efizientzia	x	✓

3.1. Taula: QEMU vs Makina fisikoa

3.1. Taulan agertzen den moduan, makina fisikoa helburuak aurrera eramateko apostu hobeagoa dela ikusi da, ondorengo arrazoiak direla eta:

- QEMU erabiltzean, sistema martxan jartzeko zein programetan ibiltzeko orduan kontuz ibili behar naiz hau ez blokeatzeko.
- Edozein ataza egiteko makina fisikoak denbora gutxiago hartzen du (efizientzia handiagoa).
- Emuladorearekin mihiztadura lengoaian programak idazteko beharrezkoak izango zaizkidan hainbat sinbolo ezin ditut zuzenean idatzi (@, parentesi, kortxeteak...). Makina fisikoan aldiz, bai.

Hala eta guztiz ere, proba txiki batzuk egiteko eta Raspbian sistemarekin hasteko, oso baliagarria da emuladorea, makina fisikoaren kostua kentzen baitu, makina fisikoaren emulazioa egiten duelarik.

3.2. Codeblocks ingurunea

Programak konpilatzeko eta arazteko ingurune baten proba egingo da: *Codeblocks*. *Codeblocks* C eta C++ lengoaietan programatzeko ingurunea da, baina, aldaketa txiki batzuk aplikatuta, ARM mihiztadura lengoaian programatzeko aukera ematen du. Programazio ingurune hau erabiltzea erabaki da ikasleek laborategiak betetzeko behar duten informazioa eskaintzen duelako, memoriaren informazioa eta erregistroen balioak atzitzea esaterako.

Codeblocks instalatzeko, aurretik Raspbian sistema instalatuta dagoela suposatzen da. Sistema hau instalatzeko bideo-tutorial bat dago memoria honetako "[10] Raspbian instalatzen" erreferentzian. Linux sistema batean egingo dudan lehenengo gauza sistema eguneratzea izango da, 3.1. Kodean agertzen den bezala.

```
Sudo aptitude update //Repositorietan gauza berriak dauden ikusteko
Sudo aptitude upgrade //Repositorietako eguneraketak jaisteko
```

3.1. Kodea: Linux sistema eguneratzeko aginduak

Ondoren, behar dudan programazio ingurunea instalatuko dut, 3.2. Kodean dagoen aginduaren bitartez. *Codeblocks*-ek C zein C++ lengoaietan programatzea ahalbidetzen du soilik, baina pauso batzuen bitartez arazorik gabe mihiztadura lengoaian programatzeko aukera izango dut.

```
Sudo aptitude install codeblocks
```

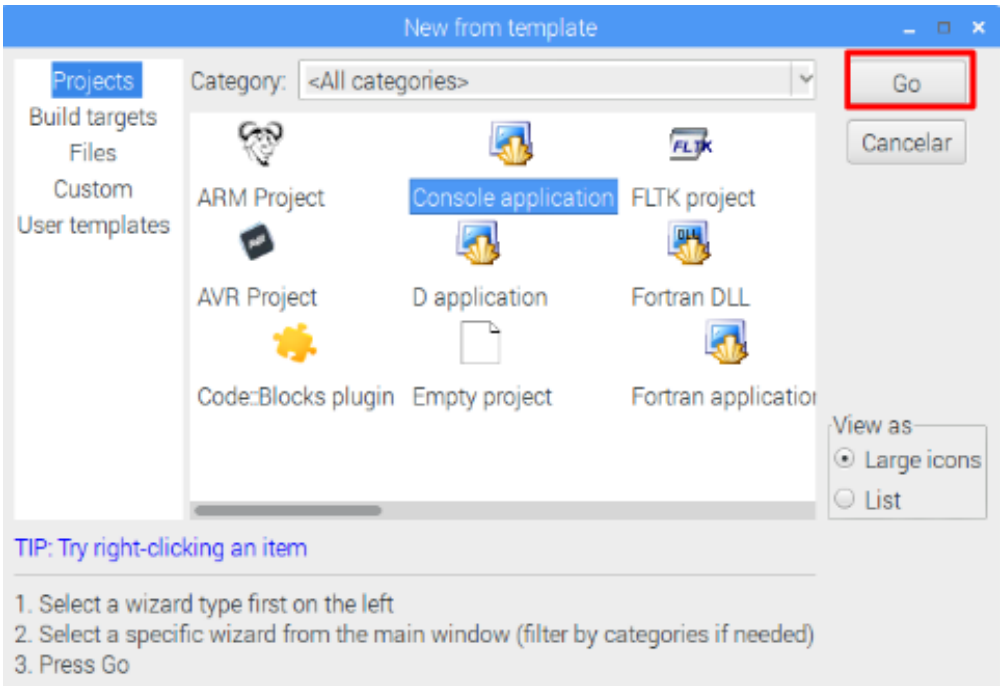
3.2. Kodea: Codeblocks instalatzeko agindua

Behin aplikazioa instalatu dudala, mihiztadura lengoaiako programa bat exekutatu ahal izateko jarraitu beharreko pausoak ikusiko dira.

3.1. Irudian hasierako menua agertzen da, hemen proiektu berri bat irekitzeko aukerari sakatuko diot.

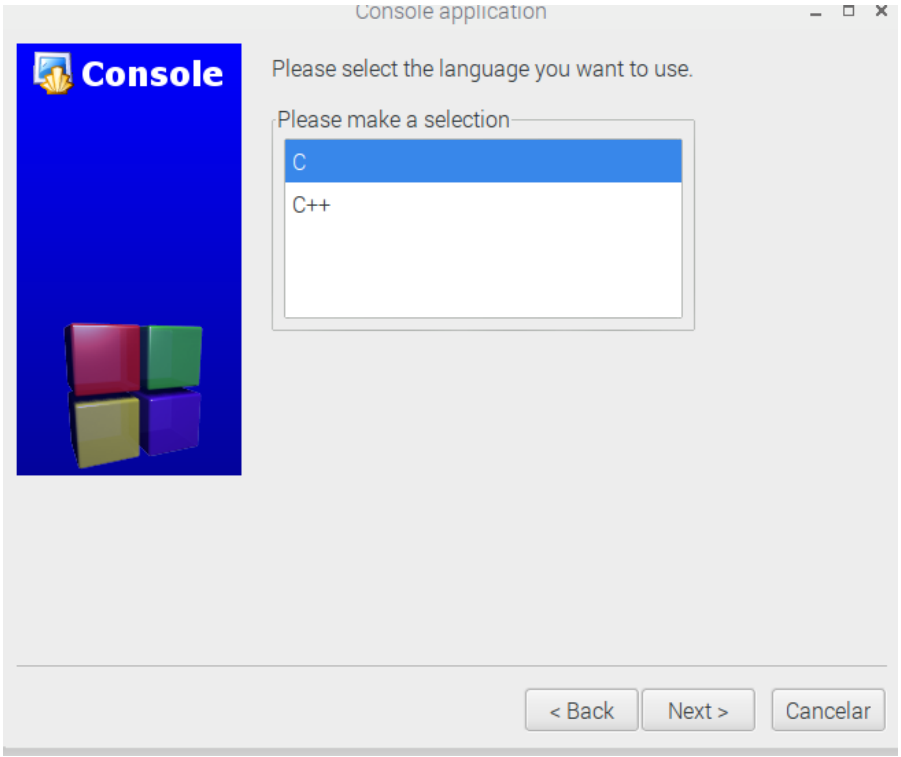
3.1.Irudia: Codeblocks-en hasierako menua

Proiektua sortzeko menura joanda, *Console Application* aukera hartuko dut, 3.2. Irudian agertzen den moduan.



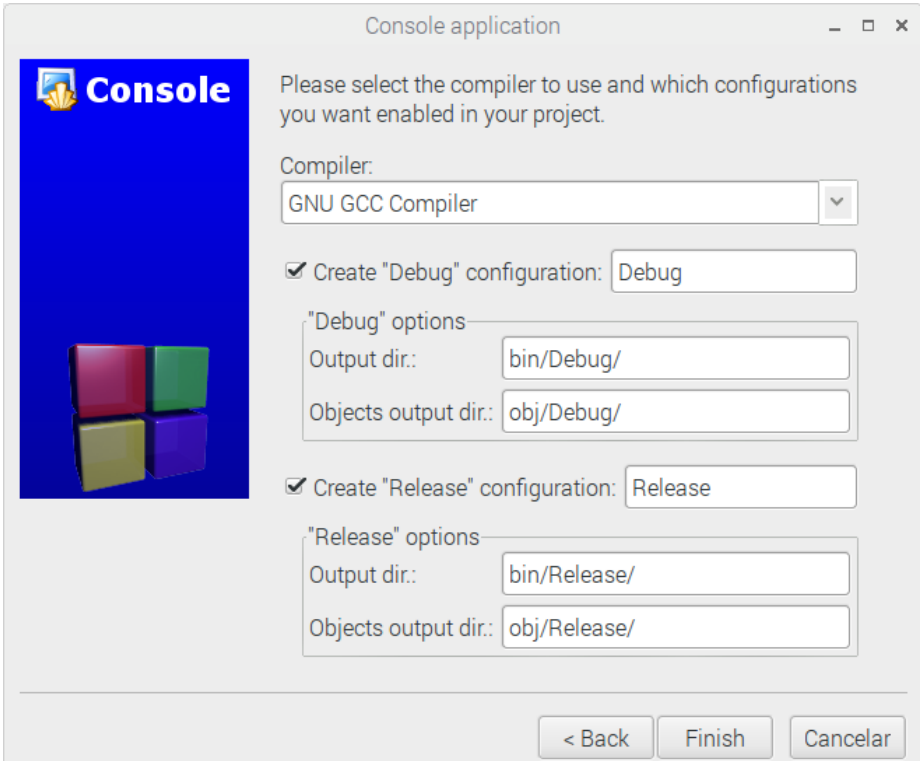
3.2.Irudia: proiektua martxan jartzen

Honen ondoren, C lengoaiko proiektua aukeratuko dut 3.3. Irudian agertzen den bezala.



3.3.Irudia: Lengoaia aukeratzen

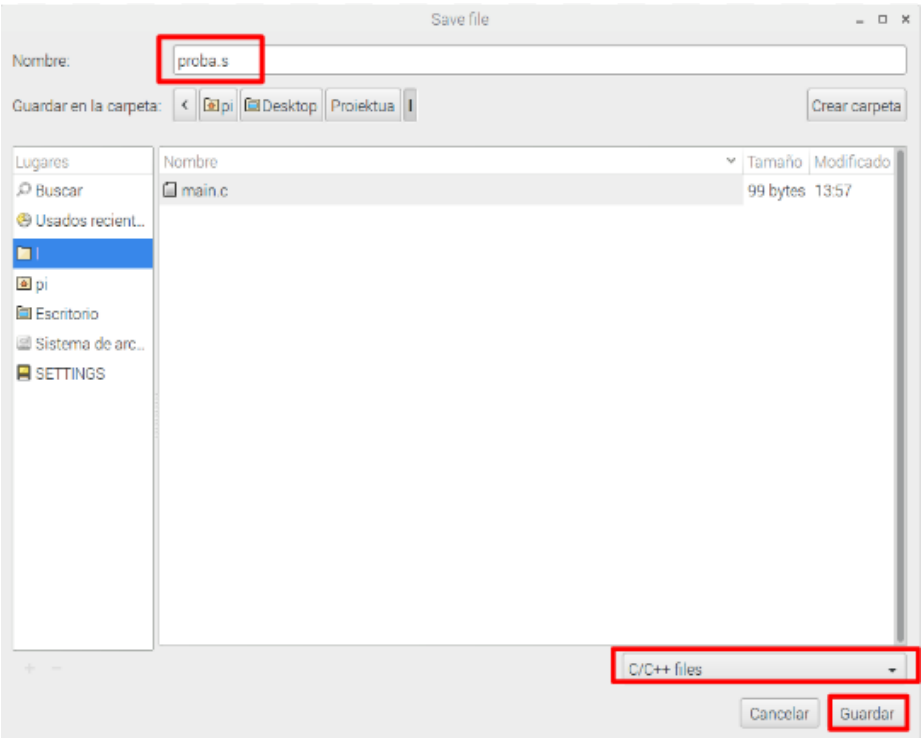
Orain proiektua non gorde nahi den esan beharko diot. Honen ondoren konpilatzailea hartzeko unea dator, 3.4. Irudian agertzen den bezala. *GNU GCC Compiler* aukera baieztatuko da (Oharra: *Debug* aukera sakatuta dagoela berrikusi).



3.4.Irudia: konpiladorea aukeratzen

Pauso honetara iritsita, proiektua sortuta egongo da. Egin beharreko lehenengo gauza sortutako *main.c* fitxategia borratzea izango da, ez dudalako C lengoaian ezer ez programatuko.

Hau eta gero, fitxategi berri bat gehituko diot proiektuari, *file -> new -> empty file* aukeratuz, ikusi 3.5 Irudia.



3.5.Irudia: mihiztadura fitxategia gordetzen

Pausorik inportanteena hemen emango dut: mihiztadura lengoaian exekutatzeko gorde beharreko fitxategiak **.s** luzapena izan beharko dute *(proba.s*, adibidez). Kodearen egitura horrelakoa izan beharko litzateke arazorik gabe konpilatzeko eta exekutatzeko, ondorengo atalean ikusiko den moduan:

```
.data
@beharrezko aldagaiak
.text
.global main
main:  @programa nagusiaren egitura

@ 1. agindua
@ 2. agindua
@ …
@ n. agindua

mov pc, lr @programaren amaiera
```

3.3. Kodea: ARM mihiztadura lengoaiako programen egitura

3.3. Proba: Raspberry Pi-ko ARM-rako moldaketa

3.3.1. Programen egitura

Aurreko atalean *Codeblocks* ingurunea ARM mihiztadura lengoaian programatzeko nola prestatu dudan ikusi da. Atal honetan proba bat egingo da ARM mihiztadura lengoaian.

```
.global main
A: .word 18
B: .word 5
C: .word 0
main:
       ldr r3, A       @Kargatu A
       ldr r2, B     @Kargatu B
       cmp r2,r3
       blt etik1
       mov r4,r3
       b etik2
etik1: mov r4,r2
etik2: str r4, C      @Idatzi C-n
mov pc,lr
```

3.4.Kodea: ARM mihiztadurako lehenengo laborategia

3.4. Kodean erakusten den programan bi zenbakiren arteko konparaketa egiten da, txikiena **C** aldagaia dagoen memoria helbidean idatziz. Programa arazorik gabe konpilatu eta arazteko hainbat arazo izan ditut, Raspberry Pi-ak memoriaren erabilera ezberdina egiten duelako eta beste konpiladore bertsio bat erabiltzen duelako.

Honen ondorioz, *Codeblocks* zein ARM mihiztadura lengoaiako hainbat sasiagindu idatziko ditut, konpilazioak zein arazketak arazorik gabe funtzionatzeko.

Hasteko, Derrigorrez bi sekzio idatzi behar dira, ondorengoak direnak:

- .data: programan erabiliko diren aldagaiak joango dira sekzio honetan.
- .text: kode sekzioa. Ez da derrigorrezkoa ipintzea programak ondo funtzionatzeko, aldiz, bai arazteko orduan arazorik ez izateko eta programaren exekuzioa arazorik gabe jarraitzeko.

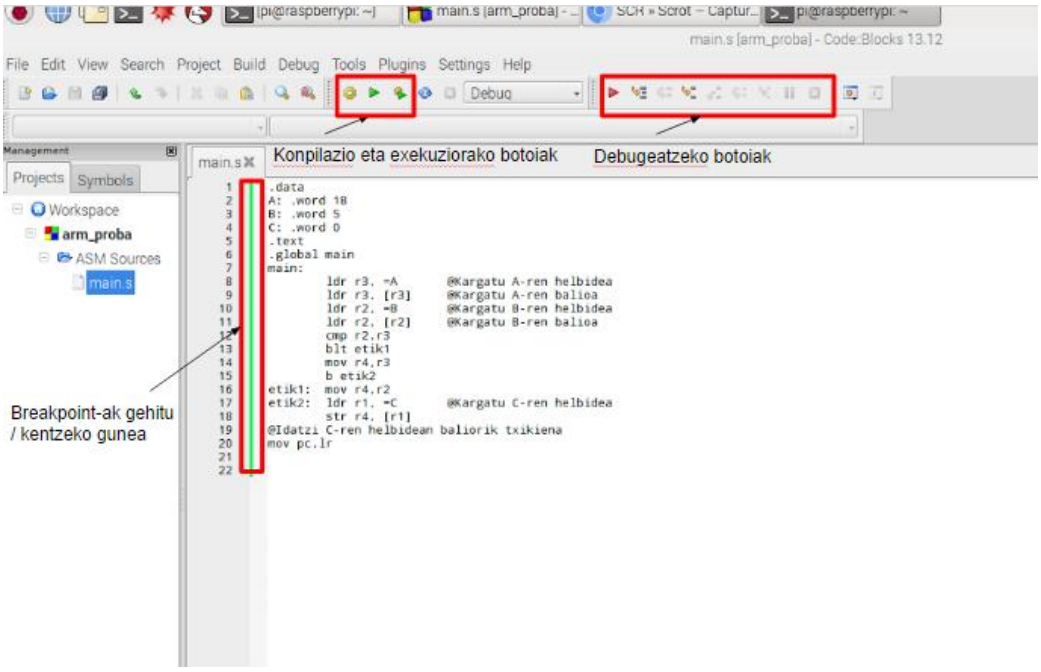
Kodeak bi sekzio dituenez, *.data* sekzioko aldagaiak atzitzeko ezin da helbideratze modu absolutua erabili. Horren ordez, erregistro bidezko zeharkakoa erabili beharko da. Helbideratze modu hau 3.5. Kodean erabiltzen da **A** eta **B** balioak kargatzeko eta **C**-n idazteko.

```
.data
A: .word 18
B: .word 5
C: .word 0
.text
.global main
main:
       ldr r3, =A       @Kargatu A-ren helbidea
       ldr r3, [r3]     @Kargatu A-ren balioa
       ldr r2, =B       @Kargatu B-ren helbidea
       ldr r2, [r2]     @Kargatu B-ren balioa
       cmp r2,r3
       blt etik1
       mov r4,r3
       b etik2
etik1:  mov r4,r2
etik2:  ldr r1, =C       @Kargatu C-ren helbidea
       str r4, [r1]      @Idatzi C-n baliorik txikiena
mov pc,lr
```

3.5.Kodea: Lehenengo laborategia moldatuta

3.3.2. Programen exekuzioa eta arazketa

Atal honetan 3.5. Kodea exekutatu eta araztuko da. 3.6. Irudiak erakusten du konpilatzeko zein arazteko beharrezkoak zaizkidan aginduak eta hauen kokapena *Codeblocks* ingurunearen interfazean.



3.6.Irudia: Kodea idazten

Baina erregistroen edukia eta memoriak programaren exekuzioaren une jakin batean duen edukia ikusi nahi bada, zer egin daiteke? *Codeblocks*-ek modu erraz batean informazio hau ikusteko aukera ematen du. 3.6. Irudian erakusten den moduan, behin *breakpoint-ak* (arratoiaren eskuineko botoiarekin, argazkiak esaten duen atalean) jarri direla eta arazteko botoiari emanda (play gorria), hiru leiho aktibatu behar dira ***Debug -> debuging Windows*** atalean:

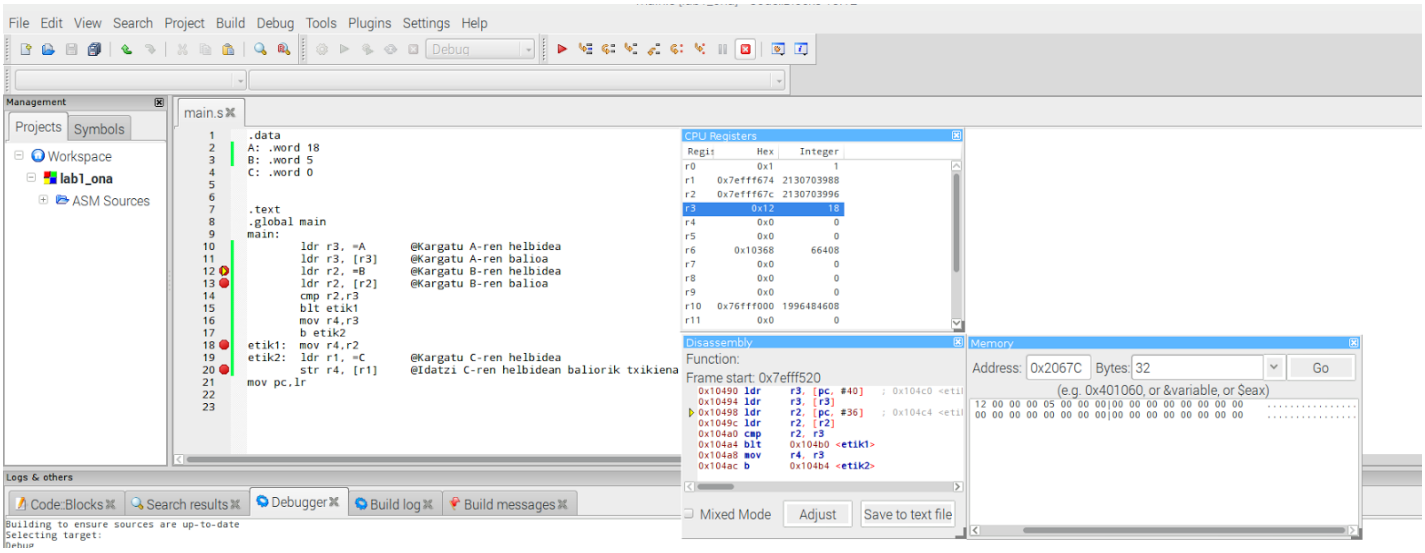
- CPU registers: Erregistro orokorren zein berezien (PC, SP, LR) uneko balioa erakusten du.

- Dissasembly: Agindu bakoitza zein memoria helbidetan gordetzen den ikus daiteke. *Memory Dump* leihoarekin batuta, agindu bakoitzak memorian zein datu gordetzen duen ikusteko balio izango du.

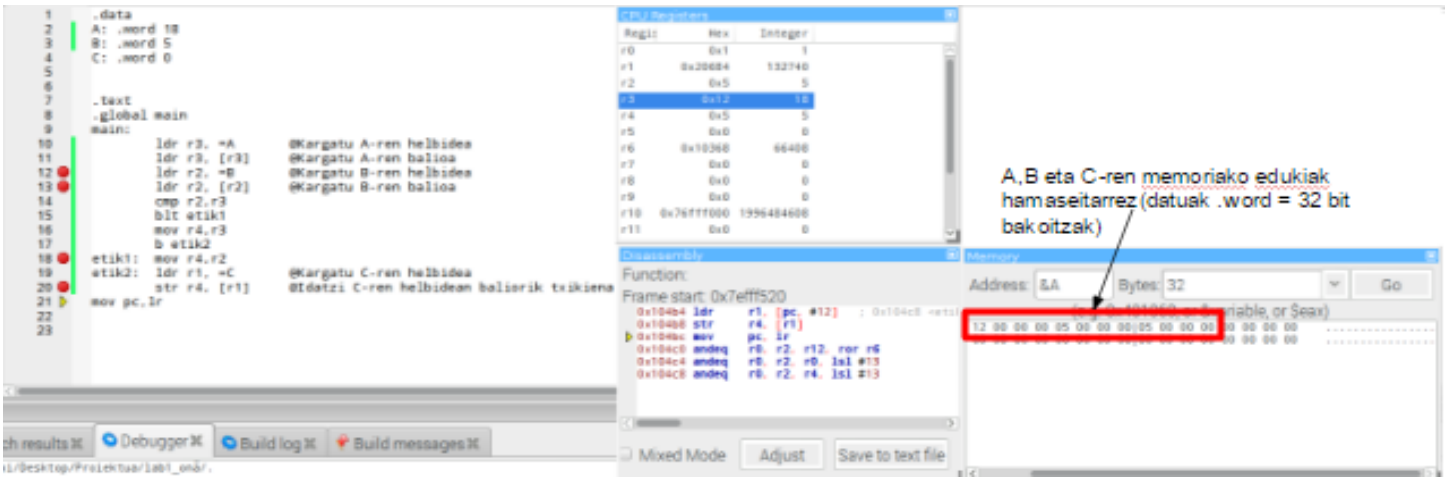
- Memory Dump: Memoria ikusteko leihoa. Bi atal nagusi ditu: ikusi nahi den memoria helbidea (*Address*) eta ikusi nahi den memoria luzera (*Bytes*). Lehenengo atalean memoria helbidea bi modutan ipini daiteke:
    - Memoria helbidea zuzenean jarrita (32 bit), honetarako metodorik egokiena hamaseitarrez jartzea da, 0x10490 esaterako.
    - Aldagai baten helbidea jarrita, programa honetarako &A, &B edo &C jarrita, memoria helbide hauetako datuak ikusteko aukera izango dut.

Hau egin ondoren, 3.7. Irudian agertzen zaigun interfazearen berdina ikusiko beharko da.

Gezi horiak erakusten du programaren exekuzioa zein momentuan dagoen. Hori oso lagungarria da programaren koherentzia mantentzen den ala ez jakiteko. Bukatzeko, programaren amaiera erakusten du 3.8. Irudiak. Programak egin beharrekoak egin dituela ikusteko erregistro zein memoriaren balioak aztertuko ditut.



3.7.Irudia: Arazteko leihoa



3.8.Irudia: Programaren amaierako balioak

Ikusten denez programaren amaieran ondorengo balioak ageri dira exekuzioan erabilitako erregistroetan:

- R1 = 0x20684 (C-ren helbidea)
- R2 = 5
- R3 = 18
- R4 = 5

eta memoria helbidetan:

- &A = 12 00 00 00 hamaseitarrez
- &B = 05 00 00 00 hamaseitarrez
- &C = 05 00 00 00 hamaseitarrez

Ondorioz, programa ondo dabilela ikusten eta ingurune honek Raspberry Pi-arekin batera aginduen behe mailako exekuzioa aztertzeko balio duela ikusi da.

Informazio gehigarri moduan:

- Programaren itzulera kodean bi aukera daude:
    - R0 erregistro orokorrak programaren amaieran duen balioa bueltatzen du. Kasu honetan bateko balioa bueltatuko du.
    - Errorea bueltatzen du, SEGMENTATION FAULT-en kasuan 139 errore kodea bueltatuz.

- Datuen Biltegiratzea RAM memorian:
    - Raspberry Pi 3-an (probatu dudan txartelean), datuak 0x20607c helbidetik aurrera biltegiratzen ditu.
    - Aginduak aldiz, 0x10490 helbidetik aurrera.

4

Sarrera / Irteera kontzeptuak Raspberry Pi txartelaren barruan

Kapitulu honetan Raspberry Pi txartelak eskaintzen dituen interfazeen bitartez, beste gailu elektronikoekin konektatzeko aukera izango dut. Txartelak dituen periferikoen artean GPIO portua eta *timerrak* erabiliko ditut. Horrela, *Konputagailuen Egitura* irakasgaian ematen diren Sarrera/Irteera azpisistemari buruzko kontzeptu teorikoak praktikara eramateko aukera izango dut.

Linux sistema batekin lanean egonda, hainbat murriztapen izango ditut periferikoak atzitzeko. Murriztapen hauek gainditzeko sistema eragilerik gabe programatzeko aukera hartu da, *bare metal* izenekoa (Geroago sakonki azalduko da). Aurkeztu zen arazo honen ondorioz, Sarrera / Irteera kontzeptuak programatzeko kapitulu hau bi zatitan banatu da, lehenengoa Raspbian sistema eragilearen barnean eta bigarrena, *bare metal* programazioaren bitartez.

4.1. Raspberry Pi-aren periferikoen atzipena

4.1.1. GPIO portua

GPIO (General Purpose Input/Output) portua Raspberry Pi txartelak kanpoko gailuekin konektatzeko duen aukeretako bat da. Raspberry Pi txartelaren hasierako bertsioek, 26 pineko GPIO portua erabiltzen zuten; A+, B+, 2 eta 3 bertsioetan aldiz, 40 pineko GPIO portua erabiltzen hasi ziren, proiektu handiagoak bezain konplexuagoak egiten ahalbidetzen dituztenak.

40 pin horiek betetzen duten funtzioaren arabera sailkatuko ditut, guztiek ez dituztelako ezaugarri berberak:

- Elikatze pinak: 5V, 3,3V eta 0V (lurra: GND), hainbat tentsioetan zirkuitu elektronikoak funtzionarazten dituzte. Bateria edo pila baten funtzio berdina egin dezakete.
- GPIO pinak: 3,3V-ko konexio programagarriak dituzten pinak gure proiektuetan erabili ahal izateko.

Behin pinen ezaugarriak ikusita, Raspberry Pi txartelean duten banaketa ikusiko da orain, 4.1. Irudian agertzen den moduan.

4.1.Irudia: Pinen kokapena GPIO portuan

Kontuz ibili behar da programatzen den GPIO pinak eta pin fisikoak zenbaki ezberdinak dituztelako. Adibidez, C-ko lehenengo programan GPIO17 pina programatzen denean interkonexio hariak 11. pin fisikora konektatu beharko dira.

4.1.2. Sarrera / Irteerako erregistroen memoria espazioa

Raspberry Pi-an Sarrera / Irteerako erregistro guztiak memorian mapeatuta daude. Raspberry Pi-ak duen memoria espazio osoa ikusi beharrean, praktikak garatzeko intereseko periferikoak zein memoria helbidetan dauden ikusiko dira soilik. Raspberry Pi-ak periferikoentzat gordeta duen memoria espazioa 4.1. Taulan ikusi ahal izango da.

	Helbide fisikoaren tartea
Raspberry Pi 1	0x20000000 / 0x20FFFFFF
Raspberry Pi 2, 3	0x3F000000 / 0x3FFFFFFF

4.1. Taula: Periferikoentzat esleitutako helbide tartea

4.1.3. GPIO erregistroen memoria espazioa

GPIO portua 0x20200000 helbide fisikotik aurrera dago atzigarri baina bertsio berriagoetan (Raspberry pi 1-etik aurrerakoak) memoria espazioaren esleipena aldatu egin zen, portu honentzat 0x3F200000 helbide fisikoa erreserbatuz, 4.2. Taulan ikus daitekeen bezala.

	Helbide fisikoaren hasiera
Raspberry Pi 1	0x20200000
Raspberry Pi 2, 3	0x3F200000

4.2. taula: GPIO portuarentzat esleitutako helbide tartea

GPIO portuak 41 erregistro ditu, guztiak 32 bitekoak. Erregistro hauetatik pin bakoitzaren informazio asko jaso daiteke:

- Zein funtzio betetzen duen pinak: Input edo output moduak erabiliko dira soilik, baina gehiago daude.
- Zein tentsio jasotzen edo ematen ari den pin jakin bat.
- Tentsio aldaketa baten ondorioz sortutako gertaerak jakinarazi, programatzaileak hauek tratatzeko.

4.3. Taulan, praktikan erabiliko diren erregistroen helbideak ikus daitezke.

Erregistro izena	Memoria helbide fisikoa
GPFSEL0	0x20200000
GPFSEL1	0x20200004
GPFSEL2	0x20200008
GPFSEL3	0x2020000C
GPFSEL4	0x20200010
GPFSEL5	0x20200014
GPSET0	0x2020001C
GPSET1	0x20200020
GPCLR0	0x20200028
GPCLR1	0x2020002C
GPLEV0	0x20200034
GPLEV1	0x20200038
GPEDS0	0x20200040
GPEDS1	0x20200044
GPAREN0	0x2020007C
GPAREN1	0x20200080
GPAFEN0	0x20200088
GPAFEN1	0x2020008C

4.3. taula: GPIO erregistroen kokapena memorian

4.3. Taulan ez dira erregistro guztiak ipini, programatzeko erabiliko direnak baizik. Raspberry Pi-ak 54 GPIO pin maneiatzeko erregistroak ditu, aldiz, portu fisikoan ez dira erdia inplementatzera iritsi. Gainontzeko pinak hurrengo bertsioetan inplementatzea aurreikusten da.

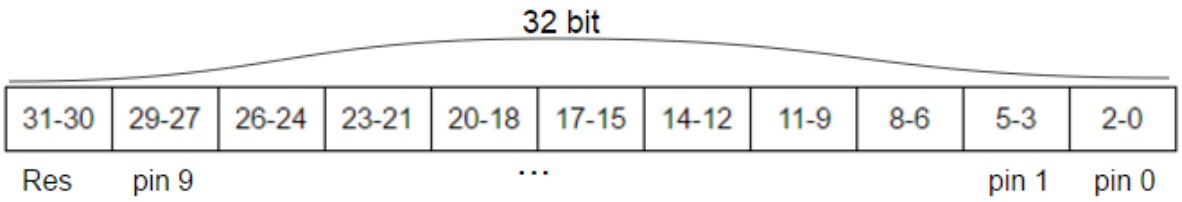
4.1.4. GPIO erregistroak beren funtzioaren arabera

Programatzerako orduan kontuan izan behar diren erregistroak agertuko dira atal honetan:

- GPFSEL erregistroak (Function Select): Pinen portaera gorde eta ezartzen duten sei erregistro. Geroago ikusiko da erregistroen egitura, baina modu sinple batean esanda, erregistro bakoitzak 10 pinen (3 bit pin bakoitzeko) portaera gordetzen du eta gainontzeko bitak erreserbatuta daude.

  Pin bakoitzak zein egoera izan ditzake?
  - 000 = IN
  - 001 = OUT

  Egoera hauetaz aparte, pinak ordezko sei funtzio egitera iritsi daitezke hainbat komunikazio protokolo erabiliz, PWM edo RX/TX serie-lerroa esaterako. Ez naiz puntu honetan gehiago sartuko praktiketan INPUT eta OUTPUT funtzioak erabiltzen direlako soilik. Erregistro hauetako bakoitzaren egitura ondorengoa da, adibidean GPFSEL0 erregistroa egonda:



- GPSET erregistroa (SET = ezarri) : OUTPUT bezala ezarrita dauden pinak aktibatzeko bi erregistro. Dagokion pinaren bitean 1 jarriz gero, pin hori HIGH bezala jarriko da, zirkuituari 3,3V eskainiz. 0 jarriz gero ez du inongo efekturik sortuko.

  Guztira bi erregistro daude, horietatik lehenengoa erabiliko da:

GPSET0

Bitak	eremu izena	Deskripzioa
31-0	SETn → (n=0..31)	n=0 → efekturik ez n=1 → ezarri n. pina

- GPCLR erregistroak (CLR = garbitu): OUTPUT bezala ezarrita dauden pinak desaktibatzeko bi erregistro. Dagokion pinaren bitean 1 jarriz gero, pin hori LOW bezala jarriko da, zirkuituari 0V eskainiz. Lehen bezala, 0 jarriz gero ez du inongo efekturik sortuko.

  Guztira bi erregistro daude, horietatik lehenengoa erabiliko da:

GPCLR0

Bitak	eremu izena	Deskripzioa
31-0	CLRn → (n=0..31)	n=0 → efekturik ez n=1 → ezabatu n. pina

- GPLEV erregistroak (LEV = tentsio maila): Pin jakin bat irakurtzeko INPUT erregistroa. Bit bat batean dagoenean, dagokion pin-a 3,3V (edo kasu batzuetan 5V) jasotzen ari dela esan nahi du, aldiz zeroan dagoenean, 0V jasotzen ari dela.

  Guztira bi erregistro daude, horietatik lehenengoa erabiliko da:

GPLEV0

Bitak	eremu izena	Deskripzioa
31-0	LEVn → (n=0..31)	n=0 → n. pina tentsio baxuan dago lanean. n=1 → n. pina tentsio altuan dago lanean.

- GPEDS erregistroak (EDS = Event Detect Status) : Pin jakin batean gertaera bat gertatzen baldin bada, erregistro hauei dagokien pineko bitean batekoa jartzen da. Erregistro hauek zirkuituetan sor daitezkeen gertaerak detektatzeko erabiliko dira, eta hauek gertatzean, eten bat sorrarazteko.

  Guztira bi erregistro daude, horietatik lehenengoa erabiliko da:

GPEDS0

Bitak	eremu izena	Deskripzioa
31-0	EDSn → (n=0..31)	n=0 → n. pinean ez da gertaerarik detektatu n=1 → Gertaera detektatu da n. pinean

Hainbat motatako gertaerak egon litezke: goranzko trantsizioa, beheranzko trantsizioa, pina tentsio jakin batean detektatzea… errazena trantsizioak detektatzea izango litzake, hau da, Raspberry Pi-ko GPIO portuan tentsio aldaketa bat sumatzen bada honek osatzen duten pinen bitartez, prozesadoreari jakinaraziko zaio hau eten bat sor dezan. Horretarako ondorengo erregistroak daude:

- GPAREN eta GPAFEN erregistroak (AREN = Async. Rising Edge Detect ; AFEN = Async. Falling edge Detect): Goranzko zein beheranzko trantsizio asinkronoak detektatzen dituzten erregistroak. Tentsio aldaketa bat gertatzen bada pin jakin batean eta pin horrek trantsizioak aktibatuta baditu, GPEDS0 erregistroan gertaera bat jaso dela igarriko da.

GPAREN0

Bitak	eremu izena	Deskripzioa
31-0	ARENn → (n=0..31)	n=0 → Goranzko trantsizioaren detekzioa n. pinean desgaituta n=1 → Goranzko trantsizioaren detekzioa n. pinean gaituta

GPAFEN0

Bitak	eremu izena	Deskripzioa
31-0	AFENn → (n=0..31)	n=0 → Beheranzko trantsizioaren detektaketa n. pinean desgaituta n=1 → Beheranzko trantsizioaren detektaketa n. pinean gaituta

Oharrak:

- Erregistro bakoitza zein alegiazko helbidetan agertzen den ikusteko, hemen klikatu.
- Pin bakoitzak zein eta zenbat funtzio izan ditzakeen ikusteko, hemen klikatu (dokumentuaren 102-103 orrialdeetan)

4.1.5. Timerrak

Denboraren kontrola izateko bi *timer* eskaintzen ditu Raspberry Pi txartelak:

- Sistemaren *timerra*: 1Mhz-ko maiztasun finkoarekin inkrementatzen doan kontagailua da hau. Praktikan *Delay* funtzioak inplementatzeko erabili daiteke, beraz, *timer* honek dituen erregistroetatik, ondorengoa erabiliko da:
    - Pisu txikieneko bitak hartzen dituen kontagailua: 0x20003004 memoria helbidean mapeatuta dagoen erregistroa. *Delay* funtzioaren inplementazioa 4.1. Kodean ageri da.

```
void timerDelay( uint32_t us )
{
   volatile uint32_t ts = rpiSystemTimer->counter_lo;
   while( ( rpiSystemTimer->counter_lo - ts ) < us )
   {
       //Ezer ez egin
   }
}
```

4.1.Kodea: Sistemaren timerraren funtzioa

4.1. Kodean ikusten denez, denbora tartearen hasiera `ts` aldagaian hartzen da eta kontagailua tarte batean inkrementatzen joango da (`rpiSystemTimer->counter_lo`, 0x20003004 helbideari erreferentzia egiten dion erakusle bat da).

- ARM *timerra*: Aurreko *timerraren* deribatu bat da, baina bere funtzionamendua ez da berdina. Kasu honetan, kontagailua 1Mhz-ko (hau maiztasun zatitzailearekin alda daiteke) maiztasunarekin dekrementatzen doa zerora iritsi arte. Zerora iristen denean eten seinale bat bidaliko dio eten kudeatzaileari, honek eten mota hau trata dezan. *Timer* honen IRQ-a 0x2000B218 helbidean dagoen `Enable_Basic_Interrupt` erregistroaren 0. bitean batekoa jarriz gaituko da. *Timer* honekin ondorengo erregistroak erabiliko dira:
    - Load erregistroa: nahi den 32 biteko balioa kargatzen da, maiztasunaren arabera periodikoki etenak sortzeko.
    - Value erregistroa: *timerraren* uneko balioa irakurtzeko erregistroa.
    - Kontrol erregistro bat: beste hainbat gauzen artean, maiztasun zatitzailea konfiguratzeko balio du. Maiztasun zatitzailea *timerraren* maiztasuna txikitzeko erabiltzen da, etenen arteko denbora handituz. Hiru modu daude:
        - Maiztasun zatitzailea 1: Maiztasun zatitzailerik gabe lan egitea (1Mhz / 1). Modu hau aktibatzeko erregistro honetako 2-3 bitetan 00 balioa jarri beharko da.
        - Maiztasun zatitzailea 16: Sistemaren *timerra* zati 16 (1Mhz / 16). Modu hau aktibatzeko erregistro honetako 2-3 bitetan 01 balioa jarri beharko da.
        - Maiztasun zatitzailea 256: Sistemaren *timerra* zati 256 (1Mhz / 256). Modu hau aktibatzeko erregistro honetako 2-3 bitetan 10 balioa jarri beharko da.
    - IRQ garbitzeko erregistroa: Erregistro honen bidez, kontagailuaren etena tratatu dela eta eten kudeatzailea IRQ honen hurrengo etenak jasotzeko prest dagoela adierazi daiteke. IRQ-a erregistro honetan bateko bat idatziz garbitzen da.

4.1.6. Timerraren erregistroen memoria espazioa

- Sistemaren *timerra*: periferiko honen memoria fisikoaren hasierako helbidea 0x20003000 (0x3F003000 Raspberry Pi 3-an) da. 4.4. Taulan erabiliko den erregistroa azaltzen da:

Erregistro izena	Memoria helbide fisikoa
CLO (Kontagailuaren pisu txikieneko 32 bitak)	0x20003004

4.4. taula: Sistemaren timerraren erregistroen kokapena memorian

- ARM *timerra*: periferiko honen erregistroak 0x2000B400 helbidetik aurrera mapeatzen dira memorian. 4.5. Taulan erabiliko diren erregistroak azaltzen dira:

Erregistro izena	Memoria helbide fisikoa
Load (balioa kargatu)	0x2000B400
Value (timerraren kontagailua)	0x2000B404
Control	0x2000B408
IRQ Clear	0x2000B40C

4.5. taula: ARM timerraren erregistroen kokapena memorian

4.1.7. Etenak

Eten bat salbuespen bat balitz bezala tratatzen da ARM prozesadoreetan. Eten bat gertatzean PC erregistroa aldatuz (programaren uneko aginduaren helbidea duen erregistroa), aurredefinitutako memoria tarte batera salto egingo da, exekutatu nahi den kodea exekutatzeko (zerbitzu errutina orokorra). Etenaren amaieran eta zerbitzu errutina exekutatuta, etendako programara bueltatuko da exekuzioa.

Lehen esan bezala, eten eskaera bat jasotzean, jauzi bat gertatuko da aurredefinitutako helbide batera joanez. Aurredefinitutako helbide hauek gordetzen dituen bektoreari **eten bektorea** deritzo.

4.1.7.1 Eten bektorea eta eten kudeatzailea (Dispatcherra)

Salbuespen izena	Memoria helbidea	Erabiltzaile modua
Reset	0x00	Supervisor
Undefined instruction	0x04	Undefined
Software	0x08	Supervisor
Data Prefetch	0x0C	Abort
Data abort	0x10	Abort
IRQ	0x18	Interrupt
IFQ (fast)	0x1C	Fast Interrupt

4.6. taula: Eten bektorea

Praktikatan IRQ etenak bakarrik erabiliko dira, baina eten mota guztiak tratatu ahal izateko eten bektorea osoa kargatuko da. 4.6. Taulan agertzen den bezala, eten bektorea 0x00 helbidean hasten da, eten bat gertatzean kodea RAM memoria helbide honetara salto egiteko. Eten kudeatzailearen (nik programatuta) lana izango da orain eten mota identifikatzea eta honi dagokion zerbitzu errutinara salto egitea.

4.6. Taulan ARM arkitekturan dauden erabiltzaile moduak agertzen dira ere. Hasieran *Supervisor* bezala hasten da exekuzioa. IRQ eten bat gertatzean hau tratatzeko exekuzioa *Interrupt* moduan kokatu beharko da, etena tratatu ondoren berriz ere *Supervisor* modura itzuliz.

Erabiltzaile modua aldatzeko, CPSR erregistroaren bitartez egingo da, gero programatzerako orduan azalduko den bezala.

4.1.7.2 Eten kudeatzailearen (Dispatcherra) memoria espazioa

Eten kudeatzailearen erregistroak 0x2000B200 helbidetik aurrera mapeatzen dira memorian. Erabiliko diren erregistroak ondorengoak dira:

Erregistro izena	Memoria helbide fisikoa
IRQ basic pending	0x2000B200
IRQ pending 1	0x2000B204
IRQ pending 2	0x2000B208
Enable IRQs 1	0x2000B210
Enable IRQs 2	0x2000B214
Enable Basic IRQs	0x2000B218
Disable IRQs 1	0x2000B21C
Disable IRQs 2	0x2000B220
Disable Basic IRQs	0x2000B224

4.7. taula: Eten kontroladorearen erregistroen kokapena memorian

4.7. Taulako erregistroak hiru multzo nagusietan banatzen dira, betetzen duten funtzioaren arabera:

- Pending erregistroak: etenen informazioa ematen duten erregistroak, dagoeneko tratatuak dauden ala ez jakiteko.
- Enable erregistroak: etenak gaitzeko.
- Disable erregistroak: etenak desgaitzeko.

IRQ eskaeren informazioa beraien zenbakiaren arabera erregistro batean edo bestean egongo da, hau da, 0-31 bitartekoak ***IRQs 1*** erregistroan egongo dira eta IRQ zenbaki handiagoa dutenak (32-63), ***IRQs 2*** erregistroan. Badago beste erregistro bat ***IRQ basic*** etenak hartzen dituena, ARM *timerraren* IRQ-a kasu.

Programatzerako orduan interesekoak diren IRQ-ak ondorengoak dira:

- GPIO portutik datozen etenak: *gpio_int[3]* IRQ-a izena du. IRQ hau 52 zenbakiarekin zenbakituta dago, honen ondorioz ***IRQs 2*** erregistroetan sartzen da, zehazki erregistro hauetako 20. bitean.
- ARM *timerretik* datozen etenentzat: ARM *timerra* ***Basic IRQs*** erregistroen barruan sartzen den IRQ-a da. Erregistro hauetako 0. bitean batekoa idatziz gero, eten hauek gaitu, irakurri edo desgaitu ahal izango dira.

Aldiz, exekuzioan etenak ez dute funtzionatuko etenak baimentzen diren arte, horren kargu *CPSR erregistroa* egiten delarik. 32 biteko erregistro honen barnean, intereseko bita zazpigarrena izango da, IRQ-ak gaitu edo desgaitzeko aukera ematen duena. Bit honetan batekoa jarriz, IRQ guztiak desgaituko dira, beraz etenak jaso eta tratatzeko bit hau zerokoa dela zihurtatu beharko naiz.

4.2. Programazioa Raspbian sistema eragilearen barnean

Kapituluaren hasieran aipatu dut sistema eragilearen barruan programatuta hainbat murriztapen daudela. Atal honetan Sistema eragilearen barruan egin daitezkeen gauzen artean ondorengoak ikusiko dira:

- GPIO portuaren atzipena.
- Inkesta bidezko sinkronizazio.

4.2.1 Raspbian sisteman GPIO portua programatu

Probak Raspberry Pi 3-an daude eginak. Raspberry Pi 1-ean erabiltzeko periferikoen helbidea 0x20000000 izan beharko luke. Sistema eragilea tartean egonda, memoriako helbide fisikoak ezin dira zuzenean atzitu. Beraz, Raspbian sistema eragilearekin lan egiteko, egin beharreko lehenengo gauza memoria atzitzeko beharrezko baimenak lortzea izango da. 4.2. Kodean dagoen bezala, memoriaren atzipena fitxategi baten bidez burutuko da.

```
/*Hasieraketak*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define BCM2710_PERI_BASE        0x3F000000 /* 1: Raspberry Pi 3-ak
periferikoentzat duen hasierako helbidea*/
#define GPIO_BASE                (BCM2710_PERI_BASE + 0x00200000) /* GPIO-
aren hasierako helbidea */

//Memoria atzipena
int  mem_fd;
void *gpio_map;
volatile unsigned *gpio;

//Konfigurazio funtzioa
void setup_io()
{
  /*2: ireki /dev/mem fitxategia*/
  if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
     printf("can't open /dev/mem \n");
     exit(-1);
  }

  /*3: RAM memorian mapeaketa */
  gpio_map = mmap(
     NULL,             //Any adddress in our space will do
     1,       //Map minimum length
     PROT_READ|PROT_WRITE,// Enable reading & writting to mapped memory
     MAP_SHARED,       //Shared with other processes
     mem_fd,           //File to map
     GPIO_BASE         //Offset to GPIO peripheral
  );

  close(mem_fd);

  if (gpio_map == MAP_FAILED) {
     printf("mmap error %d\n", (int)gpio_map);//errno also set!
     exit(-1);
  }
  // Always use volatile pointer!
  gpio = (volatile unsigned *)gpio_map;
```

```
}

/*4: Erregistroak */
// OUTPUT edo INPUT bezala jartzeko erregistroak
#define GPFSEL0  gpio
#define GPFSEL1  gpio+1
#define GPFSEL2  gpio+2

// OUTPUT bezala jarritako pina ezarri edo garbitzeko erregistroak
#define GPSET0  gpio+7
#define GPCLR0  gpio+10

// INPUT bezala jarritako pinaren tentsioa irakurtzeko erregistroa
#define GPLEV0  gpio+13
```

4.2. Kodea: Raspbian sisteman hardwarea atzitzeko fitxategia

4.2. Kodean ondorengoa egiten da:

1. Aurreko atalean ikusi bezala, GPIO periferikoaren hasierako helbidea 0x3F200000-an jartzen dut Raspberry Pi 3 batean lanean nagoelako.
2. */dev/mem* fitxategia irekitzen dut memoria fisiko osoa atzitu ahal izateko.
3. *mmap* funtzioaren bitartez eta ezarrita dituen parametroen bitartez ondorengoa egingo dut: 0x3F200000-tik 0x3F2FFFFF bitarteko helbide tartea memorian mapeatu, helbide hauek erabili eta programatu ahal izateko.
4. Mapeaketa honen hasierari erreferentzia egiten dion erakusle bat erabiliko dut (GPIO). Memoriako erregistroak 32 bitekoak direnez, erakuslearen luzera berdinekoa izango da eta honen ondorioz, memoria saltoak hitzekoak izango dira.
5. Erregistroen izenen itzulpena, prozesadorearen datu orrialdearen arabera jarrita.

Hasieraketa hauek ezarri eta gero, GPIO erakuslearen bidez, portu honetako Sarrera / Irteerako erregistroak atzitu ahal izango dira, lehenengo praktikak betetzeko baliagarria izango dena. Eranskinean praktiketan erabilitako makro zein funtzioak azalduko dira.

4.2.2 Raspbian sistemarekin egindako Probak

4.2.2.1 Sarrera / Irteerako erregistroen atzipena

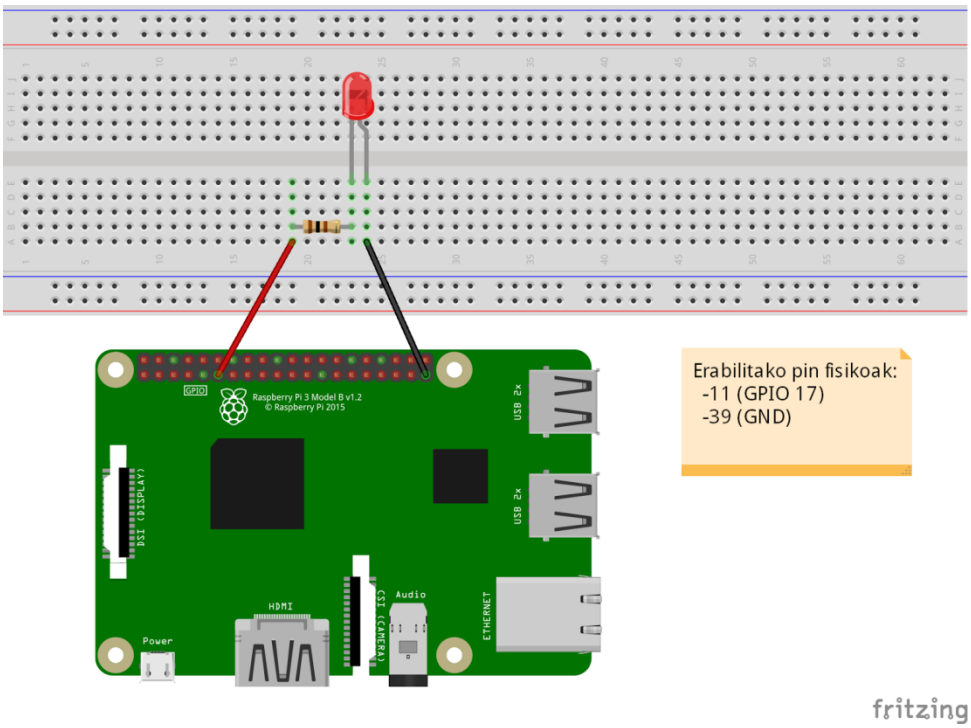
Azalpena eta kodea

Lehenengo proba honetan erregistroak nola atzitzen diren ikusiko dut. Maskarak aplikatuz erregistroak konfiguratuko ditut, hauetan idatziz Led bat piztu edo itzaltzeko gai izango naiz. Proba honetan erabiliko ditudan gailuak ondorengoak dira:

- Raspberry Pi-a
- *Breadboard* konexio taula
- 75 edo 100 Ohm-eko erresistentzia bat
- Led gorri bat (1,9V)
- Interkonexio hariak

Gailu hauek interkonektatzeko bi pin erabiliko ditut, 4.2. Irudian agertzen den bezala:

- GPIO 17 (11. pin fisikoa)
- GND pin bat (Kasu honetan 39. pin fisikoa)



4.2. Irudia: Lehenengo probaren eskema

Zirkuituan mugatutako korrontea Ledera eramatea dut helburu, hau piztu ahal izateko. Eskeman ikusten den bezala, GPIO 17 pinak 3,3 V aterako ditu gure Leda pizteko. Ohm-en legearen arabera, zirkuituan zehar egongo den korrontea mugatzeko, erresistentzia bat jarri beharko diot. Erresistentziaren kalkulua erabiliko dudan Ledak mugatzen du (eta GPIO pinaren voltaiak), kasu honetan 1,9 volt-ekoa dena. Kalkuluak aterata eta kontuan izanda Led konbentzional batek jasan dezakeen korronterik altuena 20milianpereo inguruan dagoela, 75 Ohm-eko erresistentzia bat egokia izango da (100 Ohm-ekoak berdin-berdin balio du) nire zirkuituak arazorik gabe funtzionatzeko.

Behin proba honen eskema fisikoa zein den ikusita, GPIO 17 pinean konektatuta dagoen Leda segunduro piztu eta itzaltzea nahi da. Horretarako ondorengo pausoak emango ditut:

1. 17. GPIO pina INPUT bezala jarri, horretarako, ondorengo maskara aplikatuko diot AND eragiketa logikoarekin: 0xFF1FFFFF (21-23 bitarteko bitak zerokoak dira)
2. 17. GPIO pina OUTPUT bezala jarri, horretarako, OR eragiketa logikoarekin ondorengo maskara aplikatuko diot: 0x00200000 (Pinaren bit txikiena batekoa)
3. Begizta nagusian, `GPSET` eta `GPCLR` erregistroetako 17. bitean batekoa jarri.

Kodea

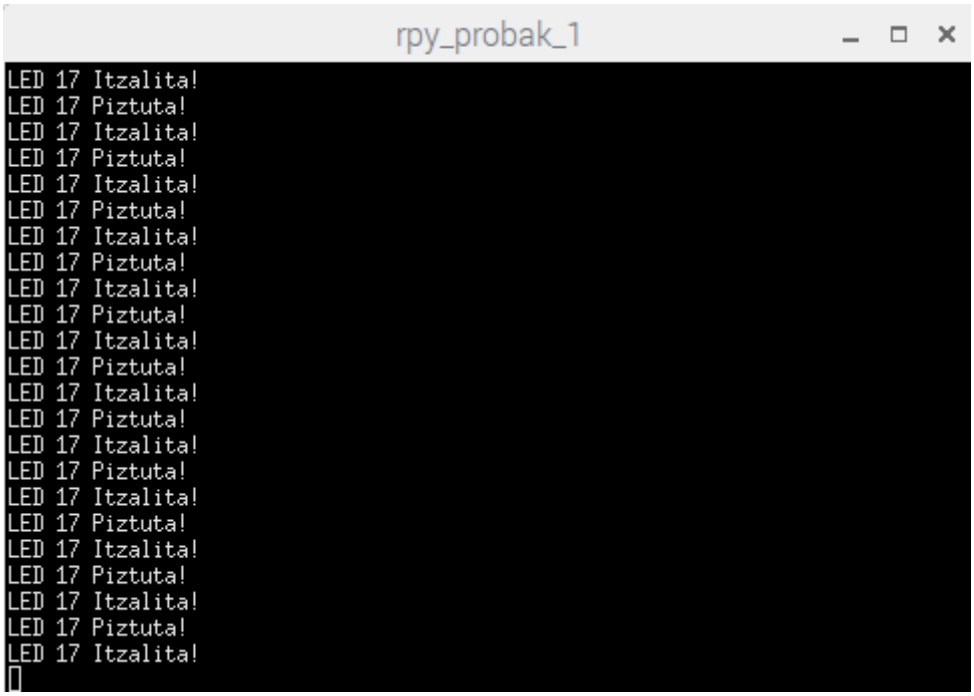
4.3. Kodean orain azaldutako aginduak kodetuta agertzen dira:

```
#include "main.h"



int main(int argc, char **argv)
{
 setup_io();   //Konfigurazio funtzioa
 int maskara,maskara2,maskara3;

 //GPIO 17 (11 fisikoa) OUTPUT bezala jarri
 maskara = 0xFF1FFFFF; // 21-23 BITAK zerokoak direla ziurtatu (INPUT)
 maskara2 = 0x00200000; //21 bita bateko bezala jarri (OUTPUT = 001)
 *(GPFSEL1) &= maskara; //21-23 BITAK zerora mantendu, besteak zeuden
bezala mantenduz
 *(GPFSEL1) |= maskara2; //21 bita batera jarri, besteak helbidekoak
mantendu
 maskara3 = 0x00020000;   //17 bita batera jarri, besteak zerora
 while(1){
     //LEDA PIZTU
     *(GPSET0) |= maskara3;    //SET0 erregistroko 17 bita batean jarri
     printf("LED 17 Piztuta!\n");
     sleep(1);
     //LEDA ITZALI
     *(GPCLR0) |= maskara3; //CLEAR0 erregistroko 17 bita batean jarri  */
     printf("LED 17 Itzalita!\n");
     sleep(1);
 }
 return 0;
}
```

4.3. Kodea: Raspbian sistemarekin egindako lehenengo praktika

Lortutako emaitzak

Lehenengo probaren helburua Leda etengabe pizten eta itzaltzen ibiltzea da, aldi berean kontsolaren bitartez sistema Ledaren egoera inprimatzen dagoela. Kodean pizteko eta itzaltzeko tarteak segundo batekoak dira. Kodea exekutatu ostean ondorengo emaitzak lortzen ditut, 4.3. Irudian ikus daitezkeenak:



4.3. Irudia: Kontsola ledaren egoera inprimatzen

4.2.2.2 Sinkronizazioa: Inkesta

Azalpena eta kodea

GPIO portuko pinak OUTPUT bezala funtziona dezaketela ikusi da, baina, Raspberry Pi txartelako pinen tentsioa irakur daiteke ere, pin hauek INPUT modura konfiguratuta. Tentsio altua eta baxua era erraz batean pin batera eramateko gailurik arruntena pultsadorea litzake. Zirkuitua irekia dagoenean, GPLEV erregistroan pin horri dagokion bitean zerokoa egongo da, aldiz, tentsio altua jasotzen badu (zirkuitua itxia), pinari dagokion bitean batekoa egongo da. Praktika honetan erregistro hauek inkesta bidez irakurriko ditut, hau da, tentsioa aldatzen den arte hau irakurtzen egongo naiz eta tentsio aldaketa detektatzean, nahi dudan kodea exekutatuko da.

Kasu honetan, aurreko praktiketan erabilitako Ledak pultsadoreen bitartez kontrolatuko ditut (bi pultsadoren bitartez). Lehenengoak argia Ledez-Led mugituko du, eta bigarrenak programa hasi eta amaituko du.

Proba honetan erabiliko ditudan gailuak ondorengoak dira:

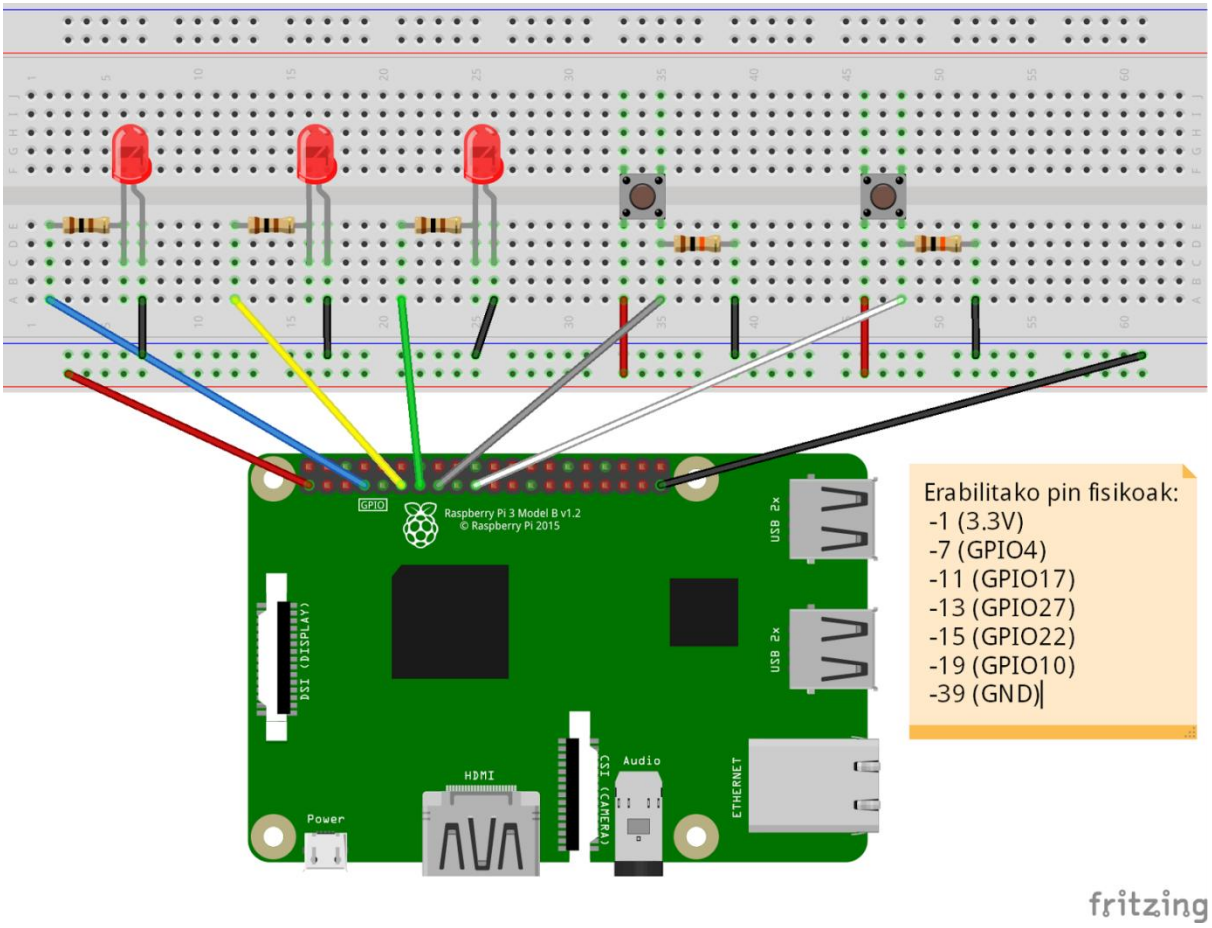
- Raspberry Pi-a
- Breadboard konexio taula
- 75 edo 100 Ohm-eko hiru erresistentzia
- 10K Ohm-eko bi erresistentzia

- Hiru Led gorri (1,9V)
- Bi pultsadore
- Interkonexio hariak

Gailu hauek interkonektatzeko zazpi pin erabiliko ditut:

- 3.3V pina (kasu honetan 1. pin fisikoa)
- GPIO 17 (11. pin fisikoa)
- GPIO 4 (7. pin fisikoa)
- GPIO 27 (13. pin fisikoa)
- GPIO 22 (15. pin fisikoa)
- GPIO 10 (19. pin fisikoa)
- GND pin bat (Kasu honetan 39. pin fisikoa)

4.4. Irudian ageri dira gailuen arteko konexioak.



4.4. Irudia: Hirugarren probaren eskema

Zirkuitu elektronikoari dagokionez eta aurreko praktikekin konparatuz, bi INPUT pin gehitu dira pultsadoreekin jolasteko. Erabilitako zirkuitua Pull-Down erresistentzia motakoa da, hau da, pultsadorea pultsatu gabe dagoenean, pinak jasotzen duen balio logikoa zerokoa da, aldiz pultsatzen dugun momentuan 3,3 voltioak jasotzen ditu, balio logikoa batekoa izanik.

Alde elektronikoa alde batera utzita, programaren funtzionamendua ondorengoa litzake:

1. Programak beti inkesta bidezko sinkronizazioa erabiliko du balioak detektatzeko. Kodean dagoen bezala, GPIO10 pinak programaren hasiera eta amaiera markatuko du. Hau pultsatzerakoan programa hasiko da.
2. Bi aukera daude:
   a. GPIO 22 pinera konektatutako pultsadorea pultsatuta argia Ledez-Led mugitzen joango da prozeduraren patroia jarraituz.
   b. GPIO 10 pinera konektatutako pultsadorea pultsatuta, programa amaituko da.

Oharra: Pultsadoreak gailu mekanikoak dira eta denbora tarte bat behar dute bere balioa egonkortzeko (0,2 segundu gutxi gora behera).

Kodea

4.4. Kodean orain azaldutako aginduak kodetuta agertzen dira.

```
#include "main.h"

#define GET_GPIO(g) (*(GPLEV0)&(1<<g)) // 0 if LOW, (1<<g) if HIGH

void input(int pina){
   int lag;
   lag = ~(7<<(((pina)%10)*3)); //111 desplazatu dagokion pineko posiziora eta
gero bitez biteko ezeztatzea  aplikatu (INPUT = 000)
   *(gpio +(pina/10)) &= lag; //AND eragiketa bi helbideen artean (erregistro
egokia eta )
}

void output(int pina){
   int lag;
   lag = (1<<(((pina)%10)*3)); //batekoa desplazatu dagokion pineko bit
txikienaren posiziora (OUTPUT = 001)
   *(gpio +(pina/10)) |= lag; //OR eragiketa bi helbideen artean
}

void set(int pina){
   *(GPSET0) = 1<<pina;  //desplazatu piztu nahi dugun pinera bateko bat.
}

void clear(int pina){
   *(GPCLR0) = 1<<pina;  //desplazatu itzali nahi dugun pinera bateko bat.
}



int main(int argc, char **argv)
{
 // Set up gpi pointer for direct register access
 setup_io();
 //Ledak
 input(27);
 output(27);
 input(17);
 output(17);
 input(4);
 output(4);
```

```
//pultsadoreak
 input(22);
 input(10);
printf("Programa hasteko itxaroten!\n");
 //Programa hasteko pultsadore egokia pultsatu eta utzi pultsatzen
 while(GET_GPIO(10)==0); // INKESTA
 usleep(200000);   //arrazoi mekanikoak
 while(GET_GPIO(10)!=0); // INKESTA
 set(4);
 printf("Pultsadoreekin jolastu!\n");

while(1){
      // BETI INKESTAZ GALDETU EA PULTSATU DUGUN ALA EZ.
       if (GET_GPIO(10)!=0){   //Programa amaitu
           printf("Programa amaituta!\n");
           clear(4);
           clear(17);
           clear(27);
           break;
       } else {    //Ledez-Led argia aldatu
           if (GET_GPIO(22)!=0){
               usleep(200000); //arrazoi mekanikoak
               if (GET_GPIO(4)!=0){
                   set(17);
                   clear(4);
               }else if (GET_GPIO(17)!=0){
                   set(27);
                   clear(17);
               }else if (GET_GPIO(27)!=0){
                   set(4);
                   clear(27);
               }
           }
       }
 }
 return 0;
}
```

4.4. Kodea: Raspbian sistemarekin egindako bigarren praktika

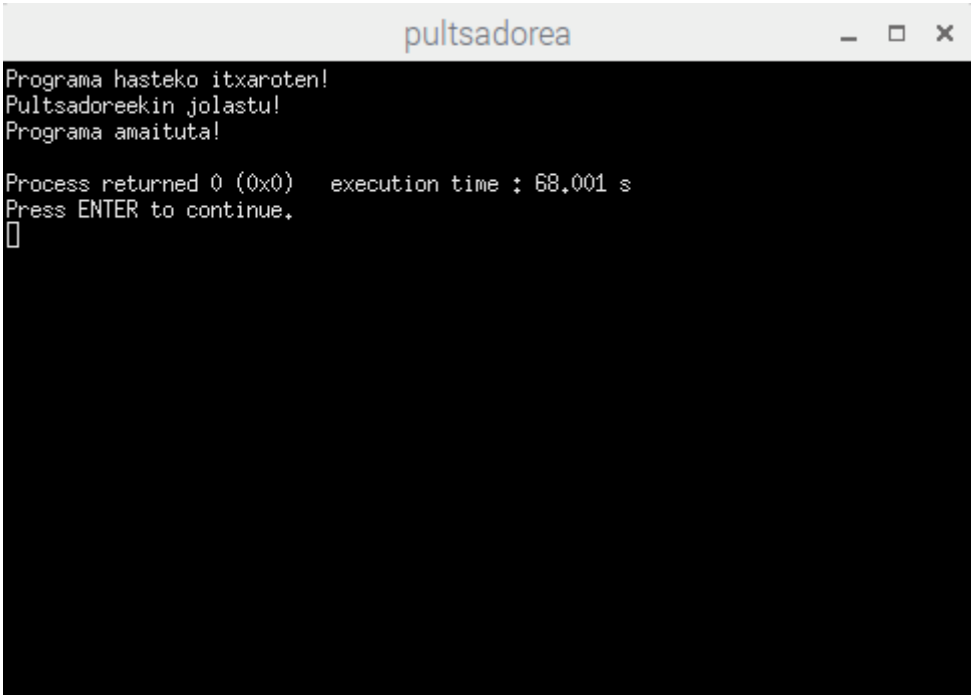
Proba honetan helburu bakarra dut:

- Eskubiko pultsadorea (GPIO 10) pultsatzen ez dudan bitartean programak ezer ez egitea.
- Behin programa hasita:
    - Ezkerreko pultsadorea pultsatuta (GPIO 22), argia Ledez-Led mugitzea.
    - Eskubiko pultsadorea pultsatuta, programa amaitzea.

Betiere sinkronizazio modu bezala inkesta erabiliz.

Lortutako emaitzak

4.5. Irudian ikus daitezke programa honetan lortutako emaitzak.



4.5. Irudia: Probaren laburpena

4.2.3 Raspbian sistemarekin izandako arazoak

Kapituluaren hasieran azaldu dudan moduan, sistema eragile batekin lan egiten denean aurkezten den limitazio nagusia hardware osoa zuzen-zuzenean atzitzen uzten ez duela da. Ez dut esango hau arazo bat denik, Sistema Eragilearen kernelean ezarritako segurtasun neurriak baitaude arrazoi honen atzetik.

Hau horrela izanda, guztiz erabakigarria izan da orain arte sistema eragileaz baliatzea memoriako helbide fisikoak atzitzeko. Linux sistema batean hau egiteko, */dev/mem* fitxategia irekitzen da eta behar dugun helbide sorta memorian mapeatzen da. Lehen aipatu bezala, sistema eragile batekin ibiltzeak gauza askotarako laguntzen du, hardwarea hondatzeko arriskuak minimizatzen dituelako. Aldiz, *Konputagailuen Egitura* ikasgaian Raspbian sistema eragileak berak maneiatzen dituen hainbat kontzeptu ikustea ondo legoke, etenak eta *timerra* esaterako.

Puntu honetara iritsita hiru aukera posible daude:

1. Linux sistema erabiltzen jarraitu eta etenak sistema eragilearen bitartez maneiatu. Aukera hau proiektu honen helburuetatik at geratzen da.
2. Linux sistemaren kernel modulua programatu zuzenean hardwarea atzitzen uzteko. Aukera hau proiektuaren helburuetatik at geratzen da ere, nahiz eta hurrengo proiektu batean aurrera eramateko ideia ona izan badaiteke ere.
3. *Bare metal* programazioa, hurrengo puntuan azalduko den aukera eta nik jorratuko dudana.

4.3. Bare metal programazioa

4.3.1 Zer da bare metal

Bare metal programazioa hardwarea zuzenean atzitzea litzake, sistema eragile baten bitarteko kontrola kenduta. Programazio modu hau sistema txertatuetan jorratzen den aukera nagusietako bat da, lan oso konkretuak bilatzen direlako.

Bare metal programazioak duen “arazoa” (berez ez da arazo bat gauzak horrela baitira sistema txertatuen munduan) hardware mota konkretu batentzat egina dagoela da, hau da, kodea hardware mota bakar batentzat egina dago. Modu honetan sistema hau helburu bakarrekoa izango da, helburu orokorrekoa izan beharrean.

4.3.2 Bare metal sistema martxan jartzen

Atal honetan sistema martxan jartzen ikasiko da, aurretik egindako programa sinple bat konpilatuz eta exekutatuz. Sistema geroago ikusiko diren hiru fitxategien bidez jartzen da martxan.

Programak Windows sistema bateko (Linuxen prozedura berdina jarrai daiteke) Notepad++ fitxategi-editorearen bitartez eraiki dira, geroago zehar-konpilazioaren bitartez Raspberry Pi-ak ulertzen duen makina lengoaira kodea itzuliz. Lehenengo programa 4.6. Irudian ikus daiteke.

etenak.c | armtimerra.c | armtimerra.h | main.c | main.c

```
#define GPFSEL4  gpio+4
#define GPFSEL5  gpio+5

#define GPSET0  gpio+7
#define GPCLR0  gpio+10

*/

//32 Biteko integerrak bezala
volatile uint32_t * const GPFSEL0 = (uint32_t *)0x20200000;
volatile uint32_t * const GPFSEL1 = (uint32_t *)0x20200004;
volatile uint32_t * const GPFSEL2 = (uint32_t *)0x20200008;
volatile uint32_t * const GPFSEL3 = (uint32_t *)0x2020000C;
volatile uint32_t * const GPFSEL4 = (uint32_t *)0x20200010;
volatile uint32_t * const GPFSEL5 = (uint32_t *)0x20200014;

volatile uint32_t * const GPSET0 = (uint32_t *)0x2020001C;

volatile uint32_t * const GPCLR0 = (uint32_t *)0x20200028;

// Erreferentzia gpio erregistroei
volatile unsigned int* gpio;

// Begista egiteko
volatile unsigned int loop;

int main(void)
{
    //GPIO erregistroen hasieran apuntatzen duen erreferentzia
    gpio = (unsigned int*)GPIO_HASIERA;

    //GPIO22 OUTPUT = GPFSEL erregistroaren 6. bitean batekoa
    *(GPFSEL2) |= (1 << 6);

    while(1)
    {
        for(loop = 0; loop < 2000000; loop++)
            ;

        //GPIO22 esarri
        *(GPSET0) |= (1 << 22);

        for(loop = 0; loop < 2000000; loop++)
            ;

        //GPIO22 itzali
        *(GPCLR0) |= (1 << 22);
    }
}
```

4.6. Irudia: Aurretik prestatutako programa notepad++ inguruan

Bare metal programazioan hasteko, buruan eduki behar den lehenengo kontzeptua zein hardwarearekin lan egingo den jakitea izango litzake, honen arabera gauzak aldatu daitezkeelako. Proba hauetan zehar Raspberry Pi B modeloa erabili da.

Atal honetako hasieran probak egiteko beharrezko materiala erabiltzeko prest utziku dut. Instalatu beharreko lehenengo gauza *gcc zehar konpiladorea* izango da. Zehar konpiladore honek C lengoaian programatutako kodea Raspberry Pi txartelak ulertu dezakeen mihiztadura lengoaian bihurtzen du.

Zehar konpiladorea jaisteko esteka: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads. Karpetaren barruan README fitxategia dago, sistema eragile bakoitzarentzat nola instalatu eta erabili era argi batean azaltzen duena.

4.5. Kodeak erakusten du GCC zehar konpiladorea instalatzeko agindua Linux sistema batean, Windows-en automatikoa baita.

4.6. Kodeak instalatuta dudan bertsioa erakusten du.

```
sudo apt-get install arm-none-eabi-gcc
```

4.5. Kodea: Zehar konpiladorea instalatzeko agindua

```
arm-none-eabi-gcc --version

erantzuna:
arm-none-eabi-gcc (15:4.7.4+svn231177-1) 4.7.4 20150529 (prerelease)
```

4.6. Kodea: GCC bertsioa ikusteko agindua

Ondoren lehenengo praktikak programatzen hasiko naiz. Gogoratu beharko da orain hardwarea eta nire artean ez dagoela inongo bitartekaririk eta sistema eragilearen barnean ez bezala, orain bai esan dezakedala defektuz GPIO erregistroen bit bakoitzaren balioa zerokoa dela. 4.7. Kodea aurretik prestutako proba sinple bat erakusten du.

```
// main.c: GPIO22 pinera konektatutako Leda pizten

//GPIO Hasierako helbidea
#define GPIO_HASIERA      0x20200000

/* Erregistroak */
#define GPFSEL0  gpio
#define GPFSEL1  gpio+1
#define GPFSEL2  gpio+2
#define GPFSEL3  gpio+3
#define GPFSEL4  gpio+4
#define GPFSEL5  gpio+5

#define GPSET0  gpio+7
#define GPCLR0  gpio+10

/* Erregistroak zuzenean aurkezteko aukera dugu ere
volatile uint32_t * const GPFSEL0 = (uint32_t *)0x20200000;
volatile uint32_t * const GPFSEL1 = (uint32_t *)0x20200004;
volatile uint32_t * const GPFSEL2 = (uint32_t *)0x20200008;
volatile uint32_t * const GPFSEL3 = (uint32_t *)0x2020000C;
volatile uint32_t * const GPFSEL4 = (uint32_t *)0x20200010;
volatile uint32_t * const GPFSEL5 = (uint32_t *)0x20200014;

volatile uint32_t * const GPSET0 = (uint32_t *)0x2020001C;

volatile uint32_t * const GPCLR0 = (uint32_t *)0x20200028;
*/

// Erreferentzia gpio erregistroei
volatile unsigned int* gpio;

// Begizta egiteko
volatile unsigned int loop;

int main(void)
{
	//GPIO erregistroen hasierara erakusten duen erreferentzia
	gpio = (unsigned int*)GPIO_HASIERA;

	//GPIO22 OUTPUT = GPFSEL erregistroaren 6. bitean batekoa
	*(GPFSEL2) |= (1 << 6);

	while(1)
	{
	for(loop = 0; loop < 2000000; loop++)
		;

	//GPIO22 ezarri
	*(GPSET0) |= (1 << 22);

	for(loop = 0; loop < 2000000; loop++)
		;

	//GPIO22 itzali
	*(GPCLR0) |= (1 << 22);
	}
}
```

4.7. Kodea: Bare metal lehenengo praktika

4.7. Kodean ikusten den diferentzia nabarmenena sistema eragilearekiko independentzia da. Raspbian sistemarekin RAM memoria atzitzeko sistema eragilearen fitxategi sistemaren beharra zegoen eta kasu honetan ez. GPIO erregistroak atzitzeko aldiz, Raspbian sistema eragilearekin egiten zen prozedura berdina jarraitzen da, behin hardwarea atzitzeko baimenak lortuta erregistroak zuzenean atzitzen direlako.

Kode hau konpilatzeko, Raspberry Pi-arentzako konpilazio marka egokiak ipini beharko dira, 4.8. Kodean ikusten den moduan. Konpilazio marka hauek "[11] Konpilazio markak" erreferentziatik hartu ditut.

Honek *kernel.elf* exekutagarria sortuko du, baina Raspberry Pi-an nire programa kargatzeko *kernel.img* exekutagarria sortu beharko dut Rasperry Pi-ak ulertzen duen makina lengoaian idatzita egongo dena. 4.9. Kodearen bitartez exekutagarria sortuko dut.

Konpilazioan warning bat jasoko dut: linkerrak *_start* etiqueta (linkerrak gure kodea exekutatzeko beharrezkoa duena) aurkitzen ez duela. Kodearen inguruko memoria helbideak ikusteko, 4.10. Kodeko sarrera agindua erabiliko dut.

```
arm-none-eabi-gcc -O0 -mfpu=vfp -mfloat-abi=hard -march=armv6zk -
mtune=arm1176jzf-s -nostartfiles -g main.c -o kernel.elf
```

4.8. Kodea: Bare metal praktika konpilatzeko agindua

```
arm-none-eabi-objcopy kernel.elf -O binary kernel.img
```

4.9. Kodea: Bare metal praktikako exekutagarria sortzeko agindua

```
Sarrera: arm-none-eabi-nm kernel.elf

Irteera:

… (helbide gehiago oraingoz interesik gabekoak)
00008000 T main
00080000 N _stack
         U _start
```

4.10. Kodea: Bare metal praktikako memoria ikusteko agindua

Raspberry Pi-ak defektuz 0x8000 helbidean kargatzen du kernela, beraz, kodea helbide horretan hasi beharko da, Raspbian sistemaren kernela ordezkatuz. 4.10. Kodean ikusten denez, Raspberry Pi-a piztu eta ondoren main funtzioa exekutatzen hasten da, honen ondorioz, programa kargatzen du. Programa konplexuagoak egiterako orduan aldiz, ez da programaren funtzionamendua bermatzen.

Banoa beraz *warning*-a konpontzera. *_start* etiketaren papera kodeak funtzionatzeko beharrezkoak diren hainbat sekzioren karga egitea da. Kargatzen dituen sekzioen artean *bss* sekzioa dago, non aldagaiak inplizituki zerora jartzeaz arduratzen den (definitu gabeko aldagai globalak eta lokal estatikoak). Lehen aipatu bezala, programak funtzionatzeko bermea izateko, etiketa hau 0x8000 helbidean joan beharko da.

Horretarako, 4.11. Kodea *_start* etiketa bere 0x8000 helbidean ipintzeaz arduratzen da.

```
//start.c fitxategian
.section ".text.startup"
.global _start
.global _get_stack_pointer

_start:
      ldr    sp, =0x8000
      b      _cstartup

_inf_loop:
      b      _inf_loop

_get_stack_pointer:
      str    sp, [sp]
      ldr    r0, [sp]
mov    pc, lr

-------------------------------------------------------------------------------

//cstartup.c fitxategian
void _cstartup()
{
      //bss sekzioa hasieratu
      int* bss = &__bss_start__;
      int* bss_end = &__bss_end__;

      while( bss < bss_end )
      *bss++ = 0;
      //Gure kodea exekutatu (honera ez garela bueltatzen suposatzen da...)
      main();
      // Bueltatu ezkero
      while(1)
      {
      // Ezer ez egin
      }
}
```

4.11. Kodea: Bare metal praktikako mihiztadura fitxategia

4.11. Kodean egiten dena ondorengoa da: Mihiztadura lengoaiaren bitartez *.text.startup* sekzioa editatzen da, (Raspberry Pi-ak kargatzen duen lehenengo sekzioa) non _Start etiketa 0x8000 helbidean kargatzen den. Hau egin ostean C funtzio bati deitzen zaio honek aldagaien sekzioak kargatzeko. Kasu honetan eta lehen aipatu bezala, bss sekzioa kargatzen da.

Hau dena egin eta gero, kodea lehen bezala exekutatzen hasiko da, inongo *warning*-ik gabe, *_start* etiketa dagokion memoria posizioan ipini dudalako, 4.12. Kodean ikusten den bezala.

```
Sarrera: arm-none-eabi-nm kernel.elf

Irteera:

… (helbide gehiago oraingoz interesik gabekoak)
00008018 T main
...
00008000 T _start
...
```

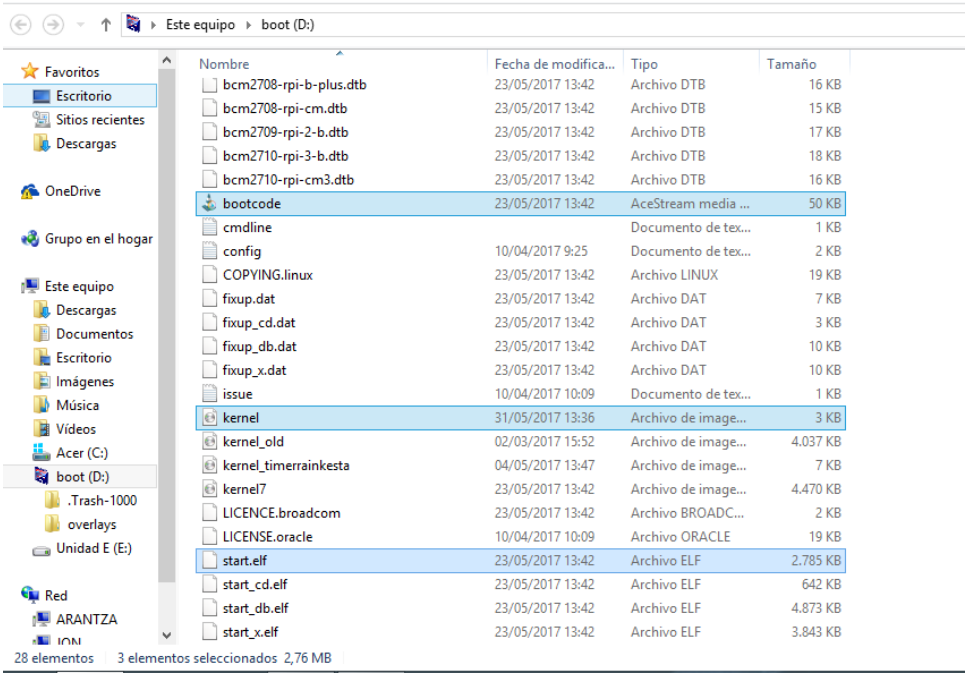
4.12. Kodea: Bare metal praktikako memoria ikusteko agindua (2)

Beraz, memorian kargatzen diren aginduen berrantolaketa bat izan da egin dudan gauza bakarra. Orain Raspberry Pi-ak programa exekutatu aurretik beharrezko etiketak eta sekzioak kargatzen ditu, funtzionamendua edozein kasutan bermatuz.

Raspberry-an programak martxan jartzeko, zehar konpiladoreak konpilatutako *Kernel.img* (*Kernel7.img* Raspberry berriagoetan) exekutagarria SD txartelean sartuko dut. Fitxategi honekin batera bi fitxategi gehiago beharko ditut SD txartelean, Raspberry Pi-aren firmware bertsio berritik aterata:

- Bootcode.bin
- Start.elf

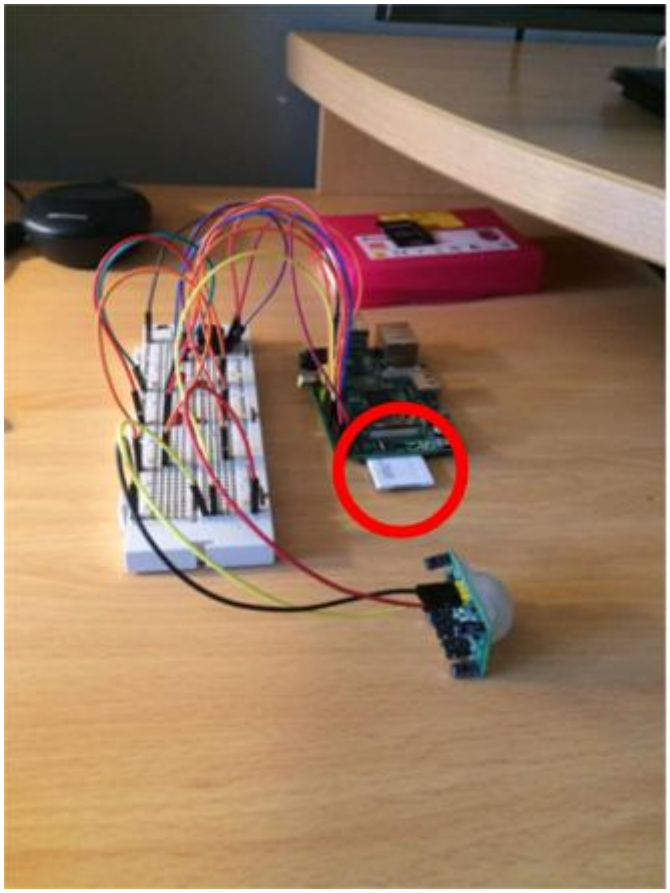
Fitxategi hauek "[12] Raspberry Pi-aren firmware berria" erreferentziatik hartu daitezke.



4.7. Irudia: SD txartela prestatua

4.7. Irudian hiru fitxategi hauek baino fitxategi gehiago ikusten dira, hau Raspbian sistema eragilea SD txartelean sartuta dudalako da, baina hiru fitxategi horiekin programak martxan jar daitezke arazorik gabe.

Amaitzeko SD txartela Raspberry Pi-an sartuko dut, 4.8. Irudian agertzen den moduan, programa has dadin.



4.8. Irudia: Raspberry Pi-a exekutatzeko prestatua

4.3.3 Probak

4.3.3.1 Sistemaren timerraren erabilpena

Azalpena eta kodea

Praktika honetan sistemaren *timerra* erabiltzen ikasiko da, *Delay* funtzioa sortzeko. Periferiko guztien erregistroak memoria helbide jarraietan ageri dira, horrek programatzerako orduan abantaila emango dit, periferiko bakoitzak dituen erregistroekin datu egitura bat sor dezakedalako. Horrela programatu dut 4.13. Kodean

```
/*
   timerra.h fitxategia:
*/
#include <stdint.h>

#define TIMERRA_HASIERA       0x20003000

// Periferikoaren erregistroak atzitzeko, datu egitura moduan.
typedef struct {
   volatile uint32_t control_status;
   volatile uint32_t counter_lo;
   volatile uint32_t counter_hi;
   volatile uint32_t compare0;
   volatile uint32_t compare1;
   volatile uint32_t compare2;
   volatile uint32_t compare3;
   } rpi_sys_timer_t;

extern rpi_sys_timer_t* RPI_GetSystemTimer(void);
extern void timerDelay( uint32_t us );

------------------------------------------------------------------------
/*
   timerra.c fitxategia:
*/
#include <stdint.h>
#include "timerra.h"

//Timerraren hasierako helbideari erreferentzia egiten dion erakuslea
static rpi_sys_timer_t* rpiSystemTimer =
(rpi_sys_timer_t*)TIMERRA_HASIERA;

//Getter funtzioa
rpi_sys_timer_t* RPI_GetSystemTimer(void)
{
   return rpiSystemTimer;
}

void timerDelay( uint32_t us )
{
   volatile uint32_t ts = rpiSystemTimer->counter_lo;

   while( ( rpiSystemTimer->counter_lo - ts ) < us )
   {
       //Ezer ez egin
   }
}
```

4.13. Kodea: Sistemaren timerra programatzeko fitxategiak

4.13. Kodean ikusten denez, `rpi_sys_timer_t` datu egituran *timerraren* erregistroak gordetzen ditut. Luzera definituta (32 bitekoak guztiak) dagoenez, atzipena zuzenekoa da. Ondoren datu mota honetako `rpiSystemTimer` erakuslea definitzen dut erregistroen hasierari erreferentzia eginez, datu egiturako aldagai guztiak dagokion erregistro parearekin lotuz.

Aurretik aipatu dut *timer* honekin ez dudala inongo etenik erabiliko. Horren ordez, sistemaren *timerra Delay* funtzio bat inplementatzeko erabiliko dut, *`timerDelay`* deiturikoa. Funtzio honen bitartez proiektuetan sartzen ditudan pultsadore mekanikoei egonkortzeko denbora tarte bat eskainiko diet.

4.14. Kodea sistemaren *timerra* probatzen duen programa da, segunduro bi GPIO pizten eta itzaltzen ibiliko dena.

```
/*
   main.c fitxategia:
*/

#include <string.h>
#include <stdlib.h>

#include "gpio.h"
#include "timerra.h"

void main()
{

   // GPIO16,GPIO22 OUTPUT bezala jarri
   *GPFSEL1 |= ( 1 << 18);
   *GPFSEL2 |= ( 1 << 6);

   while(1)
   {

       // GPIO16 pina tentsio baxuan jarri eta GPIO22 altuan
       *GPSET0 = (1 << 16); // GPIO16
       *GPSET0 = (1 << 22); // GPIO22

       //Delay timerra erabilita (mikrosegundotan)
       timerDelay( 1000000 );

       // GPIO 16 pina tentsio altuan jarri eta GPIO22 baxuan
       *GPCLR0 = (1 << 16); // GPIO16
       *GPCLR0 = (1 << 22); // GPIO16

       //Delay timerra erabilita (mikrosegundotan)
       timerDelay( 1000000 );
   }
}
```

4.14. Kodea: Bare metal praktikako probako programa nagusia

Azkeneko praktikan periferiko gehiago erabiliko direnez, periferiko horien guztien erregistroak orain azaldu ditudan moduan programatuko ditut (datu egitura baten bitartez). Orain arte erregistroak atzitzeko programatu daitezkeen aukera dexente ikusi dira modu bakar bat baina gehiagotan egin daitekeela erakusteko.

4.3.3.2 Etenen kudeaketa

Azalpena eta kodea

Bare metal programazioaren lehenengo pausoak eta periferikoak erazagutzeko modu bat ikusita, etenak nola kudeatzen diren aztertu nahi da orain, GPIO zein ARM *timerretik* sortutako etenak tratatuz.

Praktika honetan ondorengoa egingo da:

- Inkesta bidez pultsadore bat sakatzeari itxaron programa hasteko.
- Behin pultsatuta, ARM *timerrak* segunduro etengo du. Etena jasotzean zazpi segmentuko *display*-a eguneratzen da.
- Aldi-berean, pultsadorea ukitzeko aukera izango da, ACT Leda piztu eta itzaliz eta *display*-a berrabiaraziz.

Programan egin beharreko lehenengo gauza eten bektorea dagokion helbidean kokatzea da (0x0000). Hau era erraz batean egin daiteke mihiztadura lengoaiaren laguntzaz. Programa zati hau 4.15. Kodean ikus daiteke.

```
_start:
       ldr pc, =_reset_
       ldr pc, =undefined_instruction_vector
       ldr pc, =software_interrupt_vector
       ldr pc, =prefetch_abort_vector
       ldr pc, =_reset_
       ldr pc, =_reset_
       ldr pc, =interrupt_vector
       ldr pc, =fast_interrupt_vector

   _reset_:
       // Eten bektorea kopiatu 0x0000 helbidera
       ldr     r0, =_start
       ldr     r1, #0x0000
       ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
       stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

4.15. Kodea: Bigarren praktikako eten bektorea kokatzen

4.15. Kodean azaltzen den bezala, `ldmia` eta `stmia` aginduen bitartez, 0x0000 helbidean `_start` etiketako kodea (0x8000 helbidean karga) gordetzen da, hau da, eten bektorea 0x0000 helbidetik aurrera kokatzen da. Teorikoki honek funtzionatu beharko luke, baina aurreko probetan gertatu zaidan bezala, konpiladoreak memorian egiten duen antolakuntzaren ondorioz, ez du funtzionatzen. Aurreko kodearen arazketa 4.16. Kodean ikusten da, zehatzago ikusteko zer gertatzen ari den memorian.

```
00008000 <_start>:
       8000:   e59ff064    ldr pc, [pc, #100]  ; 806c
<_enable_interrupts+0x10>
       8004:   e59ff064    ldr pc, [pc, #100]  ; 8070
<_enable_interrupts+0x14>
       8008:   e59ff064    ldr pc, [pc, #100]  ; 8074
<_enable_interrupts+0x18>
       800c:   e59ff064    ldr pc, [pc, #100]  ; 8078
<_enable_interrupts+0x1c>
       8010:   e59ff054    ldr pc, [pc, #84]   ; 806c
<_enable_interrupts+0x10>
       8014:   e59ff050    ldr pc, [pc, #80]   ; 806c
<_enable_interrupts+0x10>
       8018:   e59ff05c    ldr pc, [pc, #92]   ; 807c
<_enable_interrupts+0x20>
       801c:   e59ff05c    ldr pc, [pc, #92]   ; 8080
<_enable_interrupts+0x24>

   00008020 <_reset_>:
       8020:   e3a00902    mov r0, #32768  ; 0x8000
```

4.16. Kodea: Bigarren praktikako arazketa

Kodearen hasieran `_reset_` errutina exekutatzea espero da, baina beste edozer exekutatzen dabil. 4.16. Irudiko lehenengo aginduak irakurtzen duen memoria helbidea ikusita (0x806c), `_reset_` errutina memoriako zein helbidetan gordetzen den azaltzen da, hain zuzen 0x00008020 helbidean, 4.17. Kodean erakusten den bezala. 8000 helbidetik aurrera dauden aginduek, 4.16 kodean ikusten direnak, 806c memoria helbidetik aurrera dagoen etiketa taula bat atzitzen dute. Taula horretan, etiketa horietako bakoitzak adierazten duen kodea memoriako zein helbidetatik aurrera gordeta dagoen azaltzen du. Hauxe izango litzateke benetan eten bektorea.

```
806c:   00008020    .word   0x00008020      //_reset_ dagoen helbidea
```

4.17. Kodea: Bigarren praktikako arazketa (2)

Beraz, PC + 0x6C desplazamendua eginda, `_reset_` funtzioa exekutatu beharko luke. Hau ez da guztiz egia, kodea 0x8000-tik (hasiera) 0x0000 helbidera mugitzen dudanean konpiladoreak ez dituelako eten bektoreko errutinen helbideak 0x806C helbidetik 0x006C helbidera mugitzen, memoriako hurrengo hutsunean uzten baititu.

Arazo honen aurrean Bryan-ek bere *bare metal* tutorialean ondorengoa proposatzen du: PCarekiko helbideratze erlatiboak funtzionatzen jarrai dezan, beste etiketa taula bat sortzen du, memorian `_start` errutinaren jarraian kokatuta egongo dena, hor definitu delako kodean. Modu honen bitartez, etiketak PC-arekiko desplazamendua egokia dela bermatuko dute eten bektore osoarentzat. Soluzio hau 4.18. Kodean erabiltzen da eta "[8] *Bare metal* programazioa" erreferentziatik dago hartuta.

```
_start:
       //etiketen taula
       ldr pc, _reset_h
       ldr pc, _undefined_instruction_vector_h
       ldr pc, _software_interrupt_vector_h
       ldr pc, _prefetch_abort_vector_h
       ldr pc, _data_abort_vector_h
       ldr pc, _unused_handler_h
       ldr pc, _interrupt_vector_h
       ldr pc, _fast_interrupt_vector_h

   _reset_h:                           .word   _reset_
   _undefined_instruction_vector_h:    .word   undefined_instruction_vector
   _software_interrupt_vector_h:       .word   software_interrupt_vector
   _prefetch_abort_vector_h:           .word   prefetch_abort_vector
   _data_abort_vector_h:               .word   data_abort_vector
   _unused_handler_h:                  .word   _reset_
   _interrupt_vector_h:                .word   interrupt_vector
   _fast_interrupt_vector_h:           .word   fast_interrupt_vector

   _reset_:
       //Orain etiketak kargatu behar ditugu, eten bektoreaz aparte…
       //Karga bikoitza
       mov     r0, #0x8000
       mov     r1, #0x0000
       ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
       stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
       ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
       stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

4.18. Kodea: Bigarren praktikako eten bektorea kokatzen (2)

4.18 Kodean aipatzen den moduan, `ldmia` eta `stmia` aginduen bitartez 0x8000 helbidetik aurrerako edukia 0x0000 helbidetik aurrera kargatzen hasten da. Kasu honetan lehenengo kargan eten bektorea kargatuko du eta bigarrenean etiketen taula. Honela, eten bektorea 0x0000 helbidetik aurrera egongo da atzigarri etenak tratatzeko.

```
// Start.s Fitxategia:


.section ".text.startup"

.global _start
.global _get_stack_pointer
.global _exception_table
.global _enable_interrupts

//Erabiltzaile moduak

.equ    CPSR_MODE_IRQ,          0x12
.equ    CPSR_MODE_SVR,          0x13

//IRQ-ak desgaitzeko bitean batekoa, edo IFQ-an berdin.
.equ    CPSR_IRQ_INHIBIT,       0x80
.equ    CPSR_FIQ_INHIBIT,       0x40
.equ    CPSR_THUMB,             0x20

//Soluzio partziala, bestela linkerrak ez du eten bektorea ondo kokatuko
memorian
_start:
   ldr pc, _reset_h
   ldr pc, _undefined_instruction_vector_h
   ldr pc, _software_interrupt_vector_h
   ldr pc, _prefetch_abort_vector_h
   ldr pc, _data_abort_vector_h
```

```
   ldr pc, _unused_handler_h
   ldr pc, _interrupt_vector_h
   ldr pc, _fast_interrupt_vector_h

_reset_h:                               .word   _reset_
_undefined_instruction_vector_h:        .word   undefined_instruction_vector
_software_interrupt_vector_h:           .word   software_interrupt_vector
_prefetch_abort_vector_h:               .word   prefetch_abort_vector
_data_abort_vector_h:                   .word   data_abort_vector
_unused_handler_h:                      .word   _reset_
_interrupt_vector_h:                    .word   interrupt_vector
_fast_interrupt_vector_h:               .word   fast_interrupt_vector
_reset_:
   mov     r0, #0x8000
   mov     r1, #0x0000
   ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
   stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
   ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
   stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}

   //eten bektorea 0x0000 helbidean ipinita
   mov r0, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
   msr cpsr, r0
   mov sp, #(63 * 1024 * 1024)
   mov r0, #(CPSR_MODE_SVR | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
   msr cpsr, r0
   mov     sp, #(64 * 1024 * 1024)
   bl      _cstartup


_get_stack_pointer:
   str     sp, [sp]
   ldr     r0, [sp]
   mov     pc, lr

//etenak gaitu, bic aginduaren bitartez not 0x80 egingo da eta r0-rekin
AND eragiketa.
_enable_interrupts:
   mrs     r0, cpsr
   bic     r0, r0, #0x80
   msr     cpsr, r0

   mov     pc, lr
```

4.19. Kodea: Bigarren praktikako start.s fitxategia osatuta

ARM-ko eskuliburuan begiratuta, etenak gaitzeko erabiltzaile modua *Interrupt* modura pasa behar dela aipatzen da. Erabiltzaile modua *CPSR* erregistroan maskarak aplikatuz lortuko da aldatzea. Erregistro batzuk erabiltzaile motaren arabera aldatu egiten dira, SP (*Stack Pointer*) erregistroa esaterako. 4.19. Kodean SP erregistroa RAM memoriako toki ezberdinetan kokatzen dut erabiltzaile bakoitzaren informazioa pilatzeko.

Azkenik, etenak gaitzeko funtzioa egin da, horretarako *CPSR* erregistroko 7 bita zerokoa dela ziurtatzen dut, IRQ-ak gaitzeko (`bic` aginduaren bitartez **r0 AND (not 0x80)** eragiketa egiten dut).

Hau dena egin ostean, etenen kudeaketa nola egiten den azaltzea gelditzen da bakarrik. Programa zati hau 4.20. Kodean agertzen da.

```
/*
   etenak.c fitxategia:
*/
#include <stdint.h>
#include <stdbool.h>

#include "armtimerra.h"
#include "gpio.h"
#include "etenak.h"


static rpi_irq_controller_t* rpiIRQController =
       (rpi_irq_controller_t*)ETEN_KONTROLADOREA_HASIERA;


rpi_irq_controller_t* RPI_GetIrqController( void )
{
   return rpiIRQController;
}


//Eten kudeatzailea

//Eten mota bakoitza nola kudeatuko dudan agertzen da jarraian

void __attribute__((interrupt("ABORT"))) reset_vector(void)
{

}

void __attribute__((interrupt("UNDEF"))) undefined_instruction_vector(void)
{
   while( 1 )
   {

   }
}

void __attribute__((interrupt("SWI"))) software_interrupt_vector(void)
{
   while( 1 )
   {

   }
}

void __attribute__((interrupt("ABORT"))) prefetch_abort_vector(void)
{

}

void __attribute__((interrupt("ABORT"))) data_abort_vector(void)
{

}

void __attribute__((interrupt("IRQ"))) interrupt_vector(void)
{
   static int piztuta=0;
       static int kont=0;
       static int hasiera=1;

   if (RPI_GetIrqController()->IRQ_pending_2 != 0){ //GPIO zerbitzu errutina
             timerDelay(200000);
             //etena garbitu
             RPI_GetGpio()->GPEDS0 = (1 << 4);
             garbituDisplay();
```

```
                kont=0;
                if( piztuta )
                {
                        set(16);
                        piztuta = 0;
                }
                else
                {
                        clear(16);
                        piztuta = 1;
                }
        } else { //timerraren zerbitzu errutina
                //etena garbitu
                RPI_GetArmTimer()->IRQClear = 1;
                garbituDisplay();
                switch(kont){
                        case 0: zeroDisplay(); break;
                        case 1: batDisplay(); break;
                        case 2: biDisplay(); break;
                        case 3: hiruDisplay(); break;
                        case 4: lauDisplay(); break;
                        case 5: bostDisplay(); break;
                        case 6: seiDisplay(); break;
                        case 7: zazpiDisplay(); break;
                        case 8: zortziDisplay(); break;
                        case 9: bederatziDisplay(); break;
                }
                kont++;
                if (kont >9) kont=0;
        }
}

void __attribute__((interrupt("FIQ"))) fast_interrupt_vector(void)
{

}
```

4.20. Kodea: Bigarren praktikako eten kudeatzailearen fitxategia

Kasu honetan IRQ-etatik sortutako etenak tratatuko ditut soilik, baina bi periferikoetatik jasota: GPIO zein ARM *timerretik*. Etenak bi iturritik etor daitezke, beraz GPIO-tik eten eskaera jaso den ala ez ikusiko da (`IRQ_Pending_2` erregistroaren bitartez, GPIO portuaren eten eskaera jaso den ala ez ikusten da). Erantzuna baiezkoa bada, GPIO-aren zerbitzu errutina exekutatuko da, kontrako kasuan *timerraren* zerbitzu errutina exekutatuz.

Behin eten kudeatzailea eta eten bektorea “lotuta” daudela, falta den gauza bakarra etenak gaitzea da. Etenak gaitzeko ondorengo erregistroak aldatuko ditut:

- GPIO portuaren etenak gaitu:
    - `Enable_IRQs_2` erregistroko 20 bitean batekoa ipini, GPIO-aren IRQ-ak 52 zenbakia duelako.
- *Timerraren* etenak gaitu:
    - `Enable_Basic_IRQs` erregistroko 0 bitean batekoa ipini, *timerrak* erregistro honetan 0 posizioa erreserbatua duelako.

```
/*
   main.c fitxategia:
*/

#include <string.h>
#include <stdlib.h>

#include "gpio.h"
#include "armtimerra.h"
#include "timerra.h"
#include "etenak.h"


#define GET_GPIO(pina) (RPI_GetGpio()->GPLEV0&(1<<pina)) // 0 tentsio baxua
dagoenean, 1 tentsio altuarekin


void main()
{
       //Ledak
       output(16); // OK LEDA
       //pultsadoreak
       input(4);
       //display
       output(23);
       output(25);
       output(8);
       output(9);
       output(24);
       output(22);
       output(7);

      //INPUT jarritako pultsadorea tentsio aldaketak detektatzeko gaitu
       tentsioAldaketa(4);
       //Programaren Hasiera inkestaz
       garbituDisplay();

       while(GET_GPIO(4)==0);
       timerDelay(200000); //pultsadorea egonkortzen itxaron

       //GPIO IRQ-ak gaitu
       RPI_GetIrqController()->Enable_IRQs_2 = (1 << 20); //bigarren
erregistroko 20 bita GPIO_bank[3] aktibatzen du (Pin guztiak)

       //Timerraren IRQ-a gaitu (0 bita)
   RPI_GetIrqController()->Enable_Basic_IRQs = (1 << 0);

   timerKonfigurazioa();

   //etenak gaitu (CPSR erregistroan)
   _enable_interrupts();

   while(1)
   {

   }
}
```

4.21. Kodea: Bigarren praktikako main fitxategia

4.21. Kodean ikusten denez, inkesta bidez itxaroten da pultsadorea sakatu arte, ondoren sistemaren *timerra* erabilita, *Delay* funtzioa exekutatzen da pultsadorea egonkortu arte. Beltzez markatutako lerroetan GPIO pinen zein ARM *timerraren* etenak gaitzen dira beharrezko erregistroetan batekoa ipiniz. Bukatzeko ARM *timerraren* konfigurazioa egiten da 4.22. Kodean.

```
void timerKonfigurazioa(void){
      // 1Mhz-koa izanda eta maiztasun zatitzailea 256 izanda -> 3,9 Khz ->
256 us-ro freskatu
      //Freskaketa 3900 (+- 1s emango digu)
   RPI_GetArmTimer()->Load = 0xF3C;

   /* Setup the ARM Timer */
   RPI_GetArmTimer()->Control =
           RPI_ARMTIMER_CTRL_23BIT |
           RPI_ARMTIMER_CTRL_ENABLE |
           RPI_ARMTIMER_CTRL_INT_ENABLE |
           RPI_ARMTIMER_CTRL_PRESCALE_256;
}
```

4.22. Kodea: Bigarren praktikako ARM timerraren konfigurazioa

`Control` erregistroan ARM *timerraren* etenak gaitzen dira eta maiztasun zatitzailea 256 bezala ipintzen da. Honen ondorioz, maiztasuna 3,9Khz-koa izango da. Segunduro zazpi segmentuko *display*-a eguneratu nahi badugu, 3900 zenbakia kargatu beharko dugu `Load` erregistroan.

Aurreko praktikan aipatu bezala, periferikoen erregistroak memorian datu egitura bat balira bezala erazagutu ditut, bata bestearen aldamenean baitaude. 4.23., 4.24., 4.25. eta 4.26. Kodeetan periferikoekin lotutako periferikoen kodea agertzen da.

GPIO periferikoa

```
/*
   gpio.h goiburu fitxategia:
*/

#include <stdint.h>
//GPIO Hasiera definitu
#define RPI_GPIO_HASIERA       ( 0x20000000UL + 0x200000UL )

typedef struct {
   volatile uint32_t    GPFSEL0;
   volatile uint32_t    GPFSEL1;
   volatile uint32_t    GPFSEL2;
   volatile uint32_t    GPFSEL3;
   volatile uint32_t    GPFSEL4;
   volatile uint32_t    GPFSEL5;
   volatile uint32_t    Reserved0;
   volatile uint32_t   GPSET0;
   volatile uint32_t   GPSET1;
   volatile uint32_t    Reserved1;
   volatile uint32_t   GPCLR0;
   volatile uint32_t   GPCLR1;
   volatile uint32_t    Reserved2;
   volatile uint32_t   GPLEV0;
   volatile uint32_t   GPLEV1;
   volatile uint32_t    Reserved3;
   volatile uint32_t   GPEDS0;
   volatile uint32_t   GPEDS1;
   volatile uint32_t    Reserved4;
   volatile uint32_t   GPREN0;
   volatile uint32_t   GPREN1;
   volatile uint32_t    Reserved5;
   volatile uint32_t   GPFEN0;
   volatile uint32_t   GPFEN1;
   volatile uint32_t    Reserved6;
   volatile uint32_t   GPHEN0;
   volatile uint32_t   GPHEN1;
   volatile uint32_t    Reserved7;
   volatile uint32_t   GPLEN0;
   volatile uint32_t   GPLEN1;
   volatile uint32_t    Reserved8;
   volatile uint32_t   GPAREN0;
   volatile uint32_t   GPAREN1;
   volatile uint32_t    Reserved9;
   volatile uint32_t   GPAFEN0;
   volatile uint32_t   GPAFEN1;
   volatile uint32_t    Reserved10;
   volatile uint32_t   GPPUD;
   volatile uint32_t   GPPUDCLK0;
   volatile uint32_t   GPPUDCLK1;
   volatile uint32_t    Reserved11;
} rpi_gpio_t;
```

```
/*
   gpio.c fitxategia:
*/

#include <stdint.h>
#include "gpio.h"

static rpi_gpio_t* rpiGpio = (rpi_gpio_t*)RPI_GPIO_HASIERA;

rpi_gpio_t* RPI_GetGpio(void)
{
   return rpiGpio;
}

void RPI_GpioInit(void)
{

}

void input(int pina){
   int lag;
   lag = ~(7<<(((pina)%10)*3)); //111 desplazatu dagokion pineko posiziora eta
gero bitez biteko ezeztatzea     aplikatu (INPUT = 000)
       if (pina <= 9){
              RPI_GetGpio()->GPFSEL0 &= lag; //AND eragiketa
       } else if (pina < 9 && pina <= 19){
              RPI_GetGpio()->GPFSEL1 &= lag; //AND eragiketa
       } else if (pina > 19){
              RPI_GetGpio()->GPFSEL2 &= lag; //AND eragiketa
       }
}

void output(int pina){
   int lag;
   lag = (1<<(((pina)%10)*3)); //batekoa desplazatu dagokion pineko bit
txikienaren posiziora (OUTPUT = 001)
   if (pina <= 9){
              RPI_GetGpio()->GPFSEL0 |= lag; //AND eragiketa
       } else if (pina < 9 && pina <= 19){
              RPI_GetGpio()->GPFSEL1 |= lag; //AND eragiketa
       } else if (pina > 19){
              RPI_GetGpio()->GPFSEL2 |= lag; //AND eragiketa
       }
}

void set(int pina){
   RPI_GetGpio()->GPSET0 = (1<<pina);  //desplazatu aktibatu nahi dugun pinera
bateko bat.
}

void clear(int pina){
   RPI_GetGpio()->GPCLR0 = (1<<pina);  //desplazatu desaktibatu nahi dugun pinera
bateko bat.
}

void tentsioAldaketa(int pina){
       RPI_GetGpio()->GPAREN0 = (1 << pina);
       RPI_GetGpio()->GPAFEN0 = (1 << pina);
}
```

```
void garbituDisplay(void){
       set(23);
       set(25);
       set(8);
       set(9);
       set(24);
       set(22);
       set(7);
}

void piztuDisplay(void){
       clear(23);
       clear(25);
       clear(8);
       clear(9);
       clear(24);
       clear(22);
       clear(7);
}

void zeroDisplay(void){
       clear(22);
       clear(7);
       clear(8);
       clear(23);
       clear(25);
       clear(24);
}

void batDisplay(void){
       clear(7);
       clear(8);
}

void biDisplay(void){
       clear(22);
       clear(7);
       clear(9);
       clear(25);
       clear(23);
}

void hiruDisplay(void){
       clear(22);
       clear(7);
       clear(9);
       clear(8);
       clear(23);
}

void lauDisplay(void){
       clear(24);
       clear(7);
       clear(9);
       clear(8);
}

void bostDisplay(void){
       clear(22);
       clear(24);
       clear(9);
       clear(8);
       clear(23);
}
```

```
void seiDisplay(void){
	clear(22);
	clear(24);
	clear(9);
	clear(8);
	clear(23);
	clear(25);
}

void zazpiDisplay(void){
	clear(22);
	clear(7);
	clear(8);
}

void zortziDisplay(void){
	clear(22);
	clear(7);
	clear(8);
	clear(23);
	clear(25);
	clear(24);
	clear(9);
}

void bederatziDisplay(void){
	clear(22);
	clear(7);
	clear(8);
	clear(9);
	clear(24);
}
```

4.23. Kodea: Bigarren praktikako GPIO periferikoaren fitxategiak

ARM timerra

```
/*
   armtimerra.h goiburu fitxategia:
*/

#include <stdint.h>

//ARM mikrokontroladorearen 14 sekziotik aterata definizio guztiak

#define ARMTIMERRA_HASIERA               ( 0x20000000 + 0xB400 )
#define RPI_ARMTIMER_CTRL_23BIT         ( 1 << 1 )
//Prescaler aukerak
#define RPI_ARMTIMER_CTRL_PRESCALE_1    ( 0 << 2 )
#define RPI_ARMTIMER_CTRL_PRESCALE_16   ( 1 << 2 )
#define RPI_ARMTIMER_CTRL_PRESCALE_256  ( 2 << 2 )
#define RPI_ARMTIMER_CTRL_INT_ENABLE    ( 1 << 5 )
#define RPI_ARMTIMER_CTRL_INT_DISABLE   ( 0 << 5 )
#define RPI_ARMTIMER_CTRL_ENABLE        ( 1 << 7 )
#define RPI_ARMTIMER_CTRL_DISABLE       ( 0 << 7 )


//ARM timerraren erregistroak
typedef struct {
   volatile uint32_t Load;
   volatile uint32_t Value;
   volatile uint32_t Control;
   volatile uint32_t IRQClear;
   volatile uint32_t RAWIRQ;
   volatile uint32_t MaskedIRQ;
   volatile uint32_t Reload;
   volatile uint32_t PreDivider;
   volatile uint32_t FreeRunningCounter;

   } rpi_arm_timer_t;
```

```
/*
   armtimerra.c fitxategia:
*/
#include <stdint.h>
#include "armtimerra.h"

static rpi_arm_timer_t* rpiArmTimer = (rpi_arm_timer_t*)ARMTIMERRA_HASIERA;

rpi_arm_timer_t* RPI_GetArmTimer(void)
{
   return rpiArmTimer;
}

void RPI_ArmTimerInit(void)
{

}

void timerKonfigurazioa(void){
   RPI_GetArmTimer()->Load = 0xF3C;

   /* Setup the ARM Timer */
   RPI_GetArmTimer()->Control =
           RPI_ARMTIMER_CTRL_23BIT |
           RPI_ARMTIMER_CTRL_ENABLE |
           RPI_ARMTIMER_CTRL_INT_ENABLE |
           RPI_ARMTIMER_CTRL_PRESCALE_256;
}
```

4.24. Kodea: Bigarren praktikako ARM timerraren fitxategiak

Sistemaren timerra

```
/*
   timerra.h fitxategia:
*/

#include <stdint.h>

#define TIMERRA_HASIERA       0x20003000


typedef struct {
   volatile uint32_t control_status;
   volatile uint32_t counter_lo;
   volatile uint32_t counter_hi;
   volatile uint32_t compare0;
   volatile uint32_t compare1;
   volatile uint32_t compare2;
   volatile uint32_t compare3;
   } rpi_sys_timer_t;


extern rpi_sys_timer_t* RPI_GetSystemTimer(void);
extern void timerDelay( uint32_t us );

-------------------------------------------------------------------------------

/*
   timerra.c fitxategia:
*/

#include <stdint.h>
#include "timerra.h"

//Timerraren hasierako helbideari erreferentzia
static rpi_sys_timer_t* rpiSystemTimer = (rpi_sys_timer_t*)TIMERRA_HASIERA;

rpi_sys_timer_t* RPI_GetSystemTimer(void)
{
   return rpiSystemTimer;
}

void timerDelay( uint32_t us )
{
   volatile uint32_t ts = rpiSystemTimer->counter_lo;

   while( ( rpiSystemTimer->counter_lo - ts ) < us )
   {
       //Ezer ez egin
   }
}
```

4.25. Kodea: Bigarren praktikako sistemaren timerraren fitxategiak

Eten kudeatzailearen goiburukoa

```
/*
   etenak.h goiburu fitxategia:
*/
#include <stdint.h>

#define ETEN_KONTROLADOREA_HASIERA    ( 0x20000000 + 0xB200 )

//ARM mikrokontroladorearen 7.5 sekziotik aterata
#define RPI_BASIC_ARM_TIMER_IRQ          (1 << 0)

//IRQ kontroladorearen erregistroak
typedef struct {
   volatile uint32_t IRQ_basic_pending;
   volatile uint32_t IRQ_pending_1;
   volatile uint32_t IRQ_pending_2;
   volatile uint32_t FIQ_control;
   volatile uint32_t Enable_IRQs_1;
   volatile uint32_t Enable_IRQs_2;
   volatile uint32_t Enable_Basic_IRQs;
   volatile uint32_t Disable_IRQs_1;
   volatile uint32_t Disable_IRQs_2;
   volatile uint32_t Disable_Basic_IRQs;
   } rpi_irq_controller_t;


extern rpi_irq_controller_t* RPI_GetIrqController( void );
```

4.26. Kodea: Bigarren praktikako eten kudeatzailearen goiburu fitxategia

Lortutako emaitzak

Praktika honen emaitza bideo baten bitartez erakutsiko da, Youtube plataforman igota dagoena, ondoko helbidean:

- https://www.youtube.com/watch?v=HTxUP4DWhB8

Proba honen antzekoa den beste praktika bat garatu dut ere, kasu honetan, *display*-a erabili beharrean mugimendu sentsore bat erabilita. Emaitzak esteka honetan daude atzigarri:

- https://www.youtube.com/watch?v=TAF2EMLyTv0

5

Proiektuaren ondorioak

Kapitulu honetan proiektuan zehar ateratako ondorioak azalduko dira. Hasieran proiektuan zehar ateratako ondorioak erakutsiko dira. Ondoren, proiektuan honetatik habiatuta egon daitezkeen hobekuntzak.

5.1. Ikaslearen ondorioak

Alde batetik, *Konputagailuen Egitura* ikasgaiko lehenengo partea Rasperry Pi-a erabiltzeko egokitzeari dagokionez, ARM mihiztadura lengoaian idatzitako programen egitura aldatzea nahikoa dela ikusi da. Aldaketa hau behar izatearen arrazoia ondorengoa da: memoria atzitzen duten aginduetan helbideratze modu absolutua erabili ezin izana. Hala eta guztiz ere, aldaketa hauek ez dute ondorio larririk suposatu, Raspberry Pi txartelaren bitartez ARM prozesadoreko erregistroak atzitu ahal izan direlako, Nintendo DS-an egin daitekeen modu berean.

ARM programazio lengoaian idatzitako programen exekuzioa aztertu ahal izateko berriz, *Codeblocks* programazio ingurunea oso baliagarria izan da. Izan ere, memoria zein erregistroen balioak ikusteko aukera eskaintzen ditu. Hori dela eta, lehen Nintendo DS-an erabiltzen ziren tresnak ordezkatzeko kontuan izan daitekeen baliabidea dela ondorioztatu da.

Sarrera / Irteerako atalari dagokionez, Raspbian sistema eta *Codeblocks* programazio ingurunea erabilita, hardwarea atzitzeko aukera zegoela ikusi zen. Hauen laguntzaz, GPIO portuko erregistroen balioak aldatu izan dira. Baina segurtasun arrazoiak direla eta, zenbait memoria helbide ez daude atzigarri sistema eragilea bitartean denean, denboragailuen eta eten kudeatzailearen memoria espazioa, hain zuzen. Hau horrela izanda, ezin dira etenak zuzenean kudeatu eta ikasleek ezin dituzte beraien programetan zerbitzu errutinak idatzi.

Arazo honi aurre egiteko, sistema eragilerik gabe programatzea erabaki da; hau da, *bare metal* programazioa landu da. Modu honen bitartez, Raspberry Pi txartelak ikasleen praktikak exekuzioaren hasieran kargatzen ditu, Raspbian sistema eragilea kargatu beharrean. Gainera, hardware osoa atzitzeko aukera ematen du, etenak eta periferikoen erregistroak zuzenean kudeatzea posible egiten duena.

Praktika guztiak egin eta gero, gailu elektronikoen bitartez zirkuitu konplexuak sortu dira. Nintendo DS-ak barneratuta duen interfaze grafikoak aldiz, jokuak sortzea ahalbidetzen ditu, ikasleentzat proiektu atseginagoak izan daitezkeenak. Praktika ezberdinak izan arren, bai Nintendo DS-arekin, bai Raspberry Pi txartelarekin kontzeptu teoriko berdinak lantzeko aukera dagoela ikusi da. Beraz, proiektu honen bitartez bi gailuak osagarriak direla ondorioztatu da.

5.2. Proiektuaren hobekuntzak

Proiektua garatu ostean, hau hobetzeko aukera dagoela ikusi dut. Hobekuntzen artean ondorengoak zerrendatuko ditut:

- Proiektuaren ikusgarritasuna handitu dezaketen gailu elektroniko gehiago sartzea, LCD pantaila esaterako.
- Proiektu honetatik habiatuta, prozesadore grafikoa probatzea (GPUa).
- Interfaze gehiago erabiltzea, DMA edo TX/RX serie lerroa esaterako.

Bibliografia

[1] Raspberry Pi-a https://en.wikipedia.org/wiki/Raspberry_Pi#Processor

[2] ARM arkitektura https://en.wikipedia.org/wiki/ARM_architecture

[3] CodeBlocks ARM mihiztadura lengoaiarentzat http://www.microdigitaled.com/ARM/ASM_ARM/Software/ARM_Assembly_Programming_Using_Raspberry_Pi_GUI.pdf

[4] ARM Raspberry Pi-an ikasteko tutoriala http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/

[5] Raspberry Pi-aren periferikoen informazioa https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[6] GPIO portuaren inguruko informazioa http://comohacer.eu/gpio-raspberry-pi/#Esquemas_de_los_GPIO_segun_el_modelo

[7] Programazioa Raspbian ingurunean http://www.pieter-jan.com/node/15

https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[8] *Bare metal* programazioa http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/

[9] Elektronika https://www.youtube.com/watch?v=DLI-0vXDGyQ&list=PLnwu2s7SIakSnivM8_Vt3PchE3VN38GRY

[10] Raspbian instalatzen https://www.youtube.com/watch?v=GJDIgS8nres

[11] Konpilazio markak http://elinux.org/RPi_Software#ARM

[12] Raspberry Pi-aren firmware berria https://github.com/raspberrypi/firmware/tree/master/boot

A Eranskina: Programatzerako orduan erabilitako funtzioak

Proiektuan zehar maskarak behin eta berriz kalkulatzen egon beharrean, hauek zuzenean nahi den GPIO pinerako aplikatzeko funtzioak eta makroak sortu ditut. A.1. Kodean sortutako makroek pin jakin bat OUTPUT edo INPUT moduan jartzeko balio dute.

```
//INPUT
#define INP_GPIO(pina) *(gpio+((pina)/10)) &= ~(7<<(((pina)%10)*3))
//OUTPUT
#define OUT_GPIO(pina) *(gpio+((pina)/10)) |=  (1<<(((pina)%10)*3))
```

A.1. Kodea: INPUT eta OUTPUT makroak

Makro hauetan ondorengoak egiten dira:

- *`*(gpio+((pina)/10))`*: Funtzio **erregistro egokia aukeratzeko** pinarentzat. Erregistro hauetako bakoitzak 10 pinen informazioa kodetzen duenez, zatiketa zati hamar egingo da. Adibidea: g = 17. pina -> emaitza = `*(gpio+1)` erregistroan aldaketak egingo ditugu.
- `~(7<<(((pina)%10)*3))`: Dagokion erregistroan pina INPUT bezala jartzeko. Pin bakoitzak hiru bit hartzen dituenez, bider hiru egiten da bit egokiak konfiguratzeko. Ondoren zazpi zenbakia bitarrez ipintzen da (111), aurretik kalkulatu dudan posiziora desplazatuz (<<) eta azkenik bitez bit ezeztatzen dut ~ eragiketaren bidez, dagozkion bitetan 000 geldituz, hau da, pin hori INPUT izango da.
- `(1<<(((pina)%10)*3))`: Dagokion erregistroan pina OUTPUT bezala jartzeko. Pin bakoitzak hiru bit hartzen dituenez, bider hiru egiten da bit egokiak konfiguratzeko. Ondoren bat zenbaki bitarra desplazatzen da (<<) aurretik kalkulatu dudan posizio kopuruaren arabera.

Makro hauetatik abiatuz, ondorengo funtzioak sortu ditut eragiketa berdinak burutzeko, A.2. Kodean ikusten den moduan.

```
void input(int pina){
   int lag;
   lag = ~(7<<(((pina)%10)*3)); //111 desplazatu dagokion pineko posiziora
eta gero bitez biteko ezeztatzea     aplikatu (INPUT = 000)
   *(gpio +(pina/10)) &= lag; //AND eragiketa
}

void output(int pina){
   int lag;
   lag = (1<<(((pina)%10)*3)); //batekoa desplazatu dagokion pineko bit
txikienaren posiziora (OUTPUT = 001)
   *(gpio +(pina/10)) |= lag; //OR eragiketa
}
```

A.2. Kodea: INPUT eta OUTPUT funtzioak

Pin bat OUTPUT eta INPUT moduan konfiguratzeaz aparte, badaude GPIO portuko pinekin egin daitezkeen beste hainbat eragiketa. A.3. Kodeko funtzioek pin bat aktibatu edo desaktibatzeko balioko dute.

```
void set(int pina){
   *(GPSET0) = 1<<pina;  //desplazatu aktibatu nahi dugun pinera bateko bat.
}

void clear(int pina){
   *(GPCLR0) = 1<<pina;  //desplazatu desaktibatu nahi dugun pinera bateko bat.
}
```

A.3. Kodea: SET eta CLEAR funtzioak

Amaitzeko, pin jakin baten balioa irakurtzen duen makroa A.4. Kodean dago.

```
#define GET_GPIO(pina) (*(GPLEV)&(1<<pina)) // 0 tentsio baxua dagoenean, 1
tentsio altuarekin
```

A.4. Kodea: GET_GPIO makroa

B Eranskina: Elektronika birpasatzen

Atal hau eranskin bat bezala jarri dut gure proiektuaren helburuetariko bat ez delako. Aldiz, elektronikako kontzeptuak ikastea ezinbestekoa da gero zirkuituak muntatu ahal izateko. Jarraian proiektuan zehar erabili ditugun hainbat gailu elektronikoren ezaugarriak azalduko dira, elektronikako hainbat definiziorekin batera.

B.1. Ohm-en legea

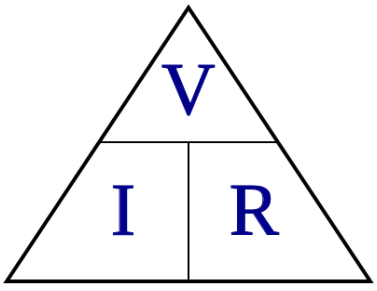
Ohmen legea Georg Simon Ohm fisikari alemaniarrak proposatutako lege fisiko bat da. Bere esanetan:

> *"erresistentzia elektriko batean zehar korronte elektrikoa pasatzerakoan, beraien muturren arteko potentzial diferentzia eta erresistentzian zehar igarotzen den intentsitatea zuzenki proportzionalak dira"*

Beraz, zirkuitu elektronikoetan, Ohm-en legean aipatzen diren hiru adierazpenak elkartrukatu genitzake, *I=intentsitatea, V= tentsioa* eta *R= erresistenzia* izanik:

$$I = \frac{V}{R} \quad \text{edo} \quad V = IR \quad \text{edo} \quad R = \frac{V}{I}.$$

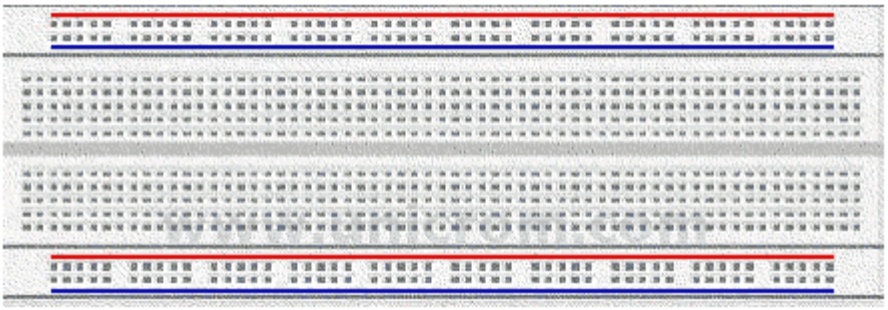
Ekuazioaren elkartrukagarritasuna triangelu batekin irudikatu daiteke, non V (tentsioa) goian jartzen den, I intentsitatea ezkerrean jartzen den, eta R erresistentzia eskuinean jartzen den. Ezker eta eskuineko atalak banatzen dituen lerro bertikalak biderketa adierazten du, eta goiko eta beheko aldeak banatzen dituen lerro horizontalak zatiketa adierazten du. B.1. Irudian ikus dezakegu grafikoki azalduta.



B.1. Irudia: Ohmen legearen hiru adierazpenak

B.2. Gailu elektronikoak

Breadboard edo *Protoboard* zirkuitu elektronikoak probatzeko txartel bat da zirkuitu osoa soldatu beharrik izan gabe. B.2. Irudian agertzen da *Breadboard* estandarra.

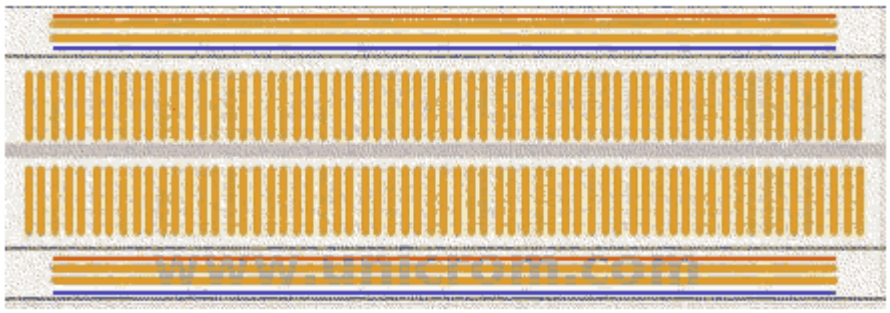


B.2. Irudia: Breadboard estandarraren adibidea

Breadboard-eko konexioei dagokionez, ondorengo ezaugarriak ditu:

- Kanpoko sekzioetan konexioak horizontalki agertzen dira. Normalean elikatze iturri bezala erabiltzen dira, lerro bat tentsio altuarekin eta bestea 0V-ko tentsioarekin.
- Barruko sekzioetan konexioak bertikalki egiten dira eta normalean zirkuitu elektronikoa hemen kokatzen da, soldatu beharrik izan gabe.

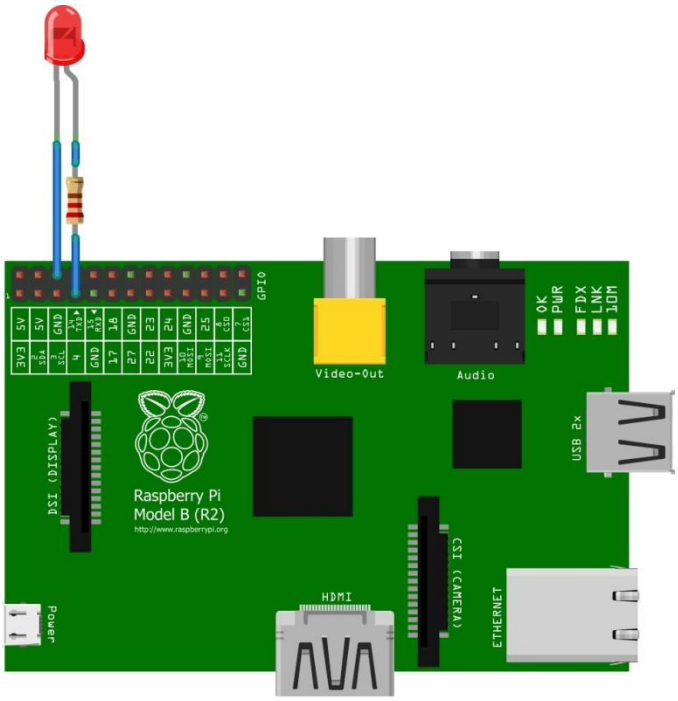
B.3. Irudian orain azaldutakoa grafikoki agertzen da.



B.3. Irudia: Breadboard-aren konexioak

Badut beraz konexioak egiten ahalbidetzen duen txartela (*Breadboard*) eta hau elikatzen duen txartela ere (Raspberry Pi). Zirkuitu elektronikoak muntatzeko gelditzen zaidan gauza bakarra, zirkuitua bera osatzen duten gailu elektronikoak azaltzea da, ondorengoak direnak:

- Led: Kontuan izan behar den lehenengo gauza Ledaren polaritatea izango da. Ledak bi hankatxo dituzte, hainbat luzerakoak. Luzeena alde positiboa da eta motzena aldiz, negatiboa. B.4. Irudian agertzen da Raspberry Pi txartelaren eta Ledaren arteko konexioa. Raspberry-ak bere GPIO pinaren bitartez 3,3V tentsioa bidaltzen dio Ledari. Erresistentzia egongo ez balitz, zirkuituan egongo zen intentsitatea oso altua izango litzake Ledarentzat eta ondorioz, erreko litzateke.



B.4. Irudia: Led zirkuitu bat Raspberry Pi-arekin

Zirkuitu elektronikoetan ondorengo Ledak erabili ohi dira:

- Gorria 1,8V
- Berdea 2,1V
- Horia 2,1V
- Txuria 3,6V
- Urdina 3,6V

Bere voltaiak zein diren jakinda eta elikadurak eskaintzen duen tentsioarekin jokatuz, Led horrek beharko duen erresistentzia kalkula daiteke, Ohm-en legea jarraituz.

- Erresistentzia: Edozein zirkuitu elektronikotik pasatzen den korrontea mugatzen duen gailu elektronikoa da, gainontzeko gailuak jasan dezaketen korronte bat jaso dezaten. Hainbat Ohm-eko erresistentziak daude merkatuan, egin beharreko gauza bakarra, gure zirkuituarentzat beharrezkoak direnak ipintzea litzateke.

  Erresistenzien balioa era erraz batean kalkula genezake, B.5. Irudiko informazioa erabilita.

Kolorea	1. eraztuna	2. eraztuna	3. eraztuna	Biderkatzailea	Perdoia	Tenperatura koefizientea
beltza	0	0	0	$\times 10^0$		
marroia	1	1	1	$\times 10^1$	±1%	100 ppm
gorria	2	2	2	$\times 10^2$	±2%	50 ppm
laranja	3	3	3	$\times 10^3$		15 ppm
horia	4	4	4	$\times 10^4$		25 ppm
berdea	5	5	5	$\times 10^5$	±0.5%	
urdina	6	6	6	$\times 10^6$	±0.25%	
morea	7	7	7	$\times 10^7$	±0.1%	
grisa	8	8	8	$\times 10^8$	±0.05%	
zuria	9	9	9	$\times 10^9$		
urrea				×0.1	±5%	
zilarra				×0.01	±10%	
kolore barik					±20%	

B.5. Irudia: Erresistentziak kalkulatzeko taula

Ondorioz, 4 eraztun dituen erresistenzia batean, ondorengoak kontuan hartuko dira:

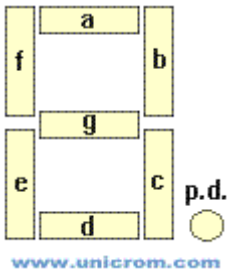
- 1. eraztuna: Lehenengo digitua.
- 2. eraztuna: Bigarren digitua.
- 3. eraztuna: Biderkatzailea
- 4. eraztuna: Errore tolerantzia ehunekoetan.

Adibidez, gorria laranja marroia eta urre eraztun bat duen erresistentzia, ondorengo Ohm tartean egongo da:

- 1. eraztuna: gorria = 2
- 2. eraztuna: laranja = 3
- 3. eraztuna: marroia = 10^1
- 4. eraztuna: errore tolerantzia +- %5
- Emaitza = 23*10 = 230 eta %5-eko errore tartea, 218,5 Ohm - 241,5 Ohm

- Pultsadorea: Proiektuen interesa handitu dezakeen beste gailu elektroniko bat. Honen bitartez, zirkuitutik pasatzen den tentsioa aldatu daiteke eta tentsio honen arabera, programaren portaera aldatu. Ledekin gertatzen den bezala, erresistentzia batekin babesteko beharra dago (10KOhm-ekoa), ez erretzeko. Pultsadoreak bi pin ditu, pultsatzen ez den bitartean komunikatzen ez direnak, aldiz, pultsatzerako orduan, korrontea pasatzen uzten dute.

- Zazpi Segmentuko *display*-a: Zenbakien errepresentazioa egiteko erabiltzen den gailu elektronikoa da hau. Segmentu bakoitza independentea da besteekiko. Ondorioz, segmentu bat itzali edo pizteko ez da beharrezkoa besteekin ezer ez egitea. Segmentu bakoitza hizki batekin errepresentatzen da, B.6. Irudian ikusten den moduan.



B.6. Irudia: Segmentuen izendapena

Bi motatako *display*-ak ikusiko dira:

- Anodo *display*-ak: segmentu guztiak pin jakin baten bitartez elikadurara lotzen dira, konexio hauen bitartez doan korrontea erresistentzia (330 Ohm) baten bitartez mugatuz.
- Katodo *display*-ak: Aurrekoen kontrako portaera dute. Segmentu guztiak pin jakin baten bitartez lurrera lotzen dira. Elikaduratik pin bakoitzera doan korrontea erresistentzia (330 Ohm) baten bitartez mugatuko da, Ledak ez erretzeko.