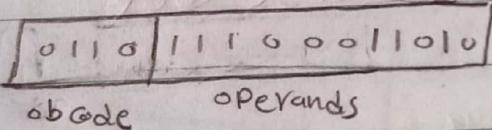


Booting Sequence :

أول ما ندوس على زرار الـ power [or بوردة] فـ CPU يدخل [reset state] طبعاً [reset entry point] هو entry point الرابع لـ CPU أو address الموجون في الـ flash memory [الذى يحتوى على الـ address] (يعود) يستعمل بعد الـ reset يروح يتناول على وـ all [all address] يكون متعدد لكل بوردة ونقدر نذكره من الـ specs.

طبعاً [all entry point] يروح عارض صيغة entry point [all address] [all address] [all address] [all address]

Fetch, decode, execute & store ← يعمل ٤ عمليات ←
assembly [as address] ← assembly [Fetch]
محظوظ في الـ ss. (١) Flash ← طبعاً سبب يكون محظوظ
كل [binary] ← binary [operands & opcode]



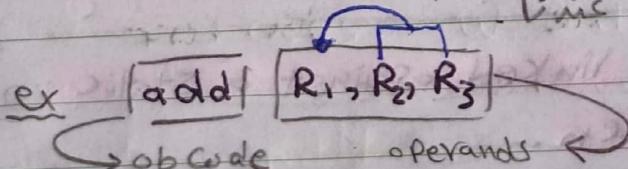
الـ all [instruction] هو ده اللي يكون فيه الـ opcode

add, move, load, ...

والـ instructions [all instruction]

binary ← قيمة طبعة الـ instruction

all [instruction] ← all [operands]



ما دلنا بنقول للـ processor اجمع R₂, R₃ واعطى

R₁ الـ add

all [opcode] ← صر اد

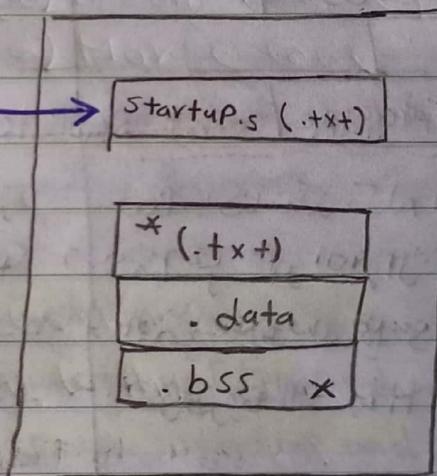
R₁, R₂, R₃ ← operands

بعد كدا decode هي العمل على بعدها instruction instruction يتابع ال operands أو opcode الخاصين بال instruction .
 بعد ما نفهم ال instruction بعدها execute .
 فببرد هنا انه ينفذ ال instruction سواء كانت add ففيه تغيير الرقمنين او اى عناصر ثانية
 يقدر كدا نوصل لآخر عملية و من store .
 في الحالات دي يتغير مكان ال الكملة فعل ليها رجع في الميموري ولا لا لوقيه زي add او sub .
 يتابع الرقمنين في ال store ففيه يعمل memory .
 . execute بعده او memory .

الموجود علني في ال Flash memory يكون هكذا binary File

(startups) has the piece of code that will execute before main function.

* (.+x+) has the instructions code of the program written in binary (opcode+operands)



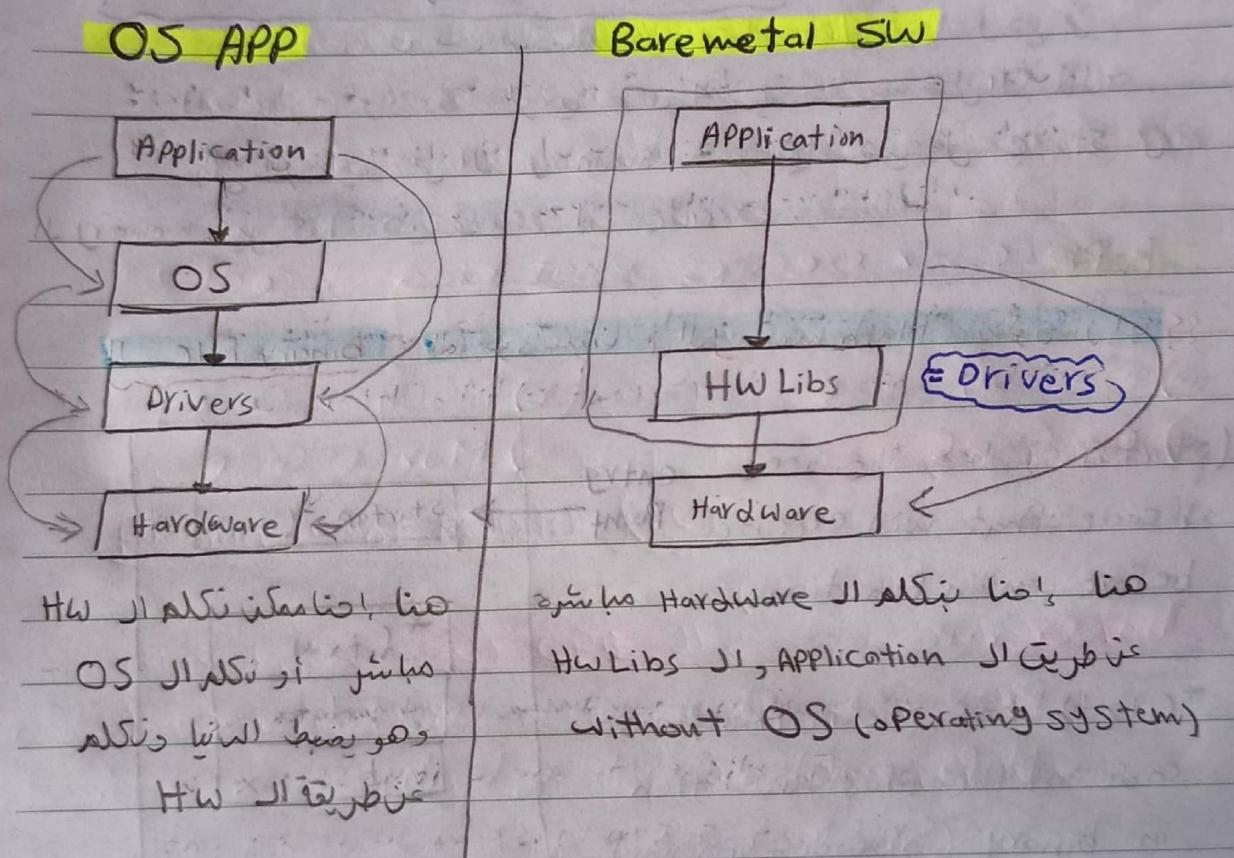
Bootloader.bin

(.data) it contains all initialized global and static variables.

Your Berametal SW

(.bss) it contains all uninitialized global variables and they will be initialize with zero.

- Most processors have a default address (entry point) from which the first bytes of code are fetched upon application of power and release of reset.
- we can get it from Specs



Two cases w/ Boot sequence !!

- Case 1
- Case 2

تسلال نسبو فوجه (cont.)

Boot sequence Case 1 :-

In some SOC you can burn your baremetal SW directly on the Flash memory and in this case you can put the reset section on the CPU entry address based on the SOC datasheet. You can do it (reset in CPU entry point) by linker script. (location counter ".")

يتعالج الكود على إحداثيا كابتيل على أول لهايلد يقوم startup code بـ reset section

entry point ليس قبله على reset section

بالاتالي إذا ممكن ندخل الكود بكله على entry point لـ OS بـ SW

SW يتعالج flush على lines
 → Bootloader SW → Your Baremetal SW

Bootloader → it is a Baremetal SW make specific functionality and it has its own startup and main function.

بعد إحداثيا ممكن نحط على reset section
 entry point على sis Bootloader
 وفي بعض الأوقات need to have boards في سلاسل
 (أكواد بتاتيك) وسلاسل (أكواد بتاتيك) هنا خارج
 entry point على sis reset section وـ
 main function نـ jump على حاجات آخر يجعل

Note run in Line Bootloader هل كل المبرمجون موجودون في

Line Software لـ

No \Rightarrow if we have entry point address
and have access to burn my own
software in flash memory so we
can run the program without boot loader

يكتبه الكود بنا على دفعات executable file

الـ هو هنا يستخدم Baremetal SW وهو من يكتب

البيانات و بمقدمة لما نعرف الـ Baremetal SW على الـ Flash

وأول ما في CPU يستعمل ساقتها أو CPU program

عـ أو entry point

الـ main function jump إلى reset section

(يكتبه في main function jump إلى .txt)

فـ موجود في الـ

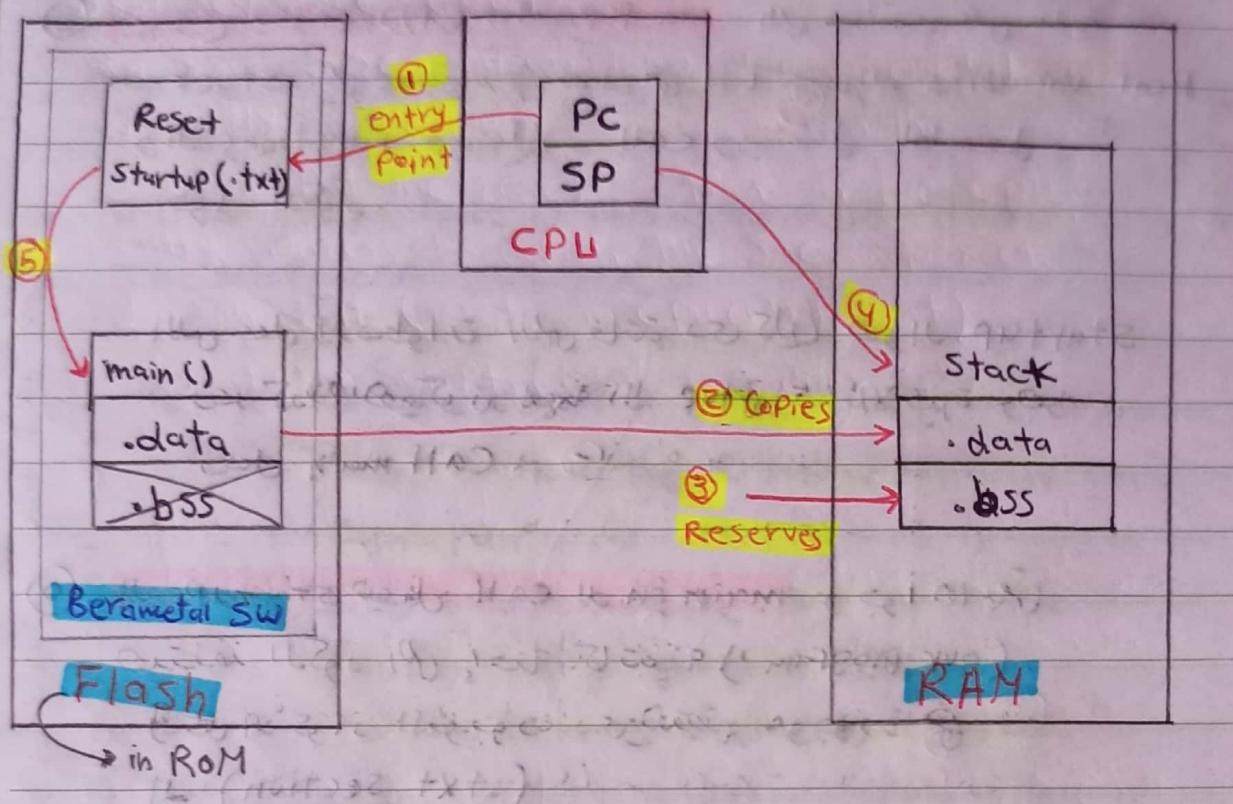
Program Count Register (PC)

address الموجود في الـ CPU ويحتوى على register هو
الـ next instruction بـ fetch وcontaining لها

Stack Pointer register (SP)

register موجود في الـ CPU ويحتوى على بـ باع

RAM أو stack بـ باع المـ موجود في الـ



entry point في قدر ال program هي تكون بيتاً وردياً او developer عيّنان هي قدر حاجة المستخدم بس لما يجيء ال user الكود على ال start up هو ال الذي يفتح ال flash وال الكود موجود في ال start up وال التالي entry point في ال start up موجود في ال flash وال power على CPU لا يحصل على ال power على processor ولا initialization

Flash الموجود في ال data section لا copy في ال RAM و خد بالع دان ال (.data) بيعتّل ال RAM و هنّتّل ال initialized global & static variables على ال

RAM في ال (.bss section) لا دخلي RAM في ال (.bss) بيعتّل على ال uninitialized global vars على ال zero وبعدها initialize

٣) بعوّض يخري ال (SP) stack pointer ال يحتوى على بداية stack ال الى متى نحن بعدهم خى إننا نعرف ال local variables وال call و يتصل functions دى عمل لها نفذ جوا الكور.

اللى عمل الرفطوار المخاتة دى كلها صوار startup بدر كدا هى كون مفهوم ال startup الاخير وهو يعمل main call.

٤) ال دى start up يعمل call main fn ال دى جوا المخاتة هيتنم الكور اى احنا كتبنا (our program). وكل الاكور الموجودة دى (التيق) موجودة من دى (text section).

(note) كل دى (bss) موجود في ال ROM موجود ١٢٨ bytes فرا بخزن في ال Flash دداخ بين اسم ال (bss) و قيمته بـ ١٢٨ bytes.

٥) لآن ال (bss) هبتحجزش أعاكن خى ال Flash لأن منه نفس أهدار مساحة ال Flash أتجز فنعا ال Uninitialized GV يمكن صيغتis أي خط ١٠٦ bytes فرا بخزن و قيمته بـ ١٠٦ bytes فريدين لأننا قادر نخلص ال startup code reset section بعوّض لينا بداية ونهاية ال bss، linker script locator فى ال خى ال location.

و بعس بدر كدا يخظمه في ال RAM بالحجم Zero initialize ليهم بـ ١٠٦ bytes و بس ١٠٦ بـ ١٠٦.

Boot Sequence Case [2]

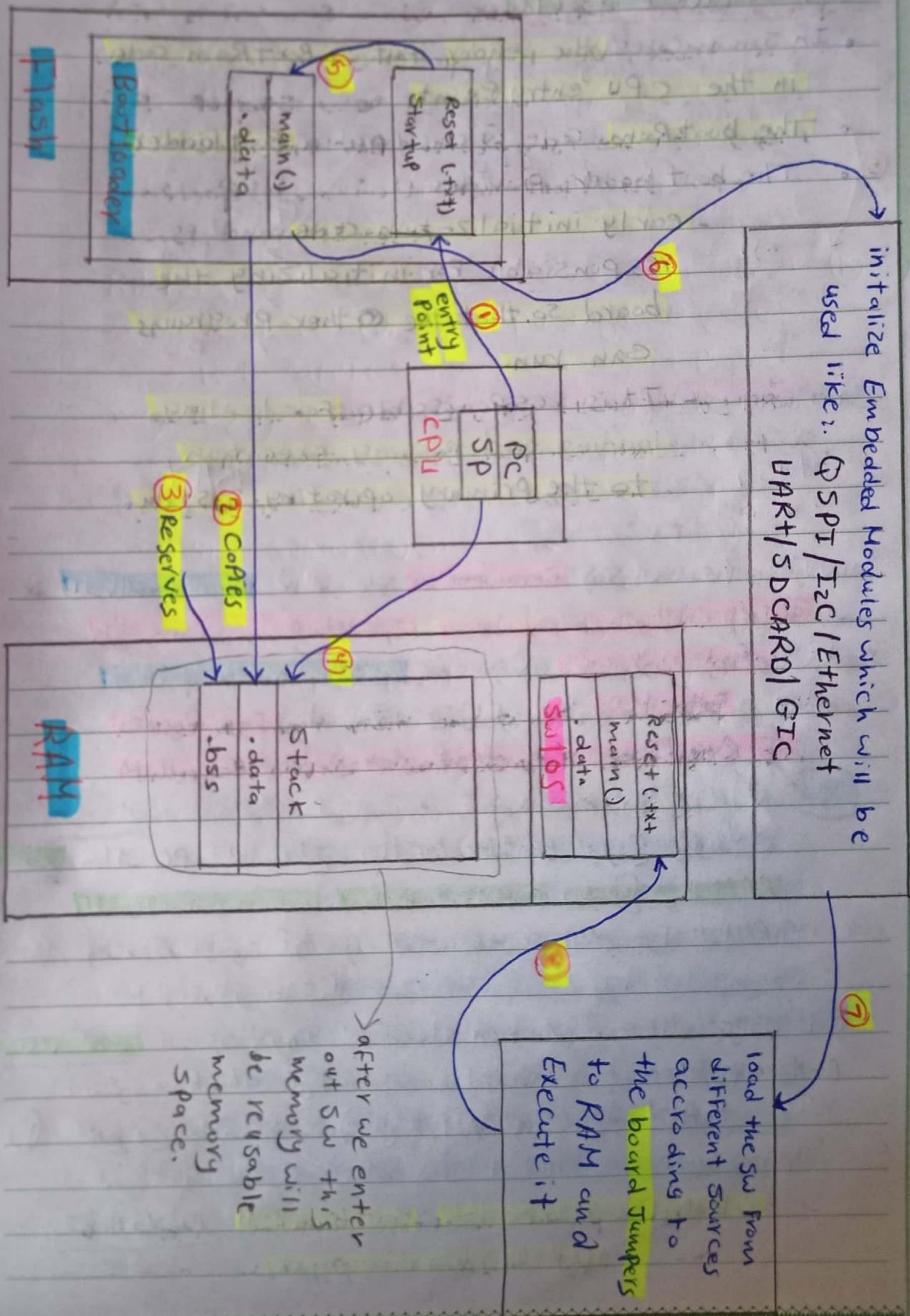
- In some cases the vendor put a BootRom code in the CPU entry Point.
- The bootRom code is considered a Bootloader
- The bootloader provides:
 - early initialization code and is responsible for initializing the board so that the other programs can run.
 - It is responsible for locating, loading and passing execution to the primary operating system.

(كود كاردي) Berametul sw يُهدى Bootloader إلَى
يُدخل بعض الوظائف المهمة startup , main كم
انه بيلدر طاجي و Bootloader ==> main و
بعض بيوت طب يدور يكتز الي ينحل طاجي
at Runtime ==> destination source من إل

الBootloader حال لما يفتح بيلور PC إلَى
RAM في C Partition ==> windows image
ويندوز جوا في RAM إلَى software ==> windows image
ذا لوندوور سائق

Linux image إلَى run alias boot linux OS
Bootloader Linux SD-card or Ethernet ==> محفوظ
صفر بيلور ويعطه في RAM ويندوز يستعمله

الBootloader يفتح الخلاصة
ويندوز يبوت منها (يستعملها).



① **CPU** زول لها الـ reset وـ Power up سايفها الـ
فروع عيده الـ reset section فولامي الـ entry Point الموجود
في الـ Bootloader اخنا في الطاولة
دي حاطين Baremetal SW من Bootloader اون بمحظ
غير CPU (Bootloader) فال BootROM
بندر الـ reset section

② **RAM** لـ Flash مـ (data) لـ copy فـ **فرمول**

linker locator هيعرف بداية ونهاية الـ (bss) من الـ
الموعد في الـ linker script ويرجع مكان ليه في الـ RAM
بالحجم الـ zero initialize فهو عزف ويحمل ليه

③ يعرق الـ stack pointer الـ stack الـ
stack pointer يحتوى على قيمة

④ main fun نـ call ينزل دى الخطا
list application main الـ (main) Bootloader

⑤ **embedded models** N initialize **use** Bootloader
المختلفة تـى اـ

- EEPROM using I2C protocol
- TCP server using Ethernet
- SD CARD
- QSPI

⑥ load the SW from different sources according
to the board Jumbers to RAM and execute it

أى بوردة يمكنون فيها jumbers على اسماح دول بيعقووا سلكتين
 أو أكثر يستخدمون selector ممكن تجعل short circuit كأنهم بين أى اثنين على حسب ما هنا على بروتين الـ Bootloader
 يلود منه يمكنه تكون في اثنين خاصين بال I2C SD card

الBootloader يكرر ما يخلص تلويد ال SW في ال RAM أو
 ساعتها يصلح ال CPU بروح الموجون reset section في ال application startup lists و
 في ال application الخاص لل خارج ال Bootloader يلهم ال CPU بروح معاشرة
 كل ال application الخاص من fun كل ما يجعل عن طريق دانتا بترجمة
 address program counter (PC) في ال next instruction

بعد ما دخلنا جوا ال SW بذاتنا خلاص فنعيش
 فيه ال memory باعتبار ال Bootloader unusable memory space في ال RAM
 لكن ال SW بذاتنا يفترض استخدام ديركت .bss أو .data أو stack فيه

The Two Running Mode

II ROM Mode

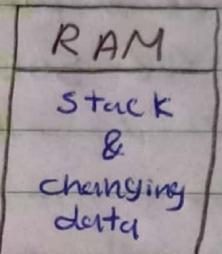
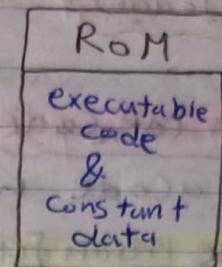
موجود برمجيات SW على ليفلز

ROM (non-volatile memory) خارج

جوار الموجودة داخل

والمتحكم في microcontroller

Case II هي



- Simple → Require smaller memory
- Fixed Code address → Relative Small Code
- entry point يعنى نقطة الدخول بذاتى من الذاكرة الموقعة في البوابة
- address of entry point ذاكرة المدخلة

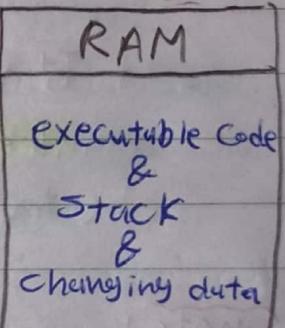
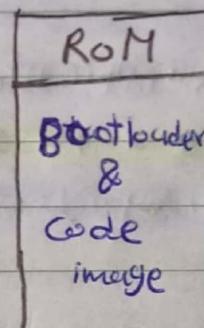
RAM mode

→ Complex

Bootloader موجود على ليفلز

يُنزل إلى مكان Flash أو SW

RAM نزل



Case II

→ Re-locatable Code

يمكن نقل الكود من أي مكان آخر

Bootloader يأخذ RAM

→ Faster RAM هي كود موجود في RAM

والستغل في RAM أسرع من ROM

→ Large Code (SDRAM)

يمكن تثبيت كود على SD-card

Boot loader vs Startup code

Startup Code

is a code which is should be run before
the main function

- initialize processor
 - copy data from ROM to RAM
 - Reserve .bss in RAM
 - define stack pointer (SP)
 - Jump to main function

Boot loader

is an executable binary (Berumetal SW) which have its own startup before its own main

It is responsible for locating, loading and passing execution to the primary operating system.

(note)

executable file can be -

bootloader or Your Beamerfile SW or OS

لینوکس (Linux) در سیستم‌های مخاطب پردازنده (embedded systems) نیز کاربرد دارد. در اینجا Bootloader را که از طبقه OS به عنوان Bootloader لینوکس شناخته می‌شود، بررسی خواهیم کرد. این Bootloader ابتدا RAM را برای بلوکه مکان وینده فرآور و بعد از آن executable file SW را که embedded Linux نام دارد، در مکان وینده قرار می‌گیرد. سپس Bootloader این SW را اجرا می‌کند تا سیستم را آغاز کرده و این کار را با کامپیوتر (PC) همکاری کرده و کار کند.

Bootloaders in depth (real cases using OS)

النقطة الأولى هي entry point، وهي نقطة الدخول إلى المعالج (Processor) على لوحة المoboard (Board). إنها تسمى Jumper numbers أو المعاينات (Jumpers). إنها تحدد(entry point) إلى معالج (processor) في جهاز الكمبيوتر (Computer). Bootloader هو البرنامج الذي يتعقب (tracks) جهاز الكمبيوتر (Computer) ويفعل (executes) الأوامر (commands) التي تم إدخالها (entered) من قبل المستخدم (User).Bootloader هو المبرمج (Programmer) الذي يتعقب (tracks) جهاز الكمبيوتر (Computer) وي�行 (executes) الأوامر (commands) التي تم إدخالها (entered) من قبل المستخدم (User).

ZImage و UImage الفرق بينهما

ZImage هو ملخص (Compressed) لـ UImage، وهو إصدار مضغوط (Zipped) من UImage.

Bootloaders is a piece of code responsible for:

- Basic Hardware initialization

- loading of an application Binary

(most used) or an operating

system kernel from:

- Flash storage
- the network
- SD Card or another type of non-Volatile storage
- USB client

Beside these basic functions, most Bootloaders provide a shell with various commands implementing different operations.

3 Phases boot sequence inside Bootloaders

① RoM code or Boot RoM

It is a code that runs immediately after a reset or power on has to be stored on chip in the SOC → (RoM Code)

دعاية عن كود دمجه صغير المتركة الى يتحقق الـ SOC
الـ ROM بـ booting والكود ده مبتدئ بـ boot جزء
reset section والـ non-volatile memory
entry point ده هو الـ main entry point الخاص بالكور ده وهو الـ main entry point
باتجاه الـ CPU وبالنهاية الكود ده يتحقق في بـ SOC
اولاً على طول الكود ده يتحقق في بـ SOC
بعد ذلك يتحقق الكود ده بـ ROM code

- It is loaded into the chip when it is manufactured and the RoM code cannot be replaced by an open source equivalent.

أول ما دل على CPU دل على entry point دل على ROM code دل على main memory دل على SRAM controller دل على SRAM memory دل على SW دل على SW دل على جواهر واتصالات دل على SPL (Secondary Program Loader) دل على SPL دل على خرائط دل على الـ memory First Stage Bootloader

SS ①

Bootloader هو المخرج من ROM code

الـ

ـ initialize كل شيء في أنه يكمل في الـ

ـ SRAM المكون على SW هي قوته

ـ SRAM و Flash و SW و Load و مكان هو موصى

ـ SD card
ـ تأميني
ـ مكن

ـ SPL هو المدخل SW الخاص بها مكان الـ

ـ SPL هو المدخل SW الذي يدخل SRAM

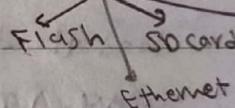
ـ مكن نلاقيه (4KB) و ساكن بتزويده و توصل له (100KB)

ـ وبالتالي لو دخل SW بقى علينا كبر فمساحة المكان

ـ SPL هو المدخل SW

The ROM code is capable of loading a small

chunk of code from one of several pre-programmed locations into the SRAM.



Most of embedded SoCs have a ROM code that works in similar way. In SoCs where the SRAM is not large enough to load a full Bootloader like UBoot there has to be an intermediate loader called the Secondary Program Loader (SPL)

(SPL) will configure SD card controller and it will load UBoot from SD card and it will put it in DRAM after that UBoot will start and configure UART controller so we can use command to implement different operations.

SRAM → static RAM
DRAM → dynamic RAM

التاريخ:

الموضوع:

UBoot will loads the Kernel (SW) From SD card or Ethernet or — to DRAM and then Run this SW

- The Functionality of the SPL is limited by the Size of SRAM.

Conclusion

بعد ما افتحنا الـ CPU power على power on

الـ entry point الـ kernel هو

Vendor tablue أو المخطوط بواسطه BootROM

وأنا مسؤول عن تنزيل الكور ده خار لـ CPU

فيتم تفعيل himem في BootROM والـ SPL

load المـ SPL من طريق انه هيطلع SPL SW

SRAM (أو مـ RAM) أو Flash في

ذالـ SPL يـ تنزيل صورـة بـ عـ مـ

SD card controller لـ Configure

UBoot (أو مـ RAM) موجود فيه الـ

DRAM خـ الـ UBoot يـ بـ حـ طـ الـ T

بعد ذلك يـ بـ حـ طـ الـ T يـ بـ حـ طـ

our (SW) أو kernel image الـ

SD card دـ يـ بـ حـ طـ الـ T load (يـ بـ حـ طـ)

Ethernet (T) (يـ بـ حـ طـ) دـ يـ بـ حـ طـ

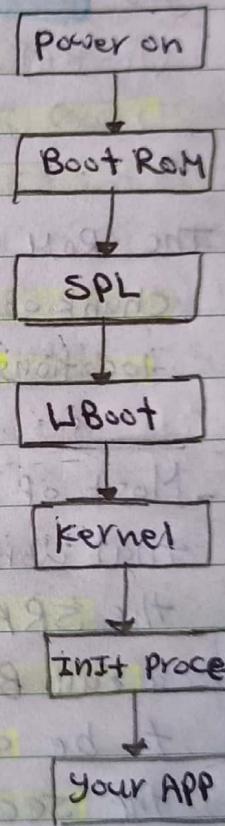
kernel دـ يـ بـ حـ طـ الـ T DRAM

يـ بـ حـ طـ initialize كلـ الحـ اـ جـ اـ

خـ الـ بـ رـ وـ بـ دـ يـ بـ حـ طـ الـ T

init process وـ بـ دـ يـ بـ حـ طـ الـ T

list application وـ بـ دـ يـ بـ حـ طـ الـ T



	Terminology 1	Terminology 2
Boot ROM	Primary program loader	N/A
SPL	Secondary program loader	1st stage Boot loader
UBoot	Main program loader	2nd stage Boot loader

لهم اتنا ادجينا ال SPL

عندما ندخلها من مخزنة SRAM الى UBoot و بالاتالى من مخزن DRAM
 نحط ال SPL في DRAM و بالاتالى في دخلي
 SPL (Secondary Program Loader) يأخذ SRAM
 تكون مخزنة مخزنة نقدر نحط فيها خار
 دار SPL فهو نستخدمه في اننا نجيب ال UBoot
 و نحطه في ال DRAM

DRAM هي microcontrollerine
 SPL هو الذي يحتوي على المخزن

SPL هي التي تخرج عن ال SD card من ال UBoot و بدورها نقدر نستخرج
 و نحطه في ال RAM و بدورها نجيبي
 image kernel or our SW

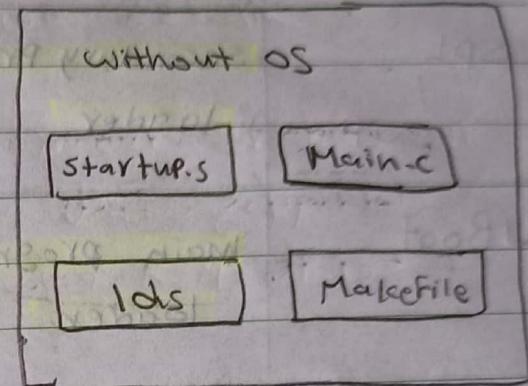
Booting Sequence Examples

→ [open slides]

Baremeta SW

Hardware \rightarrow OS \rightarrow Linker \rightarrow Application

OS direct بيون استخراج startup و بالعكس تكون \rightarrow Linker

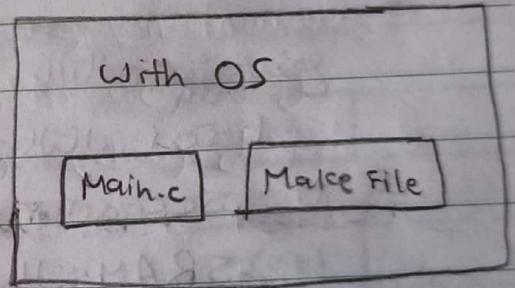


OS APP SW

Hardware \rightarrow OS \rightarrow Application

طريق OS و بالعكس من \rightarrow OS \rightarrow Linker او startup \rightarrow Application

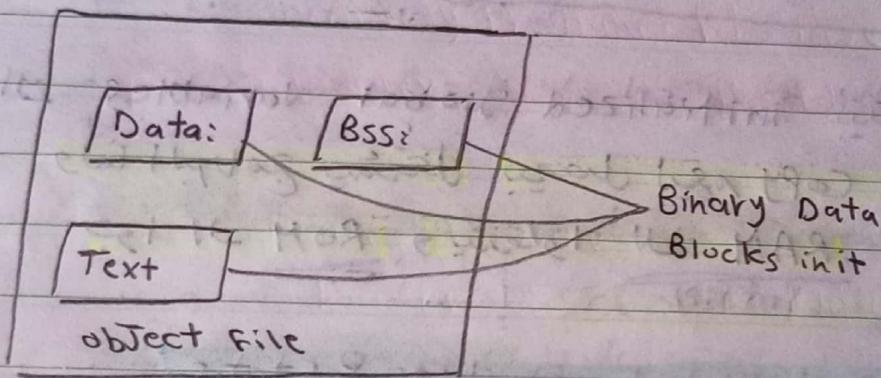
ال OS هو الذي يكمل



Navigate the .obj Files (relocatable images)

<File>.o is relocatable object file

Machine code have a virtual address
not SOC physical address



instructions → in **Text**

initialized global & static variables → in **Data**

uninitialized global variables → in **Bss**

To understand the binary file we will

use Bin utilities → **objdump**

(-h) → headers (sections → .text, .data, .bss, ...)

(-D) → disassembly

لدي مينيس ناوندال (.obj File)

Virtual addresses هي عناوين فريدة لأن كل عنوان

relocatable image is متحركة ولذلك يسمى File

Physical addresses هي معلومة في obj File

لكل عنوان كاسن على عنوان

VMA → Virtual memory address of the output section.

LMA → load (" ") " " " "

(.rodata) read only data (global)

This section will contain constant variables.

Global Variables & static Variables

initialized global Variables & static

الـ data ينطحروا في initialized global variables ذي

Flash memory لـ copy new بـ حصل على وما البرنامج يـ ذاته رام و روم

RAM و لديه ROM

uninitialized global Variables & static

الـ .bss بـ تحطوا جوا الـ uninitialized global Variables

Startup لـ is طريقة لـ zero لـ initialize لـ new وـ ويحصل

ROM و Flash memory و موجود بـ .bss لـ جوا

Flash ميزفيس أحجز (0x8) ولا حاجة جوا ال طبعا

ويكون قيمته بـ zero فاحتاجه نكرن باستخدام

الـ Startup لـ size لـ bss لـ zero لـ نكرن لـ نحو

جوا RAM و ومن size نجز مكان لـ RAM و ومن

zero لـ initialize و

(.bss) is not in Flash as it is not have a value

we reserve a section for it in RAM by

knowing its size and initialize this

Section with zero

All uninitialized (global / static) Variables

are stored in (.bss)

- Since the variables do not have initial value they are not required to be stored in (.data section in Flash memory).

Ro data read only data

it contain constant variables and it is
Store in Flash memory in ROM. → (global)

لیه اول (rdata) مینقله ایت من ای رام و ROM

Constant Variables بَصْوَى عَلَى اَنْدَار (rodata) کیمیاں اُر
at run Time ہندھل لہو تحریل
و بالتاں میں RAM میں نہحتاج منتقلہ اُر کس
ال (الی) اُر Variables فی مکن تیکرداری
RAM فی مکان کرنا منتقلہ اُر کیا؟

The .rodata is also used for

Sorting String Constants

load / Run time location

load location \Rightarrow initial address is ذريعة المكتوب
from burn like a star

run time location \Rightarrow جائزی address is مکان RAM or ROM in system و سعیل CPU لایا

ex (.data) → فيFlash موجود (البيانات) في CPU على load location

وينقل RAM خارج الذاكرة بـ runtime location بـ

Flash	RAM
Vectors	
·text	
·rodata	
·data	·data ·bss

Both load and runtime location

Load location

runtime location

global static var

extern also gives us same file

local static var

Scope → within Block

lifetime → whole program

means this var have value

and it will save in data memory

Scope → within file

lifetime → whole program

and it will save in data memory

Variable	load location	runtime location	Section
• global initialized • global static initialized • local static initialized	FLASH (RoM)	RAM	• .data Copied from Flash to Ram by Startup
• global uninitialized • global static uninitialized • local static uninitialized	X	RAM	• .bss Startup reserve space For it and initialized it by zero
• local initialized • local uninitialized • local const	X	Stack (RAM)	in Stack at runtime
• global const	Flash	X	• .rodata section

global const is exist in .rodata section in flash memory and it still in it at runtime and don't move to RAM

local const isn't exist at load location because it will be created at runtime and it is exist at stack or RAM because we don't need it after the execution of function is end so we put it in stack to be deleted.

bss is not exist at RoM it is only exist at RAM.

(note)

we open binary sections and we don't find .rodata section why??

because in our program there is

no constant global variable so

the .rodata section isn't exist

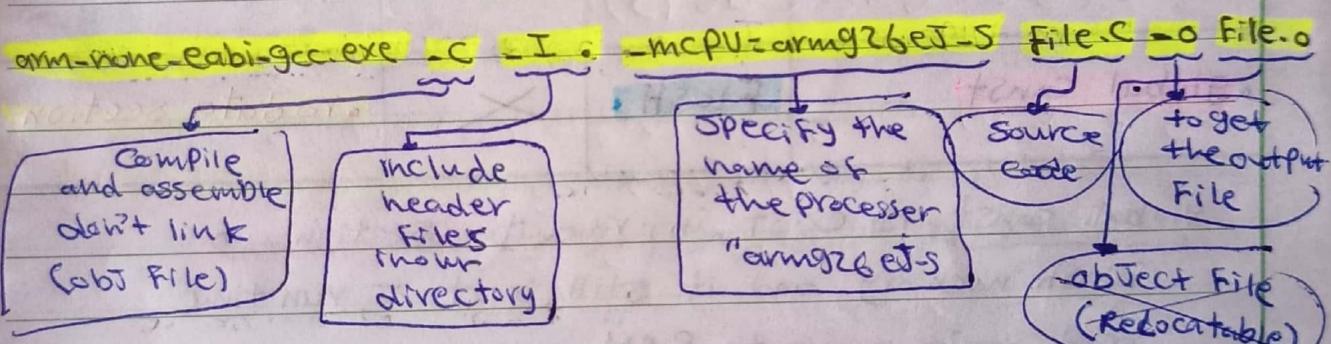
and if we define const global

variable then we will find .rodata

Section.

How To Get binary sections ??

1) get the binary file



2) see the sections from binary file using

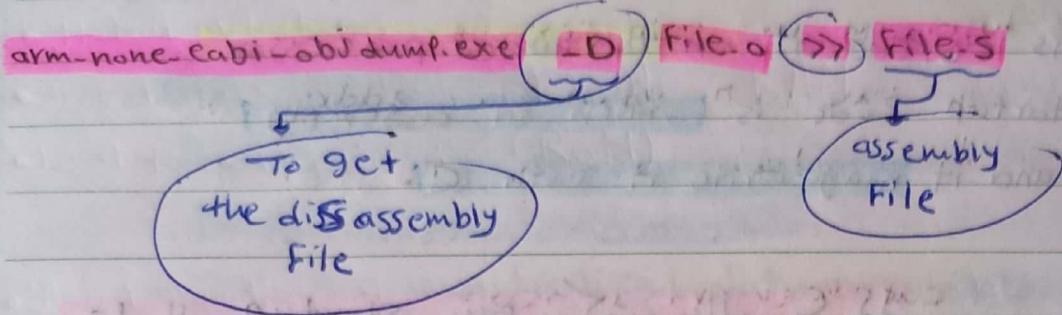
Bin Utilities "objdump"

`arm-none-eabi-objdump.exe -h File.o`

To get the headers sections

object (Binary) file

How To Get disassembly file from the binary ?



To get more information about Sections in binary

arm-none-eabi-objdump -S File.o >> File.s

Some flags used with arm-none-eabi-gcc.exe

-g → To get debug information in binary file

-O0 → No optimization

-O1 / -O2 / -O3 → to enable optimization

with different levels.



C Startup

- it is the code runs before main().
- it is dependent on the target processor.
- startup code is written in assembly
and it may write it with C

هل ار startup ينفع يكتب بال C

ار startup هيرن قبل ال main والمنفذ من ضمن
بيانات ال startup (SP) عمان نعرف نروح الكور
ار (C) بتاعنا ونشغل فيه فالمنفذ بيقى بال
بيانات ال startup فولا هو بيعمل سورة حاجات عمان
نعرف نشغل الكور بال (C).

بس خد بالك بقا لأن في بعض أنواع الprocessors

Startup ينفع يكتب بال Cortex M

بال C عارى و كذا

عمان ال Cortex M species بتاعته بتحول

اننا نحط entry point في SP ال address

فإن CPU بيأخذ address اللي إدنا حاطينه في

ال entry point وروح يحطه هناك (SP) المطلوب لو دره

و بعد كدا يروح N address اللي بدرا وينفذ اللي

فيه وبالعكس إتنا ممكن زكتب ال startup

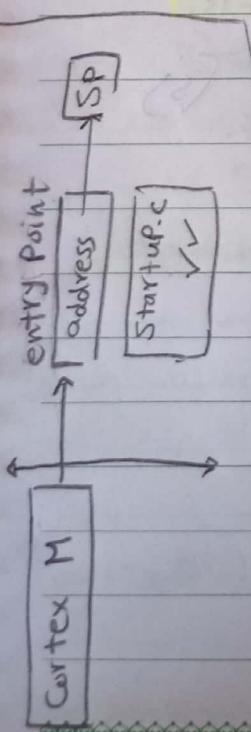
entry point في address اللي بدرا

فأولها إن CPU يستغل هيرن يرجع ال (SP)

بال entry point عن address المخطوط عليه

كـ ال C المكتوب

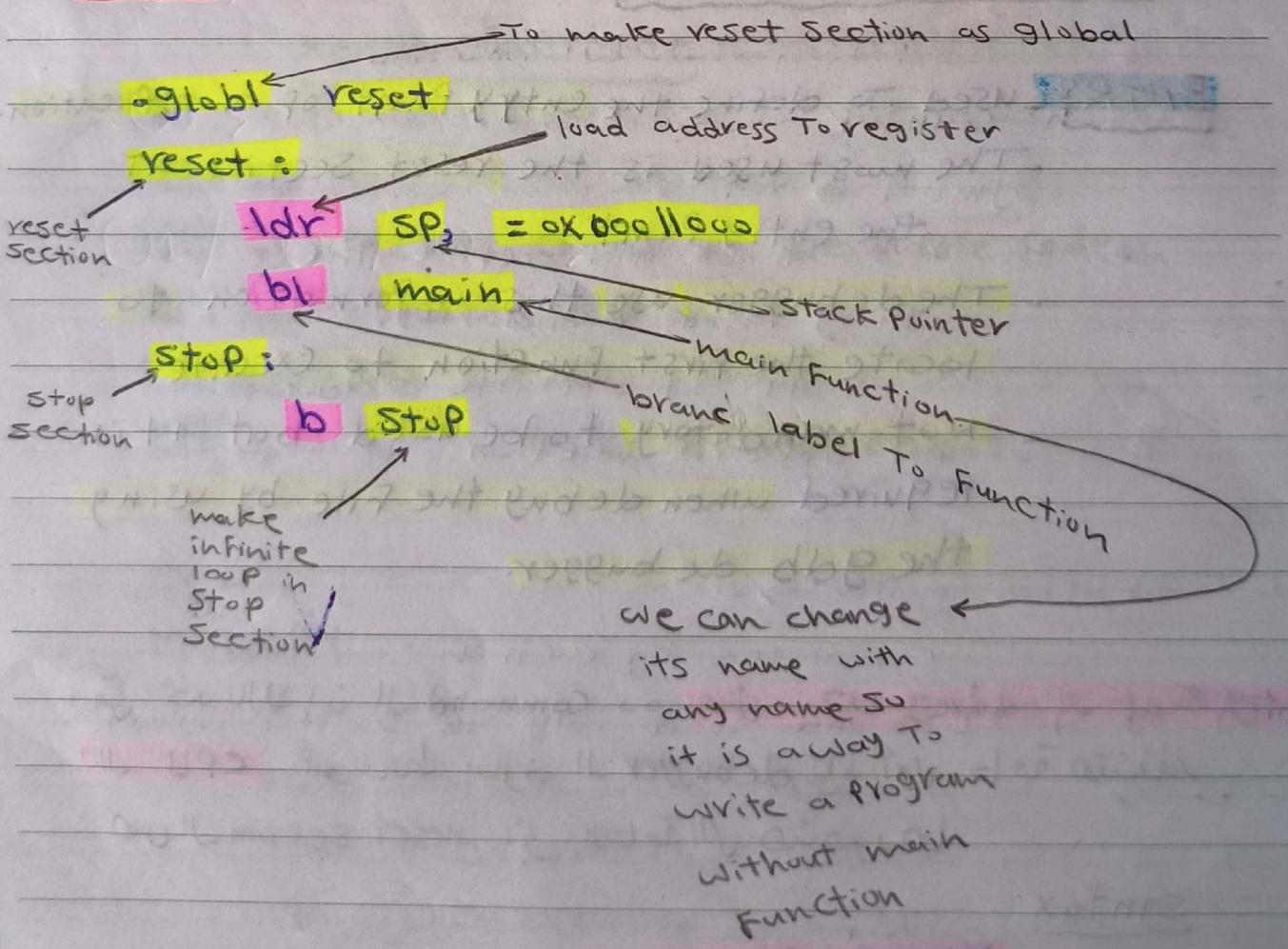
. startup ال



Startup Code for C program usually consists of
the following series of actions -

- Disable all interrupts
- Create a vector table for your microcontroller.
- Copy and initialize data from ROM to RAM
- Zero the uninitialized data area (.bss)
- Allocate space for and initialize the stack
- Initialize the processor's stack pointer
- Create and initialize the heap
- Enable interrupts
- Call main

Simple startup in lab[1]



Linker and locator

LMA → load memory address

يذكر ان الـ LMA هو عنوان الـ binary الذي ينتمي الى عنوان الـ memory address.

VMA → virtual memory address

(RAM)

الـ VMA هو عنوان الـ memory address الذي ينتمي الى عنوان الـ CPU.

- linker script is based on target microcontroller.

- linker script at linking take option **-T**

Linker Script Commands

ENTRY used to define the entry point of an application.

- The most used is the reset section at the entry point

- The debugger use this information to locate the first function to execute

- Not mandatory to be used but it is required when debug the file by using the gdb debugger.

entry point الـ address معنده الا Command الـ entry point في حين ان الـ CPU الـ memory address المفهوم في debugger الـ CPU الـ memory address المفهوم في حين ان الـ reset section الـ memory address المفهوم في حين ان الـ

Syntax

ENTRY(symbol)

MEMORY

is used To decribe any memory

(it uses all memory in one go)

so it

Syntax :

MEMORY

{

We may
write '0'

We may
write 'L'

`name(attr) : ORIGIN = origin, LENGTH = length`

`name = 0 = origin, l = length`

Define
name of
the memory
region

Start address
of the region
in physical
memory

the size in
bytes of
the region

Section attribute

'R' → read only section

'W' → read / write Section

'X' → Section containing executable code

'A' → allocated Section

'I' or 'L' → initialized Section

'!' → invert the sense of any attribute

ex:

MEMORY

{

`vect : 0 = 0 , l = 1K`

`rom : 0 = 0x400 , l = 127K`

`ram : 0 = 0x400000 , l = 4K`

{

* ⇒ الملفات التي تتحول إلى ملف (.text) أو (.data)

التاريخ:

الموضوع:

SECTIONS

- used to create different output sections in the final executable file
- you can use it to merge the i/p sections to the o/p sections

Syntax:

SECTIONS

{ .text { .secname : contents } }

.secname :

{ .contents }

}

}

SECTIONS

{ .text { }

:

* (.text)

}

}

location counter

symbol → dot `:

we use it to track and define the memory layout boundaries.

we use it to specify specific address for specific section.

location counter ⇒ it is automatically addressed

calculated by each section size.

should be used only in section command.

$\{ > (\text{Vma}) \text{ AT} > (\text{Ima}) \}$

Vma is a specify relocatable section

address in run-time located

section box بعدها يكتب الـ address الـ AT

لـ CPU ليـ run time فـ الـ AT من الـ memory

Lma is a specify relocatable section

address in load time located

بعدها section box الـ address الـ AT

لـ CPU وـ Flash فـ الـ AT من الـ memory

ex

.data :

{

* (.data)

$\{ > \text{RAM} \text{ AT} > \text{ROM} \}$

↓

(Vma)

at run time

Startup Copy .data

From ROM to RAM

↓

(Ima)

at load time

ex

location counter

= 0x10000;

.Startup :

{

startup.o (.text)

This mean that startup will

put at address 0x10000

$\{ > \text{ROM} \}$

↓

This (Vma) & (Ima)

load at start ROM at (.text)

running at start of (.text)

Linker used to resolve all symbols.

Symbols is address but not carry any value

Variable

have address

can carry value

Symbol

have address

not carry any value

- is the name of an address

- Symbol declaration is not equivalent to variable declaration

- each object have its own symbol table
the linker is resolving the symbols between all object files

- Symbol is used to specify Memory layout boundaries. (in tabz)

How To get Symbol in binary file??

arm-none-eabi-nm.exe

file.o

T → symbol in ~~text~~ section

D → " " data section

R → " " rodata Section

U → un resolved

map file

ما يوحي كل المعلومات عن ال linker

How To get it ??

arm-none-eabi-ld.exe -T linker.ld -Map = out.map

File1.o File2.o -o output.elf

How To get executable file (.elf) ??

arm-none-eabi-ld.exe -T linker.ld file1.o file2.o
-o output.elf

الاستخدام في الملف المترافق مع الملف المترافق
السكنى في الأماكن التي بها

interview)

Why .bss is not appear in the output sections ??

because we may have not any uninitialized global and static variables

use executable file to get .hex or .bin file

We will use → Binary utilities strip "objcopy"

arm-none-eabi-objcopy-exe -O binary output.elf
output-bin

To run the binary file on Qemu

qemu-system-arm
-M Versatilepb
-m 128M
-no graphic -kernel output-bin

How To export Path ??

export PATH = "Path that we want" \$PATH