МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

АРХІТЕКТУРА КОМП`ЮТЕРІВ 2. ПРОЦЕСОРИ

ТЕОРІЯ ТА ПРАКТИКУМ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського

як навчальний посібник для здобувачів ступеня бакалавра

за освітньою програмою «Комп’ютерна інженерія»

спеціальності 123 комп’ютерна інженерія

Укладачі: І. А. Клименко, А. В. Каплунов, В. А. Таранюк, В. В. Ткаченко

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Укладачі:

Клименко Ірина Анатоліївна, д.т.н., доцент, професор кафедри обчислювальної техніки

Каплунов Артем Володимирович, асистент кафедри обчислювальної техніки,

Таранюк Вікторія Анатоліївна, асистент кафедри ОТ, QA інженер Global Logic Ukraine

Ткаченко Валентина Василівна, д.т.н., доцент, доцент кафедри обчислювальної техніки

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 1 від 02.09.2022 р.)
за поданням Вченої ради факультету інформатики та обчислювальної техніки
(протокол № 11 від 11.07.2022 р.)

Навчальний посібник містить теоретичний матеріал та комплекс лабораторних робіт для навчального курсу «Архітектура комп'ютерів 2. Процесори» для студентів, що навчаються за освітньою програмою другого (бакалаврського) освітнього рівня «Комп'ютерна інженерія» за спеціальністю 123 «Комп'ютерна інженерія». Навчальний посібник буде також корисний для вивчення дисциплін в галузі розроблення вбудованих систем та низькорівневого програмування для процесорів з архітектурою ARM. Навчальний посібник призначений для отримання знань і навиків по розробленню програмно-апаратного забезпечення на процесорах архітектури ARM CORTEX-A.

Реєстр. № НП XX/XX-XXX. Обсяг Х,Х авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ВСТУП

Навчальний посібник містить комплекс лабораторних робіт для курсу «Архітектура комп'ютерів 2. Процесори» для бакалаврів, що навчаються за освітньою програмною другого бакалаврського освітнього рівня «Комп'ютерна інженерія» за спеціальністю 123 «Комп'ютерна інженерія» Комплекс лабораторних робіт розроблено для підготовки студентів технічних вищих навчальних закладів в рамках навчання розробці на базі одноплатних комп'ютерів з процесорним ядром архітектури ARM.

Комплекс лабораторних робіт може бути використаний для вивчення вибіркових дисциплін в галузі розроблення вбудованих систем та програмування мікроконтролерів для технологій Інтернету речей, а саме «Технології програмування C/Embedded», «Програмування мікроконтролерів з архітектурою STM23», «Управління IT-інфраструктурними проєктами для розроблення вбудованих систем», «Технології розроблення інтелектуальних систем», «Інтернет речей».

Теоретичний матеріал містить огляд архітектури систем на кристалі з процесорними ядрами ARM, огляд особливостей розвертання операційної системи Linux на процесорах ARM, методичні вказівки до розробки програмного забезпечення для ядра операційної системи та розвертання програмного забезпечення на платі BeagleBone Black з процесором AM335x ARM® Cortex-A8. Практичний матеріал представляє собою комплекс лабораторних робіт призначений для отримання знань і навичків по розробленню вбудованих систем на процесорі ARM.

Всі навчальні матеріали та програмно-апаратне забезпечення адаптовані до використання в режимі on-line. Для перевірки правильності створених проектів та налагодження використовуються вільно розповсюджувані САПР, емулятори та програмне забезпечення з відкритими кодами. Для виконання лабораторних робіт використовуються, емулятори Linux-орієнтованих архітектур (QEMU ARM), інструменти для програмування мікроконтролерів (vim, Eclipse, Cube IDE), апаратне забезпечення (Raspbery Pi, BeagleBone Black,

STM StarterKit GlobalLogic). На кафедрі обчислювальної техніки КПІ ім. Ігоря Сікорського підтримується репозиторій навчальних матеріалів та OpenSource проектів для забезпечення сумісного доступу до навчальних матеріалів.

Матеріали навчального посібника підготовлені за участі фахівців компанії GlobalLogic Ukraine згідно договору про співпрацю з КПІ ім. Ігоря Сікорського, ментори та тренери компанії безпосередньо приймають участь в проведенні лабораторних занять. Це допоможе підвищити впевненість студентів під час проходження інтерв'ю в компаніях по розробці програмно-апаратного забезпечення для вбудованих систем та загалом збільшити їх конкурентоспроможність на ринку фахівців.

Автори навчального посібника дякують студентам кафедри обчислювальної техніки за допомогу в проведенні експериментів та оформленні навчально-методичних матеріалів. Експериментальна частина виконана на кафедрі обчислювальної техніки КПІ ім. Ігоря Сікорського в рамках дипломного проєктування в 2022 році за тематиками «Навчальний програмно-апаратний комплекс для вбудованої системи на базі процессора ARM32» [19], виконаний Сірим І.М., керівник Таранюк В.А., «Навчальний програмно-апаратний комплекс для дистанційного навчання на базі інструментів GitLab» [20], виконаним Савченком Ю.Ю., керівник Клименко І.А.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DTS	(DeviceTree Source code) Вихідний код дерева пристроїв
MLO	(Memory LOader) Завантажувач пам’яті
RBL	(ROM Boot Loader) Завантажувач постійного запам’ятовуючого простою
SoC	(System-on-a-Chip) Система на чіпі
SPL	(Second-stage Program Loader) Програмний завантажувач другого етапу

РОЗДІЛ 1
ТЕОРЕТИЧНИЙ ВСТУП

1.1 Одноплатні комп’ютери

Обираючи одноплатну систему для розробки програмно-апаратного забезпечення як для навчальних цілей, так і для комерційних проєктів необхідно зважити переваги та недоліки цих систем та проектувати набір їх характеристик на вирішення поставленого завдання.

Ідеального рішення не існує, а тому при виборі процесора або SoC для системи, що вбудовується, необхідно враховувати безліч факторів [1, 2]. Зупинимося на ключових моментах, які варті нашої уваги:

- Продуктивність: найважливішим фактором є загальна продуктивність центрального процесора, що визначаються архітектурою та дизайном чіпа. Обов'язковим аспектом продуктивності є кількість виконуваних інструкцій на секунду та кількість операцій за такт.
- Пам'ять: головне питання, що буде виникати при розгляді цього параметра, чи підхожим є тип пам’яті системи та її об’єм для нашого програмного забезпечення.
- Потужність: у гонитві за продуктивністю системи необхідно не забувати про її енергоефективність. Так, існують методи зменшення енергоспоживання в цілому або в конкретний момент часу. Тим не менш, потужність, що необхідна процесору чи SoC, матиме вирішальне значення для нашого рішення.
- Підтримка периферійних пристроїв: зручність вбудованих рішень або ж легке додавання власних периферійних пристроїв до системи точно стане в нагоді про побудова програмного забезпечення для цих пристроїв, чи такого, що залежить від них. Також, необхідно переконатися, що ЦП коректно працює із зовнішніми периферійними пристроями.
- Архітектура: від цього фактору залежатиме вибір операційних систем та програмного забезпечення. Вибір повинен бути універсальним чи

специфічним, якщо система матиме єдине призначення протягом усього свого життя.

- Робоча напруга: за рахунок зниження робочої напруги чіпа останнім часом знижують енергоспоживання ЦП. Але в цьому випадку можливі невиправні зміни у продуктивності системи. Нам потрібно вибрати ЦП з робочою напругою, що обмежує енергоспоживання, забезпечуючи високу продуктивність.
- Безпека. Як зазначає Роб Вуд – технічний спеціаліст з безпеки апаратного забезпечення та вбудованих систем із NCC Group, безпека має життєво важливе значення при виборі ЦП чи SoC [1]. Розвиток Інтернету речей (IoT) та штучного інтелекту (II) лише збільшує проблеми безпеки у вбудованих системах. Потрібно впевнитись, що процесор має ті безпекові особливості, що необхідні, а також не містить безпекових пасток, які вплинуть на нашу систему.
- Витрати на систему: вартість, звичайно, буде ключовим фактором. Необхідно впевнитись, що матеріальне забезпечення проєкту дозволяє успішно використовувати обрану систему, що аспект витрат не стане тим каменем, через який від проєкту необхідно буде відмовитись в одній з «найгарячіших» його стадіях. Сюди входять витрати на засоби розробки, периферійні пристрої, необхідні схеми та компоненти. Варто не забувати, що складовою витрат є навчання спеціалістів роботі з обраною системою, у випадку комерційної розробки.

Порівняльну характеристику розповсюджених одноплатних систем представлено в таблиці 1.1.

Таблиця 1.1 Порівняльна характеристика поширених одноплатних систем

	Qualcom	Broadcom	Broadcom	TI
Одноплатна система	Arduino Yún Rev 2	Raspberry Pi 3 Model B+	Raspberry Pi 4 Model B	BeagleBone Black Rev C
Мікроконтролер	Atheros AR9331	BCM2837B0	BCM2711	AM3358

Архітектура	MIPSV7.4 64-bit	ARMv8 64-bit	ARMv8 64-bit	ARMv7 32-bit
Пам’ять	64MB RAM, microSD	1GB LPDDR2 SDRAM, microSD	1-, 2-, 4-, or 8GB LPDDR4-3200 SDRAM, microSD	512MB DDR3 RAM, 4GB 8-bit eMMC, microSD
Частота процесора	400MHz	1.4GHz	1.5GHz	1GHz
Периферійні пристрої	USB 2.0, WiFi 802.11b/g/n, 10/100 Ethernet	HDMI, WiFi b/g/n/ac, GigE, USB 2.0, BT 4.2	µHDMI, 4K, MIPI, VideoCore VI GPU, WiFi ac, GigE, USB 3.0+2.0, BT 5.0	microHDMI, 10/100 Ethernet, SATA, USB 2.0
Робоча напруга	5V	5V	5V	5V
Ціна	49€	45€	65€	45€

Вибір одноплатної системи BeagleBone Black [3, 4] для розроблення та налагодження вбудованої системи обумовлений її доступністю на ринку та невисокою вартістю. Це поширена та зручна система з підтримкою великої кількості ОС, зокрема операційної системи Embedded Linux, як безкоштовної для розробки в навчальних цілях системи OpenSource, зручної для студентського товариства, через попереднє ознайомлення з нею під час вивчення інших університетських курсів; мови програмування С – мови програмування, що є поширеною, має довготривалу підтримку, а також є мовою, що має широку бібліотеку для розробки програм середовища ядра саме для обраної операційної системи Linux.

1.2 Принципи розроблення програмного забезпечення для вбудованих систем

Серцем продукту є програмне забезпечення. Блоки, на яких вибудовується програмне забезпечення, визначають обсяг та межі

можливостей продукту. Розглянемо деякі з найпопулярніших варіантів у програмному стеку які визначають середовище розробки.

Компонент програмного стека, який, можливо, має найбільший вплив, це операційна система. Вибір ОС визначає програмне забезпечення, яке ви можете включити у свою систему. Правильний вибір може полегшити виконання необхідних завдань, а інші зробити страшенно складними.

В роботі [1, 2] (рис. 1.1) автори оцінили вісім найбільш популярних операційних систем на ринку вбудованих систем, та мови програмування, які використовуються для створення програм поверх ОС (рис. 2).

	Вбудований Linux	Android	QNX Neutrino RTOS	GreenHills INTEGRITY	Wind River VxWorks	Amazon FreeRTOS	webOS	Windows для IoT
Легкість розробки	★★★★	★★★★★	★★★	★★	★★★	★★	★★★★	★★★★
Ефективність	★★★	★★	★★★★	★★★★	★★★★	★★★★★	★★	★★★
Детермінована поведінка (в реальному часі)	★★★	★	★★★★★	★★★★★	★★★★★	★★★★	★★	★★
API	POSIX	POSIX	POSIX	POSIX	POSIX	FreeRTOS	POSIX	UWP
Підключення	★★★★★	★★★	★★★	★★★★	★★★	★★	★★★★★	★★★★★
Графіка	★★★★	★★★★★	★★★★	★★	★★	★	★★★★	★★★★★
Апаратна підтримка	★★★★★	★	★★★	★★	★★★★	★	★★	★★★★
Відкрите джерело	Так	Так	Ні	Ні	Ні	Так	Так	Ні
Спільнота	★★★★★	★★★★	★★★	★★	★★	★★	★★	★★★
Вартість ліцензії	$	$	$$$$	$$$$	$$$	$	$	$$$$$
Вартість налаштування та загартування	$$$	$$$$	$$	$$	$$	$$$	$$$	$

★ - це найменший рейтинг, а ★★★★★ - найбільший

Рис. 1.1 Порівняльна характеристика восьми популярних ОС [2]

	C	C++	Java	Python / MicroPython	JavaScript / HTML5 / CSS	Rust
Сильні сторони	Вбудованість, робота на ПК без ОС, IoT	Вбудованість, робота на ПК без ОС, IoT, автономні програми	Веб-розробка, хмарна розробка	Хмарна розробка, наукова аналітика	Веб-розробка	Вбудованість, робота на ПК без ОС
Легкість розробки	★★★	★★	★★★	★★★★★	★★	★★★★
Сила мови	★	★★★★	★★	★★★★★	★★★	★★★★
Простота обслуговування	★★★★	★★★★	★★★★	★★★	★	★★★★
Довговічність	★★★★★	★★★★	★★★	★★★	★	★
Ефективність часу виконання	★★★★★	★★★★★	★★★	★★	★	★★★★
Наявність бібліотек/ модулів	★★★★	★★★★	★★★	★★★★	★★★★	★★
Низькорівневий інтерфейс	★★★★★	★★★★★	★★	★★★	★	★★★★
Підтримка підключення	★★	★★★ / ★★★★★	★★★★	★★★★★	★★★★★	★★
Підтримка графіки	★★★★	★★★★★	★★★	★★★	★★★★★	★
Спільнота розробників	★★★	★★★	★ / ★★★★	★★★★	★★★★★	★★

★ - це найменший рейтинг, а ★★★★★ - найбільший

Рис. 1.2 – Порівняльна характеристика мов програмування [2]

Кожна мова програмування має сильні та слабкі сторони, які вибудовують шлях процесу розробки та мають вплив на розробку нашого вбудованого додатка. Їхні обмеження включають типи графічних інтерфейсів, які кожна мова може підтримувати. Вибір ОС та мови розробки є стратегічним. Він матиме свій вплив на складність розробки та підтримки побудованої системи, що в свою чергу відіграє на найважливішому ресурсі – на витраті часу.

Зважаючи на те, що система під керуванням ОС може бути поділена на два абстрактні рівня – віртуальний (користувацький) – *User Space* і рівень ядра операційної системи – *Kernel*. Програмування для вбудованих систем має два аспекти, які обґрунтовують вибір мов програмування. Із рис. 1.2 видно, що для програмування застосунків на рівні *User Space* можуть бути використані мови програмування високого рівня. Тепер поговоримо про програмування на рівні ядра. Тут можна розглянути два типи програмного забезпечення, яке розробляється для вбудованих систем. Це автономні застосунки (*Standalone Apps*), такі системи в принципі не потребують *User Space*, але програми виконуються під керуванням ОС на рівні ядра ОС. В цьому випадку ОС треба надати вже відкомпільований виконуваний код на машинній мові, яку зрозуміє архітектура системи. Окрім того, автономні застосунки можуть працювати без управляння операційною системою, або ОС може бути ще не розвернутою на залізі, в цьому випадку використовуються так звані мови програмування *Bare-metal* («голе залізо»). Як зазначено на рис. 1.2. до таких мов програмування належать сови C та C++ при цьому друга використовується в системах, які працюють без операційних систем, в якості *Bare-metal* мови використовується мова C.

1.3 Плата BeagleBone Black

Виходячи з попередньо визначеного завдання, – розробка навчального програмно-апаратного комплексу, цільової аудиторії та середовища, - студенти технічного університету, нам необхідно зробити вибір в сторону недорогої, розповсюдженої, вдосталь потужної та багатоцільової плати, яка матиме

достатньо низький поріг для початку роботи з нею; операційної системи, що буде недорогою, а краще, безкоштовною, універсальною та невибагливою до заліза; мови програмування, що дозволить краще зрозуміти основи розробки вбудованих систем, буде швидкою, матиме гарантію підтримки розроблених програм на довгий термін – дорога сходиться до BeagleBone Black, Embedded Linux та C.

Обираючи одноплатну систему для розробки програмно-апаратного забезпечення як для навчальних цілей, так і для комерційних проєктів необхідно зважити переваги та недоліки цих систем та проектувати набір їх характеристик на вирішення поставленого завдання.

BeagleBone Black [3, 4] — це недорога платформа розробки, яка підтримується спільнотою, для розробників і любителів. Завантаження Linux відбувається менш ніж за 10 секунд і для початку розробки необхідний лише один кабель USB.

1.4 Процедура завантаження з ROM пам’яті

Після того, як список завантажувальних пристроїв налаштовано, програма завантаження послідовно перевіряє пристрої, перераховані в списку, і виконує процедуру завантаження з пам’яті або периферійного завантаження залежно від типу завантажувального пристрою.

Процедура завантаження пам’яті зчитує дані з пристрою типу пам’яті. Якщо знайдено дійсний образ для завантаження та успішно прочитаний із зовнішнього пристрою пам’яті, код починає виконуватися.

Процедура завантаження периферійних пристроїв завантажує дані з хоста (зазвичай ПК) на пристрій пристрою за допомогою з’єднань Ethernet, USB або UART. Код ROM використовує логічний протокол для синхронізації. Після успішного підключення UART, USB або Ethernet хост надсилає двійковий вміст зображення.

Розглянемо алгоритм запуску коду в ROM на рисунку 1.1. Якщо завантаження пам’яті або периферійного пристрою не вдається для всіх

пристроїв, перерахованих у списку пристроїв, код ROM потрапляє в цикл, очікуючи, поки таймер скине систему. [5]

Рис. 1.1 – процедура запуску з коду в ПЗП [3]

1.5 Первинний та вторинний завантажувачі: RBL та SPL

Зупинимось на перших стадіях завантаження ядра Linux на Beaglebone black на базі AM335x SOC. Для цього необхідні наведені нижче двійкові файли або зображення.

Першим є RBL (ROM Boot Loader), що означає завантажувач ПЗП - первинний етап завантаження. Після цього потрібен SPL, що означає Second stage Program Loader або MLO, це завантажувач другого етапу.

Розпочнемо розгляд процесу завантаження Linux з того, що подивимося, як завантажувач другого етапу завантажується завантажувачем ПЗП.

SOC AM335x складається з ROM, тобто пам’яті лише для читання. Розмір цього ПЗП дуже малий. Для розміщення завантажувача ROM достатньо лише 176 КБ. SoC також має невелику внутрішню оперативну пам'ять, яка становить всього 128 КБ.

Коли на SoC подається живлення, він зазнає скидання, а завантажувач ROM є першим програмним компонентом, який запускається на SoC. Код ROM виконує початкові ініціалізації відповідно до блок-схеми на рисунку 1.2.

По-перше, він виконує налаштування стека. Потім викликає головну функцію, де перша ініціалізує захисний таймер. Таймер запускається на 3 хвилини. Це означає, що якщо RBL не зможе завантажити завантажувач другого етапу протягом 3 хвилин, то захисний таймер закінчиться, і він скине процесор.

Потім відбувається дуже важлива ініціалізація, яка полягає в налаштуванні годинника через PLL. PLL (Phase-Locked Loop) - це механізм генерації тактового сигналу, за допомогою якого можна генерувати широкий діапазон тактових частот для різних підсистем SoC. Для PLL необхідно надати низькочастотне джерело тактового сигналу як вхід, як кварцевий генератор, а потім PLL надасть набагато вищу вихідну частоту для роботи з різними підсистемами SoC, як-от процесор, дисплей, периферійні пристрої тощо.

Розберемося в цьому докладніше. Під час ініціалізації годинника завантажувач ПЗП перевіряє значення SYSBOOT 14 і SYSBOOT 15, і на основі значення 14-го і 15-го контактів SYSBOOT він визначає значення зовнішнього кварцевого генератора, підключеного до SoC.

На апаратному забезпеченні Beaglebone black підключений кварцевий генератор 24 МГц. Знайти його можна в нижній частині дошки «Y2».

На Beaglebone 14-й контакт SYSBOOT повинен бути підключений до високої напруги, а 15-й контакт SYSBOOT підключений до низької напруги.

Якщо RBL зчитує значення як 01, він вирішує, що тактовий сигнал 24 МГц підключений. На основі цих даних він налаштовує механізм PLL SOC.

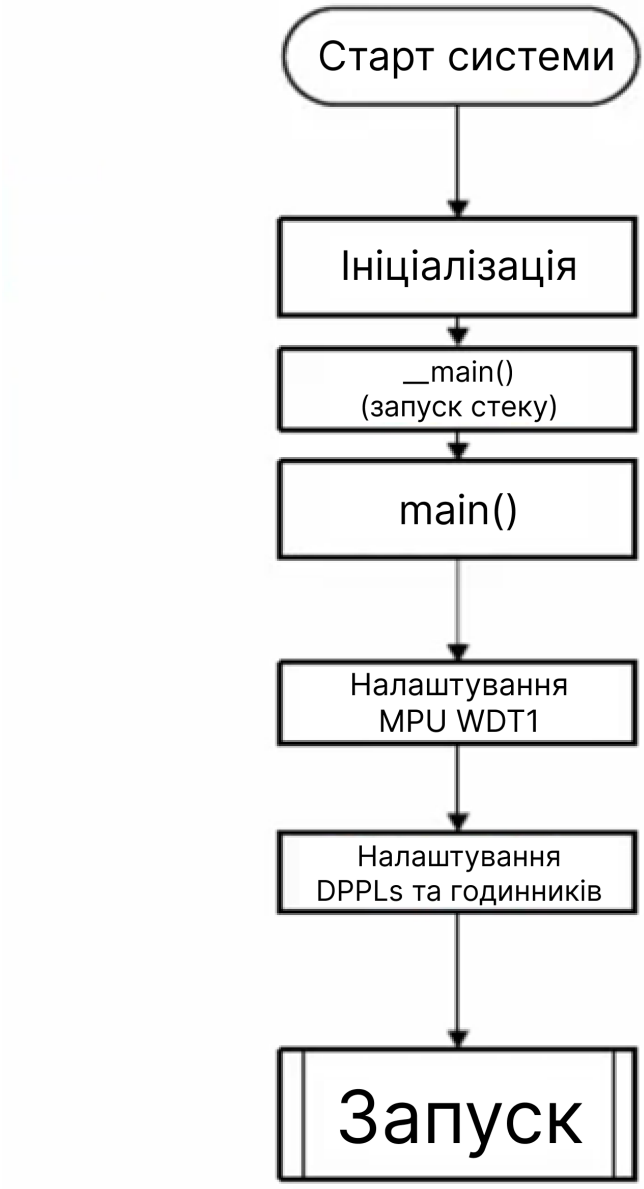


Рис. 1.2 – послідовність запуску коду в ROM пам’яті [5]

PLL подає 500 МГц тактової частоти до процесора Cortex-A8. Це потужність PLL; він просто приймає 24 МГц на вхід і виробляє набагато вищу тактову частоту.

Після ініціювання годинника Cortex-A8 повинен працювати на частоті 500 МГц.

Вторинне завантаження — це етап переналаштування PLL. Необхідно звернути увагу, що неможливо змусити завантажувач ПЗП змінювати ці тактові частоти - вони фіксовані. Я припускаю, що при потребі налаштування MPU на 300 МГц замість 500 МГц, зниження частоти можливе лише на другому етапі завантажувача.

Після цього RBL переходить до завантаження. Він пройде список завантаження, і цикл буде закритий, якщо дійсний образ для завантаження знайдений у зовнішній пам’яті або периферійних пристроях та виконано або після закінчення роботи захисного таймера.

Скажімо, RBL знаходить завантажувач другого етапу в пам'яті e-MMC. Потім код ПЗП копіює MLO/SPL у внутрішню ОЗП SOC. MLO або SPL матиме свій заголовок зображення. Цей заголовок зображення визначений вендором плати Texas Instruments. RBL спочатку зчитує заголовок зображення MLO. З заголовка зображення, він отримає дві важливі відомості. Один – адреса завантаження, а інший – загальний розмір MLO.

Отже, RBL завантажить MLO на внутрішню адресу RAM, зазначену заголовком зображення MLO. Після цього етапу робота RBL майже закінчена - вона лише розпочне виконання MLO.

Ось як завантажувач другого етапу завантажується у внутрішню оперативну пам’ять SOC за допомогою завантажувача першої стадії, яким є RBL.

Зробимо короткий підсумок, щодо розглянутого матеріалу.

Код ROM налаштовує стек, перший захисний таймер встановлюється на 3 хвилини. Конфігурується системний годинник за допомогою PLL, проводиться пошук завантажувальних пристроїв для MLO або завантажувача другого етапу. Копіюється MLO із завантажуваного пристрою у внутрішню оперативну пам’ять. І, нарешті, виконається MLO, що зберігається у внутрішній RAM.

На цьому завантаження не закінчено. Наступним етапом стане завантажувач третьої стадії, наприклад U-Boot, при цьому знадобиться образ ядра, у нашому випадку буде використано операційну систему Linux. Також потрібен Rootfs, який є кореневою файловою системою для успішного завантаження ядра Linux.

1.6 MLO

MLO ініціалізує процесор до такої міри, що завантажувач третьої стадії або U-Boot може бути завантажений у зовнішню оперативну пам’ять, тобто пристрій пам’яті DDR.

MLO виконує ініціалізацію консолі UART, щоб роздрукувати повідомлення про налагодження, переконфігуровує PLL на потрібне значення. Після чого ініціалізуються регістри пам’яті DDR для використання пам’яті DDR, оскільки основна робота MLO насправді полягає в завантаженні третього етапу завантаження, такого як U-Boot, у зовнішню пам’ять DDR.

Далі MLO виконує деякі конфігурації мультиплексування завантажувальних периферійних пристроїв, оскільки його наступна робота полягає в завантаженні U-boot із завантажувальних периферійних пристроїв. Ця стадіє я важливою, оскільки SOC містить величезну кількість периферійних пристроїв. Для кожного периферійного пристрою неможливо надати свої власні набори контактів, тому що це різко збільшить кількість контактів SOC. Щоб кількість контактів була низька, на одному штифті представлено кілька функцій.

Наприклад, pin може використовуватися для різних функцій, таких як SPI, I2C, Ethernet, USB, MMC тощо. Перш ніж використовувати цей контакт, спочатку потрібно налаштувати потрібну функціональність за допомогою мультиплексування.

Якщо MLO збирається отримати U-Boot з інтерфейсів MMC0 або MMC1, то він виконає мультиплексування, щоб виявити функціональні можливості MMC0 або MMC1 на контактах, а також виконає ініціалізацію, пов’язану з периферією MMC. Потім він копіює образ U-Boot в пам’ять DDR та передає йому управління.

U-Boot також матиме заголовок зображення, який вказує адресу завантаження та розмір зображення, яке слід мати. Отже, MLO завантажує U-Boot за адресою, як зазначено в заголовку зображення U-Boot.

MLO не завантажуватиме завантажувач третьої стадії, як U-Boot, у внутрішню оперативну пам’ять SoC. Оскільки внутрішня оперативна пам’ять становить лише 128 КБ, U-boot, очевидно, туди не поміститься. Таким чином, він копіює його в пам’ять DDR, яка є зовнішньою для SoC.

Виникає важливе запитання: чому завантажувач ПЗП не завантажує U-boot безпосередньо у внутрішню оперативну пам’ять? Чому використовується завантажувач другого етапу? Для чого потрібен завантажувач другого етапу? Чому RBL не може безпосередньо завантажити U-boot у внутрішню оперативну пам’ять SoC?

Відповідь наступна: розмір внутрішньої оперативної пам’яті в AM335x становить 128 КБ, з яких приблизно 1 КБ пам'яті вважається захищеною пам'яттю, яку не можна використовувати. А також деяке місце зарезервовано роботи зі стеком і купою.

Таким чином, доступний простір буде менше 128 КБ. Отже, це означає, що необхідно стиснути образ U-Boot до менших розмірів, щоб зберегти його у внутрішній ОЗП SoC. Але, враховуючи всі необхідні функції U-Boot, майже неможливо розмістити образ U-Boot на 128 КБ просторі.

1.7 Процес запуску MLO та U-boot

Тепер давайте подивимося практично на завантаження завантажувача ROM та MLO на апаратному забезпеченні BeagleBone Black. Для цього використаємо попередньо вбудовані двійкові файли.

Для демонстрації використаємо попередньо створений дистрибутив, який називається дистрибутивом Angstrom. Введемо Angstrom Beaglebone demo в Google, звідки завантажимо готові зображення, попередньо створені образи для успішного завантаження ядра Linux на апаратному забезпеченні Beaglebone.

Завантажимо плату через SD-карту, що є найпростішим методом. Для цього знадобиться карта micro-SD, розділена на дві частини. Один розділ має

бути типу файлової системи FAT, а інший розділ має бути типу EXT3. EXT3 і EXT4 — це власні файлові системи Linux.

Підключимо SD-карту до ПК через пристрій для читання карт SD. Відкриємо термінал. Не використовуйте термінал з правами root. Якщо вам потрібен root-доступ, то вам доведеться скористатися командою «sudo». Але працювати безпосередньо з привілеями root іноді небезпечно, і це створює багато роздратування та проблем.

Введімо команду 'dmesg'. SD-карта ідентифікується ПК як пристрій db. Розділимо цю SD-карту на дві частини. Для цього я використаю програму під назвою GParted. Запущу програмне забезпечення, дочекаюсь оновлення пристроїв пам’яті, наявних на ПК та переконаюсь у тому, що обраним пристроєм є саме SD-карта.

1.8 Точка входу в ядро

З завантажувача елемент керування переходить до файлу *head.s*, розташованого в *arch/arm/kernel/head.s*. Відкриємо *head.s* у джерелі ядра Linux.

```
/*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr, r2 = atags or dtb pointer.
 *
 * This code is mostly position independent, so if you link the kernel at
 * 0xc0008000, you call this at __pa(0xc0008000).
 *
 * See linux/arch/arm/tools/mach-types for the complete list of machine
 * numbers for r1.
 *
 * We're trying to keep crap to a minimum; DO NOT add any machine specific
 * crap here - that's what the boot loader (or in extreme, well justified
 * circumstances, zImage) is for.
 */
        .arm
```

Рис. 1.3 – Коментар до файлу *head.s*

Якщо почитати коментарі на рис. 1.3, можна дещо зрозуміти. Вони говорять про те, що зазвичай це викликається з коду декомпресора. Це означає, із завантажувача, який здійснив декомпресію ядра та ініціював виконання розпакованого ядра.

Отже, на даний момент меню вимкнено. Це означає, що віртуальна адресація не ввімкнена. Тому що ми все ще перебуваємо на етапі ініціалізації нашого обладнання.

D-кеш вимкнено. I-кеш процесора також вимкнено. Регістр r0 містить 0. А регістр r1 повинен містити значення ідентифікатора машини.

Кожна планка матиме унікальний ідентифікатор машини. R1 має містити це, оскільки цей файл використовує ідентифікатор машини для виконання специфічної для машини ініціалізації, а регістр r2 буде містити адресу двійкового дерева пристроїв, який присутній у RAM.

Розглянемо деякі важливі підпрограми, викликані цим файлом.

Перевіримо відповідний тип процесора. Цей виклик шукатиме архітектуру процесора, наявну на платі. Дізнавшись тип процесора, він викликає відповідні підпрограми ініціалізації, що стосуються процесора, знайдені у відповідних файлах, що стосуються процесора, у шляху *arch/arm/mm/proc-*.S* - mm означає керування пам’яттю.

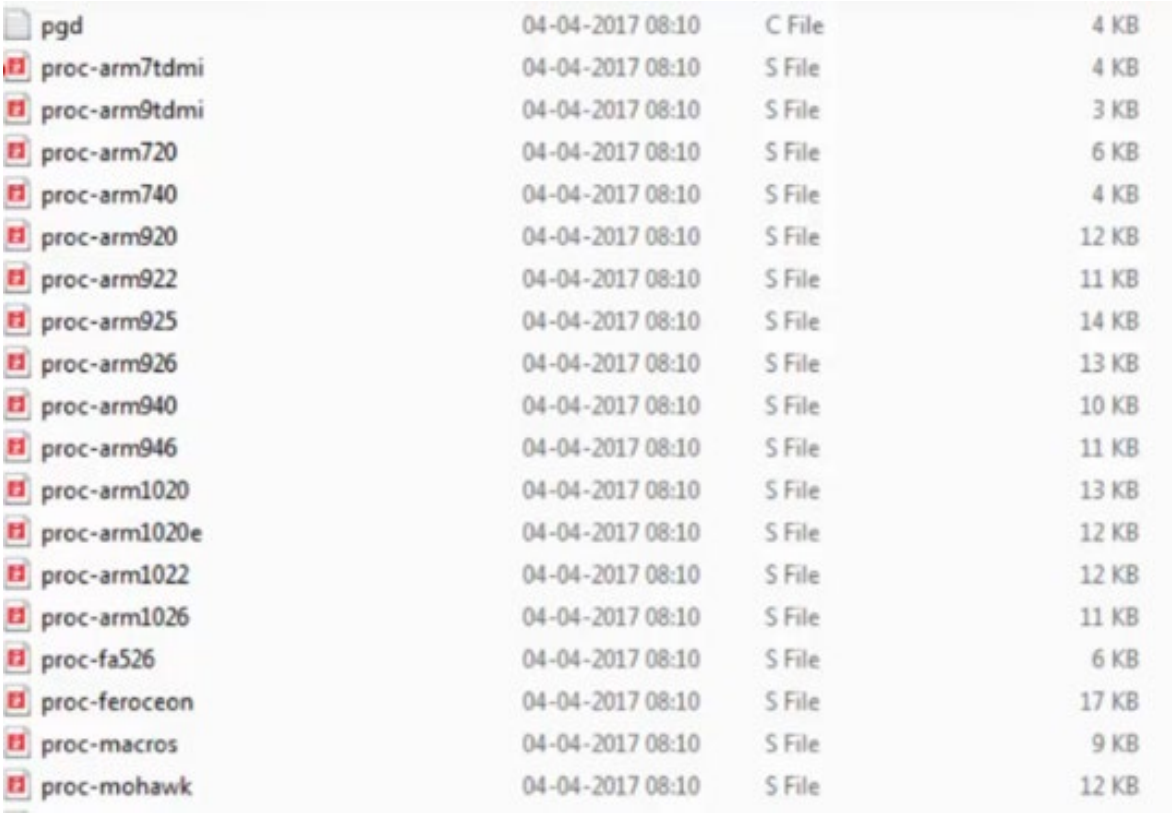
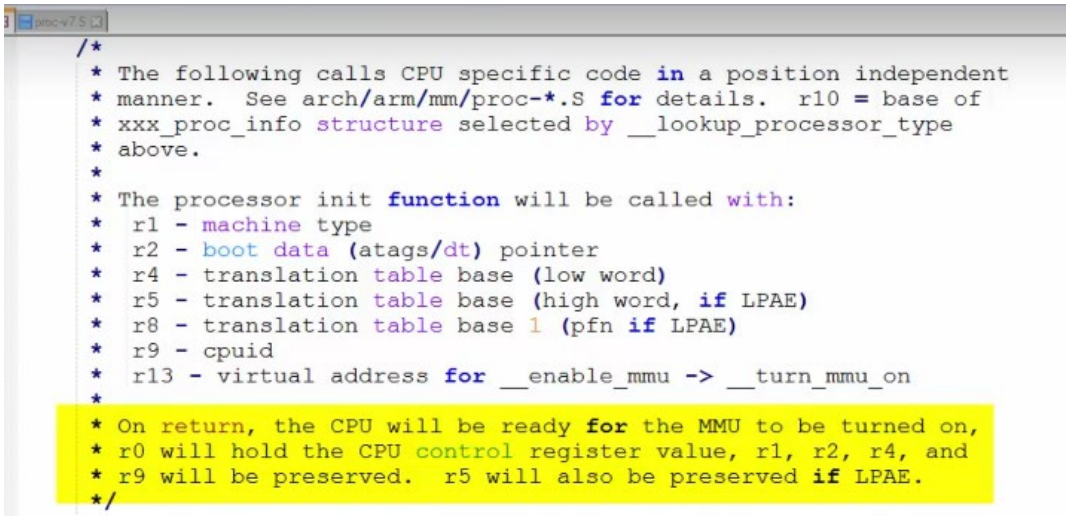


Рис. 1.4. – Вміст каталогу */arch/arm/mm*

Розглянемо рисунок 1.4. Тут можна побачити, що кожен процесор має свій, специфічний для процесора, файл ініціалізації. Спеціальні виклики процесора здійснюються для роботи з блоком управління пам’яттю, як пояснюється у коментарях на рис. 1.5.



```
/*
 * The following calls CPU specific code in a position independent
 * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
 * xxx_proc_info structure selected by __lookup_processor_type
 * above.
 *
 * The processor init function will be called with:
 *  r1 - machine type
 *  r2 - boot data (atags/dt) pointer
 *  r4 - translation table base (low word)
 *  r5 - translation table base (high word, if LPAE)
 *  r8 - translation table base 1 (pfn if LPAE)
 *  r9 - cpuid
 *  r13 - virtual address for __enable_mmu -> __turn_mmu_on
 *
 * On return, the CPU will be ready for the MMU to be turned on,
 * r0 will hold the CPU control register value, r1, r2, r4, and
 * r9 will be preserved.  r5 will also be preserved if LPAE.
 */
```

Рис. 1.5 – Коментар до файлу *proc*

Також зверніть увагу, що регістр r10 містить інформаційну структуру процесора, яка раніше була виявлена викликом *__lookup_processor_type*.

І цей коментар говорить, що натомість CPU буде готовий до ввімкнення блоку управління пам’яттю. Таким чином, це підтверджує, що всі специфічні для процесора виклики здійснюються для ініціалізації mmu перед його увімкненням.

Це означає, що перед тим, як передати контроль над загальним кодом Linux, ініціалізація MMU та увімкнення MMU є обов’язковими. Це обов’язок цих кодів, специфічних для архітектури.

Іншою важливою підпрограмою збірки є ввімкнення MMU, яка ініціалізує покажчики таблиці сторінок і вмикає MMU, щоб ядро могло почати працювати з підтримкою віртуальної адреси.

Після цього з заголовка файлу викликається функція *start_kernel()*.

Звідси управління потоком переходить до файлу *main.c* ядра Linux.

1.9 Запуск функції «init()»

Ядро запуску функцій — це функція C, реалізована в незалежному від архітектури загальному файлі, main.c, що знайдений за шляхом *init/main.c.*

На цьому етапі всі ініціалізації, що залежать від архітектури, закінчені, ваш CPU має підтримку MMU, і Linux готовий виконати незалежну від архітектури ініціалізацію ядра. *head.s* належить до специфічного для архітектури ядра Linux коду, в основному він виконує специфічні для CPU ініціалізації SOC. у цьому випадку ARM cortex-A8 є центральним процесором. І якщо ви проаналізуєте код *head.s*, його особливо цікавить пошук типу ЦП, наприклад, чи належить він до ARM 9 чи ARM 10 чи ARM cortex A8 тощо. І коли він дізнається про тип архітектури, він в основному ініціалізується MMU і створює початкові записи таблиці сторінок, а потім вмикає MMU процесора для підтримки віртуальної пам'яті, перш ніж передати керування файлу main.c ядра Linux, який є загальним.

Тепер перейдемо до функції *start_kernel()* у файлі *main.c* і дізнаємося, що вона робить.

Код main.c виконує всю роботу по запуску ядра Linux від ініціалізації першого потоку ядра, аж до монтування кореневої файлової системи та виконання першої прикладної програми Linux у просторі користувача.

Точкою входу в цей main.c є функція ядра *start(),* яка є дуже великою функцією, яка викликає безліч інших функцій ініціалізації. Ядро друкує рядок linux_banner, який буде відображено в журналі. За допомогою файлу *init/version.c* воно друкує версію Linux, версію компілятора, дату компіляції, номер збірки тощо.

Після цього ядро запуску виконує багато ранньої ініціалізації ядра Linux, наприклад, витягує аргументи командного рядка, надіслані завантажувачем, ініціалізація консолі для отримання повідомлень про помилки, ініціалізація керування пам'яттю, ініціалізація планувальника, ініціалізація таймера, висока роздільна здатність ініціалізація таймера, ініціалізація програмного IRQ тощо.

Таким чином, ініціалізуються різні підсистеми ядра Linux перед монтуванням кореневої файлової системи та запуском першої програми Linux. Можна дізнатися більше, переглянувши всі допоміжні функції одну за одною та дослідивши, що саме вони роблять. Зрештою, викликається функція під назвою *rest_init()*.

```
 */
kernel_thread(kernel_init, NULL, CLONE_FS);
numa_default_policy();
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
```

Рис. 1.6 – Створення потоків ядра

Розглянемо рисунок 1.6, на якому проводиться створення 2 потоків ядра; один називається *kernel_init*, а інший - *kthreadd*. Після цього тут запускається планувальник, і ядро переходить до циклу простою CPU, що є нескінченним циклом.

Це означає, що спочатку створиться потік ядра під назвою *kernel_init*, який отримає *PID* номер 1, а потім - інший потік ядра під назвою *kthreadd*, який отримає *PID* номер 2.

Потім потік ядра *kernel_init* виконає найпершу програму користувача, яка називається *init*. Це і є причиною, чому програма *init* успадковує свій номер *PID* потоків ядра, який дорівнює 1. Потік ядра *kthreadd* використовується для створення інших потоків ядра.

kernel_init — це потік ядра, який використовується для створення першої програми *Linux*, яка є *init*. І *kthreadd* також є ще одним потоком ядра, який використовується для створення інших потоків ядра.

```
static int __ref kernel_init(void *unused)
{
    int ret;

    kernel_init_freeable();
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    flush_delayed_fput();
```

The memory consumed by initialization functions so far will be reclaimed, because those functions are no longer needed and no one going to call them.

Рис. 1.7 – Програмний код функції *kernel_init()*

Функція *free_initmem*, що виділена на рисунку 1.7, повертає пам'ять, яка до цього часу використовується функцією ініціалізації. Тому що ці функції більше не потрібні і вся пам’ять функцій ініціалізації буде освоєна. Після завершення ініціалізації Linux більше не буде відбуватись виклик цих функцій. Це означає, що ця велика частина оперативної пам’яті може відновитись ядром. Її можна буде використовувати для інших цілей.

Після цього, відбувається намагання запустити програму ініціалізації. На рисунку 1.8 зображено логічний блок коду програми, що відповідає за неї.

```
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
    return 0;
```

Рис. 1.8 – Частина програми ініціалізації

Відбувається поступове намагання виконання ініціалізації з різних частин диску системи. Якщо не вдається виконати її в одному місці, відбувається перехід до наступних. А якщо ініціалізації не знайдено, то просто виконується програма оболонки, яка є в корені файлової системи і просто повертається інформація, як на рисунку 1.9.

```
    panic("No working init found.  Try passing init= option to kernel. "
          "See Linux Documentation/init.txt for guidance.");
}
```

Рис.1.9 – Повідомлення про відсутність робочого файлу ініціалізації

Якщо жодна з цих програм не знайдена, то видається помилка «*no working init found*». У випадку власного налаштування необхідно вказати шлях до програми ініціалізації.

На цьому моменті *Linux* нарешті запускає програму користувача. Він або запускає ініціалізацію, або якщо ініціалізації не знайдено, принаймні він запустить програму оболонки.

Контроль над запуском переходить від завантажувачів до завантажувача *Linux* до ядра *Linux* до запуску найпершої програми. І *init* — це програма, яка відповідає за запуск інших програм, як це зображено на рисунку 1.10.

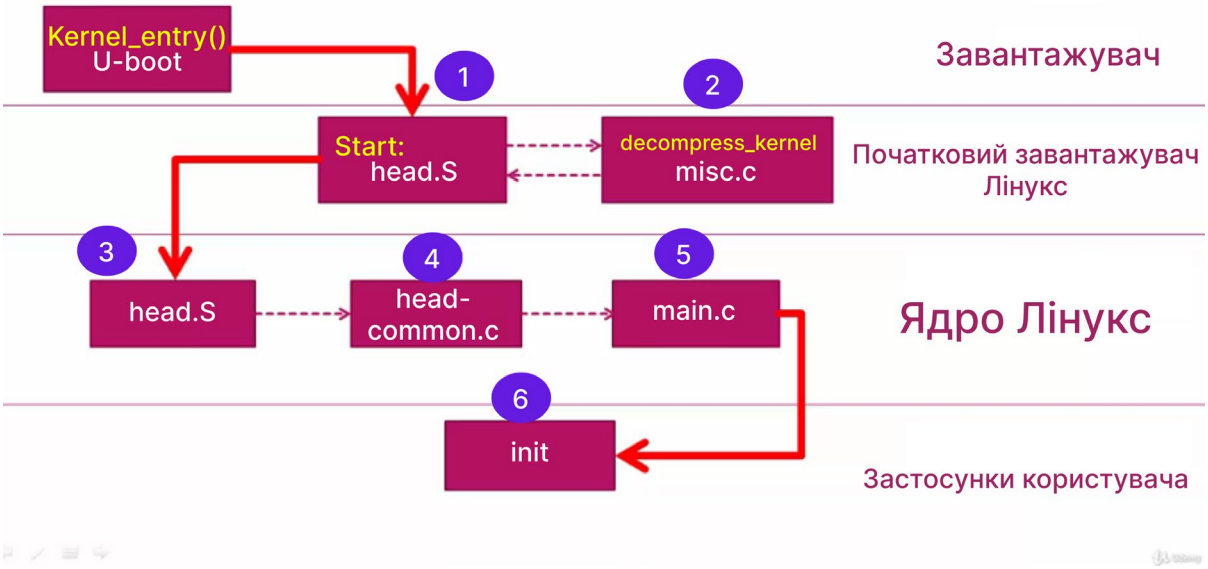


Рис.1.10 – Візуалізація переходів контролю над запуском

Тож, експериментуючи з усіма цими речами практично, створюючи нашу кореневу файлову систему та зберігаючи програму init, ми розглянули, як ми можемо використовувати програму init для запуску інших програм або інших служб ядра.

1.10 Дерево пристроїв Linux

Розберемося з деревом пристроїв Linux, для чого введено дерево пристроїв і яку проблему воно вирішує.

У нас є плата Beaglebone Black. На цій платі є SoC, який заснований на архітектурі ARM, і плата має пару вбудованих периферійних пристроїв, таких як трансивер Zigbee, serial flash, EEPROM, роз'єм SD-карти, інтерфейс USB тощо, які підключаються через інтерфейси шини, такі як USART, SPI, I2C, SDIO, USB тощо. Прикладом зображення підключених пристроїв є рисунок 1.11.

Цікаво те, що вбудовані периферійні пристрої не можна динамічно виявити. Це означає, що вбудовані периферійні пристрої, які підключаються до

SPI, I2C, SDIO, Ethernet тощо, не можуть самостійно оголосити про своє існування на платі операційній системі, як-от Linux. Чому?

Тому що такі інтерфейси, як I2C, SPI, SDIO тощо, не мають такого інтелекту для підтримки динамічного виявлення. Таким чином, навіть якщо ці периферійні пристрої є на платі, операційна система не має уявлення про їх налаштування. Але USB використовує їх по-різному. Коли ми підключаємо флеш-накопичувач або флеш-накопичувач для використання порту, він має інтелект, щоб динамічно повідомляти про свою присутність операційній системі, надсилаючи певну інформацію. Але ці пристрої не можуть цього зробити.

Рис.1.11 – Приклад периферійних пристроїв та їх інтерфейсів

Отже, всі ті, хто не може самостійно повідомити про свою присутність в операційній системі, називаються «пристроями платформи».

Тепер питання полягає в тому, як ми можемо змусити ядро Linux знати про ці пристрої платформи або периферійні пристрої, присутні на платі. Нам

потрібно якимось чином повідомити ядру про їх існування, оскільки вони самі цього зробити не можуть.

Одне з рішень, яке ми використовували, — це використовувати статичний шлях, тобто жорстке кодування цих деталей пристрою платформи у файлі, який називається файлом плати.

Наприклад, скажімо, створюємо файл плати для власної плати і коли ядро викликає функцію *board_init(),* що показана на рисунку 1.12, додається кожен пристрій платформи до підсистеми ядра з даними, що стосуються платформи.

Дані платформи — це не що інше, як структура даних, яка описує периферійний пристрій.

Файл плати є частиною ядра Linux, тому, при його зміні для додавання нових записів, нам доведеться перекомпілювати ядро. Тільки тоді зміни набудуть чинності.

При додаванні пристроїв з файлу плати, Linux дізнається про ці пристрої платформи, тож він оновлює список усіх пристроїв платформи, присутніх на платі. Коли відповідний драйвер завантажено, Linux викликає функцію «зондування» драйвера, і дані платформи будуть передані драйверу, а потім драйвер ініціалізує периферійний пристрій (рис. 1.12).

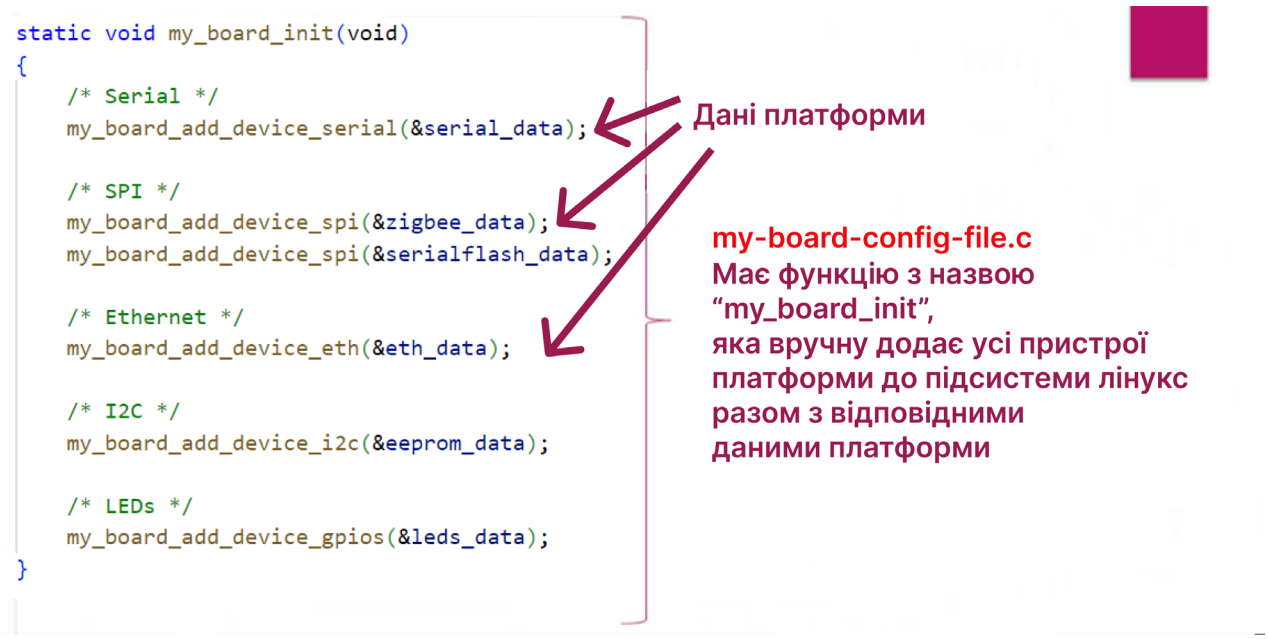


Рис.1.12 – Додавання платформенних пристроїв через жорстке кодування

Наприклад, припустимо, що пристрій платформи zigbee називається *zigbee100*. Відповідне ім'я драйвера також має збігатися з назвою пристрою платформи, тобто «*zigbee100.ko*»

Коли ви завантажуєте цей драйвер, який називається «*zigbee100.ko*», Linux пов'язує цей пристрій із цим драйвером. Linux негайно викликає функцію «зондування» цього драйвера, а потім драйвер подбає про ініціалізацію периферійного пристрою.

Різні плати мають різні вбудовані периферійні пристрої. Це означає, що плати не ідентичні, і для кожної у буде окремий файл. Це також означає, що для кожної плати у буде окремий образ ядра. Отже, цей образ ядра не працюватиме гладко на цій платі.

І постало питання необхідності відмови від залежності жорсткого перерахування пристроїв платформи від ядра Linux.

Спільнота ARM придумала ідею під назвою «Дерево пристроїв», яку також називають «Модель сплощеного дерева пристроїв».

У цьому випадку вони зробили замість жорсткого кодування обладнання у файлі плати ядра Linux, кожен постачальник плати повинен придумати файл під назвою DTS, що означає «Вихідний файл дерева пристроїв» або ж «Структура дерева пристроїв». Цей файл містить усі деталі, пов’язані з платою, написані з використанням певного попередньо визначеного синтаксису. Можна сказати, що цей файл складається з безлічі структур даних, які включатимуть усі деталі обладнання. Це означає кодування всіх деталей обладнання у файлі під назвою DTS.

Файл, що містить дані про пристрої буде скомпільовано за допомогою компілятора дерева пристроїв DTS. Це один із видів спеціального компілятора для перетворення цього файлу DTS у потік байтів, який є не що інше, як двійковий файл DTB.

DTB — це не що інше, як потік байтів, який кодує деталі апаратного забезпечення, отримане від компіляції DTS. Кожна плата матиме власний DTB.

При редагуванні файлу DTS, щоб додати новий запис, вам не потрібно компілювати ядро знову і знову, вам просто потрібно скомпілювати файл DTS і отримати новий DTB. Коли ядро завантажується, необхідно вказати ядру, де цей DTB знаходиться в пам’яті, щоб ядро Linux може завантажити цей файл DTB і витягти всі деталі апаратного забезпечення плати.

Підхід дозволяє не змінювати ядро при зміні плати. Нам потрібно змінити лише файл DTB.

РОЗДІЛ 2
ВСТУП В ПРАКТИКУМ. БАЗОВІ ОСНОВИ

2.1 Лабораторна робота 1
Вступ в LINUX. Робота в командному рядку

Теми для самостійного повторення:

1. Інсталяція операційної системиLinux на персональному компьютері.
2. Базові команди для роботи в командному рядку;
3. Менеджери пакетів.
4. Встановлення додаткових пакетів.
5. Оболонка Shell.
6. Bash скрипт.
7. Символічні посилання.

Useful links:

ArchWiki (archlinux.org) [6]
The Linux command line for beginners | Ubuntu [7]

Завдання 1.1:

Завдання до виконання лабораторної роботи видається індивідуально викладачем на аудиторному занятті і представляє собою **експрес-тест**, мета якого перевірити базові знання операційної системи Linux, які необхідні для виконання лабораторних робіт курсу «Архітектура 2. Процесори»

Контрольні запитання для перевірки базових знань і підготовки до експрес-тестування:

- Яка команда виводить директорію, в якій знаходиться користувач?
- Який із символів називається «конвеєром» і використовується для перенаправленні виводу одної програми на ввод іншої?

- Яка команда дозволяє вийти із редактора Vim?
- Яка команда дозволить встановити пакет?
- Як продивитись вміст директорії з правами доступу до файлів, датою, назвою, розміром файлу?
- Яка команда змінює дозволи прав доступу до файлів?
- Якою командою створюється символічне посилання (SimLink)?

2.2 Лабораторна робота 2
Введення в GIT

Контрольні запитання для перевірки базових знань та підготовки до екпрес-тестування:

- Чи відомі ще системи контролю версій окрім *Git*?
- Що таке GitHub?
- Що таке репозиторій Git?
- Що таке репозиторій Git?
- Як створити *Git*-репозиторій?
- Як перейменувати локальну гілку репозиторію?
- Призначення команди `git add`?
- За що відповідає команда `git config`?
- Що означає статус файлу new у виведенні команди `git status`?
- Що означає статус файлу `modified` у виведенні команди `git status`?
- Як скасувати `git add`?
- У чому різниця між `git add -A` та `git add`?
- Як скасувати дію команди `git add` на файл?
- Призначення команди `git status`?
- У чому полягає різниця між `git pull` та `git fetch`?
- Чим відрізняються команди `git push` та `git pull`?
- Призначення команди `git log`?

- Призначення команди `git show`?
- Що таке коміт?
- Як зробити коміт?
- З чого складається коміт у *Git*?
- Як об'єднати кілька окремих коммітів в один цільний коміт?
- Як можна скасувати коміт у *Git*, якщо він уже був опублікований?
- Як знайти список файлів, які змінилися у певному коміті?
- Як дізнатися, хто автор рядку у файлі, використовуючи систему Git?
- Як дізнатися, які зміни ми зробили локально?
- Як привести змінений файл до початкового стану (до зміни)?
- Що таке гілка у репозиторії Git?
- Чим відрізняються гілки `master` та `origin master`?
- Скільки всього гілок може бути у репозиторії?
- Як зробити гілку під назвою `my_branch`?
- Що зробить команда `git branch` без будь-якого параметра?
- Як зробити коміт для гілки `my_branch`?
- Що таке злиття двох гілок?
- Як вирішити конфлікт у Git?
- Як скасувати злиття гілок, якщо стався конфлікт?
- Чому бувають конфлікти при злитті гілок?
- Як додати нову директорію до Git?

Useful links:

Git - Book (git-scm.com) [8]

Git - Reference (git-scm.com) [9]

Завдання 2.1: Завдання містить кілька конфліктів злиття, які необхідно вирішити.

1). Перед виконанням завдання необхідно ознайомитися з розділами книги Pro Git [1.1]: «Вступ», «Основи Git», «Галуження в *Git*».

2). Клонуйте репозиторій, посилання, на який видане викладачем.

3). Використовуючи `rebase -i`, змініть місцями коміти: "fix truncation error" та "formatting: use tabs instead of spaces".
4). Підпишіть їх і створіть два патчі, використовуючи `git format-patch`.
5). Поєднайте "improve calculation accuracy" та "fix truncation error" в один коміт, використовуючи `rebase -i`. Підпишіть його.
Потім видаліть верхній коміт "formatting: use tabs instead of spaces", використовуючи `git rebase -i`.

6). Перейменуйте поточне віддалене сховище на "*github*". Додайте ще одне віддалене сховище, посилання, на яке видане викладачем в пункті 2 завдання 1.2, і назвіть його "*gitlab*".
7). Отримайте код з віддаленого сховища "*gitlab*" за допомогою "git fetch" та дослідіть його за допомогою `git log gitlab/master`.
8). Потім додайте коміт: "add a multiplication operation" використовуючи `git cherry-pick`.
9). Створіть власне виправлення на 5 – 15 рядків (довільні зміни).
10). Надішліть результат у власний репозиторій.

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- результати виконання
- два патчі, які були підготовлені "git format-patch" (окремим каталогом у репозиторії);
- висновки по роботі.

2.3 Лабораторна робота 3

Розроблення makefile та cmake

Теми для самостійного повторення

- Призначення утиліти make;
- Набір правил перетворення, які задаються в скрипті Makefile;
- Де повинен знаходитися Makefile;
- Використання правил, реквизитів, команд, пререквизитів;
- Використання залежностей в Makefile;
- Абстрактні цілі в Makefile;
- Автоматичні змінні.

Useful links:

GNU make [10]

Makefile cheatsheet (devhints.io) [11]

Завдання 3.1:

1). Напишіть програму на C++, яка приймає і аналізує будь-яку кількість параметрів командного рядка (для тих хто не знайомий з C++, можна звернути увагу на функцію `getopt()`). Повинна бути відповідність між довгими і короткими формами ключів, наприклад, `--help` і `-h` – це дві форми одного і того ж ключа. Дублікати повинні ігноруватися (ключ передається двічі або в різних формах). Для невідомого параметра необхідно надрукувати відповідне попередження. Після розбору повинні бути надруковані тільки унікальні аргументи. Визначте певний фіктивний обробник для кожного аргумента.

Приклади використання програми:

```
./cmd_parce_app -h                    // 1 унікальний ключ
./cmd_parce_app --help                // 1 унікальний ключ
./cmd_parce_app -l -h                 // 2 унікальних ключа
./cmd_parce_app -lh                   // 2 унікальних ключа
```

```
./cmd_parce_app --list --version -v -lh // 3 унікальних ключа
```

Виводом для останнього прикладу може бути:

```
Arg: List
Arg: Version
Arg: Help
```

Визначте свої власні довгі і короткі ключі для практики.

Додатково (не обов'язково).

* Деякі ключі можуть приймати числове значення в якості додаткового параметра:

`-V100` і `--value = 1000` – це різні форми одного і того ж параметра.

* Деякі ключі можуть приймати список числових значень, розділених комами, в якості додаткових параметрів:

`-L1,2,3,4 i --list = 1,2,3,4` – це різні форми одного і того ж параметра.

2). Написати *make* або *cmake* файл для збірки проекту.

3). Залити проект на *github* або *gitlab*.

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- репозиторій повинен містити файли з вихідним кодом, а також make/cmake файл за вашим вибором; в *README.md* файлі необхідно описати які ключі та їх параметри підтримує ваша реалізація;
- висновки по роботі.

Контрольні запитання для експерес-тестування:

- Як пов'язані між собою правила, реквизити, команди, пререквизити?
- Які ще конструкції, окрім правил може містити *Makefile*?
- Яка ціль є за замовчанням?
- Що виконує наступна команда

  `g++ main.cpp hello.cpp foo.cpp -o hello` ?

- З якого обов’язкового символу мають починатися рядки, що містять команди?
- Чи має пререквізіти правило з ціллю `clean` ("очищення"), що містить команди видалення?
- Для чого використається ціль `clean: rm -rf *.o hello`?
- Для чого призначені автоматичні змінні в makefile?

РОЗДІЛ 3
ПРАКТИКУМ ПО РОЗВЕРТАННЮ ЯДРА ОПЕРАЦІЙНОЇ СИСТЕМИ ДЛЯ ПРОЦЕСОРА АРХІТЕКТУРИ ARM

3.1 Лабораторна робота 4
Збирання прошивки ядра ОС LINUX із вихідних кодів

Мета: ознайомитися з основними етапами збирання виконуваних програм з вихідних кодів для ОС Linux. Здобути базові знання з використання системи збирання *kbuild* та основних підходів до розроблення *make* та *cmake* файлів. Навчитися скачувати вихідні коди завантажувальника *u-boot* та ядра *linux kernel* для цільових архітектур з репозиторіїв вихідних кодів. Навчитися виконувати аналіз вихідних кодів ПЗ для збирання програм на прикладі *make* файлу для завантажувальника *u-boot*. Навчитися збирати виконувані файли для прошивки ядра ОС на архітектурі *ARMv*7.

Структура лабораторної роботи:

- Основні етапи збирання програм з вихідних кодів u-boot в системі *kbuild*
- Завантаження вихідних кодів U-boot
- Встановлення кроскомпілятора gcc
- Конфігурація та сбирання виконуваних файлів SPL(MLO), U-boot images
- Аналіз make файлу u-boot
- Поняття про Linux Kernel Source Tree
- Конфігурація та сбирання виконуваних файлів zimage, *.dtb

Стислі теоретичні відомості та порядок виконання завдання лабораторної роботи

3.1.1 Основні етапи збирання програм з вихідних кодів u-boot в системі kbuild

Узагальнений алгоритм збирання та компілювання програми під Linux:

- Закачати програмне забезпечення
- Переключитися на потрібну гілку або тег (Checkout to desired branches ot tag)
- Прочитати рекомендації, як зібрати програмне забезпечення (Consult with README та INSTALL)
- Інсталювати залежності збирання, необхідні пакети для збирання (Install all build dependencies)
- Сконфігурувати необхідні змінні оточення для кроскомпіляції (Configure shall environment for cross-compiling)
- Конфігурація (Configure the SW for build with options you desire)
- Збирання (Build the SW)
- Інсталяція/прошивка (Install/flash the build SW)

3.1.2. Завантаження вихідних кодів завантажувальника U-boot

Useful links:

The U-Boot Documentation [12]

U-Boot Reference Manual [13]

https://source.denx.de/u-boot/u-boot [14]

Додаткова інструкція: https://git.comsys.kpi.ua [18]

Для завантаження U-boot треба перейти на FTP-сервер DENX, який підтримує U-Boot і завантажити необхідні вихідні коди [14].

Вихідний код U-boot підтримує різні архітектури та різні порти, які працюють із різними архітектурами, такими як ARM, PowerPC, AVR, ARC

тощо. Якщо ви перейдете до */arch*, ви можете знайти різні папки, де зберігаються пов'язані з архітектурою коди. А якщо ви перейдете до директорії */board*, вам буде представлено список плат, які підтримуються U-Boot.

Тепер давайте перейдемо до board/ti, де ви можете знайти різні папки для різних SoC.

У нашому випадку Beaglebone black працює на *AM335x SoC*, тому всі файли, пов'язані з платою, ви знайдете в папці *AM335x*. Перегляньте її вміст *ls*, де побачите файл *board.c*, що є лише одним файлом, який фактично займається ініціалізацією портів *AM335x*. Цей файл є загальним для всіх плат, які базуються на *AM335x*.

Крім того, у репозиторії знаходяться файли ініціалізації для конкретних плат та процесорів. Вам потрібно перейти до */arch/arm*, де розташовується папка під назвою */cpu*. Там знаходиться код ініціалізації для різних архітектур CPU(*core processor unit)*.

У нашому випадку нас цікавить *armv7*, тому що ми використовуємо *cortex a8*, оснований на заданій архітектурі.

Файл *start.S* у директорії *armv7* — це точка, де завантажувач ROM (read-only memory) передає керування SPL (second stage boot loader) — завантажувачу другого етапу. Він переходить від ініціалізації CPU до ініціалізації SoC та до ініціалізації плати.

Є ще одна цікава папка під назвою */configs*. У ній можна побачити різні файли конфігурації, пов'язані з різними портами. Ввівши *ls -1 | wc -l*, одержимо близько 12 сотень різних варіантів конфігураційних файлів.

Використовуйте файли конфігурації за замовчуванням, якщо ви не знаєте, як налаштувати вихідний код U-Boot для вашої плати. Коли ви застосовуєте цей файл конфігурації за замовчуванням до U-Boot, гарантується, що основні та необхідні функції U-Boot будуть доступні на вашій платі.

Щоб вивести всі файли конфігурації для Beaglebone black введіть

```
$ ls -l am33*
```

У нашому випадку ми будемо використовувати файл конфігурації за замовчуванням *am335x_boneblack_defconfig*.

Використовуйте файли конфігурації за замовчуванням, якщо ви не знаєте, як налаштувати вихідний код U-Boot для вашої плати. Коли ви застосовуєте цей файл конфігурації за замовчуванням до U-Boot, гарантується, що основні та необхідні функції U-Boot будуть доступні на вашій платі.

У нашому випадку Beaglebone працює з AM335x SOC. Отже, усі файли, пов'язані з платою, ви знайдете в папці AM335x, просто введіть ls, і тут ви побачите, що є *board.c*, є лише один файл C, який фактично займається ініціалізації портів на основі дефектів AM335x.

Таким чином, цей файл є загальним для всіх плат, які базуються на AM335x. І після цього ініціалізація, пов'язана з процесором, є в цій папці. Вам потрібно перейти до *arch/arm/* і тут є папка під назвою CPU. І, якщо ви перевірите тут, це різні архітектури процесора, вироблені ARM, наприклад, це для arm11, це для arm9, це для архітектури arm версії 7. Наприклад, кора ARM a8 або a9 заснована на архітектурі ARMV7. І останній, armv8, також підтримується і має міцну руку.

Таким чином, папка CPU насправді складається з пов'язаного з CPU коду ініціалізації для різних архітектур CPU.

Якщо ви можете знайти файл *start.s*, це точка, де завантажувач ПЗП передає управління SPL, тобто завантажувача другого етапу. Отже, все відбувається прямо з цього файлу. Він переходить від ініціалізації ЦП до ініціалізації SOC до ініціалізації плати. Якщо у вас є час і якщо ви зацікавлені, ви можете вивчити ці файли збірки.

Таким чином, ця конфігурація за замовчуванням використовується, коли ви не знаєте, як налаштувати вихідний код U-Boot для вашої плати. Отже, які функції вам потрібно ввімкнути, які функції потрібно вимкнути, ви, можливо, не знаєте. У цьому випадку розробник дошки надасть вам файл конфігурації за замовчуванням. Коли ви застосовуєте цей файл конфігурації за замовчуванням

до U-Boot, то гарантується, що основні та необхідні функції U-Boot будуть доступні на вашій платі.

Таким чином, весь пов'язаний з архітектурою код розділений у різні папки, пов'язані з архітектурою. Отже, якщо ваша архітектура ARM, то вам потрібно подивитися на *arch/ARM/* в arch arm, всі файли, пов'язані з процесором, є в *arch/arm/CPU/*, а потім ви повинні згадати архітектуру свого процесора, у нашому випадку це armv7. Отже, ця папка містить усі файли ініціалізації, що стосуються процесора.

Після цього *board/ti* містить усі пов'язані з платою функції ініціалізації, які базуються на SOC, створених ti.

Завдання 4.1

Для виконання завдання закачайте із репозиторію, посилання, на який виданий викладачем детальну інструкцію [18]. Виконайте наступні кроки:

1. Для завантаження програмного забезпечення для збирання вихідних кодів завантажувальника U-boot перейдіть на ресурс https://source.denx.de/u-boot/u-boot [14], що займається підтриманням U-boot, та завантажте останню версію завантажувальника в директорію `~/repos/`. Зклонувати репозиторій треба в директорію `~/repos/u-boot`.

/*

робота з git через термінал

Скачайте вихідний код U-Boot:

```
$ cd ~/repos
$ git clone https://gitlab.denx.de/u-boot/u-boot.git
$ cd u-boot
```

Переключимося на актуальну (визначену) гілку:

```
$ git checkout v2019.07
```

Також скористаємося такою серією патчів, встановивши попередньо утиліту curl:

```
$ sudo apt install curl
$ curl https://patchwork.ozlabs.org/series/130450/mbox/ | git am
```

*/

3.1.3. Встановлення toolchains

Toolchain — це набір інструментів для крос-компіляції програм, що мають відмінну архітектуру від архітектури хоста. Ми будемо використовувати тулчейни, аби створювати програми для пристроїв на ARM архітектурах на персональному комп'ютері x86_64 (хост). Зазвичай тулчейн складається з таких основних компонентів:

- компілятор C, лінкер і пов'язані інструменти (gcc, ld, as, cpp)
- libc (та інші пов'язані бібліотеки)
- налагоджувач (GDB)
- інструменти для роботи з крос-компільованими бінарними файлами (readelf, objdump тощо)

Існують два види *Toolchains*:

1. *Bare Metal* (*EABI*) *toolchain*: для створення програм, які запускаються без ОС

2. *Linux toolchain*: для створення програм середовища користувача, які запускаються під ОС Linux

Виконувані файли тулчейнів зазвичай мають префікс, наприклад, «*arm-eabi-*» або «*arm-linux-gnueabihf-*». Сам префікс містить так званий кортеж:

- повна форма: arch — постачальник — OS — libc/ABI
- скорочена форма: arch — OS — libc/ABI
- ще більш скорочена форма: arch — libc/ABI

де:

- arch — це ваша цільова архітектура

- vendor — це назва компанії, яка розробила цей тулчейн
- ОС — це назва ОС, на якій повинна працювати ваша крос-компільована програма
- libc/ABI вказує, які бібліотеки C і ABI надаються цим тулчейном

Linux toolchain використовується для створення user-space програм, що захоплюють середовище Linux. Він містить повну версію GNU libc (glibc), яка спирається на інтерфейс ядра системного виклику (syscall kernel interface). Нам він знадобиться для створення *rootfs* (BusyBox) та user-space інструментів.

Завдання 4.2

Завантажте Linux toolchain для процесорів Cortex-A (32-bit)

Розпакуйте завантажений тулчейн у */opt*:

```
$ sudo tar xJvf gcc-arm-8.3-2019.03-x86_64-arm-linux-
gnueabihf.tar.xz \-C /opt/
```

Другий крок — додати нову змінну середовища.

Для використання тулчейну в середовищі оболонки (shell) нам потрібно додати шлях до каталогу bin/ завантаженого тулчейна до змінної середовища `$PATH`.

Крім того, зазвичай потрібно надавати змінну `$CROSS_COMPILE` з рядком префікса вашого тулчейна (це буде далі використовуватися в Makefile скомпільованого проєкту для запуску правильного інструменту, наприклад, `${CROSS_COMPILE}gcc`).

Для додавання нової змінної перейдіть у свій домашній каталог і відкрийте файл під назвою *bashrc* за допомогою текстового редактора vi або будь-якого іншого.

```
$ vi /home/$USERNAME/.bashrc
```

Прокрутіть униз файлу, вставте команду експорту змінної (нижче) та збережіть файл.

/

для редактора vi: Прокрутіть униз файлу, натисніть «i» для переходу в режим вводу (insert mode), вставте команду експорту змінної, натисніть ***Esc*** *та збережіть файл. Для цього увійдіть в режим останньої строки (last line mode) натиснувши «:» та введіть* ***wq*** *(записати файл та покинути редактор vi) чи* ***q!*** *(вийти з редактора vi без збереження змін), якщо у вас щось не вийшло. Виконайте вихід, натиснувши* ***Enter****.*

**/*

Після редагування файлу введіть

```
$ source /home/$USERNAME/.bashrc
```

Ви повинні виконати цю команду, оскільки зміни, які ви щойно внесли у файл *.bashrc*, не набудуть чинності, доки ви не запустите цю команду.

Перевірка коректності встановлення змінної:

```
$ gcc --version
```

3.1.4. Конфігурація та генерація вихідних кодів SPL, MLO, U-boot

Завдання 4.3

Першим кроком є видалення всіх файлів у поточному каталозі (або створені цим make-файлом), які створені шляхом налаштування або створення програми. Якщо ваша папка вже чиста, то distclean нічого не видалить.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
```

Другим кроком є застосування конфігурації плати за замовчуванням для U-Boot. Вам не слід редагувати *defconfig* вручну. Ви зіткнетеся з великою кількістю проблем. Цей файл можна редагувати за допомогою *menuconfig*.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-/
/am335x_boneblack_defconfig
```

Третій крок — запустити *menuconfig*. *Menuconfig* використовується для виконання будь-яких налаштувань над конфігурацією за замовчуванням.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

Наприклад, за допомогою цієї опції можна змінити архітектуру (наразі вибрана архітектура ARM):

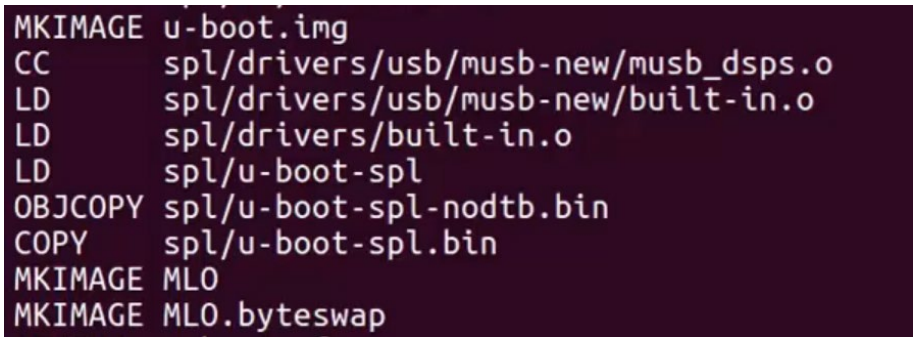
Запустіть *menuconfig*, лише якщо вам потрібні більш конкретні налаштування.

Четвертий крок — компіляція.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- j4
// -j<кількість ядер вашої системи>
```

Для цього ви повинні використовувати цю команду разом із прапорцем *-j*. Якщо ви не згадаєте цей прапор, для компіляції буде використано лише одне ядро.

Коли компіляція закінчиться, ви побачите *u-boot.img*, *MLO* і *u-boot-spl.bin*, також створений разом із *u-boot.img*:



3.1.5. Аналіз make файлу u-boot

Useful links:

Makefile U-Boot / U-Boot GitLab (denx.de) [15]

Розглянемо певні частинки файлу Makefile створеного для роботи з u-boot.

У залежності від апаратного імені машини `shell uname -m` визначається хост-архітектура:

```
# Determine target architecture for the sandbox
include include/host_arch.h
ifeq ("", "$(CROSS_COMPILE)")
 MK_ARCH="${shell uname -m}"
else
 MK_ARCH="${shell echo $(CROSS_COMPILE) | sed -n 's/^\s*\([^\/]*\/\)*\([^-]*\)-\S*/\2/p'}"
endif
unexport HOST_ARCH
ifeq («x86_64», $(MK_ARCH))
 export HOST_ARCH=$(HOST_ARCH_X86_64)
else ifneq (,$(findstring $(MK_ARCH), «i386» «i486» «i586» «i686»))
 export HOST_ARCH=$(HOST_ARCH_X86)
else ifneq (,$(findstring $(MK_ARCH), «aarch64» «armv8l»))
 export HOST_ARCH=$(HOST_ARCH_AARCH64)
else ifneq (,$(findstring $(MK_ARCH), «arm» «armv7» «armv7l»))
 export HOST_ARCH=$(HOST_ARCH_ARM)
else ifeq («riscv32», $(MK_ARCH))
 export HOST_ARCH=$(HOST_ARCH_RISCV32)
else ifeq («riscv64», $(MK_ARCH))
 export HOST_ARCH=$(HOST_ARCH_RISCV64)
endif
undefine MK_ARCH
```

Перевірка відповідності розміру бінарного файлу встановленим лімітам:

```
# Check ths size of a binary:
# Args:
# $1: File to check
# #2: Size limit in bytes (decimal or 0xhex)
define size_check
```

```
	actual=$$(wc -c $1 | awk '{print $$1}'); \
	limit=$$(printf "%d» $2); \
	if test $$actual -gt $$limit; then \
		echo "$1 exceeds file size limit:" >&2; \
		echo " limit: $$(printf%#x $$limit) bytes» >&2; \
		echo " actual: $$(printf%#x $$actual) bytes» >&2; \
		echo " excess: $$(printf%#x $$((actual – limit))) bytes» >&2;\
		exit 1; \
	fi
endef
export size_check
```

Конфігурація *.img* файлів:

```
	# TPL + SPL
ifeq ($(CONFIG_SPL)$(CONFIG_TPL),yy)
MKIMAGEFLAGS_u-boot-tpl-rockchip.bin = -n $(CONFIG_SYS_SOC) -T rksd
tpl/u-boot-tpl-rockchip.bin: tpl/u-boot-tpl.bin FORCE
	$(call if_changed,mkimage)
idbloader.img: tpl/u-boot-tpl-rockchip.bin spl/u-boot-spl.bin FORCE
	$(call if_changed,cat)
else
MKIMAGEFLAGS_idbloader.img = -n $(CONFIG_SYS_SOC) -T rksd
idbloader.img: spl/u-boot-spl.bin FORCE
	$(call if_changed,mkimage)
endif
```

Процес підчистки файлів після запуску різноманітних інструкцій. Відбувається за допомогою виклику `make` разом з такими назвами інструкцій `clean/mrproper/distclean`

```
	###
# Cleaning is done on three levels.
# make clean Delete most generated files
# Leave enough to build external modules
# make mrproper Delete the current configuration, and all generated files
# make distclean Remove editor backup files, patch leftover files and the like

# Directories & files removed with 'make clean'
```

```
CLEAN_DIRS                  +=                  $(MODVERDIR)                  \
       $(foreach       d,       spl       tpl,       $(patsubst%,$d/%,       \
                   $(filter-out  include,  $(shell  ls  -1  $d  2>/dev/null))))

CLEAN_FILES  +=  include/bmp_logo.h  include/bmp_logo_data.h  tools/version.h  \
       u-boot*       MLO*       SPL       System.map       fit-dtb.blob*       \
       u-boot-ivt.img.log    u-boot-dtb.imx.log    SPL.log    u-boot.imx.log    \
       lpc32xx-*    bl31.c    bl31.elf    bl31_*.bin    image.map    tispl.bin*    \
       idbloader.img     flash.bin     flash.log     defconfig     keep-syms-lto.c

#     Directories     &     files     removed     with     'make     mrproper'
MRPROPER_DIRS    +=    include/config    include/generated    spl    tpl    \
            .tmp_objdiff                doc/output                include/asm

#   Remove   include/asm   symlink   created   by   U-Boot   before   v2014.01
MRPROPER_FILES  +=.config.config.old  include/autoconf.mk*  include/config.h  \
             ctags  etags  tags  TAGS  cscope*  GPATH  GTAGS  GRTAGS  GSYMS  \
             drivers/video/fonts/*.S                                 include/asm

#  clean  -  Delete  most,  but  leave  enough  to  build  external  modules
#
clean:                        rm-dirs:=                        $(CLEAN_DIRS)
clean:                       rm-files:=                       $(CLEAN_FILES)

clean-dirs   :=      $(foreach      f,$(u-boot-alldirs),$(if      $(wildcard
$(srctree)/$f/Makefile),$f))

clean-dirs:=          $(addprefix          _clean_,          $(clean-dirs))

PHONY          +=          $(clean-dirs)          clean          archclean
$(clean-dirs):
      $(Q)$(MAKE)             $(clean)=$(patsubst             _clean_%,%,$@)

clean:                                                          $(clean-dirs)
      $(call                                                      cmd,rmdirs)
      $(call                                                     cmd,rmfiles)
      @find  $(if  $(KBUILD_EXTMOD),  $(KBUILD_EXTMOD),.)  $(RCS_FIND_IGNORE)  \
            \(-name   '*.[oas]'   -o   -name   '*.ko'   -o   -name   '.*.cmd'   \
            -o   -name   '*.ko.*'   -o   -name   '*.su'   -o   -name   '*.pyc'   \
            -o   -name   '.*.d'   -o   -name   '.*.tmp'   -o   -name   '*.mod.c'   \
            -o     -name     '*.lex.c'     -o     -name     '*.tab.[ch]'     \
```

```
            -o              -name              '*.asn1.[ch]'              \
            -o    -name    '*.symtypes'    -o    -name    'modules.order'    \
            -o    -name    modules.builtin    -o    -name    '.tmp_*.o.*'    \
            -o -name 'dsdt.aml' -o -name 'dsdt.asl.tmp' -o -name 'dsdt.c' \
            -o  -name  '*.efi'  -o  -name  '*.gcno'  -o  -name  '*.so'  \)  \
            -type       f       -print       |       xargs       rm       -f

#    mrproper    —    Delete    all    generated    files,    including.config
#
mrproper:           rm-dirs:=           $(wildcard           $(MRPROPER_DIRS))
mrproper:           rm-files:=          $(wildcard           $(MRPROPER_FILES))
mrproper-dirs:=                $(addprefix                _mrproper_,scripts)

PHONY        +=        $(mrproper-dirs)        mrproper        archmrproper
$(mrproper-dirs):
      $(Q)$(MAKE)            $(clean)=$(patsubst            _mrproper_%,%,$@)

mrproper:                        clean                        $(mrproper-dirs)
      $(call                                                        cmd,rmdirs)
      $(call                                                       cmd,rmfiles)
      @rm                      -f                      arch/*/include/asm/arch

#                                                                  distclean
#
PHONY                              +=                              distclean

distclean:                                                          mrproper
      @find            $(srctree)            $(RCS_FIND_IGNORE)            \
            \(-name   '*.orig'   -o   -name   '*.rej'   -o   -name   '*~'   \
            -o   -name   '*.bak'   -o   -name   '#*#'   -o   -name   '.*.orig'   \
            -o   -name   '.*.rej'   -o   -name   '*%'   -o   -name   'core'   \
            -o            -name            '*.pyc'            \)            \
            -type       f       -print       |       xargs       rm       -f
      @rm               -f               boards.cfg               CHANGELOG

backup:
      F=`basename               $(srctree)`;               cd.;               \
      gtar --force-local -zcvf `LC_ALL=C date "+$$F-%Y-%m-%d-%T.tar.gz»` $$F
```

3.1.6 Поняття про Linux Kernel Source Tree (Device Tree Database)

Useful links:

About the Device Tree [16]

Розглянемо процедуру компіляції вихідного коду Linux для створення двійкового файлу ядра Linux. Перед цим давайте завантажимо вихідний код ядра Linux.

Давайте спочатку дослідимо структуру каталогів джерела ядра Linux, щоб зрозуміти, як Linux працює на різних архітектурах процесора, таких як x86, arm, powerPC, AVR тощо.

Вихідний код для роботи з усіма цими трьома компонентами, тобто процесором, SOC і платою, систематично розміщений у дереві джерел ядра Linux у різних каталогах.

Оскільки наша архітектура заснована на ARM, усі коди, пов'язані з обробкою тих пристроїв, які засновані на цій архітектурі, присутні в каталозі, який називається *ARCH/arm*.

Кожен постачальник SOC має свій машинний каталог, щоб зберігати свої вихідні коди, специфічні для SOC, які необхідні для роботи з SOC, виробленими постачальником.

Тепер цей специфічний для машини каталог складається з двох частин. Одина з них пов'язана з платою, а інша – із загальним кодом, спільним для машини.

Файл плати є вихідним файлом, який пояснює різні периферійні пристрої на вашій платі за межами SOC.

Цей файл зазвичай містить реєстраційні функції різних пристроїв, які знаходяться поза межами SoC, як-от Ethernet phy, eeproms, світлодіоди, РК-дисплеї тощо. Постачальник використовує цей файл для реєстрації різних периферійних пристроїв плати в підсистемі Linux. І загальний код машини містить різні вихідні файли, які містять допоміжні функції для ініціалізації та

роботи з різними вбудованими периферійними пристроями SOC, а ці коди є загальними для всіх SOC, що мають загальний IP для периферійних пристроїв.

Наприклад, AM335x SOC належить до сімейства SOC Sitara від Texas Instruments.

Але все одно в Linux немає каталогу під назвою Machine-Sitara, але є каталог під назвою Machine-DaVinci. Це тому, що SoC DaVinci призначені для різних ринкових цілей.

Ці SOC призначені для відеододатків. IP-адреси периферійних пристроїв абсолютно різні, і для їх обробки потрібні різні коди.

У випадку AM335x, незважаючи на те, що SoC належить до іншого сімейства, ніж OMAP, більшість периферійних IP-адрес є такими ж, як сімейство SoC OMAP2/3/4. Тому вони все ще посилаються на каталог OMAP2 машини для AM335x.

3.1.7. Вихідний код ядра Linux

На початку свого існування ядро Linux містило вбудовані у нього конфігурації різних плат та їхніх пристроїв. Для кожної плати розроблювалась власна конфігурація. Зі збільшенням кількості та варіативності систем стало зрозуміло, що подальша підтримка кожної, шляхом розширення ядра, не має майбутнього через збільшення розмірів ядра, та постійну роботу його розробників, спрямовану на додавання у ядро підтримки чергової SoC.

Таким чином недоцільність такого підходу призвела до розробки нового - дерева пристроїв (Device Tree).

За нового підходу завантажувач системи використовує два файли: образ ядра та двійковий скомпілбований файл, що містить опис апаратного забезпечення. Ядро, при запуску системи, зчитує надану інформацію про обладнання та відповідним чином конфігурує систему.

3.1.8. Конфігурація та генерація zimage, *.dtb

Useful links:

U-boot documentation [12]

U-boot reference manual [13]

Додаткова інструкція: https://git.comsys.kpi.ua [18]

Завдання 4.4

1). Завантажити вихідний код ядра Linux з git репозиторію:

```
$ cd ~/repos
$ git clone
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-
stable.git
$ cd linux-stable
```

2). Переключитися на визначену гілку:

```
$ git checkout linux-4.19.y
```

3). Сконфігурувати середовище тулчейну в за допомогою змінних оточення:

```
$ export PATH=/opt/gcc-arm-8.3-2019.03-x86_64-arm-
eabi/bin:$PATH
$ export CROSS_COMPILE='ccache arm-eabi-'
$ export ARCH=arm
```

Ми використовуватимемо конфігураційний файл *multi_v7_defconfig* за основу, який є загальновживаним для плат ARMv7. Доповнимо його деякими опціями ядра для плати BeagleBone Black. Для цього створимо фрагмент майбутньої конфігурації:

```
$ mkdir fragments
$ vim fragments/bbb.cfg
```

Наповнимо його наступним чином та збережемо файл:

```
# Use multi_v7_defconfig as a base for merge_config.sh
# --- USB ---
# Enable USB on BBB (AM335x)
CONFIG_USB_ANNOUNCE_NEW_DEVICES=y
CONFIG_USB_EHCI_ROOT_HUB_TT=y
CONFIG_AM335X_PHY_USB=y
CONFIG_USB_MUSB_TUSB6010=y
CONFIG_USB_MUSB_OMAP2PLUS=y
CONFIG_USB_MUSB_HDRC=y
CONFIG_USB_MUSB_DSPS=y
CONFIG_USB_MUSB_AM35X=y
CONFIG_USB_CONFIGFS=y
CONFIG_NOP_USB_XCEIV=y
# For USB keyboard and mouse
CONFIG_USB_HID=y
CONFIG_USB_HIDDEV=y
# For PL2303, FTDI, etc
CONFIG_USB_SERIAL=y
CONFIG_USB_SERIAL_PL2303=y
CONFIG_USB_SERIAL_GENERIC=y
CONFIG_USB_SERIAL_SIMPLE=y
CONFIG_USB_SERIAL_FTDI_SIO=y
# For USB mass storage devices (like flash USB stick)
CONFIG_USB_ULPI=y
CONFIG_USB_ULPI_BUS=y
# --- Networking ---
CONFIG_BRIDGE=y
# --- Device Tree Overlays (.dtbo support) ---
CONFIG_OF_OVERLAY=y
```

Об’єднаємо самостійно створений фрагмент з базовим файлом конфігурації:

```
$ ./scripts/kconfig/merge_config.sh \
arch/arm/configs/multi_v7_defconfig fragments/bbb.cfg
```

4). Скомпілювати образ ядра, двійковий файл дерева пристроїв (*.dtb*) та модулі ядра:

```
$ make -j4 zImage modules am335x-boneblack.dtb
```

Отримаємо наступні файли:

- *arch/arm/boot/zImage*
- *arch/arm/boot/dts/am335x-boneblack.dtb*

Після закінчення процесу компіляції ядра Linux буде згенерованим стислий образ ядра *zImage* в директорії */arch/arm/boot*, і там же в директорії */dts* знаходитиметься скомпільоване дерево пристроїв *am335x-boneblack.dtb*. Скомплектований FAT розділ SD карти міститиме:

- *MLO* – завантажувач другого порядку
- *u-boot.img* - завантажувач третього порядку
- *uEnv.txt* - додаткові команди завантажувача
- *zImage* - образ ядра Linux
- *am335x-boneblack.dtb* - скомпіловане дерево пристроїв

Завдання 4.5. Створення мінімалістичної файлової системи Busybox rootfs.

Завдання виконати згідно інструкції, що розміщується в репозиторії [18]. Виконати наступні дії:

1). Зайдіть до репозиторію проєкта і закачайте файл інструкції.

2). Зклонуйте репозиторій з вихідними кодами утілити *Busybox* в директорію `repos/busybox/`;

3). Перейдіть до останньої стабільної гілки;

4). Налаштуйте необхідні змінні окруження та використайте Linux toolchains для збирання виконуваних кодів;

5). Зконфігуруйте *Busybox* за допомогою конфігуратора `defconfig;`

6). Інсталлюйте *Busybox* в директорію `_install/`;

7). Створіть каталоги, необхідні для заповнення *rootfs* файлами ініціалізації та відсутніми вузлами:

```
$ mkdir -p
_install/{boot,dev,etc\/init.d,lib,proc,root,sys\/kernel\/deb
ug,tmp}
```

8). Створіть `init script ( _install/etc/init.d/rcS ):`

```
#!/bin/sh
mount -t sysfs none /sys
mount -t proc none /proc
mount -t debugfs none /sys/kernel/debug
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

Зробіть файл `_install/etc/init.d/rcS` виконуваним за допомогою команди `$ chmod`;

9). Зробіть символічне посилання на систему ініціалізації в кореневому каталозі `bin/busybox` (щоб ядро могло її запустити):

```
$ ln -s bin/busybox _install/init;
```

10). Заповнити директорію `/boot` файлової системи файлами ядра Linux:

```
$ cd _install/boot
$ cp ~/repos/linux-stable/arch/arm/boot/zImage .
$ cp ~/repos/linux-stable/arch/arm/boot/dts/am335x-
boneblack.dtb .
$ cp ~/repos/linux-stable/System.map .
$ cp ~/repos/linux-stable/.config ./config
$ cd –
```

11). Заповнити `/lib.` Виконайте динамічне лінкування *BusyBox*. Налаштуйте змінні оточення для Linux toolchains. Зкопіюйте модулі ядра до `lib/modules/$(uname -r)/` в *rootfs:*

```
$ cd ~/repos/linux-stable
$ export INSTALL_MOD_PATH=~/repos/busybox/_install
```

```
$ export ARCH=arm
$ make modules_install
$ cd -
```

Оскільки *BusyBox* було динамічно злінковано, нам потрібно скопіювати системні бібліотеки з toolchain у `lib/` каталог. Двійковий файл *busybox* залежить від *libc.so* , *libm.so* та *ld-linux.so* (див. інструкцію).

12). Заповнити `/etc`. Виконайте конфігурацію *mdev* (підтримка *hotplug* модуля). У таких складних системах, як Debian, події *hotplug* (і багато іншого) обробляються пристроєм менеджеру пристроїв *udev*. Для *BusyBox* є більш спрощений інструмент *mdev* для цього. Конфігурується в директорію `/etc.`

Завдання 4.6. Завантаження ядра *Linux* файлової системи *rootfs* на емуляторі *QEMU* процесорного ядра *ARM*.

1). Ви можете перевірити створені виконувані коди ядра Linux та файлової системи rootfs розробленої вбудованої системи фактично без плати BeagleBone Black, використовуючи емулятор QEMU. QEMU це програмний емулятор або віртуальна машина для різних систем, в тому числі і для процесора ARM.

2). Інсталлюйте *QEMU*:

```
$ sudo apt install qemu-system-arm
```

3). Запустіть ядро *Linux* та файлової системи *rootfs* на емуляторі згідно інструкції:

4). Ви побачите журнал ядра з запущеної машини. Ядро розпакує *rootfs*, а потім запустить `/init` процес.

5). Після завершення процесу завантаження віртуальної машини натисніть *Enter*, щоб побачити командний рядок. В якості експериментів виконайте команди Linux в консолі, щоб переконатися, що ви створили прошивку ядра на процесорі ARM і дослідити систему, наприклад:

```
/ # uname -a
/ # ls -l
/ # dmesg | grep init
```

```
/ # busybox --help | head -15
/ # poweroff
```

Додайте власні команди, внесіть результати виконуваних команд в протокол з результатами виконання лабораторної роботи. Для обробки виключних ситуацій зверніться до інструкції.

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- звіт повинен містити скриншоти на вміст директорії з файлами вихідних кодів , журналу ядра операційної системи запущеної віртуальної машини, результати виконання команд Linux в консолі запущеного ядра Linux на віртуальній машині;
- висновки по роботі.

Контрольні запитання

- Вкажіть основні етапи розвертання ОС Linux (Linux bootstrap).
- Типи toolchains.
- Вкажіть, які набори інструментів для кроскомпіляції містять toolchains?
- Якиа функція завантажувальника?
- В якому файлі визначаються рівні виконання (Run Level) програмою init під час розвертання Linux?
- Який номер PID має init процес?
- Який дистрибутив використовується для розгортання rootfs в убудованих ситсемах?
- В які основні етапи виконується збирання виконуваної програми із вихідних кодів під ОС Linux?
- Призначення менеджеру пристроїв mdev?
- Опишіть призначення каталогів ОС Linux.

- Яка команда використовується для передавання змінної оточення процесу, що запускається?
- Яка команда використовується для перегляду значення змінної оточення?
- Для передавання яких налаштувань ОС використовується змінна оточення $PATH?

3.2. Лабораторна робота 5

ПРОШИВКА ПЛАТИ. РОБОТА З ІНТЕФЕЙСОМ UART

3.2.1. Способи прошивки плати

Ця плата для програмування вбудованих систем дозволяє проводити завантаження системи за допомогою великого розмаїття джерел завантаження:

1. Flash NAND
2. NOR Flash (XIP – (англ. eXecute In Place) виконання на місці)
3. USB
4. eMMC
5. SD-карта
6. Ethernet
7. UART
8. SPI

Це означає, що можна зберігати завантажувальні образи в будь-якій, із зазначених вище, пам’яті або периферійному пристрої.

Table 26-7. SYSBOOT Configuration Pins[4]

SYSBOOT[15:14]	SYSBOOT[13:12]	SYSBOOT[11:10]	SYSBOOT[9]	SYSBOOT[8]	SYSBOOT[7:6]	SYSBOOT[5]	SYSBOOT[4:0]	Boot Sequence			
For all boot modes: Crystal Frequency	**For all boot modes**: Set to 00b for normal operation	**For XIP boot**: Muxed or non-muxed device **For NAND boot**: must be 00b	**For NAND and NANDI2C boot**: NAND ECC **For Fast External Boot**: must be 0b	**For XIP boot**: Bus width	**For EMAC boot**: PHY mode	**For all boot modes**: CLKOUT1 output enabled/disabled on XDMA_EVENT_INTR0					
CONTROL_STATUS[23:22]	CONTROL_STATUS[21:20]	CONTROL_STATUS[19:18]	CONTROL_STATUS[17]	CONTROL_STATUS[16]	CONTROL_STATUS[7:6]	CONTROL_STATUS[5]	CONTROL_STATUS[4:0]				
								1st	2nd	3rd	4th
							00000b	Reserved			
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)	**For XIP boot**: 00b = non-muxed device 10b = muxed device x1b = reserved	Don't care for ROM code[3]	0 = 8-bit device 1 = 16-bit device	Don't care for ROM code	0 = CLKOUT1 disabled 1 = CLKOUT1 enabled	00001b	UART0	XIP w/ WAIT[1] (MUX2)[2]	MMC0	SPI0
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)	**For NAND boot**: must be 00b	0 = ECC done by ROM 1 = ECC handled by NAND	Don't care for ROM code	Don't care for ROM code	0 = CLKOUT1 disabled 1 = CLKOUT1 enabled	00010b	UART0	SPI0	NAND	NANDI2C
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)	**For XIP boot**: 00b = non-muxed device 10b = muxed device x1b = reserved	Don't care for ROM code	0 = 8-bit device 1 = 16-bit device	Don't care for ROM code	0 = CLKOUT1 disabled 1 = CLKOUT1 enabled	00011b	UART0	SPI0	XIP (MUX2)[2]	MMC0
00b = 19.2MHz 01b = 24MHz 10b = 25MHz	00b (all other values reserved)	**For XIP boot**: 00b = non-muxed device 10b = muxed	0 = ECC done by ROM 1 = ECC handled by	0 = 8-bit device 1 = 16-bit device	Don't care for ROM code	0 = CLKOUT1 disabled 1 = CLKOUT1 enabled	00100b	UART0	XIP w/ WAIT[1] (MUX1)	MMC0	NAND

Рисунок 2.1 – Конфігурація пінів для завантаження системи [3]

Розглянемо рисунок 2.1. Зосередимося на SYSBOOT [4:0]. “SYSBOOT” є одним із регістрів цього SOC, і його перші п’ять бітів визначають порядок завантаження. Розглянемо приклад.

SYSBOOT [4:0] = 00000b (зарезервовано, не можна використовувати цю конфігурацію). Беремо наступний біт.

SYSBOOT [4:0] = 00001b. Після RESET, якщо SYSBOOT [4:0] = 00001b, SOC спробує спочатку завантажитися з UART0, якщо це не вдасться, потім спробує завантажитися з XIP, якщо також спіткатиме невдача, тоді він спробує завантажитися з MMC0, і в останню чергу він намагатиметься завантажитися з SPI0. Невдача приводить до виводу повідомлення про помилку та зупинки завантаження.

Розглянемо будову системи на чіпі AM335x, що зображена на рисунку 2.2. При скиданні системи на чіпі, код, збережений у її ПЗП, запускається першим. Код, що зберігається в ПЗП, називається завантажувачем ПЗП, він програмується в ПЗП SOC під час вироблення мікросхеми і не може бути зміненим, бо знаходиться в ROM пам’яті, яка слугує лише для читання.

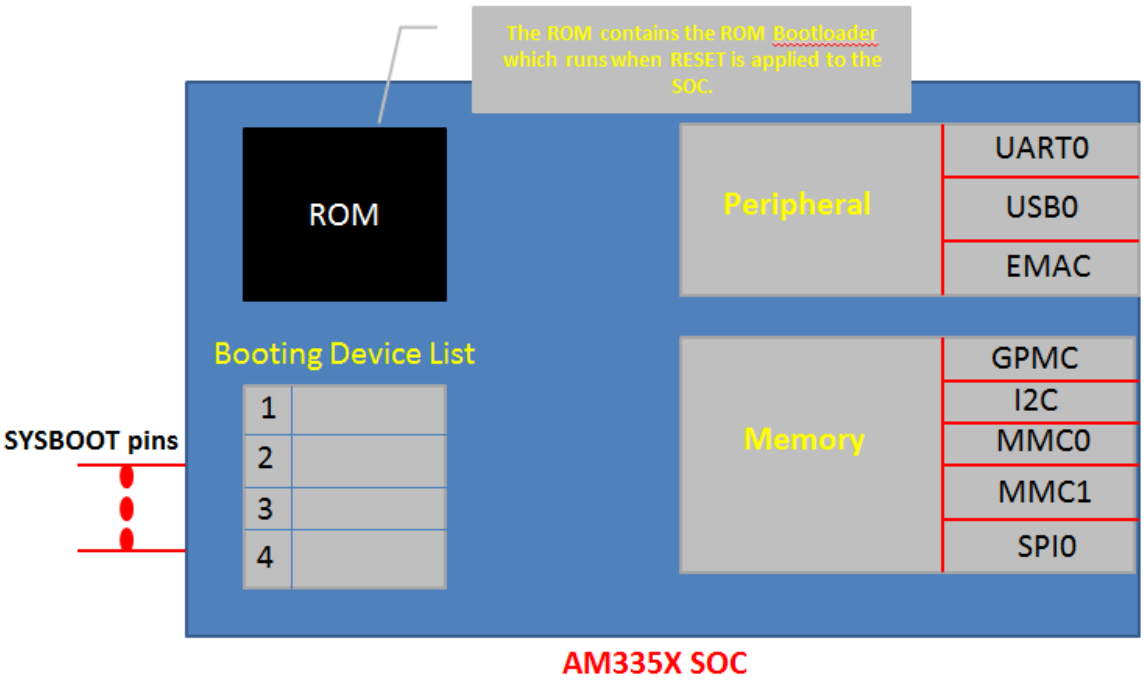


Рисунок 2.2 – будова системи на чіпі AM335x [4]

Завдання ПЗП — налаштувати системний годинник SOC, а також його основна робота — завантажити завантажувач другого етапу, що має назву MLO або SPL, які будуть розглянуті згодом.

Для завантаження завантажувача другого етапу код ПЗП читає регістр SYSBOOT [15:0] і на основі значення SYSBOOT [4:0] готує список завантажувальних пристроїв. Значення регістра SYSBOOT [15:0] визначається рівнем напруги на контактах SYSBOOT. Тобто, якщо SYSBOOT [4:0] = 00011b, то порядок завантаження буде наступним: UART0, SPI0, XIP, MMC0.

Отже, можна сказати, що контакти SYSBOOT налаштовують порядок завантажувальних пристроїв.

Деякі плати нададуть вам контроль, щоб змінити значення SYSBOOT [15:0] за допомогою DIP-перемикачів, як на рисунку 2.3.

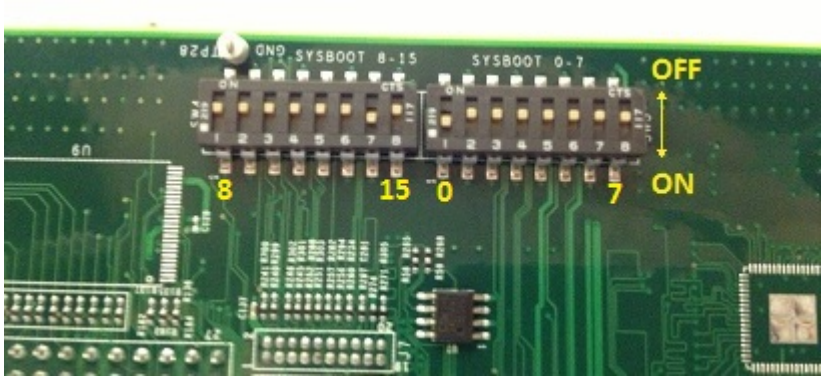


Рисунок 2.3 – DIP-перемикачі налаштування SYSBOOT

Налаштувавши DIP-перемикачі, можна визначити значення контактів SYSBOOT, Самостійно визначивши таким чином порядок завантаження.

Але BeagleBone Black не має таких перемикачів для налаштування контактів SYSBOOT.

Обрана плата наділена схемою для визначення рівня напруги на контактах SYSBOOT, що зображена на рисунку 2.4. Тут зауважте, що SYS_BOOT2 підключений до кнопки S2 (S2 є кнопкою завантаження). Подавши живлення на плату, знайдемо рівень напруги.

SYS_BOOT0 = 0B; SYS_BOOT1 = 0 B;

SYS_BOOT2 = 1 B; SYS_BOOT3 = 1 B; SYS_BOOT4 = 1 B.

Можна впевнитись у цьому вимірявши напругу за допомогою вольтметра на 45,44,43,41,40 контактах роз'єму P8 плати, матимемо SYSBOOT [4:0] = 11100 відповідно.

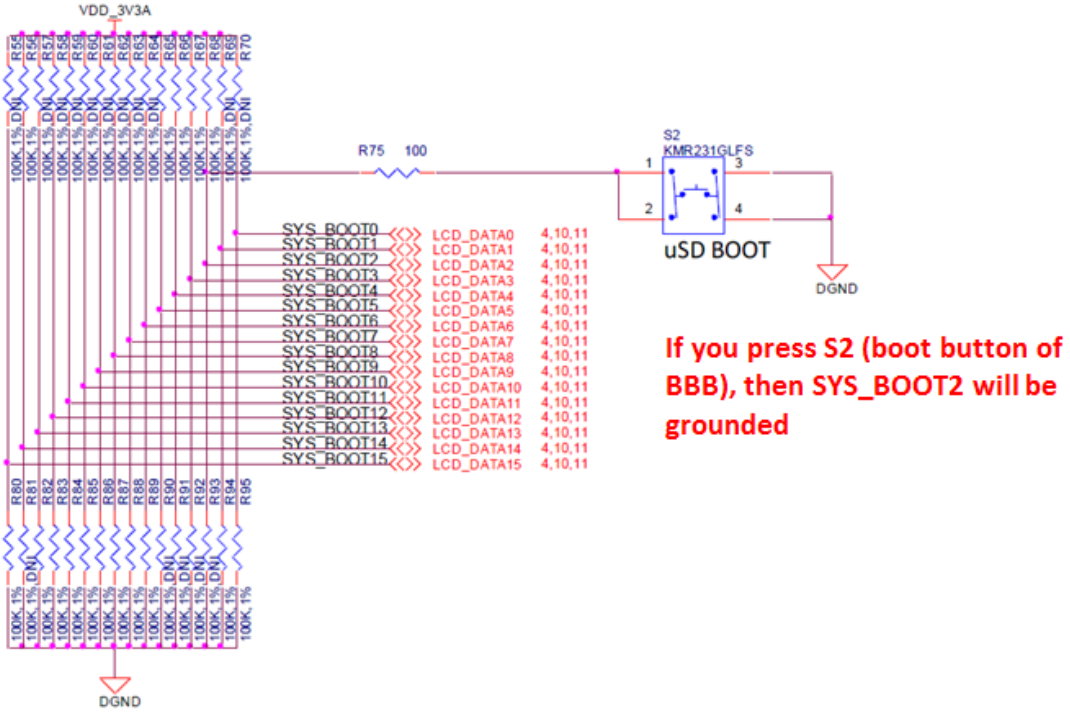


Рисунок 2.4 – дизайн-конфігурація завантаження у процесорі

Натиснувши кнопку S2, SYS_BOOT2 буде заземлений, відповідно отримаємо SYSBOOT [4:0] = 11000.

Тепер на основі S2 (кнопка завантаження BBB) ми отримали 2 конфігурації завантаження:

Віджато S2 (SYSBOOT [4:0] = 11100). Матимемо наступний порядок завантаження:

MMC1 (eMMC) → MMC0 (карта SD) → UART0 → USB0

Натиснуто S2 (SYSBOOT [4:0] = 11000), порядок завантаження буде таким

SPI0 → MMC0 (карта SD) → USB0 → UART0

Призначення пінів плати BeagleBone Black можна розглянути на рисунку 2.5.

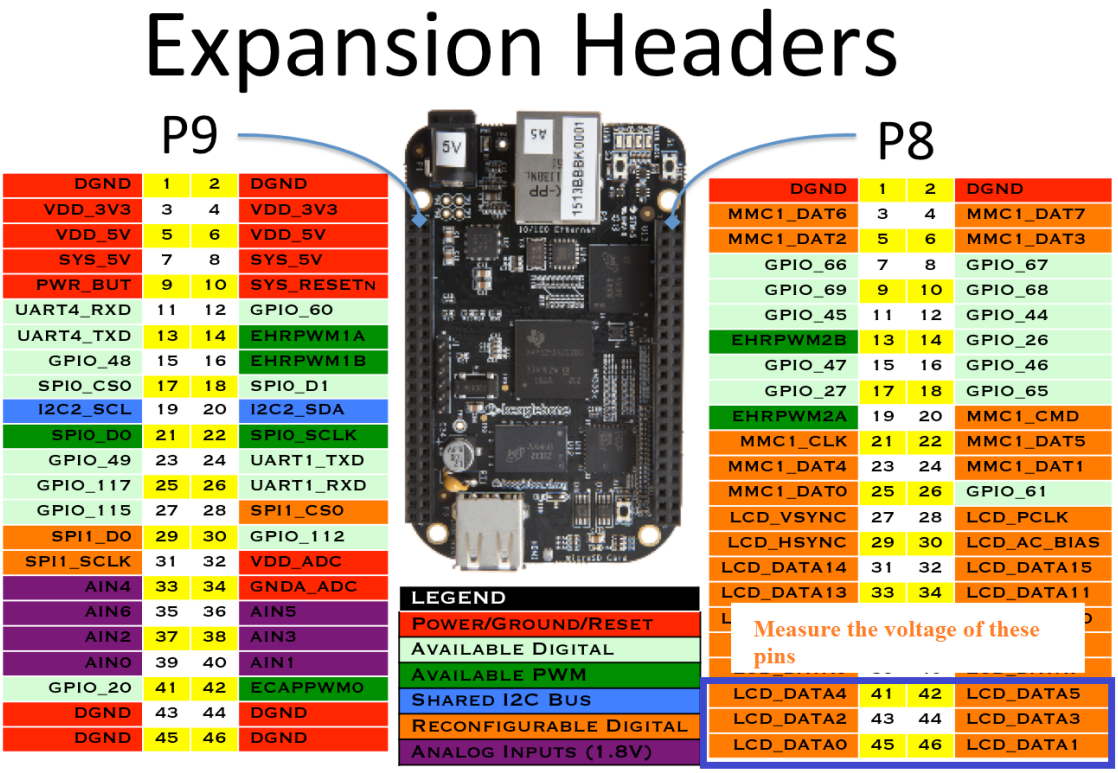


Рисунок 2.5 – призначення контактів плати BeagleBone Black

Розглянемо детальніше джерела завантаження.

Завантаження з eMMC Boot (MMC1):

eMMC підключається через інтерфейс MMC1. Це найшвидший можливий режим завантаження, eMMC знаходиться прямо на платі, тому не має необхідності купувати будь-які зовнішні компоненти або мікросхеми пам’яті. Це режим завантаження за замовчуванням. При скиданні плати, вона почне завантажуватися із завантаження зображень, збережених у eMMC. Якщо в eMMC не знайдено належного образу завантаження, процесор автоматично спробує завантажитися з наступного пристрою у списку.

Завантаження SD:

Якщо режим завантаження за замовчуванням (тобто завантажується з eMMC) не вдається, він спробує завантажитися з SD-карти, яку ви підключили до роз’єму SD-карти в інтерфейсі MMC0.

Якщо натиснути S2, а потім застосувати живлення, плата спробує спочатку завантажитися з SPI, а якщо нічого не підключено до SPI, вона спробує завантажитися з MMC0, де знайдена наша SD-карта.

Також ми можемо використовувати завантаження з SD-карти для флеш-завантаження зображень на eMMC. Отже потрібно записати нові образи на eMMC, можна завантажитися через SD-карту, потім записати нові зображення на eMMC, після чого скинути плату, щоб плата могла завантажуватися з використанням нових зображень, збережених у eMMC.

Серійне завантаження:

У цьому режимі код ПЗП SOC намагатиметься завантажити завантажувальні образи з послідовного порту.

Завантаження через USB:

Такий тип завантаження знайомий усім з встановлення операційної системи на персональний комп'ютер. Послідовність наступна: перезавантажити комп'ютер, потім натиснути кнопку BIOS, щоб перевести комп'ютер у режим BIOS, там обрати завантаження з USB.

Це дуже схоже, коли ви скидаєте плату, ви можете змусити її завантажуватися з USB-накопичувача.

Useful links:

BeagleBoard.org - black [3]
AM3358 data sheet, product information and support | TI.com [5]
Додаткова інструкція: https://git.comsys.kpi.ua [18]

Завдання 5.1.

Прошити плату з SD картки

1) Виконати форматування SD картки за інструкцією, що розміщується в робочому репозиторії проєкту, посилання на репозиторій видається викладачем [18].

2). Завдання виконати згідно інструкції, що розміщується в робочому репозиторії проєкту, посилання на репозиторій видається викладачем.

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- звіт повинен містити скриншоти на вміст директорії з файлами вихідних кодів , журналу ядра операційної системи запущеної віртуальної машини, результати виконання команд Linux в консолі запущеного ядра Linux у віддаленій консолі.
- висновки по роботі.

Контрольні запитання

- Який режим завантаження використовується за замовчуванням?
- Що таке UART?

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЯДРА ОС

4.1 Лабораторна робота №7
Ядро LINUX. Створення базових завантажуваних модулів ядра

Мета: ознайомитися з теоретичними основами та технологією розроблення завантажуваних модулів ядра для процесорів архітектури ARM v7. Ознайомитися з методами збірки, процесом компіляції та завантаження модулів ядра. Вивчити основну базову структуру модуля ядра та файлу збірки. Створити, завантажити та відлагодити власний модуль ядра.

Структура лабораторної роботи:

- Огляд архытектури ядра ARM v7
- Основні відомості про технологію розроблення модулів ядра
- Стек технологій для розробки базового модуля
- Приклад розробки базового модуля
- Розробка, компіляція та завантаження власного модуля ядра

Useful links:

GNU make [10]
Makefile cheatsheet (devhints.io) [11]
procfs - ArchWiki (archlinux.org) [17]

Теоретичні відомості та послідовність виконання завдання до лабораторної роботи

4.1.1 Архітектура ядра операційної системи Linux

Аби розібратися з функціональними особливостями ядра linux ми повинні мати уяву про зв’язок між користувацькими програмами та ресурсами комп’ютера. Ці зв’язки визначені архітектурою, що була обрана для побудови ядра конкретної системи.

Задачі, що покладені на ядро й визначають його архітектуру. Відмінності часто використовуваних архітектур побудови ядра операційної системи зображены на рис. 4.1.

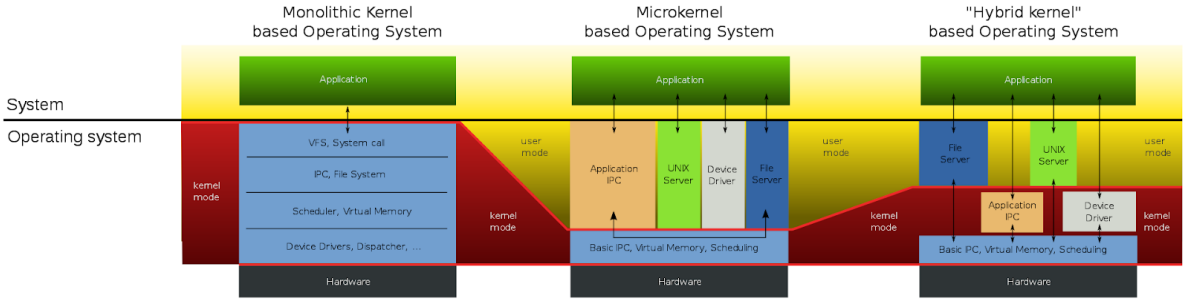


Рис. 4.1. Види архітектур: монолітне, мікро-, гібридне ядро.

Ядро linux є монолітним – воно опрацьовує усі процеси в системі, окрім користувацьких. Тобто користувач спілкується із системою лише через ядро. Задачі ядра мають найвищий пріоритет у системі. Ядро linux має контроль над

- створенням програмного інтерфейсу для доступу до файлових систем
- обробкою користувацьких запитів
- міжпроцесною взаємодією
- роботою з файловою системою
- розподіленням процесорного часу
- виділенням процесам віртуальної пам’яті
- роботою з пристроями
- роботою програм із процесором

4.1.2 Модулі ядра Linux

За своєю природою операційна система Linux є системою з монолітним ядром та можливістю завантажувати модулі. Ядро має обмежений функціонал і потребує його розширення при додаванні нових пристроїв, файлових систем. Саме це завдання лягає на плечі завантажуваних модулів.

Завантажуваний модуль ядра є об’єктним файлом, що може розширювати можливості ядра операційної системи без потреби в перекомпіляції ядра або

перезавантаженні системи. А коли функціональність, що була надана завантажуваним модулем, вже не буде потрібна, модуль може бути вивантаженим для звільнення ресурсів.

Без можливості розширення функціональності ядра за допомогою завантажуваних модулів, саме ядро Linux було би дуже великим, бо потребувало би вмісту всіх необхідних/існуючих драйверів.

Аби модуль був сумісним з ядром, він має бути скомпільованим під версію ядра, що співпадає з запущеним (якщо модуль сумісний за допомогою dkms (Dynamic Kernel Module Support), є спосіб його запуску незалежно від версії ядра).

Під час інсталяції системи відбувається визначення обладнання комп'ютера. Залежно від виявленого обладнання та заданих користувачем параметрів системи програма інсталяції вирішує, які модулі необхідно завантажувати при завантаженні.

Якщо після інсталяції додається нове обладнання для якого потрібен новий модуль ядра, то необхідно налаштувати систему для завантаження цього модуля. Для цього при завантаженні системи з новим обладнанням запускається утиліта Kudzu, яка визначає чи підтримується це обладнання і налаштовує модуль для нього. Також можна вказати цей модуль вручну, відредагувавши файл конфігурації модуля */etc/modprobe.conf*.

Керування модулями за допомогою команд відбувається завдяки інсталюванню пакета module-init-tools. За допомогою його основних утиліт відбувається управління модулями:

- *insmod* – завантаження модуля.
- *rmmod* – видалення модуля.
- *lsmod* – вивести список завантажених модулів.
- *modinfo* – вивести вміст секції *.modinfo* об’єкного файлу модуля.
- *modprobe* – розумне завантаження чи видалення модуля чи набору модулів. Якщо потребується завантажити А перед завантаженням Б, то при запиті на завантаження Б, А буде завантажено автоматично.

Щоб завантажити модуль ядра, необхідно скористатися утилітою modprobe, вказавши в параметрах назву модуля ядра. За замовчуванням modprobe намагається завантажити модуль із підкаталогів */lib/modules/<kernel-version>/kernel/drivers/*. Для різних типів модулів призначені різні підкаталоги, наприклад, підкаталог *net/* містить драйвери мережевих плат. Деякі модулі ядра мають залежності, це означає, що перед ними мають бути завантажені інші модулі. Команда `/sbin/modprobe` перевіряє залежності та перш ніж завантажити зазначений модуль, завантажує необхідні йому модулі.

Після розробки модуля і потрібно створити Makefile, що буде відповідати за збірку модуля в файл необхідного формату (*.ko*). Потрібно зазначити, що процес збірки модулів ядра відрізняється від звичайного процесу збірки за допомогою make. Головна відмінність полягає в тому, що замість утиліти make процесом займається система збірки ядра, таким чином можна полегшити розробку модуля. Створити Makefile досить просто, якщо у модулі присутній лише один файл, то достатньо лише даного рядка:

```
obj-m := name.o
```

Де name це назва об'єктного файлу, з якого має бути зібраний модуль. Якщо ж модуль має бути зібраний з більш ніж одного файлу, наприклад *file1.c* та *file2.c*, то скрипт мав би наступний вигляд:

```
obj-m := module.o
module-objs := file1.o file2.o
```

Як вже було сказано, процесом збірки займається система збірки ядра, тому щоб Makefile працював, він має бути використаний в контексті системи збірки ядра. Така команда виглядатиме наступним чином:

```
make -C ~/kernel_dir M='pwd'
```

Ця команда спочатку змінює робочий каталог на той, в якому знаходиться вихідний код ядра, знаходить там Makefile верхнього рівня, після чого знову змінює робочу директорію на ту, що знаходиться в `M=''` та використовує Makefile ядра, щоб зібрати код, що знаходиться в даному каталозі.

Після того як модуль був зібраний, можна нарешті його завантажити до ядра ОС.

4.1.3. Розробка базового модуля Linux

Модулі ядра написані на мові програмування С, але вони значно відрізняються від звичайних користувацьких програм. Ключовими відмінностями є те, що завантажувані модулі:

- не мають функції main();
- не виконуються послідовно, подібно до програмування на основі подій;
- будь-які ресурси, виділені модулю, повинні бути вивільнені вручну при вивантаженні модуля;
- повинні мати постійну та прогнозовану поведінку при виникненні переривань;
- мають вищий рівень привілеїв виконання;
- не мають підтримки значень з плаваючою комою.

Під час написання та тестування LKM дуже легко покласти ОС. Цілком можливо, що при цьому може пошкодитись ваша файлова система та завантаження системи стане неможливим. Тому для розробки необхідно використовувати віртуальну машину, яку можна відновити без втрати важливих даних або використовувати вбудовану систему на платі, такій як BeagleBone Black.

Щоб збілдити завантажуваний модуль, необхідно мати встановленими заголовки ядра Linux. Заголовки ядра Linux — це файли заголовків С, які визначають інтерфейси між різними модулями ядра, ядром та простором

користувача. Наявні файли заголовків мають бути тієї ж версії, що і ядро, для якого ви хочете створити модуль.

В ОС Debian, Ubuntu або Mint Linux ви можете встановити заголовки ядра так:

```
$ sudo apt install linux-headers-$(uname -r)
```

4.1.4. Приклад розробки найпростішого модуля ядра Linux

hello_world.c

```
/**
 * Базовий завантажуваний модуль "Hello World!", що відображає повідомлення в
 * /var/log/kern.log файл при завантаженні та вивантаженні. Модуль може приймати
 * аргумент при завантаженні -- ім'я, що з'явиться в log файлі ядра.
 *
 * Якщо не вказати ім'я при завантаженні модуля, буде використано аргумент за замовчуванням
 */
документація linux/ файлів заголовку
#include <linux/init.h>      // містить макроси функцій e.g. __init
#include <linux/module.h>    // динамічне завантаження модулів у ядро
#include <linux/kernel.h>    // типи, макроси, функції ядра

// Тип ліцензії впливає на доступ модуля до ресурсів ядра. Тип ліцензії як у прикладі дозволяє
//  доступитись до всіх ресурсів.
MODULE_LICENSE("GPL");

// Інформація про модуль
MODULE_AUTHOR("College Student");
MODULE_DESCRIPTION("Basic LKM");
MODULE_VERSION("1.0");

// Ім'я змінної оголошується як статичне, щоб обмежити її область дії в межах модуля
```

```
	// Вам слід уникати використання глобальних змінних у модулях ядра, оскільки вони використовуються для всього ядра.
	static char *name = "world"; // "world" - значення аргументу за замовчуванням

	/**
	 * Макрос module_param(ім'я, тип, дозволи) має три параметри:
	 * 1) ім'я, ім'я параметра, яке відображається користувачеві, і ім'я змінної в модулі
	 * 2) тип, тип параметра - один із byte, int, uint, long, ulong, short, ushort, bool,
	 * invool, або charp (вказівник char)
	 * 3) дозволи, це права доступу до параметра при використанні sysfs
	 */

	module_param(name, charp, S_IRUGO); // S_IRUGO дозвіл на читання
	MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");

	/** Функція ініціалізації LKM
	 * Ключове слово static обмежує видимість функції в межах цього файлу C. __init
	 * макрос означає, що для вбудованого драйвера (не LKM) функція використовується лише під час ініціалізації
	 * і що його можна відкинути, а його пам'ять звільнити після цього моменту.
	 * У разі успіху return повертає 0
	 */
	static int __init hello_init(void)
	{
		// printk() схожий на printf(), і ви можете викликати його з будь-якого місця в коді LKM.
		// Основна відмінність полягає в тому, що ви повинні вказати рівень журналу під час виклику функції.
		// Рівні журналу визначені в linux/kern_levels.h.
		printk(KERN_INFO "Hello: Hello %s from the HelloWorld LKM!\n", name);
		return 0;
	}

	/** Функція очищення LKM
	 * Подібно до функції ініціалізації, вона статична. Макрос __exit повідомляє, що якщо цей код
```

```
 * використовується для вбудованого драйвера (не LKM), що ця функція не потрібна.
 */
static void __exit hello_exit(void)
{
    printk(KERN_INFO "Hello: Goodbye %s from the HelloWorld LKM!\n", name);
}

/** Модуль повинен використовувати макроси module_init() module_exit() з linux/init.h, які
 * визначають функцію ініціалізації та функцію очищення.
 *
 * Функції в модулі можуть мати будь-які назви; однак імена повинні
 * передаватись до спеціальних макросів module_init() і module_exit() в самому кінці модуля.
 */

// Коли цей модуль завантажується, виконується функція hello_init().
module_init(hello_init);

// Коли цей модуль вивантажується, виконується функція hello_exit().
module_exit(hello_exit);
```

Makefile

```
# Для створення завантажуваного модуля ядра (LKM) потрібен Makefile. Насправді це особливий kbuild
# Makefile.
#
# Попередження: Make-файли чутливі до табуляції, вимагають її перед правилами (перед викликами «make» нижче)
#

# визначення мети, визначає модуль, який буде створено (hello_world.o)

# obj-m визначає ціль завантажуваного модуля
obj-m+=hello_world.o

# Решта цього Makefile подібна до звичайного Makefile
```

```
	# "$(shell uname -r)" є корисним викликом для повернення поточної версії збірки ядра
	# Параметр -C перемикає каталог на каталог ядра перед виконанням будь-яких завдань створення
	# Призначення змінної "M=$(PWD)" повідомляє команді make, де існують фактичні файли проекту

	# Ціль "modules" є цільовою за замовчуванням для зовнішніх модулів ядра.
	# Альтернативною метою є "modules_install", яка встановить модуль (потрібен sudo)

	all:
	    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
	clean:
	    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

При виклику

```
$ make
```

проект буде зібраний, та буде отримано файл модуля *hello_world.ko,* який ми завантажимо за допомогою

```
$ insmod hello_world.ko name=<значення аргументу>
```

Завдання 7.1

Розробимо завантажуваний модуль для парсингу DeviceTree.

1). Для цього в папці проєкта створимо файл *overlay_test.dts*

```
$ touch overlay_test.dts
```

та наповнимо його наступним змістом:

```
/dts-v1/;   /* syntax version 1 */
/plugin/;   /* required plugin statement */
/ {         /* root node identificator */
```

```
      compatible = "ti, am33xx"     /* "<manufacturer>, <model>" (ti - Texas
Instruments)*/
      fragment@0 {                /* start node of customization the device tree*/
            target-path = "/";          /* target for overlay - root path */
            __overlay__ {               /* overlay node */
                  bbb_board {           /* self-created name of the node */
                        compatible = "ti, beaglebone-black";
                        /* some properties ... */
                        label = "bbb_test";
                        status = "okay";
                        value = <2022>;
                  };
            };
      };
  };
```

2). Збережемо файл та скомпілюємо його за допомогою компілятора *dtc*, який попередньо встановимо наступним чином:

```
$ sudo apt install device-tree-compiler
$ dtc -@ -I dts -O dtb -o overlay_test.dtbo overlay_test.dts
```

3). Перевіримо створення об’єктного файлу *overlay_test.dtbo*:

```
$ ls
```

```
ivan_main@ivanMain-VirtualBox:~/development/diploma/basic_klm$ dtc -@ -I dts -O
dtb -o overlay_test.dtbo overlay_test.dts
ivan_main@ivanMain-VirtualBox:~/development/diploma/basic_klm$ ls
overlay_test.dtbo  overlay_test.dts
```

4). Створимо *Makefile* з наступним вмістом:

```
  obj-m += dt_parse.o

  all:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

  clean:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

5). Розробимо модуль dt_parse.c:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/mod_devicetable.h>
#include <linux/property.h>
#include <linux/platform_device.h>
#include <linux/of_device.h>

/* Module info */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ivan Siryi");
MODULE_DESCRIPTION("LKM example for parsing DeviceTree source file");

/* Declate functions */
static int dt_probe(struct platform_device *p_pdev); /* function called while loading the driver */
static int dt_remove(struct platform_device *p_pdev); /* function called while unloading the driver */

static struct of_device_id bbb_driver_ids[] = {  /* list of compatible devices*/
    {
          .compatible = "ti, beaglebone-black",
    }, { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, bbb_driver_ids);   /* assign compatible device list to a module */

static struct platform_driver bbb_driver = {
    .probe = dt_probe,
    .remove = dt_remove,
    .driver = {
          .name = "bbb_driver",
          .of_match_table = bbb_driver_ids,
    },
};

static int dt_probe(struct platform_device *p_pdev) {
    struct device *dev = &p_pdev->dev;
    const char *label;
```

```
    int value, ret;

    printk("Starting dt_probe function\n");

    /* Checking presence of properties */
    if(!device_property_present(dev, "label")) {
          printk("Error! Property 'label' not found!\n");
          return -1;
    }
    if(!device_property_present(dev, "value")) {
          printk("Error! Property 'value' not found!\n");
          return -1;
    }

    /* Read device properties */
    ret = device_property_read_string(dev, "label", &label);
    if(ret) {
          printk("Error! Could not read 'label'\n");
          return -1;
    }
    printk("label property is: %s\n", label);

    ret = device_property_read_u32(dev, "value", &value);
    if(ret) {
          printk("Error! Could not read 'value'\n");
          return -1;
    }
    printk("value property is: %d\n", value);

    return 0;
}

static int dt_remove(struct platform_device *p_pdev) {
    printk("Starting dt_remove function\n");
    return 0;
}


static int __init my_init(void) {
    printk("Loading the bbb_driver...\n");
```

```
    if(platform_driver_register(&bbb_driver)) {
          printk("Error! Could not load bbb_driver\n");
          return -1;
    }
    return 0;
}

static void __exit my_exit(void) {
    printk("Unload bbb_driver");
    platform_driver_unregister(&bbb_driver);
}

module_init(my_init);
module_exit(my_exit);
```

6). Зберемо наш розроблений модуль:

```
$ make
```

7). Застосуємо дерево пристроїв, перевіримо його коректність та впевнимося у його застосуванні до нашої системи:

```
$ sudo dtoverlay overlay_test.dtbo
$ sudo ls /proc/device-tree/
$ sudo ls /proc/device-tree/beaglebone-black
$ sudo cat /proc/device-tree/beaglebone-black/label
/proc/device-tree/beaglebone-black/value
```

8). Завантажимо розроблений модуль, запустивши попередньо *qemu*:

```
$ sudo insmod dt_parse.ko
```

9). Подивимось на результати виконання:

```
$ sudo dmesg | tail
```

10). Видалимо модуль та перевіримо коректність його завершення:

```
$ sudo rmmod dt_parse.ko
```

```
$ sudo dmesg | tail
```

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- звіт повинен містити скриншоти виконання модулів ядра;
- висновки по роботі.

Контрольні запитання

- Які переваги та недоліки монолітного ядра?
- Якщо ядро Linux монолітне, чому можливе підключення модулів ядра?
- Які переваги це надає підключення модулів ядра?
- Які можливості відкриває використання тулчейнів?

4.2. Лабораторна робота №8

Ядро LINUX. Створення базових завантажуваних модулів ядра

Мета: ознайомитися з теоретичним підґрунтям технології завантажуваних модулів ядра на основі існуючих архітектур ядра. Ознайомитися з методами збірки, процесом компіляції та завантаження модулів ядра. Вивчити основну базову структуру модуля ядра та файлу збірки. Створити, завантажити та відлагодити власний модуль ядра.

Структура лабораторної роботи:

- Відомості щодо використання файлової системи proc
- Робота з файловою системою proc
- Розроблення, компіляція та завантаження власних модулів ядра

Useful links:

GNU make [10]

Makefile cheatsheet (devhints.io) [11]

procfs - ArchWiki (archlinux.org) [17]

Теоретичні відомості та послідовність виконання завдання до лабораторної роботи

4.2.1. Файлова система *proc*

Для сильнішого поглиблення у розробку модулів пригадаємо, що система linux побудована за принципом “усе є файл”.

Proc file system або ж procfs є способом зберігання інформації про систему. Вона є віртуальною файловою системою, що надає інформацію через, окремо існуючий для кожного процесу, файл. Також procfs містить інформацію про ядро та його параметри (sysctl), та різноманітні метрики системи, такі як час роботи системи, інформація про центральний процесор, фізичну та віртуальну пам’ять, завантажені за допомогою модулів ядра файлові системи.

4.2.2. Використання файлової системи *proc*

Кожен процес, що виконується представлений у файловій системі proc у вигляді файлу */proc/<pid>*, де pid - унікальний ідентифікатор процесу. Взємодія з файлами системи proc відбувається так само, як зі звичайними файлами. Їх можна відкривати на читання

```
$ cat /proc/cmdline
```

або ж записувати у них:

```
$ echo 1 > /proc/sys/kernel/sysrq
```

Зазирнемо в один з файлів процесів нашої системи, попередньо проглянувши увесь їх список:

```
$ ls -l /proc/
```

```
total 0
dr-xr-xr-x  9 root    root                0 Sep  8 18:17 1
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 10
dr-xr-xr-x  9 daemonx daemonx             0 Sep  9 03:02 1057
dr-xr-xr-x  9 daemonx daemonx             0 Sep  8 18:18 1077
dr-xr-xr-x  9 daemonx daemonx             0 Sep  9 03:02 1087
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 11
dr-xr-xr-x  9 daemonx daemonx             0 Sep  9 03:02 1103
dr-xr-xr-x  9 daemonx daemonx             0 Sep  9 03:02 1107
dr-xr-xr-x  9 daemonx daemonx             0 Sep  9 03:02 1159
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 12
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 124
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 125
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 127
dr-xr-xr-x  9 root    root                0 Sep  9 03:02 128
...
```

```
$ ls -l /proc/1057
```

```
total 0
dr-xr-xr-x 2 daemonx daemonx 0 Sep  9 03:12 attr
-rw-r--r-- 1 daemonx daemonx 0 Sep  9 03:12 autogroup
-r-------- 1 daemonx daemonx 0 Sep  9 03:12 auxv
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 cgroup
--w------- 1 daemonx daemonx 0 Sep  9 03:12 clear_refs
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 cmdline
-rw-r--r-- 1 daemonx daemonx 0 Sep  9 03:12 comm
-rw-r--r-- 1 daemonx daemonx 0 Sep  9 03:12 coredump_filter
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 cpuset
lrwxrwxrwx 1 daemonx daemonx 0 Sep  9 03:12 cwd -> /home/daemonx
-r-------- 1 daemonx daemonx 0 Sep  9 03:12 environ
lrwxrwxrwx 1 daemonx daemonx 0 Sep  9 03:12 exe -> /usr/lib/gvfsd-metadata
dr-x------ 2 daemonx daemonx 0 Sep  9 03:12 fd
dr-x------ 2 daemonx daemonx 0 Sep  9 03:12 fdinfo
-rw-r--r-- 1 daemonx daemonx 0 Sep  9 03:12 gid_map
-r-------- 1 daemonx daemonx 0 Sep  9 03:12 io
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 latency
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 limits
-rw-r--r-- 1 daemonx daemonx 0 Sep  9 03:12 loginuid
dr-x------ 2 daemonx daemonx 0 Sep  9 03:12 map_files
-r--r--r-- 1 daemonx daemonx 0 Sep  9 03:12 maps
-rw------- 1 daemonx daemonx 0 Sep  9 03:12 mem
...
```

Детальніше про деякі з полів:

- *cmdline* – аргументи, які приймає прогграма під час свого запуску;
- *cwd* – директорія, в якій відбувається робота процесу;
- *environ* – змінні середовища всередині процесу;
- *fd/* – директорія, яка зберігає список відкритих дескрипторів системи;
- *exe* – символічне посилання на виконуваний файл процесу;
- *maps* – інформація про розміщення процесу в оперативній пам’ті;
- *mem* – віртуальна пам’ять процесу.

Завдання 8.1.

Розширимо можливості побудованого у попередній лабораторній роботі модуля ядра, для чого виконайте наступні завдання.

1). Створимо нову директорію проєкта, скопіюємо у нього попередньо створені файли, та змінимо ім’я *.c* файлу. Не забудемо перед цим очистити директорію проєкта попередньої лабораторної роботи за допомогою виклику

```
$ make clean
```

```
$ cd ..
$ cp -r dt_parse/ dt_gpio
$ cd dt_gpio
$ mv dt_parse.c dt_gpio.c
```

2). Почемо з модифікації *Makefile.* Аби не компілювати окремо наш файл дерева пристроїв *.dts* додамо до *Makefile* нове правило.

```
obj-m += dt_gpio.o

all: module dt
	echo My kernel loadable module

module:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

dt: overlay_test.dts
	dtc -@ -I dts -O dtb -o overlay_test.dtbo overlay_test.dts

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
	rm -rf overlay_test.dtbo
```

3). Додамо до нашого дерева пристроїв рядок з абстрактним апаратним забезпеченням, що буде підключений до роз'єму входу/виходу загального призначення (*gpio*).

```
/dts-v1/;   /* syntax version 1 */
/plugin/;   /* required plugin statement */
/ {         /* root node identificator */
	compatible = "ti, am33xx"   /* "<manufacturer>, <model>" */
	fragment@0 {            /* start node of customization the device tree*/
		target-path = "/";         /* target for overlay */
		__overlay__ {              /* overlay node */
			bbb_board {  /* self-created name of the device */
```

```
                              compatible = "ti, beaglebone-black";        /*
Texas Instruments, BeagleBone Black */
                              /* some properties ... */
                              label = "bbb_test";
                              status = "okay";
                              value = <2022>;
                              simple-gpio = <&gpio 23 0>;                 /*
<gpio-controller, gpio index, active low/high> */
                        };
                  };
            };
      };
```

4). Перепишемо наш модуль, додавши у нього парсинг нашої нової властивості та можливість керування роботою пристрою, через запис відповідного сигналу у файл *procfs*.

```
      #include <linux/module.h>
      #include <linux/init.h>
      #include <linux/mod_devicetable.h>
      #include <linux/property.h>
      #include <linux/platform_device.h>
      #include <linux/of_device.h>
      #include <linux/gpio/consumer.h>
      #include <linux/proc_fs.h>

      /* Module info */
      MODULE_LICENSE("GPL");
      MODULE_AUTHOR("Student Student");
      MODULE_DESCRIPTION("LKM example for parsing DeviceTree source file with
simple gpio");

      /* Declate functions */
      static int dt_probe(struct platform_device *p_pdev);
      static int dt_remove(struct platform_device *p_pdev);

      static struct of_device_id bbb_driver_ids[] = {
            {
```

```
                .compatible = "ti, beaglebone-black",
        },
        {/* sentinel */}};
MODULE_DEVICE_TABLE(of, bbb_driver_ids);

static struct platform_driver bbb_driver = {
        .probe = dt_probe,
        .remove = dt_remove,
        .driver = {
                .name = "bbb_driver",
                .of_match_table = bbb_driver_ids,
        },
};

static struct gpio_desc *simple_gpio = NULL; /* GPIO variable */

static struct proc_dir_entry *proc_file;

static ssize_t write_to_buff(struct file *File, const char *buffer, size_t
count, loff_t *off) {
        switch (buffer[0]) {
                case '0':
                case '1':
                        gpiod_set_value(simple_gpio, buffer[0] - '0');
                default:
                        break;
        }
        return count;
}

static struct proc_ops file_operations = {
        .proc_write = write_to_buff,
};


static int dt_probe(struct platform_device *p_pdev) {
        struct device *dev = &p_pdev->dev;
        const char *label;
        int value, ret;
```

```
        printk("Starting dt_probe function\n");

        /* Checking presence of properties */
        if (!device_property_present(dev, "label")) {
              printk("Error! Property 'label' not found!\n");
              return -1;
        }
        if (!device_property_present(dev, "value")) {
              printk("Error! Property 'value' not found!\n");
              return -1;
        }
        if(!device_property_present(dev, "simple-gpio")) {
              printk("Error! Property 'simple-gpio' not found!\n");
              return -1;
        }

        /* Read device properties */
        ret = device_property_read_string(dev, "label", &label);
        if (ret){
              printk("Error! Could not read 'label'\n");
              return -1;
        }
        printk("label property is: %s\n", label);

        ret = device_property_read_u32(dev, "value", &value);
        if (ret) {
              printk("Error! Could not read 'value'\n");
              return -1;
        }
        printk("value property is: %d\n", value);

        simple_gpio = gpiod_get(dev, "simple-gpio", GPIOD_OUT_LOW); /* Init
GPIO */
        if(IS_ERR(simple_gpio)) {
              printk("Error! Could not setup the GPIO\n");
              return -1 * IS_ERR(simple_gpio);
        }

        proc_file = proc_create("my-simple-gpio", 0666, NULL,
&file_operations); /* Creating procfs file */
```

```
	if(proc_file == NULL) {
		printk("Error creating /proc/my-simple-gpio\n");
		gpiod_put(simple_gpio);
		return -ENOMEM;
	}

	return 0;
}

static int dt_remove(struct platform_device *p_pdev) {
	printk("Starting dt_remove function\n");
	gpiod_put(simple_gpio);
	proc_remove(proc_file);
	return 0;
}

static int __init my_init(void) {
	printk("Loading the bbb_driver...\n");
	if (platform_driver_register(&bbb_driver)) {
		printk("Error! Could not load bbb_driver\n");
		return -1;
	}
	return 0;
}

static void __exit my_exit(void) {
	printk("Unload bbb_driver");
	platform_driver_unregister(&bbb_driver);
}

module_init(my_init);
module_exit(my_exit);
```

5). Зберемо наш розроблений модуль:

```
$ make
```

6). Застосуємо дерево пристроїв, перевіримо його коректність та впевнимося у його застосуванні до нашої системи:

```
$ sudo dtoverlay overlay_test.dtbo
$ sudo ls /proc/device-tree/
$ sudo ls /proc/device-tree/beaglebone-black/my-simple-gpio
```

7). Завантажимо розроблений модуль, запустивши попередньо qemu:

```
$ sudo insmod dt_gpio.ko
```

8). Подивимось на результати виконання:

```
$ sudo dmesg | tail
```

9). Подамо сигнал включення до нашого апаратного пристрою - запишемо 1 в його proc файл.

```
$ echo 1 > /proc/
```

10). Видалимо модуль та перевіримо коректність його завершення:

```
$ sudo rmmod dt_gpio.ko
$ sudo dmesg | tail
```

Структура звіту:

Звіт повинен містити

- посилання на репозиторій, де розміщений результат виконання завдання;
- звіт повинен містити скриншоти виконання модулів ядра;
- висновки по роботі.

Контрольні запитання

- Що таке дерева пристроїв (Device Tree)?

- Які переваги надає використання Device Tree?
- Для чого використовуэться утиліта make?
- Які альтернативи утиліти make існують?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Avila R. How to Select the Right CPU or SoC for your Embedded Project [Електронний ресурс] / Risto Avila. – 2021. – Режим доступу до ресурсу: https://www.qt.io/embedded-development-talk/how-to-select-the-right-cpu-or-soc-for-your-embedded-project.

2. Embedded product planning and requirements guide [Електронний ресурс] // The QT company. – 2021. – Режим доступу до ресурсу: https://resources.qt.io/home/the-embedded-product-planning-and-requirements-guide.

3. BeagleBone Black [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://beagleboard.org/black.

4. Coley G. BeagleBone Black System Reference Manual [Електронний ресурс] / Gerald Coley. – 2013. – Режим доступу до ресурсу: https://cdn-shop.adafruit.com/datasheets/BBB_SRM.pdf.

5. AM335x and AMIC110 Sitara™ Processors Technical Reference Manual (Rev. Q) [Електронний ресурс] // Texas Instruments. – 2022. – Режим доступу до ресурсу: https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf?ts=1660308115041&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FAM3357.

6. Arch Linux documentation. ArchWiki [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://wiki.archlinux.org.

7. The Linux command line for beginners | Ubuntu [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://ubuntu.com/tutorials/command-line-for-beginners#1-overview .

8. Chacon S., Straub B. Pro Git book. 2nd Edition [Електронний ресурс] // Apress. – 2014. – 548 с. – Режим доступу до ресурсу: https://git-scm.com/book/uk/v2.

9. Quick reference guides: GitHub Cheat Sheet | Visual Git Cheat Sheet [Електронний ресурс] // Режим доступу до ресурсу: https://git-scm.com/docs/.

10. Stallman R.M., McGrath R., Paul D.S. GNU Make Manual. A Program for Directing Recompilation GNU make Version 4.3 [Електронний ресурс] //

2020. – Режим доступу до ресурсу: https://www.gnu.org/software/make/manual/make.html.

11. 3.2. Makefile cheatsheet. Free Software Foundation [Електронний ресурс] // 2020. – Режим доступу до ресурсу: https://devhints.io/makefile.

12. 3.4 The U-Boot Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://u-boot.readthedocs.io/en/latest/index.html.

13. 3.5 U-Boot Reference Manual [Електронний ресурс] – Режим доступу до ресурсу: https://hub.digi.com/dp/path=/support/asset/u-boot-reference-manual/.

14. U-Boot / U-Boot GitLab [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://source.denx.de/u-boot/u-boot.

15. Makefile master U-Boot / U-Boot GitLab [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://source.denx.de/u-boot/u-boot/-/blob/master/Makefile.

16. About the Device Tree [Електронний ресурс] – Режим доступу до ресурсу: http://www.ofitselfso.com/BeagleNotes/AboutTheDeviceTree.pdf.

17. Arch Linux documentation: Procfs [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://wiki.archlinux.org/title/Procfs.

18. Embedded systems and the Internet of Things. Git for comsys (kpi.ua) [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://git.comsys.kpi.ua.

19. Protsenko S. BeagleBone Black: Platform Bring-Up with Upstream Components [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.slideshare.net/GlobalLogicUkraine/beaglebone-black-platform-bringup-with-upstream-components

20. Сірий І.М. Навчальний програмно-апаратний комплекс для вбудованої системи на базі процесора ARM32 : дипломний проєкт бакалавра : керівник В.А. Таранюк: 123 Комп'ютерна інженерія / І.М. Сірий [Електронний ресурс] // Київ, 2022. – 79 с. – Режим доступу до ресурсу: