КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Курсова робота

за спеціальністю 121 Інженерія програмного забезпечення
на тему:

СТВОРЕННЯ МОДУЛЕЙ І ДРАЙВЕРІВ ПІД ЯДРО LINUX

Виконав студент 3-го курсу
Опанюк Микита Ігорович _____
(підпис)

Науковий керівник:
доцент
Галкін Олександр Володимирович _____
(підпис)

Засвідчую, що в цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ –2019

РЕФЕРАТ

Обсяг роботи 57 сторінок, 22 ілюстрацій, 3 таблиці, 27 джерело посилань.

РОЗРОБКА МОДУЛІВ ТА ДРАЙВЕРІВ ПІД ОПЕРАЦІЙНУ СИСТЕМУ LINUX, ДОСЛІДЖЕННЯ ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX, ДОСЛІДЖЕННЯ РІЗНИХ ПРИКЛАДНИХ ПРОГРАМНИХ ІНТЕРФЕЙСІВ ДЛЯ РОБОТИ З ПЕРИФІРІЙНИМИ ПРИСТРОЯМИ, РІЗНОВИДИ ПРИСТРОЇВ

Об’єктом роботи є процес дослідження поведінки різних типів пристроїв та налаштування їх працездатності за допомогою драйверів, як програмного забезпечення, завдяки якому операційна система (Linux) отримує доступ до апаратної складової комп’ютера для виконання операцій, налаштувань, обробки інформації тощо. Предметом роботи є набір програмного забезпечення операцій взаємодії цих функціональних блоків між собою.

Метою роботи є створення програмного забезпечення для налаштування та реалізації взаємодії функціональних блоків комп’ютера між собою та операційною систему для можливості отримання інфромації з пристроїв користувачем.

Методи розроблення: комп’ютерне моделювання, методи роботи з різними типами пристроїв, інформаційних шин комп’ютера, розробка програмного продукту на основі сучасної операційної системи.

Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки Eclipse IDE 4.11, мова програмування C, GNU toolchain (набір необхідних пакетів програм для компіляції та генерації виконуваного коду з текстів програм на мові C).

Результати роботи: виконано загальний огляд електронних засобів, налагодження роботи та взаємодії електронних засобів з вбудовуваним одноплатним комп'ютером, на основі процесора архітектури ARM, BeagleBone Black, проаналізовано переваги та недоліки використання рзіних стандартів та інтерфейсів для роботи з окремими модулями комп’ютера, розроблено програмні продукти (драйвера та модулі), які дозволяють наочно демонструвати процеси роботи та взаємодії різних апаратних засобів між собою, а також зчитувати інформацію з пристроїв на рівні операційної системи - як в просторі ядра, так і в просторі користувача.

За методами розробки та інструментальними засобами робота виконувалася сумісно з прикладами драйверів, реалізованих в операційній системі Linux.

Програмні продукти можуть застосовуватися в навчальному процесі університетського курсу системного програмування під час вивчення модулів та драйверів операційної системи для роботи з апаратної складовою комп’ютера.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

VFS – Virtual file-system, віртуальна файлова система.
BBB – BeagleBone Black, вбудовуваний одноплатний компьютер;
API – Application programming interface, прикладний програмний інтерфейс;
GPIO – General-purpose input/output, інтерфейс введення/виведення загального призначення;
I2C – Inter-Integated Circuit, послідовна шина даних для зв'язку інтегральних схем;
PCI – Peripheral component interconnect, шина вводу/виводу для підключення периферійних пристроїв до материнської плати комп’ютера;
USB – Universal serial bus, універсальна послідовна шина, призначена для з'єднання периферійних пристроїв копм’ютера.
DTS – Device tree source, представляє спеціальну структуру даних, що описує апаратні компоненти конкретного комп'ютера, так що ядро операційної системи може використовувати і керувати тими компонентами, включаючи процесор або процесор, пам'ять, шини і периферійні пристрої.
MMU – Memory management unit, модуль керування пам'яттю.
PM – Power manager, менеджер живлення.
IRQ – Interrupt request, "запит на прерывання". Так прийнято називати спеціальний сигнал, який повідомляє процесору про необхідність припинення виконання поточної програми, зберігаючи її стан, і перехід до заздалегідь заданої адреси пам'яті для її обробки.
ARM – Advanced RISC Machine, поліпшена RISC машина. 32-бітна RISC архітектура процесорів.
RISC – Reduced Instruction Set Computing, обчислення зі скороченим набором команд. Архітектура процесорів зі скороченим набором команд.
DMA – Direct memory access, технологія прямого доступу до пам’яті комп’ютера, минуючи центральний процесор.
RTC – Real-time clock, годинник реального часу, комп'ютерний годинник (найчастіше у вигляді інтегральної схеми), який відстежує поточний час.
CISC – complex instruction set computer, комп’ютер з повним набором команд.

ВСТУП

Оцінка актуальності розробки драйверів під операційну систему Linux. В епоху панування та розвитку технологій людина створює нові пристрої для вимірювання, спостереження, зберігання інформації в першу чергу для покращення та спрощення свого існування. Завдяки швидкому розвитку в комп'ютерній техніці та значним розширенням можливостей було створено величезну кількість різновидів обчислювальних машин для виконання різних задач, проте з кожним новим поколінням, з кожною новою архітектурою виникала інша проблема – налаштування та створення можливості взаємодії самої людини з технікою.

Саме тому людство створило зручне рішення - операційні системи, що допомагали нам вирішити такі проблеми, як: багатозадачність та можливість виконання декількох операції за одиницю часу, розподілення ресурсів комп'ютера між задачами, зберігання та обробка результатів, вирішення проблем конкурентності між задачами за ресурси комп'ютера і найголовніше – створення можливості узгодженої роботи між окремими фізичними модулями комп'ютера. Саме цю можливість узгодженості пристроїв створюють окремі модулі операційної системи – драйвера.

Чому ж саме Linux в наш час займає перші місця по використанню на різних апаратних платформах, таких як сервери, смартфони, мейнфрейми, суперкомп'ютери, центральні комп'ютери автомобілів, маршрутизатори мереж, ігрові консолі тощо? Перша та найголовніша причина – Linux являє собою вільне програмне забезпечення з відкритим кодом, що на відміну від інших актуальних операційних систем на сьогодні є досить важливим фактором в комп'ютерній індустрії. Вихідний код Linux доступний всім для використання, зміни та поширення абсолютне вільно, а головне – безкоштовно.

Актуальність роботи та підстави для її виконання. Весь час на ринку з'являються нові прилади, апаратні платформи, поновлються старі технології, створюются нові канали зв'язку. Саме тому людство завжди потребує можливості налагодженої роботи з новими пристроями, а головне – простим, "абстрагованим" способом використовувати новий функціонал без будь-яких потреб у додаткових знаннях. Все це нам допомогає отримати операційна система та драйвери, що вона використовує для створення налагодженості та взаємодії між окремими частинами комп'ютера.

Саме тому Linux, як один з найкращих варіантів легко-адаптавного відкритого програмного забезпечення, буде продовжувати розвиватися та займати лідируючі місця для вбудованих платформ, серверів, мейнфреймів, смартфонів та багатьох інших пристроїв, а разом з ним буде рости потреба в покращенні та

розробки нових драйверів, створенні нових стандартів. Саме тому цей напрямок буде актуальний ще довгий час.

Мета й завдання роботи. Метою роботи є дослілити електронні засоби, на основі яких потрібно реалізувати програмне забезпечення для налаштування взаємодії функціональних блоків комп'ютера між собою та операційною системою для можливості роботи користувача з інфромацією з відповідних пристроїв. Для досягнення цієї мети поставлено такі завдання:

- Дослідження операційної системі Linux, її особливості та можливості.
- Дослідити різновили пристроїв, з якими працює операційна система Linux та прикладні програмні інтерфейси для роботи з ними.
- Розробка примітивних прикладів драйверів для роботи з простими пристроями на основі отриманих знань.

Об'єкт, методи й засоби розроблення. Об'єктом роботи є процес дослідження поведінки різних типів пристроїв та налаштування їх працездатності за допомогою драйверів, як програмного забезпечення, завдяки якому операційна система (Linux) отримує доступ до апаратної складової комп'ютера для виконання операцій, налаштувань, обробки інформації тощо. Предметом роботи є набір програмного забезпечення операцій взаємодії цих функціональних блоків між собою.

Методи розроблення: комп'ютерне моделювання, методи роботи з різними типами пристроїв, інформаційних шин комп'ютера, розробка програмного продукту на основі сучасної операційної системи.

Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки Eclipse IDE 4.11, мова програмування C, GNU toolchain (набір необхідних пакетів програм для компіляції та генерації виконуваного коду з текстів програм на мові C).

Результати роботи: виконано загальний огляд електронних засобів, налагодження роботи та взаємодії електронних засобів з вбудовуваним одноплатним комп'ютером, на основі процесора архітектури ARM, BeagleBone Black, проаналізовано примітиви та методи ядра Linux, переваги та недоліки використання рзіних методів, структур даних, стандартів та інтерфейсів для роботи з окремими модулями комп'ютера, розроблено програмні продукти (драйвера та модулі), які дозволяють наочно демонструвати процеси роботи та взаємодії різних апаратних засобів між собою, а також зчитувати інформацію з пристроїв на рівні операційної системи - як в просторі ядра, так і в просторі користувача.

Взаємозв'язок з іншими роботами. За методами розробки та інструментальними засобами робота виконувалася сумісно з прикладами драйверів, реалізованих в операційній системі Linux.

Можливі сфери застосування. Програмні продукти можуть застосовуватися в навчальному процесі університетського курсу системного програмування під час вивчення модулів та драйверів операційної системи для роботи з апаратної складовою комп’ютера.

РОЗДІЛ 1. ОПЕРАЦІЙНА СИСТЕМА LINUX, РОБОТА З РІЗНИМИ АРХІТЕКТУРАМИ ПРОЦЕСОРА КОМП’ЮТЕРА

1.1 Архітектури процесора комп’ютера. Відмінності.

Процесор - основний компонент комп'ютера, призначений для керування всіма його пристроями та виконання арифметичних і логічних операцій над даними.

Архітектура процесора - це набір інструкцій, які можуть використовуватися при складанні програм і реалізовані на апаратному рівні за допомогою певних сполучень транзисторів процесора. Саме вони дозволяють програмам взаємодіяти з апаратним забезпеченням і визначають яким чином будуть передаватися дані в пам`ять і зчитуватися звідти **[1]**.

На даний момент існують два типи архітектур: CISC і RISC. Перша передбачає, що в процесорі будуть реалізовані інструкції на всі випадки життя, друга, RISC - ставить перед розробниками завдання створення процесора з набором мінімально необхідних для роботи команд. Інструкції RISC мають менший розмір і простіші.

Архітектура процесора x86 була розроблена в 1978 році і вперше з`явилася в процесорах компанії Intel і відноситься до типу CISC. Її назва взято від моделі першого процесора з цієї архітектурою - Intel 8086. Згодом, через брак кращої альтернативи цю архітектуру почали підтримувати і інші виробники процесорів, наприклад, AMD. Зараз вона є стандартом для настільних комп`ютерів, ноутбуків, нетбуків, серверів та інших подібних пристроїв. Але також іноді процесори x86 застосовуються в планшетах, це досить звична практика. У x86 є кілька суттєвих	**ARM архітектура** була представлена трохи пізніше за x86 - в 1985 році. Вона була розроблена відомою в Британії компанією Acorn, тоді ця архітектура називалася Arcon Risk Machine і належала до типу RISC, але потім була випущена її поліпшена версія Advanted RISC Machine, яка зараз і відома як ARM. При розробці цієї архітектури інженери ставили перед собою мету усунути всі недоліки x86 і створити абсолютно нову і максимально ефективну архітектуру. ARM чіпи отримали мінімальне енергоспоживання і низьку ціну, але мали низьку продуктивність роботи в порівнянні з x86, тому спочатку вони

недоліків. По-перше - це складність команд, їх заплутаність, яка виникла через довгу історію розвитку. По-друге, такі процесори споживають занадто багато енергії і через це виділяють багато теплоти. Інженери x86 спочатку пішли шляхом отримання максимальної продуктивності, а швидкість вимагає ресурсів.	не завоювали велику популярність на персональних комп`ютерах. На відміну від x86, розробники спочатку намагалися отримати мінімальні витрати на ресурси, вони мають менше інструкцій процесора, менше транзисторів, але і відповідно менше всяких додаткових можливостей. Але за останні роки продуктивність процесорів ARM поліпшувалася. З огляду на це, і низьке енергоспоживання вони почали дуже широко застосовуватися в мобільних пристроях, таких як планшети і смартфони.
X86 кращий у таких напрямках: продуктивність та швидкість виконання команд та операцій (проте в наші дні створюють все більше ARM процесорів, потужність яких на рівні з X86 процесорами);	**ARM кращий у таких напрямках:** виробництво процесорів значно дешевше і простіше, велика кількість виробників на ринку через можливість отримання ліцензії на випуск власного процесора; низьке споживання електроенергії, низьке тепловиділення, що дозволяє економити в плані охолодження (проте сучасні x86 не сильно відстають в економічності споживання електроенергії порівняно з ARM);

Таб. 1. Порівняння архітектур процесорів

Існують і інші архітектури процесорів (як приклад – MISC - Minimal Instruction Set Computer — «комп’ютер с мінімальним набором команд»), проте найбільш популярні на ринку саме x86 і ARM архітектури.

Звідси виникає закономірне питання: чому багато хто все ще використовують CISC, коли є RISC? Вся справа в сумісності. x86_64 все ще лідер в desktop-сегменті і тільки з історичних причин. Більшість старих програм та величезна кількість інструментів реалізовані тільки на x86, тому і нові

desktop-системи повинні бути x86, щоб всі старі програми, ігри, утиліти і тому подібне могли працювати на новій машині.

Для Open Source це здебільшого не є проблемою, так як користувач може знайти в інтернеті потрібну версію програми або ж самому оптимізувати та скомпілювати сирцевий код під під іншу архітектуру. На жаль, зробити ж версію платної, з закритим вихідним кодом, програми під іншу архітектуру може тільки власник цього вихідного коду програми.

1.2 Операційна система Linux

Операці́йна систе́ма — це базовий комплекс програм, що виконує управління апаратною складовою комп'ютера або віртуальної машини; забезпечує керування обчислювальним процесом і організовує взаємодію з користувачем.

До **UNIX-подібних ОС [25]** відноситься велика кількість операційних систем, котрі можна умовно поділити на три категорії — System V, BSD та Linux. Сама назва «UNIX» є торговою маркою, що належить «The Open Group», котра власне й ліцензує кожну конкретну ОС на предмет того, чи відповідає вона стандарту. Тому через ліцензійні чи інші неузгодження деякі ОС, котрі фактично є UNIX-подібними, не визнані такими офіційно.

Системи UNIX запускаються на великій кількості процесорних архітектур. Вони широко використовуються як серверні системи у бізнесі, як стільничні системи у академічному та інженерному середовищі. Тут популярні вільні варіанти UNIX, такі як Linux та БСД-системи. Окрім того, деякі з них останнім часом набувають широкого поширення в корпоративному середовищі, особливо це стосується орієнтованих на кінцевого користувача дистрибутивів Linux, в першу чергу Ubuntu, Mandriva, Red Hat Enterprise Linux та Suse. Linux також є популярною системою на стільницях розробників, системних адміністраторів та інших IT-спеціалістів.

Linux (повна назва - GNU/Linux) - загальна назва UNIX-подібних операційних систем на основі однойменного ядра. Це один із найкращих прикладів розробки вільного (вільного) та відкритого (з відкритим кодом) програмного забезпечення (програмне забезпечення). Плюси операційної системи Linux:

- Безпека. Linux не може приховати свої недоліки. Його код переглядається багатьма експертами, котрі роблять свій внесок до кожного випуску нової версії ядра.
- Стабільність і надійність роботи.
- Модульність. Може включати тільки те, що система потребує, навіть під час роботи.
- Легко програмувати. Можливість вчитися на прикладі існуючого коду.
- Багато корисних ресурсів в інтернеті. Повна підтримка мережі.
- Портативність і апаратна підтримка. Запускається на більшості архітектур, через що набуває великої популярності на ринку смартфонів, планшеті.
- Масштабованість. Може працювати як на суперкомп'ютерах, так і на маленьких пристроях (достатньо 4 Мб оперативної пам'яті).

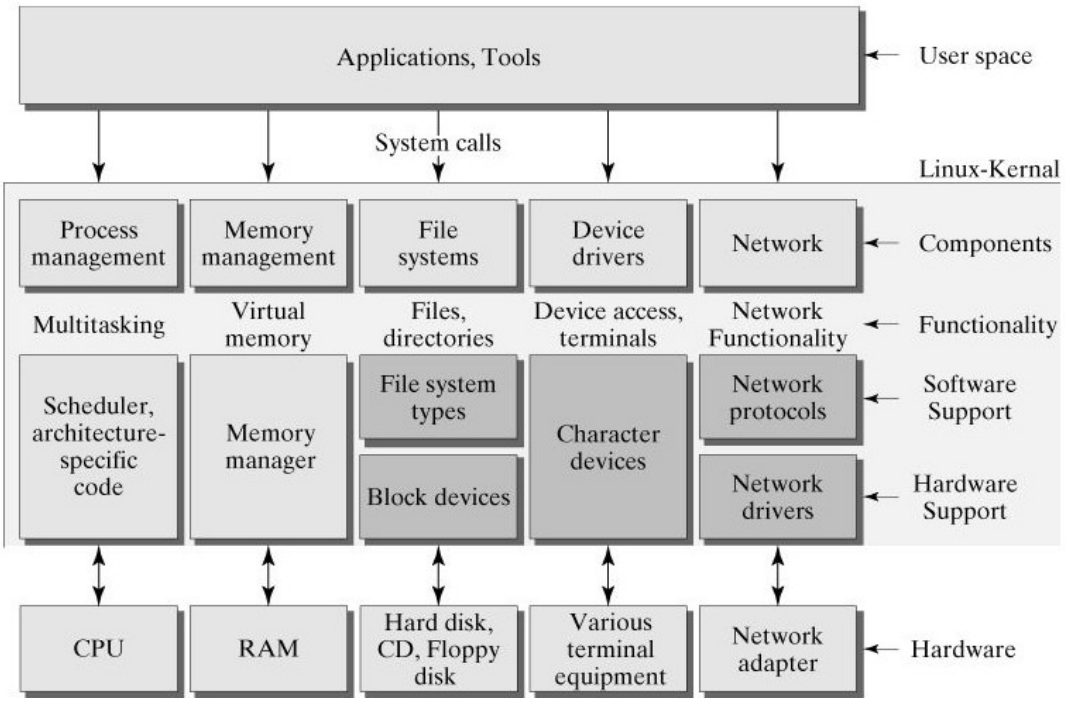


Рис.1 Архітектура ядра Linux. [2]

Ядро поділене, в плані доступу до ресурсів комп’ютера, на 2 частини:

1. Простір користувача (user space), в якому відбувается виконання програм користувача. Цей простір спрощує (абстрагує) розуміння роботи програм та вихідного коду користувачів, дозволяє уникнути проблем з розподіленням пам’яті комп’ютера, розподіленням процесорного часу,

рішенням проблеми конкуретності за ресурси, методи використання периферійниих пристроїв, формат збереження інофрмації, тощо.

2. За рішення всіх вище описаних проблем відповідає простір ядра – простір Linux-Kernel. Для виконання вихідного коду користувача в найбільш оптимальних умовах простір користувача має постійних звязок з простором ядра у вигляді використання System calls – системні сигнали, які викликають стандартні бібліотеки в просторі користувача, тоді як за їхню обробку відповідає простір ядра. Створення нової програми, отримання доступу до файлу, отримання пакетів інформації з мережі чи простий запуск виконуваного файлу – все це виконують підсистеми ядра, такі як менеджер виконуваних процесів, менеджер пам'яті, файлові системи, драйвера та підсистема для роботи з мережею. У кожної підсистеми є власний функціонал разом зі спеціальним програмним забезпеченням, присутнім всередині ядра Linux, завдяки якому підсистеми взаємодіють з апаратною частиною комп'ютера.

1.3 Компіляція програм та драйверів під різні архітектури центрального процесора. Використання toolchains для компіляції програм.

GNU toolchain — набір необхідних пакетів програм для компіляції та генерації виконуваного коду з сирцевих текстів програм **[3]**.

GNU toolchain в першу чергу означає каскадно з'єднаний набір інструментів, що мають спільне завдання - перетворення програми, написаної на мові програмування високого рівня, в виконуваний файл. GCC - лише одна з ланок цього ланцюжка, яким, при всій його важливості, необхідно кооперуватися з іншими ланками. У цьому джерело більшості проблем збірки тулчейнів.

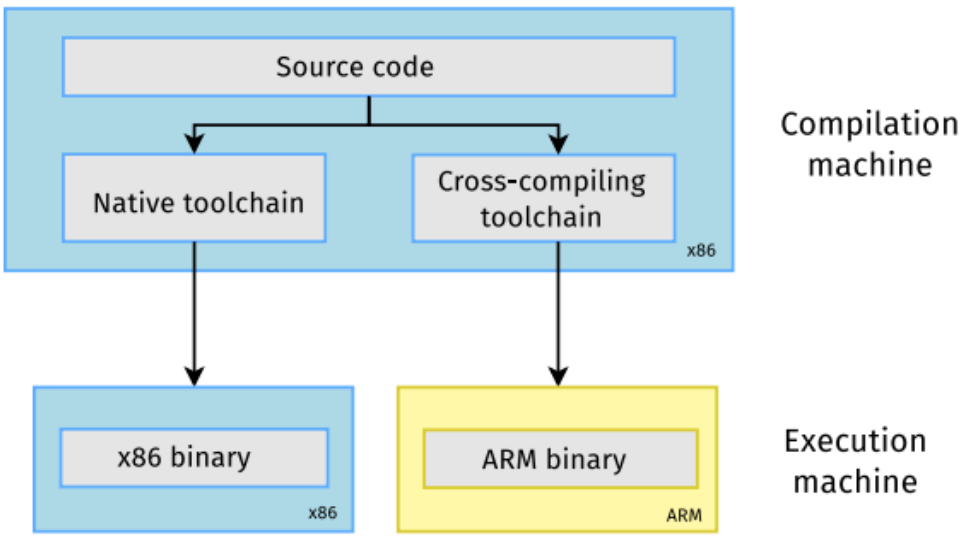


Рис. 2. Принцип роботи toolchain для компіляції коду під різні архітектури.

Набір програм, що потрібні розробнику для крос-компіляції (компіляції програм під іншу архітектуру процесора):

- gcc - *GNU Compiler Collection* — набір компіляторів проекту GNU.
- binutils: ld, as, objdump, objcopy, readelf, та інші – набір інструментів та утиліт для лінковки та роботи з об’єктними файлами коду,
- glibc – стандартний набір бібліотек мови C, та інші системні бібліотеки
- загаловні файли Linux
- gdb – GNU debugger – відлагоджувач

В нашому випадку для роботи з драйверами потрібні 2 типи toolchain:

• **Bare-metal targeted** (arm-eabi): для компіляції самого ядра за завантажувача Uboot - комп'ютерний завантажувач операційної системи, орієнтований на вбудовані пристрої архітектур MIPS, PowerPC, ARM і інші.

• **Linux targeted** (arm-linux-gnueabihf): для BusyBox - набір UNIX-утиліт командного рядка (у вигляді одного файлу), котрий є майже повним POSIX-середовищем для вбудованих та інших систем з невеликим обсягом доступного місця.

Компіляція відбувається на хості з архітектурою = x86_64, під архітектуру = ARM.

РОЗДІЛ 2. ЯДРО LINUX. РЕАЛІЗАЦІЯ МОДУЛІВ НА ОСНОВІ МЕТОДІВ ТА ПРИМІТИВІВ ЯДРА LINUX

2.1 Налаштування та компіляція модулів під ядро Linux.

Перш ніж приступати до вивчення особливостей та можливостей ядра Linux та написання більш складних драйверів, розглянемо приклад написання та компіляції досить простого та звичного модуля “Hello World”.

Але для початку треба скомпілювати саме ядро Linux, під яке і будемо збирати наші модулі. Відповідно нам треба скомпілювати ядро під архітектуру ARM, тому як всі драйвера та модулі надалі будуть писатися під міні-комп’ютер BeagleBone Black. Стосовно того як саме налаштувати та скомпілювати ядро під плату BBB **[4]**.

Почнемо з прикладу коду модуля ядра, пізніше напишемо makefile під цей модуль:

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

MODULE_AUTHOR(“Mykyta Opanyuk <mykyta@opanyuk.com”);
MODULE_DESCRIPTION(“Hello, world in Linux Kernel Training”);
MODULE_LICENSE(“Dual BSD/GPL”);

static int __init hello_init(void) {
        printk(KERN_EMERG “Hello, world!\n”);
return 0;
}

static void __exit hello_exit(void) {
        /* Do nothing here right now */
}

module_init(hello_init);
module_exit(hello_exit);
```

Рис. 3. Приклад драйверу hello_world.

Давайте розглянемо цей модуль детальніше:

- На перших рядках ми включаємо <linux/init.h>, <linux/module.h> і файли заголовків <linux/printk.h> з дерева ядра Linux. Перший використовується

для забезпечення макросів __init / __ exit, другий для забезпечення макросів MODULE _ *() та module_init() - функція ініціалізації / module_exit() - функції деініціалізації, третій - для забезпечення прототипу функції printk(), а також KERN_EMERG define. KERN_EMERG - це найвищий рівень для логування програми.

- hello_init() викликається під час ініціалізації модуля. Зазвичай деякі статичні дані ініціалізуються тут, виділяються деякі динамічні дані або виконуються інші дії. У нашому випадку ми просто надрукуємо рядок “Hello, world!”. Зверніть увагу, що повернення негативного значення errno (наприклад, -EINVAL) з цієї функції призводить до помилки модуля init і запобігання завантаження модуля.
- Далі, hello_exit(), викликаний під час вивантаження модуля, не може вийти з ладу, тому повернення значення тут не приймається.

Тепер настав час створити **Makefile [5]** і побудувати наш модуль:

Гарна практика - створити загальний файл makefile, який дозволяє користувачеві будувати зовнішній модуль ядра за допомогою простого виклику make або make <target>. Однак ми знаємо, що Linux Kernel використовує makefile стилю kbuild, який не є таким же, як звичайні файли Makefile, які будуть виконані безпосередньо командою оболонки make. Саме тому далі вказаний приклад можна використовувати під будь які інші драйвера, змінюючи назву obj-m.

```
ifneq ($(KERNELRELEASE),)

# kbuild part of makefile

CFLAGS_hw2.o := -DDEBUG
obj-m := hello.o

else

# normal makefile

KDIR ?= /lib/modules/$(shell uname -r)/build

module:
	$(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILING) -C $(KDIR) M=$(PWD) C=1 modules

clean:
	$(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILING) -C $(KDIR) M=$(PWD) C=1 clean

.PHONY: module clean

endif
```

Рис. 4. Приклад Makefile.

Для того, щоб виклик make був робочий — треба задати значення з**мінної оболочки shell** та експортувати ці змінні в Makefile**:**

$ export ARCH=arm
$ export KDIR=/home/mykyta/Learning_linux_kernel_BBB/BeagleBone_Black/linux-stable
$ export CROSS_COMPILING=/opt/gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

Проаналізувавши Makefile — можна побачити, що make вклиикається тут в 2 варіантах:

- make module — практично присвоєння значень для ARCH та CROSS_COMPILE та виконання компіляції модуля під вказану архітектуру під вказане скомпільоване ядро з використанням вказаних інструментів крос компіляції.
- make clean — видалення всіх створених тимчасових файлів (об’єктні файли) та самих скомпільованих модулів ядра.

Після того, як скомпілюємо модуль — отримаємо відповідно hello.ko (kernel object file), який відповідно можна завантажити у систему:
$ insmod hello.ko (виклик hello_init())- після чого в журналі ядра :
$ dmesg — можемо вичитати повідомлення Hello world.
Вивантажити з системи модуль :
$ rmmod hello.ko (виклик hello_exit())

2.2 Базові структури даних ядра

Ядро - це окрема частина програмного забезпечення, що не лише імплементує використання бібліотек мови С. Воно реалізує величезну кількість інших механізмів, з якими можна зіткнутися в більш сучасних розширених бібліотеках, а саме різні типи даних, різновиди реалізацій багатопочності та методів синхронізації в ядрі, стиснення, хешування, пошук інформації тощо.

Зв'язані списки (Linked lists) в ядрі Linux [6]:

Найбільш поширені, прості і зручні структури даних. Загалом розробники вільні у виборі реалізації. Вони можуть або використовувати власні структури даних і примітиви маніпулювання списком (ітератори, помічники вставки / видалення і т.д.), але спочатку краще подивитися ядро Linux, раптом схожий тип даних вже реалізований.

Списки також можуть бути подвійними, де кожен вузол має вказівник на попередній елемент списку. Також список може завершуватися відповідним NULL або вказувати на ту ж голову заглушки (або ж — як варіант використання циклічних списків).

У найпростішому випадку однонаправлений зв'язаний список може виглядати наступним чином:

```
struct my_data {
	struct my_data *next;
	unsigned long canary;
};
```

Стандартні списки, пов'язані з ядром Linux, реалізовані у вигляді подвійних зв'язаних списків.

У поширених випадках вони використовують заглушку для зберігання посилання на весь список, а не просто вказівник на перший (останній) елемент списку. Ця властивість використовується більшістю користувачами примітивів маніпулювання списками, а також користувачами ітераторів.
Ці списки є загальними, вбудовуються в поворотну структуру даних клієнтів цих структур у зв'язаному списку. Тип пов'язаного списку настільки поширений в ядрі, тому його оголошують у <linux/types.h>. Загальні приміти для маніпулювання зв'язаними списками оголошені в <linux / list.h>. У <linux/rculist.h> існують примітиви зі списку варіантів RCU. Як очікується, складність O(N) для обходу списку і O(1) для маніпулювання списком.

Нижче наведено короткий огляд того, як перетворити власну структуру даних у зв'язаний список можна маніпулювати стандартними примитивами списку.

```
	#include <linux/types.h>
	#include <linux/list.h>

	struct my_data {
		struct list_head list_node;
		struct list_head another_list_node;
		unsigned long canary;
	};
```

Саме на основі зв’язних списків реалізується велика кількість інших структур даних, такі як стеки tasklet, workqueue та інші, що будут розглянуті в наступному розділі.

Хеш-таблиці в ядрі Linux

Крім зв'язаного списку в ядрі Linux є ще один, більш потужні структури, які можна використовувати для реалізації розширеного сховища даних в ядрі. Це хеш-таблиці **[7]**, які реалізуються подібним, загальним способом як зв'язані списки в ядрі Linux. В першу чергу - особливість і відмінність хеш-таблиць в Linux полягає в тому, що існує декілька доступних допоміжних примітивів, а також структур даних, які можна вбудовувати у власні структури даних, щоб дозволити їх використання у вигляді хеш-таблиць. Крім того Linux реалізовані перевірені часом, швидкі хеш-функції для генерації хеш-ключів. Для реалізації хеш-таблиць для зберігання даних можуть бути використані наступні компоненти:

1. Тип даних для вбудовування в власні структури даних. Оголошено в <linux/types.h>.

2. Один розмірний масив, що служить таблицею хеш-відра. Для цього найпростіший випадок може бути визначений як статичний масив фіксованого розміру, але якщо необхідна масштабованість під великий обсяг даних, то зазвичай реалізується масив з динамічним розміром.

Рис. Хеш-таблиця:

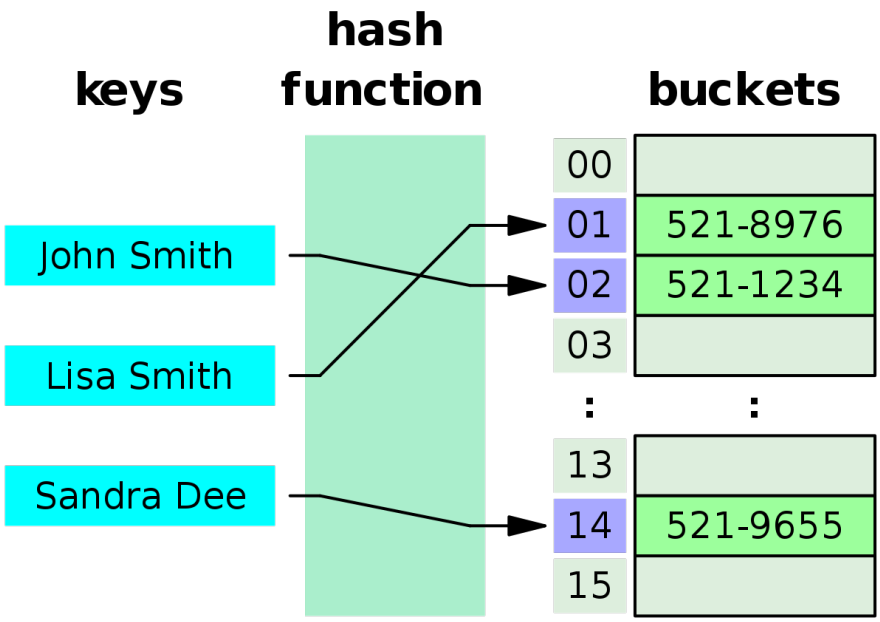


Рис. 5. Приклад використання хеш функції.

Двійкові дерева пошуку (Binary Search Tree) в ядрі Linux

BST - це двійкове дерево з збереженим ключем і необов'язковим значенням, відповідним цьому ключу, і посиланнями на два піддерева, звані відповідним чином "лівим піддеревом" і "правим піддеревом" з наступними властивостями, застосованими до кожного вузла:

- Ключ лівого піддерева завжди менше, ніж ключ батьківського вузла;
- Ключ в правому піддереві завжди більше, ніж ключ батьківського вузла;
- Час складності BST на звичайних операціях, таких як вставка, видалення та пошук (O(log N) у середньому або кращому випадку, коли дерево добре збалансоване. O(N) для найгіршого випадку, коли дерево вироджується до зв'язаного списку (і зауважимо, що складність операцій вставки та видалення списку також O(N)));
- Бінарні дерева пошуку перш за все являються будівельними блоками для більш високорівневих структур, таких як набори, асоціативні масиви словники та навіть файлові системи.

Навіщо використовувати дерева, якщо хеш-таблиці мають кращу тимчасову складність у середньому випадку?

- Деяка інформація має структуру дерева (наприклад, словник чи файлова система);
- Ефективність використання пам'яті: не потрібно додатково пам’яті для масиву хеш-відра;
- Немає необхідності підтримувати розмір масиву хеш-таблиць, коли кількість даних зростає/зменшується;
- Дерева, при правильному розміщенні вузлів, добре кешуються;
- Усі ключі або значення можуть бути отримані в певному порядку шляхом обходу (наприклад, зліва направо);
- Хороша постійна середня тимчасова складність, коли BST реалізується як збалансоване дерево;

2.3 Поняття часу, затримок, відкладеної роботи та обробки переривань в Linux

Джерела часу в ядрі Linux:[8][25][26][27]

- RTC (Real Time Clock): на основі годинника, вбудованого в плату (Hardware clock). Використовується для встановлення та збереження поточної дати і часу, навіть коли система вимкнена;

- Системні таймери (низька роздільна здатність): kernel/time/timer.c: існують з різною тактовою частотою лічильники (100,250,1000 Гц); Підтримка системного часу; На основі системних таймерів реалізовані планування завдань і вікладених подій (waitqueue, tasklets, workqueue, тощо);

- Таймери високої точності: kernel/time/hrtimer.c: Може підтримувати роздільну здатність такту, що перевищує за точністю 1 мс. Вказані годинники можуть підтримувати роздільну здатність 1 наносекунду, зазвичай округлюється до роздільної здатності годинника певної платформи.

Одиниці вимірювання часу в ядрі Linux:

Jiffies - лічильник, який збільшувався при кожному перериванні системного годинника. Залежно від архітектури процесора платформи, jiffies можуть бути:

- 32 бітні, з частотою 1000 Гц: близько 50 днів, після чого значення jiffies обнуляється;
- 64 бітні, 1000 Гц: вистачає близько на 600 мільйонів років;

На 32 бітах, jiffies вказує на низький порядок 32 біта, jiffies_64 на біти високого порядку (тобто значення jiffies зберігається в 2 змінних системного годинника), тоді як на 64 бітних процесорах використовують лише jiffies_64;

HZ, визначає частоту переривання годинника, яка за замовчуванням 1000 на x86 або 1 мілісекунда. Налаштувати значення частоти переривання можна під час компіляції або часу завантаження ядра Linux.

- Інші типові значення - 100 (10 мс) або 250 (4 мс).
- Низькі значення HZ: менше накладних витрат.
- Великі значення HZ: краща точність роботи годинника.

Приклад таймерів в ядрі Linux (**struct timer_list *new_timer**):

```
timer_setup(&new_timer, timer_func, 0); /* new API for init timers */
mod_timer(&new_timer, jiffies + 1 * HZ); /* set time for timer callback */
```

Вище — ініціалізація системного таймера, та встановлення часу його спрацювання - виклик функції обробника:

```
static void timer_func(struct timer_list *unused) /* timer callback func*/
{
        printk(KERN_ALERT "We are at timer_func!\n");
        printk(KERN_ALERT "Getting flags from timer : %u\n", unused->flags);
        printk(KERN_ALERT "Jiffies at timer_func : %lu\n", jiffies);
        printk(KERN_ALERT "Ret value after timer_func = %u!\n<->\n", ret);

        ret++;

        tasklet_schedule(&new_task);

        if (ret < 5) /* reset a new callback time for a timer 5th times */
                mod_timer(&new_timer, jiffies + 1 * HZ);
}
```

Рис. 6. Приклад функції-обробника таймера.

Відповідно при завершені роботи модуля — зв’язний список таймерів треба звільнити.

del_timer(new_timer);

Ознайомившись з таймерами в ядрі, можна перейти до поняття **відкладеної роботи** в операційній системі.

Загальна проблема будь-якого **ISR [26] (Interrupt Service Routine – обробник переривань)** - це затримка. Оскільки найчастіше виконання всіх інших процесів та переривань відключено під час виконання ISR, ISR, як очікується, буде коротким. Але, що, якщо потрібно зробити обробку величезної кількості даних, виділення пам'яті, виконання складних арифметичних операцій в ISR? Linux долає цю проблему, надаючи в ядрі інфраструктуру, на яку можна розділити ISR:

- **Top-Half — верхня половина:** Це критична секція, в якій інші процеси та переривання блокуються. Крім того в цій половині переривання, що обробляється, виконується в атамарному контексті, а тому
- **Bottom-Half – нижня половина:** Менш критична секція, яка не впливає на виконання інших процесів та переривань.

Не обов'язково розділяти ISR в Linux. Це вибір розробників драйверів пристроїв. Якщо розробник відчуває, що ISR буде дуже коротким за виконанням і може бути керованим, то він може не реалізовувати нижню

половину. Аналогічно, якщо розробник драйвера вважає, що відключення переривання на тривалий час є розумний, то він може ще мати верхню половину з багатьма операціями. Отже, це дизайнерське рішення. Linux надає деякі технічні засоби для реалізації нижньої половини.

Існує 3 набільш популярних способів обробки переривань в ядрі:

- **Tasklet [10][[26]** — як один з різновидів обробників переривань верхньої половини. Tasklet-и виконуються в атамарному контексті, і не можуть спати під час обробки переривання. Tasklet потрібно обов'язково ініціалізувати, перш ніж з ними працювати.

```
tasklet_init(&fake_dev.taskl, taskl_func, (unsigned long)&fake_dev);
```

і лише після цього можна планувати виконання обробки в верхній половині, і при успішній обробці повертати IRQ_HANDLED. Запланований tasklet виконуються зразу, якщо в системі в даний момент не виконується обробка в верхній половині.

Приклад функції-обробника tasklet:

```
/* tasklet function */
static void taskl_func(unsigned long param)
{
        struct fake_dev_ctx *p_ctx = (struct fake_dev_ctx*)param;
        unsigned long long num = 0;

        num = p_ctx->button;

        printk(KERN_ALERT "number of button up/down %llu\n", num);
}
```

Рис. 7. Приклад функції-обробника tasklet.

\ Відповідно, при завершені роботи модуля — треба звільнити tasklet.

tasklet_kill(tasklet_struct *new_task);

- **Workqueue [11][26]** - це більш складні і важкі сущності, ніж tasklet'и. Workqueue, як і tasklet'и, служать для відкладеної обробки переривань (хоча їх можна використовувати і для інших цілей), але, на відміну від tasklet'ов, виконуються в контексті kernel-процесу, відповідно, вони не зобов'язані бути атомарними і можуть використовувати функцію sleep(), різні засоби синхронізації і т.п.

На зображенні він показаний дуже наближено і спрощено, як все відбувається насправді, детально описано нижче.

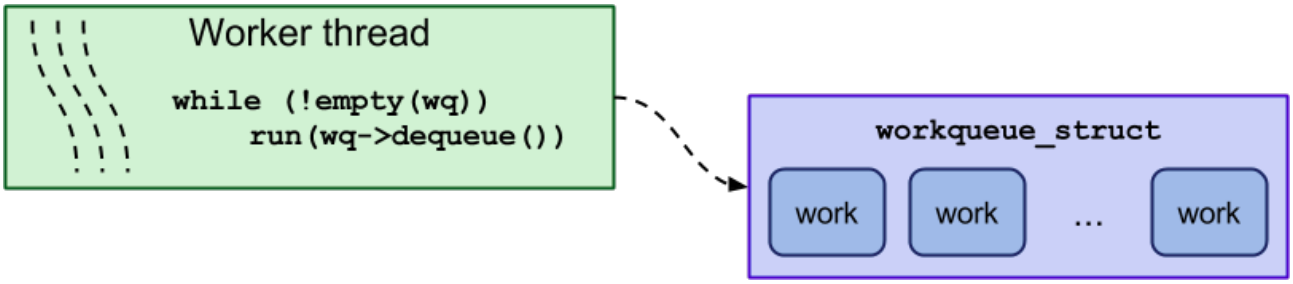


Рис. 8. Механізм роботи workqueue.

По-перше, **work item** (для стислості просто **work**) - це структура, що описує функцію (наприклад, обробник переривання), яку ми хочемо запланувати Його можна сприймати як аналог структури tasklet. Tasklet'и при плануванні додавалися в черзі, приховані від користувача, тепер же нам потрібно використовувати спеціальну чергу — **workqueue**.

Tasklet'и розгрібаються функцією-планувальником, а workqueue обробляється спеціальними потоками, які звуться **worker**'ами. Worker'и забезпечують асинхронне виконання work'ов з workqueue. Хоча вони викликають work'і в порядку черги, в загальному випадку про суворе, послідовне виконання мови не йде: все-таки тут мають місце витіснення, сон, очікування і т.д. Взагалі, worker'и - це kernel-потоки, тобто ними керує основний планувальник ядра Linux. Але worker'и частково втручаються в планування для додаткової організації паралельного виконання work'ов. Приклад буде пізніше, коли розглянемо device tree.

- **Реєстрація обробника переривань [12][26][27] (Registering an Interrupt Handler)** — процес створення налаштованого обробника переривань. Вказаний обробник можна налаштувати як під роботу в верхній половині, так і в нижній половині. Крім того — налаштовувати вказаний обробник можна і під тип переривань, що отримує система.

Приклад реєстрації обробника:

```
ret = request_irq(irqNumber, button_irq_handler_up_down,
        IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING, "user boot", &fake_dev);
```

irqNumber — номер відповідного IRQ, по якому будуть посилатися переривання (нижче : виклик функції **gpio_to_irq()**). В цьому випадку — переривання GPIO контакту, до якого під'єднана кнопка. Нижче можна побачити процес перевірки працездатності вказаної за номером GPIO2_8 лінії GPIO — виклики **gpio_request()**. **gpio_direction_input()** — інформує мікроконтролер GPIO як саме буде працювати лінія — в даному випадку на введення даних.

```
ret = gpio_request(GPIO2_8, "user_boot");
if (ret != 0)
        printk("=== gpio_request FAILED. ret = %d;\n", ret);

ret = gpio_direction_input(GPIO2_8);
if (ret != 0)
        printk("=== gpio_direction_input FAILED.\n");

irqNumber = gpio_to_irq(GPIO2_8);
if (irqNumber == -EINVAL)
        printk("=== gpio_to_irq FAILED.\n");

printk("=== The button is mapped to IRQ: %d\n", irqNumber);
```

Рис. 9. Приклад ініціалізації gpio-переривань.

Всередині функції можна побачити виклик запланованого виконання іншого типу обробників переривань Tasklet.

2.4 Поняття паралелізму та стану гонки між потоками. Методи синхронізації

У програмах, що використовують спільну пам'ять, розробники повинні забезпечити захист спільних ресурсів від одночасного доступу. Ядро не є винятком. Спільні ресурси вимагають захисту від одночасного доступу, оскільки, якщо кілька потоків виконання мають доступ і керують даними одночасно, потоки можуть перезаписати зміни один одного або отримати доступ до даних, коли вони перебувають у вільному стані. Паралельний доступ до загальних даних часто призводить до нестабільності, яку важко відстежувати і налагоджувати. **[13][26][25]**

Де може зустрітися ситуація конкурентності, паралелізму:

- Задачі в ядрі (приклад **work queue**);
- Обробник переривань (**Реєстрація обробника переривань, синхронізація всередині функції обробника);**
- Процеси, що виконуються не в атомарному контексті, нижня половина;
- Потоки ядра;

У ядрі 2.0 була введена симетрична підтримка багатопроцесорної обробки. Підтримка багатопроцесорної обробки означає, що код ядра може одночасно працювати на двох або більше процесорах. Отже, без захисту, код в

ядрі, що працює на двох різних процесорах, може одночасно отримувати доступ до спільних даних точно в той же час. З введенням ядра 2.6 ядро Linux стало попереджуючим. Це означає, що (за відсутності захисту) планувальник може вивантажити код ядра практично в будь-яку точку і перенести інше завдання. Сьогодні ряд сценаріїв дозволяють для паралелізму всередині ядра, і всі вони потребують захисту.

Приклад паралелізму та змагань за ресурси розглянемо в розділі про Драйвери пристроїв платформи та драйверів символьних пристроїв.

2.5 Виділення пам’яті в Linux. Робота з підсистемами розподілу пам’яті в ядрі

Важливою відмінністю у програмуванні ядра є те, як отримати доступ до пам'яті та виділити її. У зв'язку з тим, що програмування ядра дуже близьке до фізичної машини, існують важливі правила управління пам'яттю.

По-перше, ядро працює з декількома типами пам'яті **[14][26][25]**:

Фізична пам'ять — прямий доступ до оперативної пам’яті в обхід менеджера розподілення пам’яті між процесами (kmalloc() — kernel memory allocation).

Віртуальна пам'ять з адресного простору ядра (**vmalloc** — virtual memory allocation).

Віртуальна пам'ять з адресного простору процесу (**malloc**)

Резидентна пам'ять - ми точно знаємо, що доступні сторінки присутні у фізичній пам'яті для конкретного, і вони не будуть звільнені під виконання інших процесів, та доступ з інших процесів до цих сторінок буде закритий.

Віртуальна пам'ять в адресному просторі процесу не може вважатися резидентною через механізми віртуальної пам'яті, реалізовані операційною системою: сторінки можуть бути замінені або просто не можуть бути присутніми у фізичній пам'яті в результаті механізму пошукового виклику. Пам'ять в адресному просторі ядра може бути резидентною чи ні. Як дані, так і сегменти коду модуля та стек ядра процесу є резидентними. Динамічна пам'ять може бути або не бути резидентною, залежно від її розподілу.

При роботі з резидентною пам'яттю все просто: доступ до пам'яті можна отримати в будь-який час. Але якщо працювати з нерезидентною пам'яттю, то до неї можна звертатися тільки з певних контекстів. Доступ до нерезидентної пам'яті можна отримати лише з контексту процесу. Доступ до нерезидентної пам'яті з контексту переривання має непередбачувані результати і, отже, коли

операційна система виявляє такий доступ, вона прийме радикальні заходи: блокування або скидання системи для запобігання серйозної корупції.

Віртуальну пам'ять процесу неможливо отримати безпосередньо з ядра. Загалом, це абсолютно не рекомендується для доступу до адресного простору процесу, але є ситуації, коли драйвер пристрою повинен це робити. Типовий випадок, коли драйвер пристрою повинен отримати доступ до буфера з простору користувача. У цьому випадку драйвер пристрою повинен використовувати спеціальні функції і не мати безпосереднього доступу до буфера. Це необхідно для запобігання доступу до недійсних областей пам'яті.

Інша відмінність від планування простору користувача, відносно пам'яті, обумовлена стеком, розмір якого є фіксованим і обмеженим. Стек 4K використовується в Linux, а стек 12K використовується в Windows. З цієї причини слід уникати виділення великих структур на стеку або використання рекурсивних викликів.

Найпростішим способом виділення пам'яті є використання функції з сім'ї **kmalloc()**. І, щоб бути на безпечному розмірі, краще використовувати підпрограми, які встановлюють пам'ять до нуля, наприклад **kzalloc()**. Якщо вам потрібно виділити пам'ять для масиву, є **kmalloc_array()** і помічники **kcalloc()**.

Максимальний розмір шматка, який можна виділити за допомогою kmalloc, обмежений. Фактичний ліміт залежить від апаратного забезпечення та конфігурації ядра, але добре використовувати kmalloc для об'єктів, менших розміру сторінки, що дорівнює 4 кілобайтам.

Для великих розподілів можна використовувати сторінки **vmalloc()** і **vzalloc()**, або безпосередньо запитувати сторінки з розподільника сторінок. Пам'ять, що виділяється за допомогою vmalloc та пов'язаних з нею функцій, фізично не є суміжною. Перш за все цю пам’ять нам надає менеджер розподілу оперативної пам’яті.

Якщо ви не впевнені, що розмір виділення занадто великий для kmalloc, можна використовувати **kvmalloc()** та його похідні. Він спробує виділити пам'ять за допомогою kmalloc, і якщо виділення не відбудеться, то буде повторно виконано за допомогою vmalloc. Існують обмеження, за якими не всі

можливі параметри GFP (Get Free Page) можна використовувати з kvmalloc; Зауважимо, що kvmalloc може повернути пам'ять, яка не є фізично суміжною.

Якщо потрібно виділити багато ідентичних об'єктів, можна використовувати розподільник кеша, як один з найкращих варіантів швидкого доступу до пам’яті. Кеш слід налаштовувати за допомогою **kmem_cache_create()** або **kmem_cache_create_usercopy()**, перш ніж він може бути використаний. Другу функцію слід використовувати, якщо частина кешу може бути скопійована до простору користувачів. Після створення кешу за допомогою операції **kmem_cache_alloc()** можна виділяти шматки пам’яті (slab) з цього кешу.

Коли виділена пам'ять більше не потрібна, вона повинна бути звільнена. Ви можете використовувати **kvfree()** для пам'яті, виділеної за допомогою kmalloc, vmalloc і kvmalloc. Для кешованої пам’яті слід застосовувати для звільнення відповідних виділених шматків **kmem_cache_free()**. І головне — треба не забувати знищувати сам проініціалізований кеш за допомогою **kmem_cache_destroy()**.

РОЗДІЛ 3.РОЗРОБКИ ДРАЙВЕРІВ ПІД РІЗНІ ТИПИ ПРИСТРОЇВ

3.1 Драйвери пристроїв платформи. Дерево пристроїв

[15] Основна інформація про дерево пристроїв:

● Визначено стандартом для вимог платформи для вбудованої архітектури (ePAPR);

● ePAPR визначає концепцію, яка називається деревом пристроїв для опису системного обладнання;

● Програма-завантажувач (в моєму випадку u-boot system bootloader) завантажує дерево пристроїв у пам'ять клієнтської програми і передає клієнту вказівник на дерево пристроїв;

● Дерево пристроїв - це деревоподібна структура даних з вузлами, які описують фізичні пристрої в системі;

● Дерево пристрою, сумісного з ePAPR, описує інформацію про пристрої платформи у системі, які не можуть бути динамічно виявлені операційною системою;

Прилад платформи:

У перші дні користувачам Linux часто доводилося говорити ядрі, де конкретні пристрої повинні бути знайдені, перш ніж їхні системи працюватимуть. За відсутності цієї інформації драйвер не міг знати, які порти вводу-виводу і переривання (лінії) пристрою було налаштовано для використання. На щастя, тепер ми живемо в часи шин, подібних до PCI чи USB, які мають вбудовані в них знання; будь-який пристрій, що сидить на шині PCI, може повідомити системі, який це пристрій і де його ресурси. Таким чином, ядро може під час завантаження перераховувати доступні пристрої і легко з ними працювати.

На жаль, життя не таке просте - є багато пристроїв, які все ще не виявляються процесором. У вбудованому світі і в системі на чіпі невидимі пристрої зростають у кількості. Тому ядро все ще має можливості забезпечити способи розповісти про наявні апаратні засоби. "Пристрої платформи" вже давно використовуються в цій ролі в ядрі.

Процес компіляції та завантаження файлу дерева пристроїв. Приклад синтаксису device tree файлу.

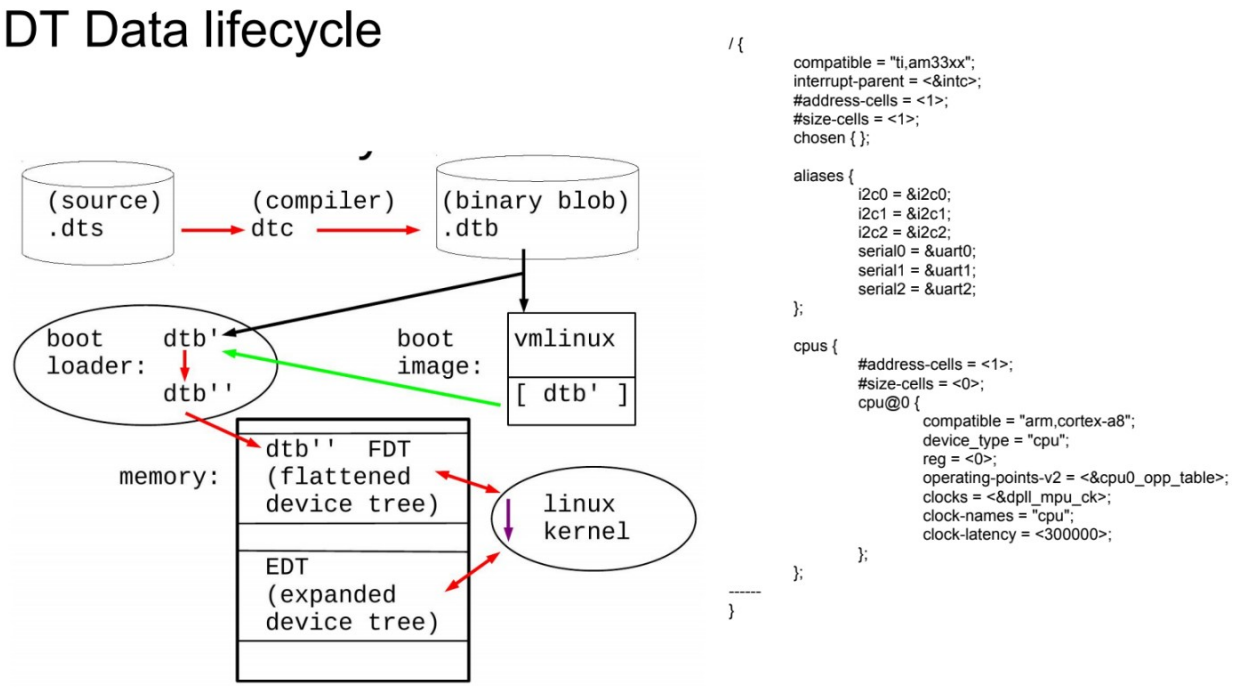


Рис. 10. Приклад ініціалізації DTS файлу та зчитування даних про пристрої платформи в пам’ять. Приклад структури DTS файлу.

● всі вихідні файли компіляції дерева пристроїв (DTS) на даний момент розташовані в arch/XXX/boot/dts, де XXX — відповідна архітектура;
● .dtsi файли для включення (за допомогою директив), які зазвичай містять додаткову інформацію про конкретний SoC;
● Інструмент, компілятор Device Tree Compiler компілює потрібні DTS файли в двійкову форму.
● Device Tree Blob є результатом компіляції і є двійковим файлом, який завантажується u-boot-ом і аналізується ядром Linux під час завантаження.
● arch /XXX/boot/dts/Makefile перераховує, які DTB повинні генеруватися під час збирання.
● Файли Дерева пристроїв не є монолітними, їх можна розділити на декілька файлів;
● Файли .dtsi включаються до файлів, а файли .dts - до кінцевих дерев пристроїв;
● Як правило, .dtsi буде містити визначення інформації на рівні SoC — кристал процесора;
● Файл .dts містить інформацію на рівні плати.

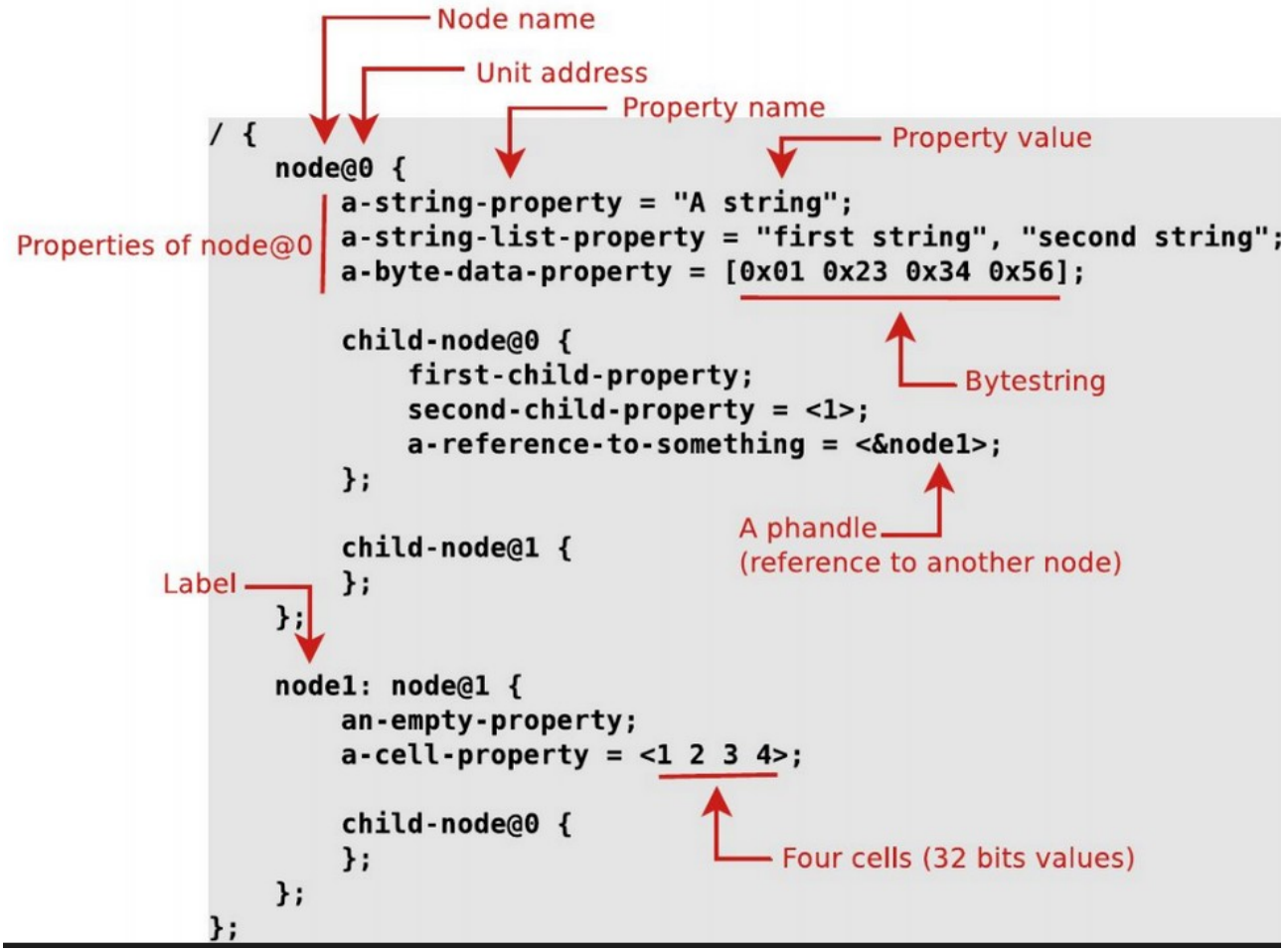


Рис. 11. Синтаксис device tree файлу.

Пристрій платформи **[16][26][27]** представлений структурою struct platform_device, яку, як і інші відповідні декларації, можна знайти в <linux/platform_device.h>. Ці пристрої вважаються підключеними до віртуальної "автобусної платформи"; драйвери платформних пристроїв повинні таким чином реєструватися як такі з кодом шини платформи. Ця реєстрація здійснюється за допомогою структури platform_driver:

Рис. 12. Опис структури platform_driver:

```
static struct of_device_id mxs_auart_dt_ids[] = {
        {
                .compatible = "fsl,imx28-auart",
                .data = &mxs_auart_devtype[IMX28_AUART]
        }, {
                .compatible = "fsl,imx23-auart",
                .data = &mxs_auart_devtype[IMX23_AUART]
        }, { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, mxs_auart_dt_ids);
[...]
static struct platform_driver mxs_auart_driver = {
        .probe = mxs_auart_probe,
        .remove = mxs_auart_remove,
        .driver = {
                .name = "mxs-auart",
                .of_match_table = mxs_auart_dt_ids,
        },
};
```

в структурі of_device_id, значення поля compatible зберігає рядок, який інформує на що орієнтуватися в dts для пошуку та зчитування інформації стосовно конкретного пристрою платформи.

3.2 Пристрої вводу/виводу. Прикладні програмні інтерфейси для роботи з ними. Приклади роботи з реальним пристроєм - матричнична клавіатура. GPIO інтерфейс

У UNIX-подібних системах пристрої доступні користувачеві через спеціальні файли пристроїв. Ці файли згруповані в каталог /dev, а системні виклики відкриті, читання, запис, закриття, lseek, mmap і т.д. перенаправляються операційною системою на драйвер пристрою, пов'язаний з фізичним пристроєм. Драйвер пристрою - це компонент ядра (зазвичай модуль), який взаємодіє з пристроєм.

У світі UNIX існують дві категорії файлів пристроїв і, таким чином, драйвери пристроїв: символьні **[19]** і блочні**[23].** Цей розподіл здійснюється за швидкістю, обсягом і способом організації даних, що передаються від пристрою до системи, і навпаки. У першій категорії існують повільні пристрої, які керують невеликою кількістю даних, а доступ до даних не вимагає частого пошуку запитів. Прикладами є такі пристрої, як клавіатура, миша, послідовний порт, звукова карта, джойстик. Загалом, операції з цими пристроями (читання, запис) виконуються послідовно байтом по байту. Друга категорія включає пристрої, де обсяг даних великий, дані організовані за блоками, а пошук інформації відбуваються часто. Прикладами пристроїв, які підпадають під цю категорію, є жорсткі диски, компакт-диски, паличні диски, магнітні накопичувачі. Для цих пристроїв читання і запис виконуються на рівні блоку даних.

Для двох типів драйверів пристроїв ядро Linux пропонує різні API. Якщо для символьних пристроїв системні виклики переходять безпосередньо до драйверів пристроїв, у випадку блочних пристроїв драйвери не працюють безпосередньо з системними викликами. У разі блокових пристроїв зв'язок між користувацьким простором і драйвером блочного пристрою опосередковується підсистемою керування файлами і підсистемою блочних пристроїв. Роль цих підсистем полягає в тому, щоб підготувати необхідні ресурси драйвера пристрою (буфери), зберегти недавно прочитані дані в буфері кешу, а також замовити операції читання і запису з причин продуктивності.

Коротка характеристика символьного пристрою:

- Доступ до пристроїв здійснюється через імена у файловій системі;
- Ці імена називаються спеціальними файлами або файлами пристроїв або просто вузлами дерева файлової системи (зберігаються в / dev);
- Ядро Linux представляє символьні та блокові пристрої як пари чисел <major>: <minor>.
- Деякі major номера зарезервовані для певних драйверів символьних пристроїв;
- Інші major номери динамічно призначаються драйверу пристрою під час завантаження Linux (інформація з DTS) або при завантаженні драйвера;
- Пристрої одного і того ж major номера належать до одного класу;

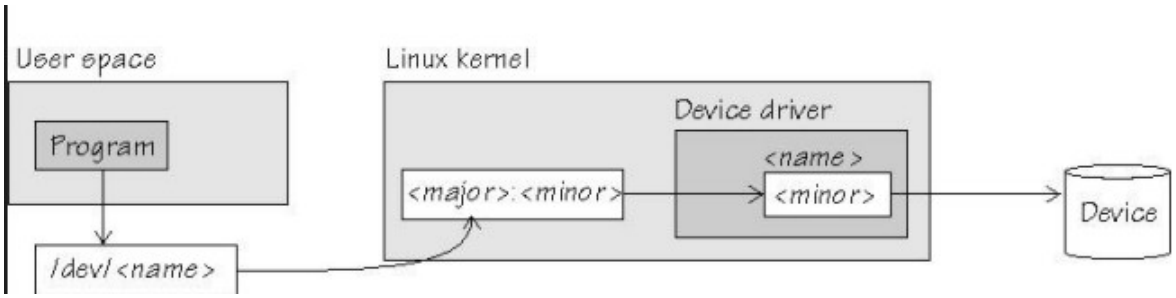


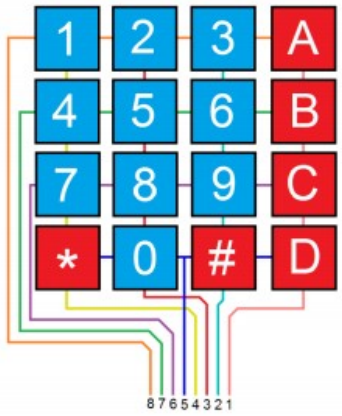
Рис. 13. Схема взаємодії простору користувача з драйвером пристрою в проторі ядра Linux

Для символьних пристроїв існує структура, що зберігає вказівники на можливі методи, які може використовувати користувач для взаємодії з пристроєм:

```
struct file_operations {
        struct module *owner;
                loff_t (*llseek) (struct file *, loff_t, int);
                ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
                ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
                ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
                ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
                int (*iterate) (struct file *, struct dir_context *);
                unsigned int (*poll) (struct file *, struct poll_table_struct *);
                long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
                long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
                int (*mmap) (struct file *, struct vm_area_struct *);
                int (*mremap)(struct file *, struct vm_area_struct *);
                int (*open) (struct inode *, struct file *);
                int (*flush) (struct file *, fl_owner_t id);
                int (*release) (struct inode *, struct file *);
                int (*fsync) (struct file *, loff_t, loff_t, int datasync);
                int (*fasync) (int, struct file *, int);
                ----
        };
```

● llseek: викликається, коли VFS потрібно перемістити індекс позиції файлу;
● read: викликається read (2) і пов'язаними системними викликами;
● read_iter: можливо асинхронне читання з iov_iter як призначення;
● write: викликається записом (2) і пов'язаними системними викликами;
● write_iter: можливо асинхронне записування з iov_iter як джерело;
● iterate: викликається, коли VFS потрібно прочитати вміст каталогу;
● poll: викликається VFS, коли процес хоче перевірити, чи є активність на;
цьому файлі, і (за бажанням) перейти в режим сну, поки не буде вказаної в драйвері діяльності. Викликається системними викликами select (2) і poll (2);
● unlocked_ioctl: викликається системним викликом ioctl (2);
● compat_ioctl: викликається системним викликом ioctl (2), коли 32-бітні системні виклики використовуються на 64-бітових ядрах;
● mmap: викликається системним викликом mmap (2);
● open: викликається VFS при відкритті inode;
● flush: викликається системним викликом close (2) для видалення файлу;
● release: викликається після закриття останньої посилання на відкритий файл;
● fsync: викликається системним викликом fsync (2);
● fasync: викликається системним викликом fcntl (2), коли для файлу включений асинхронний (неблокуючий) режим;

Більшість з цих операцій не реалізовано у вказаному прикладі, а тому ми позначаємо відповідний вказівник на метод макросом на зразок `no_llseek`



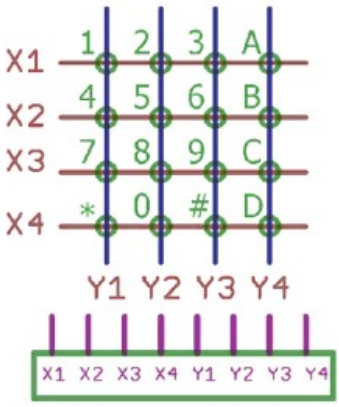


Рис. 14. Розглянемо приклад драйверу символьного пристрою для роботи матричної клавіатури, що представляє із себе матричну клавіатуру вигляду

Всього GPIO **8** ліній може бути використано для роботи клавіатури. Перш ніж почати працювати з клавіатурою, треба вибрати які саме лінії використати для підключення, в моєму випадку гарний варіант буде:

GPIO line	Pin name	BBB pin
gpio0_26	gpmc_ad10	P8.14
gpio0_27	gpmc_ad11	P8.17
gpio1_12	gpmc_ad12	P8.12
gpio1_13	gpmc_ad13	P8.11
gpio1_14	gpmc_ad14	P8.16
gpio1_15	gpmc_ad15	P8.15
gpio1_17	gpmc_a1	P9.23
gpio1_29	gpmc_csn0	P8.26

Таб. 2. GPIO-лінії для підключення

Ініціалізація клавіатури:
• виділити 4 лінії на сканування та 4 лінії на читання;
• Встановіть всі лінії сканування та читання на режим “вхід” (input);
• Налаштуйте час очікування на лініях читання (між 2 сигналами переривання на 1 лінії);

Опитування клавіатури (цей процес починається, коли натискається будь-яка клавіша і створюється переривання):
• Встановіть всі лінії сканування на вхід (Hi-Z);
• Встановіть одну лінію сканування в режим “виходу” (output);
• Повторіть для наступної лінії сканування;

Сканування (на переривання):
• Прочитати стан всіх ліній читання (відповідно до попереднього пункту, коли ми встановлюємо послідовно лінії сканування в output і зчитуємо їх значення стану, порівнюючи зі станом в попередньому циклі оброки переривання);
• Виявити, яка кнопка була натиснута (стан, порівняно з попередньою ітерацією – змінився);

Перш ніж приступити до реалізації, розглянемо нюанси ініціалізації драйвера:

У реальних драйверах ми рідко використовуємо тільки module_init() чи module_exit(), взагалом ці методи заміняють probe() и remove(), які в свою чергу мають власні особливості:

Виконавши insmod драйвера, викликається pci_register_driver(), яка дає ядру список пристроїв, які він може обслуговувати, а також покажчик на функцію probe() Після цього ядро викликає функцію probe() драйвера один раз для кожного пристрою, що описані під вказаний драйвер. Ця probe() функція запускає ініціалізацію пристрою: ініціалізацію апаратного забезпечення, виділення ресурсів і реєстрацію пристрою з ядром у вигляді символьного, блочного, мережевого або будь-якого іншого пристрою.

Це полегшує роботу драйверів, оскільки їм ніколи не потрібно шукати пристрої, які були підключені “гарячим” (hot-plug) способом. Саме ядро відповідає за обробку цієї частини і сповіщає потрібний драйвер, коли він отримує пристрій для обробки.

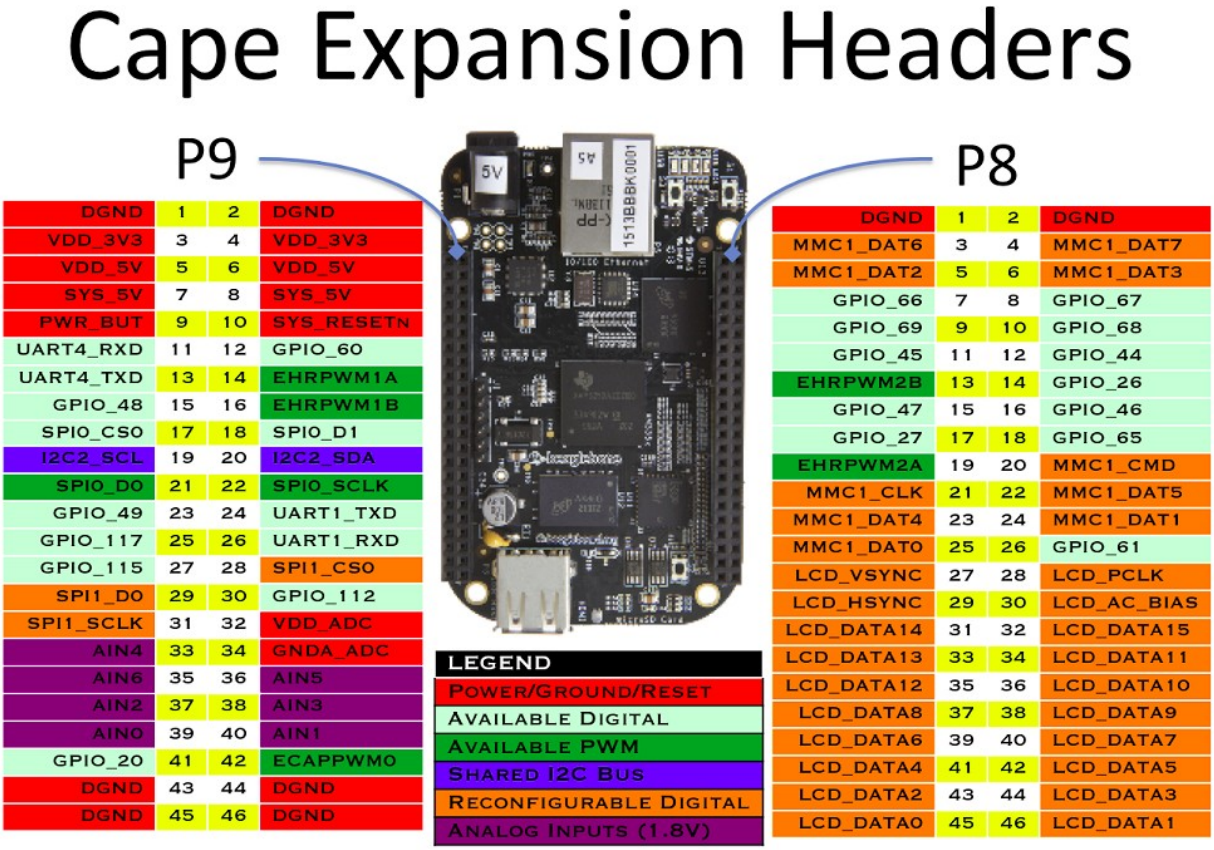
Нюанси коректної реалізації структури драйвера:

- Драйвер повинен бути незалежним від платформи. Це означає, що в коді драйвера, спираючись на конкретний приклад з матричною клавіатурою 4x4, не повинно бути зафіксовано значення номерів GPIO, по яким відбувається підключення та отримання інформації;
- Дані пристрою отримуються з дерева пристроїв;
- Прив'язка драйверів – це автоматичний процес асоціації пристрою з відповідним драйвером, що працює з цим пристроєм;

Інтерфейс введення/виведення загального призначення (англ. General-purpose input/output, GPIO) — інтерфейс для зв'язку між компонентами комп'ютерної системи, наприклад, мікропроцесором і різними периферійними пристроями. Контакти GPIO можуть діяти і як входи, і як виходи, і це, як правило, підлягає налаштуванню. GPIO контакти часто групуються в порти. **[26]**

GPIO контакти не мають спеціального призначення і зазвичай залишаються невикористаними. Ідея полягає в тому, що іноді системному інтегратору для побудови повної системи, яка використовує чип, може виявитися корисним мати кілька додаткових ліній цифрового управління. З них можна організувати додаткові схеми, які інакше довелося б створювати з нуля.

Адреса портів пристроїв, що використовують переривання GPIO, повідомляються центральному процесору (CPU), і лише тоді процесор зможе їх використовувати.



P9			P9
DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
UART4_RXD	11	12	GPIO_60
UART4_TXD	13	14	EHRPWM1A
GPIO_48	15	16	EHRPWM1B
SPI0_CS0	17	18	SPI0_D1
I2C2_SCL	19	20	I2C2_SDA
SPI0_D0	21	22	SPI0_SCLK
GPIO_49	23	24	UART1_TXD
GPIO_117	25	26	UART1_RXD
GPIO_115	27	28	SPI1_CS0
SPI1_D0	29	30	GPIO_112
SPI1_SCLK	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	ECAPPWM0
DGND	43	44	DGND
DGND	45	46	DGND

P8			P8
DGND	1	2	DGND
MMC1_DAT6	3	4	MMC1_DAT7
MMC1_DAT2	5	6	MMC1_DAT3
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
EHRPWM2B	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
EHRPWM2A	19	20	MMC1_CMD
MMC1_CLK	21	22	MMC1_DAT5
MMC1_DAT4	23	24	MMC1_DAT1
MMC1_DAT0	25	26	GPIO_61
LCD_VSYNC	27	28	LCD_PCLK
LCD_HSYNC	29	30	LCD_AC_BIAS
LCD_DATA14	31	32	LCD_DATA15
LCD_DATA13	33	34	LCD_DATA11
LCD_DATA12	35	36	LCD_DATA10
LCD_DATA8	37	38	LCD_DATA9
LCD_DATA6	39	40	LCD_DATA7
LCD_DATA4	41	42	LCD_DATA5
LCD_DATA2	43	44	LCD_DATA3
LCD_DATA0	45	46	LCD_DATA1

Рис. 15. Схема GPIO та інших можливих підключень(живлення, земля, I2C шина і т.д.) для BBB [17]

```
&am33xx_pinmux {
        hw3_pins: hw3_pins {
                pinctrl-single,pins = <
                        AM33XX_IOPAD(0x828, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x82c, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x830, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x834, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x838, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x83c, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x844, PIN_INPUT | MUX_MODE7)
                        AM33XX_IOPAD(0x87c, PIN_INPUT | MUX_MODE7)
                >;
        };
};

/ {
        matrix_keypad {
                compatible = "matrix_keypad";
                row-gpios = <&gpio0 26 GPIO_ACTIVE_LOW
                        &gpio0 27 GPIO_ACTIVE_LOW
                        &gpio1 12 GPIO_ACTIVE_LOW
                        &gpio1 13 GPIO_ACTIVE_LOW>;
                column-gpios = <&gpio1 14 GPIO_ACTIVE_LOW
                        &gpio1 15 GPIO_ACTIVE_LOW
                        &gpio1 17 GPIO_ACTIVE_LOW
                        &gpio1 29 GPIO_ACTIVE_LOW>;
                debounce-delay-ms = <5>;
                col-scan-delay-us = <400>;

                wakeup-source;
                pinctrl-names = "default";
                pinctrl-0 = <&hw3_pins>;
        };
};
```

Рис. 16. Вигляд допису додаткових вершин до DTS, в яких зберігається інформація про клавіатуру

Вершина **am33xx_pinmux** зберігає інформацію про те в якому режимі завантажувати під конкретним номером лінію (всього 8) **[17][18]**.

Вершина **matrix_keypad** зберігає інформацію про compatible-рядок, який використовує драйвер для пошуку інформації про пристрій. В row-gpios та column-gpios призначається один GPIO (від контролера GPIO з міткою gpio0 або мікроконтролера GPIO з міткою gpio1) (на яку посилається символ амперсанда &) і передається їм два значення 10 — номер лінії, і GPIO_ACTIVE_LOW -

вказує полярність GPIO (GPIO_ACTIVE_LOW - макрос препроцесора, визначений як 1). wakeup-source – вказує на те, що пристрій може працювати з менеджером енергії. Pinctrl - це частина апаратного забезпечення, зазвичай набір регістрів, які можуть керувати PIN-кодами. Він може мультиплексувати, зміщувати, встановлювати ємність навантаження, встановлювати міцність приводу і т.д. для окремих контактів або груп контактів. **[18]**

Набір функцій в модулі matrix_keypad_v1.c:

Основна структура, що зберігає інформацію про клавіатуру:

```
struct matrix_keypad {
/* для реєстрації misc пристрою, з відповідним major і minor номерами [20]. */
	struct miscdevice mdev;

/* для синхронізації обробки різних work-ів в workqueue, щоб не зіпсувати перевірку яка кнопка була нажата; */
	spinlock_t lock;

/*  відповідно черга для work-ів (нижче), що відповідає за зчитування стану клавіатури після кожного отриманого переривання. */
	struct workqueue_struct *work_queue;

/*  відповідно черга для work-ів (нижче), що відповідає за зчитування стану клавіатури після кожного отриманого переривання. */
	wait_queue_head_t wait;

/* структура work-ів, які використовуться в черзі, зберігають потрібну інформацію для роботи обробника workqueue; */
	struct delayed_work dwork;

/* вказівник на масив, що зберігає стан колонок (ліній сканування) */
	int *cols_state;

/* масив із 4 irqs для відповідно отримання переривань з рядок на зчитування для подальшого пошуку яка саме кнопка була натиснута */
	int *irq_rows;

/* час між перериваннями рядків на зчитування */
	char data_buffer[9];

/* GPIO API – using GPIO descriptors – новий інтерфейс для роботи з GPIO, що спрощує виділення памяті та работи з перериваннями вказаних ліній */
	struct gpio_desc **row_gpios;
	struct gpio_desc **col_gpios;

	u32 num_row_gpios; /* кількість рядків на зчитування */
	u32 num_col_gpios; /* кількість рядків на сканування */

/* час між перериваннями рядків на зчитування */
```

```
	u32 debounce_ms;

/* час між перериваннями рядків на зчитування */
	u8 count_of_data_ready;

/* час очікування між переведенням рядків на сканування з input в output режими */
	u32 col_scan_delay_us;

/* змінна, що потрібна для перевірки завершення процесу знаходження нажатої кнопки (поки операція пошуку кнопки не завершиться – інші works сплять) */
	bool scan_pending;

/* значення, залежне від режиму роботи електроспоживання (менеджер енергозбереження) */
	bool stopped;

/* значення, залежне від режиму роботи електроспоживання (менеджер енергозбереження) */
	bool wakeup;

/* значення, яке зберігає поточний статус буфера на зчитування */
	bool data_is_ready;
};
```

Основний функціонал:

- **Методи для взаємодії користувача з символьним пристроєм:**

```
/* Структура, що зберігає інформацію про файлові операції з символьним пристроєм (приклад операції: echo 123 > /dev/matrix_keypad_v1 – це операція write). Ця структура присвоюється в probe() для нашого MISC пристрою, з яким ми взаємодіємо на протязі роботи драйвера. */
static const struct file_operations keypad_fops = {
	.owner		= THIS_MODULE,
	.read		= keypad_read,
	.write		= keypad_write,
	.poll		= keypad_poll,
	.unlocked_ioctl	= keypad_ioctl,
	.llseek		= no_llseek,
};

/* Допомагає з файлу  /dev/matrix_keypad_v1 витягнути інфорацію про miscdevice для подальшого отримання структури matrix_keypad за допомогою макроса container_of [20] */
static inline struct matrix_keypad *to_matrix_keypad_struct(struct file *file);

/* коли користувач намагається отримати дані з /dev/matrix_keypad_v1, то викликається метод відповідний read, що повертає користувачеві масив символів. Основний метод: */
static ssize_t keypad_read(struct file *file, char __user *buf, size_t count, loff_t *ppos);
```

```
/* Копіює побайтово значення, отриманих по вказівнику from (дані передаються по стандартному потоку виведення даних) в вказівник to в кількості n байтів */
static __always_inline unsigned long __must_check
copy_to_user(void __user *to, const void *from, unsigned long n);

/* коли користувач намагається передати дані на /dev/matrix_keypad_v1, то викликається відповідний write-метод, який заносить передані користувачем дані в масив буфера символів і повертає користувачеві кількість символів, записаних в цей масив. Основний метод : */
static ssize_t keypad_write(struct file *file, const char __user *buf,
           size_t count, loff_t *ppos);

  /* Копіює побайтово значення, отриманих по вказівнику from (дані передаються по стандартному потоку введення) в вказівник to в кількості n байтів */
static __always_inline unsigned long __must_check
copy_from_user(void *to, const void __user *from, unsigned long n);

/* Користувач засинає доти, доки не будуть отримані нові дані з клавіатури */
static __poll_t keypad_poll(struct file *file, poll_table *wait);

/* Input-Output control, завдяки якому в kernel space можна передати певну конфігурацію з вказаним значенням. В моєму випадку обробляється 1 команда : kEYPAD_STATUS, яка залежно від того яке значення було передано (0 або 1) деактивує або активує GPIO лінії зчитування на переривання. */
static long keypad_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
```

- **Додатковий функціонал для обробки GPIO ліній:**

```
static void activate_col(const struct matrix_keypad *pdata, int col,
           bool on) {
     if (on) {
           /* якщо потрібно вичитати значення з рядку сканування – перехід лінії GPIO відповідного рядка в режим виводу */
           gpiod_direction_output(pdata->col_gpios[col], 0);
           /* очікування, перш ніж змінити режим роботи іншої лінії */
           udelay(pdata→col_scan_delay_us);
     } else {
           /* інакше -  перехід лінії GPIO відповідного рядка в режим вводу даних */
           gpiod_direction_input(pdata→col_gpios[col]);
     }
}

/* переведення режиму роботи всіх ліній сканування на введення/виведення */
```

```
static void activate_all_cols(const struct matrix_keypad pdata, bool on);

/* Методом gpiod_get_value_cansleep – отримання значення GPIO лінії на зчитування */
static bool row_asserted(const struct matrix_keypad *pdata, int row);

/* примітивна функція виводу нажатої кнопки в dmesg */
static void check_what_output(int x, int y);
```

- **Функції, що на пряму опрацьовують стан клавіатури при натискані будь-якої кнопки:**

```
static void matrix_keypad_scan(struct work_struct *work)
{
/* завдяки макросу container_of [21] ми можемо витягнути потрібну нам для обробки інформацію */
	struct matrix_keypad *keypad =
		container_of(work, struct matrix_keypad, dwork.work);
	int row, col;

	unsigned int i;
	int *new_cols_state =
		(int*)kzalloc(keypad→num_col_gpios *sizeof(int), GFP_KERNEL);

/* Дективуємо всі лінії на сканування – переводимо в режим input */
	activate_all_cols(keypad, 0);

	memset(new_cols_state, 0, sizeof(int) * keypad->num_col_gpios);

/* Активуючи по черзі лінії сканування, перевіряємо чи є зміни відповідно на лініях для читання. Якщо зміни є – зберігаємо це з врахуванням того яка саме лінія для читання змінила свій стан (побітовим зсувом) в масив new_cols_state */
	for (col = 0; col < keypad->num_col_gpios; col++) {
		/* Переводимо лінію сканування в режим output щоб вичитати стан */
		activate_col(keypad, col, 1);

		/* Після того як відновили роботу відповідної лінії сканування – зчитуємо стан ліній зчитування */
		for (row = 0; row < keypad->num_row_gpios; row++)
			new_cols_state[col] |=
				row_asserted(keypad, row) ? (1 << row) : 0;
		/* Переводимо лінію сканування в режим input */
		activate_col(keypad, col, 0);
	}

/* Порівнюємо новий стан клавіатури (а саме стан ліній сканування) з попереднім станом – якщо ж присутні зміни – виводимо яка клавіша була натиснута (у випадку, коли натискають одну й ту ж клавішу – переривання клавіатури працють в 2 режими – натискання та
```

```
відпускання клавіши, обидва режими змінюють стан відповідної лінії сканування (і завжди спочатку спрацьовує натискання, що, як приклад, зсуває 1 на відповідну позицію (номер лінії для читання), тоді як якщо відпустити клавішу – в лінії сканування туди ж задається 0, а тому ми можемо спокійно натискати одну й ту ж клавішу */
	for (col = 0; col < keypad->num_col_gpios; col++) {
	/* Перевірка зміни відповідної лінії сканування */
		u32 changing = new_cols_state[col] ^ keypad->cols_state[col];
		/* Якщо змін немає – продовжуємо з наступної ітерації циклу */
		if (changing == 0)
			continue;
		/* Якщо зміни є – перевіряємо на якій саме лінії зчитування відбулись зміни */
		for (row = 0; row < keypad->num_row_gpios; row++) {
			if ((changing & (1 << row)) == 0)
				continue;
			check_what_output(row, col);
		}
	}

/* Зберігаємо новий стан клавіатури */
	memcpy(keypad->cols_state, new_cols_state,
			sizeof(int) * keypad->num_col_gpios);

/* Після завершення перевірки на зміну стану – відновлюємо всі GPIO лінії сканування в режим input – вводу даних */
	activate_all_cols(keypad, 1);

/* Змінюємо статус обробки стану клавіатури, відновлюємо переривання для наступних ітерацій */
	spin_lock_irq(&keypad->lock);
	keypad->scan_pending = 0;

	for (i = 0; i < keypad->num_row_gpios; i++)
		enable_irq(keypad->irq_rows[i]);

	kfree(new_cols_state);

/* Звільняємо spinlock блокування переривань даючи можливість іншим worker-ам почати обробку нових пакетів work */
	spin_unlock_irq(&keypad->lock);
}

/* Функція-обробник, що викликається для обробки переривання, отриманого на 1 із 4 ліній для читання */
static irqreturn_t matrix_keypad_interrupt(int irq, void *id)
{
	struct matrix_keypad *keypad = id;
	unsigned long flags;
	unsigned int i;

/* Намагаємось захопити spinlock для перевірки чи знаходиться клавіатура в стані обробки або в режимі сну. Якщо так – то в данний момент не можливо виконати обробку (якщо ж ми
```

```
захопили спінлок блокування в момент блокування – отже виникла помилка з призупиненням роботи переривань ліній на читання, а отже в будь-якому випадку звільняємо спінлок, виходимо з функції */
      spin_lock_irqsave(&keypad->lock, flags);

      if (unlikely(keypad->scan_pending || keypad->stopped)) {
            spin_unlock_irqrestore(&keypad->lock, flags);
            return IRQ_HANDLED;
      }

/* Вимикаємо обробку переривань на лініях читання, завдяки чому можемо спокійно обробити дані стану клавіатури */
      for (i = 0; i < keypad->num_row_gpios; i++)
            disable_irq_nosync(keypad→irq_rows[i]);

/* Встановлюємо стан – “в обробці” */
      keypad->scan_pending = 1;

/* Передаємо в чергу завдань – workqueue, де у відповідному work ми перевіримо зміну стани клавіатури */
      queue_delayed_work(keypad->work_queue, &keypad->dwork,
                  msecs_to_jiffies(QUEUE_DELAY_MS));

/* Звільняємо спінлок блокування з відновленням налаштувань відповідних запитів переривань */
      spin_unlock_irqrestore(&keypad->lock, flags);
      return IRQ_HANDLED;
}
```

- **Функції, що відповідають за роботу драйвера у випадку коли менеджер споживання електроенергії переводить роботу ядра Linux з режиму очікування/сну (suspend) в режим повноцінної роботи(resume)**

```
/* Запуск драйвера, відновлення переривань на лініях, викликається в методі
 probe() та matrix_keypad_resume() */
static int matrix_keypad_start(struct platform_device *dev);

/* Повна зупинка драйвера, вимкнення переривань, викликається в методі remove() та matrix_keypad_suspend() */
static void matrix_keypad_stop(struct platform_device *dev);

/* Відновлення роботи ліній на переривання */
static void matrix_keypad_enable_wakeup(struct matrix_keypad *keypad);

/* Зупинка роботи ліній на переривання */
static void matrix_keypad_disable_wakeup(struct matrix_keypad *keypad);
```

```
/* Основна функція призупинення роботи драйвера, виклик
matrix_keypad_disable_wakeup() та matrix_keypad_stop() */
static int matrix_keypad_suspend(struct device *dev);

/* Основна функція відновлення роботи драйвера, виклик
matrix_keypad_enable_wakeup() та matrix_keypad_start() */
static int matrix_keypad_resume(struct device *dev);
```

- **Функція, що ініціалізує GPIO лінії, відповідно задаючи irq для відповідних ліній. Встановлення часу очікування між обробкою різних ліній зчитування, відключення переривань до моменту запуску драйверу:**

```
static int matrix_keypad_init_gpio(struct platform_device *pdev,
          struct matrix_keypad *keypad);
```

- **Функція, що відповідає за зчитування інформації про пристрій зі структури дерева пристроїв. В першу чергу це потрібно для ініціалізації роботи GPIO ліній:**

```
static struct matrix_keypad *matrix_keypad_parse_dt(struct device *dev);
```

- **Основні методи, що використуються для зчитування з dts:**

```
/* повертає кількість елементів в масиві propname у відповідній вершині дерева пристроїв
np. */
static inline int of_gpio_named_count(struct device_node *np, const
char* propname);

/* в out_value записує відповідне значення елемента propname у відповідній вершині
дерева пристроїв np. Повертає 0 – якщо зчитування успішне, інакше – помилку про
відсутність такого значення у вказаній вершині. */
static inline int of_property_read_u32(const struct device_node *np,
          const char *propname, u32 *out_value);

/* Повертає значення булівської змінної propname з відповідної вершини дерева пристроїв
np . */
static inline bool of_property_read_bool(const struct device_node *np,
          const char *propname);

/* Дозволяє зчитувати значення масива GPIOD-ліній propname по вказаному індексу index у
відповідній вершині дерева пристроїв node з відповідними ініціалізуючими параметрами
dflags. Label – додаткова інформація для додавання до запитаного GPIO. Структура dev -
```

пристрій -користувач GPIO лініями. Взагалом цип пристроєм являється **MISC** пристрій, якій ініціалізується в **matrix_keypad_probe** */

```
struct gpio_desc *devm_gpiod_get_from_of_node(struct device *dev,
          struct device_node *node, const char *propname, int index,
          enum gpiod_flags dflags, const char *label);
```

- **Функції ініціалізації та завершення модуля ядра Linux:**

/* Функція – ініціалізація самого драйвера, виділення пам'яті під всі поля структури matrix_keypad, виклик matrix_keypad_parse_dt(), matrix_keypad_init_gpio() та matrix_keypad_start() для подальшоъ роботи драйвера. */

```
static int matrix_keypad_probe(struct platform_device *pdev);
```

- **Основний метод виділення пам'яті, який взятий за основу для роботи вказаного драйвера матричної клавіатури:**

/* Відповідно функція звільнення пам'яті структури matrix_keypad (деініціалізація GPIO ліній, видалення workqueue і т.д.) */

```
static int matrix_keypad_remove(struct platform_device *dev);
```

- **Структури даних, які відповідають за ініціалізацію та деініціалізацію драйвера та пошук потрібної інформації в DTS:**

```
static const struct of_device_id matrix_keypad_of_match[] = {
      { .compatible = "matrix_keypad" },
      {}
};
MODULE_DEVICE_TABLE(of, matrix_keypad_of_match);

static struct platform_driver matrix_keypad_driver = {
      .probe = matrix_keypad_probe,
      .remove = matrix_keypad_remove,
      .driver = {
            .name = DRIVER_NAME,
            .pm = &matrix_keypad_pm,
            .of_match_table = matrix_keypad_of_match,
      }
};
```

3.3 I2C шина для передачі даних. Приклад роботи з нею, використання різних API

I²C - послідовна шина даних для зв'язку інтегральних схем, розроблених фірмою Philips на початку 1980-х як проста шина внутрішнього зв'язку для створення керуючої електроніки. Використовується для з'єднання

низькошвидкісних периферійних компонентів з материнською платою, вбудованими системами та мобільними телефонами. **[26][27]**

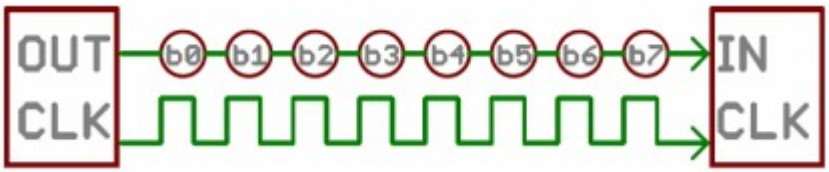


Рис. 17. Послідовна шина.

Плюси послідовних шин:
- Легше реалізувати HW;
- Може використовуватися на більш високих частотах;

Характеристики синхронної шини:
- Велика швидкість передачі даних завдяки синхронізації з годинником;
- Добре працює лише на коротких дротах;

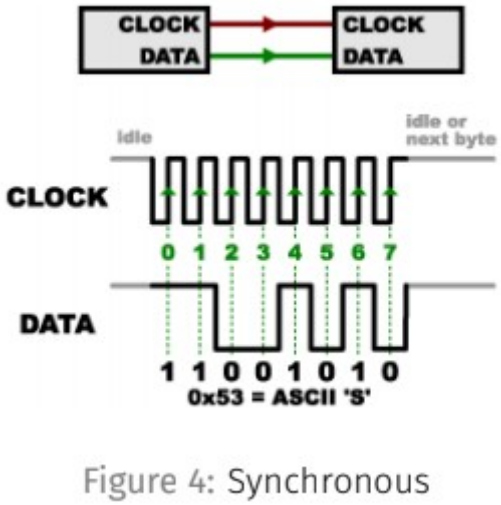


Figure 4: Synchronous

Рис. 18. I2C шина

Основні характеристики:
• Низькошвидкісна шина для підключення бортових і зовнішніх пристроїв до SoC;
• Тільки два дроти (годинник, дані);
• Дозволяє кілька пристроїв на одній шині (завдяки адресації);
• Пристрої мають адреси I2C;
• Схема роботи з комп’ютера з пристроями - Майстер / підлеглі;
• Майстер ініціює зв'язок і забезпечує синхронізуючий сигнал;
• Дозволена схема із багатьох майстрів;

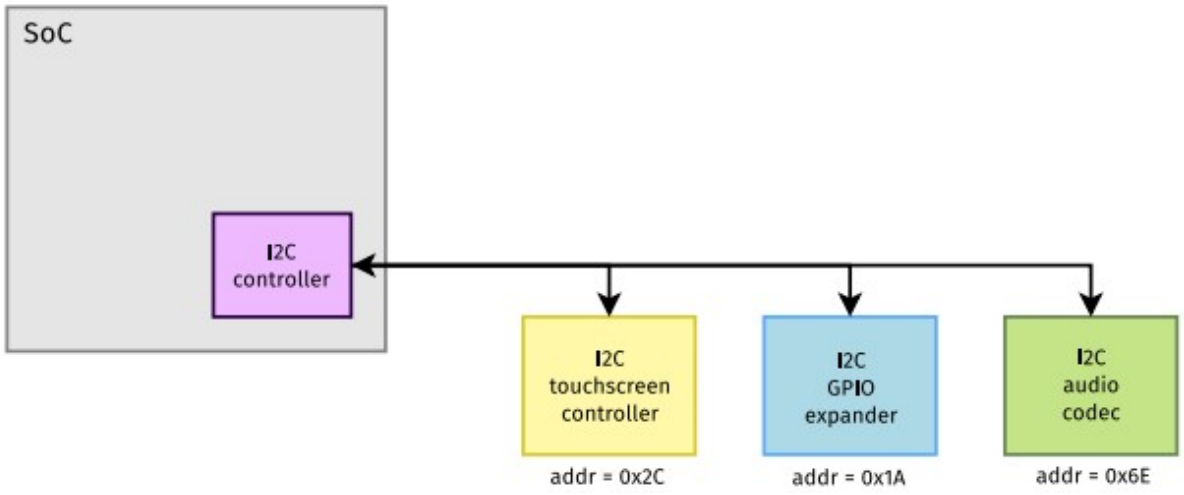


Рис. 19. Примітивна схема I2C шини (а саме адресація пристроїв на шині)

І перш ніж працювати з пристроями, розглянемо в якому форматі шина передає дані: **[22]**

Під час операції передачі даних в регістри пристроя-прислуги, ми передаємо масив значень, що складається з 3 відокремлених частин:

- адрес самого пристрою;
- адрес регістру, в який записують дані;
- масив із байтів, що буду записані у вказаний регістр;

Всі ці частини відокремлені за допомогою ACK — спеціального байту контролю підлеглого (SDA — означає лінію даних, SCL — лінія годинника).
R/W — означає режим, в якому будуть відбуватися операції з підлеглим.

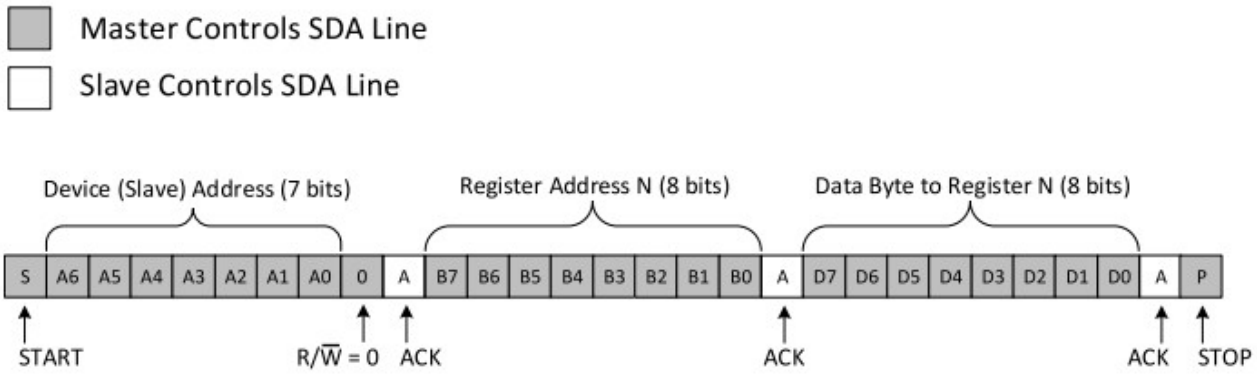


Рис. 20. Передача даних пристрою-прислуги.

Схожа ситуація зі зчитуванням даних з підлеглого пристрою:

- для початку — визначаємо пристрій, та адрес регістру, з якого будемо зчитувати, практично повторюючи дії запису в підлеглого;
- потім призупиняємо процес запису (Repeated START), вказавши повторно пристрій, проте R/W на зчитування, і вичитуємо 8 байтів із регістру.

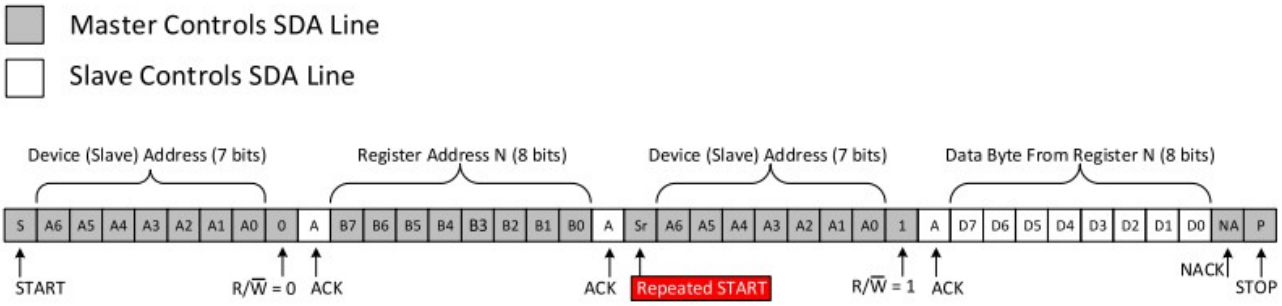


Рис. 21. Читання даних з пристрою-прислуги.

Основні API, що використовувались для реалізації зчитування даних з RTC модуля, підключеного до BBB 4 лініями GPIO (1 – земля, 2 – годинник, 3 – дані, 4 – напруга):

Проаналізувавши таблицю регістрів відповідного модуля RTC, що використовувався для роботи з I2C шиною:

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00h	CH	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03h	0	0	0	0	0	DAY			Day	01–07
04h	0	0	10 Date		Date				Date	01–31
05h	0	0	0	10 Month	Month				Month	01–12
06h	10 Year				Year				Year	00–99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08h–3Fh									RAM 56 x 8	00h–FFh

0 = Always reads back as 0.

Таб. 3. Регістри таймера DS1307

З специфікації DS1307:

• “Мережевий ланцюг скидається, коли записується регістр секунд. Передача запису відбувається на підтвердженні I2C з DS1307. Після скидання ланцюга дільника, щоб уникнути проблем із перекиданням, реєстрації часу і дати повинні бути записані протягом однієї секунди;

• Тому рекомендується записувати всі регістри (час / дата) в одній передачі, починаючи з запису регістру секунд”;

Створивши структуру для отримання даних з I2C підлеглого (2 типи повідомлень – 1) для встановлення адресу регістру та пристрою, 2) для зчитування регістру:

```
struct i2c_msg msg[2] = {
	{
		.addr = new_rtc->client->addr,
		.len = 1,
		.buf = &reg,
	},
	{
		.addr = new_rtc->client->addr,
		.flags = I2C_M_RD,
		.len = 1,
		.buf = &reg_status,
	}
};
```

Де відповідно addr – адрес підлеглого RTC модуля на шині I2C, reg – адрес регістру підлеглого RTC, len – довжина даних для зчитування, I2C_M_RD – режим зчитування (R/W = 1). Крім того в 2 повідомлені змінюється значення buf – тепер в нього записується посилання куди зчитувати дані з регістру:

```
i2c_transfer(new_rtc->client->adapter, msg, 2);
pr_info("### read data (i2c_transfer) = %#x!\n", reg_status);
```

a)`i2c_transfer` - посилає серію із 2 повідомлень

• Кожне повідомлення може бути читанням або записом, і їх можна будь-яким чином змішувати

• Транзакції можуть об'єднуватися - між транзакціями не передається стоп-біт

Структура i2c_msg містить для кожного повідомлення:

• адреса клієнта

• кількість байтів повідомлення

• і самі дані повідомлення

```
num_of_data = i2c_smbus_read_i2c_block_data(new_rtc->client, 0x00,
            7, new_rtc->buffer);
pr_info("### read data (i2c_smbus_read_byte_data)!\n");

for (i = 0; i < len; i++)
	pr_info("%#x\n", new_rtc→buffer[i]);
```

б)SMBus - шина керування системою:

• Протокол SMBus - підмножина протоколу I2C;

• Багато пристроїв використовують тільки одну підмножину;

• Якщо ви пишете драйвер для деякого пристрою I2C, спробуйте використати команди SMBus, якщо це можливо;

• Це дозволяє використовувати драйвер пристрою як на адаптерах SMBus, так і на адаптерах I2C;
• (набір команд SMBus автоматично переводиться на I2C на адаптерах I2C, але звичайні команди I2C взагалі не можуть оброблятися на більшості чистих адаптерів SMBus)

```
regs[DS1307_REG_SECS] = bin2bcd(t->tm_sec);
regs[DS1307_REG_MIN] = bin2bcd(t->tm_min);
regs[DS1307_REG_HOUR] = bin2bcd(t->tm_hour);
regs[DS1307_REG_WDAY] = bin2bcd(t->tm_wday + 1);
regs[DS1307_REG_MDAY] = bin2bcd(t->tm_mday);
regs[DS1307_REG_MONTH] = bin2bcd(t->tm_mon + 1);
regs[DS1307_REG_YEAR] = bin2bcd(t->tm_year - 100);

pr_info("%s: %7ph\n", "write", regs);

result = regmap_bulk_write(ds1307->regmap, REG_OFFSET, regs,
                           sizeof(regs));
```

в) regmap = Карта регістрів
• Реєстрація I/O потоків для I2C і SPI (аналогічно реалізації read/write операцій в символьному пристрої);
• Ми можемо використовувати його замість обговорюваного I2C API (вище);
• Може кешувати регістри (memory mapping method);
• Може управляти блокуванням шини;
• Може обробляти процес конверсації різних форматів передачі даних (big-endian to little-endian як приклад);
• BCD = binary-coded decimal - двійково-кодовані десяткові числа, тобто
Кожна з двох половинок кожного байта являє собою десяткову цифру у вигляді:

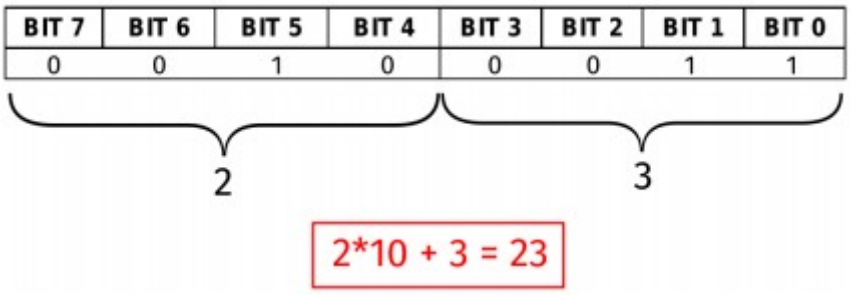


Рис. 21. Читання даних з пристрою-прислуги.

bcd2bin(n) = (((n) >> 4) * 10 + ((n) & 0x0F)).

3.4 Блокові пристрої.

Блокові спеціальні файли або блокові пристроїв забезпечують буферизований доступ до апаратних пристроїв і забезпечують деяку абстракцію від їх специфіки. **[25][26][27]** На відміну від пристроїв з символами, блокові пристрої завжди дозволять програмісту читати або записувати блок будь-якого розміру (включаючи окремі символи/байти) і будь-яке вирівнювання. Недоліком є те, що, оскільки блокові пристрої буферизовані, програміст не знає, скільки часу буде потрібно для того, щоб записані дані передавалися з буферів ядра до фактичного пристрою, або ж в якому порядку будуть надходити дві окремі операції запису на фізичний пристрій. Додатково, якщо одне і те ж апаратне забезпечення виставляє як символьні, так і блокові пристрої, існує ризик пошкодження даних через того, що клієнти використовують символьний пристрій, не знаючи про зміни, внесені в буфери блочного пристрою. **[23]**

Більшість систем створюють як блокові, так і символьні пристрої для представлення апаратних засобів, таких як, в першу чергу, жорсткі диски.

• Блок-драйвер надає доступ до пристроїв, які передають випадково доступні дані в блоках фіксованого розміру - насамперед, це дискові накопичувачі.
• Ядро Linux бачить, що блокові пристрої принципово відрізняються від символьних пристроїв; в результаті драйвери блоків мають чіткий інтерфейс.
• Виділена підсистема — block layer - відповідає за керування блоковими пристроями, разом з драйверами пристроїв.
• З простору користувача доступ до блокових пристроїв здійснюється за допомогою файлів пристроїв у /dev/.
• У більшості випадків вони зберігають файлові системи. До них не звертаються безпосередньо, а скоріше файлові системи монтуються, так що вміст файлової системи відображається в глобальній ієрархії файлів.
• Блок-пристрої також видно через файлову систему sysfs, в /sys/block/.

Приклад функціоналу для роботи з блоковими пристроями:

Прототипи функцій для роботи з блоковим пристроєм:

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

а) Перший крок — реєстрація блокового пристрою під відповідним major номером. В будь-якому випадку цей номер підбирається динамічно, або ж може бути вказаний статистично.

```
int register_blkdev(unsigned int major, const char *name);
```

Після реєстрації в каталозі /proc/devices зможе побачити новий пристрій.

При звільнені пристрою — обов’язково треба звільнити major номер.
void unregister_blkdev(unsigned int major, const char *name);

б) struct gendisk задає блоковий пристрій, прототип:
#include <linux/genhd.h>

• major - major номер пристрою;
• first_minor - minor номер пристрою. Потрібне в тому випадку, коли пристрій розбивають на декілька частин;
• minors — кількість номерів minor. У випадку нерозбитого на частини — 1;
• *fops — вказівник на структуру операцій з блоковим пристроєм;
• *queue — вказівник на чергу запитів, що генерує користувач;

```
struct gendisk {
        int major;
        int first_minor;
        int minors;
        /*...*/
        const struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        /*...*/
};
```

в) • Виділення пам’яті під структуру gendisk:
#define alloc_disk(minors) alloc_disk_node(minors, NUMA_NO_NODE)
• Виділення пам’яті під чергу запитів, де rfn – вказівник на функцію обробки запитів.
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);

• Для ініціалізації структури поля major, first_minor, fops, disk_name і queue повинні бути задані.

г) Задаємо пропускну здатність диску, а саме розмір сектору (традиційно 512 байт):

```
static inline void set_capacity(struct gendisk *disk, sector_t size);
```

• Додаємо диск до системи:

```
void add_disk(struct gendisk *disk);
```

• З цього моменту ми можемо отримати доступ до блокового пристрою. Проте пристрій повинен мати можливість обробляти I/O запити перш ніж викликати add_disk().

• I/O запити можуть мати місце навідь під час виклику add_disk().

д)• Видалення диску:

```
void del_gendisk(struct gendisk *gp);
```

• Звільнення черги:

```
void blk_cleanup_queue(struct request_queue *);
```

• Звільняємо пам’ять, що була виділена alloc_disk():

```
void put_disk(struct gendisk *disk);
```

е) struct block_device_operations задана в linux/blkdev.h

Це структура, що описує функціонал, який потрібен для роботи користувачеві з блоковим пристроєм.

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        void (*release) (struct gendisk *, fmode_t);
        /*...*/
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        unsigned int (*check_events) (struct gendisk *disk unsigned int clearing);
        /*...*/
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        /*...*/
};
```

Деякі основні функції:

• open () і release () - викликаються, коли до дескриптора пристрою, який обробляється драйвером, намагаються отримати доступ за допомогою open() і close();

• ioctl() - для конкретних операцій драйвера і compat_ioctl() для 32 бітних процесів, що виконуються на 64-бітному ядрі;
• check_events() і revalidate() - необхідні для обробки подій, наприклад, підтримка знімних носіїв (замість застарілих media_changed());

Getgeo() - надає інформацію про геометрію пристрою користувачам;

є) Приклад функції-обробника запиту:
Інформація про передачу доступна в структурі запиту.
• blk_rq_pos() - повертає позицію (сектор) в пристрої, на якому має здійснюватися передача.
• blk_rq_cur_sectors() - повертає кількість переданих байтів.
• bio_data() - повертає покажчик буфера: розташування в пам'яті, де дані повинні бути прочитані або записані.
• rq_data_dir() - тип передачі: READ або WRITE.
• __blk_end_request() або blk_end_request() - використовується для повідомлення про завершення запиту.
• __blk_end_request() - слід використовувати, коли блокування черги вже проведено.
Сама функція (викликається, коли в черзі присутні необроблені запити):

```
static void sbull_request(struct request_queue *q)
{
        struct request *req;
        blk_status_t error;

        req = blk_fetch_request(q);

        while (req) {
                struct sbull_dev *dev = req->rq_disk->private_data;
                error = BLK_STS_OK;

                if (blk_rq_is_passthrough(req)) {
                        pr_err("Skip non-fs request\n");
                        error = BLK_STS_IOERR;
                        goto done;
                }
```

```
            sbull_transfer(dev, blk_rq_pos(req), blk_rq_cur_sectors(req),
            bio_data(req->bio), rq_data_dir(req));

        done:
            if(!__blk_end_request_cur(req, error))
            req = blk_fetch_request(q);

        }
}
```

Обмеження простого запиту:

• sbull виконує запити синхронно, один раз передається один буфер;
• весь буфер передається одночасно;

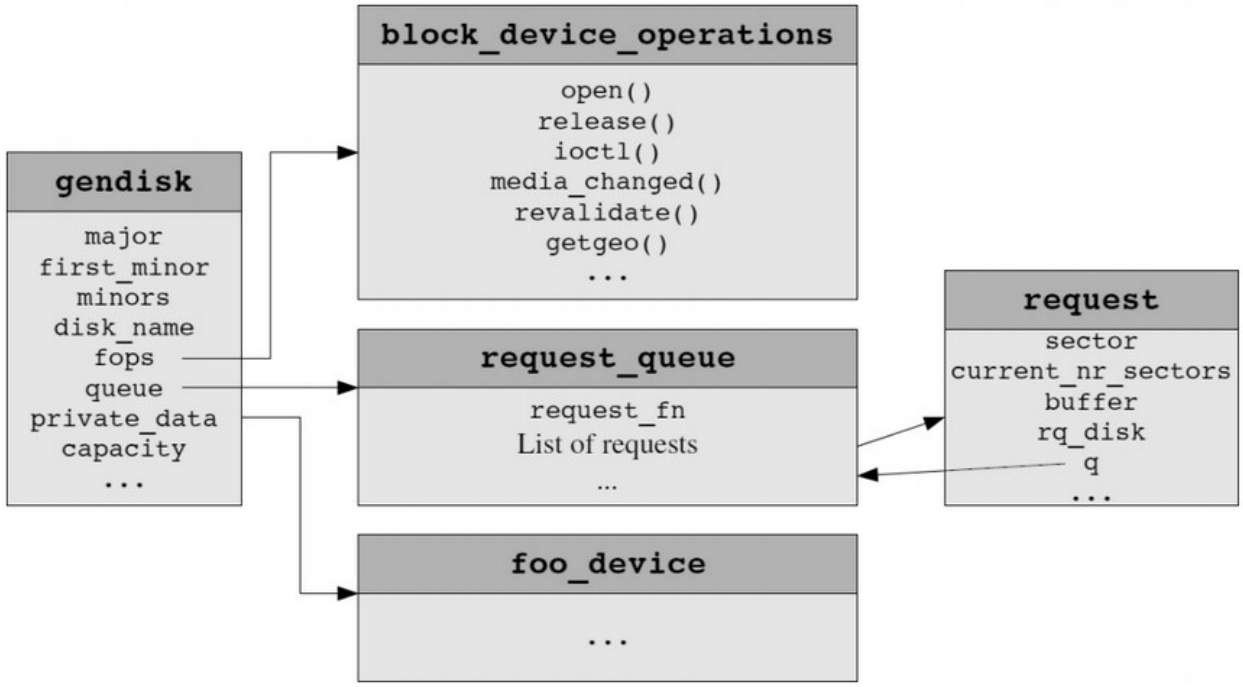


Рис. 22. Структура драйвера блокового пристрою.

ВИСНОВКИ

Так як метою роботи є створення програмного забезпечення для налаштування та реалізації взаємодії функціональних блоків комп’ютера між собою та операційною систему для можливості отримання інфромації з пристроїв користувачем, а тому як результат роботи - було розглянуто причини використання саме операційної системи Linux, її плюси та мінуси, реалізували приклади драйверів на основні символьних пристроїв, для яких існує величезна кількість способів взаємодії відповідного пристрою з комп’ютером (GPIO інтерфейс, I2C шина, та багато інших типів шин та підключень). Відповідно до кожного типу підключення або шини існують власні API для роботи з ними, деякі з них були розглянуті в розділі 3. Реалізовано драйвер роботи матричної клавіатури для BBB, на основі якого було розглянуто стандартні структури даних ядра Linux, методи синхронізації та методи відкладеної роботи, обробка переривань, функціонал для взаємодії простору користувача напряму з символьним пристроєм. Все це – лише мала частина можливостей реалізації драйверів та методів для роботи з величезним різновидом пристроїв, які кожен день люди створюють нові. А саме тому напрямок реалізації драйверів під операційну систему Linux буде продовжувати свій розвиток, покращуючи методи для роботи різних типів пристроїв, спрощуючи інтерфейси для роботи з різними типами підключень, створюючи нові інтерфейси – більш оптимальні. І разом з цим буде рости потреба в спеціалістах-розробниках. Завдяки цьому досить актуально розглядати цей напрямок програмування серед навчальних програм факультетів, спеціалізованих в комп’ютерних науках.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. різновиди архітектур процесорів [Електронний ресурс] :
https://tproger.ru/articles/processors-architectures-review/
2. Інформація про архітектуру ядра Linux [Електронний ресурс]:
https://www.engineersgarage.com/tutorials/introduction-linux-part-715
3. Програмні забезпечення GNU Toolchains [Електронний ресурс]:
https://elinux.org/Toolchains
4. Компіляція ядра Linux [Електронний ресурс] :
https://elinux.org/Building_BBB_Kernel
5. Інформація про makefile, приклади [Електронний ресурс] :
https://opensource.com/article/18/8/what-how-makefile
6. Зв'язні списки в ядрі Linux [Електронний ресурс] :
https://kernelnewbies.org/FAQ/LinkedLists
7. Хеш таблиці [Електронний ресурс] :
https://www.usenix.org/legacy/publications/library/proceedings/als00/2000papers/papers/2000/2000papers/papers/lever/lever_html/
8. Про таймери, затримки та приклади роботи таймерів в ядрі Linux [Електронний ресурс] :
www.cs.uni.edu/~diesburg/courses/dd/notes/lecture_8_time.ppt
9. Принцип роботи обробника переривань, верхня та нижня половини [Електронний ресурс] :
https://kerneltweaks.wordpress.com/2014/08/21/concept-of-isr-in-linux/
10. Tasklets и переривання в ядрі Linux [Електронний ресурс] :
https://habr.com/ru/company/embox/blog/244071/
11. Workqueue та нижня половина [Електронний ресурс] :
https://habr.com/ru/company/embox/blog/244155/
12. Реєстрація обробника переривань, API [Електронний ресурс] :
https://notes.shichao.io/lkd/ch7/#interrupt-handlers
13. Linux kernel synchronization - синхронізація процесів в ядрі Linux [Електронний ресурс] :
https://notes.shichao.io/lkd/ch9
14. Linux allocation memory (пам'ять в Linux) [Електронний ресурс] :
https://www.linuxjournal.com/article/6930
15. Device tree, інформація та синтаксиз, робота зі структурою дерева приладів [Електронний ресурс] :

https://events.static.linuxfound.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf
16. The platform device API [Електронний ресурс] :
https://lwn.net/Articles/448499
17. Інформація про роз'єм P8 P9 плати BeagleBone Black
[Електронний ресурс] :
http://credentiality2.blogspot.com/2015/09/beaglebone-pru-gpio-example.html
18. Pinctrl інформація [Електронний ресурс] :
https://01.org/linuxgraphics/gfx-docs/drm/driver-api/pinctl.html
19. Інформація про символьні пристрої [Електронний ресурс] :
https://linux-kernel-labs.github.io/master/labs/device_drivers.html
20. Miscellaneous Character Drivers - Змішані драйвери символів [Електронний ресурс] :
https://www.linuxjournal.com/article/2920
21. Макрос container_of в Linux, приклади використання [Електронний ресурс] :
https://www.stupid-projects.com/container_of/
22. I2C master-slave devices communication – методи спілкування пристроїв майстра з підлеглими [Електронний ресурс] :
http://www.ti.com/lit/an/slva704/slva704.pdf
23. Блочні пристрої в ядрі Linux [Електронний ресурс] :
https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch14s01.html
24. Офіційний репозиторій Лінуса Торвальдса для завантажування ядра Linux [Електронний ресурс]:
https://github.com/torvalds/linux
25. Andrew S. Tanenbaum, Herbert Bos. MODERN OPERATING SYSTEMS 4th Edition / Vrije Universiteit, Amsterdam, The Netherlands – К.:«PEARSON», 2015. – 1136 с.
26. John Madieu, Linux Device Drivers Development / BIRMINGHAM - MUMBAI – К.:«Packt Publishing», 2017. – 548 с.
27. Jonathan Corbet, AlessandroRubini, and Greg Kroah-Hartman. Linux device drivers, 3th edition / O'Reilly Media, Inc., 2005. - 630 с.
28. Приклади реалізації різних типів драйверів ядра Linux під архітектуру ARM [Електронний ресурс] : https://github.com/MykytaOpanyuk/Learn_Linux_Kernel