Міністерство освіти та науки України
Чорноморський національний університет імені Петра Могили

Я. М. Крайник

Embedded Systems

Методичні вказівки до виконання практичних робіт

Випуск 259

Миколаїв – 2018

УДК 004.3(076)
К 78

Рекомендовано до друку вченою радою ЧНУ ім. Петра Могили (витяг з протоколу № 3 від 8 листопада 2018 року).

***Рецензенти*:**
Журавська І. М. канд. техн. наук, доцент кафедри комп'ютерної інженерії Чорноморського національного університету ім. Петра Могили;
Пузирьов С. В. канд. фіз.-мат. наук, доцент кафедри комп'ютерної інженерії Чорноморського національного університету ім. Петра Могили.

К 78 Крайник Я. М. Embedded Systems : метод. вказівки до виконання практичних робіт. Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2018. 44 с. (Методична серія ; вип. 259).

Методичні рекомендації містять завдання для виконання практичних робіт з дисципліни «Embedded Systems». Наведений необхідний теоретичний матеріал, а також описаний процес практичного виконання з необхідними командами та прикладами програмної реалізації.

Методичні рекомендації призначені для студентів денної форми навчання, які навчаються за освітньо-кваліфікаційним рівнем «магістр» за спеціальністю 123 «Комп'ютерна інженерія».

УДК 004.3(076)

ISSN 1811–492X

ЗМІСТ

Вступ 4

Практична робота № 1. Збірка Linux засобами Yocto 6

Практична робота № 2. Побудова модуля ядра 9

Практична робота № 3. Дослідження вбудованої системи
на базі емулятора QEMU 15

Практична робота № 4. Компіляція та розробка
програмного забезпечення під вбудовані платформи 21

Практична робота № 5. Робота з одноплатним комп'ютером 25

Практична робота № 6. Програмування
на цільовій платформі 34

Практична робота № 7. Відлагодження засобами GDB 38

Перелік посилань 41

ВСТУП

Вбудовані системи (англ. Embedded Systems) представляють собою системи, які є складовою частиною іншої більшої системи з механічними та електронними компонентами. На них покладено виконання окремих функцій, які забезпечують коректну роботу всього комплексу. Такі функції можуть бути як відносно простимим, так і дуже складними з точки зору обчислень, програмної реалізації та ін. Особливо важливими є програмні компоненти вбудованих систем, оскільки саме вони являють собою реалізацію функцій і від них залежить ефективність, надійність, швидкодія систем.

Операційна система Linux завдяки багатьом своїм особливостям знайшла широке застосування у вбудованих системах. Фактично, вона є готовою програмною платформою, на базі якої організовується робота програм. Linux має велику кількість сервісів, що спрощують розробку. Розробникам для desktop-систем Linux простіше перейти на розробку вбудованих систем за наявності такої платформи. Більше того, певна частина програмного коду може бути перенесена на цільову платформу без модифікацій.

Тим не менш, є велика кількість особливостей, які необхідно враховувати при розробці вбудованих систем на базі Linux. Безпосередньо початок роботи потребує наявності образу ядра та супутніх файлів для того, щоб ініціалізувати апаратне забезпечення з урахуванням його характеристик та складу. Найчастіше образ ядра можна отримати самостійно шляхом збірки з вихідних файлів. Велику кількість деталей необхідно враховувати для коректної роботи апаратного забезпечення.

Загалом, варто відзначити, що переваги від використання Linux переважають над недоліками, тому для того, щоб ефективно користуватись можливостями цієї операційної системи при розробці вбудованих систем, необхідно розуміти особливості роботи ядра Linux та його компонентів, роботу програмного забезпечення на такій платформі, вміти виконувати побудови образу ядра та його налаштування під конкретне апаратне забезпечення, реалізовувати програми під конкретну платформу як з використанням засобів безпосередньо операційної системи, так і на базі сторонніх бібліотек.

За рахунок цього можна отримати перевагу у швидкодії, надійності та інших важливих параметрах функціонування вбудованої системи. Саме тому використання Linux у вбудованих системах є актуальним та важливим.

Практичні роботи мають комплексний характер та передбачають роботу як з апаратною, так і програмною складовою.

ПРАКТИЧНА РОБОТА № 1.
ЗБІРКА LINUX ЗАСОБАМИ YOCTO

Метою даної роботи є збірка Linux для подальшого запуску на емуляторі.

Yocto Project надає можливість автоматизувати процес збірки ОС Linux для запуску на вбудованих системах. За рахунок уніфікації процесу збірки, до якого входять завантаження необхідних компонентів, їх конфігурація, компіляція та підготовка до встановлення у тому чи іншому вигляді. Це досягається за рахунок того, що засоби побудови у складі Yocto отримують усе програмне забезпечення, яке пов'язане з тим чи іншим проектом. У результаті цього необхідно надати доволі значний дисковий простір для побудови однієї або декількох версій Linux для одного цільового пристрою. Для виконання цієї роботи рекомендований дисковий простір, який рекомендується виділити, становить 35–40 Гб під віртуальну машину. Приклади скріншотів виконані для версії ОС Ubuntu 16.04.5.

Встановлення засобів побудови. Для початку необхідно встановити клієнт git для того, щоб завантажити усі засоби побудови з репозиторіїв. Для цього у вікні терміналу вводимо команду

```
sudo apt-get install git
```

Після цього можна перейти до завантаження менеджера посилань та залежностей у Yocto:

```
git clone git://git.yoctoproject.org/poky poky
```

Для виконання роботи буде використовуватись гілка під назвою morty. Для того, щоб переключитись на вказану гілку, слід ввести команди:

```
cd poky
git checkout morty
```

Метою наступних команд є додавання списку залежностей для побудови ОС Linux для обраної платформи (Xilinx Zynq-7000).

```
git clone git://github.com/Xilinx/meta-xilinx
cd meta-xilinx
git checkout morty
cd ..
git clone https://github.com/openembedded/meta-
openembedded.git
```

```
cd meta-openembedded
git checkout morty
cd ..
```

Після цього слід провести налаштування цільової платформи у файлах конфігурації та встановити оточення для побудови командою

```
source oe-init-build-env
```

Редагування файлів налаштувань рекомендується виконувати за допомогою текстового редактора nano:

```
nano conf/local.conf
```

При вказанні цільової платформи замінити MACHINE ??= "qemux86" на MACHINE ??= "qemu-zynq7". Дана конфігурація задає побудову для запуску на емуляторі QEMU. Обов'язково зберегти зміни командою Ctrl+O та вийти (Ctrl+X).

У файлі налаштувань шарів (layers) додати раніше встановлені:

```
nano conf/bblayers.conf
```

та додати у секцію BBLAYERS:

```
"
...
/home/work/poky/meta-xilinx \
/home/work/poky/meta-openembedded/meta-oe \
"
```

Зберегти зміни та вийти.

Побудова ядра та запуск на емуляторі. Після проведення вказаних дій залишається лише одна операція для побудови ядра:

```
bitbake core-image-minimal
```

У деяких випадках може видаватись помилка, пов'язана з відсутністю певного програмного забезпечення, на роботі якого базується bitbake, тому можна одразу встановити залежності командою

```
sudo apt-get install gawk wget git-core diffstat
unzip texinfo gcc-multilib \
build-essential chrpath socat cpio python
python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping libsdl1.2-
dev xterm
```

При тестуванні достатньо було встановити лише пакети chrpath та texinfo. Після того, як команда `bitbake core-image-minimal` успішно запущена, будуть проведені дії з побудови ядра. Зазвичай такий процес займає декілька (3–5) годин. Скріншот вікна терміналу під час виконання збірки показаний на рисунку нижче (рис. 1.1).

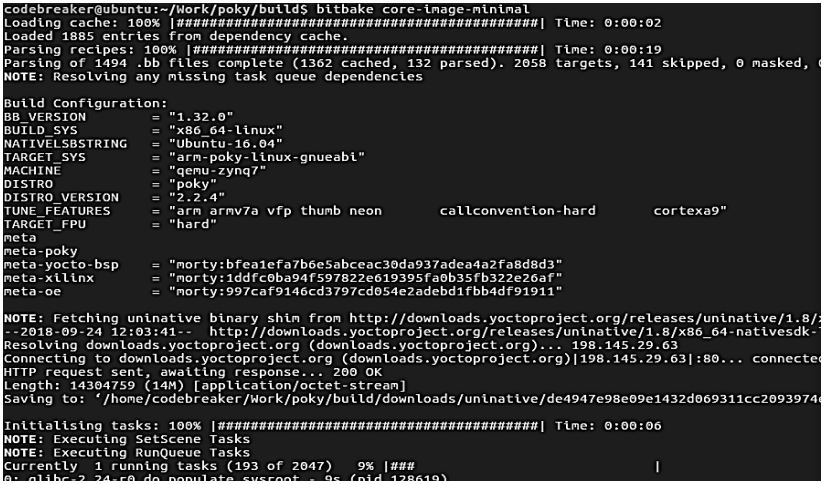

Рис. 1.1. Вигляд терміналу в процесі побудови ядра

Після завершення побудови можна запустити отриману версію Linux у емуляторі QEMU командою:

```
runqemu qemu-zynq7
```

Для здачі роботи продемонструвати скріншоти встановлення ПЗ, налаштування, збірки під час її виконання та результат після завершення та запуск на емуляторі.

ПРАКТИЧНА РОБОТА № 2.
ПОБУДОВА МОДУЛЯ ЯДРА

Метою роботи є побудова модуля ядра засобами Yocto та подальше його розгортання на цільовій системі (емуляторі).

Хід роботи

Програмування для ядра відрізняється від програмування прикладних застосувань. В основному внесення змін до його функціональності відбувається за рахунок реалізації *модулів*. Ядро Linux підтримує завантаження та вивантаження модулів, що є дуже зручним для розробки, оскільки не вимагає постійного перезавантаження системи.

Проект Yocto підтримує автоматичну збірку модулів шляхом додавання нових мета-шарів (*meta-layers*) для конкретної платформи. Для того, щоб створити новий шар необхідно скористатись скриптом, який знаходиться у директорії `scripts` у місці розташування *poky*. Запуск скрипту *yocto-layer* з параметром create дозволяє створити новий шар. Також необхідно вказувати назву нового шару при виклику скрипта. Виклик даної команди з подальшою перевіркою наявності директорії нового шару наведено на рис. 2.1.

```
codebreaker@ubuntu:~/Work/poky$ ./scripts/yocto-layer create hellomod
Please enter the layer priority you'd like to use for the layer: [default: 6]
Would you like to have an example recipe created? (y/n) [default: n]
Would you like to have an example bbappend file created? (y/n) [default: n]

New layer created in meta-hellomod.

Don't forget to add it to your BBLAYERS (for details see meta-hellomod/README).
codebreaker@ubuntu:~/Work/poky$ ls
bitbake         meta-hellomod      meta-xilinx               README
build           meta-openembedded  meta-yocto                README.hardware
documentation   meta-poky          meta-yocto-bsp            scripts
LICENSE         meta-selftest      oe-init-build-env         yocto-layer.log
meta            meta-skeleton      oe-init-build-env-memres
codebreaker@ubuntu:~/Work/poky$
```

Рис. 2.1. Створення нового шару

Новий шар містить шаблонну структуру директорій, яка необхідна для роботи з ним інструмента побудови *bitbake.* У директорії *poky* також наявний шаблон для шару, який розміщений у директорії *meta-skeleton*. Його зручно використовувати в якості основи для побудови власного модуля. Для цього слід скопіювати вміст даної папки у

директорію нового шару. Шаблон містить інструкції для побудови модуля ядра, а також додаткові інструкції, що розміщені у директорії linux. Останні слід видалити, оскільки в цьому випадку нас цікавить лише побудова модуля. Послідовність виконання команд у терміналі командного вікна наведена на рис. 2.2 (зверніть увагу на директорію, з якої виконуються команди).

```
codebreaker@ubuntu:~/Work/poky/meta-hellomod$ cp -rp ../meta-skeleton/recipes-ke
rnel/ .
codebreaker@ubuntu:~/Work/poky/meta-hellomod$ cd recipes-kernel/
codebreaker@ubuntu:~/Work/poky/meta-hellomod/recipes-kernel$ ls
hello-mod  linux
codebreaker@ubuntu:~/Work/poky/meta-hellomod/recipes-kernel$ rm -rf linux/
codebreaker@ubuntu:~/Work/poky/meta-hellomod/recipes-kernel$
```

Рис. 2.2. Формування структури нового шару

Модулі ядра реалізуються на мові С. У скопійованій директорії *hello-mod* можна ознайомитись з тим, як виглядає найпростіший модуль ядра. Варто звернути увагу на те, що підключаються заголовкові файли, які розміщені в директорії *linux*. Ці версії файлів адаптовані спеціально для роботи в режимі ядра та надають мінімально необхідний для цього функціонал. Заголовкові файли стандартної бібліотеки не мають підключатись у вихідних файлах модулів. Також варто звернути увагу на модифікатори `__init` та `__exit`. Вони дозволяють вказати, що ці функції придатні для виклику при завантаженні модуля та його вивантаженні. У прикладі використовується інший варіант найменування, який одразу реалізує функції з необхідними іменами (рис. 2.3).

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

int init_module(void)
{
        printk("Hello World!\n");
        return 0;
}

void cleanup_module(void)
{
        printk("Goodbye Cruel World!\n");
}

MODULE_LICENSE("GPL");
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

Рис. 2.3. Файл реалізації модуля

Окремо слід розглянути *make-файл* для побудови модуля, оскільки його зміст (рис. 2.4) відрізняється від традиційних *make-файлів*. Директива *obj-m* вказує, на основі якого двійкового файлу має бути побудований модуль (модулі мають розширення .ko – від kernel object).

У файлі визначається, де виконувати пошук файлу для компіляції, та визначено три задачі, які можна виконати: побудова, встановлення та очистка. Під час виконання побудови керування процесом передається спеціальній підсистемі, пов'язаній з побудовою модулів ядра. Для отримання вихідного модуля, даний файл використовується інструментом *bitbake* хоча його можна виконати безпосередньо з директорії.

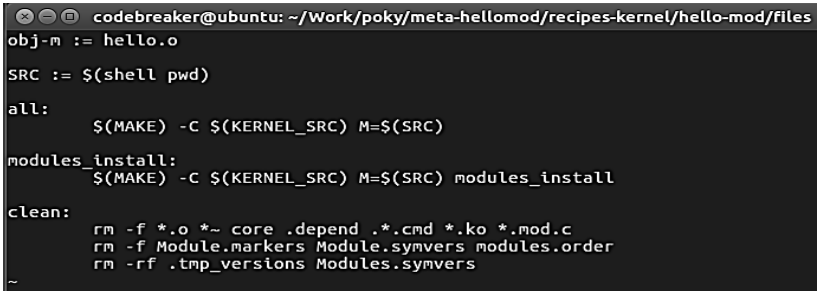

Рис. 2.4. Зміст make-файлу

Для того, щоб виконати побудову з використанням *poky*, слід повернутись у головну директорію та налаштувати робоче оточення командою

```
source oe-init-build-env
```

Наступним кроком є додавання у файл налаштувань conf/bblayers.conf нового шару (рис. 2.5).

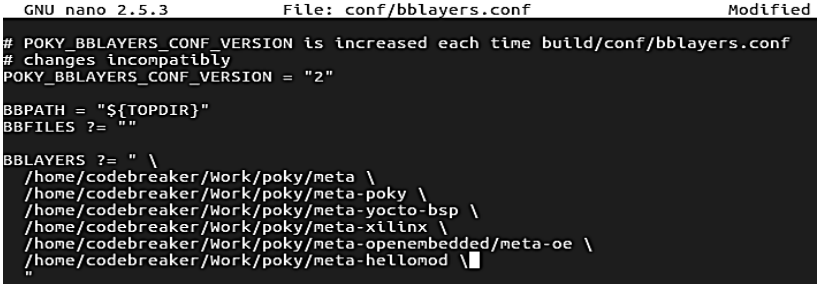

Рис. 2.5. Додавання нового шару у файл конфігурацій

Після цього потрібно запустити виконання побудови командою

```
bitbake hello-mod
```

Зверніть увагу на наявність дефісу в назві модуля. Результуючий вивід роботи даної команди показаний на рис. 2.6.

```
codebreaker@ubuntu:~/Work/poky/build$ bitbake hello-mod
Loading cache: 100% |########################################| Time: 0:00:00
Loaded 1290 entries from dependency cache.
Parsing recipes: 100% |######################################| Time: 0:00:26
Parsing of 1495 .bb files complete (994 cached, 501 parsed). 2059 targets, 141 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION        = "1.32.0"
BUILD_SYS         = "x86_64-linux"
NATIVELSBSTRING   = "universal"
TARGET_SYS        = "arm-poky-linux-gnueabi"
MACHINE           = "qemu-zynq7"
DISTRO            = "poky"
DISTRO_VERSION    = "2.2.4"
TUNE_FEATURES     = "arm armv7a vfp thumb neon       callconvention-hard       c
ortexa9"
TARGET_FPU        = "hard"
meta
meta-poky
meta-yocto-bsp    = "morty:bfea1efa7b6e5abceac30da937adea4a2fa8d8d3"
meta-xilinx       = "morty:1ddfc0ba94f597822e619395fa0b35fb322e26af"
meta-oe           = "morty:997caf9146cd3797cd054e2adebd1fbb4df91911"
meta-hellomod     = "morty:bfea1efa7b6e5abceac30da937adea4a2fa8d8d3"

Initialising tasks: 100% |###################################| Time: 0:00:00
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 481 tasks of which 467 didn't need to be rerun an
d all succeeded.
codebreaker@ubuntu:~/Work/poky/build$
```

Рис. 2.6. Лог команди побудови модуля

На відміну від процесу побудови ядра, для побудови модуля необхідно значно менше часу. Загалом, весь процес займе не більше кількох хвилин. Таким чином, буде отримано двійковий файл модуля з розширенням .ko.

Проте, даний модуль наразі не є інтегрований у ядро і його необхідно перенести на цільову платформу. Для цього рекомендовано відкрити нове термінальне вікно та запустити виконання віртуальної машини з попередньої роботи (команда `runqemu qemu-zynq7`). Запущена віртуальна та поточна машина, на якій відбувалась побудова, об'єднуються у логічну мережу за допомогою мережевого інтерфейсу. Для них мають проходити запити команди `ping`. Адреса системи, що працює на емуляторі, зазвичай, встановлюється як 192.168.7.2, а сервер отримує адресу 192.168.7.1. Передати файл з однієї системи в іншу можна з використанням протоколу *tftp*.

Сервером у цьому випадку виступатиме *host-система,* а клієнтом – вбудована система, яка емулюється. Для того, щоб запустити TFTP-сервер, необхідно встановити пакет *tftpd-hpa* та розпочати роботу сервера (рис. 2.7).

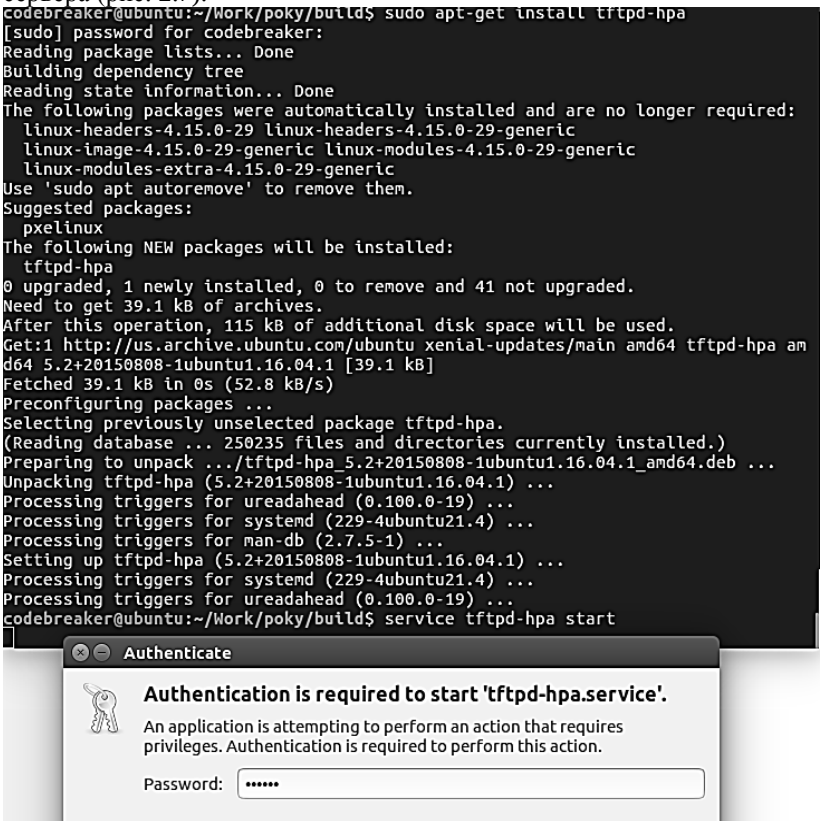

Рис. 2.7. Встановлення та запуск TFTP-сервера

Для систем, які відрізняються від *Ubuntu*, команда запуску може відрізнятись.

TFTP-клієнт може скопіювати дані з папки, яка пов'язана з TFTP-сервером. У цьому випадку вона розміщена у `/var/lib/tftfpboot/`. Скомпільований модуль знаходиться у директорії, яка показана на рис. 2.8. Його необхідно скопіювати у вищезазначену директорію сервера.

```
codebreaker@ubuntu:~/Work/poky/build$ sudo cp /home/codebreaker/Work/poky/build/
tmp/sysroots/qemu-zynq7/lib/modules/4.6.0-xilinx-v2016.3/extra/hello.ko /var/lib
/tftpboot/
[sudo] password for codebreaker:
codebreaker@ubuntu:~/Work/poky/build$ ls /var/lib/tftpboot/
hello.ko
```

Рис. 2.8. Переміщення модуля у директорію TFTP-сервера

Як вже зазначалось, емулятор рекомендується запустити у новому термінальному вікні (рис. 2.9).

```
codebreaker@ubuntu: ~/Work/poky/build
codebreaker@ubuntu:~$ cd Work/poky/
codebreaker@ubuntu:~/Work/poky$ source oe-init-build-env

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
codebreaker@ubuntu:~/Work/poky/build$ runqemu qemu-zynq7
```

Рис. 2.9. Запуск емулятора

Отримати файл з директорії сервера та запустити, після цього вивантажити модуль ядра можна за допомогою послідовності команд, яка вказана на рис. 2.10.

```
root@qemu-zynq7:~# tftp -g -r hello.ko 192.168.7.1
root@qemu-zynq7:~# ls -l
total 4
-rw-r--r-- 1 root root 2928 Oct  5 13:47 hello.ko
root@qemu-zynq7:~# insmod hello.ko
[  217.696788] Hello World!
root@qemu-zynq7:~# rmmod hello.ko
[  230.881779] Goodbye Cruel World!
root@qemu-zynq7:~#
```

Рис. 2.10. Копіювання модуля, запуск та зупинка роботи

Команда `insmod` дозволяє запустити модуль ядра. Запуск модуля відбувається одразу після завершення виконання команди. Команда `rmmod` зупиняє роботу модуля, викликаючи функцію завершення його роботи (`cleanup_module`). Переглянути список завантажених у ядро модулів можна командою `lsmod`.

Для здачі роботи у звіті представити скріншоти відповідних операцій.

ПРАКТИЧНА РОБОТА № 3. ДОСЛІДЖЕННЯ ВБУДОВАНОЇ СИСТЕМИ НА БАЗІ ЕМУЛЯТОРА QEMU

Мета роботи – ознайомлення з головними особливостями вбудованої системи при роботі з віртуальною системою для платформи *Xilinx Zynq-7* на базі емулятора *QEMU*, а також з вихідними файлами, які використовуються для побудови у системі *Yocto.*

Наступні завдання потребують роботи з графічним менеджером файлів (наприклад, Nautilus в Ubuntu), а також можуть потребувати виконання мінімального набору команд у термінальному вікні. Усі кінцеві двійкові файли, які були отримані у результаті побудови ядра для емулятора можна знайти у директорії <path_to_poky>/poky/build/tmp/deploy/images/<target_machine_name> (рис. 3.1). У ній, окрім самих файлів знаходяться також посилання на них з більш короткими іменами (наприклад, uImage).

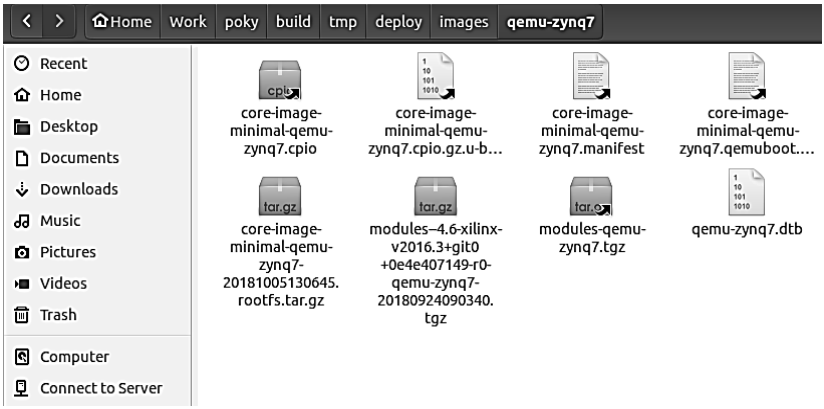

Рис. 3.1. Вміст директорії
з результуючими файлами після побудови

Побудова завантажувача для вбудованої системи. Завантажувачем операційної системи виконується робота щодо ініціалізації найбільш базової периферії на найбільш низькому рівні для запуску подальших процесів у операційній системі. Порівняно з самою операційною

системою, завантажувач потребує набагато менше ресурсів для запуску. Після того, як основні апаратні ресурси проініціалізовані, керування розгортанням програмних засобів передається ядру системи. У якості завантажувача найчастіше використовується програмне забезпечення *U-Boot.* У випадку емулятора дане програмне забезпечення не потрібне, проте, при побудові для апаратної платформи такий файл є необхідним.

Дерево пристроїв у вбудованій системі. Дерево пристроїв надає програмному забезпеченню інформацію про те, які апаратні ресурси наявні у системі, чи потрібно їх ініціалізовувати, налаштування тактової частоти, номери асоційованих переривань, а також деяку іншу інформацію, яка може бути специфічною для конкретного модуля. Дерево пристроїв у початковому вигляді описується у текстових файлах (мають розширення .dts або .dtsi – від англ. device tree source). Один файл опису може включати інший (.dtsi), що є зручним для платформ, які мають спільну частину. Інформація з даних файлів компілюється окремим компілятором – *Device Tree Compiler – dtc.* У результаті отримується двійковий файл (розширення .dtb), який і використовується під час завантаження.

Складові дерева пристрою системи, що використовується у цьому курсі, мають розташовуватись у директорії <path_to_poky>/poky/build/tmp/work/qemu-zynq7-poky-linux-gnueabi/device-tree/1.0-r0/devicetree.

Виконати пошук опису пристрою за варіантом (табл. 1) та навести скріншот опису відповідного пристрою у файлі. Більшість пристроїв системи описані у файлі zynq7-base.dtsi.

Таблиця 1.

Варіанти для завдання пошуку модулів у sysfs

№ варіанта	**Значення <dev_name>**
1	gpio
2	i2c
3	spi
4	timer
5	sdhci
6	ethernet
7	serial
8	adc
9	watchdog
10	interrupt-controller

Побудова ядра для вбудованої системи. Побудова ядра системи з вихідних файлів виконується командою

```
make arch=<target_arch>
```

з зазначенням цільової архітектури. Система побудови Yocto приховує цей момент від користувача, оскільки призначена для того, щоб зберегти час на налаштування системи. У випадку, якщо необхідно внести зміни у результуючі налаштування, необхідно їх змінити перед виконанням автоматизованої побудови. Для цього слід перейти у директорію, де розміщуються вихідні файли ядра та внести зміну у конфігурацію. Цільова директорія має бути розміщена у папці <path_to_poky>/poky/build/tmp/work-shared/qemu-zynq7/kernel-sources.
Якщо з даної директорії у терміналі запустити команду *`make menuconfig`*, то термінал має перейти у редактор налаштувань образу ядра (для роботи редактора необхідна бібліотека libncurses5-dev, тому, можливо, її необхідно буде встановити командою `sudo apt-get install libncurses5-dev`). Інтерфейс користувача редактора показано на рис. 3.2.

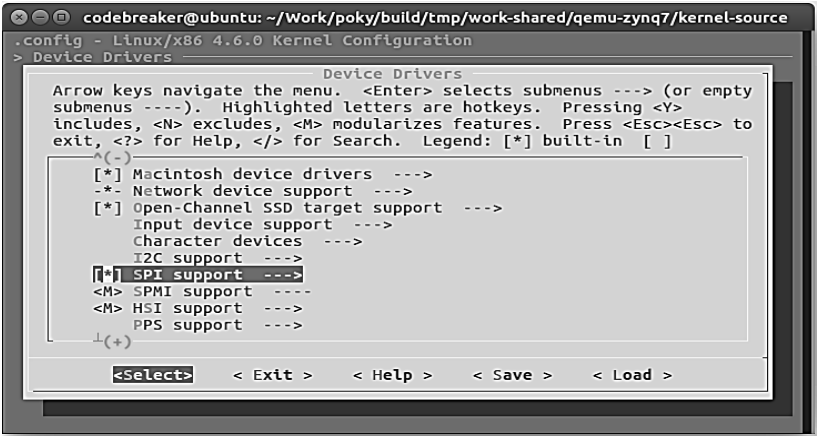

Рис. 3.2. Консоль налаштувань ядра

Переміщення між налаштуваннями здійснюється стрілками на клавіатурі. Встановлення опції відбувається при натисненні клавіші “*”. Змінивши конфігурацію її можна зберегти командою Save. *У звіті продемонструвати запущене вікно редактора конфігурацій*.

Модулі вбудованої системи. Зазвичай, модулі у ОС Linux розташовані у директорії /lib/modules/<version>/. Проте, з метою зменшення використання дискового простору у вбудованій системі така директорія є відсутньою.

Дослідження файлової системи вбудованої системи. Для виконання цього етапу слід запустити отриманий образ віртуальної машини командою

```
runqemu qemu-zynq7
```

Далі потрібно виконати вхід з ім'ям користувача *root* та переглянути вміст кореневої директорії системи командою:

```
ls /
```

Порівняти результат виконання цієї команди з аналогічним результатом для вказаної команди та системи, у якій запущений емулятор. *Зробити висновок відносно цього порівняння з представленням скріншотів*.

За допомогою команди `df` отримати інформацію про доступний для використання дисковий простір. *Зробити порівняння з основною системою з представленням відповідних скріншотів*.

Зазвичай, основні команди, які доступні з інтерфейсу терміналу розташовуються у директоріях /bin та /usr/bin. *Зробити порівняння з основною системою відносно кількості доступних команд, з представленням відповідних скріншотів*.

Встановлення програм на систему, зазвичай, відбувається з репозиторіїв з використанням менеджера (apt-get, rpm, opkg). *Перевірити наявність цих менеджерів та зробити висновок щодо встановлення програм на систему*.

Після цього перейти до директорії віртуальної файлової системи sysfs командою (слід звернути увагу на назву директорії)

```
cd /sys
```

У більшості випадків вона монтується автоматично, проте, якщо з певних причин вона недоступна після запуску, її необхідно змонтувати командою:

```
mount -t sysfs sysfs /sys
```

Переглянути вміст даної директорії та переконатись у наявності директорій для опису пристроїв та модулів:

```
ls
```

За допомогою команди:

```
find . -name <dev_name>
```

виконати пошук за варіантом (відповідно до табл. 1) директорій, які відносяться до конкретного модуля.

Навести результат виконання цієї команди відповідно до варіанту.

Файлова система *sysfs* надає не лише можливість отримати інформацію про зовнішні пристрої, але і є найпростішим інтерфейсом для керування ними безпосередньо з командного рядка.

Проведемо перевірку роботи з пристроями через *sysfs* на прикладі GPIO. Для цього перейдемо у директорію /sys/class/gpio та переглянемо вміст даної директорії (рис. 3.3).

```
root@qemu-zynq7:/sys/class/gpio# ls -l
total 0
--w------- 1 root root 4096 Jan  1  1970 export
lrwxrwxrwx 1 root root    0 Jan  1  1970 gpiochip906 -> ../../devices/soc0/amba/
e000a000.gpio/gpio/gpiochip906
--w------- 1 root root 4096 Jan  1  1970 unexport
```

Рис. 3.3. Представлення контролера GPIO у файловій системі sysfs

У даній директорії міститься посилання на директорію з назвою *gpiochip906*. Цифра 906 у кінці найменування вказує, з якого порядкового номера є доступними для використання піни. Їхня загальна кількість залежить від пристрою та вказана у документації. Також у директорії наявні два файли, *export* та *unexport*, які доступні лише для запису. Їхнє призначення полягає у наданні доступу до інтерфейсу GPIO за допомогою sysfs. Для прикладу розглянемо, як відбувається доступ до другого піну в інтерфейсі. Для цього слід виконати операцію експорту (рис. 3.4).

```
root@qemu-zynq7:/sys/class/gpio# echo 907 > /sys/class/gpio/export
root@qemu-zynq7:/sys/class/gpio# ls
export  gpio907  gpiochip906  unexport
```

Рис. 3.4. Експорт піну

У результаті має з’явитись нова директорія, яка призначена для роботи з піном. Слід звернути увагу на процедуру виконання експорту: у відповідний файл записується номер піну, який необхідно експортувати (команда `echo`), після чого формується директорія з відповідною назвою (*gpio907*). Проведемо більш детальне ознайомлення з вмістом нової директорії (рис. 3.5).

```
root@qemu-zynq7:/sys/class/gpio# ls -lL gpio907
total 0
-rw-r--r-- 1 root root 4096 Oct  5 14:16 active_low
drwxr-xr-x 4 root root    0 Jan  1  1970 device
-rw-r--r-- 1 root root 4096 Oct  5 14:16 direction
-rw-r--r-- 1 root root 4096 Oct  5 14:16 edge
drwxr-xr-x 2 root root    0 Oct  5 14:16 power
drwxr-xr-x 2 root root    0 Jan  1  1970 subsystem
-rw-r--r-- 1 root root 4096 Oct  5 14:11 uevent
-rw-r--r-- 1 root root 4096 Oct  5 14:16 value
```

Рис. 3.5. Вміст директорії для керування експортованим піном

П’ять файлів з доступом на читання та запис (лише для користувача root) дозволяють отримати поточну інформацію про налаштування піну (рис. 3.6).

```
root@qemu-zynq7:/sys/class/gpio/gpio907# cat edge
none
root@qemu-zynq7:/sys/class/gpio/gpio907# cat value
0
root@qemu-zynq7:/sys/class/gpio/gpio907# cat direction
in
root@qemu-zynq7:/sys/class/gpio/gpio907# cat active_low
0
```

Рис. 3.6. Значення за замовчанням для налаштувань піну

Як зрозуміло з попереднього рисунку, пін працює на вхід, має поточний логічний рівень 0 та не реагує на появу фронтів або спадів на вході. Параметр *active_low* більш актуальний, коли пін налаштований як вихід.

Для того, щоб переналаштувати пін на вихід достатньо змінити вміст файлу *direction*. Після цього ним можна керувати шляхом запису значення до файлу *value*. Проте, через обмеження емулятора ця опція не працює і не змінює значення на протилежне (рис. 3.7). Для безпосередньо апаратної платформи такий підхід має працювати.

```
root@qemu-zynq7:/sys/class/gpio/gpio907# echo out > direction
root@qemu-zynq7:/sys/class/gpio/gpio907# cat direction
out
root@qemu-zynq7:/sys/class/gpio/gpio907# echo 1 > value
root@qemu-zynq7:/sys/class/gpio/gpio907# cat value
0
```

Рис. 3.7. Зміна налаштувань експортованого піну

Єдиним варіантом змінити значення у емуляторі є зміна налаштування параметра *active_low* (рис. 3.8).

```
root@qemu-zynq7:/sys/class/gpio/gpio907# echo 1 > active_low
root@qemu-zynq7:/sys/class/gpio/gpio907# cat active_low
1
root@qemu-zynq7:/sys/class/gpio/gpio907# cat value
1
root@qemu-zynq7:/sys/class/gpio/gpio907# echo 1 > value
root@qemu-zynq7:/sys/class/gpio/gpio907# echo value
value
root@qemu-zynq7:/sys/class/gpio/gpio907# cat value
1
```

Рис. 3.8. Зміна вихідного значення піну

Після цього значення сигналу переходить у високий логічний рівень, однак, змінити його записом значення у *value* не вдається.

Навести скріншоти виконання усіх наведених команд на емуляторі.

ПРАКТИЧНА РОБОТА № 4. КОМПІЛЯЦІЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПІД ВБУДОВАНІ ПЛАТФОРМИ

Мета роботи – здобути початкові навички роботи з компіляторами для вбудованих систем та процесом розробки вбудованих систем.

Після того, як ядро операційної системи та супутні файли зібрані та запущені на цільовій платформі, постає питання розробки програмного забезпечення для неї. Основна особливість цього процесу полягає у тому, що найчастіше, щоб запустити програму для цільової платформи, слід розроблювати її на іншій платформі з більшими можливостями для розробки (середовище, потужніший процесор, більше оперативної пам'яті та ін.). Цільова система через обмеженість ресурсів позбавлена таких можливостей. Відповідно, розробка має вестись на одній машині, а запуск відбуватись на іншій. Причому, характерним також є те, що дві системи не є сумісними між собою, тому програма, написана для однієї системи, не буде виконуватись на іншій. Такий варіант є бажаним та більш зручним для розробки, проте, можливий також і інший варіант.

Одним з варіантів розробки для такого випадку є використання засобів побудови безпосередньо на цільовій платформі. Такий варіант підтримується також Yocto. Для цього у налаштуваннях властивостей цільового образу слід зазначити опції, які вказані на рис. 4.1.

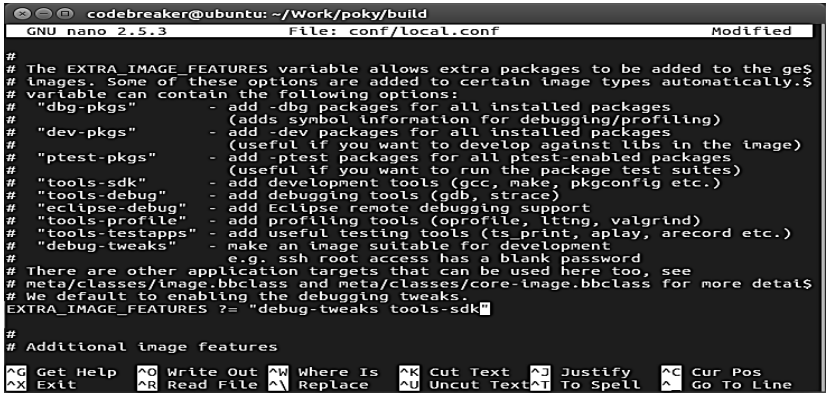

Рис. 4.1. Розділ налаштувань опцій відлагодження та розробки образу

До опції за замовчанням *debug_tweaks* додана опція *tools-sdk*, яка дозволяє встановити засоби розробки безпосередньо на цільову систему. Зміна цієї опції потребує повторної побудови системи. Сам процес у такому випадку має зайняти менше часу, оскільки більшість модулів вже готові, необхідно лише додати модулі для розробки (компілятор, *make* та деякі інші залежності).

Для того, щоб перевірити, що отриманий образ підтримує функції, пов'язані з розробкою, реалізуємо найпростішу програму, яка виводить повідомлення на дисплей. Текст програми показаний на рис. 4.2.

```
#include <stdio.h>

int main()
{
        printf("Hello from Zynq7\n");
        return 0;
}
```

Рис. 4.2. Код програми для компіляції

Перевірити працездатність програми можна стандартним способом: виконавши компіляцію (рис. 4.3), та запустивши отриманий файл на виконання.

```
root@qemu-zynq7:~# gcc -o hello hello.c
root@qemu-zynq7:~# ./hello
Hello from Zynq7
root@qemu-zynq7:~#
```

Рис. 4.3. Компіляція програми

Як видно з попереднього рисунку, компіляція та запуск нічим не відрізняється від цих процесів на звичайній машині. Компіляція забезпечується засобами, які встановлені у директорії */usr/bin*. Якщо переглянути вміст даної директорії, то можна побачити компілятори та інші засоби (рис. 4.4). Вони мають префікс `arm-poky-linux-gnueabi-`. Стандартні команди (наприклад, gcc) є посиланням на відповідний файл у даній директорії.

З теоретичної точки зору слід відмітити, що процес створення засобів розробки під конкретну платформу отримав назву *Canadian Cross*. Традиційно, вона передбачає наявність трьох машин:

1. Машини, на якій отримуються засоби побудови.
2. Машини, на якій отримані засоби побудови розгортаються (*SDK machine*).
3. Цільова машина (*target machine*).

При цьому, усі розглянуті машини можуть мати різну платформу. Наприклад, в якості першої машини використовується машина з операційною системою Windows на платформі IA32. Отримані засоби SDK працюють на машині під управлінням системи Linux на платформі x86, а результуючий вихідний файл розгортається на системі Android на платформі ARM. Загалом, типовою є ситуація, коли перші дві стадії виконуються на одній машині, щоб спростити розробку. Також, зазвичай, розробка ведеться саме під операційною системою Linux.

```
root@qemu-zynq7:~# ls /usr/bin/
[                                 g++                 printenv
[[                                gcc                 printenv.coreutils
aclocal                           gencat              printf
aclocal-1.15                      getconf             printf.coreutils
addr2line                         getent              prlimit
ar                                gettextize          ptx
arch                              gnu-configize       ptx.coreutils
arch.coreutils                    gprof               quilt
arm-poky-linux-gnueabi-addr2line  groups              ranlib
arm-poky-linux-gnueabi-ar         groups.coreutils    readelf
arm-poky-linux-gnueabi-as         head                readlink
arm-poky-linux-gnueabi-c++filt    head.coreutils      readlink.coreutils
arm-poky-linux-gnueabi-cpp        hexdump             realpath
arm-poky-linux-gnueabi-dwp        hexdump.util-linux  realpath.coreutils
arm-poky-linux-gnueabi-elfedit    hostid              recode-sr-latin
arm-poky-linux-gnueabi-g++        hostid.coreutils    rename
arm-poky-linux-gnueabi-gcc        iconv               renice
arm-poky-linux-gnueabi-gcc-ar     id                  renice.util-linux
arm-poky-linux-gnueabi-gcc-nm     id.coreutils        reset
arm-poky-linux-gnueabi-gcc-ranlib ifnames             resize
arm-poky-linux-gnueabi-gprof      install             rev
arm-poky-linux-gnueabi-ld         install.coreutils   rpcgen
arm-poky-linux-gnueabi-ld.bfd     intltool-extract    runcon
arm-poky-linux-gnueabi-ld.gold    intltool-merge      runcon.coreutils
arm-poky-linux-gnueabi-nm         intltool-prepare    script
arm-poky-linux-gnueabi-objcopy    intltool-update     scriptreplay
arm-poky-linux-gnueabi-objdump    intltoolize         sdiff
arm-poky-linux-gnueabi-pkg-config ionice              seq
arm-poky-linux-gnueabi-ranlib     ipcmk               seq.coreutils
arm-poky-linux-gnueabi-readelf    ipcrm               setarch
arm-poky-linux-gnueabi-size       ipcs                setsid
arm-poky-linux-gnueabi-strings    isosize             setsid.util-linux
arm-poky-linux-gnueabi-strip      join                setterm
as                                join.coreutils      sha1sum
```

Рис. 4.4. Засоби для кросс-компіляції

Завдання

Реалізувати програму з використанням стандартних засобів та бібліотек.

1. Програма, яка показує вміст директорії, вказаної в якості параметра.

2. Програма, яка визначає 10 процесів з найбільшим показником використання пам’яті.

3. Програма, яка показує вміст директорії, переданої в якості параметра (рекурсивно обходить кожну вкладену піддиректорію).

4. Програма для керування процесом створення файлів у заданій директорії. У файл записується інформація про момент його створення.

ПРАКТИЧНА РОБОТА № 5.
РОБОТА З ОДНОПЛАТНИМ КОМП'ЮТЕРОМ

Мета роботи – дослідження апаратних та програмних засобів одноплатного комп'ютера з точки зору використання апаратного забезпечення.

Nano Pi – серія одноплатних комп'ютерів від виробника FriendlyARM. Зазвичай, у них використовуються процесори Allwinner, які представляють собою систему на кристалі (у мікросхемі доступне також графічне ядро). Найменшим комп'ютером з даної серії є Nano Pi Neo Air. Розмір такого комп'ютера становить усього 4x4 см. У ньому відсутні роз'єми для підключення стандартних дисплеїв (HDMI-інтерфейс), тому працювати з ним можна лише через термінал послідовного порту або через SSH. Бездротове з'єднання забезпечується за допомогою Wi-Fi. Комп'ютер може працювати як в режимі точки доступу, так і в режимі станції (клієнта).

Для виконання робіт цього розділу рекомендується використовувати програму MobaXTerm (рис. 5.1), яка, окрім власне з'єднання по SSH, дозволяє виконати велику кількість інших корисних функцій, наприклад, одразу дозволяє обмінюватись файлами з цільової системою за допомогою протоколу SFTP.

Рис. 5.1. Термінал програми MobaXTerm\
після підключення до плати

Перш за все, потрібно переглянути вміст директорії /dev. Окрім стандартних пристроїв звертає на себе увагу наявність модулів SPI (spidev0.0) та I2C (i2c-0) – рис. 5.2.

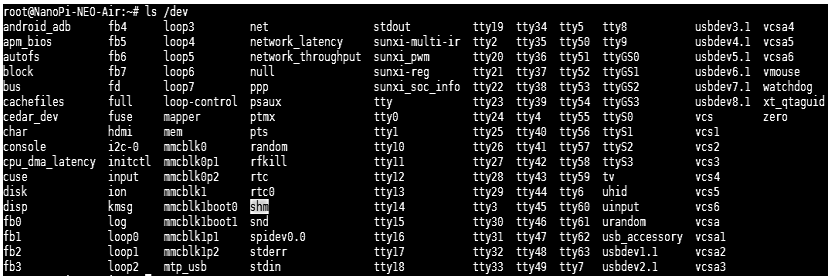

Рис. 5.2. Вміст директорії /dev

Вони є доступними завдяки тому, що були обрані як активні та доступні у налаштуваннях за замовчанням для образу операційної системи. Для того, щоб змінити ці налаштування слід викликати відповідну конфігураційну утиліту. У випадку Nano Pi така утиліта має назву npi-config (рис. 5.3). Залежно від версії системи ця можливість може бути відключена, тоді усі модулі повинні працювати.

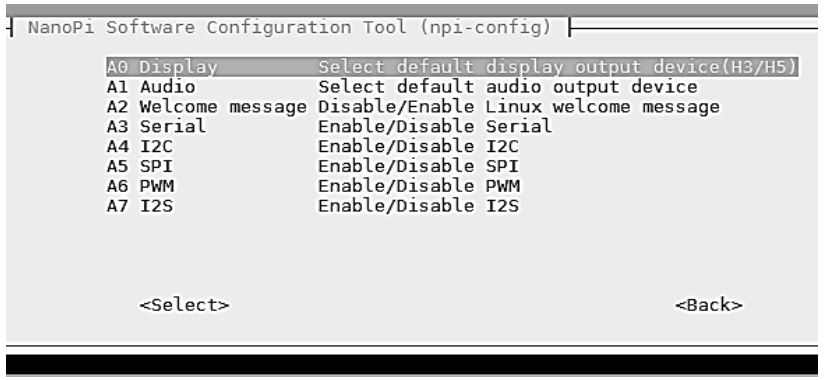

Рис. 5.3. Налаштування інтерфейсів у npi-config

Далі переглянемо вміст директорії /sys (рис. 5.4). У ній можна побачити стандартний для даного розташування набір директорій.

Рис. 5.4. Вміст директорії /sys

Дослідимо інфраструктуру, на якій базується робота з GPIO (рис 5.5).

```
root@NanoPi-NEO-Air:~# find . -name gpio
./WiringNP/gpio
./WiringNP/gpio/gpio
root@NanoPi-NEO-Air:~# cd /sys/
root@NanoPi-NEO-Air:/sys# find . -name "*gpio*"
./bus/platform/devices/leds-gpio
./bus/platform/drivers/leds-gpio
./bus/platform/drivers/leds-gpio/leds-gpio
./bus/platform/drivers/gpio-rc-recv
./devices/platform/leds-gpio
./devices/platform/sunxi-pinctrl/gpio
./devices/platform/sunxi-pinctrl/gpio/gpiochip0
./devices/platform/sunxi-pinctrl/gpio/gpiochip0/ngpio
./class/gpio
./class/gpio/gpiochip0
./kernel/debug/tracing/events/gpio
./kernel/debug/tracing/events/gpio/gpio_direction
./kernel/debug/tracing/events/gpio/gpio_value
./kernel/debug/gpio
./kernel/debug/pinctrl/sunxi-pinctrl/gpio-ranges
./module/bcmdhd/parameters/dhd_oob_gpio_num
```

Рис. 5.5. Результат пошуку пристроїв для управління GPIO

Основне, на що необхідно звернути увагу, – це наявність одного чіпу для управління інтерфейсом GPIO – `gpiochip0`. Далі необхідно продемонструвати роботу з GPIO засобами sysfs. Перейдемо у директорію, де розміщений верхній рівень опису контролера GPIO (рис. 5.6).

```
root@NanoPi-NEO-Air:/sys/class/gpio# ls
export  gpiochip0  unexport
```

Рис. 5.6. Вміст директорії для класу gpio

Як видно з рисунку, доступні операції експорту, які використовувались при роботі з образом операційної системи на QEMU. Більше деталей щодо інтерфейсу GPIO платформи можна отримати з файлів, які розміщені безпосередньо у директорії контролера (рис. 5.7).

```
root@NanoPi-NEO-Air:/sys/class/gpio/gpiochip0# cat ngpio
384
```

```
root@NanoPi-NEO-Air:/sys/class/gpio/gpiochip0# cat base
0
```

Рис. 5.7. Кількість пінів та початковий номер пінів на платі

Відповідно, контролер GPIO керує 384 пінами, а базова адреса для доступу до них починається з 0. Звичайно, не всі піни виведені для користування; доступна лише невелика їх частина. Для прикладу налаштуємо пін GPIOA0 (див. рис. з назвою пінів далі). Він матиме номер 0 з точки зору контролера. Переглянемо його поточний стан командою gpio readall (більш детально про дану команду – далі) – рис. 5.8.

```
root@NanoPi-NEO-Air:/sys/class/gpio# gpio readall
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+---+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 |     |     |     3.3V |      |   |  1 || 2  |   |      | 5V       |     |     |
 |  12 |   8 |  GPIOA12 | ALT5 | 0 |  3 || 4  |   |      | 5V       |     |     |
 |  11 |   9 |  GPIOA11 | ALT5 | 0 |  5 || 6  |   |      | 0v       |     |     |
 | 203 |   7 |  GPIOG11 |  OUT | 1 |  7 || 8  | 0 | ALT5 | GPIOG6   | 15  | 198 |
 |     |     |       0v |      |   |  9 || 10 | 0 | ALT5 | GPIOG7   | 16  | 199 |
 |   0 |   0 |  GPIOA0  | ALT5 | 0 | 11 || 12 | 0 | OFF  | GPIOA6   | 1   | 6   |
 |   2 |   2 |  GPIOA2  | ALT5 | 0 | 13 || 14 |   |      | 0v       |     |     |
 |   3 |   3 |  GPIOA3  | ALT5 | 0 | 15 || 16 | 0 | OFF  | GPIOG8   | 4   | 200 |
 |     |     |     3.3v |      |   | 17 || 18 | 0 | OFF  | GPIOG9   | 5   | 201 |
 |  64 |  12 |  GPIOC0  | ALT4 | 0 | 19 || 20 |   |      | 0v       |     |     |
 |  65 |  13 |  GPIOC1  | ALT4 | 0 | 21 || 22 | 0 | ALT5 | GPIOA1   | 6   | 1   |
 |  66 |  14 |  GPIOC2  | ALT4 | 0 | 23 || 24 | 0 | ALT4 | GPIOC3   | 10  | 67  |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+------+----------+-----+-----+
```

Рис. 5.8. Визначення стану піну

Пін перебуває у логічному стані 0 і налаштований як альтернативна функція (ALT5). Відповідну інформацію можна отримати за допомогою sysfs. Розглянемо даний процес, а також процес налаштування піну більш детально. Експортуємо пін командою

```
echo 0 > export
```

При цьому необхідно, щоб поточна директорія була /sys/class/gpio. Пересвідчимось, що інтерфейс керування тепер є доступним (рис. 5.9).

```
root@NanoPi-NEO-Air:/sys/class/gpio# ls
export  gpio0  gpiochip0  unexport
```

Рис. 5.9. Вміст директорії після експорту

Проведемо налаштування засобами sysfs (рис. 5.11).

```
root@NanoPi-NEO-Air:/sys/class/gpio# cd gpio0
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# ls
active_low  device  direction  edge  power  subsystem  uevent  value
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# cat value
0
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# cat direction
in
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# echo out > direction
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# cat direction
out
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# echo 1 > value
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# cat value
1
root@NanoPi-NEO-Air:/sys/class/gpio/gpio0# gpio readall
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 |     |     |     3.3V |      |   |  1 || 2  |   |      | 5V       |     |     |
 |  12 |   8 |  GPIOA12 | ALT5 | 0 |  3 || 4  |   |      | 5V       |     |     |
 |  11 |   9 |  GPIOA11 | ALT5 | 0 |  5 || 6  |   |      | 0v       |     |     |
 | 203 |   7 |  GPIOG11 |  OUT | 1 |  7 || 8  | 0 | ALT5 | GPIOG6   | 15  | 198 |
 |     |     |       0v |      |   |  9 || 10 | 0 | ALT5 | GPIOG7   | 16  | 199 |
 |   0 |   0 |  GPIOA0  |  OUT | 1 | 11 || 12 | 0 | OFF  | GPIOA6   | 1   | 6   |
 |   2 |   2 |  GPIOA2  | ALT5 | 0 | 13 || 14 |   |      | 0v       |     |     |
 |   3 |   3 |  GPIOA3  | ALT5 | 0 | 15 || 16 | 0 | OFF  | GPIOG8   | 4   | 200 |
 |     |     |    3.3v  |      |   | 17 || 18 | 0 | OFF  | GPIOG9   | 5   | 201 |
 |  64 |  12 |  GPIOC0  | ALT4 | 0 | 19 || 20 |   |      | 0v       |     |     |
 |  65 |  13 |  GPIOC1  | ALT4 | 0 | 21 || 22 | 0 | ALT5 | GPIOA1   | 6   | 1   |
 |  66 |  14 |  GPIOC2  | ALT4 | 0 | 23 || 24 | 0 | ALT4 | GPIOC3   | 10  | 67  |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+------+----------+-----+-----+
```

Рис. 5.11. Зміна виходу піну засобами sysfs

Загалом, порівнюючи роботу з реальним комп'ютером та з емулятором, робота через sysfs для управління GPIO не відрізняється для різних платформ. *За завданням викладача потрібно налаштувати пін на одноплатному комп'ютері засобами GPIO.*

Більш зручним, з програмної точки зору, інтерфейсом для контролю пінів є використання спеціалізованих бібліотек. Бібліотека wiringPi дозволяє отримати доступ до пінів введення/виведення. Дана бібліотека вже встановлена на комп'ютері та має бути доступна усім користувачам. Також вона надає набір інструментів, які дозволяють керувати станом GPIO безпосередньо з командного рядка. Дані функції забезпечуються командою `gpio` (рис. 5.12).

```
root@NanoPi-NEO-Air:~# gpio readall
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 |     |     |     3.3V |      |   |  1 || 2  |   |      | 5V       |     |     |
 |  12 |   8 |  GPIOA12 | ALT5 | 0 |  3 || 4  |   |      | 5V       |     |     |
 |  11 |   9 |  GPIOA11 | ALT5 | 0 |  5 || 6  |   |      | 0v       |     |     |
 | 203 |   7 |  GPIOG11 |  OUT | 1 |  7 || 8  | 0 | ALT5 | GPIOG6   | 15  | 198 |
 |     |     |       0v |      |   |  9 || 10 | 0 | ALT5 | GPIOG7   | 16  | 199 |
 |   0 |   0 |  GPIOA0  | ALT5 | 0 | 11 || 12 | 0 | OFF  | GPIOA6   | 1   | 6   |
 |   2 |   2 |  GPIOA2  | ALT5 | 0 | 13 || 14 |   |      | 0v       |     |     |
 |   3 |   3 |  GPIOA3  | ALT5 | 0 | 15 || 16 | 0 | OFF  | GPIOG8   | 4   | 200 |
 |     |     |    3.3v  |      |   | 17 || 18 | 0 | OFF  | GPIOG9   | 5   | 201 |
 |  64 |  12 |  GPIOC0  | ALT4 | 0 | 19 || 20 |   |      | 0v       |     |     |
 |  65 |  13 |  GPIOC1  | ALT4 | 0 | 21 || 22 | 0 | ALT5 | GPIOA1   | 6   | 1   |
 |  66 |  14 |  GPIOC2  | ALT4 | 0 | 23 || 24 | 0 | ALT4 | GPIOC3   | 10  | 67  |
 +-----+-----+----------+------+---+----++----+---+------+----------+-----+-----+
 | BCM | wPi |   Name   | Mode | V | Physical | V | Mode | Name     | wPi | BCM |
 +-----+-----+----------+------+---+-NanoPi-NEO-Air--+------+----------+-----+-----+

 +-----+---NanoPi-NEO-Air Debug UART-+---+
 | BCM | wPi |   Name   | Mode | V | Ph |
 +-----+-----+----------+------+---+----+
 |   4 |  17 |  GPIOA4  | ALT5 | 0 | 37 |
 |   5 |  18 |  GPIOA5  | ALT5 | 0 | 38 |
 +-----+-----+----------+------+---+----+
```

Рис. 5.12. Результат виконання команди gpio readall

Інформація, представлена на рис. надається у вигляді таблиці з наступними даними:

1. Ім’я порту (Name).
2. Режим (Mode).
3. Значення логічного рівня (V).
4. Номер порту в wiringPi (wPi).
5. Номер порту в системі, представлення контролера чіпа (BCM).

Саме даний номер використовується під час роботи за допомогою інтерфейсу sysfs.

За допомогою такої утиліти можна керувати виводами GPIO у більш зручному форматі, ніж за допомогою sysfs. *Ознайомитись з можливостями команди gpio. Налаштувати, за завданням викладача, пін засобами команди gpio*.

Важливим інструментом, з точки зору моніторингу стану системи, є утиліта `cpu_freq` (рис. 5.13).

```
root@NanoPi-NEO-Air:~# cpu_freq
INFO: HARDWARE=sun8i
CPU0 online=1 temp=50 governor=interactive cur_freq=480000
CPU1 online=1 temp=50 governor=interactive cur_freq=480000
CPU2 online=1 temp=50 governor=interactive cur_freq=480000
CPU3 online=1 temp=50 governor=interactive cur_freq=480000
DDR governor=userspace cur_freq=432000 max=432000 min=408000
```

Рис. 5.13. Результат виконання команди cpu_freq

Використання оперативної пам’яті під час роботи з одноплатними комп’ютерами є важливим чинником, який визначає можливість запуску програм. На платі Nano Pi Neo Air встановлена мікросхема пам’яті на 512 Мбайт. Для того, щоб переглянути кількість доступної оперативної пам’яті слід ввести команду `free` (рис. 5.14).

```
root@NanoPi-NEO-Air:~# free
              total        used        free      shared  buff/cache   available
Mem:         506124       47812      315004        2644      143308      402820
Swap:             0           0           0
```

Рис. 5.14. Приклад результату виконання команди free

Незважаючи на те, що за допомогою sysfs можна отримати велику кількість інформації про апаратне забезпечення пристрою, більш зручними є засоби, які орієнтовані на роботу з конкретним типом пристроїв. Прикладами таких засобів є набір утиліт для роботи з інтерфейсом I2C. Раніше було визначено, що на платі доступний один модуль I2C, який пронумерований як пристрій з номером 0.

Перш за все, варто ознайомитись з тим, яким чином можна підключити до плати пристрій з I2C-інтерфейсом. Для цього потрібно переглянути діаграму призначення пінів (рис. 5.15). Як видно з діаграми, у пристрої доступний лише один інтерфейс I2C. У цій роботі до плати буде підключений модуль акселерометра-гіроскопа IntelliSense MPU6050.

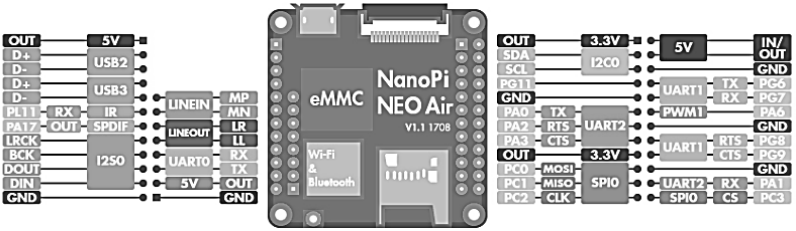

Рис. 5.15. Діаграма призначення пінів

Процедура визначення того, чи підключений певний пристрій, може бути реалізована за рахунок особливостей роботи інтерфейсу I2C. Початком взаємодії завжди є передача адреси пристрою від керуючого пристрою до керованого. Таким чином, перебравши у циклі усі можливі адреси, можна отримати інформацію про те, які пристрої очікують на шині (до шини можуть бути підключені одразу декілька пристроїв, які мають різні адреси). Фактично, дану функцію виконує команда `i2cdetect` (рис. 5.16).

```
root@NanoPi-NEO-Air:~# i2cdetect -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Рис. 5.16. Ідентифікація пристроїв на шині I2C

Ця утиліта сканує усі можливі адреси та виводить ті значення, від яких була отримана відповідь. У цьому випадку отримана відповідь від пристрою з адресою 0x68. Для того, щоб переконатись, що це дійсно є акселерометр-гіроскоп, слід переглянути налаштування, які вказані виробником у документації на пристрій (рис. 5.17).

PARAMETER	CONDITIONS	MIN	TYP
SERIAL INTERFACE			
SPI Operating Frequency, All Registers Read/Write	MPU-6000 only, Low Speed Characterization		100 ±10%
	MPU-6000 only, High Speed Characterization		1 ±10%
SPI Operating Frequency, Sensor and Interrupt Registers Read Only	MPU-6000 only		20 ±10%
I^2C Operating Frequency	All registers, Fast-mode		
	All registers, Standard-mode		
I^2C ADDRESS	AD0 = 0		1101000
	AD0 = 1		1101001

Рис. 5.17. Значення адреси пристрою з документації

Значення 1101000 у двійковому вигляді відповідає значенню 0x68 у шістнадцятковому. У документі наведені два варіанти, оскільки у пристрої наявний вхід AD0, на який можна подати або логічний 0 або логічну 1 для того, щоб мати можливість керувати адресою у випадку підключення кількох пристроїв.

I2C-пристрої зберігають дані про свій стан, а також про виміряні показники у регістрах. Утиліта i2cdump дозволяє зчитати дані з усіх регістрів (діапазон адрес – 0–255), що показано на рис. 5.18.

```
root@NanoPi-NEO-Air:~# i2cdump -y 0 0x68
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 87 03 7e e1 21 01 f9 46 f9 99 05 70 28 6f 4d b3    ??~?!??F???p(oM?
10: 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    T...............
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 40 00 00 00 00 00    ..........@.....
70: 00 00 00 00 00 68 00 00 00 00 00 00 00 00 00 00    .....h..........
80: 87 03 7e e1 21 01 f9 46 f9 99 05 70 28 6f 4d b3    ??~?!??F???p(oM?
90: 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    T...............
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
e0: 00 00 00 00 00 00 00 00 00 00 40 00 00 00 00 00    ..........@.....
f0: 00 00 00 00 00 68 00 00 00 00 00 00 00 00 00 00    .....h..........
```

Рис. 5.18. Зчитування усіх регістрів пристрою

Як видно з рисунку, у MPU6050 наявні декілька секцій, які заповнені даними, а всі інші мають значення 0. *Дослідити карту регістрів пристрою та вказати, які значення мають відповідні регістри*.

Завдання

Nano Pi працює в режимі точки доступу з параметрами, які наведені на рис. 5.19.

```
Configuration Saved!

Please connect your computer to the wireless network:
Wireless Name: NanoPi
     Password: 123456789
done.
```

Рис. 5.19. Налаштування точки доступу, розгорнутої на платі

Кожному студенту виділений окремий акаунт з ім'ям користувача userN (N – номер за списком у журналі) та пароль (отримати від викладача). Nano Pi у режимі точки доступу матиме наступну IP-адресу (рис. 5.20). Підключитись до одноплатного комп'ютера за допомогою будь-якого SSH-клієнта на основі отриманої інформації.

```
root@NanoPi-NEO-Air:~# ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:2884 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2884 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:213360 (213.3 KB)  TX bytes:213360 (213.3 KB)

wlan0     Link encap:Ethernet  HWaddr 02:1a:11:f6:b9:d2
          inet addr:192.168.8.1  Bcast:192.168.8.255  Mask:255.255.2
          inet6 addr: fe80::1a:11ff:fef6:b9d2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1548 errors:0 dropped:1 overruns:0 frame:0
          TX packets:289 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:118570 (118.5 KB)  TX bytes:41381 (41.3 KB)
```

Рис. 5.20. IP-адреса точки комп'ютера

До плати буде підключений модуль MPU6050 для роботи з засобами I2C.

Виконати усі завдання, які описані у ході роботи, навести скріншоти їх успішного виконання.

ПРАКТИЧНА РОБОТА № 6. ПРОГРАМУВАННЯ НА ЦІЛЬОВІЙ ПЛАТФОРМІ

Мета роботи – отримати знання щодо програмування, безпосередньо на цільовій платформі шляхом написання програм, для обробки даних з сенсора та управління інтерфейсом введення/ виведення на мові програмування C/C++.

Незважаючи на той факт, що розробка безпосередньо на цільовій платформі не є найзручнішим варіантом для написання програм для вбудованих систем, така опція є актуальною, коли йдеться про програми невеликого розміру (наприклад, утиліти або тестові приклади). Такі програми призначені для того, щоб швидко перевірити концепції використання бібліотек, доступних команд та ін. За умови достатньої ознайомленності з платформою, а також відповідними інструментами, час на розробку такої програми є співставним з часом, що витрачається під час розробки за допомогою кросс-компіляції.

У даній роботі при написанні програми необхідно використовувати бібліотеку wiringPi. Вона була початково розроблена для одноплатного комп'ютера Raspberry Pi, але у подальшому була адаптована для продуктів виробників Orange Pi та Nano Pi і деяких інших. У випадку Nano Pi репозиторій бібліотеки має назву WiringNP (https://github.com/friendlyarm/WiringNP). Ця бібліотека дозволяє керувати виводами GPIO, а також працювати з послідовними інтерфейсами (UART, I2C, SPI), які доступні у таких пристроях. Додатково реалізована можливість роботи виводу в режимі ШІМ.

Після завантаження з репозиторію бібліотека має бути побудована за допомогою команди make, але з метою проведення роботи, бібліотека побудована і встановлена заздалегідь, тому цей крок не розглядається.

Загалом, процес розробки при використанні даної бібліотеки є доволі простим, оскільки відносно складні команди реалізовані за допомогою компактних функцій. Робота з послідовним інтерфейсами є простішою порівняно з їх використанням у мікроконтролерах. Реалізовані стандартні послідовності інструкції, що характерні під час роботи з відповідними зовнішніми пристроями, які дозволяють організувати обмін даними середньої складності у відносно простій програмі.

Бібліотека має залежність від системної бібліотеки pthread, тому проводити компіляцію програми з використанням wiringPi слід вказавши обидві бібліотеки.

Розглянемо приклад того, як виглядає робота з інтерфейсом I2C за допомогою бібліотеки wiringPi (рис. 6.1).

```
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int fd;
int accx, accy, accz;
int gyrox, gyroy, gyroz;

int read_i2c(int addr) {
        int val;
        val = wiringPiI2CReadReg8(fd, addr);
        val <<= 8;
        val += wiringPiI2CReadReg8(fd, addr);
        if (val >= 0x8000)
                val = -(65536 - val);
        return val;
}

int main() {
        fd = wiringPiI2CSetup(0x68);
        wiringPiI2CWriteReg8(fd, 0x6B, 0x00);
        printf("set 0x6B=%X\n", wiringPiI2CReadReg8(fd, 0x6B));
        while(1) {
                gyrox = read_i2c(0x43);
                gyroy = read_i2c(0x45);
                gyroz = read_i2c(0x47);
                printf("Gx = %i, Gy = %i, Gz = %i\n", gyrox, gyroy, gyroz);
                delay(100);
        }
        return 0;
}
```

Рис. 6.10. Приклад програми для роботи з датчиком

Для того, щоб підключити функції, які надає бібліотека, слід за допомогою директиви include вказати відповідні .h-файли. Для роботи з пінами достатньо файлу wiringPi.h, а функції послідовних інтерфейсів описані в окремих файлах (наприклад, для I2C – у wiringPiI2C.h). Ініціалізація інтерфейсу I2C реалізована функцією wiringPiI2CSetup(). У якості параметра передається адреса пристрою, з яким планується вести комунікації. Як було з'ясовано у попередній роботі, адреса MPU6050 – 0x68. Після цього встановлюється заборона переходу в режим сну шляхом запису значення у відповідний регістр та починається процес зчитування даних з датчика. Зчитування реалізоване у вигляді окремої функції, яка зчитує дані з двох регістрів, розташованих послідовно та об'єднує їх.

Окремо слід звернути увагу на команду компіляції файлу програми (рис. 6.2).

```
root@NanoPi-NEO-Air:~/test_wp# gcc test_acc.c -otest_acc -lwiringPi -lpthread
```

Рис. 6.2. Команда компіляції програми

Саме в такій послідовності потрібно розташувати аргументи для команди компілятора. Зміна місцями імен бібліотек, що мають бути скомпоновані також не допускається, оскільки призведе до помилки.

Окрім розглянутого прикладу, бібліотека супроводжується великою кількістю прикладів, що демонструють основні можливості. Безпосередньо на платі вони розташовані у директорії /root/WiringNP/examples. Також приклади можна переглянути безпосередньо у репозиторії. Розглянемо приклад, який демонструє роботу піну в режимі широтно-імпульсної модуляції (ШІМ) – рис. 6.3.

```
#include <wiringPi.h>

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main (void)
{
  int bright ;

  printf ("Raspberry Pi wiringPi PWM test program\n") ;

  if (wiringPiSetup () == -1)
    exit (1) ;

  pinMode (1, PWM_OUTPUT) ;

  for (;;)
  {
    for (bright = 0 ; bright < 1024 ; ++bright)
    {
      pwmWrite (1, bright) ;
      delay (1) ;
    }

    for (bright = 1023 ; bright >= 0 ; --bright)
    {
      pwmWrite (1, bright) ;
      delay (1) ;
    }
  }

  return 0 ;
}
```

Рис. 6.3. Приклад програми для роботи з ШІМ

Для реалізації даного режиму достатньо лише одного .h-файлу. У прикладі демонструється робота з одним піном. Коефіцієнт заповнення поступово збільшується від мінімального значення до максимального і навпаки. Після закінчення процес починається спочатку завдяки наявності нескінченного циклу.

Завдання

1. Реалізувати програму, яка зчитує дані з датчика MPU6050 та відповідно до кута нахилу по осі X встановлює значення ШІМ на виході. До виходу ШІМ підключений світлодіод.
2. Реалізувати програму, яка реагує на натискання кнопки шляхом переривання. У програмі задіяні 3 виходи ШІМ. За замовчанням демонструється ефект поступового послідовного збільшення та зменшення яскравості на всіх світлодіодах. Натиснення на кнопку зупиняє процес. Повторне натиснення – відновлює.
3. Реалізувати відслідковування позиції датчика на основі показників акселерометра.
4. Реалізувати управління жестами на основі показників гіроскопа для сигналу ШІМ. Використовується один вихід ШІМ. Можна використовувати дані по одній осі.
5. Реалізувати управління жестами на основі показників гіроскопа для сигналу ШІМ. Використовується три виходи ШІМ. Можна використовувати дані по одній осі.

ПРАКТИЧНА РОБОТА № 7. ВІДЛАГОДЖЕННЯ ЗАСОБАМИ GDB

Мета роботи – навчитись відлагоджувати програму для одноплатного комп'ютера за допомогою засобів відлагоджувача GDB.

GNU Debugger (GDB) – один з найпоширеніших відлагоджувачів для програм, які написані мовами C/C++. Даний програмний продукт є безкоштовним. Під час роботи з середовищами розробки для відлагодження часто використовується саме GDB. GDB підтримує також роботу на платформі ARM, тому саме його буде використано як засіб відлагодження у даній роботі.

Перш за все, відлагоджувач необхідно встановити командою:

```
root@NanoPi-NEO-Air:~# apt-get install gdb
```

або аналогічною командою для встановлення пакетів.

У випадку, якщо пакет не знайдено, то слід оновити списки програмного забезпечення репозиторіїв командою apt-get update. Після цього потрібно повторно виконати наведену вище команду. Варто зазначити, що для більшості операційних систем для одноплатних комп'ютерів GDB є доступним з репозиторіїв, тому з його встановленням на інші подібні платформи не має бути проблем.

Відлагодження у консольному вікні дуже відрізняється від традиційного процесу відлагодження у середовищах розробки. Тим не менш, практично всю інформацію, яку надають засоби розробки можна отримати за допомогою GDB. Робота з GDB на системах, які не мають графічного інтерфейсу користувача ведеться шляхом введення команд з запущеною програмою.

Для того, щоб відлагодження було можливим, слід вказати про необхідність включення відлагоджувальної інформації у файл. Це робить додаванням опції -g у команду компіляції. Відповідно, команда виглядатиме таким чином

```
gcc -o <output_name> -g <source_file_name>
```

Після успішної компіляції запустити відлагодження можна командою

```
gdb <output_file>
```

У результаті з'являється новий рядок запрошення (зазвичай, `gdb)`), який очікує введення команд. Вийти з GDB можна командою quit.

Та перш ніж почати працювати з відлагоджувачем, необхідна наявність програми. Як приклад, використовується проста програма (рис. 7.1) де проходить обчислення суми чисел до досягнення певного значення. У результаті значення суми виводиться на дисплей.

```
 GNU nano 2.5.3                                File: debug_test.c

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

int main() {
        int i;
        int sum = 0;
        for (i = 0; i < 25; i++) {
                sum += i;
        }
        printf("The sum equals to %d\n", sum);
        return 0;
}
```

Рис. 7.1. Програма для тестування відлагодження

Відкомпілювати програму варто компілятором gcc, який вже встановлений на одноплатному комп'ютері. Для цього слід ввести команду наступного виду (рис. 7.2).

```
root@NanoPi-NEO-Air:/home/pi/debug_test# gcc -g -o debug_test debug_test.c
```

Рис. 7.2. Команда компіляції
для можливості подальшого відлагодження

Зверніть увагу на наявність опції -g у списку опцій. Ця опція є ключовою для процесу відлагодження, оскільки вона вказує на необхідність включення відлагоджувальної інформації у результуючий двійковий файл. Після запуску відлагодження програми (див. попередні команди), можна запустити виконання програми, проте, це не матиме сенсу оскільки не можна переглянути значення змінних у конкретний момент часу. Тому перед запуском необхідно встановити точки зупинки (англ. breakpoints). Точки зупинки встановлюються шляхом вказання номера рядка у файлі. Якщо програма описана в одному файлі, то ім'я можна опустити, але для випадку кількох файлів бажано вказувати номер рядка. Це не є достатньо зручним, оскільки необхідно або запам'ятати номери рядків у файлах або переключатись між запущеним відлагоджувачем та редактором коду для того, щоб

узгодити ці налаштування. Для встановлення точки зупинки слід ввести команду:

```
break <line_number>
```

У прикладі точка зупинки встановлюється на 10 рядку (рис. 7.3). Після цього слід ввести команду `run` для початку виконання програми. Зупинка відбудеться саме у вказаному рядку.

```
(gdb) break 10
Breakpoint 1 at 0x104f4: file debug_test.c, line 10.
(gdb) run
Starting program: /home/pi/debug_test/debug_test
Cannot parse expression `.L1185 4@r4'.
warning: Probes-based dynamic linker interface failed.
Reverting to original interface.


Breakpoint 1, main () at debug_test.c:11
11              sprintf(message, "The sum equals to %d\n", sum);
(gdb) p i
$1 = 25
(gdb) p sum
$2 = 300
(gdb) p message
$3 = ",\001\000\000\241\004\001\000]\005\001\000PK\375\266,\001\000\000\000\000\000\000\061\005\001\000\000"
```

Рис. 7.3. Відлагодження програми

Найбільшою перевагою, яку надає відлагоджувач, є можливість переглянути значення зміни безпосередньо у процесі виконання програми. Для цього використовується функція print або її скорочена версія p. Після неї потрібно вказати ім'я змінної.

Після того, як програма була запущена на виконання один раз і дійшла до першої точки зупинки, коректним способом продовження роботи програми є введення команди continue або c. Проте, часто після того як програма зупинилась у відповідній точці необхідно виконати декілька наступних інструкцій, а не повністю відновлювати роботу. Для цього слід вказати команду next або step (з відповідними скороченими версіями n та s) – рис. 7.4.

```
Breakpoint 1, main () at debug_test.c:11
11              printf("The sum equals to %d\n", sum);
(gdb) next
The sum equals to 300
12              return 0;
```

Рис. 7.4. Виконання кроку у програмі

Завдання

Провести відлагодження програми, яка була розроблена у попередній роботі з демонстрацію усіх команд.

ПЕРЕЛІК ПОСИЛАНЬ

1. Yocto Project [Електронний ресурс]. – Режим доступу : https://www.yoctoproject.org/ – Загол. з екрана.
2. Yocto Project Overview and Concepts Manual [Електронний ресурс]. – Режим доступу : https://www.yoctoproject.org/docs/2.5.1/overview-manual/overview-manual.html – Загол. з екрана.
3. Zynq-7000 [Електронний ресурс]. Режим доступу : https://www.digikey.com/eewiki/display/linuxonarm/Zynq-7000 – Загол. з екрана.
4. Writing a Linux Kernel Module – Part 1: Introduction [Електронний ресурс]. – Режим доступу : http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/ – Загол. з екрана.
5. Understanding The sysfs File System (/sys) in Linux [Електронний ресурс]. – Режим доступу : https://www.thegeekdiary.com/understanding-the-sysfs-file-system-in-linux/ – Загол. з екрана.
6. Cross compiling for ARM with Ubuntu 16.04 LTS [Електронний ресурс]. – Режим доступу : http://jensd.be/800/linux/cross-compiling-for-arm-with-ubuntu-16-04-lts – Загол. з екрана.
7. NanoPi NEO Air [Електронний ресурс]. – Режим доступу : http://wiki.friendlyarm.com/wiki/index.php/NanoPi_NEO_Air – Загол. з екрана.
8. Wiring Pi GPIO interface library for Raspberry Pi [Електронний ресурс]. – Режим доступу : http://wiringpi.com/ – Загол. з екрана.
9. WiringNP [Електронний ресурс]. – Режим доступу : https://github.com/friendlyarm/WiringNP – Загол. з екрана.
10. GDB: The GNU Project Debugger [Електронний ресурс]. – Режим доступу : https://www.gnu.org/software/gdb/ – Загол. з екрана.

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчальне видання

Ярослав Михайлович
КРАЙНИК

Embedded Systems

Методичні вказівки до виконання практичних робіт

Випуск 259

Редактор *Р. Грубкіна.*
Технічний редактор, комп'ютерна верстка *Д. Кардаш.*
Друк *С. Волинець.* Фальцювально-палітурні роботи *О. Кутова.*

Підп. до друку 30.11.2018
Формат 60x84$^{1}/_{16}$. Папір офсет.
Гарнітура "Times New Roman". Друк ризограф.
Ум. друк. арк. 2,55. Обл.-вид. арк. 1,14.
Тираж 5 пр. Зам. № 5600.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.