



МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ
КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Методичні вказівки до лабораторних робіт
з курсу **“Архітектура комп’ютерів-3. Мікропроцесорні засоби”**
для студентів спеціальності “Комп’ютерна інженерія”

Розглянуто
на засіданні кафедри ОТ
протокол №____ від _____2020 р.

Укладач: Клименко І.А.
Рецензент: Клименко І.А.

ЗМІСТ

Вступ	4
Розділ 1. Робота з мікроконтролером STM32F407VG	5
1.1. Теоретична інформація	5
1.1.1 Архітектура ARM	5
1.1.2 Опис мікроконтролера МК STM32F407VG	8
1.1.2.1 Процесорне ядро	8
1.2. Встановлення програмного забезпечення	12
1.3. Лабораторна робота №1.1	15
1.3.1 Порядок виконання	15
1.3.2 Додаткові матеріали	18
1.4. Лабораторна робота №1.2	19
1.4.1 Скорочені теоретичні відомості	19
1.4.1.1 Основні мікрокоманди асемблера в ARM архітектурі	19
1.4.1.2 Основні директиви асемблера в ARM архітектурі	22
1.4.1.3 Типи ARM інструкцій	23
1.4.1.4 Адресний простір та робота з пам'яттю	24
1.4.1.5 Основні інструкції переходу	26
1.4.1.6 GDB	27
1.4.2 Завдання	27
1.4.3 Варіанти	27
1.4.4 Порядок виконання	28
1.4.5 Література та вказівки	30
1.5. Лабораторна робота №1.3	32
1.5.1 Скорочені теоретичні відомості	32
1.5.1.1 Відлагодження	32
1.5.1.2 Переривання	33
1.5.1.3 Основні команди для лабораторної роботи	34
1.5.1.4 Робота з командами LDR та STR	35
1.5.1.4.A З числовим зсувом	35
1.5.1.4.B З регістровим зсувом	36
1.5.1.4.B Операції доступу до пам'яті з декількома регістрами	37
1.5.1.5 Робота з командою BKPT	38
1.5.1.6 Завантажувач	39
1.5.1.5 Основні директиви для лабораторної роботи	42
1.5.2 Література	43
1.5.3 Завдання	43
1.5.4 Варіанти	44
1.6. Лабораторна робота №1.4	45
1.6.1 Скорочені теоретичні відомості	45
1.6.1.1 Загальні відомості про WH1602B	45
1.6.1.2 General-purpose I/Os (GPIO)	46
1.6.1.3 Reset and clock control (RCC)	47

1.6.2	Порядок виконання	47
1.6.2.1	Етапи роботи	47
1.6.2.2	Створення проекту	47
1.6.2.3	Початкові налаштування	50
1.6.2.4	Робота з периферією	50
1.6.2.5	Ініціалізація дисплея	50
1.6.2.6	Конфігурація портів	51
1.6.2.7	Реалізація керуючих функцій.	52
1.6.2.8	Збірка проекту та його відлагодження	53
Розділ 2.	Робота з платою Beagle Bone Black	55
2.1.	Лабораторна робота №2.1	55
2.1.1	Скорочені теоретичні відомості	55
2.1.1.1	Основна теоретична інформація	55
2.1.1.2	Приклади	57
2.1.2	Література	58
2.1.3	Примітки до роботи	59
2.1.4	Завдання	60
2.1.4.1	Завдання Basic	60
2.1.4.2	Завдання Advanced:	60
2.1.5	Оформлення звіту	61
2.2.	Лабораторна робота №2.2	62
2.2.1	Теоретичні відомості	62
2.2.1.1	Зв'язні списки ядра Linux	62
2.2.1.2	Розподіл пам'яті	64
2.2.2	Література	66
2.2.3	Примітки до роботи	67
2.2.4	Завдання	67
2.2.4.1	Завдання Basic	67
2.2.4.2	Завдання Advanced	67
2.2.5	Оформлення звіту	68
2.3.	Лабораторна робота №2.3	69
2.3.1	Скорочені теоретичні відомості	69
2.3.1.1	Основна теоретична інформація	69
2.3.1.2	Пошук помилок під час компіляції	69
2.3.1.3	Параметри конфігурації налагодження ядра	70
2.3.2	Література	71
2.3.3	Примітки до роботи	71
2.3.4	Завдання	73
2.3.4.1	Завдання Basic	73
2.3.5	Оформлення звіту	73

Вступ

Курс “Архітектура комп’ютерів-3. Мікропроцесорні засоби” розділений на дві частини:

- 1) Робота з мікроконтролером STM32F407VG на архітектурі ARM.
- 2) Вивчення ядра Linux та робота з платою Beagle Bone Black.

Перша частина містить 4 лабораторні роботи, на яких ви вивчите базові команди архітектури ARM, навчитесь створювати базові програми на мові асемблера, відлагоджувати їх за допомогою GDB, а також навчитесь працювати з периферією (монітором) мікроконтролера STM32F407VG.

Друга частина містить 3 лабораторні роботи, на яких ви вивчите основи ядра Linux, навчитесь створювати та відлагоджувати примітивні програми та запускати їх на платі Beagle Bone Black.

Для виконання робіт передбачається, що студент пройшов курс “Операційні системи Linux” та “Системне програмування-1”, тобто знає основні команди Linux та мову асемблера. Також для робіт другої частини потрібні базові знання мови C.

Усі матеріали та приклади коду для всіх робіт доступні за посиланням на репозиторій:

https://github.com/emidot32/AK3_Labs

Розділ 1. Робота з мікроконтролером STM32F407VG

1.1. Теоретична інформація

1.1.1 Архітектура ARM

Advanced RISC Machine (ARM) є сім'єю найбільш розповсюджених ядер у вбудованих системах в світі (станом на 2019 рік). Розроблене компанією Arm Holdings, що дає можливість іншим компаніям розробляти на основі цього сімейства процесори та системи на чіпі будь-якої складності.

Хоч повний опис архітектури і закритий (на нього можна купити ліцензію), все-таки є достатньо відкритої інформації, що стосується саме ядра ARM процесора, щоби можна було вивчати сучасні RISC процесори, їх можливості, та деякі відкриті системи, в які вбудовані ці процесори.

Існує ціла низка спеціалізованих ARM мікроархітектур: ARM1-7, ARM7T, ARM10E, Cortex-A (32-bit), Neoverse та інші. Найбільш поширеними зараз є ARM-Cortex ядра.

Нижче приводиться таблиця застосування та особливостей різних ARM-Cortex сімейств.

Таблиця 1.1. Використання та особливості ARM-Cortex сімейств

Сімейство	Використання	Особливості
ARM-Cortex-M M=Microcontroller Application	Мікроконтролери високої та низької швидкодії загального та спеціалізованого призначення, сигнальні процесори (DSP).	Найдешевше сімейство, порівняно невисока швидкодія, низьке енергоспоживання, маленький розмір ядра на кристалі, віртуальна пам'ять відсутня.
ARM-Cortex-R R=Real-Time	Сигнальні процесори високої якості, системи на чіпі(SOC), мікроконтролери спеціального призначення.	Схоже на Cortex-M, але більш підходить для систем індустріального призначення, найдорожче сімейство.
ARM-Cortex-A A=Application	Системи на чіпі(SOC), мобільні процесори, кластерні суперкомп'ютерні системи,	Висока швидкодія, високе енергоспоживання, існує як 32-бітна так і 64-бітна архітектура.

	процесори загального призначення	
--	----------------------------------	--

В цьому курсі ми будемо вивчати саме **M** сімейство.

Сімейство ARM Cortex-M - це мікропроцесорні ядра ARM, призначені для використання в мікроконтролерах, ASIC, ASSP, FPGA і SoC. Ядра Cortex-M зазвичай використовуються як спеціальні чіпи мікроконтролера, але також "приховані" всередині мікросхем SoC як контролери управління потужністю, контролери вводу / виводу, системні контролери та інше. Cortex-M стали популярною заміною для 8-бітових мікросхем у додатках, які користуються 32-бітовими математичними операціями та заміною старих застарілих ядер ARM, таких як ARM7 та ARM9.

Нижче приведена таблиця використання та особливостей різних ARM-Cortex-M ядер.

Таблиця 1.2. Використання та особливості ARM-Cortex-M ядер.

Ядро\Архітектура	Використання	Особливості
Cortex-M0 ARMv6-M **	Заміна 8 та 16-бітних мікроконтролерів в старих системах.	Оптимізована для малих розмірів та використання в найменших чіпах в дешевих системах.
Cortex-M0+ ARMv6-M * **	Системи на батарейках.	Те ж саме, що і Cortex-M0, але оптимізована ще й для енергозбереження.
Cortex-M1 ARMv6-M *	FPGA.	Розроблене для того, щоб запускатися з програмованих інтегральних схем.
Cortex-M3 ARMv7-M *	Мікроконтролери загального призначення.	Оптимізоване для операційних систем реального часу(RTOS), можлива підтримка ОС загального призначення (uClinux).
Cortex-M4 ARMv7E-M *	Заміна Cortex-M3.	Те ж саме, що і Cortex-M3, але більша швидкодія.
Cortex-M7 ARMv7E-M *	Мікроконтролери високої швидкодії.	Те ж саме, що і Cortex-M4, але більша швидкодія.

Cortex-M23 ARMv8-M * **	Заміна Cortex-M0+	Те ж саме, що і Cortex-M0+, але більший набір інструкцій.
----------------------------------	-------------------	--

*- опціонально є блок захисту пам'яті (MPU)

** - блок операцій з плаваючою комою (FPU) завжди відсутній

Ми будемо вивчати Cortex-M4. Навчившись працювати з будь-якою мікроархітектурою можна легко перейти до іншої у рамках Cortex-M.

Існує декілька компаній, що розробляють мікроконтролери на Cortex-M4 архітектурі: Microchip(Atmel), TI, ST Microelectronics, Cypress Semiconductors, Silicon Labs, NXP, Nordic

Даний документ описує ST Microelectronics STM32F4XX архітектуру, що представлена цілою низкою мікроконтролерів (далі МК).

ARM® Cortex®-M4 (DSP + FPV) – Up to 180 MHz

<div><div>ART Accelerator™ enabling 0 wait state executing from internal Flash</div><div>Up to 2x USB2.0 OTG FS/HS (except for access lines)</div><div>SDIO</div><div>USART, SPI, I2C</div><div>I2S + audio PLL</div><div>16 and 32-bit timers</div><div>Up to 3x 12-bit ADC (0.41 μs)</div><div>Up to 2x 12-bit DAC</div><div>External memory controller</div><div>Low voltage 1.71 to 3.6 V</div></div>	<div><div><div>STM32 F4</div></div></div>	FCPU (MHz)	Flash (bytes)	RAM (KB)	Ethernet I/F IEEE 1588		Camera I/F	SDRAM I/F		SAI3 I/F	Chrom-ART Graphic Accelerator™	TFT LCD controller	MPI DSI																																																																																																									
	2x CAN		Dual Quad-SPI		SPDIF RX																																																																																																																	
	Advanced lines																																																																																																																					
	STM32F469 ²	180	512 K to 2 M	384	<div><div>•</div><div>•</div></div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																									
	STM32F429 ²	180	512 K to 2 M	256	<div><div>•</div><div>•</div></div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																										
	STM32F427 ²	180	1 to 2 M	256	<div><div>•</div><div>•</div></div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																										
	Foundation lines																																																																																																																					
	STM32F446	180	256 K to 512 K	128	<div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																										
	STM32F407 ²	168	512 K to 1 M	192	<div><div>•</div><div>•</div></div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																										
	STM32F405 ²	168	512 K to 1 M	192	<div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div> <div>•</div>	<div>•</div>	<div>•</div>	<div>•</div>																																																																																																										
<table><tr><td>Product lines</td><td>FCPU (MHz)</td><td>Flash (Kbytes)</td><td>RAM (KB)</td><td>Run current (μA/MHz)</td><td>STOP current (μA)</td><td>Small package (mm)</td><td>FSMC (NOR/PSRAM/LCD support)</td><td>QSPI</td><td>DSPDM</td><td>CAN 2.0B</td><td>DAC</td><td>TRNG</td><td>DMA Batch Acquisition Mode</td><td>USB 2.0 OTG FS</td></tr><tr><td colspan="15">Access lines</td></tr><tr><td>STM32F401</td><td>84</td><td>128 to 512</td><td>up to 96</td><td>Down to 128</td><td>Down to 10</td><td>Down to 3x3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>•</td></tr><tr><td>STM32F410</td><td>100</td><td>64 to 128</td><td>32</td><td>Down to 89</td><td>Down to 6</td><td>Down to 2.553x 2.579</td><td></td><td></td><td></td><td></td><td>•</td><td>•</td><td>BAM</td><td>-</td></tr><tr><td>STM32F411</td><td>100</td><td>256 to 512</td><td>128</td><td>Down to 100</td><td>Down to 12</td><td>Down to 3.034x 3.22</td><td></td><td></td><td></td><td></td><td></td><td></td><td>BAM</td><td>•</td></tr><tr><td>STM32F412</td><td>100</td><td>512 to 1024</td><td>256</td><td>Down to 112</td><td>Down to 18</td><td>Down to 3.653x 3.651</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>BAM</td><td>• +LPM</td></tr><tr><td>STM32F413²</td><td>100</td><td>1024 to 1536</td><td>320</td><td>Down to 115</td><td>Down to 18</td><td>Down to 3.951x 4.039</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>BAM+</td><td>• +LPM</td></tr></table>														Product lines	FCPU (MHz)	Flash (Kbytes)	RAM (KB)	Run current (μA/MHz)	STOP current (μA)	Small package (mm)	FSMC (NOR/PSRAM/LCD support)	QSPI	DSPDM	CAN 2.0B	DAC	TRNG	DMA Batch Acquisition Mode	USB 2.0 OTG FS	Access lines															STM32F401	84	128 to 512	up to 96	Down to 128	Down to 10	Down to 3x3								•	STM32F410	100	64 to 128	32	Down to 89	Down to 6	Down to 2.553x 2.579					•	•	BAM	-	STM32F411	100	256 to 512	128	Down to 100	Down to 12	Down to 3.034x 3.22							BAM	•	STM32F412	100	512 to 1024	256	Down to 112	Down to 18	Down to 3.653x 3.651	•	•	•	•	•	•	BAM	• +LPM	STM32F413 ²	100	1024 to 1536	320	Down to 115	Down to 18	Down to 3.951x 4.039	•	•	•	•	•	•	BAM+	• +LPM
Product lines	FCPU (MHz)	Flash (Kbytes)	RAM (KB)	Run current (μA/MHz)	STOP current (μA)	Small package (mm)	FSMC (NOR/PSRAM/LCD support)	QSPI	DSPDM	CAN 2.0B	DAC	TRNG	DMA Batch Acquisition Mode	USB 2.0 OTG FS																																																																																																								
Access lines																																																																																																																						
STM32F401	84	128 to 512	up to 96	Down to 128	Down to 10	Down to 3x3								•																																																																																																								
STM32F410	100	64 to 128	32	Down to 89	Down to 6	Down to 2.553x 2.579					•	•	BAM	-																																																																																																								
STM32F411	100	256 to 512	128	Down to 100	Down to 12	Down to 3.034x 3.22							BAM	•																																																																																																								
STM32F412	100	512 to 1024	256	Down to 112	Down to 18	Down to 3.653x 3.651	•	•	•	•	•	•	BAM	• +LPM																																																																																																								
STM32F413 ²	100	1024 to 1536	320	Down to 115	Down to 18	Down to 3.951x 4.039	•	•	•	•	•	•	BAM+	• +LPM																																																																																																								

Notes:

1. 1.7 V min on specific packages

2. The same devices are also found with embedded Hardware crypto/hash

3. Serial Audio Interface

4. Link Power Management

Рис 1.1. Мікроконтролери STM32F4XX

1.1.2 Опис мікроконтролера МК STM32F407VG

Ми будемо працювати з МК STM32F407VG, що має:

1. ядро Cortex-M4 з FPU на архітектурі ARMv7E-M;
2. адаптивний прискорювач реального часу загрузки програми з флеш пам'яті;
3. 1 МБ флеш пам'яті (FLASH);
4. 192 кб оперативної пам'яті (SRAM);
5. контролер зовнішньої статичної пам'яті;
6. паралельний порт виводу на екран 8080/6080;
7. годинник реального часу(RTC) на 32khz резонаторі;
8. блок живлення SRAM;
9. контролер прямого доступу до пам'яті (DMA);
10. 17 таймерів загального призначення;
11. модуль програмування\відладки SWD\JTAG;
12. 140 портів вводу виводу з можливістю переривання;
13. інтерфейси:
 1. 4 i2c порта,
 2. 4 UART\USART порта,
 3. 3 SPI шини,
 4. 2 CAN шини
 5. інтерфейс карти пам'яті SD (SDIO),
 6. USB2.0 OTG FULL SPEED,
 7. USB2.0 SLAVE,
 8. 10/100 Ethernet MAC.

1.1.2.1 Процесорне ядро

Існує багато сімейств МК на ядрі Cortex-M4. В них багато відмінностей, але ядро та зв'язок ядра з периферією МК незмінне.

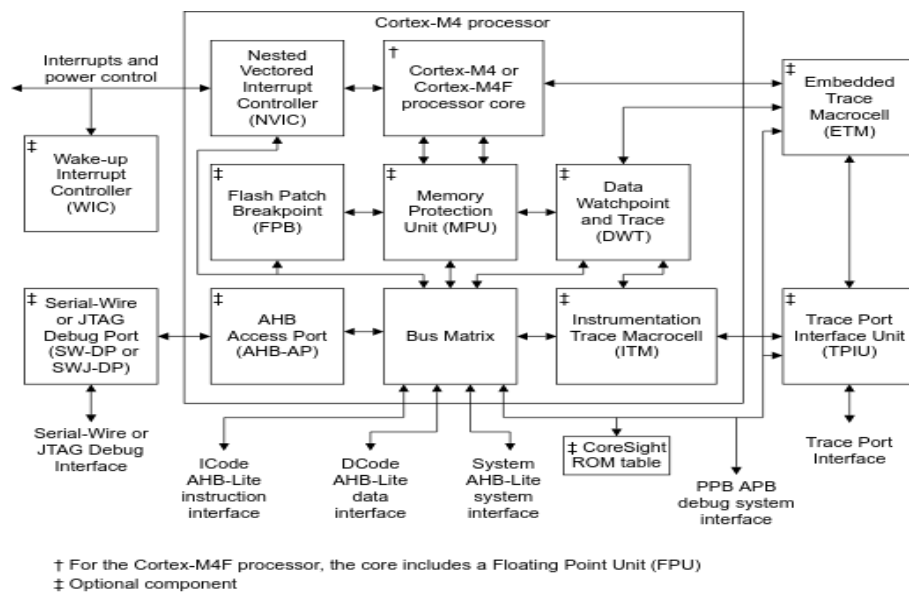


Рис 1.2. Структурна схема процесора (основної частини МК)

- NVIC - Nested Vectored Interrupt Controller (вкладений векторизований контролер переривань), що організовує внутрішні і зовнішні переривання;
- Serial Wire/JTAG — стандартний інтерфейс відладки;
- WIC — Wakeup Interrupt Controller (контролер запуску перериванням), що дозволяє тримати МК виключеним поки не прийде зовнішнє переривання.
- MPU - Memory protection unit - підвищує надійність системи, визначаючи атрибути пам'яті для різних регіонів пам'яті.

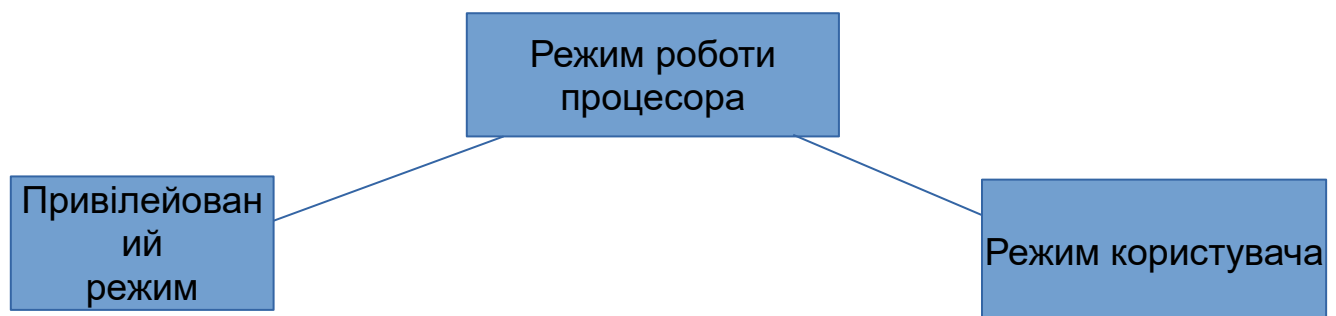


Рис 1.3. Модель програмування

Привілейований (Handler mode) – використовується в обробках переривань, виключень, завантажувачем, операційною системою. Всі інструкції та пам'ять доступні у такому режимі.

Користувача (Thread mode) – використовується для виконання програми користувача, це режим за замовчуванням.

Регістри ядра та АЛП

Name	Type ⁽¹⁾	Required privilege ⁽²⁾	Reset value
R0-R12	read-write	Either	Unknown
MSP	read-write	Privileged	See description
PSP	read-write	Either	Unknown
LR	read-write	Either	0xFFFFFFFF
PC	read-write	Either	See description

PSR	read-write	Privileged	0x01000000
ASPR	read-write	Either	Unknown
IPSR	read-only	Privileged	0x00000000
EPSR	read-only	Privileged	0x01000000
PRIMASK	read-write	Privileged	0x00000000
FAULTMASK	read-write	Privileged	0x00000000
BASEPRI	read-write	Privileged	0x00000000
CONTROL	read-write	Privileged	0x00000000

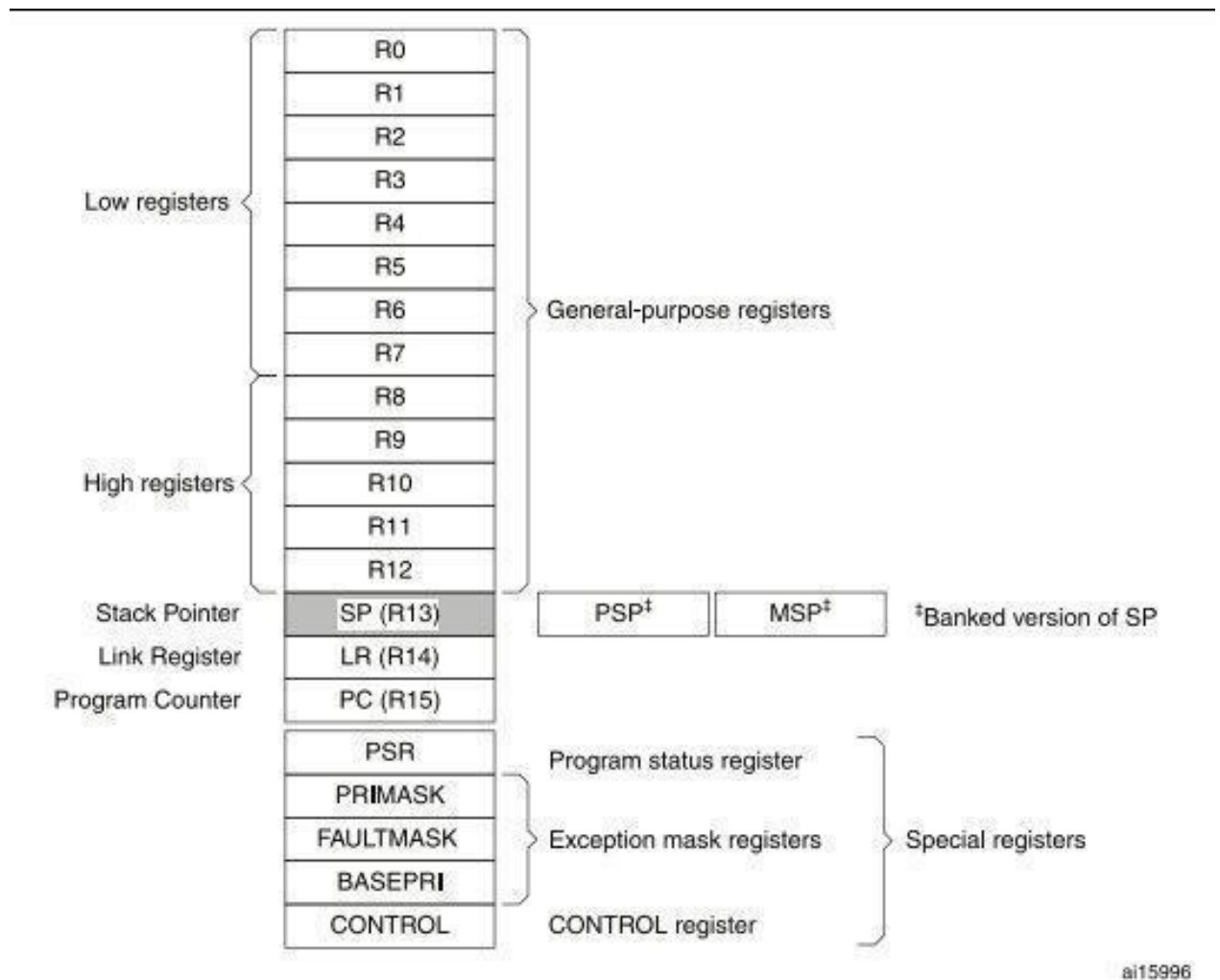


Рис 1.4. Регістри ядра та АЛП

- 32-бітні регістри загального призначення: R0-R12
- Вказівник стеку (Stack Pointer), R13
- В залежності від біту 1 в регістрі CONTROL:

bit[1]=1:

PSP: Process Stack Pointer

bit[1]=0:

MSP: Main Stack Pointer – за замовчуванням

- Link регістр (LR), R14 - зберігає інформацію про повернення для підпрограм, виклики функцій та винятки. Після сбросу процесор завантажує значення LR 0xFFFFFFFF.
- Програмний лічильник (PC), R15. Він містить поточну адресу програми. Після старту процесор завантажує PC на значення вектора RESET, яке знаходиться за адресою 0x00000004. Біт [0] значення завантажується в T-біт EPSR при скиданні процесора і повинен бути 1.

1.2. Встановлення програмного забезпечення

Для виконання робіт вам знадобиться операційна система Linux (будь-який дистрибутив, але бажано Arch-based, для простоти встановлення допоміжних інструментів) та власне нижчевказане ПО:

1. Встановіть емулятор комп'ютерів qemu-system-gnuarmecclipse:

<https://xpack.github.io/qemu-arm/>

2. Завантажте архів з:

<https://github.com/xpack-dev-tools/qemu-arm-xpack/releases/>

та виконайте наступні команди:

```
>>> mkdir -p ~/opt
```

```
>>> cd ~/opt
```

```
>>> tar xvf ~/Downloads/xpack-qemu-arm-2.8.0-7-linux-x64.tgz
```

```
>>> chmod -R -w xPacks/qemu-arm/2.8.0-7
```

Перевірити виконання:

```
>>> ~/opt/xPacks/qemu-arm/2.8.0-7/bin/qemu-system-gnuarmecclipse --version
```

Результат успішного виконання:

```
xPack 64-bit QEMU emulator version 2.8.0-7 (v2.8.0-4-20190211-47-g109b69f49a-dirty)
```

```
Copyright (c) 2003-2016 Fabrice Bellard and the QEMU Project developers
```

3. Встановіть тулчейни:

1) arm-none-eabi-gcc

Debian-based:

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

Або

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

Arch-based:

```
>>> sudo pacman -S arm-none-eabi-gcc
```

2) arm-none-eabi-newlib

Debian -based:

```
>>> sudo apt-get install arm-none-eabi-newlib
```

Або

```
>>> sudo apt-get install libnewlib-arm-none-eabi
```

Arch-based:

```
>>> sudo pacman -S arm-none-eabi-newlib
```

3) arm-none-eabi-gdb

Debian -based:

```
>>> sudo apt-get install arm-none-eabi-gdb
```

Або

```
>>> sudo apt-get install gdb-arm-none-eabi
```

Або

```
>>> sudo apt-get install gdb-multiarch
```

Arch-based:

```
>>> sudo pacman -S arm-none-eabi-gdb
```

4) arm-none-eabi-binutils

Debian -based:

```
>>> sudo apt-get install arm-none-eabi-binutils
```

Або

```
>>> sudo apt-get install binutils-arm-none-eabi
```

Arch-based:

```
>>> sudo pacman -S arm-none-eabi-binutils
```

4. Встановіть утиліти:

1) stlink Firmware programmer for STM32 STLINK v1/v2 protocol

Debian -based:

Щоб установити stlink-tools введіть в терміналі:

```
>>> sudo apt-get install stlink-tools
```

Якщо пакет не було знайдено, вставте в */etc/apt/sources.list* репозиторій
deb <http://ftp.de.debian.org/debian> sid main.

Або командою:

```
>>> sudo apt-add-repository deb http://ftp.de.debian.org/debian sid main
```

Arch-based:

```
>>> sudo pacman -S stlink
```

```
2) make
```

Debian -based:

```
>>> sudo apt-get install make
```

Arch-based:

```
>>> sudo pacman -S make
```

Додаткова література:

PM0214 Programming manual STM32 Cortex®-M4 MCUs and MPUs programming manual [Електронний ресурс] – Режим доступу: https://www.st.com/content/ccc/resource/technical/document/programming_manual/6c/3a/cb/e7/e4/ea/44/9b/DM00046982.pdf/files/DM00046982.pdf/jcr:content/translations/en.DM00046982.pdf

1.3. Лабораторна робота №1.1

Тема: Створення мінімального програмного проекту на мові асемблера.

Мета: Створити мінімальний проект, перевірити виконання відлагоджувачем.

1.3.1 Порядок виконання

1) Створіть файл start.S у директорії проекту(створіть директорію проекту).

Цей файл потрібен для зберігання таблиці векторів виключень (про неї детальніше буде в лабораторній роботі №3) та для мітки hard_reset з якої починається виконання програми.

Зверніть увагу на розширення файлу, воно має значення для компілятора:

```
.syntax unified
.cpu cortex-m4
//.fpu softvfp
.thumb

// Global memory locations.
.global vtable
.global reset_handler
/*
 * vector table
 */
.type vtable, %object
vtable:
    .word __stack_start
    .word __hard_reset__+1
    .size vtable, .-vtable
__hard_reset__:
    ldr r0, =__stack_start
    mov sp, r0
    b __hard_reset__
```

2) Створіть файл lscript.ld. Це скрипт лінкера, який вказує скільки пам'яті може використовувати програма:

```
// linker script for stm32f1 platforms
// Define the end of RAM and limit of stack memory

MEMORY
{
    // We mark flash memory as read-only, since that is where the program lives. STM32
    // chips map their flash memory to start at 0x08000000, and we have 32KB of flash
    // memory available.
    FLASH ( rx )      : ORIGIN = 0x08000000, LENGTH = 1M
    // We mark the RAM as read/write, and as mentioned above it is 4KB long starting at
    // address 0x20000000.
    RAM ( rxw )       : ORIGIN = 0x20000000, LENGTH = 128K
}
__stack_start = ORIGIN(RAM) + LENGTH(RAM); // Start of the stack address
```

3) Зберіть проект, для цього виконайте:

```
>>> arm-none-eabi-gcc -x assembler-with-cpp -c -O0 -g3 -mcpu=cortex-m4 -mthumb
-Wall start.S -o start.o
```

```
>>> arm-none-eabi-gcc start.o -mcpu=cortex-m4 -mthumb -Wall --specs=nosys.specs
-nostdlib -lgcc -T./lscript.ld -o firmware.elf
```

```
>>> arm-none-eabi-objcopy -O binary -F elf32-littlearm firmware.elf firmware.bin
```

На виході маємо бінарний файл.

4) Після цього можна виконати цей код в qemu, використавши відлагоджувач gdb.

Записати в PATH де знаходиться qemu, зверніть увагу, це потрібно робити кожного разу коли запускаєте консоль, тому краще всього додати наступний рядок в, наприклад, /home/user/.bashrc файл:

```
PATN=$PATH:~/opt/xPacks/qemu-arm/2.8.0-7/bin/
```

Виконати:

```
>>> qemu-system-gnuarmeclipse --verbose --verbose --board STM32F4-Discovery --
mcu STM32F407VG -d unimp,guest_errors --image firmware.bin --semihosting-
config enable=on,target=native -s -S
```

З флагами -s -S qemu очікує підключення зовнішнього відлагоджувального ПО з портом tcp::1234. Відкриваємо нове вікно терміналу та виконуємо:

```
>>> arm-none-eabi-gdb firmware.elf
```

В разі успіху:

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from firmware.elf...

(gdb)

Вводимо *target extended-remote:1234*. Тепер програма готова для відлагоджування.

Далі програма очікує введення команд gdb. Для того щоб виконати по кроково вводимо:

>>> step

Далі натискаємо Enter та виконуємо програму.

5) Тепер можна автоматизувати створення прошивки.

Створіть GNU Makefile:

```
SDK_PREFIX?=arm-none-eabi-
CC = $(SDK_PREFIX)gcc
LD = $(SDK_PREFIX)ld
SIZE = $(SDK_PREFIX)size
OBJCOPY = $(SDK_PREFIX)objcopy
QEMU = qemu-system-gnuarmelclipse
BOARD ?= STM32F4-Discovery
MCU=STM32F407VG
TARGET=firmware
CPU_CC=cortex-m4
TCP_ADDR=1234
deps = \
    start.S \
    lscript.ld
all: target
target:
    $(CC) -x assembler-with-cpp -c -O0 -g3 -mcpu=$(CPU_CC) -Wall start.S -o start.o
    $(CC) start.o -mcpu=$(CPU_CC) -Wall --specs=nosys.specs -nostdlib -lgcc -
T./lscript.ld -o $(TARGET).elf
    $(OBJCOPY) -O binary -F elf32-littlearm $(TARGET).elf $(TARGET).bin
qemu:
    $(QEMU) --verbose --verbose --board $(BOARD) --mcu $(MCU) -d unimp,guest_errors
--image $(TARGET).bin --semihosting-config enable=on,target=native -gdb
tcp::$(TCP_ADDR) -S
clean:
```

```
-rm *.o  
-rm *.elf  
-rm *.bin
```

Команда *make* створить прошивку, команда *make qemu* запустить емулятор з вищезазначеними налаштуваннями.

1.3.2 Додаткові матеріали

- 1) The GNU MCU Eclipse QEMU command line options [Електронний ресурс]: Режим доступу - <https://gnu-mcu-eclipse.github.io/qemu/options/>
- 2) GDB and Debugging [Електронний ресурс]: Режим доступу - <https://web.stanford.edu/class/archive/cs/cs107/cs107.1196/resources/gdb>
- 3) GDB QUICK REFERENCE [Електронний ресурс]: Режим доступу - <http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

1.4. Лабораторна робота №1.2

Тема: Основні інструкції 32-бітного ARM процесора для мікроконтролерів

Мета: Навчитися використовувати асемблерні інструкції ядра Cortex-M4, працювати з процедурами і базово зрозуміти архітектуру ядра.

1.4.1 Скорочені теоретичні відомості

1.4.1.1 Основні мікрокоманди асемблера в ARM архітектурі:

1) ADD, ADC, SUB, SBC, and RSB

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2  
op{cond} {Rd,} Rn, #imm12; ADD and SUB only
```

Рис 1.5. Приклад синтаксиса ADD, SUB

*cond - умовний код. Rd - регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn. Rn - регістр, в якому знаходиться перший операнд. Operand2 - константа або регістр. #imm12 - значення з проміжку 0-4095

- Мікрокоманда **ADD** додає значення operand2 або #imm12 до значення в Rn.
- Мікрокоманда **ADC** додає значення operand2 до значення в Rn, враховуючи флаг переносу.
- Мікрокоманда **SUB** віднімає значення operand2 або #imm12 від значення в Rn.
- Мікрокоманда **SBC** віднімає значення operand2 або #imm12 від значення в Rn, враховуючи флаг переносу
- Мікрокоманда **RSB** віднімає значення Rn від значення в operand2.

2) MUL, MLA, and MLS

Syntax

```
MUL{S}{cond} {Rd,} Rn, Rm ; Multiply  
MLA{cond} Rd, Rn, Rm, Ra ; Multiply with accumulate  
MLS{cond} Rd, Rn, Rm, Ra ; Multiply with subtract
```

Рис 1.6. Приклад синтаксиса MUL,MLA,MLS

*cond - умовний код. Rd - регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn. Rn, Rm - регістри, в яких знаходяться операнди. Ra - регістр, що зберігає значення, до якого додається або від якого віднімається результат

- Мікрокоманда **MUL** множить значення в Rn та Rm, молодші 32 біти результату зберігаються в Rd.
- Мікрокоманда **MLA** множить значення в Rn та Rm, а потім додає то результату значення з Rd, молодші 32 біти результату зберігаються в Rd.
- Мікрокоманда **MLS** множить значення в Rn та Rm, а потім віднімає результат від значення з Rd, молодші 32 біти результату зберігаються в Rd.

3) SDIV and UDIV

Syntax

```
SDIV{cond} {Rd,} Rn, Rm
UDIV{cond} {Rd,} Rn, Rm
```

Рис 1.7. Приклад синтаксиса SDIV, UDIV

*cond - умовний код. Rd - регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn. Rn - регістр, в якому знаходиться ділене. Rm - регістр, в якому знаходиться дільник.

- Мікрокоманда **SDIV** виконує ділення зі знаком значення в Rn на Rm, результат зберігається в Rd.
- Мікрокоманда **UDIV** виконує ділення без знаку значення в Rn на Rm, результат зберігається в Rd.
- Для обох мікрокоманд виконується наступне правило:
- Якщо значення в Rn не ділиться на Rm, результат округлюється до 0.

4) ASR, LSL, LSR, ROR and RRX

Syntax

```
op{S}{cond} Rd, Rm, Rs
op{S}{cond} Rd, Rm, #n
RRX{S}{cond} Rd, Rm
```

Рис 1.8. Приклад синтаксиса ASR, LSL, LSR, ROR and RRX

*cond - умовний код. Rd - регістр призначення (куди зберігається результат

– обов’язковий). Rm - регістр, що містить значення, яке зсувається. Rs - регістр, що містить значення довжини зсуву регістра Rm. n - значення довжини зсуву регістра Rm.

Допустимі значення n:

- ARS: Shift length from 1 to 32
- LSL: Shift length from 0 to 31
- LSR: Shift length from 1 to 32
- ROR: Shift length from 1 to 31

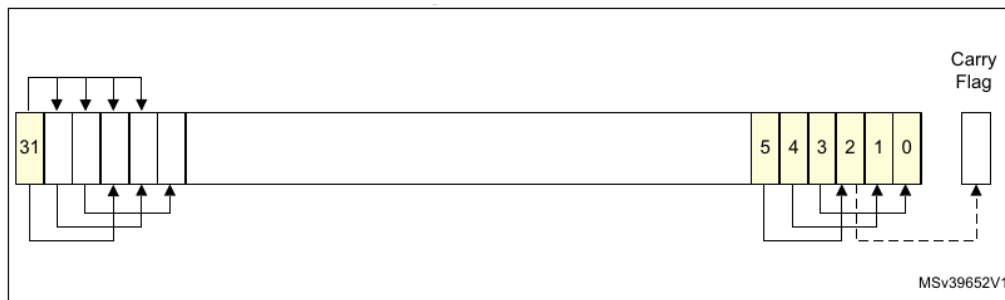


Рис 1.9. Побітова схема роботи команди ARS

Мікрокоманда **ARS** виконує арифметичний зсув праворуч значення в Rm на Rs, результат зберігається в Rd.

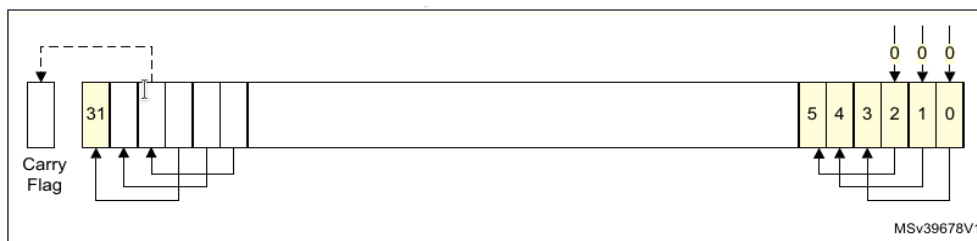


Рис 1.10. Побітова схема роботи команди LSL

Мікрокоманда **LSL** виконує логічний зсув ліворуч значення в Rm на Rs, результат зберігається в Rd.

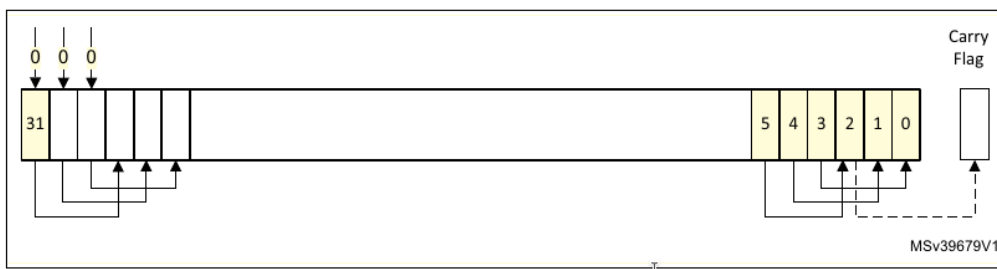


Рис 1.11. Побітова схема роботи команди LSR

Мікрокоманда **LSR** виконує логічний зсув праворуч значення в Rm на Rs, результат зберігається в Rd.

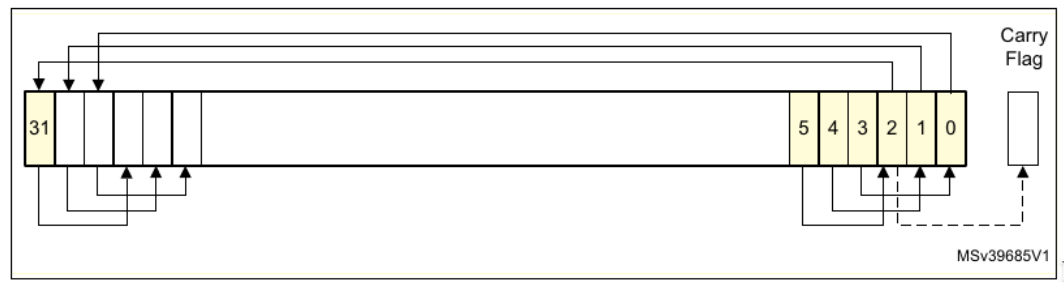


Рис 1.12. Побітова схема роботи команди ROR

Мікрокоманда **ROR** виконує поворот праворуч значення в Rm на Rs, результат зберігається в Rd.

5) AND, ORR and EOR

op{S} {cond} Rd, Rn, Operand2

*cond - умовний код. Rd - регістр призначення (куди зберігається результат). Якщо не зазначений, результат зберігається в Rn. Rn - регістр, в якому знаходиться перший операнд. Operand2 - константа або регістр.

- Мікрокоманда **AND** виконує побітову операцію І на операндах Rn та Operand2.
- Мікрокоманда **ORR** виконує побітову операцію АБО на операндах Rn та Operand2.
- Мікрокоманда **EOR** виконує побітову операцію ВИКЛЮЧНЕ АБО на операндах Rn та Operand2.

6) PUSH and POP

Push – «заштовхує» регістри в стек. Pop – «виштовхує» регістри із стека.

Команди приймають список регістрів. Наприклад:

push {r0, r1, r2}

pop {r0, r1, r2}

1.4.1.2 Основні директиви асемблера в ARM архітектурі

.syntax - Директива **.syntax** дозволяє вибрати синтаксис мови *thumb2* (новий або застарілий). Ми використовуємо останню версію за замовчуванням (*unified*). Директива є архітектурно-залежною (директива має бути використана в кожному асемблерному файлі).

.size - Вказує скільки місця займають дані, на які вказує певний символ. Наприклад `.size __hard_reset__, .- __hard_reset__` обчислить розмір функції `__hard_reset__` в байтах.

.word - (`.word __hard_reset__ +1`) Встановлює комірку пам'яті 32 -бітним числом, що є адресою процедури(переривання RESET)+1. Біт 0 в даному випадку – флажок переходу в режим виконання thumb інструкцій, якби він був поставлений в 0, процесор видав би помилку, оскільки у нас є тільки 1 режим інструкцій(перевірте це).

.thumb - Дає зрозуміти асемблеру, що треба генерувати виключно thumb інструкції процесора.

.global - За замовчуванням асемблер створює невидимі ззовні файлу символи коли ми створюємо нові процедури, змінні, будь-які інші об'єкти. Щоб процедура могла звернутись з іншого файлу в процедуру в цьому файлі, її потрібно створити видимою за допомогою директиви `.global`

1.4.1.3 Типи ARM інструкцій

Існує 2 типа ARM інструкцій:

1. Стандартний ARM набір, що дозволяє швидко виконувати операції.
2. Thumb, що дозволяє значно зменшити розміри виконуваної програми.

Стандартний ARM набір складається з інструкцій, що мають різну довжину (від 16 біт до 64 і навіть більше), що робить конвеєр в ядрі досить складним, але швидкодія більша ніж у процесорів, що мають тільки thumb набір інструкцій.

Thumb набір – підмножина стандарту, що складається з 16-ти та 32-бітних інструкцій.

Наприклад, сімейство Cortex-M0 реалізує в основному тільки 32 бітні інструкції. Сімейство Cortex-M4 реалізує повний набір thumb інструкцій, але має тільки thumb режим. Сімейства Cortex-A4 можуть працювати в 2х режимах: thumb та ARM.

1.4.1.4 Адресний простір та робота з пам'яттю

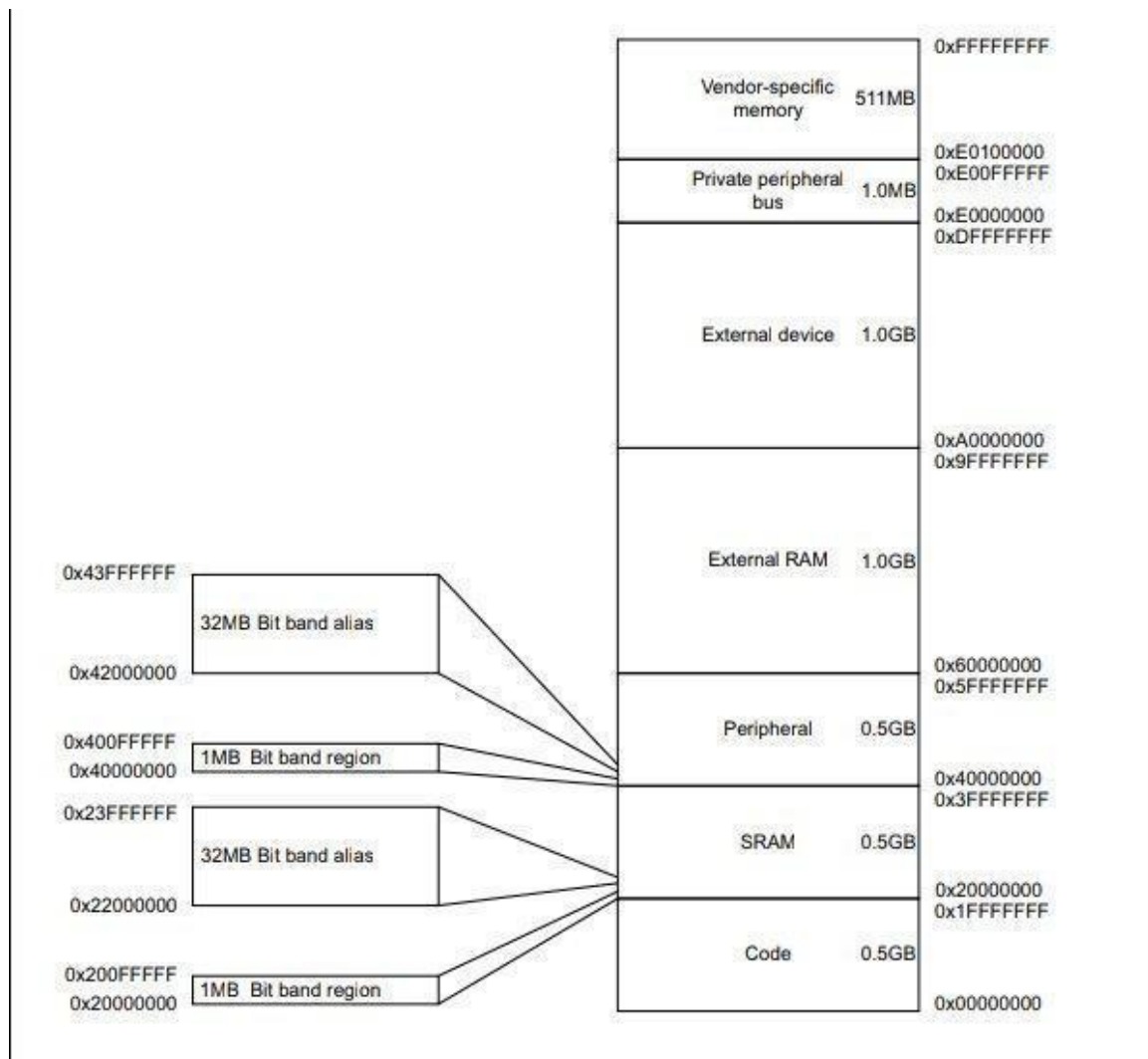


Рис 1.13. Адресний простір

Це фон-Неймановська архітектура, інструкції можна виконувати і з розділу code і з розділу SRAM (STATIC RAM), і з розділу External RAM, якщо підключено зовнішню пам'ять. Зверніть увагу на регіон 0x0000 0000 – 0x2000 0000, зазвичай там є доступ до постійної (FLASH) пам'яті.

При чому в STM32 девайсах це адресний простір постійної пам'яті.

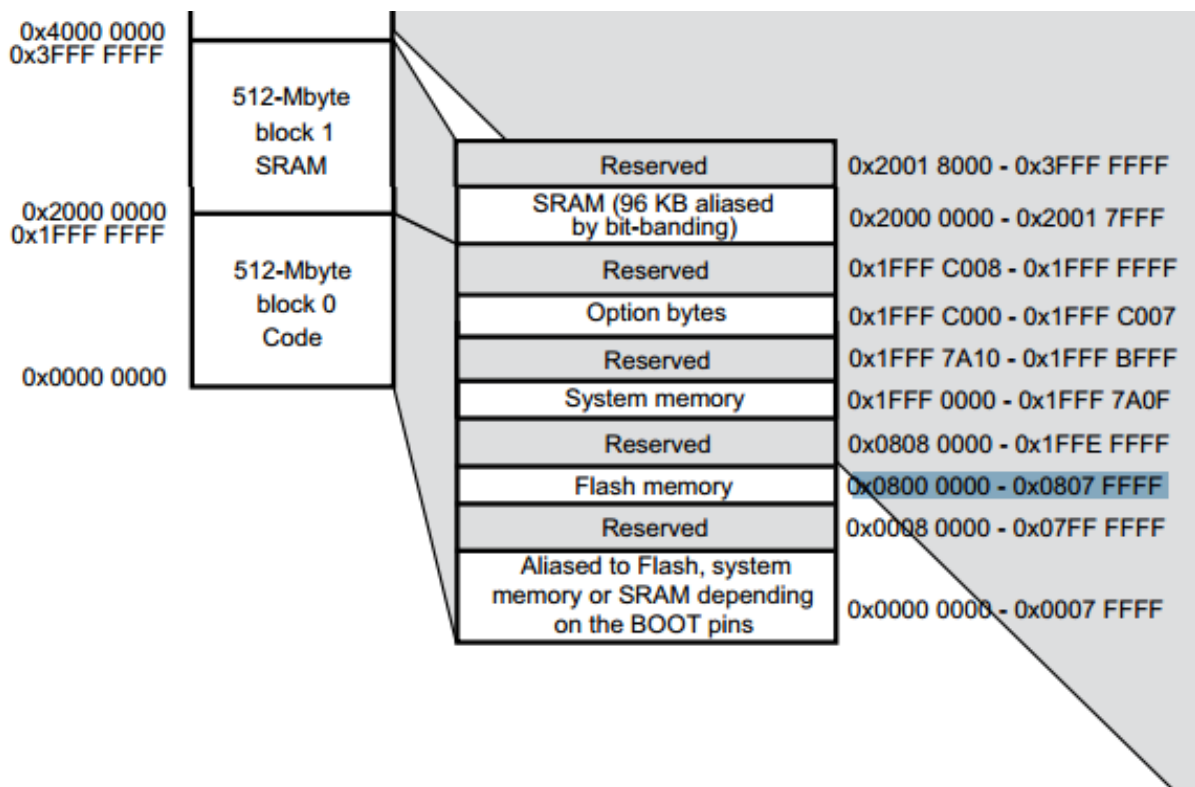


Рис 1.14. Адресний простір

0x0000 0000 - 0x0007 FFFF- це *Віртуальна пам'ять*, в залежності від початкової конфігурації може виступати і SRAM (0x2000 0000), і флеш пам'ять (0x0800 0000), і системна пам'ять (ROM) 0x1FFF 0000.

Процесор вважає, що в стартовій адресі 0x0000 0000 знаходиться таблиця векторів переривань, при чому кожний вектор є 32-бітним, 1-ий вектор – початкове значення вказівника стеку SP, друге значення – вказівник на початкове значення PC.

Рядок:

```
.word __stack_start
```

Встановлює вказівник стеку у вказану адресу, в даному випадку:

```
{
    RAM ( rxw )      : ORIGIN = 0x20000000, LENGTH = 128K
}
__stack_start = ORIGIN(RAM) + LENGTH(RAM);
```

Вона встановлює його в кінець оперативної пам'яті.

В флеш пам'яті зазвичай знаходиться наша програма і стартує звідти за замовчуванням.

1.4.1.5 Основні інструкції переходу

Mnemonic	Brief description	See
B	Branch	<i>B, BL, BX, and BLX on page 142</i>
BL	Branch with Link	<i>B, BL, BX, and BLX on page 142</i>
BLX	Branch indirect with Link	<i>B, BL, BX, and BLX on page 142</i>
BX	Branch indirect	<i>B, BL, BX, and BLX on page 142</i>
CBNZ	Compare and Branch if Non Zero	<i>CBZ and CBNZ on page 144</i>
CBZ	Compare and Branch if Non Zero	<i>CBZ and CBNZ on page 144</i>
IT	If-Then	<i>IT on page 145</i>
TBB	Table Branch Byte	<i>TBB and TBH on page 147</i>
TBH	Table Branch Halfword	<i>TBB and TBH on page 147</i>

Рис 1.15. Інструкції переходу

Всі інструкції переходу відносні.

Особливу увагу слід звернути на інструкцію **BL <відносна адреса>** та її відмінність від **B <відносна адреса>**. Вона не тільки переходить (змінює **PC**), а ще й зберігає стару адресу **PC** в регістр **LR** та збільшує її на 1 (при переході за адресою в регістрі перевіряється тип інструкцій за першим бітом(флагом)). Визвана процедура має зберігати **LR** (або значення з нього) щоб мати можливість завершитись і передати виконання процедурі, що її визвала. Якщо значення адреси зберігалось в **LR** можна визвати **BX LR**.

BX переходить за адресою у регістрі. Є і інший більш використовуваний спосіб. Це - використання стеку. Викликана процедура зберігає у стек разом з регістрами **змінних** (див AAPCS (Procedure Call Standard for the ARM Architecture)), що вона буде використовувати і **LR**.

`push { r4, r6, lr }` - зверніть Увагу, що лише 1 інструкція може зберегти декілька регістрів у стек. `pop { r4, r6, pc }` одразу витаскує значення всіх регістрів, у тому числі виходить з процедури.

Проведемо наступний дослід - спробуємо визвати коректно процедуру без **BL**.

В коді до 1 лабораторної знайдіть:

```
__hard_reset__:
// initialize stack here
// if not initialized yet
bl lab1
_loop: b _loop
.size __hard_reset__, .-__hard_reset__
```

Замініть `bl lab1` на:

```

mov lr, _loop
add lr, 1
b lab1

```

Отже 3 інструкції замість одної правильно її замінюють. `mov lr, _loop` записує адресу повернення, `add lr, 1` дає процесору зрозуміти при виході з процедури, що використовується набір `thumb` інструкцій, `b lab1` просто переходить до виконання процедури.

1.4.1.6 GDB

GDB – відлагоджувач програм. З його допомогою можна покроково виконати програму та слідкувати за станами регістрів.

Основні команди:

- `layout regs` -- Відображає усі регістри разом з вихідним кодом.
- `help` -- Відображає список класів команд
- `kill` -- Примусово зупиняє запущену програму
- `step` -- Виконати наступний рядок коду
- `stepi` -- Виконати наступну машинну інструкцію
- `continue` -- Продовжити виконання програми
- `quit` -- Вийти з GDB

1.4.2 Завдання

Написати програму на GAS, яка рахує функцію за варіантом та запустити її в відлагоджувачі `gdb`, як показано в лабораторній роботі 1. Показати регістри з вірним результатом.

Варіант визначається остачею від ділення на 5 номера залікової книжки:
 $XXXX\%5 = \text{№ варіанта.}$

1.4.3 Варіанти

Таблиця 1.3. Таблиця варіантів

№	Функція
0	$(a+b)/2 + c!$
1	$(a-b)*3 + 2^c$
2	$(a\&b)>> + c!$
3	$\begin{cases} (a+b)/c, & (a-b) \geq 0 \\ (a-b)/c, & (a-b) < 0 \end{cases}$

4	$\begin{cases} (a \mid b) * c, & (a * b) \geq 10 \\ (a \& b) * c, & (a * b) < 10 \end{cases}$
---	---

*>> - логічний зсув вправо

1.4.4 Порядок виконання

- 1) Створити файли start.S та lscript.ld як в лабораторній роботі №1, але зміст файлу start.S наступний:

```
.syntax unified
.cpu cortex-m4
//.fpu softvfp
.thumb

// Global memory locations.
.global vtable
.global __hard_reset__

/*
 * vector table
 */
.type vtable, %object
.type __hard_reset__, %function
vtable:
    .word __stack_start
    .word __hard_reset__+1
    .size vtable, .-vtable
__hard_reset__:
// initialize stack here
// if not initialized yet
    bl lab1
    _loop: b _loop
    .size __hard_reset__, .-
__hard_reset__
```

- 2) Створіть файл lab1.S, в якому і буде програма за варіантом. Приклад:

lab1.S

```
.global lab1
.syntax unified
```

```

#define A #4
lab1:
    push {lr}
    // calculate
    mov r0, A
    mov r1, #0
    bl test_var
    pop {pc}
test_var:
    push { r0, r1, lr }
    cmp r0, r1
    ITE GE
    movGE r3, r0
    movLT r2, r0
    pop { r0, r1, r2, pc }

```

3) Створіть Makefile для автоматичної збірки проекту:

```

SDK_PREFIX?=arm-none-eabi-
CC = $(SDK_PREFIX)gcc
LD = $(SDK_PREFIX)ld
SIZE = $(SDK_PREFIX)size
OBJCOPY = $(SDK_PREFIX)objcopy
QEMU = qemu-system-gnuarmeclipse
BOARD ?= STM32F4-Discovery
MCU=STM32F407VG
TARGET=firmware
CPU_CC=cortex-m4
TCP_ADDR=1234
deps = \
    start.S \
    lscript.ld

all: target

target:
    $(CC) -x assembler-with-cpp -c -O0 -g3 -mcpu=$(CPU_CC) -Wall start.S -o
    start.o

```

```

$(CC) -x assembler-with-cpp -c -O0 -g3 -mcpu=$(CPU_CC) -Wall lab1.S -o
lab1.o

$(CC) start.o lab1.o -mcpu=$(CPU_CC) -Wall --specs=nosys.specs -nostdlib
-lgcc -T./lscript.ld -o $(TARGET).elf

$(OBJCOPY) -O binary -F elf32-littlearm $(TARGET).elf $(TARGET).bin

qemu:

$(QEMU) --verbose --verbose --board $(BOARD) --mcu $(MCU) -d
unimp,guest_errors --image $(TARGET).bin --semihosting-config
enable=on,target=native -gdb tcp::$(TCP_ADDR) -S

clean:

    -rm *.o
    -rm *.elf
    -rm *.bin

flash:

st-flash write $(TARGET).bin 0x08000000

```

4) Виконайте збірку проекту за допомогою make:

```
>>> make
```

5) Запустіть емулятор qemu за допомогою make qemu:

```
>>> make qemu
```

6) В іншому терміналі запустіть відлагоджувач gdb командою *arm-none-eabi-gdb firmware.elf*. Та виконайте програму крок за кроком. Продемонструйте значення регістрів.

1.4.5 Література та вказівки

1. STM32 Cortex®-M4 MCUs and MPUs programming manual [Електронний ресурс]: Режим доступу - https://www.st.com/content/ccc/resource/technical/document/programming_manual/6c/3a/cb/e7/e4/ea/44/9b/DM00046982.pdf/files/DM00046982.pdf/jcr:content/translations/en.DM00046982.pdf
2. Useful assembler directives and macros for the GNU assembler [Електронний ресурс]: Режим доступу - <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/useful-assembler-directives-and-macros-for-the-gnu-assembler>
3. Assembler Directives [Електронний ресурс]: Режим доступу - https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html

Вони дозволять вам розуміти синтаксис мови асемблера GAS (GNU Assembly), що є частиною стандартного пакету тулчейну GCC (GNU Compiler Collection) для арм (arm-none-eabi-). А також ви зможете вже працювати з GDB відлагоджувальником самостійно.

Слід звернути увагу на те, що препроцесор C, що є частиною пакету GCC, може бути використаний (і так слід робити) в GAS. Тобто Сшні макроси, коментарі, і т.і там підтримуються. В GAS є свій препроцесор, але він вмикається після роботи Сішного препроцесора. Деякий його функціонал ми також використовуємо.

4. ARM7TDMI Technical Reference Manual. The Thumb instruction set
[Електронний ресурс]: Режим доступу - <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>

Але рекомендується використовувати мануал від ST, оскільки ми працюємо з їх реалізацію Cortex мікропроцесорів (див. перше посилання)

1.5. Лабораторна робота №1.3

Тема: Завантажувач основної програми. Обробка виключень. Вивід даних на відлагоджувальний порт або консоль.

Мета: Навчитися працювати з оперативною пам'яттю, використовувати інструкції спеціального призначення, використовувати виключення процесора Cortex-M4. Створення мінімального завантажувача системи. Навчитися користуватися виводом даних через відлагоджувальний порт (або консоль).

1.5.1 Скорочені теоретичні відомості

1.5.1.1 Відлагодження

Відлагоджувальний функціонал процесора дозволяє передавати інформацію через програматор на реальному пристрої на базі ARM-Cortex M, та в емуляторах, таких як **Qemu**. Qemu дозволяє використати системні виводи ОС, на якій він виконується через системні(відлагоджувальні) виводи пристрою, що емулюється.

Для відлагодження в цій лабораторній роботі використовується мікрокоманда BKPT (детальніше в розділі **5.1.5**). Відлагодження програми буде проходити в режимі Semihosting.

Semihosting - це механізм, який дає змогу коду, що працює у вбудованій системі (його також називають цільовим), обмінюватись даними та використовувати пристрої вводу / виводу хост-комп'ютера. В основному, цільовий процесор зупиняється, або шляхом запуску команди на зупинку, або через якусь іншу операцію, яка зупиняє виконання програми і ставить цільовий процесор під контроль відлагоджувального агента. Відлагоджувальний агент зчитує один або кілька регістрів, щоб визначити тип операції, яку повинен виконувати хост від імені цілі, а потім виконує дію, після чого перезапускає ціль.

У більшості систем важливою частиною є введення / виведення до терміналу або принаймні тільки вивід. Це можливість надсилати текст хосту, відображати його на консолі відлагодження або терміналі. В цій лабораторній роботі буде використана лише дана операція, для перевірки правильності роботи коду.

1.5.1.2 Переривання

В даній лабораторній роботі використовується механізм переривання, на прикладі переривання **RESET**. Дане переривання відбувається при включенні живлення, або при сбросі системи. Найпершою під час старту системи запускається програма, що відповідає за обробку цього переривання.

Задати програму, що буде виконуватись відразу після старту системи, або оброблювати будь-яке інше переривання, можна за допомогою векторної таблиці переривань.

Векторна таблиця складається з: початкового значення **Stack Pointer**, та адрес, які називаються векторами переривань. 1-ий біт кожного вектора має дорівнювати 1, щоб вказати процесору на використання **thumb** інструкцій.

Кожний вектор переривання є по суті адресою програми, що буде виконуватись якщо дане виключення з'явиться.

Таблиця 1.4. Вектори переривання

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

MS30018V1

Таблиця 1.5. Переривання Cortex - M4 процесора

Exception number ⁽¹⁾	IRQ number ⁽¹⁾	Exception type	Priority	Vector address or offset ⁽²⁾	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable ⁽³⁾	0x00000010	Synchronous
5	-11	Bus fault	Configurable ⁽³⁾	0x00000014	Synchronous when precise Asynchronous when imprecise
6	-10	Usage fault	Configurable ⁽³⁾	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable ⁽³⁾	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable ⁽³⁾	0x00000038	Asynchronous
15	-1	SysTick	Configurable ⁽³⁾	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable ⁽⁴⁾	0x00000040 and above ⁽⁵⁾	Asynchronous

Детальніше з механізмом виключень та з їх видами можна в розділі 2.3 PM0214

1.5.1.3 Основні команди для лабораторної роботи

Основними командами, що використовуються в даній лабораторній роботі, є команди доступу до пам'яті **STR** (store register) та **LDR** (load register).

LDR – завантажує дані з пам'яті до регістрів.

STR – зберігає значення регістрів у пам'ять.

Існує декілька варіантів цих команд, в залежності від виду зсуву, адресації, типу даних та кількості регістрів. Також, як уже було сказано в 5.1.1, в даній лабораторній роботі використовується відлагодження програми в режимі **Semihosting**, що виконується за допомогою мікрокоманди **БКРТ**.

1.5.1.4 Робота з командами LDR та STR

1.5.1.4.A З числовим зсувом

- **Зсувна адресація:** значення зсуву (offset) додається (чи віднімається) до адреси, що знаходиться в регістрі Rn. По адресі, що утворилась в результаті, відбувається доступ до пам'яті. При цьому значення в регістрі Rn не змінюється.
- **Пре-індексна адресація:** значення зсуву (offset) додається (чи віднімається) до адреси, що знаходиться в регістрі Rn. Адреса, що утворилась в результаті, використовується для доступу до пам'яті та записується в регістр Rn.
- **Пост-індексна адресація:** адреса, що знаходиться в регістрі Rn, використовується для доступу до пам'яті. Значення зсуву (offset) додається (чи віднімається) до адреси, та записується назад в регістр Rn.

Таблиця 1.6. Можливі значення зсуву (offset)

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	-255 to 4095	-255 to 255	-255 to 255
Two words	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020

Синтаксис операції:

- `op{type}{cond} Rt, [Rn {, #offset}];` безпосередня адресація
- `op{type}{cond} Rt, (Rn, #offset]!;` пре-індексна адресація
- `op{type}{cond} Rt, (Rn], #offset;` пост-індексна адресація
- `opD{cond} Rt, Rt2, [Rn {, #offset}];` безпосередня адресація, два машинних слова
- `opD{cond} Rt, Rt2, (Rn, #offset]!;` пре-індексна адресація, два машинних слова
- `opD{cond} Rt, Rt2, [Rn], #offset;` пост-індексна адресація, два машинних слова

Де:

- `op` - LDR або STR;

- type - один з допустимих типів:
 - B: Байт без знаку, незаповнені біти заповнюються нулями;
 - SB: Байт зі знаком, незаповнені біти заповнюються знаком (лише LDR);
 - H: Півслова без знака, незаповнені біти заповнюються нулями;
 - SH: Півслова зі знаком, незаповнені біти заповнюються знаком (лише LDR)
 - —: Не вказувати тип (для адресації повного машинного слова),
- cond - умовний код;
- Rt - регістр, в який завантажуються, чи з якого вивантажуються дані;
- Rn - регістр, в якому знаходиться адреса пам'яті;
- offset - зсув (відступ) в байтах від адреси, що знаходиться в Rn (якщо не вказаний - адресою вважається вміст Rn);
- Rt2 - додатковий регістр для 2-слівних операцій.

Приклади використання:

LDR R8, [R10];	Завантажує в R8 слово за адресою в R10
STRH R3, [R4], #4;	Вивантажує півслова з R3 в пам'ять за адресою в R4, після чого додає до R4 4
STRD R0, R1, [R8], #-16;	Вивантажує слово з R0 за адресою в R8, та вивантажує слово з R1 за адресою на 4 байти більшою від R8, після чого декрементує R8 на 16

1.5.1.4.Б З регістровим зсувом

Адреса в пам'яті, за якою відбувається завантаження (вивантаження), отримується зі зсуву від адреси в регістрі Rn. Зсув визначається значенням в регістрі Rm, яке може бути додатково зсунуто до 3 бітів вліво, використовуючи LSL.

Синтаксис операції:

- op{type}{cond} Rt, [Rn, Rm {, LSL #n}];

Де:

- op - LDR або STR;
- type - один з допустимих типів:
 - B: Байт без знаку, незаповнені біти заповнюються нулями;

- SB: Байт зі знаком, незаповнені біти заповнюються знаком (лише LDR);
- H: Півслова без знака, незаповнені біти заповнюються нулями;
- SH: Півслова зі знаком, незаповнені біти заповнюються знаком (лише LDR)
- —: Не вказувати тип (для адресації повного машинного слова),
- cond - умовний код;
- Rt - регістр, в який завантажуються, чи з якого вивантажуються дані;
- Rn - регістр, в якому знаходиться адреса пам'яті;
- offset - зсув (відступ) в байтах від адреси, що знаходиться в Rn (якщо не вказаний - адресою вважається вміст Rn);
- LSL #n - необов'язковий логічний зсув зі значенням n = 0..3

Приклади використання:

STR R0, [R5, R1]; Вивантажує значення в R0 в пам'ять за адресою, що дорівнює сумі R5 та R1

STR R0, [R1, R2, LSL #2]; Вивантажує значення R0 за адресою, що дорівнює сумі R1 та $4 * R2$

1.5.1.4.В Операції доступу до пам'яті з декількома регістрами

Адреси, що використовуються для доступу до пам'яті, поділяються на 4-байтні інтервали від Rn до $(Rn + 4 * (n - 1))$, де n - кількість регістрів в reglist. Доступ відбувається в порядку нумерації регістрів, тобто регістр з найнижчим порядковим номером у reglist використовує найнижчу адресу, а регістр з найвищим порядковим номером - найвищу. При цьому, якщо заданий суфікс !, то значення $(Rn + 4 * n)$ буде записано назад в Rn.

Синтаксис операції:

- op{addr_mode}{cond} Rn{!}, reglist;

Де:

- op - LDM або STM
- addr_mode - Один з наступних:
 - IA -- Інкрементувати адресу після кожного доступу (за замовчуванням)
 - DB -- Декрементувати адресу перед кожним доступом.
- cond - умовний код.

- Rn - регістр, в якому знаходиться адреса пам'яті.
- ! - суфікс запису до Rn.
- reglist - Список з 1-го чи більше регістрів, в які завантажуються (чи з яких вивантажуються) дані, що задані в дужках {}. Може бути заданий в якості ряду регістрів. Якщо задається більше, ніж 1 регістр, то вони мають бути відокремлені комами.

Приклади використання:

LDM R8, {R0, R2, R9};	Завантажує значення наступним чином: R0 := [R8], R2 := [R8 + 4], R9 := [R8 + 8] Значення в регістрі R8 лишається без змін.
STMDB R1!, {R3-R6, R11, R12};	Вивантажує значення наступним чином: [R1 - 4] := R3, [R1 - 8] := R4, [R1 - 12] := R5, [R1 - 16] := R6, [R1 - 20] := R11, [R1 - 24] := R12. Новим значенням адреси в регістрі R1 стає: R1 := R1 + 24.

1.5.1.5 Робота з командою ВКРТ

Інструкція ВКРТ приводить процесор в стан налагодження. Інструменти для налагодження можуть використовувати це, щоб дослідити стан системи, коли буде досягнута інструкція за певною адресою. Процесор ігнорується. Якщо потрібно, відладчик може використовувати його для зберігання додаткової інформації про точку розриву.

Інструкція ВКРТ затримує процесор на виконання, поки відлагоджувальний пристрій не відповість. Тобто, якщо відлагоджувача нема, код, що містить цю інструкцію, не виконується. Тому, бажано зробити програму таким чином, щоб було легко відключити всі включення цієї інструкції в коді, оскільки виконання програми буде перевірятись на реальному мікроконтролері.

Синтаксис операції:

ВКРТ #imm

Де imm - вираз, що оцінює ціле число в діапазоні 0-255.

Значення 0xAB - означає Semihosting, тобто використання можливостей зовнішнього пристрою для відладки.

Також ВКРТ має регістрові параметри, для 0xAB: перший параметр - тип операції, що потрібно зробити хосту, другий - данні для операції. Наприклад, операція 0x4 (згідно з Semihosting) – вивід рядку, параметр 2 - строка символів з 0 в кінці.

Приклади використання:

```
#define
SEMIHOSTING_SYS_WRITE0  #0x04
data: .asciz "Hello World"
      mov r0, SEMIHOSTING_SYS_WRITE0
      ldr r1, =data
      bkpt #0xAB
```

Цей фрагмент коду виведе рядок у відлагоджувальний порт(або в термінал з якого запущено Qemu).

1.5.1.6 Завантажувач

Метою даної лабораторної роботи є розробка примітивного завантажувача програми, що була написана в попередній лабораторній роботі, тобто побудова такої програми, що зможе побайтно перенести іншу програму в оперативну пам'ять та запустити її звідти. Розглянемо цю програму детальніше на прикладі:

kernel.S:

```
.syntax unified
.cpu cortex-m4
.thumb
#define A #4

// Global memory locations.
.global vtable_kernel
.global __kernel_reset__

.type vtable_kernel, %object
.type __kernel_reset__, %function

.section .interrupt_vector
vtable_kernel:
    .word __stack_start
    .word __kernel_reset__+1
    .size vtable_kernel, .-vtable_kernel

.section .rodata
    data: .asciz "kernel started!\n"
    final: .asciz "Value in register #3: "
.section .text
__kernel_reset__:
    ldr r0, =data
    bl dbgput_line

    // calculate
    mov r0, A
```

```

mov r1, #0
cmp r0, r1
ITE GE
movGE r3, r0
movLT r2, r0

ldr r0, =final
bl dbgput
mov r0, r3
bl dbgput_num

end:
b end

```

Ця програма буде відігравати роль завантажуваної програми. Зверніть увагу на те, як код з попередньої лабораторної роботи розміщений в даному файлі. Розберемо її трохи детальніше.

За допомогою директив **.section .interrupt_vector** ми задаємо векторну таблицю переривань. Першим рядком таблиці ми визначаємо початковий стан **Stack Pointer**, тобто адресу, з якої починається стек. Далі ми вказуємо для оброблення виключення **RESET** адресу програми `__kernel_reset__`, вказуючи при цьому використання **thumb** інструкцій.

Дана програма виводить в консоль відлагодження рядок `data`, після чого виконує приклад варіанту з лабораторної роботи №2, виводить вміст **r3** в консоль відлагодження та крутиться у нескінченному циклі. Для виводу результату виконання програми за варіантом використовується процедура **dbgput_num**, з принципом роботи якої можна ознайомитись в **print.s**. Для виводу вмісту певного регістра потрібно скопіювати його значення в **r0** і лише після цього викликати процедуру.

Тепер розглянемо приклад роботи завантажувача:

bootloader.S

```

.syntax unified
.cpu cortex-m4
//.fpu softvfp
.thumb

.global bootload
.section .rodata
    image: .incbin "kernel.bin"
    end_of_image:
    str_boot_start: .asciz "bootloader started"
    str_boot_end: .asciz "bootloader end"
    str_boot_indicate: .asciz "#"

```


Початок файлу такий самий, як і в **kernel.S**. Нижче ми вказуємо мітку, що буде відповідати за завантаження **kernel.S** в оперативну пам'ять, як глобальну, щоб цю мітку було видно зі **start.S**. Далі, в секції **rodata** ми створюємо рядки з **ASCII** символів, для перевірки правильності роботи програми, та мітки **image** та **end_of_image**, що містять адресу пам'яті, в якій знаходиться початок та кінець програми (**end_of_image** вказує на наступне слово після останнього слова програми).

Для прикладу ми будемо завантажувати програму послівно, за допомогою інструкцій **ldr** та **str**:

```
.section .text
bootload:
    ldr r0, =str_boot_start
    bl dbgput_line
    ldr r0, =end_of_image
    ldr r1, =image
    ldr r2, =_ram_start
```

Спочатку потрібно завантажити в регістри адреси початку та кінця програми, а також адресу початку оперативної пам'яті. Також за допомогою процедури **dbgput_line** виводиться в консоль відлагодження рядок **str_boot_start**. Ознайомитись з принципом роботи процедур можна у файлі **print.S**.

Завантаживши у відповідні регістри адреси початку та кінця програми, будемо послівно завантажувати її в оперативну пам'ять:

```
loop:
    ldr r3, [r1], #4
    str r3, [r2], #4
    cmp r0, r1
    bhi loop
```

Спочатку за допомогою інструкції **ldr** ми завантажуюмо слово програми, що знаходиться за адресою в **r1**, після чого вивантажуємо його в оперативну пам'ять за адресою в **r2**, та переходимо до наступного слова. Цикл закінчується як тільки буде завантажене останнє слово програми.

Після завершення завантаження, залишилось тільки перейти на початок оперативної пам'яті, щоб почати виконання завантаженої програми:

```
bl newline
ldr r0, =str_boot_end
bl dbgput_line
```

За допомогою процедур `newline` та `dbgput_line` виводиться текст до відлагоджувальної консолі, після чого командою `ldr` ми завантажуюмо адресу початку оперативної пам'яті:

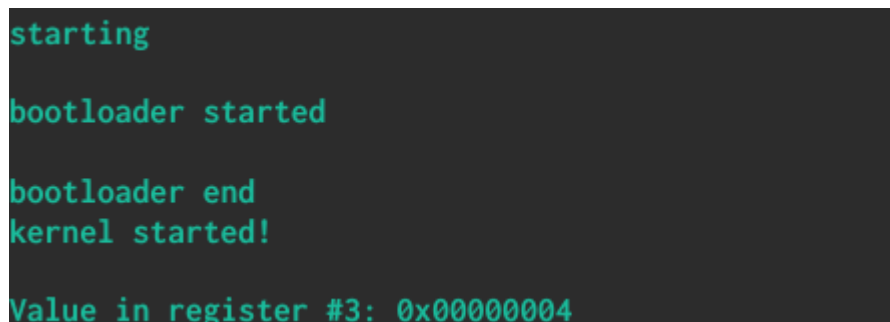
```
ldr lr, =bootload_end
add lr, #1
ldr r2, =_ram_start
```

Оскільки на початку програми знаходиться векторна таблиця, то нам потрібно перейти на наступне слово, яке містить адресу підпрограми, що відповідає за обробку виключення RESET (див. 5.1.2). В даному випадку це адреса підпрограми `__reset_lernel__`:

```
add r2, #4 // go to __reset_kernel__
ldr r0, [r2]
bx r0

bootload_end:
b bootload_end
```

Після того як ми отримали з векторної таблиці адресу `__kernel_reset__`, можна нарешті перейти до виконання завантаженої програми, що відбувається за допомогою команди `bx r0`. Також, в `lr` була збережена адреса мітки `bootload_end`, на випадок повернення з завантаженої програми.



```
starting
bootloader started
bootloader end
kernel started!
Value in register #3: 0x00000004
```

Рис.1.16 Приклад результату роботи завантажувача:

1.5.1.5 Основні директиви для лабораторної роботи

.incbin [some_file] - дозволяє зберегти в пам'ять файл в тому вигляді, в якому він є

.section – вказує на певний блок коду. Може використовуватися с директивами `.text`, `.data`, `.rodata`.

.interrupt_vector - вказує використання об'єкта в якості векторної таблиці переривань.

.text – вказує на блок виконуваного коду

.rodata – read-only data, тобто вказує на блок даних тільки для читання.

.asciz “Some string” – кодує рядок в ASCII та повертає адресу його початку.

1.5.2 Література

Курс для розуміння програм на асемблері в архітектурі STM32 Cortex-M4 [Електронний ресурс]: Режим доступу - <https://vivonomicon.com/2018/04/02/bare-metal-stm32-programming-part-1-hello-arm/>

1.5.3 Завдання

1) «Зпулити» шаблон лабораторної роботи №2 з репозиторію <https://github.com/Igor1101/CompArch2STM32/tree/master/lab2>

2) Розібратись, як мають працювати завантажувач, обробка виключення та **Semihosting**.

3) Скопіювати код розрахунку виразу з 2-ої роботи та вставити його в **kernel.s**.

Або модифікувати файл обчислення виразу з 2-ої роботи, тобто вставити в нього векторну таблицю, та модифікувати **Makefile**, щоб **<назва_вашого_файлу>.bin** файл автоматично створювався. Це не є обов'язковим, але такий підхід є заохочувальним.

4) Створити завантажувач за варіантом, який буде виконувати програму (наприклад, **kernel.bin**).

* - Для варіантів з декрементною адресацією перед початком роботи завантажувача потрібно обчислити різницю адрес початку та кінця програми, після чого додати її до адреси початку оперативної пам'яті. Результат розрахунку має виводитись в консоль.

5) Запустити програму в **gdb**, продемонструвати запуск завантажуваної програми та виведення результату в консоль.

Варіант обраховується за формулою:

$$V = N \bmod 16$$

де V - отриманий варіант, N - номер залікової книжки

1.5.4 Варіанти

Таблиця 1.7. Варіанти

№	Команди для роботи з пам'яттю	Інкремент/Декремент регістру адреси	Вид зсуву	Кількість байт для зсуву
0	LDR, STR	інкремент	числовий	4
1	LDR, STR	інкремент	регістровий	4
2	LDR, STR	декремент*	числовий	4
3	LDR, STR	декремент*	регістровий	4
4	LDRB, STRB	інкремент	числовий	1
5	LDRB, STRB	інкремент	регістровий	1
6	LDRB, STRB	декремент*	числовий	1
7	LDRB, STRB	декремент*	регістровий	1
8	LDRH, STRH	інкремент	числовий	2
9	LDRH, STRH	інкремент	регістровий	2
10	LDRH, STRH	декремент*	числовий	2
11	LDRH, STRH	декремент*	регістровий	2
12	LDM, STM	інкремент	регістровий	8 (2 регістра по 4)
13	LDM, STM	інкремент	регістровий	12 (3 регістра по 4)
14	LDM, STM	декремент*	регістровий	8 (2 регістра по 4)
15	LDM, STM	декремент*	регістровий	12 (3 регістра по 4)

1.6. Лабораторна робота №1.4

Тема: Драйвер складного зовнішнього пристрою, використання периферії МК та зовнішньої периферії.

Мета: Навчитися використовувати периферію ядра МК та периферію МК. Створення програми для контролю дисплею

1.6.1 Скорочені теоретичні відомості

Виконання даної лабораторної роботи передбачає безпосереднє використання даного embedded kit та невикористання будь-яких зовнішніх бібліотек.

1.6.1.1 Загальні відомості про WH1602B

Як вже було зазначено вище, даний LCD дисплей встановлений на embedded kit **STM32f407g**. Datasheet на даний дисплей можна знайти [за посиланням](#). Для успішного виконання лабораторної роботи необхідно мати відомості про піни даного дисплею:

Таблиця 1.8. Піни WH1602B

1	Vss	GND	9	DB2	H/L Data Bus
2	Vdd	+3V or +5V	10	DB3	H/L Data Bus
3	Vo	Contrast adjustment	11	DB4	H/L Data Bus
4	RS	H/L Register Select signal	12	DB5	H/L Data Bus
5	R/W	H/L Read / Write signal	13	DB6	H/L Data Bus
6	E	H→L Enable signal	14	DB7	H/L Data Bus
7	DB0	H/L Data Bus	15	A/Vee	Enabling/Disabling LED lightning
8	DB1	H/L Data Bus	16	K	Power supply for B/L

1.6.1.2 General-purpose I/Os (GPIO)

Кожен порт вводу-виводу загального призначення має цілий набір 32-бітних регістрів, призначений для різнопланового керування I/O пристроєм. З них для виконання лабораторної роботи потрібні:

- регістр конфігурації **GPIOx_MODER**;
- регістр даних **GPIOx_ODR**;
- регістр конфігурації **GPIOx_PUPDR**;
- регістр встановлення/скидання (set/reset) **GPIOx_BSRR**;
- регістр вибору альтернативної функції **GPIOx_AFRH**;
- регістр вибору альтернативної функції **GPIOx_AFRL**;

де *x* - код, який відповідає певному I/O пристрою. Повну інформацію по регістрам, їх призначенню та адресам можна знайти [за посиланням \(розділ 8.4, GPIO registers\)](#).

З урахуванням специфічних апаратних характеристик кожного I/O порту, зазначеного в таблиці даних, кожен біт I/O портів загального призначення (GPIO) може бути індивідуально налаштований програмним забезпеченням. Кожен біт портів вводу-виводу вільно програмується, проте значення що присвоюються регістрам портів вводу-виводу мають бути лише 32-бітними, 16-бітними чи 8-бітними (стандартне слово, півслово та байт відповідно). Регістр **GPIOx_BSRR** призначений для встановлення дозволу побітового зчитування та модифікації доступу до будь-якого з регістрів **GPIO**.

Піни вводу-виводу мікроконтролера підключаються до периферійних пристроїв плати через мультиплексор, який дозволяє підключити одночасно лише одну альтернативну функцію (AF). Таким чином, це дозволяє уникнути конфлікт між периферійними пристроями, які спільно використовують спільний I/O пін. Кожен I/O пін має мультиплексор з 16 альтернативними входами функції (AF0 до AF15), які можна налаштувати через регістри **GPIOx_AFRL** (для пінів 0-7) та регістри **GPIOx_AFRH** (для пінів 8-15).

Біти регістру **GPIOx_PUPDR** керують режимами роботи транзисторів I/O пристроїв. За допомогою нього можна виставити піни вводу у режим **pull-up** (на транзистори подається ненульова напруга) чи у режим **pull-down** (на транзистори нульова напруга).

[Докладніше \(англомовна документація\) \(розділ 8, GPIO\).](#)

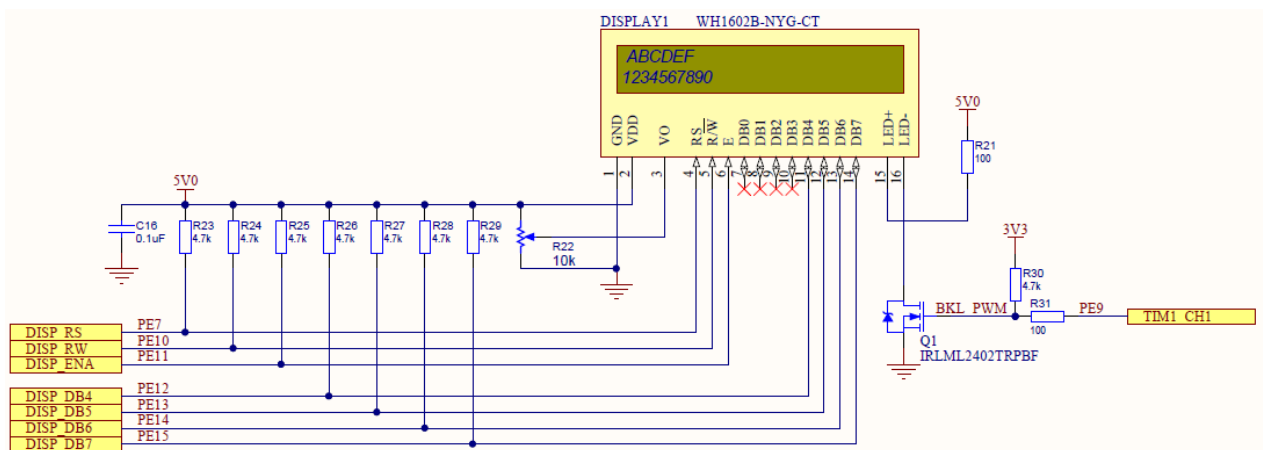


Рисунок 1.17. Схема підключення WH1602B до STM32f407g. [У кращій якості.](#)

1.6.1.3 Reset and clock control (RCC)

Для правильного налаштування I/O пристроїв під час виконання лабораторної роботи необхідно ознайомитись з керуванням скидання та тактуванням (RCC). За замовчуванням тактування периферійних пристроїв виключене, для його налаштування необхідний доступ до регістру **RCC AHB1 peripheral clock register (RCC_AHB1ENR)**.

[Докладніше \(англомовна документація\) \(розділ 6.3, RCC registers\).](#)

1.6.2 Порядок виконання

1.6.2.1 Етапи роботи

За основу необхідно взяти [Лабораторну роботу №2](#) та реалізувати вивід результатів на дисплей. Для цього необхідно:

1. Ініціалізувати порт
2. Ініціалізувати дисплей
3. Зберегти результати лабораторної роботи у вигляді масиву ASCII-символів
4. Вивести даний масив на дисплей.

1.6.2.2 Створення проекту

Для виконання лабораторної роботи скористаємся CubeIDE. Ця IDE значно зручніша в плані налаштування та сбороки проекту, а також спрощує відлагодження.

1. Створимо проект в середовищі STM32CubeIDE:

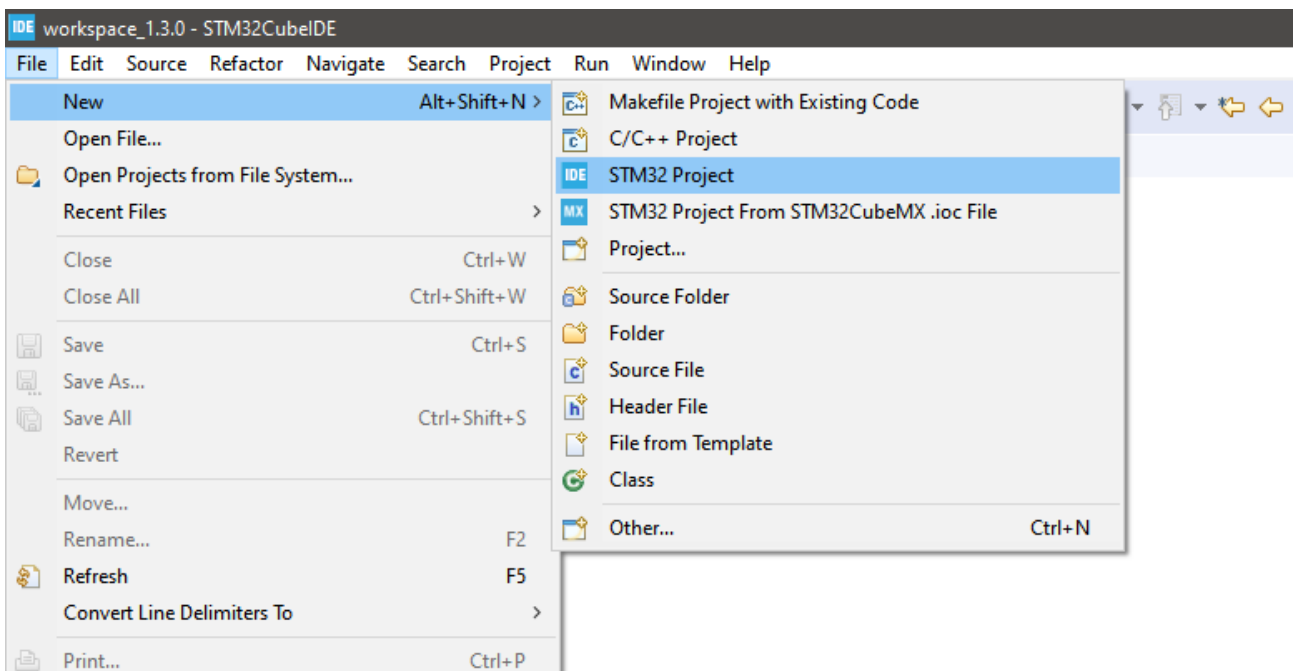


Рисунок 1.18 Створення проекту (перший етап)

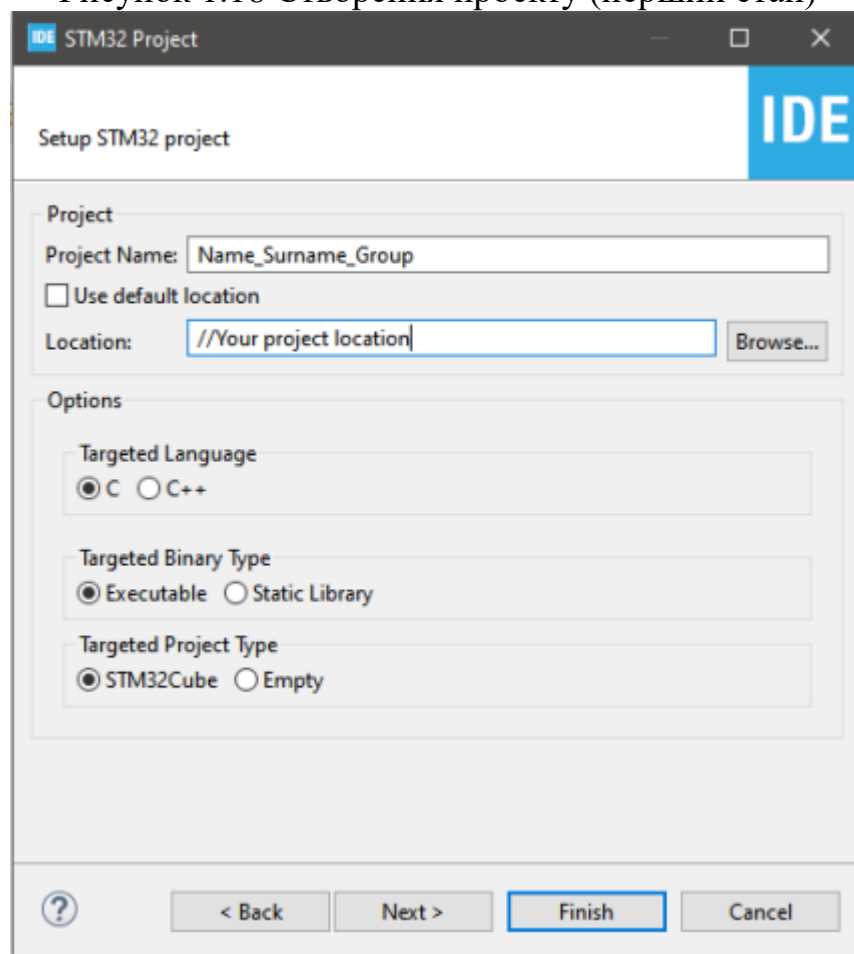


Рисунок 1.19 Створення проекту (другий етап)

Додаткові налаштування в **Device configuration tool** не виконуються.

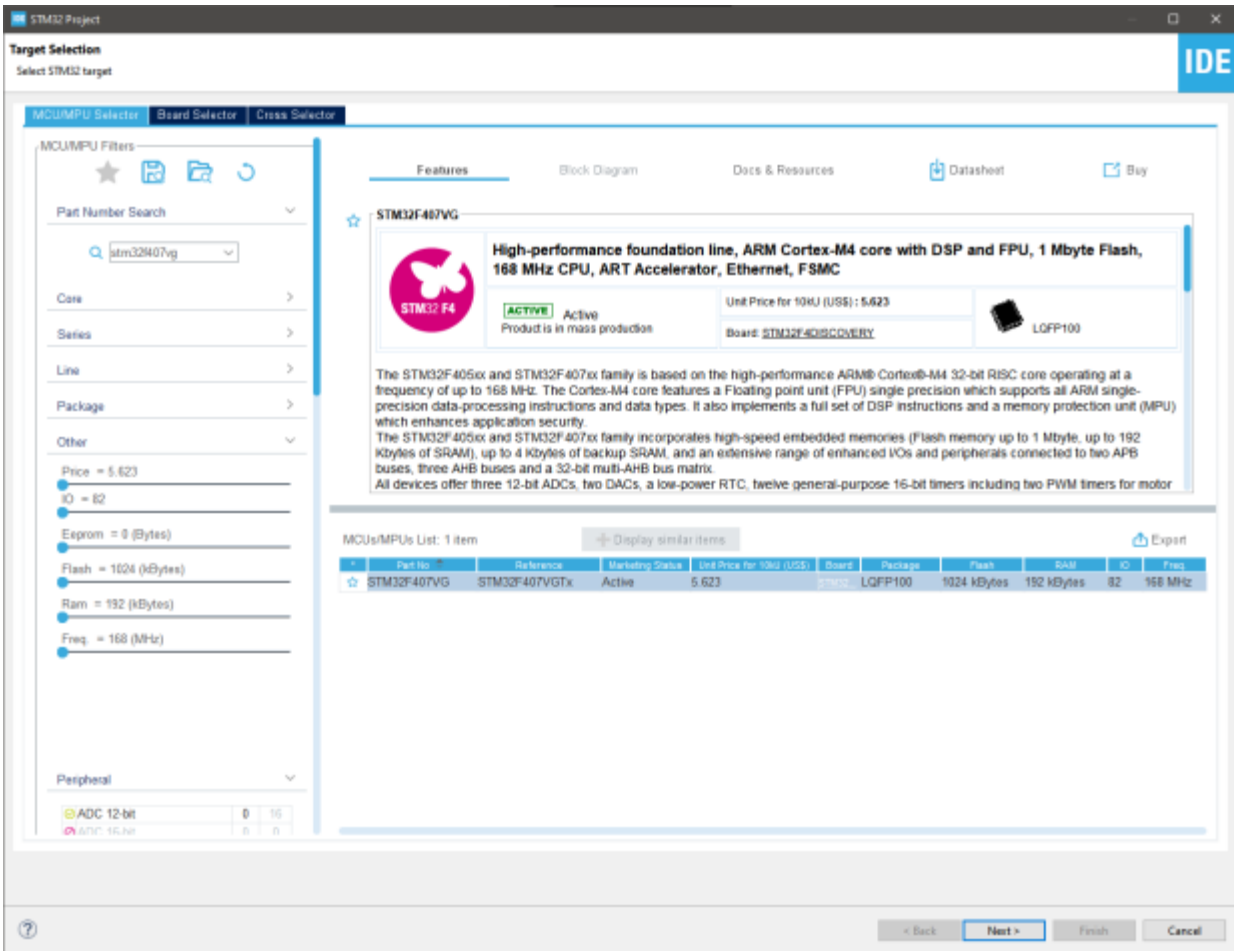


Рисунок 1.20 Створення проекту (третій етап)

2. Перевіримо новостворені файли:

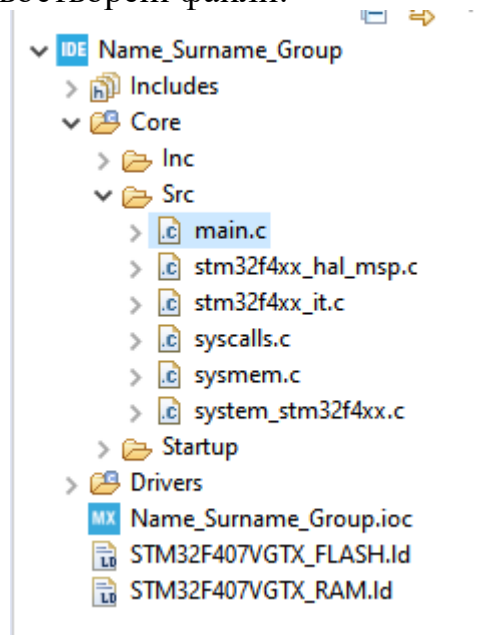


Рисунок 1.21 Файли, створені IDE

Відкриємо файл **main.c** і побачимо, що IDE сама згенерувала код та додала імпорт необхідних бібліотек. Для коректної роботи код необхідно писати в секціях, що виділені спеціальним коментарем:

```
/* Private user code -----*/  
/* USER CODE BEGIN 0 */  
  
/* USER CODE END 0 */
```

Рисунок 1.22 Секція користувацького коду

1.6.2.3 Початкові налаштування

В **Доповненнях** прочитайте розділ про встановлення додатків в eclipse-срр та встановіть їх. Використання самої платформи **Eclipse** не є обов'язковим але бажаним, використання C або C++ є обов'язковим. Виконувати можна використовуючи лише **CMSIS** та додатки до нього від **ST**, що підключаються заголовним файлом:

```
#include <stm32f407xx.h>
```

1.6.2.4 Робота з периферією

Всі операції проводяться бітовими масками, наприклад:

```
RCC->AHB3ENR |= RCC_AHB3ENR_FSMCEN;
```

виставить тільки потрібні біти регістру **AHB3ENR** під маскою **RCC_AHB3ENR_FSMCEN**, інші залишаться без змін. Цією операцію (зі вказанням потрібного регістру) можна включити тактування будь-якого периферійного пристрою. У нашому випадку, [згідно схеми](#), дисплею призначені піни E.

1.6.2.5 Ініціалізація дисплея

Дисплей працює у 8-бітному режимі, водночас до нього підключено лише 4 піни 8-бітної шини, [що зображено на схемі](#). Контролер дисплея зчитує з шини байт лише тоді, коли на виводі E з'являється перехід 1==>0, тобто пін E використовується для тактування. Біт **R\W** використовується для вибору режиму роботи дисплею - **Read** (зчитування) чи **Write** (запис). Читання з дисплею не використовується у даній лабораторній роботі, отже можна його завжди тримати в 0. Біт **RS** використовується для переключення режимів зчитування наступної послідовності сигналів. Режимів 2 - сприймати як команду, чи як код символу, що потрібно вивести.

Але на платі нам недоступна 8-бітна шина, а лише її 4 бітна верхня частина. Контролер дисплею дозволяє працювати з 4-бітною шиною. Команда/дані розподіляються, і спочатку передається старші 4 біти, потім молодші. Виникає потреба в передачі контроллеру інформацію режиму роботи, використовуючи лише 4 біти.

Послідовність ініціалізації дисплея для роботи в 4-бітному режимі (перші 4 команди - старші половинки повноцінних команд):

1. передати лише старші 4 біти FUNCTION SET з встановленим бітом DL.
2. передати лише старші 4 біти FUNCTION SET з встановленим бітом DL.
3. передати лише старші 4 біти FUNCTION SET з встановленим бітом DL.
4. передати лише старші 4 біти FUNCTION SET з невстановленими бітами.
5. передати FUNCTION SET з встановленим N.
6. передати DISPLAY CONTROL з включеними D, C, B (ця команда має викликати блимання курсору на екрані).
7. передати команду ENTRY_MODE_SET з встановленим бітом ID
8. передати команду CURSOR DISPLAY SHIFT SC
9. передати команду CLEAR DISPLAY
10. передати команду RETURN HOME

Після цього можна повноцінно використовувати всі команди в 4-бітному режимі.

1.6.2.6 Конфігурація портів

Після ініціалізації дисплею необхідно налаштувати його порти.

GPIOx-> - CMSIS

Де x - фіксований вказівник на будь-які регістри GPIO (не забуваємо, що у нашому випадку це E).

За допомогою регістру **GPIOx_MODER** ми можемо визначити, як сприймати інформацію, що подається на кожен пін I/O пристрою. На кожен пін відводиться по 2 біти, отже усього доступно 4 режиму конфігурації:

Таблиця 1.9. Режими конфігурації вводу-виводу пристроїв

00	Input (reset state)
01	General purpose output mode
10	Alternate function mode
11	Analog mode

У нашому випадку ми маємо виставити відповідні значення регістру в режим 01 - General purpose output mode, для виводу інформації на дисплей.

Окрім цього, оскільки ми виводимо інформацію на дисплей, нам необхідно виставити його піни вводу у нульове значення (тобто зняти з них напругу) Цього можна домогтися спеціальною маскою **GPIO_PUPDR_PUPDRx**, де x – номер потрібного піну.

1.6.2.7 Реалізація керуючих функцій.

Заради спрощення роботи з дисплеєм необхідно створити функції, що здатні змінювати значення окремих пінів, що підключені до дисплея.

Символи для виводу передаються у регістри BSRR [у вигляді ASCII-коду](#), звідки йдуть на дисплей.

Для змінювання значення використовуються регістри **BSRR** та **ODR**. Вони є взаємопов'язаними (завжди мають однакові значення), щоправда відрізняються за функціоналом.

BSRR дозволяє незалежно змінювати кожний окремий біт (так зване atomic set/reset, “атомарні операції”), водночас для **BSRR** недоступна операція читання, отже реалізація деяких функцій, що передбачають зміну відносно наявного стану регістру (наприклад toggle), неможлива. Наприклад, команда:

```
GPIOE->BSRR = GPIO_BSRR_BS_11 | GPIO_BSRR_BR_10;
```

де:

GPIO_BSRR_BS_x - Set, встановити (змінити значення на 1) порт x
GPIO_BSRR_BR_x - Reset, обнулити порт x

встановить 1 у порт GPIOE11 і 0 у порт GPIOE10.

ODR, на противагу йому, дозволяє виконати читання, але зміна значення регістру можлива за умови його повного перезапису.

Також не буде зайвим слідкувати за часом операції, є можливість додати цикл затримки, наприклад:

```
for(volatile int a=0; a<30000; a++);
```

або використовувати SysTick. Зручніше всього поставити затримку максимально довгою (до секунди), а потім знижувати за потреби.

1.6.2.8 Збірка проекту та його відлагодження

Після написання коду зібрати готовий проект можна за допомогою відповідної опції на вкладці **Project**:

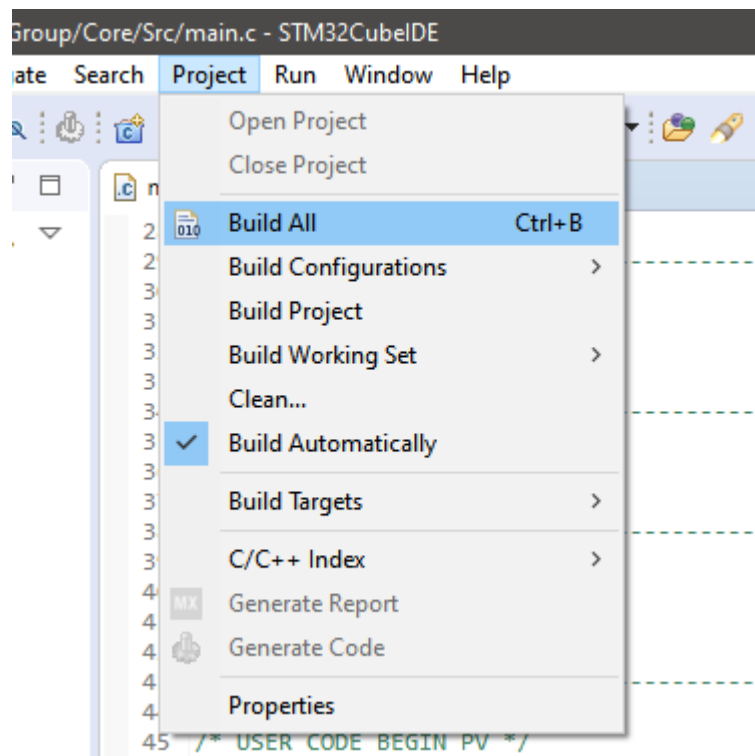


Рисунок 1.23 Зборка проекту

Після зборки проекту стає доступним його відагодження. Для цього скористаємося опцією **Debug**. За допомогою панелі керування відлагоджувача можна виконати покроковий прохід по коду для перевірки потрібних значень у керуючих регістрах потрібного I/O пристрою. Обираємо в правій частині відлагоджувача вкладку **SFRs**. Всі задіяні в проекті регістри (у нашому випадку це RCC та GPIOx) скомпоновані по групах:

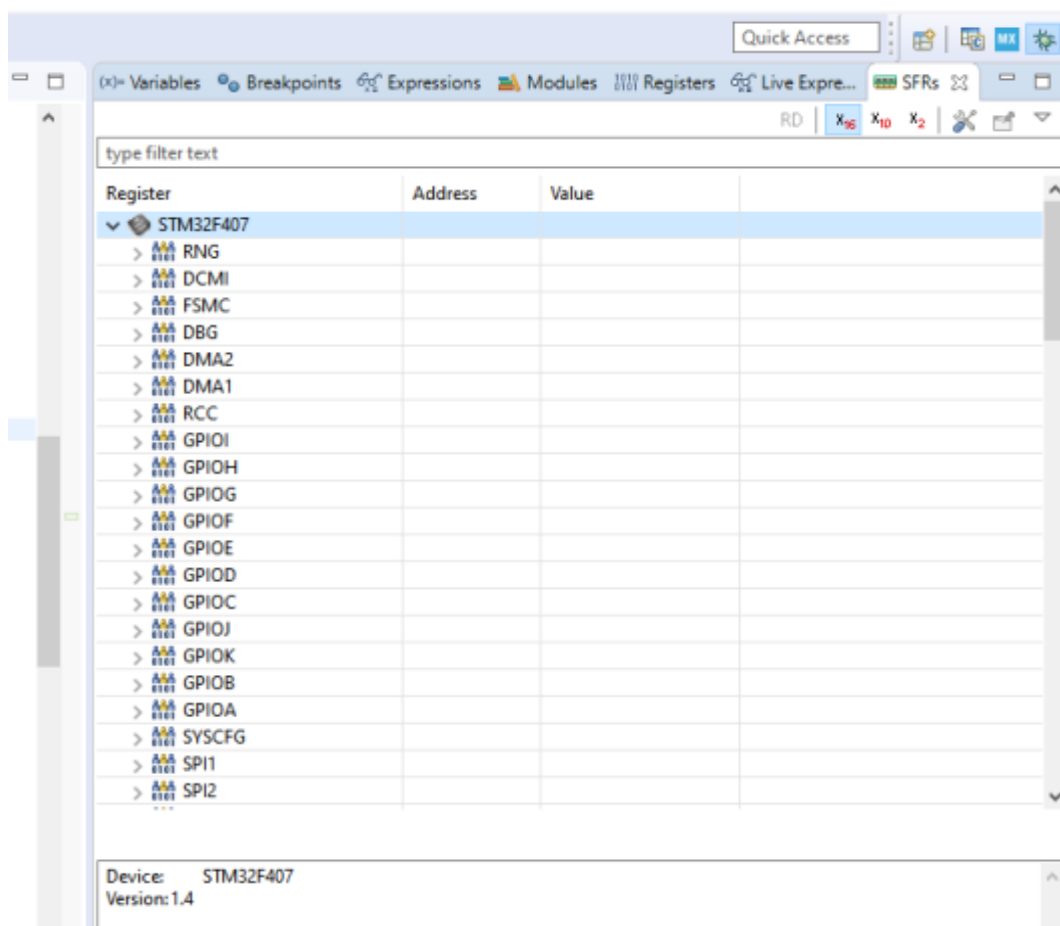


Рисунок 1.24 Вкладка SFRs

>	GPIOI			
>	GPIOH			
>	GPIOG			
>	GPIOF			
▼	GPIOE			
>	MODER	0x40021000	0x0	
>	OTYPER	0x40021004	0x0	
>	OSPEEDR	0x40021008	0x0	
>	PUPDR	0x4002100c	0x0	
>	IDR	0x40021010	0x0	
>	ODR	0x40021014	0x0	
>	BSRR	0x40021018		
>	LCKR	0x4002101c	0x0	
>	AFRL	0x40021020	0x0	
>	AFRH	0x40021024	0x0	
>	GPIOD			
>	GPIOC			
>	GPIOJ			

Dec: 0
Hex: 0x0
Bin: 0

Рисунок 1.25 Перевірка необхідних значень регістрів

Розділ 2. Робота з платою Beagle Bone Black

2.1. Лабораторна робота №2.1

Тема: Збірка та запуск найпростіших модулів ядра ОС Linux за допомогою BeagleBone Black.

Мета: Ознайомитись з основними елементами модуля ОС Linux, навчитись створювати найпростіші модулі (драйвери) за допомогою BeagleBone Black.

2.1.1 Скорочені теоретичні відомості

2.1.1.1 Основна теоретична інформація

Дана лабораторна робота є вступом до розробки власних модулів (драйверів) операційної системи Linux. Перед тим як розпочати розгляд команд та процесу збірки модуля, необхідно зазначити деякі важливі відмінності програмування модулів ядра від програмування звичних програм.

Перш за все, існує суттєва різниця в циклі життя програми та модуля ядра Linux, в той час як програма від початку запуску виконує одне й те саме завдання, модуль певним чином “реєструє” себе в ядрі операційної системи з ціллю багаторазового виконання наступних запитів. Головна відмінність модуля полягає в наявності функції ініціалізації, що готує його основні функції для подальшого використання. Таким чином модуль дає про себе знати ядру операційної системи. Коли модуль більше не є потрібним виконується “функція виходу”, таким чином модуль сповіщає ядро про свою відсутність. Такий підхід до програмування є дуже схожим на подійно-орієнтоване програмування.

Також, важливою є особливість модулів, що полягає в їх зв’язаності тільки з ядром операційної системи. Таким чином функції, що можуть бути використані в будь-якому модулі, експортуються лише ядром.

Ще однією суттєвою відмінністю в програмуванні модулів є те, що навіть найпростіші модулі виконуються в конкурентному середовищі, тобто ситуація при якій декілька процесів одночасно звертаються до модуля є цілком нормальною. Таким чином код драйвера для ядра ОС Linux має бути спроможним виконуватись в більш ніж 1 контексті одночасно, тому при проектуванні програми слід звертати увагу проблеми, що часто виникають при паралельному програмуванні (race condition).

Розглянемо програмування модуля на прикладі (Приклад #1).
Даний модуль визначає 2 функції:

- `hello_init` - Виконується при завантаженні модуля до ядра
- `hello_exit` - Виконується при видаленні модуля з ядра

Поведінку цих функцій задають макроси `module_init` та `module_exit`.

Функція `printk` - має той самий принцип що й `printf`, але визначена в ядрі ОС, при цьому `KERN_EMERG` задає пріоритет повідомлення.

Також в файлі присутні наступні необов'язкові макроси:

- `MODULE_AUTHOR` - визначає автора модуля
- `MODULE_DESCRIPTION` - задає короткий опис роботи модуля
- `MODULE_LICENSE` - задає ліцензію під якою може поширюватись код ("Dual BSD/GPL")

Таким чином, був створений простий модуль, що виводить в консоль "Hello, world!" при завантаженні до ядра ОС Linux. Однак, перш ніж почати його завантаження, потрібно створити `Makefile`, що буде відповідати за збірку модуля в файл необхідного формату (`.ko`). Приклад даного файлу знаходиться також в Прикладі #1.

Потрібно зазначити, що процес збірки модулів ядра також відрізняється від звичайного процесу збірки за допомогою `make`. Головна відмінність полягає в тому, що замість утиліти `make` процесом займається система збірки ядра, таким чином можна полегшити розробку модуля. Створити `Makefile` досить просто, якщо модулі присутній лише один файл, то достатньо лише даного рядка:

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

Де `name` це назва об'єктного файлу, з якого має бути зібраний модуль.

Якщо ж модуль має бути зібраний з більш ніж одного файлу, наприклад `file1.c` та `file2.c`, то скрипт мав би наступний вигляд:

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

Як вже було сказано, процесом збірки займається система збірки ядра, тому щоб `Makefile` як в Приклад #1 працював, він має бути використаний в контексті системи збірки ядра. Така команда виглядатиме наступним чином:

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

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

Після того як модуль був зібраний, можна нарешті його завантажити до ядра ОС. Для цього використовуються команди `insmod` та `modprobe`:

- `insmod` - завантажує код та дані модуля до ядра ОС, шукає всі незнайомі символи в символній таблиці ядра. Також приймає параметри, що можуть бути використані для `load-time` налаштування модуля, детальніше в `Module Parameters` в [1]
- `modprobe` - так само як `insmod` завантажує модуль до ядра ОС, але шукає всі незнайомі символи в модулі в інших модулях. Коли такі

модулі знайдені, вони теж імпортуються до ядра ОС
Також використовуються наступні команди:

- *rmmod* - видаляє модуль з ядра ОС
- *lsmod* - виводить список модулів що зараз використовуються ядром ОС.

Більш детально ознайомитись з командами та їх параметрами можна в [\[7.2.1\]](#).

2.1.1.2 Приклади

Для всіх прикладів *hello.c* однаковий, змінюється тільки Makefile
В Makefile потрібно замінити ``uname -r`` на вивід цієї команди в терміналі.

Приклад #1:

hello.c

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

MODULE_AUTHOR("Serhii Popovych <serhii.popovych@globallogic.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);
```

Makefile

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m := hello.o
else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
$(MAKE) -C $(KDIR) M=$$PWD
clean:
$(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

Приклад #2:

Kbuild

```
# kbuild part of makefile
obj-m := hello.o
```

Makefile

```
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build
default:
    $(MAKE) -C $(KDIR) M=$$PWD
clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean
```

Приклад #3:

Kbuild як в прикладі #2

Makefile

```
ifneq ($(KERNELRELEASE),)
include Kbuild
else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD
clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

2.1.2 Література

1) Building and Running modules, Linux Device Drivers, Second Edition by Jonathan Corbet, Alessandro Rubini [Електронний ресурс]: Режим доступу -

https://drive.google.com/file/d/1Sns7rNavGOuMqYMsVwI_1fWMDLJU46lT/view

2) BeagleBone Black: Platform Bring Up with Upstream Components

[Електронний ресурс]: Режим доступу -

<https://drive.google.com/file/d/1oG36av0X-D7m-s5aaWJyIwX-NPG0IyJa/view>

2.1.3 Примітки до роботи

- 1) Ядро Linux вже має бути зібране в лабораторній роботі №2 на курсі “Архітектура комп’ютерів-2. Процесори”, щоб були готові всі необхідні для компіляції модулів файли (*BBB* до пункту 3.1 включно).
- 2) Перед виконанням `make`, до команд `export`, які використовувалися для збирання ядра, слід додати: `export KDIR=<шлях_до_каталогу_ядра>` (наприклад, `$HOME/repos/linux-stable`)
- 3) Для довідки:
 - A. Каталог `$KDIR/Documentation/kbuild`, зокрема `modules.txt`
 - B. `$KDIR/Documentation/process/coding-style.rst`
 - C. Параметри модуля описані в додатку *Building and Running Modules* у *Module Parameters* (сторінка 35)

2.1.4 Завдання

В цій роботі є два варіанта завдань - **Basic** та **Advanced**. Вони відрізняються складністю та максимальним балом за роботу. Студент вибирає один із варіантів.

2.1.4.1 Завдання Basic

1. Обрати один із прикладів, наведених в теоретичних відомостях для подальшої роботи, зібрати і виконати `insmod` та `rmmmod` на платі BBB (або емуляторі QEMU).
2. Модифікувати модуль, додавши до нього параметр типу `uint`, який визначає, скільки разів має бути надрукований рядок "Hello, world!"

2.1 Значення параметра за умовчанням 1.

2.2 Якщо значення параметра 0 або знаходиться між 5 і 10, надрукувати попередження і продовжити роботу.

2.3 Якщо значення параметра більше 10, то функція ініціалізації повинна надрукувати повідомлення про помилку і повернути значення `-EINVAL` (модуль не має завантажити взагалі).

3. Додати опис параметра. Подивитися його командою `modinfo`.

4. Виконати `insmod/rmmmod` модуля на платі BBB без параметра у командному рядку, зі значеннями параметра 0, довільним між 1 і 10, довільним більше 10.

Після котрогось із `insmod` подивитися значення встановленого параметра (каталог `/sys/module/hello/parameters`)

2.1.4.2 Завдання Advanced:

1. Розділити проект на два модулі, `hello1` та `hello2`. Модуль `hello1` повинен експортувати функцію `print_hello()`, яку використовуватиме модуль `hello2` (параметр кількості викликів функції перенести у модуль `hello2`).

2. Заголовковий файл `hello1.h`, який використовуватимуть обидва модулі, винести у підкаталог `inc`, який додати у систему збирання так, щоб файли `*.c` могли використовувати директиву `#include "hello1.h"` лише з іменем файлу, без шляху (див. `ccflags-y`).

3. Замінити `printk` на відповідні ситуації `pr_err`, `pr_warn`, `pr_info` (для друку привітання використати `pr_info`)

4. Виконати `insmod hello1.ko`, потім `insmod hello2.ko` з такими значеннями параметра, щоб отримати всі можливі повідомлення і знайти їх (може знадобитися `dmesg`, `grep`).

5. Спробувати завантажити `hello2.ko`, не завантажуючи `hello1.ko`, пояснити результат.

2.1.5 Оформлення звіту

Звіт має містити:

1. Лістинг коду (для завдання Basic) або посилання на `github/gitlab` репозиторій, що містить файли з вихідним кодом та `Makefile` (для завдання Advanced)
2. Скріншоти виконання `insmod/rmmod` з різними параметрами команди `modinfo`
3. Вивід значення встановленого параметра.

2.2. Лабораторна робота №2.2

Тема: Зв'язні списки та розподіл пам'яті в ядрі Linux.

Мета: Ознайомитись із зв'язними списками ядра Linux, навчитись працювати із виділенням та звільненням пам'яті.

2.2.1 Теоретичні відомості

2.2.1.1 Зв'язні списки ядра Linux

Зв'язні списки в ядрі Linux

- Найпоширеніші, прості та зручні структури даних
- Зазвичай розробники вільні у виборі реалізації. Вони можуть або використовувати власну структуру даних та перелік примітивів для маніпулювання (ітератори, помічники з введення / видалення тощо), але спочатку краще подивитися ядро Linux "Стандартне" зв'язане виконання списку, якщо воно відповідає потребам завдання.

- Списки також можна подвійно зв'язати, де кожен вузол має вказівник на попередній. Також список може закінчуватися NULL або вказувати на ту саму заглушку.

- У найпростішому випадку окремо спрямований список може виглядати наступним чином:

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

- Стандартні зв'язні списки ядра Linux, реалізовані у вигляді подвійних круглих списків.

- У звичайних випадках вони використовують заглушку для посилання на весь список, а не просто вказівник на перший (останній) елемент списку. Ця властивість використовується більшістю списку примітивних маніпуляцій, а також ітераторів.

- Ці списки є загальними, вбудовуються в структуру даних клієнтів, перетворюючи ці структури в пов'язаному списку.

- Тип пов'язаного списку настільки поширений у ядрі, тому він оголошується в `<linux / types.h>`.

- Примітиви маніпуляції із загальним пов'язаним списком оголошуються в `<linux / list.h>`.

- У примітці `<linux / rculist.h>` є список варіантів маніпуляцій із списком варіантів RCU.

- Як очікується, складність $O(N)$ стосується переходу списків та $O(1)$ для маніпулювання списком

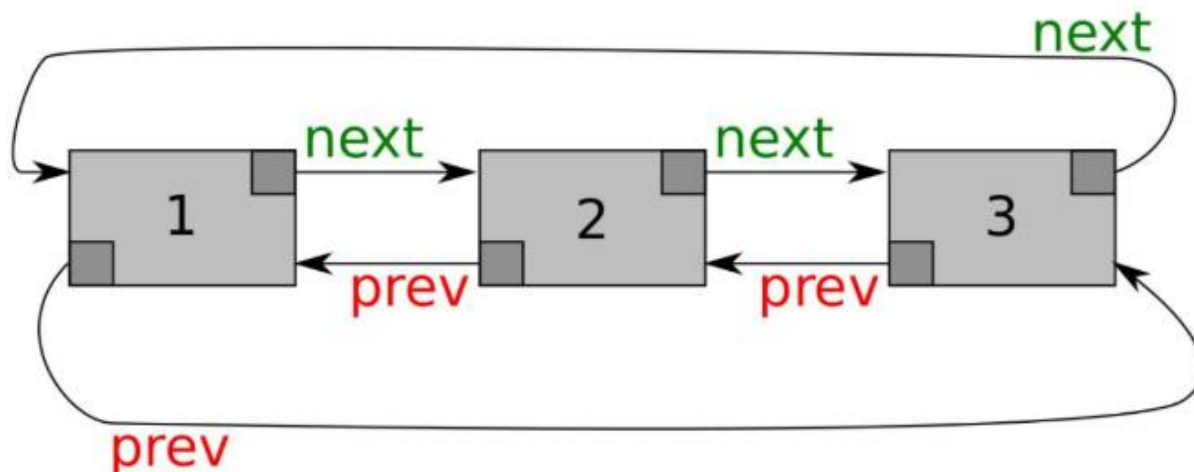


Рис.2.1 Зв'язний список

- Ось як визначити заголовок зв'язного списку ядра Linux

```
/* if list is global this can be used to define stub list
head */
static LIST_HEAD(my_list_head);
/* which is in turn equivalent to */
static struct list_head my_list_head =
LIST_HEAD_INIT(my_list_head);
static struct my_data *cool_init_function(void)
{
/* here @os some other struct
* containing stub list head is
* allocated at runtime (e.g. in heap)*/
INIT_LIST_HEAD(&os->my_list_head);
}
```

- Додамо кілька статичних даних до статичного, загального списку загальнолюдських списків

```
struct LIST_HEAD(my_list_head);
/* static array of lists! */
static struct my_list ml[] = {
{ .canary = 0xfade0f01, },
{ .canary = 0xfade0f02, },
{ .canary = 0xfade0f03, },
};
/* somewhere in .text */
for (i = 0; i < ARRAY_SIZE(ml); i++) {
list_add(&ml[i].list_node,
&my_list_head);
}
```

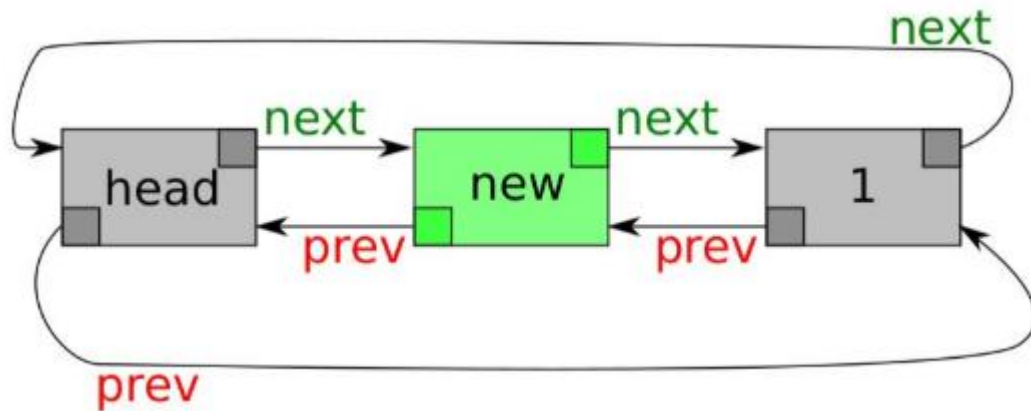


Рис.2.2 Зв'язний список

- Тепер, щоб перевірити, чи список порожній, можна використовувати `list_empty()`

```
static LIST_HEAD(my_list_head);
static int init_my_data_list(unsigned int nr_items)
{
    /* Calling this function with non-empty
     * list head isn't supported for now.
     */
    BUG_ON(!list_empty(&my_list_head));
    /* rest of the code goes here */
}
```

- Щоб з'єднати два списки, можна використовувати `list_splice()` або `list_splice_tail()`

```
static struct my_data *odd_canary(unsigned int nr_items);
static struct my_data *even_canary(unsigned int nr_items);
static LIST_HEAD(my_list_head);
static LIST_HEAD(my_list_even), LIST_HEAD(my_list_odd);
/* somewhere in the .text */
even_canary(&my_list_even, 10);
odd_canary(&my_list_odd, 10);
list_replace_init(&my_list_even, &my_list_head);
list_splice_tail_init(&my_list_odd, &my_list_head);
```

2.2.1.2 Розподіл пам'яті

Функції та символи, пов'язані з розподілом пам'яті:

1. `#include <linux/slab.h>`
`void *kmalloc(size_t size, int flags);`
`void kfree(void *obj);`
 Найчастіше використовуваний інтерфейс для розподілу пам'яті.
2. `#include <linux/mm.h>`
`GFP_USER`
`GFP_KERNEL`
`GFP_NOFS`

GFP_NOIO
GFP_ATOMIC

Прапори, які керують виконанням розподілу пам'яті, від найменшого обмеження до найбільшого. Пріоритети GFP_USER та GFP_KERNEL дозволяють увімкнути поточний процес, щоб задовольнити запит. GFP_NOFS та GFP_NOIO відключають файлову систему операцій та всіх операцій вводу / виводу відповідно..

- 3. __GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY

Ці прапори змінюють поведінку ядра під час розподілу пам'яті.

- 4. #include <linux/malloc.h>

```
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset,  
unsigned long flags, constructor( ), destructor( ));  
int kmem_cache_destroy(kmem_cache_t *cache);
```

Створити та знищити кеш-таблицю. Кеш може використовуватися для виділення декількох об'єктів однакового розміру.

- 5. SLAB_NO_REAP
SLAB_HWCACHE_ALIGN

SLAB_CACHE_DMA - Прапори, які можна вказати під час створення кешу.

- 6. SLAB_STOR_ATOMIC

SLAB_STOR_CONSTRUCTOR - Прапори, які алокатор може передавати конструктору та функціям деструктора.

- 7. void *kmem_cache_alloc(kmem_cache_t *cache, int flags);

void kmem_cache_free(kmem_cache_t *cache, const void *obj); - Виділити та випустити з кеша один об'єкт.

- 8. /proc/slabinfo - Віртуальний файл, що містить статистику використання кеш-пам'яті.

- 9. #include <linux/mempool.h>

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,  
mempool_free_t *free_fn, void *data);
```

void mempool_destroy(mempool_t *pool); - Функції для створення пулів пам'яті, які намагаються уникати відмов, зберігаючи "список надзвичайних ситуацій" виділених предметів.

- 10. void *mempool_alloc(mempool_t *pool, int gfp_mask);

`void mempool_free(void *element, mempool_t *pool);` - Функції для виділення елементів із (і повернення їх до) пулів пам'яті.

11. `unsigned long get_zeroed_page(int flags);`

`unsigned long __get_free_page(int flags);`

`unsigned long __get_free_pages(int flags, unsigned long order);` - Функції розподілу, орієнтовані на сторінку. `get_zeroed_page` повертає одиничний, нульово-заповнену сторінку. Усі інші версії виклику не ініціалізують вміст

12. `void free_page(unsigned long addr);`

`void free_pages(unsigned long addr, unsigned long order);` - Функції, які звільняють виділені сторінки.

13. `#include <linux/vmalloc.h>`

`void * vmalloc(unsigned long size);`

`void vfree(void * addr);`

`#include <asm/io.h>`

`void * ioremap(unsigned long offset, unsigned long size);`

`void iounmap(void *addr);` - Функції, що виділяють або звільняють суміжний віртуальний адресний простір. `ioremap` отримує доступ до фізичної пам'яті за допомогою віртуальних адрес, тоді як `vmalloc` виділяє вільні сторінки. Регіони, відображені за допомогою `ioremap`, звільняються за допомогою `iounmap`, а сторінки отримані з `vmalloc` вивільняються з `vfree`.

14. `#include <linux/percpu.h>`

`DEFINE_PER_CPU(type, name);`

`DECLARE_PER_CPU(type, name);` - Макроси, що визначають та оголошують змінні на CPU.

15. `int get_cpu();`

`void put_cpu();`

`per_cpu_ptr(void *variable, int cpu_id)` - `get_cpu` отримує посилання на поточний процесор і повертає ідентифікаційний номер процесора; `put_cpu` повертає це посилання. Щоб отримати доступ до динамічно розподіленої змінної CPU, використовуйте `per_cpu_ptr` з ідентифікатором процесора.

2.2.2 Література

1) Allocating Memory, Chapter 8 [Електронний ресурс]: Режим доступу -

<https://drive.google.com/file/d/1jgUxWBrthCfzsWPTevuKqKptT9UtLpDe/view?usp=sharing>

2) Описання роботи звязного списку [Електронний ресурс]: Режим доступу -

https://drive.google.com/file/d/1ykPmhcUXVci30jFWR1IM__4BtAretCXF/view?usp=sharing

2.2.3 Примітки до роботи

1. Завдання розраховане на вже виконане завдання [Лабораторної роботи №2.1](#), і полягає у модифікації реалізації того завдання.
2. Для відповідності Linux Kernel Coding Style ознайомтеся зі скриптом *checkpatch.pl*.

2.1 Приклад використання \$KDIR/scripts/checkpatch.pl -f файл.c

2.2 Позначеного ERROR не повинно бути взагалі. З позначеного WARNING частина вас ще не стосується, проте простих речей на зразок “не треба на початку рядка ставити пробіл, а за ним табуляцію” також не повинно залишатися.

3. При виконанні роботи, не забувайте виконувати необхідні перевірки.

2.2.4 Завдання

В цій роботі також два варіанта завдань - **Basic** та **Advanced**. Студент вибирає один із варіантів.

2.2.4.1 Завдання Basic

1. Оголосити структуру даних для розміщення у списку, яка крім елемента `struct list_head` містить поле типу `ktime_t` (*include/linux/ktime.h* у вашому репозиторії linux-stable).
2. Створити статичну змінну голови списку.
3. Перед кожним друком привітання виділити пам'ять для екземпляра оголошеної структури, занести в неї поточний час ядра, отриманий функцією `ktime_get()`.
4. У функції `hello_exit()` пройти по списку і надрукувати час кожної події в наносекундах, вилучити елемент списку і звільнити виділену пам'ять. Приклад проходження по списку з вилученням елемента є у *appendix*.

На даному етапі досить виділяти пам'ять викликом

```
ptr = kmalloc(sizeof(*ptr), GFP_KERNEL);
```

і звільнити її викликом

```
kfree(ptr);
```

2.2.4.2 Завдання Advanced

У цьому завданні вся робота зі списком виконується в модулі *hello1*. Також додати ще одне поле типу `ktime_t` і засікати час до та після виклику функції друку, а на вивантаженні модуля надрукувати час, який пішов на кожен друк.

2.2.5 Оформлення звіту

Звіт має містити:

1. Лістинг коду (для завдання Basic) або посилання на github/gitlab репозиторій, що містить файли з вихідним кодом та Makefile (для завдання Advanced).
2. Скріншоти виконання insmod/rmmod.

2.3. Лабораторна робота №2.3

Тема: Використання засобів відлагодження та зневадження при написанні модулів ядра ОС Linux.

Мета: Ознайомитися з методологією та основними засобами відлагодження, доступними при написанні модулів ядра ОС Linux.

2.3.1 Скорочені теоретичні відомості

2.3.1.1 Основна теоретична інформація

Процес написання модулів ядра операційної системи супроводжується підвищеною складністю процесу відлагодження такого коду. Моніторингу коду ядра та відстеження помилки під час його виконання значно складніше, ніж відлагодження звичайного виконавчого коду, оскільки код модулів ядра є набором функціональних можливостей, не пов'язаних з конкретним процесом. Окрім того, що помилки коду ядра дуже важко відтворюються, вони ще й здатні призвести до “падіння” всієї операційної системи. Часто це призводить до знищення значної частини інформації, яка могли б бути використана для зневадження.

2.3.1.2 Пошук помилок під час компіляції

В Linux є кількість макросів препроцесора, які оцінюють стан під час збирання та видають помилки збірки.

Ці підпрограми визначені в `<linux / bug.h>`:

- `BUILD_BUG ()` - припиняє будувати беззастережно.
- `BUILD_BUG_ON (умова)` - зупиняє збірку, коли `@condition` оцінюється як true. Зауважте, що `@condition` має складати час, що підлягає оцінці (наприклад, ціла константа, відоме значення вказівника та будь-який вираз зі значеннями, відомими під час компіляції).
- `BUILD_BUG_ON_ZERO (умова)` - припиняє збірку, коли `@condition` є істинним. Може використовуватися як ініціалізатор члена структури і повертає `(size_t) 0`, якщо `@condition` є хибним.
- `BUILD_BUG_ON_NULL (умова)` - зупиняє збірку, коли `@condition` є істинним, може використовуватися як ініціалізатор члена структури і повертає `NULL`, якщо `@condition` є хибним.
- `BUILD_BUG_ON_NOT_POWER_OF_2 (expr)` - зупиняє збірку, якщо `@expr` не має двох потужностей.

OOPS and Panic

- OOPS - не фатальний стан, що відбувається у відповідь на деяку неуточнену / необроблену поведінку у функціональності або викликається зовнішніми подіями під час роботи системи. Система може або відновитись із такого стану без будь-яких втрат даних / функціональності і продовжувати працювати навіть при обмежених функціональних можливостях.

- Panic - це фатальний стан, коли система не може продовжувати роботу без значної втрати функціональності та / або втрати даних, і вона не може відновитись через це безпечно. Часто після того, як система відновлюється від певних не фатальних OOPS пізніше може викликатися Panic, оскільки нормальна функціональність системи порушена і важливі структури даних можуть бути пошкоджені.

Макроси попередження під час виконання:

Вони визначаються і в `<asm-generic / bug.h>` як макроси препроцесора.

- WARN (умова, формат ...) - тригерне попередження, коли умовна оцінюється на true. Використовуйте `printk ()` для друку `@format` з додатковим повідомленням аргументів

- WARN_ON (умова) - тригерне попереджувальне повідомлення, коли умова оцінюється на true.

- WARN_ONCE (умова, формат ...) - як WARN (), але друкує попереджувальне повідомлення лише один раз

- WARN_ON_ONCE (умова) - як WARN_ON (), але друкуйте попереджувальне повідомлення лише один раз

Формат повідомлення OOPS

- Взагалі це залежить від архітектури, але має певну загальну структуру
- Повідомлення може містити такі дані для відстеження походження проблеми:

- Зареєстрований вміст
- Рядки, що описують обладнання, на якому спрацював повідомлення
- Рядок, що описує тип проблеми (наприклад, помилка сторінки, поділ на нуль тощо)
- Список модулів, пов'язаних з ядром
- Позиція вказівника інструкції (IP) та символічна назва функції, що володіє ним

2.3.1.3 Параметри конфігурації налагодження ядра

У Linux існує ряд функцій, пов'язаних із налагодженням:

- CONFIG_DEBUG_KERNEL - головний перемикач
- CONFIG_DEBUG_SLAB - кожен байт виділеної пам'яті встановлений на спеціальні значення, захист додається до та після виділеного об'єкта пам'яті.

- CONFIG_INIT_DEBUG- дає змогу перевіряти код, який намагається отримати доступ до пам'яті часу ініціалізації після завершення ініціалізації
- CONFIG_DEBUG_LIST - додає перевірку на відповідність у список функцій

Ядро буде працювати повільніше, але зможе ловити та повідомляти про помилки.

Контроль поведінки printk ()

Ось, як визначаються loglevel повідомлення в ядрі Linux

- 0 (KERN_EMERG) система є непридатною
- 1 (KERN_ALERT) дії потрібно вжити негайно
- 2 (KERN_CRIT) критичні умови
- 3 (KERN_ERR) помилкові умови
- 4 (KERN_WARNING) умови попередження
- 5 (KERN_NOTICE) нормальний, але важливий стан
- 6 (KERN_INFO) інформаційний
- 7 (KERN_DEBUG) повідомлень рівня налагодження

Усі повідомлення ядра з рівнем логів, меншим за рівень консолі будуть надруковані в консоль. Можна змінити ліміт, використовуючи: # echo 4 > / proc / sys / kernel / printk

2.3.2 Література

- 1) Debugging Techniques [Електронний ресурс]: Режим доступу - https://drive.google.com/file/d/1LJ_Gs1eMQqda-tkER-P1AAUkXJT3Bdof/view?usp=sharing
- 2) Kernel debugging config options [Електронний ресурс]: Режим доступу - https://drive.google.com/file/d/1aam7VigjN9_NlaRQBPOqcmtr1qAmoK8B/view?usp=sharing
- 3) Hunting the bugs at compile time [Електронний ресурс]: Режим доступу - https://drive.google.com/file/d/19sHqmSAYHD_Ie_jt-3_jFjVxA6BGJ-Tb/view?usp=sharing

2.3.3 Примітки до роботи

Завдання розраховане на вже виконане завдання [Лабораторної роботи №2.2](#), і полягає у модифікації реалізації того завдання.

Зауваження:

- 1) Подивіться адреси завантаження своїх модулів на sysfs у каталозі поруч з параметрами модуля: /sys/module/<modulename>/sections/

A. Файли з адресами мають такі ж назви, як було названо секції, тобто починаються з крапки (.text, .init.text, ...), врахуйте це.

- 2) Будь-ласка, додатково ознайомтесь з debugfs.
- 3) Докладніше у додатку *Debugging Techniques*.
- 4) Для довідки:

A. \$KDIR/Documentation/admin-guide/dynamic-debug-howto.rst

B. \$KDIR/Documentation/filesystems/debugfs.txt

C. \$KDIR/Documentation/filesystems/proc.txt

D. \$KDIR/Documentation/filesystems/sysfs.txt

E. \$KDIR/Documentation/admin-guide/sysfs-rules.rst

2.3.4 Завдання

Для **Basic** потрібно виконати одне з двох завдань на вибір.

Для **Advanced** потрібно виконати **обидва**.

2.3.4.1 Завдання Basic

- 1) Додайте BUG_ON() замість друку повідомлення та повернення -EINVAL для неприпустимого значення параметра.
- 2) Додайте примусове внесення помилки “начебто kmalloc() повернув 0” під час формування елемента списку для якогось повідомлення (останнього із серії, 5-го, ... — на ваш вибір).
- 3) Модифікуйте Makefile аналогічно *appendix1*.
- 4) Отримайте обидва повідомлення, роздивіться їх та для одного з них виконайте пошук місця аварії аналогічно *appendix1*.

Зауважте, що при виконанні BUG_ON() модуль буде “зайнятий”, і ви не зможете виконати rtmmod.

2.3.4.2 Завдання Advanced

- 5) Упевніться у відсутності каталогу: `/sys/kernel/debug/dynamic_debug`
Це означає вимкнену опцію CONFIG_DYNAMIC_DEBUG (якщо збиралося по методичці, то не повинно бути).
- 6) Замініть у функції exit модуля hello (hello1) друк вмісту списку на pr_debug і додайте два виклики pr_debug до та після друку списку.
- 7) Перевірте залежність друку повідомлень від #define DEBUG на початку файлу.
- 8) Перезберіть ядро з увімкненим CONFIG_DYNAMIC_DEBUG, замініть його на nfs. Перезберіть модуль.
- 9) Аналогічно показаному в *appendix2*, поекспериментуйте з друком з прапорцями p, f, m, а також зі встановленням їх для всього модуля та для окремих рядків.

2.3.5 Оформлення звіту

Протокол має містити лістинг коду (для завдання Basic) або посилання на github/gitlab репозиторій, що містить файли з вихідним кодом та Makefile (для завдання Advanced), а також скріншоти виконання відлагоджування.