Лабораторна робота №2   
Програмування на C і ASM   
(Приклад обробки тексту и обчислення суми на ASM)

# оГЛЯД

Більшість програм для мікроконтролерів пишуться на мові С, оскільки це простіше і швидше (а значить і дешевше), ніж на мові асемблера. Але іноді буває так, що деякі критичні до часу виконання частини програм необхідно писати на асемблері. Використовуючи асемблер, ви можете по максимуму оптимізувати код програми так, як вважаєте за потрібне. Але все ж більшість з систем закодовано на мові C з мовою асемблера, який використовується тільки для критично важливої обробки. Також мова С (при використанні таких бібліотек, як CMSIS), забезпечує переносимість програм на різні мікроконтролери.   
З іншого боку, при написанні програми на мові Сі може виникнути необхідність у використанні асемблера для реалізації функцій, які не можуть бути написані на Сі, наприклад:

* функцій, що безпосередньо маніпулюють вмістом стека або використовують спеціальні команди, які не можуть бути отримані за допомогою стандартних конструкцій мови;
* для написання процедур, критичних до часу виконання;
* при обмежених обсягах пам'яті, що вимагає написання частини програми на асемблері для отримання якомога більш компактного коду

В підсумку, запис асемблерного коду у вигляді функцій, які можуть бути викликані з C-коду як функції C, призводить до модульності програм, що дає нам найкраще з обох світів: швидке, модульне C-програмування, і висока продуктивність асемблерного коду.

## ТЕХНІКИ ЗМІШУВАННЯ мови АСЕМБЛЕРУ І C-кодУ

Можливі три техніки, змішування коду С і асемблера:

1. при використанні так званого *вбудованого асемблера* (або, в разі засобів GNU, inline-асемблера) в тексті програми на Сі:
2. при виклику з програми на Сі функції або процедури, написаної на асемблері і реалізованої в окремому файлі;
3. при виклику з асемблерної програми процедури або функції, написаної на Сі.

При цьому необхідно чітко уявляти собі, яким чином здійснюється передача параметрів в функцію, що викликається, і повернення результату її роботи в програму, яка її викликала. Механізм такої взаємодії детально описаний у [стандарті AAPCS](http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf) [5]. Кратко це виглядає так:

* Вхідні параметри передаються в регістрах R0-R3. Якщо ж обсяг переданих даних більше 4 \* 32 біт (важливо не кількість переданих, а їх загальний розмір), то вони передаються через R0-R3 + стек.
* Значення, що повертається, зазвичай зберігається в R0, а якщо це значення займає 2\*32 біта, то в R0-R1.
* Регістри R0-R3 можуть змінюватися в функції, тоді як вміст R4-R11, R12 і LR слід зберігати при вході в підпрограму і відновлювати при виході з неї. Зазвичай для цього використовують стек.
* Значення SP має бути вирівняно за подвійним словом.

## ЗАУВАЖЕННЯ

Перш ніж приступити до роботи, вам необхідно ознайомитися з документом «Початок роботи» (Додаток А «Встановлення і налаштування Keil uVision5» до лабораторної роботи №1), в якому міститься інструкція по налаштуванню середовища програмування. Корисним буде також знайомство з етапами створення проекту і прийомами роботи в IDE Keil μVision5, а також з основами мови Асемблера для ARM, отримане вами при виконанні лабораторної роботи №1.

# заВДАННЯ на рОботу

У цій роботі ви познайомитеся зі структурою і технологією створення проекту, в якому основна програма, написана на С, викликає функції (підпрограми), написані на asm. Ви будете писати і компілювати код на С і asm, і налагоджувати образ програми на отладочной платі (а саме, на платі «Nucleo» або «Discovery») за допомогою інструменту Keil μVision5 MDK-ARM. Ви виконаєте дві вправи:

У першої - відпрацюєте техніку вбудованого асемблера;

У другої – техніку виклику asm-функцій, реалізованих в окремому файлі.

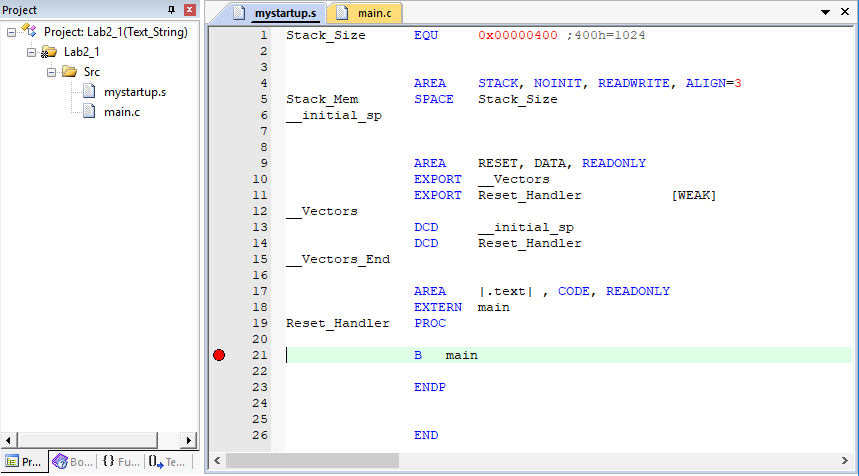
## **Обробка текстового рядку (вбудований асемблер)**

У проект з програмою main на C, добавімо підпрограми на asm для виконання операцій копіювання символьних рядків і їх «капіталізації» - перекладу кодів малих символів (літер) в прописні (великі літери), застосовуючи техніку вбудованого асемблера (inline assembly).

Будемо прагнути дотримуватися модульності програм.

## STARTUP

Виділимо процедуру запуску програми в окремий asm-файл **mystartup.s** (рис. 1).



Точка останову  
 (Breakpoint)

Функція надана у іншому файлі

Рисунок 1 ‒ Структура проекту и програма **mystartup**

## Main

Спочатку створимо основну функцію на C. Ця функція містить дві змінні (**а** і **b**) типу масиву символів. Масив a ініційований ASCIIZ-рядком символів, а в спочатку «порожній» масив b функцією **my\_strcpy** буде скопійована рядок з масиву а. Потім функція **my\_capitalize** перекодовує в масиві b всі зустрінуті рядкові символи в прописні. Обидві функції будуть написані на асемблері.

int main(void)

{

const char a[] = "Hello world!";

char b[20];

my\_strcpy(a, b);

my\_capitalize(b);

while (1);

}

## УГОДИ ПРО регістрИ і ПЕРЕДАЧУ параметрів

Будемо дотримуватися загальноприйнятих угод про використання регістрів, щоб наш asm-код співіснував з кодом C.

### ВИКЛИК функцІй І передача аргументІв

Коли функція викликає підпрограму, вона поміщає адресу повернення в регістр посилань **lr.** Аргументи (якщо вони є) передаються в регістрах від **r0** до **r3**, починаючи з **r0**. Якщо є більше чотирьох аргументів або вони занадто великі, щоб вписатися в 4\*32-розрядні регістри, вони передаються через стек.

### ТИМЧАСОВЕ СХОВИЩЕ

Регістри ***r0-r3*** можуть використовуватися в процедурі для тимчасового зберігання будь-яких значень, якщо вони не використовувалися для передачі аргументів, або якщо значення аргументу вже більше не потрібно.

### резервування регістрів

Регістри ***r4-r11, R12***, якщо вони будуть використані підпрограмою, повинні бути спочатку збережені і відновлені перед поверненням. Виконаємо це шляхом збереження їх у стекі.

### повернення з функцій

Оскільки зворотну адресу було збережено в регістрі зв'язку **Lr**, інструкція **BX Lr** наприкінці підпрограми (процедури) буде перезавантажувати процесор з повернутим значенням адреси з **Lr**. Якщо функція повертає значення, вона буде передана через регістр **r0**. Як що функція повертає два параметри типу int (32-біт), або один double або long long (64-біт), то їх значення повинно бути у **r1,r0.**

## функІя копіЮванНя Рядків

Функція **my\_strcpy** має два аргументи (**src, dst**). Кожен з них являє собою 32-розрядний покажчик на символ. У нашому випадку покажчики вписуються в регістри, тому аргумент **src** передається через регістр **r0**, а **dst** передається через r1.

Функція буде завантажувати символи з рядка (масива) пам'яті **a[](src)** і копіювати його у рядок **b[] (dst).**

\_\_asm void my\_strcpy(const char \*src, char \*dst)

{

loop

LDRB r2, [r0] ; Завантажуємо у r2 байт з комірки пам'яті на яку показує r0 (src)

ADDS r0, #1 ; Нарощуємо покажчик src на 1 (переводимо його на наступний байт) src

STRB r2, [r1] ; Збережемо байт з r2 в коміркі памяті, на яку вказує r1 (покажчик dst)

ADDS r1, #1 ; Нарощуємо покажчик dst

CMP r2, #0 ; Чи дорівнював байт нулю? /(Кінець рядка src?)

BNE loop ; Якщо ні (прапорець Z=0/"not equal" - "не дорівнює"), повторимо цикл

BX lr ; Инакше - повернення з підпрограми

}

## переКЛАД в ЛІТЕРИ верхнЬОго регІстра (ВЕЛИКІ ЛІТЕРИ)

Тепер подивимося на підпрограму, яка переводить всі букви в рядку у верхній регістр. Нам потрібно завантажити кожен символ в регістр, перевірити, чи є він символом малої літери, і, якщо ні, то пропустити його, а якщо так, змінити його код на код символу відповідної великої літери і повернути в рядок.

Кожен символ в рядку представлений [кодом ASCII](ASCII.bmp). Наприклад, '**A'** »представляє собою 65 (0x41), **'B'**» - 66 (0x42) і т. Д. До **'Z'** », який має код 90 (0x5a). Букви нижнього регістра починаються з «**a**» (97 або 0x61) і закінчуються на «**z**» (122 або 0x7a). Ми можемо перетворити малу літеру в відповідну їй прописну, віднімаючи з її коду 32 (0x20).

\_\_asm void my\_capitalize(char \*str)

{

cap\_loop

LDRB r1, [r0] ; Завантажимо у r1 байт з комірки пам'яті, на яку показує r0 (\*str)  
 ; (r0 - покажчик на рядок, переданий як аргумент функції)

CMP r1, #'a'-1 ; Порівняємо його з кодом символу, що передує "а" в ASCII таблиці

BLS cap\_skip ; Якщо менше або дорівнює, проминемо цей байт

CMP r1, #'z' ; Инакше порівняємо його з кодом символу 'z'

BHI cap\_skip ; Якщо він вище, теж проминемо цей байт

; Тобто, це не код малої (рядкової) англійської літери

SUBS r1,#32 ; Інакше віднімемо різницю 32 (0х20), щоб перевести в код великої літери

STRB r1, [r0] ; Збережемо код великої літери в пам'яті

cap\_skip

ADDS r0, r0, #1 ; Нарощуємо покажчик str

CMP r1, #0 ; Чи дорівнював байт нулю?

BNE cap\_loop ; Якщо ні, повторюємо цикл

BX lr ; Инакше - повернення з підпрограми

}

Код процедури показаний вище. Він завантажує байт в ***r1***. Якщо байт менше, ніж ***'a'*** або більше ніж '***z***', тоді модифікація байта пропускається і код переходить до завершення ітерації циклу ***cap\_skip***, де перевіряє, чи не досягнуто кінець рядка (код 0). Якщо немає, то операція повторюється з наступним символом - виконується повернення ***BNE*** *(branch no equal)* на мітку ***cap\_loop***.

Цей код має прийом (quirk) - перша команда порівняння **CMP** *(Compare)* порівнює **r1** з кодом, розташованим в ASCII-таблиці безпосередньо перед кодом символу «**a**». Навіщо? Адже нам хотілося б порівняти **r1** з **'a'**, а потім перейти, якщо він виявиться нижче (lower). Однак, інструкції умовного переходу **branch lower** в системі команд немає (див. Додаток 2 до Лабораторної роботи №1), а є тільки **BLS** *(branch lower or same)***.** Щоб використовувати цю інструкцію, нам потрібно зменшити на одиницю значення, яке ми порівнюємо з **r1**.

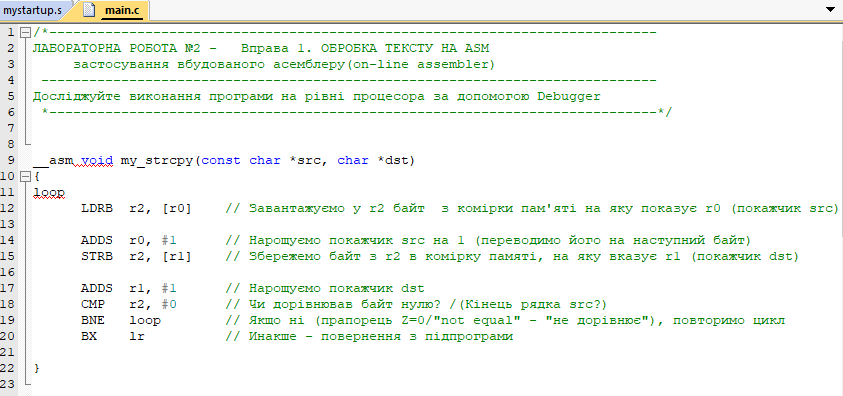


Рисунок 2 ‒ Функція копіювання рядка з файлу main

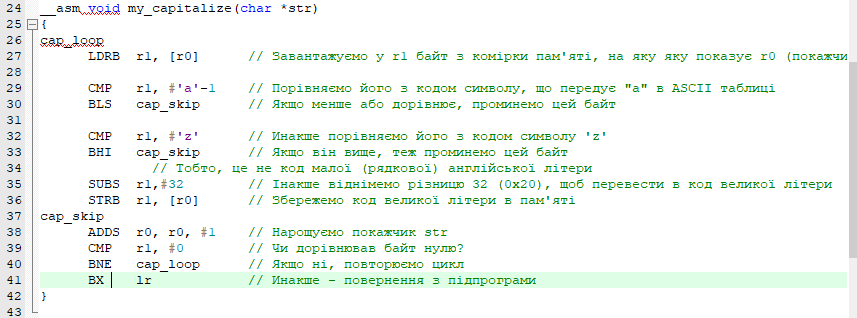


Рисунок 3 ‒ Функція my\_capitalize з файлу main

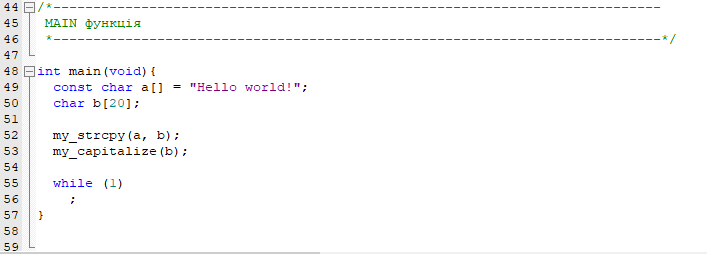
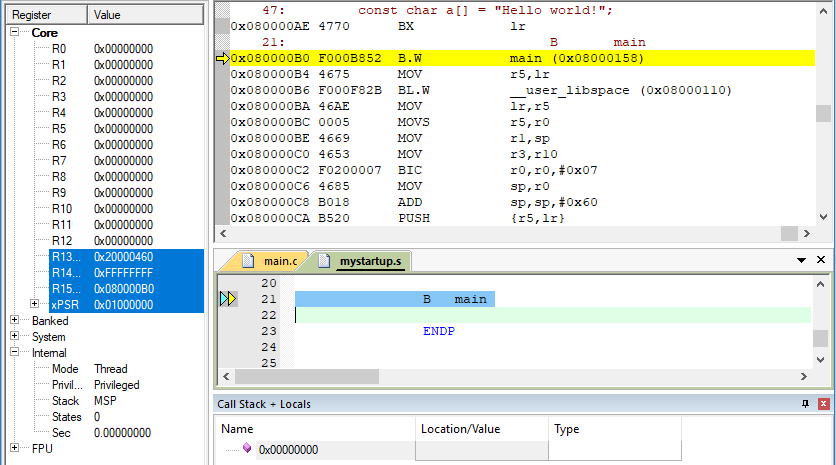


Рисунок 4 ‒ Функція main

# процедура вИКОНАНня

1. Створить проект із структурою (рис 1).
2. Скомпілюйте код.
3. Завантажте код на плату.
4. Запустіть сеанс налагодження (Ctrl + F5) і запустіть програму до тих пір, поки не буде досягнена точка останову (Breakpoint) (див. рис. 1). Які стартові значення покажчика стека (**r13**), регістра посилань (**r14**) і лічильника програм (**r15**)?(рис. 5)

**Рисунок 5 ‒ Старт програми

*(Майте на увазі, що значення регістра і адреси, показані на знімках, можуть відрізнятися від того, що отримуєте ви)*

1. Вікно «Disassembly» (View->Disassembly Window). На яку інструкцію показує жовта стрілка, і яка її адреса? Як ця адреса відноситься до значення **PC**?   
   На рисунку виділено фрагмент коду початкових дій з підготовки сегментів стеку і коду. Зокрема готується і заноситься у Flash масив символів (констант – const) **a[].**
2. Виконуйте покрокове трасування (**F11**) програми. На яку адресу перейде програма? Значення якого регістра змінилося (він повинен бути виділеним у вікні «Registers»)? (см. рис. 6)
3. Додайте вікна для дослідження дампів пам’яті Flash (Memory1) та стеку (Memory3).
4. Що буде виконувати фрагмент програми, виділений на рис.6.?   
   Подивіться інструкції у вікні «Disassembly». Поясніть дію інструкцій, довжина яких становить чотири байти? (Див. рис. 6). Що і як змінюється після виконання виділеного фрагменту ?

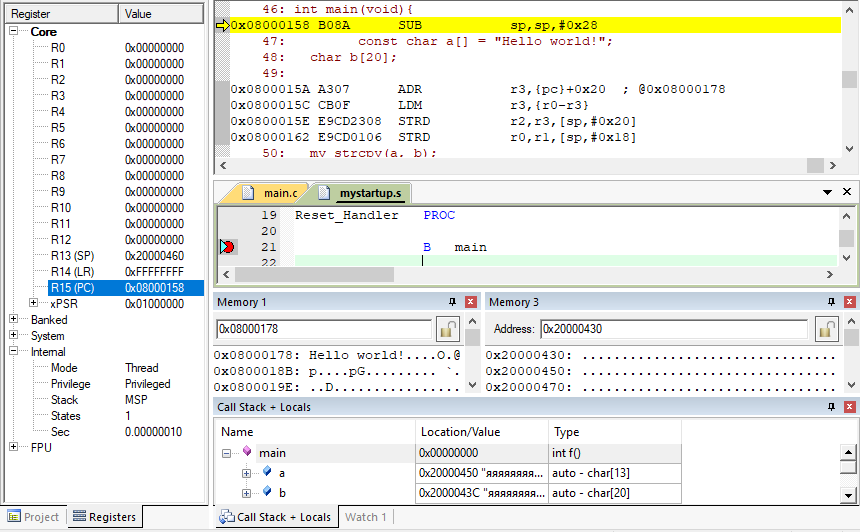


Рисунок 6 ‒ Стан ділянок пам’яті. Вхідні константи (рядок a[) у Flash

1. Що відображається у вікі «Call Stack + Locals»?

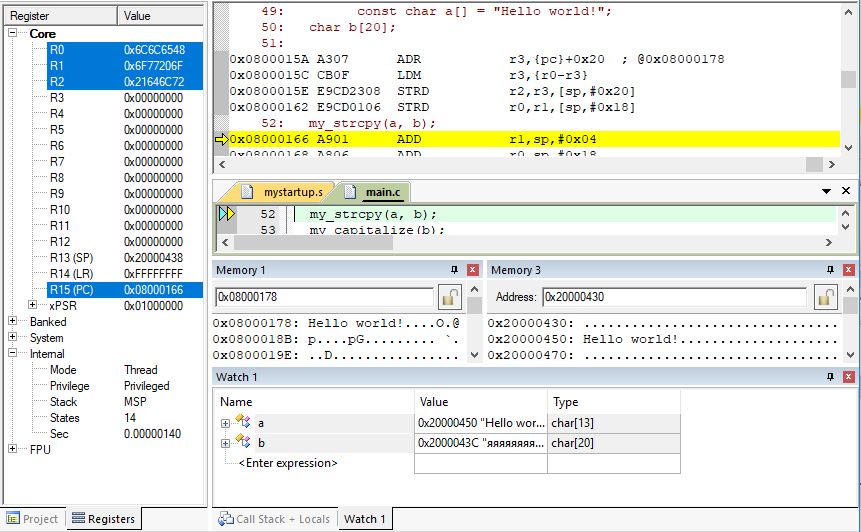
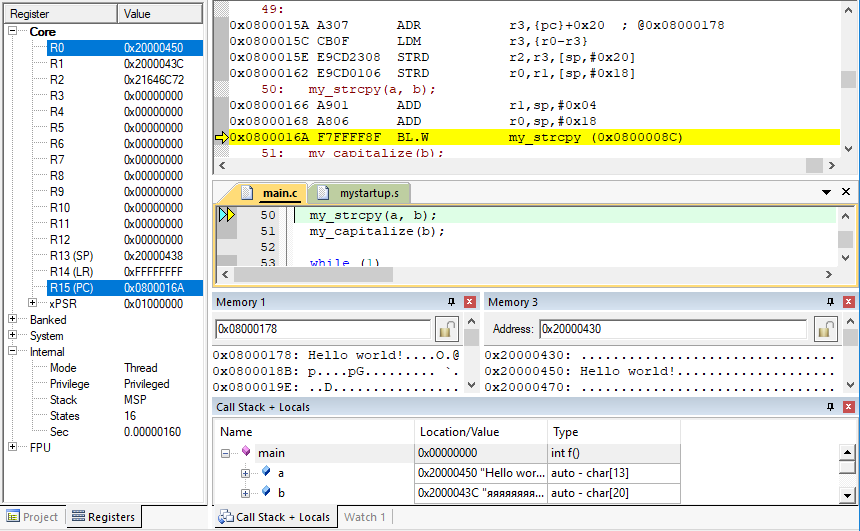


Рисунок 7 ‒ Розміщення вмісту рядка **a[]** у стеку, копіюванням з сегменту коду

1. Продовжте виконання (**F11**) до досягнення інструкції **my\_strcpy BL.W**. Які значення **SP, PC** і **LR**? (Рис.8).
2. Що змінилося і чому? Чи відповідає значення **PC** тому, що показано в вікні «Disassembly»? (див. рис.8).

R0: value of scr pointer

R1:value of dst pointer



Stack Pointer, Link Register and Program Counter

Рисунок 8 ‒ Підготовка вхідних параметрів для функції

1. Які регістри тримають аргументи функції my\_strcpy, і який їх вміст?   
   Яким чином (під впливом яких команд) змінився їх вміст? (див. рис. 8)
2. Подивіться вікно «Call Stack + Locals», щоб проаналізувати змінні «a» і «b» (див. малюнок 9))

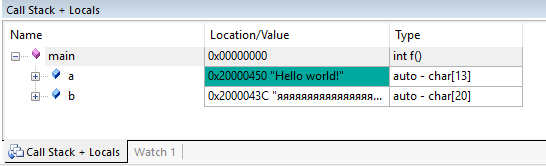


Рисунок 9 ‒ Початковий стан рядків **a[ ]** і **b[ ]**

1. Чому дорівнює «а»? Чому дорівнює «b»?
2. Виконайте команду **BL.W my\_strcpy**. Які значення **SP**, **PS** ? Чому дорівнює значення **LR**?
3. Виконуйте команди крок за кроком (**F11**), спостерігаючи за вікном стеку (Memory 3) і ««Call Stack + Locals», щоб побачити, що рядок **a[]**символом за символом копіюється в **b[].**
4. Яка команда виконує переcилання? У якому регістрі знаходиться символ, що пересилається (операнд команди)?

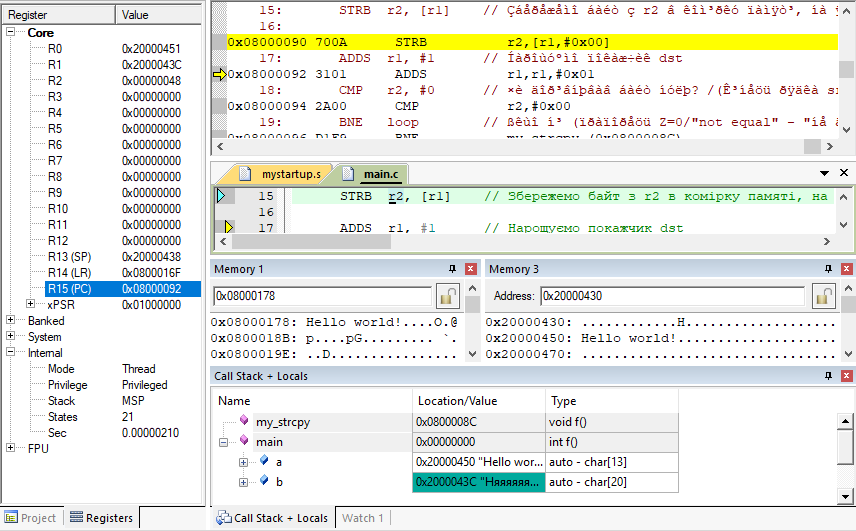
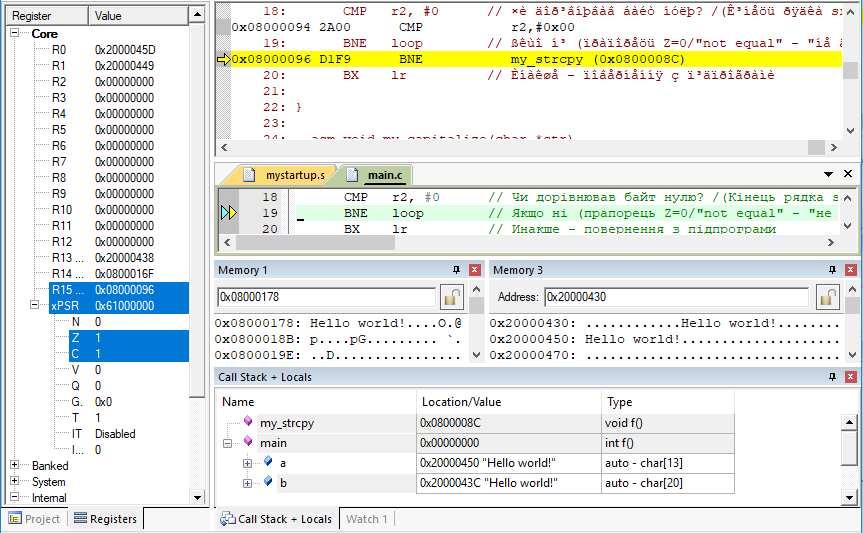


Рисунок 10 ‒ Пересилання першого символу з рядка **a[ ]** у рядок **b[ ]**

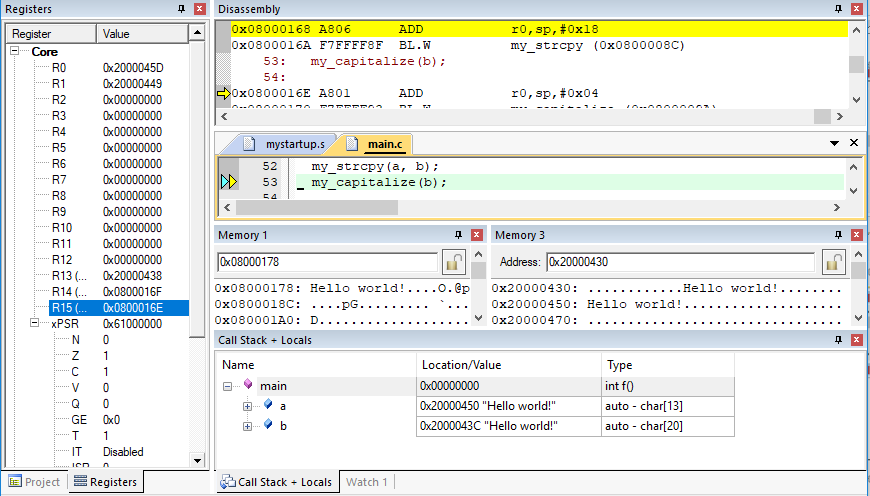
1. Простежте заповнення рядка **b[]** в стеці до останнього символу («**0»**). Що вказує процесору на завершення циклу? (рис. 11).



**?**

Рисунок 11 ‒ Завершення копіювання рядків **a[ ]** і **b[ ]**

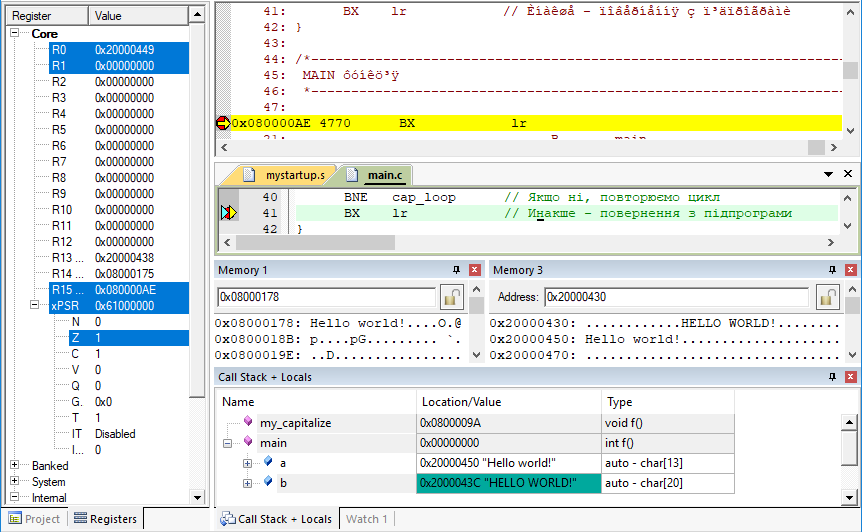
1. Які значення символу, покажчикf **SP**, покажчик **Ds**t (**?**), Регістр зв'язку **lr** (**R1**4) і лічильник команд (**R15**), коли код досягає останню команду в підпрограмі (**BX lr**)?
2. Виконайте команду **BX Lr**. Тепер, яке значення **PC** ? (Див. Рис.12).



**PC**

Рисунок 12 ‒ Повернення з підпрограми my\_strcpy

1. Який зв'язок між значенням **PC** і попереднім значенням **LR**? Поясніть.
2. Тепер крокуйте через підпрограму **my\_capitalize** і переконайтеся, що віна правильно працює, перетворюючи "Hello world!" В "HELLO WORLD!".



Символи «капіталізуються» один за одним від a до b

Рисунок 13 ‒ Завершення підпрограми **my\_capitalize**

## **Обчислення суми декількох чисел (окремі asm-процедури)**

Створимо проект з 4-х файлів. Файл **mystartup.s** (рис. 14) має той же зміст і призначення, що і в попередньому прикладі.

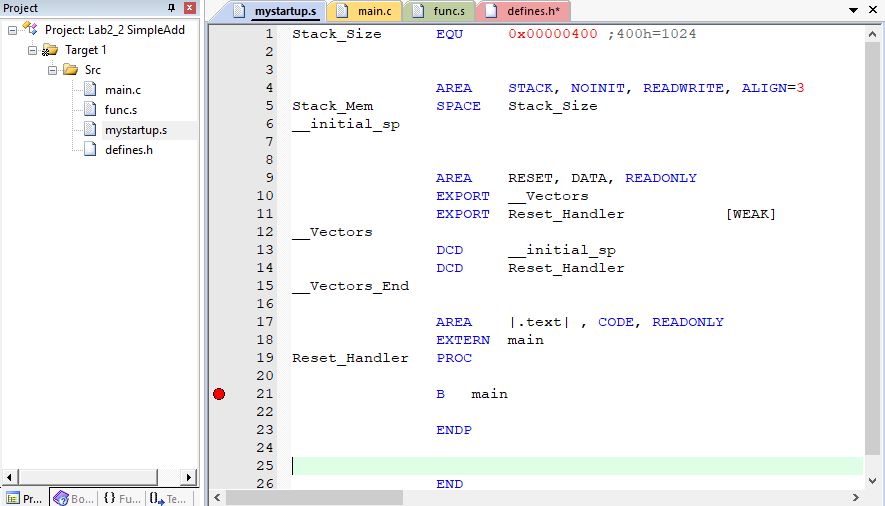


Рисунок 14 ‒ Структура проекту и програма **mystartup**

У файлі **main.c** ініціюємо шість беззнакових цілих чисел **a**, **b**, **d**, **e**, **f**, **h** і викличемо функцію **Simple\_add()**, яка визначить суму цих чисел, і поверне її у змінної **с** (рис. 15).

В файлах **func.c** і **defines.h** (рис.16, рис.17) наведено, відповідно, *визначення* і *об’ява* функції **Simple\_add()**. Зверніть увагу на те, що, оскільки кількість вихідних параметрів функції перевищує 4, то перші ‒ **a**, **b**, **d**, **e** за замовчанням будуть передані функції через регістри **r0 – r3**, а решта **f** і **h** – через стек.

Нагадуємо, що *об’явлення* *функції* (*прототип*, *заголовок*, *сигнатура*) повинно находитися у тексті програми раніше її виклику (див. директиву **#include ”defines.h”** у першому рядку файлу **main.c** на рис.15).

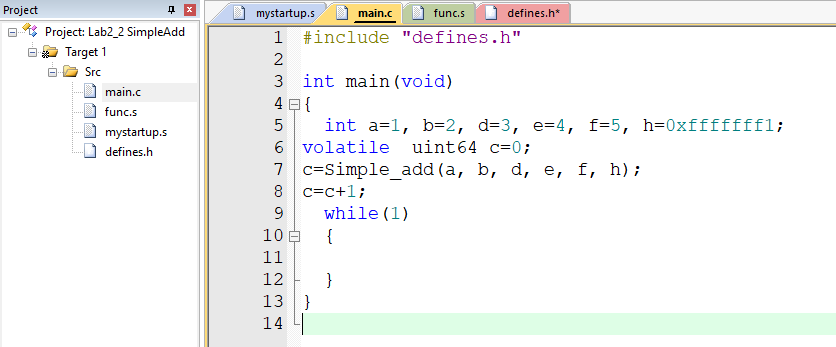


Рисунок 15 ‒ Функція **main**

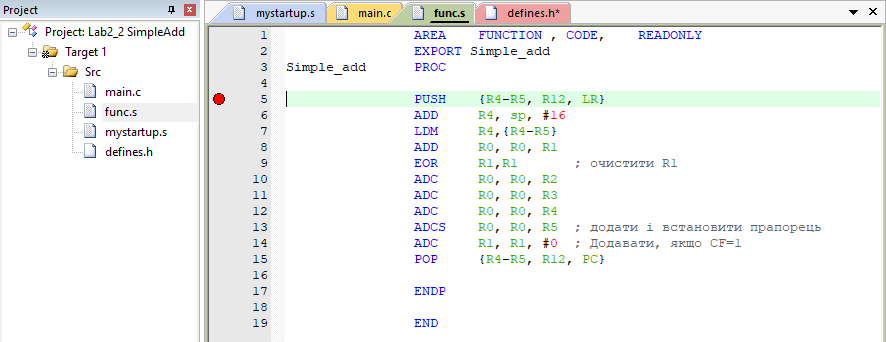


Рисунок 16 ‒ ASM-функція **Simple\_add** у окремому файлі **func.s**

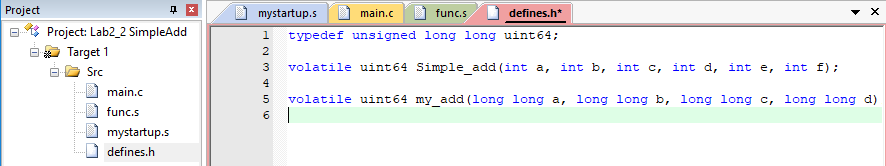


Рисунок 17 ‒ Файл **defines.** об’яв функцій і позначень

Зверніть увагу на інший спосіб повернення з підпрограми (див. рис. 16):

;На початку підпрограми  
**push {r4 – r5},r12,lr** ; Зберігаємо значення регістрів **r4**, **r5** і **r12** разом з **lr**  
 **….** ; Виконуємо потрібні дії (тіло підпрограми)

;В кінці підпрограми  
**pop {r4 – r5},r12,pc** ; Відновлюємо вміст регістрів, але останнім знімаємо зі стеку   
 ; збережену адресу повернення (вміст **lr)** і записуємо її у   
 ; програмний лічильник **pc** , і таким чином повертаємося з підпрограми.

Тут також показано як однією командою зберегти у стеку, а потім відновити декілька регістрів.

Конструкція **{Rn ‒ Rn+k}** називається списком регістрів (*reg list*) і замінює запис **Rn,Rn+1,...,Rn+k**.

1. Поясніть дію команд з рядків 6 і 7 функції **Simple\_add** (рис. 16).
2. Яку дію виконує (для чого введена) команда з рядка 14?
3. Що являє собою результат функції **Simple\_add** і яким чином він повертається в основну програму main?

Створить наведену тут програму і дослідить її виконання процесором Cortex-M4 зі складу МК STM32f4xx у Debugger

# заВДАННЯ до самостійной розробки

Для закріплення матеріалу виконайте у якості індивідуального завдання наступне:

**До вправи 1**. Копіювання рядка символів у другий рядок «з заду на перед»

**До вправи 2**. Зробіть дві asm-функції. Одна обчислює суму 32-бітних «без знакових» елементів масиву, а інша враховує їх знак. Результат 64-бітне «без знакове» та «знакове» значення. Вхідні масиви завдайте у С-модулі. При виклику функції з С-модулю в якості аргументів слід передавати посилання (покажчик) на адресу масиву і його розмір (кількість елементів).

# Література

* 1. ARMv7-M Architecture Application Level Reference Manual [Електронний ресурс] Режим доступу: [www.arm.com/products/CPUs/ARM\_Cortex-М3\_v7.html](http://www.arm.com/products/CPUs/ARM_Cortex-М3_v7.html)
  2. Cortex M4 Technical Reference Manual [Електронний ресурс]Режим доступу:   
     <http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf>
  3. STM32F3xx/F4xxx Cortex®-M4 programming manual (PM0214) [Електронний ресурс] Режим доступу:  
     [www.st.com/resource/en/programming\_manual/dm00046982.pdf](http://www.st.com/resource/en/programming_manual/dm00046982.pdf)
  4. **Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство** / Джозеф Ю ; пер. с англ. А.В. Евстегнеева, -- М.: Дедока-ХХІ, 2002. – 552 с.:ил. – (Мировая електроника). – Доп.тит. л. англ.. – ISBN 978-5-94120-343-0
  5. **The Definitive Guide to ARM ® Cortex ® -M3 and Cortex-M4 Processors.** 3rd.Edition. /**Joseph Yiu.** ARM Ltd., Cambridge, UK. Nov.2013.ISBN.0124080820.pdf   
     (Электронний варіант на Google Disk)
  6. AAPCS Procedure СаН Standard for the ARM Architecture [електронний ресурс] режим доступу:   
     <http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf>