Проектиране на вградени системи
Лабораторно упражнение №3

Работа с кросдебъгер GDB и сървърно приложение OpenOCD. Дебъгване на вградени системи през Интернет.

==========

1. Сървърното приложение OpenOCD има за цел да уеднакви протоколите за комуникация на няколко дебъг интерфейса и да предостави дебъг достъп на потребителя до неговия микроконтролер посредством стандартен TCP/IP терминал или софтуерния дебъгер GDB [1].
Отворете два терминала с комбинацията CTRL + T под Линукс. В терминал 1 стартирайте сървърното приложение OpenOCD:

```
openocd -f board/stm32f7discovery.cfg
```

а в терминал 2 се свържете към него чрез клиентското приложение telnet през виртуалния мрежови интерфейс на вашия компютър с IP → 127.0.0.1 и порт → 4444 :

```
telnet 127.0.0.1 4444
```

В ново-излезлият команден ред на telnet инициализирайте дебъг връзка и спрете ядрото на микропроцесора със следните команди:

```
init
reset halt
```

След това заредете във флаш предварително компилираната програма от директория **03_1** с командите:

```
flash banks
flash probe 0
flash write_image erase /path/to/stm32_demo.bin 0x08000000
reset run
```

където първата команда показва наличните банки Flash, а втората инициализира вътрешната Flash банка 0 от паметта. Третата команда записва двоичния файл посредством SWD дебъгера във въпросната Flash, а адресът 0x08000000 е нейният начален адрес. Ако записът е бил успешен, на екрана на макета трябва да се появи анимация след последната команда, която задава ресет на микроконтролера.

2. Прочетете току-що записаната програма с командата:

flash read_bank 0 /path/to/Desktop/firmware.bin 0x0 2097152

където 0 е номер на Flash банка, 0x0 е отместване от началния ѝ адрес (в случая 0x08000000), а 2097152 е брой байтове който да прочете (2 x 1024 x 1024 = 2 MB).

3. Изпълнете програмата стъпка по стъпка. За целта трябва първо да спрете изпълнените на програмата с командата:

```
halt
```

и след това да напишете командата няколко пъти:

```
step
```

която ще ви изписва следната информация:

```
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x08003026 msp: 0x2007feb8
```

Тук “Thread” е режимът на работа на ARM ядрото, xPSR е обобщеният регистър на състоянието (status register), pc е програмният брояч, msp е основният стеков указател.

След няколко стъпки можете да пуснете отново програмата да се изпълнява до край с командата:

```
resume
```

4. С команди на OpenOCD може да се четат и записват регистри на микроконтролера. Това са командите:

md[x] [адрес] [брой думи]
mdd - чете 64-битови двойни думи
mdw - чете 32-битови думи
mdh - чете 16-битови полу-думи
mdb - чете 8-битово число

mw[x] [адрес] [данни] [брой думи]
mwd - записва 64-битови двойни думи
mww - записва 32-битови думи
mwh - записва 16-битови полу-думи
mwb - записва 8-битово число

Прочетете първите 32 32-битови думи от началото на Flash паметта

(0x08000000):

```
mdw 0x08000000 32
```

5. Пуснете светодиода от предишното лабораторно да свети без да компилирате кода на програмата и без да я зареждате в паметта. За целта първо изтрийте демо програмата с:

```
reset halt
flash probe 0
stm32f2x mass_erase 0
reset halt
```

Необходимите адреси и данните на съответните адреси са:

//AHB1ENR, Enable GPIOA
0x40023830 0x01
//MODER, Set PA12 as output
//**ВНИМАНИЕ** – на този порт се намират SWD сигналите!
0x40020000 0x1000000
//TYPER, Set PA12 as push-pull
0x40020004 0x00
//ODR, Set logic 1 on PA12
0x40020014 0x1000

6. В Линукс терминал влезте в debug директорията на проекта ви от миналото упражнение и стартирайте софтуерната дебъг програма GDB [2]:

```
arm-none-eabi-gdb
```

Тази команда ще стартира дебъг сесия. Свържете GDB с OpenOCD с помощта на командата:

```
target remote 127.0.0.1:3333
```

Заредете програмата ви за изчисляване на RMS стойност във Flash паметта на микроконтролера посредством GDB командата:

```
load main.axf
```

Заредете символната и дебъгерната информация от .axf файла в GDB, за да можете да дебъгвате със съответните команди:

```
file main.axf
```

Клиентската програма GDB позволява да се предават команди директно на

сървърната OpenOCD посредством командата monitor. Например следните команди са аналогични на тези от предходните точки:

```
monitor init
monitor reset halt
monitor mdw 0x08000000 32
```

и други.

Поставете точка на прекъсване в началото на main( ) функцията ви посредством (действието се нарича **set breakpoint**):

```
break main (или само b main)
```

Аналогично можете да сложите точки и в други места на програмата ви:

```
break led_set
break led_clear
```

Точка на прекъсване може да се сложи на произволен ред така:

```
break <номер на ред>
break <име на файл : номер на ред>
```

Списък с всички активни точки може да се покаже с командата:

```
info break
```

Премахване на всички точки или само на някои от тях може да стане с :

```
delete breakpoints
delete breakpoint <номер на точката от списъка “info break”>
```

Пуснете програмата да се изпълнява, до безкрай или докато не срещне точка на прекъсване (действието се нарича **resume**):

```
continue (или съкратено с)
```

Ако всичко е минало успешно в терминала ви ще се изпише:

```
(gdb) continue
Continuing.
Note: automatically using hardware breakpoints for read-only
addresses.

Breakpoint 1, main () at main.c:22
22        double val3 = 4.002503;
```

Сега можете да изпълнявате програмата стъпка по стъпка, т.е. ред по ред във вашата С програма посредством командата:

`step` (или съкратено s)

След всяка стъпка дебъгерът ще ви показва информация за програмния брояч (не забравяйте, че програмният брояч нараства в зависимост от размера на изпълняваните инструкции, както и от размера на конвейера на съответния микропроцесор). С тази команда, ако срещнете функция в сорс кода ви ще влезете в нея да я дебъгвате (действието се нарича **step into**).
Възможно е стъпките да са на ниво Асемблер, тогава трябва да се използва:

`stepi`

Ако е излишно дебъгването на всяка срещната функция, може да се използва командата:

`next` (или съкратено n)

която срещайки функция, ще изпълни кода ѝ и ще предаде контрол обратно на програмиста на излизане от функцията (действието се нарича **step over**).
Ако сте влезли в много голяма функция и искате да излезете на ниво функцията, която я извикала първоначално, използвайте командата (действието се нарича **step out**):

`finish`

Ако е необходимо, паметта на микроконтролера може да се прочита с командата:

`x <адрес>`

Разгледайте първите две думи в началото на Flash паметта, както направихте в предходните точки:

```
x 0x08000000
x 0x08000004
```

Докато е спряно изпълнението на програмата ви, разгледайте променливите ѝ с помощта на командата:

```
p <име на променлива>
p <име на файл :: име на променлива>
p <име на функция :: име на променлива>
```

Командата p може да приема и форматиращи спецификатори (подобно на printf) като се сложи наклонена черта и буква:

```
p/<форматиращ спецификатор> <име на променлива>
```

Форматиращ спецификатор	Детайли
x	Целочислена стойност в шестнадесетичен вид
d	Целочислена стойност в десетичен вид със знак
u	Целочислена стойност в десетичен вид без знак
o	Целочислена стойност в осмичен вид
t	Целочислена стойност в двоичен вид
a	Адрес на променливата в паметта + отместване от най-близкия символ
c	Символ + числен еквивалент на символа
f	Число с плаваща запетая
s	Символен низ
z	Същото като 'x', но стойността ще бъде предшествана от нули в зависимост от размера ѝ в паметта
r	Python pretty-printer

Съдържанието на регистрите на ядрото може да бъде показано с командата:

```
info registers
```

7. Дебъгерната програма GDB може да чете стекови извадки (stack frame). За да тествате тази команда, напишете във вашия main.c файл три функции, всяка от която извършва някакви изчисления и вика следващата, например:

main → my_func_1 → my_func_2 → my_func_3

Спрете дебъг сесията, компилирайте и заредете програмата, стартирайте нова дебъг сесия. Сложете точка на прекъсване в най-вътрешната функция (my_func_3) и напишете някоя от командите:

```
backtrace (или само bt)
backtrace full
```

Втората команда показва всички локални променливи, които се съхраняват в стек паметта (SRAM) в дадения момент от програмата ви.

8. Дебъгерната програма GDB може да работи в мрежи с отдалечени target-и. За да изпробвате тази възможност нека една от работните маси да бъде OpenOCD сървър с демо платка, а друга маса да бъде GDB клиент с фърмуер, който ще се дебъгва.
На сървърното PC стартирайте терминал и напишете:

```
ifconfig
```

Оттам ще разберете IP адреса на въпросното PC. Стартирайте OpenOCD както направихте в задача 1.
На клиентското PC стартирайте GDB и на командата target remote напишете IP-то на сървъра на вашите колеги. Например, ако IP-то на сървъра е 192.168.1.143, то изпълнете следните команди:

```
arm-none-eabi-gdb
target remote 192.168.1.143:3333
```

Тествайте командите от предишните задачи.

ВНИМАНИЕ: Примерът работи в локална мрежа. На практика, ако двата компютъра са отдалечени, със свои реални IP адреси, ще е необходимо да настроите т.нар. Port Forwarding на вашия рутер, свързан към компютъра, който ще има сървърна роля (т.е. там където ще е микроконтролера). Фиксирайте едно IP от DHCP pool-а за вашия MAC адрес, след което с това IP и порт 3333 направете Port Forwarding. Когато трябва да се свържете с клиента GDB, укажете реалното IP (т.е. адреса на рутера), а не локалното, което ви е дадено от DHCP.

9. Настройка на развойната среда Eclipse за работа с GDB и OpenOCD. Развойната среда с отворен код Eclipse поддържа toolchain-а GCC и е добре интегриран с него. Вземете архивния файл на Eclipse и го разархивирайте (най-новата версия може да бъде свалена от [3], от раздела “Eclipse IDE for C/C++ Developers”). Влезте в новосъздадената папка “eclipse” и натиснете два пъти върху файла “eclipse”. Укажете работна директория на Desktop-а и натиснете OK.
За да отворите вашия проект, изберете File → Import → C/C++ → Existing Code as Makefile Project → Next → Existing Code Location → Browse → укажете пътя до вашия проект от миналите задачи, в същия прозорец изберете Cross GCC, задайте примерно име на проекта в полето Project Name → Finish
Ако сега натиснете бутона за build-ване (иконка на чук) в таб-а Console ще ви излезе грешката:

```
make: arm-none-eabi-gcc: Command not found
```

Това е нормално, защото в средата (environment) на операционната система може да не е указан пътя до GCC. Това може да бъде направено от Eclipse по следния начин: десен бутон на top-level директорията на току-що импортнатия ви проект → Properties → C/C++ Build → Environment → изберете променливата PATH → Edit → в края на полето Value добавете двуеточие и укажете пътя до GCC → OK → OK → Build. Сега не би трябвало да излязат грешки, а в конзолата ще видите това, което досега виждахте в терминала на Линукс.

Забележка: за да разберете пътя до компилатора, в Линукс терминал напишете `which arm-none-eabi-gcc`.

За да може Eclipse да използва графичен front-end за GDB в режим на TCP/IP комуникация първо трябва да се инсталира Java Plug-in наречен “GDB Hardware Debugging”. Това може да стане от Help → Install New Software → Work with → Изберете от падащото меню Kepler - http://download.eclipse.org/releases/kepler (заменете името на версията Kepler с името на вашата конкретна версия) → в полето “Type filter text” напишете “GDB Hardware Debugging” и натиснете Enter → Ще се появи категория (за конкретната версия името ѝ е “Mobile and Device Development”), която трябва да отворите и да сложите тикче върху “C/C++ GDB Hardware Debugging” → натиснете Next и следвайте указанията за инсталацията.
Когато инсталацията приключи, средата ще се рестартира. Тогава от Run → Debug Configurations → кликнете два пъти върху GDB Hardware Debugging → ще ви се отвори таб New Configuration, който можете да конфигурирате от десния панел.
Най-важните полета, които трябва да попълните са:

- C/C++ Application – път до изпълнимия файл **с дебъгерна информация** main.axf
- Project – директория, в която се намират сорс файловете на проекта;
- GDB Command – да се укаже точния префикс на GDB командата – в случая arm-none-eabi-gdb;
- Use remote target – това тикче трябва да е сложено, за да укажем TCP/IP връзка;
- JTAG Device – избира се Generic TCP/IP;
- Host name or IP address – указва се IP адреса на сървъра, т.е. на компютъра, на който върви вече пуснат OpenOCD;
- Port number – номер на порт, на който се очакват връзки от сървъра, за OpenOCD видяхме, че това е 3333;
- Load Image и Load Symbols – и двете трябва да сочат към изпълнимия файл, т.е. “Use project binary” (тези радио бутони са еквивалентни на командите load и file от GDB, които тествахме по-рано);

- Set breakpoint at – пише се main, аналогично на командата breakpoint main;
- Run commands – добре е да се запишат monitor init и monitor reset halt, за да започваме винаги от ресет хендлера.

След като сме готови с всички настройки по менютата може да натиснем debug, което ще програмира Flash паметта на микроконтролера през мрежата и ще изкара графичен интерфейс на GDB, с помощта на който може да се дебъгне програмата. Забележете, че сега за всяка команда си има по един бутон, с който лесно можете да дебъгвате.

Тествайте различните бутони на графичния дебъгер. Сложете точки на прекъсване, наблюдавайте локални и глобални променливи, и т.н.

Забележка: дебъг режимът може да бъде използван и само на един компютър, но въпреки това отново дебъгването се нарича “remote”, защото протоколът си остава TCP/IP. Ако желаете да дебъгвате на един компютър, вместо IP на сървър напишете IP на виртуалния ви мрежови интерфейс 127.0.0.1 (също известен като localhost), като не забравяйте преди това да стартирате OpenOCD на същия този компютър.

Забележка: в последните години се появиха много Plugin-и за Eclipse, които са специално оптимизирани и с повече възможности за работа с OpenOCD. Такъв е проектът GNU MCU on Eclipse [4].

10. Комбинацията GDB + OpenOCD може да бъде използвана и при дебъгване на микроконтролер с операционна система [5]. За целта сървърът OpenOCD трябва да бъде конфигуриран предварително за конкретната ОС. В настоящата задача ще бъде показано как се работи с FreeRTOS, но настройките са аналогични и за други поддържани ОС. Към момента на писането на този документ, поддържаните ОС са: eCos, ThreadX, FreeRTOS, Linux, ChibiOS, embKernel, mqx, uCOS-III.
За FreeRTOS дебъг са необходими три настройки:

- да бъде включен следния код някъде в програмата ви

```
#include "FreeRTOS.h"

#ifdef __GNUC__
#define USED __attribute__((used))
#else
#define USED
#endif

const int USED uxTopUsedPriority = configMAX_PRIORITIES – 1;
```

- да бъде предаден аргумент на линкера (през компилатора)

```
-Wl,--undefined=uxTopUsedPriority
```

- да бъде указан вида на операционната система в OpenOCD посредством командата configure, предшествана от името на дебъгваният микроконтролер (използва се OpenOCD променлива)

  $_TARGETNAME configure -rtos FreeRTOS

Вземете предварително компилираният проект от директория **03_10** и използвайте готовия .elf файл (от директория 03_10/Debug). Разгледайте main.c – програмата инициализира микроконтролера и пуска две задачи да се изпълняват в паралел. Всяка задача мига различен светодиод на платката.

```
openocd -f board/stm32f7discovery.cfg (в отделен терминал)
arm-none-eabi-gdb
target remote localhost:3333
monitor $_TARGETNAME configure -rtos FreeRTOS
monitor reset halt
load free_rtos_debug.elf
file free_rtos_debug.elf
continue
CTRL + C
info threads
```

където последната команда ще покаже списък с всички задачи, които се изпълняват в момента от ядрото на ОС. За да изберем фокус върху конкретна задача, трябва да се използва командата:

```
thread <номер на задача>
```

11. Тествайте командата за извличане на символи от .elf файл [6]. Под символи се разбират имена на всички функции и глобални променливи в обектовия файл. С Линукс терминал влезте в директорията с .elf файла на примера с операционната система и изпълнете командата:

```
arm-none-eabi-nm -a -l free_rtos_debug.elf
```

което ще принтира адрес в паметта, където ще се намира символа при изпълнението на програмата, буква-спецификатор, която указва вида на символа, името на символа, така както е дадено от програмиста и накрая (най-дясна колонка) файлът, в който е деклариран символът. Таблицата по-долу показва обобщение на символните спецификатори. Главна буква указва външен (глобален) символ, докато с малка буква се указва локален символ.

Ако желаете, можете да подредите символите по адреси, а не по азбучен ред на имената им. Това става с аргумента -n, или -v, или --numberic-sort.

12. Тествайте командата за статичен анализ на използваната памет arm-none-eabi-size [7]. С нейна помощ може да се определи размера на .text, .data и .bss регионите. Регионите stack и heap зависят от изпълнението на програмата и

Символен спецификатор	Значение
A	Абсолютна стойност на символа.
B, b	Символът ще се намира в .bss региона на паметта.
C	Неинициализиран символ.
D, d	Символът ще се намира в .data региона на паметта.
G, g	Символът ще се намира в .data региона за малки обекти (глобална int променлива е малък обект, глобален масив с int е голям обект).
i	Символът се намира в специална секция от DLL. Адресът е условен и ще бъде известен едва по време на изпълнението на програмата.
I	Символът е указател към друг символ.
N	Символът е за подпомагане на дебъгването.
n	Символът е в .const data региона на паметта
p	Символът се намира в stack unwinding региона на паметта (C++ деструктори).
R, r	Същото като n.
S, s	Символът ще се намира в .bss региона за малки обекти.
T, t	Символът ще се намира в .text региона на паметта.
U	Символът е неуточнен.
u	Символът е уникален глобален символ. Линкерът ще се погрижи в цялото адресно поле да няма други символи с такова име.
V, v	Символ, деклариран като weak.
W, w	Същото като V.
?	Символът е неуточнен или специфичен

	за дадения обектов файл.

техния размер може да бъде определен приблизително със симулация. Поради сложността на проблема, GCC няма програма, която да е способна на това. Изпълнете програмата с .elf файла на операционната система:

```
arm-none-eabi-size free_rtos_debug.elf
```

В миналото лабораторно упражнение беше споменато, че байтовете, които ще се запишат във Flash паметта са:

FLASH total = text + data

По подразбиране командата използва форматът Berkeley за изобразяване на данните. По-детайлна разбивка на секциите от паметта дава форматът System V:

```
arm-none-eabi-size --format=sysv free_rtos_debug.elf
```

където по-горната формула може да се разшири:

FLASH total = isr_vector + text + rodata + ARM.extab + ARM + preinit_array + init_array + fini_array + data

13. Програмата arm-none-eabi-objcopy копира съдържанието на един обектов файл в друг [8]. В процеса на копиране може да се приложат трансформации на данните. Именно заради това тази програма се използва често. Най-широкото ѝ използване е за преобразуване на обектови файлове с дебъгерна информация в обектови файлове без дебъгерна информация (“чисти” двоични файлове). Или накратко казано – с нейна помощ от .axf (.elf) файл се прави .bin файл.

Направете от .elf файлът един .bin, един .hex и един .rec файл като знаете аргументите на командата:

arm-none-eabi-objcopy -O [вид-изходен-файл] [име-на-входния-файл] [име-на-изходния-файл]

където “вид-изходен-файл” може да бъде binary, srec, ihex и др. (всички видове можете да видите с аргумента –info).

ВНИМАНИЕ: първо направете .bin файлът, след това от него направете другите два, като изрично укажете вид-входен-файл да е binary:

arm-none-eabi-objcopy -I [вид-входен-файл] -O [вид-изходен-файл] [име-на-входния-файл] [име-на-изходния-файл]

* * *

[1] S. Oliver, O. Harboe, D. Ellis, D. Brownell, “Open On-Chip Debugger: OpenOCD's User's Guide”, http://www.openocd.org, 2017.
[2] https://www.gnu.org/software/gdb/documentation
[3] https://www.eclipse.org
[4] https://eclipse-embed-cdt.github.io
[5] https://mcuoneclipse.com/2016/04/09/freertos-thread-debugging-with-eclipse-and-openocd
[6] https://sourceware.org/binutils/docs/binutils/nm.html
[7] https://embarc.org/man-pages/binutils/size.html
[8] https://sourceware.org/binutils/docs/binutils/objcopy.html

гл. ас. д-р инж. Любомир Богданов, 2021 г.