



**УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**



Димитрије Ћук

**Од изворног С кода до извршне бинарне
слике - компилација и распоред у меморији
микроконтролера**

Дипломски рад

Нови Сад 2025.



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР :	
Идентификациони број, ИБР :	
Тип документације, ТД :	Монографска публикација
Тип записа, ТЗ :	Текстуални штампани примерак
Врста рада, ВР :	Дипломски рад
Аутор, АУ :	Димитрије Ћук
Ментор, МН :	проф. др Дарко Марчетић
Наслов рада, НР :	Од изворног С кода до извршне бинарне слике - компилација и распоред у меморији микроконтролера
Језик публикације, ЈП :	српски
Језик извода, ЈИ :	српски
Земља публикавања, ЗП :	Република Србија
Уже географско подручје, УГП :	АП Војводина
Година, ГО :	2024.
Издавач, ИЗ :	Ауторски репринт
Место и адреса, МА :	Факултет техничких наука, 21000 Нови Сад, Трг Доситеја Обрадовића 6
Физички опис рада, ФО : (поглавља/страна/ цитата/табела/слика/графика/прилога)	(9/72/15/2/35/0/0)
Научна област, НО :	Електротехничко и рачунарско инжењерство
Научна дисциплина, НД :	Рачунарске науке и уграђени системи
Предметна одредница/Кључне речи, ПО :	Меморија у микроконтролеру, компајлер и компајлирање, линкер и линковање
УДК	
Чува се, ЧУ :	Библиотека ФТН, Трг Доситеја Обрадовића 6, Нови Сад
Важна напомена, ВН :	
Извод, ИЗ :	Циљ рада је да се систематски прикаже цео ток развоја embedded софтвера – од писања изворног кода на С језику до добијања финалне извршне бинарне слике која се уписује у меморију микроконтролера. Конкретно, рад обухвата детаљан приказ компилационог ланца уз коришћење GCC алатке (GNU компајлера) и анализу меморијског распореда кроз подешавање линкерске скрипте. Фокус је на томе како се изворни С код преводи (преко фазе препроцесирања, компилације и асемблирања) у објектне датотеке, затим линковањем обликује у извршну ELF датотеку, која се најзад конвертује у hex или bin формат погодан за учитавање у микроконтролер. Рад настоји да прикаже повезаност свих корака – од нивоа изворног кода до коначне бинарне слике, наглашавајући улогу сваког елемента у ланцу алата.
Датум прихватања теме, ДП :	10.09.2024.
Датум одбране, ДО :	27.09.2024.
Чланови комисије, КО :	Председник: др Владимир Поповић, доц., ФТН Нови Сад
	Члан: др Владимир Рајс, ван. проф., ФТН Нови Сад
	Члан:
	Члан:
	Члан, ментор: др Дарко Марчетић, ред. проф., ФТН Нови Сад
	Потпис ментора




UNIVERSITY OF NOVI SAD • FACULTY OF TECHNICAL SCIENCES

21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, ANO :	
Identification number, INO :	
Document type, DT :	Monographic publication
Type of record, TR :	Textual Printed Material
Contents code, CC :	Bachelor thesis
Author, AU :	Dimitrije Ćuk
Mentor, MN :	prof. dr Darko Marčetić
Title, TI :	Memory Mapping in a Microcontroller and the Use of Digital Signatures
Language of text, LT :	Serbian
Language of abstract, LA :	Serbian
Country of publication, CP :	Republic of Serbia
Locality of publication, LP :	AP of Vojvodina
Publication year, PY :	2024.
Publisher, PB :	Author's reprint
Publication place, PP :	Faculty of technical sciences, 21000 Novi Sad, Trg Dositeja Obradovića 6
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	(9/72/15/2/35/0/0)
Scientific field, SF :	Electrical and computer engineering
Scientific discipline, SD :	Computer science and embedded systems
Subject/Key words, S/KW :	From Source C Code to Executable Binary Image – Compilation and Memory Layout in Microcontrollers
UC	
Holding data, HD :	Library of Faculty of technical sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, N :	
Abstract, AB :	The aim of this paper is to systematically present the entire process of embedded software development—from writing source code in the C language to obtaining the final executable binary image that is programmed into the microcontroller's memory. Specifically, the paper includes a detailed overview of the compilation toolchain using the GCC tool (GNU Compiler) and an analysis of the memory layout through linker script configuration. The focus is on how the source C code is translated (through the preprocessing, compilation, and assembly phases) into object files, then shaped by linking into an executable ELF file, which is finally converted into a hex or bin format suitable for loading into the microcontroller. The paper seeks to demonstrate the interconnection of all steps—from the level of source code to the final binary image—emphasizing the role of each element in the toolchain.
Accepted by the Scientific Board on, ASB :	10.09.2024.
Defended on, DE :	27.09.2024.
Defended Board, DB :	President: dr Vladimir Popović, assist. prof., FTN Novi Sad
	Member: dr Vladimir Rajs, assoc. prof., FTN Novi Sad
	Member:
	Member:
	Member, mentor: dr Darko Marčetić, full prof., FTN Novi Sad
	Menthor's sign

	УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА	Број:
	21000 НОВИ САД, Трг Доситеја Обрадовића 6	012-40/1732
	ЗАДАТАК ЗА ДИПЛОМСКИ РАД	Датум: 10.09.2024.

(Податке уноси предметни наставник - ментор)

СТУДИЈСКИ ПРОГРАМ:	Е1 – енергетика, електроника, телекомуникације
РУКОВОДИЛАЦ СТУДИЈСКОГ ПРОГРАМА:	др Милан Сечујски

Студент:	Димитрије Ћук	Број индекса:	ЕЕ 3/2016
Област:	Рачунарске науке и уграђени системи		
Ментор:	др Дарко Марчетић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ДИПЛОМСКИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА: <ul style="list-style-type: none"> - проблем – тема рада; - начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна; 			

НАСЛОВ ДИПЛОМСКОГ РАДА:

Од изворног С кода до извршне бинарне слике - компилација и распоред у меморији микроконтролера
--

ТЕКСТ ЗАДАТКА:

--

Руководилац студијског програма:	Ментор рада:
др Милан Сечујски	др Дарко Марчетић

Примерак за: <input type="radio"/> - Студента; <input type="radio"/> - Ментора
--

Садржај

1. Увод.....	8
1.1. Контекст и значај теме.....	8
1.2. Циљ рада.....	8
1.3. Методологија	9
1.4. Објашњење термина и скраћеница	9
1.4.1. Хардвер и архитектура.....	9
1.4.2. Основни појмови софтверског система	10
1.4.3. Софтверски слојеви и библиотеке	10
1.4.4. Компилациони алати и процеси.....	11
1.4.5. Формати излазних датотека	11
1.5. Релевантност теме.....	12
2. Улога C језика у програмирању микроконтролера	13
2.1. Историјска еволуција C језика у embedded окружењима.....	13
2.2. Кључне особине C језика у embedded контексту	14
2.3. Упоредна анализа C језика и алтернативних језика	14
3. Организација C изворног кода за embedded окружења.....	15
3.1. Основне компоненте embedded пројекта	16
3.1.1. main.c (улазна тачка програма)	16
3.1.2. Модули и драјвери (.c/.h парови)	17
3.1.3. startup.s (векторска табела, Reset рутина, итд.).....	18
3.1.4. linker.ld (меморијско мапирање)	19
3.1.5. system_*.c (иницијализација такта, PLL, напајања).....	20
3.1.6. Makefile (компилација и линковање)	21
3.2. Употреба CMSIS и HAL слојева.....	23
3.2.1. CMSIS (Cortex Microcontroller Software Interface Standard)	23
3.2.2. HAL (Hardware Abstraction Layer)	24
3.3. Стил програмирања и стандардизација.....	28
3.3.1. Индустијски стандарди кодирања за безбедност и поузданост.....	28
3.3.2. Конзистентност стила и одрживост кода	29
3.4. Приступ меморијски мапираним регистрима.....	30
3.4.1. Меморијски мапирани улази и излази у микроконтролерима.....	30
3.4.2. Адресни простор и распоред периферија	30
3.4.3. Приступ регистрима у програму (C језик).....	31
3.4.4. Моделирање хардверских регистра у C	31
3.4.5. Предности и значај апстракције регистра.....	32
4. Фазе компилације (превођења) у GCC.....	33
4.1. Препроцесирање.....	35
4.1.1. Улога препроцесора и резултујући код.....	35
4.1.2. Основни пример позива препроцесора	35
4.1.3. Проширени позив са укључивањем путања и макроа	36
4.1.4. Значај препроцесирања у embedded пројекту – аналитички осврт.....	37
4.2. Компилација (превођење C кода у асемблер).....	38
4.2.1. Улога фазе компилације и излазни асемблерски код	38
4.2.2. Најједноставнији пример превођења у асемблер	38

4.2.3. Додавање архитектурских опција (умерено сложен пример).....	39
4.2.4. Опције за оптимизацију, дебаг и секционисање кода (средње сложен пример).....	40
4.2.5. Додавање путања до заглавља и условних макроа (комплекснији пример).....	41
4.2.6. Комплетан build позив – индустријски сценарио (најсложенији облик).....	42
4.3. Асемблирање (превођење асемблерског кода у објектни модул).....	44
4.3.1. Улога асемблера и објектни фајл (.o).....	44
4.3.2. Најједноставнији позив асемблера.....	44
4.3.3. Спецификација архитектуре при асемблирању.....	45
4.3.4. Укључивање оптимизационих и дебаг информација у објектни фајл.....	46
4.3.5. Асемблер са додатним include путањама и макроима.....	47
4.3.6. Индустријски позив асемблера (пуно build окружење).....	48
4.4. Линковање.....	50
4.4.1. Улога линкера и повезивање објектних модула.....	50
4.4.2. Основни пример линковања једног модула.....	51
4.4.3. Коришћење линкерске скрипте за меморијски распоред.....	52
4.4.4. Постепена надоградња линкер команде (пример).....	56
4.5. Интеграција фаза компилације у оквиру GCC алатке.....	59
5. Формати резултујућих датотека.....	60
5.1. ELF формат.....	62
5.1.1. Структура формата.....	62
5.1.2. Улога у развојном току.....	63
5.1.3. Предности и мане.....	63
5.1.4. Употреба у контексту микроконтролера (CYT2BL5CAS).....	64
5.1.5. Веза са GNU алатима.....	64
5.2. Intel HEX формат.....	65
5.2.1. Структура формата.....	65
5.2.2. Улога у развојном току.....	66
5.2.3. Предности и мане.....	67
5.2.4. Употреба у контексту микроконтролера (CYT2BL5CAS).....	67
5.2.5. Веза са GNU алатима.....	68
5.3. RAW бинарни формат (.bin).....	69
5.3.1. Структура формата.....	69
5.3.2. Улога у развојном току.....	70
5.3.3. Предности и мане.....	70
5.3.4. Употреба у контексту микроконтролера (CYT2BL5CAS).....	71
5.3.5. Веза са GNU алатима.....	71
5.4. Motorola S-Record формат (S19).....	72
5.4.1. Структура формата.....	72
5.4.2. Улога у развојном току.....	73
5.4.3. Предности и мане.....	73
5.4.4. Употреба у контексту микроконтролера (CYT2BL5CAS).....	74
5.4.5. Веза са GNU алатима.....	74
5.5. Критеријуми избора формата у производном току.....	75
6. Практична анализа GNU binutils алата за обраду објектних и бинарних датотека.....	78
6.1. arm-none-eabi-objcopy: креирање HEX, BIN и S19 датотека.....	80
6.1.1. ARM-специфична варијанта алата.....	80
6.1.2. Основне могућности алата.....	81
6.1.3. Практичне напомене.....	81

6.2. arm-none-eabi-readelf: испитивање интерне структуре ELF фајлова.....	82
6.2.1. Преглед ELF заглавља и секција.....	82
6.2.2. Симболи и програмски сегменти.....	82
6.2.3. Релокације, динамичке информације и дебаг подаци.....	83
6.2.4. Примена у анализи меморијског распореда	83
6.2.5. Практични примери	83
6.3. arm-none-eabi-nm: преглед симбола у бинарним датотекама	85
6.3.1. Приказ симбола и њихова класификација	85
6.3.2. Практична вредност у анализи меморијског распореда.....	86
6.3.3. Напредне опције за дубљу анализу	86
6.3.4. Примери употребе у embedded контексту.....	86
6.4. arm-none-eabi-size: анализа меморијске потрошње по секцијама.....	88
6.4.1. Основни принцип рада.....	88
6.4.2. Аналитички значај података	89
6.4.3. Напредне могућности	89
6.4.4. Практична примена у embedded развоју	89
6.5. arm-none-eabi-objdump: дубинска анализа ELF структуре	91
6.5.1. Дисасемблирање машинског кода	91
6.5.2. Хексадецимални приказ садржаја.....	91
6.5.3. Табела симбола и метаподаци ELF датотеке	92
6.5.4. Напредне могућности	92
6.5.5. Практична примена у embedded контексту.....	92
6.6. Комплементарна употреба GNU алата у embedded развоју	94
7. Линкерска скрипта.....	96
7.1. Пример линкер скрипте за GNU C компајлер	97
7.2. Почетне директиве и дефиниције симбола	105
7.3. MEMORY дефиниција	111
7.3.1. Flash меморија	112
7.3.2. RAM меморија	113
7.3.3. Специјални меморијски региони (Supervisory Flash и eFuse)	113
7.4. SECTIONS расподела	115
7.4.1. Секција .cy_app_header: заглавље апликације и векторска табела.....	121
7.4.2. Секција .text: програмски код, векторска табела и константни подаци	121
7.4.3. Секције .ARM.extab и .ARM.exidx: табеле изузетака и stack unwinding	124
7.4.4. Секција .copy.table: табела за копирање секција из флеша у RAM.....	125
7.4.5. Секција .zero.table: иницијализација BSS региона занулирањем	127
7.4.6. Секција .ramVectors: RAM копија векторске табеле	128
7.4.7. Секција .data: иницијализовани подаци, низови конструктора и RAM функције	130
7.4.8. Секција .noinit: променљиве које задржавају вредности преко ресета	133
7.4.9. Секција .bss: неиницијализовани подаци у RAM-у	134
7.4.10. Секције .heap и .stack_dummy: динамичка меморија и стек у RAM-у	135
7.4.11. Специјални меморијски региони: Supervisory Flash, eFuse и cymeta	139
7.4.12. Закључак о SECTIONS расподели	143
7.5. Закључак о линкер скрипти	144
8. Закључак.....	145
9. Литература	146

1. Увод

1.1. Контекст и значај теме

Микроконтролери као уграђени рачунарски системи данас су свеprisутни у аутомобилској индустрији, индустрији аутоматике, и IoT (Internet of Things) системима. Од аутоматизованих производних погона до „паметних” уређаја и сензорских мрежа, ови мали контролери управљају широким спектром функција у реалном времену. Процењује се да је број таквих уређаја и њихова комплексност у сталном порасту, што чини знање о развоју програма за микроконтролере све важнијим. Посебан акценат је на **bare-metal** програмском моделу – односно програмирању без оперативног система – које омогућава директну контролу хардвера ради постизања максималне ефикасности. Овакво програмирање блиско хардверу сматра се кључном вештином јер пружа увид у рад система на најнижем нивоу и омогућава оптимизацију перформанси и потрошње енергије, без сувишног трошења ресурса. У контексту све веће примене уграђених система, дубље разумевање процеса развоја софтвера за те уређаје је од суштинске важности за пројектовање поузданих и безбедних система.

Језик C се издваја као доминантан у области програмирања микроконтролера захваљујући својој ефикасности и близини хардверу. Овај језик генерише машински код минималног оверхеда, који се извршава готово једнако брзо као ручно писани асемблер, што је од пресудне важности за задатке који се извршавају у реалном времену. Истовремено, C је знатно читљивији и одрживији од чистог асемблера, што олакшава тимски рад на развоју фирмвера (firmware). Због тих својстава, C (заједно са својим надскупом C++) постао је стандард у развоју **bare-metal** софтвера – према проценама индустрије, око 80% свих уграђених система данас користи управо C језик. Његова преносивост и стандардизација (нпр. кроз ISO/IEC C стандарде и ARM-ову CMSIS спецификацију) додатно су учврстили улогу C језика у embedded домену. Стога, савремени инжењери уграђених система морају добро познавати не само сам језик, већ и читав процес преводјења C кода у извршни облик прилагођен мети (таргету) – хардверској архитектури микроконтролера.

1.2. Циљ рада

Циљ овог рада је да се систематски прикаже цео ток развоја embedded софтвера – од писања изворног кода на C језику до добијања финалне извршне бинарне слике која се уписује у меморију микроконтролера. Конкретно, рад обухвата детаљан приказ компилационог ланца уз коришћење GCC алатке (GNU компајлера) и анализу меморијског распореда кроз подешавање линкерске скрипте. Фокус је на томе како се изворни C код преводи (преко фазе препроцесирања, компилације и асемблирања) у објектне датотеке, затим линковањем обликује у извршну ELF датотеку, која се најзад конвертује у hex или bin формат погодан за учитавање у микроконтролер. Рад настоји да прикаже међусобну повезаност свих корака – од нивоа изворног кода до коначне бинарне слике – наглашавајући улогу сваког елемента у ланцу алата.

1.3. Методологија

Приступ истраживању и излагању је дескриптивно-аналитички, ослоњен превасходно на званичну документацију и стандарде. Користе се референтни извори као што су техничка упутства компаније **ARM** (за архитектуру процесора и стандарде попут ARM Cortex-M архитектуре), документација самог **GCC** компилационог окружења и пратећих **GNU** алата, одговарајући IEEE/ISO стандарди (нпр. стандарди програмског језика C и бинарних формата), као и технички подаци произвођача микроконтролера (Infineon) – укључујући технички лист (datasheet) и референтни приручник (Technical Reference Manual – TRM) за конкретни модел чипа. Анализа је спроведена кроз праћење конкретног пример пројекта и теоријско објашњење сваке фазе превођења (компилације) и распореда у меморији. Теоријски садржај је периодично илустрован практичним примером из пројекта за микроконтролер **CYT2BL5CAS** (породица TRAVEO™ T2G-B-E), развијеног у оквиру ModusToolbox™ окружења уз коришћење одговарајућег *Board Support Package*-а (BSP) **KIT_T2G-B-E_LITE**. Овај пројекат је преузет из званичног “Hello World” темплејта и служи да демонстрира стварну имплементацију концепата обрађених у раду, чиме се обезбеђује спој између теоријских појмова и практичне реализације на циљном хардверу.

1.4. Објашњење термина и скраћеница

1.4.1. Хардвер и архитектура

- **ARM** – Првобитно акроним за *Advanced RISC Machines*, данас представља и назив компаније *Arm Ltd.* и породице RISC процесорских архитектура. ARM архитектуре, нарочито линија Cortex-M, доминирају у свету микроконтролера због повољног односа између перформанси, потрошње енергије и цене. Примену налазе у мобилним уређајима, аутомобилским системима, индустријској аутоматици и IoT уређајима.
- **TRAVEO™ T2G** – Породица 32-битних микроконтролера компаније *Infineon*, заснована на ARM Cortex архитектури. Другу генерацију (Traveo II) карактеришу високе перформансе, сигурносне карактеристике, као и богата периферијска подршка за аутомобилске примене. Обухвата више потпородица попут Body High и Body Entry.
- **CYT2BL5CAS** – Конкретан модел микроконтролера из TRAVEO™ T2G Body Entry серије. Поседује два процесорска језгра (Cortex-M4 до 160 MHz и Cortex-M0+ до 100 MHz), 4 MB Flash меморије и велики број интегрисаних периферија. Представља циљни хардвер у оквиру овог рада.
- **Адресни простор** – Целокупан скуп адреса које микроконтролер може да адресира. У Cortex-M архитектури адресни простор је линеаран и обједињује програмски код, RAM, регистре периферија и системске регистре.

- **Периферије** – Хардверске компоненте интегрисане у микроконтролер које омогућавају комуникацију, управљање, мерење и генерисање сигнала (нпр. UART, ADC, PWM, GPIO, Timers). Контролишу се преко меморијски мапираних регистара.
 - **Регистри** – Меморијске јединице фиксне дужине којима се управља хардвером. Разликују се системски регистри (нпр. SCB->VTOR) и регистри специфични за периферије. Често се користе у low-level програмирању и у иницијализацији система.
-

1.4.2. Основни појмови софтверског система

- **Фирмвер (firmware)** – Специфичан тип софтвера који се трајно налази у Flash меморији и блиско је повезан са конкретним хардвером. Задужен је за управљање основним функцијама уређаја и често не зависи од присуства оперативног система.
 - **bootloader** – Иницијални програм који се извршава по укључењу уређаја. Налази се у ROM/Flash меморији и служи за читавање и покретање апликативног фирмвера, као и за могућност надоградње (нпр. преко UART, CAN, OTA). Може да врши криптографску валидацију садржаја.
 - **Дигитални потпис** – Криптографски механизам који обезбеђује аутентичност и интегритет бинарног кода. Најчешће се заснива на алгоритмима јавног кључа (нпр. RSA, ECDSA) и користи се у secure boot механизмима ради спречавања извршавања неауторизованог фирмвера.
-

1.4.3. Софтверски слојеви и библиотеке

- **CMSIS (Cortex Microcontroller Software Interface Standard)** – Званични ARM стандард који дефинише API, заглавља и структуре за приступ системским и периферним регистрима. Омогућава униформно програмирање Cortex-M уређаја независно од произвођача.
- **HAL (Hardware Abstraction Layer)** – Софтверски слој који апстрахује приступ периферијама и хардверским ресурсима, пружајући унификоване API функције. HAL имплементације се често базирају на CMSIS-у и испоручују се као део SDK-а произвођача.
- **Векторска табела** – Структура смештена на почетку меморијског простора (обично у Flash), која садржи адресе прекидних рутина. Први унос указује на почетак стека, други на функцију Reset_Handler, док следећи одговарају одређеним прекидима.
- **Reset рутина** – Почетна функција (Reset_Handler) која се извршава по ресетовању уређаја. Задужена је за иницијализацију меморијских секција, системских параметара и покретање главне апликације (main). Представља део startup кода.

1.4.4. Компилациони алати и процеси

- **GCC** – Скуп компајлера отвореног кода (GNU Compiler Collection), који подржава различите језике (C, C++, Fortran итд.) и хардверске архитектуре. У embedded контексту, користи се GCC ARM toolchain за генерисање машинског кода за микроконтролере.
 - **GNU** – Пројекат слободног софтвера чији је циљ био развој потпуно отвореног оперативног система. Изнедрио је многе алате попут GCC-а, GNU binutils-а и make-а, који чине окосницу embedded build система.
 - **GNU binutils** – Колекција алата за рад са бинарним датотекама: as (асемблер), ld (линкер), objcopy, objdump, nm, readelf итд. Користе се у фазама компилације, линковања и анализа извршних датотека.
 - **Препроцесирање** – Прва фаза компилационог процеса. Обрађује директиве као што су #include, #define, #ifdef, проширује макрое и формира јединствен .i фајл са чистим C кодом без препроцесорских ознака.
 - **Компилација** – Претварање .i фајла у асемблерски (.s). Анализира се синтакса и семантика C кода, врши се типска провера и генерише се машински независан асемблерски код.
 - **Асемблирање** – Преводи .s у објектни бинарни фајл (.o). У овој фази се генеришу машинске инструкције, секције, табеле симбола и relocation уноси.
 - **Мапирање меморије / Линковање** – Процес који спаја више .o и библиотека у једну .elf датотеку. Линкер, по инструкцијама из скрипте, смешта секције (.text, .data, .bss итд.) у одговарајуће адресне просторе.
 - **Линкерска скрипта** – Текстуална датотека која описује распоред и величине меморијских региона (MEMORY {}) и секција (SECTIONS {}). Одређује како ће објектни код бити постављен у Flash и RAM уређаја.
-

1.4.5. Формати излазних датотека

- **ELF** – *Executable and Linkable Format* је стандардни бинарни формат за Unix-подобне системе. ELF фајл садржи све секције програма, симболе и релокационе информације, што га чини погодним за дебаг и анализу.
- **Intel HEX формат** – ASCII формат у којем се бинарни подаци представљају као хексадекадне вредности по линијама (record-има). Сваки запис садржи адресу, тип података, сам садржај и контролни збир. Широко је подржан од стране програматора и дебагера.
- **Бинарни формат** – "RAW" представљање садржаја меморије, без заглавља и симболичких информација. .bin фајлови садрже само податке предвиђене за Flash и погодни су за директно програмирање.

- **Motorola S-Record формат** – Алтернативни текстуални формат сличан Intel HEX-у. Користи S0–S9 записе који садрже адресу, дужину, податке и контролни збир. Широко подржан у индустријским програматорима и legacy алатима.

1.5. Релевантност теме

Разматрана тематика директно одговара потребама реалног развоја софтвера у уграђеним окружењима, нарочито за системе који раде без оперативног система (bare-metal). Правилна организација изворног кода, разумевање формата резултујућих датотека и контрола над меморијским распоредом представљају предуслов за функционалан и безбедан рад микроконтролера у критичним применама – попут аутомобилских управљачких система, медицинских уређаја или индустријских контролера. Незнање или превид у овим областима може довести до тешко уочљивих грешака које компромитују поузданост система. Са друге стране, темељно разумевање компилационог процеса и меморијске структуре фирмвера отвара пут ка напреднијим темама. Конкретно, знања изложена у овом раду представљају основу за имплементацију **bootloader**-а (који захтева прецизно управљање меморијским адресним просторима и секцијама кода), за увођење механизма безбедности фирмвера (нпр. верификацију кода путем дигиталног потписа), као и за оптимизацију потрошње ограничених ресурса микроконтролера. На тај начин, ова тема је значајна не само теоријски, већ и практично – као неопходан део знања за инжењере који се баве развојем поузданог и сигурног embedded софтвера.

2. Улога C језика у програмирању микроконтролера

Језик C је најзаступљенији у програмирању микроконтролера због своје флексибилности и ефикасности (мали оверхед). Под флексибилношћу се подразумева могућност директног приступа хардверским ресурсима, као што су меморијске адресе и регистри. Са друге стране, мали оверхед значи да компајлирани код заузима минимално меморије и омогућава брзо извршавање. Захваљујући овим особинама, C омогућава оптимално коришћење ограничених ресурса карактеристичних за уграђене системе.

Практично све – од малих контролера у уређајима до оперативних система – може бити написано у C-у због његове преносивости и способности да уз минималне наредбе пружи максималну контролу над хардвером. У домену уграђених система, C пружа низак ниво апстракције: омогућава директан приступ меморијским адресама и периферијама, управљање битовима и регистрима, као и прецизну контролу над временски критичним секцијама кода. За разлику од виших језика, компајлер C језика генерише ефикасан машински код који се извршава готово једнако брзо као ручно писани асемблер, чиме је погодан за примене у реалном времену (real-time). Истовремено, C код је знатно читљивији и одрживији од чистог асемблера, што је важно при тимском развоју софтвера. Захваљујући овим особинама, C (и његов надскуп C++) је постао стандард у развоју фирмвера за микроконтролере, омогућивши и преносивост кода између различитих архитектура. Другим речима, C обезбеђује **директан приступ хардверу и високе перформансе**, што су кључни захтеви у embedded систему.

2.1. Историјска еволуција C језика у embedded окружењима

Језик C је развијен раних 1970-их у Bell Labs-у за потребе оперативног система UNIX, али је због своје минималистичке и ефикасне структуре веома брзо нашао примену у embedded системима. Током развоја микроконтролера и појаве 8-битних и 16-битних архитектура, C је потиснуо асемблер као доминантни језик због боље читљивости и лакше преносивости. Данас, C представља основну полазну тачку за развој софтвера у реалновременским (real-time) и ресурсно ограниченим системима, посебно захваљујући стандардизацији преко ISO/IEC 9899:2018 (C18) и спецификацијама као што је CMSIS (Cortex Microcontroller Software Interface Standard) од стране ARM-a.

Ознака **ISO/IEC 9899:2018 (C18)** односи се на званични међународни стандард за програмски језик C, који су усвојиле две признате организације за стандардизацију: *International Organization for Standardization (ISO)* и *International Electrotechnical Commission (IEC)*. Означење **9899** представља број стандарда који се односи на језик C, док **2018** указује на годину последње ревизије. Ознака **C18** је неформална, али широко прихваћена у стручној заједници, и односи се на ову верзију C стандарда. Стандард C18 представља мању ревизију претходне верзије C11 (ISO/IEC 9899:2011), фокусирајући се на техничке исправке, унапређење формалне прецизности и уклањање неусаглашености, без увођења нових синтаксних или семантичких елемената. Због тога се C18 сматра најстабилнијом и најсавременијом верзијом стандарда језика C која се тренутно примењује у индустријским и академским окружењима.

2.2. Кључне особине С језика у embedded контексту

У контексту програмирања микроконтролера, С се издваја следећим особинама:

- **Директан рад са меморијом** — Показивачи омогућавају манипулацију на нивоу бајта и регистра, што је неопходно за рад са периферијама. Кроз `volatile` квалификатор могуће је обезбедити исправно понашање при асинхроним изменама вредности (нпр. од стране хардвера или прекида).
- **Фино управљање меморијским распоредом** — Коришћењем `__attribute__((section("example")))`, програмер може утицати на то у коју меморијску регију ће одређена функција или променљива бити смештена, што је критично у систему без оперативног система.
- **Интеграција са асемблером** — Када је неопходна максимална оптимизација или директна манипулација регистрима, С омогућава уметање асемблерских инструкција (`__asm__`) или позиве на екстерне рутине, што га чини флексибилним алатом за развој близак хардверу (low-level programming).
- **Прецизна контрола времена извршавања** — У real-time системима детерминистичност је од пресудне важности. С не садржи runtime механизме попут garbage collector-а, чиме обезбеђује временски предвидљиво понашање.

2.3. Упоредна анализа С језика и алтернативних језика

Иако се у embedded индустрији појављују и други језици (нпр. Rust, Ada, Python/MicroPython), ниједан не достиже ниво подршке и зрелости који има С. Python се користи углавном у образовне и прототипске сврхе, Rust још увек има ограничену подршку за специфичне архитектуре и toolchain-ове, док је Ada присутна углавном у високо-регулисаним индустријама (авионика, нуклеарна техника).

С се тако намеће као оптималан компромис између перформанси, поузданости, контроле и одрживости. Он пружа довољну блискост хардверу за временски критичне апликације, а истовремено омогућава тимски развој, проверу стандарда као што су MISRA и лако повезивање са библиотекама и хардверским апстракционим слојевима.

3. Организација С изворног кода за embedded окружења

Развој софтвера за микроконтролере у програмском језику С захтева дисциплиновану и модуларну организацију изворног кода, посебно у условима где не постоји оперативни систем и где је програм одговоран за директно управљање хардвером — такозвано **bare-metal програмирање**. У овом контексту, коректна организација пројекта није само питање структуралне естетике, већ предуслов за исправно иницијализовање система, ефикасну меморијску расподелу и могућност проширивости и тестирања.

3.1. Основне компоненте *embedded* пројекта

Типичан C-пројекат за ARM Cortex-M4 микроконтролер у *bare-metal* окружењу састоји се из више међусобно повезаних компоненти. Произвођач микроконтролера обично испоручује почетни *startup* код (нпр. асемблерску или C датотеку) са векторском табелом прекида и *reset* рутином, као и одговарајућа CMSIS заглавља и HAL библиотеке за рад са периферијама. На основу тога, програмер развија сопствени изворни код у **main.c** датотеци и повезаним модулима (нпр. сензори, управљање мотором), укључујући потребне хедере за сваки модул. Поред тога, пројекат садржи и линкерску скрипту (нпр. **linker.ld**) која описује меморијске регије микроконтролера (Flash, RAM) и дефинише распоред секција програма (.text, .data, .bss, *stack* итд.) у те регије. Оваква структура обезбеђује да сваки део кода и података буде смештен на предвиђено место током линковања, што је основа за стабилан и поуздан *embedded* систем.

3.1.1. main.c (улазна тачка програма)

main.c је главна изворна јединица програма која садржи функцију `main()`, односно улазну тачку извршавања. У овој датотеци врши се иницијализација хардвера и система по покретању микроконтролера, након чега програм прелази у главну петљу (тзв. *super-loop*) у оквиру које се обавља његова главна функционалност. Типично се у `main()` функцији омогућавају прекиди и покрећу иницијализације свих потребних периферија, па затим следи бесконачна петља која одржава рад програма. Уколико систем користи RTOS, у `main()` се уместо бесконачне петље може стартовати *scheduler*, али у *bare-metal* приступу `main()` сам управља током извршавања.

У наставку је приказан један могући поједностављен пример структуре функције `main()`. После ресета, најпре се иницијализују плоча и периферије позивом функције за подешавање хардвера (у овом случају `cybsp_init()`), а резултат иницијализације се проверава. Потом се омогућавају глобални прекиди, након чега би следила подешавања комуникације (нпр. UART за *debug* излаз) и осталих уређаја, па улазак у главну петљу програма:

```
int main(void)
{
    result = cybsp_init(); /* Иницијализација платформе и периферија */

    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0); /* Заустави извршење ако иницијализација није успела */
    }

    __enable_irq(); /* Омогућавање глобалних прекида */

    /* Укључење UART-а и LED */
    cy_retarget_io_init_fc(...);
    cyhal_gpio_init(CYBSP_USER_LED, ...);
}
```



```

timer_init(); /* Старт тајмера који генерише прекид сваке секунде */

while (true)
{
    if (timer_interrupt_flag)
    {
        timer_interrupt_flag = false;
        cyhal_gpio_toggle(CYBSP_USER_LED); // трептање LED-ом
    }
}
}

```

Горњи пример демонстрира типичне кораке на почетку `main()` функције – иницијализацију хардвера и омогућавање прекида. Након тога, `main()` обично улази у бесконачну петљу (`while(1)` или `for(;;)`) у којој се обрађују догађаји или сензорски подаци, шаљу поруке, управља актуаторима и сл. (нпр. читавање УАРТ улаза и укључивање/искључивање LED диоде у примеру). *Напомена:* конкретна реализација `main()` може знатно да варира у зависности од пројекта – наведени код је само једна могућа варијанта имплементације.

3.1.2. Модули и драјвери (.c/.h парови)

Већи пројекти се организују на модуларан начин, тако да се поједине функционалне целине реализују у виду одвојених модула (нпр. `gpio.c`, `uart.c`, `sensor.c`, `motor_control.c`, итд.). Сваки такав модул обично долази у пару: изворна датотека са имплементацијом (.c) и одговарајућа заглавна датотека (.h) која декларише његов јавни интерфејс. Заглавни (.h) фајл садржи прототипове функција, декларације структура, *enum* типова и глобалних променљивих које модул излаже другим деловима програма. На тај начин се постиже јасна подела кода и боља могућност поновне употребе и тестирања – остале јединице укључују само потребне хедере и позивају функције модула преко дефинисаног интерфејса.

Пример једног таквог модула је драјвер за тајмер. У његовом заглављу може бити декларација функције за покретање тајмера, на пример:

```
cy_rslt_t cyhal_timer_start(cyhal_timer_t *obj);
```

Ова функција је у .h датотеци само најављена (прототип), а у изворној .c датотеци дата је њена реализација. У коду испод видимо делић имплементације функције `cyhal_timer_start` у оквиру драјвера тајмера: након провера објекта и услова, функција позива рутине блиске хардверу за подешавање периферије – укључивање бројача и стартовање тајмера:

```

if (CY_RSLT_SUCCESS == result)
{
    result = Cy_TCPWM_Counter_Init(obj->tcpwm.base, ... , config); /* Иницијализација хардверског
тајмер блока */
}

```

```

if (CY_RSLT_SUCCESS == result)
{
    Cy_TCPWM_Counter_Enable(obj->tcpwm.base, ... ); /* Enable тајмер (покретање бројања) */
}

```

У овом сегменту кода функција драјвера користи функције из произвођачке PDL библиотеке (Cy_TCPWM_Counter_Init/Enable) да конфигурише и покрене одговарајући тајмерски периферни блок микроконтролера. На тај начин се остварује апстракција – виши нивои кода позивају једноставну `cyhal_timer_start()` функцију, док она интерно обавља комплексне операције над регистрима. *Напомена:* структура модула и стил имплементације могу се разликовати; приказани пример је само један могући начин организације `.c/.h` пара датотека у склопу драјвера.

3.1.3. startup.s (векторска табела, Reset рутина, итд.)

Startup датотека (често названа **startup.s** за асемблерску или **startup.c** за C имплементацију) садржи код који се извршава први након укључења или ресетовања микроконтролера. Њене главне улоге су: (1) дефинисање **векторске табеле прекида**, која на познатој адреси (нпр. почетак Flash меморије) садржи почетне адресе свих прекидних рутина, укључујући и почетну вредност стек показивача и адресу *Reset_Handler*-а; (2) имплементација саме *Reset_Handler* рутине, која припрема извршно окружење пре него што се позове функција `main()`.

Векторска табела је низ од 32-битних вредности које одговарају почетном стек показивачу и адресама свих излазних тачака прекида. На примеру испод видимо почетак векторске табеле за један Cortex-M4 уређај – прва вредност је иницијални адресни врх стека (**Stack Top**), а затим следе адресе обрадних рутина: *Reset_Handler*, *Non-Maskable Interrupt* (NMI), *HardFault*, и осталих дефинисаних изузетака и прекида:

```

__Vectors:
__long   __StackTop      /* Почетна адреса стека */
__long   Reset_Handler  /* Reset Handler */
__long   CY_NMI_HANDLER_ADDR /* NMI Handler */
__long   HardFault_Handler /* Hard Fault Handler */
__long   MemManage_Handler /* MPU Fault Handler */
__long   BusFault_Handler /* Bus Fault Handler */
__long   UsageFault_Handler /* Usage Fault Handler */
...      /* (наставак листе прекида) */

```

Након векторске табеле, *startup* код реализује саму *Reset_Handler* функцију. Ова рутина се извршава на самом почетку (на њу упућује други елемент векторске табеле) и њен задатак је да припреми окружење за C програм. То типично обухвата: постављање почетног стека, копирање иницијализационих података из Flash у RAM (секција **.data**), брисање (иницијализација на нуле) неиницијализованих статичких променљивих (секција **.bss**), потенцијално омогућавање FPU јединице, подешавање векторске табеле ако се преселила у RAM, и позив функције за почетно подешавање система (нпр. `SystemInit()`). Тек након тога, *startup* рутина позива корисничку функцију `main()` и предаје јој даљу контролу извршавања програма. На крају *Reset_Handler*-а се обично

налази бесконачна петља као заштита ако `main` икада врати управљање (што се у исправном програму не дешава):

```
/* ... (иницијализација .data и .bss секција) ... */
#ifndef __NO_SYSTEM_INIT
bl  SystemInit      /* Позив функције за системску иницијализацију */
#endif

/* ... (позив конструктора C++ објеката) ... */
bl  __libc_init_array

/* Execute main application */
bl  main            /* Позив корисничке main() функције */

/* Call C/C++ static destructors */
bl  __libc_fini_array

/* Should never get here */
b   .              /* Бесконачна петља (dead loop) */
```

Горњи код илуструје завршни део *startup* секвенце: након припреме меморије, позива се `SystemInit` (осим ако није искључен макроом), затим рутина за статичке конструкторе (`__libc_init_array`), па корисничка функција `main`. По повратку из `main` (који се у правилу не дешава у *bare-metal* програмима), позвали би се деструктори статичких објеката и програм улази у бесконачну петљу. *Напомена*: конкретан садржај *startup* кода зависи од конкретног архитектурног језгра и алатног ланца – приказани пример одговара CMSIS шаблону за Cortex-M4 и једно могуће извођење *reset* рутине.

3.1.4. linker.ld (меморијско мапирање)

Линкерска скрипта или директива (најчешће названа **linker.ld**) одређује како ће се секције кода и података распоредити у физичкој меморији микроконтролера током процеса линковања. Она описује расположиве меморијске регије (нпр. флеш и рам) и правила смештања различитих секција програма у те регије. Тиме линкер зна тачно на које адресе треба ставити сваки део извршног кода и података, што је од критичне важности у *bare-metal* систему где нема оперативног система да динамички управља меморијом.

У делу ниже видимо пример дефиниције меморијских регија у линкерској скрипти. Дефинисана су два главна региона: FLASH (са атрибутима *rx* (*read execute*) – за извршавање и читање) од адресе `0x10000000` дужине `0x410000` бајтова, и RAM (са *rw* (*read write execute*) атрибутима) од адресе `0x08020000` дужине `0x5F800` бајтова. Ове адресе и величине одговарају конкретном микроконтролеру (овде пример двојезгарног система где CM4 језгро користи одређени део меморије):

```
ram (rwx) : ORIGIN = 0x08020000, LENGTH = 0x5F800
flash (rx) : ORIGIN = 0x10000000, LENGTH = 0x410000
sflash_user_data (rx) : ORIGIN = 0x17000800, LENGTH = 0x800 /* специјални Flash */
...
```

Након дефинисања меморије, линкерска скрипта описује распоред секција. На пример, код за Cortex-M4 језгро може поставити секцију **.text** (садржи извршни код програма) у Flash на адресу одмах након резервисаног простора за M0+ језгро (ако постоји). Секција **.data** (иницијализовани подаци) мора бити смештена у RAM, али њене иницијалне вредности треба сачувати у Flash – што се постиже директивом `AT> flash` у скрипти. Извод из линкер скрипте који то илуструје:

```
.data __ram_vectors_end__ : AT>flash {
    ...
    __data_end__ = .;
} > ram
```

У горњем примеру, секција **.data** се алоцира у RAM (ознака `> ram`), али јој је *Load Memory Address* постављена на Flash (`AT>flash`). То значи да ће сви бајтови **.data** секције бити уписани у извршну датотеку на одговарајућим Flash адресама, одакле ће их *startup* код копирати у RAM при покретању. Слично, секција **.bss** (неиницијализовани подаци) дефинише се са атрибутом `NOLOAD` и смешта у RAM, чиме линкер означава да за њу не треба резервисати простор у Flash фајлу већ ће бити само обележена за касније попуњавање нулама у RAM-у. Линкерска скрипта обично додељује и симболе као што су `__StackTop` и `__StackLimit` на крају RAM меморије, чиме се дефинише позиција и величина стека програма.

Добро осмишљена линкерска скрипта обезбеђује исправно мапирање целокупног програма у меморију микроконтролера. *Напомена:* иако постоје унапред припремљене генеричке скрипте, увек је потребно прилагодити их конкретном чипу (према подацима из *datasheet*-а) како би се сви сегменти (нпр. више блокова RAM-а, посебне меморије) исправно обухватили.

3.1.5. `system_*.c` (иницијализација такта, PLL, напајања)

Уз *startup* код, уобичајено је да постоји и посебна датотека назива облика **`system_<device>.c`**, која садржи функције за почетну конфигурацију система такта и напајања. ARM CMSIS стандард предвиђа функцију `SystemInit()` у овој датотеци, коју *startup* позива непосредно пре корисничког кода. Улога `SystemInit()` је да подеси основне параметре система: такт микроконтролера (нпр. учитава унутрашњи осцилатор или подешава PLL множилац и делитеље такта за жељену фреквенцију), подеси брзину рада Flash меморије (нпр. *wait-state*-ове) у складу са тактом, омогући FPU (уколико постоји) и припреми глобалну променљиву **`SystemCoreClock`** која садржи вредност фреквенције језгра. Ова датотека је специфична за сваки *device* и типично је испоручује произвођач – програмер је обично не мења, осим ако је потребно прилагодити такт нестандартно.

Уколико се користи произвођачки *Hardware Abstraction Layer* (HAL), део системске иницијализације може бити распоређен и у функције за иницијализацију плоче или периферија. На пример, Infineon-ова функција `cybsp_init()` позива низ потпроцедура које укључују подешавање хардвер менаџера ресурса и система напајања:

```
cy_rslt_t cybsp_init(void)
{
    cy_rslt_t result = cyhal_hwmgr_init();    /* Иницијализација менаџера хардверских ресурса */
    if (CY_RSLT_SUCCESS == result)
    {
        result = cyhal_syspm_init();          /* Иницијализација система напајања/такта */
    }
    ...
    return result;
}
```

Овај фрагмент кода илуструје да позивом једне функције (`cybsp_init`) у `main.c` заправо покрећемо вишеструка подешавања у позадини – од менаџмента тактова и напона до резервисања ресурса за вишејезгарне системе (нпр. функције `syscfg_config_init()` и друге у наставку кода). У класичној CMSIS поставци, сличне акције обавља `SystemInit()`, али у овом примеру оне су део HAL иницијализације специфичне за произвођача. *Напомена:* без обзира на конкретну реализацију, суштина `system_*.c` јесте да се сви кључни системски параметри микроконтролера подесе на самом почетку (пре апликационог кода), како би остатак програма могао да ради на предвидљивој тактној фреквенцији и конфигурацији.

3.1.6. Makefile (компилација и линковање)

Makefile представља скрипт за аутоматизацију процеса превођења кода и линковања у извршну бинарну слику. У GCC окружењу, *Makefile* прописује кораке: који се фајлови требају компајлирати, са којим опцијама, и како их затим повезати линкером. На пример, наредба:

```
arm-none-eabi-gcc -O2 -mcpu=cortex-m7 -o program.elf main.c uart.c startup.s -T linker.ld
```

илуструје како се у једном кораку могу обавити све фазе превођења – наведеном командом GCC ће аутоматски препроцесирати и компајлирати `main.c`, `uart.c` и асемблерски `startup.s`, а затим их линковати користећи линкер скрипту `-T linker.ld`, производећи извршни ELF фајл (`program.elf`). У пракси, *Makefile* управо генерише овакве командне позиве за све изворне јединице пројекта, укључујући и додавање неопходних путева до заглавља, библиотека и дефинисање макроа за условну компилацију. Он такође води рачуна о редоследу извршавања – да се сваки `.c` преведе у `.o` пре линковања, да се асемблерске датотеке такође преведу, и на крају да се позове линкер са свим насталим објектним фајловима и одговарајућом `.ld` скриптом.

У случају интегрисаних развојних окружења (IDE) као што су IAR или Keil, не постоји експлицитан *Makefile*, али концепт је исти – пројекат садржи подешавања која дефинишу који се фајлови компајлирају и како, а IDE интерно генерише командне позиве компајлера и линкера. Било да се користи ручно написан *Makefile* или IDE, резултат је на крају исти: сви претходно описани делови пројекта (startup код, main.c, модули, системске функције и линкерска скрипта) бивају састављени и повезани у једну извршну бинарну слику спремну за читавање у микроконтролер. *Напомена*: конкретна синтакса и организација *Makefile*-а могу бити различити (нпр. коришћење CMake уместо ручног *Makefile*-а), али увек служе истој сврси – аутоматизацији и контролисању процеса грађења *embedded* софтвера.

3.2. Употреба CMSIS и HAL слојева

У развоју софтвера за микроконтролере, уобичајено је ослањање на стандардизоване слојеве апстракције (библиотеке) који поједностављују руковање хардвером. Два најважнија таква слоја су **CMSIS** и **HAL**. Они заједно обезбеђују структуриран приступ компонентама система – од самог процесорског језгра до периферијских уређаја – чиме се смањује сложеност директног руковања регистрима и убрзава развој. У наставку су описани ови слојеви и њихова улога, уз пример који илуструје њихову употребу у пракси.

3.2.1. CMSIS (Cortex Microcontroller Software Interface Standard)

CMSIS је стандард који је развио **ARM** са циљем да уједначи софтверски интерфејс за Cortex-M микроконтролере. CMSIS обезбеђује дефиниције и функције блиске хардверу за сам процесор и основне периферије, независно од произвођача конкретног чипа. Кроз CMSIS, произвођачи микроконтролера испоручују сет заглавља и рутина које описују хардвер на симболичком нивоу – од регистара језгра до специјализованих периферијских јединица – на стандардизован начин.

Конкретно, CMSIS укључује **CMSIS-Core** део, који обухвата дефиниције за све регистре процесорског језгра и основне периферије. На пример, заглавља попут *core_cm4.h* или *core_cm7.h* садрже структуре и адресне симболе за Cortex-M4/M7 регистре (попут регистра за векторски адресер прекида **SCB->VTOR**), док заглавље *system_<i>Device</i>.h* (нпр. *system_stm32f4xx.h* за STM32 или одговарајуће Infineon заглавље за Traveo T2G) садржи параметре такта и почетну функцију за подешавање система. Захваљујући тим дефиницијама, програмер може да приступа регистрима на читљив начин уместо кроз „магичне“ бројеве адреса – на пример, да упише вредност у регистар контролера прекида употребом симбола **NVIC->ISER** уместо ручног адресирања меморије.

Важно је нагласити да CMSIS пружа и стандардизовану шаблон-рутину за стартап (startup) система. При укључивању микроконтролера, извршава се *startup* код (написан у асемблеру или C-у) који долази уз CMSIS пакет за тај уређај. Овај почетни код дефинише **векторску табелу прекида** (листа адреса свих прекидних рутина) и садржи *Reset_Handler* функцију (рутину на коју процесор прелази након ресетовања). У оквиру *Reset_Handler*-а се обично иницијализују основне ствари: пуњење почетних вредности података у RAM, брисање *BSS* секције, конфигурирање такта система и осталих низводних компоненти. Према CMSIS стандарду, уобичајено је да се у склопу стартап кода позове функција **SystemInit()** – дефинисана у *system_* заглављу – која подешава системски часовник (такт) и друге основне параметре пре него што се настави ка функцији *main*. На тај начин, CMSIS обезбеђује да сваки микроконтролер има предвидљиво иницијално окружење за извршавање корисничког кода, без оперативног система.

Кроз овакав слој, програмер има конзистентан и типски безбедан начин да управља хардвером. На пример, ако је потребно померање векторске таблице (нпр. при коришћењу bootloader-a), довољно је подесити регистар **SCB->VTOR** на адресу нове таблице – CMSIS је већ обезбедио симболички назив *SCB* (*System Control Block*)

структуре и поље *VTOR*. Слично томе, омогућавање или забрана прекида врши се стандардизованим функцијама попут `__enable_irq()` и `__disable_irq()`, које су имплементирани као инлајн асемблерске инструкције у CMSIS заглављима. Ове функције раде униформно без обзира на конкретан компајлер, што доприноси преносивости кода.

CMSIS самим својим постојањем смањује могућност грешака и повећава читљивост кода. Захваљујући њему и одговарајућим заглављима које обезбеђује произвођач, инжењери више не морају да користе непрегледне изразе за директан упис у меморију (нпр. `*(volatile uint32_t*)0x50000004 = 0x1;`), већ могу да користе симболичке и читљиве облике као што је `GPIO->OUT = 0x1;` за управљање излазима – што значајно унапређује одрживост и транспарентност софтвера. При томе, CMSIS не додаје практично никакав оверхед при превођењу: коришћење CMSIS макроа и структура своди се на директне операције над регистрима, па су перформансе таквог кода једнаке као да су регистри адресирани ручно. Ова особина чини CMSIS погодним и за критичне делове система где је битна брзина извршавања и детерминистичко понашање.

3.2.2. HAL (Hardware Abstraction Layer)

HAL представља слој **апстракције хардвера** који углавном обезбеђује произвођач микроконтролера у виду библиотеке. За разлику од CMSIS-а, који је оријентисан на сам процесор и основне регистре, HAL библиотеке циљају више нивое – пружају готове функције за управљање разним периферијама (тајмерима, UART-ом, GPIO линијама, A/D конверторима итд.), скривајући детаље реализације. Идеја HAL-а је да понуди униформан интерфејс за честе операције, тако да програмер може, на пример, једноставном функцијом да пошаље податке преко серијског порта или генерише PWM сигнал, без потребе да познаје сваки бит у неколико регистара тог периферног модула.

Типичан пример је STMicroelectronics-ова HAL библиотека за STM32 серију контролера: она нуди функције као што је `HAL_UART_Transmit()` за слање података преко UART-а или `HAL_GPIO_WritePin()` за подешавање излаза на пину. Слично томе, Infineon (раније Cypress) за своје PSoC/Traveo микроконтролере пружа HAL функције попут `cyhal_gpio_init()` за конфигурацију пина или `cyhal_timer_start()` за управљање тајмером. Ове функције унутар себе обављају читав низ корака – од укључивања такта периферије, конфигурисања режима рада, до провере валидности параметара – али су ка кориснику изложене као једноставан API (Application Programming Interface). На тај начин, **HAL слојеви смањују количину хардверски зависног кода у главном програму**: велики део посла обављају HAL рутине, а код програмера постаје краћи и јаснији.

Важно је напоменути да HAL библиотеке пишу сами произвођачи за своје породице уређаја, па оне нису универзално преносиве између различитих брендова микроконтролера. Међутим, унутар једне породице или једног произвођача, HAL настоји да уједначи интерфејсе. То значи да прелазак са једног модела на други (нпр. између различитих STM32 чипова или између различитих Infineon PSoC модела) захтева минималне измене у коду ако се користи HAL. Библиотека апстракује разлике: сваки конкретан модел ће имати другачије регистре „испод хаубе“, али ће позив функције `HAL_UART_Transmit()` радити на свима њима на исти начин са аспекта програмера.

Овакав приступ **убрзава развој** и чини прототипе брже готовим, јер се инжењер може фокусирати на логику програма уместо на детаље иницијализације сваког подсистема.

Цена те погодности је одређени **оверхед** – у погледу меморије и брзине. HAL функције су општије и садрже додатне провере и слојеве позива, па генерисани код може бити спорији у односу на ручно оптимизовани приступ регистрима. Ипак, у већини случајева овај оверхед је прихватљив, поготово имајући у виду добитак у преносивости и уштеди времена приликом софтверског развоја. За **перформансно критичне секције**, добра пракса је да се HAL користи за већи део система, а да се само у уским грлима где је потребна максимална брзина прибегне директном приступу регистрима (коришћењем CMSIS симбола или специјализованих *Low Level* драјвера). На тај начин се постиже баланс између брзине и одрживости кода.

Пример употребе HAL и CMSIS: Размотримо једноставан програм који трепће диодом на развојној плочи са Infineon Traveo II T2G микроконтролером. Захваљујући HAL слоју, целокупна иницијализација хардвера и периферија своди се на неколико позива функција из библиотеке, док CMSIS обезбеђује основне операције на нивоу језгра. У наставку је извод из функције main таквог програма (поједностављено за приказ):

```
int main(void)
{
    cy_rslt_t result;
    result = cybsp_init();    /* Иницијализација хардвера платформе и периферија */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);        /* Заустави извршавање ако иницијализација није успела */
    }

    __enable_irq();          /* Омогућавање глобалних прекида */

    /* Иницијализација корисничке LED диоде (GPIO пина) */
    cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                    CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);

    /* Покретање тајмера који ће генерисати прекид сваке секунде */
    cyhal_timer_t led_blink_timer;
    cyhal_timer_init(&led_blink_timer, NC, NULL);
    cyhal_timer_set_frequency(&led_blink_timer, 10000);    // подеси извор такта
    cyhal_timer_configure(&led_blink_timer, &led_blink_timer_cfg); // конфигурација периода
    cyhal_timer_register_callback(&led_blink_timer, isr_timer, NULL);
    cyhal_timer_enable_event(&led_blink_timer, CYHAL_TIMER_IRQ_TERMINAL_COUNT, 7, true);
    cyhal_timer_start(&led_blink_timer);

    for(;;)
    {
        /* У главној петљи проверава се флаг који поставља тајмерски прекид */
        if (timer_interrupt_flag)
        {
            timer_interrupt_flag = false;
        }
    }
}
```

```
        cyhal_gpio_toggle(CYBSP_USER_LED); // трептање LED-ом - инверзија стања пина
    }
}
}
```

Горњи код илуструје како се **HAL функционалност користи на високом нивоу**, док су детаљи скривени у позадини. На пример, позив `cybsp_init()` иницира читав низ операција неопходних да микроконтролер правилно проради: подешава се системски такт, покрећу се модули за управљање напајањем, резервишу се хардверски ресурси и иницијализују подразумеване периферије на плочи. Све те активности се одвијају „испод хаубе“ у оквиру неколико функција које ова рутина позива. У конкретном случају Infineon Traveo T2G платформе, `cybsp_init()` интерно позива, између осталог, функцију за покретање менаџера хардверских ресурса (`cyhal_hwmgr_init()`) и функцију за подешавање система напајања (`cyhal_syspm_init()`). Такође се примењују унапред генерисана подешавања такта и пинова (у оквиру функција као што су `cyscfg_config_init()` и `cyscfg_config_reservations()`), и региструју се повратни позиви за промену такта при уласку микроконтролера у режим ниске потрошње. Све ово је спаковано у једно апстрактно **HAL** позивно место, тако да у `main` функцији имамо само један ред којим „магично“ спремамо читав систем за рад. Овај приступ очигледно поједностављује структуру програма и смањује могућност пропуста у иницијализацији.

Након успешне иницијализације, у главној рутини се позива **CMSIS** функција `__enable_irq()` да се омогуће глобални прекиди на нивоу процесора. Ово је неопходан корак који је илустрација сарадње између CMSIS-а и HAL-а: CMSIS брине о контролеру прекида (NVIC) и другим системским аспектима, док HAL преузима конфигуравање периферијских модула који ће те прекиде користити. У примеру, након што је тајмер конфигуриран и стартован позивима `cyhal_timer_*` функција (HAL апстракција за тајмерски блок), тајмерски хардвер аутономно броји време и генерише прекид сваке секунде. Тај прекид се обрађује у позадини (HAL је регистровао `isr_timer` обрађивач преко `cyhal_timer_register_callback()`), при чему та обрада постави заставицу `timer_interrupt_flag`. Главна петља програма (`for(;;)`) затим, уз помоћ тог флага, зна када је једна секунда протекла и у одговору позива `cyhal_gpio_toggle()` – HAL функцију која мења стање излазног пина где је прикључена LED диода. Резултат је трептање диоде у интервалу од 1Hz, остварено без иједног директног уписа у хардверски регистар у корисничком коду. Сви уписи (нпр. подешавање излаза пина или конфигурација тајмера) реализовани су унутар HAL функција, користећи при том CMSIS дефинисане симболе за приступ одговарајућим регистрима.

Овај пример показује предности слојевитог приступа. Код је знатно читљивији и краћи него што би био уколико бисмо ручно конфигурирали сваки регистар. Истовремено, захваљујући CMSIS-у, имамо сигурност да су системски ресурси (попут векторске таблице, стања прекида, почетних секција меморије) исправно постављени пре него што HAL крене са иницијализацијом периферија. **CMSIS обезбеђује формалну доследност и стабилну основу система**, док **HAL омогућава бржи развој и већу преносивост** кода између различитих контролера исте породице. Комбинацијом ова два слоја, програмер може да достигне оптималан спој поузданости и ефикасности: критични делови се по потреби могу писати ближе хардверу (користећи CMSIS директно за приступ регистрима), док се већи део апликације ослања на проверене HAL рутине које убрзавају израду и смањују могућност грешака. Таква систематична организација кода

у слојевима значајно олакшава разумевање целокупног система и одржавање програма током његовог животног циклуса.

3.3. Стил програмирања и стандардизација

Развој **C** кода за уграђене (embedded) системе мора да следи дисциплинован стил и јасно дефинисане стандарде због високе поузданости и дугог животног циклуса који се од ових система очекују. Посебно у критичним доменима (аутомобилска индустрија, индустријска аутоматика, медицински уређаји), софтверски инжењери примењују строге смернице програмирања како би смањили могућност грешака и неодређеног понашања програма. Ове смернице обухватају како општи стил кодирања (конзистентно форматирање, именовање и организацију кода), тако и формалне стандарде безбедног програмирања усмерене на спречавање грешака на нивоу језика.

3.3.1. Индустрijски стандарди кодирања за безбедност и поузданост

Најзначајнији скуп правила за стил и безбедност кода у индустрији уграђених система је **MISRA C** стандард (*Motor Industry Software Reliability Association*). MISRA C дефинише строга правила којих се програмери требају придржавати како би избегли неодређено или потенцијално опасно понашање програма. Ова правила, између осталог, укључују забрану коришћења динамичке алокације меморије (нпр. функција *malloc*), неконтролисаних конверзија типова (*cast* операција) и употребе *goto* наредби. Придржавање оваквих стандарда омогућава примену формалне верификације и аутоматизоване статичке анализе кода (помоћу алата као што су *PC-lint* или *Coverity*), чиме се значајно повећава поузданост и безбедност резултујућег софтвера. У домену аутоматива, поштовање MISRA смерница је де-факто обавезно за испуњавање захтева функционалне безбедности (нпр. у оквиру стандарда **ISO 26262** за аутомобилске системе).

Поред MISRA-е, постоје и други сетови смерница усмерени на побољшање квалитета и сигурности кода. Један од њих је **CERT C** стандард, који представља смернице за сигурно програмирање на **C** језику. Док је MISRA првенствено фокусиран на безбедност система и избегавање кварова (*safety*) у уграђеним уређајима, **CERT C** нагласак ставља на обезбеђивање софтвера од рањивости и напада (*security*), пружајући препоруке за спречавање уобичајених софтверских пропуста као што су прекорачење бафера, неконтролисано руковање меморијом и слично. Оба стандарда се широко примењују – MISRA пре свега у аутомобилској и другим безбедносно критичним индустријама, а CERT C у областима где је кључна заштита од сајбер-напада. Важно је нагласити да се MISRA и CERT C не искључују међусобно; напротив, могу се користити комплементарно. Применом MISRA смерница се поставља темељ поузданог и структурно исправног кода, након чега CERT C препоруке додају додатни ниво заштите од злонамерних сценарија, чинећи софтвер и безбедним и сигурним. Поред тога, у пракси се могу срести и други доменски стандарди и препоруке – на пример, **ISO 26262** захтева да произвођачи у аутомобилској индустрији користе одговарајуће стандарде кодирања као део процеса обезбеђивања функционалне безбедности, док **CERT C** допуњује ту причу аспектима сајбер безбедности. У неким организацијама примену налазе и интерни стилски водичи или алтернативни стандарди (попут *Barr-C* смерница за уграђено програмирање), којима се додатно прецизирају правила кодирања у складу са специфичностима пројекта.

3.3.2. Конзистентност стила и одрживост кода

Осим придржавања формалних стандарда, одржавање конзистентног стила кодирања у целом пројекту има велики утицај на читљивост и одрживост софтвера. Под **стилом програмирања** подразумева се читав скуп правила и навика које код чине једноставним за праћење: конзистентно форматирање (увлачење линија, постављање заграда и размака), смислено именовање променљивих, константи и функција, структурирање кода по логичким целинама, као и писање јасних коментара где год је потребно. Уједначен стил олакшава тимски рад – различити програмери ће брже разумети туђи код ако сви прате исте конвенције. Стилска усклађеност такође поједностављује **code review** поступак (мануелну проверу кода од стране колега) и доприноси смањењу броја грешака у касним фазама развоја.

Стандардизација стила и придржавање договорених смерница данас су саставни део процеса развоја софтвера за микроконтролере. Коришћењем индустријских стандарда као што су MISRA C и CERT C, потпомогнутим алатима за статичку анализу који аутоматски откривају одступања од правила, успоставља се висок ниво квалитета кода. Доследан и добро документован код је не само мање склон грешкама већ је и лакше преносив на нове платформе и одржив током времена. На тај начин, **стил програмирања** и **стандардизација** представљају два повезана аспекта квалитета софтвера – први осигурава читљивост и једнообразност, а други уводи проверљива правила која подижу поузданост и безбедност система у целини. Поштовањем ових принципа, развојни тим гради основу за софтвер који ће бити отпоран на грешке, предвидив у понашању и усклађен са строгим захтевима уграђених критичних апликација.

3.4. Приступ меморијски мапираним регистрима

3.4.1. Меморијски мапирани улази и излази у микроконтролерима

У типичној *embedded* архитектури, периферни уређаји се контролишу путем меморијски мапираних регистара – посебних хардверских регистара који су изложени у заједничком адресном простору процесора. Део расположивих адреса рачунара резервисан је за ове уређаје, па упис података на одређену меморијску адресу заправо шаље податак периферном уређају, док читање са те адресе доводи до читавања податка из уређаја. Ово значи да се исте инструкције које CPU користи за приступ обичној меморији (нпр. *load/store* операције) могу користити и за приступ периферијама. Хардверски адресни декодер на системској магистрали препознаје да ли дата адреса припада меморији или уређају и усмерава сигнале и податке ка одговарајућој компоненти. За разлику од тзв. *port-mapped I/O* приступа (карактеристичног за неке раније архитектуре са посебним I/O инструкцијама), меморијски мапиран I/O поједностављује дизајн процесора и омогућава јединствен и ефикасан начин комуникације са уређајима, у складу са RISC филозофијом.

3.4.2. Адресни простор и распоред периферија

Микроконтролери обично имплементирају *Von Neumann* модел меморије са јединственим адресним простором за програмски код, податке и периферије. На пример, ARM Cortex-M архитектура дефинише 4 GB адресног простора подељеног на регионе за Flash (код), SRAM и периферије. Типично је велики блок од 512 MB резервисан за регистре on-chip периферних уређаја – код ARM Cortex-M језгара овај *Peripheral* регион обухвата адресе отприлике од 0x4000_0000 до 0x5FFF_FFFF. У том опсегу смештени су регистри разноврсних модула као што су GPIO, тајмери, UART, A/D конвертори и др; сваки уређај добија сопствени подпојас адреса за своје регистре. Читањем или писањем на било коју адресу у оквиру тог опсега, CPU у ствари приступа одговарајућем регистру периферије. Поред уобичајених периферија, и поједини системски контролни регистри (нпр. регистри за управљање прекидима, тактом или дебагом) такође су мапирани у посебан регион адресног простора – на пример, *Private Peripheral Bus* регион око адресе 0xE000_0000 код ARM Cortex-M садржи NVIC, SysTick и друге кључне регистре језгра. Овако дефинисана меморијска мапа поједностављује пројектовање *boot* софтвера и олакшава преносивост програма, јер сва Cortex-M језгра имају сличну организацију адресног простора за основне компоненте система.

3.4.3. Приступ регистрима у програму (С језик)

Са становишта софтвера, рад са меморијски мапираним регистрима своди се на уписивање и читање одређених адреса у меморији. Језик С омогућава веома директан приступ – корисник може декларацијом показивача на дату адресу или коришћењем одговарајућег *header*-а читати и мењати вредности хардверских регистра као да су променљиве у меморији. Међутим, да би се очувала исправна семантика, неопходно је те променљиве означити као *volatile*. Кључна реч *volatile* упозорава компајлер да се вредност дате променљиве може мењати изван тренутног програма (нпр. од стране хардвера или другог *thread*-а) те да не сме оптимизовати приступе – сваки упис или читање у изворном коду мора резултирати стварним уписом или читањем на датој адреси. У супротном, могло би се десити да компајлер негенерише очекивану инструкцију (нпр. ако „закључи“ да се вредност није променила) или да је задржи у регистру процесора, што би нарушило комуникацију са уређајем. Из тог разлога, регистарске константе у *header*-има микроконтролера увек су декларисане као *volatile*.

3.4.4. Моделирање хардверских регистра у С

Да би се олакшало коришћење меморијски мапираних регистра, у пракси се примењује техника мапирања регистара на С структуре. Идеја је да се дефинише *typedef struct* чија поља тачно одговарају регистрима једног периферног модула редом којим су они распоређени у меморијском простору. Затим се креира показивач (или макро) на ту структуру на базној адреси периферије. На тај начин, сваки регистар се може именовано адресирати преко поља структуре уместо преко „магичних“ хексадецималних константи. Ознака *volatile* се може применити на саму структуру или на показивач, чиме се гарантује да ће сваки приступ пољима структуре заиста приступити физичком регистру. Практично сваки савремени произвођач микроконтролера уз своје уређаје испоручује и одговарајуће заглавље са већ унапред дефинисаним структурама и базним адресама периферија. ARM је стандардизовао овај приступ кроз CMSIS (*Cortex Microcontroller Software Interface Standard*), па се у CMSIS *header*-има налазе описне структуре и макрои за све регистре циљаног система. На пример, у наставку је приказана поједностављена дефиниција једног GPIO модула и коришћење његових регистара:

```
typedef struct {
    volatile uint32_t IN;    // регистар улазних вредности пинова
    volatile uint32_t OUT;   // регистар излазних вредности пинова
    volatile uint32_t DIR;   // регистар правца (0 = улаз, 1 = излаз)
    // ... остали регистри периферије
} GPIO_TypeDef;

#define GPIO ((GPIO_TypeDef *) 0x50000000UL) // базна адреса GPIO модула

// Пример употребе:
GPIO->DIR |= 0x1; // поставља пин 0 као излаз
GPIO->OUT = 0x1;  // поставља логичку '1' на пин 0
```

Горњи код илуструје принцип меморијски мапираног приступа периферији. Структура `GPIO_TypeDef` декларативно описује низ од три 32-битна регистра – замислимо да су то улазни, излазни и регистар правца GPIO порта. Макро GPIO дефинише показивач на ову структуру на меморијској адреси `0x50000000`, за коју претпостављамо да је базна адреса одговарајућег GPIO контролера у датом микроконтролеру. Када у програму извршимо наредбу `GPIO->OUT = 0x1;`, компајлер ће генерисати машинску инструкцију за упис вредности 1 на меморијску адресу која одговара регистру `OUT` тог модула (нпр. инструкцију `STR` на ARM архитектури). Овим уписом се хардверски излаз на пину 0 поставља на високи ниво (под условом да је тај пин претходно конфигурисан као излаз, као у примеру где се `GPIO->DIR` подешава). Читљивост је знатно побољшана – уместо неразумљивог израза `*(volatile uint32_t *) (0x50000004) = 0x1;` који директно адресира меморију, програмер користи симболичко име `GPIO->OUT`, што јасно означава шта се догађа. Савремени преводиоци ће овакву употребу структура оптимизовати једнако ефикасно као и коришћење директних показивача или макроа; резултујући машински код је идентичан, па нема казне у погледу перформанси. Дакле, главна разлика је у побољшаној прегледности и типској безбедности кода, без жртвовања ефикасности.

3.4.5. Предности и значај апстракције регистра

Стандарди попут CMSIS-а и званични *header*-и произвођача обезбеђују да програмери не морају ручно да дефинишу сваки регистар и адресу – већ су им на располагању унапред проверене дефиниције структуре и базних адреса. Ово смањује могућност грешке и унапређује преносивост софтвера између различитих платформи. Код написан уз коришћење симболичких регистра (нпр. `RCC->AHB1ENR` или `GPIO->ODR` у случају STM32 микроконтролера) много је разумљивији него код са „магичним“ бројевима адреса, што доприноси бољој одрживости. У академском и индустријском контексту, овакав ниво апстракције се препоручује као део добрих пракси пројектовања: повећава се кохезија и јасно раздвајање надлежности софтверских модула, чинећи систем лакшим за верификацију и одржавање. На крају, приступ меморијски мапираним регистрима представља основни механизам којим *bare-metal* фирмвер остварује интеракцију са физичким светом – кроз промишљено коришћење овог механизма, постиже се детерминистичко, брзо и предвидиво извршавање управљачког кода, што је од пресудне важности за реалновременске примене.

4. Фазе компилације (превођења) у GCC

Превођење C програма у машински код одвија се кроз више дискретних фаза. GCC компајлер (GNU Compiler Collection) интерно дели процес на четири главна корака: **препроцесирање**, **компилацију** (у ужем смислу, превођење C у асемблер), **асемблирање** и **линковање**, тим редоследом. Свака фаза има специфичну улогу у претварању изворног .c кода у извршну бинарну слику и производи излаз који постаје улаз наредној фази. Коначни резултат ланца је извршна датотека (на пример ELF формат) спремна за учитавање у меморију микроконтролера. Табела 1 даје преглед ових фаза, алата и типичних улаза/излаза сваке фазе.

Табела 1. Фазе превођења C програма уз GCC (улаз и излаз сваке фазе)

Фаза	GCC алат/позив	Улаз	Излаз
Препроцесирање	gcc -E	*.c, *.h (изворни код)	Препроцесирани код (*.i)
Компилација (C→ASM)	gcc -S	*.i (препроцесирани)	Асемблерски код (*.s)
Асемблирање	gcc -c (или as)	*.s (асемблерски код)	Објектни фајл (*.o)
Линковање	gcc (или ld)	*.o (+ библиотеке)	Извршна датотека (нпр. ELF)

При развоју за *host* системе (нпр. Linux/Windows), програмер може да не приметити ове међуфазе – једноставан позив попут `gcc main.c -o main` аутоматски покреће све кораке (препроцесирање, компилацију, асемблирање, линковање) у једном пролазу. Међутим, у *embedded* окружењу (нпр. за ARM Cortex-M4 микроконтролере) користи се крос-компајлер *arm-none-eabi-gcc* који генерише машински код за циљну архитектуру (ARM) и захтева прецизнију контролу сваке фазе. GCC у *host* режиму користи подразумеване библиотеке и линкер скрипте које одговарају систему са оперативним системом, док *arm-none-eabi-gcc* користи специјализовани *bare-metal* toolchain без подршке оперативног система (ОС подршке). Због тога је у *embedded* пројектима неопходно експлицитно навођење параметара за циљни процесор (нпр. `-mcpu=cortex-m4`, `-mthumb`, `-mfloat-abi=softfp`, `-mfpu=fpv4-sp-d16`) и коришћење прилагођене линкерске скрипте (`-T linker.ld`) која описује меморијску мапу микроконтролера.

Аналитички осврт: У наставку су истакнуте кључне разлике и значај појединих фаза у контексту *host* vs. *embedded* компилације:

- У *host* окружењу (нпр. развој програма за РС), корисник ретко уочава међукорак јер GCC све извршава аутоматски. На пример, један позив `gcc main.c -o main` обухвата комплетан процес превођења од С кода до извршног фајла.
- У *embedded* окружењу, неопходна је фина контрола над сваком фазом. Програмер често ручно извршава или барем надгледа: препроцесирање (ради провере условне компилације и укључених заглавља), компилацију у асемблер (да види како се С код преводи у инструкције циљног процесора), асемблирање (стварање објектних `.o` модула) и линковање (спајање свих модула у јединствену бинарну целину прилагођену меморији).
- Додатне опције специфичне за платформу (нпр. `-mcpu=cortex-m4`, `-mthumb`, `-mfloat-abi=softfp`, `-mfpu=fpv4-sp-d16`) су од пресудног значаја у *embedded* компилацији. Оне прецизирају инструкциони сет, ABI (Application Binary Interface) и коришћење FPU јединице, обезбеђујући да генерисан код буде исправан за циљну архитектуру.

Следе детаљнији описи сваке фазе компилационог процеса, са примерима команди у GNU алатима. Приказана је постепена надоградња од најједноставнијих ка сложенијим позивима, уз анализу шта свака додатна опција доноси у односу на претходни корак. Сви примери су дати за ARM Cortex-M4 *bare-metal* окружење коришћењем алатке `arm-none-eabi-gcc`.

4.1. Препроцесирање

4.1.1. Улога препроцесора и резултујући код

Препроцесирање је прва фаза компилационог ланца, у којој се извршава *С препроцесор*. Овај алат обрађује директиве из изворног кода које почињу знаком `#`. Конкретно, препроцесор: уклапа садржаје укључених заглавља на места `#include` директива, проширује макрое дефинисане директивама `#define`, и условно уклања или укључује делове кода на основу `#ifdef` и сродних услова. Резултат ове фазе је **препроцесирани изворни код** – типично смештен у датотеку са екстензијом `.i` (за `C`) или `.ii` (за `C++`). Препроцесирани фајл садржи чист `C` код без иједне препроцесорске директиве, али са већ уметнутим садржајима свих хедер фајлова и проширеним макроима. Овако "очишћен" код представља конзистентан улаз за наредну фазу компилације.

4.1.2. Основни пример позива препроцесора

Најједноставнији пример покретања препроцесирања користећи `GCC` јесте позив са опцијом `-E`, која означава "само препроцесирај". На пример, уколико имамо изворни фајл `main.c`, основни позив би био:

```
arm-none-eabi-gcc -E main.c -o main.i
```

Објашњење основних елемената ове команде:

- **-E** – наредба `GCC`-у да изврши *само препроцесирање* и заустави се након те фазе (не наставља на компилацију у машински код).
- **main.c** – улазни изворни `C` фајл који се препроцесуира.
- **-o main.i** – задаје назив излазне датотеке. Овде ће сав препроцесирани код бити уписан.

Након извршења, добија се датотека **main.i**. Она садржи чист `C` код: сав код из свих `#include` хедер фајлова биће уметнут унутар `main.i`, сви макрои из `#define` директива биће замењени одговарајућим текстом, а ниједна препроцесорска директива више није присутна. Ова `.i` датотека представља улаз за компилацију у ужем смислу.

4.1.3. Проширени позив са укључивањем путања и макроа

У реалним пројектима, препроцесирање ретко када користи подразумеване путање и подешавања. Обично је потребно навести додатне опције, као што су путање до заглавља и дефинисање одређених макроа, да би препроцесор имао приступ свим потребним декларацијама. Следи један сложенији пример типичан за *embedded* пројекат (нпр. у оквиру Infineon ModusToolbox окружења):

```
arm-none-eabi-gcc -E -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp \
-mfpu=fpv4-sp-d16 -Og -Wall -isystem C:/path/to/gcc/arm-none-eabi/include \
-lbssps/TARGET_APP_KIT_T2G-B-E_LITE -I../mtb_shared/cmsis/release-v5.8.2/Core/Include \
-D CYT2BL5CAS main.c -o main.i
```

У овој продуженој наредби уведене су следеће компоненте:

- **Опције за архитектуру:** `-mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16` прецизирају да се код генерише за Cortex-M4 језгро, са *Thumb* сетом инструкција и коришћењем FPUv4-SP-D16 јединице са софтверским ABI-јем за покретни зарез. Ове опције осигуравају да препроцесор (и наредне фазе) знају циљну архитектуру и подесе одговарајуће дефиниције (нпр. кроз заглавља специфична за Cortex-M4).
- **Опције оптимизације и упозорења:** `-Og` значи оптимизација прилагођена дебаговању (умерена оптимизација која не омета поступак отклањања грешака), док `-Wall` укључује сва важнија компајлерска *упозорења*. Иако ове опције директно не мењају резултат препроцесора, оне су део конзистентног скупа опција које ће се пренети и на касније фазе, обезбеђујући доследне услове превођења.
- **Путање до заглавља:** опције `-I` и `-isystem` наводе додатне директоријуме са хедер фајловима. У примеру, `-lbssps/TARGET_APP_KIT_T2G-B-E_LITE` и `-I../mtb_shared/cmsis/.../Include` додају пројектне путање (Board Support Package, CMSIS) у којима се налазе неопходни хедери. Опција `-isystem <path>` укључује системске путање (нпр. до gcc библиотечких хедера) и третира их као системске (што може утицати на ниво упозорења).
- **Дефинисање макроа:** опција `-D CYT2BL5CAS` дефинише макро CYT2BL5CAS (једнако као да је у коду присутна линија `#define CYT2BL5CAS`). Укључене су и друге `-D` опције (попут `-DCOMPONENT_CM4` у другим контекстима) ради условне компилације – оне омогућавају да одређени делови кода буду укључени или изостављени у зависности од циљне плоче, микроконтролера, или конфигурације.
- **Изназ:** на крају, `-o main.i` означава да се резултат препроцесирања снима у `main.i`. Ова датотека ће садржати комплетан C код са свим укљученим заглављима и активираним макроима.

Помоћу ове команде, препроцесор обавља ширу интеграцију: учитава све потребне путеве до заглавља, активира релевантне макрое за условну компилацију, и припрема излазни `.i` фајл спреман за следећу фазу.

4.1.4. Значај препроцесирања у *embedded* пројекту – аналитички осврт

У оквиру једног сложеног *embedded* пројекта, попут софтвера за Cortex-M4 микроконтролер, препроцесор игра критичну улогу у повезивању различитих софтверских слојева. Конкретно, у ModusToolbox пројекту за развојни комплет KIT_T2G-B-E_LITE, препроцесирање обезбеђује да наредне фазе компилације добију конзистентан и потпун извор, тако што спаја следеће компоненте у једну целину:

1. **CMSIS дефиниције** – преко укључених хедер датотека као што је *core_cm4.h*, препроцесор обезбеђује унификован приступ регистрима и специјалним функцијама Cortex-M4 језгра. CMSIS (Cortex Microcontroller Software Interface Standard) поставља стандарде који омогућавају да се софтвер лакше преноси између различитих произвођача микроконтролера.
2. **HAL библиотеке** – укључивањем HAL (Hardware Abstraction Layer) заглавља, нпр. *cyhal_timer.h* и повезаних .с имплементација, код добија апстракцију периферија. То значи да исти С код може да ради на више варијанти хардвера, јер HAL слој брине о разликама у регистрима и контролерима периферија.
3. **BSP слој** – преко Board Support Package компоненти, нпр. *cybsp.h* и *cybsp.c*, дефинише се иницијализација система специфична за дату плочу и микроконтролер. BSP обухвата ствари као што су постављање такта, пинова, меморијских региона, подешавање векторске табеле, итд.

Препроцесор стога гради мост између високог нивоа изворног кода (нпр. *main.c* који садржи логику програма) и хардверски специфичних подешавања. Он обезбеђује да су све потребне декларације и дефиниције присутне пре него што започне превођење самог кода. Тако наредне фазе (компилација, асемблирање) добијају **конзистентан и комплетан изворни код**, где су сви међузависни делови већ решени у препроцесору.

4.2. Компилација (превођење С кода у асемблер)

4.2.1. Улога фазе компилације и излазни асемблерски код

У фази **компилације** (у ужем смислу те речи) GCC преводи препроцесирани С код у *асемблерски код* за циљну процесорску архитектуру. Другим речима, конструкције и изрази високог нивоа у С језику транслирају се у секвенцу асемблерских инструкција које остварују исту функционалност. На пример, петља, услов или позив функције у С-у биће мапирани у одговарајуће ARM Cortex-M4 инструкције (попут MOV, LDR, BL итд.). Излаз компилације је читљив текстуални фајл са екстензијом .s, који садржи асемблерски програм.

Компилација укључује и низ оптимизација које компајлер спроводи како би генерисани код био што ефикаснији у погледу брзине извршавања и/или величине. Ниво и тип оптимизација зависе од задатих опција (нпр. -O0 без оптимизације, -O2 агресивна оптимизација, -Og оптимизација погодна за дебаговање, итд.). Такође, ако је укључена подршка за дебаг (-g флаг), компилација у .s фајл ће укључити и информације потребне за дебагер (симболичка имена, линије кода, итд., које ће касније бити смештене у објектном фајлу).

У наставку је приказана **градација примера компилације** – од најједноставнијег позива (без икаквих специјалних опција) до индустријског нивоа позива са бројним параметрима. Пратићемо шта се додавањем сваке групе опција добија и како то утиче на излазни асемблерски код.

4.2.2. Најједноставнији пример превођења у асемблер

Прво посматрамо минималан пример компилације, где преводимо претходно добијени препроцесирани код у асемблер, без навођења икаквих додатних опција осим оних нужних. Претпоставимо да већ имамо main.i (резултат препроцесирања). Основни позив GCC компајлера да произведе .s из .i је:

```
arm-none-eabi-gcc -S main.i -o main.s
```

Објашњење опција и резултата:

- **-S** – преведи С код у асемблер (заустави се након генерисања .s фајла). Ова опција каже компајлеру да не иде до машинског кода у овој фази, већ да излаз треба да буде асемблерски извор.
- **main.i** – улазна датотека, препроцесирани С код (може се уместо тога навести и .c фајл; GCC ће интерно прво обавити препроцесирање, па компилацију).
- **-o main.s** – назив излазног асемблерског фајла.

Резултат ове команде је датотека **main.s** која садржи асемблерске инструкције. Уколико нисмо навели специфичну архитектуру, GCC ће употребити неке подразумеване вредности (често дефолтну ARM архитектуру). Такав .s би био исправан асемблерски код, али можда не оптималан или потпуно усклађен са одређеним језгром микроконтролера. На овом нивоу, секције кода (нпр. .text) би биле подразумеване, без посебних атрибута.

4.2.3. Додавање архитектурских опција (умерено сложен пример)

Следећи корак је да прецизирамо циљну архитектуру како би асемблерски код тачно одговарао жељеном микроконтролеру (нпр. Cortex-M4) и његовим могућностима. Додајемо најважније *параметре за архитектуру*: тип процесора, сет инструкција (Thumb или ARM), тип ABI-ја за покретни зарез и евентуално модел FPU јединице. На пример:

```
arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp \
-mfpu=fpv4-sp-d16 main.c -o main.s
```

У односу на претходни основни позив, овде смо укључили следеће опције за архитектуру циљног процесора:

- **-mcpu=cortex-m4** – циљна CPU архитектура је Cortex-M4 језгро. Ово осигурава да компајлер генерише инструкције које постоје на M4 и да оптимизује код за ту микро-архитектуру.
- **-mthumb** – користи *Thumb* сет инструкција (који је доминантан на Cortex-M серији, уместо пуне 32-битне ARM инструкционе конфигурације). Без ове опције, компајлер би могао генерисати ARM инструкције које Cortex-M4 не подржава, јер Cortex-M4 извршава искључиво Thumb-2 код.
- **-mfloat-abi=softfp** и **-mfpu=fpv4-sp-d16** – ове опције дефинишу употребу јединице за покретни зарез. Cortex-M4 може имати уграђен *Floating Point Unit* (FPU) за операције са покретним зарезом. Опција -mfpu=fpv4-sp-d16 означава FPU који подржава 32-битне (single precision) операције са 16 регистара (FPv4-SP), док -mfloat-abi=softfp наводи да се у бинарном интерфејсу користи "софтверски" ABI за FPU (процесор ће користити FPU инструкције, али ће позивни интерфејс бити компатибилан са кодом који нема FPU - параметри се прослеђују преко стека/регистара као да FPU не постоји, чиме се задржава компатибилност).

У овом примеру, улаз смо поново навели као main.c уместо main.i – GCC ће сам обавити препроцесирање main.c пре компилације у асемблерски код, јер опција -S не искључује претходне фазе, већ само зауставља након генерисања .s. Добијени фајл **main.s** сада је прецизно прилагођен архитектури ARM Cortex-M4 са Thumb-2 инструкцијама и FPv4-SP-D16 јединицом за покретни зарез. Другим речима, свака инструкција у .s датотеци биће легитимна за Cortex-M4 и у складу са очекивањима за тај микроконтролер, што је предуслов да касније асемблер успешно преведе тај код у машинске инструкције.

4.2.4. Опције за оптимизацију, дебаг и секционисање кода (средње сложен пример)

Када је основна архитектура подешена, следећи ниво сложености је укључивање опција за оптимизацију кода, генерисање дебаг информација, као и организацију излазног кода у посебне секције ради ефикаснијег руковања меморијом. Додајемо неколико важних флагова:

```
arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16 \
-Og -g -Wall -ffunction-sections -fdata-sections \
main.c -o main.s
```

Нове опције уведене у овом позиву:

- **-Og** – оптимизација која је намењена дебаговању. GCC овим флагом покушава да балансира између оптимизације перформанси и очувања структуре кода ради лакшег дебаговања. Код генерисан са -Og ће радити прилично ефикасно, али ће и даље кореспондирати директно са изворним линијама, што помаже при извршавању линију по линију у дебагеру.
- **-g** – укључује дебаг информације у генерисани код. То значи да ће .s садржати податке о томе којој изворној .c линији припада свака инструкција, као и имена променљивих, функција и сл. Ове информације ће бити уграђене у објектни фајл и ELF, што омогућава дебагеру (нпр. GDB) да повеже машински код са изворним кодом.
- **-Wall** – омогућава приказ "свих" важних упозорења (*All warnings*). Иако ова опција сама по себи не утиче на асемблерски излаз, добра је пракса укључити је да би се током компилације указало на потенцијалне проблеме у коду (нпр. неиницијализоване променљиве, неупотребљене параметре, имплицитне конверзије типова итд.).
- **-ffunction-sections** и **-fdata-sections** – ове две опције инструишу компајлер да стави сваку функцију и сваки глобални податак у засебну секцију у асемблерском излазу. Конкретно, свака функција ће бити у својој .text.<func_name> секцији у .s фајлу, а сваки глобални или статички податак у својој .data.<var_name> или .bss.<var_name> секцији (зависно од тога да ли је иницијализован). Ово изгледа као унутрашњи детаљ, али је изузетно корисно касније приликом линковања: омогућава линку да елиминише неупотребљене функције и податке (функционалност звана *garbage collection of sections*, активира се помоћу линкер опције --gc-sections). Дакле, -ffunction-sections -fdata-sections припрема терен да се величина финалне бинарне слике смањи уклањањем „мртвог кода” који нигде није позван или искоришћен.

Резултат позива са овим додатним опцијама је асемблерски код main.s који у себи носи дебаг симболе, а свака функција и глобална променљива су издвојене у своје секције. Ово чини излазни .s фајл обимнијим и детаљнијим (због додатних секцијских директива и симболичких информација), али уз бројне предности: могућност покретања у дебагеру са прегледом изворног кода, бољу анализу приликом линковања и могућност да се неукључене секције касније избаце из финалне слике.

4.2.5. Додавање путања до заглавља и условних макроа (комплекснији пример)

Да бисмо компилацију приближили реалним условима *embedded* пројекта, у овом кораку укључујемо опције за додатне `include` путање и макро дефиниције, слично као код препроцесирања. Замислимо да желимо да компајлирамо `main.c` који зависи од одређених заглавља у пројекту и да условно укључује код на основу макроа (нпр. за различите компоненте или периферије). Команда може изгледати овако:

```
arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16 \
-Og -g -Wall -ffunction-sections -fdata-sections \
-I ../mtb_shared/cmsis/release-v5.8.2/Core/Include \
-I bsp/TARGET_APP_KIT_T2G-B-E_LITE \
-D CYT2BL5CAS -D COMPONENT_CM4 \
main.c -o main.s
```

Нове ствари у овом позиву:

- Додате су **путање до заглавља** специфичних за пројекат: две путање задате са `-I` заставицама (једна до `CMSIS` језгра, друга до `BSP` директоријума за одређену плочу). Сада компилација има приступ свим потребним декларацијама (нпр. регистарским дефиницијама, прототиповима функција драјвера и сл.) које нису у стандардним системским локацијама.
- Додати су **макрони** специфични за циљну плочу/микроконтролер: у примеру, `-D CYT2BL5CAS` и `-D COMPONENT_CM4`. Они могу утицати на то који ће се делови кода из заглавља и `.c` фајлова укључити. На пример, код може имати услове `#ifdef CYT2BL5CAS` да би укључио одређена подешавања само за тај микроконтролер, или `#ifdef COMPONENT_CM4` да би одвојио код који се компајлира за Cortex-M4 језгро у SoC-овима са више језгара.

Овај ниво команде је типичан за практичан *embedded* пројекат: сада већ укључујемо `CMSIS` и `BSP` заглавља, и дефинишемо макрое специфичне за платформу, тако да компајлер има *пуну слику* пројекта. Излаз `main.s` генерисан овом командом садржаће, поред раније поменутих информација, и све позиве функција и референце на променљиве из додатних библиотека (нпр. `HAL`) са исправним именима, јер су захваљујући `-I` путањама пронађене одговарајуће декларације.

4.2.6. Комплетан build позив – индустријски сценарио (најсложенији облик)

На крају, долазимо до најсложенијег облика позива компајлера, какав се налази у реалним build системима велике компаније или отвореног кода. Овде укључујемо све опције које су у пракси потребне да би превођење било успешно интегрисано у окружење пројекта. Пример такве *целовите* команде (апсолутне путање скраћене ради прегледности):

```
arm-none-eabi-gcc -S -mcpu=cortex-m4 --specs=nano.specs -Og \
-mfloat-abi=softfp -mfpu=fpv4-sp-d16 -mthumb -ffunction-sections -fdata-sections \
-ffat-lto-objects -g -Wall -pipe -MMD -MP -MF main.d -MT main.o \
-isystem C:/.../arm-none-eabi/11.3.1/include \
-I bsps/TARGET_APP_KIT_T2G-B-E_LITE -I ../mtb_shared/cmsis/release-v5.8.2/Core/Include \
... (остале -I и -D опције) ... \
-o main.s main.i
```

Ова команда је репрезент типичног излаза build система генерисаног од алата као што је CMake или пројектни менаџери. Карактеристично, садржи:

- **Мноштво -I и -D опција:** све потребне путање до заглавља (за BSP, HAL, PDL, CMSIS, итд.) и макро дефиниције за условну компилацију свих модула. У пројектима са више зависности, број ових опција може бити велик, како би се компајлеру предочиле све локације хедер фајлова и све неопходне конфигурационе константе.
- **Спецификације runtime-a:** --specs=nano.specs означава да се линковање/компајлирање ради са *Newlib-nano* C стандардном библиотеком (која је лакша и мања, намењена микроконтролерима). Ово је важно за финално линковање, али се наводи већ у компилационој фази како би компајлер знао да циља на скуп библиотечких функција оптимизованих за embedded.
- **LTO подршка:** опција -ffat-lto-objects укључује *Link-Time Optimization* метаподатке у објектне фајлове. Ово значи да ће се касније, приликом линковања, моћи радити глобалне оптимизације преко граница фајлова. -ffat-lto-objects конкретно чува и нормалан објектни код и LTO биткод у .о датотеци, за случај да је linker без LTO подршке.
- **Опције за праћење зависности:** -MMD -MP -MF main.d -MT main.o су флагови који се тичу генерисања *dependency* фајлова за Makefile. Они инструктишу компајлер да, поред .s излаза, генерише и .d датотеку (овде main.d) са списком свих укључених хедер фајлова које main.c непосредно или посредно користи. Ово је корисно да build систем зна када да поново компајлира main.c – ако се измени неки хедер наведен у main.d, main.o ће бити реконструисан. Опција -pipe нема утицаја на резултат већ само утиче на метод комуникације између фаза (кроз pipeline уместо привремених фајлова, ради бржег извршења).
- **Редослед аргумената:** Приметимо да је у овом примеру излаз (-o main.s) наведен непосредно пре улазног фајла main.i. Пошто је наведена .i датотека као улаз, то значи да је препроцесирање већ обављено (можда коришћењем опције -save-

temps или раније ручно), а овде се експлицитно компајлира тај препроцесирани код. Уколико би се навео main.c уместо main.i, GCC би сам покренуо препроцесор. Обе варијанте су важеће, али коришћење .i директно је ретко осим у дубокој анализи; обично се наводи .c и GCC одради све међукораке аутоматски.

Овакав позив представља *пун индустријски сценарио*: прецизно је дефинисан цео toolchain, укључене су све релевантне путање до библиотека, активирани су све потребне оптимизације и поставке за генерисање додатних артефаката (попут .d dependency фајлова). Резултујући main.s биће потпуно конзистентан са остатком пројекта, спреман за асемблирање у објектни фајл.

Резиме градације (компилација C→ASM): На основу наведених примера, можемо истакнути еволуцију GCC позива за фазу компилације и шта је у ком кораку додато:

1. **Минимално:** gcc -S main.c -o main.s – основна генерација асемблерског кода из C извора (подразумевана архитектура, без посебних опција).
2. **Архитектура:** додавање -mcpu, -mthumb, -mfloat-abi, -mfpu опција – циљање конкретног процесора (нпр. Cortex-M4) и његовог инструкционог сета/FPU.
3. **Дебаг и оптимизација:** додавање -Og, -g, -Wall, -ffunction-sections, -fdata-sections – омогућава лакше отклањање грешака и припрему за елиминацију неупотребљених секција.
4. **Embedded специфичности:** додавање -I путања и -D макроа – укључивање свих потребних заглавља (CMSIS, HAL, BSP) и конфигурационих дефиниција за конкретан хардвер.
5. **Индустријско окружење:** веома дугачке команде са свим путањама, макроима, специјалним спецификацијама (--specs), LTO и dependency опцијама – какве генеришу аутоматизовани build системи у пракси, за потпуно поновљиву и контролисану компилацију.

4.3. Асемблирање (превођење асемблерског кода у објектни модул)

4.3.1. Улога асемблера и објектни фајл (.o)

Када је асемблерски код генерисан (један или више .s фајлова), следећа фаза је **асемблирање**. Асемблер (нпр. `arm-none-eabi-as`, који се може позвати и кроз `gcc -c` опцију) преводи асемблерски извор у бинарни облик – релокативни *објектни фајл*. Објектни фајлови имају екстензију .o и обично су у ELF формату или сличном формату објектних модула. Они садрже машински код (бинарне инструкције) за дати модул, али још увек нису самостални извршни програм. Наиме, у .o фајлу симболи који представљају адресе функција или података из других модула остају нерешени – то су *референце* које ће бити повезане у следећој фази (линковању). Такође, .o фајл задржава табеле релокација (које кажу линкеру где треба уписати конкретне адресе касније), симболичке информације о функцијама и променљивим, и секцијске информације (нпр. одвојене секције за код, податке, итд.).

Другим речима, асемблер узима .s (који је текстуални опис инструкција и података) и емитује одговарајуће машинске инструкције и бинарне податке у .o, али оставља линкеру да "залепи" све .o модуле заједно и попуни коначне адресе. Ово омогућава да се један модул компајлира/асемблује независно од осталих.

У наставку поново демонстрирамо градацију позива за асемблерску фазу, пратећи сличан образац као за компилацију: од минималног позива ка све сложенијим, уз анализу додатих опција.

4.3.2. Најједноставнији позив асемблера

Минимални корак је да узмемо генерисани асемблерски код и преведемо га у објектни фајл. На претпоставци да имамо `main.s`, основни позив изгледа:

`arm-none-eabi-gcc -c main.s -o main.o`

Овде:

- **-c** – каже GCC-у да изврши само компилацију/асемблирање до објектног фајла, али без линковања. То јест, зауставља се након генерисања .o из .s (или директно из .c, али у контексту већ имамо .s па ради као чист асемблер).
- **main.s** – улазни асемблерски извор.
- **-o main.o** – излаз је објектна датотека.

Ова команда позива интерни `arm-none-eabi-as` у оквиру GCC алатке, који транслира инструкције из `main.s` у одговарајући машински код и смешта га у `main.o`. Међутим, ако

не специфицирамо архитектуру, асемблер ће претпоставити неке подразумеване вредности (нпр. ARM архитектура без FPU-a). За демонстрацију једноставног примера, `main.o` ће бити произведен – али ако код у `main.s` користи специфичне Cortex-M4 инструкције, асемблер без одговарајућих параметара може приговорити (или чак погрешно протумачити инструкције). Дакле, овај минимални позив је исувише генеричан за реалну употребу на Cortex-M, али служи да прикаже суштину: трансформацију из `.s` у `.o` фајл.

4.3.3. Спецификација архитектуре при асемблирању

Да би објектни фајл био исправан за конкретан микроконтролер, практично увек морамо при асемблирању навести исте оне опције за циљну архитектуру које смо користили и приликом компилације у `.s`. Ако смо, на пример, у асемблерском коду користили Cortex-M4 инструкције, морамо то рећи асемблеру. Дакле, проширујемо команду додавањем истих флагова `-mcpu`, `-mthumb`, `-mfloat-abi`, `-mfpu`:

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp \
-mfpu=fpv4-sp-d16 main.s -o main.o
```

Ново у односу на претходни позив:

- Наведене су опције за архитектуру: **`-mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16`**, исто као у секцији 4.2.3. Сада асемблер тачно зна да парсира и генерише код за Cortex-M4 и његову FPU јединицу. Ако `main.s` садржи, рецимо, инструкцију `VMUL.F32` (множење бројева са покретним зарезом), асемблер без `-mfpu=fpv4-sp-d16` не би препознао ту инструкцију, док са овом опцијом хоће и знаће да је емитује као одговарајући `opcode`.
- Остали делови су исти: `-c` (assembling only), `main.s` као улаз и `-o main.o` као излаз.

Резултат је објектна датотека **`main.o`** која је сада тачно генерисана за ARM Cortex-M4 са FPv4-SP-D16. Сви машински кодови унутар `main.o` одговарају циљном процесору. Ова `.o` датотека и даље садржи нерешене симболе (ако их је било у коду, нпр. позив функције из другог модула), али је потпуно спремна за линковање у финални извршни програм.

4.3.4. Укључивање оптимизационих и дебаг информација у објектни фајл

Иако главни посао асемблера није да "мисли" о оптимизацијама (то ради компајлер), можемо му проследити одређене опције које ће утицати на објектни фајл. Примера ради, ако желимо да задржимо дебаг симболе у .о (који су пренети из .s захваљујући опцији `-g` у претходној фази) или да осигурамо да су секције у .о подељене (због `-ffunction-sections` и `-fdata-sections` које смо раније користили), треба да користимо *исте те опције* и у фази асемблирања. Стога проширујемо команду:

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16 \
-Og -g -Wall -ffunction-sections -fdata-sections \
main.s -o main.o
```

Овде смо додали:

- **-Og -g -Wall** – ове опције су већ објашњене и њихов ефекат је углавном у фази компилације. Међутим, задржавање `-g` осигурава да ће дебаг информације које су можда у .s (коментари `.file`, `.loc` директиве итд.) бити укључене у финални .о. Опција `-Wall` нема директан утицај на асемблерски корак осим што би могла исписати упозорења ако примети нешто необично у .s коду.
- **-ffunction-sections -fdata-sections** – ове опције смо такође користили раније да поделимо код у секције. Важно је знати да су секције већ одређене у .s (путем директива `.section` које је компајлер тамо ставио). Асемблер ће те директиве свакако поштовати, али задржавање истих опција осигурава конзистентност и да се у случају компајлирања директно из С у .о (прескачући .s фазу) секције исправно формирају. У овом конкретном сценарију (асемблирање .s генерисаног од раније), `-ffunction-sections -fdata-sections` не уводе нове промене јер је .s већ конструисан са тим секцијама; ипак, добра је пракса наводити их и у овој фази ради јасноће и уједначености.

После извршавања ове команде, добијамо **main.o** у којем се налазе: машинске инструкције (секција `.text` подељена на потсекције по функцијама), подаци (секције `.data/.bss` можда такође подељене), симболичке табеле (са именима свих функција и глобалних променљивих које су дефинисане или референциране) и дебаг информације (ако је укључен `-g`). Овај објектни фајл је већи (због дебаг секција) али садржајно богатији и омогућиће линку касније да uklони неупотребљене делове и да се програм може дебаговати.

4.3.5. Асемблер са додатним include путањама и макроима

У неким случајевима, асемблерски код .s може да користи директиве сличне *С препроцесору*. На пример, GNU асемблер (gas) допушта коришћење директиве `.include` за укључивање других фајлова, а такође може разумети и неке макро дефиниције (иако не на истом нивоу као *С препроцесор*). Из тог разлога, чак и у фази асемблирања, понекад је потребно навести путање до заглавља или дефинисати макрое, нарочито ако .s није у потпуности самосталан већ укључује друге фајлове (нпр. дефиниције регистара, константи и сл.).

Замислимо да `main.s` садржи линије попут `.include "core_cm4.h"` (што значи да асемблер треба да учита тај хедер) или условне секције које зависе од неке константе. У том случају, команду за асемблирање проширујемо слично као код компилације:

```
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16 \
-Og -g -Wall -ffunction-sections -fdata-sections \
-I ../mtb_shared/cmsis/release-v5.8.2/Core/Include \
-I bsp/TARGET_APP_KIT_T2G-B-E_LITE \
-D CYT2BL5CAS -D COMPONENT_CM4 \
main.s -o main.o
```

Овде су додате исте оне путање до заглавља (`-I`) и макро дефиниције (`-D`) које су коришћене у претходним фазама. То осигурава да ако `main.s` директно укључује неки хедер (нпр. `core_cm4.h` за регистарске дефиниције), асемблер може пронаћи тај фајл на задатој путањи. Такође, ако асемблерски код користи условне директиве (`.if` са неким константама), опције `-D` ће обезбедити да су те константе дефинисане.

Резултат је да ће **улазни .s фајл** бити асемблиран у `main.o` уз укључивање свих *CMSIS* дефиниција и макроа специфичних за плочу, баш као што је урађено током препроцесирања у *С* фази. Ова конзистентност је битна – све фазе користе исте дефиниције и исте хедере, како би финални програм био исправан.

4.3.6. Индустијски позив асемблера (пуно build окружење)

Најсложенији облик командне линије за асемблерску фазу биће сличан најсложенијем примеру из компилације, јер ће build систем проследити већину истих опција и при асемблирању. У наставку је сажетак једног таквог позива (путање скраћене):

```
C:/path/arm-none-eabi-gcc -c \
-mcpu=cortex-m4 --specs=nano.specs -Og -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -mthumb \
-ffunction-sections -fdata-sections -ffat-lto-objects -g -Wall -pipe -MMD -MP -MF main.d -MT main.o \
-isystem C:/.../arm-none-eabi/11.3.1/include \
...(остале -isystem, -I и -D опције)... \
-o .../build/Debug/main.o main.s
```

У овој "пуноправној" команди видимо:

- **Све путање** (-I и -isystem) за BSP, HAL, PDL, CMSIS, исте као код фазе компилације.
- **Све макрое** (-D) за различите конфигурације.
- **Спецификацију runtime-а** --specs=nano.specs за коришћење мале C библиотеке.
- **LTO опције и праћење зависности** (-ffat-lto-objects -MMD -MP ...) – иако LTO и праћење зависности нису директно релевантни за асемблер, build систем их ставља у све командне линије ради једноставности, а GCC их углавном прослеђује где треба (LTO метаподаци ће бити убачени у .o, а .d датотека ће бити генерисана ако се асемблира директно из .с – у овом случају main.s је већ финалан, па dependency фајл није толико битан).
- **Пуну путању излазног фајла** (-o build/Debug/main.o) – у изградњи пројекта објектни фајлови се смештају у одређене директоријуме, па команда то директно наводи.
- **Улазни фајл** (main.s на крају) – који је у овом случају .s генерисан у ранијој фази.

Ова команда представља прави индустријски сценарио – у њој је садржано све што је потребно да се добије конзистентан објектни фајл у складу са читавим пројектом. Резултат, main.o, биће идентичан као и у претходном кораку, само произведен у контексту великог пројекта.

Резиме градације (асемблирање ASM→OBJ): Кораци кроз које смо прошли могу се сажети на следећи начин:

1. **Минимум:** gcc -c main.s -o main.o – основно асемблирање без навођења архитектуре (преводи .s у .o, али за конкретан Cortex-M можда неадекватно без додатних параметара).
2. **Архитектура:** додавање -mcpu, -mthumb, -mfpu – асемблер циља тачно одређени процесор (Cortex-M4, Thumb-2, FPUv4-SP).

3. **Дебаг и секционисање:** додавање -Og, -g, -Wall, -ffunction-sections, -fdata-sections – задржавање дебаг симбола у .о и сегментација по функцијама и подацима ради касније оптимизације.
4. **Интеграција пројектних заглавља:** додавање -I и -D опција – обезбеђује да асемблер пронађе све укључене фајлове и примени условне секције, исто као компајлер.
5. **Пун build систем:** врло дугачка команда са свим путањама, макроима, спецификацијама и осталим опцијама – обично генерисана аутоматски, обезбеђује да је сваки објектни модул компатибилан и спреман за даље линковање.

4.4. Линковање

4.4.1. Улога линкера и повезивање објектних модула

Линковање је завршна фаза у генерисању извршног програма. Линкер (нпр. GNU ld, који се позива командом `arm-none-eabi-gcc` без `-c` опције) узима један или више објектних фајлова (*.o) и спаја их у јединствену извршну датотеку. При томе може укључити и библиотеке (нпр. стандардну C библиотеку, или специфичне хардверске библиотеке) уколико програм користи функције дефинисане у њима. Главни задаци линкера су:

- **Разрешавање референци:** линкер повезује симболе из различитих .o модула. Ако функција у `main.o` позива функцију из `uart.o`, линкер ће у машинском коду заменити привремену референцу стварном адресом те функције из `uart.o`. Слично важи и за глобалне променљиве – све неодређене референце морају бити попуњене правим адресама или вредностима.
- **Комбинација секција:** објектни фајлови имају секције (.text, .data, .bss, итд.). Линкер их све спаја у заједничке секције извршне датотеке и распоређује на одређене меморијске адресе. Резултат је типично ELF датотека која у себи носи информацију о томе где ће свака секција бити смештена у меморији када се програм учита.
- **Додавање runtime startup кода:** у системима са оперативним системом, линкер често аутоматски додаје иницијализациони код (нпр. `crt0` за C програме) који поставља окружење пре него што се позове `main`. У *bare-metal* окружењу за микроконтролере, стартап код (укључујући векторску табелу прекида и функцију `Reset_Handler`) обично се обезбеђује као засебан објектни фајл (нпр. `startup.o` генерисан из `startup.s` асемблерског фајла) који се укључује у линковање. Линкер ће и њега повезати са осталим деловима програма.

Крајњи излаз линковања је **извршна бинарна датотека** – у нашем случају у ELF формату – у којој су све функције и подаци смештени на својим коначним меморијским адресама. Та датотека (`program.elf`, на пример) представља самосталан програм који се може учитати у Flash меморију микроконтролера и извршавати.

4.4.2. Основни пример линковања једног модула

Најједноставнији могући сценарио линковања је када имамо само један објектни фајл (са свим кодом) који желимо да претворимо у извршну датотеку. Команда би изгледала овако:

```
arm-none-eabi-gcc main.o -o program.elf
```

Овде нема -с опције, чиме смо рекли GCC-у да након компилације/асемблирања (које у овом случају није ни потребно јер већ имамо .о) настави до линковања. Навели смо:

- **main.o** – улазни објектни модул за линкер. Подразумева се да main.o може садржати референце (нпр. позиве библиотечких функција printf, memcpy итд.). Ако су оне присутне, линкер ће аутоматски тражити у стандардној библиотеци њихове дефиниције (осим ако му се изричито каже да не линкује стандардну библиотеку).
- **-o program.elf** – назив излазне извршне датотеке у ELF формату.

Резултат је фајл **program.elf**. Уколико је main.o садржао само програмски код без зависности или са свим унутрашњим функцијама, program.elf би већ могао бити исправан извршни програм (за хост систем). Међутим, за микроконтролер, овај корак још увек није довољан – потребно је обезбедити да код буде смештен у одговарајућу меморију (Flash/RAM), што укључује коришћење линкерске скрипте. У овом базичном примеру, ако се изврши линковање без икаквих додатних параметара, користиће се **подразумевана linker скрипта** уграђена у GCC, која није прилагођена микроконтролеру. То значи да ће program.elf вероватно имати секцију .text мапирану на стандардну адресу за Linux (попут 0x08048000) или на подразумевани почетак меморије (попут 0x00000000), што не мора да одговара стварној физичкој адреси Flash меморије микроконтролера. Стога, иако је program.elf формално креиран, он не би био исправан за читавање у микроконтролер без додатних корака.

Објашњење: Без експлицитног навођења linker скрипте, arm-none-eabi-ld (позван кроз gcc) ће применити подразумевана правила за распоред секција. То укључује коришћење своје уграђене скрипте намењене *bare-metal* EABI окружењу, али та скрипта и даље претпоставља одређене стандардне вредности (нпр. да код почиње од адресе 0x00000000 или 0x08000000, али често са поставкама за симулацију). Уопштено, **сваки embedded пројекат мора имати прилагођену linker скрипту**, јер без ње нема начина да линкер зна стварни распоред меморије циљног система.

4.4.3. Коришћење линкерске скрипте за меморијски распоред

Уобичајена пракса за микроконтролере је да се линкеру проследи прилагођена *линкерска скрипта* (обично екстензије `.ld`). Линкерска скрипта описује расположиве меморијске регионе (нпр. где се налази Flash, где RAM, колики су), као и правила како секције програма треба распоредити у те регионе. GCC-у се линкерска скрипта прослеђује опцијом `-T <file>`.

Минимална измена претходне команде, да би радила исправно у `embedded` окружењу, била би додавање `-T linker.ld` параметра са одговарајућом скриптом. Претпоставимо да имамо датотеку *linker.ld* која описује наш микроконтролер. Команда постаје:

```
arm-none-eabi-gcc main.o -T linker.ld -o program.elf
```

Ово је сада **прави минимални облик за `embedded`**: из објектног кода добијамо ELF који је распоређен према меморијским ограничењима микроконтролера. Линкер ће користити информације из *linker.ld* да одлучи на коју адресу ставља `.text` секцију (тј. бинарни код програма), где иде `.data` (иницијализовани подаци у RAM, са копијом у Flash), `.bss` (неиницијализовани подаци у RAM), стек, хип, и друге секције.

Да нагласимо значај: **Без линкерске скрипте, линковање за микроконтролер није исправно.** У `embedded` систему нема оперативног система који би накнадно распоређивао меморију, па је на програмеру/линкеру да од почетка зна тачне адресе. Зато сваки `embedded` пројекат мора имати или ручно написану или аутоматски генерисану линкерску скрипту. У случају да не користимо `-T` флег, линкер ће применити своју генеричку расподелу која скоро сигурно не одговара нашем уређају.

Напомена: Постоје одређени изузеци у смислу алата – неки власнички компајлери које развијају и одржавају конкретне компаније (попут Keil μ Vision који развија Arm или IAR који развија IAR Systems) имају графичке конфигурације меморије које у позадини креирају одговарајућу скрипту. У GCC свету, морамо експлицитно навести `.ld` датотеку, осим ако користимо готов BSP или *startup* код који већ укључује генерисану скрипту (нпр. Infineon ModusToolbox у својим пројектима испоручује спреман *linker.ld* за циљну плочу).

Илустрација: У наставку је једноставан пример минималне линкерске скрипте за Cortex-M4 микроконтролер са једним Flash и једним RAM меморијским простором. Ова скрипта дефинише меморијске регионе и распоређује главне секције програма у њих:

```
/* Минимална линкер скрипта за Cortex-M4 */
ENTRY(Reset_Handler) /* Почетна рутина након ресета */

/* Дефинисање доступне меморије */
MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K /* Flash почиње на 0x08000000 */
    RAM (rwx): ORIGIN = 0x20000000, LENGTH = 128K /* RAM почиње на 0x20000000 */
}

/* Распоред секција */
SECTIONS
{
    /* Програмски код (.text) иде у FLASH */
    .text :
    {
        KEEP(*(.isr_vector)) /* Vektorska tabla на почетку */
        *(.text*) /* Sav programski kod funkcija */
        *(.rodata*) /* Konstantni podaci */
        _etext = .; /* Kraj .text sekcije */
    } > FLASH

    /* Иницијализовани подаци (.data) – у RAM, али се иницијалне вредности чувају у FLASH-а */
    .data : AT(_etext)
    {
        _sdata = .;
        *(.data*)
        _edata = .;
    } > RAM

    /* Неиницијализоване променљиве (.bss) – у RAM */
    .bss :
    {
        _sbss = .;
        *(.bss*)
        *(COMMON)
        _ebss = .;
    } > RAM

    /* Дефинишемо stack на крају RAM-а */
    ._user_stack :
    {
        . = ALIGN(8);
        _estack = ORIGIN(RAM) + LENGTH(RAM);
    } > RAM
}
```

У овој скрипти:

- У **MEMORY** блоку дефинисана су два региона: FLASH (атрибути *r* за читање, *x* за извршавање) који почиње на адреси 0x08000000 и величине је 512 KB, и RAM (атрибути *rwx*, читање/писање/извршавање, мада код се типично не извршава из RAM па *x* није критичан) који почиње на 0x20000000 и величине је 128 KB. Ове бројке би се прилагодиле конкретном микроконтролеру – у примеру одговарају типичној конфигурацији.
- У **SECTIONS** делу се налазе правила како секције из објектних фајлова скупити и распоредити:
 - **.text секција:** Овде се ставља сав програмски код и константни подаци у Flash. Прво се укључује секција `*(.isr_vector)` – ово је важно за Cortex-M: векторска табела прекида (која обично долази из *startup.o*) мора бити на почетку Flash-а. Директива `KEEP()` осигурава да се та секција не уклони чак и ако линкер ради *колекцију смећа* (*garbage collection*) некоришћених секција. Затим `*(.text*)` укључује све остале .text секције (код функција) из свих објектних фајлова, и `*(.rodata*)` све константе. На крају, поставља се симбол `_etext` на крај .text секције (овај симбол може касније да користи startup код да зна где се завршавају константе у Flash-у). Цела .text секција је мапирана у FLASH меморијски регион.
 - **.data секција:** Овде се распоређују иницијализоване глобалне/статичке променљиве. Кључна је директива `AT(_etext)`, која каже да се почетак .data секције у Flash-у налази на адреси `_etext` (одмах након краја .text). То значи да ће иницијалне вредности ових променљивих бити смештене у Flash непосредно иза кода, а сам садржај .data секције ће током извршавања бити пребачен у RAM (који је овде таргетиран са `> RAM`). Унутар блока, `_sdata` и `_edata` су симболи који обележавају почетак и крај .data у RAM-у. Сви сегменти података `*(.data*)` биће смештени између. Овај механизам омогућава да глобалне променљиве које имају иницијалне вредности буду учитане у RAM при старту програма (обично у `Reset_Handler` рутини) копирањем из Flash-а.
 - **.bss секција:** Овде иду све неиницијализоване глобалне/статичке променљиве (симболички, оне су иницијализоване на нулу). Оне ће бити смештене у RAM (одмах иза .data). Симболи `_sbss` и `_ebss` означавају почетак и крај .bss секције, а `*(.bss*)` и `*(COMMON)` укључују све варијабле које нису добиле експлицитну вредност (`COMMON` обухвата оне који су декларисани а нису дефинисани, типично). Ове променљиве ће такође у `Reset_Handler` бити постављене на нулу (пошто у бинарној слици не заузимају место, већ се само резервише простор у RAM-у).
 - **Стек (stack):** Овде је дефинисана секција `._user_stack` која само поставља симбол `_estack`. Он се израчунава као `ORIGIN(RAM) + LENGTH(RAM)`, што заправо даје крај RAM меморије (адресу након последњег бајта RAM-а). Cortex-M архитектура користи ту адресу као почетну вредност стека (`SP` регистра) – зато се често `_estack` ставља као први елемент векторске таблице. Овде га дефинишемо ради потпуности. Ова секција се мапира у RAM, али сама не заузима простор (осим резервисања симбола) – зато је

на почетку поравнање на 8 бајтова (ALIGN(8)) да би стек био усклађен на 8-бајтну границу, како архитектура захтева.

- Наредба **ENTRY(Reset_Handler)** близу врха скрипте означава *улазну тачку* програма. Она каже линкеру да је симбол Reset_Handler (који би требало да се налази у секцији .text на почетку, јер га startup.o дефинише) почетна адреса на коју ће програм скочити по ресету. У ELF фајлу, овај ENTRY се упише као ознака entry point-а програма.

Ова минимална скрипта је довољна да се изврши успешно линковање и добије ELF чији ће секцијски распоред одговарати микроконтролеру. Наравно, у реалним пројектима скрипта је комплекснија – додају се додатни меморијски региони (нпр. посебан Bootloader регион, различити RAM сегменти ако их има више), секције за *heap*, за C++ *static constructors/destructors* (иницијализација глобалних објеката), секције за меморијски мапиране периферије, и слично. Али суштински концепт је исти.

Из горње скрипте важно је запазити да смо јасно дефинисали где почиње Flash, где RAM, и како су критичне програмске секције распоређене. Без овога, линкер не би знао те информације, па би финална слика била нефункционална на правом хардверу.

4.4.4. Постепена надоградња линкер команде (пример)

Слично као за претходне фазе, приказаћемо етапе побољшања линкер командне линије за један једноставан пројекат. Претпоставимо да имамо `main.o` и одговарајући `startup.o` (који садржи векторску табелу и `Reset_Handler` функцију), јер је то минимум за један микроконтролерски програм. Полазимо од најједноставније команде и постепено додајемо елементе неопходне за потпуни `embedded` сценарио:

1. Основно линковање једног модула:

```
arm-none-eabi-gcc main.o -o main.elf
```

Ово генерише `main.elf`, али као што је објашњено, секције нису правилно распоређене за микроконтролер (користи се подразумевана скрипта). Такав ELF није спреман за флешовање.

2. Додавање стартап кода:

У `embedded` окружењу неопходно је укључити и објектни модул који садржи *startup* код (пре свега векторску таблицу и почетну функцију). Дакле:

```
arm-none-eabi-gcc main.o startup.o -o main.elf
```

Сада `main.elf` садржи и главну апликацију и стартап сегмент. Међутим, секције су *и даље* распоређене по подразумеваној логици, јер још нисмо задали сопствени распоред. Добијамо ELF који укључује векторску табелу и код програма, али вероватно све у погрешном делу меморије (нпр. векторска табела није на почетку стварног Flash-а, већ где је год подразумевана скрипта одредила).

3. Навођење архитектурских опција при линковању:

Иако линкер углавном само копира бинарне кодове из `.o` фајлова, опције попут `-mcpu=cortex-m4 -mthumb -mfloat-abi=softfp -mfpu=fpv4-sp-d16` могу бити корисне и овде. Оне обезбеђују да линкер бира праве верзије runtime библиотеке за дату архитектуру (нпр. може утицати на то коју варијанту `libc` повезује). Додајемо их:

```
arm-none-eabi-gcc main.o startup.o -mcpu=cortex-m4 -mthumb \
-mfloat-abi=softfp -mfpu=fpv4-sp-d16 -o main.elf
```

Сада је генерисан ELF специфичан за ARM Cortex-M4 архитектуру, мада још немамо правилан меморијски мапинг. Овај корак осигурава да ће, ако се линкује било шта из стандардне библиотеке, бити узета варијанта оптимизована за Cortex-M4 (нпр. *nano.specs libc* за M4, ако је доступна).

4. Додавање прилагођене линкерске скрипте:

Критични корак – специфицирамо меморијски распоред помоћу наше скрипте:

```
arm-none-eabi-gcc main.o startup.o -mcpu=cortex-m4 -mthumb \
-mfloat-abi=softfp -mfpu=fpv4-sp-d16 \
-T linker.ld -o main.elf
```


Са -T linker.ld, линкер распоређује секције тачно у складу са оним што смо дефинисали у *linker.ld*. Сада main.elf има .text, .data, .bss и друге секције постављене на исправне адресе у Flash и RAM. Векторска табела ће бити на почетку Flash-а, стек на крају RAM-а, итд. Ово је први потпуно исправан ELF за наш микроконтролер.

5. Додавање опција за дебаг, оптимизацију и библиотеке:

Уводимо опције које смо користили и у претходним фазама, као и спецификацију коришћења оптимизоване C библиотеке. На пример:

```
arm-none-eabi-gcc main.o startup.o -mcpu=cortex-m4 --specs=nano.specs -Og \
-mfloat-abi=softfp -mfpu=fpv4-sp-d16 -mthumb -Wall -g \
-T linker.ld -o main.elf
```

Новине:

- **-Wall** и **-g**: укључују упозорења и дебаг симболе у финалном ELF-у (симболи су већ били у .о, овде се само преносе).
- **-Og**: мада је оптимизација примарно битна у фази компилације, навођење нивоа оптимизације и при линку сигнализира конзистентност (у случају LTO, линкер би могао да зна како да оптимизује преко модула).
- **--specs=nano.specs**: врло значајно – ово каже линкеру да користи *Newlib-nano* библиотеку уместо пуне Newlib. Newlib-nano је верзија C стандардне библиотеке намењена за микроконтролере, знатно мања по величини. Са овом опцијом, ако програм користи функције попут printf, линкер ће их повезати из *libc_nano.a* уместо стандардног *libc.a*. Резултујући ELF је компатибилан са Newlib-nano, што штеди меморију.

6. Индустриски облик линковања (са Map фајлом и оптимизацијом секција):

Последњи корак укључује неке напредне линкер опције преко префикса -Wl, (којим се прослеђују аргументи директно линкеру). Додаћемо генерисање *map* датотеке и укључити опцију уклањања некоришћених секција:

```
arm-none-eabi-gcc main.o startup.o -mcpu=cortex-m4 --specs=nano.specs -Og \
-mfloat-abi=softfp -mfpu=fpv4-sp-d16 -mthumb -Wall -g \
-T linker.ld -Wl,-Map=main.map,--gc-sections \
-o main.elf
```

Нове ставке овде:

- **-Wl,-Map=main.map** – прослеђује линкеру опцију да генерише *map* датотеку са именом *main.map*. Map датотека садржи детаљан распоред свих секција и симбола унутар финалног ELF-а (које адресе су додељене свакој функцији, колико бајтова заузимају секције, који симболи су у којој секцији, итд.). Ово је изузетно корисно за анализу употребе меморије и проверу да ли је неки код правилно смештен.

- **--gc-sections** – (скраћено од *garbage collect sections*) каже линкеру да избаци све секције које нису референциране. Ово ради у комбинацији са `-ffunction-sections -fdata-sections` које смо користили у фази компилације: свака функција је у засебној секцији, па ако не постоји референца на ту функцију (ниједна друга функција је не позива, и није наведена као `entry` или у векторској табlici), линкер ће ту секцију прогласити неповезаном и одстранити је из финалне слике. Резултат је мањи бинарни код, јер се нпр. из библиотека укључују само стварно коришћене функције, остале опадају.

У овом облику, имамо потпуну контролу и увид у процес линковања. `main.elf` је финална извршна слика, `main.map` садржи људски читљив извештај о распореду меморије, а `--gc-sections` осигурава да ништа сувишно није укључено.

Након ове серије корака, можемо рећи да смо постепено изградили команду од најпростије до високо конфигурисане, каква се користи у професионалним окружењима.

Резиме градације (линковање): Етапе развоја линкер командне линије и њихов значај су:

1. **Минимум:** `gcc main.o -o main.elf` – линкује један модул, али без меморијске мапе (неупотребљиво директно на MCU без модификација).
2. **Стартап модул:** додавање `startup.o` – укључивање основног `startup` кода (векторска табела и `Reset_Handler`) што је потребно за *embedded* програм.
3. **Архитектура:** додавање `-mcpu`, `-mthumb`, `-mfpu` – обезбеђивање да линкер бира исправне библиотеке и поставке за Cortex-M (иако сама машинска код мапа долази из `.o` фајлова).
4. **Линкер скрипта:** додавање `-T linker.ld` – исправан распоред секција у меморији, коначно чини ELF валидним за микроконтролер.
5. **Дебаг и runtime:** додавање `-Og -g -Wall --specs=nano.specs` – укључивање дебаг информација, оптимизације, и коришћење оптимизоване runtime библиотеке за *embedded* (Newlib-nano).
6. **Напредне линкер опције:** додавање `-Wl,-Map,...,--gc-sections` – индустријски сценарио: генерисање `map` датотеке за анализу и аутоматско избацивање некоришћеног кода ради минимизације величине.

4.5. Интеграција фаза компилације у оквиру GCC алатке

У пракси, већина описаних корака обавља се аутоматски и транспарентно за корисника, путем једне наредбе за gcc. GCC је пројектован тако да интерно **аутоматски позива препроцесор, компајлер, асемблер и линкер** у исправном редоследу. На пример, један једини позив:

```
arm-none-eabi-gcc -O2 -mcpu=cortex-m7 -o program.elf main.c uart.c startup.s -T linker.ld
```

ће обавити све фазе – препроцесирање оба изворна .c фајла, њихову компилацију у асемблер, асемблирање у објектне модуле, као и линковање заједно са startup.s (претходно асемблованим) – и произвести коначну датотеку **program.elf**. Корисник дакле не мора ручно да креира .i, .s и .o датотеке; они се праве привремено и бришу се по успешном завршетку процеса.

Ипак, важно је разумети постојање ових међукорака и знати како изгледају, јер комплексни *embedded* системи често захтевају увид у сваки од њих. GCC пружа опције попут **-save-temps** које омогућавају да се сачувају привремени резултати – препроцесирани код (.i) и асемблерски код (.s) – тако да могу бити прегледани или анализирани након компилације. Ово је корисно када се сумња на проблем у некој фази (нпр. да ли је препроцесор укључио све што треба, или да ли компајлер генерише очекиване асемблерске инструкције).

Треба имати на уму и да GCC документација експлицитно описује ове четири фазе компилације и одговарајуће суфиксе датотека које настају. Поред тога, постоје и помоћни излази – на пример, *listing* фајл са измешаним C и асемблерским кодом може се добити опцијом **-Wa,-adhln** (assembler listing), али то је само додатни артефакт за преглед, не и улаз ни излаз за наредну фазу.

Коначно, вреди нагласити још једном важност линкерске скрипте у интегрисаном процесу. Док на *host* системима линковање може проћи без посебне скрипте (ослањајући се на стандардне поставке оперативног система), у *embedded* случају **извршна датотека неће бити исправна без тачних информација о меморији**. Зато је део командне линије који се односи на линкер (**-T linker.ld**, опције **-Wl,...**) подједнако битан као и опције компајлера. Структура и садржај линкерске скрипте директно утичу на валидност добијене бинарне слике програма и морају бити прилагођени циљном хардверу (пример смо видели у одељку 4.4.3).

У summary, **GCC интегрише све фазе** тако да је кориснику на располагању једноставан интерфејс (једна наредба за компилацију целог програма), али знање о томе шта се дешава "иза сцене" омогућава дебаговање на нивоу међуфаза и прилагођавање процеса посебним потребама пројекта. На пример, опције које смо демонстрирали (**-E, -S, -c, -o, -I, -D, -Wall, -g, -O, -T** итд.) могу се комбиновано користити да би се по потреби нека фаза извршила одвојено или с прилагођеним параметрима, чиме се добија флексибилност у развојном процесу.

5. Формати резултујућих датотека

Финална фаза процеса превођења С програма за микроконтролер подразумева добијање извршне датотеке у формату који омогућава даљу анализу, тестирање и програмирање циљног уређаја. У овој етапи, сви објектни модули и библиотеке, претходно обједињени током линковања, организовани су у јединствену бинарну слику чија је структура дефинисана изабраним форматом.

Избор формата резултујуће датотеке није произвољан, већ зависи од више практичних чинилаца. Сам формат у основи може бити примењен на различитим архитектурама, али конкретан ELF излаз увек садржи ознаке и атрибуте дефинисане ABI (Application Binary Interface) спецификацијом дате архитектуре (нпр. ARM EABI или x86 ABI), што условљава његову употребу унутар одређеног toolchain-а. Поред тога, избор зависи од подршке коју пружају компајлери, линкери и програматорски алати – нека окружења подразумевано генеришу HEX или S19, док друга користе ELF као примарни артефакт. На крају, на одлуку утичу и захтеви производног и одржавачког процеса: фабричка линија или сервисни алат могу очекивати баш одређени формат (нпр. HEX), док се унутар развоја паралелно користе ELF за дебаговање и BIN за криптографску обраду.

У савременом embedded развоју, најчешће се користи ELF (Executable and Linkable Format) као интермедијарни и вишенаменски формат, који задржава све секције, симболе и debug информације. За разлику од њега, формати Intel HEX, RAW BIN и Motorola S-Record представљају једноставније и компактније облике погодне за директно програмирање Flash меморије микроконтролера. Сваки од ових формата има своје предности и ограничења, па је њихово правилно разумевање предуслов за успешну интеграцију процеса компилације са поступцима програмирања у реалним системима.

Следећа табела пружа сажет и систематизован преглед основних карактеристика ова четири формата:

Табела: Поређење формата резултујућих датотека у развоју firmware-a

Формат	Структура и садржај	Предности	Мане	Типична употреба
ELF (Executable and Linkable Format)	Бинарни формат са заглављем, табелом секција, табелом сегмената, симболима, релокацијама и debug информацијом.	Богатство метаподатака; преносивост; подршка за више архитектура; погодан за дебаговање и анализу.	Велика величина; непрактичан за директно програмирање Flash меморије.	Развој и дебаг (GDB, objdump, readelf); основа за конверзију у HEX, BIN или S19.
Intel HEX	ASCII текстуални формат; сваки ред (record) садржи адресу, дужину података, тип записа, податке и checksum.	Читљив у едитору; подржава дисконтинуиране меморијске регионе; интегритет кроз checksum; индустријски стандард.	~2× већи од бинарног фајла; потребно парсирање при програмирању.	Најчешћи формат за програмирање микроконтролера у производњи; дистрибуција firmware-a у аутомобилској и индустријској електроници.
RAW BIN	Чист низ бајтова (memory dump) без заглавља или адресних ознака.	Најмањи могући фајл; брзо учитавање; једноставан за checksum и криптографске операције.	Нема адресних информација; нема уграђену проверу интегритета; може постати огроман ако постоје офсети.	OTA (Over-The-Air) ажурирања; bootloader протоколи који примају бинарни stream; криптографска верификација (SHA, потписивање).
Motorola S-Record (S19)	ASCII текстуални формат; редови почињу са „S“, садрже тип, адресу, дужину података, податке и checksum. Подржава 16, 24 и 32-битне адресе.	Читљивост; подршка за дисконтинуиране адресе; уграђени checksum; историјски формат за Motorola/NXP платформе.	Већи од бинарног; ређе коришћен ван Motorola/NXP екосистема; потребно парсирање.	Алтернативни индустријски стандард; употреба у системима са Motorola/NXP наслеђем; захтеви одређених производних линија.

5.1. ELF формат

По завршеном линковању, добија се извршна датотека, најчешће у **ELF формату** (Executable and Linkable Format). ELF је стандардни бинарни формат који се користи на Unix/Linux системима за извршне датотеке, објектне модуле, па чак и библиотеке. Он је прихваћен и код крос-компајлера за микроконтролере јер је флексибилан и независан од архитектуре – подржава различите процесоре, ендијаност и величине адресног простора. За потребе *embedded* програмирања, ELF садржи све потребне информације о програму: машински код сегментиран у секције (.text, .data, .bss, итд.), али и симболичке табеле, таблице релокација, програмска заглавља са описом сегмената за извршавање, и опционе дебаг информације. ELF формат подржава двоструку анализу: према табели секција (*section header table*), која описује структуру изворног кода и симболе, или према табели сегмената (*program header table*), која описује начин читавања у меморију при извршавању. У контексту микроконтролера, важнији је распоред секција, јер сегменти одговарају меморијским регијама у које ће секције бити смештене (Flash, RAM).

Иако ELF датотека садржи извршни код, она се обично **не програмира директно** у микроконтролер. Разлог је што ELF носи и метаподатке (нпр. симболе, одређене секције које нису потребне за сам рад програма) и није у формату који типични програмабилни хардвер очекује. Зато се из ELF-а изводе *чисти бинарни формати* погодни за флешовање. Најчешће се срећу три таква формата у *embedded* свету: **Intel HEX**, **RAW BIN** (сиров бинарни фајл), и **Motorola S-Record (S19)** формат.

5.1.1. Структура формата

Executable and Linkable Format (ELF) је стандардни бинарни формат извршних датотека и објектних модула, првобитно развијен за Unix систем V, а данас усвојен широм индустрије. Карактерише га изузетна флексибилност и преносивост – подржана је разноврсна хардверска архитектура, различита ендијаност и ширина адресног простора. ELF датотека почиње главним ELF заглављем, након чега следе табела секција и табела програмских сегмената. Садржи више секција које сегментирају програм (нпр. .text, .rodata, .data, .bss), као и симболичке табеле, таблице релокација и опционе дебаг информације. ELF формату су на тај начин ингерентни сви потребни метаподаци програма – од машинског кода, преко информација за линковање, до симбола и ознака за дебагер. Захваљујући томе, ELF омогућава двоструки приказ садржаја: кроз *section header* табелу (која описује логичку организацију кода по секцијама и симболима) и кроз *program header* табелу (која описује меморијске сегменте намењене читавању при извршавању).

5.1.2. Улога у развојном току

ELF је примарни излаз компилационо-линкерског ланца у *embedded* пројектима. Након линковања, добијена ELF извршна датотека представља комплетну слику фирмвера прилагођену циљном процесору. У току развоја, ELF служи као носилац свих информација потребних за даљу обраду: користи се за анализу величине и распореда секција (size алат), за симболичко дебаговање (алатима попут GDB који из ELF-а читају симболе и дебаг информацију), као и за дисасемблирање машинског кода (objdump). **Дакле, ELF обезбеђује да се изворни код може повезати са машинским инструкцијама током тестирања и дебаговања програма на микроконтролеру.** Међутим, у завршној фази испоруке софтвера (нпр. програмирање у производњи), ELF се типично не користи директно, већ се из њега генеришу једноставнији формати погодни за учитавање у меморију циљног уређаја.

5.1.3. Предности и мане

Основна предност ELF формата је богатство информација и независност од платформе. Како је део званичне ABI (Application Binary Interface) спецификације, ELF је подржан на различитим процесорским архитектурама и оперативним системима. Подржава различите ендијаности и величине адреса, не ограничавајући се на одређени CPU или сет инструкција (Instruction Set Architecture, ISA). ELF обједињује код и податке са симболима, што омогућава *loader*-у или дебагеру да распакује програм управо онако како је линкован. Такође, ELF је проширив – могу се додавати нове секције (нпр. за отисак верзије, проверне суме, прилагођене сегменте) без нарушавања компатибилности. **Мане** ELF-а у контексту микроконтролера произилазе управо из његове сложености. Пошто садржи метаподатке (симболе, секције које нису потребне за извршавање итд.), ELF датотека је знатно већа од минималне бинарне слике и није у формату који типични програматори очекују за упис у Flash. Програматори обично раде са простим хексадецималним или бинарним садржајем меморије, па директно програмирање ELF-а није могуће без претходне екстракције битних делова. Додатно, ELF је бинарни формат специфичан за алатни ланац – нпр. ELF генерисан за ARM EABI садржи ознаке архитектуре, сегменте и секције у складу са ARM ABI конвенцијама – те захтева да алати који га читају подржавају ту специфичну варијанту.

5.1.4. Употреба у контексту микроконтролера (CYT2BL5CAS)

За циљни микроконтролер CYT2BL5CAS (Infineon Traveo II породице) који поседује ARM Cortex-M4 и Cortex-M0+ језгра, компилација пројекта у GCC алатима генерише ELF извршну датотеку у складу са ARMv7-M EABI спецификацијом. У случају мулти-језгарних *firmware*-а, обично се добија засебан ELF за свако језгро (нпр. један за M4, други за M0+), који затим садрже одговарајуће секције за код тог процесора. Тако на пример, у оквиру *ModusToolbox* развојног окружења за CYT2BL5CAS, један *application* може обухватати два *пројекта* – за главно и помоћно језгро – чијом се градњом добијају две ELF датотеке. Ове ELF датотеке потом се могу комбиновано обрађивати: *make* алати Infineon окружења генеришу јединствену HEX слику која укључује податке из оба ELF-а, тако да се читав двојезгарни систем може испрограмирати једним фајлом. Током развоја, ELF за главно језгро се користи за покретање дебаг сесије – нпр. *debugger* (GDB или IDE) учита ELF како би знао адресе симбола и линија кода, при чему се програм извршава на микроконтролеру корак по корак. У производном сценарију, међутим, финална испорука фирмвера за CYT2BL5CAS се не врши у облику ELF датотеке; уместо тога ELF служи као извор из ког се изводе HEX или BIN фајлови који се прослеђују програматору или *bootloader*-у на самом уређају.

5.1.5. Веза са GNU алатима

У GCC/GNU алатном ланцу, ELF формат је централни елемент. GNU линкер (*ld*) по дифолту производи ELF извршну датотеку за задату мету (у нашем случају ARM Cortex-M). Низ пратећих *binutils* алата намењен је манипулацији ELF-ом: *readelf* читава заглавља и приказује структуру ELF-а (нпр. листинг секција, сегмената, симбола), *objdump* може да дисасемблира секције машинског кода или прикаже хексадецимални *dump*, *nm* листа симболе, док *size* приказује величину секција. Алат *objcopy* се користи за конверзију ELF-а у друге формате – он може „*препаковати*“ садржај било које повезане (линковане) извршне датотеке у жељени излазни формат. На пример, команда:

```
arm-none-eabi-objcopy -O ihex program.elf program.hex
```

генерише Intel HEX фајл из ELF-а, док опција *-O binary* производи сирови бинарни фајл. Приликом овакве конверзије, *objcopy* избацује све симболе и релокационе записе – резултат је чист *memory dump* садржај ELF-а, почев од најниже адресе која се у њему појављује. За производњу S-Record фајлова користи се опција *-O srec*. Поред *objcopy*, у оквиру GNU алата постоји и *objdump* који може приказати садржај ELF-а у хексадецималном облику (опција *-s*), што помаже у валидацији да ли ELF садржи очекиване секције и вредности пре конверзије у крајњи формат. Укратко, ELF формат тесно је интегрисан са GNU компајлерско-линкерским током, а *binutils* алати пружају пуну подршку за његову анализу и трансформацију у формате погодне за програмирање микроконтролера.

5.2. Intel HEX формат

Intel HEX формат је текстуални формат у ASCII нотацији који представља садржај меморије у хексадецималном облику. Датотека се састоји од више линија, где свака линија представља један *рекорд* са одређеним бројем бајтова, њиховом адресом у меморији и контролном сумом. Конкретно, свака линија почиње двотачком, затим следи бајт бројача (колико бајтова података та линија носи), па 16-битна почетна адреса, бајт типа записа (нпр. 00 за податке, 01 за крај датотеке, 04 за проширену адресу код већих адресних простора итд.), затим сами подаци (парови хекс цифара), и на крају једна контролна сума за проверу тачности. Овај формат је веома погодан јер је читљив и садржи адресе – нпр. ако програм није континуиран у меморији, HEX фајл може имати "рупе" у адресама између линија. Програматори (алати за флешовање) читају HEX датотеку линију по линију и уписују бајтове на наведене адресе у флеш меморију микроконтролера. Intel HEX је историјски настао 1970-их за потребе учитавања програма са папирне траке у Intel MCS systems, али је и данас широко коришћен због једноставности и поузданости провере (свака линија носи своју контролну суму). Генерисање HEX фајла се обично ради алатом **objcopy**, о чему ће бити речи касније.

5.2.1. Структура формата

Intel HEX је текстуални формат за представљање бинарних података у ASCII нотацији. Датотека је организована у записе (рекорде) по линијама текста, при чему свака линија садржи информацију о низу бајтова и њиховој локацији у меморији. Сваки рекорд почиње двотачком (:) која означава почетак линије, након чега следе поља: (1) један бајт дужине података (у хекс представи, 2 карактера) који казује колико бајтова података тај запис носи (зелени карактери на слици доле), (2) 16-битна почетна адреса тих података (4 хекс цифре) (љубичасти карактери), (3) један бајт типа записа (2 хекс цифре) (црвени карактери), (4) следе N бајтова стварних података ($2*N$ хекс карактера) (плави карактери) и (5) један бајт контролне суме за проверу исправности (2 хекс цифре) (сиви карактери). Типови записа омогућавају проширење адресног простора и означавање почетка/краја фајла. Најважнији су: 00 – запис података (садржи бајтове за упис одређене меморијске области), 01 – крај фајла, 02 – проширена сегментна адреса (за 20-битне адресе), 04 – проширена линеарна адреса (за 32-битне адресе) итд. Захваљујући овим проширеним записима, Intel HEX може адресирати цели 32-битни адресни простор (до 4 GB) упркос томе што појединачни записи података носе 16-битне адресе. Сваки запис садржи и контролни бајт који је израчунат као двокомплементни збир свих претходних бајтова тог записа, што омогућава проверу интегритета – парсер може детектовати грешку ако контролна сума не одговара садржају. Формат тако обезбеђује високу поузданост при преносу података, јер је свака линија самостална целина са уграђеном провером.

```

96:103DF0000070C9545D7ACD033E7BCD033ECD413C7E
97:103E0000626BC94F0F0F0F0FE60FCDDF3D79E60F45
98:103E1000C3DF3D0E3C0600CDA93E0DC2153EC921B3
99:103E200003007EE6C0CA4C3CFE40CA5E3CFE80CA2F
100:103E30001237C315372100004636AA7E7024FEAA29
101:103E4000CA383E25252D7D4CC9D630D8C6E9D8C6FE
102:103E500006F2573EC607D8C60AB7C9CD463FFE3D53
103:103E6000C25B3ECD463FFE20CA633EC94F0F0F0FD7
104:103E70000FE60FCDB83CCDA93E79E60FCDB83CCDCD
105:103E8000A93E798257C9060DCDA93E060AC3A93EAF
106:103E90007EEFFFD30B3E80D3093E00D3097A161481
107:103EA000CD1D3D15C2A03E57C92103007EE630CA94
108:103EB0004C3CFE10C2CA3EDB01E640CAB73E78D396
109:103EC0000B3E0AD3093E08D309C9FE20CA0C37C3EA
110:103ED0000F372103007EE60CC2FD3E3E09D3093EAA
111:103EE00008D30926FADB01E601CAF83ECD1D3D25BF
112:103EF000C2E53EB73E011FC9DB00EEFFC9FE04C2AA
113:103F0000213F3E0CD3093E08D30926FADB01E62007
114:103F1000C21D3FCD1D3D25C20C3FC3F33EDB03B7A1
115:103F2000C9FE08CA0637CA0937CDD23EDA393CE69F
116:103F30007FC9444D210800702C712C722C73C97BF1
117:103F4000875F7A8F57C9CD8A3CE67F47CD433C785F
118:103F5000C954004301420231035400500431083275
119:103F60000C54005010312032305400434031803224
120:013F7000C090
121:000ED80119

```

Слика 1 – Приказ секција (рекорда) унутар HEX фајла

5.2.2. Улога у развојном току

Intel HEX се обично користи као крајњи извозни формат фирмвера који ће бити уписан у меморију микроконтролера. Компилацијом и линковањем добијени ELF се у завршној фази развоја конвертује у .hex датотеку, која садржи исте бајтове машинског кода и иницијалних података као ELF, али организоване у горе описане ASCII записе. Овај формат је погодан за дистрибуцију и читавање на различитим системима – како за ручно прегледање (датотека је читљива у текст едитору), тако и за машинску обраду. У типичној употреби, произвођач софтвера доставља .hex фајл сервисном особљу или производној линији, где програматорски уређај читава тај фајл и уписује његов садржај у *non-volatile* меморију циљног система. HEX формат је тиме постао део *de facto* стандарда у испоруци уграђеног софтвера – многи *integrator*-и и сервисери очекују баш .hex фајл као испратни артефакт уз извршни код. HEX датотека се током развоја не користи само за програмирање микроконтролера, већ и за тестирање у софтверским алатима – може се учитати у виртуелну меморију емулатора који извршава код као на стварном чипу или у симулатор који моделује само поједине делове система. На тај начин омогућено је проверавање рада програма и без физичког хардвера.

5.2.3. Предности и мане

Предности Intel HEX формата укључују читљивост, преносивост и садржајност информација о адресама. Пошто је у текстуалном облику, .hex датотека се може отворити у било ком едитору, што олакшава брзу инспекцију или чак ручну измену малог броја бајтова ако је неопходно. Формат носи експлицитне адресе за сваки блок података, те може представљати и не-континуирану меморију без потребе да се празни опсези испуњавају нулама (за разлику од сировог бинарног фајла који би морао садржати и „рупе“ као стварне бајтове). Тако, ако програм користи меморију од 0x1000 до 0x1FFF и од 0x3000 до 0x30FF, HEX фајл ће имати два блока записа и неће расти због празног простора између. Ова економичност формата погодна је за уређаје са више раздвојених меморијских региона. Штавише, сваки запис има контролну суму, што повећава поузданост – програматор може да верификује интегритет сваког блока приликом читавања. Intel HEX је и историјски проверен формат: настао је 1970-их за Intel MCS системе са папирном траком, али се одржао до данас управо због једноставности и поузданости.

Мане HEX формата произилазе углавном из његове ASCII природе. У поређењу са бинарним фајлом, .hex датотека је већа (јер сваки бајт представља два карактера, плус структурални карактери за адресу и др.), што значи и спорије читавање ако је проток ограничен. Међутим, та разлика се делимично ублажава јер укупна величина HEX фајла и даље остаје релативно мала у односу на капацитет меморије микроконтролера (нпр. ~2× већа од бинарног за исти садржај). Дакле, HEX је већи и спорији за пренос од BIN-а, али пошто се ради о релативно малим датотекама, разлика у пракси најчешће није проблем. Такође, при обради HEX датотеке свака линија мора да се парсира и провери путем контролне суме, што уноси мало додатног посла за програматор у односу на директан упис чистог бинарног тока. Поред тога, HEX формат је замишљен за једноставан, линеаран меморијски простор: иако омогућава адресирање до 32-бита (запис типа 04), није погодан за системе са сложенијим меморијским организацијама као што су банковање или произвођачки специфичне мапе меморије. У пракси, ове мане ретко представљају проблем, па је HEX изузетно распрострањен у употреби.

5.2.4. Употреба у контексту микроконтролера (CYT2BL5CAS)

У случају микроконтролера CYT2BL5CAS, Intel HEX формат је типичан избор за програмирање Flash меморије. Развојни алати Infineon ModusToolbox, након успешне израде пројекта, могу генерисати .hex датотеку која садржи целокупан фирмвер. Како је поменуто, код овог модела постоје два процесорска језгра – уколико се оба користе, HEX фајл ће укључити записе за секције оба језгра (нпр. један континуирани блок за М4 код од 0x10000000 па надаље, затим блок за М0+ код на одговарајућој адреси, итд.). Инфинионов (Infineon) систем градње, заснован на *Makefile*, аутоматски спаја појединачне .hex из сваког потпројекта у јединствену .hex датотеку за целу апликацију. Ова датотека се затим може директно проследити алату за програмирање – било да се ради о командној линији (cyflash, OpenOCD скрипти) или GUI алату (нпр. Infineon/Cypress Programmer) – који ће је анализирати линију по линију и уписати одговарајуће бајтове у Flash меморију микроконтролера. Уколико се CYT2BL5CAS фирмвер дистрибуира трећим странама (нпр. интеграторима у возилу), највероватније ће то бити управо у HEX формату, јер је широко подржан и компатибилан са различитом

опремом за програмирање у аутомобилској индустрији. Треба напоменути да је Intel HEX у овом контексту само контејнер – сам микроконтролер нема „свест“ о формату, већ очекује да програматор упише одговарајуће бајтове на права места у меморији. HEX формату зато претходи корак парсирања од стране *host* алата, који за CYT2BL5CAS типично обавља рачунар или програматорски уређај повезан са линијом за производњу.

5.2.5. Веза са GNU алатима

GNU *binutils* пакет подржава генерисање и манипулацију Intel HEX фајлова кроз алат *objcopy* и сродне. Као што је већ илустровано, конверзија ELF-а у HEX се остварује командом попут:

```
arm-none-eabi-objcopy -O ihex program.elf program.hex
```

Алат аутоматски преводи све секције означене за учитавање у одговарајуће HEX записе. Сваку секцију (нпр. *.text* која почиње на 0x0, или *.rodata* која се може налазити на вишој адреси) *objcopy* ће преточити у низ одговарајућих :NN AAAA 00 ... CC линија у излазу. Уколико адресни размак између две секције пређе 64 KB, *objcopy* убацује потребан *extended linear address (04)* запис да би повећао базну адресу за наредне податке. GNU алат заправо води рачуна о свим детаљима формата, укључујући и рачунање контролних сума на крају сваке линије. Програмер стога не мора ручно руковати структуром *.hex* фајла – довољно је да позове *objcopy* са одговарајућим параметром. Осим *export-a*, *binutils* омогућава и *import*: на пример, *objcopy* или *objdump* могу прочитати постојећи HEX фајл (наводећи *--input-target=ihex*) и третирати га као да је у питању објектна датотека, што је корисно ако се, рецимо, жели преbacити *.hex* назад у ELF ради анализе или упоредити са оригиналним ELF-ом. Поред *objcopy-a*, корисни су и специјализовани алати као што је *srec_cat* из *SRecord* пакета, који могу спајати, делити или мењати HEX фајлове – нпр. спајање више *.hex* датотека за различите меморијске регије у један, или генерисање филера (*--fill* опција) за попуњавање празнина одређеним бајтом (попут 0xFF). У контексту GNU алата, Intel HEX се третира као један од могућих излазних формата BFD библиотеке, што значи да је подржан на свим архитектурама и да је директно доступан у уобичајеним *toolchain*-овима за ARM Cortex-M (нпр. *arm-none-eabi* верзију *objcopy*). Практично, свака интеграција GCC компајлера за микроконтролере долази и са могућношћу да се добије HEX – било позивом *objcopy* ручно, било подешавањем опција линковања/постпроцесирања у IDE окружењима да то ураде аутоматски након успешне компилације.

5.3. RAW бинарни формат (.bin)

RAW бинарни формат (.bin) је најједноставнији могући формат – низ бајтова идентичан бајтовима који треба да се упишу у меморију, без икакве додатне структуре или информација. Сирови бинарни фајл представља *меморијски дамп* програма, обично тачно оних секција које се налазе у непрекидном опсегу адреса. При конверзији ELF-а у .bin, одбацују се сви симболи, заглавља и вишак информација, и добија се само секвенца бајтова која одговара садржају флеша (и евентуално других меморија ако се спајају у један фајл). GNU објектору алат омогућава ову конверзију: на пример команда

```
arm-none-eabi-objcopy -O binary program.elf program.bin
```

узима ELF и ствара .bin фајл. Према документацији, када се објектору користи за генерисање raw binary датотеке, он ефективно производи меморијски дамп укупног садржаја ELF-а – од најнижег до највишег адресног бајта садржаног у ELF-у – одбацујући све симболе и релокације. Битно је напоменути да .bin не носи информацију о томе на коју адресу ти бајтови треба да се упишу; претпоставља се подразумевани почетак (нпр. код већине микроконтролера почетак флеша је или 0x00000000 или нека позната базна адреса). Због тога, .bin формат се углавном користи када се читава слика програма ставља на почетак флеш меморије. Ако је потребно флешовати програм који не почиње од 0 или ако програм обухвата више одвојених меморијских области, Intel HEX је погоднији, јер носи адресе.

У пракси, многи произвођачи алата и IDE-ови (нпр. KEIL uVision, IAR EWARM, GCC toolchain) омогућавају генерисање HEX или BIN датотека из ELF-а. Неки *debugger*-и и *програматори* могу чак директно учитати ELF (користећи информације из ELF заглавља о сегментима за читавање). Ипак, имајући у виду да HEX и BIN представљају стандард у размени фирмвер слика (нпр. HEX за надоградњу софтвера у сервису, или BIN за брзо читавање преко bootloader-а), важно је разумети њихову структуру и разлике.

5.3.1. Структура формата

Сирови бинарни формат (.bin) представља низ бајтова идентичан низу који треба уписати у меморију, без икаквих додатних заглавља или метаподатака. Другим речима, .bin фајл је чист *dump* садржаја меморије – секвенцијални приказ бајт по бајт, у истом распореду како ће бити смештени у циљном уређају. Не постоје информације о адресама, величинама секција, нити контролне суме уграђене у сам фајл. На пример, ако је програм компилацијом и линковањем позициониран тако да заузима простор Flash меморије од адресе 0x00000000 до 0x0000FFFF, резултујућа бинарна датотека ће садржати управо тих 64 KB, распоређених у непрекидном низу. Први бајт датотеке одговара садржају на адреси 0x00000000, а последњи на адреси 0x0000FFFF. Формат не разликује празну меморију од стварне нуле – ако у програму постоје „рупе“ између секција, оне ће се у .bin извозу појавити као бајтови попуњени нулама (или подразумеваним *fill pattern*-ом) како би се сачувала континуалност адресног опсега. Дакле, .bin директно одражава бинарни садржај целокупне меморијске слике програма, без логичких разграничења.

5.3.2. Улога у развојном току

RAW бинарни формат се често користи у ситуацијама када је потребно брзо и директно учитати програм у меморију или пренети *firmware* преко протокола који не трпи додатне форматиране структуре. Пример је *bootloader* који очекује да добије бинарни *stream* бајтова преко UART-а или USB-а – у том случају, слање .hex датотеке би захтевало да *bootloader* парсира ASCII линије и рачуна контролне суме, што повећава комплексност, док пријем .bin фајла значи да се сваки добијени бајт директно уписује на следећу адресу у Flash. У развоју, бинарни фајл је користан и за емулацију целе меморије: неки симулатори или софтверски модели микроконтролера могу директно учитати .bin као слику Flash-а. Такође, .bin се користи за генерисање *checksum*-а или дигиталног потписа фирмвера – често се најпре из ELF-а извуче BIN, па се над њим израчуна нпр. SHA-256, јер BIN представља тачан низ бајтова који ће бити у уређају. У производњи, међутим, чист BIN формат се ређе размењује самостално, управо због недостатка информације о адреси: ако се фирмвер не линкује од почетне адресе меморије или ако постоји офсет, .bin фајл то не садржи, па би програматор морао „знати“ на коју адресу да га прочита. Због тога је улога BIN формата најчешће комплементарна HEX-у – користи се интерно у оквиру алатног ланца или у специфичним сценаријима, али не као универзални формат испоруке трећим лицима.

5.3.3. Предности и мане

Највећа предност сировог BIN формата је једноставност. Будући да нема заглавља, парсирање није потребно – било који алат за програмирање може једноставно учитати цео фајл у меморију почев од подразумеваног базног адресног офсета. Ово га чини и најефикаснијим по простору: .bin фајл је најмањи могући приказ програма (сваки бајт у фајлу директно одговара бајту у уређају). За скучене меморијске медијуме или брзи пренос, мања величина може бити значајна предност. Такође, једноставност формата доноси и повољности у погледу интероперабилности – готово сваки програматор или *flasher* алат подржава учитавање бинарног дампа. **Мане** овог формата произилазе из одсуства адресне информације. За разлику од HEX/S19, BIN не зна ништа о томе на коју апсолутну адресу се његов први бајт односи. Уобичајено је да се претпостави да BIN почиње од 0x00000000 (тј. од почетка Flash-а код већине микроконтролера). Ако је то тачно, нема проблема – фирмвер који је линкован од 0x0 ће бити коректно представљен BIN фајлом који почиње садржајем вектор табеле на 0x0 итд. Међутим, ако програм није смештен од почетка Flash-а (рецимо, код има офсет јер на почетку меморије резидира *bootloader*), .bin фајл који га садржи морао би имати празне бајтове за тај офсет, иначе би се све адресе помериле. То значи да BIN може потенцијално постати веома велик ако програм почиње на високој адреси – нпр. програм од 1 KB који почиње на 0x8000000 захтевао би BIN фајл од 128 MB (испуњених нулама до те адресе). Из тог разлога, BIN је непрактичан за не-континуиране меморијске слике; HEX или S-Record су ту далеко ефикаснији. Још један недостатак је одсуство провере интегритета – у BIN фајлу нема контролних сума нити редундантних информација, па ако се један бајт промени услед грешке при преносу, то се не може детектовати осим упоређивањем са оригиналом. Стога се BIN обично преноси у комбинацији са додатним механизмима за проверу (нпр. спољни CRC који се шаље посебно, или ослањањем на комуникациони протокол који гарантује доставу).

5.3.4. Употреба у контексту микроконтролера (CYT2BL5CAS)

За циљни микроконтролер CYT2BL5CAS, BIN формат се углавном користи посредно. На пример, током развоја у ModusToolbox окружењу, могуће је добити .bin фајл позивањем `make build` циљева који интерно користе објсору за конверзију ELF-а у BIN. Овај бинарни дамп садржи садржај Flash меморије предвиђен за програмирање. Infineon-ов bootloader (ако је присутан) не користи читавање BIN датотека преко USB диска, већ се фирмвер програмира у HEX формату или путем debug порта. Ипак, један реалан сценарио за BIN на овој платформи је употреба у *OTA* (Over-The-Air) надоградњама или преко ауто-дијагностичких алата: уколико се *firmware* шаље преко CAN-а, LIN-а или другог протокола до микроконтролера, серверски део софтвера може слати управо сурови бинарни садржај, бајт по бајт, а микроконтролер (његов bootloader) ће те бајтове уписивати у Flash. У тим ситуацијама, уносни формат на страни микроконтролера је бинарни *stream*, па се исплати држати *firmware* у BIN формату и на *PC* страни да би се избегло парсирање. У случају CYT2BL5CAS, који има вишемегабајтни Flash, BIN формат за целокупан *firmware* може бити више мегабајта велик, али на производној линији или сервисном алату то обично није препрека. Треба приметити и да неки *debug* алати (нпр. Segger J-Link) могу директно примити ELF или HEX, али у одсуству истих, корисник може проследити BIN уз ручно задавање базне адресе (нпр. командама у конзоли: `loadbin program.bin, 0x10000000`). То показује да BIN има улогу *најнижег заједничког имениоца* – формат који ће увек бити могуће уписати, мада понекад уз додатно знање о адреси.

5.3.5. Веза са GNU алатима

GNU објсору омогућава лак прелаз из ELF у BIN формат. Већ наведеном командом `-O binary` добија се .bin датотека избацавањем свих секција које нису за читавање и конкатенацијом осталих према њиховим меморијским адресама. Важно је нагласити да објсору при том поштује реалне адресе секција: он ће у .bin фајл уврстити и нуле за све празнине између секција ако постоји размак. Корисник може опционо задати *попуну* празнина помоћу опције `--gap-fill=<вредност>` (нпр. `0xFF` уместо нула) ако жели специфичан образац. Друга корисна опција је `--pad-to=<адреса>` којом се .bin фајл продужава нулама до одређене дужине – рецимо да би се осигурало да фајл обухвата читав дефинисани меморијски регион. Ове могућности су битне уколико се BIN користи за израчунавање *checksum*-а: потребно је прецизно дефинисати које бајтове он садржи. Поред објсору-а, ту је и алат `xxd` (није део GNU binutils, већ Unix алат) којим се BIN фајл може пресликати у хексадецимални текст ради лакшег прегледа – корисно за визуелну проверу садржаја. У контексту комплетног *toolchain*-а, многи IDE-ови (KEIL uVision, IAR EWARM, GCC Makefile скрипте) нуде директно генерисање BIN датотеке као артефакта билд процеса. У случају GCC, ово је често реализовано позивом објсору након линковања, док комерцијални алати имају интегрисане конвертере. За инверзну операцију (читавање BIN и претварање у ELF за анализу), GNU асемблер или линкер могу користити *binary input* формат: нпр. `ld -b binary program.bin -o program.o` ће BIN третирати као објектни фајл без архитектуре и омогућити да се његови садржаји мапирају на секцију у новом ELF-у. Овакав поступак је ређи, али сведочи о томе да BIN формат има своје место чак и у оквирима напредних алата – као најосновнији облик представљања низова бајтова.

5.4. Motorola S-Record формат (S19)

Поред Intel HEX-а, чест је и **Motorola S-Record (S19)** формат – сличан ASCII хекс запису линија. Алати **objcopy** са опцијом **-O srec** може генерисати S-Record фајл. Разлика је углавном у синтакси линија (S-Record линије почињу са 'S' и имају мало другачију организацију адресних поља).

5.4.1. Структура формата

Motorola S-Record (често називан *.s19* по екстензији) је формат сличне намене као Intel HEX, али другачије синтаксе, настао у Motorola компанији средином 1970-их за M6800 и касније M68000 породицу процесора. Као и HEX, S-Record је ASCII формат који линијама текста представља адресиране бајтове са контролном сумом. Сваки запис почиње словом S уместо двотачке. Одмах након тога следи број који означава тип записа (нпр. S1, S2, S3 за записе података са 16, 24 или 32-битним адресама; S0 за хедер/коментар; S5 за бројач записа; S7, S8, S9 за завршне записе са почетном адресом извршавања). После ознаке типа долази један бајт дужине (број бајтова који укључује адресу + податке + један бајт за чексум), затим адресно поље (дужине зависно од типа записа: 2 бајта за S1, 3 за S2, 4 за S3), потом сами подаци, и на крају 1 бајт контролне суме. Контролна сума се рачуна као једнобајтни комплемент (1-комплемент) збира свих бајтова дужине, адреса и података – тако да укупан збир свих бајтова укључујући суму даје 0xFF. Будући да постоји више типова завршних записа (S7, S8, S9) условљених величином адресе почетне тачке извршавања, S-Record формат је у старту био предвиђен да подржи и 32-битне системе (S3/S7 за 32-бит адресе) паралелно са 16-битним (S1/S9). Уобичајена екстензија *.s19* долази отуда што су 16-битни записи података означени са S1, а завршни са S9 – међутим, за веће адресе могу се користити и *.s28* или *.s37* екстензије (у зависности од највишег типа записа података). Формат дозвољава и коментарске линије (оне које не почињу са 'S'), што може бити корисно за додавање описа унутар фајла.

```

program.s19
1  S00E000070726F6772616D2E733139EE
2  S31510000000000002081B0100100D0000007D01001009
3  S3151000001000000000000000000000000000000CA
4  S315100000200000000000000000000000000007901001030
5  S31510000030000000000000000000000790100107901001096
6  S31510000040B5010010F10100102D0200106902001018
7  S31510000050A5020010E10200101D0300105903001044
8  S315100000607901001079010010790100107901001052
9  S315100000707901001079010010790100107901001042
10 S315100000800448054B10B5834203D0044B002B00D017
11 S31510000090984710BD880800088080008000000006E
12 S315100000A006480749091A8B10C90FC91810B5491007
13 S315100000B003D0044B002B00D0984710BD88080008C9
14 S315100000C088080008000000010B5074C2378002BA4
15 S315100000D009D1FFF7D5FF054B002B02D0044800E0ED
16 S315100000E000BF0123237010BDC00E00080000000E1
17 S315100000F0000F0010054B10B5002B03D0044905481E
18 S3151000010000E000BFFFF7CCFF10BDC04600000000A6

```

Слика 2 – Приказ секција унутар S19 фајла

5.4.2. Улога у развојном току

S-Record се историјски користио у развоју софтвера за Motorola/FreeScale (данас NXP) микроконтролере и процесоре. Данас је његова улога веома слична Intel HEX-у: служи као један од стандардних формата за дистрибуцију firmware-a, нарочито у оквирима или алатима који потичу из Motorola екосистема. На пример, софтвер за програмирање EEPROM/EPROM меморија 80-их и 90-их година често је захтевао S19 фајл као улаз. У савремено доба, NXP CodeWarrior, Green Hills и други компајлери могу да емитују S-Record директно. У *embedded* току, тимови који су традиционално на Motorola/NXP технологији могу наставити да користе S19 као примарни формат за флешовање, посебно ако су алати на производној линији подешени тако. Такође, уколико постоји потреба да се firmware учита у симулатор или да се споји више image-ева (рецимо *bootloader* и *application*), постоје алати специјализовани за S-Record манипулацију. На крају, S-Record и Intel HEX су функционално еквивалентни – улога им је да пренесу апсолутне бајтове програма са свим потребним адресним информацијама до програматора. Често је избор између њих питање алатне подршке или договора у оквиру организације. Ако, на пример, добављач захтева S19 датотеку за надоградњу ECU уређаја, онда ће се користити тај формат.

5.4.3. Предности и мане

Предности S-Record формата су углавном исте као и код Intel HEX-a: читљив је (ASCII), носи адресе, има контролне збирове по запису. Одређене синтаксне разлике не дају значајну техничку предност једном над другим – оба формата су довољно поуздана и раширена. S-Record има флексибилност у смислу адресног опсега (типа записа S1/S2/S3) и може кодирати до 32-битних адреса. Једна од потенцијалних предности Motorola S-Record формата јесте могућност употребе краћих записа, што је нарочито корисно код уређаја са ограниченим пријемним бафером. Сваком S-Record реду претходи поље које дефинише укупну дужину записа, па се број података у једној линији може смањити у складу са ограничењима пријемне стране. Уобичајена је дужина од 16 бајтова података по линији, иако је овај параметар могуће конфигурисати. За поређење, Intel HEX формат често користи линије са 16 или 32 бајта података, али и он омогућава прилагођавање дужине у складу са потребама система. **Мане** S-Record формата су махом исте природе као и код HEX-a: већа величина у односу на бинарни, потреба за парсирањем. Једна разлика је у нумерацији типова – S-Record има чак 10 типова записа (S0–S9) што је више од HEX-ових 6 – али то не чини формат сложенијим за коришћење, већ само захтева да алати и програмери буду свесни који типови су релевантни (најчешће S0, S1/2/3, S7/8/9). Пошто не постоји универзални стандард који диктира да ли ће се користити HEX или S19, једна од практичних мана је могућа некомпатибилност између тимова или алата који очекују различите формате – но, пошто су конверзије тривијалне уз одговарајуће алате, ово се лако премости.

5.4.4. Употреба у контексту микроконтролера (CYT2BL5CAS)

Платформа CYT2BL5CAS и њен развојни екосистем углавном преферирају Intel HEX, али теоријски ништа не спречава употребу S-Record формата. На пример, ако би производни процес у некој фабрици захтевао .s19, могло би се из постојећег ELF-а генерисати S-Record уместо HEX-а. Infineon алати као *Cypress Programmer* тренутно користе HEX, па у оквиру овог рада S-Record није коришћен у пракси. Ипак, CYT2BL5CAS је Cortex-M микроконтролер – за њега важи да су S-Record и HEX подједнако способни да пренесу firmware. Уколико би се, рецимо, фирмвер испоручивао неком *third-party* програматору који инсистира на S19 формату, једноставна конверзија би обезбедила компатибилност. Такви случајеви нису неуобичајени у индустрији – разне компаније имају интерне стандарде. Због тога је добро што CYT2BL5CAS нема ограничења у том погледу: сам контролер не „види“ формат, а алати за њега (нпр. *open-source* OpenOCD или *proprietary* Segger J-Flash) обично подржавају и HEX и S19 учитавање. Укратко, иако S-Record није експлицитно коришћен у нашем примеру, знање о њему осигурава да се firmware за овај микроконтролер може лако адаптирати за било који тражени формат дистрибуције.

5.4.5. Веза са GNU алатима

GNU објектору директно подржава креирање S-Record датотека преко опције `-O srec`. На пример:

```
arm-none-eabi-objcopy -O srec program.elf program.s19
```

резултира у томе да сваки сегмент из ELF-а буде преточен у одговарајуће S-записе. При том, BFD библиотека унутар *objcopy*-а аутоматски бира да ли да користи S1, S2 или S3 тип записа на основу величине адресе која се јавља. За Cortex-M4/M0+ адресни простор (до 32-бит), користиће S3 записе за податке и S7 као крајњи запис са стартном адресом. Ако је пожељно форсирати одређени тип (нпр. да увек користи S3 иако су адресе 24-битне), објектору нуди опцију `--srec-forceS3`. Такође, може се ограничити дужина линије са `--srec-len=<N>` ако циљни програматор има ограничење у броју бајтова по запису. Осим објектору, споменути алат `srec_cat` из *SRecord* пакета пружа веома грануларну контролу над S-Record датотекама – може комбиновати више .s19 датотека, генерисати контролне скупове на нивоу целог фајла, конвертовати S19 у HEX и обрнуто, итд. У GCC алатном ланцу, директна генерација .s19 најчешће иде преко објектору пост-процеса (ретко ће сам *IDE* понудити S19 без додатне скрипте). Међутим, кориснику је то транспарентно – исто као и за HEX, може једним позивом добити S-Record. Уколико бисмо желели да упоредимо HEX и S19 генерисане из истог ELF-а, могли бисмо користити `srec_cmp` алат који проверава еквиваленцију две датотеке независно од формата. Ово само илуструје да GNU/open-source екосистем има добру подршку и за Motorola S-Record паралелно са Intel HEX-ом, премда је други можда нешто чешћи у ARM свету.

5.5. Критеријуми избора формата у производном току

При избору одговарајућег излазног формата за дистрибуцију и програмирање *firmware*-а, инжењер мора узети у обзир више техничких и практичних критеријума. Сва три главна формата (HEX, BIN, S19) могу носити исти корисни садржај – машински код микроконтролера – али се разликују по начину на који га представљају и по захтевима које намећу на околни алатни ланац. У наставку су наведени кључни фактори који утичу на избор формата у оквиру производног процеса:

- **Структура меморијске слике:** Ако је *firmware* распоређен у више одвојених регија меморије (нпр. више неповезаних партиција Flash-а, или комбинација Flash и EEPROM садржаја), формати који подржавају „скокове“ у адресама су пожељни. Intel HEX и Motorola S-Record су у стању да обухвате дисконтинуиране адресне опсеге у једној датотеци – сваки сегмент меморије биће представљен посебним записима са својим адресама. Насупрот томе, RAW BIN може представљати дисконтинуирани садржај само убацивањем пуно *padding*-а (нпр. нула) за празне делове, што може учинити фајл огромним и непрактичним. Стога, за комплексније меморијске мапе и веће офсете, HEX или S19 ће обично бити први избор. У случају CYT2BL5CAS, који има монолитну интерну Flash од 4 MB и опционално додатну EEPROM/Emulated-EEPROM, HEX је погодан да у једној датотеци носи и главни код и евентуалне додатне податке (нпр. калибрације у EEPROM сегменту), док би BIN захтевао одвојене фајлове или један велики фајл са пуно празног простора.
- **Захтеви алата и протокола:** Производни ток често подразумева коришћење одређених програматора или *in-circuit* тестера који имају сопствени софтвер. Треба проверити које формате ти алати подржавају *out-of-the-box*. Историјски гледано, многи комерцијални програматори (Data I/O, Segger, P&E, итд.) подржавају Intel HEX и Motorola S-Record као стандардне опције за увоз *firmware*-а. Ако је алат ограничен на један од та два, избор је јасан. На пример, ако линија за програмирање у фабрици користи опрему која „очекује“ .hex фајл, онда нема разлога форсирати S19. Слично томе, ако се *firmware* дистрибуира крајњем кориснику који ће га уписивати преко, рецимо, bootloader-а са SD картице, можда је прикладније дати бинарни .bin (јер је вероватно да је bootloader на уређају програмиран да учита бинарни фајл са познате локације на картици). Критеријум је дакле компатибилност са постојећим алатима и протоколима. У случају нашег циљног система, Infineon/Cypress алати нативно раде са .hex, а OpenOCD такође лако прихвата Intel HEX – то је утицало да се HEX користи као примарни формат током развоја и програмирања.
- **Контрола интегритета и грешака:** У производном окружењу, поузданост при преносу *firmware*-а је од највише важности. Формати са уграђеним проверама (HEX, S19) имају предност јер сваки запис носи контролни збир. Иако сам програматорски софтвер типично имплементира додатну верификацију (нпр. чита назад уписане податке и пореди их са оригиналом, или одржава CRC током преноса), додатни ниво контроле на нивоу фајла није на одмет – посебно у случају ручног руковања датотекама. На пример, приликом слања *firmware*-а е-поштом или складиштења на FTP-у, ASCII формат је мање склон проблемима интерпретације у односу на бинарни (где би, рецимо, одређена комбинација бајтова могла изгледати као управљачки карактер). Такође, ако се *firmware*

едитује или генерише скриптом, лакше је уочити грешку у HEX-у/S19 (нпр. погрешна дужина линије) јер ће провера суме заказати, док бинарни фајл нема непосредну индикацију грешке. Стога, за високо-заштићене процесе (аутомобилска индустрија, авио индустрија) обично се преферирају формати са интерним интегритетом.

- **Величина и брзина учитавања:** Ако је брзина програмирања критична – нпр. пуњење *firmware*-а у фабрици мора да се обави што брже због великог обима производа – бинарни формат може имати благу предност. Парсирање HEX/S19 формата додаје неколико милисекунди или чак стотинки по фајлу, што се у масовној производњи може сматрати занемарљивим у односу на саме времене програмирања Flash-а (које се мери у секунди за мегабајте). Ипак, ако програматорски алгоритам пише бајт-по-бајт док чита ток (*stream*) без великог бафера, бинарни *stream* ће теоретски бити најефикаснији. Величина фајла може играти улогу када се *firmware* дистрибуира преко ограничених канала (нпр. ОТА ажурирање преко мобилне мреже): ту је сваки бајт битан, па бинарни формат штеди ~50% у односу на HEX. У тим случајевима, инжењери неретко бирају да ОТА пакет буде бинарни, евентуално криптован, како би се минимизирао саобраћај. За CYT2BL5CAS од 4 MB Flash-а, HEX фајл би био ~8 MB, што ни данас није превелико за преузимање, али ако се ажурирање врши преко скувих или спорих веза, уштеда није занемарљива. Стога критеријум брзине/величине може превагнути у корист BIN-а у специфичним сценаријима (нпр. *field update* преко мреже).
- **Компатибилност са безбедносним механизмима:** Уколико је део производног процеса додавање криптографске заштите (потписивање или енкрипција *firmware*-а), важно је размотрити који формат се најлакше уклапа у тај ток. Рецимо, да би се израчунао дигитални потпис на читав *firmware*, најједноставније је радити над бинарним сажетком; ако је доступан BIN фајл, он се директно *hash*-ује. Ако је на располагању само HEX, потребно је или га претходно превести у бинарни облик (што је опет један корак више) или дефинисати да се потпис рачуна по секцијама на основу садржаја HEX записа. Слично, ако се *firmware* шифрује, бинарни формат је природни носилац шифрованих података (шифровани бафер одређене дужине), док би HEX у том случају носио шифроване бајтове али би и даље био већи ~2× због ASCII експанзије, што не доноси бенефит већ само повећава *overhead*. Због тога, многи системи који имплементирају сигурно бутовање и потписивање, интерно користе BIN формат (или неки сопствени контејнер) за криптографске операције, чак и ако се корисницима споља и даље дистрибуира HEX ради компатибилности. Дакле, критеријум безбедносне интеграције може утицати на то да ли се у одређеној фази конвертује у BIN.

У закључку, избор формата у производном току није универзална одлука, већ резултат балансирања наведених фактора. Уколико је примарни циљ преносивост и стандардизација – посебно када више страна размењује *firmware* – Intel HEX је често први избор због широко распрострањене подршке. Ако се ради о унутрашњем процесу где су алати под контролом и битна је ефикасност, RAW BIN може бити погодан (уз услов да је меморијска слика компактна од почетка адресног простора). Motorola S-Record се најчешће бира у окружењима која већ имају тај *legacy* или специфичан захтев, али технички нуди исте предности као HEX и равноправна је алтернатива. У случају

платформе као што је CYT2BL5CAS, где је развој кроз GCC и производња ослоњена на модерне алате, разумно је пратити задати ток – компајлирати до ELF, затим генерисати HEX за програмирање, а BIN користити по потреби (нпр. за верификацију или ОТА пакете). На тај начин се постиже и тачност и ефикасност, уз минималан ризик од неслагања алата у ланцу.

6. Практична анализа GNU binutils алата за обраду објектних и бинарних датотека

У развоју софтвера за микроконтролере, посебно у сложеним *embedded* пројектима, неопходно је добро познавање алата за испитивање, анализу и конверзију објектних и извршних датотека. GNU компилациони ланац, поред самог компајлера (*gcc*), обухвата и пакет алата познат као **GNU binutils** (*GNU Binary Utilities*), који омогућавају увид у садржај ELF датотека, конверзију у различите формате, дисасемблирање машинског кода, мерење меморијске потрошње, анализу симбола и низ других активности кључних за развој у *bare-metal* окружењима.

Формално, појам „GNU binutils алати“ означава целину свих корисничких програма из овог пакета, међу којима су најважнији: **objcopy**, **readelf**, **nm**, **size**, **objdump**, али и други попут **as** (GNU асемблер) и **ld** (GNU линкер). Сви они се дистрибуирају у оквиру истог пакета, независно од архитектуре, али приликом крос-компајлирања морају бити специјализовани за конкретну мету (*target*).

GNU binutils у крос-компајлерским окружењима

При развоју *firmware*-а за ARM Cortex-M микроконтролере (као у овом раду, са примером CYT2BL5CAS из Infineon TRAVEO™ T2G-B-E породице), програмер ради на хост систему заснованом на x86 архитектури (нпр. Windows или Linux), док је извршни код намењен ARM процесору. У таквом сценарију користи се **крос-компајлер** (*cross-compiler*), који генерише машински код за архитектуру различиту од хост архитектуре.

GNU binutils алати у том контексту увек носе префикс који указује на архитектуру мете. На пример:

- **arm-none-eabi-objcopy**, **arm-none-eabi-readelf**, **arm-none-eabi-objdump** – за ARM Cortex-M (без оперативног система, *embedded application binary interface*),
- **aarch64-none-linux-gnu-objcopy** – за ARM Cortex-A са Linux окружењем,
- **riscv64-unknown-elf-objcopy** – за RISC-V архитектуру без ОС-а,
- **mipsel-none-elf-objcopy** – за MIPS,
- **powerpc-eabi-objcopy** – за PowerPC.

Овај префикс је интегрални део назива сваког алата, што гарантује да је компајлиран са подршком за специфичну *target* архитектуру. Уколико би програмер користио „обичан“ **objcopy** (без префикса) на хост систему, алат би очекивао објектне датотеке за x86 архитектуру и не би исправно функционисао при обради ARM ELF датотека. Због тога је у *embedded* развоју пресудно увек користити ARM-специфичне варијанте са префиксом **arm-none-eabi-**.

Практичне импликације

У анализи која следи (поглавља 6.1–6.6 овог рада) користе се искључиво GNU *binutils* алати са ARM префиксом, инсталирани као део **arm-none-eabi toolchain**-а. То је посебно важно јер:

1. **Исправност резултата** – само алати компајлирани за ARM ELF формате могу коректно тумачити, дисасемблирати и конвертовати датотеке настале GCC ARM превођењем.
2. **Униформност развоја** – целокупан компилациони ланац (gcc, ld, as, objcopy, objdump итд.) мора бити усаглашен и доследно коришћен у истој ARM-none-eabi варијанти.
3. **Преносивост и транспарентност** – GNU алати имају идентичну синтаксу и начин рада на свим платформама, па се стекнута знања могу применити на различитим архитектурама уз минималне измене.

6.1. arm-none-eabi-objcopy: креирање HEX, BIN и S19 датотека

Алат **objcopy** представља део GNU *binutils* пакета и служи за копирање, модификацију и конверзију формата објектних датотека. У embedded развоју његова основна намена је да из ELF датотеке, која настаје као резултат компилације и линковања, генерише излазне формате погодне за програмирање микроконтролера, као што су **Intel HEX (.hex)**, **RAW бинарни (.bin)** или **Motorola S-Record (.s19)**.

Поред конверзије формата, **objcopy** омогућава и низ додатних операција над бинарним датотекама: уклањање табела симбола, издвајање или спајање појединих секција, премештање секција у друге меморијске области, или додавање пуњења (*padding*) како би се датотека прилагодила захтевима меморијске мапе.

6.1.1. ARM-специфична варијанта алата

Кључна разлика постоји између „генеричког“ **objcopy** и варијанте **arm-none-eabi-objcopy**:

1. arm-none-eabi-objcopy

- представља ARM-специфичну верзију алата која се дистрибуира у оквиру *arm-none-eabi GCC toolchain-a*;
- подразумевано подржава ARM ELF формате (*elf32-littlearm, bare-metal, EABI*);
- може директно и без додатних параметара обрадити ELF датотеку генерисану компајлером *arm-none-eabi-gcc*.
- пример употребе:
- `arm-none-eabi-objcopy -O binary program.elf program.bin`

2. Генерички objcopy (нпр. из MSYS2 MinGW окружења)

- компајлиран је за `x86_64-pc-mingw32` и није подразумевано свестан ARM ELF формата;
- приликом покушаја конверзије може пријавити грешку типа *file format not recognized* или произвести неисправан излаз;
- да би исправно функционисао, морају се експлицитно навести улазни и излазни формати, на пример:
- `objcopy -I elf32-littlearm -O binary program.elf program.bin`

С обзиром на то да овај рад обрађује ARM Cortex-M архитектуру, у пракси је неопходно користити ARM-специфичну варијанту **arm-none-eabi-objcopy**.

6.1.2. Основне могућности алата

Неке од најважнијих функција **arm-none-eabi-objcopy** алата могу се систематизовати на следећи начин:

1. **Конверзија ELF у Intel HEX формат:**

```
arm-none-eabi-objcopy -O ihex program.elf program.hex
```

2. **Конверзија ELF у RAW бинарни формат:**

```
arm-none-eabi-objcopy -O binary program.elf program.bin
```

3. **Конверзија ELF у Motorola S-Record (S19):**

```
arm-none-eabi-objcopy -O srec program.elf program.s19
```

4. **Издавање појединачне секције (нпр. .text):**

```
arm-none-eabi-objcopy -j .text -O binary program.elf text.bin
```

5. **Искључивање табеле симбола:**

```
arm-none-eabi-objcopy --strip-symbols program.elf program_stripped.elf
```

6. **Уклањање свих симбола ради минимизације величине:**

```
arm-none-eabi-objcopy --strip-symbols=symbols.txt program.elf program_stripped.elf
```

7. **Преименовање или спајање секција:**

```
arm-none-eabi-objcopy --rename-section .data=.mydata program.elf program_mod.elf
```

8. **Додавање пуњења празнина (padding):**

```
arm-none-eabi-objcopy --pad-to 0x20000 --gap-fill 0xFF program.elf padded.bin
```

9. **Конверзија између различитих ELF формата (нпр. 32/64-бит, endian):**

```
arm-none-eabi-objcopy -O elf32-littlearm program.elf program32.elf
```

6.1.3. Практичне напомене

При конверзији у RAW бинарни формат (.bin) неопходно је бити опрезан. Уколико ELF датотека садржи „празнине“ у меморијској мапи (нпр. секције које нису континуалне), **objcopy** ће приликом генерисања .bin датотеке ове празнине попунити нулама, чиме добијени бинарни фајл може бити знатно већи од стварне величине корисног кода. Ово понашање проистиче из чињенице да .bin представља континуални дамп меморије од најниже до највише адресе у ELF-у.

6.2. arm-none-eabi-readelf: испитивање интерне структуре ELF фајлова

Алат **readelf** је специјализовани део GNU *binutils* пакета намењен анализи ELF датотека (*Executable and Linkable Format*). За разлику од алата као што је **objdump**, који се углавном користи за дисасемблирање и приказ машинских инструкција, **readelf** пружа структурисан и свеобухватан увид у унутрашњу организацију ELF формата. Он омогућава приказ ELF заглавља, програмских и секцијских табела, симбола, релокација, динамичких информација и дебаг података. Управо због тога, **readelf** је кључан у *embedded* развоју, јер омогућава да се теоријски меморијски распоред из линкерске скрипте упоређи са стварним излазом компајлера и линкера.

6.2.1. Преглед ELF заглавља и секција

Основна сврха алата је да прикаже садржај ELF заглавља и секција. На пример, команда:

```
arm-none-eabi-readelf -h program.elf
```

приказује опште ELF заглавље, у коме су наведени тип датотеке (нпр. EXEC – извршна), архитектура (ARM), ендијаност (*little-endian*), улазна тачка програма, као и офсети на којима почињу програмске и секцијске табеле. На овај начин програмер добија увид у најосновније карактеристике бинарне датотеке и може да потврди да ли је генерисана у складу са очекиваним архитектурским параметрима.

Са друге стране, опција **-S** омогућава приказ секцијске табеле, односно списка свих секција у ELF-у. Ту се могу видети имена (*.text*, *.data*, *.bss*, *.rodata*, итд.), њихове величине, виртуелне адресе у меморији, као и позиције унутар самог фајла. Ови подаци су од посебног значаја у *embedded* контексту, јер директно показују како је код распоређен у Flash меморији и где се налазе иницијализовани и неиницијализовани подаци у RAM-у.

6.2.2. Симболи и програмски сегменти

Једна од најчешћих примена алата је приказ табеле симбола. Команда:

```
arm-none-eabi-readelf -s program.elf
```

изводи листу свих симбола у програму – од функција и глобалних променљивих до интерних системских симбола које користи компајлер и линкер. На тај начин се, на пример, лако може пронаћи адреса функције *main*, што је честа провера исправности изграђене апликације.

Поред секција и симбола, **readelf** омогућава и приказ програмских заглавља (**-l** опција), у којима су описани сегменти који се заиста читавају у меморију. Ово је суштинска разлика између секција и сегмената: док секције представљају логичке целине унутар

ELF-а (нпр. код, подаци, константе), сегменти дефинишу шта ће оперативни систем или, у случају *bare-metal* система, стартап код микроконтролера заиста пребацити у RAM или Flash током извршавања.

6.2.3. Релокације, динамичке информације и дебаг подаци

Иако се у *embedded* систему најчешће користи статичко линковање, **readelf** омогућава и преглед релокационих записа (-r) и динамичких информација (-d). Ови подаци нису увек присутни у *bare-metal* окружењима, али су од великог значаја у системима који користе динамичке библиотеке. Поред тога, **readelf** може да прикаже садржај дебаг секција у DWARF формату (-w опције), што је важно за дубинско отклањање грешака и праћење симболичких информација током дебаговања.

6.2.4. Примена у анализи меморијског распореда

У контексту анализе линкерске скрипте, најкориснија функција је управо приказ секција (-S), јер она омогућава директно поређење дефинисаног распореда у скрипти и стварно изграђене бинарне слике. На пример, излаз команде може показати да је:

- .text секција величине 0x1000 бајтова смештена у Flash на адреси 0x10000000,
- .data секција величине 0x100 бајтова распоређена у RAM-у на адреси 0x08040000, али са иницијалним вредностима које се чувају у Flash-у (LMA),
- .bss секција обухвата 0x200 бајтова у RAM-у и иницијализује се на нулу током стартапа.

Ови подаци се морају у потпуности поклопити са дефиницијом у линкерској скрипти, јер било какво одступање може довести до неочекиваног понашања микроконтролера.

6.2.5. Практични примери

Примери употребе који илуструју најчешће примене у *embedded* развоју су:

- провера архитектуре и улазне тачке:
`arm-none-eabi-readelf -h program.elf`
- анализа секција ради провере усаглашености са линкер скриптом:
`arm-none-eabi-readelf -S program.elf`
- тражење адресе функције `main`:
`arm-none-eabi-readelf -s program.elf | grep main`

- приказ сегмената за читавање у RAM/Flash:
`arm-none-eabi-readelf -l program.elf`
- приказ садржаја константних података у секцији `.rodata`:
`arm-none-eabi-readelf -x .rodata program.elf`

6.3. arm-none-eabi-nm: преглед симбола у бинарним датотекама

Алат **nm** из GNU *binutils* пакета служи за анализу симбола унутар објектних и извршних датотека. У контексту *embedded* развоја, овај алат има посебан значај јер омогућава програмеру да стекне детаљан увид у распоред функција, глобалних променљивих и других симболичких ентитета које компајлер и линкер генеришу током изградње пројекта.

У свакодневној пракси **arm-none-eabi-nm** се користи како би се проверило:

- где је функција или променљива смештена у меморији,
- да ли је неки симбол правилно разрешен након линковања,
- да ли је стек (*stack*) дефинисан на очекиваној адреси,
- и да ли постоје недефинисани или преклопљени симболи који могу изазвати проблеме у извршавању.

6.3.1. Приказ симбола и њихова класификација

Када се покрене у основној форми, команда:

```
arm-none-eabi-nm program.elf
```

исписује листу симбола у три колоне: адресу симбола, тип симбола (ознаку једним словом) и његово име. Ознака типа омогућава да се одмах уочи природа симбола:

- **T** означава функцију у сегменту извршног кода (*.text*),
- **D** иницијализовани податак у секцији *.data*,
- **B** неиницијализовани податак у секцији *.bss*,
- **R** податак у секцији само за читање (*.rodata*),
- **U** недефинисани симбол (који није разрешен приликом линковања),
- **W** слаби симбол (*weak*), који може бити замењен јачом дефиницијом.

Ова класификација је од велике помоћи при верификацији исправности линкерске скрипте: ако је, на пример, функција означена словом **T**, њена адреса мора бити у Flash опсегу; ако је променљива у **.bss** или **.data**, адреса мора припадати RAM региону.

6.3.2. Практична вредност у анализи меморијског распореда

У системима без оперативног система програмер има директну контролу над целокупним меморијским простором, па алат **nm** служи као поуздана метода за проверу да ли се распоред симбола заиста поклапа са оним што је дефинисано у линкерској скрипти.

На пример:

- глобалне променљиве које треба да буду у RAM-у морају се појавити са ознаком **B** или **D**, уз адресу унутар RAM опсега (нпр. 0x08020000 – 0x0807FFFF);
- функције морају бити у Flash-у, означене словом **T** и смештене у опсег од 0x10000000 па навише;
- симбол **_estack**, који представља врх стека, мора се појавити са тачном адресом краја RAM-а, што је кључна провера исправности linker.ld датотеке.

Ако се након финалног линковања појаве **U** (undefined) симболи, то указује на грешке у систему – или недостајуће библиотеке, или функције које нису реализоване.

6.3.3. Напредне опције за дубљу анализу

Иако је основни излаз често довољан, **nm** нуди и неколико напредних могућности:

- симболи се подразумевано сортирају по имену, али се опцијом **-n** могу приказати по адреси, што омогућава увид у редослед смештаја функција и података у меморији;
- коришћењем опције **-S** уз сваки симбол се приказује и његова величина, што олакшава анализу меморијске потрошње појединачних функција или променљивих;
- уз опцију **--defined-only** могуће је излистати искључиво дефинисане симболе, чиме се елиминише „шум“ који уносе недефинисани симболи.

Ови механизми омогућавају прецизнију контролу и ефикаснију проверу интегритета пројекта.

6.3.4. Примери употребе у embedded контексту

Да би се илустровала практична примена, могу се навести следеће ситуације:

- **Провера адресе функције main:**
`arm-none-eabi-nm program.elf | grep main`

- **Анализа меморијског распореда:**

```
arm-none-eabi-nm -n program.elf
```

Овим се добија списак свих симбола у редоследу по меморијским адресама, што је корисно за проверу да ли је распоред у Flash-у и RAM-у доследан.

- **Верификација стека:**

```
arm-none-eabi-nm program.elf | grep _estack
```

Овим се осигурава да је врх стека дефинисан у складу са MEMORY регијом у linker.ld.

- **Претрага недефинисаних симбола:**

```
arm-none-eabi-nm program.elf | grep " U "
```

Ако се било који такав симбол појави, неопходно је анализирати да ли недостаје библиотека или имплементација функције.

6.4. arm-none-eabi-size: анализа меморијске потрошње по секцијама

Алат **arm-none-eabi-size** представља једно од најједноставнијих, али и најкориснијих средстава у GNU *binutils* пакету када је потребно брзо проценити меморијску потрошњу програма. Његова основна сврха није дубинска анализа ELF структуре, већ пружање јасног резимеа о томе колико простора поједине кључне секције заузимају у меморији микроконтролера. Управо због тога он је често први алат који се користи након успешног линковања – као тренутна потврда да програм заиста „стаје“ у расположиви Flash и RAM.

6.4.1. Основни принцип рада

Позивом:

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

корисник добија табеларни приказ величина секција у ELF датотеци. Уобичајени излаз садржи шест колона:

- **text** – простор који заузима извршни код заједно са константним подацима (.text + .rodata), смештеним у Flash меморији,
- **data** – иницијализовани подаци (.data) који ће бити смештени у RAM, али који истовремено заузимају и место у Flash-у, јер тамо морају бити сачуване њихове почетне вредности,
- **bss** – неиницијализовани подаци (.bss), који ће приликом стартапа бити алоцирани у RAM-у и постављени на нулу,
- **dec** – укупан збир величина (у децималном формату),
- **hex** – исти тај збир, али у хексадекадном облику,
- **filename** – име ELF датотеке над којом је анализа извршена.

Пример излаза:

text	data	bss	dec	hex	filename
429444	688	4112	434244	6a044	program.elf

Из овог податка се одмах може закључити да код и константни подаци заузимају око **429 kB Flash-а**, иницијализовани подаци **688 бајтова RAM-а** (уз исту количину у Flash-у за њихово иницијално стање), док неиницијализовани подаци захтевају **4112 бајтова**

RAM-а. Ово представља збирну слику меморијског „отиска“ програма и основу за процену да ли он може бити успешно смештен у циљни микроконтролер.

6.4.2. Аналитички значај података

Резултат који даје **size** није само бројчани извештај, већ и средство за проверу исправности целокупног система. Уколико, на пример, вредност у колони **bss** знатно превазилази расположиви RAM, програм неће бити извршив и то се мора отклонити редукцијом статичких структура или оптимизацијом кода. Са друге стране, величина секције **data** одмах указује на то колико RAM-а ће бити заузето одмах по старту, а истовремено показује колико Flash меморије мора бити резервисано за иницијалне вредности.

Оваква анализа је од посебне важности код система као што је Infineon CYT2BL5CAS, који располаже са **4 MB Flash** и **320 KB RAM-а**. Поређењем излаза алата са овим граничним вредностима може се проценити:

- колико простора остаје за **heap** и **stack**,
 - да ли програм заузима превише меморије у односу на доступне ресурсе,
 - да ли је потребна оптимизација кода или корекција линкерске скрипте.
-

6.4.3. Напредне могућности

Иако је најчешће довољан основни приказ, **arm-none-eabi-size** нуди и додатне опције које омогућавају детаљнију анализу:

- опција **-A** или **--format=SysV** приказује величине свих секција појединачно (нпр. **.vectors**, **.rodata**, **.heap**, кориснички дефинисане секције), чиме се добија прецизнија слика о томе који делови кода или података заузимају највише меморије,
 - опција **--common** укључује и *common* симболе – глобалне променљиве које нису иницијализоване, али нису ни смештене у класичну **.bss** секцију,
 - опција **--totals** даје збирне вредности када се истовремено анализира више ELF датотека, што је корисно у пројектима са више извршних модула.
-

6.4.4. Практична примена у embedded развоју

У пракси, алат **arm-none-eabi-size** често се користи у последњем кораку процеса изградње као део аутоматизованог извештаја (нпр. у *Makefile*-у или у CI/CD окружењу). На тај начин се програмеру одмах сигнализира ако је нека од секција прешла расположиви капацитет меморије.

У комбинацији са анализом коју дају алати као што су **readelf** или **nm**, добија се потпуна слика:

- **readelf** пружа распоред секција и њихове адресе,
- **nm** омогућава увиде у појединачне симболе и функције,
- **size** резимира укупан меморијски утицај.

Ова три алата заједно представљају основно средство сваког embedded инжењера за проверу да ли изграђени *firmware* задовољава ограничења циљног хардвера.

6.5. arm-none-eabi-objdump: дубинска анализа ELF структуре

Алат **objdump** представља најмоћнији и најсвестранији инструмент GNU *binutils* пакета за анализу бинарних датотека. Његова специфичност огледа се у томе што омогућава увид у све слојеве ELF извршног фајла: од метаподатака и секција, преко симболичких таблица, па све до дисасемблирања машинског кода у људски читљив асемблер. За разлику од алата као што су **readelf** или **nm**, који пружају структурне или симболичке информације, **objdump** комбинује оба приступа и омогућава непосредан увид у то како је компајлер превео изворни С код у конкретне ARM инструкције. Управо због тога овај алат је незаменљив у фазама дебаговања и оптимизације кода на ниском нивоу.

6.5.1. Дисасемблирање машинског кода

Најзначајнија функционалност алата **objdump** јесте дисасемблирање, односно превођење бинарних инструкција у асемблерски листинг. Наредба:

```
arm-none-eabi-objdump -d -M reg-names-std program.elf
```

изводи детаљан листинг свих секција које садрже код (нпр. *.text*, *.init*). Опција *-M reg-names-std* осигурава да се регистри приказују стандардним Cortex-M именима (*r0*, *r1*, *sp*, *lr*), што повећава читљивост.

Овакав излаз има двоструку вредност: прво, програмер добија могућност да упореди изворни С код са генерисаним инструкцијама и процени ефикасност компајлера; друго, омогућава се тражење потенцијалних аномалија или неефикасних секвенци које могу утицати на перформансе или потрошњу енергије. На пример, уколико GCC није препознао могућност оптимизације петље, програмер може уочити сувишне инструкције и одлучити се за другачију организацију кода.

6.5.2. Хексадецимални приказ садржаја

Поред дисасемблирања, **objdump** омогућава и директан увид у сирове бинарне податке ELF секција. Командом:

```
arm-none-eabi-objdump -s program.elf
```

приказује се садржај свих секција у хексадецималном и ASCII формату. Ова функција је корисна када је потребно испитати структуре података у меморији, као што су векторска табела прекида, *lookup* таблице или иницијализовани низови. За разлику од дисасемблирања, овде се добија „чиста“ репрезентација бајтова, што је неопходно у случајевима када се проверавају тачне вредности података уписаних у меморију.

6.5.3. Табела симбола и метаподаци ELF датотеке

Алат такође може да прикаже табелу симбола, коришћењем опције `-t`. Овај излаз је функционално сличан резултату алата **nm**, али је интегрисан са осталим информацијама које пружа **objdump**. Симболи се приказују сортирани по секцијама, уз своје атрибуте и адресе, што олакшава идентификацију функција и променљивих унутар конкретних меморијских региона.

За свеобухватни преглед користи се опција `-x`, која изводи комплетан приказ ELF заглавља, програмских сегмената, секција и симбола. Иако је овај излаз веома обиман и мање прегледан од специјализованих алата као што су **readelf** или **nm**, његова предност је у томе што на једном месту пружа целокупну слику о извршном фајлу. То је нарочито значајно када је потребно анализирати усаглашеност линкерске скрипте и стварно изграђене бинарне слике.

6.5.4. Напредне могућности

Objdump поседује и низ додатних функција које га чине погодним за сложене анализе:

- опција `-d -l` омогућава дисасемблирање уз приказ изворних C линија кода (ако ELF садржи debug информације),
- `-D` дисасемблира целокупан фајл, укључујући делове који нису у стандардним секцијама кода,
- `--start-address` и `--stop-address` ограничавају приказ на задати адресни опсег,
- опција `-C` врши деманглирање C++ симбола, чинећи их читљивим у изворном облику,
- опција `-g` омогућава преглед одељака са debug информацијама, што је значајно за отклањање грешака.

6.5.5. Практична примена у embedded контексту

У развоју за ARM Cortex-M микроконтролере, алат **arm-none-eabi-objdump** има незаменљиву улогу у неколико критичних сегмената:

- **верификација меморијског распореда** – програмер може проверити да ли су поједине функције или табеле смештене у тачно предвиђене регионе (Flash или RAM),
- **анализа ефикасности кода** – увиди у генерисане инструкције омогућавају процену да ли компајлер користи оптималне инструкцијске секвенце,
- **дијагностика на ниском нивоу** – при грешкама као што су неисправно иницијализован стек, погрешне адресе векторске табеле или извршавање

неочекиваних инструкција, дисасемблирање представља најпоузданији начин за разумевање стварног понашања програма,

- **идентификација „тешких“ функција** – анализом величине и дужине инструкцијских секвенци могу се уочити функције које заузимају непропорционално много меморије или циклуса извршавања.

6.6. Комплементарна употреба GNU алата у embedded развоју

Наведени GNU алати представљају незаобилазан део практичног развојног циклуса у embedded окружењима. Иако се уобичајено посматрају као пратећи инструменти компајлера и линкера, њихова права вредност огледа се у могућности да програмеру омогуће **директан увид у интерну структуру резултујућих датотека**, као и у верификацију да је програм правилно смештен у меморију микроконтролера.

Алати се међусобно допуњују:

- **arm-none-eabi-objcopy** служи за трансформацију ELF датотеке у крајње формате употребљиве за програмирање (Intel HEX, бинарни .bin, Motorola S-Record). На тај начин се обезбеђује компатибилност са програматорима и bootloader механизмима.
- **arm-none-eabi-readelf** омогућава да се испита тачан распоред секција и сегмената у ELF-у, чиме се потврђује да је линкерска скрипта коректно одредила позиције критичних делова програма (нпр. .text у флешу, .data у RAM-у).
- **arm-none-eabi-nm** пружа преглед и класификацију симбола, што је од кључног значаја за дијагностику проблема са глобалним променљивима, стеком или неповезаним функцијама.
- **arm-none-eabi-size** резимира укупно заузеће меморије по секцијама, дајући програмеру јасну слику о томе да ли програм стаје у расположиви Flash и RAM, и колико простора остаје за динамичке структуре података.
- **arm-none-eabi-objdump** омогућава дисасемблирање и дубинску анализу машинских инструкција, што је од непроцењиве важности при дебаговању и оптимизацији кода на нивоу асемблера.

У пракси, комплементарна употреба ових алата може изгледати овако:

1. Након успешног линковања, програмер прво покреће **arm-none-eabi-size program.elf** да би брзо проценио да ли програм одговара расположивим ресурсима микроконтролера.
2. Уколико постоји сумња у исправност секција, нарочито .data и .bss, користи се **arm-none-eabi-readelf -S program.elf** ради детаљног увида у адресе и величине.
3. У случају дебаговања глобалних променљивих или провере симбола дефинисаних у линкер скрипти (попут _estack), примењује се **arm-none-eabi-nm program.elf**.
4. За анализу извршног кода и проверу генерисаних инструкција, посебно код функција критичних по перформансе, користи се **arm-none-eabi-objdump -d -M reg-names-std program.elf**.
5. На крају, приликом припреме финалне датотеке за програмирање у меморију микроконтролера, ELF се уз помоћ **arm-none-eabi-objcopy** конвертује у .hex или .bin формат, у складу са потребама програматора или bootloader-a.

Ови кораци показују да су GNU binutils алати не само помоћна средства већ и **неопходан продужетак компајлера и линкера** у embedded развоју. Они програмеру пружају увид „испод хаубе“, односно у све фазе након превођења С кода, чиме се омогућава потпуна контрола над меморијским распоредом и извршним кодом. На тај начин, њихова употреба није само практична, већ и суштинска за обезбеђивање поузданости и предвидљивости система у реалним условима рада.

7. Линкерска скрипта

Линкерска скрипта или линкер директива (обично .ld фајл, нпр. linker.ld) представља суштински део embedded пројекта – она одређује начин распоређивања секција програма у физичку меморију микроконтролера током процеса линковања. Ова скрипта повезује свет апстрактних секција C кода са конкретним Flash/RAM адресним простором хардвера, обезбеђујући да сваки део извршног кода и података буде смештен на предвиђену локацију у меморији. У систему *bare-metal* (без оперативног система), исправно мапирање свих секција на фиксне меморијске адресе је критично, јер не постоји динамички менаџмент меморије у току извршавања.

У наставку разматрамо пример линкерске скрипте за конкретан микроконтролер (Infineon Traveo II, породице CYT2BL5CAS – двојезгарни Cortex-M систем). Та скрипта је класичног обрасца за вишеструка језгра: састоји се из три дела – (1) почетне глобалне директиве и дефиниције симбола, (2) секција MEMORY са описом расположивих меморијских регија (нпр. Flash, RAM, специјални сегменти) и (3) секција SECTIONS која прописује смештај сваке програмске секције (.text, .data, .bss, стек, хип, итд.) у одговарајуће меморијске регије. У наредним потпоглављима систематски се анализира сваки од ових делова линкерске скрипте.

7.1. Пример линкер скрипте за GNU C компајлер

Предметна линкерска скрипта започиње глобалним поставкама излазног формата и библиотека, затим дефинише кључне параметре (нпр. величину стека) као симболичке константе, а потом описује расположиве меморијске регије циљног система. На основу тих подешавања, у блоку `SECTIONS` врши се расподела преведених програмских секција у одговарајуће меморијске регионе.

Конкретно, скрипта прво одређује да ће резултујући бинарни извршни фајл бити ELF за 32-битни ARM у *little-endian* формату (директива `OUTPUT_FORMAT("elf32-littlearm", ...)`), а као улазну тачку програма поставља адресу симбола *Reset_Handler* (директива `ENTRY(Reset_Handler)`). Затим се задају константне вредности за важне величине меморијских резерви, попут величине стека (`STACK_SIZE`) и резервисаног простора за друго језгро у flash меморији. На основу ових параметара, могу се израчунати изведене адресе и границе меморијских региона – у посматраном примеру, реч је о двојезгарном микроконтролеру па се рачуна основна адреса и величина SRAM меморије за помоћно Cortex-M0+ језгро, као и одговарајуће резервације у Flash-у за Cortex-M4 језгро (пратећи да величина резервисаног Flash блока буде поравната на границу сектора). Скрипта укључује и `ASSERT` наредбу којом се осигурава да резервисани део Flash-а за M0+ језгро завршава на граници сектора одређене величине (нпр. `0x8_0000` бајтова = 512 KB), како би се спречило да попуњавање M0+ програма ненамерно обрише почетак M4 апликације. Након тога, директива `EXTERN(Reset_Handler)` у линкер скрипти осигурава да симбол *Reset_Handler* буде експлицитно уведен у излазну датотеку као недефинисан, чиме се линкер приморава да из библиотеке или `startup` модула увуче одговарајућу дефиницију. На тај начин се гарантовано укључује почетни код ресет рутине неопходан за покретање програма.

Следећи део скрипте је блок `MEMORY`, у коме се именују главни меморијски региони система. У овом примеру дефинисана су два кључна региона: интерна Flash меморија за програмски код и интерна SRAM за податке и стек. Flash регион почиње од адресе `0x1000_0000` и дуг је `0x41_0000` бајтова, док RAM регион почиње од `0x0802_0000` са дужином `0x5_F800` бајтова. Ове адресе и величине одговарају конкретном микроконтролеру – у овом случају, део меморије је резервисан за M0+ језгро (почетних 128 KB SRAM-а и 512 KB Flash-а за програм помоћног језгра), а главно M4 језгро користи преостали део меморије. Поред тога, у `MEMORY` секцији су описани и специјални меморијски сегменти карактеристични за овај чип: неколико области *Supervisory Flash* меморије (нпр. за корисничке податке, криптографске кључеве, табеле садржаја) на фиксним адресама `0x1700_0800`, `0x1700_1A00` итд., као и *eFuse* меморија на адреси `0x9070_0000`. Ови посебни региони су резервисани за системску употребу и не треба их мењати. Служе за смештај одређених секција ако се оне појаве у програму (нпр. секција за корисничке податке у *Supervisory Flash*-у).

Коначно, у блоку `SECTIONS` прецизира се распоред излазних ELF секција по меморијским регијама. Векторска табела прекида и извршни код програма смештају се у Flash, иницијализовани подаци (`.data`) у RAM (али са копијом иницијалних вредности у Flash-у), неиницијализовани подаци (`.bss`) такође у RAM (без заузимања места у Flash-у), док су секције за стек и хип (*stack* и *heap*) позициониране на крају RAM меморије. Овим распоредом се остварује статичко меморијско мапирање целокупног програма, што је основа за правилно покретање и рад микроконтролера у *bare-metal* окружењу. У наставку детаљно разматрамо сваку од наведених целина скрипте.

```

OUTPUT_FORMAT ("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
SEARCH_DIR(.)
GROUP(-lgcc -lc -lnosys)
ENTRY(Reset_Handler)

/* The size of the stack section at the end of CM0+ SRAM */
STACK_SIZE = 0x1000;

/* Additions MLGH to incorporate the SROM Sram requirement */
sram_start_reserve          = 0;

/* Private SRAM for SROM (e.g. API processing). Reserved at the beginning */
sram_private_for_srom       = 0x00000800;

cm0plus_sram_reserve        = 0x00020000; /* cm0 sram size */
cm0plus_code_flash_reserve  = 0x00080000; /* cm0 flash size */

sram_base_address          = 0x08000000;
code_flash_base_address    = 0x10000000;
code_flash_total_size      = 0x00080000;

_base_SRAM_CM0P = sram_base_address + sram_start_reserve + sram_private_for_srom;
_size_SRAM_CM0P = cm0plus_sram_reserve - sram_start_reserve - sram_private_for_srom;

/* CM0+ flash reservation must end on a sector boundary in order to avoid partial
erasure of CM4 application. */
code_flash_sector_size     = 0x8000;

/* Enforce CM0+ flash size ends on a boundary.
 * Comment this assert out if you need to prioritize CM4
 * application space, but note that if the sector boundary
 * does not match the CM0+ flash size, the CM4
 * application must always be flashed again after the CM0+ application,
 * since flashing the CM0+ program
 * will erase the start of the CM4 program that is placed inside the last
 * CM0+ application sector.
 */
ASSERT(cm0plus_code_flash_reserve % code_flash_sector_size == 0, "CM0 code space
does not end on a sector boundary, which will cause the start of the CM4
application space to be erased when modifying CM0 application. Fix CM0
application size.")

/* Force symbol to be entered in the output file as an undefined symbol. Doing
 * this may, for example, trigger linking of additional modules from standard
 * libraries. You may list several symbols for each EXTERN, and you may use
 * EXTERN multiple times. This command has the same effect as the -u command-line
 * option.
 */
EXTERN(Reset_Handler)

```

```

/* The MEMORY section below describes the location and size of blocks
 * of memory in the target.
 * Use this section to specify the memory regions available for allocation.
 */
MEMORY
{
    /* The ram and flash regions control RAM and flash
     * memory allocation for the CM0+ core.
     * You can change the memory allocation
     * by editing the 'ram' and 'flash' regions.
     * Note that 2 KB at the end of the system SRAM are reserved for system use.
     * Using this memory region for other purposes will lead to unexpected
     * behavior (however, this is usually only a concern for the CM4 processor.)
     * Your changes must be aligned with the corresponding memory regions
     * for the CM4 core in 'xx_cm4_dual.ld',
     * where 'xx' is the device group; for example, 'cyb06xx7_cm4_dual.ld'.
     */

    ram    (rxw) : ORIGIN = _base_SRAM_CM0P, LENGTH = _size_SRAM_CM0P
    flash (rx)  : ORIGIN = 0x10000000,      LENGTH = 0x80000

    /* The following regions define device specific
     * memory regions and must not be changed. */

    /* Supervisory flash: User data */
    sflash_user_data (rx)      : ORIGIN = 0x17000800, LENGTH = 0x800

    /* Supervisory flash: Normal Access Restrictions (NAR) */
    sflash_nar        (rx)      : ORIGIN = 0x17001A00, LENGTH = 0x200

    /* Supervisory flash: Public Key */
    sflash_public_key (rx)      : ORIGIN = 0x17005A00, LENGTH = 0xC00

    /* Supervisory flash: Table of Content # 2 */
    sflash_toc_2      (rx)      : ORIGIN = 0x17007C00, LENGTH = 0x200

    /* Supervisory flash: Table of Content # 2 Copy */
    sflash_rtoc_2     (rx)      : ORIGIN = 0x17007E00, LENGTH = 0x200

    efuse              (r)       : ORIGIN = 0x90700000, LENGTH = 0x100000 /* 1 MB */
}

/* Library configurations */
GROUP(libgcc.a libc.a libm.a libnosys.a)

/* Linker script to place sections and symbol values. Should be used together
 * with other linker script that defines memory regions FLASH and RAM.
 * It references following symbols, which must be defined in code:
 *   Reset_Handler : Entry of reset handler
 *
 * It defines following symbols, which code can use without definition:
 *   __exidx_start
 *   __exidx_end
 *   __copy_table_start__
 *   __copy_table_end__
 *   __zero_table_start__
 *   __zero_table_end__
 *   __etext
 *   __data_start__

```

```
* __preinit_array_start
* __preinit_array_end
* __init_array_start
* __init_array_end
* __fini_array_start
* __fini_array_end
* __data_end__
* __bss_start__
* __bss_end__
* __end__
* end
* __HeapLimit
* __StackLimit
* __StackTop
* __stack
* __Vectors_End
* __Vectors_Size
*/
```

SECTIONS

```
{
    .cy_app_header :
    {
        KEEP(*(.cy_app_header))
    } > flash

    /* Cortex-M0+ application flash area */
    .text ORIGIN(flash) :
    {
        . = ALIGN(4);
        __Vectors = . ;
        KEEP(*(.vectors))
        . = ALIGN(4);
        __Vectors_End = .;
        __Vectors_Size = __Vectors_End - __Vectors;
        __end__ = .;

        . = ALIGN(4);
        *(.text*)

        KEEP(*(.init))
        KEEP(*(.fini))

        /* .ctors */
        *crtbegin.o(.ctors)
        *crtbegin?.o(.ctors)
        *(EXCLUDE_FILE(*crtend?.o *crtend.o) .ctors)
        *(SORT(.ctors.*))
        *(.ctors)

        /* .dtors */
        *crtbegin.o(.dtors)
        *crtbegin?.o(.dtors)
        *(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors)
        *(SORT(.dtors.*))
        *(.dtors)

        /* Read-only code (constants). */
```

```

        *(.rodata .rodata.* .constdata .constdata.* .conststring .conststring.*)

        KEEP(*(.eh_frame*))
    } > flash

    .ARM.extab :
    {
        *(.ARM.extab* .gnu.linkonce.armextab.*)
    } > flash

    __exidx_start = .;

    .ARM.exidx :
    {
        *(.ARM.exidx* .gnu.linkonce.armexidx.*)
    } > flash
    __exidx_end = .;

/* To copy multiple ROM to RAM sections,
 * uncomment .copy.table section and,
 * define __STARTUP_COPY_MULTIPLE in startup_tviibe4m_cm0plus.S */
.copy.table :
{
    . = ALIGN(4);
    __copy_table_start__ = .;

    /* Copy interrupt vectors from flash to RAM */
    LONG (__Vectors)                /* From */
    LONG (__ram_vectors_start__)    /* To */
    LONG (__Vectors_End - __Vectors) /* Size */

    /* Copy data section to RAM */
    LONG (__etext)                  /* From */
    LONG (__data_start__)           /* To */
    LONG (__data_end__ - __data_start__) /* Size */

    __copy_table_end__ = .;
} > flash

/* To clear multiple BSS sections,
 * uncomment .zero.table section and,
 * define __STARTUP_CLEAR_BSS_MULTIPLE in startup_tviibe4m_cm0plus.S */
.zero.table :
{
    . = ALIGN(4);
    __zero_table_start__ = .;
    LONG (__bss_start__)
    LONG (__bss_end__ - __bss_start__)
    __zero_table_end__ = .;
} > flash

__etext = . ;

.ramVectors (NOLOAD) : ALIGN(8)
{

```

```

    __ram_vectors_start__ = .;
    KEEP(*(.ram_vectors))
    __ram_vectors_end__ = .;
} > ram

.data __ram_vectors_end__ :
{
    . = ALIGN(4);
    __data_start__ = .;

    *(vtable)
    __sdata_start__ = .;
    *(.data*)
    __sdata_end__ = .;

    . = ALIGN(4);
    /* preinit data */
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP(*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);

    . = ALIGN(4);
    /* init data */
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP(*(SORT(.init_array.*)))
    KEEP(*(.init_array))
    PROVIDE_HIDDEN (__init_array_end = .);

    . = ALIGN(4);
    /* finit data */
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP(*(SORT(.fini_array.*)))
    KEEP(*(.fini_array))
    PROVIDE_HIDDEN (__fini_array_end = .);

    KEEP(*(.jcr*))
    . = ALIGN(4);

    KEEP(*(.cy_ramfunc*))
    . = ALIGN(4);

    __data_end__ = .;
} > ram AT>flash

/* Place variables in the section that should not be initialized during the
 * device startup.
 */
.noinit (NOLOAD) : ALIGN(8)
{
    KEEP(*(.noinit))
} > ram

/* The uninitialized global or static variables are placed in this section.
 *
 * The NOLOAD attribute tells linker that .bss section does not consume

```

```

* any space in the image. The NOLOAD attribute changes the .bss type to
* NOBITS, and that makes linker to A) not allocate section in memory, and
* A) put information to clear the section with all zeros during application
* loading.
*
* Without the NOLOAD attribute, the .bss section might get PROGBITS type.
* This makes linker to A) allocate zeroed section in memory, and B) copy
* this section to RAM during application loading.
*/
.bss (NOLOAD):
{
    . = ALIGN(4);
    __bss_start__ = .;
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
    __bss_end__ = .;
} > ram

.heap (NOLOAD):
{
    __HeapBase = .;
    __end__ = .;
    end = __end__;
    KEEP(*(.heap*))
    . = ORIGIN(ram) + LENGTH(ram) - STACK_SIZE;
    __HeapLimit = .;
} > ram

/* .stack_dummy section doesn't contains any symbols. It is only
* used for linker to calculate size of stack sections, and assign
* values to stack symbols later */
.stack_dummy (NOLOAD):
{
    KEEP(*(.stack*))
} > ram

/* Set stack top to end of RAM, and stack limit move down by
* size of stack_dummy section */
__StackTop = ORIGIN(ram) + LENGTH(ram);
__StackLimit = __StackTop - SIZEOF(.stack_dummy);
PROVIDE(__stack = __StackTop);

/* Check if data + heap + stack exceeds RAM limit */
ASSERT(__StackLimit >= __HeapLimit, "region RAM overflowed with stack")

/* Supervisory Flash: User data */
.cy_sflash_user_data :
{
    KEEP(*(.cy_sflash_user_data))
} > sflash_user_data

/* Supervisory Flash: Normal Access Restrictions (NAR) */
.cy_sflash_nar :

```

```
{
    KEEP(*(.cy_sflash_nar))
} > sflash_nar

/* Supervisory Flash: Public Key */
.cy_sflash_public_key :
{
    KEEP(*(.cy_sflash_public_key))
} > sflash_public_key

/* Supervisory Flash: Table of Content # 2 */
.cy_toc_part2 :
{
    KEEP(*(.cy_toc_part2))
} > sflash_toc_2

/* Supervisory Flash: Table of Content # 2 Copy */
.cy_rtoc_part2 :
{
    KEEP(*(.cy_rtoc_part2))
} > sflash_rtoc_2
/* eFuse */
.cy_efuse :
{
    KEEP(*(.cy_efuse))
} > efuse

/* These sections are used for additional metadata (silicon revision,
 * Silicon/JTAG ID, etc.) storage.
 */
.cymeta          0x90500000 : { KEEP(*(.cymeta)) } :NONE
}

/* The following symbols used by the cymcuelftool. */
/* Flash */
__cy_memory_0_start    = 0x10000000;
__cy_memory_0_length   = 0x00410000;
__cy_memory_0_row_size = 0x200;

/* Supervisory Flash */
__cy_memory_2_start    = 0x17000000;
__cy_memory_2_length   = 0x8000;
__cy_memory_2_row_size = 0x200;

/* eFuse */
__cy_memory_4_start    = 0x90700000;
__cy_memory_4_length   = 0x100000;
__cy_memory_4_row_size = 1;

/* EOF */
```


7.2. Почетне директиве и дефиниције симбола

Први део скрипте садржи глобалне директиве и иницијализацију симболичких константи, које ће касније бити коришћене у MEMORY и SECTIONS одељцима. Ове директиве одређују формат излазне датотеке, путеве до библиотека, улазну тачку програма, као и почетне величине за стек и друге резервисане меморијске области.

```
OUTPUT_FORMAT ("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
SEARCH_DIR(.)
GROUP(-lgcc -lc -lnosys)
ENTRY(Reset_Handler)

/* The size of the stack section at the end of CM0+ SRAM */
STACK_SIZE = 0x1000;

/* Additions MLGH to incorporate the SROM Sram requirement */
sram_start_reserve = 0;

/* Private SRAM for SROM (e.g. API processing). Reserved at the beginning */
sram_private_for_srom = 0x00000800;

cm0plus_sram_reserve = 0x00020000; /* cm0 sram size */
cm0plus_code_flash_reserve = 0x00080000; /* cm0 flash size */

sram_base_address = 0x08000000;
code_flash_base_address = 0x10000000;
code_flash_total_size = 0x00080000;

_base_SRAM_CM0P = sram_base_address + sram_start_reserve + sram_private_for_srom;
_size_SRAM_CM0P = cm0plus_sram_reserve - sram_start_reserve - sram_private_for_srom;

/* CM0+ flash reservation must end on a sector boundary in order to avoid partial
erasure of CM4 application. */
code_flash_sector_size = 0x8000;

/* Enforce CM0+ flash size ends on a boundary.
* Comment this assert out if you need to prioritize CM4
* application space, but note that if the sector boundary
* does not match the CM0+ flash size, the CM4
* application must always be flashed again after the CM0+ application,
* since flashing the CM0+ program
* will erase the start of the CM4 program that is placed inside the last
* CM0+ application sector.
*/
ASSERT(cm0plus_code_flash_reserve % code_flash_sector_size == 0, "CM0 code space
does not end on a sector boundary, which will cause the start of the CM4
application space to be erased when modifying CM0 application. Fix CM0
application size.")

/* Force symbol to be entered in the output file as an undefined symbol. Doing
* this may, for example, trigger linking of additional modules from standard
* libraries. You may list several symbols for each EXTERN, and you may use
* EXTERN multiple times. This command has the same effect as the -u command-line
* option.
*/
EXTERN(Reset_Handler)
```

Код приказан изнад представља уводни део линкерске скрипте који претходи формалном опису MEMORY региона. У њему се дефинишу кључни параметри потребни за каснију расподелу секција, као што су изведене адресе, величине и додатне заштитне провере (assert), чиме се обезбеђује доследност и исправност распоређивања у SECTIONS одељку. Такав приступ одговара устаљеној пракси у GNU ld језику скрипти и заснива се на званичној спецификацији линкера.

OUTPUT_FORMAT

Директива

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
```

представља механизам GNU линкера за дефинисање формата излазног објекта на нивоу BFD (Binary File Descriptor) апстракције. Основни параметар, први у низу, дефинише примарни формат у којем ће бити генерисана резултујућа извршна датотека. У случају микроконтролера CYT2BL5CAS, заснованог на ARM Cortex-M4 језгру, тај формат је **ELF 32-битни са „little-endian“ распоредом бајтова**, што је у складу са архитектурским ограничењима самог језгра.

Други параметар у директиви, у овом случају „elf32-bigarm“, служи као алтернативни формат и користи се уколико би током процеса обраде постојала потреба за променом ендјанског распореда у „big-endian“ варијанту. Ова могућност унапређује преносивост и универзалност GNU алата, иако Cortex-M4 језгро, као што је описано у **Arm® Architecture Reference Manual** за ARMv7-M архитектуру, подржава искључиво „little-endian“ модел извршавања.

Трећи параметар, који је у овом примеру поново дефинисан као „elf32-littlearm“, историјски је повезан са такозваним *unixes* операцијама, односно процедурама снимања тренутног стања процеса у извршну датотеку. Та функционалност развијена је у оквиру GNU окружења ради специфичних алата (нпр. Emacs), али у embedded домену и у контексту микроконтролера нема практичну примену. Ипак, синтаксно присуство трећег параметра и даље је обавезно уколико се наводи проширена форма директиве.

Сумирано, у конкретном случају Cortex-M4 архитектуре ефективно је значајан искључиво први параметар, док су други и трећи задржани ради компатибилности и универзалности GNU алатки у ширем спектру архитектура и развојних окружења.

Цела еквивалентна командна линија за ову директиву, уколико се користи спољашњи позив ld линкера, изгледала би овако:

```
ld --oformat elf32-littlearm -T linker.ld -o output.elf input.o
```

где:

- --oformat elf32-littlearm одређује излазни BFD формат,
- -T linker.ld задаје коришћену линкерску скрипту,
- -o output.elf је име излазне ELF датотеке,
- input.o је улазни објектни фајл добијен компилацијом.

Детаљна синтакса и семантика директиве `OUTPUT_FORMAT`, као и објашњење њене трочлане структуре, доступни су у оквиру одељка *Format Commands* званичне GNU документације за `ld`.

SEARCH_DIR

Директива `SEARCH_DIR(".")` додаје **текући директоријум** `(.)` у интерну листу путања за претрагу библиотека и архива, што је функционално еквивалентно опцији командне линије `-L.` На тај начин, GNU линкер ће током процеса повезивања (linking) претражити и локални директоријум у потрази за библиотекама као што су `libc.a`, `libgcc.a`, или другим архивама `(*a)` специфичним за пројекат.

Ова наредба је изузетно корисна у `embedded` окружењима, где се често користе прилагођене или претходно компајлиране верзије стандардних библиотека које се налазе директно унутар пројектне структуре. Уместо да се ослања на системски глобални пут (нпр. `/usr/lib`), `SEARCH_DIR(".")` омогућава потпуну контролу над тим **које тачно библиотеке ће бити повезане**, што је кључно за предвидљивост, минимализацију и безбедност `firmware-a`.

Стандардне библиотеке попут **`libc`** (стандардна C библиотека) и **`libgcc`** (унутрашња GCC помоћна библиотека) пружају основне функције без којих већина C програма не би могла да се линкује или извршава. То укључује имплементацију функција као што су `memcpy`, `strlen`, `printf`, `malloc`, али и низ *runtime* функција неопходних за коректно управљање регистрима, стеком, и позивима функција у ARM архитектури (нпр. `__aeabi_*` функције).

Командна линија која одговара ефекту `SEARCH_DIR(".")` у линкерској скрипти изгледала би овако:

```
ld -L. -T linker.ld -o output.elf startup.o main.o -lc -lgcc
```

где:

- `-L.` додаје текући директоријум у претрагу библиотека;
- `-T linker.ld` задаје линкерску скрипту;
- `-o output.elf` је излазна ELF датотека;
- `startup.o` и `main.o` су улазни објектни модули;
- `-lc` и `-lgcc` указују да треба повезати `libc` и `libgcc`, које ће бити пронађене унутар текућег директоријума ако се тамо налазе.

Синтакса и функција директиве `SEARCH_DIR` документоване су у званичној GNU `ld` документацији, у оквиру одељка *File Commands*.

GROUP (библиотеке)

Директива `GROUP(-lgcc -lc -lnosys)` у оквиру линкерске скрипте формира **групу архива** коју линкер треба да претражује **итеративно**, све док се не разреши све међузависности између наведених библиотека. Ово је скриптни еквивалент коришћењу опција `--start-`

group ... --end-group на командној линији. Груписање је посебно важно у ситуацијама када библиотеке **међусобно позивају симболе**, као што је случај са libgcc (унутрашња GCC библиотека), libc (стандардна C библиотека) и libnosys (задата „по-OS“ имплементација системских позива). Без употребе GROUP, линкер би могао да прекине претрагу након првог пролаза и пријави нерешене симболе, док груписањем обезбеђујемо да се линковање наставља док се све међузависности успешно не разреше.

ENTRY(Reset_Handler)

Директива ENTRY(Reset_Handler) дефинише почетну извршну тачку програма у ELF фајлу. Овим се означава да ће након ресетовања процесора извршавање почети од адресе симбола Reset_Handler. У Cortex-M архитектури, Reset_Handler је рутина стартап кода (део *векторске табеле*) која се налази на другој позицији у табели прекида и позива се одмах након што процесор учита почетну вредност стек показивача из прве позиције. Постављање Reset_Handler као ENTRY симбола осигурава да ће, при генерисању извршног фајла или конверзији у бинарни формат, ова рутина бити третирана као главна тачка уласка у програм (иако сам Cortex-M контролер у старту стварно користи векторску табелу за проналажење те адресе).

Додела симболичких константи

Након наведених глобалних директива, скрипта дефинише више константи коришћењем синтаксе *симбол = вредност;*. Примери су: STACK_SIZE = 0x1000;, sram_private_for_srom = 0x00000800;, cm0plus_sram_reserve = 0x00020000;, итд. Овакве изјаве представљају *доделе симбола* у LD језику и креирају апсолутне симболе који се могу користити у изразима даље у скрипти. Линкер их евалуира *лењо*, тј. тек када су потребни. У нашем случају ови симболи служе за параметризацију меморијских адреса и величина:

- STACK_SIZE = 0x1000 дефинише величину стека од 4096 бајтова (4 KB) по језгру. Ово је меморија која ће се резервисати при врху одговарајућег RAM-а за стек програма. Уколико би била дефинисана макро константа __STACK_SIZE при превођењу, користила би се њена вредност (то је усклађено са startup кодом који условно поставља величину стека).
- sram_private_for_srom = 0x00000800 резервише 0x800 бајтова (2048 B) SRAM-а за намене *SRAM* рутина система. Traveo T2G микроконтролери имају интерни *System ROM* (SRAM) са API функцијама (нпр. за флешовање, сигурносне функције) које користе део SRAM-а као привремену “scratch” област. Овом константом се предвиђа та област на почетку SRAM-а коју неће користити апликација.
- cm0plus_sram_reserve = 0x00020000 дефинише да је 128 KB (0x20000) SRAM-а намењено Cortex-M0+ језгру у овом двојезгарном систему. Слично, cm0plus_code_flash_reserve = 0x00080000 означава да је 512 KB Flash меморије резервисано за програм Cortex-M0+ језгра. Ове резервације служе да би се одвојили ресурси између секундарног (M0+) и главног (M4) језгра, пошто оба језгра деле исту физичку меморију.
- sram_base_address = 0x08000000 и code_flash_base_address = 0x10000000 су почетне базе адреса интерне SRAM и Flash меморије на овом микроконтролеру.

Конкретно, 0x1000_0000 је базна адреса главне Flash (Code Flash) меморије у Traveo T2G чиповима, док је 0x0800_0000 база SRAM-а.

- `code_flash_total_size = 0x00080000` (512 KB) је у овом примјеру наведена тотална величина Code Flash-а. *Напомена:* За предметни модел CYT2BL5CAS, укупан Flash је стварно 4 MB, али овде је `code_flash_total_size` подешен на 512 KB, што одговара резервисаном простору за M0+ језгро. Пратећи код у скрипти ће израчунати преосталу величину за M4 језгро.

Уз помоћ ових константи изводе се даље вредности:

```
_base_SRAM_CM0P = sram_base_address + sram_start_reserve + sram_private_for_srom;
_size_SRAM_CM0P = cm0plus_sram_reserve - sram_start_reserve - sram_private_for_srom;
```

Овде је `_base_SRAM_CM0P` почетна адреса SRAM-а намењеног M0+ језгру након почетне резервације (овде `sram_start_reserve = 0`) и SROM резерве, дакле резултат је $0x0800_0000 + 0x0 + 0x800 = \mathbf{0x0800_0800}$.

То значи да првих 2048 бајтова SRAM-а заузима SROM, а од 0x0800_0800 почиње M0+ меморија. `_size_SRAM_CM0P` израчунава ефективну дужину SRAM-а за M0+: $0x20000 - 0x800 = \mathbf{0x1F800}$ (126 KB). Слично, у делу за Flash проверава се поравнање резерве:

```
code_flash_sector_size = 0x8000; /* 32 KB */
```

```
ASSERT(cm0plus_code_flash_reserve % code_flash_sector_size == 0, "CM0 code space does not end on a sector boundary...");
```

Овим се осигурава да 512 KB резервисаних за M0+ заузима цео број сектора од 32 KB (што јесте случај, 0x80000 је $16 * 32KB$).

ASSERT директива генерише грешку при линковању ако услов није испуњен – у овом примеру то је заштита да граница између M0+ и M4 Flash региона падне тачно на границу сектора, како би се избегло нежељено брисање почетка M4 програма приликом репрограмирања M0+ апликације.

```
ASSERT(cm0plus_code_flash_reserve % code_flash_sector_size == 0, "...")
```

ASSERT је скриптна провера инваријанте: ако услов није испуњен, линкер прекида са грешком и исписује поруку. Овде се формално обезбеђује да резервација Code Flash-а за CM0+ завршава на граници сектора, како би се избегло делимично брисање почетка CM4 апликације приликом репрограмирања.

EXTERN

Команда EXTERN(Reset_Handler) експлицитно декларише симбол Reset_Handler као екстерни (недефинисани) симбол који мора постојати у излазу. Практично, ово осигурава да ће објектни модул који садржи Reset_Handler (startup датотека са векторском табелом) бити увучен у линку (еквивалентно коришћењу опције -u Reset_Handler). На тај начин стартап код и векторска табела неће бити одбачени од стране линкера чак и ако на њих нема других референци у програму.

Наведени блок директива и дефиниција поставља темеље за даљи ток скрипте. Сумирано, пре MEMORY дела ми смо:

- (i) дефинисали формат излазног ELF-а и политику претраживања библиотека,
- (ii) поставили улазну тачку на Reset_Handler у складу са ARM Cortex-M моделом,
- (iii) увели параметризоване симболе за адресе и величине меморијских регија (укључујући резерве за SROM и M0+ језгро у складу са архитектуром Traveo T2G), и
- (iv) додали проверу исправности за поравнање Flash поделе. Оваква организација чини остатак скрипте читљивијом и поузданијом – касније дефинисане MEMORY и SECTIONS секције користе ове симболе да прецизно позиционирају програмске секције у оквиру расположивог адресног простора микроконтролера.

7.3. MEMORY дефиниција

Блок MEMORY у линкерској скрипти описује расположиве меморијске регионе циљног микроконтролера, њихове почетне адресе, величине и атрибуте. Ова секција служи да линкеру стави до знања које све меморије постоје и колике су, како би могао да распореди секције програма у њих и да пријави грешку уколико нека секција превазилази границе додељеног региона. У нашем примеру, MEMORY блок је дефинисан овако:

```
MEMORY
{
    /* The ram and flash regions control RAM and flash
    * memory allocation for the CM0+ core.
    * You can change the memory allocation
    * by editing the 'ram' and 'flash' regions.
    * Note that 2 KB at the end of the system SRAM are reserved for system use.
    * Using this memory region for other purposes will lead to unexpected
    * behavior (however, this is usually only a concern for the CM4 processor.)
    * Your changes must be aligned with the corresponding memory regions
    * for the CM4 core in 'xx_cm4_dual.ld',
    * where 'xx' is the device group; for example, 'cyb06xx7_cm4_dual.ld'.
    */

    ram    (rxw) : ORIGIN = _base_SRAM_CM0P, LENGTH = _size_SRAM_CM0P
    flash  (rx)  : ORIGIN = 0x10000000,      LENGTH = 0x80000

    /* The following regions define device specific
    * memory regions and must not be changed. */

    /* Supervisory flash: User data */
    sflash_user_data  (rx)      : ORIGIN = 0x17000800, LENGTH = 0x800

    /* Supervisory flash: Normal Access Restrictions (NAR) */
    sflash_nar        (rx)      : ORIGIN = 0x17001A00, LENGTH = 0x200

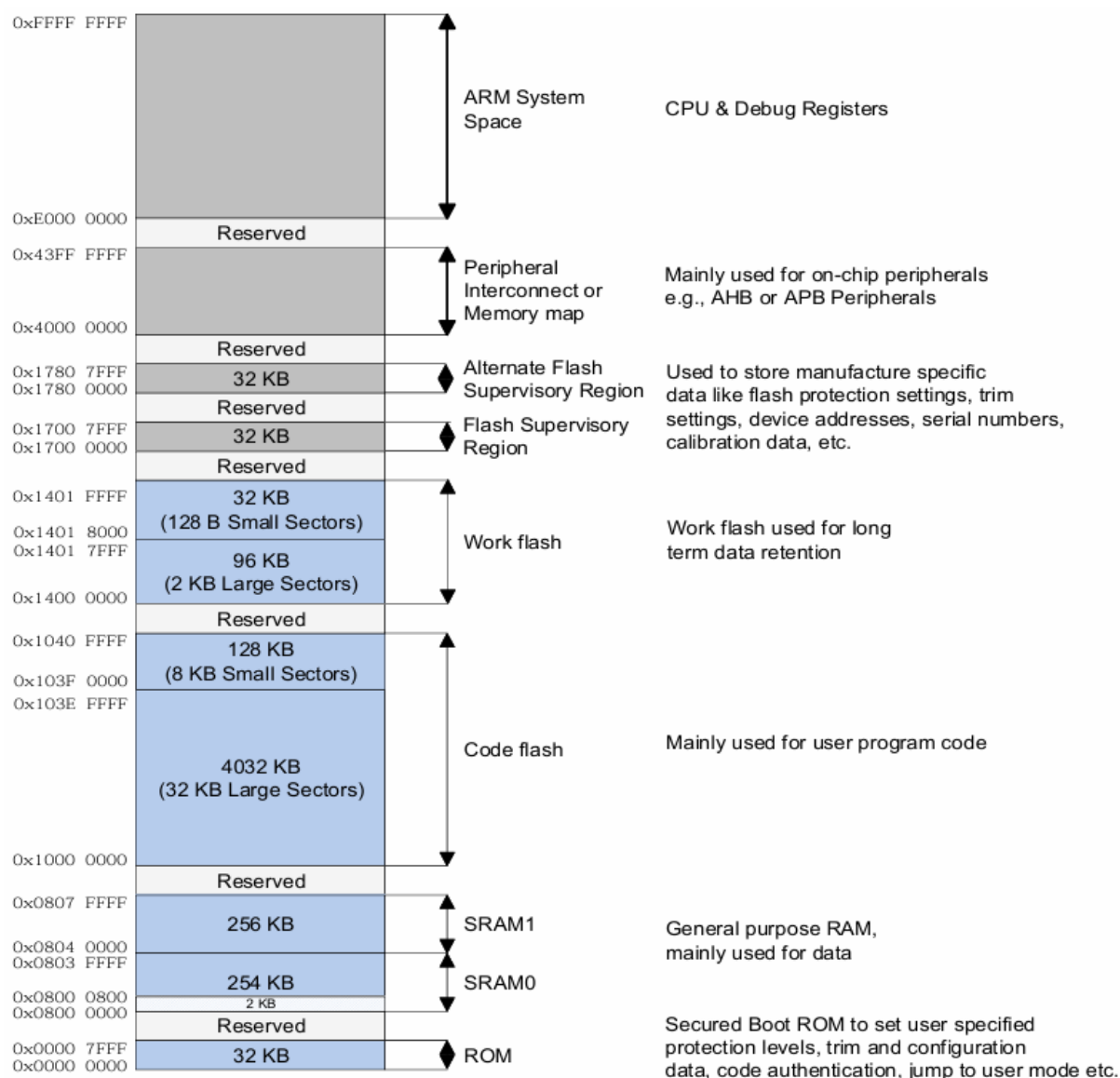
    /* Supervisory flash: Public Key */
    sflash_public_key (rx)      : ORIGIN = 0x17005A00, LENGTH = 0xC00

    /* Supervisory flash: Table of Content # 2 */
    sflash_toc_2      (rx)      : ORIGIN = 0x17007C00, LENGTH = 0x200

    /* Supervisory flash: Table of Content # 2 Copy */
    sflash_rtoc_2     (rx)      : ORIGIN = 0x17007E00, LENGTH = 0x200

    efuse              (r)       : ORIGIN = 0x90700000, LENGTH = 0x100000 /* 1 MB */
}
```

Овде су именовани главни сегменти меморије. Прва два региона, *ram* и *flash*, представљају интерну SRAM и Flash меморију намењену корисничком програму. Атрибути у заградама означавају дозволе: r (читање), w (уписивање), x (извршавање). Према томе, *ram* је означен са (rxw) јер се у SRAM-у налазе подаци над којима програм може да чита, пише и по потреби извршава код (нпр. извршавање из RAM-а ако би се копирале функције у RAM), док је *flash* означен са (rx) – из њега је дозвољено читање и извршавање, али не и упис (Flash је трајна меморија (non-volatile memory), па програм не сме покушати да мења код у њој у току рада).



Слика 3 – Адресна мапа из Infineon Traveo 2G CY2BL datasheet-a

7.3.1. Flash меморија

Додељене адресе и величине ових региона одговарају хардверским спецификацијама микроконтролера. Уочавамо да *flash* почиње на 0x1000_0000 (типично база интерне Flash меморије на ARM Cortex-M системима) и има дужину 0x41_0000 бајтова (4 MB + 64 KB = 4.096 KB + 64 KB = 4.032 KB Large Sectors + 128 KB Small Sectors).

Дужина ове меморије од 0x41_0000 бајтова је подељена на 0x3F_0000 бајтова (4032 KB = 32 KB * 126) који представљају 126 великих сектора меморије од 32 KB (32 KB Large Sectors), и 0x2_0000 бајтова (128 KB = 8 KB * 16) који представљају 16 малих сектора меморије од 8 KB (8 KB Small Sectors).

Неуобичајена величина (није пун степен двојке) указује да од укупне доступне Flash меморије део може бити резервисан за Bootloader или безбедносне функције. У нашем случају, 0x410000 = 4.259.840 бајтова, што је за 65.536 бајтова више од 4 MB – управо тих 64 KB могу да представљају Bootloader област или сектор вишка.

7.3.2. RAM меморија

SRAM за CY2BL породицу микроконтролера почиње на адреси 0x0800_0000 и завршава на 0x0808_0000, што значи да је ова меморија дугачка 0x8_0000 бајтова ($524.288 \text{ B} = 512 \text{ KB} = 256 \text{ KB SRAM0} + 256 \text{ KB SRAM1}$). Интерно је подељена на SRAM0 и SRAM1 меморијске блокове дужине 0x4_0000 бајтова.

ram регион у линкер скрипти коју обрађујемо за M0+ језгро почиње на адреси `_base_SRAM_CM0P` и има дужину `_size_SRAM_CM0P` бајтова. Величину `_base_SRAM_CM0P` смо добили тако што смо на почетак SRAM меморије додали 0x800 бајтова (2 KB) који су резервисани од стране микроконтролера и добили број 0x0800_0800 као почетну адресу. Резервисана меморија је описана симболом `sram_private_for_srom` дужине 0x800 који је дефинисан у почетном делу линкер скрипте. Величина `_size_SRAM_CM0P` је израчуната тако што је од укупне SRAM меморије дужине 0x2_0000 (128 KB) резервисане за M0+ језгро одузета дужина резервисане меморије од 0x800 (2 KB) бајтова и добијена вредност од 0x1_F800 (126 KB) бајтова.

ram регион за M4 језгро који је описан у линкер скрипти коју не обрађујемо, али је део пројекта коришћеног као пример за овај рад, почиње на адреси 0x0802_0000 и има дужину од 0x6_0000 (384 KB) бајтова. Овај број проистиче из расподеле SRAM-а између језгара: укупно има 0x8_0000 (524.288) бајтова SRAM-а, од чега је првих 0x2_0000 (128 KB) додељено M0+ језгру (0x1_F800 B = 126 KB) и резервисаној меморији за систем (0x800 B = 2 KB), а преосталих 384 KB чини RAM простор за M4 језгро. То потврђују вредности у MEMORY секцији – RAM регион M4 језгра почиње након 128 KB (на адреси 0x08020000) и завршава се на крају физичког SRAM-а.

7.3.3. Специјални меморијски региони (Supervisory Flash и eFuse)

Поред два главна региона, у MEMORY блоку видимо низ мањих регија означених као *sflash_user_data*, *sflash_nar*, *sflash_public_key*, *sflash_toc_2*, *sflash_rtoc_2* и *efuse*. Ово одговара специјалним меморијама на датој платформи: Supervisory Flash је посебан део интерне Flash меморије намењен за складиштење података који треба да опстану независно од апликације (нпр. фабричка подешавања, сигурносни кључеви, табеле садржаја за Bootloader), док *efuse* представља електронски фьюз (OTP меморију) за јединствене кључеве и конфигурације чипа.

У литератури термин *fuse* (енг. „осигурач“) у ширем смислу означава елемент заштите у електричним и електронским системима, док се у контексту микроконтролера односи на специјализовану полупроводничку структуру познату као *electronic fuse* (eFuse). У процесу програмирања eFuse меморије, одређени проводни елементи унутар интегрисаног кола подвргавају се трајној физичкој модификацији, најчешће променом отпора или прекидом проводне путање услед контролисаног електричног импулса. Таква промена се на логичком нивоу интерпретира као бинарна вредност (0 или 1), чиме eFuse функционише као једнократно програмабилна меморија (OTP, One-Time Programmable). Ово омогућава да се у чип трајно упишу вредности попут криптографских кључева, конфигурационих параметара или јединствених идентификатора. Због ове неповратности, eFuse се често користи у безбедносним механизмима и заштити интелектуалне својине, јер обезбеђује да једном уписане информације не могу бити измењене у току животног циклуса микроконтролера.

Свака од ових регија има задате адресе (у примеру, нпр. *sflash_user_data* почиње на 0x1700_0800) и величине (нпр. 0x800 бајтова). Атрибут им је (rx) јер се из њих обично само чита, понекад и извршава у случају сигурносних рутина, али не пише, осим путем специјалних процедура. У линкерској скрипти стоји напомена да ови региони не смеју бити мењани – они су део хардверске меморијске мапе. Ипак, њихово постојање у MEMORY секцији омогућава да програмер по потреби у свом коду дефинише секције са тачно тим именима (нпр. секцију *.cy_sflash_user_data*) и да линкер онда те податке смести у предвиђене адресне опсеге. Ако таквих података нема, ови региони остаће празни, али су дефинисани ради комплетности меморијске мапе.

Важно је истаћи да дефинисање региона са њиховим дужинама омогућава линкеру да приликом линковања провери да ли секције стају у додељене меморије. Уколико збир величина секција које су смештене у одређени регион пређе назначену LENGTH дужину, линкер ће пријавити грешку попут *“region FLASH overflowed by X bytes”*, чиме упозорава да је потребно или оптимизовати код/податке или прилагодити расподелу меморије. Овај механизам је од суштинске важности за рано уочавање проблема са прекорачењем ограничених меморијских ресурса на микроконтролеру.

7.4. SECTIONS расподела

SECTIONS блок је срж линкерске скрипте – у њему се одређује како ће све излазне секције (које носе стандардна имена попут .text, .data, .bss, или кориснички дефинисане) бити мапиране у адресни простор дефинисан у MEMORY блоку. Овај део скрипте описује редослед секција у излазном фајлу, њихово поравнање, симболичке ознаке (симболе) које се везују за кључне адресе, као и меморијске регионе у које се свака секција смешта. Анализираћемо редом најважније секције из нашег примера.

```
/* linker_cm0p.ld */

...

/* Library configurations */
GROUP(libgcc.a libc.a libm.a libnosys.a)

/* Linker script to place sections and symbol values. Should be used together
 * with other linker script that defines memory regions FLASH and RAM.
 * It references following symbols, which must be defined in code:
 *   Reset_Handler : Entry of reset handler
 *
 * It defines following symbols, which code can use without definition:
 *   __exidx_start
 *   __exidx_end
 *   __copy_table_start__
 *   __copy_table_end__
 *   __zero_table_start__
 *   __zero_table_end__
 *   __etext
 *   __data_start__
 *   __preinit_array_start
 *   __preinit_array_end
 *   __init_array_start
 *   __init_array_end
 *   __fini_array_start
 *   __fini_array_end
 *   __data_end__
 *   __bss_start__
 *   __bss_end__
 *   __end__
 *   end
 *   __HeapLimit
 *   __StackLimit
 *   __StackTop
 *   __stack
 *   __Vectors_End
 *   __Vectors_Size
 */

SECTIONS
{
    .cy_app_header :
    {
        KEEP(*(.cy_app_header))
    } > flash
```

```

/* Cortex-M0+ application flash area */
.text ORIGIN(flash) :
{
    . = ALIGN(4);
    __Vectors = . ;
    KEEP(*(.vectors))
    . = ALIGN(4);
    __Vectors_End = .;
    __Vectors_Size = __Vectors_End - __Vectors;
    __end__ = .;

    . = ALIGN(4);
    *(.text*)

    KEEP(*(.init))
    KEEP(*(.fini))

    /* .ctors */
    *crtbegin.o(.ctors)
    *crtbegin?.o(.ctors)
    *(EXCLUDE_FILE(*crtend?.o *crtend.o) .ctors)
    *(SORT(.ctors.*))
    *(.ctors)

    /* .dtors */
    *crtbegin.o(.dtors)
    *crtbegin?.o(.dtors)
    *(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors)
    *(SORT(.dtors.*))
    *(.dtors)

    /* Read-only code (constants). */
    *(.rodata .rodata.* .constdata .constdata.* .conststring .conststring.*)

    KEEP(*(.eh_frame*))
} > flash

.ARM.extab :
{
    *(.ARM.extab* .gnu.linkonce.armextab.*)
} > flash

__exidx_start = .;

.ARM.exidx :
{
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > flash
__exidx_end = .;

/* To copy multiple ROM to RAM sections,
 * uncomment .copy.table section and,
 * define __STARTUP_COPY_MULTIPLE in startup_tviibe4m_cm0plus.S */
.copy.table :
{
    . = ALIGN(4);
    __copy_table_start__ = .;

```

```

/* Copy interrupt vectors from flash to RAM */
LONG (__Vectors)                /* From */
LONG (__ram_vectors_start__)    /* To   */
LONG (__Vectors_End - __Vectors) /* Size */

/* Copy data section to RAM */
LONG (__etext)                  /* From */
LONG (__data_start__)           /* To   */
LONG (__data_end__ - __data_start__) /* Size */

__copy_table_end__ = .;
} > flash

/* To clear multiple BSS sections,
 * uncomment .zero.table section and,
 * define __STARTUP_CLEAR_BSS_MULTIPLE in startup_tviibe4m_cm0plus.S */
.zero.table :
{
    . = ALIGN(4);
    __zero_table_start__ = .;
    LONG (__bss_start__)
    LONG (__bss_end__ - __bss_start__)
    __zero_table_end__ = .;
} > flash

__etext = . ;

.ramVectors (NOLOAD) : ALIGN(8)
{
    __ram_vectors_start__ = .;
    KEEP(*(.ram_vectors))
    __ram_vectors_end__ = .;
} > ram

.data __ram_vectors_end__ :
{
    . = ALIGN(4);
    __data_start__ = .;

    *(vtable)
    __sdata_start__ = .;
    *(.data*)
    __sdata_end__ = .;

    . = ALIGN(4);
    /* preinit data */
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP(*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);

    . = ALIGN(4);
    /* init data */
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP(*(SORT(.init_array.*)))
    KEEP(*(.init_array))

```

```

        PROVIDE_HIDDEN (__init_array_end = .);

        . = ALIGN(4);
        /* finit data */
        PROVIDE_HIDDEN (__fini_array_start = .);
        KEEP(*(SORT(.fini_array.*)))
        KEEP(*(.fini_array))
        PROVIDE_HIDDEN (__fini_array_end = .);

        KEEP(*(.jcr*))
        . = ALIGN(4);

        KEEP(*(.cy_ramfunc*))
        . = ALIGN(4);

        __data_end__ = .;
} > ram AT>flash

/* Place variables in the section that should not be initialized during the
 * device startup.
 */
.noinit (NOLOAD) : ALIGN(8)
{
    KEEP(*(.noinit))
} > ram

/* The uninitialized global or static variables are placed in this section.
 *
 * The NOLOAD attribute tells linker that .bss section does not consume
 * any space in the image. The NOLOAD attribute changes the .bss type to
 * NOBITS, and that makes linker to A) not allocate section in memory, and
 * A) put information to clear the section with all zeros during application
 * loading.
 *
 * Without the NOLOAD attribute, the .bss section might get PROGBITS type.
 * This makes linker to A) allocate zeroed section in memory, and B) copy
 * this section to RAM during application loading.
 */
.bss (NOLOAD):
{
    . = ALIGN(4);
    __bss_start__ = .;
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
    __bss_end__ = .;
} > ram

.heap (NOLOAD):
{
    __HeapBase = .;
    __end__ = .;
    end = __end__;
    KEEP(*(.heap*))
    . = ORIGIN(ram) + LENGTH(ram) - STACK_SIZE;

```

```
    __HeapLimit = .;
} > ram

/* .stack_dummy section doesn't contains any symbols. It is only
 * used for linker to calculate size of stack sections, and assign
 * values to stack symbols later */
.stack_dummy (NOLOAD):
{
    KEEP(*(.stack*))
} > ram

/* Set stack top to end of RAM, and stack limit move down by
 * size of stack_dummy section */
__StackTop = ORIGIN(ram) + LENGTH(ram);
__StackLimit = __StackTop - SIZEOF(.stack_dummy);
PROVIDE(__stack = __StackTop);

/* Check if data + heap + stack exceeds RAM limit */
ASSERT(__StackLimit >= __HeapLimit, "region RAM overflowed with stack")

/* Supervisory Flash: User data */
.cy_sflash_user_data :
{
    KEEP(*(.cy_sflash_user_data))
} > sflash_user_data

/* Supervisory Flash: Normal Access Restrictions (NAR) */
.cy_sflash_nar :
{
    KEEP(*(.cy_sflash_nar))
} > sflash_nar

/* Supervisory Flash: Public Key */
.cy_sflash_public_key :
{
    KEEP(*(.cy_sflash_public_key))
} > sflash_public_key

/* Supervisory Flash: Table of Content # 2 */
.cy_toc_part2 :
{
    KEEP(*(.cy_toc_part2))
} > sflash_toc_2

/* Supervisory Flash: Table of Content # 2 Copy */
.cy_rtoc_part2 :
{
    KEEP(*(.cy_rtoc_part2))
} > sflash_rtoc_2
```

```
/* eFuse */
.cy_efuse :
{
    KEEP(*(.cy_efuse))
} > efuse

/* These sections are used for additional metadata (silicon revision,
 * Silicon/JTAG ID, etc.) storage.
 */
.cymeta      0x90500000 : { KEEP(*(.cymeta)) } :NONE
}

/* The following symbols used by the cymcuelftool. */
/* Flash */
__cy_memory_0_start    = 0x10000000;
__cy_memory_0_length   = 0x00410000;
__cy_memory_0_row_size = 0x200;

/* Supervisory Flash */
__cy_memory_2_start    = 0x17000000;
__cy_memory_2_length   = 0x8000;
__cy_memory_2_row_size = 0x200;

/* eFuse */
__cy_memory_4_start    = 0x90700000;
__cy_memory_4_length   = 0x100000;
__cy_memory_4_row_size = 1;

/* EOF */
```


7.4.1. Секција `.cy_app_header`: заглавље апликације и векторска табела

У делу `SECTIONS` линкер скрипте врши се распоређивање свих секција извршног програма у одговарајуће меморијске регионе и дефинишу се посебни симболи који помажу стартап рутини да правилно иницијализује меморију. Први део је секција `.cy_app_header`, која се смешта на сам почетак флеш меморије (адреса `0x10000000`) и служи за складиштење заглавља апликације у системима са безбедним бутовањем. Ово заглавље садржи метаподатке о апликацији (нпр. величину, ID типа апликације, број језгара) и омогућава `bootloader`-у или `ROM` коду да верификује и правилно учита апликацију. Дефиниција `KEEP*(.cy_app_header)` обезбеђује да се садржај секције заглавља не оптимизује од стране линкера, чак и ако на њега нема референци из кода (ово је важно јер `bootloader` чита ово заглавље, иако га сама апликација директно не користи). Напомена је да се у `Cortex-M` архитектури векторска табела прекида обично очекује на почетку флеша (на адреси ресет-вектора) ради исправног функционисања `NVIC`-а. У нашем случају, због присуства `.cy_app_header` испред, почетак векторске табеле је померен. Овај проблем се решава тако што `bootloader` или стартап рутина знају за заглавље и користе његове информације (нпр. поље `core0Vt` у заглављу) да пронађу и поставе векторску табелу на исправно место.

```
/* linker_cm0p.ld */  
  
...  
  
SECTIONS  
{  
    .cy_app_header :  
    {  
        KEEP*(.cy_app_header)  
    } > flash  
  
    ...  
}
```

7.4.2. Секција `.text`: програмски код, векторска табела и константни подаци

Следећа секција `.text` дефинише се са атрибутом `> flash` (тј. смешта се у флеш меморију) и почиње на `ORIGIN(flash)` – то је адреса почетка флеша (`0x10000000`). На самом почетку `.text` секције поравнате на 4 бајта, лоцира се векторска табела прекида: поставља се симбол `__Vectors` на текућу адресу (тј. на почетак векторске табеле), затим се укључује садржај свих улазних секција именованих `.vectors` (помоћу `KEEP*(.vectors)`) да би се осигурало да линкер задржи табелу). Векторска табела, коју обично генерише стартап асемблерски код, садржи почетну вредност стек показивача и адресе свих обрађивача прекида (`Reset_Handler`, `NMI`, `HardFault`, итд.). У асемблерском стартап фајлу ова табела је обележена управо у `.vectors` секцији и започиње вредношћу почетног садржаја стек-показивача (`__StackTop`), затим адресом функције за ресет, па даље осталим векторима. Након што се укључи садржај `.vectors`, локаунтер – *location counter* (означава интерни бројач који `GNU ld` линкер користи како би пратио *текућу адресу* у којој ће бити смештен следећи бајт или реч излазне датотеке) се поново поравна на 4 бајта, и постављају се симболи `__Vectors_End` (адреса након краја векторске табеле) и

`__Vectors_Size` (величина табеле, израчуната као разлика те две адресе). Ови симболи се могу користити за копирање табеле или провере величине. У наставку, поставља се привремено и симбол `__end__ = .;` – он тренутно добија вредност након векторске табеле, али ће касније бити поново дефинисан када се заврши распоред података у RAM-у (симбол **end** се традиционално користи да означи крај статичких података у RAM-у, односно почетак heap-a).

```
/* linker_cm0p.ld */

...

/* Cortex-M0+ application flash area */
.text ORIGIN(flash) :
{
    . = ALIGN(4);
    __Vectors = . ;
    KEEP(*(.vectors))
    . = ALIGN(4);
    __Vectors_End = .;
    __Vectors_Size = __Vectors_End - __Vectors;
    __end__ = .;

    . = ALIGN(4);
    *(.text*)

    KEEP(*(.init))
    KEEP(*(.fini))

    /* .ctors */
    *crtbegin.o(.ctors)
    *crtbegin?.o(.ctors)
    *(EXCLUDE_FILE(*crtend?.o *crtend.o) .ctors)
    *(SORT(.ctors.*))
    *(.ctors)

    /* .dtors */
    *crtbegin.o(.dtors)
    *crtbegin?.o(.dtors)
    *(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors)
    *(SORT(.dtors.*))
    *(.dtors)

    /* Read-only code (constants). */
    *(.rodata .rodata.* .constdata .constdata.* .conststring .conststring.*)

    KEEP(*(.eh_frame*))
} > flash

...
```

После векторске табеле, унутар `.text` секције смешта се остатак програмског кода и константи. Прво се локаунтер поравна на 4 бајта, а затим се укључују све `.text*` секције из улазних објеката, што обухвата машински код свих функција програма. Након кода, директивама `KEEP(*(.init))` и `KEEP(*(.fini))` обезбеђује се задржавање евентуалних секција за иницијализацију и финализацију (нпр. код који се извршава пре `main` или по завршетку програма). Ове секције се ретко користе у једноставним `embedded`

пројектима, али их GCC runtime може креирати (нпр. `.init` за иницијализацију хардвера). У наставку су обрађене `.ctors` и `.dtors` секције, које носе листе конструктора и деструктора глобалних објеката у C++ програмима. Линкер овде сакупља све објекте који садрже листе конструктора (`.ctors`) и деструктора (`.dtors`): прво се додају уноси из стартних објеката (`crtbegin`) који означавају почетак листе, затим сви `.ctors` из корисничких објеката (осим оних у `crtend` који означавају крај), па сортирани уноси (директива `SORT(.ctors.*)` хвата све секције које почињу на `.ctors.` и сортира их), и најзад секција `.ctors` ако постоји. Сличан поступак важи и за `.dtors` (деструктори). Овако припремљени низови показивача на конструкторске функције биће позвани током покретања програма, типично кроз позив функције попут `_init` или путем извршања `__libc_init_array()` у стартуп коду, која пролази кроз листе и позива функције конструктора и преиницијализације. На тај начин се обезбеђује да се, рецимо, глобални објекти у C++-у правилно конструишу пре корисничког кода, а деструктори би се позвали приликом гашења програма (мада у `embedded` контексту програм обично не излази из `main`, па се деструктори рутински не извршавају).

У наставку `.text` секције смештају се **константни подаци** – све секције означене као `.rodata`, `.constdata` или чак стрингови константи. Линкер директива `*(.rodata .rodata.* .constdata .constdata.* .conststring .conststring.*)` сакупља све потсекције константних података и смешта их ту, непосредно иза кода. Ови подаци остају у флешу (само за читање) и доступни су програму као константе (нпр. низови литерала, `lookup` табеле и сл.). Одмах за њима, специјалном директивом `KEEP(*(eh_frame*))` укључује се `.eh_frame` секција, ако постоји. `.eh_frame` садржи информације за `stack unwinding` и руковање изузецима (`exception handling frame data`) коришћене углавном у C++ и при дебаговању. Иако мали `microcontroller` пројекти обично немају бацање C++ изузетака, ова секција је предвиђена за случај да је укључена подршка за изузетке, јер садржи метаподатке неопходне за праћење стек-фрејмова током „`unwind`“-а стека.

7.4.3. Секције .ARM.extab и .ARM.exidx: табеле изузетака и stack unwinding

Након завршетка .text секције (која је обухватила код и константе у флешу), линкер скрипта обрађује две специјалне ARM-везане секције: .ARM.extab и .ARM.exidx. Секција .ARM.extab (ARM Exception Table) смешта табеле изузетака за C++ (детаље о баченим изузецима, потребне за „stack unwinding“). Одмах потом, дефинише се симбол __exidx_start на текућу адресу, па се отвара .ARM.exidx секција у коју се смешта индекс табела изузетака – низ записа који указују на рутину за отклањање стека за сваки део кода. Након укључивања свих .ARM.exidx* секција, затвара се ова секција и дефинише симбол __exidx_end. Ови симболи __exidx_start и __exidx_end обележавају почетак и крај табеле изузетака и користе се од стране runtime-а (нпр. при бацању изузетка, функција за отклањање стека зна докле да иде кроз таблицу изузетака).

```
/* linker_cm0p.ld */

...

.ARM.extab :
{
    *(.ARM.extab* .gnu.linkonce.armextab.*)
} > flash

__exidx_start = .;

.ARM.exidx :
{
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > flash
__exidx_end = .;

...
```

7.4.4. Секција .copy.table: табела за копирање секција из флеша у RAM

До овде смо распоредили сав код и константе у флешу. Следећи блок у скрипти дефинише структуре које омогућавају копирање садржаја из флеша у RAM и иницијализацију секција при покретању програма. Коментар у скрипти наводи да се, ради копирања више секција из ROM у RAM, може користити структура .copy.table уз дефинисање макроа __STARTUP_COPY_MULTIPLE у startup коду. У нашој скрипти .copy.table секција је активно наведена и изгледа овако:

```
/* linker_cm0p.ld */

...

/* To copy multiple ROM to RAM sections,
 * uncomment .copy.table section and,
 * define __STARTUP_COPY_MULTIPLE in startup_tviibe4m_cm0plus.S */
.copy.table :
{
    . = ALIGN(4);
    __copy_table_start__ = .;

    /* Copy interrupt vectors from flash to RAM */
    LONG (__Vectors)                /* From */
    LONG (__ram_vectors_start__)    /* To   */
    LONG (__Vectors_End - __Vectors) /* Size */

    /* Copy data section to RAM */
    LONG (__etext)                  /* From */
    LONG (__data_start__)           /* To   */
    LONG (__data_end__ - __data_start__) /* Size */

    __copy_table_end__ = .;
} > flash

...
```

Поравна се локаунтер на 4 бајта, затим се бележи почетак табеле у __copy_table_start__. У табелу се уписују троструки записи (сваки се састоји од три LONG вредности од 32 бита) који описују сегменте за копирање. Први такав запис односи се на векторску табелу прекида: он садржи адресу извора (__Vectors, почетак векторске табеле у флешу), одредишну адресу у RAM-у (__ram_vectors_start__, место где треба копирати таблицу у RAM) и величину блока (рачунату као __Vectors_End - __Vectors). Овим записом се омогућава опционално копирање целе векторске табеле у RAM, уколико то стартап рутина предвиди. Други запис у табели покрива главну .data секцију иницијализованих података: он садржи адресу извора података у флешу (__etext – место где се у флешу налазе почетне вредности .data секције), затим одредишну адресу у RAM-у где ти подаци треба да буду смештени (__data_start__), као и величину иницијализационих података (рачунату као __data_end__ - __data_start__). Након списка секција за копирање, поставља се симбол __copy_table_end__ на крај ове табеле. Ова табела ће у току покретања бити коришћена ако је у стартап коду омогућена логика за вишеструко копирање (нпр. у CMSIS шаблонским startup датотекама постоји условни код који пролази кроз записе између __copy_table_start__ и __copy_table_end__ и извршава копирања). У нашем случају, пошто табела обухвата баш два сегмента (векторе и .data), стартап рутина за Cortex-M0+ језгро (фајл startup_tviibe4m_cm0plus.S из пример

пројекта) је конфигурисана да користи ову табелу – у њој је дефинисан `__STARTUP_COPY_MULTIPLE` симбол, што покреће петљу за копирање више секција. У тој startup рутини постоје екстерно декларисани симболи `__copy_table_start__` и др, и код који их обрађује. Исечак из `startup_tviibe4m_cm0plus.S` то приказује.

```
/* startup_tviibe4m_cm0plus.S*/

...

    /* Copy flash vectors and data section to RAM */
    #define __STARTUP_COPY_MULTIPLE

...

#ifdef __STARTUP_COPY_MULTIPLE
/* Multiple sections scheme.
 *
 * Between symbol address __copy_table_start__ and __copy_table_end__,
 * there are array of triplets, each of which specify:
 *   offset 0: LMA of start of a section to copy from
 *   offset 4: VMA of start of a section to copy to
 *   offset 8: size of the section to copy. Must be multiply of 4
 *
 * All addresses must be aligned to 4 bytes boundary.
 */
    ldr    r4, =__copy_table_start__
    ldr    r5, =__copy_table_end__

...

```

7.4.5. Секција `.zero.table`: иницијализација BSS региона занулирањем

Сличан принцип важи и за `.zero.table` секцију, која следи у скрипти. У нашој скрипти `.zero.table` је дефинисана и садржи: поравнање на 4, затим `__zero_table_start__` на почетак, па један запис који представља пар (адреса почетка `.bss` секције, и дужина `.bss` секције). Конкретно, уписује се `LONG(__bss_start__)` и `LONG(__bss_end__ - __bss_start__)`. Овде видимо да је тренутно предвиђена само једна непрекидна BSS регија за чишћење (сви неиницијализовани подаци заједно), па су довољна ова два параметра – почетак и величина тог блока. Симбол `__zero_table_end__` се ставља на крај табеле. Ако би постојало више одвојених BSS секција (рецимо, посебна BSS у другом RAM сегменту), оне би могле бити додате у ову табелу као додатни парови. Уколико је у стартап коду омогућена логика за чишћење више BSS подручја, онда ће он читати ову табелу и обавити задато чишћење меморије. У супротном (као што је чешће), стартап код ће једноставно обрисати цео BSS према симболима `__bss_start__` и `__bss_end__` без коришћења табеле. Видимо да су симболи `__STARTUP_CLEAR_BSS_MULTIPLE`, `__zero_table_start__`, `__zero_table_end__` дефинисани у `startup_tviibe4m_cm0plus.S` фајлу.

```
/* startup_tviibe4m_cm0plus.S */

...

#ifdef __STARTUP_CLEAR_BSS_MULTIPLE
/* Multiple sections scheme.
 *
 * Between symbol address __zero_table_start__ and __zero_table_end__,
 * there are array of tuples specifying:
 *   offset 0: Start of a BSS section
 *   offset 4: Size of this BSS section. Must be multiple of 4
 */
    ldr    r3, __zero_table_start__
    ldr    r4, __zero_table_end__

...

```

Исечак из `linker_cm0p.ld` фајла који приказује део скрипте за иницијализацију више BSS секција уколико је симбол `__STARTUP_CLEAR_BSS_MULTIPLE` дефинисан у `startup_tviibe4m_cm0plus.S`:

```
/* linker_cm0p.ld */

...

/* To clear multiple BSS sections,
 * uncomment .zero.table section and,
 * define __STARTUP_CLEAR_BSS_MULTIPLE in startup_tviibe4m_cm0plus.S */
.zero.table :
{
    . = ALIGN(4);
    __zero_table_start__ = .;
    LONG (__bss_start__)
    LONG (__bss_end__ - __bss_start__)
    __zero_table_end__ = .;
} > flash

...

```

7.4.6. Секција .ramVectors: RAM копија векторске табеле

Након затварања .zero.table секције, линкер поставља симбол __etext на текућу адресу. Овај симбол тиме означава крај свих секција које се налазе у флешу и уједно представља адресу одакле почињу иницијализациони подаци за .data секцију. Уобичајено, __etext (енд-текст) или сличан симбол (нпр. _sidata) користи се у стартуп коду као изворишна адреса за копирање .data секције – тј. то је адреса у ROM-у одакле треба почети копирање података у RAM. У нашем случају, __etext је управо адреса на којој се у флешу налази први бајт података који треба пресликати у RAM секцију .data. Из горње .copy.table видимо да се __etext користи као „From“ поље за копирање .data сегмента. Важно је напоменути да __etext није само крај .text сегмента, већ крај свих флеш секција које претходе .data LMA (Load Memory Address) – пошто смо у флешу иза кода и константи додали и таблице за копирање/чишћење, __etext долази тек након њих. Тиме је осигурано да сви ти подаци (укључујући и саме табеле) остану у флешу и да се неће случајно преклопити са сегментом .data у флешу. Другим речима, иницијализациони блок за .data биће постављен у флешу тек након .copy.table и .zero.table, тако да они не ремете једни друге.

```
/* linker_cm0p.ld */

...

__etext = . ;

.ramVectors (NOLOAD) : ALIGN(8)
{
    __ram_vectors_start__ = .;
    KEEP(*(.ram_vectors))
    __ram_vectors_end__ = .;
} > ram

...
```

Након што је дефинисан __etext, прелазимо на секције које се налазе у RAM меморији. Прва међу њима је .ramVectors, дефинисана са атрибутом > ram и NOLOAD, и поравнањем на 8 бајтова. Ова секција представља **RAM копију векторске табеле**. Унутар ње, симбол __ram_vectors_start__ се поставља на тренутну адресу (што означава почетак простора у RAM-у за векторе), затим се укључује (KEEP) садржај свих улазних секција именованих .ram_vectors, и на крају се симбол __ram_vectors_end__ поставља на крај те секције. Сама .ram_vectors секција нема LOAD адрес у флешу (атрибут NOLOAD значи да се она не појављује у извршној слици на диску), већ само резервише простор у RAM-у. Како садржај бива постављен? У нашем случају, startup асемблерски код је већ резервисао ово место: у фајлу startup_tviiibe4m_cm0plus.S налази се дефиниција секције .ram_vectors која резервише довољно бајтова (позивом .space) колико износи величина векторске табеле. Он то постиже тако што је раније израчунао __VectorsSize (или еквивалент) у асемблеру и онда у .ram_vectors секцији рекао да се резервише толико бајтова. При линковању, овај резервисани блок бива смештен управо у .ramVectors излазну секцију. Зашто се ово ради? Код одређених микроконтролера (поготово са безбедносним функцијама или dual-core архитектуром), уобичајена је пракса да се након иницијализације векторска табела пресели у RAM, како би се по потреби могла модификовати у runtime-у (на пример, да bootloader преусмери одређене прекиде, или да

се омогући променљив векторски сто током рада система). Код PSoC 6/Traveo II породица (чији је овај пример линкер скрипте) конкретно, CM0+ језгро често служи као безбедносно или бут језгро, па може копирати своју векторску табелу у RAM ради заштите, или може припремити RAM векторску табелу за апликацију која ће се касније покренути на другом језгру. У нашем случају, `.copy.table` предвиђа копирање векторске табеле (први запис), што значи да ће стартап код копирати све вредности из флеша (оригинална табела у `.vectors`) у овај резервисани RAM простор (`__ram_vectors_start__...__ram_vectors_end__`). Након тога, по потреби, процесор може подесити VTOR (Vector Table Offset Register) регистар на адресу `__ram_vectors_start__` како би процесор убудуће користио RAM верзију табеле прекида. VTOR је специјални системски регистар у ARM Cortex-M архитектури, смештен у оквиру блока System Control Block (SCB). Његова намена је да одреди *почетну адресу векторске табеле прекида* коју процесор користи за тражење адреса сервисних рутина (ISR). Напомињемо да Cortex-M0+ језгра у овој фамилији подржавају релокацију векторске табеле (VTOR регистар), тако да је овај механизам изводљив. Дакле, секција `.ramVectors` обезбеђује да простор у RAM-у за векторе буде резервисан и познатих адреса (преко симбола), а стварно копирање ће се обавити приликом покретања (тј. у `Reset_Handler` рутини).

7.4.7. Секција .data: иницијализовани подаци, низови конструктора и RAM функције

Главна секција података у RAM-у је секција .data. У нашој скрипти, она је дефинисана овако: .data __ram_vectors_end__ : { ... } > ram AT>flash. Ово значи да ће .data секција имати своју виртуелну адресу (VMA – Virtual Memory Address) у RAM-у непосредно након краја .ramVectors секције (симбол **ram_vectors_end** одређује почетак .data), али ће јој садржај бити смештен у флеш (LMA – Load Memory Address) на позицији која следи након претходних флеш секција (пошто је назначено AT>flash). Другим речима, линкер ће резервисати место у RAM-у за све .data променљиве, али ће иницијални битови за те променљиве бити упаковани у извршну слику у флешу (након __etext адресе). При покретању, ти битови ће бити копирани у RAM, чиме се успостављају почетне вредности глобалних и статичких променљивих које имају иницијализаторе. Унутар .data секције, прво поравнавамо адресу на 4 (. = ALIGN(4)) ради сигурности да су сви објекти правилно поравнати у меморији. Затим постављамо симбол __data_start__ = .; да означимо почетак података у RAM-у.

```
/* linker_cm0p.ld */

...

.data __ram_vectors_end__ :
{
    . = ALIGN(4);
    __data_start__ = .;

    *(vtable)
    __sdata_start__ = .;
    *(.data*)
    __sdata_end__ = .;

    . = ALIGN(4);
    /* preinit data */
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP(*(SORT(.preinit_array)))
    PROVIDE_HIDDEN (__preinit_array_end = .);

    . = ALIGN(4);
    /* init data */
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP(*(SORT(.init_array.*)))
    KEEP(*(SORT(.init_array)))
    PROVIDE_HIDDEN (__init_array_end = .);

    . = ALIGN(4);
    /* finit data */
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP(*(SORT(.fini_array.*)))
    KEEP(*(SORT(.fini_array)))
    PROVIDE_HIDDEN (__fini_array_end = .);

    KEEP(*(SORT(.jcr*)))
    . = ALIGN(4);

    KEEP(*(SORT(.cy_ramfunc*)))
}
```

```

    . = ALIGN(4);

    __data_end__ = .;

} > ram AT>flash

```

...

Први садржај у `.data` који се смешта је све из улазних секција означених са `(vtable)` – овај шаблон обухвата секције које компајлер генерише за табеле виртуелних функција у C++ (виртуелне таблице класа могу бити смештене у посебну `.vtable` секцију) или потенцијално друге мале податке који су означени тим именом. Пошто су ове табеле променљиве (пуне се адресама функција које су такође фиксне у програму), може их бити потребно копирати у RAM ако нису константне. Одмах након тога, поставља се симбол `__sdata_start__ = .;`, па се укључују све преостале улазне `.data*` секције (све глобалне променљиве које имају иницијализацију). Након што су све те променљиве распоређене, поставља се `__sdata_end__ = .;`. Овде ознаке **`sdata_start`** и **`sdata_end`** указују на евентуалне границе "small data" сегмента – у неким ABI-јевима (Application Binary Interface) за RISC архитектуре, као што је ARM EABI, одређени мали глобални подаци (који стају у пар бајтова) могу се држати у посебном сегменту за који постоје оптимизоване инструкције приступа. Овде се сва `.data` сматра „`sdata`“ сегментом, па ове ознаке могу помоћи дебагеру или runtime-у ако је то потребно.

Након што су све променљиве смештене, поново поравнавамо на 4 и започињемо секције низова функција које се морају позвати при стартапу или гашењу програма. Прво је `.preinit_array`: поставља се скривени (HIDDEN) симбол `__preinit_array_start` на тренутну адресу, затим се `KEEP*(.preinit_array)` укључују све секције које садрже низ „pre-init“ функција, па се ставља `__preinit_array_end`. Pre-initialization низ је предвиђен за функције које треба позвати *pre* стандардне иницијализације библиотека – ово се ретко користи, углавном за системске ствари пре конструктора (нпр. за *sanity check* тј. *sanity test* или слично). Потом следи `.init_array`: на сличан начин, дефинишу се `__init_array_start`, укључују сви елементи из `.init_array` секција (прво сортирани именовани `.init_array.*`, па несортирани `.init_array` остатак), и онда `__init_array_end`. `.init_array` секције садрже низове показивача на функције конструктора (иницијализациони код) који треба да се позову након што су статичке променљиве већ у RAM-у. Глобални C++ објекти, као и неке иницијализације библиотека (нпр. за `libc`), региструју своје конструктор функције у `.init_array`. При покретању програма, након што се изврши копирање `.data` и брисање `.bss`, обично се позива функција која пролази од `__preinit_array_start` до `__preinit_array_end`, затим од `__init_array_start` до `__init_array_end` и позива све те функције. На тај начин се извршавају конструктори глобалних објеката пре уласка у `main`. Након тога у меморијском распореду следи `.fini_array`, са својим почетним и крајњим симболима (`__fini_array_start`, `__fini_array_end`), у који се смештају показивачи на функције које треба позвати *на крају* програма (деструктори глобалних објеката). Иако се у уграђеним системима ретко зове функција `exit()` која би покренула те деструкторе, секција је ту ради конзистентности са ABI-јем и за случај да се ипак користи. Симболи ових низова означени су као `PROVIDE_HIDDEN`, што значи да ће бити дефинисани овде ако нису већ негде дефинисани и биће локални за `exe` (не извозе се глобално).

Даље, `KEEP*(.jcr*)` директива укључује све `.jcr` секције. `.jcr` је секција која се користи у GNU toolchain-у за „Java-powered“ регистре, односно садржи `pointer` на тзв.

`_Jv_RegisterClasses` функцију у случају да се користи GNU Java компајлер. У нашим условима, ово најчешће није примењиво, па ће та секција бити празна, али linker шаблон је задржава за случај да постоји (да је не би одбацио оптимизатор). Потом се поново поравна на 4 и укључује `.cy_ramfunc*` секције: `KEEP(*(.cy_ramfunc*))`. Овај сегмент је специфичан за Cypress/Infineon платформу и користи се за функције које су означене да треба да се извршавају из RAM-а (такве функције корисник може означити атрибутом, нпр. `CY_SECTION(".cy_ramfunc")`), да би критичне рутине копирао у RAM). Тиме линкер све такве функције ставља у ову секцију, која је део `.data` региона, што значи да ће њихов машински код бити у бинарном фајлу, односно у FLASH-у, али ће при покретању бити копиран у RAM и одатле извршаван. Ово је битно за функције које морају брзо да раде или да раде док је флеш меморија недоступна (нпр. рутине за програмирање флеша сама себе мора да се извршава из RAM-а, или рутине које искључују привремено кеш/cache или флеш/flash). Програмер корисник такве функције обележава специјалним атрибутом (нпр. `__attribute__((section(".ramfunc")))`), а линкер обухвата њихово премештање. Један пример како се функција може сместити у RAM је дат и на Memfault форуму – коришћењем `__attribute__((section(".data")))` програмер је успео да распореди тело функције у RAM сегмент. У нашем случају, коришћена је посебна секција `.cy_ramfunc` уместо опште `.data`, али принцип је исти: машински код функције биће присутан у извршној слици (у флешу), а након ресета стартап копира у RAM тачно као и обичне иницијализационе податке. Директива `KEEP` овде осигурава да линкер не одстрани ову функцију мислећи да је неискоришћена (јер је можда нигде не зовемо директно, већ је позивамо преко показивача или прекида).

Након укључених `.cy_ramfunc` секција, поново се поравна адреса и поставља симбол `__data_end__ = .`; да означи крај свих података у RAM-у. Ово је важан симбол – он заједно са `data_start` одређује опсег иницијализованих података. У комбинацији са `__etext`, он је већ коришћен за креирање `.copy.table` записа (видели смо `LONG(__data_end__ - __data_start__)` као величину за копирање). Стартап код типично користи `data_start`, `data_end` и адресу почетка података у флешу (коју смо означили са `__etext`) да би урадио једноставно копирање иницијализационих података – у случају да није омогућен `__STARTUP_COPY_MULTIPLE`, обично у `startup` функцији наиђемо на нешто попут: `LDR r1, __etext`; `LDR r2, __data_start`; `LDR r3, __data_end`; и затим копирају у петљи ту количину бајтова. Наш линкер скрипт је спреман за оба сценарија (вишеструко копирање преко табеле, или једноставно копирање целог блока).

```
/* startup_tviibe4m_cm0plus.S */

...

#ifdef __STARTUP_COPY_MULTIPLE
/* Multiple sections scheme. */

...

#else
/* Single section scheme.
 *
 * The ranges of copy from/to are specified by following symbols
 * __etext: LMA of start of the section to copy from. Usually end of text
 * __data_start__: VMA of start of the section to copy to
 * __data_end__: VMA of end of the section to copy to
 *
 * All addresses must be aligned to 4 bytes boundary.
```

```

*/
    ldr    r1, __etext
    ldr    r2, __data_start__
    ldr    r3, __data_end__

    subs   r3, r2
    ble    .L_loop1_done

.L_loop1:
    subs   r3, #4
    ldr    r0, [r1,r3]
    str    r0, [r2,r3]
    bgt    .L_loop1

.L_loop1_done:
#endif /*__STARTUP_COPY_MULTIPLE */

...

```

7.4.8. Секција .noinit: променљиве које задржавају вредности преко ресета

Сада прелазимо на секције које се такође налазе у RAM-у, али не садрже иницијализационе податке у флешу (NOLOAD секције) – оне се само алоцирају у меморијском простору RAM-а, а њихову иницијализацију обавља код након ресета, обично постављањем на нулу или их уопште не дира. Прва је .noinit секција. Дефинисана је као .noinit (NOLOAD) : ALIGN(8) { KEEP(*(.noinit)) } > ram. Она обухвата променљиве које *не треба* иницијализовати при старту. То значи да ако нека глобална или статичка променљива користи атрибут (нпр. у GCC је то `__attribute__((section(".noinit")))` или специјални атрибут `__no_init`), они ће пасти у ову секцију. Стартуп код их неће ни копирати ни брисати. Ово је корисно за променљиве које треба да задрже вредност преко софтверског ресета (нпр. одређене заставице које сигнализирају да ли је систем већ био покренут или не, или подаци који треба да преживе излазак из `main`-а и поновни улазак). Пошто је NOLOAD, линкер их неће укључити у бинарну слику (нема потребе, нема иницијалних вредности), али ће резервисати простор у RAM-у да се не преклопе са другима.

```

/* linker_cm0p.ld */

...

/* Place variables in the section that should not be initialized during the
 * device startup.
 */
.noinit (NOLOAD) : ALIGN(8)
{
    KEEP(*(.noinit))
} > ram

...

```

7.4.9. Секција .bss: неиницијализовани подаци у RAM-у

Следи секција .bss, такође NOLOAD. Она садржи све неиницијализоване глобалне и статичке променљиве (оне које у C/C++ коду нису експлицитно постављене на неку вредност, подразумевано треба да буду 0 према C стандарду). Дефиниција је:

```
/* linker_cm0p.ld */

...

/* The uninitialized global or static variables are placed in this section.
 *
 * The NOLOAD attribute tells linker that .bss section does not consume
 * any space in the image. The NOLOAD attribute changes the .bss type to
 * NOBITS, and that makes linker to A) not allocate section in memory, and
 * B) put information to clear the section with all zeros during application
 * loading.
 *
 * Without the NOLOAD attribute, the .bss section might get PROGBITS type.
 * This makes linker to A) allocate zeroed section in memory, and B) copy
 * this section to RAM during application loading.
 */
.bss (NOLOAD):
{
    . = ALIGN(4);
    __bss_start__ = .;
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
    __bss_end__ = .;
} > ram

...
```

Поравна се на 4, постави симбол `__bss_start__` на почетак ове секције, затим се укључе све .bss секције из објеката `*(.bss*)` као и сви *COMMON* симболи. *COMMON* симболи представљају непридружене глобалне променљиве (оне декларисане а недефинисане, које линкер ставља у .bss ако нема другу дефиницију). Након укључивања свих њих, опет се поравна крај и стави `__bss_end__`. Атрибут *NOLOAD* означава да ова секција нема података у самом извршном фајлу – она ће једноставно заузети простор у RAM-у између `bss_start` и `bss_end`, али програм неће садржати то место (уместо тога, loader или стартап код треба да га обришу). У стартап рутини, након копирања .data, типично следи петља која сетује све адресе од `bss_start` до `bss_end` на нулу, чиме се испуњава обећање C стандарда да неиницијализоване глобалне буду 0. Уколико је омогућена .zero.table, та петља ће можда бити комплекснија (креће се кроз табелу опсега за брисање), али ефекат је исти. Коментари у скрипти објашњавају да ако .bss секција **нема** *NOLOAD* атрибут, линкер би је третирао као *PROGBITS* (део слике) што би значило да би снимио низ нула у бинарном фајлу и потом при учитавању копирао нуле у RAM (што је непотребно трошење меморије). *NOLOAD* то избегава и каже да нема шта да се снима – довољно је знати да тај простор треба попунити нулама.

```

/* startup_tviibe4m_cm0plus.S */

...

#elif defined (__STARTUP_CLEAR_BSS)
/* Single BSS section scheme.
 *
 * The BSS section is specified by following symbols
 * __bss_start__: start of the BSS section.
 * __bss_end__: end of the BSS section.
 *
 * Both addresses must be aligned to 4 bytes boundary.
 */
    ldr    r1, __bss_start__
    ldr    r2, __bss_end__

    movs   r0, 0

    subs   r2, r1
    ble    .L_loop3_done

.L_loop3:
    subs   r2, #4
    str    r0, [r1, r2]
    bgt    .L_loop3
.L_loop3_done:
#endif /* __STARTUP_CLEAR_BSS_MULTIPLE || __STARTUP_CLEAR_BSS */

...

```

7.4.10. Секције .heap и .stack_dummy: динамичка меморија и стек у RAM-y

Након BSS, долазимо до дефиниције простора за динамичку алокацију (heap) и стека (stack). Ово у скрипти почиње са секцијом .heap означеном NOLOAD. Унутар ње видимо: на почетку `__HeapBase = .;` – симбол који означава почетак heap-а, постављен на текућу адресу (што је одмах након **bss_end**, јер до тада смо све претходно сместили у RAM). Затим `__end__ = .;` и `end = __end__;`. Овим се два симбола, `__end__` и `end`, постављају на исту ту адресу, дакле на почетак heap-а. Симбол `end` (без подвлаке) је традиционалан ознака која се користи у C runtime-y – стандардна имплементација системског позива `_sbrk` (који управља динамичком алокацијом у одсуству оперативног система) користи променљиву `end` као почетак неалоцираног дела меморије. У стандардном `libc-y`, `end` означава прву адресу иза свих статичких података, одакле алокаатор може да почне да додељује меморију за `malloc`. Зато је важно да је `end` правилно постављен. Наш линкер га везује за интерни `__end__` (који је, како видимо, управо постављен на `__HeapBase`). Практично, `__HeapBase`, `__end__` и `end` сада све указују на исту адресу – место где почиње heap. Затим `KEEP(*(.heap*))` укључује све улазне секције назване .heap ако их има. Обично програмер не креира ручно променљиве у секцији .heap; ово је ту за случај да runtime библиотека или неки специјалан код жели да резервише унапред неки део heap-а. Такође, у startup асемблерском коду за наш MCU, уочавамо да постоји дефинисана .heap секција која резервише одређен број бајтова ако је дефинисан `__HEAP_SIZE` макро. На пример, ако је `__HEAP_SIZE` подешен на `0x400`,

startup ће у .heap секцији .о фајла резервисати 1KB простора. Овде KEEP*(.heap*)) осигурава да тај резервисани блок уђе у излазну .heap секцију линкера. Ако није ништа резервисано, ова директива нема ефекат (неће бити садржаја, али .heap секција ионако нема шта да учитава). Након тога, веома је важно шта линкер ради: линија . = ORIGIN(ram) + LENGTH(ram) - STACK_SIZE; помера текућу локацију тј. локаунтер (.) на адресу која представља почетак региона за стек. Подсећања ради, у MEMORY делу скрипте дефинисан је ram регион са ORIGIN и LENGTH. Овде се од краја тог RAM региона одузима вредност STACK_SIZE (која је на почетку скрипте дефинисана као константа – у нашем случају STACK_SIZE = 0x1000;, тј. 4096 бајтова). Дакле, локаунтер се сада поставља на адресу (крај RAM-а - 4096). Та адреса ће бити почетак стека (стек расте надолу на ARM-у). Након овог померања, симбол __HeapLimit = .; се дефинише на том месту. Практично, __HeapLimit означава крај дозвољеног heap-а, тј. границу докле динамички алокаатор сме да заузима меморију. Он је постављен тачно на дно стек простора, што значи да heap сме да расте од __HeapBase па навише све до __HeapLimit, а stack ће расти од врха RAM-а па наниже до те исте границе. Овим моделом, сва расположива слободна меморија између статичких података и врха RAM-а подељена је између heap-а и стека. Уколико би алокације на heap-у прекорачиле __HeapLimit или стек нарастао испод те границе, дошло би до сукоба (преклапања). Тај сценарио се проверава касније assertion-ом.

```
/* linker_cm0p.ld */

...

    .heap (NOLOAD):
    {
        __HeapBase = .;
        __end__ = .;
        end = __end__;
        KEEP*(.heap*)
        . = ORIGIN(ram) + LENGTH(ram) - STACK_SIZE;
        __HeapLimit = .;
    } > ram

...
```

Вреди приметити да наш линкер скрипт двоструко обезбеђује простор за стек: једном преко ове манипулације .heap секцијом (померањем . за STACK_SIZE бајтова), а други пут преко следеће секције .stack_dummy. Ова секција је такође NOLOAD и дефинисана је као:

```
/* linker_cm0p.ld */

...

/* .stack_dummy section doesn't contains any symbols. It is only
 * used for linker to calculate size of stack sections, and assign
 * values to stack symbols later */
.stack_dummy (NOLOAD):
{
    KEEP*(.stack*)
} > ram

...
```


Она служи за израчунавање величине стека и постављање одговарајућих симбола, али сама неће постојати у уčitаном програму (њен тип је NOBITS). Овде KEEP(*(.stack*)) укључује све улазне .stack секције – а у нашем случају постоји једна таква из startup асемблерског кода. Наиме, у startup_tviibe4m_cm0plus.S фајлу видимо:

```
/* startup_tviibe4m_cm0plus.S */

...

    .section .stack
    .align    3
#ifdef __STACK_SIZE
    .equ      Stack_Size, __STACK_SIZE
#else
    .equ      Stack_Size, 0x00001000
#endif
    .globl    __StackTop
    .globl    __StackLimit
__StackLimit:
    .space    Stack_Size
    .size     __StackLimit, . - __StackLimit
__StackTop:
    .size     __StackTop, . - __StackTop

...
```

Асемблер је дакле направио секцију .stack у којој резервише Stack_Size бајтова (стандардно 0x1000, што кореспондира нашем STACK_SIZE од 4096) и означио два симбола: __StackLimit на почетку тог блока и __StackTop на крају тог блока (који ће бити на адреси __StackLimit + Stack_Size). При линковању, ова .stack секција из асемблерског модула биће спојена у нашу .stack_dummy излазну секцију (због matching имена). Тако .stack_dummy заправо заузима тачно величину која је потребна стеку (према дефиницији у асемблеру). Линкер неће ту секцију ставити у бинарни фајл (NOLOAD), али ће знати њену величину (SIZEOF(.stack_dummy)). То сада користимо: одмах након дефиниције .stack_dummy, у линкер скрипти постављамо:

```
/* linker_cm0p.ld */

...

/* Set stack top to end of RAM, and stack limit move down by
 * size of stack_dummy section */
__StackTop = ORIGIN(ram) + LENGTH(ram);
__StackLimit = __StackTop - SIZEOF(.stack_dummy);
PROVIDE(__stack = __StackTop);

...
```

__StackTop се дефинише као крај RAM-а (ORIGIN+LENGTH даје прву адресу изван RAM региона). То је заправо физички врх меморије, који ће бити коришћен као почетна вредност стек-показивача SP приликом ресета. Симбол __StackLimit се израчунава као __StackTop - SIZEOF(.stack_dummy), што је еквивалентно крај_RAM - Stack_Size. Пошто је SIZEOF(.stack_dummy) једнако броју бајтова које је асемблер резервисао за стек, __StackLimit ће пасти на адресу која означава "дно" стека (најнижу безбедну адресу стека). Другим речима, стек може да расте наниже од __StackTop до (али не преко)

__StackLimit. Симбол __stack се обезбеђује (PROVIDE) као алијас за __StackTop, за случај да негде у коду неко очекује симбол __stack (неке библиотеке или алати га реферишу као почетак стека). Сада можемо разумети зашто је линкер померио локаунтер (.) у .heap секцији за STACK_SIZE бајтова: тим померањем је ефективно оставио „рупу“ у меморији између __HeapLimit и краја RAM-а, величине STACK_SIZE. Затим је .stack_dummy секција (од асемблера) тачно испунила ту рупу резервисаним бафером за стек. Да није померена ., .stack_dummy би почела одмах на __HeapBase (крај .bss), што није жељено – желимо стек на врху меморије. Овако, комбинацијом оба механизма, и heap и stack су добили свој простор. Остаје још да се провери да се не преклапају. Последња линија у секцији је:

```
/* linker_cm0p.ld */  
  
...  
  
    /* Check if data + heap + stack exceeds RAM limit */  
    ASSERT(__StackLimit >= __HeapLimit, "region RAM overflowed with stack")  
  
...
```

Ова наредба каже линкеру да баца грешку ако услов __StackLimit >= __HeapLimit није испуњен, тј. ако је стек „прегазио“ heap. Убацавањем конкретних вредности, то значи да се захтева да је адреса дна стека већа или једнака адреси краја heap-а. Уколико имамо превише глобалних података или превелики STACK_SIZE, може се десити да __HeapLimit (дно стека) испадне испод __HeapBase (почетка heap-а). Тада би овај ASSERT реаговао и при линковању бисмо добили грешку "region RAM overflowed with stack". Ово је корисна провера интегритета расподеле меморије.

Тиме је завршен распоред главних секција програма у RAM меморији.

7.4.11. Специјални меморијски региони: Supervisory Flash, eFuse и cymeta

Преостале секције у скрипти односе се на специјалне меморијске регионе ван главног RAM-а и флеша, специфичне за PSoC 6/Traveo II архитектуру, познате као Supervisory Flash и eFuse. Меморијски регион "sflash" (Supervisory Flash) је посебан мали флеш сегмент који је обезбеђен за складиштење података као што су безбедносни кључеви, табеле садржаја (ТОС) и конфигурације заштите од дебаговања. Наш линкер скрипт дефинише неколико секција које се мапирају баш у те регионе, користећи називе меморија > sflash_user_data, > sflash_nar, итд.

```
/* linker_cm0p.ld */

...

/* Supervisory Flash: User data */
.cy_sflash_user_data :
{
    KEEP(*(.cy_sflash_user_data))
} > sflash_user_data

/* Supervisory Flash: Normal Access Restrictions (NAR) */
.cy_sflash_nar :
{
    KEEP(*(.cy_sflash_nar))
} > sflash_nar

/* Supervisory Flash: Public Key */
.cy_sflash_public_key :
{
    KEEP(*(.cy_sflash_public_key))
} > sflash_public_key

...
```

Прва секција је .cy_sflash_user_data, намењена за корисничке податке у supervisory flash-у. Њен почетак тј. ORIGIN је 0x17000800 (према MEMORY делу) а дужина 0x800 бајтова – тако је одређен простор од 2KB за податке које корисник може трајно сместити у овај посебан флеш (нпр. калибрациони подаци, конфигурације које треба да опстану и када се апликација поново програмира, јер ови делови флеша могу бити изузети од ре-програмирања).

Следећа је .cy_sflash_nar (Normal Access Restrictions) секција, смештена у регион sflash_nar (ORIGIN 0x17001A00, Length 0x200). Ових 512 бајтова обично држи подешавања за дебаг портове и сигурносне дозволе у различитим lifecycle стањима чипа (тзв. NAR регија садржи битове који контролишу да ли је Debug Access Port дозвољен у Normal, Secure, Dead режимима уређаја, као и да ли се може трајно закључати).

Затим `.cy_sflash_public_key` секција (ORIGIN 0x17005A00, Length 0xC00 што је 3KB). Овај простор је предвиђен за складиште јавног криптографског кључа који се користи за проверу потписа апликације при secure boot-у. Infineon-ова документација наводи да се RSA2048 јавни кључ (модулус од 256 бајтова, експонент и остали параметри) смешта управо у SFlash, и то као структура од ~724 бајта, тако да 3KB покрива довољно места за различите формате кључа (укључујући неколико копија, коефицијенте за бржу верификацију итд.). На пример, у коду може да постоји фајл `cy_ps_keystorage.c` и да је кориснички јавни кључ означен атрибутом `__attribute__((section(".cy_sflash_public_key")))` како би био лоциран у овом региону.

```
/* linker_cm0p.ld */

...

/* Supervisory Flash: Table of Content # 2 */
.cy_toc_part2 :
{
    KEEP(*(.cy_toc_part2))
} > sflash_toc_2

/* Supervisory Flash: Table of Content # 2 Copy */
.cy_rtoc_part2 :
{
    KEEP(*(.cy_rtoc_part2))
} > sflash_rtoc_2

...
```

Следе две секције: `.cy_toc_part2` и `.cy_rtoc_part2`, обе дужине 0x200 (512 бајтова), на адресама 0x17007C00 и 0x17007E00 респективно. TOC2 означава "Table of Contents #2" – структуру у supervisory flash-у коју BootROM користи да пронађе апликације и одреди параметре бутирања. Према документацији, TOC2 садржи магични број, показиваче на апликације (нпр. адресу главне апликације у флешу), величине, верзије, као и флегове за подешавања такта и дебаг прозора. Такође, често садржи флег који указује да ли да се врши верификација потписа главне апликације у NORMAL режиму, и слично. Уређај на ресету учита TOC2 из SFlash-а да би знао коју апликацију да покрене и како (нпр. да ли постоји друга апликација, bootloader, и сл.). Секција `.cy_toc_part2` је примарна TOC2 табела, док је `.cy_rtoc_part2` вероватно резервисана за резервну копију те табеле (редундантни TOC2) – "r" може да значи "redundant". У неким уређајима Cypress обезбеђује две копије TOC2 ради сигурности од корупције. Подаци у овим секцијама се обично не мењају често; попуњавају их алати при програмирању уређаја или посебан софтвер (нпр. `sumselftool` при креирању финалне слике уме да ажурира TOC2 записе). У сваком случају, линкер их резервише и KEEPER осигурава да ће било шта што смо означили тим секцијама остати у фајлу.

```
/* linker_cm0p.ld */  
  
...  
  
    /* eFuse */  
    .cy_efuse :  
    {  
        KEEP(*(.cy_efuse))  
    } > efuse  
  
...
```

Следи секција `.cy_efuse`, мапирана у регион `efuse` на адреси `0x90700000` (дужине `0x100000`, тј. 1MB). Ово је простор одређен за `eFuse` податке. Реално, `eFuse` (електрично програмабилни фјузи) нису меморија која се учитава у програм, али се може у `.hex` датотеци резервисати простор за њих или генерисати подаци који ће програмирати `eFuse`. Овде `KEEP(*(.cy_efuse))` секција омогућава да, ако постоји секција са тим именом (нпр. можда дефинишемо константе које желимо да упишемо у фјузе), она се издвоји у одвојени део `image`-а. Атрибут `> efuse` значи да ће те вредности бити мапирани на одговарајуће `eFuse` адресе. Чип вероватно не дозвољава читање/писање `eFuse` кроз стандардну меморијску мапу (обично се они програмски уписују преко специјалних команди), али ова секција може да послужи при генерисању датотека за програмирање (нпр. Cypress алати могу из `ELF`-а извући шта треба уписати у `eFuse`). У већини случајева, ова секција ће остати празна, осим ако корисник експлицитно не упише нешто.

```
/* linker_cm0p.ld */  
  
...  
  
    /* These sections are used for additional metadata (silicon revision,  
    * Silicon/JTAG ID, etc.) storage.  
    */  
    .cymeta          0x90500000 : { KEEP(*(.cymeta)) } :NONE  
} /* End of SECTIONS */  
  
...
```

На крају `SECTIONS` блока, видимо и секцију `.cymeta` која има специфичну синтаксу: `.cymeta 0x90500000 : { KEEP(*(.cymeta)) } :NONE`. Овим се секција `.cymeta` поставља на фиксну апсолутну адресу `0x9050_0000`, без придруживања неком дефинисаном меморијском региону (`:NONE` значи да линкер не проверава да ли је та адреса унутар дефинисаних `MEMORY` региона). Ова адреса је у опсегу `0x9050_0000–0x905F_FFFF`, који изгледа припада истом домену као и `eFuse` (мада наш регион `efuse` почиње тек од `0x9070_0000`). Могуће је да је `0x9050_0000` адреса у споредној меморији коју `BootROM` очекује да садржи неке податке о силицијуму. Коментар у скрипти каже да се ове секције користе за додатне метаподатке (ревизија силицијума, `JTAG ID` итд.). Дакле, `.cymeta` би могла да садржи структуру са информацијама о чипу, верзији, идентификаторима, које алат `sumsueelftool` или `bootloader` могу да очекују. Пошто је означена са `KEEP`, ако програм или алат дефинишу нешто у секцији `.cymeta`, то ће бити укључено на тачно то место у излазном `.hex/.elf`. Атрибут `:NONE` осигурава да линкер то неће покушати да

стави у неки од већ дефинисаних региона (који можда немају ту адресу). Ово је специфично за cymcuelftool употребу, иначе би линкер овакву дефиницију сматрао потенцијално проблематичном (ван домета меморије).

Након што су све секције дефинисане и затворена витичаста заграда, на крају скрипте видимо дефиниције неколико симбола који описују меморијске регионе, уз напомену "The following symbols used by the cymcuelftool.". Ово су три групе од по три симбола: за **Flash** (memory 0), **Supervisory Flash** (memory 2) и **eFuse** (memory 4).

```
/* linker_cm0p.ld */

...

/* The following symbols used by the cymcuelftool. */
/* Flash */
__cy_memory_0_start    = 0x10000000;
__cy_memory_0_length   = 0x00410000;
__cy_memory_0_row_size = 0x200;

/* Supervisory Flash */
__cy_memory_2_start    = 0x17000000;
__cy_memory_2_length   = 0x8000;
__cy_memory_2_row_size = 0x200;

/* eFuse */
__cy_memory_4_start    = 0x90700000;
__cy_memory_4_length   = 0x100000;
__cy_memory_4_row_size = 1;

/* EOF */
```

Ови симболи дају алату информацију о распону адреса и организацији различитих типова меморије на уређају. На пример, **Memory0 (главни флеш)** почиње на 0x10000000 и дугачак је 0x410000 бајтова. Row size од 0x200 указује да је флеш организован у програмске странице (flash rows) величине 512 бајтова. Програмска страница представља најмању јединицу уписа у флеш – док је читање могуће по бајтовима, упис се увек изводи над целом страницом поравнатом на границу од 512 бајтова. Ово је типична величина за Infineon PSoC 6/Traveo II архитектуру и одражава ограничења технологије флеш меморије. **Memory2 (SFlash)** почиње на 0x17000000, величине 0x8000 (32KB) са редом од 0x200 (опет 512 бајтова по страници) – овај регион обухвата цели supervisory flash. **Memory4 (eFuse)** је од 0x90700000, величине 0x100000 (1MB), а row_size=1, што значи да се еФјуз програмски адресира на нивоу бајта (нема страничне организације као флеш). Алат **СyMCUElfTool** користи ове симболе да би знао где се који тип меморије налази и колико је велик, приликом израде криптографског потписа или .cyacd2 датотека за ОТА надоградње. Конкретно, при генерисању дигиталног потписа, потребно је знати опсег флеша који се потписује – cymcuelftool користи симболе __cy_memory_0_start и __cy_memory_0_length (или алтернативно __cy_app_verify_start/length) да утврди од ког до ког бајта ELF слике треба израчунати signature (тј. које делове софтвера штитимо потписом). У нашем случају није експлицитно дат __cy_app_verify_start, али можемо претпоставити да је еквивалентан __cy_memory_0_start (0x10000000 – почетак апликације) а __cy_app_verify_length би био стварна дужина корисне апликације. Осим тога, ови симболи служе и за потребе patch-

овања – алат може на пример додати CRC на крај дефинисаног memory региона. Заиста, Infineon упутство каже: "Define the range of Flash memory to be patched using the `__cy_memory_0_XXXX` sections in your linker script...", након чега се позива `symcuelftool` да генерише CRC или signature и стави га на предвиђено место (нпр. у `.cy_app_signature` секцију).

Треба напоменути и да у овом линкер скрипту нема директно дефинисане секције `.cy_app_signature`, али у упутствима Infineon-а се види да се она обично дефинише на крају flash региона – нпр. на адреси `FLASH_END - 0x100` или слично, где се оставља место за 4 бајта CRC-а или 256 бајтова потписа. Када се `symcuelftool` алат покрене са опцијом за CRC или RSA потпис, он ће пронаћи `.cy_app_signature` секцију уколико постоји или је сам креирати, уписати вредност потписа, и прилагодити TOC2 записе у SFlash-у да означе да је апликација аутентична.

7.4.12. Закључак о SECTIONS расподели

На крају, `/* EOF */` коментар означава крај линкер скрипте (End of File). Овај SECTIONS одељак је врло детаљно распоредио све делове извршног програма: од заглавља апликације, преко кода, константи, C++ конструктора/деструктора, па до различитих табела за startup, и на крају свих врста података у RAM-у (иницијализованих, неиницијализованих, heap/stack) и специјалних регија. Сваку линију смо анализирали: видели смо како се линкерским симболима повезују сегменти са стартап кодом – на пример, `__Vectors` и повезани симболи за постављање векторске таблице, табеле `__copy_table_*` и `__zero_table_*` које `Reset_Handler` рутина користи за копирање/чишћење секција, као и симболи `__StackTop`, `__StackLimit`, `__HeapBase`, `__HeapLimit` који обезбеђују правилну иницијализацију стека и динамичке алокације. Овако генерисан извршни ELF фајл може успешно да прође кроз алат `symcuelftool` ради додавања потписа – алат ће препознати све потребне симболе/секције, уписати signature у предвиђену секцију и ажурирати TOC2 запис са величином и потписом апликације, након чега систем може да бутује уз криптографску проверу интегритета. На тај начин, овај линкер скрипт повезује све делове процеса – од генерисања бинарне слике па до механизма secure boot-а и расподеле меморије микроконтролера – у једну конзистентну целину.

7.5. Закључак о линкер скрипти

Она повезује свет С кода са физичком меморијом хардвера. MEMORY секција описује *где* може шта да иде, а SECTIONS секција *како* да се садржај распореди. За типичан Cortex-M пројекат, већина програмера користи унапред припремљену скрипту (било од произвођача или генерисану алатом), али разумевање исте је кључно при решавању проблема као што су: преливање меморије, смештање специфичних функција у одређени регион (нпр. у посебан сегмент који се може ажурирати независно), прављење боотлоадер/апликација поделе, итд.

8. Закључак

9. Литература