



**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:		Потпис ментора
	Члан:		
	Члан, ментор:	др Дарко Марчетић, ред. проф., ФТН Нови Сад	

Образац Q2.НА.06-05 - Издање 1



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:

<p>Циљ овог 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 <i>hex</i> или <i>bin</i> 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.</p>
--

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

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

Садржај

1. Увод.....	6
1.1. Контекст и значај теме.....	6
1.2. Циљ рада.....	6
1.3. Методологија	7
1.4. Објашњење термина и скраћеница	7
1.4.1. Хардвер и архитектура.....	7
1.4.2. Основни појмови софтверског система	8
1.4.3. Софтверски слојеви и библиотеке	8
1.4.4. Компилациони алати и процеси.....	8
1.4.5. Формати излазних датотека	9
1.5. Релевантност теме.....	9
2. Улога C језика у програмирању микроконтролера	11
2.1. Историјска еволуција C језика у embedded окружењима.....	11
2.2. Кључне особине C језика у embedded контексту	12
2.3. Упоредна анализа C језика и алтернативних језика	12
3. Организација C изворног кода за embedded окружења.....	13
3.1. Основне компоненте embedded пројекта	14
3.1.1. main.c (улазна тачка програма)	14
3.1.2. Модули и драјвери (.c/.h парови)	15
3.1.3. startup.s (векторска табела, Reset рутина, итд.).....	16
3.1.4. linker.ld (меморијско мапирање)	17
3.1.5. system_*.c (иницијализација такта, PLL, напајања).....	18
3.1.6. Makefile (компилација и линковање)	19
3.2. Употреба CMSIS и HAL слојева.....	21
3.2.1. CMSIS (Cortex Microcontroller Software Interface Standard)	21
3.2.2. HAL (Hardware Abstraction Layer)	22
3.3. Стил програмирања и стандардизација.....	26
3.3.1. Индустијски стандарди кодирања за безбедност и поузданост.....	26
3.3.2. Конзистентност стила и одрживост кода	27
3.4. Приступ меморијски мапираним регистрима.....	28
3.4.1. Меморијски мапирани улази и излази у микроконтролерима.....	28
3.4.2. Адресни простор и распоред периферија	28
3.4.3. Приступ регистрима у програму (C језик).....	29
3.4.4. Моделирање хардверских регистра у C	29
3.4.5. Предности и значај апстракције регистра.....	30
4. Фазе компилације (превођења) у GCC	31
4.1. Препроцесирање.....	32
4.2. Компилација	33
4.3. Асемблирање.....	34
4.4. Линковање.....	35
4.5. Интеграција фаза компилације у оквиру GCC алатке	36
5. Формати резултујућих датотека	37
5.1. ELF формат	38
5.2. Intel HEX формат	39

5.3. RAW бинарни формат (.bin).....	40
5.4. Motorola S-Record формат (S19)	41
6. Практична анализа GNU binutils алата за обраду објектних и бинарних датотека	42
6.1. Алат objcopy за креирање HEX, BIN и S19 датотека	44
6.2. Алат readelf за испитивање интерне структуре ELF фајлова.....	47
6.3. Алат nm за преглед и класификацију симбола у бинарним датотекама	48
6.4. Алат size за анализу меморијске потрошње по секцијама.....	49
6.5. Алат objdump као средство за дубинску анализу ELF структуре	50
6.6. Комплементарна употреба GNU алата у embedded развоју	51
7. Линкерска скрипта.....	52
7.1. Пример линкер скрипте за GNU C компајлер	53
7.2. Почетне директиве и дефиниције симбола	61
7.3. MEMORY дефиниција	66
7.4. SECTIONS расподела	68
7.5. Закључак о линкер скрипти	75
8. Компилација и меморијски распоред за Infineon TRAVEO T2G....	76
9. Закључак.....	77
10. Литература	78

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 учитавање 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** – Текстурална 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 текстурални формат сличан Intel HEX-у. Користи S0–S9 записе који садрже адресу, дужину, податке и контролни збир. Широко подржан у индустријским програматорима и `legasy` 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-а.

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-a, 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 нестандардно.

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 конфигураисање периферијских модула који ће те прекиде користити. У примеру, након што је тајмер конфигурисан 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 регистара (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 корака: **препроцесирање**, **компилацију** (у ужем смислу), **асемблирање** и **линковање**, тим редоследом. Свака фаза има своју улогу у претварању изворног **.c** кода у извршну бинарну датотеку. Следи преглед ових фаза у табели 1.

Табела 1. Фазе превођења C програма уз GCC

Фаза	Алат (GCC позив)	Улаз	Израз
Препроцесирање	<code>gcc -E</code>	*.c, *.h (изворник)	Препроцесирани код (*.i)
Компилација (C->ASM)	<code>gcc -S</code>	*.i (из претходног)	Асемблерски код (*.s)
Асемблирање	<code>gcc -c</code> или <code>as</code>	*.s (асемблерски код)	Објектни фајл (*.o)
Линковање	<code>gcc</code> или <code>ld</code>	*.o (+ библиотеке)	Извршна датотека (ELF)

4.1. Препроцесирање

Препроцесирање је прва фаза у којој се извршава С препроцесор. Он обрађује директиве које почињу знаком `#` – на пример, убацује садржај хедер датотека на место `#include` директива, проширује макрое дефинисане са `#define`, и условно уклања/укључује делове кода на основу `#ifdef` услова. Резултат ове фазе је *препроцесирани исходни код*, типично са екстензијом `.i` (или `.ii` за C++) који више не садржи препроцесорске директиве, већ само “чист” C код.

4.2. Компилација

У фази *компилације* (у ужем смислу) GCC преводи препроцесирани С код у *асемблерски код* за циљну архитектуру. То значи да се синтакса и конструкције С језика преводе у низ асемблерских инструкција (нпр. ARM Cortex-M7 инструкције) које остварују еквивалентну функционалност. Излаз из ове фазе је .s датотека (асемблерски код у текстуалном облику). Ова фаза укључује и различите оптимизације које компајлер примењује (према подешеним опцијама, нпр. -O2) како би генерисани код био што ефикаснији.

4.3. Асемблирање

Генерисани .s асемблерски код затим пролази кроз асемблер (саставни део GCC алата, нпр. arm-none-eabi-as), који га претвара у *релокативни објектни фајл* – машински код са нерешеним релокацијама и симболима. Ова датотека обично има екстензију .o и у формату је објектне датотеке (најчешће ELF формат, о чему ће бити речи касније). Објектни фајл садржи машинске инструкције за дати модул, али још увек није самостално извршна целина, јер адресе функција и података који се налазе у другим модулима нису још познате (остају као симболи које треба повезати).

4.4. Линковање

Последња фаза је позив линкера (нпр. GNU ld) који узима један или више објектних фајлова (.o), као и евентуално предефинисане библиотеке (нпр. libc, или драјверске библиотеке), и **повезује** их у јединствену извршну датотеку. Линкер разрешава све међусобне референце – нпр. када функција у main.o зове функцију која је имплементирана у uart.o, линкер ће уписати исправну адресу те функције у машински код позива. Такође, линкер припаја и стандардни стартуп код (нпр. **crt0** за C) ако је део toolchain-а, мада у embedded окружењу стартуп и векторска табела обично долазе као засебан модул пројекта. Резултат линковања је извршна бинарна датотека у *ELF формату* (или сличном), са свим спојеним секцијама на одговарајућим меморијским адресама. Ова датотека је сада самосталан програм који се може учитати у меморију микроконтролера и покренути.

4.5. Интеграција фаза компилације у оквиру GCC алатке

Напоменимо да се у пракси већина ових корака обавља "у пролазу" помоћу исте **gcc** наредбе, јер GCC аутоматски позива препроцесор, па компајлер, асемблер и линкер. На пример, позив

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

ће обавити све кораке и произвести коначни program.elf. Ипак, корисно је разумети ове међукоре, јер алати омогућавају да се сваки корак изведе одвојено (нпр. опција -save-temps чува привремене .i и .s датотеке). GCC документација наглашава постојање наведене четири фазе и одговарајуће суфиксе/екстензије фајлова. Током компилације могу настати и помоћни фајлови као што је *листинг* (са мешовитим C кодом и асемблером, ако је затражено), али они нису нужни у даљем процесу.

Важно је истаћи да линкер за успешно повезивање за *embedded* мету мора знати распоред меморије циљног микроконтролера – ту ступа на снагу *линкерска скрипта* која описује меморијске регије (Flash, RAM) и како распоредити секције програма у њих. Линкерска скрипта је критична за добијање исправне бинарне слике програма и њена структура биће детаљно анализирана у посебном одељку.

5. Формати резултујућих датотека

Финална фаза процеса превођења C 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 спојени током линковања, организовани су у јединствену бинарну слику чија је структура дефинисана изабраним форматом. Избор формата резултујуће датотеке није произвољан – он зависи од карактеристика циљне архитектуре, доступних алата за програмирање (programmers/debuggers), као и од захтева процеса производње и одржавања уграђеног система. У пракси, у области embedded програмирања, најчешће се користе ELF (Executable and Linkable Format) као интермедијарни и вишенаменски формат, те његове конверзије у једноставније, „чисте“ формате погодне за флешовање, као што су Intel HEX, RAW бинарни (.bin) и Motorola S-Record (S19). Сваки од ових формата има специфичну намену, структуру и предности у одређеним сценаријима – од задржавања симболичких и дебаг информација, до минималистичког представљања података за директно уписивање у меморију. Разумевање њихових карактеристика и међусобних разлика представља предуслов за правилно интегрисање компилационог процеса са поступцима програмирања микроконтролера у реалним системима.

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 cross-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.

5.2. 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 (*alati* 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.

5.3. 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 објсору 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 објсору 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 размену фирмвер слика (нпр. HEX за надоградњу софтвера у сервису, или BIN за брзо читавање преко bootloader-а), важно је разумети њихову структуру и разлике.

5.4. 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). Пошто је у питању алтернативни формат, нећемо детаљно разматрати његову структуру, али вреди споменути да алат **srec_cat** (део SRecord пакета) може манипулисати и HEX и S19 фајловима.

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

При раду на озбиљним embedded пројектима, корисно је познавати алате за анализу и конверзију објектних и извршних датотека. GCC toolchain долази са низом **GNU binutils** алата који омогућавају увид у ELF садржај, конверзију формата, дисасемблирање, мерење величине и друго. У наставку су описани најважнији алати и њихова примена у контексту програмирања микроконтролера.

„GNU binutils алати“ је формално исправан и технички прецизан појам:

- обухвата све алате као што су objcopy, readelf, nm, size, objdump...
- независно од архитектуре или toolchain-a, сви су део истог пакета: GNU Binary Utilities

Упркос томе што користиш ARM специфичну варијанту (arm-none-eabi-objcopy), **то је и даље конкретна инстанца GNU binutils алата.**

У свим примерима анализе у овом поглављу користе се GNU binutils алати специфични за ARM архитектуру, инсталирани као део arm-none-eabi toolchain-a.

У embedded развоју није свеједно да ли се ради о objcopy за x86 или arm-none-eabi-objcopy за Cortex-M;

С обзиром на то да је у питању **embedded систем заснован на ARM Cortex-M архитектури** и да се у пракси користи **крос-компајлерски toolchain**: arm-none-eabi-gcc и пратећи binutils алати (са префиксом arm-none-eabi-).

Када развијаш за архитектуру **различиту од хост машине**, потребно је користити алате са префиксом:

Архитектура	Префикс алата
ARM Cortex-M	arm-none-eabi-*
ARM Cortex-A (Linux)	aarch64-none-linux-gnu-*
RISC-V	riscv64-unknown-elf-*
MIPS	mipsel-none-elf-*
PowerPC	powerpc-eabi-*

Ови алати су део **крос-toolchain-a**, а *binutils* део (*objdump*, *objcopy*, *readelf*...) мора бити компајлиран да подржава *target* архитектуру.

Дипломски рад обрађује **ARM Cortex-M4 и Cortex-M0+** микроконтролер као пример, дакле:

- Мора се користити **arm-none-eabi-objcopy**, **arm-none-eabi-nm**, итд.
- „Обични“ *objcopy* и остали алати **неће исправно радити**, осим ако је систем на ARM архитектури (што није случај код твоје Windows/Linux машине)

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

Ово је алат за копирање и конверзију формата објектних датотека. Користи се за генерисање HEX или BIN фајла из ELF-а. Примери:

objcopy -O ihex program.elf program.hex (створи Intel HEX из ELF-а),

objcopy -O binary program.elf program.bin (створи raw бинарни дамп).

Овај алат може издвајати и поједине секције, уклањати debug информације опцијом -S, итд. Базиран је на BFD библиотеци, тако да подржава бројне формате и аутоматски препознаје улазни и излазни формат. Треба бити опрезан: при конверзији у .bin, ако ELF садржи "празнину" (нпр. секције које нису континуалне у меморији), objcopy ће ту празнину испунити нулама у .bin фајлу (јер прави континуални дамп од најнижег до највишег адресног бајта). Стога .bin може бити већи него износ корисног кода ако постоје велике неиницијализоване секције на високим адресама.

Кључна разлика између „генеричког“ objcopy и крос-компајлерског arm-none-eabi-objcopy:

- **1. arm-none-eabi-objcopy**
- то је варијанта GNU objcopy алата из **ARM GCC toolchain-a**;
- подешен је тако да подразумевано зна за **ARM ELF формате** (little-endian, bare-metal, EABI);
- без додатних параметара тачно разуме program.elf који је генерисао arm-none-eabi-gcc.

Пример (ради одмах):

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

- **2. „Генерички“ objcopy (нпр. из MSYS2 MinGW-a)**
- такође је GNU objcopy, али је компајлиран за **x86_64-pc-mingw32** (Windows окружење);
- не зна сам од себе да је улаз ARM ELF, па може да пријави грешку типа *“file format not recognized”* или да направи неисправан излаз;
- да би радио, мораш експлицитно да наведеш формат улаза и излаза, нпр:

objcopy -I elf32-littlearm -O binary program.elf program.bin

Пошто дипломски рад користи пример на ARM архитектури, онда је потребно користити `arm-none-eabi-objcopy` уместо `objcopy`.

Алат **GNU objcopy** је део *binutils* пакета и служи за манипулацију бинарним датотекама, нарочито ELF излазом компајлера. Његова суштинска намена у `embedded` окружењу је да из *ELF извршне датотеке* изведе различите алтернативне формате попут **Intel HEX (.hex)**, **RAW бинарног (.bin)** или **Motorola S-Record (.s19)** формата, који су погодни за програмирање микроконтролера. Поред конверзије, `objcopy` може да избаци, промени или модификује одређене секције, да релокације претвори у апсолутне адресе, да обрише табеле симбола или да селективно издвоји сегменте кода.

1. Основне могућности алата `objcopy`

Неке од најчешћих команди (покрећу се у командној линији) су:

2. Конверзија ELF у Intel HEX

```
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. Издвајање само једне 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 (минимизација величине)

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

7. Спајање или премештање 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 (32/64-bit, endian)

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


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

Алат **readelf** служи за читање ELF датотека – даје детаљан увид у ELF заглавља, секције, сегменте, симболе, итд. На пример, *readelf -h program.elf* приказује опште заглавље (тип, машина, ендијаност, улазну тачку...), *readelf -S program.elf* листа секције (са именима, величинама, стартним адресама у фајлу и у меморији), а *readelf -s program.elf* листа симбол таблицу (са именима функција/променљивих и њиховим адресама или офсетовима). У анализи линкер скрипте и меморијског распореда, *readelf -S* је посебно користан да видимо где су .text, .data, .bss и друге секције смештене и колике су.

Пример: излаз *readelf -S* може показати да је .text секција величине, 0x1000 бајтова на адреси 0x10000000 (у флешу), .data величине 0x100 бајтова на адреси 0x08040000 (RAM), са LMA (Load Memory Address) у флешу, итд. Ово нам јасно говори распоред по регијама.

6.3. Алат `arm-none-eabi-nm` за преглед и класификацију симбола у бинарним датотекама

Овај алат служи за листање симбола (из објектних или извршних фајлова). `nm program.elf` ће исцртати листу свих симбола (функција, глобалних променљивих) које постоје у програму, са њиховим адресама и ознаком типа (Т=текст/функција, D=иницијализовани податак, В=неиницијализовани податак `bss`, и сл.). Ово је корисно кад желимо да знамо на којој адреси се налази одређена функција или променљива након линковања. На пример, можемо проверити да ли је глобална променљива мапирана у RAM (биће означена са В или D и имаће адресу у опсегу RAM меморије), или да ли је функција у флешу (ознака Т са адресом у опсегу флеша). `nm` је посебно драгоцен за грубу проверу исправности линкер скрипте – нпр. да ли симбол `_estack` (врх стека) има очекивану вредност, да ли су неки симболи преклопљени итд.

6.4. Алат `arm-none-eabi-size` за анализу меморијске потрошње по секцијама

Овај алат приказује резиме величина секција у извршној датотеци. Обично се позива као `size program.elf` и избацује три колоне: величину `.text` (код + константни подаци у флешу), `.data` (иницијални подаци који ће бити учитани у RAM) и `.bss` (неиницијализовани подаци који ће заузети RAM), као и укупан збир. Ово је врло прегледно да се види колики је „отисак“ програма у флешу и RAM-у. На пример, output може бити:

text	data	bss	dec	hex	filename
4528	128	256	4912	1330	program.elf

што значи да код заузима ~4528 бајтова флеша, статички иницијализовани подаци 128 бајтова RAM-а (плус још толико у флешу за њихове почетне вредности), а `.bss` (нпр. глобалне променљиве иницијализоване на 0) заузимају 256 бајтова RAM-а. Ови бројеви су битни у планирању да ли програм стаје у меморију микроконтролера. (Напомена: `size` уз опцију `-A` или `--format=SysV` даје детаљнији приказ по именованим секцијама.)

6.5. Алат arm-none-eabi-objdump као средство за дубинску анализу ELF структуре

objdump је многофункционални алат за испис садржаја објектних датотека. Може да прикаже хексадецимални *dump* секција (*objdump -s program.elf*), али најкориснија функционалност је **дисасемблирање** машинског кода у читљив асемблер.

Наредба *objdump -d -M reg-names-std program.elf* произвешће асемблерски листинг свих секција које имају код (нпр. `.text`, `.init`) са људски читљивим именима регистара. Ово је изузетно корисно за дебаговање ниског нивоа – можемо видети тачно које је инструкције компајлер генерисао из нашег C кода, што помаже у оптимизацији или трагању за баговима на ниском нивоу.

objdump -t program.elf исписује таблицу симбола (слично *nm*).

objdump -x program.elf исписује пуна ELF заглавља, секције, сегменте, симболе (комбинација информација, мање читљива од специјализованих алата попут *readelf* или *nm*).

Углавном, *objdump* је згодан за брзи увид у садржај бинарног кода – било у хекс или у асемблерском облику.

6.6. Комплементарна употреба GNU алата у embedded развоју

Набројани алати покривају најважније аспекте: конверзију формата (*objcopy*, *srec_cat*), анализу садржаја (*readelf*, *nm*, *objdump*) и величину (*size*). У типичном развојном циклусу, након добијања *program.elf*, програмер може покренути *size* да провери заузеће меморије, *objdump -d* ако сумња у неку оптимизацију компајлера, или *nm* да пронађе адресу битне променљиве за дебаговање. При припреми HEX-а за програмирање, користи се *objcopy* или *srec_cat*. На тај начин, ови алати представљају продужетак функционалности самог компајлера и линкера, дајући увид "испод хаубе" готовог програма.

7. Линкерска скрипта

Линкерска скрипта или линкер директива (LD фајл – linker.ld) је суштински део embedded пројекта – она одређује начин распоређивања секција програма у физичку меморију микроконтролера током линковања. Она повезује свет С кода са конкретним Flash/RAM адресним простором хардвера, обезбеђујући да сваки део извршног кода и података буде на предвиђеној адреси у меморији. У примеру пројекта за микроконтролер CYT2BL5CAS (Infineon Traveo II, KIT_T2G-B-E_LITE), линкерска скрипта је класичног обрасца за Cortex-M систем са четири језгра (четворојезгарни систем). Састоји се из три дела: (1) почетне глобалне директиве и дефиниција симбола, (2) секције MEMORY са описом расположивих меморијских регија (Flash, RAM, специјални Flash сегменти), и (3) секције SECTIONS која прописује смештај сваке програмске секције (.text, .data, .bss, стек, хип и др.) у одговарајуће меморијске регије. У наставку се систематски анализира свака компонента ове скрипте.

7.1. Пример линкер скрипте за GNU C компајлер

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 система. На основу тога, у блоку `SECTIONS` врши се расподела преведених програмских секција у одговарајуће меморијске регионе. У нашем примеру, скрипта најпре одређује да ће резултујући извршни фајл бити ELF за 32-битни ARM у little-endian формату, а као улазну тачку програма поставља рутину за ресет (симбол `Reset_Handler`). Потом се задају константе за величине меморијских резерви (нпр. `STACK_SIZE`) и израчунавају изведени адресни параметри. Блок `MEMORY` именује главне регије: интерну Flash меморију (за програмски код) и интерну SRAM (за податке и стек), уз додатне специјалне сегменте (supervisory flash за кључеве, eFuse и сл.). Коначно, у блоку `SECTIONS` прецизира се распоред ELF секција: векторска табела прекида и извршни код смештени су у Flash, иницијализовани подаци (.data) су предвиђени у RAM (али са копијом иницијалних вредности у Flash-у), неиницијализовани подаци (.bss) такође у RAM (без заузимања места у Flash-у), док су стек и хип секције позициониране на крају RAM меморије. На овај начин се остварује статичко меморијско мапирање целог програма, које је основа за правилно покретање и рад микроконтролера у *bare-metal* окружењу.

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

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. */
    sflash_user_data (rx) : ORIGIN = 0x17000800, LENGTH = 0x800 /* Supervisory flash: User data
    */
    sflash_nar      (rx)  : ORIGIN = 0x17001A00, LENGTH = 0x200 /* Supervisory flash: Normal
    Access Restrictions (NAR) */
    sflash_public_key (rx) : ORIGIN = 0x17005A00, LENGTH = 0xC00 /* Supervisory flash: Public
    Key */
    sflash_toc_2     (rx)  : ORIGIN = 0x17007C00, LENGTH = 0x200 /* Supervisory flash: Table of
    Content # 2 */
    sflash_rtoc_2    (rx)  : ORIGIN = 0x17007E00, LENGTH = 0x200 /* Supervisory flash: Table of
    Content # 2 Copy */
    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_tvii4m_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;
sram_private_for_srom   = 0x00000800; /* Private SRAM for SROM (e.g. API processing).
Reserved at the beginning */

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 приказани 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")

одређује **BFD** формат излазног објекта који ће линкер генерисати. **BFD** је скраћеница од *Binary File Descriptor* — то је унутрашњи апстракциони интерфејс GNU ld линкера и сродних алата (objcopy, readelf, nm и др.), који омогућава подршку за различите бинарне формате на транспарентан начин, без потребе да се мења логика алата при раду са ELF, COFF, a.out и другим форматима.

Унутар директиве OUTPUT_FORMAT наводе се **три параметра**:

1. Први параметар је **подразумевани формат** који ће се користити током генерисања излазне датотеке;
2. Други је **алтернативни формат** који се може употребити уколико се током процеса обраде појави потреба за променом ендјанског распореда (нпр. big-endian варијанта);
3. Трећи параметар је **формат који ће се користити у unpexes операцијама**, односно када се врши снимање тренутног стања меморије у извршну датотеку.

У типичним сценаријима за микроконтролере који користе ARM архитектуру у *little-endian* режиму, **сви параметри** се могу поставити на идентичну вредност — као у примеру elf32-littlearm — јер нема реалне потребе за подршком другим варијантама формата. Међутим, синтакса GNU ld линкера подразумева **експлицитно задавање сва три параметра**, чак и када су идентични, ради очувања униформности и избегавања неодређеног понашања у случајевима када је unpexes потребан или када алат интерно промени режим обраде.

Разлог зашто су **први и трећи параметар исти** лежи у чињеници да се у embedded окружењима unpexes готово никада не користи, па је најбезбедније и најконзистентније рециклаирати вредност подразумеваног формата (elf32-littlearm). То значи да ће и током нормалне компилације и током евентуалног снимања меморије (ако се уопште деси) линкер произвести датотеку истог формата.

Израз **unpexes** (од *undo exes*) односи се на снимање садржаја меморије у тренутном стању у нову извршну датотеку. Ова техника се ређе користи у embedded контексту, али је подржана од стране GNU алата ради компатибилности са другим окружењима (на пример, GNU Emacs користи unpexes приликом израде snapshot-а унапред учитаних модула). У тим случајевима, потребно је да линкер зна у ком формату да произведе излаз и тада користи **трећи параметар** OUTPUT_FORMAT директиве.

Дакле, иако су у овом конкретном примеру сви аргументи једнаки (elf32-littlearm), њихово присуство одражава интерну логику GNU линкера и осигурава стабилност и предвидивост понашања у свим фазама линковања. За системског програмера, препорука је увек да се **експлицитно наведу сва три параметра**, чак и када су идентични, ради пуне контроле над понашањем линкера.

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 „но-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 разреши.

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 етикете 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 *SROM* рутина система. Traveo T2G микроконтролери имају интерни *System ROM* (SROM) 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 почетног *ресерва* (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 М4 програма приликом репрограмирања М0+ 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 М0+ завршава на граници сектора, како би се избегло делимично брисање почетка М4 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 библиотеке, (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 SRAM и М0+ 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

У скрипти се најпре дефинишу именовани *меморијски региони*. Пример за неки Cortex-M7 микроконтролер може бити:

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

sflash_nar	(rx)	: ORIGIN = 0x17001A00, LENGTH = 0x200	/* Supervisory flash: Normal Access Restrictions (NAR) */
sflash_public_key	(rx)	: ORIGIN = 0x17005A00, LENGTH = 0xC00	/* Supervisory flash: Public Key */
sflash_toc_2	(rx)	: ORIGIN = 0x17007C00, LENGTH = 0x200	/* Supervisory flash: Table of Content # 2 */
sflash_rtoc_2	(rx)	: ORIGIN = 0x17007E00, LENGTH = 0x200	/* Supervisory flash: Table of Content # 2 Copy */
efuse	(r)	: ORIGIN = 0x90700000, LENGTH = 0x100000	/* 1 MB */
}			

7.4. SECTIONS расподела

```
/* 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 */
```

7.5. 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. Компилација и меморијски распоред за Infineon TRAVEO T2G

9. Закључак

10. Литература