УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Димитрије Ћук

Од изворног C 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 публиковања, **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**:		С обзиром да у микроконтролеру не постоји оперативни систем који управља меморијом, потребно је конфигурисати линкер кроз линкер директиву, односно линкер скрипту, на начин да уважи физичку расподелу меморије по врстама и адресама, на основу техничког листа, тј. спецификације коју произвођач доставља. Дигитално потписивање је процес који осигурава аутентичност и интегритет података коришћењем криптографије. Користи се асиметрична енкрипција која подразумева постојање приватног и јавног кључа.
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 Стеван Цветићанин, ван. проф., ФТН Нови Сад
	Члан:	
	Члан:	
	Члан, ментор:	др Дарко Марчетић, ред. проф., ФТН Нови Сад

Потпис ментора

Образац **Q2.НА.06-05** - Издање 1

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**

21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual Printed Material
Contents code, **CC**:	Bachelor thesis
Author, **AU**:	Dimitrije Ćuk
Mentor, **MN**:	prof. dr Darko Marčetić
Title, **TI**:	Memory Mapping in a Microcontroller and the Use of Digital Signatures
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	AP of Vojvodina
Publication year, **PY**:	2024.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Faculty of technical sciences, 21000 Novi Sad, Trg Dositeja Obradovića 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes	(9/72/15/2/35/0/0)
Scientific field, **SF**:	Electrical and computer engineering
Scientific discipline, **SD**:	Computer science and embedded systems
Subject/Key words, **S**/**KW**:	From Source C Code to Executable Binary Image – Compilation and Memory Layout in Microcontrollers
UC	
Holding data, **HD**:	Library of Faculty of technical sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, **N**:	
Abstract, **AB**:	Since there is no operating system in a microcontroller to manage memory, it is necessary to configure the linker through a linker directive or linker script in such a way that it takes into account the physical distribution of memory by type and address, based on the datasheet or specification provided by the manufacturer. Digital signing is a process that ensures the authenticity and integrity of data through cryptography. It uses asymmetric encryption, which involves the existence of both a private and a public key.
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 Stevan Cvetićanin, assoc. prof., FTN Novi Sad	
	Member:		Menthor's sign
	Member:		
	Member, mentor:	dr Darko Marčetić, full prof., FTN Novi Sad	

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број: 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:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ ДИПЛОМСКОГ РАДА:

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

ТЕКСТ ЗАДАТКА:

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

Примерак за: o - Студента; o - Ментора

Садржај

1. Увод 6
1.1. Циљ и мотивација рада 6
1.2. Методологија и приступ 6
1.3. Релевантност теме 6
2. Улога C језика у програмирању микроконтролера 7
2.1. Историјска еволуција C језика у embedded окружењима 7
2.2. Кључне особине C језика у embedded контексту 8
2.3. Упоредна анализа C језика и алтернативних језика 8
3. Организација C изворног кода за embedded окружења 9
3.1. Основне компоненте embedded пројекта 10
3.1.1. main.c (улазна тачка програма) 10
3.1.2. Модули и драјвери (.c/.h парови) 11
3.1.3. startup.s (векторска табела, *Reset* рутина, итд.) 12
3.1.4. linker.ld (меморијско мапирање) 13
3.1.5. system_*.c (иницијализација такта, PLL, напајања) 14
3.1.6. Makefile (компилација и линковање) 15
3.2. Употреба CMSIS и HAL слојева 17
3.2.1. CMSIS (Cortex Microcontroller Software Interface Standard) 17
3.2.2. HAL (Hardware Abstraction Layer) 18
3.3. Стил програмирања и стандардизација 22
3.3.1. Индустријски стандарди кодирања за безбедност и поузданост 22
3.3.2. Конзистентност стила и одрживост кода 23
3.4. Приступ меморијски мапираним регистрима 24
3.5. Пример за Infineon TRAVEO™ T2G 26
4. Фазе компилације (превођења) у GCC 27
4.1. Препроцесирање 28
4.2. Компилација 29
4.3. Асемблирање 30
4.4. Линковање 31
4.5. Интеграција фаза компилације у оквиру GCC алатке 32
5. Формати резултујућих датотека 33
5.1. ELF формат 34
5.2. Intel HEX формат 35
5.3. RAW бинарни формат (.bin) 36
6. Употреба GNU алата: objcopy, readelf, nm, size, objdump 38
6.1. objcopy 39
6.2. readelf 40
6.3. nm 41
6.4. size 42
6.5. objdump 43
6.6. Закључак 44
7. Линкерска скрипта: MEMORY и SECTIONS дефиниције 45
7.1. MEMORY дефиниција 46
7.2. SECTIONS расподела 47

8. Закључак..... 49

1. Увод

Програмски језик C доминира у развоју софтвера за микроконтролере захваљујући својој ефикасности и близини хардверу. Овај рад обрађује процес развоја једног C програма за 32-битни аутомобилски микроконтролер TRAVEO™ T2G (породица CYT3BB/4BB) заснован на ARM Cortex-M7 архитектури. Кроз логичку целину, рад покрива фазе писања и организације изворног кода, преко превођења (компилације) и линковања у извршну бинарну датотеку. Биће описан формат генерисаних датотека (ELF, HEX, BIN) и алати за њихову анализу (GNU binutils алати), као и структура линкерске скрипте прилагођена конкретној меморијској архитектури TRAVEO T2G чипа.

1.1. Циљ и мотивација рада

Циљ овог рада је да се систематски прикаже цео ток развоја уграђеног софтвера — од писања изворног C кода до добијања финалне бинарне слике која се програмира у меморију микроконтролера. Фокус је на компилационом процесу у оквиру GCC алатке, као и на детаљној анализи распореда меморијских секција кроз линкерску скрипту прилагођену конкретном микроконтролеру. У контексту растуће примене микроконтролера у аутомобилској индустрији, индустрији аутоматике и IoT систему, дубље разумевање унутрашњег функционисања овог процеса је од суштинске важности за пројектовање поузданих и безбедних система.

1.2. Методологија и приступ

Рад прати практичан инжењерски приступ: анализира се развој C програма за Infineon TRAVEO™ T2G микроконтролер (CYT4BB породица), који користи ARM Cortex-M7 језгро. Изложена су сва релевантна техничка средства: GCC компајлер и GNU binutils алати, структура startup кода и организација пројекта, као и конфигурација линкерске скрипте на основу спецификације меморијске мапе циљног хардвера. Кроз све фазе (препроцесирање, компилација, асемблирање, линковање), обрађен је начин на који се C код претвара у ELF датотеку, која се затим конвертује у HEX или BIN формат погодан за флешовање.

1.3. Релевантност теме

Обрада теме је уско везана за потребе реалног развоја софтвера у embedded окружењу, посебно у системима који раде без оперативног система (bare-metal). Правилна организација извора, разумевање формата резултујућих датотека и контрола над меморијским распоредом су предуслов за функционалан и безбедан рад микроконтролера у критичним апликацијама. Такође, разумевање ових аспеката представља основу за проширене теме као што су имплементација bootloader-a, сигурносна верификација кода (нпр. путем дигиталног потписа), и оптимизација потрошње ресурса.

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 и ресурсно ограниченим системима, посебно захваљујући стандардизацији преко ISO/IEC 9899:2018 (C18) и спецификацијама као што је CMSIS (Cortex Microcontroller Software Interface Standard) од стране ARM-a.

2.2. Кључне особине C језика у embedded контексту

У контексту програмирања микроконтролера, C 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(".секција"))), 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, C омогућава уметање асемблерских инструкција (__asm__) или позиве на екстерне рутине, што га чини флексибилним алатом за нисконивоовски развој.

- **Прецизна контрола времена извршавања** — У real-time системима детерминистичност је од пресудне важности. C не садржи runtime механизме попут garbage collector-a, чиме обезбеђује временски предвидљиво понашање.

2.3. Упоредна анализа C 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 C. 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).

C се тако намеће као оптималан компромис између перформанси, 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 библиотекахама и хардверским апстракционим слојевима.

3. Организација C изворног кода за embedded окружења

Развој софтвера за микроконтролере у програмском језику C 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. очитавање UART 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 модуларни 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 ниско-нивоске рутине 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_HANLDER_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 библиотечка 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 (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* – за извршавање и читање) од адресе 0x10000000 дужине 0x410000 бајтова, и ram (са *rwx* 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 занулавање 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 генераичке 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 (arm-software.github.ioarm-software.github.io). 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 cycfg_config_init() и др. 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 учитавање 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 регистрацијама 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 ниско-нивоске дефиниције и функције за сам процесор и основне периферије, независно од произвођача конкретног чипа. Кроз CMSIS, произвођачи микроконтролера испоручују сет заглавља и рутини које описују хардвер на симболичком нивоу – од регистара језгра до специјализованих периферијskih 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_Device.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** umesto ручног адресирања меморије.

Важно је нагласити да CMSIS пружа и стандардизовану шаблон-рутину за стартуп 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, конфигурисање такта система и осталих низводних компоненти. Према CMSIS стандарду, уобичајено је да се у склопу стартуп 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 консистентан и типски безбедан начин да управља хардвером. На пример, ако жељено 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 имплементиране 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 је перформантност таквог кода једнака 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 библиотека. 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, УАРТ-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. На тај начин, **HAL слојеви смањују количину хардверски зависног кода у главном програму**: велики део посла обавља библиотечка рутина, 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 апстрахује 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 прототипове брже готовим, јер се инжењер може

фокусирати на логику програма уместо на ниско-нивоске детаље иницијализације сваког подсистема.

Цена те погодности је одређен **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 1 секунду */
    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 cycfg_config_init() и cycfg_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 конфигурисање 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 конфигурисали 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 библиотечке 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. Индустријски 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 аутомотиве, 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 размакница), смислено именовање променљивих, константи и функција, структурисање 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

Комуникација између процесора и хардвера у *embedded* системима реализује се кроз меморијски мапиране регистре, што подразумева директан упис у специфичне адресе у меморијском простору. Програмски језик C омогућава дефинисање структура које одговарају распореду регистра, уз употребу кључне речи `volatile`, која обавештава компајлер да не сме оптимизовати приступ тим променљивим, јер се њихове вредности могу мењати асинхроно — нпр. од стране прекидне рутине, хардверског тајмера или другог језгра процесора.

На пример:

```
typedef struct {
    volatile uint32_t IN;
    volatile uint32_t OUT;
    volatile uint32_t DIR;
    // ... остали регистри
} GPIO_TypeDef;
#define GPIO ((GPIO_TypeDef *) 0x50000000UL)
```

Горњи код показује дефиницију структуре која моделује GPIO регистре и симболичку адресу на коју она мапира. У том случају, приступ регистру `OUT` врши се на следећи начин:

- `GPIO->OUT = 0x1;`

Ова једноставна линија кода представља суштину рада са периферијама у *bare-metal* окружењу. Она подразумева:

- да `GPIO` представља показивач на структуру која моделује GPIO регистре;
- да оператор `->` омогућава приступ пољу `OUT` те структуре;
- да се вредност `0x1` уписује у регистар `OUT`, чиме се поставља логичка „1“ на пин број 0, док се остали пинови постављају на „0“.

Под условом да је претходно извршена конфигурација правца рада пинова (нпр. `GPIO->DIR |= 0x1;`), ова наредба резултује у појави високог логичког нивоа на излазном пину. У Assembly коду, оваква операција се мапира на једну инструкцију (`STR`) која уписује вредност у конкретну адресу у меморијском простору, чиме се постиже детерминистичко и брзо извршавање — особина кључна за реалновременске системе.

Захваљујући CMSIS библиотеци и заглављима произвођача, програмери више не морају да користе непрегледне изразе склоне грешкама попут:

```
*(volatile uint32_t *)(0x50000004) = 0x1;
```

већ могу користити симболичке, типски безбедне и читљиве форме попут `GPIO->OUT = 0x1;`, чиме се обезбеђује значајно већа одрживост, транспарентност и преносивост кода.

3.5. Пример за Infineon TRAVEO™ T2G

У случају микроконтролера TRAVEO T2G (нпр. из породице CYT4BB), произвођач Infineon испоручује почетни код (*startup* assembly или C датотеку) који садржи векторску табелу прекида и функцију за ресет. Такође, обезбеђују се CMSIS-заглавља специфична за циљани модел, као и HAL библиотеке за руковање периферијама. Програмер затим развија сопствени код у `main.c`, организује логичке модуле (нпр. `motor_control.c`, `sensors.c`), и укључује потребна заглавља.

Пројекат поред тога садржи и линкерску скрипту (`.ld` датотеку), као што је `TRAVEO_T2G.ld`, прилагођену меморијској мапи микроконтролера. Ова скрипта дефинише секције као што су `.text`, `.data`, `.bss`, `.stack`, и њихово постављање у Flash или RAM меморију. Тиме се омогућава да свака компонента програма буде смештена у предвиђено меморијско подручје током фазе линковања.

На овај начин, добром организацијом извора, доследним коришћењем CMSIS-а и HAL библиотека, као и строго дефинисаним фазама иницијализације и петље извршавања, обезбеђује се основа за стабилан, поуздан и преносив *embedded* систем, који може бити успешно преведен и повезан у потпуности током следеће фазе — компилације у оквиру GCC окружења.

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 до четири корака: **препроцесирање**, **компилацију** (у ужем смислу), **асемблирање** и **линковање**, тим редоследом (Overall Options (Using the GNU Compiler Collection (GCC))). 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	Излаз
Препроцесирање	`gcc -E`	*.c, *.h (изворник)	Препроцесирани код (*.i)
Компилација (C->ASM)	`gcc -S`	*.i (из претходног)	Асемблерски код (*.s)
Асемблирање	`gcc -c` или `as`	*.s (асемблерски код)	Објектни фајл (*.o)
Линковање	`gcc` или `ld`	*.o (+ библиотеке)	Извршна датотека (ELF)

4.1. Препроцесирање

Препроцесирање: Прва фаза у којој се извршава C препроцесор. Он обрађује директиве почевши 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

Компилација (у ужем смислу): У овој фази GCC преводи препроцесирани C код у *асемблерски код* за циљну архитектуру. То значи да се синтакса и конструкције C 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

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

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-a, 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 (Overall Options (Using the GNU Compiler Collection (GCC))). 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

Увод.

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 (Executable and Linkable Format - Wikipedia). Он је прихваћен и код cross-компајлера за микроконтролере јер је флексибилан и независан од архитектуре – подржава различите процесоре, ендијаност и величине адресног простора (Executable and Linkable Format - Wikipedia). 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 учитавања у меморију при извршавању (Executable and Linkable Format - Wikipedia). 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 (Intel HEX - Wikipedia). 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 (Intel HEX - Wikipedia). 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 za флешовање) читају HEX датотеку линију по линију и уписују бајтове на наведене адресе у флеш меморију микроконтролера (Intel HEX - Wikipedia). 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) (Intel HEX - Wikipedia). 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 `objcopy` 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 objcopy 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 (objcopy (GNU Binary Utilities)). 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 учитавање). 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 учитавање преко bootloader-а), важно је разумети њихову структуру и разлике.

Motorola S-Record (S19)

Поред Intel HEX-а, чест је и **Motorola S-Record (S19)** формат – сличан ASCII хекс запису линија. Alati `objcopy` са опцијом `-O srec` може генерисати S-Record фајл (objcopy (GNU Binary Utilities)). 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. Употреба GNU алата: objcopy, readelf, nm, size, objdump

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 програмигања микроконтролера.

6.1. objcopy

`objcopy`: 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). `objcopy` може и издвајати 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 (objcopy (GNU Binary Utilities)) (objcopy (GNU Binary Utilities)). 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.

6.2. readelf

`readelf`: Алат 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 меморијским регијама.

6.3. nm

`nm`: Листирање 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 (T=текст/функција, D=иницијализовани податак, B=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 `B` или `D` и имаће адресу у опсегу RAM меморије), или да ли је функција у флешу (ознака `T` 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. size

`size`: 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. objdump

`objdump`: 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. Закључак

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: MEMORY и SECTIONS дефиниције

Линкерска скрипта или линкер директива (`.ld` датотека) је суштински део embedded пројекта – она диктира како ће линкер распоредити преведене секције програма у меморијски адресни простор микроконтролера. Пошто микроконтролер има одређене величине и адресе флеш и RAM меморије, потребно је линкер обавестити где може смештати код, а где податке. GNU ld линкер има свој интерни језик за скрипте који омогућава опис меморијских региона и расподелу секција.

7.1. MEMORY дефиниција

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

```
MEMORY
{
    FLASH (rx)  : ORIGIN = 0x10000000, LENGTH = 4096K
    WORK_FLASH (rx) : ORIGIN = 0x14000000, LENGTH = 256K
    SRAM (rwx)  : ORIGIN = 0x28000000, LENGTH = 768K
}
```

Овим смо линкеру дали до знања да постоје три региона: `FLASH` који почиње на адреси 0x1000_0000 величине 4 MB, додатни `WORK_FLASH` од 256 KB, и `SRAM` од 768 KB почев од 0x2800_0000. Сваки регион има атрибуте: `r` (читљив), `w` (уписив), `x` (извршан). Флеш је обично означен са `rx` (читање + извршавање, није уписив у току рада програма), RAM са `rwx` (читање/писање, извршавање ако се код копира у RAM). Ове MEMORY декларације омогућавају линкеру да разуме расположиви адресни простор и да пријави грешку ако одређена секција премашује додељени меморијски блок. На пример, ако код постане превелик (прелази 4MB флеша), линкер ће јавити да `FLASH` регион нема довољно простора. MEMORY секција дакле описује распон и намену меморијских блокова циљног система (The most thoroughly commented linker script (probably) - Stargirl (Thea) Flowers).

7.2. SECTIONS расподела

Након дефинисања меморије, линкерска скрипта садржи `SECTIONS` блок, који је срце скрипте – ту се наводи како се свакој излазној секцији ELF-а додељује меморијски регион и адреса. Пример делимичне SECTIONS скрипте за ARM:

```
SECTIONS {
    .text :
    {
       _stext = .;           /* почетак .text секције */
       KEEP(*(.isr_vector))  /* векторска табела, ако је дефинисана у секцији
.isr_vector */
       *(.text*)             /* све .text секције из свих објеката */
       *(.rodata*)           /* све .rodata (константе) секције */
       _etext = .;           /* крај .text/.rodata секције */
    } > FLASH                /* овај блок иде у FLASH меморију */

    .data : AT (ADDR(FLASH) + SIZEOF(.text)) /* AT(...) каже да се почетни
садржај .data налази након .text у FLASH */
    {
       _sdata = .;           /* почетак .data у RAM */
       *(.data*)             /* све иницијализоване статичке промењиве */
       _edata = .;
    } > SRAM                 /* овај блок се мапира у SRAM на извршавање */

    .bss (NOLOAD) :
    {
       _sbss = .;            /* почетак .bss */
       *(.bss*)
       *(COMMON)
       _ebss = .;
    } > SRAM                 /* .bss је у SRAM, NOLOAD јер нема почетних
података у флешу */

    /* ... остале секције (стек, heap, секције за C++ EH, итд.) ... */
}
```

Овај пример илуструје кључне концепте. Прво, `.text` (са .rodata) је смештен у `FLASH` регион. У њему се чак *принудно чува* (`KEEP`) векторска табела ако је дефинисана у посебној секцији `.isr_vector` (многи startup кодови стављају векторску таблицу у засебну секцију на почетак флеша). Затим .text обухвата сав програмски код и неизменљиве податке. Симболи као `_stext` и `_etext` су обележивачи које можемо касније користити (нпр. `_etext` указује на крај садржаја који се налази у флешу, што је уједно почетак података за копирање у RAM).

Део за `.data` је интересантан јер демонстрира одвајање *LOAD адресе* и *RUN адресе*: секција `.data` ће током извршавања бити у SRAM (због `> SRAM`), али је наведено `AT (ADDR(FLASH) + SIZEOF(.text))`. Ово каже линкеру да резервише простор у FLASH-у за иницијалне вредности `.data` секције, одмах након `.text` блока.

Практично, то значи да ће у ELF-у `.data` имати два различита почетка: *VMA* (Virtual Memory Address) у RAM (нпр. 0x2800_0000 ако ту почиње SRAM) и *LMA* (Load Memory Address) у флешу (нпр. 0x1000_1234 ако .text заузима до 0x1000_1233). Линкер ће поред ELF секције уписати и тај садржај (иницијалне бајтове) на LMA адресу. Потом, стартуп код микроконтролера копира те бајтове из флеша на одговарајућу VMA адресу у SRAM током покретања програма. На овај начин, глобалне променљиве које имају иницијалне вредности (нпр. `int count = 5;`) налазе се у .data секцији: након ресета, у RAM-у на адреси `_sdata` треба да се нађе вредност 5, коју је програм негде морао сачувати – управо у флешу, у склопу .data LMA. Директива `AT()` у линкер скрипти се користи да раздвоји те адресе и веома је битна за исправно покретање програма (The most thoroughly commented linker script (probably) - Stargirl (Thea) Flowers).

Секција `.bss` у примеру има атрибут `NOLOAD` – то значи да за њу не постоји садржај у флешу; она представља блок RAM меморије коју треба испунити нулама. Линкерски симболи `_sbss` и `_ebss` означавају почетак и крај .bss. Важно је и да су сви *COMMON* симболи (неиницијализоване глобалне променљиве које нису експлицитно у .bss) такође стављени у .bss (директива `*(COMMON)`). Стартуп код ће једноставно обрисати (испунити нулама) овај RAM опсег при покретању.

Поред ових, линкер скрипта често резервише простор за стек (нпр. дефинише симбол `_estack` на крај SRAM-а), за heap (ако се користи), и укључује специјалне секције за C++ ако су присутне (нпр. `.init_array` за конструкторе објеката). У случају ARM Cortex-M, потребне су и секције `.ARM.exidx` и `.ARM.extab` за информације о изузецима (корисно ако се користе одређене библиотеке и функције или C++ бацање изузетака – throw exception). Они се обично смештају на крај `.text` сегмента. Укључујући све те ставке, линкерска скрипта обезбеђује да у излазном ELF-у стоје исправно дефинисане адресе и границе секција.

За пример TRAVEO T2G микроконтролера, линкерска скрипта би била прилагођена тачним величинама меморија тог чипа. На основу **меморијске архитектуре**, могли бисмо дефинисати MEMORY са `FLASH (rx) : ORIGIN = 0x10000000, LENGTH = 4160K` (главна флеш меморија од ~4.16 MB), евентуално `WORKFLASH (rx) : ORIGIN = 0x14000000, LENGTH = 256K`, `SRAM (rwx) : ORIGIN = 0x28000000, LENGTH = 768K`, и можда посебно `ITCM` и `DTCM` меморије ако би се користиле (16KB + 16KB по језгру). За једноставност, претпоставимо да сав код иде у главни FLASH, а сав RAM садржај у главни SRAM. Линкерска скрипта би онда одразила тај модел: .text, .rodata, евентуално векторска табела и startup код у FLASH; .data LMA у FLASH али VMA у SRAM; .bss у SRAM.

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

8. Закључак