ChatGPT

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 [1]. На основу тога, програмер развија сопствени изворни код у датотеци **main.c** и повезаним модулима (нпр. сензори, управљање мотором), укључујући потребне хедере за сваки модул [2]. 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 [2]. 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.

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 [3].

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:

```c
    result = cybsp_init();                  /* Иницијализација плоче и периферија */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);                       /* Заустави извршење ако иницијализација није успела */
    }
    __enable_irq();                         /* Омогућавање глобалних прекида */
```
[4]

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 УАРТ улаза и укључивање/искључивање LED диоде у примеру) [5]

[6]. *Напомена:* конкретна реализација `main()` може знатно да варира у зависности од пројекта – наведени код је само једна могућа варијанта имплементације.

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 [7]. 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:

```c
cy_rslt_t cyhal_timer_start(cyhal_timer_t *obj);
```
[8]

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:

```c
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, ... );                    /* ЕнEnable тајмер (покретање бројања) */
}
```
[9]

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.

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()` [10].

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:

```
asm
__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) */
```
[11]

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):

```asm / ... (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) */
```

``` [12] [13]

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.

## 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 [14]. 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):

```
ld
    ram   (rwx) : ORIGIN = 0x08020000, LENGTH = 0x5F800
    flash (rx) : ORIGIN = 0x10000000, LENGTH = 0x410000
     sflash_user_data (rx) : ORIGIN = 0x17000800, LENGTH = 0x800   /* специјални
Flash */
    ...
```
[15]

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) [16] [17]. 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:
```

```
ld
    .data __ram_vectors_end__ : AT>flash {
        ...
        __data_end__ = .;
    } > ram
```
[18]

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 [19] [20]. 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 [21]. Линкерска скрипта обично додељује и симболе као што су **__StackTop** и **__StackLimit** на крају RAM меморије, чиме се дефинише позиција и величина стека програма [22].

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 [14].

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 [23]. 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 [24] [25]. 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:

```
c
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;
}
```
[26]

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.

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:

```bash
arm-none-eabi-gcc -O2 -mcpu=cortex-m7 -o program.elf main.c uart.c startup.s -T linker.ld
```
[27]

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 [28].

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.

---

1 2 3 7 10 14 27 28 Од изворног C кода до извршне бинарне слике - компилација и распоред у меморији микроконтролера.pdf
file://file-U7PzurVU4PZSr7Xo3Pct4S

4 5 6 main.c
file://file-MKeyr29RjHoZXXT2LGAyhK

8 cyhal_timer.h
file://file-CfKtpinpMG7735bymQS27J

9 cyhal_timer.c
file://file-RqZiBoA2CXgmcsFbjcrNz9

11 12 13 19 20 23 startup_tviibe4m_cm4.S
file://file-7KsoiFLuNMtVmP8i8N1eYJ

15 16 17 18 21 22 linker.ld
file://file-6JUsHqRo1wzpSeH1dyjdWe

24 25 CMSIS-Core (Cortex-M): System and Clock Configuration
https://arm-software.github.io/CMSIS_6/main/Core/group__system__init__gr.html

26 cybsp.c
file://file-MaFaFLFMQ3so4WyRPhtt9w