

## Употреба CMSIS и HAL слојева

У развоју софтвера за микроконтролере, уобичајено је ослањање на стандардизоване слојеве абстракције 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.

### 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 рутине 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 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, конфигурирање 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.

## 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, 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. 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 платформи. 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 конфигуравање периферијских модула који ће те прекиде користити. У примеру, након што је тајмер конфигуриран и стартован позивима `cyhal_timer_*` функција (HAL апстракција за тајмерски блок), тајмерски хардвер аутономно броји време и генерише прекид сваке секунде. Тај прекид се обрађује у позадини (HAL је регистровао `isr_timer` обрађивач преко `cyhal_timer_register_callback()`), при чему та обрада постави заставицу `timer_interrupt_flag`. Главна петља програма (`for(;;)`) затим, уз помоћ тог флага, зна када је једна секунда протекла и у одговору позива `cyhal_gpio_toggle()` – HAL функцију која мења стање излазног пина где је прикључена LED диода. Резултат је трептање диоде у интервалу од 1Hz, остварено без иједног директног уписа у хардверски регистар у корисничком коду. Сви уписи (нпр. подешавање излаза пина или конфигурација тајмера) реализовани су унутар HAL функција, користећи при том CMSIS дефинисане симболе за приступ одговарајућим регистрима.

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

---