Интегрирани среди за развой

Автор: доц. д-р инж. Любомир Богданов

Европейски съюз

ПРОЕКТ BG051PO001--4.3.04-0042
„Организационна и технологична инфраструктура за учене през целия живот и развитие на компетенции”
Проектът се осъществява с финансовата подкрепа на
Оперативна програма „Развитие на човешките ресурси”,
съфинансирана от Европейския социален фонд на Европейския съюз
Инвестира във вашето бъдеще!

ЕСФ България
Европейски социален фонд

Съдържание

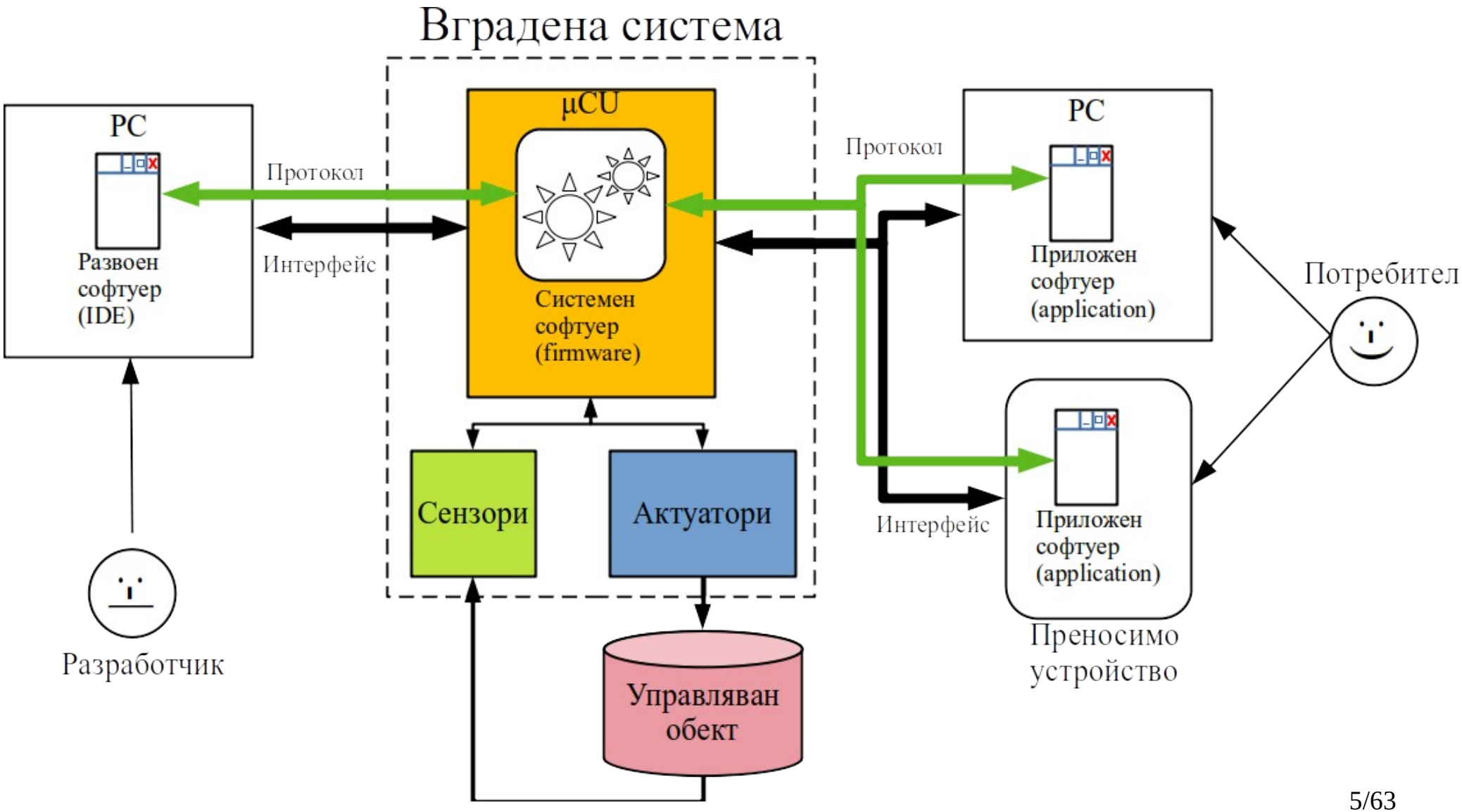
Разработка на системен софтуер

Системен софтуер (firmware) – софтуер, работещ на ниско ниво на абстракция, който модифицира регистри със специално предназначение и по този начин управлява хардуера.

Приложен софтуер (application) софтуер, работещ на високо ниво на абстракция, който задава команди и чете състоянието на вградената система посредством фърмуера и интерфейс, поддържащ даден протокол за обмен на данни.

Развоен софтуер (development software) – набор от програми, чрез които се създава изпълнимия код на системния софтуер. Използват се т.нар. развойни среди (виж по-следващия слайд).

Разработка на системен софтуер

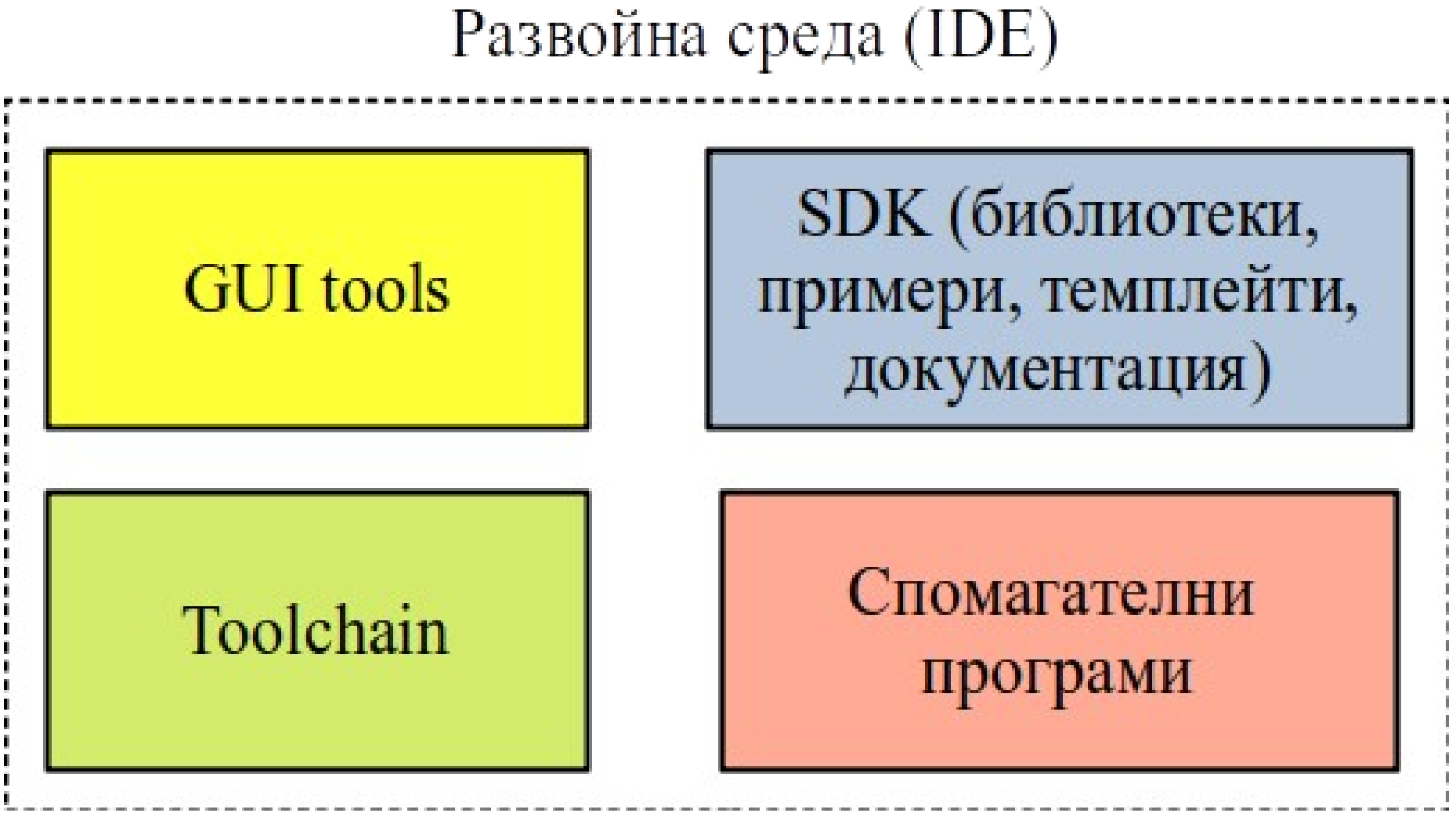


Разработка на системен софтуер

Интегрирана развойна среда (Integrated Development Environment) – набор от програми от командния ред и програми с графичен интерфейс (Graphical User Interface) GUI), с помощта на които се създава изпълнимия код на системния софтуер (build), зарежда се в паметта на системата и позволява да се откриват и отстраняват грешки (debug).

IDE може да бъде разделена на 4 части:
*GUI програми
*Toolchain
*Спомагателни програми
*SDK

Разработка на системен софтуер



Разработка на системен софтуер

GUI програмите са това, което проектантът вижда на разойната среда. Те включват:

***GUI текстови редактор** – мястото, където се въвежда сорс кода в текстови вид;

***GUI дебъгер** – графична програма за управление на софтуерен дебъгер;

***GUI файлов експлорър** – гр. програма за представяне файловата йерархия на проекта/проектите;

***GUI команден ред** – гр. програма за показване на съобщенията на всички други програми, които работят от командния ред;

***GUI блоков редактор** – гр. програма за представяне и редактиране блоковата схема на система, реализирана върху FPGA;

***други**.

Разработка на системен софтуер

Toolchain - набор от програми за командния ред, чрез които се създава изпълнимия код на системния софтуер и позволява да се откриват и отстраняват грешки. Toolchain-ът съдържа:

*__компилатор__ на C/C++ (compiler)
*__асемблер__ (assembler)
*__линкер__ (linker)
*__дебъгер__ (debugger)
*програми за **обработка на двоични файлове** (binary utilities)
*програми за **статичен анализ** на кода (profiling tools)

Разработка на системен софтуер

Спомагателни програми - набор от програми за командния ред, спомагащи създаването на изпълнимия код:
*__команден ред__ (**C**ommand **L**ine **I**nterface, CLI)
*__програми за автоматизация на създаването__ (build automation tools);
*__програми за зареждане на системния софтуер__ в паметта на системата (flash utilities)
*__програми за контрол на версията__ (version control)
*__програми за автоматично документиране__ (documentation generator)
*__други__

Разработка на системен софтуер

SDK (**S**oftware **D**evelopment **K**it) – спомагателен софтуер за фърмуера. Обикновено се дава от производителя на микроконтролера и включва следните компоненти:

*__библиотеки за µPU__ (съдържат вътрешни функции и дефиниции на регистри от микропроцесорната периферия);

*__библиотеки за µCU__ (съдържат библиотеки за работа с периферните модули на контролера, Hardware Abstraction Layer, HAL);

*__библиотеки за демо платки__ и еталонни дизайни (такива библиотеки се наричат Board Support Package, BSP);

*__външен софтуер__ (наричат го third party: файлови системи, програми за статистика, CLI за микроконтролери, и т.н.)

*__примерни проекти__;

*__заготовки__ (темплейти) за проекти;

*__документация__ (на библиотеките, на хардуера – технически спецификации, примерни приложения, и др.);

*__други__

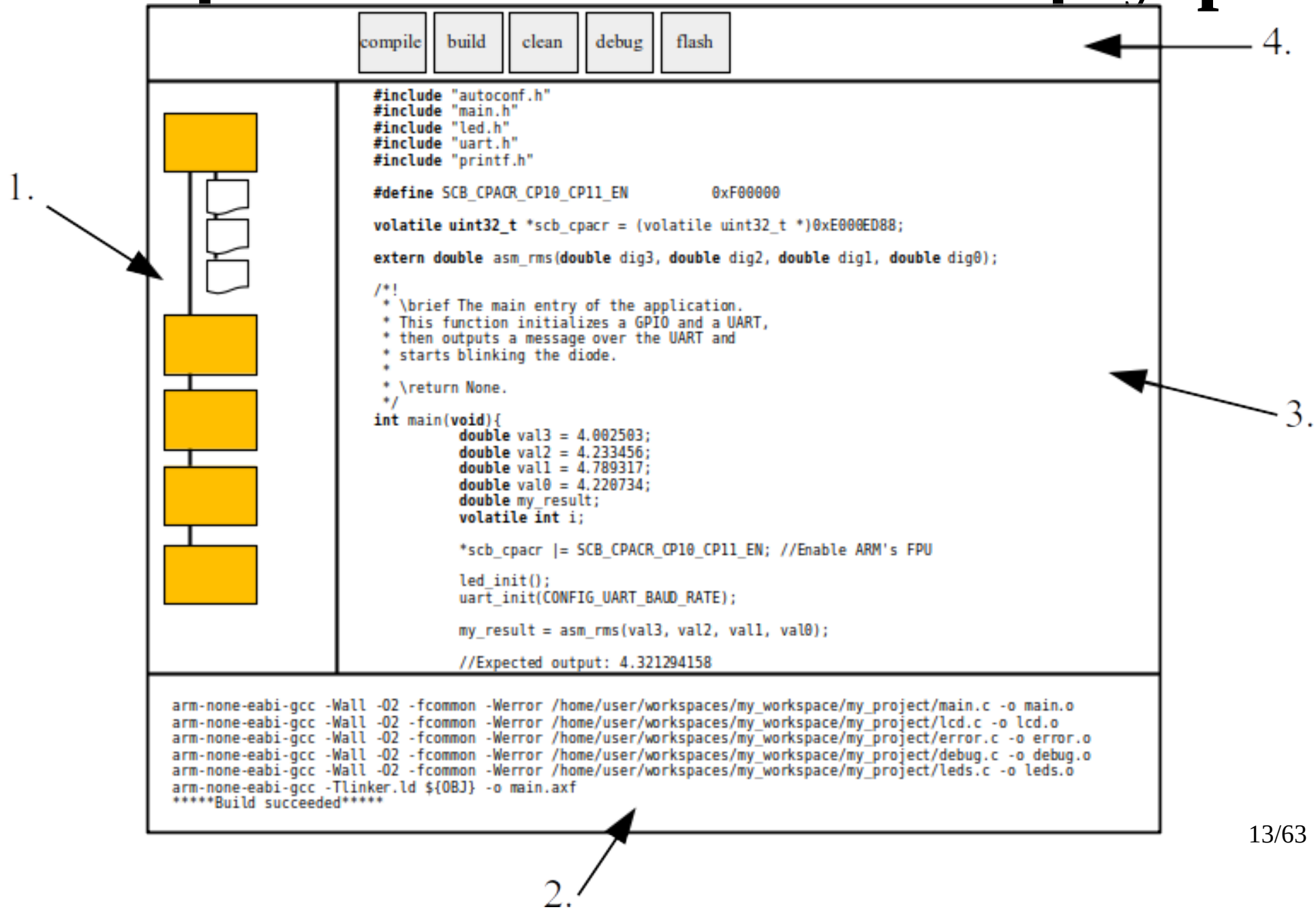
Разработка на системен софтуер

Всички развойни среди за μCU имат общи черти и външният им изглед може да се обобщи, както е направено на следващия слайд. В **режим на въвеждане на код** (edit) има следните полета:

1. **Файлов експлорър** – покава директории и файлове на проекта/работното място (project/workspace);
2. **Команден ред** – показва кои команди се изпълняват в момента от средата;
3. **Текстов редактор** – съдържа сорс кода на фърмуера;
4. **Панел с бутони** – съдържа:

*compile – компилиране отворения в редактора файл;
*build – компилиране, асемблиране и линкване на всички сорс; файлове в проекта/работното място;
*clean – изтриване на всички обектови и двоични файлове;
*debug – стартиране на дебъг сесия;
*flash – програмиране на контролера без да се стартира дебъг сесия;

Разработка на системен софтуер

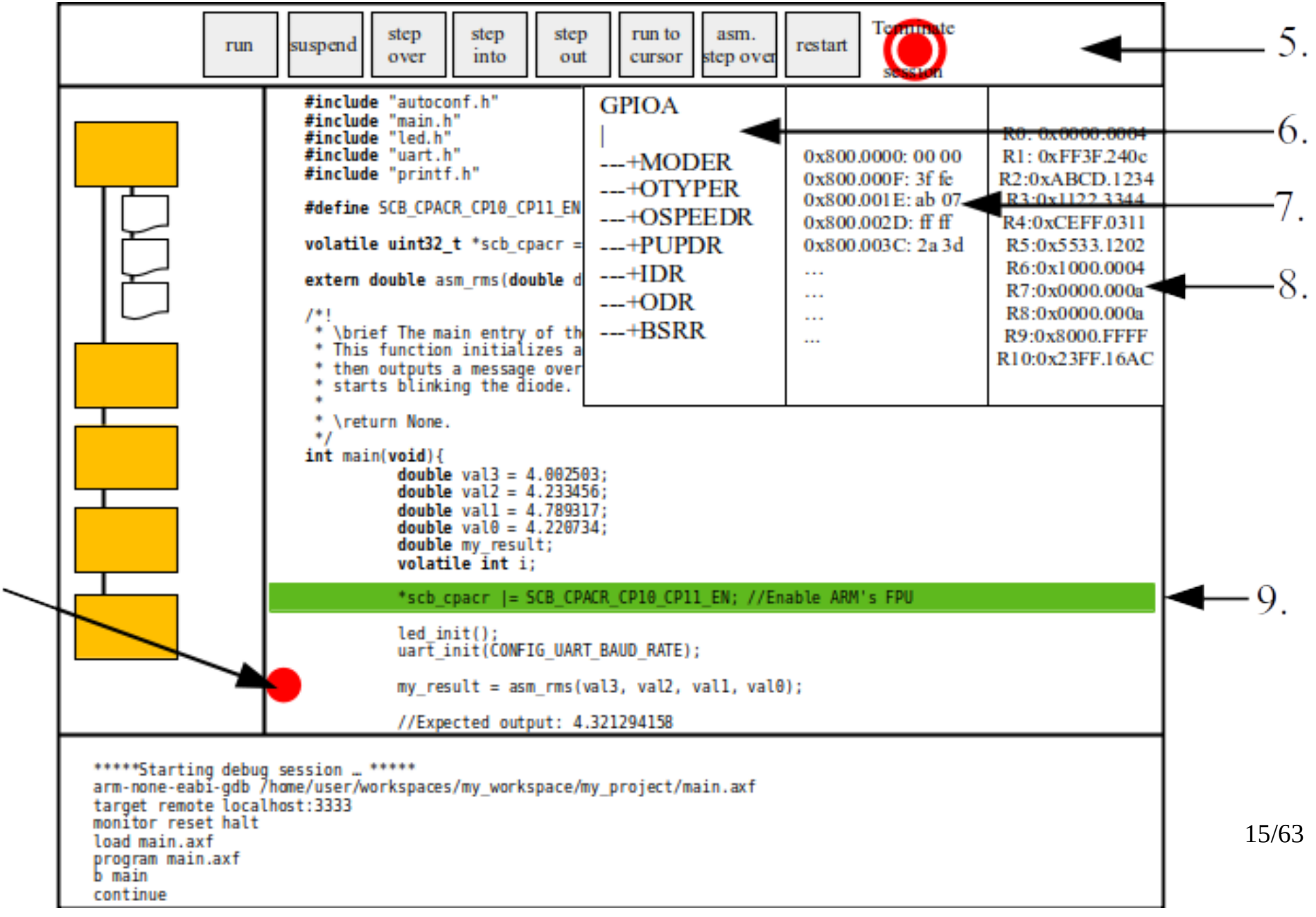


Разработка на системен софтуер

Всички развойни среди променят “лицето си” (изгледа, perspective), когато влязат в **режим на дебъг сесия** (debug) при натискане на бутона debug. Отново може да се каже, че имат общи черти (виж следващия слайд):

5. панел с бутони за дебъгване;
6.панел, показващ съдържанието на регистрите на даден I/O модул;
7.панел, показващ съдържанието на регистрите на паметите (Flash, Ferro, SRAM);
8.панел, показващ съдържанието на регистрите на микропроцесорното ядро;
9.засветяване на ред от редактора, показващо докъде е стигнал μPU в изпълнението на програмата;
10.точка на прекъсване, която ще пренуди μPU да спре изпълнението на програмата и да върне контрола на хардуерния/софтуерния дебъгер ;

Разработка на системен софтуер



Разработка на системен софтуер

Бутоните на дебъг панела са:
*__run -__ μPU се пуска да изпълнява програмата безкрайно, докато тя не завърши, или докато не се натисне бутона suspend;
*__suspend__ – спира изпълнението на фърмуера от μPU и се чакат команди от другите бутони;
*__step over__ – върви се ред по ред в програмата и ако се срещне функция, изпълнява се нейното тяло и контрола се предава обратно на дебъгера след края на функцията;
*__step in__ – върви се ред по ред в програмата и ако се срещне функция, влиза се в нея и започва да се върви ред по ред там;

Разработка на системен софтуер

***step out** – ако μPU се намира в средата на много дълга функция, натискането на този бутон ще придвижи изпълнението до края на функцията и ще спре μPU във функцията, която е едно ниво по-нагоре;
***run to cursor** – изпълнението на програмата се придвижва до позицията на курсора на мишката, след което се спира μPU;
***assembly step over** – върви се ред по ред в програмата, но дебъгера спира μPU след изпълнението на всяка инструкция (един ред на C може да е съставен от няколко реда на Асемблер), което прави стъпкуването по-фино;
***restart** – връща се програмния брояч в началото на програмата.

Етапи в създаването на изпълним код

Етапите в създаването на изпълним код са следните:

1. Въвежда се сорс код на C/C++ в текстови редактор.

2.Текстовият файл се подава на компилатор, който създава асемблерния еквивалент на С програмата с условни адреси.

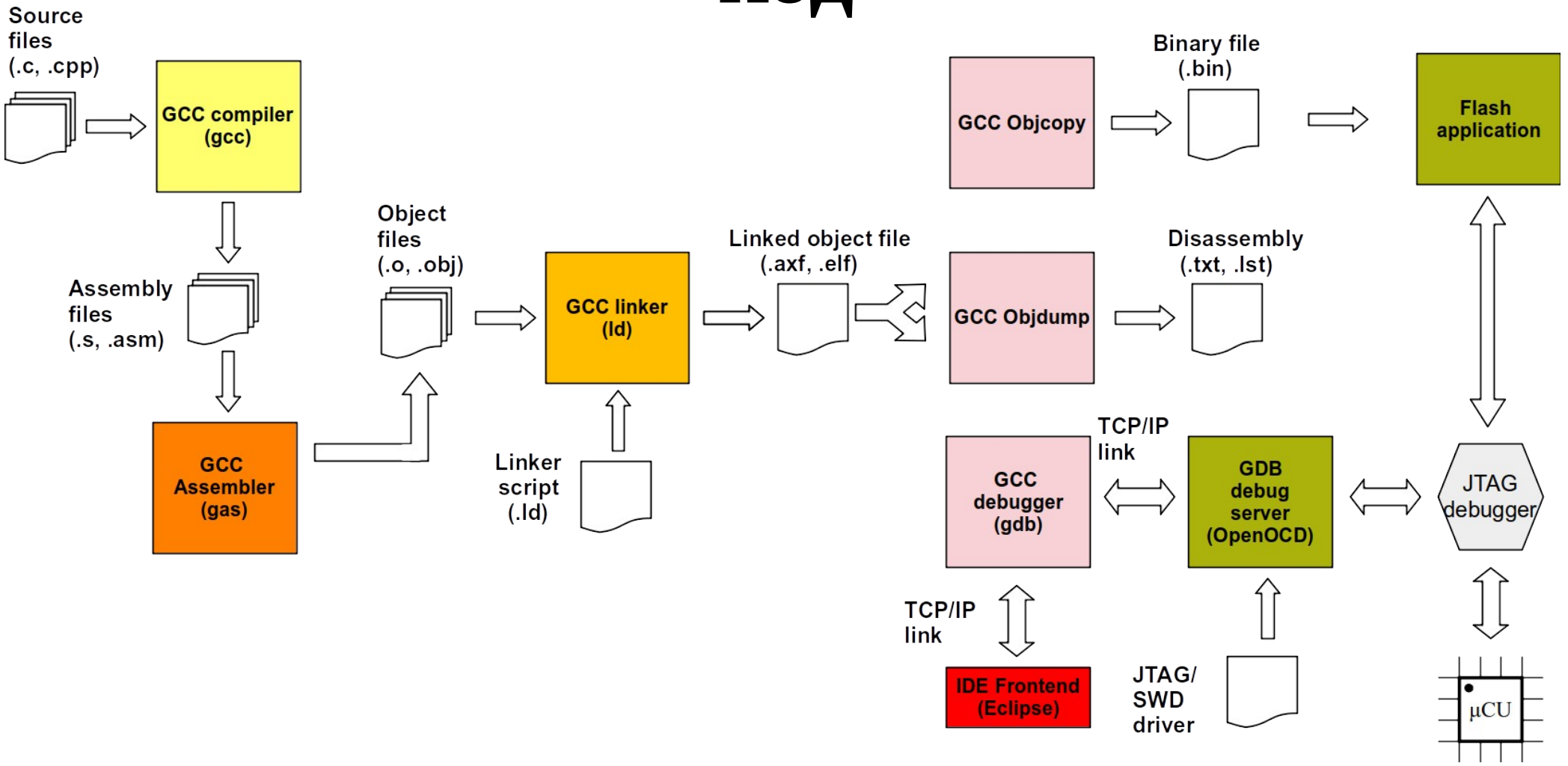
3.Асемблерният еквивалент се подава на програмата асемблер, която създава обектов код с условни адреси.

4.Обектовият код с условни адреси се подава на линкера, който създава обектов код с абсолютни адреси и дебъгерна информация. Ако в проекта има други обектови файлове, те се линкват с настоящия.

5. Обектовият код с абсолютни адреси се подава на програма за обработка на двоични файлове и дебъгерната информация се премахва. Остава чист, изпълним, двоичен файл.

6. Двоичният изпълним файл се зарежда в паметта на системата посредством специализирана приложна програма.

Етапи в създаването на изпълним код



Етапи в създаването на изпълним код

Компилаторите, асемблерите и линкерите прилагат **оптимизации на кода**, за да се подобри някои от параметрите на програмата:
*бързодействие
*размер
Възможно е генерираният код да е грешен. Затова се използва програма, наречена дисасемблер**.**
Дисасемблер – програма, която преобразува двоичния код на фърмуера в код на Асемблер (текстови вид), и кода на Асемблер в код на С (текстови вид). Полученият файл се нарича **дисасемблерен листинг.** С негова помощ може да станат видни оптимизациите, приложени върху кода и да се поправи грешката чрез пренаписване оригиналния фърмуер, или чрез вмъкване на директиви за временно изключване на оптимизациите.

Етапи в създаването на изпълним код

Едва ли има човек, който да напише сложна програма правилно от първия път. Създаването на изпълним код е итеративен процес, в който се използват **софтуерни и хардуерни дебъгери**, за да се отстранят грешките в кода.

На блоковата схема от по-предишния слайд GCC debugger е **софтуерния дебъгер.** Той се нуждае от графичен фронтенд (IDE Frontend) и GDB debug server. С тяхна помощ се управлява μPU, така че той да изпълнява команди при натискане на бутон от графичния интерфейс.

За да може софтуерния дебъгер да се свърже с μPU, необходим е хардуерен интерфейс. Този интерфейс се осигурява от RS232/USB/Ethernet ↔ JTAG/SWD/SBW адаптер, който може да се нарече **хардуерен дебъгер**.

Етапи в създаването на изпълним код

Пример: сорс код

```
int main(void){
    volatile int i;

    led_init();
    uart_init(CONFIG_UART_BAUD_RATE);

    printf("This is an example usage of printf and USART%d\n", 1);

    while (1){
        led_set();
        for(i = 0; i < LED_BLINK; i++){ }
        led_clear();
        for(i = 0; i < LED_BLINK; i++){ }
    }
}
```

Етапи в създаването на изпълним код

Пример: асемблерен еквивалент с условни адреси.

```
main:
    push   {r7, lr}
    sub sp, sp, #8
    add r7, sp, #0
    bl  led_init
    mov r0, #9600
    bl  uart_init
    movs   r1, #1
    ldr r0, .L7
    bl  printf_
.L6:
    bl  led_set
    movs   r3, #0
    str r3, [r7, #4]
    b   .L2
.L3:
    ldr r3, [r7, #4]
    adds   r3, r3, #1
    str r3, [r7, #4]
.L2:
    ldr r3, [r7, #4]
    ldr r2, .L7+4
    cmp r3, r2
    ble .L3
    bl  led_clear
    movs   r3, #0
    str r3, [r7, #4]
    b   .L4
.L5:
    ldr r3, [r7, #4]
    adds   r3, r3, #1
    str r3, [r7, #4]
.L4:
    ldr r3, [r7, #4]
    ldr r2, .L7+4
    cmp r3, r2
    ble .L5
    b   .L6
.L8:
    .align 2
.L7:
    .word  .LC0
    .word  399999
```

Етапи в създаването на изпълним код

Пример: асемблерен обектов еквивалент с условни адреси.

```
00000000 <main>:
   0:   b580        push    {r7, lr}
   2:   b082        sub sp, #8
   4:   af00        add r7, sp, #0
   6:   f7ff fffe   bl  0
   a:   f44f 5016   mov.w   r0, #9600
   e:   f7ff fffe   bl  0
  12:   2101        movs    r1, #1
  14:   480d        ldr r0, [pc, #52]
  16:   f7ff fffe   bl  0
  1a:   f7ff fffe   bl  0
  1e:   2300        movs    r3, #0
  20:   607b        str r3, [r7, #4]
  22:   e002        b.n 2a
  24:   687b        ldr r3, [r7, #4]
  26:   3301        adds    r3, #1
  28:   607b        str r3, [r7, #4]
  2a:   687b        ldr r3, [r7, #4]
  2c:   4a08        ldr r2, [pc, #32]
  2e:   4293        cmp r3, r2
  30:   ddf8        ble.n   24
  32:   f7ff fffe   bl  0
  36:   2300        movs    r3, #0
  38:   607b        str r3, [r7, #4]
  3a:   e002        b.n 42
  3c:   687b        ldr r3, [r7, #4]
  3e:   3301        adds    r3, #1
  40:   607b        str r3, [r7, #4]
  42:   687b        ldr r3, [r7, #4]
  44:   4a02        ldr r2, [pc, #8]
  46:   4293        cmp r3, r2
  48:   ddf8        ble.n   3c
  4a:   e7e6        b.n 1a
  4c:   00000000    andeq   ...
  50:    00061a7f   andeq   ...
```

Етапи в създаването на изпълним код

Пример: асемблерен обектов еквивалент с абсолютни адреси (след линкване).

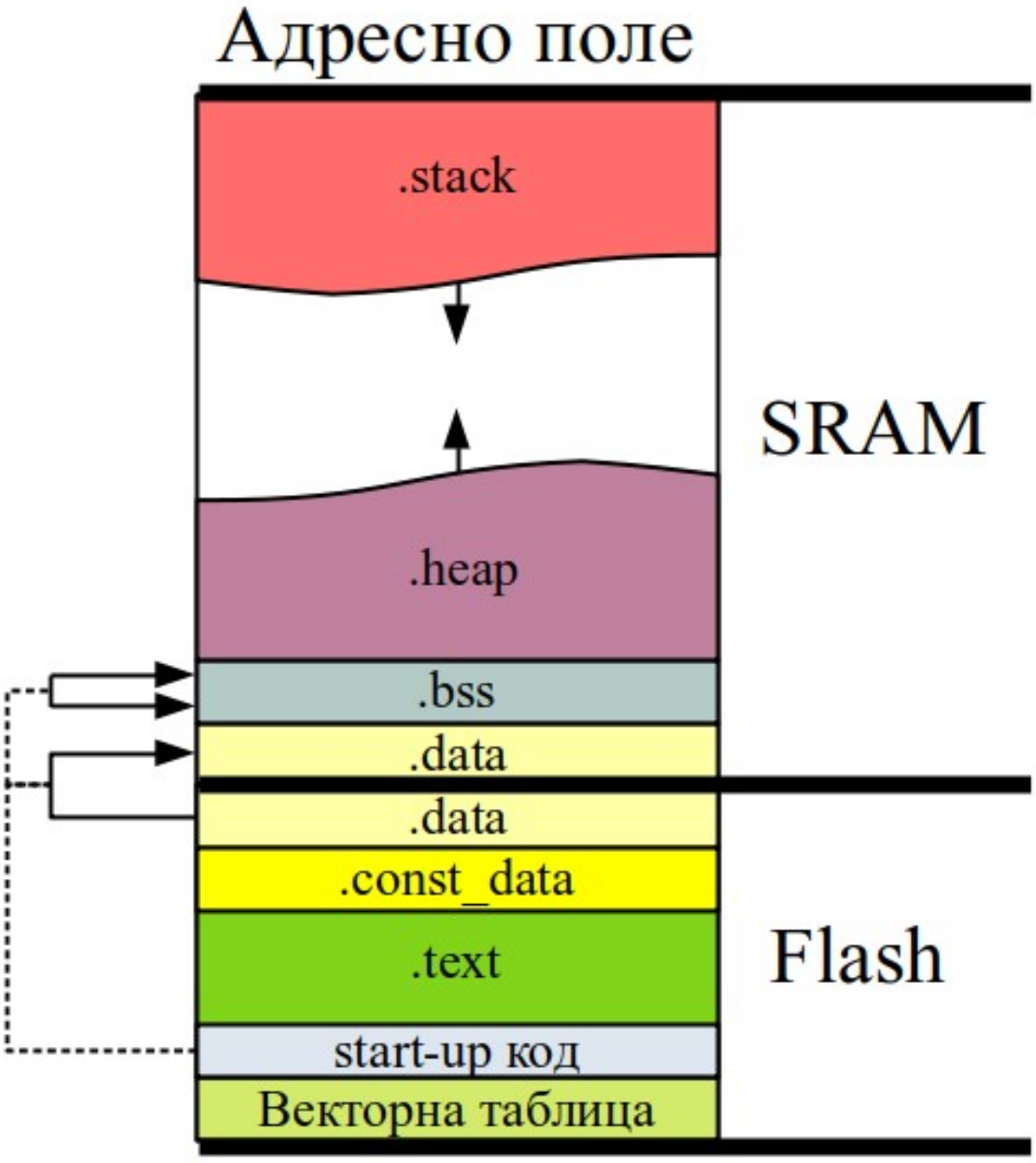
```
080001f8 <main>:
80001f8:    b580        push    {r7, lr}
80001fa:    b082        sub sp, #8
80001fc:    af00        add r7, sp, #0
80001fe:    f000 f825   bl  800024c
8000202:    f44f 5016   mov.w   r0, #9600
8000206:    f000 f86f   bl  80002e8
800020a:    2101        movs    r1, #1
800020c:    480d        ldr r0, [pc, #52]
800020e:    f001 fdeb   bl  8001de8
8000212:    f000 f849   bl  80002a8
8000216:    2300        movs    r3, #0
8000218:    607b        str r3, [r7, #4]
800021a:    e002        b.n 8000222
800021c:    687b        ldr r3, [r7, #4]
800021e:    3301        adds    r3, #1
8000220:    607b        str r3, [r7, #4]
8000222:    687b        ldr r3, [r7, #4]
8000224:    4a08        ldr r2, [pc, #32]
8000226:    4293        cmp r3, r2
8000228:    ddf8        ble.n   800021c
800022a:    f000 f84d   bl      80002c8
800022e:    2300        movs    r3, #0
8000230:    607b        str r3, [r7, #4]
8000232:    e002        b.n     800023a
8000234:    687b        ldr r3, [r7, #4]
8000236:    3301        adds    r3, #1
8000238:    607b        str r3, [r7, #4]
800023a:    687b        ldr r3, [r7, #4]
800023c:    4a02        ldr r2, [pc, #8]
800023e:    4293        cmp r3, r2
8000240:    ddf8        ble.n   8000234
8000242:    e7e6        b.n     8000212
8000244:    08002b6c    stmdaeq ...
8000248:    00061a7f    andeq   ...
```

Етапи в създаването на изпълним код

За да може една програма на C/C++ да се изпълнява, трябва **първо паметта да се раздели на сегменти** от стартиращия код (start-up code). Този код е първият код, който се изпълнява след получаване на прекъсване за Reset и извличане на адреса на Reset хендлера. Кодът може да е написан на асемблер или на C. Ако се пише **стартиращ код на C не трябва да се използват променливи,** освен указатели (които имат директно представяне на Асемблер чрез абсолютна адресация), защото паметта все още не е разделена.

Имената на сегментите се дават от програмиста, чрез линкерния скрипт. На следващия слайд са показани най-често използваните имена за тях.

Етапи в създаването на изпълним код



Етапи в създаването на изпълним код

.bss – съдържа глобалните и статични (декларирани като static) неинициализирани или инициализирани на нула променливи. Сегментът се създава от стартиращия код, като между началният му и крайният му адрес се записват нули. Променливите могат да бъдат четени, записвани и презаписвани.

.data – съдържа глобалните и статични променливи, инициализирани от програмиста. Сегментът се създава от стартиращия код, като се копират инициализиращите данни от Flash в SRAM. Променливите могат да бъдат четени, записвани и презаписвани.

.const_data – съдържа глобални и локални константи (променливи, декларирани като const). Сегментът се създава от дебъгера във Flash, при запис на програмата в микроконтролера. Променливите могат да бъдат само четени.

Етапи в създаването на изпълним код

.text (или **.code**) – съдържа инструкциите на програмата. Сегментът се създава от дебъгера във Flash, при запис на програмата в микроконтролера. Инструкциите могат да бъдат само извличани от микропроцесора.

.stack – съдържа служебна информация, копирана от хардуера (виж стекова група, stack frame), аргументи (предавани по стойност) на функции и локални променливи (които не са декларирани като static). За запис и четене в този регион се използват push и pop инструкции, които организират достъпа до паметта по схемата LIFO буфер. Регионът не се създава при стартирането, а само се пълни от потребителската програма, когато се извикват функции една в друга.

Етапи в създаването на изпълним код

Пояснение – понякога програмистите от високо ниво използват думата “стек” с друго значение – като библиотека, която сама си организира и разделя паметта за нейни, вътрешни нужди. Обикновено сложните библиотеки се наричат стек – USB stack, TCP/IP stack, Bluetooth stack и т.н.

Хардуерен стек (dedicated hardware stack) – специализирани регистри от μPU ядро или друг контролерен елемент, които се използват за стекова памет. Повечето μPU използват SRAM за тази цел, но има някои които са реализирани с допълнителна памет (например PIC18 има 31 регистъра за стек, FPU копроцесорът 8087 има 8 регистъра стек).

Етапи в създаването на изпълним код

.heap – съдържа динамично заделени променливи (т.е. такива създадени с malloc( ) и new оператора). Регионът не се създава при стартирането, а само се пълни от съответните функции при заделяне на памет от потребителската програма.

Етапи в създаването на изпълним код

.vector_table – съдържа векторната таблица. Обикновено се разполага в началото или края на Flash (зависи от конкретния μPU). В този регион няма инструкции, а само адреси на функции. Микропроцесорното ядро знае предварително, че точно там се намира таблицата. Сегментът се създава от дебъгера във Flash, при запис на програмата в микроконтролера. Адресите могат да бъдат само извличани от микропроцесора при настъпване на прекъсване (включително и Reset).

Векторната таблица може да бъде релокирана на друго място в паметта, ако микропроцесорът има регистър за офсет на тази таблица (например ARM Cortex имат VTOR).

Етапи в създаването на изпълним код

Стартиращ код (start-up code) – първият код, който се изпълнява от микропроцесора след рестарт. Неговата функция е да създаде .data и .bss сегменти, да инициализира тактовите честоти на системата, да пусне FPU /ако има/ и да прехвърли изпълнението към main( ) функцията.

Етапи в създаването на изпълним код

```
void reset_handler(void){
   uint32_t *source;
   uint32_t *destination;

   source = &_sdata_lma;

   //.data
   for (destination = &_sdata; destination < &_edata; ){
      *destination++ = *source++;
   }

   //.bss
   for (destination = &_sbss; destination < &_ebss; ){
      *(destination++) = 0x00000000;
   }

   main();

    while(1){ }
}
```

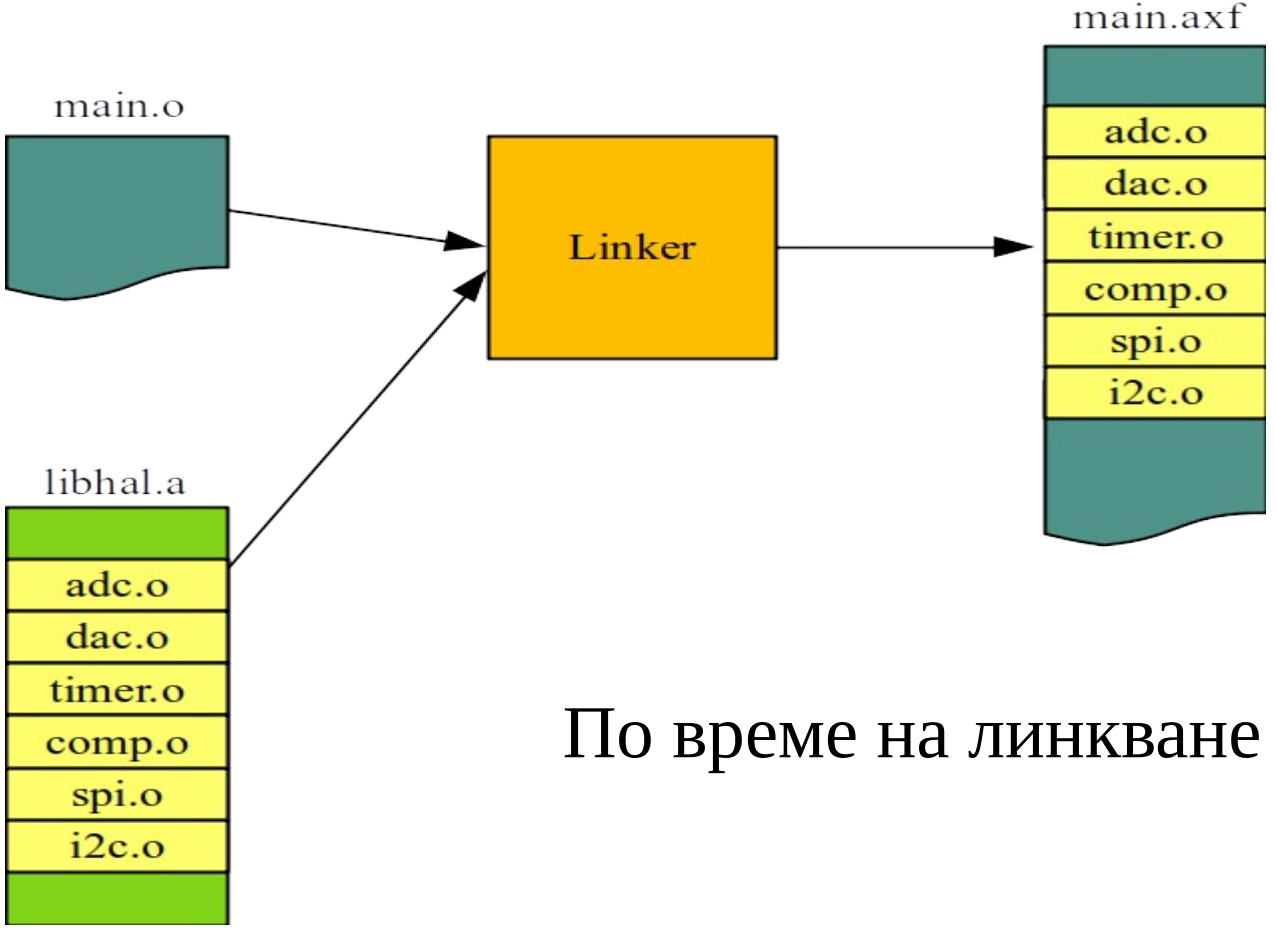
Етапи в създаването на изпълним код

Софтуерна библиотека – софтуерен архив от предварително компилирани обектови файлове, които се използват от потребителския фърмуер.

Съществуват два вида библиотеки:
*статични (static library, .a, .lib)
*динамични:
- → динамично-линкнати (dynamically linked, .so, .dll)
- → динамично-заредени (dynamically loaded)

Етапи в създаването на изпълним код

Статична библиотека – обектовият код на библиотеката се линква с обектовия код на потребителския фърмуер и става част от крайния двоичен файл. Използва се в bare-metal фърмуер (т.е. системен софтуер без операционна система).



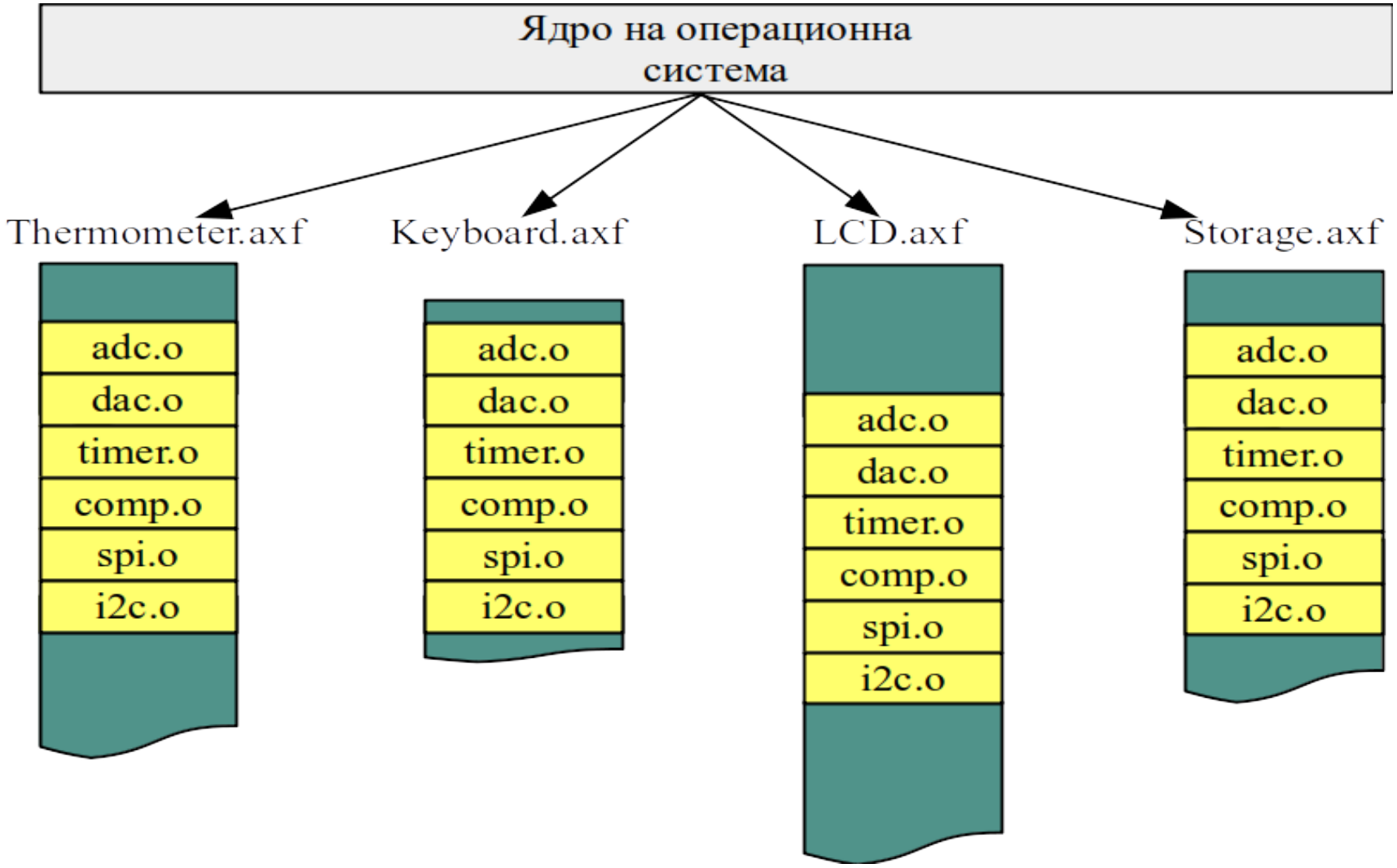
По време на линкване (link-time).

Етапи в създаването на изпълним код

Недостатък – в многозадачна система може да се използват статични библиотеки, но **обектовият код ще се копира във всяка една задача**, отнемайки от паметта на системата.

Недостатък – ако се направи корекция в библиотеката, тя трябва да се компилира и архивира наново, след което **да се линкне със всяка една програма, която я използва**.

Етапи в създаването на изпълним код



По време на изпълнение (run time, multi-thread).

Етапи в създаването на изпълним код

Динамично-линкната библиотека – обектовият код трябва да се подаде на линкера по време на линкването, но не се добавя към крайния изпълним файл. Вместо това, к**огато се стартира програмата**, процес на операционната система (наречен динамичен зареждач, dynamic linker) ще потърси обектовия код на библиотеката в системни директории (или предварително указани системни променливи, в Линукс - LD_LIBRARY_PATH) и ако намери такъв файл, обектовия код на потребителския софтуер ще може да вика функции от тази библиотека. Този процес е “невидим” за потребителския код. Той “вижда” същото, което би видял със статична библиотека [1]. Използваните библиотеки са записани в потребителския софтуер.

Този вид библиотеки се използва с операционни системи.

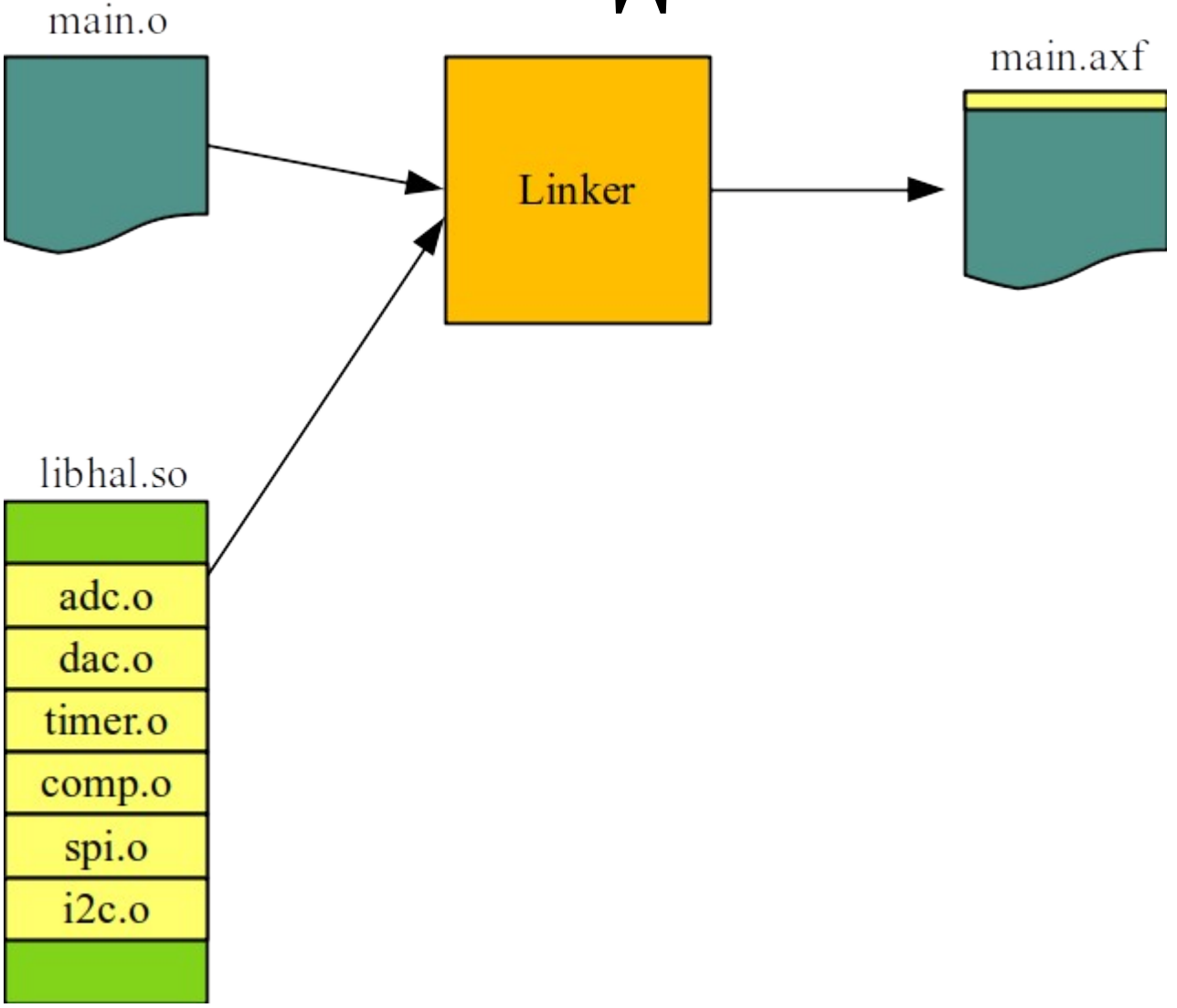
Етапи в създаването на изпълним код

Предимство **-** веднъж заредена, библиотеката **може да бъде използвана и от други процеси** [2], [3], [4]. Това е възможно, защото се прави по едно копие на променливите от библиотеката (.data сегмента) за всяка нишка. Сегментът с инструкции (.text) си остава само един. Ако кодът си взаймодейства с хардуер, библиотеката трябва да е безопасна за многозадачно изпълнение (thread-safe), т.е. да се направят проверки, че в даден момент само 1 нишка достъпва ресурс.

Предимство – ако се наложат корекции на библиотеката, само нейния обектов код трябва да бъде подменен в системата. Приложенията, които я използват **няма нужда да се компилират** и **линкват наново**.

Недостатък – поради операцията “динамично линкване”, **изпълнението на програмата се забавя** спрямо варианта със статични библиотеки.

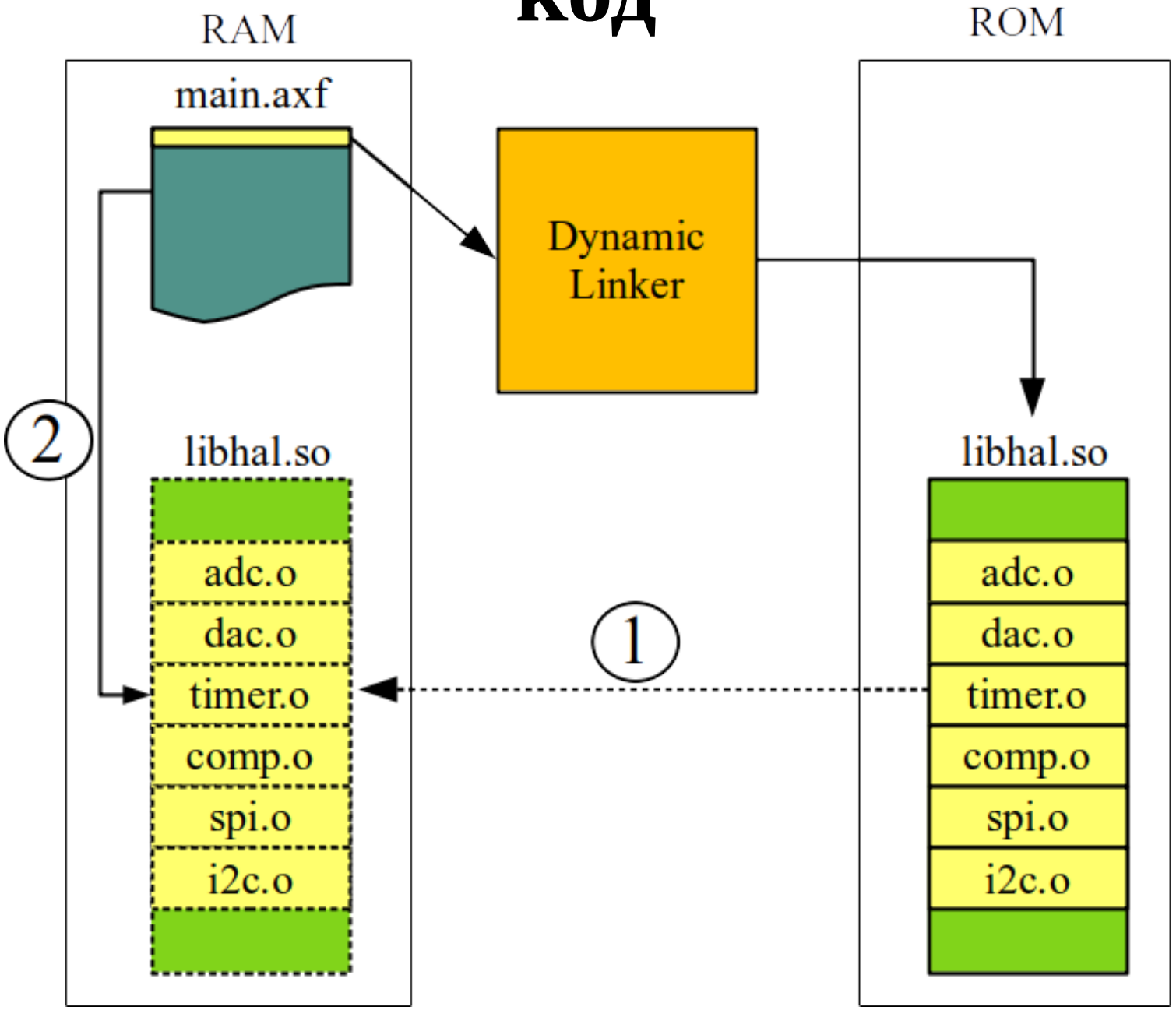
Етапи в създаването на изпълним код



По време на линкване (link-time).

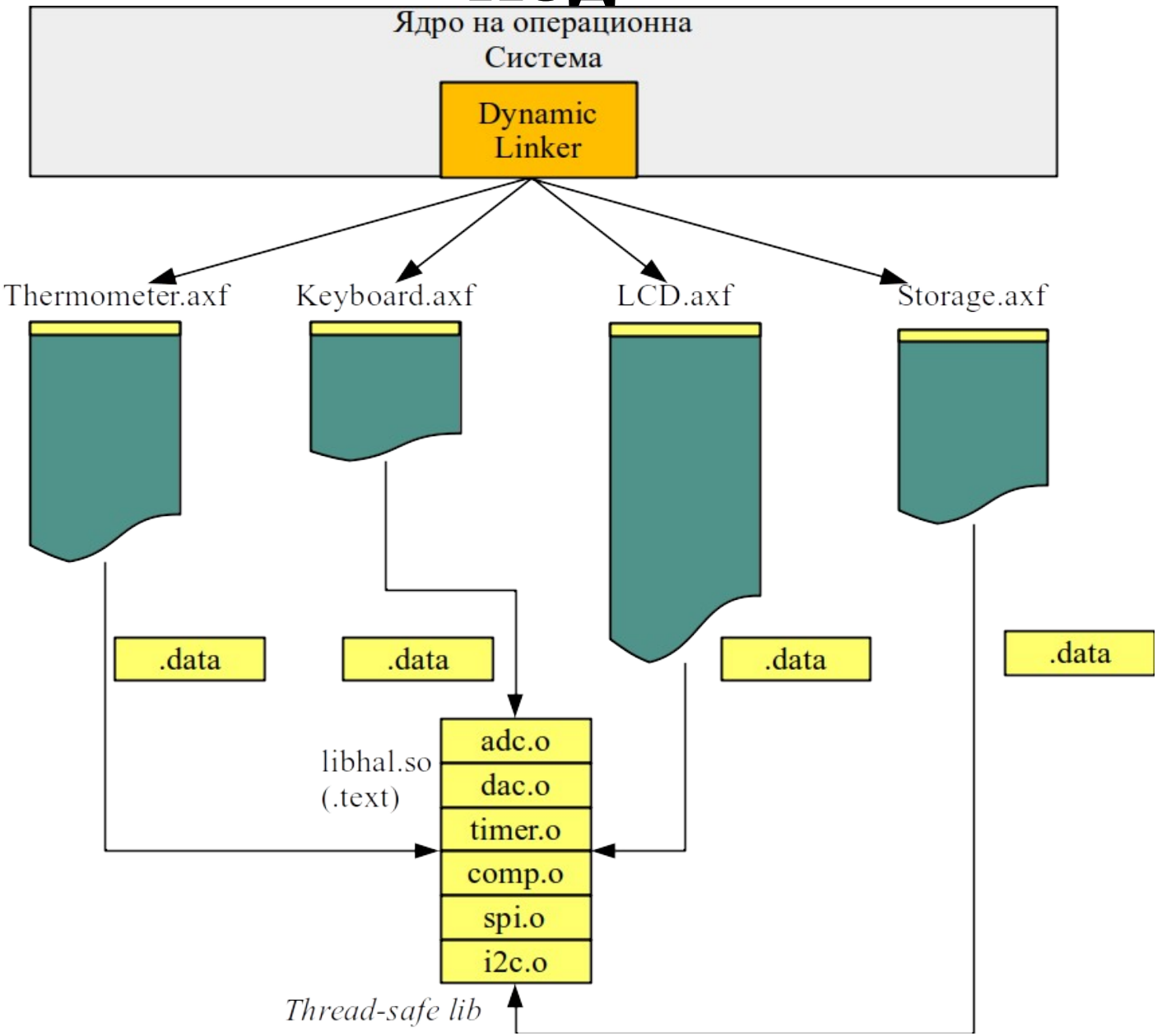
Етапи в създаването на изпълним код

По време на изпълнение (run time).



Етапи в създаването на изпълним код

По време на изпълнение (run time, multi-thread).



Етапи в създаването на изпълним код

Код с независима позиция (**P**osition **I**ndependent **C**ode, PIC) – код, асемблерният еквивалент на който не използва абсолютна адресация, а само символна (symbolic, PC-relative) Динамичните библиотеки трябва да се компилират като PIC, защото при стартиране на програмата, не се знае точно къде динамичният линкер ще релокира обектовия им код.

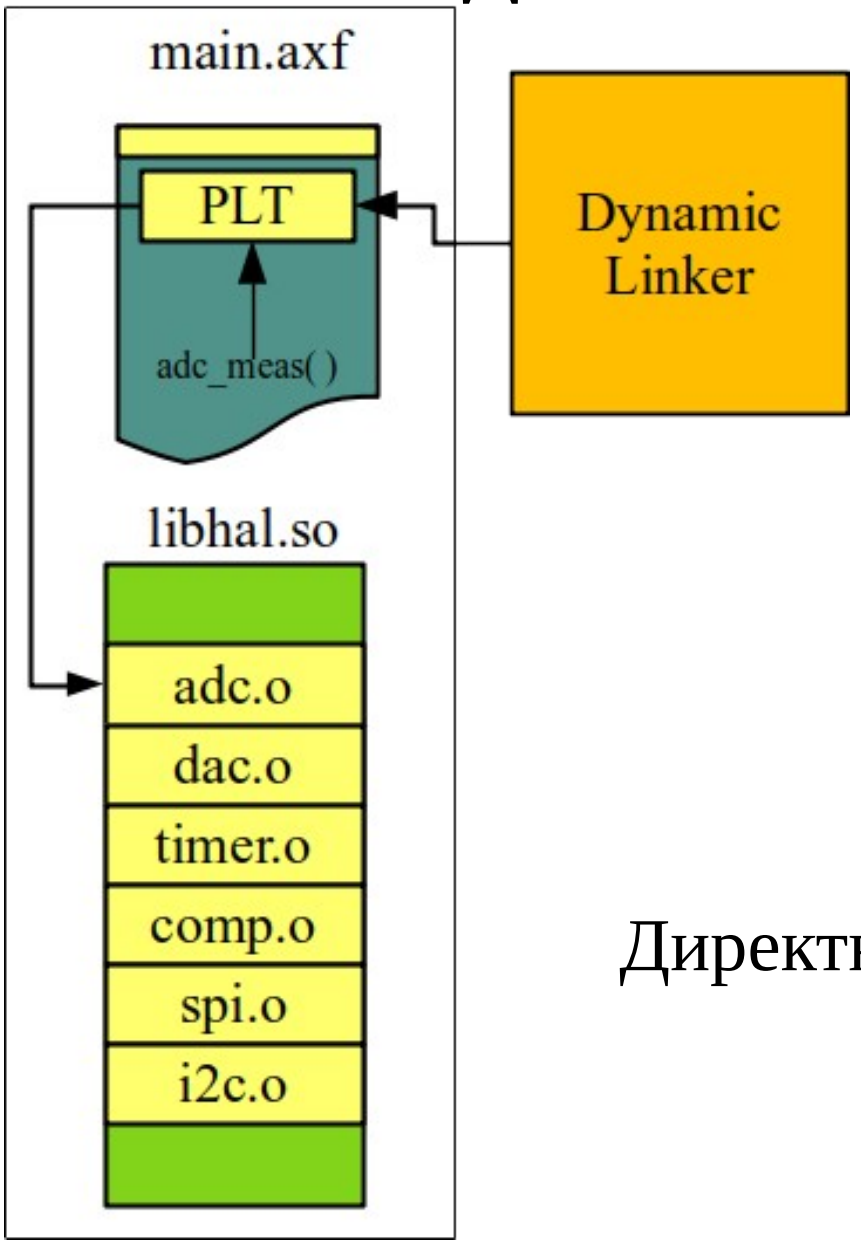
Етапи в създаването на изпълним код

Процедурна таблица (**P**rocedure **L**inkage **T**able, PLT) – малък регион код, внедрен в потребителската програма, който се попълва с адреси на библиотечни функции от динамичния линкер при стартиране на програмата. Използва се, когато в потребителската програма **има извикване на функция** от библиотеката.

Позволени са преходи навсякъде в 32-битовото поле (veneer).

За всяко извикване на библиотечна функция се поставя по една PLT.

Етапи в създаването на изпълним код

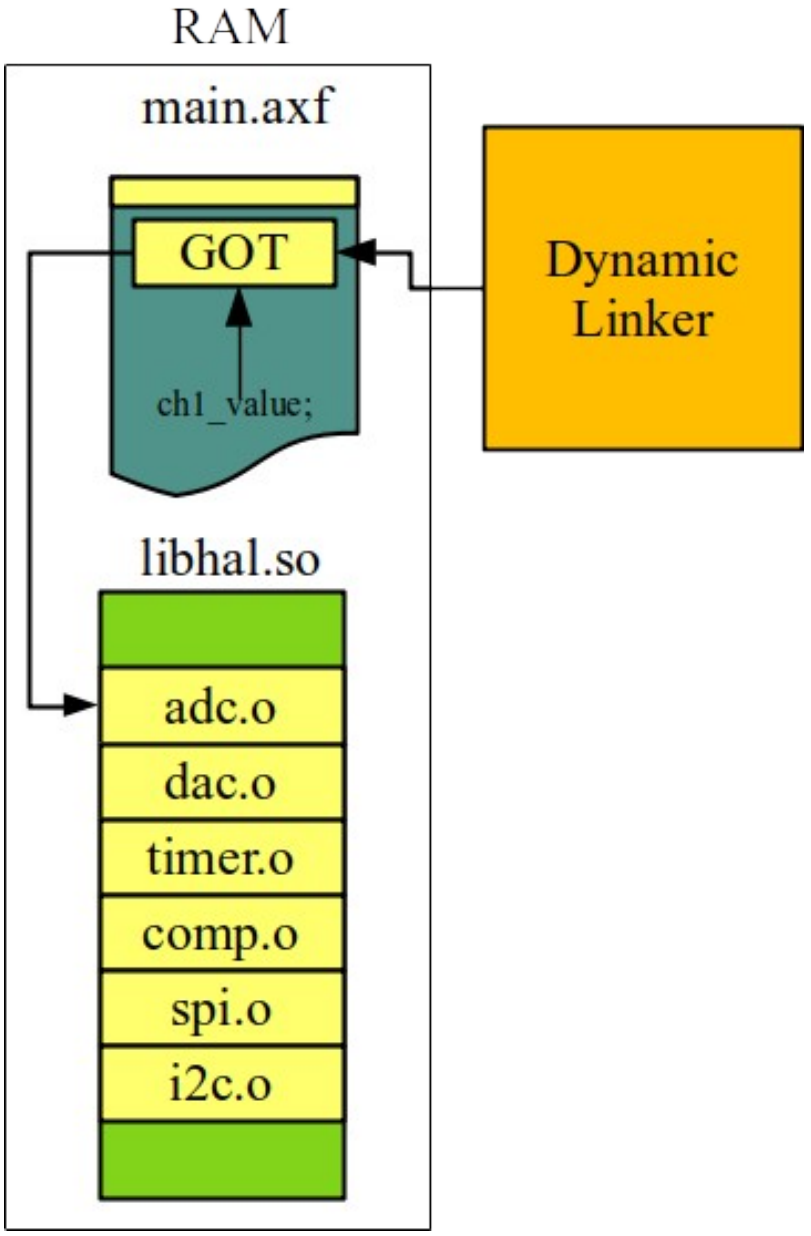


Директна PLT таблица.

Етапи в създаването на изпълним код

Глобална таблица с отмествания (**G**lobal **O**ffset **T**able, GOT) **-** малък регион с данни, внедрен в потребителската програма, който се попълва с адреси на библиотечни променливи от динамичния линкер при стартиране на програмата. Използва се, когато в потребителската програма **има достъп до глобална променлива** от библиотеката.

Етапи в създаването на изпълним код



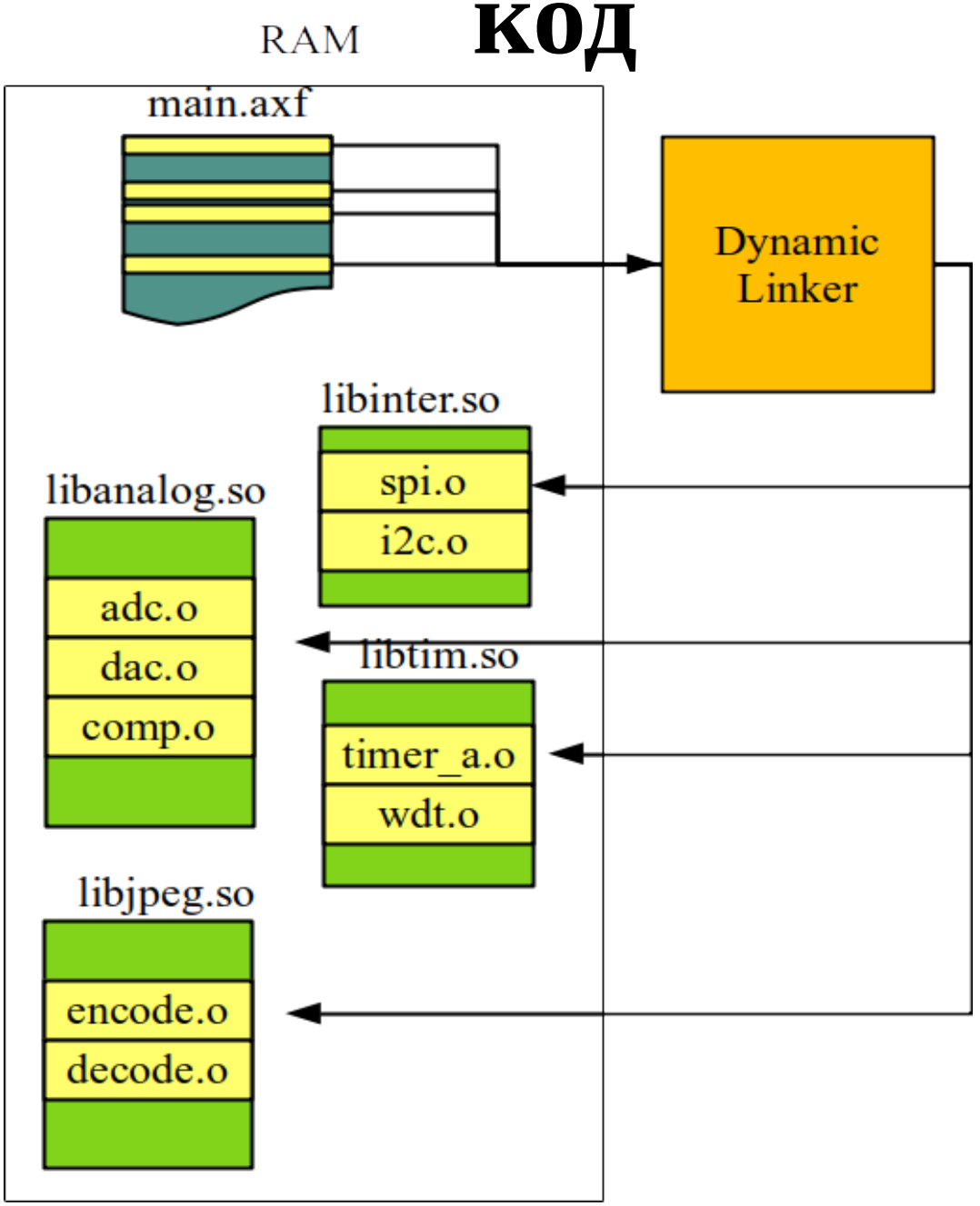
Етапи в създаването на изпълним код

Динамично-заредена библиотека – обектовият код не се добавя към крайния изпълним файл. Вместо това в потребителската програма се поместват API функции, които зареждат съответните библиотеки. Този процес е “видим” за потребителския код.

Прилича на зареждането на “плъгин” и е по-гъвкаво от динамично-линкнатата библиотека, защото може да се правят проверки дали библиотеката съществува, както и да се взимат динамично решения дали въобще да бъде заредена библиотеката.

Този вид библиотеки се използва с операционни системи.

Етапи в създаването на изпълним код



Етапи в създаването на изпълним код

Пример – под Линукс, динамично-зареждаеми API са [2]:

```
#include <dlfcn.h>

//Прави обектов файл на библиотека да бъде достъпен за програмата.
void *dlopen( const char *file, int mode );

//Извлича указател към символ (функция/променлива) по име, което се задава като низ
void *dlsym( void *restrict handle, const char *restrict name );

//Връща последно възникналата грешка
char *dlerror();

//Известява OS, че библиотеката повече няма да се използва
char *dlclose( void *handle );
```

Откриване на грешки в кода

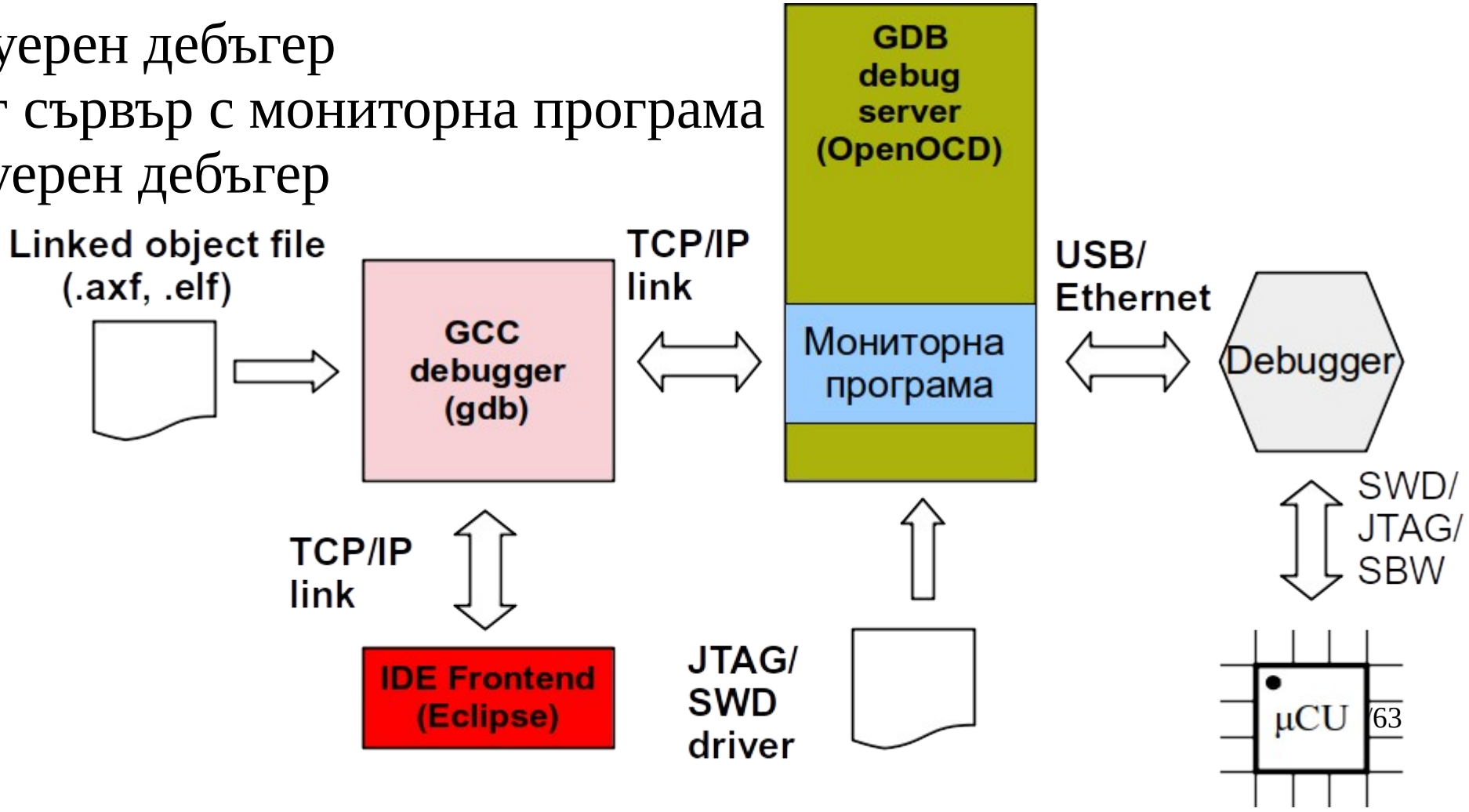
Дебъгване (debugging) – процес на откриване и отстраняване на грешки в кода.

Три важни компонента:

*софтуерен дебъгер

*дебъг сървър с мониторна програма

*хардуерен дебъгер



Откриване на грешки в кода

Софтуерен дебъгер – прави връзката между конструкция на C и асемблерните инструкции, за да може да изпълнява операциите, стъпкуване, наблюдаване на регистри и др.

Пример – дебъгерът GDB от GNU toolchain. Поддържа командите:
*file – зареди линкнат обектов файл, съдържащ дебъгерна информация (.axf, .elf);
*load – изпрати команда на мониторната програма за запис на фърмуера във Flash
*monitor [ABCD] – изпрати командата ABCD към мониторната програма;
* break [име-на-функция] – постави точка на прекъсване;
*continue – пусни програмата да се изпълнява до безкрай (run);
*step – напредни с една C конструкция напред (step over);
*finish – излез от тялото на функцията (step out);
*p [име-на-променлива] – покажи стойността на променлива;
*други

Откриване на грешки в кода

Дебъг сървър с мониторна програма – извършва операции на ниско ниво, като знае особеностите на конкретния микропроцесор. Този вид операции дублират в голяма степен операциите, които софтуерния дебъгер поддържа, но тук се познават конкретни адреси на регистри от JTAG/SWD модула, конкретни регистри на микропроцесора и конкретни регистри на контролера на Flash паметта.

*Дебъг сървърът стартира сървърно приложение и очаква команди от софтуерния дебъгер. Сървърът поддържа командите на софтуерния дебъгер.
*Мониторна програма – преобразува командите на софтуерния дебъгер в команди за хардуерния дебъгер. Мониторът трябва да знае конкретния микроконтролер, с който се работи.
*Драйвер за комуникация – използва се, за да се осъществи връзка по съответния интерфейс (най-често USB, Ethernet) с хардуерния дебъгер.

Откриване на грешки в кода

Пример – команди поддържани от GDB сървъра OpenOCD:
*halt – задръж микропроцесора на един адрес;
*step – придвижи микропроцесора една инструкция напред;
*resume – пусни програмата да се изпълнява до безкрай;
*reset – рестартирай микропроцесора;
*md [0x1234] – покажи съдържанието на регистър на адрес “0x1234”;
*mw [0x1234] – запиши (ако не е read-only) стойност в регистър на адрес “0x1234”;
*flash mass_erase – изтрии цялата Flash памет;
*flash write_image – запиши файл във Flash паметта на микроконтролера.

Откриване на грешки в кода

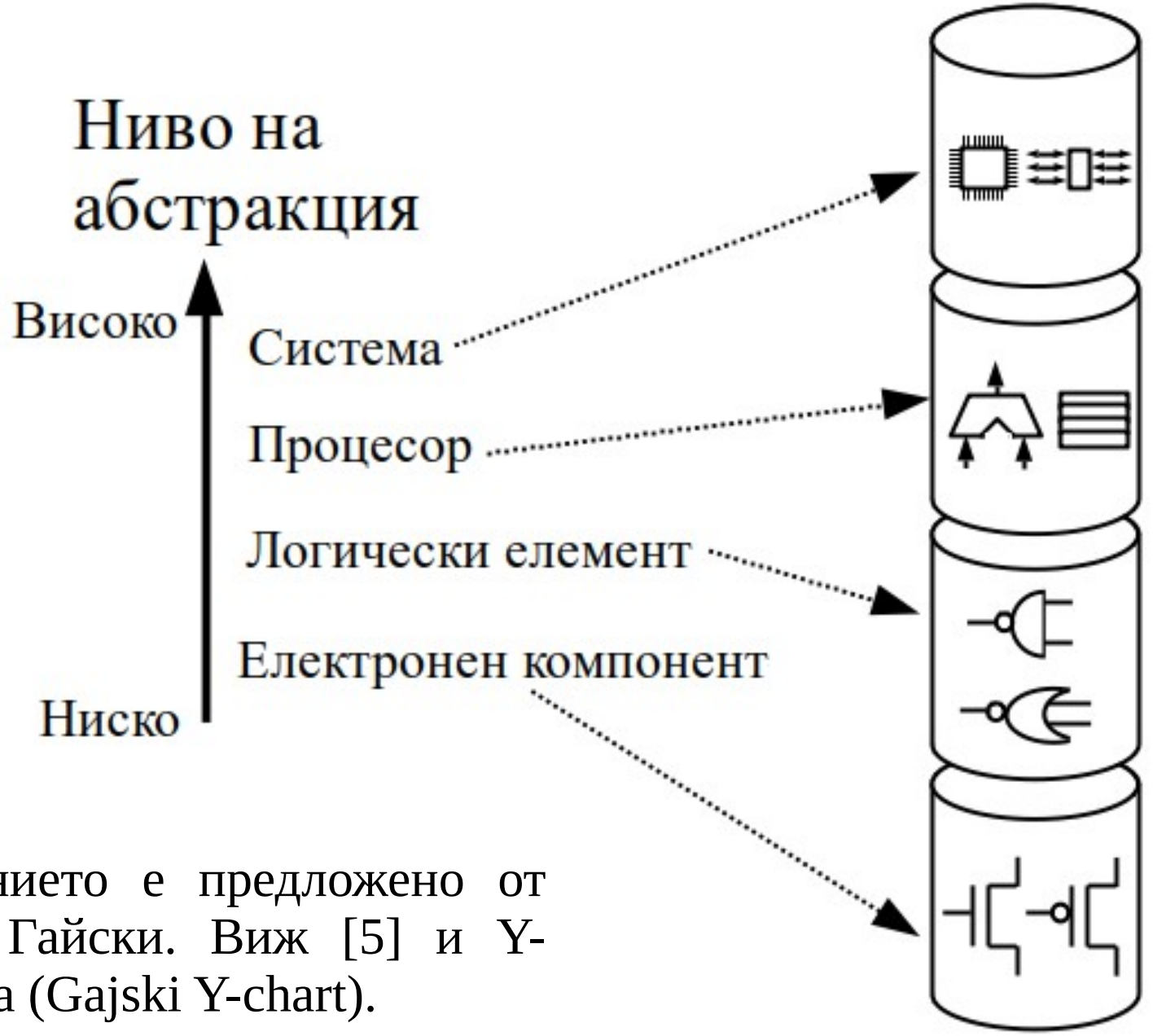
Хардуерен дебъгер – както беше споменато в лекцията за паметите, хардуерният дебъгер осъществява връзката между мониторната програма на дебъг сървъра и дебъг контролера на микроконтролера. Той прави преобразуване на:
*електрическите нива;
*кодирането на електрическите нива;
*протоколи за комуникация.

Проектиране на вградени системи

Проектирането на вградени системи може да стане на 4 нива на абстракция [5]:
*ниво транзистор;
*ниво логически елемент;
*ниво регистър (**R**egister **T**ransfer **L**evel, RTL);
*ниво система (**E**lectronic **S**ystem **L**evel, ESL, понякога се среща “поведенческо описание”, behavioral description).

В настоящия курс се работи на RTL ниво.

Проектиране на вградени системи



Разделението е предложено от Даниел Гайски. Виж [5] и Y-диаграма (Gajski Y-chart).

Проектиране на вградени системи

Нивото, на което работи един проектант, може да бъде определено по библиотеките, които използва той/тя:

*ако се използват библиотеки с транзистори, резистори, кондензатори – нивото е транзисторно;

*ако се използват библиотеки с комбинационна логика (И, ИЛИ, ИИЛИ, ИЛИ-НЕ, НЕ, и т.н.) - нивото е логически елемент;

*ако се използват библиотеки с последователностна логика (тригери, брочи, паралелни/премествящи регистри, АЛУ, барелни премествачи, и т.н.) - нивото е регистрово;

* ако се използват библиотеки с микропроцесори, памети, хардуерни ускорители, магистрални мултиплексори, и т.н.) - нивото е системно.

Проектиране на вградени системи

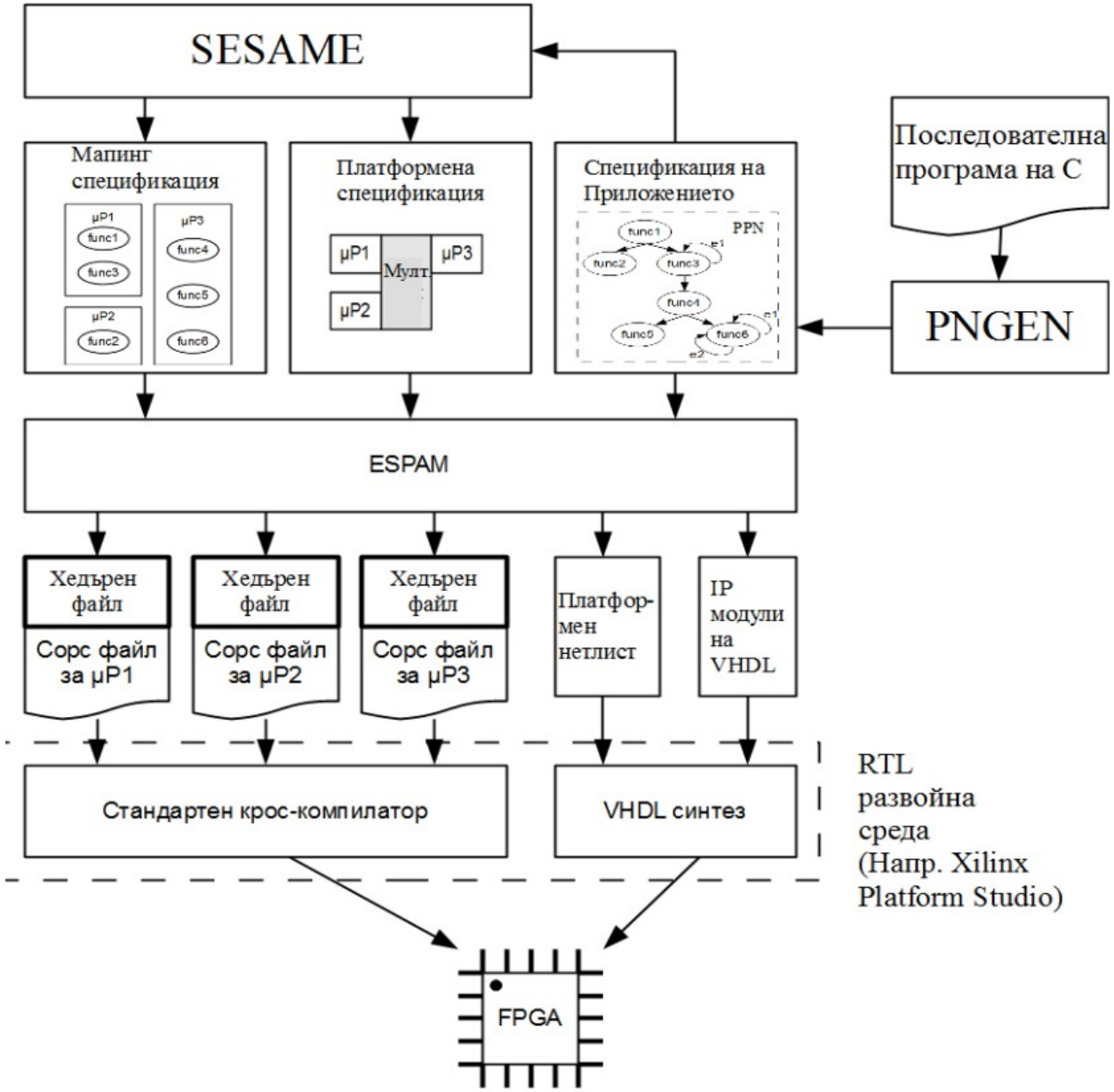
Първите три нива са добре познати. Последното четвърто, ESL, е най-новото и най-често разработваното в последните години.

Примерна реализация на среда с отворен код за развой на ESL ниво е показана на следващия слайд. С нея се проектират вградени многопроцесорни системи върху чип (MPSoC).

Това е средата Дедал (Daedalus Design Framework), разработена в Лайденски Университет, Нидерландия.

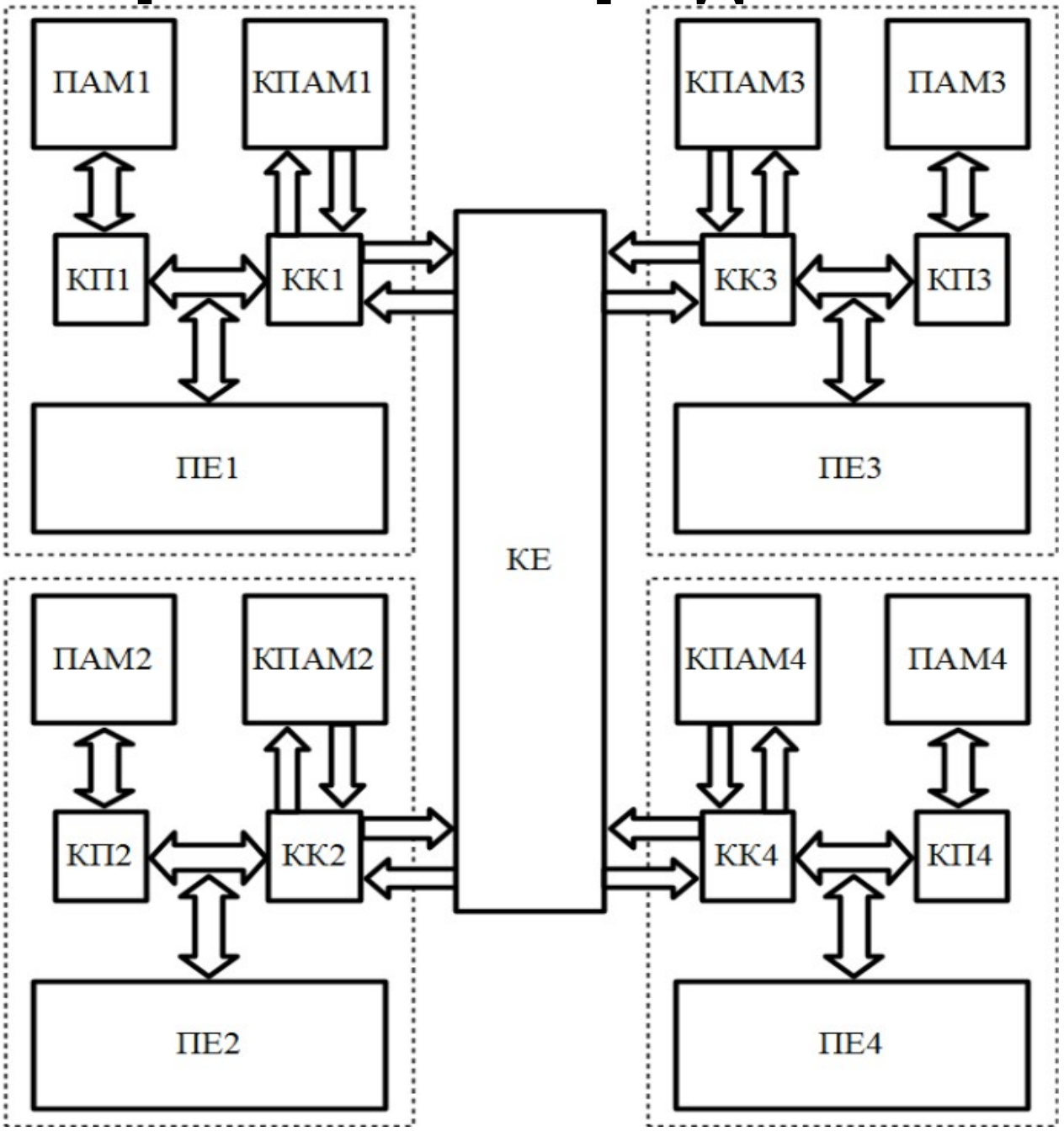
Има сериозно българско участие.

Проектиране на вградени системи



Фиг. 5.2 – Развойна среда ДЕДАЛ за синтез на системи от високо ниво.

Проектиране на вградени системи



Литература

[1] G. Matzigkeit, A. Oliva, T. Tanner, G. Vaughan, “GNU libtool”, pp. 48 – 51, v.2.4.6, Free Software Foundation Inc, 2015.
[2]M. Jones, “Anatomy of Linux dynamic libraries”, 2008.
https://developer.ibm.com/tutorials/l-dynamic-libraries/
[3] “Dynamic Linking with the ARM Compiler toolchain”, Application note 242, DAI0242A, ARM Ltd, 2010.
[4] Ian Wienand, “PLT and GOT – the key to code sharing and dynamic libraries”, online, 2021.
https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html
[5] D. Gajski *et al*, “Embedded System Design: Modeling, Synthesis and Verification”, Springer Science + Business Media, LLC 2009