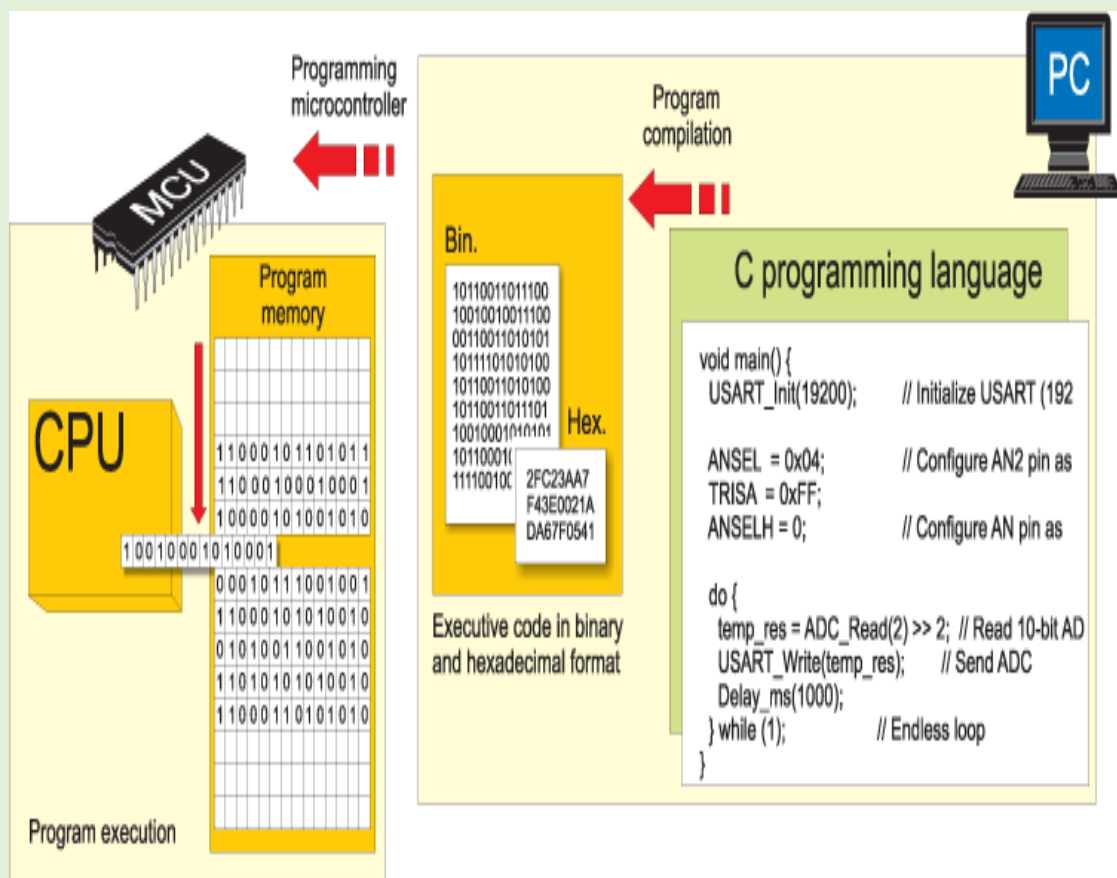


Програмиране на микроконтролери на езика C



Димо Петков

Програмиране на микроконтролери на езика C

Първо издание

София, 2016 г.

Всички права запазени. Нито една част от тази книга не може да бъде размножавана или предавана под никаква форма или начин, електронен или механичен, включително фотокопиране, записване или чрез каквито и да е системи за съхранение на информация, без предварително писмено разрешение от автора.

© Димо Петков Петков, автор

www.progstarter.com

ISBN xxxx-xxxx-xxxx-xxxx

Кратко съдържание:

За автора	15
От автора	15
Предговор	16
За кого е предназначена тази книга	19
Защо микроконтролера Z8F083A.....	19
Използвани конвенции	21
Как е организирана книгата.....	22
Част I: Описание на микроконтролера Z8F083A	25
1 Основни характеристики	27
2 Описание на процесора eZ8	29
3 Описание на изводите	47
4 Адресно пространство	51
5 Карта на регистровата памет.....	52
6 Ресет и излизане от STOP-режим	56
7 Контролер на прекъсванията.....	65
8 Генераторен блок.....	76
9 Режими с ниска консумация.....	87
10 Портове.....	91
11 WDT таймер.....	105
12 Таймери	112
13 Компаратор	142
14 Аналого-цифров преобразувател.....	146
15 Флаш-памет.....	152
16 Флаш-конфигурационни битове	165
17 Енерго-независима памет за данни (NVDS).....	168
18 Дебъгер	171
19 Електрически параметри	175
20 Информация за поръчка.....	176
Част II: Кратък C справочник	179
21 Въведение.....	181
22 Основи	184
23 Типове данни	212
24 Константи.....	291
25 Операции	302
26 Преобразувания	303
27 С оператори.....	304
28 Функции	305
29 Предпроцесор	306

30 Стандартна C библиотека	307
31 C програмни техники	310
Част III; Правила и съвети при писане на C-код	311
32 Въведение.....	313
33 Стил на писане на C-програми.....	315
34 Слабости в езика C	316
35 Тестване на C код	317
Част IV: Описание на развойните средства.....	319
36 Общи сведения	321
37 Описание на основната платка Z8Encore_MB01.....	322
38 Описаниена разширителната платка Z8Encore_EB01(MB01)	328
39 Описание на развойната среда Zilog Developer Studio II.....	332
Част V: Въведение в програмирането на микроконтролера Z8F083A на езика C	435
40 Предисловие.....	437
41 Управление на светодиода (вариант 1)	438
42 Управление на светодиода (вариант 2)	448
43 Управление на светодиода (вариант 3)	453
44 Управление на светодиода (вариант 4)	460
45 Управление на 7-сегментен дисплей	471
46 Четене на бутон.	502
47 Измерване на период (честота)	518
48 Измерване на коефициент на запълване	534
49 Детектор на ниско напрежение	549
50 Измерване на напрежение	553
51 Заключение	564
Част VI: Описание на драйверите за микроконтролера Z8F083A	566
Част VII: Практически примери.....	568
Пример1. Четене на матрична клавиатура. Реализация на кодова брава.	570
Пример2. Управление на буквено-цифров LCD-дисплей (Hitachi HD44780 LCD-контролер).	571
Пример3. Управление на графичен LCD-дисплей (Nokia 5110 LCD- дисплей). Комуникация по SPI шина.	572
Пример4. Аналогов термометър с AD22103.	573
Пример5. Съвързване на външна EEPROM-памет. Комуникация по I2C шина.	574

Пример6. Цифров термометър с DS18B20.	575
Пример7. Влагомер с DHT11.	576
Пример8. Инфрочервен термометър с MLX90614.....	577
♦ Приложение А	578
♦ Приложение В	580
Заклучение.....	581

Съдържание:

За автора	15
От автора	15
Предговор	16
За кого е предназначена тази книга.....	19
Защо микроконтролера Z8F083A	19
Използвани конвенции	21
Как е организирана книгата	22
Част I	25
Описание на микроконтролера Z8F083A	25
1 Основни характеристики	27
2 Описание на процесора eZ8	29
2.1 Организация на паметта	29
2.2 Програмен модел	33
2.3 Стек	36
2.4 Прекъсвания	38
2.4.1 Общи сведения	38
2.4.2 Обработка на разрешено прекъсване	39
2.4.3 Обработка на неразрешено прекъсване	42
2.4.4 Вложени прекъсвания	43
2.4.5 Генериране на софтуерни прекъсвания.....	45
2.4.6 Аварийни прекъсвания	45
3 Описание на изводите	47
4 Адресно пространство	51
5 Карта на регистровата памет.....	52
6 Ресет и излизане от STOP-режим	56
6.1 Източници и типове ресет	56
6.2 Описание на източниците на ресет	58
6.3 Излизане от STOP-режим.....	60
6.4 Описание на управляващите регистри.....	62
Регистър RSTSTAT (Reset Status).....	62
7 Контролер на прекъсванията.....	65
7.1 Общи сведения	65
7.2 Описание на управляващите регистри.....	69
Регистър IRQ0 (Interrupt Request 0).....	69
Регистър IRQ1 (Interrupt Request 1).....	70
Регистър IRQ2 (Interrupt Request 2).....	70
Регистър IRQ0ENH (IRQ0 Enable High Bit).....	71
Регистър IRQ0ENL (IRQ0 Enable Low Bit).....	71
Регистър IRQ1ENH (IRQ1 Enable High Bit).....	72
Регистър IRQ1ENL (IRQ1 Enable Low Bit).....	72

Регистър IRQ2ENH (IRQ2 Enable High Bit).....	72
Регистър IRQ2ENL (IRQ2 Enable Low Bit).....	73
Регистър IRQES (Interrupt Edge Select)	73
Регистър IRQSS (Shared Interrupt Select)	74
Регистър IRQCTL (Interrupt Control)	74
8 Генераторен блок.....	76
8.1 Общи сведения	76
8.2 Детектиране и възстановяване на повреден системен тактов сигнал	78
8.3 Кварцов генератор	79
8.4 Вътрешен прецизен генератор.....	82
8.5 Описание на управляващите регистри.....	82
Регистър OSCCTL (Oscillator Control Register)	82
8.6 Избор и превключване на системния тактов сигнал	84
9 Режими с ниска консумация.....	87
9.1 Общи сведения	87
STOP-режим	87
HALT-режим.....	88
9.2 Описание на управляващите регистри.....	89
Регистър PWRCTL0 (Power Control Register 0).....	89
10 Портове.....	91
10.1 Общи сведения	91
10.2 Алтернативни функции на изводите.....	91
10.3 Управляващи регистри.....	95
10.4 Описание на управляващите регистри	96
Регистър PxADDR (Port Address)	96
Регистър PxCTL (Port Control)	97
Регистър PxDD (Port Direction)	97
Регистър PxAF(Port Alternate Function).....	98
Регистър PxAFS1 (Port Alternate Set 1)	99
Регистър PxOC (Port Output Control).....	99
Регистър PxHDE (Port High Drive Enable)	100
Регистър PxSMRE (Port Stop Mode Recovery Enable).....	100
Регистър PxPUE (Port Pull-up enable)	101
Регистър PxIN (Port Input Data).....	101
Регистър PxOUT (Port Output Data).....	102
Регистър LEDEN (LED Drive Enable).....	103
Регистър LEDLVLH (LED Drive Level High)	103
Регистър LEDLVLL (LED Drive Level Low)	104
11 WDT таймер.....	105
11.1 Общи сведения	105
11.2 Устройство и принцип на работа	105
11.3 Реакция при препълване на WDT-таймера	107

11.4	Запис в регистрите WDTU, WDTN и WDTL	108
11.5	Описание на управляващите регистри	109
	Регистър WDTCTL (Watchdog Timer Control).....	109
	Регистър WDTU (Watchdog Timer Reload Upper Byte).....	109
	Регистър WDTN (Watchdog Timer Reload High Byte).....	110
	Регистър WDTL (Watchdog Timer Reload Low Byte).....	110
12	Таймери	112
12.1	Общи сведения	112
12.2	Устройство и принцип на работа	112
12.3	Режими на работа.....	113
	Еднократен режим (single-shot mode)	113
	Непрекъснат режим (continuous mode)	115
	Броячен режим	116
	Компараторен броячен режим	118
	Едноизходен PWM режим.....	119
	Двуизходен PWM режим.....	123
	Режим на прехващане (capture mode).....	124
	Режим на прехващане с рестартиране.....	126
	Режим на сравнение (compare mode).....	128
	Режим на контролирано броене (gated mode).....	130
	Режим на прехващане/сравнение	132
12.4	Четене на текущата броячна стойност на таймера	133
12.5	Описание на управляващите регистри	134
	Регистри TxH (Timer High Byte) и TxL (Timer Low Byte).....	134
	Регистри TxRH (Timer Reload High Byte) и TxRL (Timer Reload Low Byte)	135
	Регистри TxPWMH (PWM High Byte) и TxPWML (PWM Low Byte).....	136
	Регистър TxCTL0 (Timer x Control 0).....	137
	Регистър TxCTL1 (Timer x Control 1).....	138
13	Компаратор	142
13.1	Общи сведения	142
13.2	Устройство и принцип на работа	142
13.3	Описание на управляващите регистри	144
	Регистър CMP0 (Comparator Control Register).....	144
14	Аналого-цифров преобразувател	146
14.1	Общи сведения	146
14.2	Устройство и принцип на работа	146
14.3	Описание на управляващите регистри	148
	Регистър ADCCTL0 (ADC Control 0).....	148
	Регистър ADCD_H (ADC Data High Byte).....	149
	Регистър ADCD_L (ADC Data Low Bits)	150
	Регистър ADCSST (ADC Settling Time).....	150

Регистър ADCST (ADC Sample Time).....	151
Регистър ADCCP (ADC Clock Prescale)	151
15 Флаш-памет.....	152
15.1 Общи сведения.....	152
15.2 Флаш-информационна област	155
15.3 Флаш-контролер	155
15.4 Описание на управляващите регистри	160
Регистър FCTL (Flash Control).....	160
Регистър FSTAT (Flash Status).....	161
Регистър FPS (Flash Page Select).....	162
Регистър FPROT (Flash Sector Protect)	162
Регистър FFREQH (Flash Frequency High).....	163
Регистър FFREQL (Flash Frequency Low).....	163
16 Флаш-конфигурационни битове	165
16.1 Общи сведения.....	165
16.2 Типове конфигурационни битове	166
17 Енерго-независима памет за данни (NVDS)	168
17.1 Общи сведения.....	168
17.2 Запис на байт в NVDS	169
17.3 Четене на байт от NVDS	170
18 Дебъгер.....	171
18.1 Общи сведения.....	171
18.2 Свързване на дебъгера към микроконтролера	172
18.3 Тестване/дебъгване с USB SmartCable	172
19 Електрически параметри	175
20 Информация за поръчка.....	176
Част II	179
Кратък С справочник	179
21 Въведение.....	181
22 Основи	184
22.1 Структура на С програма	184
22.2 Среда	192
22.2.1 Транслираща среда.....	192
22.2.2 Изпълняваща среда	195
22.3 Характеристики на средата.....	197
22.3.1 Символни таблици	197
22.3.2 Диграфи и Триграфи	200
22.3.3 Универсални символи.....	201
22.3.4 Широки и многобайтови символи	203
22.4 Лексеми.....	204
22.5 Ключови думи	205
22.6 Идентификатор	205
22.7 Константи	208

22.8	Операции и Пунктуатори.....	208
22.9	Имена на хедър-файлове.....	209
22.10	Предпроцесорни числа.....	210
22.11	Коментари.....	210
23	Типове данни.....	212
23.1	Променливи.....	212
23.2	Категории типове.....	214
23.2.1	Базови типове данни.....	218
23.2.2	Изброявания.....	221
23.2.3	Масиви.....	225
	Едномерни масиви.....	225
	Многомерни масиви.....	227
	Операции с масиви.....	230
	Абстрактен изглед.....	232
23.2.4	Указатели.....	236
	Указатели към обекти.....	236
	Указатели към функции.....	237
	Обобщени указатели.....	239
	Нулеви указатели.....	239
	Операции с указатели.....	240
	Връзка между указатели и масиви.....	249
23.2.5	Структури.....	250
	Деклариране на структура.....	250
	Операции със структури.....	254
	Абстрактен изглед.....	256
23.2.6	Обединения.....	257
	Деклариране на обединение.....	257
	Операции с обединения.....	262
23.2.7	Битови полета.....	265
	Дефиниране на битово поле.....	265
	Ограничения на битовите полета.....	270
23.3	Квалификатори на типа.....	271
23.4	Област на видимост на променливите.....	273
23.5	Време на живот на променливите.....	278
23.6	Клас памет на променливите.....	278
23.7	Инициализация на променливите.....	289
24	Константи.....	291
24.1	Целочислени константи.....	291
24.2	Константи с плаваща запетая.....	293
24.3	Символни константи.....	295
24.4	Низови константи.....	297
25	Операции.....	302
26	Преобразувания.....	303

27 С оператори.....	304
28 Функции	305
29 Предпроцесор	306
30 Стандартна С библиотека.....	307
31 С програмни техники	310
Част III	311
Правила и съвети при писане на С-код.....	311
32 Въведение.....	313
33 Стил на писане на С-програми.....	315
34 Слабости в езика С	316
35 Тестване на С код	317
Част IV	319
Описание на развойните средства	319
36 Общи сведения	321
37 Описание на основната платка Z8Encore_MB01.....	322
38 Описаниена разширителната платка Z8Encore_EB01(MB01)	328
39 Описание на развойната среда Zilog Developer Studio II.....	332
39.1 Общи сведения.....	332
39.2 Отваряне на проект в ZDS II.....	332
39.3 Създаване на проект в ZDS II	333
39.3.1 Създаване на С проект	333
39.3.2 Добавяне на файл към проекта	335
39.3.3 Конфигуриране на проекта.....	336
39.3.3.1 Общи настройки.....	337
39.3.3.2 Настройки на Асемблера.....	339
39.3.3.3 Настройки на Компилятора	342
39.3.3.4 Настройки на ZSL-библиотеката	356
39.3.3.5 Настройки на Линкера.....	356
39.3.3.6 Настройки на Дебъгера	370
39.4 Компилиране на проекта.....	372
39.5 Описание на С-компилятора.....	373
39.5.1 Съответствие с ANSI С стандарта	373
39.5.2 Разширения спрямо ANSI С стандарта	376
39.5.2.1 Нови ключови думи за спецификатори на клас-памет	376
39.5.2.2 Модели на паметта	379
39.5.2.3 Контекст на функции.....	380
39.5.2.4 Прекъсвания	382
39.5.2.5 Разполагане на низове	383
39.5.2.6 Въмъкване на асемблерен код	385
39.5.2.7 Директиви за поставяне и подравняване	385
39.5.2.8 Изброявания тип char и short	387

39.5.2.9	Конфигуриране на флаш-конфигурационните байтове в С	388
39.5.2.10	Поддръжка на стандарта C99	388
39.5.3	Размер на типовете	390
39.5.4	Предефинирани макроси	391
39.5.5	Конвенции при извикване на функции	396
39.5.5.1	Механизъм на извикване на функция: Динамичен контекст	396
39.5.5.2	Механизъм на извикване на функция: Статичен контекст	396
39.5.5.3	Механизъм на извикване на функция: Съхраняване на параметри в регистри	396
39.5.5.4	Връщана стойност	396
39.5.5.5	Специални случаи	396
39.5.6	Извикване на асемблерни функции от С	398
39.5.7	Извикване на С-функции от Асемблера	399
39.5.8	RTL-библиотека (Run-time Library)	400
39.5.8.1	Общи сведения	400
39.5.8.2	Zilog хедър-файлове	400
39.5.8.3	Описание на Zilog функциите	402
39.5.9	Стартови файлове	411
39.5.10	Именуване на сегменти	412
39.6	Описание на Линкера	413
39.6.1	Общи сведения	413
39.6.2	Линкер-командни файлове	413
39.6.3	Линкерни команди	413
39.6.4	Линкерни изрази	413
39.7	Описание на Дебъгера	414
39.7.1	Общи сведения	414
39.7.2	Кодови индикатори на Дебъгера	415
39.7.3	Описание на менюто Debug	416
39.7.4	Описание на прозорците на дебъгера	419
39.7.4.1	Registers	419
39.7.4.2	Special Function Registers	420
39.7.4.3	Clock	421
39.7.4.4	Memory	422
39.7.4.5	Watch	423
39.7.4.6	Locals	425
39.7.4.7	Call Stack	426
39.7.4.8	Symbols	426
39.7.4.9	Disassembly	427
39.7.4.10	Simulated UART Output	428
39.7.5	Точки на прекъсване	429

39.7.5.1	Общи сведения.....	429
39.7.5.2	Вмъкване на точка на прекъсване	430
39.7.5.3	Преглед на всички точки на прекъсване	431
39.7.5.4	Преместване на курсора в точка на прекъсване	431
39.7.5.5	Разрешаване на точки на прекъсване.....	432
39.7.5.6	Забраняване на точки на прекъсване	432
39.7.5.7	Премахване на точки на прекъсване	433
Част V		435
Въведение в програмирането на микроконтролера Z8F083A на езика C		435
40	Предисловие.....	437
41	Управление на светодиодиод (вариант 1)	438
42	Управление на светодиодиод (вариант 2)	448
43	Управление на светодиодиод (вариант 3)	453
44	Управление на светодиодиод (вариант 4)	460
45	Управление на 7-сегментен дисплей	471
46	Четене на бутон.	502
47	Измерване на период (честота)	518
48	Измерване на коефициент на запълване	534
49	Детектор на ниско напрежение	549
50	Измерване на напрежение	553
51	Заклучение	564
Част VI		566
Описание на драйверите за микроконтролера Z8F083A		566
Част VII	568	
Практически примери.....		568
Пример1. Четене на матрична клавиатура. Реализация на кодова брава.		570
Описание на програмата.....		570
Описание на хардуера.....		570
Описание на софтуера		570
Пример2. Управление на буквено-цифров LCD-дисплей (Hitachi HD44780 LCD-контролер).		571
Описание на програмата.....		571
Описание на хардуера.....		571
Описание на софтуера		571
Пример3. Управление на графичен LCD-дисплей (Nokia 5110 LCD-дисплей). Комуникация по SPI шина.		572
Описание на програмата.....		572
Описание на хардуера.....		572
Описание на софтуера		572
Пример4. Аналогов термометър с AD22103.		573
Описание на програмата.....		573

Описание на хардуера.....	573
Описание на софтуера	573
Пример5. Свързване на външна EEPROM-памет. Комуникация по I2C шина.	574
Описание на програмата.....	574
Описание на хардуера.....	574
Описание на софтуера	574
Пример6. Цифров термометър с DS18B20.	575
Описание на програмата.....	575
Описание на хардуера.....	575
Описание на софтуера	575
Пример7. Влагомер с DHT11.	576
Описание на програмата.....	576
Описание на хардуера.....	576
Описание на софтуера	576
Пример8. Инфрачервен термометър с MLX90614.....	577
Описание на програмата.....	577
Описание на хардуера.....	577
Описание на софтуера	577
♦ Приложение А	578
♦ Приложение В.....	580
Заклучение	581

За автора



ДИМО ПЕТКОВ завършва Техническия Университет в Габрово през 2001г. Има магистърска степен по специалността „**Комуникационна Техника и Технологии**“. Започва кариерата си като сервизен инженер на аудио-видео техника. По-късно работи няколко години в белгийската фирма „**Мелексис**“ като приложен инженер. През цялото това време се занимава с изучаване на програмирането на микроконтролери. Понастоящем работи като С-програмист в немската компания **КОСТАЛ**, занимаваща се с разработката на електронни модули за автомобили.

От автора

Създадох тази книга не защото съм гуру в С-програмирането на микроконтролери, а поради недостига на подобна литература на българския пазар и желанието ми да променя това. Като начинаещи програмисти на микроконтролери вероятно сте се сблъскали с въпросите "Откъде да започна?", "Коя книга е подходяща за мен?". Създадох тази книга като част от поредица други книги в отговор на тези въпроси, за да спестя на начинаещите лутането в търсене на правилния път. Надявам се, че съм успял да постигна до голяма степен целта си. Може да изпращате Вашите въпроси и забележки на dimope@abv.bg.

Предговор

Развитието на микроелектрониката доведе до създаване на "интелигентни" интегрални схеми, наречени микроконтролери. Микроконтролерът е малък компютър, събран в един корпус и с ограничени ресурси в сравнение с един персонален компютър.



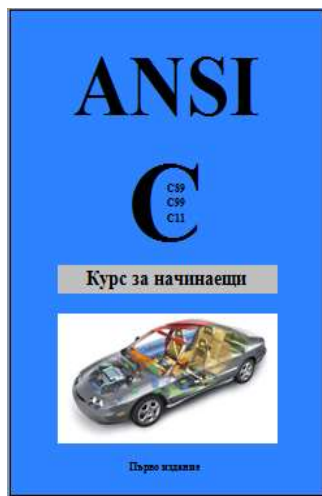
Микроконтролерите се използват във всички сфери на живота. Почти всички съвременни електронни устройства използват микроконтролери, от малки домакински уреди до печки, хладилници и т.н. Всички съвременни автомобили съдържат редица електронни модули с микроконтролери.



Терминът "интелигентен" е метафоричен, тъй като микроконтролерът се превръща в такъв само когато се програмира да върши определена задача, в противен случай

той е просто една безполезна интегрална схема. Езикът, който микроконтролерите разбират, се нарича **машинен език**. Той е набор от машинни инструкции. Една машинна инструкция е комбинация от единици (1) и нули (0). Директното използване на машинни инструкции от програмистите при създаване на програма е много труден и бавен процес. За улеснение на програмистите наборът от машинни инструкции на един микроконтролер се заменя с псевдоинструкции, които се състоят от по-лесни за разбиране и запомняне от човек символи. Псевдоинструкциите се наричат още **асемблерни инструкции**. Те се преобразуват в машинни инструкции чрез специална програма, наречена **Асемблер**. Различните микроконтролери имат различна архитектура и различен набор от машинни (съответно и асемблерни) инструкции. Например PIC микроконтролерите имат свой набор от инструкции, който се различава от набора инструкции на AVR микроконтролерите. За съжаление няма универсален асемблерен език. Това означава, че ако искате да прехвърлите програма, писана на асемблерен език за един микроконтролер - върху друг, ще трябва да я пренапишете наново, като използвате асемблерните инструкции на втория микроконтролер. Това е и един от големите недостатъци на асемблерния език. Освен това писането на асемблерен език изисква много добро познаване на архитектурата на микроконтролера. Също така, въпреки че асемблерните инструкции са по-разбираеми в сравнение с машинните, научаването и помненето им също е голямо предизвикателство. Тези недостатъци дават тласък в търсенето на други методи за създаване на по-големи и по-сложни програми. Така се появяват езици от по-високо ниво какъвто е езикът C. Езикът C предоставя едно абстрактно ниво на програмиране, което не зависи (или поне в по-голямата си част) от микроконтролера, за който се пише програмата. Именно в програмирането на микроконтролери езикът C остава незаменим, а това води и до нужда от C-програмисти. Тази книга е четвъртата от поредица книги, които ще Ви

дадат пълна информация в максимално достъпен вид, която е необходима, за да се научите да програмирате микроконтролери на езика C. Процесът включва три стъпки. Първата стъпка е да се запознаете с елементите на езика C. За тази цел е предвидена книгата **ANSI C. Курс за начинаещи**. Тя се фокусира върху базовите знания свързани



с езика C. Освен, че ще придобиете добри познания за структурата на езика ще разберете и как той функционира. За максимално улеснение на абсолютно начинаещите C-програмисти е добавена и предварителна информация, необходима за разбирането на описания материал. Книгата разглежда почти всички аспекти на езика C, но не влиза в дълбоки детайли, които биха затруднили начинаещия програмист.



Предвидена е втора книга „ANSI C. Пълен справочник”, която разглежда детайлно езика C и може да се използва за справочник дори от опитни програмисти. Прочитайки книгата „ANSI C. Пълен справочник”, Вие ще станете истински C експерти.

След като усвоите езика C, втората стъпка е да се запознаете с устройството и принципа на действие на един микроконтролер. За тази цел е предвидена книгата „Микроконтролери. Архитектура и принцип на действие”. Тази книга ще Ви запознае с представянето на



информацията в микроконтролерите, бройните системи и т.н. Тя разглежда също градивните елементи на микроконтролерите и архитектурите, по които се изграждат. Последната част на книгата ще Ви запознае с архитектурата и програмирането на асемблерен език на микроконтролера **Z8F083A**, част от фамилията [**Z8 Encore!**](#) на фирмата [**Zilog**](#).

Програмирането се учи най-вече с практика. Това е и целта на настоящата книга. Тя комбинира наученото от предходните две стъпки и ще Ви даде практически познания за софтуерните и хардуерни средства, необходими при написването, тестването и дебъгването (откриването и коригирането на грешки) на С код, на базата на микроконтролера **Z8F083A** на фирмата **Zilog**.

За кого е предназначена тази книга

Тази книга е предназначена за хора, които са запознати с езика С и искат да приложат своите знания в програмирането на микроконтролери. Ако не сте запознати с езика С, ще трябва да го направите преди да продължите. За целта може да използвате книгите [**ANSI C. Курс за начинаещи**](#) и [**ANSI C. Пълен справочник**](#). Книгата допуска също, че сте запознати с базовите архитектури и принципа на действие на микроконтролерите. Ако тези знания Ви липсват, обърнете се към книгата [**Микроконтролери. Архитектура и принцип на действие**](#).

Защо микроконтролера Z8F083A

Тъй като книгата е предназначена за начинаещи, микроконтролерът също трябва да бъде подходящ за такива.

Стартът винаги е много важен. За съжаление много начинаещи се отказват още в началото при сблъсък с първите трудности. По мое мнение основна причина за това е не само липсата на мотивация, но и неправилния подход (най-вече избор на микроконтролер със сложна архитектура и твърде академичния начин на представяне на информацията в повечето университети у нас). На пазара има богато разнообразие от микроконтролери (може да погледнете в таблицата в края на книгата, където съм направил списък с производители на микроконтролери), но не всеки е подходящ за целта. Микроконтролерът **Z8F083A** е част от фамилията микроконтролери **Z8Encore!** на фирмата Zilog. Той се характеризира с малък брой периферни модули, което го прави удобен за описване в книга. Периферните модули са значително опростени, което от своя страна позволява на начинаещия лесно и бързо да разбере как работят. Други причини, които привлякоха вниманието ми към този микроконтролер, са евтините софтуерни и хардуерни програмни средства. Развойната среда **Zilog Developer Studio II** е опростена и лесна за работа. Тя осигурява текстов редактор, асемблер и C компилатор, и е абсолютно безплатна. Дебъгерът **USB Smart Cable** е от порядъка на 40-45 евро, което е напълно приемлива цена. Всички тези неща правят серията Z8Encore! изключително подходяща за въвеждане на начинаещите в света на микроконтролерите. Въпреки че не е познат на българския пазар, реших да избира този микроконтролер. Идеята ми е да отделим колкото се може по-малко време в изучаването на конкретна архитектура и развойните й средства и да се концентрираме върху разбирането на базовите принципи. Доброто познаване на тези принципи е предпоставка за по-лесно ориентиране в различните микроконтролерни архитектури. Убеден съм, че бързото и лесно навлизане в материята ще Ви мотивира още повече да продължите сами да развивате уменията си и да обогатявате знанията си в дадената посока (*„Покажи на магарето плевнята и то само ще намери вратата”*☺). **Z8F083A**

ПОЗВОЛЯВА ВСИЧКО ТОВА ДА СЕ СЛУЧИ. Ще са Ви необходими не повече от две седмици да разучите микроконтролера и развойната среда **Zilog Developer Studio II**. След този период ще можете да се отдадете свободно на програмирането му на езика C. Ако сте ученик или студент, ще можете да използвате микроконтролера **Z8F083A** в курсови и дипломни работи. Знанията, които ще придобиете ще Ви служат за основа и ще Ви помогнат да преминете по-лесно към програмирането на микроконтролери с друга архитектура и по-висока степен на сложност.

Използвани конвенции



Текстът в това поле отбелязва някои по-важни моменти, които е добре да запомните.



Текстът в това поле отбелязва съвети, които е препоръчително да следвате.



Текстът в това поле отбелязва важна информация, на която трябва да обърнете внимание.



Текстът в това поле съдържа кратка помощна информация, свързана с описанието на теми от електрониката.

Забележка: Текстът в това поле съдържа пояснения и коментари.

текст_{опц} – суфиксът _{опц}, добавен в края на текст, е указание,

че текстът е опционален и може да се изпусне.

текст^{схх} – суфиксът ^{схх} (хх = 89, 95, 99, 11), добавен в края на текст, указва версията на стандарта, за която информацията в текста е валидна.

Как е организирана книгата

Част I описва микроконтролера Z8F083A. Ако сте чели книгата „Микроконтролери. Архитектура и принцип на действие“, нещата в тази част ще са Ви познати. Въпреки това реших, че е добре да добавя тази информация и тук за по-голямо удобство.

Част II представлява кратък справочник на езика C. Може да използвате информацията в тази част в случай, че се наложи да си припомните дадена конструкция на езика C.

Част III съдържа практически правила и съвети при писането на C-код. Това включва организация на програмата и файловете, форматиране на кода и др. Ще Ви запозная с някои слабости в езика C, които трябва да избягвате, особено при разработката на системи с повишени изисквания за безопасност.

Част IV описва развойните средства, които ще използвам в книгата. Това включва всички развойни платки. Ще се запознаете с развойната среда **Zilog Developer Studio II**, с особеностите на C-компилятора, ще се научите още как да използвате дебъгера, за да изследвате поведението на кода и да локализирате дефекти в кода.

Част V представлява въведение в програмирането на микроконтролера Z8F083A на езика C. В тази част ще започнем с няколко сравнително малки и прости примери, които ще Ви

помогнат да се запознаете с програмирането на по-голяма част от периферните модули на микроконтролера Z8F083A. С това ще поставим и начало на писането на С-код за микроконтролери.

Част VI описва пакет от драйвери¹ за микроконтролера Z8F083A.

Забележка: В случай, че сте забравили, „драйвер“ се нарича програма, която директно си взаимодейства с хардуера (периферните модули на микроконтролера), т.е. . това е програма от най-ниско ниво. Типично за всеки периферен модул на микроконтролера отговаря отделен драйвер. Приложните програми комуникират с хардуера, ако са написани професионално, чрез драйверите. Това ги прави независими изцяло или в по-голямата си част от хардуера, а това от своя страна води до по-висока степен на преносимост на приложните програми на други хардуерни платформи. Във всички случаи драйверите са тясно свързани с архитектурата на микроконтролера, което означава, че при прехвърляне на програмата на друга хардуерна платформа, драйверите трябва да бъдат пренаписани за тази хардуерна платформа. Съществуват стандарти, които определят начина на писане на драйверите. Такъв например е стандартът AUTOSAR (www.autosar.org). Стандартизирането на драйверите прави приложните програми напълно независими от хардуерната платформа.

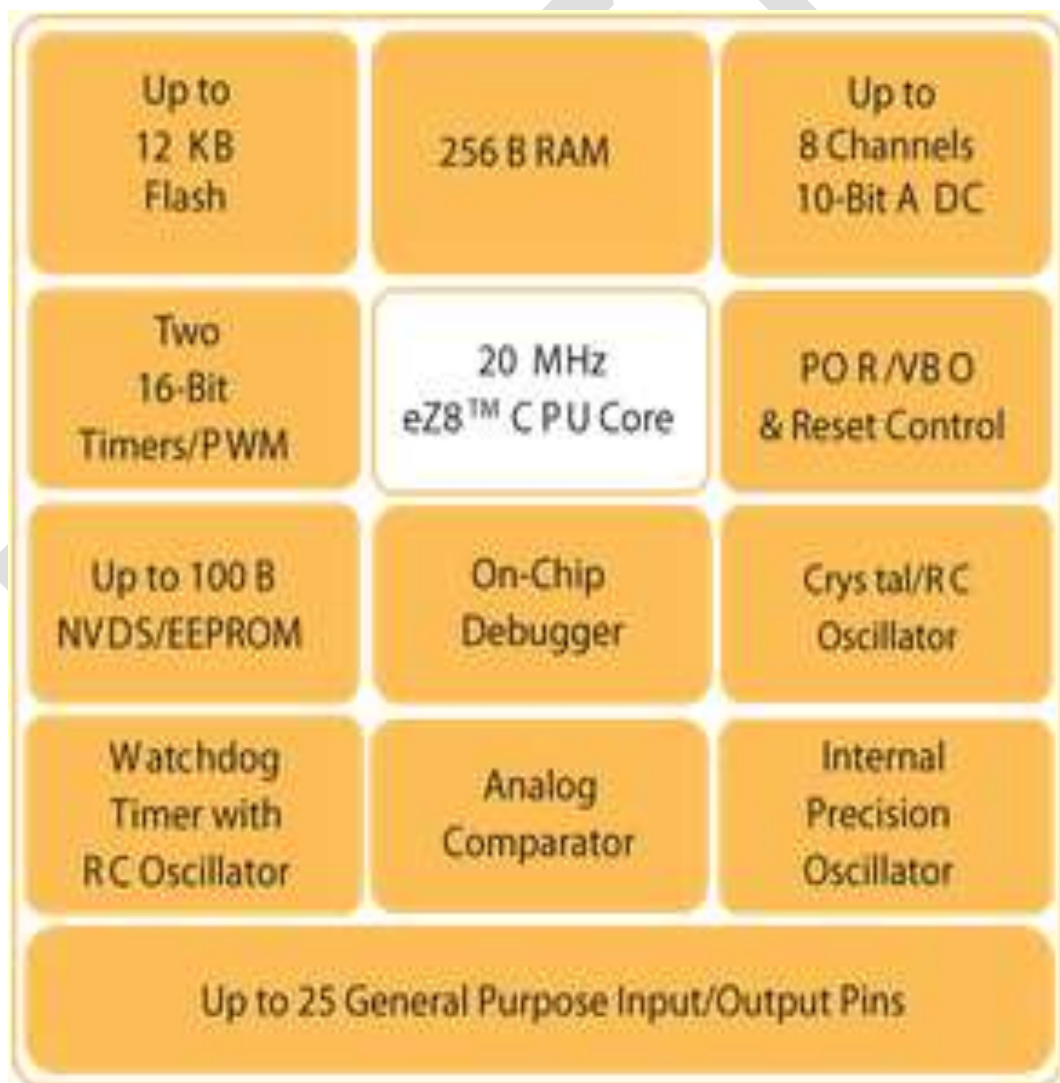
Драйверите са написани съгласно правилата и съветите, описани в част III. Целта е от рано да се научите да оформяте кода си професионално, а не хаотично.

Част VII се състои от практически примери, използващи драйверите описани в част VI, и илюстрира употребата на езика С в програмирането на микроконтролери.

DRAFT

Част I

Описание на микроконтролера Z8F083A



DRAFT

1 Основни характеристики

Z8Encore! F083A серията микроконтролери има следните характеристики:

- 20MHz процесор eZ8;
- До 8kB Флаш памет;
- До 256 байта Регистрова Памет (**RAM-памет**);
- 100 байта енергонезависима памет за данни (**NVDS - Nonvolatile Data Storage**);
- До 23 Входни-Изходни изводи;
- Вътрешен прецизен генератор;
- Два 16-битови многофункционални таймери;
- WDT-таймер със собствен вътрешен RC генератор;
- Еднопроводна двупосочна връзка с дебъгер;
- Бърз 8-канален, 10-битов АЦП;
- Аналогов компаратор;
- До 17 източника на прекъсване;
- Защита от временно понижаване на захранването (**VBO – Voltage Brown-Out**);
- Ресет при включване на захранването (**POR – Power - On Reset**);

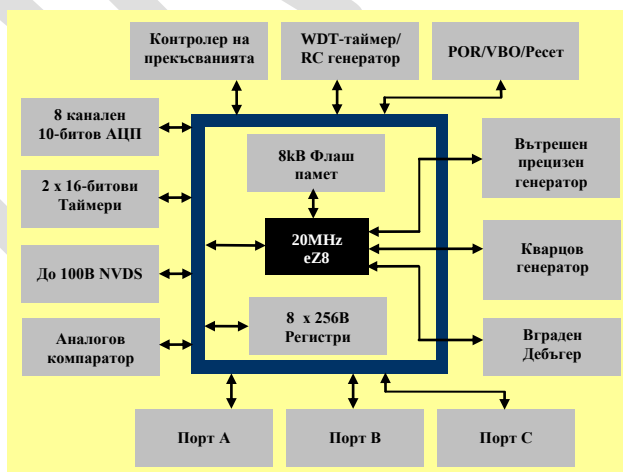
- Работно напрежение $2.6 \div 3.6V$;
- До тринадесет 5V-толерантни изводи;
- 20- и 28-изводни корпуси;
- Работен температурен обхват:
 - Стандартен $0 \div 70\text{ }^{\circ}C$;
 - Разширен $-40 \div 105\text{ }^{\circ}C$.

Табл.1 показва основните характеристики на микроконтролерите от серията Z8Encore! F083A.

Микроконтролер	Флаш (kB)	RAM (B)	NVDS (100B)	ADC	Входни/Изходни изводи
Z8F083A	8	256	Да	Да	17/23
Z8F043A	4	256	Да	Да	17/23

Табл. 1 Микроконтролери от серията Z8 Encore! F083A

Следващата фигура показва блоковата схема на Z8Encore! F083A.



Фиг. 1 Блокова схема на Z8Encore! F083A

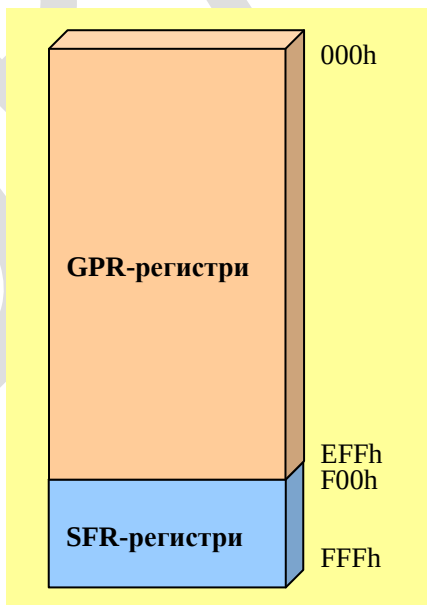
2 Описание на процесора eZ8

2.1 Организация на паметта

Процесорът **eZ8** може да осъществява достъп до три различни типа памет. Всяка памет има свое отделно адресно пространство.

- Регистрова Памет (РП)

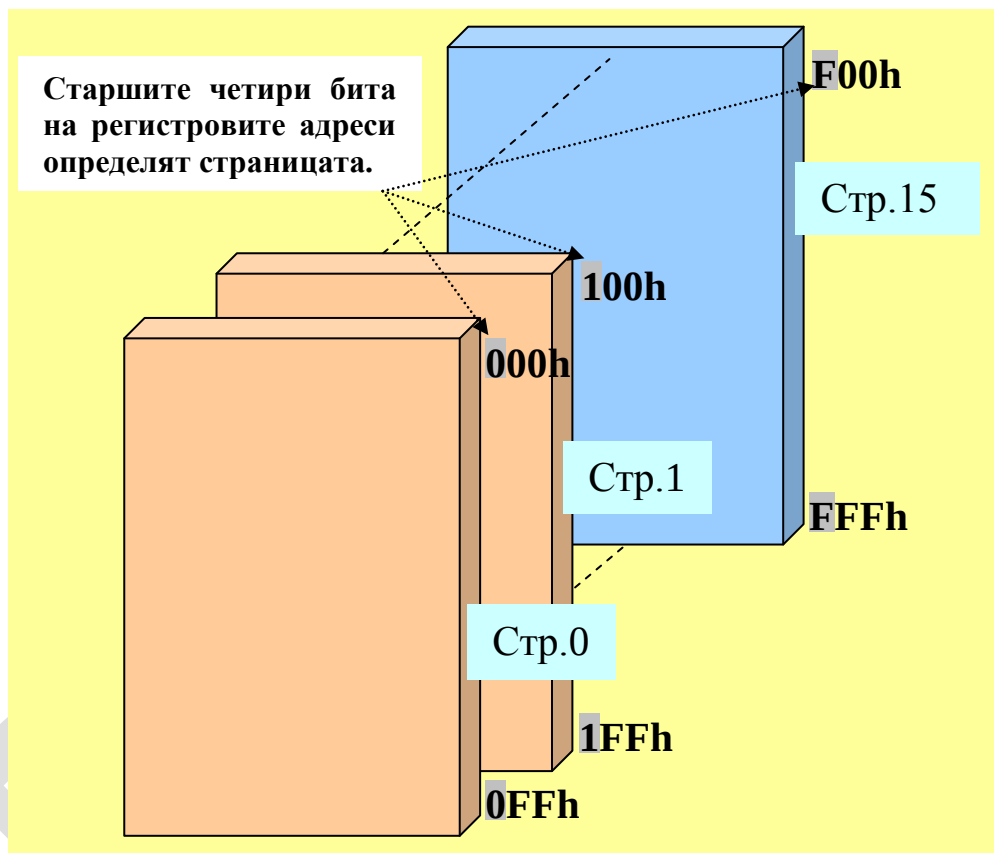
Регистровата Памет се състои от два типа регистри: регистри за обща употреба (**GPR – General Purpose Register**) и управляващи регистри, които се наричат още регистри със специална функция (**SFR – Special Function register**). GPR-регистрите могат да се използват като акумулатори, адресни указатели, индексни регистри, за стек или временно съхранение на данни. SFR-регистрите служат за управление на микроконтролера. Процесорът **eZ8** може да адресира до 4096 байта (4kB) регистри. Фиг.2 показва организацията на РП.



Фиг. 2 Регистрова Памет

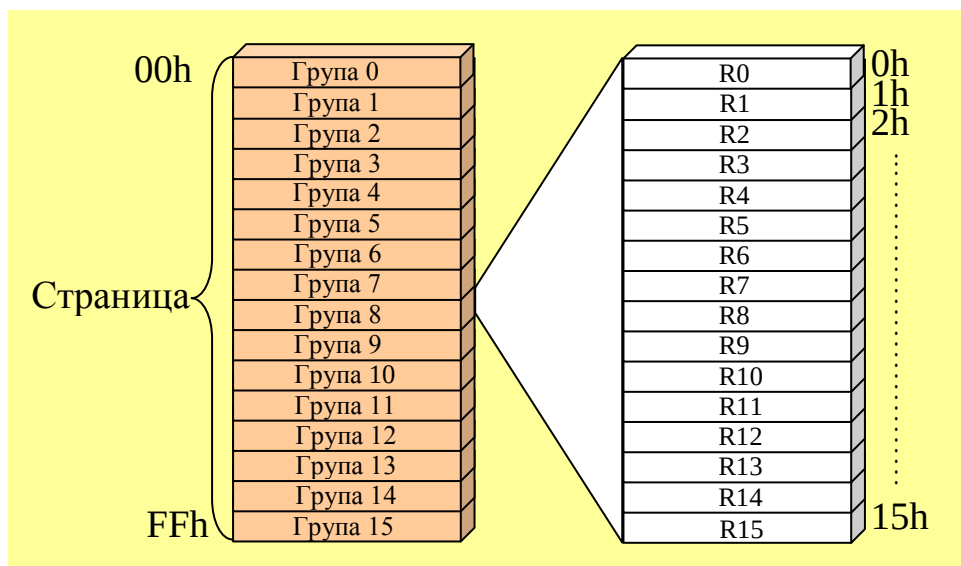
Както се вижда от фигурата, SFR-регистрите заемат последните 256 байта от Регистровата Памет.

Регистровата Памет се разделя логически на 16 страници, всяка състояща се от 256 регистри (Фиг.3).

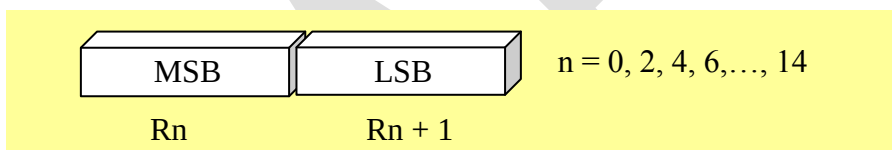


Фиг. 3 Страниране на Регистровата Памет

Всяка страница от своя страна се разделя на 16 групи, всяка състояща се от 16 регистъра, наричани още **работни регистри**. Регистрите във всяка група се обозначават като R0, R1, R2,..., R15 (Фиг.4).



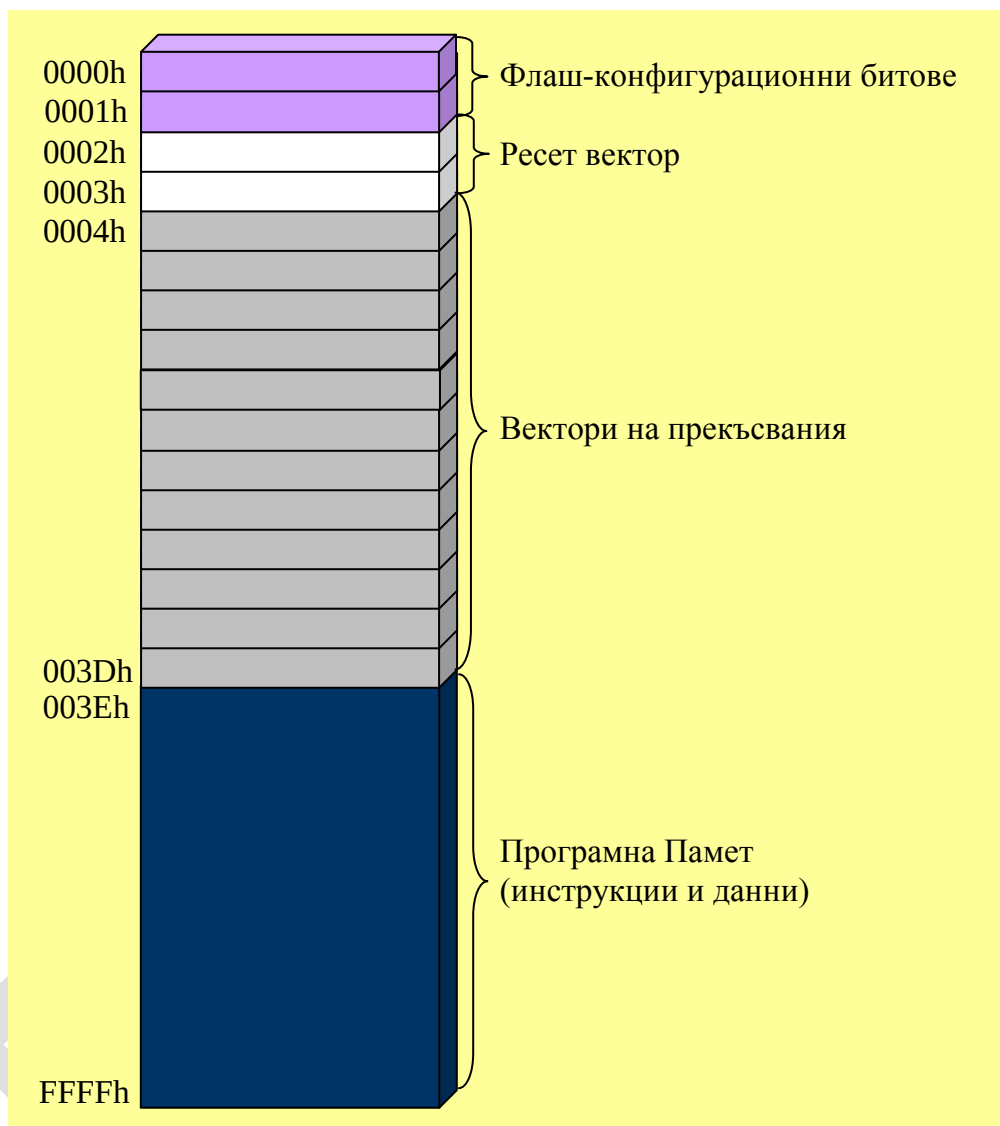
Фиг. 4 Разделяне на страница на 16 групи по 16 регистъра



Фиг. 5 16-битова регистрова дума

■ Програмна Памет (ПП)

Процесорът **eZ8** може да адресира 65 536 байта (64 kB) Програмна Памет. Програмната Памет се използва за съхранение на инструкциите на програмата и за съхранение на данни. Следващата фигура показва организацията на ПП. Първите два байта са заети от флаш-конфигурационни битове (вижте [16 Флаш-конфигурационни битове](#)), след тях е разположена таблицата с векторите на прекъсване, и най-накрая паметта за инструкциите и данни.



Фиг. 6 Програмна Памет

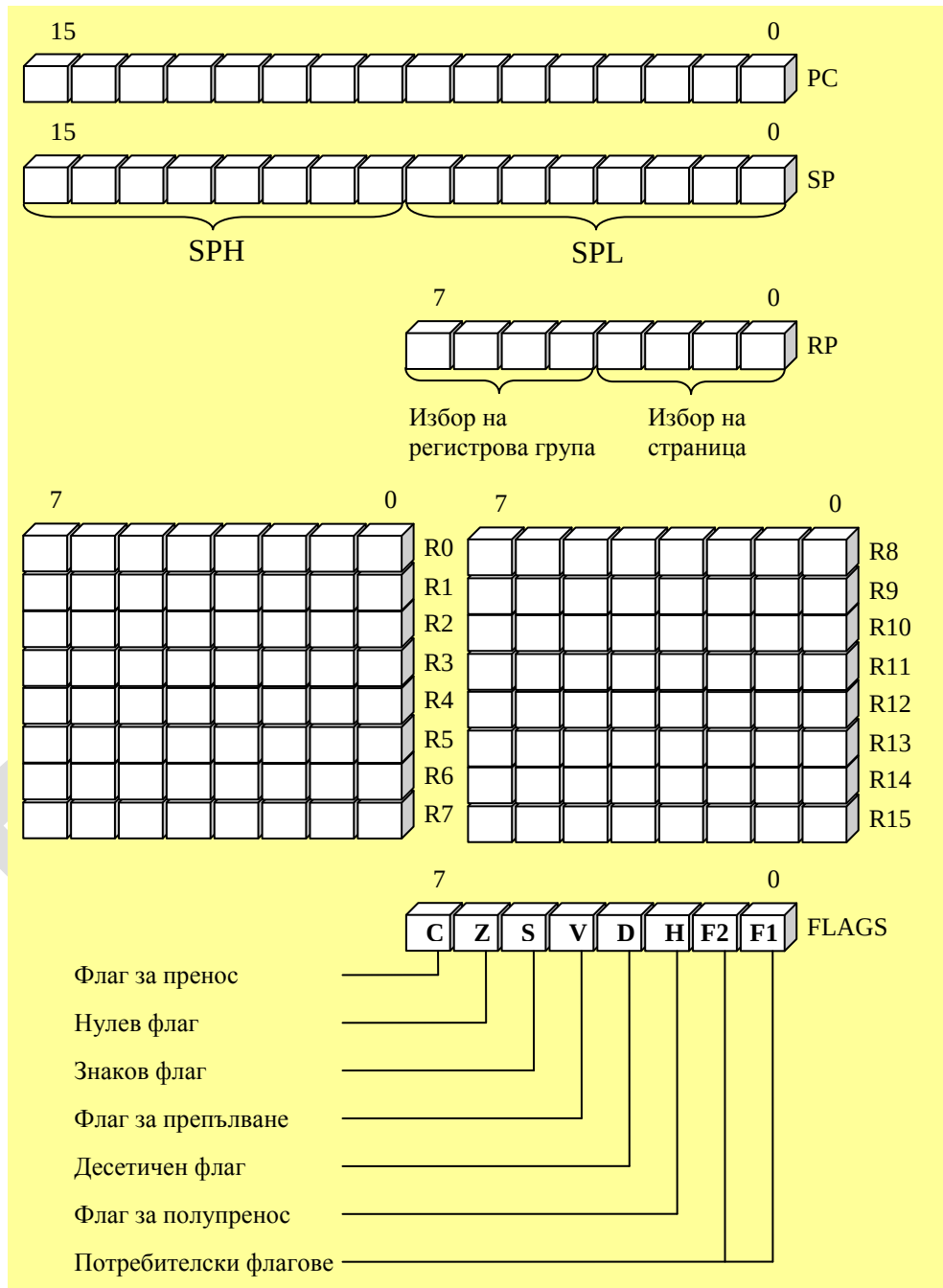
- Памет за Данни (ПД)

Процесорът **eZ8** също така може да адресира и 65 536 байта (64 kB) Памет за Данни¹.

Забележка¹: В микроконтролерите Z8Encore!F083A тази памет не е имплементирана.

2.2 Програмен модел

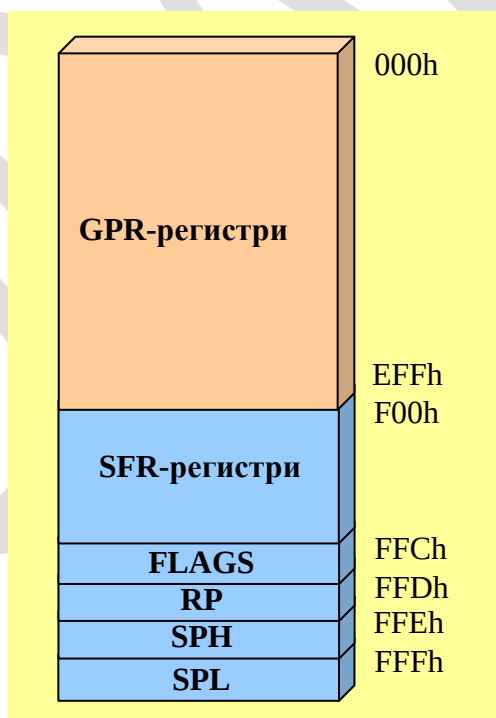
Фиг.7 описва програмния модел на процесора eZ8.



Фиг. 7 Програмен модел на процесора eZ8

Програмният Брояч PC (**Program Counter**) е 16-битов и съдържа адреса в Програмната Памет на поредната инструкция за изпълнение. Указателят на Стекa SP (**Stack Pointer**) също е 16-битов и сочи към последната записана стойност в стека. SP се състои от два 8-битови регистри: SPH и SPL. Указателят към Регистрите RP (**Register Pointer**) е 8-битов и се използва за избор на страница (RP[3:0]) и регистрова група (RP[7:4]) от Регистровата Памет. R0÷R15 е текущата избрана регистрова група. Регистърът на Флаговете FLAGS е 8-битов и съдържа флагове, указващи различни ситуации (като препълване при сумиране, нулев резултат и др.) от последната изпълнена от процесора инструкция.

С изключение на Програмния Брояч всички останали регистри за управление на процесора са част от Регистровата Памет и винаги се разполагат на едни и същи адреси (Фиг.8).



Фиг. 8 Процесорни регистри

Флаг за пренос C

Флагът за пренос C е 1, когато резултатът от аритметична операция генерира пренос от или заем в бит 7 на данните. В противен случай е 0.

Нулев Флаг Z

Ако резултатът от аритметична или логическа операция е нула, този флаг е 1, в противен случай е 0.

Знаков флаг S

Знаковият флаг S съдържа стойността на най-старшия бит на резултата от аритметична или логическа операция. Процесорът eZ8 използва допълнителен код (**2's complement**) за представяне на знакови числа. При този код ако най-старшият бит на числото е 0, то е положително ($S = 0$), а ако е 1, то е отрицателно ($S = 1$).

Флаг за препълване V

Флагът за препълване V е 1, ако резултатът от знакови аритметични, за преместване и ротация операции е по-голям от максималната възможна стойност (> 127) или по-малък от минималната възможна стойност (< -128), която може да се представи от 8-битово число в допълнителен код, т.е. . ако резултатът е извън диапазона $[-128 \div 127]$. В противен случай $V = 0$.

Десетичен флаг D

Флагът D се използва за BCD (**B**inary-**C**oded **D**ecimal) аритметични операции. Тъй като алгоритъмът за коригиране на BCD операции е различен за събиране и изваждане, този флаг специфицира типа на последната изпълнена инструкция, позволявайки последваща операция за

десетична настройка. След изваждане $D = 1$. След сумиране $D = 0$.

Флаг за полупренос H

Флагът за полупренос H е 1, когато операция събиране генерира пренос от бит 3 или операция изваждане генерира заем в бит 4.

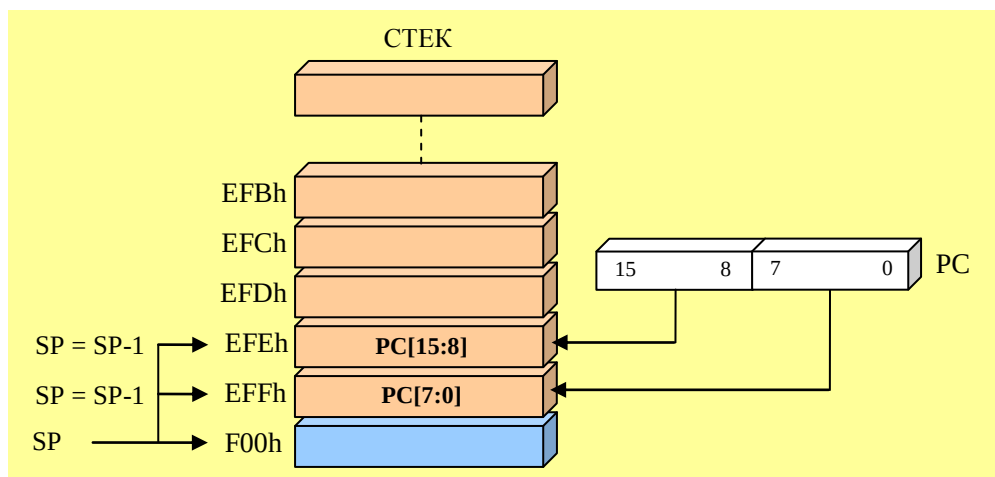
Потребителски флагове F1/F2

Тези флагове не се използват от процесора и могат да се използват от потребителя свободно за собствени цели.

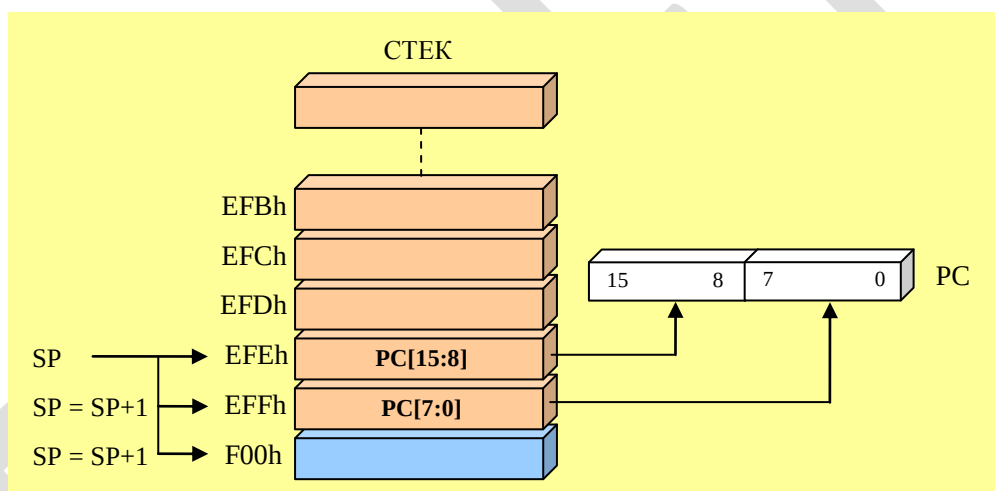
2.3 Стек

Стекът в eZ8 се разполага в Регистровата Памет. Указателят на стека е 16-битов и се състои от регистрите SPH (старши байт на SP) и SPL (младши байт на SP), т.е. $SP = SPH:SPL$. SP сочи винаги към последно записаната стойност в стека и се изменя по механизма предекремент-постинкремент (при запис SP първо се намалява и след това стойността се записва в стека, при четене стека се прочита и след това SP се увеличава).

При извикване на инструкцията CALL (извикване на подпрограма) съдържанието на PC се съхранява в стека автоматично. При изпълнение на инструкцията RET (връщане от подпрограма) съхранената в стека стойност се връща обратно в PC. Следващите фигури илюстрират този процес. Приема се, че стекът е разположен в края Регистровата Памет, заемана от GPR-регистрите, и указателят на стека SP сочи към първия SFR-регистър. Това означава, че първата записана стойност в стека ще се разположи в последния GPR-регистър (адрес EFFh).

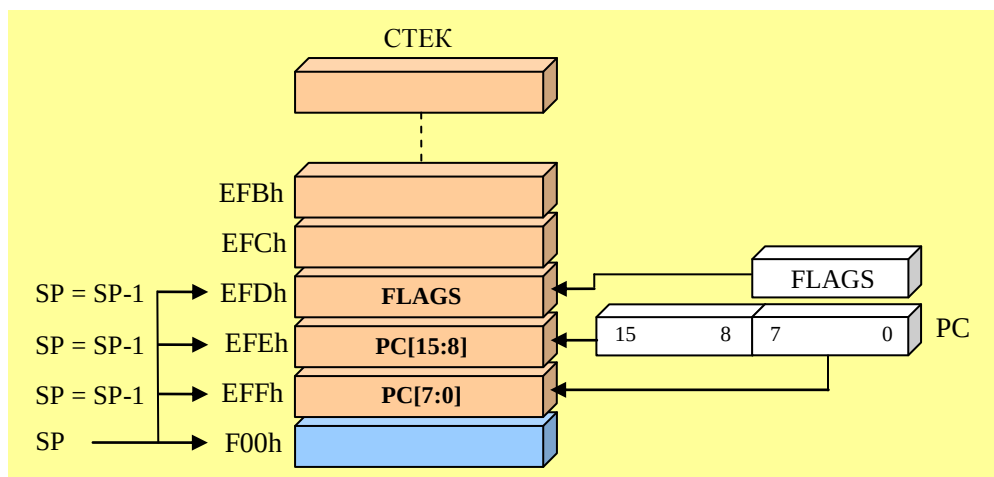


Фиг. 9 Стек и инструкцията CALL

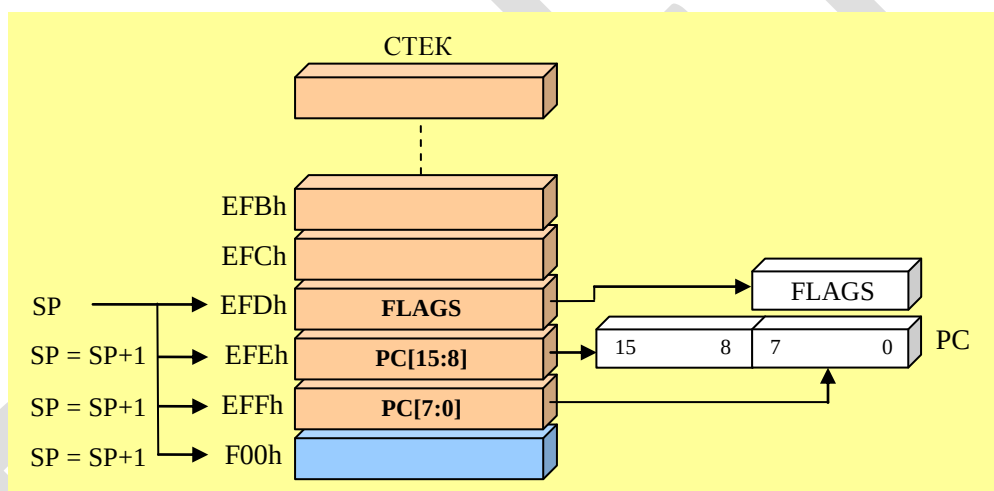


Фиг. 10 Стек и инструкцията RET

При възникване на прекъсване освен PC в стека се съхранява и Регистъра на Флаговете FLAGS. Изпълнението на инструкцията IRET (връщане от прекъсване) възстановява стойностите на тези регистри обратно. Следващите фигури илюстрират този процес. Приема се, че стекаът е разположен в края Регистровата Памет, заемана от GPR-регистрите и указателят на стека SP сочи към първия SFR-регистър. Това означава, че първата записана стойност в стека ще се разположи в последния GPR-регистър (адрес EFFh).



Фиг. 11 Стек и възникване на прекъсване



Фиг. 12 Стек и инструкцията IRET

2.4 Прекъсвания

2.4.1 Общи сведения

Прекъсванията позволяват на периферните модули на микроконтролера да прекъсват текущо изпълняваната от процесора програма и да го превключват към изпълнение на подпрограма, обслужваща това прекъсване (**ISR – Interrupt**

Service Routine). Когато тази подпрограма завърши изпълнението си, процесорът отново се връща към изпълнение на инструкциите на прекъснатата програма. Всички прекъсвания в eZ8 могат да бъдат разрешени с инструкцията EI (установява в 1 глобалния разрешаващ бит IRQE в регистър IRQCTL) и да бъдат забранени с инструкцията DI (нулира глобалния разрешаващ бит IRQE в регистър IRQCTL). Също така източниците на прекъсване имат приоритет, който определя кое прекъсване ще се обработи първо в случай, че възникнат няколко прекъсвания едновременно.

2.4.2 Обработка на разрешено прекъсване

Всеки източник на прекъсване има собствен адрес в Програмната Памет, на който се съхранява **векторът на прекъсването**. Векторът на прекъсването¹ заема два байта и представлява адреса на подпрограмата, обработваща това прекъсване.

Забележка¹: Отговорност на програмиста е да запише адреса (вектора на прекъсване) на дадена ISR-подпрограма на съответния адрес в Програмната Памет.

Когато едно прекъсване възникне, се извършват следните действия:

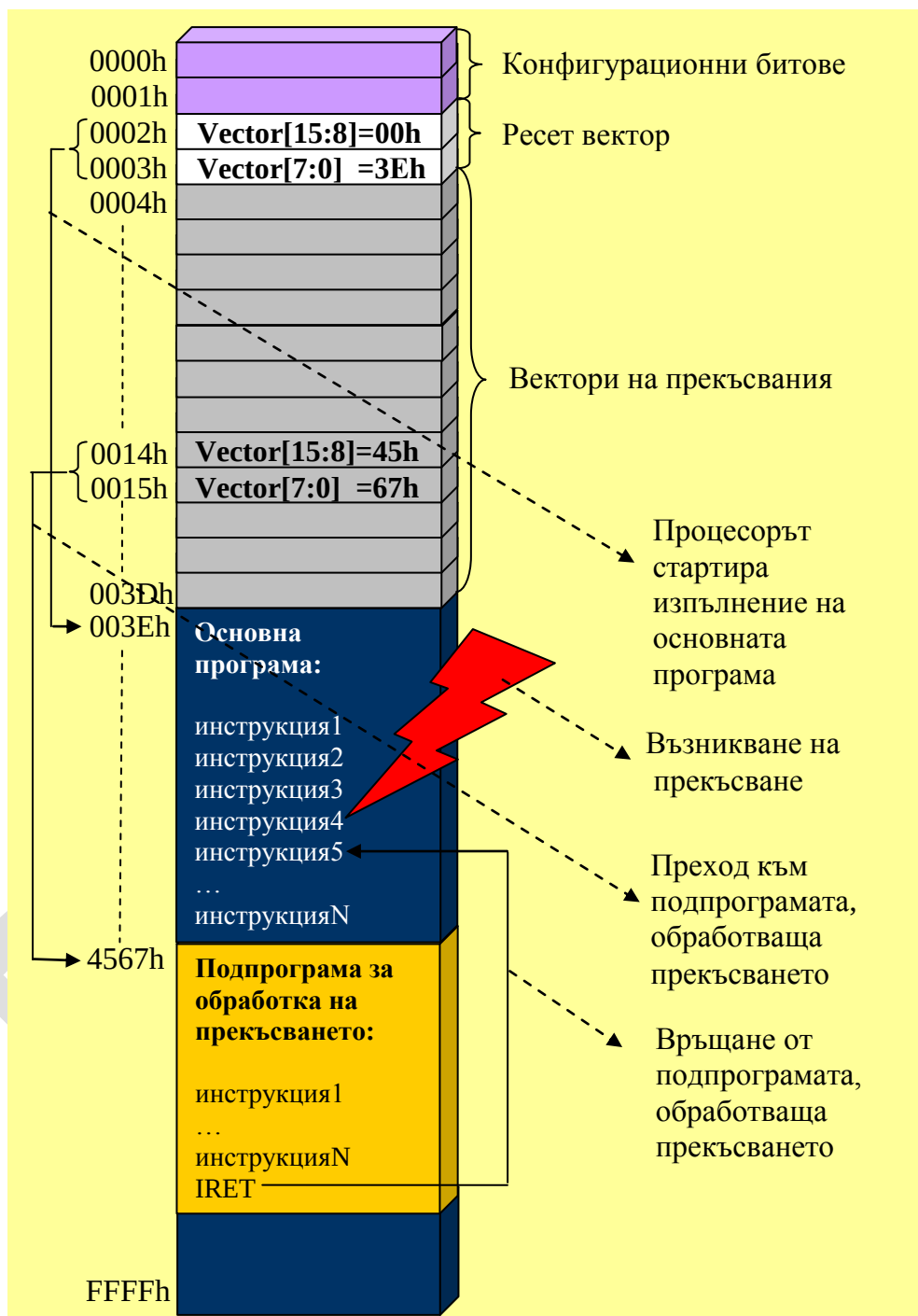
1. Бит IRQE се нулира (всички прекъсвания се забраняват).
2. PC[7:0] се записва в стека;
3. PC[15:8] се записва в стека;

4. FLAGS се записва в стека;
5. Старшият байт на вектора на прекъсване се записва в PC[15:8];
6. Младшият байт на вектора на прекъсване се записва в PC[7:0];

След стъпки 4 и 5 PC сочи към подпрограмата, обработваща прекъсването.

7. Извършва се преход към подпрограмата, обработваща прекъсването.

Следващата фигура илюстрира обработката на прекъсване. Приема се, че векторът на прекъсване се съхранява на адреси 0014h и 0015h в Програмната Памет.



Фиг. 13 Обработка на прекъсване

Фиг.13 описва следната ситуация: ресет векторът съдържа

адреса на основната програма и процесорът изпълнява нейните инструкции. По време на изпълнение на **инструкция4** възниква прекъсване. След като процесорът завърши изпълнението на тази инструкция, съдържанието на PC (който вече сочи към **инструкция5**), се съхранява в стека. Съдържанието на FLAGS също се съхранява в стека и в PC се зарежда адреса на подпрограмата, обработваща това прекъсване. В нашия пример този вектор (4567h) е разположен на адреси 0014h и 0015h. След това процесорът преминава към изпълнение на инструкциите на ISR-подпрограмата. При достигане на инструкцията IRET (връщане от прекъсване), регистрите PC и FLAGS възстановяват стойностите си от стека и процесорът продължава с изпълнението на следващата инструкция от основната програма (**инструкция5**).

2.4.3 Обработка на неразрешено прекъсване

Флаговете на прекъсване се установяват в 1 при всяко генериране на прекъсване, независимо дали прекъсването е разрешено или забранено. Ако едно прекъсване е забранено, Вие може периодично да проверявате съответния флаг на прекъсване дали е установен в 1 и да извършите извикване на подпрограма, която обработва това прекъсване.

```
...
if (0 != (IRQ & 0x10)) // Проверка на флаг 4 в IRQ1
{
    InterruptHandler(); // Извикай подпрограмата, която
                        // обработва това прекъсване
}
...
void InterruptHandler (void)
{
}
}
```

2.4.4 Вложени прекъсвания

Когато процесорът стартира обработка на прекъсване, всички останали прекъсвания се забраняват (бит IRQE се нулира). Това означава, че обработката на високоприоритетно прекъсване може да бъде забавена от ниско приоритетно прекъсване, текущо обработвано от процесора. Този проблем може да се разреши най-лесно ако в подпрограмата, обработваща ниско приоритетното прекъсване, разрешите отново прекъсванията, като установите в 1 бит IRQE.

```
void InterruptHandler (void)
{
    EI(); // Разрешаване на прекъсванията
    ...
    DI(); // Забраняване на прекъсванията

    return; // Връщане от подпрограмата
}
```

Този подход има и един съществен недостатък, а именно, че и по-ниско приоритетни прекъсвания ще могат да прекъсват подпрограмата. Този недостатък може да се избегне, ако изпълните следните стъпки в подпрограмата:

1. Съхранете регистрите IRQCTL (Interrupt Control), IRQnENH (Interrupt Enable High, $n = 0 \div 2$) и IRQnENL (Interrupt Enable Low, $n = 0 \div 2$).
2. Конфигурирайте регистрите IRQnENH и IRQnENL, така че да забраните прекъсванията с по-нисък приоритет.
3. Разрешете прекъсванията.
4. Продължете с обработката на текущото ниско приоритетно прекъсване.

5. Възстановете обратно регистрите IRQCTL, IRQnENH и IRQnENL.
6. Забранете прекъсванията.

```
void InterruptHandler (void)
{
    // Стъпка 1
    uint08 u08IrqCtlTmp  = IRQCTL;
    uint08 u08Irq0EnhTmp = IRQ0ENH;
    uint08 u08Irq1EnhTmp = IRQ1ENH;
    uint08 u08Irq2EnhTmp = IRQ2ENH;
    uint08 u08Irq0EnlTmp = IRQ0ENL;
    uint08 u08Irq1EnlTmp = IRQ1ENL;
    uint08 u08Irq2EnlTmp = IRQ2ENL;

    // Стъпка 2
    IRQ0ENH = НоваСтойност;
    IRQ1ENH = НоваСтойност;
    IRQ2ENH = НоваСтойност;
    IRQ0ENL = НоваСтойност;
    IRQ1ENL = НоваСтойност;
    IRQ2ENL = НоваСтойност;

    // Стъпка 3
    EI(); // Разрешаване на прекъсванията

    // Стъпка 4
    ...
    // Кодът на подпрограмата се разполага тук
    ...

    // Стъпка 5
    IRQ2ENL = u08Irq2EnlTmp;
    IRQ1ENL = u08Irq1EnlTmp;
    IRQ0ENL = u08Irq0EnlTmp;
    IRQ2ENH = u08Irq2EnhTmp;
    IRQ1ENH = u08Irq1EnhTmp;
    IRQ0ENH = u08Irq0EnhTmp;
    IRQCTL  = u08IrqCtlTmp;

    // Стъпка 6
    DI(); // Забраняване на прекъсванията
}
```

```
    return; //Връщане от подпрограмата  
}
```

Ако изпълните тези стъпки, само прекъсвания с по-висок приоритет ще могат да прекъсват обработката на текущото по-ниско приоритетно прекъсване.

Не се безпокойте, ако имената на регистрите не Ви говорят нищо. Когато разгледаме описанието контролерът на прекъсванията на микроконтролера Z8F083A ([7 Контролер на прекъсванията](#)), всичко ще Ви се изясни напълно.

2.4.5 Генериране на софтуерни прекъсвания

Флаговете на прекъсване могат да бъдат установявани в 1 и от софтуера. Контролерът на прекъсванията обработва софтуерните прекъсвания по същия начин както и хардуерните.

Пример:

```
IRQ1 = 0x10; // IRQ1[4] = 1
```

Тази конструкция установява в 1 бит 4 на регистър IRQ1. Ако прекъсванията са разрешени и няма друго по-високо приоритетно прекъсване, процесорът ще стартира процедура по обработка на това прекъсване.

2.4.6 Аварийни прекъсвания

Аварийното прекъсване (**trap**) е вид прекъсване, което се генерира само в определени аварийни ситуации, т.е. . то не трябва да се генерира при нормално изпълнение на

програмата. Например микроконтролерът Z8F083A може да генерира аварийно прекъсване при възникване на една от следните ситуации:

- Изпълнение на грешна инструкция;
- Грешка в основния генератор на тактовия сигнал;
- Грешка в генератора на WDT-таймера.

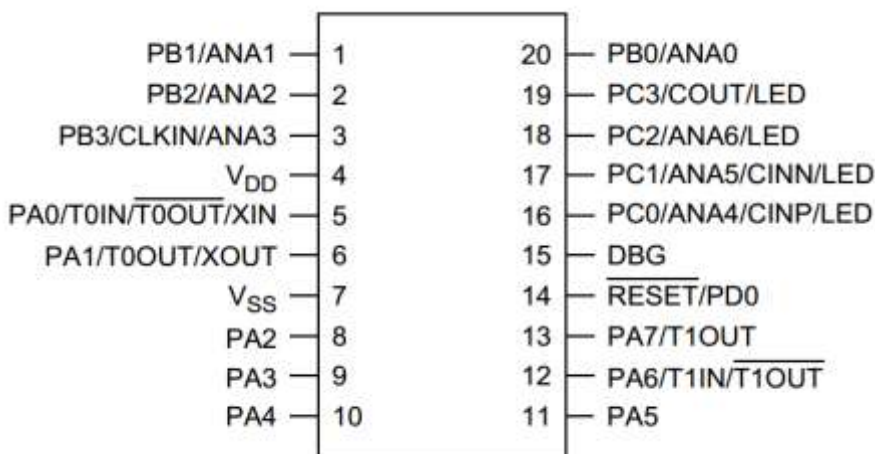
Процесорът обработва аварийните прекъсвания по същия начин както и обичайните прекъсвания. Аварийното прекъсване, генерирано от изпълнение на грешна инструкция, обаче има следните особености: процесорът съхранява регистър FLAGS и PC в стека както и при обичайните прекъсванията, с тази разлика, че PC не се увеличава, а остава да сочи към невалидната инструкция. След това векторът на аварийното прекъсване от грешна инструкция се зарежда в PC.



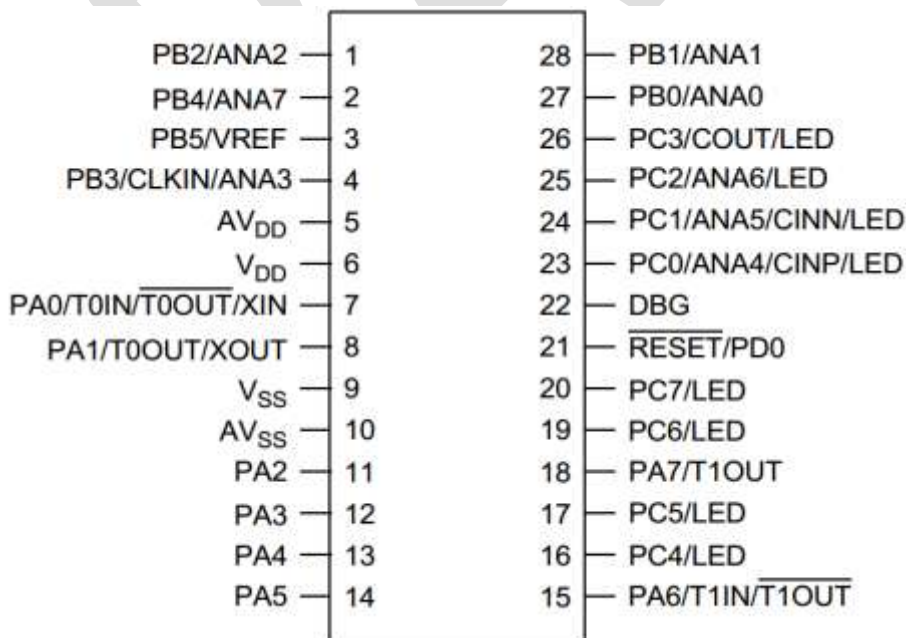
Подпрограмата, обработваща аварийното прекъсване от изпълнение на грешна инструкция, не трябва да завършва с инструкцията IRET. Причината е, че PC ще се зареди отново с адреса на невалидната инструкция, което на свой ред ще предизвика повторно генериране на същото аварийно прекъсване.

3 Описание на изводите

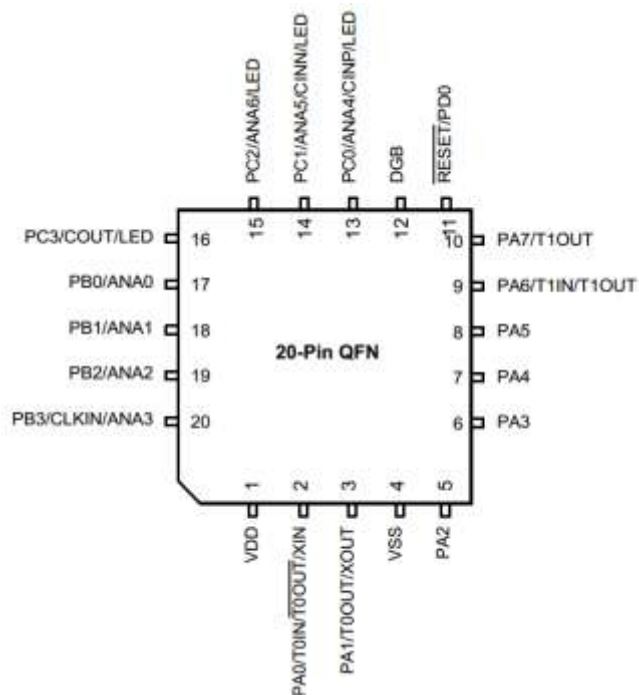
Следващите няколко фигури показва наличните корпуси, а Табл.2 описва функциите на всички изводи. Обърнете внимание, че някои изводи имат няколко различни функции.



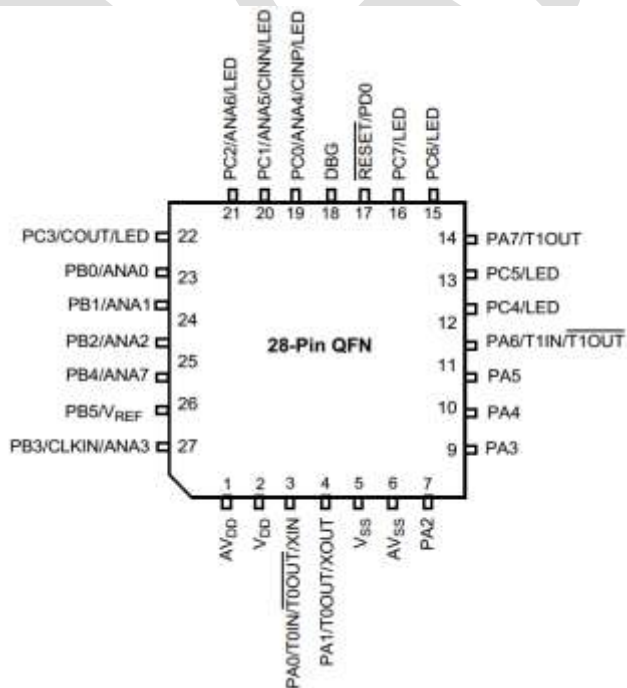
Фиг. 14 20-изводен SOIC, SSOP, PDIP корпус



Фиг. 15 28-изводен SOIC и SSOP корпус



Фиг. 16 20-изводен QFN корпус



Фиг. 17 28-изводен QFN корпус

Функция	В/И	Описание
Портове		
PA[7:0]	В/И	Порт А. Тези изводи се използват като GPIO ¹
PB[5:0]	В/И	Порт В. Тези изводи се използват като GPIO ¹
PC[7:0]	В/И	Порт С. Тези изводи се използват като GPIO ¹
PD[0]	И	Порт D. Този извод се използват само като изход
Таймери		
T0OUT/ T1OUT	И	Таймерен изход 0/1
$\overline{T0OUT}$ / $\overline{T1OUT}$	И	Инвертиран таймерен изход 0/1
T0IN/T1IN	В	Таймерен вход 0/1
Компаратор		
CINP/CINN	В	Компараторни входове (положителен и отрицателен)
COUT	И	Компараторен изход
АЦП		
ANA[7:0]	В	Аналогови входове за АЦП
VREF	В	Вход за опорно напрежение за АЦП
Генератори		
XIN	В	Вход за външен кварцов кристал
XOUT	И	Иход за външен кварцов кристал
CLKIN	В	Вход за външен TTL системен тактов сигнал
LED драйвери		
LED	И	LED изход. Всички изводи на порт С имат способността да управляват LED директно без допълнителни компоненти.
Дебъгер		
DBG	В/И	Вход/Изход за дебъгер. Този извод е с отворен дрейн и изисква външен изтеглящ (pull up) резистор.
Ресет		
$\overline{RESET}$	В/И	Вход за ресет. Също служи и за ресет индикатор.
Захранване		

V_{DD}	B	Захранване на цифровата част на устройството
AV_{DD}	B	Захранване на аналоговата част на устройството
V_{SS}	B	Маса на цифровата част на устройството
AV_{SS}	B	Маса на аналоговата част на устройството

Табл. 2 Описание на функциите на изводите

Забележка¹: **GPIO (General Purpose Input/Output)** – Цифров Вход/Изход за Обща Употреба.

4 Адресно пространство

Регистрова памет

Z8Encore!F083A серията устройства имат до 256 байта RAM. Четене от регистър извън наличната RAM памет връща недефинирана стойност, а записът няма никакъв ефект.

Програмна памет

Z8Encore!F083A серията устройства имат до 8kB флаш памет. Четене от адрес извън наличната флаш памет връща FFh, а записът няма никакъв ефект. Табл.3 описва картата на паметта на наличните устройства.

Устройство	Адрес	Функция
Z8F083A	0000h-0001h	Флаш-конфигурационни битове
	0002h-0003h	Ресет вектор
	0004h-003Dh	Таблица с вектори на прекъсванията
	003Eh-1FFFh	Програмна памет
Z8F043A	0000h-0001h	Флаш-конфигурационни битове
	0002h-0003h	Ресет вектор
	0004h-003Dh	Таблица с вектори на прекъсванията
	003Eh-0FFFh	Програмна памет

Табл. 3 Карта на програмната памет

Памет за данни

Z8Encore!F083A серията устройства не използва 64kB адресно пространство за данни на процесора eZ8.

5 Карта на регистровата памет

Табл.4 описва всички регистри на Регистровата Памет, адресите, които заемат и стойностите им след ресет.

xx – недефинирана стойност

Адрес	Обозначение	Описание	Ресет
GPR-регистри			
000h-0FFh	-	RAM	xx
100h-1FFh	-	Резервиран	xx
SFR-регистри			
Таймер0			
F00h	T0H	Timer 0 High Byte	00
F01h	T0L	Timer 0 Low Byte	01
F02h	T0RH	Timer 0 Reload High Byte	FF
F03h	T0RL	Timer 0 Reload Low Byte	FF
F04h	T0PWMH	Timer 0 PWM High Byte	00
F05h	T0PWML	Timer 0 PWM Low Byte	00
F06h	T0CTL0	Timer 0 Control 0	00
F07h	T0CTL1	Timer 0 Control 1	00
Таймер1			
F08h	T1H	Timer 1 High Byte	00
F09h	T1L	Timer 1 Low Byte	01
F0Ah	T1RH	Timer 1 Reload High Byte	FF
F0Bh	T1RL	Timer 1 Reload Low Byte	FF
F0Ch	T1PWMH	Timer 1 PWM High Byte	00
F0Dh	T1PWML	Timer 1 PWM Low Byte	00
F0Eh	T1CTL0	Timer 1 Control 0	00
F0Fh	T1CTL1	Timer 1 Control 1	00
F10h-F6Fh	-	Резервиран	xx
АЦП			
F70h	ADCCTL0	ADC Control 0	00
F71h	-	Резервиран	xx
F72h	ADCD_H	ADC Data High Byte	xx

F73h	ADCDL	ADC Data Low Bits	xx
F74h	ADCSST	ADC Sample Settling Time	0F
F75h	ADCST	ADC Sample Time	3F
F76h	ADCCP	ADC Clock Prescale	00
F77h-F7Fh	-	Резервиран	xx
LPC (Low Power Control)			
F80h	PWRCTL0	Power Control 0	88
F81h	Резервиран	-	xx
LED контролер			
F82h	LEDEN	LED Drive Enable	00
F83h	LEDLVLH	LED Drive Level High	00
F84h	LEDLVLL	LED Drive Level Low	00
F85h	-	Резервиран	xx
Генератор			
F86h	OSCCTL	Oscillator Control	A0
F87h-F8Fh	-	Резервиран	xx
Компаратор0			
F90h	CMP0	Comparator 0 Control	14
F91h-FBFh	-	Резервиран	xx
Контролер на прекъсванията			
FC0h	IRQ0	Interrupt Request 0	00
FC1h	IRQ0ENH	IRQ0 Enable High Bit	00
FC2h	IRQ0ENL	IRQ0 Enable Low Bit	00
FC3h	IRQ1	Interrupt Request 1	00
FC4h	IRQ1ENH	IRQ1 Enable High Bit	00
FC5h	IRQ1ENL	IRQ1 Enable Low Bit	00
FC6h	IRQ2	Interrupt Request 2	00
FC7h	IRQ2ENH	IRQ2 Enable High Bit	00
FC8h	IRQ2ENL	IRQ2 Enable Low Bit	00
FC9h-FCCh	-	Резервиран	xx
FCDh	IRQES	Interrupt Edge Select	00
FCEh	IRQSS	Shared Interrupt Select	00
FCFh	IRQCTL	Interrupt Control	00
Порт А			

FD0h	PAADDR	Port A Address	00
FD1h	PACTL	Port A Control	00
FD2h	PAIN	Port A Input Data	xx
FD3h	PAOUT	Port A Output Data	00
Порт В			
FD4h	PBADDR	Port B Address	00
FD5h	PBCTL	Port B Control	00
FD6h	PBIN	Port B Input Data	xx
FD7h	PBOUT	Port B Output Data	00
Порт С			
FD8h	PCADDR	Port C Address	00
FD9h	PCCTL	Port C Control	00
FDAh	PCIN	Port C Input Data	xx
FDBh	PCOUT	Port C Output Data	00
Порт D			
FDCh	PDADDR	Port C Address	00
FDDh	PDCTL	Port C Control	00
FDEh	-	Резервиран	xx
FDFh	PDOUT	Port D Output Data	00
FE0h-FEFh	-	Резервиран	xx
WDT таймер			
FF0h	RSTSTAT	Reset Status	xx
	WDTCTL	WDT Control	xx
FF1h	WDTU	WDT Reload upper byte	FF
FF2h	WDTH	WDT Reload High Byte	FF
FF3h	WDTL	WDT Reload Low Byte	FF
FF4h-FF5h	-	Резервиран	xx
Флаш контролер			
FF6h	TRMADR	Trim Bit Address	00
FF7h	TRMDR	Trim Data	xx
FF8h	FCTL	Flash Control	00
	FSTAT	Flash Status	00
FF9h	FPS	Flash Page Select	00
	FPROT	Flash Sector Protect	00

FFAh	FFREQH	Flash Programming Frequency High Byte	00
FFBh	FFREQL	Flash Programming Frequency Low Byte	00
Процесорни регистри			
FFCh	FLAGS	Flags	xx
FFDh	RP	Register Pointer	xx
FFEh	SPH	Stack Pointer High Byte	xx
FFFh	SPL	Stack Pointer Low Byte	xx

Табл. 4 Карта на регистровата памет

6 Ресет и излизане от STOP-режим

6.1 Източници и типове ресет

Серията микроконтролери Z8Encore!F083A има следните източници на начално установяване (системен ресет) на микроконтролера:

- Включване на захранването (**POR – Power-On Reset**);
- Понижаване на захранването (**VBO – Voltage Brown-Out Reset**);
- Препълване на WDT-таймера¹;
- Ресет сигнал от външния извод $\overline{\text{RESET}}$.

Забележка¹: WDT-таймерът ще предизвика ресет само ако тази функция е разрешена от флаш-конфигурационния бит **WDT_RES** (вижте [16 Флаш-конфигурационни битове](#)).

Излизане от STOP-режим¹ се разглежда като форма на ресет на микроконтролера. Следващата таблица изброява типовете ресет и тяхната продължителност.

тип ресет	продължителност
системен ресет	66 такта на вътрешния прецизен генератор
системен ресет с разрешен кварцов генератор	5000 такта на вътрешния прецизен генератор
излизане от STOP-режим	66 такта на вътрешния прецизен генератор
излизане от STOP-режим с	5000 такта на вътрешния прецизен генератор

разрешен кварцов
генератор

Табл. 5 Типове ресет

Забележка¹: Процесорът eZE8 има два режима на ниска консумация: STOP-режим и HALT-режим. Те са разгледани подробно в [9 Режими с ниска консумация](#)

При ресет процесорът eZ8 и всички периферни модули са спрени, кварцовият генератор и вътрешният генератор на WDT-таймера продължават да работят, SFR-регистрите се зареждат с началните си стойности (вижте Табл.4), всички GPIO изводи се конфигурират като входове със забранени изтеглящи резистори. Извод $\overline{\text{PD0}}$ е изключение и се конфигурира като двупосочен $\overline{\text{RESET}}$ извод с отворен дрейн. След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес. След ресет системният тактов сигнал винаги идва от вътрешния прецизен генератор. Табл.6 описва източниците на ресет на микроконтролера в различни работни режими на процесора.

Работен режим	Източник на ресет	Специални условия
Нормален или HALT-режим	POR/VBO	-
	Препълване на WDT-таймера	-
	$\overline{\text{RESET}}$ извод	Продължителността на импулса трябва да е минимум 4 системни такта.
	Дебъгер	-
STOP-режим	POR/BOR	-
	$\overline{\text{RESET}}$ извод	Продължителността на импулса трябва да е минимум 12ns.

Табл. 6 Ресет на микроконтролера в различни работни режими

6.2 Описание на източниците на ресет

Включване на захранването (POR)

Всяко устройство от серията Z8Encore!F083A има вътрешна POR схема, която наблюдава захранването на цифровата част и държи микроконтролера в ресет състояние, докато захранването превиши определено прагово напрежение V_{POR} , при което се стартира брояч. Микроконтролерът се държи в ресет състояние до препълване на брояча. Времето за препълване на брояча е по-дълго, ако кварцовият генератор е разрешен (вижте Табл.5). След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес, а бит POR в регистър RSTSTAT се установява в 1.

Понижаване на захранването (VBO)

Всяко устройство в серията Z8Encore!F083A има вътрешна VBO схема, която наблюдава захранването на цифровата част. Когато захранването падне под определено прагово напрежение V_{VBO} ($V_{VBO} < V_{POR}$), микроконтролерът влиза в ресет състояние и остава така, докато напрежението е по-малко от V_{POR} . След като напрежението превиши V_{POR} , се стартира брояч. Микроконтролерът се държи в ресет състояние до препълване на брояча. Времето за препълване на брояча е по-дълго, ако кварцовият генератор е разрешен (вижте Табл.5). След излизане от ресет стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес, а бит POR в регистър

RSTSTAT се установява в 1.

VBO схемата може да бъде изключена в STOP-режим с помощта на флаш-конфигурационния бит VBO_AO (вижте [16 Флаш-конфигурационни битове](#)).

Препълване на WDT-таймера

Препълването на WDT-таймера инициира системен ресет, ако флаш-конфигурационният бит WDT_RES е 1. Ако флаш-конфигурационният бит WDT_RES е 0, препълването на WDT-таймера генерира прекъсване, но не и системен ресет. При ресет от WDT-таймера бит WDT в регистър RSTSTAT се установява в 1.

Извод RESET

Този извод има входен тригер на Шмит и вътрешен изтеглящ резистор. Преход от високо в ниско ниво на този извод, за минимум четири такта на системния генератор, инициира системен ресет. Ако извод RESET е в ниско ниво и след изтичане на периода на ресета, микроконтролерът остава в ресет, докато нивото на извода стане високо. След излизане от ресет бит EXT в регистър RSTSTAT се установява в 1.

Освен като вход за външен ресет на микроконтролера, този извод служи и за индикатор, че процесорът е в ресет състояние. След като настъпи ресет от вътрешен източник (POR, BOR, WDT-таймер) този извод се държи в ниско ниво от вътрешна схема до изтичане на ресет периода както е описано в Табл.5. Тази особеност позволява на микроконтролерите да установяват в начално състояние външни елементи, свързани към него, и по този начин да се синхронизират.

Дебъгер

Вградения в микроконтролера дебъгер също може да инициира системен POR ресет чрез установяване в 1 на бит RST в регистър OCDCTL. При излизане от ресет този бит автоматично се нулира и бит POR в регистър RSTSTAT се установява в 1.

6.3 Излизане от STOP-режим

Когато процесорът eZ8 изпълни инструкцията STOP, микроконтролерът влиза в режим на ниска консумация^{1,2}, наречен STOP-режим.

Забележка¹: Често на жаргон вкарването на микроконтролера в режим с ниска консумация се нарича „заспиване” на микроконтролера.

Забележка²: Често на жаргон изкарването на микроконтролера от режим с ниска консумация се нарича „събуждане” на микроконтролера.

Излизането на микроконтролера от STOP-режим не влияе на регистрите с изключение на регистрите RSTSTAT и OSCCTL. При всяко излизане от STOP-режим системният тактов сигнал винаги идва от вътрешния прецизен генератор. Стойността на ресет вектора се зарежда в програмния брояч PC и програмата започва изпълнението си от този адрес, а бит STOP в регистър RSTSTAT се установява в 1.

Табл.7 описва източниците, които водят до излизане от STOP режим и действията, които съпровождат този процес.

източник	действие
----------	----------

Препълване от WDT-таймера, конфигуриран за ресет	Излизане от STOP-режим
Препълване от WDT-таймера, конфигуриран за прекъсване	Излизане от STOP-режим, последвано от прекъсване (ако прекъсването е разрешено)
Промяна на нивото на извод	Излизане от STOP-режим
$\overline{\text{RESET}}$ извод	Системен ресет
Ниско ниво на извод DBG	Системен ресет

Табл. 7 Източници на излизане от STOP-режим

Излизане от STOP-режим при препълване на WDT-таймера

Ако WDT-таймерът се препълни по време на STOP-режим, микроконтролерът преминава в нормален режим на работа. Битове WDT и STOP в регистър RSTSTAT се установяват в 1. Ако WDT-таймерът е конфигуриран за прекъсване, и микроконтролерът е конфигуриран да го обработи, то след излизане от STOP-режим процесорът обработва това прекъсване.

Излизане от STOP-режим при промяна на нивото на извод

Всеки от GPIO изводите може да се конфигурира да буди процесора при промяна на нивото на извода ($\uparrow$ или $\downarrow$). Бит STOP в регистър RSTSTAT се установяват в 1.



В STOP-режим входните регистри за данни PxIN са забранени. Стойността на входен извод се записва в съответния бит за данни само ако сигналът остане на извода до края на процедурата по събуждане на процесора. Ето защо кратки импулси на такъв извод инициират събуждане на процесора, но не се записват в регистъра за данни, нито се стартира прекъсване (ако е разрешено за този извод).

Излизане от STOP-режим от извод RESET

Ниско ниво на този извод по време на STOP-режим инициира системен ресет. Минималната широчина на импулса трябва да бъде 12ns. Бит EXT в регистър RSTSTAT се установява в 1.

Излизане от STOP-режим по ниско ниво на извод DBG

По време на STOP-режим дебъгерът инициира системен ресет при детектиране на определени грешни сигнали на извод DBG (ниско ниво с определена продължителност в зависимост от ситуацията). Дебъгерната част не се инициализира, докато останалата част на микроконтролера преминава през нормална процедура на системен ресет. Бит POR в регистър RSTSTAT се установява в 1.

6.4 Описание на управляващите регистри

Регистър RSTSTAT (Reset Status)

Табл.8 описва регистър RSTSTAT. Този регистър е само за четене. Четене на RSTSTAT нулира старшите четири бита. Този регистър използва същия адрес като регистър WDTCTL (Watchdog Timer Control Register), който е само за запис.

Ч/З – Четене/Запис

Бит	7	6	5	4	3	2	1	0
Име	POR	STOP	WDT	EXT				
Ресет	вижте Табл.47				0	0	0	0
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FF0h							

Бит	Описание
7	Индикатор на POR ресет Установява се в 1 ако настъпи POR ресет и се нулира, ако настъпи препълване от WDT-таймера или излизане от STOP-режим. Четене на регистъра също нулира този бит.
6	Индикатор на излизане от STOP-режим Установява се в 1 ако настъпи излизане от STOP-режим и се нулира при POR ресет или ресет от препълване на WDT-таймера, докато устройството не е в STOP-режим. Четене на регистъра също нулира този бит.
5	Индикатор на препълване на WDT-таймера Установява се в 1 ако настъпи препълване от WDT-таймера. POR ресет нулира този бит. Излизане от STOP-режим при промяна на нивото на извод също нулира този бит. Четене на регистъра нулира този бит.
4	Индикатор на ресет от извод $\overline{\text{RESET}}$ Установява се в 1 при ресет от извод $\overline{\text{RESET}}$. POR ресет или излизане от STOP-режим при промяна на нивото на извод нулира този бит. Четене на регистъра също нулира този бит.
3 - 0	Тези битове са резервирани и трябва да бъдат 0.

Табл. 8 RSTSTAT

Събитие	POR	STOP	WDT	EXT
Ресет от POR	1	0	0	0
Ресет от извод $\overline{\text{RESET}}$	0	0	0	1
Ресет от препълване на WDT-таймера	0	0	1	0
Ресет от дебъгера (OCTCTL[1] = 1)	1	0	0	0
Ресет от STOP-режим при ниско ниво на извод DBG	1	0	0	0
Излизане от STOP-режим при промяна на нивото на извод	0	1	0	0

Излизане от STOP-режим при препълване на WDT- таймера	0	1	1	0
---	---	---	---	---

Табл. 9 Индикатори на източника на ресет

7 Контролер на прекъсванията

7.1 Общи сведения

Z8Encore!F083A серията микроконтролери има 17 източници на прекъсване, използващи 16 вектора на прекъсване, и 3 източника на аварийни прекъсвания.

- 12 прекъсвания от GPIO-изводи;
- 5 прекъсвания от периферни модули;
- Аварийно прекъсване от изпълнение на грешна инструкция;
- Аварийно прекъсване от грешка в основния генератор;
- Аварийно прекъсване от грешка в генератора на WDT-таймера.

Прекъсванията от извод РА6 и Компаратора използват общ адрес, на който се разполага вектора на прекъсване.

Табл.10 описва всички източници на прекъсване, адресите в програмната памет, където се разполагат векторите на прекъсване и хардуерния приоритет на прекъсване. Един вектор на прекъсване се разполага с най-старшия байт на четен адрес и най-младшия байт на нечетен (big-endian).

Приоритет	Адрес	Източник на прекъсване
най-висок	0002h	Ресет (не е прекъсване)
	0004h	WDT-таймер
	0006h	Грешна инструкция (аварийно прекъсване)
	0008h	-
	000Ah	Таймер1
	000Ch	Таймер0

	000Eh	-
	0010h	-
	0012h	-
	0014h	-
	0016h	АЦП
	0018h	РА7 (избираем нарастващ или падащ фронт)
	001Ah	РА6 (избираем нарастващ или падащ фронт, или компараторен изход)
	001Ch	РА5 (избираем нарастващ или падащ фронт)
	001Eh	РА4 (избираем нарастващ или падащ фронт)
	0020h	РА3 (избираем нарастващ или падащ фронт)
	0022h	РА2 (избираем нарастващ или падащ фронт)
	0024h	РА1 (избираем нарастващ или падащ фронт)
	0026h	РА0 (избираем нарастващ или падащ фронт)
	0028h	-
	002Ah	-
	002Ch	-
	002Eh	-
	0030h	РС3 (по двата входни фронта)
	0032h	РС2 (по двата входни фронта)
	0034h	РС1 (по двата входни фронта)
	0036h	РС0 (по двата входни фронта)
	0038h	-
	003Ah	Основен генератор (аварийно прекъсване)
най-нисък	003Ch	РС генератор на WDT-таймера (аварийно прекъсване)

Табл. 10 Източници на прекъсване в Z8F083A

Прекъсванията се разрешават с глобалния бит IRQE в регистър IRQCTL.

Бит IRQE се установява в 1 (прекъсванията са разрешени) при едно от следните действия:

- Изпълняване на инструкцията EI;

- Изпълняване на инструкцията IRET;
- Запис на 1 в бит IRQE.

Бит IRQE се установява в 0 (прекъсванията са забранени) от следните действия:

- Изпълняване на инструкцията DI;
- Приемане на прекъсване за обработка от eZ8 процесора;
- Запис на 0 в бит IRQE;
- Ресет;
- Изпълнение на инструкцията TRAP;
- Генериране на аварийно прекъсване от изпълнение на грешна инструкция;
- Генериране на аварийно прекъсване от грешка в главния генератор;
- Генериране на аварийно прекъсване от грешка в генератора на WDT-таймера.

Всеки източник на прекъсване има локален механизъм за разрешаване. Този механизъм също се използва за задаване и на софтуерен приоритет на прекъсванията. Процесорът eZ8 има 3 нива на софтуерен приоритет: ниво 3 (най-високо), ниво 1 (най-ниско). Ако възникнат няколко прекъсвания с еднакъв софтуерен приоритет, процесорът взема под внимание хардуерния приоритет. Прекъсването с най-висок хардуерен приоритет се обработва първо.

Ресет контролерът, прекъсването от WDT-таймера и трите

аварийни прекъсвания винаги имат най-високо софтуерно ниво на прекъсване. Всички останали източници на прекъсване имат по два бита, чрез които се извършва локално разрешаване и се задава софтуерно ниво на прекъсване. Табл.11 описва възможните комбинации на тези битове.

IRQnENH[x]	IRQnENL[x]	Софтуерен приоритет	Описание
0	0	-	Забранено
0	1	Ниво 1	Ниско
1	0	Ниво 2	Средно
1	1	Ниво 3	Високо

$n = 0-2, x = 0-7$

Табл. 11 Разрешаване и задаване на софтуерен приоритет

За повече информация вижте описанието на регистрите IRQnENH и IRQnENL ($n = 0 - 2$).

Когато процесорът стартира обработка на прекъсване, съответния флаг за заявка на прекъсването автоматично се нулира. Запис на 0 във флага също го нулира.

	ZiLOG препоръчва следния начин за нулиране на флаг на прекъсване: IRQ0 &= MASK; където MASK съдържа 0 в позицията, която трябва да се нулира в регистър IRQ0.
---	--

Флаговете за заявка за прекъсване освен хардуерно, могат да се установяват в 1 и софтуерно. Софтуерните прекъсвания се обработват по същия начин както и хардуерните.

Пример:

IRQ1 |= 0x20; // IRQ1<5> = 1

Тази инструкция установява бит 5 (заявка за прекъсване от PA5) в регистър IRQ1. Ако прекъсването е разрешено и няма друго активно прекъсване с по-висок приоритет, процесорът стартира неговата обработка.

7.2 Описание на управляващите регистри

Регистър IRQ0 (Interrupt Request 0)

Бит	7	6	5	4	3	2	1	0
Име		T1I	T0I					ADCI
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC0h							
Бит	Описание							
7	Този бит е резервиран и трябва да бъде 0.							
6	Заявка за прекъсване от Таймер1 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							
5	Заявка за прекъсване от Таймер0 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							
4-1	Тези битове са резервирани и трябва да бъдат 0.							
0	Заявка за прекъсване от АЦП 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							

Табл. 12 IRQ0

Регистър IRQ1 (Interrupt Request 1)

Бит	7	6	5	4	3	2	1	0
Име	PA7I	PA6CI	PA5I	PA4I	PA3I	PA2I	PA1I	PA0I
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC3h							
Бит	Описание							
7	Заявка за прекъсване от PA7 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							
6	Заявка за прекъсване от PA6 или Компаратора 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							
5-0	Заявка за прекъсване от PAx (x = 5-0) 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							

Табл. 13 IRQ1

Регистър IRQ2 (Interrupt Request 2)

Бит	7	6	5	4	3	2	1	0
Име					PC3I	PC2I	PC1I	PC0I
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF6h							
Бит	Описание							
7-4	Тези битове са резервирани и трябва да бъдат 0.							
3-0	Заявка за прекъсване от PCx (x = 3-0) 0 – няма заявка за прекъсване 1 – има заявка за прекъсване							

Табл. 14 IRQ2

Регистър IRQ0ENH (IRQ0 Enable High Bit)

Бит	7	6	5	4	3	2	1	0
Име		T1ENH	T0ENH					ADCENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC1h							
Бит	Описание							
7	Този бит е резервиран и трябва да бъде 0.							
6	Старши бит за разрешаване на прекъсване от Таймер1							
5	Старши бит за разрешаване на прекъсване от Таймер0							
4-1	Тези битове са резервирани и трябва да бъдат 0.							
0	Старши бит за разрешаване на прекъсване от АЦП							

Табл. 15 IRQ0ENH

Регистър IRQ0ENL (IRQ0 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име		T1ENL	T0ENL					ADCENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч	Ч/З	Ч/З	Ч/З	Ч/З	Ч	Ч	Ч/З
Адрес	FC2h							
Бит	Описание							
7	Този бит е резервиран и трябва да бъде 0.							
6	Младши бит за разрешаване на прекъсване от Таймер1							
5	Младши бит за разрешаване на прекъсване от Таймер0							
4-1	Тези битове са резервирани и трябва да бъдат 0.							
0	Младши бит за разрешаване на прекъсване от АЦП							

Табл. 16 IRQ0ENL

Регистър IRQ1ENH (IRQ1 Enable High Bit)

Бит	7	6	5	4	3	2	1	0
Име	PA7ENH	PA6CENH	PA5ENH	PA4ENH	PA3ENH	PA2ENH	PA1ENH	PA0ENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC4h							
Бит	Описание							
7	Старши бит за разрешаване на прекъсване от PA7							
6	Старши бит за разрешаване на прекъсване от PA6/Компаратор							
5	Старши бит за разрешаване на прекъсване от PAx (x = 5-0)							

Табл. 17 IRQ1ENH

Регистър IRQ1ENL (IRQ1 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име	PA7ENL	PA6CENL	PA5ENL	PA4ENL	PA3ENL	PA2ENL	PA1ENL	PA0ENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC5h							
Бит	Описание							
7	Младши бит за разрешаване на прекъсване от PA7							
6	Младши бит за разрешаване на прекъсване от PA6/Компаратор							
5	Младши бит за разрешаване на прекъсване от PAx (x = 5-0)							

Табл. 18 IRQ1ENL

Регистър IRQ2ENH (IRQ2 Enable High Bit)

Бит	7	6	5	4	3	2	1	0
Име					C3ENH	C2ENH	C1ENH	C0ENH
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC7h							
Бит	Описание							
7-4	Тези битове са резервирани и трябва да бъдат 0.							
3-0	Старши бит за разрешаване на прекъсване от PCx (x = 3-0)							

Табл. 19 ERQ2ENH

Регистър IRQ2ENL (IRQ2 Enable Low Bit)

Бит	7	6	5	4	3	2	1	0
Име					C3ENL	C2ENL	C1ENL	C0ENL
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FC8h							
Бит	Описание							
7-4	Тези битове са резервирани и трябва да бъдат 0.							
3-0	Младши бит за разрешаване на прекъсване от PCx (x = 3-0)							

Табл. 20 IRQ2ENL

Регистър IRQES (Interrupt Edge Select)

Бит	7	6	5	4	3	2	1	0
Име	IES7	IES6	IES5	IES4	IES3	IES2	IES1	IES0
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FCDh							

Бит	Описание
7-0	Избор на активен фронт на прекъсване 0 – Прекъсване по падащ фронт на PAx или PDx (x = 7-0) 1 – Прекъсване по нарастващ фронт на PAx или PDx (x = 7-0)

Табл. 21 IRQES

Регистър IRQSS (Shared Interrupt Select)

Бит	7	6	5	4	3	2	1	0
Име		PA6CS						
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FCEh							
Бит	Описание							
7	Този бит е резервиран и трябва да бъде 0.							
6	Избор на източник на прекъсване от PA6/Компаратор 0 – Прекъсване от PA6 1 – Прекъсване от Компаратора							
5-0	Тези битове са резервирани и трябва да бъдат 0.							

Табл. 22 IRQSS

Регистър IRQCTL (Interrupt Control)

Бит	7	6	5	4	3	2	1	0
Име	IRQE							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FCFh							
Бит	Описание							

7	Глобален бит за разрешаване на прекъсванията 0 – Прекъсванията са забранени 1 – Прекъсванията са разрешени
6-0	Тези битове са резервирани и трябва да бъдат 0.

Табл. 23 IRQCTL

8 Генераторен блок

8.1 Общи сведения

Z8Encore!F083A серията микроконтролери има 4 източника на тактови сигнали, които могат да се използват за системен тактов сигнал:

- Вътрешен прецизен настройваем RC генератор (F_{IPRO});
- Вътрешен генератор с външен кварцов или керамичен резонатор, или външна RC верига (F_{IQO});
- Тактов сигнал от външен генератор (F_{EXT});
- Ниско прецизен генератор на WDT-таймера (F_{WDT}).

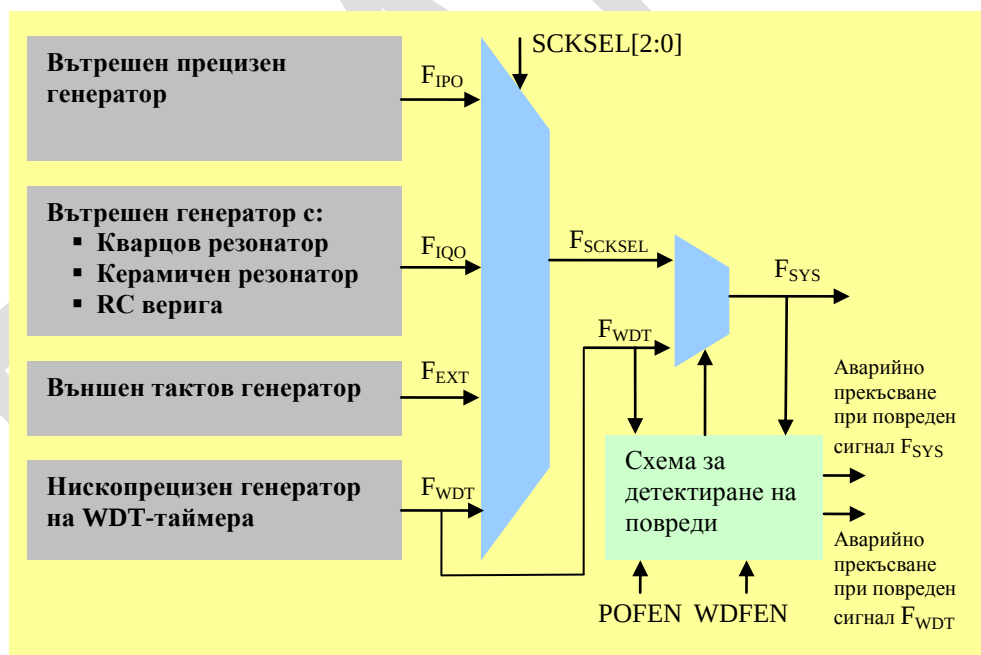
Табл.24 описва характеристиките на описаните по-горе източници на тактови сигнали.

Генератор	Характеристики
Вътрешен прецизен RC генератор	▪ 119kHz или 20MHz ▪ $\pm 4\%$ точност, когато е настроен ▪ не изисква външни компоненти
Вътрешен генератор с външен кварцов или керамичен резонатор	▪ 32kHz ÷ 20MHz ▪ много висока точност (зависи от използвания кварцов или керамичен резонатор) ▪ изисква външни компоненти
Вътрешен генератор с външна RC верига	▪ 32kHz ÷ 4MHz ▪ точността зависи от външни компоненти
Тактов сигнал от външен	▪ 0 ÷ 20MHz

генератор	■ точността зависи от външния генератор
Нископрецизен генератор на WDT-таймера	■ 10kHz ■ $\pm 40\%$ точност ■ не изисква външни компоненти ■ ниска консумация-

Табл. 24 Генератори на тактови сигнали

В допълнение Z8Encore!F083A серията микроконтролери има схема за детектиране и възстановяване на повреден системен тактов сигнал. Следващата фигура илюстрира обобщения изглед на генераторния блок.



Фиг. 18 Генераторен блок

8.2 Детектиране и възстановяване на повреден системен тактов сигнал

Повреда в генератора на системния тактов сигнал

Z8Encore!F083A серията микроконтролери генерира немаскируемо прекъсване в случай на детектиране на повреден системен тактов сигнал. За да поддържа устройството работоспособно, детектиращата схема автоматично превключва RC генератора на WDT-таймера да бъде източник на системния тактов сигнал (този генератор трябва да бъде разрешен преди това). Тази функционалност не е налична, ако RC генераторът на WDT-таймера е забранен. Функционалността не е налична също, ако RC генераторът на WDT-таймера се използва като източник на системния тактов сигнал.

Повреда в системния тактов сигнал се детектира, ако честотата му падне под $1\text{kHz} \pm 50\%$. Фалшива повреда може да се детектира и ако системният тактов сигнал се генерира от външен генератор с честота под допустимата. В този случай детектиращата схема може да бъде изключена (бит POFEN = 0 в регистър OSCCTL).

Повреда в RC генератора на WDT-таймера

Z8Encore!F083A серията микроконтролери генерира немаскируемо прекъсване в случай на детектиране на повреден тактов сигнал на RC генератора на WDT-таймера. Това не води до превключване на системния тактов сигнал. След детектиране на такава повреда, детектиране на повреда в системния тактов сигнал е невъзможна. Детектиращата схема не работи, ако RC генераторът на WDT-таймера е забранен или се използва като източник на системния тактов сигнал. Детектиращата схема може да бъде изключена (бит

WDFEN = 0 в регистър OSCCTL). Схемата за детектиране на грешка в RC генератора на WDT-таймера отброява 8004 системни такта, докато следи тактовия сигнал на RC генератора преди да определи дали има повреда. Скоростта на системния такт определя скоростта на детектиране на грешка в RC генератора на WDT-таймера. Много бавен системен такт води до много дълго време за детектиране.

8.3 Кварцов генератор

Кварцовият генератор (IQO – **I**nternal **Q**uartz **O**scillator) използва външен кварцов резонатор, за да генерира честоти в обхвата 32KHz ÷ 20MHz. Може да се използва също и керамичен резонатор с честота до 8MHz или RC верига с честота до 4MHz. Кварцовият генератор се използва да генерира основния тактов сигнал за процесора eZ8 и за по-голямата част от периферните модули. В допълнение входният извод XIN¹ също може да приема CMOS-тактов сигнал с честоти 32KHz ÷ 20MHz. Ако се използва външен тактов генератор, извод XOUT трябва да бъде оставен несвързан.

Забележка¹: Въпреки че XIN може да се използва като вход за външен тактов сигнал, извод CLKIN е по-подходящ за тази цел.

Кварцовият генератор поддържа 4 режима на работа:

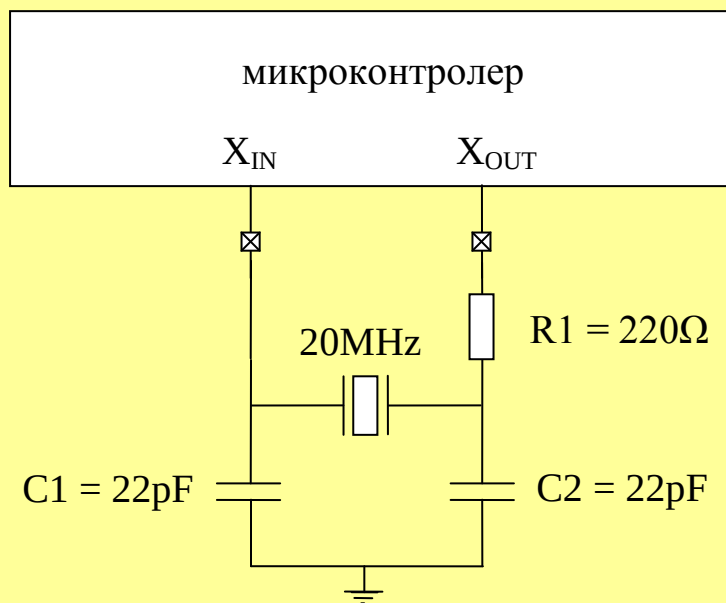
- режим с ниска консумация с кварцови резонатори 32kHz ÷ 1MHz;
- режим със средна консумация с кварцови резонатори 0.5MHz ÷ 8MHz;

- режим с висока консумация с кварцови резонатори 8MHz ÷ 20MHz;
- външна RC верига (< 4MHz).

Режимът на работа се избира с флаш-конфигурационните битове (вижте [16 Флаш-конфигурационни битове](#)).

Кварцовият генератор може да бъде разрешен автоматично след ресет посредством флаш-конфигурационният бит XTLDIS или ръчно чрез бит XTLEN в регистър OSCCTL. В последния случай потребителският код трябва да изчака поне 5000 цикъла на вътрешния прецизен генератор за стабилизация на тактовия сигнал на кварцовия генератор. След изтичане на това време кварцовият генератор може да бъде избран за системен тактов генератор.

Следващата фигура показва препоръчителните стойности на външните компоненти на кварцовия генератор с 20MHz кварцов резонатор.



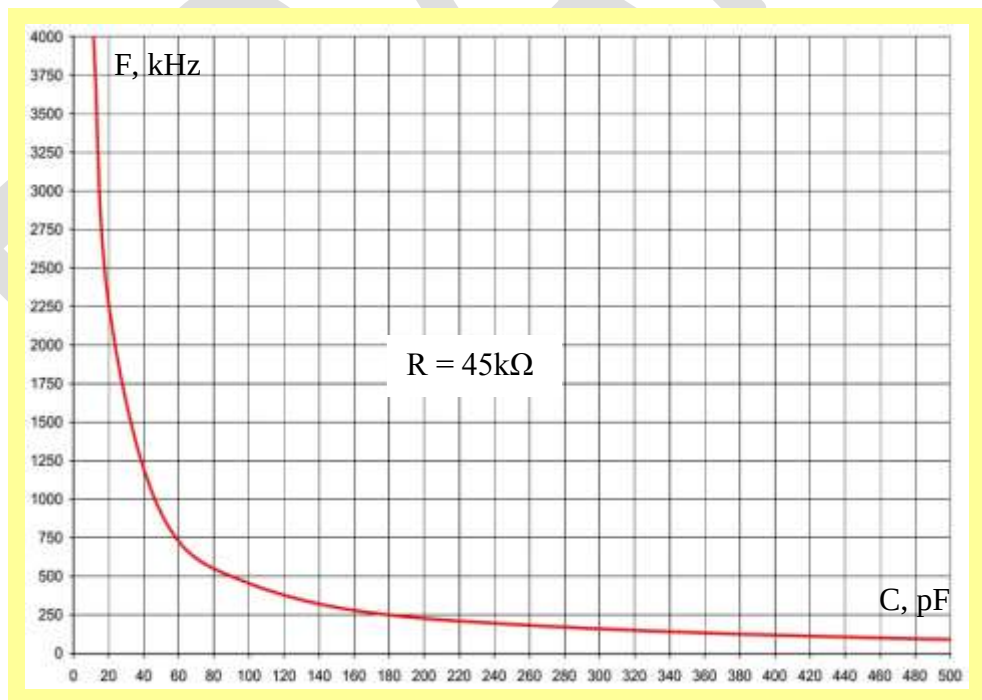
Фиг. 19 Външни компоненти на кварцов генератор

Следващата фигура показва използването на външна RC група.



Фиг. 20 Външни компоненти на RC генератор

Фиг.21 показва типични честоти при $V_{DD} = 3.3\text{V}$, $T = 25^\circ\text{C}$ и $R = 45\text{k}\Omega$.



Фиг. 21 Типични честоти при $R=45\text{k}\Omega$ и различни стойности на C



В RC режим на работа генераторът спира да осцилира, ако захранващото напрежение падне под 2.7V, но преди захранването да падне до V_{VBO} . Генераторът възстановява работата си, когато захранващото напрежение превиши 2.7V.

8.4 Вътрешен прецизен генератор

Вътрешният прецизен генератор (IPO - Internal Precision Oscillator) работи без външни компоненти с честота 20MHz или 119 kHz. Максималното време за стартиране на IPO е 25 μ s. IPO-генераторът е фабрично настроен на горните честоти.

8.5 Описание на управляващите регистри

Регистър OSCCTL (Oscillator Control Register)

Бит	7	6	5	4	3	2	1	0
Име	INTEN	XTLEN	WDTEN	POFEN	WDFEN	SCKSEL		
Ресет	1	0	1	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F86h							
Бит	Описание							
7	Разрешаване на IPO генератора 0 – IPO е забранен 1 – IPO е разрешен							
6	Разрешаване на кварцовия генератор 0 – Кварцовият генератор е забранен 1 – Кварцовият генератор е разрешен							
5	Разрешаване на RC генератора на WDT-таймера							

	0 – RC генераторът е забранен 1 – RC генераторът е разрешен
4	Разрешаване на детектиращата схема на системния такт 0 – детектиращата схема е забранена 1 – детектиращата схема е разрешена
3	Разрешаване на детектиращата схема на RC генератора на WDT-таймера 0 – детектиращата схема е забранена 1 – детектиращата схема е разрешена
2-0	Избор на източник на системен тактов сигнал 000 – IPO с честота 20MHz 001 – IPO с честота 119kHz 010 – Кварцов генератор (кварцов или керамичен резонатор, RC) 011 – Генератор на WDT-таймера 100 – Външен тактов генератор 111 – Резервирана комбинация

Табл. 25 OSCCTL



Неволният достъп до регистър OSCCTL може да доведе до спиране на работата на микроконтролера, поради превключване към нефункциониращ генератор. За предпазване от подобна ситуация, регистър OSCCTL използва специален отключващ/заклучващ механизъм. За да запишете нова стойност в този регистър, той трябва да бъде отключен чрез последователен запис на стойностите E7h и 18h, след което може да запишете новата стойност. Записите на стойностите E7h и 18h трябва да бъде в този ред, но не е задължително да са последователни.

Пример:

OSCCTL = 0xE7;

OSCCTL = 0x18;

OSCCTL = нова-стойност;

8.6 Избор и превключване на системния тактов сигнал

Следващата таблица описва необходимите стъпки при избор на системния тактов сигнал от наличните генератори.

Генератор	Необходими стъпки
Вътрешен прецизен RC генератор 119kHz или 20MHz	Отключете и запишете регистър OSCCTL, за да разрешите и изберете IPO-генератора като източник на системния тактов сигнал с честота 119kHz или 20MHz.
Вътрешен генератор с външен кварцов или керамичен резонатор - кварцов резонатор (32kHz ÷ 1MHz) - кварцов (керамичен) резонатор (0.5MHz ÷ 5MHz) - кварцов резонатор (5MHz ÷ 20MHz)	- Конфигурирайте флаш-конфигурационните битове OSC_SEL[1:0], за да изберете режима на работа на IQO-генератора. - Отключете и запишете регистър OSCCTL, за да разрешите IQO-генератора. - Изчакайте сигнала на IQO-генератора да се стабилизира¹. - Отключете и запишете регистър OSCCTL, за да изберете IQO-генератора като източник на системния тактов сигнал.
Вътрешен генератор с външна RC верига 32kHz ÷ 4MHz	- Конфигурирайте флаш-конфигурационните битове OSC_SEL[1:0], за да изберете режима на работа на IQO-генератора. - Отключете и запишете регистър OSCCTL, за да разрешите и изберете IQO- като източник на системния тактов сигнал.

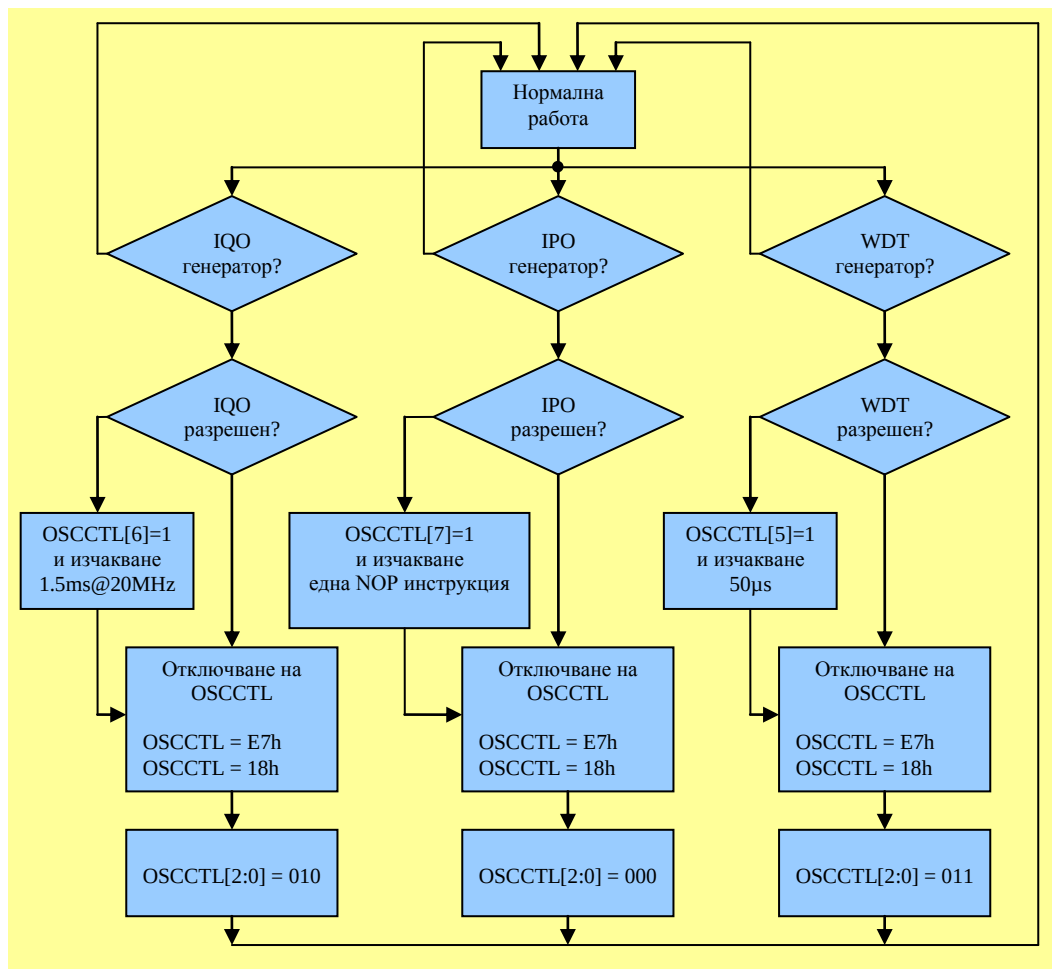
Тактов сигнал от външен генератор ▪ 0 ÷ 20MHz	▪ Конфигурирайте извод PB3 да работи като вход за външен тактов сигнал (алтернативна функция CLKIN). ▪ Отключете и запишете регистър OSCCTL, за да изберете външен тактов генератор като източник на системния тактов сигнал.
Нископрецизен генератор на WDT-таймера ▪ 10kHz	▪ Отключете и запишете регистър OSCCTL, за да разрешите и изберете WDT- като източник на системния тактов сигнал.

Табл. 26 Избор на системен тактов сигнал

Забележка¹: Ако флаш-конфигурационният бит XTLDIS е 0 (кварцовият генератор се включва при ресет), изчакване не е необходимо.

	При превключване на системния тактов сигнал към друг източник, схемата за детектиране на повреди в системния тактов сигнал и сигнала от WDT-генератора трябва да бъде изключена (битове POFEN и WDTFEN трябва да се нулират). В противен случай при превключване на тактовия сигнал може да се детектира фалшива повреда.
---	--

Следващата диаграма описва механизма на превключване на системния тактов сигнал от един източник към друг.



Фиг. 22 Диаграма на превключване на системния тактов сигнал

9 Режими с ниска консумация

9.1 Общи сведения

Z8Encore!F083A серията микроконтролери има два режима на ниска консумация:

- STOP-режим;
- HALT-режим.



Потребителят не трябва да разрешава изтеглящите резистори на неизползваните GPIO изводи.

STOP-режим

Микроконтролерът влиза в STOP-режим при изпълнение на инструкцията STOP. Работните характеристики на този режим са следните:

- Кварцовият и вътрешният прецизен генератор са спрени;
- Системният тактов сигнал е спрян;
- Процесорът eZ8 е спрян (не изпълнява инструкции, програмният брояч не се увеличава);
- RC генераторът на WDT-таймера продължава да работи (ако е разрешен);
- WDT-таймера продължава да работи (ако е разрешен);
- VBO схемата продължава да работи (ако е разрешена);
- Всички останали периферни модули са спрени.

За допълнително намаляване на консумацията, всички GPIO изводи, които са конфигурирани като цифрови входове трябва да бъдат отведени до V_{DD} , когато изтеглящите резистори са разрешени, или към една от захранващите шини (V_{DD} или GND), когато изтеглящите резистори са забранени.

За излизане от този режим вижте [6.3 Излизане от STOP-режим](#).

HALT-режим

Микроконтролерът влиза в HALT-режим при изпълнение на инструкцията HALT. Работните характеристики на този режим са следните:

- Основният генератор е разрешен и продължава да работи;
- Системният тактов сигнал продължава да работи;
- Процесорът eZ8 е спрян (не изпълнява инструкции, програмният брояч не се увеличава);
- RC генераторът на WDT-таймера продължава да работи;
- WDT-таймера продължава да работи (ако е разрешен);
- Всички останали периферни модули продължават да работят.

За допълнително намаляване на консумацията, всички GPIO изводи, които са конфигурирани като цифрови входове, трябва да бъдат отведени до V_{DD} , когато изтеглящите

резистори са разрешени, или към една захранващите шини (V_{DD} или GND), когато изтеглящите резистори са забранени.

Микроконтролерът излиза от този режим при възникване на някое от следните събития:

- Прекъсване;
- Препълване на WDT-таймера (ресет или прекъсване);
- POR ресет;
- VBO ресет;
- Ресет от извод $\overline{\text{RESET}}$.

9.2 Описание на управляващите регистри

Регистър PWRCTL0 (Power Control Register 0)

Всеки бит в този регистър забранява даден периферен модул или чрез изключване на тактовия сигнал или чрез изключване на захранването към този модул.

Бит	7	6	5	4	3	2	1	0
Име				VBO			COMP	
Ресет ¹	1	0	0	0	1	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F80h							
Бит	Описание							
7-5	Тези битове са резервирани и трябва да бъдат 0.							
4	Забраняване на VBO схемата 0 – VBO схемата е разрешена							

	1 – VBO схемата е забранена
3-2	Тези битове са резервирани и трябва да бъдат 0.
1	Забраняване на Компаратора 0 – Компараторът е разрешен 1 – Компараторът е забранен
0	Този бит е резервиран и трябва да бъде 0.

Табл. 27 PWRCTL0

Забележка¹: Този регистър се установява в начално състояние само от POR ресет. Други видове ресет не оказват влияние върху него.

10 Портове

10.1 Общи сведения

Z8Encore!F083A серията микроконтролери имат максимум 23 входни/изходни изводи за обща употреба (**GPIO – General Purpose Input Output**). Тези изводи са групирани в портове (Порт А ÷ Порт D). Всеки порт има набор от конфигурационни регистри, които настройват функцията на всеки извод (цифров вход/изход, аналогов вход, таймерен вход/изход и т.н.) и други (вижте [3 Описание на изводите](#)).

Следващата таблица описва портовете и изводите в тях за различните устройства от Z8Encore!F083A серията.

Устройство	Корпус	АЦП	Порт А	Порт В	Порт С	Порт D	Брой изводи
Z8F083A Z8F043A	20-извода	Да	7:0	3:0	3:0	0	17
Z8F083A Z8F043A	28-извода	Да	7:0	5:0	7:0	0	23

Табл. 28 Брой налични изводи според корпуса

10.2 Алтернативни функции на изводите

Ако се върнете отново към точка [3 Описание на изводите](#), ще видите, че повечето изводи могат да изпълняват по няколко функции, а не само изводи за цифров вход/изход. Когато един извод е конфигуриран да работи в някой от алтернативните си режими, управлението му се поема от съответния периферен модул.

PD0 / RESET

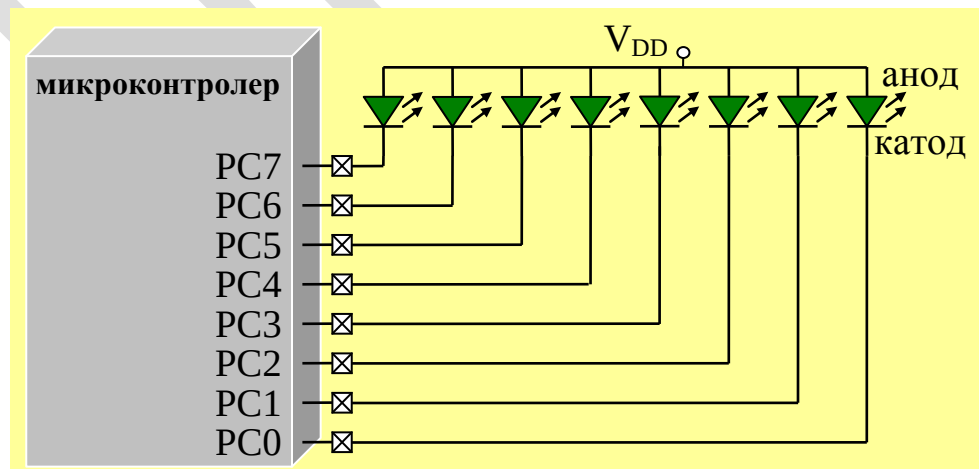
PD0 и $\overline{\text{RESET}}$ използват общ извод. За разлика от изводите на другите портове този извод не работи като GPIO след ресет, а като двупосочен (вход/изход) ресет извод с отворен-дрейн с вътрешен изтеглящ резистор. В GPIO-режим PD0 може да работи само като изход и трябва да бъде конфигуриран като такъв. PD0 може да работи като изход с повишена мощност, но не може да се използва да извежда микроконтролера от STOP-режим.

PA0/XIN и PA1/XOUT

Функционалността на кварцовия генератор не се управлява от GPIO блока. Когато този генератор е разрешен, GPIO функционалността на изводи PA0 и PA1 е изключена и те работят като XIN и XOUT съответно.

Директно управление на светодиоди

Изводите на Порт С могат да управляват директно светодиоди (**LED – Light Emitting Diode**). Изходите поемат ток с програмируеми нива 3mA, 7mA, 13mA и 20mA. Анодът на светодиода трябва да бъде свързан към V_{DD} , а катодът към извода (Фиг.23)



Фиг. 23 Свързване на светодиоди към Порт С

5V-толерантни изводи

Всички изводи издържат на напрежение 5V, с изключение на изводите, които работят като аналогови входове, изводите на кварцовия генератор, изводите на компаратора, PA[1:0], PB[5:0] и PC[2:0].

Външен тактов сигнал

За устройства, които работят с външен TTL тактов сигнал, извод PB3 служи за тактов вход. В такъв случай конфигурирайте извода да използва алтернативната си функция CLKIN и чрез регистър OSCCTL изберете външния тактов сигнал за системен такт.

Прекъсвания от изводите

Много от изводите могат да генерират прекъсвания по нарастващ, падащ или и по двата фронта. За повече информация вижте [7 Контролер на прекъсванията](#).

Следващите таблици описват всички изводи на Порт А, В, С и D, алтернативните функции, които могат да изпълняват и управляващите битове, с които се избира съответната алтернативна функция.

Порт А			
Извод	Алтернативна Функция	Описание	Регистър
PA0	T0IN/ <u>T0OUT</u>	Таймерен вход 0/ Инвертиран таймерен изход 0	RAAF[0] = 1
PA1	T0OUT	Таймерен изход 0	RAAF[1] = 1
PA2	-	-	-
PA3	-	-	-
PA4	-	-	-
PA5	-	-	-
PA6	T1IN/ <u>T1OUT</u>	Таймерен вход 1/ Инвертиран	RAAF[6] = 1

		таймерен изход 1	
PA7	T1OUT	Таймерен изход 1	PAAF[7] = 1

Табл. 29 Алтернативни функции на Порт А

Порт В			
Извод	Алтернативна Функция	Описание	Регистър
PB0	-	-	PBAFS1 [0] = 0
	ANA0	Аналогов вход 0	PBAFS1 [0] = 1
PB1	-	-	PBAFS1 [1] = 0
	ANA1	Аналогов вход 1	PBAFS1 [1] = 1
PB2	-	-	PBAFS1 [2] = 0
	ANA2	Аналогов вход 2	PBAFS1 [2] = 1
PB3	CLKIN	Вход за външен тактов сигнал	PBAFS1 [3] = 0
	ANA3	Аналогов вход 3	PBAFS1 [3] = 1
PB4	-	-	PBAFS1 [4] = 0
	ANA7	Аналогов вход 7	PBAFS1 [4] = 1
PB5	-	-	PBAFS1 [5] = 0
	VREF	Вход за опорно напрежение	PBAFS1 [5] = 1
PB6	-	-	PBAFS1 [6] = 0
	-	-	PBAFS1 [6] = 1
PB7	-	-	PBAFS1 [7] = 0
	-	-	PBAFS1 [7] = 1

Табл. 30 Алтернативни функции на Порт В

Порт С			
Извод	Алтернативна Функция	Описание	Регистър
PC0	-	-	PCAFS1 [0] = 0
	ANA4/CINP	Аналогов вход 4/Компараторен вход +	PCAFS1 [0] = 1
PC1	-	-	PCAFS1 [1] = 0
	ANA5/CINN	Аналогов вход 5/Компараторен вход -	PCAFS1 [1] = 1
PC2	-	-	PCAFS1 [2] = 0
	ANA6	Аналогов вход 6	PCAFS1 [2] = 1
PC3	COUT	Компараторен изход	PCAFS1 [3] = 0
	-	-	PCAFS1 [3] = 1

PC4	-	-	PCAFS1 [4] = 0
	-	-	PCAFS1 [4] = 1
PC5	-	-	PCAFS1 [5] = 0
	-	-	PCAFS1 [5] = 1
PC6	-	-	PCAFS1 [6] = 0
	-	-	PCAFS1 [6] = 1
PC7	-	-	PCAFS1 [7] = 0
	-	-	PCAFS1 [7] = 1

Табл. 31 Алтернативни функции на Порт С

Порт D			
Извод	Алтернативна Функция	Описание	Регистър
PD0	RESET	Външен ресет. По подразбиране работи като ресет извод.	PDAF[0] = 1

Табл. 32 Алтернативни функции на Порт D

10.3 Управляващи регистри

Всеки порт има 4 управляващи регистри (Табл.33).

Регистър	Описание
RxADDR	Адресен регистър (избира подрегистър)
RxCTL	Управляващ регистър (осъществява достъп до подрегистър)
RxIN	Входен регистър
RxOUT	Изходен регистър

x = A,B,C и D

Табл. 33 Управляващи регистри

Освен тези регистри всеки порт има и няколко управляващи подрегистри (Табл.34).

Подрегистър	Адрес	Описание
-------------	-------	----------

PxDD	01h	Управление на посоката на данните (вход или изход)
PxAF	02h	Разрешаване на алтернативна функция
PxOC	03h	Изходен контрол (отворен-дрейн)
PxHDE	04h	Разрешаване на мощно управление
PxSMRE	05h	Разрешаване за излизане от STOP-режим
PxPUE	06h	Разрешаване на изтеглящите резистори
PxAFS1	07h	Алтернативна функция регистър 1
PxAFS2 ¹	08h	Алтернативна функция регистър 2

x = A,B,C и D

Табл. 34 Управляващи подрегистри

Забележка¹: Този регистър не се използва от Z8Encore!F083A серията микроконтролери.

Подрегистрите не са част от Регистровата Памет като всички останали управляващи регистри, а се адресират чрез регистър PxADDR (x = A, B, C и D). Достъпът до подрегистър става чрез регистър PxCTL (x = A, B, C и D). Например ако PAADDR = 01h, достъпът до регистър PACTL осъществява достъп до подрегистър PADD.

10.4 Описание на управляващите регистри

Регистър PxADDR (Port Address)

PxADDR регистъра служи като указател към управляващите подрегистри на всеки порт.

Бит	7	6	5	4	3	2	1	0
Име	PADDR[7:0]							
Ресет	00h							
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З

Адрес	FD0h, FD4h, FD8h, FDCh
Бит	Описание
7-0	Адрес на подрегистър 00 – 01 – PxDD 02 – PxAF 03 – PxOC 04 – PxHDE 05 – PxSMRE 06 – PxPUE 07 – PxAFS1 08 – PxAFS2

Табл. 35 PxADDR

Регистър PxCTL (Port Control)

PxCTL регистърът служи за достъп до управляващите подрегистри на всеки порт. Регистър PxADDR определя подрегистъра.

Бит	7	6	5	4	3	2	1	0
Име	PCTL[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FD1h, FD5h, FD9h, FDDh							
Бит	Описание							
7-0	Осъществява достъп до подрегистъра, сочен от PxADDR							

Табл. 36 PxCTL

Регистър PxDD (Port Direction)

Този регистър управлява посоката на изводите, когато

изводът е конфигуриран като GPIO. В алтернативен режим на работа, посоката на извода се управлява от съответния периферен модул.

Бит	7	6	5	4	3	2	1	0
Име	DD[7:0]							
Ресет	1	1	1	1	1	1	1	1
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 01h							
Бит	Описание							
7-0	Посока на данните 0 – Цифров Изход. Данните в регистър PxOUT се извеждат на извода. 1 – Цифров вход. Стойността на извода се чете и се записва в PxIN.							

Табл. 37 PxDD

Регистър PxAF(Port Alternate Function)

Този регистър разрешава алтернативната функция на съответния извод. Конкретната алтернативна функция на извода се избира с регистрите PxAFS1 и PxAFS2 (вижте Табл.80 и 81). Изключение са Порт А и D, при които алтернативната функция на извода се разрешава и избира директно чрез регистър PxAF (вижте Табл.29 и 32).

Бит	7	6	5	4	3	2	1	0
Име	AF[7:0]							
Ресет	00h (Порт А-С), 01h (Порт D)							
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 02h							
Бит	Описание							

7-0	Разрешаване на алтернативна функция на извод 0 – Алтернативната функция е забранена. Изводът работи като GPIO (цифров вход или изход). 1 – Алтернативната функция е разрешена.
-----	---

Табл. 38 PxAF

Регистър PxAFS1 (Port Alternate Set 1)

Този регистър избира алтернативната функция на специфицирания извод на Порт x (x = В или С) (вижте Табл.30 и 31). Алтернативната функция на извода също трябва да бъде разрешена в регистър PxAF.

Бит	7	6	5	4	3	2	1	0
Име	PAFS1[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 07h							
Бит	Описание							
7:0	0 – Вижте Табл. 30 и 31 1 – Вижте Табл. 30 и 31							

Табл. 39 PxAFS1

Регистър PxOC (Port Output Control)

Този регистър конфигурира специфицирания извод като отворен-дрейн независимо дали извода работи в GPIO или алтернативен режим.

Бит	7	6	5	4	3	2	1	0
Име	POC[7:0]							
Ресет	0	0	0	0	0	0	0	0

Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 03h							
Бит	Описание							
7:0	Управление на изхода 0 – Режим отворен-дрейн забранен 1 – Режим отворен-дрейн разрешен							

Табл. 40 PxOC

Регистър PxHDE (Port High Drive Enable)

Този регистър конфигурира специфицирания извод за мощни изходни токови операции независимо дали извода работи в GPIO или алтернативен режим.

Бит	7	6	5	4	3	2	1	0
Име	PHDE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 04h							
Бит	Описание							
7:0	Разрешаване на мощно управление 0 – Изводът е конфигуриран за стандартни изходни токови операции 1 – Изводът е конфигуриран за мощни изходни токови операции							

Табл. 41 PxHDE

Регистър PxSMRE (Port Stop Mode Recovery Enable)

Този регистър разрешава специфицирания извод да извежда процесора от STOP-режим. Всеки преход на извода

(нарастващ или падащ фронт) стартира процедура по извеждане на процесора от STOP-режим (вижте [6.3 Излизане от STOP-режим](#)).

Бит	7	6	5	4	3	2	1	0
Име	PSMRE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 05h							
Бит	Описание							
7-0	0 – Излизането от STOP-режим забранено 1 – Излизането от STOP-режим разрешено							

Табл. 42 PxSMRE

Регистър PxPUE (Port Pull-up enable)

Този регистър разрешава вътрешния изтеглящ резистор на изводите на Портa A÷D.

Бит	7	6	5	4	3	2	1	0
Име	PPUE[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	PxADDR = 06h							
Бит	Описание							
7:0	0 – Изтеглящият резистор е забранен 1 – Изтеглящият резистор е разрешен							

Табл. 43 PxPUE

Регистър PxIN (Port Input Data)

Когато специфицираният извод е конфигуриран като цифров вход, нивото на този извод (лог.0 или 1) се записва в регистър PxIN (x = A, B и C). Извод PD0 не може да работи като цифров вход.

Бит	7	6	5	4	3	2	1	0
Име	PIN[7:0]							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FD2h, FD6h, FDAh							
Бит	Описание							
7:0	0 – Логическа 0 на входа 1 – Логическа 1 на входа							

Табл. 44 PxIN

Регистър PxOUT (Port Output Data)

Данните в този регистър се извеждат на съответните изводи, когато са конфигурирани като цифрови изходи.

Бит	7	6	5	4	3	2	1	0
Име	POUT[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FD3h, FD7h, FDBh, FDFh							
Бит	Описание							
7:0	0 – Извеждане на лог.0 на извода 1 – Извеждане на лог.1 на извода. Логическата 1 не се извежда на извода, ако той е конфигуриран в режим отворен-дрейн (вижте Регистър PxOC).							

Табл. 45 PxOUT

Регистър LEDEN (LED Drive Enable)

Този регистър включва вътрешните схеми на изводите на Порт С, които директно могат да управляват светодиоди.

Бит	7	6	5	4	3	2	1	0
Име	LEDEN[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F82h							
Бит	Описание							
7:0	0 – Вътрешната схема за управление на LED е изключена 1 – Вътрешната схема за управление на LED е включена							

Табл. 46 LEDEN

Когато извод PCx ($x = 0 \div 7$) е конфигуриран да работи в LED режим, битовите в регистрите LEDLVLH и LEDLVLL конфигурират големината на тока (Табл.47).

LEDLVLHn	LEDLVLLn	LED ток
0	0	3mA
0	1	7mA
1	0	13mA
1	1	20mA

$n = 0 \div 7$

Табл. 47 LED големина на тока

Регистър LEDLVLH (LED Drive Level High)

Бит	7	6	5	4	3	2	1	0
Име	LEDLVLH[7:0]							
Ресет	0	0	0	0	0	0	0	0

Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F83h							
Бит	Описание							
7:0	Вижте Табл.47							

Табл. 48 LEDLVLH

Регистър LEDLVLL (LED Drive Level Low)

Бит	7	6	5	4	3	2	1	0
Име	LEDLVLL[7:0]							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F84h							
Бит	Описание							
7:0	Вижте Табл.47							

Табл. 49 LEDLVLL

11 WDT таймер

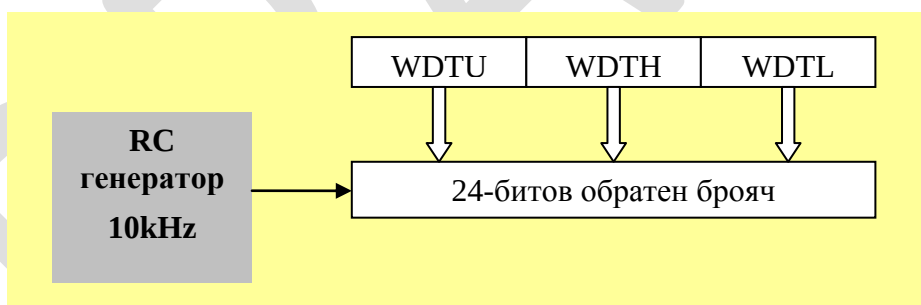
11.1 Общи сведения

WDT-таймерът защитава програмата от „зависване”. Той има следните свойства:

- Собствен вътрешен RC генератор на тактов сигнал;
- 24-битов програмируем период на препълване;
- Избираема реакция при препълване: ресет или прекъсване.

11.2 Устройство и принцип на работа

Фиг.24 показва блоковата схема на WDT-таймера.



Фиг. 24 Блокова схема на WDT-таймера

Веднъж разрешен таймерът брои постоянно и трябва да бъде обновяван периодично, за да се избегне препълване. WDT-таймерът се разрешава чрез изпълнение на инструкцията WDT или чрез флаш-конфигурационния бит WDT_AO. Този флаш-конфигурационен бит включва таймера веднага след излизане на процесора от ресет състояние, дори инструкцията WDT да не е била изпълнена.

Таймерът работи в режим на обратно броене. При разрешаването му 24-битовата стойност, формирана от 8-битовите регистри WDTU, WDTN и WDTL, се зарежда в брояча. На всеки тактов сигнал от RC генератора броячът се намалява с 1. Броячът се намалява до 000000h, освен ако не се изпълни инструкцията WDT. Изпълнението на тази инструкция принуждава презареждане на брояча с 24-битовата стойност от регистрите WDTU, WDTN и WDTL и броенето се възобновява. По този начин се избягва изтичането на периода на брояча и генериране на ресет или прекъсване от WDT-таймера.

Времето за изтичане на периода на таймера се изчислява по следната формула:

$$T_{WDT} (ms) = \frac{\text{Стойност за презареждане}}{10}$$

където *Стойност за презареждане* е 24-битовата десетична стойност формирана от 8-битовите регистри WDTU, WDTN и WDTL.

WDT-таймерът не може да се презареди след като достигне 000002h. Презареждащата стойност не трябва да бъде под 000004h. Табл.50 дава информация за приблизителното време за препълване на таймера при минимална и максимална презареждаща стойност.

Презареждаща стойност	Време за препълване	Описание
000004h	400μs	минимално
000400h	102ms	по подразбиране
FFFFFFh	28 минути	максимално

Табл. 50 Времена за препълване на таймера

WDT-таймерът се препълва (изтича периодът на броене),

когато броячът достигне 000000h. Реакцията при препълване може да бъде ресет или прекъсване. Това зависи от флаш-конфигурационния бит WDT_RES.

11.3 Реакция при препълване на WDT-таймера

Генериране на прекъсване в нормален режим

Ако е конфигуриран да генерира прекъсване по препълване, WDT-таймерът изпраща заявка за прекъсване към контролерът на прекъсванията и вдига флаг WDT в регистър RSTSTAT. Ако прекъсванията са разрешени, процесорът извлича вектора на прекъсване и изпълнява кода, намиращ се на този адрес. WDT-таймерът се зарежда с максималната си стойност FFFFFFFh и продължава да брои. Таймерът не се презарежда автоматично с презареждащата стойност в регистрите WDTU, WDTN и WDTL.

Регистър [RSTSTAT](#) трябва да бъде прочетен преди изчистване на прекъсването от таймера. Това четене изчиства флага WDT.

Генериране на прекъсване в STOP-режим

Ако е конфигуриран да генерира прекъсване по препълване и процесорът е в STOP-режим, WDT-таймерът автоматично инициира излизане от STOP-режим, генерира заявка за прекъсване и вдига флаговете WDT и STOP в регистър RSTSTAT. За повече детайли относно излизане от STOP-режим вижте [6.3 Излизане от STOP-режим](#).

Ако прекъсванията са разрешени, след излизане от STOP-режим процесорът извлича вектора на прекъсване и изпълнява кода, намиращ се на този адрес.

Генериране на ресет в нормален режим

Ако е конфигуриран да генерира ресет по препълване, WDT-таймерът вкарва процесорът в ресет състояние и вдига флага WDT в регистър RSTSTAT. За повече информации относно ресет състоянието вижте [6 Ресет и излизане от STOP-режим](#).

Генериране на ресет в STOP-режим

Ако е конфигуриран да генерира ресет по препълване и процесорът е в STOP-режим, WDT-таймерът автоматично инициира излизане от STOP-режим и вдига флаговете WDT и STOP в регистър RSTSTAT. За повече детайли относно излизане от STOP-режим вижте [6.3 Излизане от STOP-режим](#).

11.4 Запис в регистрите WDTU, WDTH и WDTL

Директният записът в тези регистри е забранен и изисква специална отключваща последователност.

1. Запишете 55h в регистър WDTCTL;
2. Запишете AAh в регистър WDTCTL;
3. Запишете стойност в WDTU;
4. Запишете стойност в WDTH;
5. Запишете стойност в WDTL.

Презареждащите регистри трябва да бъдат записани в указания по-горе ред. Не извършвайте други междинни записи между тези операции, в противен случай заключващият механизъм се активира отново.

Презареждащата стойност в тези регистри се зарежда в брояча при първото разрешаване на таймера и всеки път при изпълняване на инструкцията WDT.

11.5 Описание на управляващите регистри

Регистър WDTCTL (Watchdog Timer Control)

Този регистър е само за запис. Запис на отключващата последователност 55h AAh в него отключва достъпа до презареждащите регистри WDTU, WDTN и WDTL. Операциите за запис в регистър WDTCTL нямат ефект върху неговите битове.

Бит	7	6	5	4	3	2	1	0
Име	WDTUNLK							
Ресет	x	x	x	x	x	x	x	x
Ч/З	3	3	3	3	3	3	3	3
Адрес	FF0h							
Бит	Описание							
7:0	Потребителският софтуер трябва да запише правилната отключваща последователност в този регистър преди да запише презареждащите регистри							

Табл. 51 WDTCTL

Регистър WDTU (Watchdog Timer Reload Upper Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTU							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3
Адрес	FF1h							

Бит	Описание
7:0	Битове 23:16 на презареждащата стойност

Табл. 52 WDTU

Забележка¹: Четенето на този регистър връща текущата стойност на брояча.

Регистър WDTN (Watchdog Timer Reload High Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTN							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3
Адрес	FF2h							
Бит	Описание							
7:0	Битове 15:8 на презареждащата стойност							

Табл. 53 WDTN

Забележка¹: Четенето на този регистър връща текущата стойност на брояча.

Регистър WDTL (Watchdog Timer Reload Low Byte)

Бит	7	6	5	4	3	2	1	0
Име	WDTL							
Ресет	x	x	x	x	x	x	x	x
Ч/З	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3	Ч ¹ /3
Адрес	FF3h							

Бит	Описание
7:0	Битове 7:0 на презареждащата стойност

Табл. 54 WDTL

Забележка¹: Четенето на този регистър връща текущата стойност на брояча.

12 Таймери

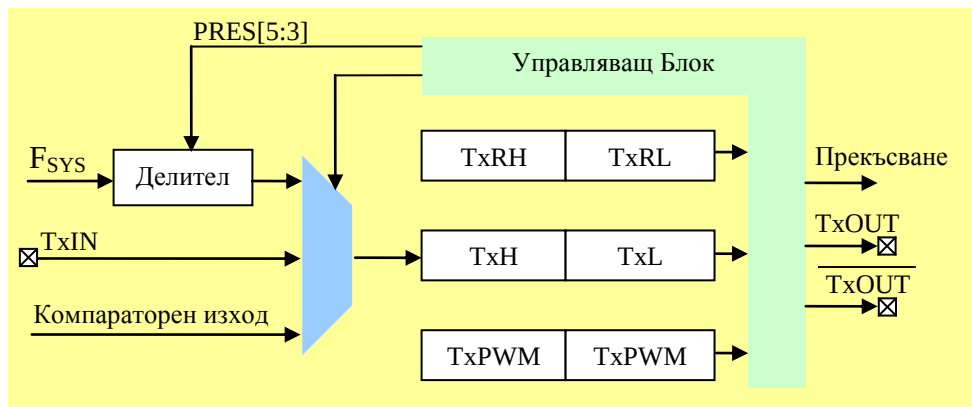
12.1 Общи сведения

Z8Encore!F083A серията микроконтролери съдържа два идентични 16-битови многофункционални таймери (Таймер 0 и Таймер 1) за отмерване на време, броене на събития, генериране на импулси, прихващане (**capture**) и сравнение (**compare**). Таймерите имат следните характеристики:

- 16-битови таймери;
- Програмируем входен делител с коефициенти на делене от 1 до 128;
- Генериране на PWM (**Pulse-Width Modulation**);
- Режим на прехващане;
- Режим на сравнение;
- Извод за таймерен вход, прихващане или контрол на броенето. Честотата на външния входен сигнал може да е максимум $\frac{1}{4}$ от честотата на системния тактов сигнал;
- Извод за таймерен изход;
- Генериране на прекъсване.

12.2 Устройство и принцип на работа

Фиг.25 показва блоковата схема на Таймерите. Таймерите са 16-битови прави броячи, състоящи се от два 8-битови броячни регистри TxH и TxL, два 8-битови регистри TxRH



Фиг. 25 Блокова схема на Таймер x (x = 0 или 1)

и TxRL, съдържащи максималната броячна стойност и два 8-битови регистри TxPWMH и TxPWML, които се използват в различните режими на работа на таймерите.

Принципът на работа на таймера зависи от избрания режим. Следващата точка описва всички режими на работа на таймерите.

12.3 Режими на работа

Еднократен режим (single-shot mode)

В този режим таймерът брои до 16-битовата броячна стойност, съхранена в 8-битовите регистри TxRH и TxRL. Системният тактов сигнал F_{SYS} служи за тактов сигнал на таймера. При достигане на броячната стойност таймерът генерира прекъсване, броячните регистри се презареждат с 0001h, таймерът автоматично се забранява и спира да брои. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки когато конфигурирате таймера в еднократен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в еднократен режим;
 - Конфигурирате входния делител;
 - Конфигурирате началното ниво, ако използвате таймерния изход TxOUT.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL.
3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Ако е необходимо, разрешете прекъсването от таймера.
5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.
6. Разрешете таймера.

Периодът на препълване на таймера може да се определи от следното уравнение:

$$T(s) = \frac{(\text{Максимална Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{Системен Такт(Hz)}}$$

Непрекъснат режим (continuous mode)

В този режим таймерът брои до 16-битовата броячна стойност, съхранена в 8-битовите регистри TxRH и TxRL. Системният тактов F_{SYS} сигнал служи за тактов сигнал на таймера. При достигане на броячната стойност таймерът генерира прекъсване, броячните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TROL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки, когато конфигурирате таймера в непрекъснат режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в непрекъснат режим;
 - Конфигурирате входния делител;
 - Конфигурирате началното ниво, ако използвате таймерния изход TxOUT.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).
3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Ако е необходимо, разрешете прекъсването от таймера.

5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.
6. Разрешете таймера.

Периодът на препълване на таймера може да се определи от следното уравнение:

$$T(s) = \frac{\text{Максимална Броячна Стойност} * \text{Делител}}{\text{Системен Такт(Hz)}}$$

Ако началната стойност в броячните регистри е различна от 0001h, използвайте уравнението за еднократен режим, за да определите първия период на препълване на таймера.

Броячен режим

В този режим таймерът брои преходите на таймерния вход TxIN. Бит TPOL в регистър TxCTL1 определя активния фронт на входния сигнал, по който става броенето (падащ или нарастващ фронт). В броячен режим входният делител е забранен.



Честотата на входния сигнал трябва да бъде максимум $\frac{1}{4}$ от честотата на системния тактов сигнал.

При достигане на броячната стойност, съхранена в регистрите TxRH и TxRL, се генерира прекъсване, таймерните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки, когато конфигурирате таймера в

броячен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в броячен режим;
 - Изберете активния входен фронт (падащ или нарастващ). Тази настройка задава също и началното ниво на таймерния изход (ако е разрешен).
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL. Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера броенето винаги започва от 0001h.
3. Запишете максималната броячна стойност в регистрите TxRH и TxRL.
4. Ако е необходимо, разрешете прекъсването от таймера.
5. Конфигурирайте съответния извод в режим на таймерен вход.
6. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.
7. Разрешете таймера.

Броят на активните фронтове на таймерния вход може да се определи от следното уравнение:

$$N = \text{Текуща Броячна Стойност} - \text{Начална Броячна Стойност}$$

Компараторен броячен режим

В този режим таймерът брои преходите от компараторния изход. Бит TPOL в регистър TxCTL1 определя дали броенето се извършва по нарастващия или падащия фронт на компараторния сигнал. В компараторен броячен режим входният делител е забранен.



Честотата на компараторния сигнал трябва да бъде максимум $\frac{1}{4}$ от честотата на системния тактов сигнал.

При достигане на броячната стойност, съхранена в регистрите TxRH и TxRL, се генерира прекъсване, броячните регистри се презареждат с 0001h и броенето продължава. Ако таймерният изход TxOUT е разрешен, нивото на извода се превключва (от ниско във високо или обратно) при презареждането на таймера в 0001h. Бит TPOL в регистър TxCTL1 определя началното ниво на извода.

Следвайте следните стъпки, когато конфигурирате таймера в компараторен броячен режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в компараторен броячен режим;
 - Изберете активния входен фронт на компараторния сигнал (падащ или нарастващ). Тази настройка задава също и началното ниво на таймерния изход (ако е разрешен).
2. Запишете начална стойност на таймера в броячните

регистри TxH и TxL. Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Ако е необходимо, разрешете прекъсването от таймера.
5. Ако е необходимо, конфигурирайте съответния изход в режим на таймерен изход.
6. Разрешете таймера.

Броят на активните фронтове от компараторния изход може да се определи от следното уравнение:

$$N = \text{Текуща Броячна Стойност} - \text{Начална Броячна Стойност}$$

Едноизходен PWM режим

В този режим таймерът генерира широчинно-модулирани импулси (**PWM – Pulse-Width Modulation**) на таймерния изход TxOUT. Системният тактов сигнал F_{sys} служи за тактов сигнал. Таймерът първо брои до 16-битовата широчинна стойност съхранена в 8-битовите регистри TxPWMH и TxPWML. Когато таймерът достигне тази стойност, таймерният изход се превключва (от ниско във високо ниво или обратно). Таймерът продължава да брои, докато достигне максималната 16-битова броячна стойност в 8-битовите регистри TxRH и TxRL. При достигане на тази стойност таймерът генерира прекъсване, таймерните регистри се презареждат с 0001h, таймерният изход се превключва отново и процесът се повтаря отначало. Бит TPOL в регистър TxCTL1 определя началното ниво на

извода TxOUT.

Ако $T_{POL} = 0$, началното ниво на извода е ниско и се превключва във високо, когато $T_{xH}:T_{xL} == T_{xPWMH}:T_{xPWML}$. Когато $T_{xH}:T_{xL} == T_{xRH}:T_{xRL}$, изходът отново се превключва в ниско ниво.

Ако $T_{POL} = 1$, началното ниво на извода е високо и се превключва в ниско, когато $T_{xH}:T_{xL} == T_{xPWMH}:T_{xPWML}$. Когато $T_{xH}:T_{xL} == T_{xRH}:T_{xRL}$, изходът отново се превключва във високо ниво.

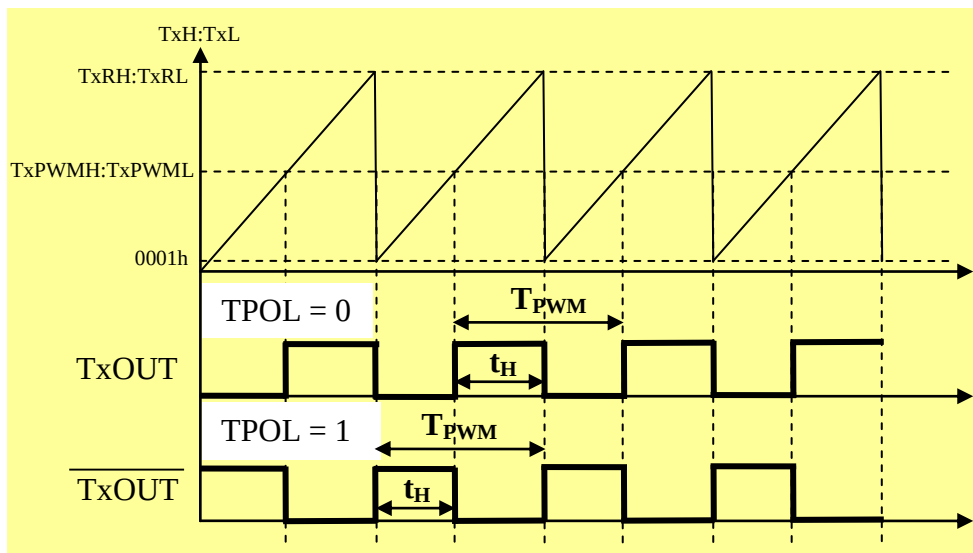
Периодът T_{PWM} на изходния сигнал се определя от максималната броячната стойност в $T_{xRH}:T_{xRL}$. По-голяма стойност съответства на по-голям период.

Коефициентът на запълване $k = t_H/T_{PWM}$ се определя от стойността в регистрите $T_{xPWMH}:T_{xPWML}$.

При $T_{POL} = 1$ по-голяма стойност съответства на по-голям коефициент на запълване. Когато $T_{xPWMH}:T_{xPWML} = 0001h$, $k = 0$. Когато $T_{xPWMH}:T_{xPWML} = T_{xRH}:T_{xRL}$, $k = 1$.

При $T_{POL} = 0$ по-малка стойност съответства на по-голям коефициент на запълване. Когато $T_{xPWMH}:T_{xPWML} = 0001h$, $k = 1$. Когато $T_{xPWMH}:T_{xPWML} = T_{xRH}:T_{xRL}$, $k = 0$.

Фиг.26 илюстрира описания процес.



Фиг. 26 Едноизходен PWM режим

Следвайте следните стъпки, когато конфигурирате таймера в едноизходен PWM режим:

1. Запишете управляващите регистри $TxCTL0$ и $TxCTL1$ за да:
 - Забраните таймера;
 - Конфигурирате таймера в едноизходен PWM режим;
 - Конфигурирате входния делител;
 - Конфигурирате началната ниво на таймерния изход $TxOUT$.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено $0001h$). Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от $0001h$.

3. Запишете широчинната стойност в регистрите TxPWMH и TxPWML.
4. Запишете максималната броячна стойност в регистрите TxRH и TxRL. Тази стойност трябва да е по-голяма от широчинната стойност.
5. Ако е необходимо, разрешете прекъсването от таймера.
6. Конфигурирайте съответния извод в режим на таймерен изход.
7. Разрешете таймера.

Периодът на PWM сигнала се определя от следното уравнение:

$$T_{PWM}(s) = \frac{\text{Максимална Броячна Стойност} * \text{Делител}}{\text{Системен Такт(Hz)}}$$

Ако TROL = 0, коефициентът на запълване се определя от следното уравнение:

$$k(\%) = \left(1 - \frac{\text{Широчинна Стойност}}{\text{Максимална Броячна Стойност}} \right) * 100$$

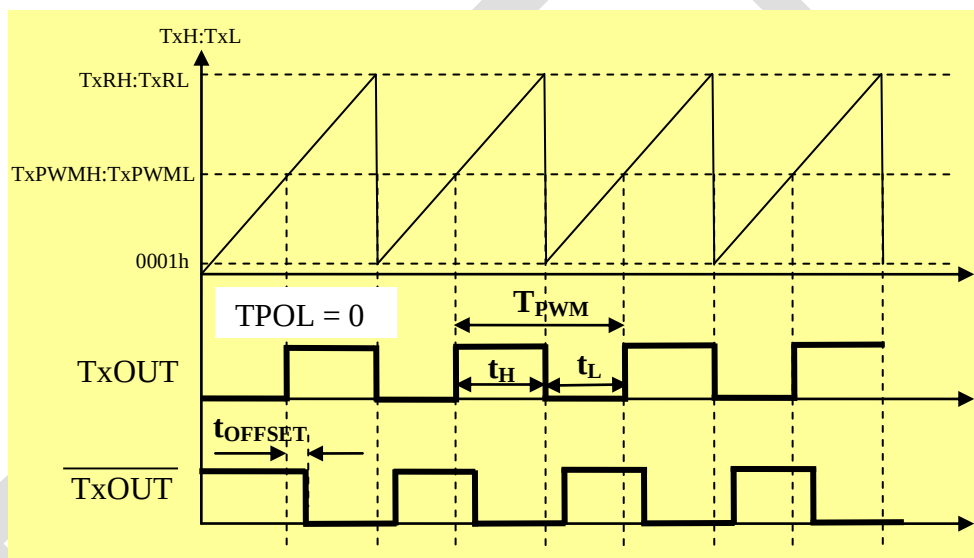
Ако TROL = 1, коефициентът на запълване се определя от следното уравнение:

$$k(\%) = \frac{\text{Широчинна Стойност}}{\text{Максимална Броячна Стойност}} * 100$$

Двуизходен PWM режим

Този режим е като предходния с тази разлика, че таймерът генерира два широчинно-импулсни сигнала, един нормален сигнал на изход TxOUT и втори инвертиран сигнал на изход $\overline{\text{TxOUT}}$. Между двата сигнала може да бъде добавено и конфигурируемо дефазиране t_{OFFSET} (от 0 до 128 такта на системния тактов сигнал).

Фиг.27 илюстрира описания процес при $\text{TPOL} = 0$.



Фиг. 27 Двуизходен PWM режим

Следвайте следните стъпки, когато конфигурирате таймера в двуизходен PWM режим:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в двуизходен PWM режим;

- Конфигурирате входния делител;
 - Конфигурирате началната ниво на таймерния изход TxOUT.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h). Тази настройка има ефект само при първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.
 3. Запишете широчинната стойност в регистрите TxPWMH и TxPWML.
 4. Конфигурирайте дефазирането между двата таймерни изхода. Дефазирането трябва да бъде по-малко от t_H и t_L .
 5. Запишете максималната броячната стойност в регистрите TxRH и TxRL. Тази стойност трябва да е по-голяма от широчинната стойност.
 6. Ако е необходимо, разрешете прекъсването от таймера.
 7. Конфигурирайте съответния извод в режим на таймерен изход.
 8. Разрешете таймера.

Режим на прехващане (capture mode)

В този режим при появата на активен фронт (нарастващ или падащ) на таймерния вход TxIN, текущата броячна стойност в регистрите TxH:TxL се копира (прехваща) в регистрите TxPWMH:TxPWML. Системният тактов сигнал F_{SYS} служи за тактов сигнал на таймера. Бит TPOL в регистър TxCTL1

определя по кой фронт (нарастващ или падащ) на таймерния вход да става прехващането на текущата броячна стойност на таймера. При появата на активен фронт на таймерния вход се генерира прекъсване и таймерът продължава да брои. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прехващащо събитие, а не от препълване на таймера.

Таймерът продължава да брои, докато стигне броячната стойност в регистрите TxRH:TxRL. При достигане на тази стойност, таймерът генерира прекъсване, **броячните регистри се презареждат с 0001h¹** и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването е поради препълване на таймера, а не поради прехващащо събитие.

Забележка¹: В оригиналната документация е описано, че при тази ситуация таймерът продължава да брои, но по-вероятно е пропуснато да се каже, че преди това броячните регистри TxH:TxL се презареждат с 0001h.

Следвайте следните стъпки, когато конфигурирате таймера в режим на прихващане:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в режим на прехващане;
 - Конфигурирате входния делител;
 - Изберете активния фронт на таймерния вход.

2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).
3. Запишете максималната броячна стойност в регистрите TxRH и TxRL.
4. Запишете 0000h в регистрите TxPWMH:TxPWML. Нулирането на тези регистри позволява на софтуера да определи дали прекъсването е генерирано от прехващащо събитие или от препълване на таймера. Ако стойността в TxPWMH:TxPWML е все още нула, значи прекъсването е от препълване.
5. Ако е необходимо, разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прехващащо събитие и при препълване на таймера. Ако е необходимо, конфигурирайте таймера да генерира прекъсване само при прехващане или само при препълване чрез бит TICONFIG в регистър TxCTL1.
6. Конфигурирайте съответния извод в режим на таймерен вход.
7. Разрешете таймера.

Времето от стартирането на таймера до появата прехващащо събитие може да се определи от уравнението:

$$T_{CAPTURE} (s) = \frac{(\text{Прехваната Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{СистемнаТактоваЧестота (Hz)}}$$

Режим на прехващане с рестартиране

В този режим при появата на активен фронт (нарастващ или падащ) на таймерния вход TxIN, текущата броячна стойност в регистрите TxH:TxL се копира (прехваща) в регистрите

TxPWMH:TxPWML. Системният тактов сигнал F_{SYS} служи за тактов сигнал на таймера. Бит TPOL в регистър TxCTL1 определя по кой фронт (нарастващ или падащ) на таймерния вход да става прехващането на текущата броячна стойност на таймера.

При появата на активен фронт на таймерния вход, се генерира прекъсване и броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прехващащо събитие.

Ако не възникне прехващащо събитие, таймерът продължава да брои, докато стигне броячната стойност в регистрите TxRH:TxRL. При достигане на тази стойност, таймерът генерира прекъсване и броячните регистри се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването е поради препълване на таймера.

Следвайте следните стъпки, когато конфигурирате таймера в режим на прехващане с рестартиране:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в режим на прехващане с рестартиране;
 - Конфигурирате входния делител;
 - Изберете активния фронт на таймерния вход.
2. Запишете начална стойност на таймера в броячните

регистри TxH и TxL (обикновено 0001h).

3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Запишете 0000h в регистрите TxPWMH:TxPWML. Нулирането на тези регистри позволява на софтуера да определи дали прекъсването е генерирано от прехващащо събитие или от препълване на таймера. Ако стойността в TxPWMH:TxPWML е все още нула, значи прекъсването е от препълване.
5. Ако е необходимо, разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прехващащо събитие и при препълване на таймера. Ако е необходимо, конфигурирайте таймера да генерира прекъсване само при прехващане или само при препълване чрез бит TICONFIG в регистър TxCTL1.
6. Конфигурирайте съответния извод в режим на таймерен вход.
7. Разрешете таймера.

Времето от стартирането на таймера до появата прехващащо събитие може да се определи от уравнението:

$$T_{CAPTURE}(s) = \frac{(\text{Прехваната Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{Системна Тактова Честота (Hz)}}$$

Режим на сравнение (compare mode)

В този режим таймерът брои до 16-битовата стойност съхранена в регистрите TxRH:TxRL (извършва се сравнение на стойността в броячните регистри TxH:TxL с броячната стойност в регистрите TxRH:TxRL). Системният тактов

сигнал F_{SYS} служи за тактов сигнал на таймера. При достигане до тази стойност таймерът генерира прекъсване и броенето продължава (броячните регистри $TxH:TxD$ не се презареждат с 0001h). Допълнително, нивото на таймерния изход $TxDOUT$ (ако е разрешен) се превключва (от ниско във високо или обратно). Бит $TROL$ в регистър $TxDCTL1$ определя началното ниво на таймерния изход.

Ако таймерът достигне стойност $FFFFh$, броячните регистри се презареждат с 0000h и броенето продължава.

Следвайте следните стъпки, когато конфигурирате таймера в режим на сравнение:

1. Запишете управляващите регистри $TxDCTL0$ и $TxDCTL1$ за да:
 - Забраните таймера;
 - Конфигурирате таймера в режим на сравнение;
 - Конфигурирате входния делител;
 - Изберете начално ниво на таймерния изход.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxD .
3. Запишете сравняващата броячната стойност в регистрите $TxRH$ и $TxRL$.
4. Разрешете прекъсването от таймера.
5. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.

6. Разрешете таймера.

Времето от стартирането на таймера до възникване на сравнението може да се определи от уравнението:

$$T_{\text{COMPARE}} (s) = \frac{(\text{Сравняваща Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{СистемнаТактоваЧестота (Hz)}}$$

Режим на контролирано броене (gated mode)

В този режим активен сигнал (ниско или високо ниво) на таймерния вход TxIN служи като разрешаващ сигнал за таймера, т.е. таймерът брои само когато на таймерния вход има валиден разрешаващ сигнал. Бит TPOL в регистър TxCTL1 определя активното ниво на разрешаващия сигнал. При деактивиране на разрешаващия сигнал се генерира прекъсване. Системният тактов сигнал F_{SYS} служи за тактов сигнал на таймера

Таймерът брои до 16-битовата броячна стойност в регистрите TxRH:TxRL. При достигане на тази стойност таймерът генерира прекъсване, броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява (ако разрешаващият сигнал е все още активен). Допълнително, ако таймерният изход е разрешен, нивото на извода се превключва (от високо в ниско или обратно) при презареждане на таймера.

За да определите причината за прекъсването (дали идва от препълване на таймера или от деактивиране на разрешаващия сигнал), прочетете нивото на таймерния вход TxIN и го сравнете със стойността на бит TPOL.

Следвайте следните стъпки, когато конфигурирате таймера в режим на контролирано броене:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в контролиран режим на броене;
 - Конфигурирате входния делител.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL. Тази настройка има ефект само до първото препълване на таймера. След първото препълване на таймера, броенето винаги започва от 0001h.
3. Запишете максималната броячната стойност в регистрите TxRH и TxRL.
4. Разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при деактивиране на разрешаващия сигнал и при препълване на таймера. Вие може да конфигурирате прекъсването да се генерира само при деактивиране на разрешаващия сигнал или само при препълване на таймера, чрез бит TICONFIG в регистър TxCTL1.
5. Конфигурирайте съответния извод като таймерен вход.
6. Ако е необходимо, конфигурирайте съответния извод в режим на таймерен изход.
7. Разрешете таймера.
8. Активирайте разрешаващия сигнал, за да стартирате броенето.

Режим на прехващане/сравнение

В този режим таймерът започва да брои по първия активен фронт (нарастващ или падащ) на таймерния вход TxIN. Бит TROL в регистър TxCTL1 определя активния фронт. Системният тактов сигнал F_{SYS} служи за тактов сигнал на таймера.

При всеки следващ активен фронт на входа текущата броячната стойност на таймера се копира (прехваща) в регистрите TxPWMH:TxPWML, генерира се прекъсване, броячните регистри TxH:TxL се презареждат в 0001h и броенето продължава. Флаг INTCAP в регистър TxCTL1 се вдига, за да укаже, че прекъсването се дължи на прехващащо събитие.

Ако не възникне прихващащо събитие (няма активен фронт на таймерния вход), таймерът брои до 16-битовата сравняваща стойност в регистрите TxRH:TxRL. При достигане на тази стойност таймерът генерира прекъсване, броячните регистри TxH:TxL се презареждат с 0001h и броенето се възобновява. Флаг INTCAP в регистър TxCTL1 се нулира, за да укаже, че прекъсването не се дължи на прехващащо събитие.

Следвайте следните стъпки, когато конфигурирате таймера в режим на прехващане/сравнение:

1. Запишете управляващите регистри TxCTL0 и TxCTL1 за да:
 - Забраните таймера;
 - Конфигурирате таймера в режим на прехващане/сравнение;

- Конфигурирате входния делител;
 - Изберете активен фронт на таймерния вход.
2. Запишете начална стойност на таймера в броячните регистри TxH и TxL (обикновено 0001h).
 3. Запишете сравняващата броячната стойност в регистрите TxRH и TxRL.
 4. Разрешете прекъсването от таймера. По подразбиране прекъсване се генерира при прехващащо събитие и препълване на таймера. Може да конфигурирате прекъсването да се генерира само при прехващащо събитие или само при препълване на таймера, чрез бит TICONFIG в регистър TxCTL1.
 5. Конфигурирайте съответния извод в режим на таймерен вход.
 6. Разрешете таймера.
 7. Броенето започва по първия активен фронт на таймерния вход без да се генерира прекъсване.

Времето от стартирането на таймера до възникване на прехващане може да се определи от уравнението:

$$T_{\text{COMPARE}}(s) = \frac{(\text{Прехваната Броячна Стойност} - \text{Начална Броячна Стойност}) * \text{Делител}}{\text{Системна Тактова Честота (Hz)}}$$

12.4 Четене на текущата броячна стойност на таймера

Текущата броячна стойност може да бъде прочетена, докато таймерът брои. Когато таймерът е разрешен, четенето на регистър TxH води и до копиране на TxL във временен регистър. Последващо четене на TxL, връща стойността, съхранена във временния регистър.

Тази операция позволява точно изчитане на 16-битовата стойност на таймера, докато е разрешен. Когато таймерът е забранен, четене на TxL връща действителната стойност в този регистър.

12.5 Описание на управляващите регистри

Регистри TxH (Timer High Byte) и TxL (Timer Low Byte)

Тези регистри съдържат текущата броячна стойност на таймера. Когато таймерът е разрешен, четене на TxH води и до копиране на TxL във временен регистър. Последващо четене на TxL връща стойността, съхранена във временния регистър. Когато таймерът е забранен, четене на TxL връща действителната стойност в този регистър.

Бит	7	6	5	4	3	2	1	0
Име	ТН							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F00h, F08h							
Бит	Описание							
7-0	Старши байт на 16-битовия брояч							

Табл. 55 TxH

Бит	7	6	5	4	3	2	1	0
-----	---	---	---	---	---	---	---	---

Име	TL							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F01h, F09h							
Бит	Описание							
7-0	Младши байт на 16-битовия брояч							

Табл. 56 TxL

Регистри TxRH (Timer Reload High Byte) и TxRL (Timer Reload Low Byte)

Тези регистри съдържат максималната 16-битова броячна стойност до която брой таймера. Запис в TxRH води до запис във временен регистър. Запис в TxRL води и до копиране на временния регистър в TxRH. Тази операция позволява едновременно обновяване на 16-битовата броячната стойност в тези регистри. В режим на сравнение тези регистри съдържат 16-битовата сравняваща стойност.

Бит	7	6	5	4	3	2	1	0
Име	TRH							
Ресет	1	1	1	1	1	1	1	1
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F02h, F0Ah							
Бит	Описание							
7-0	Старши байт на 16-битовата броячна/сравняваща стойност							

Табл. 57 TxRH

Бит	7	6	5	4	3	2	1	0
Име	TRL							
Ресет	1	1	1	1	1	1	1	1

Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F03h, F0Bh							
Бит	Описание							
7-0	Младши байт на 16-битовата броячна/сравняваща стойност							

Табл. 58 TxRL

Регистри TxPWMH (PWM High Byte) и TxPWML (PWM Low Byte)

Тези регистри съдържат 16-битовата стойност, определяща продължителността на високото ниво на импулсите в едноизходен и двуизходен PWM режим на работа. В режим на прихващане и прихващане/сравнение тези регистри съдържат прихванатата броячна стойност при възникване на прихващащо събитие.

Бит	7	6	5	4	3	2	1	0
Име	TPWMH							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F04h, F0Ch							
Бит	Описание							
7-0	Старши байт на 16-битовата широчинна стойност в PWM режими или старши байт на прихванатата броячна стойност в режими на прихващане.							

Табл. 59 TxPWMH

Бит	7	6	5	4	3	2	1	0
Име	TPWML							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З

Адрес	F05h, F0Dh
Бит	Описание
7-0	Младши байт на 16-битовата широчинна стойност в PWM режими или младши байт на прихванатата броячна стойност в режими на прихващане.

Табл. 60 TxPWML

Регистър TxCTL0 (Timer x Control 0)

Бит	7	6	5	4	3	2	1	0
Име	TMODEHI	TICONFIG			PWMD			INCAP
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F06h, F0Eh							
Бит	Описание							
7	Работен режим на таймера Този бит заедно с битове TMODE[2:0] в регистър TxCTL1 определят работния режим на таймера.							
	TMODEHI:TMODE[2:0]			Режим				
	0000			Еднократен				
	0001			Непрекъснат				
	0010			Броячен				
	0011			Едноизходен PWM				
	0100			Прихващане				
	0101			Сравнение				
	0110			Контролирано броене				
	0111			Прихващане/Сравнение				
	1000			Двуизходен PWM режим				
	1001			Прихващане с рестартиране				
	1010			Компараторен				
6-5	Конфигуриране на прекъсването от таймера 0x – Прекъсване при презареждане (препълване), сравнение и прихващане							

	10 – Прекъсване при прихващане/деактивиране 11 – Прекъсване при презареждане (препълване)/сравнение
4	Резервиран
3-1	Конфигуриране на t_{OFFSET} 000 – 0 цикъла 001 – 2 цикъла 010 – 4 цикъла 011 – 8 цикъла 100 – 16 цикъла 101 – 32 цикъла 110 – 64 цикъла 111 – 128 цикъла
0	Индикатор на източник на прекъсване 0 – Прекъсването не се дължи на прихващащо събитие 1 – Прекъсването се дължи на прихващащо събитие

Табл. 61 TxCTL0

Регистър TxCTL1 (Timer x Control 1)

Бит	7	6	5	4	3	2	1	0
Име	TEN	TPOL	PRES			TMODE		
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F07h, F0Fh							
Бит	Описание							
7	Разрешаване на таймера 0 – Таймерът е забранен 1 – Таймерът е разрешен							
6	Полярност на входа/изхода <u>Еднократен режим</u> Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при презареждане (препълване) на таймера.							

Непрекъснат режим

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.

Броячен режим

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.

0 – Броене по нарастващ фронт

1 – Броене по падащ фронт

Компараторен броячен режим

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен, таймерният изход се превключва при всяко презареждане (препълване) на таймера.

Когато таймерният изход TxOUT е разрешен, той винаги извежда стойността съхранена в бит TPOL, независимо дали таймерът е разрешен или забранен. Промяна на TPOL, докато таймерът е разрешен и брои, не променя веднага стойността на TxOUT.

Еднозиходен PWM режим

0 – Извод TxOUT е в ниско ниво, когато таймерът е забранен. TxOUT се превключва във високо ниво, когато таймерът е разрешен и $TxH:TxL == TxPWMH:TxPWML$. Когато $TxH:TxL == TxRH:TxRL$, TxOUT отново се превключва в ниско ниво.

1 – Извод TxOUT е във високо ниво, когато таймерът е забранен. TxOUT се превключва в ниско ниво, когато таймерът е разрешен и $TxH:TxL == TxPWMH:TxPWML$. Когато $TxH:TxL == TxPH:TxPL$, TxOUT отново се превключва във високо ниво.

Двувизходен PWM режим

- 0 – Извод TxOUT е в ниско ниво и извод $\overline{\text{TxOUT}}$ е във високо ниво, когато таймерът е забранен. TxOUT се превключва във високо ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, TxOUT отново се превключва в ниско ниво.
- 1 – Извод TxOUT е във високо ниво и извод $\overline{\text{TxOUT}}$ е в ниско ниво, когато таймерът е забранен. TxOUT се превключва в ниско ниво, когато таймерът е разрешен и TxH:TxL == TxPWMH:TxPWML. Когато TxH:TxL == TxRH:TxRL, TxOUT отново се превключва във високо ниво.

Режим на прихващане

- 0 – Прихващане по нарастващ входен фронт
- 1 – Прихващане по падащ входен фронт

Режим на прихващане с рестартиране

- 0 – Прихващане по нарастващ входен фронт
- 1 – Прихващане по падащ входен фронт

Режим на сравнение

Когато таймерът е забранен, таймерният изход извежда стойността на този бит. Когато таймерът е разрешен и TxH:TxL == TxRH:TxRL, таймерният изход се превключва.

Режим на контролирано броене

- 0 – Таймерът брои, когато нивото на входа TxIN е високо и прекъсване се генерира по падащ фронт на входния сигнал.
- 1 – Таймерът брои, когато нивото на входа TxIN е ниско и прекъсване се генерира по нарастващ фронт на входния сигнал.

Режим на прихващане/сравнение

	0 – Броенето се стартира по първия нарастващ фронт на входа TxIN. При всеки следващ нарастващ фронт броячната стойност се прихваща. 1 – Броенето се стартира по първия падащ фронт на входа TxIN. При всеки следващ падащ фронт броячната стойност се прихваща.																								
5-3	Коефициент на деление на F_{SYS} 000 – 1 001 – 2 010 – 4 011 – 8 100 – 16 101 – 32 110 – 64 111 – 128																								
2-0	Работен режим на таймера Тези битове заедно с бит TMODEHI в регистър TxCTL0 определят работния режим на таймера. <table border="1"> <thead> <tr> <th>TMODEHI:TMODE[2:0]</th><th>Режим</th></tr> </thead> <tbody> <tr> <td>0000</td><td>Еднократен</td></tr> <tr> <td>0001</td><td>Непрекъснат</td></tr> <tr> <td>0010</td><td>Броячен</td></tr> <tr> <td>0011</td><td>Едноизходен PWM</td></tr> <tr> <td>0100</td><td>Прихващане</td></tr> <tr> <td>0101</td><td>Сравнение</td></tr> <tr> <td>0110</td><td>Контролирано броене</td></tr> <tr> <td>0111</td><td>Прихващане/Сравнение</td></tr> <tr> <td>1000</td><td>Двуизходен PWM режим</td></tr> <tr> <td>1001</td><td>Прихващане с рестартиране</td></tr> <tr> <td>1010</td><td>Компараторен</td></tr> </tbody> </table>	TMODEHI:TMODE[2:0]	Режим	0000	Еднократен	0001	Непрекъснат	0010	Броячен	0011	Едноизходен PWM	0100	Прихващане	0101	Сравнение	0110	Контролирано броене	0111	Прихващане/Сравнение	1000	Двуизходен PWM режим	1001	Прихващане с рестартиране	1010	Компараторен
TMODEHI:TMODE[2:0]	Режим																								
0000	Еднократен																								
0001	Непрекъснат																								
0010	Броячен																								
0011	Едноизходен PWM																								
0100	Прихващане																								
0101	Сравнение																								
0110	Контролирано броене																								
0111	Прихващане/Сравнение																								
1000	Двуизходен PWM режим																								
1001	Прихващане с рестартиране																								
1010	Компараторен																								

Табл. 62 TxCTL1

13 Компаратор

13.1 Общи сведения

Компараторът в Z8Encore!F083A серията микроконтролери има следните характеристики:

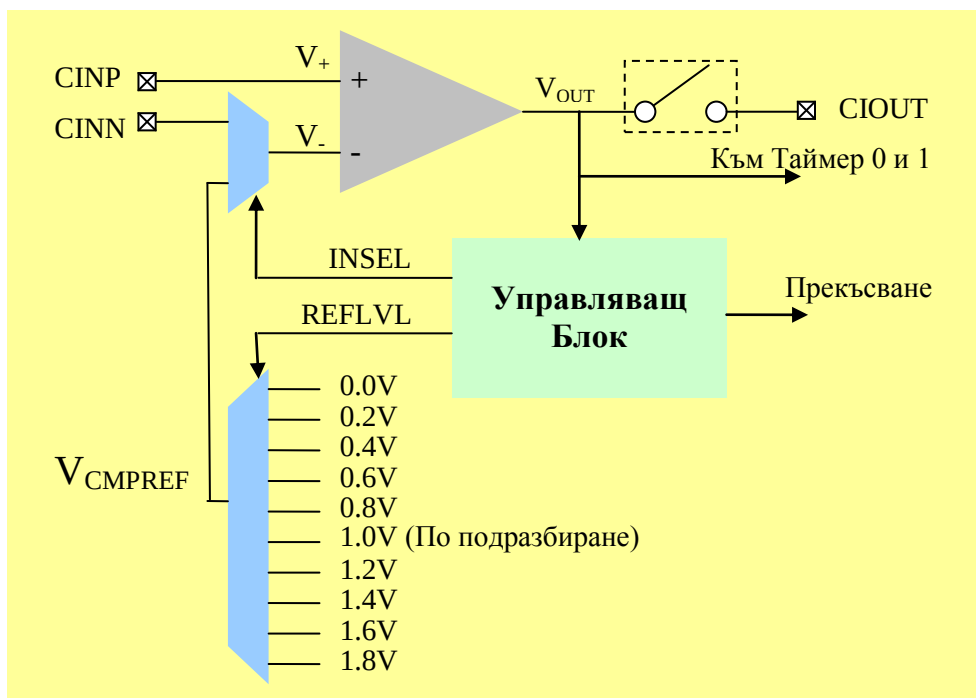
- Положителен вход свързан към извод CINP;
- Отрицателен вход CINN свързан към извод CINN или към вътрешно конфигурируемо опорно напрежение V_{CMPREF} ;
- Изходът на компаратора може да се изведе на извод COUT;
- Време за отговор 100ns;
- Входен хистерезис 8mV.

13.2 Устройство и принцип на работа

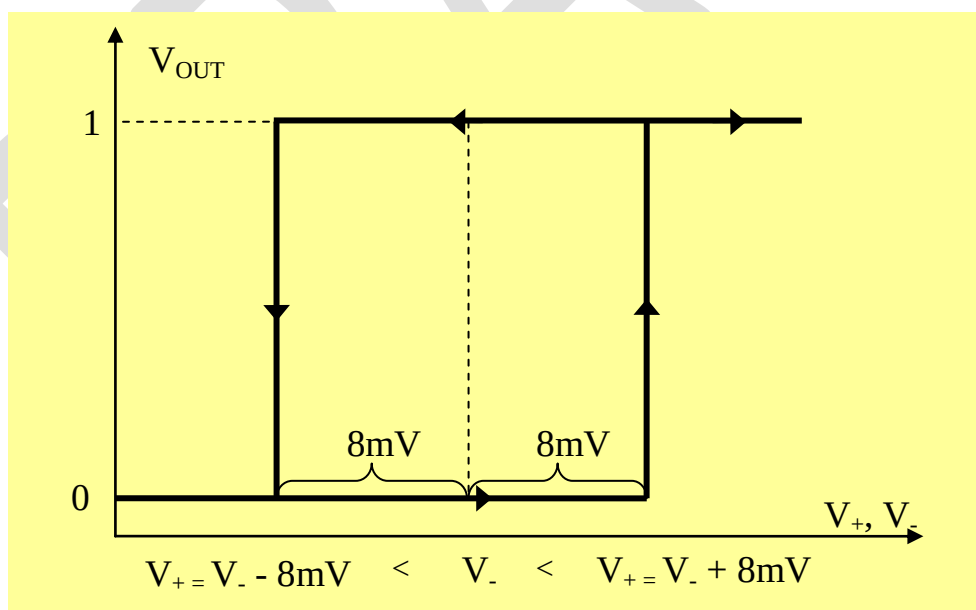
Фиг.28 показва блоковата схема на компаратора, а Фиг.29 описва принципа на работа.

Когато напрежението V_+ е по-голямо от V_- , V_{OUT} се превключва във високо ниво и се генерира прекъсване. Когато напрежението V_+ е по-малко от V_- , V_{OUT} се превключва в ниско ниво и се генерира прекъсване.

За намаляване на консумацията, захранването на компаратора може да бъде изключено. За повече детайли вижте [Регистър PWRCTL0](#).



Фиг. 28 Блокова схема на компаратора



Фиг. 29 V_{OUT} като функция на V_+ и V_-



В резултат на времето за отговор на компаратора, Zilog не препоръчва разрешаване на компаратор без първо забраняване на прекъсванията и изчакване на изхода да се установи. Това закъснение предпазва от фалшиви прекъсвания след разрешаване на компаратора.

Следващият пример показва как безопасно да разрешите компаратора.

```
DI(); // Забраняване на прекъсванията
CMP0 = нова-конфигурация; // Зареждане на нова конфигурация
asm ("NOP");
asm ("NOP"); // Изчакване на установяване на изхода
IRQ0 = 0x00; //Изчистване на всички фалшиви чакащи
           // прекъсвания
EI(); // Разрешаване на прекъсванията
```

13.3 Описание на управляващите регистри

Регистър CMP0 (Comparator Control Register)

Бит	7	6	5	4	3	2	1	0
Име		INNSEL	REFLVL					
Ресет	0	0	0	1	0	1	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F90h							
Бит	Описание							
7	Този бит е резервиран и трябва да се поддържа в 0.							
6	Избор на V. 0 – CINN е източник на опорно напрежение 1 – V _{REF} е източник на опорно напрежение							
5-2	Избор на вътрешно опорно напрежение V_{CMPREF}							

	0000 – 0.0V 0001 – 0.2V 0010 – 0.4V 0011 – 0.6V 0100 – 0.8V 0101 – 1.0V (По подразбиране) 0110 – 1.2V 0111 – 1.4V 1000 – 1.6V 1001 – 1.8V 1010 – 1111 (Резервирани)
1-0	Тези битове са резервирани и трябва да се поддържат в 0.

Табл. 63 CMP0

14 Аналого-цифров преобразувател

14.1 Общи сведения

Аналого-цифровият преобразувател в Z8Encore!F083A серията микроконтролери има следните характеристики:

- 8 аналогови входа;
- Време за преобразуване под $2.8\mu s$ (тактовият сигнал за АЦП трябва да е под 10MHz);
- Програмируемо време за семплиране на входното напрежение;
- Генериране на прекъсване при завършване на преобразуването;
- Генератор на вътрешно опорно напрежение;
- Извод V_{REF} за подаване на външно опорно напрежение.

14.2 Устройство и принцип на работа

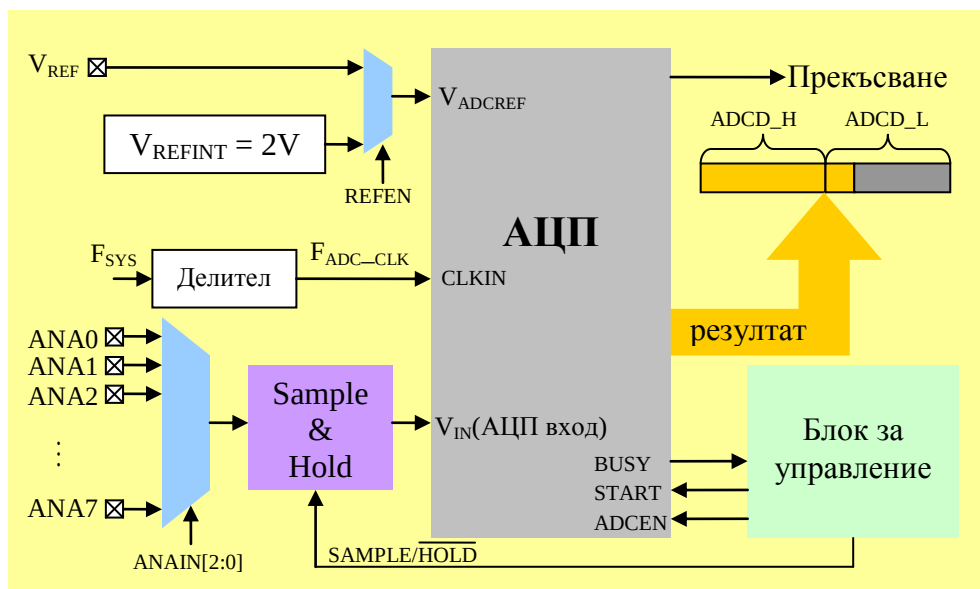
Фиг.30 показва блоковата схема на АЦП. АЦП преобразува избрания аналогов вход $ANAx$ ($x = 0 - 7$) в 10-битов цифров код. Резултатът от преобразуването може да се изчисли от следното уравнение:

$$АЦПрезултат = \frac{V_{IN}}{V_{ADCREF}} \times 1024$$

където,

V_{IN} – Напрежение на измервания аналогов вход

V_{ADCREF} – Опорно напрежение на АЦП



Фиг. 30 Блокова схема на АЦП

Опорното напрежение V_{ADCREF} може да се получи от вътрешен източник или от извод V_{REF} :

$$V_{ADCREF} = \begin{cases} 2V \text{ от вътрешния източник } V_{REFINT} \\ V_{DD} \text{ от извод } V_{REF} \end{cases}$$

Максималното входно напрежение, което се преобразува от АЦП е:

$$V_{INTMAX} = \begin{cases} V_{REFINT}, \text{ ако се използва вътрешното опорно напрежение} \\ 0.9V_{DD}, \text{ ако се използва външно опорно напрежение (от извод } V_{REF}) \end{cases}$$

Преобразуването се стартира с бит START в регистър ADCCTL0. Този бит служи и за индикатор дали текущото преобразуване е в прогрес или е завършило (прочитане на 1 в този бит указва, че в момента се извършва преобразуване).

Стартиране на ново преобразуване води до прекратяване на текущото и започване на ново.

При завършване на преобразуването АЦП генерира прекъсване. Чакащи заявки за прекъсване, когато АЦП е забранено, не се нулират автоматично.

Времето за преобразуване се определя от следното уравнение:

$$T_{\text{CONV}} = T_S + T_H + 13 \cdot T_{\text{ADC_CLK}}$$

където

T_S – Време за семплиране (**sample time**). Това време може да се конфигурира чрез регистър ADCST и трябва да е минимум $1\mu\text{s}$.

T_H – Време за задържане (**hold time**). Това време може да се конфигурира чрез регистър ADCSST и трябва да е минимум $0.5\mu\text{s}$.

$T_{\text{ADC_CLK}}$ – Период на тактовия сигнал за АЦП. Максималната честота на тактовия сигнал $F_{\text{ADC_CLK}}$ е 10MHz .

14.3 Описание на управляващите регистри

Регистър ADCCTL0 (ADC Control 0)

Бит	7	6	5	4	3	2	1	0
Име	START		REFEN	ADCEN		ANAIN		
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	F70h							

Бит	Описание
7	START/BUSY 0 – Прочитане на 0 означава, че АЦП е свободен. Запис на 0 няма ефект. 1 – Прочитане на 1 означава, че АЦП извършва преобразуване в момента. Запис на 1 стартира преобразуване.
6	Този бит е резервиран и трябва да се поддържа в 0.
5	Избор на опорно напрежение 0 – Извод V_{REF} е източник на опорно напрежение. 1 – V_{REFIN} е източник на опорно напрежение. Стойността на V_{REFIN} се появява на извод V_{REF} .
4	Разрешаване на АЦП 0 – АЦП е забранен. 1 – АЦП е разрешен.
3	Този бит е резервиран и трябва да се поддържа в 0.
2-0	Избор на аналогов канал 000 – ANA0 е избран да се преобразува. 001 – ANA1 е избран да се преобразува. 010 – ANA2 е избран да се преобразува. 011 – ANA3 е избран да се преобразува. 100 – ANA4 е избран да се преобразува. 101 – ANA5 е избран да се преобразува. 110 – ANA6 е избран да се преобразува. 111 – ANA7 е избран да се преобразува.

Табл. 64 ADCCTL0

Регистър ADCD_H (ADC Data High Byte)

Бит	7	6	5	4	3	2	1	0
Име	ADCDH							
Ресет	x							
Ч/З	ч	ч	ч	ч	ч	ч	ч	ч
Адрес	F72h							
Бит	Описание							

7-0	00h-FFh: Старши байт на резултата от преобразуването.
-----	---

Табл. 65 ADCD_H

Регистър ADCD_L (ADC Data Low Bits)

Бит	7	6	5	4	3	2	1	0
Име	ADCDL							
Ресет	x		x					
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	F73h							
Бит	Описание							
7-6	00h-03h: Младши битове на резултата от преобразуването.							
5-0	Тези битове са резервирани и трябва да се поддържат в 0.							

Табл. 66 ADCD_L

Регистър ADCSST (ADC Settling Time)

Бит	7	6	5	4	3	2	1	0
Име					SST			
Ресет	0				1	1	1	1
Ч/З	Ч				Ч/З			
Адрес	F74h							
Бит	Описание							
7-4	Тези битове са резервирани и трябва да се поддържат в 0.							
3-0	Избор на време T_H Измерва се в ситемни тактови периода. Трябва да бъде минимум 0.5μs.							

Табл. 67 ADCDSST

Регистър ADCST (ADC Sample Time)

Бит	7	6	5	4	3	2	1	0
Име	ST							
Ресет	0	1	1	1	1	1	1	1
Ч/З	Ч/З							
Адрес	F75h							
Бит	Описание							
7-6	Тези битове са резервирани и трябва да се поддържат в 0.							
5-0	Избор на време T_s Измерва се в системни тактови периода. Трябва да бъде минимум 1 μ s.							

Табл. 68 ADCST

Регистър ADCCP (ADC Clock Prescale)

Бит	7	6	5	4	3	2	1	0
Име						DIV8	DIV4	DIV2
Ресет	0					0	0	0
Ч/З	Ч/З							
Адрес	F76h							
Бит	Описание							
7-3	Тези битове са резервирани и трябва да се поддържат в 0.							
2	$F_{SYS}/8$ (Най-висок приоритет) 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 8.							
1	$F_{SYS}/4$ 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 4.							
0	$F_{SYS}/2$ (Най-нисък приоритет) 0 – Системният такт не се разделя. 1 – Системният такт се разделя на 2.							

Табл. 69 ADCSP

15 Флаш-памет

15.1 Общи сведения

Микроконтролерите от серията Z8Encore!F083A имат 4kB (4096 байта) или 8kB (8192 байта) флаш-памет за четене/запис/изтриване. Паметта може да се програмира и изтрива, докато микроконтролерът е на платката, с помощта на потребителски код или с дебъгер/програматор.

Флаш-паметта е разделена на страници по 512 байта. Флаш-страницата е минималният блок памет, който може да бъде изтрит. Всяка страница е разделена на 8 реда всеки с по 64 байта.

За защита на програмата/данните, флаш-паметта е разделена също и на сектори. В **Z8F043A** секторът заема една флаш-страницата, а в **Z8F083A** секторът заема две флаш-страници.

Първите два байта на флаш-паметта се заемат от флаш-конфигурационни битове (вижте [16 Флаш-конфигурационни бита](#)).

Табл.70 описва конфигурацията на Програмната Памет за всяко устройство от серията Z8Encore!F083A.

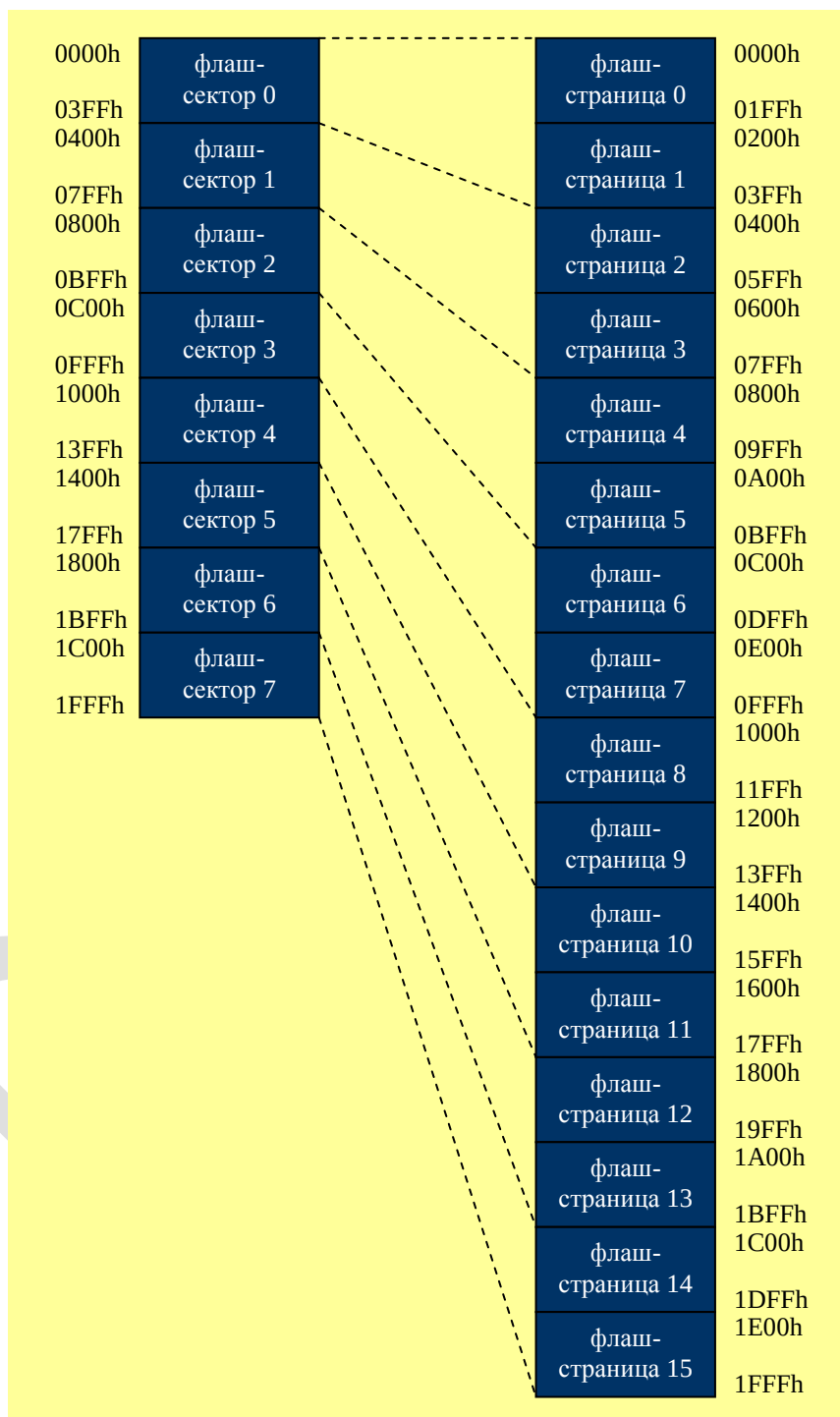
	Флаш-памет kB (байтове)	Флаш- страници	адреси	Флаш-сектор (байтове)
Z8F083A	8 (8196)	16	0000h-1FFFh	1024
Z8F043A	4 (4096)	8	0000h-0FFFh	512

Табл. 70 Конфигурация на флаш-паметта на серията Z8Encore!F083A

Следващите фигури показват разделянето на флаш-паметта на сектори и страници.

0000h	флаш-сектор 0		флаш-страница 0	0000h
01FFh				01FFh
0200h	флаш-сектор 1		флаш-страница 1	0200h
03FFh				03FFh
0400h	флаш-сектор 2		флаш-страница 2	0400h
05FFh				05FFh
0600h	флаш-сектор 3		флаш-страница 3	0600h
07FFh				07FFh
0800h	флаш-сектор 4		флаш-страница 4	0800h
09FFh				09FFh
0A00h	флаш-сектор 5		флаш-страница 5	0A00h
0BFFh				0BFFh
0C00h	флаш-сектор 6		флаш-страница 6	0C00h
0DFFh				0DFFh
0E00h	флаш-сектор 7		флаш-страница 7	0E00h
0FFFh				0FFFh

Фиг. 31 Флаш-памет на Z8F043A



Фиг. 32 Флаш-памет на Z8F083A

15.2 Флаш-информационна област

Флаш-информационната област е физически разделена от Програмната Памет и се разполага на адреси FE00h ÷ FE7Fh. В тази област се съхраняват фабрични настройки на VBO-модула и вътрешния прецизен генератор, и фабрични калибрационни данни на АЦП-модула (Табл.71).

Адрес	Описание
FE00h – FE3Fh	Zilog-конфигурационни битове
FE40h – FE53h	Номер на устройството. 20-символен ASCII буквено-цифров код
FE54h – FE5Fh	-
FE60h – FE7Fh	Zilog-калибрационни данни

Табл. 71 Флаш-информационна област

Флаш-информационната област е достъпна когато бит 7 на регистър FPS е 1. Когато достъпът е разрешен, флаш-информационната област се „наслажда“ върху Програмната Памет с адреси FE00h ÷ FE7Fh и четене от тези адреси води до четене на флаш-информационната област вместо от Програмната Памет. Флаш-информационната област е достъпна само за четене.

15.3 Флаш-контролер

Флаш-контролерът програмира и изтрива флаш-паметта. Той съдържа няколко защитни механизми срещу случайно програмиране или изтриване. Преди извършване на операции програмиране или изтриване, потребителят трябва да конфигурира регистрите FFREQH и FFREQL (**Flash Frequence High/Low**). Тези регистри формират 16-битова честотна стойност FFREQ, която задава времевите характеристики на операциите програмиране и изтриване.

$$FFREQ[15:0] = \frac{\text{СистемнаТактоваЧестота (Hz)}}{1000}$$



Изтриването и програмирането на флаш-паметта изисква системният тактов сигнал да бъде в диапазона 10kHz ÷ 20MHz.

Защита на кода от външен достъп

Потребителският код може да бъде защитен от прочитане с дебъгер/програматор чрез нулиране на флаш-конфигурационния бит FRP (вижте [16 Флаш-конфигурационни бито](#)).

Защита на кода от случайно програмиране и изтриване

Z8Encore!F083A микроконтролерите имат няколко нива на защита от случайно програмиране и изтриване на флаш-паметта.

- Защита чрез флаш-конфигурационния бит FWP.

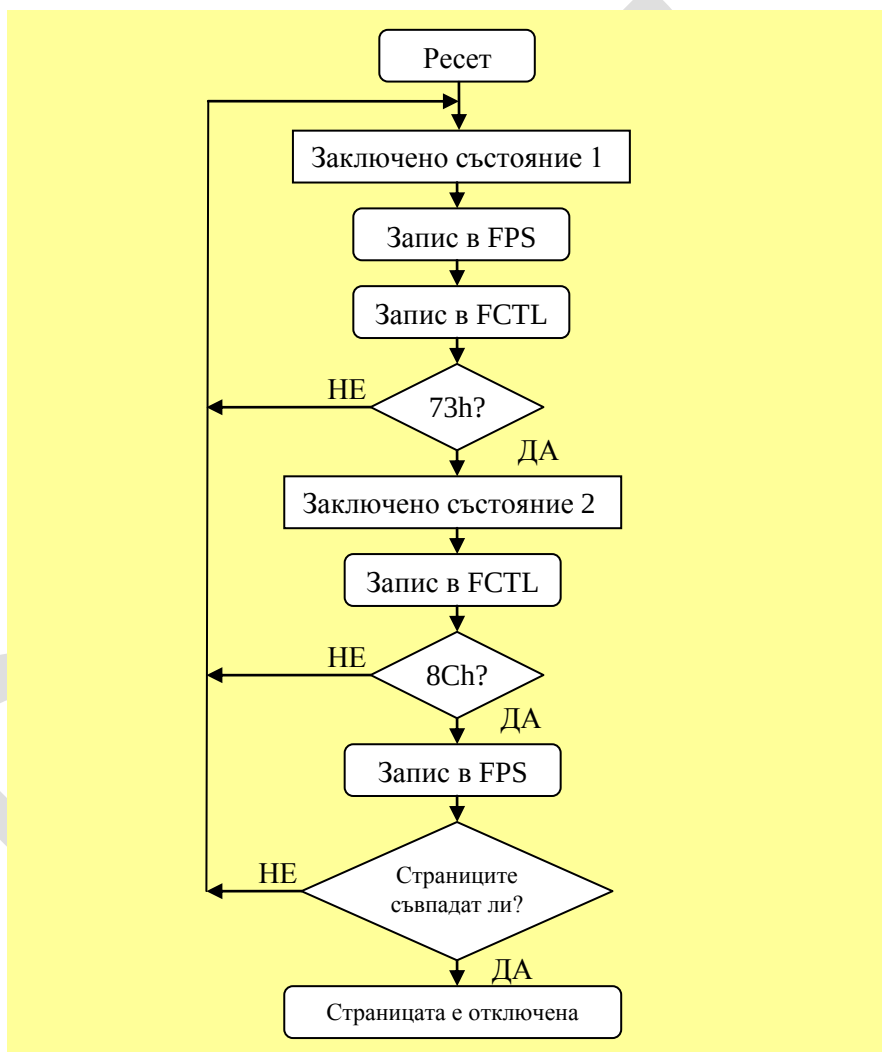
FWP	Описание
0	Програмирането и изтриването е забранено за цялата флаш-памет. В потребителския код програмирането, изтриването на страница и на цялата памет също е забранено. Цялото изтриване на паметта е възможно чрез дебъгера.
1	Програмирането, изтриването на страница и на цялата памет е разрешено.

Табл. 72 Защита на кода с флаш-конфигурационния бит FWP

- Защита чрез специална последователност от действия.

При ресет флаш-контролерът е заключен, за да предпази флаш-паметта от случайно изтриване и програмиране. За да програмирате или изтриете паметта, първо запишете номера

на страницата в регистър FPS. Отключете флаш-контролера чрез последователен запис в регистър FCTL на стойностите 73h и 8Ch. Запишете отново същата страница в регистър FPS. Ако страниците се различават, флаш-контролерът отново се връща в заключено състояние. Ако двете страници съвпадат, избраната страница става активна (фиг. 33)



Фиг. 33 Отключване на флаш-страница

След отключването на дадена страница, потребителят трябва да разреши програмирането или изтриването ѝ. Запис на стойност 95h в регистър FCTL води до изтриване на

страница само ако отключената страница се намира в незащитен сектор. Запис на друга стойност в регистър FCTL заключва флаш-контролера. Цяло изтриване на паметта не е позволено чрез потребителски код, а само чрез дебъгера.

След като отключи дадена страница, потребителят може да запише байт в нея. След като байтът е записан, страницата остава отключена, позволявайки последващи записи на байтове. Последващи записи в регистър FCTL водят до заключване на страницата.

- Сектор-базирана защита

Финалният защитен механизъм е имплементиран на базата на сектори. Флаш-паметта е разделена на 8 сектори (вижте фиг.31 и 32).

Регистър FPROT съдържа битове за разрешаване на програмирането или изтриването на секторите. След като един сектор е защитен (съответният бит в регистър FPROT се запише в 1) той не може да бъде отключен от потребителския код (съответният бит не може да бъде нулиран с помощта на инструкции). Регистър FPROT се нулира след ресет.

Регистрите FPROT и FPS използват един и същ регистров адрес FF9h. Достъпът до FPROT се разрешава чрез запис на 5Eh в регистър FCTL. Запис на стойност различна от 5Eh в FCTL забранява достъпа до регистър FPROT и разрешава достъпа до регистър FPS.

Спазвайте следната процедура, когато конфигурирате регистър FPROT:

1. Запишете 00h във FCTL, за да нулирате флаш-контролера;

2. Запишете 5Eh във FCTL, за да разрешите достъпа до регистър FPROT;
3. Конфигурирайте регистър FPROT;
4. Запишете 00h във FCTL, за да нулирате флаш-контролера.

Регистър FPROT се инициализира в 0 след ресет, поставяйки всеки сектор в незащитено състояние. Когато бит в регистър FPROT се запише в 1, съответният сектор не може да бъде записван или изтриван. След установяване на бит в регистър FPROT, битът не може да бъде нулиран от потребителя.

Изтриване на страница

Флаш-страницата е минималния блок памет, който може да бъде изтрит. Изтрити байтове имат стойност FFh. Само страници, които се намират в незащитени сектори могат да бъдат изтрити. За да изриете страница, първо трябва да я отключите (Фиг.33) и след това да запишете стойност 95h в регистър FCTL. По време на изтриването процесорът спира да изпълнява инструкции, но тактовият сигнал и периферийните модули продължават да работят.

Изтриване на цялата памет

Цялата памет може да бъде изтрита наведнъж само с дебъгер.

Програмиране на байт

Записът на байт във флаш-паметта от потребителския код става с помощта на инструкциите LDC и LDCI. Записът е разрешен само след отключване на съответната страница, в която ще се извършва записа (фиг.33). Изтрит байт има стойност FFh. Операцията програмиране се използва само да

промени даден бит от 1 в 0. Един или повече битове могат да бъдат променени от 0 в 1 само чрез операция изтриване.



Един байт не може да бъде записан повече два пъти преди страницата да бъде изтрита.

15.4 Описание на управляващите регистри

Регистър FCTL (Flash Control)

Флаш-контролерът трябва да бъде отключен преди флаш-паметта да може да бъде програмирана или изтривана. Запис на стойностите 73h и 8Ch последователно в регистър FCTL отключва флаш-контролера. След като флаш-контролерът е отключен, флаш-паметта е разрешена за пълно изтриване¹ или изтриване на страници чрез запис на съответните команди в регистър FCTL.

Забележка¹: Цялото изтриване на флаш-паметта е възможно само чрез дебъгера.

Запис на невалидна стойност или невалидна последователност от стойности във FCTL води до заключване на флаш-контролера. Регистър FCTL и регистър FSTAT използват един и същ регистров адрес. При запис се използва регистър FCTL, а при четене FSTAT.

Бит	7	6	5	4	3	2	1	0
Име	FCMD							
Ресет	0	0	0	0	0	0	0	0
Ч/З	3	3	3	3	3	3	3	3
Адрес	FF8h							

Бит	Описание
7-0	Флаш-команда 73h – Първа команда за отключване 8Ch – Втора команда за отключване 95h – Команда за изтриване на страница 63h – Команда за изтриване на цялата памет 5Eh – Разрешаване на достъпа до регистър FPROT

Табл. 73 FCTL

Регистър FSTAT (Flash Status)

Регистър FSTAT указва текущото състояние на флаш-контролера. Регистър FSTAT и регистър FCTL използват един и същ регистров адрес. При запис се използва регистър FCTL, а при четене FSTAT.

Бит	7	6	5	4	3	2	1	0
Име				FSTAT				
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч	Ч	Ч	Ч	Ч	Ч	Ч	Ч
Адрес	FF8h							
Бит	Описание							
7-6	Тези битове са резервирани и трябва да бъдат 0.							
5-0	Състояние на флаш-контролера 000000 – Флаш-контролерът е заключен 000001 – Първата команда за отключване е приета 000010 – Втората команда за отключване е приета 000011 – Флаш-контролерът е отключен 000100 – Регистър FPROT е избран 001xxx – Операция програмиране е в прогрес 010xxx – Операция изтриване на страница е в прогрес 100xxx – Операция пълно изтриване е в прогрес							

Табл. 74 FSTAT

Регистър FPS (Flash Page Select)

Регистър FPS избира флаш-страница, която ще бъде програмирана или изтривана. Регистър FPS и регистър FPROT използват един и същ регистров адрес. FPROT е достъпен когато флаш-контролерът е заключен и регистър FCTL е записан с 5Eh. Във всички други случаи е достъпен регистър FPS.

Бит	7	6	5	4	3	2	1	0
Име	INFO_EN	PAGE						
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF9h							
Бит	Описание							
7	Разрешаване на флаш-информационната област 0 – Флаш-информационната област е забранена 1 – Флаш-информационната област е разрешена							
6-0	Избор на страница Това 7 битово поле указва флаш-страница за изтриване и отключване. Тези битове се явяват старшите битове [15-9] на адресите във флаш-паметта. За Z8F04xx микроконтролерите старшите 4 бита трябва да бъдат 0.							

Табл. 75 FPS

Регистър FPROT (Flash Sector Protect)

Регистър FPROT съдържа разрешаващи битове за всеки флаш-сектор. Тези битове могат да бъдат установявани с инструкции, но не могат да бъдат нулирани с инструкции.

Регистър FPROT и регистър FPS използват един и същ регистров адрес. FPROT е достъпен когато флаш-контролерът е заключен и регистър FCTL е записан с 5Eh.

Във всички други случаи е достъпен регистър FPS.

Бит	7	6	5	4	3	2	1	0
Име	SPROT7	SPROT6	SPROT5	SPROT4	SPROT3	SPROT2	SPROT1	SPROT0
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FF9h							
Бит	Описание							
7-0	Защита на сектор 0 – Секторът е незащитен 1 – Секторът е защитен							

Табл. 76 FPROT

Регистър FFREQH (Flash Frequency High)

Бит	7	6	5	4	3	2	1	0
Име	FFREQH							
Ресет	0	0	0	0	0	0	0	0
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FFAh							
Бит	Описание							
7-0	Старши байт на 16 битовата флаш-честотна стойност FFREQ							

Табл. 77 FFREQH

Регистър FFREQL (Flash Frequency Low)

Бит	7	6	5	4	3	2	1	0
Име	FFREQL							
Ресет	0	0	0	0	0	0	0	0

Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	FFBh							
Бит	Описание							
7-0	Младши байт на 16 битовата флаш-честотна стойност FFREQ							

Табл. 78 FFREQ

16 Флаш-конфигурационни битове

16.1 Общи сведения

Флаш-конфигурационните битове позволяват на потребителя да управлява следните функции на микроконтролера:

- Генериране на прекъсване или ресет при препълване на WDT-таймера;
- Разрешаване на WDT-таймера след ресет;
- Защиаване на потребителския код от прочитане;
- Защиаване на части или на целия потребителски код от случайно програмиране или изтриване;
- Изключване на VBO схемата в STOP-режим;
- Конфигуриране на кварцовия генератор:
 - кварцови резонатори $32\text{kHz} \div 1\text{MHz}$;
 - кварцови резонатори $0.5\text{MHz} \div 5\text{MHz}$;
 - кварцови резонатори $5\text{MHz} \div 20\text{MHz}$;
 - външна RC верига ($< 4\text{MHz}$).
- Настройка на вътрешния прецизен генератор.

След всяко препрограмиране на флаш-конфигурационните битове, микроконтролерът трябва да се ресетне, за да може новите настройки да станат активни. При всеки тип ресет (системен или излизане от STOP-режим) флаш-конфигурационните битове се изчитат от Програмната

Памет и се записват в конфигурационни регистри. Тези регистри не са част от Регистровата Памет и не са достъпни за запис или четене.

16.2 Типове конфигурационни битове

Потребителски флаш-конфигурационни битове

Тези битове заемат първите два байта на Програмната Памет. Информацията в тези байтове се губи при изтриване на флаш-страница 0 на Програмната Памет.

Бит	7	6	5	4	3	2	1	0
Име	WDT_RES	WDT_AO	OSC_SEL[1:0]		VBO_AO	FRP		FWP
Ресет	Н	Н	Н	Н	Н	Н	Н	Н
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	Програмна Памет 0000h							
Н - непроменен								
Бит	Описание							
7	Реакция на WDT-таймера при препълване 0 – WDT-таймерът генерира прекъсване 1 – WDT-таймерът генерира системен ресет							
6	Включване на WDT-таймера 0 – WDT-таймерът е включен и не може да бъде забранен 1 – WDT-таймерът не е включен. WDT-таймерът се включва при изпълнение на инструкцията WDT. Веднъж включен таймерът може да бъде забранен само от ресет.							
5-4	Режим на генератора 00 – Външна RC верига (< 4MHz) 01 – кварцов резонатор (32kHz ÷ 1MHz) 10 – кварцов (керамичен) резонатор (0.5MHz ÷ 5MHz) 11 – кварцов резонатор (5MHz ÷ 20MHz)							
3	Разрешаване на VBO защитата 0 – VBO защитата е забранена в STOP-режим 1 – VBO защитата винаги е разрешена (дори и в STOP-режим)							

2	Защита на флаш-паметта (програмния код) от прочитане 0 – Програмният код е защитен 1 – Програмният код е незащитен
1	Този бит е резервиран и трябва да бъде 1.
0	Защита на флаш-паметта от програмиране и изтриване 0 – Програмирането и изтриването е забранено за цялата флаш-памет. В потребителския код програмирането, изтриването на страница и на цялата памет също е забранено. Цялото изтриване на паметта е възможно чрез дебъгера. 1 – Програмирането, изтриването на страница и на цялата памет е разрешено.

Табл. 79 Потребителски флаш-конфигурационен байт 0

Бит	7	6	5	4	3	2	1	0
Име	VBO_RES			XTLDIS				
Ресет	Н	Н	Н	Н	Н	Н	Н	Н
Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З	Ч/З
Адрес	Програмна Памет 0001h							
Н – непроменен								
Бит	Описание							
7	1 – VBO-модула генерира системен ресет							
6-5	Тези битове са резервирани и трябва да бъдат 1.							
4	Включване на кварцовия генератор при ресет 0 – Кварцовият генератор се включва при ресет 1 – Кварцовият генератор не се включва при ресет							
3-0	Тези битове са резервирани и трябва да бъдат 1.							

Табл. 80 Потребителски флаш-конфигурационен байт 1

Фабрични флаш-конфигурационни битове

Тези флаш-конфигурационни битове са настроени фабрично и не се нуждаят от потребителска намеса. По тази причина тази тема няма да бъде разгледана в книгата.

17 Енерго-независима памет за данни (NVDS)

17.1 Общи сведения

Z8Encore!F083A микроконтролерите съдържат енергонезависима памет NVDS (**Non-Volatile Data Storage**) с обем до 100 байта, която издържа на 100 000 цикъла за запис.

За достъп до NVDS се използват две подпрограми: подпрограма за запис и подпрограма за четене. Тези подпрограми са предефинирани и се разполагат на адреси извън Програмната Памет, използвана от потребителския код. Достъпът до тях става с инструкцията CALL.

За да не разрушават потребителския код, тези подпрограми съхраняват в стека текущия набор от работни регистри R0 ÷ R15. При излизане от тези подпрограми съдържанието на работните регистри се възстановява.

При достъпа до NVDS прекъсванията не са забранени. Всяко прекъсване, което възниква по време на достъп до NVDS, не трябва да променя работните регистри и текущото съдържание на стека, в противен случай съдържанието на NVDS може да бъде повредено. Zilog препоръчва прекъсванията да бъдат забранени преди достъп до NVDS.

За правилна работа на NVDS, флаш-честотните регистри FFREQH:FFREQH трябва да бъдат програмирани в зависимост от стойността на системната тактова честота (вижте [15 Флаш-памет](#)).

17.2 Запис на байт в NVDS

За да запише байт в NVDS, потребителският код трябва първо да запише адреса (00h ÷ 63h) и след това данните (00h ÷ FFh) в стека. Записът на данните в NVDS се осъществява чрез извикване на подпрограмата за запис на адрес 20B3h. При връщане от подпрограмата състоянието на резултата от записа се съхранява в работния регистър R0 (Табл.81). Потребителският код трябва да махне адреса и байта с данни от стека.

Запис на невалиден адрес няма никакъв ефект.

Бит	7	6	5	4	3	2	1	0
Име						FE	IGADDR	WE
Адрес	Работен регистър R0							
Бит	Описание							
7-3	-							
2	Флаш-грешка 0 – Няма флаш-грешка 1 – Възниканала е флаш-грешка							
1	Невалиден адрес 0 – Валиден адрес 1 – Невалиден адрес							
0	Грешка при запис 0 – Няма грешка 1 – Има грешка. При запис записаната стойност автоматично се изчита и ако прочетената стойност се различава от записваната, този бит се установява.							

Табл. 81 Статус байт при NVDS запис

17.3 Четене на байт от NVDS

За да прочете байт от NVDS, потребителският код трябва първо да запише адреса (00h ÷ 63h) в стека. Четенето на данните от NVDS се осъществява чрез извикване на подпрограмата за четене на адрес 2000h. При връщане от подпрограмата прочетеният байт се намира в работния регистър R0, а състоянието на резултата от четенето се съхранява в работния регистър R1 (Табл.82). Потребителският код трябва да махне адреса от стека.

Четене от невалиден адрес връща стойност FFh.

Бит	7	6	5	4	3	2	1	0
Име				DE		FE	IGADDR	
Адрес	Работен регистър R1							
Бит	Описание							
7-5	-							
4	Грешка на данните 0 – Няма грешка 1 – Възниканала е грешка. Когато се чете NVDS адрес, ако се намери грешка в най-последните данни, съответстващи на този адрес, този бит се установява.							
3	-							
2	Флаш-грешка 0 – Няма флаш-грешка 1 – Възниканала е флаш-грешка							
1	Невалиден адрес 0 – Валиден адрес 1 – Невалиден адрес							
0	-							

Табл. 82 Статус байт при NVDS четене

18 Дебъгер

18.1 Общи сведения

Z8Encore! фамилията микроконтролери съдържа вграден дебъгер (**OCD – On-Chip Debugger**), който има следните характеристики:

- Четене и запис на Регистровата Памет;
- Четене и запис на Програмната Памет;
- Четене и запис на Паметта за Данни;
- Задаване на точки на прекъсване (**breakpoints**);
- Изпълняване на инструкции на процесора eZ8.

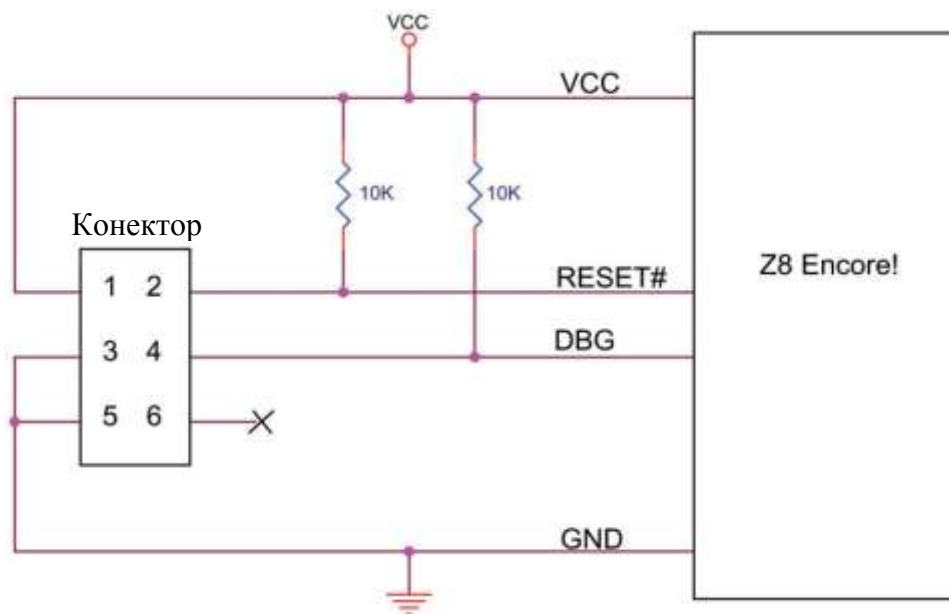
Връзката между вградения дебъгер на микроконтролера и компютър става посредством интерфейса **USB SmartCable** (Фиг. 34)



Фиг. 34 USB SmartCable интерфейс

18.2 Свързване на дебъгера към микроконтролера

Следващата фигура показва начина на свързване на **USB SmarCable** интерфейса с микроконтролера.



Фиг. 35 Свързване на USB SmartCable интерфейс с Z8 Encore! микроконтролер

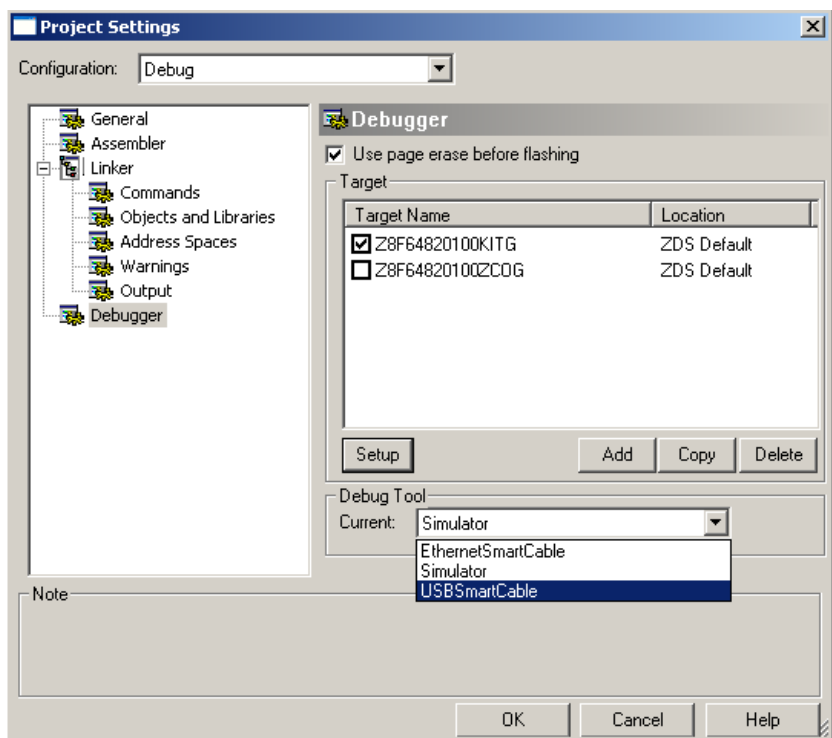
18.3 Тестване/дебъгване с USB SmartCable

За да използвате дебъгера, трябва да го изберете като инструмент за тестване/дебъгване. За целта в развойната среда **Zilog Developer Studio II** изберете **Project** → **Settings...** Отваря се прозореца **Project Settings** (Фиг.36). Изберете категорията **Debugger** и от менюто **Debug Tool:Current** изберете **USBSmartCable**.

Свържете дебъгера към компютъра и платката (Фиг.37).



Захранването на развойната платка трябва да бъде изключено преди да свържете/откачите дебъгера към/от нея.



Фиг. 36 Project Ssettings



Фиг. 37 Свързване на дебъгера към платката и компютъра

Използвайте следните бутони в средата ZDS II, за да се свържете с микроконтролера на платката и да заредите програмата в Програмната Памет.



- | ☐ Зареждане на програмата в Програмната Памет на микроконтролера
- | ☐ Свързване на ZDS II с микроконтролера посредством дебъгера

Може да използвате и ресет бутона . ZDS II средата първо ще се свърже с микроконтролера на платката, след което ще зареди програмата в Програмната Памет.

Допълнителна информация може да намерите [39 Описание на развойната среда Zilog Developer Studio II](#).

19 Електрически параметри

Пълно описание на електрическите параметри на Z8Encore!F083A серията микроконтролери може да намерите в оригиналната документация [Z8Encore!®F083A Series. Product Specification \(PS026310-1212\).](#)

20 Информация за поръчка

Може да използвате следващата таблица при избора си на микроконтролер от серията Z8Encore!F083A.

Номер на устройството	Флаш	RAM	NVDS	АЦП канали	Корпус
Z8 Encore!F083A с 8kB Флаш (0°C до 70°C)					
Z8F083ASH020SG	8kB	256B	100B	7	SOIC-20
Z8F083ANH020SG				7	SSOP -20
Z8F083APH020SG				7	PDIP-20
Z8F083AQH020SG				7	QFN-20
Z8F083ASJ020SG				8	SOIC-28
Z8F083AHJ020SG				8	SSOP-28
Z8F083AQJ020SG				8	QFN-28
Z8 Encore!F083A с 8kB Флаш (-40°C до 105°C)					
Z8F083ASH020EG	8kB	256B	100B	7	SOIC-20
Z8F083ANH020EG				7	SSOP -20
Z8F083APH020EG				7	PDIP-20
Z8F083AQH020EG				7	QFN-20
Z8F083ASJ020EG				8	SOIC-28
Z8F083AHJ020EG				8	SSOP-28
Z8F083AQJ020EG				8	QFN-28
Z8 Encore!F083A с 4kB Флаш (0°C до 70°C)					
Z8F043ASH020SG	4kB	256B	100B	7	SOIC-20
Z8F043ANH020SG				7	SSOP -20
Z8F043APH020SG				7	PDIP-20
Z8F043AQH020SG				7	QFN-20
Z8F043ASJ020SG				8	SOIC-28
Z8F043AHJ020SG				8	SSOP-28
Z8F043AQJ020SG				8	QFN-28
Z8 Encore!F083A с 4kB Флаш (-40°C до 105°C)					
Z8F043ASH020EG	4kB	256B	100B	7	SOIC-20
Z8F043ANH020EG				7	SSOP -20

Z8F043APH020EG				7	PDIP-20
Z8F043AQH020EG				7	QFN-20
Z8F043ASJ020EG				8	SOIC-28
Z8F043AHJ020EG				8	SSOP-28
Z8F043AQJ020EG				8	QFN-28

Табл. 83 Z8 Encore!F083A серия микроконтролери

Значение на символите в номера на устройството:

Z8 F 08 3A S H 020 S G

G = Green Plastic Packaging Compound

Температурен обхват

S = Стандартен, 0°C до 70°C

E = Разширен, -40°C до 105°C

Скорост

020 = 20MHz

Брой изводи

H = 20

J = 28

Корпус

P = PDIP

Q = QFN

S = SOIC

H = SSOP

Тип устройство

3A = Екипирано с бърз АЦП

Размер памет

08 = 8kB Флаш

04 = 4kB Флаш

Тип памет

F = Флаш

Фамилия

Z8 = Zilog 8 битови микроконтролери

DRAFT

Част II

Кратък С справочник



DRAFT

21 Въведение

Програмният език C е създаден през 70-те години на миналия век от Денис Ричи в лабораторията Бел. През 1978г., заедно с Браян Кърнигън, той публикува първата версия на езика C, която в днешно време е известна като **K&R стандарт**.

Езикът C е стандартизиран първо през 1989 г. от ANSI (**American National Standards Institute**) комитета. През 1990 г. този стандарт е приет и от международната организация ISO (**International Standards Organization**). Затова той се нарича още ANSI/ISO стандарт на езика C. Тези два стандарта са еднакви и често се наричат C89 или C90, и се означават съответно като **ANSI X3.159-1989** и **ISO/IEC 9899:1990**. През следващите няколко години стандартът претърпява няколко корекции и добавки както е показано долу:

ISO/IEC 9899:1990/Cor 1:1994

ISO/IEC 9899:1990/Amd 1:1995 (Наричан формално C95)

ISO/IEC 9899:1990/Cor 2:1996

През 1999 г. е приет нов стандарт **ISO/IEC 9899:1999**, който заменя предходния с всички корекции и добавки. Този стандарт е известен още като C99. Той също претърпява изменения през следващите няколко години както е показано долу:

ISO/IEC 9899:1999/Cor 1:2001

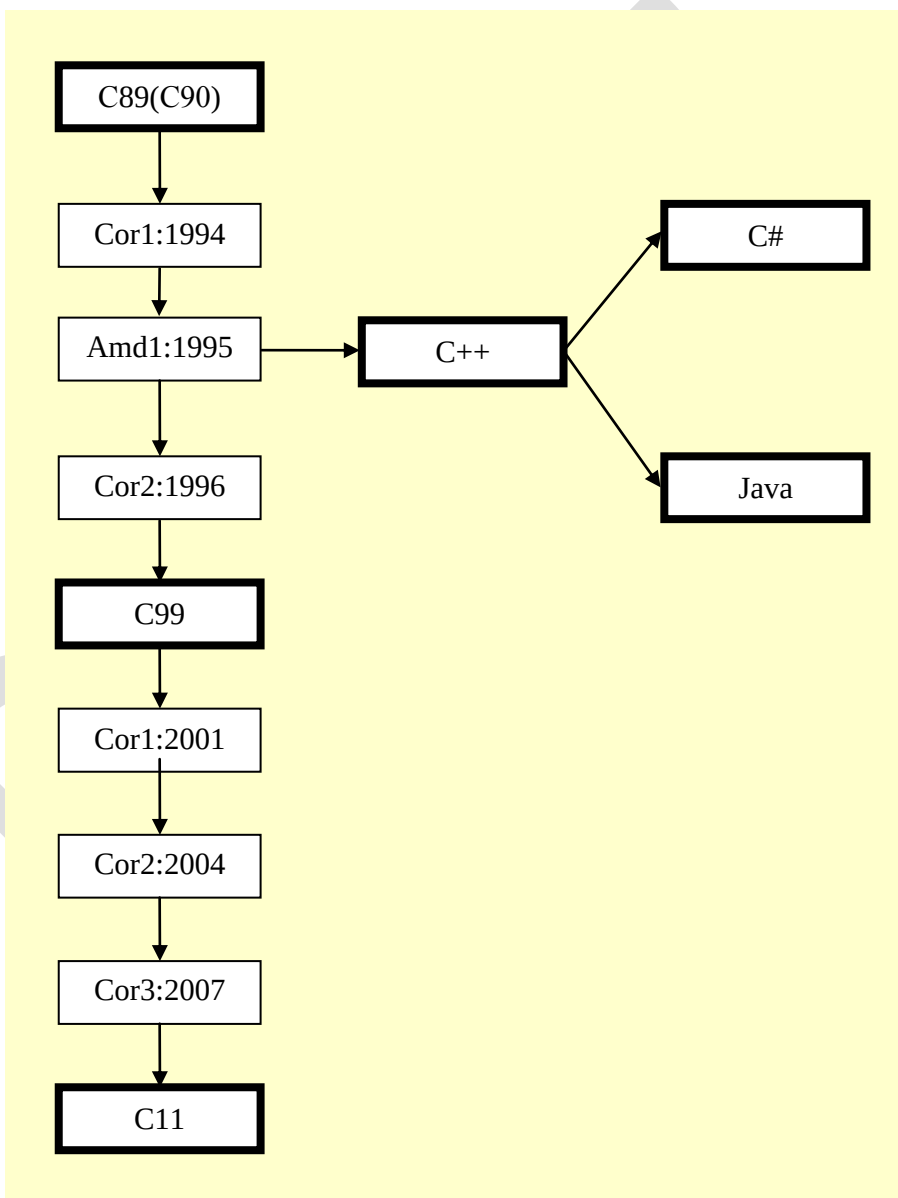
ISO/IEC 9899:1999/Cor 2:2004

ISO/IEC 9899:1999/Cor 3:2007

През 2011 г. е приет нов стандарт **ISO/IEC 9899:2011**, формално наричан C11.

Стандартът C95 (ISO/IEC 9899:1990/Amd 1:1995) е основа на езика C++, който пък е основа на двата най-популярни езика за обектно-ориентирано програмиране: C# и Java.

Фиг.38 показва нагледно описаните по-горе периоди на стандартизиране на езика C.



Фиг. 38 Стандартизиране на езика C

Въпреки че С е език за обща употреба, той най-често се използва в системното програмиране. Това включва създаването на софтуер, който управлява компютърния хардуер директно, като драйвери, операционни системи, и софтуер за микроконтролери (**firmware**). Езикът С е ориентиран към архитектурата на днешните компютри и се характеризира със следните предимства:

- Малък, сбит, но мощен език;
- Сорс-кодът е с висока степен на преносимост;
- Машинният код е ефикасен;
- С-компилатори са налични за всички днешни компютърни системи.

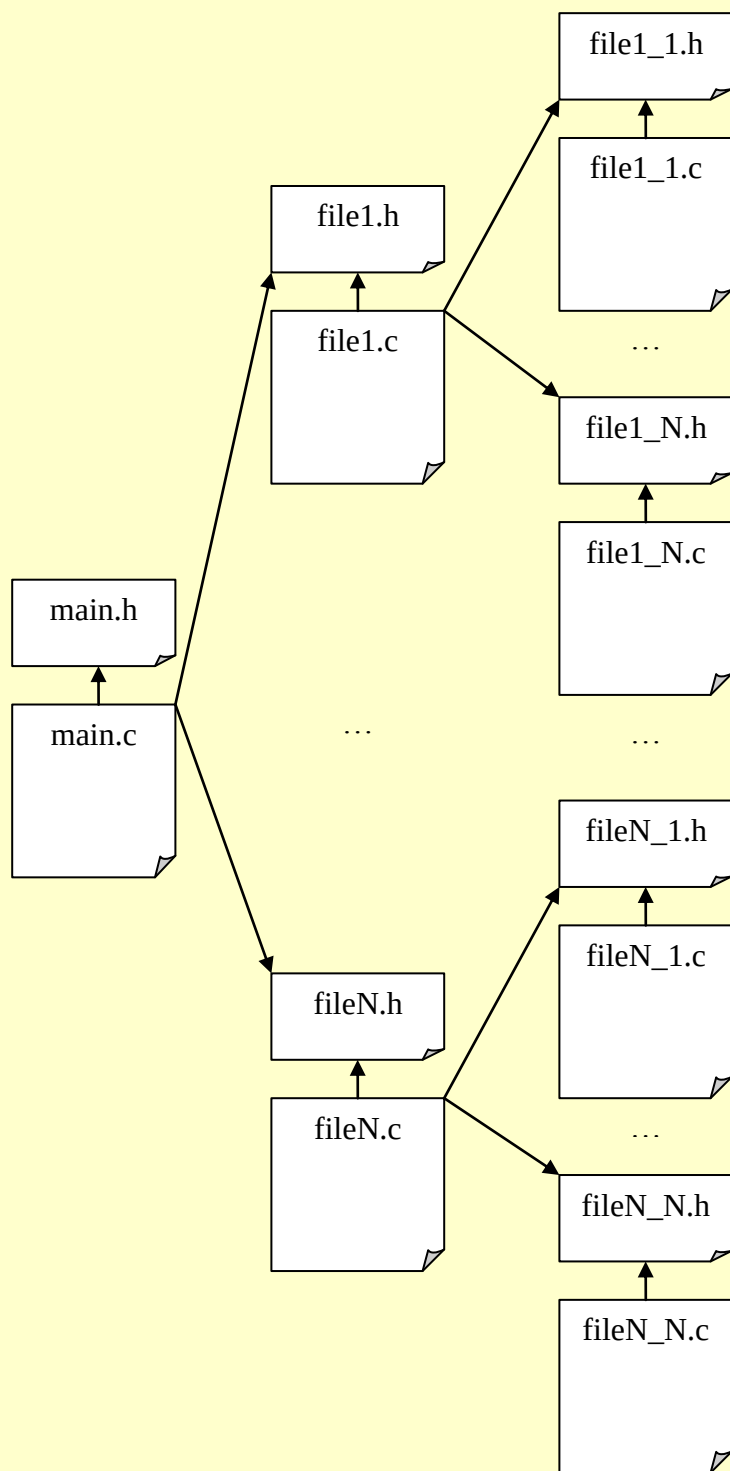
Описанието на езика С в тази част се базира основно на **C89** стандарта и някои нововъведения на **C99** стандарта.

22 Основи

22.1 Структура на C програма

Градивните блокове на една C програма са функциите. Функцията е подпрограма, която се изпълнява при нейното стартиране (извикване). Всяка C програма започва изпълнението си от специална функция, наречена **main()**, която се извиква автоматично при стартиране на програмата. Тази функция е определена от стандарта C и задължително трябва да присъства във всяка C програма. Функциите се разделят на библиотечни и потребителски. Библиотечните функции се предоставят от производителя на компилатора и се използват наготово от потребителя. Библиотечните функции от своя страна се разделят на стандартни и нестандартни. Стандартни са тези, които са дефинирани от C стандарта. Към тях всеки производител на C компилатор може да добави свои библиотечни функции. Потребителските функции се създават от самия програмист.

Една C програма се състои от един или повече сорс-файлове. Фиг.39 представя типична организация на многофайлова C програма. Обикновено сорс-файлът, където се намира функцията `main`, се обозначава като `main.c`. За всеки сорс-файл се създава хедър-файл със същото име, който се включва в началото на сорс-файла с помощта на директивата `#include`. Хедър-файлът съдържа дефиниции на макроси, декларации на потребителски типове (с `struct`, `union`, `enum` и `typedef`), декларации на външни променливи и прототипи на функции, които трябва да бъдат достъпни и за други сорс-файлове. Всеки сорс-файл, който използва изброените по-горе дефиниции и декларации, трябва да включва и съответния хедър-файл. Това е отбелязано на фигурата със стрелки. Например `main.c` използва дефинициите и декларациите на `file1.h`, ..., `fileN.h` и затова тези хедър-файлове трябва да бъдат добавени в него. Аналогично `file1.c`



Фиг. 39 Типична структура на C програма

използва дефинициите и декларациите на file1_1.h,..., file1_N.h и затова тези хедър-файлове трябва да бъдат добавени в него, и т.н.

Най-общо една С програма може да се представи по следния начин:

```
/*
 * 1. Включване на хедър-файлове
 */
#include < хедър-файл1.h >
#include < хедър-файл2.h >
....
#include < хедър-файлN.h >

/*
 * 2. Описание на прототипите на потребителските функции
 */
тип име-функция1(списък-параметри);
тип име-функция2(списък-параметри);
....
тип име-функцияN(списък-параметри);

/*
 * 3. Дефиниране на глобални променливи
 */
тип име-променлива1;
тип име-променлива2;
....
тип име-променливаN;

/*
 * 4. Начало на С програмата
 * Изпълнението на С програмата започва тук!
 */
int main(void)
{
    /* 4.1 Дефиниране на локални променливи */
    тип име-променлива1;
    тип име-променлива2;
    ....
    тип име-променливаN;

    /* 4.2 Потребителски код */
    оператор1;
    оператор2;
    ....
    операторN;
```

```

        return 0;
    }

/*****
 *           5. Дефиниране на потребителските функции           *
 *****/
тип    име-функция1(списък-параметри)
{
    /* 5.1 Дефиниране на локални променливи */
    тип    име-променлива1;
    тип    име-променлива2;
    ....
    тип    име-променливаN;

    /* 5.2 Потребителски код */
    оператор1;
    оператор2;
    ....
    операторN;
}

тип    име-функция2(списък-параметри)
{
    /* 5.1 Дефиниране на локални променливи */
    тип    име-променлива1;
    тип    име-променлива2;
    ....
    тип    име-променливаN;

    /* 5.2 Потребителски код */
    оператор1;
    оператор2;
    ....
    операторN;
}
....

тип    име-функцияN(списък-параметри)
{
    /* 5.1 Дефиниране на локални променливи */
    тип    име-променлива1;
    тип    име-променлива2;
    ....
    тип    име-променливаN;

    /* 5.2 Потребителски код */
    оператор1;
    оператор2;
    ....
    операторN;
}

```

Основните части на една C програма са :

- **Хедър-файлове** – Това са файлове с разширение .h, които съдържат информация необходима за изпълнението на програмата, например прототипи на библиотечни функции. Съдържанието на тези файлове се включва с помощта на предпроцесорната директива **#include**.
- **Прототипи на функции** – Прототипът на една функция дава информация на компилатора за самата функция. Въпреки че те не са задължителни, всяка професионално написана програма използва прототипи.
- **Глобални променливи** – променливата е област от паметта, използвана за съхранение на данни. Когато една променлива е обявена извън всички функции, тя се нарича глобална. Глобалните променливи могат да се използват от всички функции.
- **Функция main()** – Основните градивни елементи на една C програма са функциите. Функцията в езика C представлява подпрограма, която се изпълнява при нейното извикване. Всяка една C програма започва изпълнението си от специална функция, наречена **main()**, която се извиква автоматично при стартиране на програмата. Тази функция е определена от стандарта C и задължително трябва да присъства във всяка една C програма. Изпълнението на програмата започва след отварящата фигурна скоба { и завършва след затварящата фигурна скоба } на **main()**.
- **Локални променливи** – когато една променлива е обявена вътре в една функция, тя се нарича локална. Локалните променливи могат да се използват само от функцията, в която са обявени. Те не са достъпни за

други функции.

- **Функции** - Функциите се разделят на библиотечни и потребителски. Библиотечните функции се предоставят от производителя на компилатора и се използват наготово от потребителя. Библиотечните функции от своя страна се разделят на стандартни и нестандартни. Стандартни са тези, които са дефинирани от C стандарта. Към тях всеки производител на C компилатор може да добави свои библиотечни функции. Потребителските функции се създават от самия програмист. Подобно на функцията **main()**, изпълнението на кода на всяка функция започва и завършва с отваряща и затваряща фигурна скоба. Тялото (кодът) на една функция се изгражда с помощта на оператори. Функциите могат да приемат стойности и да връщат резултат.
- **Оператори** - Операторът е програмна конструкция, която извършва някакво действие. Всички оператори в C завършват с точка и запетая. Тя служи за разпознаване на край на оператор. След като един оператор се изпълни, програмата преминава към изпълнението на следващия оператор.
- **Коментари** - Коментарът представлява потребителско съобщение затворено между **/*** и ***/**. Всичко поставено между тези символи се игнорира от компилатора и служи единствено за пояснение на програмата.

Например:

/* Това е коментар */

или

/* Това също е коментар, но на няколко реда */

Въпреки че коментарите не са задължителни, те са много полезни. Всяка професионално написана програма използва коментари, за да документира кода и действията, които се извършват.

Най-кратко казано, една С програма се състои от функции, които от своя страна се състоят от оператори.

Фиг.40 и 41 показват примерна структура на .h и .c файл съответно.

```
#ifndef FILENAME_H
#define FILENAME_H

/* 1. Описание на файла: име, автор, версия и т.н. */
...
/* 2. Системни хедър-файлове */
#include <SystemHeader.h>
...
/* 3. Потребителски хедър-файлове */
#include "UserHeader.h"
...
/* 4. Дефиниции на външни макроси */
#define име-макрос заменящ-текст
...
/* 5. Декларации на външни потребителски типове */
struct име-структура {членове};
...
union име-обединение {членове};
...
enum име-обединение {членове};
...
typedef тип ново-име-на-тип;
...
/* 6. Декларации на локални константи */
static const тип име-константа;
...
/* 7. Декларации на глобални външни променливи */
extern тип име-променлива;
...
/* 8. Прототипи на външни функции */
extern тип име-функция(списък-параметри);
...

#endif
```

Фиг. 40 Примерна структура на .h файл


```

/* 1. Описание на файла: име, автор, версия и т.н. */
...
/* 2. Системни хедър-файлове */
#include <SystemHeader.h>
...
/* 3. Потребителски хедър-файлове */
#include "UserHeader.h"
...
/* 4. Дефиниции на вътрешни макроси */
#define име-макрос заменящ-текст
...
/* 5. Декларации на вътрешни потребителски типове */
struct име-структура {членове};
...
union име-обединение {членове};
...
enum име-обединение {членове};
...
typedef тип ново-име-на-тип;
...
/* 6. Дефиниции на глобални константи */
const тип име-константа = инициализатор;
...
/* 7. Дефиниции на глобални външни променливи */
тип име-променлива;
...
/* 8. Дефиниции на локални константи */
static const тип име-константа = инициализатор;
...
/* 9. Дефиниции на глобални статични променливи */
static тип име-променлива;
...
/* 10. Прототипи на статични функции */
static тип име-функция(списък-параметри);
...

/* 11. Дефиниции на глобални и статични функции */
тип име-функция(списък-параметри)
{
    /* код */
}

...

static тип име-функция(списък-параметри)
{
    /* код */
}
...

```

Фиг. 41 Примерна структура на .c файл

22.2 Среда

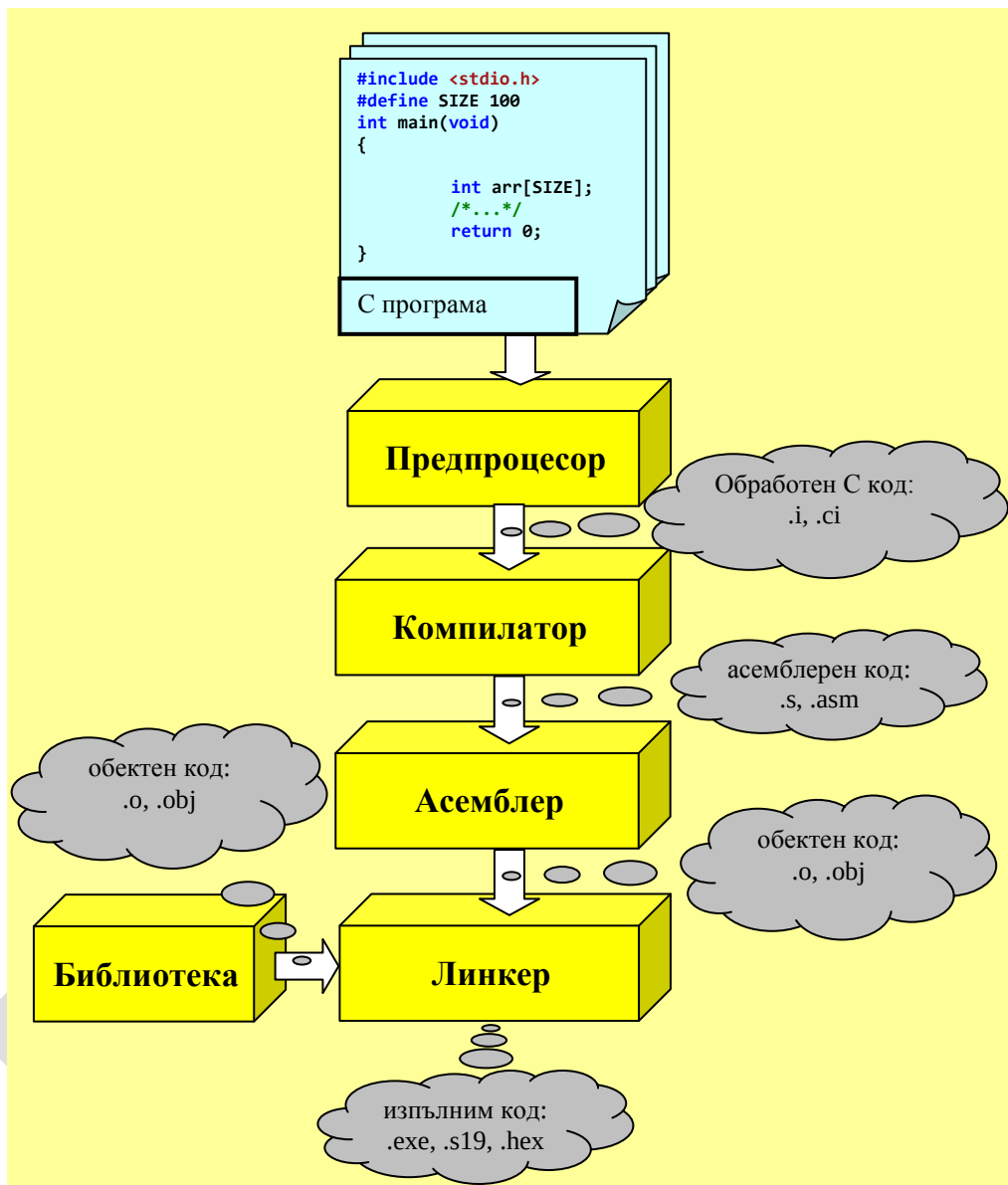
Една С-програма може да бъде написана и компилирана на една компютърна система и да се изпълнява на друга.



Когато една програма се компилира в една среда, а се изпълнява в друга, процесът на транслиране е известен като крос-компилиране.

22.2.1 Транслираща среда

Транслираща се нарича компютърната система, на която програмата се компилира. Фиг.42 описва процеса на компилиране на С-програма. С програмата се състои от един или няколко сорс-файла с разширение **.c**. Преди да се компилират, тези файлове се обработват от програма наречена Предпроцесор. Обикновено обработените от Предпроцесора С файлове имат разширение **.i** или **.ci**. След това, всеки обработен файл се компилира. Процесът на компилация представлява преобразуване на обработения С код в асемблерен код. Обикновено обработените от Компилятора файлове имат разширение **.s** или **.asm**. Асемблерният код от своя страна се обработва от програма, наречена Асемблер, която го преобразува в т.нар. обектен код (машинни инструкции). Обикновено обработените от Асемблера файлове имат разширение **.o** или **.obj**. Линкерът свързва обектните файлове в един изпълним файл. Това е крайният файл от целия процес на компилация, който е готов за изпълнение. На Линкера могат да се подават и външни обектни файлове (например получени от стандартни С библиотечни файлове). Изпълнимите файлове имат различни разширения в зависимост от това къде ще се изпълнява кода. Например изпълним файл предназначен да се изпълнява на персонален компютър има разширение **.exe**.



Фиг. 42 Създаване и компилиране на С програма

Изпълними файлове, предназначени за изпълнение в микроконтролери, имат разширения като **.s19**, **.hex**, **.elf** и други.

Компилирането на една С-програма включва следните фази:

1. Символите на сор-файла се четат и се преобразуват,

ако е необходимо, в символите на сорс символната таблица. Индикаторите за край на ред се заменят със символ за нов ред. Триграфите се заменят с техните еквивалентни символи (диграфите не се заменят с техните еквивалентни символи).

2. Всички редове, завършващи с обратно наклонена черта (`\`), последвана от символ за нов ред, се свързват със следващия ред, формирайки логически ред от няколко физически реда. Освен ако е празен, един сорс-файл трябва да завършва със символ за нов ред, който не е предшестван от обратно наклонена черта (`\`).
3. Сорс-файлът се разбива на предпроцесорни лексеми. Всеки коментар се заменя с интервал. Символите за нов ред се запазват.
4. Предпроцесорните директиви се изпълняват и се извършва разширяване на макросите. Фази 1 ÷ 4 се прилагат рекурсивно за всички файлове вмъкнати с предпроцесорната директива `#include`.
5. Символите и ескаре-последователостите в символните и низовите константи се преобразуват в съответните символи от изпълняващата символна таблица.
6. Съседните низове се свързват в един низ.
7. Извиква се компилатора. Всички лексеми се анализират синтактично и се преобразуват в обектен код.
8. Извиква се линкера. Всички референции към външни обекти и функции се разрешават. Ако линкерът „засече“ използването на идентификатор на

променлива или функция, но не открива дефиницията му в нито един от файловете, се генерира грешка.

22.2.2 Изпълняваща среда

Изпълняваща се нарича компютърната система, на която програмата се изпълнява. ANSI C стандартът дефинира два типа изпълняващи среди и съответно два типа имплементации на C-компилаторите - фрийстендинг (**freestanding**) и хостед (**hosted**).

Неформално, хостед-компилатор се нарича компилатор, който поддържа всички възможности на езика C и цялата стандартна библиотека. Съответно, компютърна система, която може да изпълнява хостед C-програми, се нарича хостед изпълняваща среда. Такива компютърни системи обикновено работят под управлението на операционна система, имат файлова система, поддържат работа с входни/изходни устройства (клавиатура, монитор, принтер и др.). Типичен пример за хостед изпълняваща компютърна система е персоналният компютър.

Функцията, която се извиква при стартирането на хостед C-програма се казва **main**. ANSI C стандартът дефинира следните два типа:

```
int main(void) {...}
```

и

```
int main(int argc, char* argv[]) {...}
```

където

argc

Съдържа броя на всички аргументи подадени от командия ред, в това число и името на програмата. Ако няма подадени входни аргументи, параметърът **argc** съдържа стойност 1, тъй като

името на програмата се брои за първи аргумент по подразбиране.

argv

Масив от указатели към `char`. Всички аргументи от командния ред представляват символни низове. Първият елемент `argv[0]` сочи към името на програмата, втория елемент `argv[1]` сочи към първия аргумент, следващ името на програмата, и т.н. Масивът `argv[]` винаги съдържа един елемент повече от броя на аргументите от командния ред (в това число влиза и името на програмата). По дефиниция този елемент е нулев указател, т.е. `argv[argc] == NULL`.



Името на програмата, въведено в командния ред, както и всички входни аргументи, се разглеждат като символни низове. Ако входните аргументи са числа, то преди да се използват в програмата, те трябва да се преобразуват от низове в съответния им числов еквивалент.

Стойността връщана от **main()** се подава на извикващата я среда. Връщаната стойност може да бъде 0 или **EXIT_SUCCESS**, което е индикация, че програмата е завършила нормално, или **EXIT_FAILURE**, което е индикация за някакъв проблем. Макросите **EXIT_SUCCESS** и **EXIT_FAILURE** са дефинирани в хедъра **stdlib.h**. Използването на други стойности е имплементационно-зависимо. Някои среди игнорират връщаната стойност.

Фрийстендинг-компилятор се нарича компилатор, който поддържа напълно или по-голямата част от езика C, без стандартната библиотека, с изключение на средствата осигурени в хедър-файловете **float.h**, **iso646.h**, **limits.h**, **stdarg.h**, **stdbool.h**^{C99}, **stddef.h** и **stdint.h**^{C99}. Казано на жаргон фрийстендинг-компиляторът е „олекотена” версия на хостед-компилятор. Типично фрийстендинг-компиляторите са предназначени за компютърни системи с ограничени възможности, такива, които обикновено не работят под управлението на операционна система, нямат файлова система и не поддържат работа с входни/изходни устройства, като клавиатура, монитор, принтер и т.н. Типичен пример за

фрийстендинг компютърни системи са микроконтролерите.

Името и типът на функцията, която се извиква при стартирането на фрийстендинг С-програма, зависят от имплементацията. На практика фрийстендинг С-компилаторите използват следните типове:

```
int main(void) {...}
```

или

```
void main(void) {...}
```

Ефектът от завършването на програмата във фрийстендинг среда зависи от имплементацията.

22.3 Характеристики на средата

22.3.1 Символни таблици

С-програмата представлява съвкупност от символи. В компютърните системи зад всеки символ стои определен числов код, т.е. . на най-ниско ниво компютърът „вижда” число, а не символ. Различните компютърни системи могат да използват различно кодиране на символите. Например символът 'A' има стойност 65 в ASCII символната таблица и стойност 193 в EBCDIC символната таблица.

ANSI C стандартът дефинира две таблици със символи:

- **Сорс символна таблица (source character set)**

Това е символната таблица, използвана от компютърната система, на която се създава и компилира програмата. Тя включва всички символи, които могат да се използват в един сорс-файл.

- **Изпълняваща символна таблица (execution character set)**

Това е символната таблица, използвана от компютърната система, на която се изпълнява програмата.

В много C-компилатори двете таблици са идентични (използват еднакви символи и кодиране на символите).

Стандартът дефинира общ набор от символи (**basic character set**), които тези две таблици трябва да съдържат.

- 52 букви от латинската азбука в долен и горен регистър;

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V X Y Z

- 10 арабски цифри;

0 1 2 3 4 5 6 7 8 9

- 29 графични символи;

! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~

- 5 празни символи - интервал, хоризонтална табулация, вертикална табулация, нова страница, нов ред

ANSI C стандартът изисква членовете на базовата символна таблица да бъдат еднобайтови.

Освен гореизброените символи, базовата символна таблица в изпълняващата компютърна ситема (**basic execution**

character set) трябва да съдържа управляващи символи за:

- звуков сигнал (**alert**);
- връщане на курсора с една позиция назад (**backspace**);
- връщане на курсора в началото на реда (**carriage-return**);
- нулев символ – представлява байт, чийто битове имат стойност 0.

При компилиране на програмата компилаторът заменя символите в символните и низовите константи с числовите стойности на съответните символи от изпълняващата символна таблица¹. Числовите стойности на символите в изпълняващата символна таблица зависят от имплементацията.

Забележка¹: Ако управляващ-израз на директивата `#if` съдържа символни константи, предпроцесорът може да използва числовото кодиране на сорс символната таблица или на изпълняващата символна таблица. Това зависи от имплементацията на компилатора. Например няма гаранция, че константният израз в следната `#if` директива и `if` оператор ще има една и съща стойност.

```
#if ('z' - 'a' == 25)
```

```
if ('z' - 'a' == 25)
```

Типична програмна грешка е допускането, че символите, използвани в изпълняващата компютърна система, използват специфично кодиране, например ASCII. Кодирането на символите в изпълняващата символна таблица зависи от имплементацията. Например:

израз	резултат	
	ASCII	EBCDIC
'A'+'Z' - 1	26	41

Двете таблици могат да съдържат допълнителни символи (**extended characters**). Базовата символна таблица и допълнителните символи образуват разширена символна таблица (**extended character set**), т.е..

extended character set = basic character set + extended characters

Стандартът не специфицира разширените символи и тяхното кодиране.



Освен ако не е абсолютно необходимо, избягвайте използването на символи извън базовата символна таблица. Използването на разширени символи намалява преносимостта на програмата.

22.3.2 Диграфи и Триграфи

Някои компютърни системи не поддържат определени символи от базовата символна таблица. Такива компютърни системи могат да използват определени комбинации от символи, наречени **диграфи**^{C99} и **триграфи** за представяне на липсващите символи. Диграфите се състоят от два символа, докато триграфите от три (Табл.84).

символ	триграф	диграфи ^{C99}
[	??(	<:
\	??/	
]	??)	:>
^	??'	
{	??<	<%
	??	
}	??>	%>
~	??-	

#	??=	%:
##	-	%: %:

Табл. 84 Триграфи и Диграфи

Пример:

```
int main(void)
<% /* Еквивалентен запис на { */

    return 0;
%> /* Еквивалентен запис на } */
```

Пример:

```
char arr??(??) = "Hello C??/n"; /* char arr[] = "Hello C\n"; */
```

Диграфите не се интерпретират като такива, когато са част от символна или низова константа. Във всички останали позиции те се държат като символа, който представляват.

Триграфите се заменят с техните символни еквиваленти в първата фаза на компилирането на кода. Това означава, че триграфите, за разлика от диграфите, се преобразуват в техните символни еквиваленти без значение дали се намират в символна константа, низова константа, коментари и предпроцесорни директиви.

22.3.3 Универсални символи

C99 въвежда т.нар. универсални символи (**universal character names**) за представяне разширените символи, независимо от начина им на кодиране, избран от имплементацията. Универсалните символи имат следния формат:

\uXXXX

или

`\UXXXXXXXX`

където

`X`

Шестнайсетична цифра ($0 \div 9$ и $a(A) \div f(F)$).

Универсалните символи могат да се използват в:

- идентификатори;
- символни константи;
- низови константи.

Те представляват символи от т.нар. **Универсален Символен Набор - (UCS - Universal Character Set)**. UCS е дефиниран от международния стандарт ISO/IEC-10646 и съдържа над 1 000 000 символа. Всеки универсален символ се представя чрез т.нар. **кодова точка (code point)**, която е цяло число в диапазона $0 \div 1\,114\,111$ ($0x000000 \div 0x10FFFF$). За всяка кодова точка стандартът ISO/IEC-10646 дефинира т.нар. кратък идентификатор (**short identifier**). Универсален символ, изразен чрез `\UXXXXXXXX`, обозначава символ, чийто осемцифрен кратък идентификатор е `XXXXXXXX`. Универсален символ, изразен чрез `\uXXXX`, обозначава символ, чийто кратък четирицифрен идентификатор е `XXXX`. Четирицифрените кратки идентификатори са еквивалентни на осемцифрени кратки идентификатори, чийто 4 старши цифри са нула, т.е.. $XXXX \approx 0000XXXX$.

Универсални символни имена, чийто кратък идентификатор е в диапазона $0000 \div 00A0$ и $D800 \div DFFF$, не са позволени. Изключение са символите с кратки идентификатори 0024 (\$), 0040 (@) и 0060(').

Пример:

```
wchar_t value\u00F6\u00F8 = '\u00f6';
```



Освен ако не е абсолютно необходимо, избягвайте използването на универсални символи. Използването им прави кода нечетлив и не всички компилатори поддържат универсални символи. Това влошава качеството на кода и намалява преносимостта на програмата.

Повече за стандарта ISO/IEC-10646 ще намерите в Приложение В.

22.3.4 Широки и многобайтови символи

Широките и многобайтовите символи са два различни подхода за представяне на символи от разширената символна таблица.

ANSI C стандартът дефинира типовете **wchar_t** и **wint_t** за работа с широки символи. Тези типове не са вградени в езика, а са дефинирани в **stddef.h**. Типът **wchar_t** е дефиниран като един от целочислените типове, достатъчно голям да представи всички широки символи, поддържани от имплементацията. Типът **wint_t** е дефиниран като един от целочислените типове, достатъчно голям да побере всички стойности на **wchar_t** и стойността на макроса **WEOF** (константа от тип **wint_t**, чиято стойност се различава от всички символи на разширената символна таблица). Различни функции могат да върнат като резултата **WEOF**, за да укажат, че е достигнат края на файла или е настъпила грешка. Широките символи изпозват фиксиран брой битове (типично 16 или 32).

Многобайтовите символи се състоят от един или няколко байта. Те могат да се използват в символни константи, низови константи, коментари, идентификатори и имена на

хедър-файлове.

Методите на кодиране на широките и многобайтовите символи зависи от имплементацията. Повечето имплементации използват форматите UTF-16 или UTF-32 за кодиране на широки символи и UTF-8 за кодиране на многобайтови символи.

Хедър-файловете **wchar.h** и **wctype.h** съдържат средства за работа с широки и многобайтови символи.

22.4 Лексеми

Минималният лексикален елемент, до който една C програма се разбива по време на компилиране, се нарича лексема (**token**). Лексемите, от които се състои една C програма, са:

Лексема:

- Ключова дума
- Идентификатор
- Константа
- Операция^{C89}
- Пунктуатор

Предпроцесорна лексема:

- Име на хедър-файл
- Идентификатор
- Предпроцесорно число
- Символна константа
- Низова константа
- Операция^{C89}
- Пунктуатор

Всяка предпроцесорна лексема се преобращува в лексема в процеса на компилация.

22.5 Ключови думи

Следващата таблица описва ключовите думи в C.

auto	enum	restrict ^{C99}	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool ^{C99}
continue	if	static	_Complex ^{C99}
default	inline ^{C99}	struct	_Imaginary ^{C99}
do	int	switch	-
double	long	typedef	-
else	register	union	-

Табл. 85 Ключови думи

22.6 Идентификатор

Един идентификатор може да обозначава една от следните единици в езика C:

- име на променлива;
- име на функция;
- етикет на структура, обединение , изброяване (tag);
- етикет на оператор (label);
- член на на структура, обединение или изброяване;

- **typedef** име;
- **макрос**;
- **параметър на макрос**.

Идентификаторът се състои от:

- Латински символи;

```

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V X Y Z

```

- Арабски цифри;

```

0 1 2 3 4 5 6 7 8 9

```

- Подчертаващо тире;

```

_

```

- Универсални символи^{C99};
- Многобайтови символи^{C99}.

Първият символ на идентификатора не може да бъде цифра или универсален символ, представляващ цифра.

Пример:

```

sum1 – валиден идентификатор
_sum – валиден идентификатор
1sum – НЕвалиден идентификатор

```

Езикът C прави разлика между малки и големи букви. Това

означава, че `sum1`, `Sum1`, `SUM1`, `sUm1` и т.н. се възприемат като различни идентификатори от компилатора.

Идентификаторите, обозначаващи различни C единици, могат да съвпадат стига да се намират в различни категории (именовани пространства). Именованите пространства в езика C са:

- етикети на оператори;
- етикети на структури, обединения и изброявания;
- членове на структури и изброявания;
- всички останали (имена на променливи, функции, typedef имена и enum-константи).

Пример:

```
int sum; /* sum е идентификатор на променлива */

int sum_xy (int x, int y); /* sum_xy е идентификатор на функция */

struct mystruct /* mystruct е идентификатор на етикет(име) на
{               структура */
    int a; /* a е идентификатор на член на структура */
    int b; /* b е идентификатор на член на структура */
};
union myunion /* myunion е идентификатор на етикет(име) на
{             обединение */
    int i; /* i е идентификатор на член на обединение */
    double d; /* d е идентификатор на член на обединение */
};
enum myenum /* myenum е идентификатор на етикет(име) на
{           изброяване */
    I, /* I е идентификатор на член на изброяване */
    You, /* You е идентификатор на член на изброяване */
    He, /* He е идентификатор на член на изброяване */
    She /* She е идентификатор на член на изброяване */
};

typedef unsigned char uint8; /* uint8 е идентификатор на typedef
име */
```

```

addition: /* addition е идентификатор на етикет на оператор */
sum = sum_xy (10, 20);

#define message "Hello C!\n" /* message е идентификатор на макрос
*/

#define print_message(m) printf("%s\n", m); /* m е идентификатор
на параметър на макрос */

```

22.7 Константи

Константите са разгледани в [24 Константи](#).

22.8 Операции и Пунктуатори

C89 включва следните символи за операции:

[]	()	.	->						
++	--	&	*	+	-	~	!	sizeof	
/	%	<<	>>	<	>	<=	>=	==	
!=	^		&&		? :	=	*=	/=	
%=	+=	-=	<<=	>>=	&=	^=	=	,	

Забележка: В заглавния-файл `<iso646.h>` са дефинирани макроси, които се разширяват до определени операции както е показано:

```

#define and      &&
#define and_eq  &=
#define bitand  &
#define bitor   |
#define compl   ~
#define not     !
#define not_eq  !=
#define or      ||
#define or_eq   |=
#define xor     ^
#define xor_eq  ^=

```

C89 включва следните символи за пунктуатори:

[] () { } * , : = ; ...

C99 обединява горните символи (без sizeof) плюс някои допълнителни символи под общото понятие пунктуатор:

[] () . ->
++ -- & * + - ~ !
/ % << >> < > <= >= ==
!= ^ | && || ? : = *= /=
%= += -= <<= >>= &= ^= |= ,

<: :> <% %> %: %:%:

22.9 Имена на хедър-файлове

<име-хедър-файл>

или

"име-хедър-файл"

Този вид предпроцесорна лексема може да се появява само в #include предпроцесорната директива. Имената на хедър-файлове могат да включват всички символи от сорс символната таблица без символ за нов ред и ". Допълнително са в сила следните ограничения:

- Ако символите ' , \ , " , //^{C99} или /* се появят между разделителите < и >, поведението е недефинирано.
- Ако символите ' , \ , //^{C99} или /* се появят между разделителите " и ", поведението е недефинирано.

22.10 Предпроцесорни числа

Едно предпроцесорно число започва с цифра, която по избор може да бъде предшествана от точка, и може да бъде последвана от букви, подчертаващи тирета, цифри, точки и последователностите $e+$, $e-$, $E+$, $E-$, $P+^{C99}$ или $P-^{C99}$. Предпроцесорните числа включват целочислени константи и константи с плаваща запетая плюс много други форми, които може да не са първоначално валидни числа. Предпроцесорното число няма тип или стойност. То може да ги "получи" след като се преобразува до число с плаваща запетая или целочислена константата след разгъване на макроса, в който участва.

Пример:

```
#define glue(a, b) a ## b  
  
float f = glue(1e+, 20); /* float f = 1e+20 */
```

22.11 Коментари

Коментарите представлява потребителски съобщения, които не влияят на поведението на програмата. Те са служат за добавяне на пояснения към кода. ANSI C стандартът дефинира два начина за добавяне на коментари в кода:

```
/* коментар */  
  
или  
  
// коментарC99
```

Всичко поставено между `/*` и `*/` се игнорира от компилатора. Този тип коментар се нарича още многоредов коментар, тъй като маркерите за начало `/*` и край `*/` на коментара могат да са разположени на различни редове. Това е единственият

коментар дефиниран в стандарта C89.

Пример:

```
/* Това е коментар */
```

```
/* Това е коментар  
   на няколко реда */
```

C99 добавя т.нар. едноредов коментар. Всичко след маркера за начало `//` на коментара до края на реда се игнорира от компилатора.

Пример:

```
// Това е коментар
```

Особености на коментарите:

- Коментарите не се разпознават като такива, ако са част от низови константи;
- По време на компилиране на кода всеки коментар се заменя с интервал;
- Вложени коментари не се поддържат¹.

Забележка¹: Много компилатори нарушават това изискване на ANSI C стандарта и поддържат вложени коментари.



Ако искате да изключите определена секция с код от процеса на компилация, използвайте `#if ... #endif`, вместо `/* ... */`. Например:

```
#if (0)  
...  
някакъв код  
...  
#endif
```

23 Типове данни

23.1 Променливи

Променливата е област от паметта на компютъра, която се използва за съхранение на данни. Както и самото понятие подсказва, в процеса на изпълнение на програмата, стойността на променливата може да се променя. В C една променлива трябва да бъде дефинирана (обявена) преди да бъде използвана някъде в кода. Общата форма на дефиниране на променлива е:

Дефиниране на променлива

клас-памет_{опц} квалификатор-тип_{опц} спецификатор-тип_{опц} тип име-променлива = инициализатор_{опц};

където

клас-памет_{опц}¹ може да е:

`static`
`auto`
`register`

квалификатор-тип_{опц}¹ може да е:

`const`
`volatile`
`const volatile` или `volatile const`

спецификатор-тип_{опц}¹ се прилага само върху някои базови типове и може да е:

<code>signed</code>	<code>(int</code> или <code>char)</code>
<code>unsigned</code>	<code>(int</code> или <code>char)</code>
<code>short</code>	<code>(int)</code>
<code>long</code>	<code>(int)</code>
<code>signed short</code>	<code>(int)</code>
<code>signed long</code>	<code>(int)</code>
<code>signed long long</code> ^{C99}	<code>(int)</code>
<code>unsigned short</code>	<code>(int)</code>
<code>unsigned long</code>	<code>(int)</code>
<code>unsigned long long</code> ^{C99}	<code>(int)</code>

Забележка¹: Редът на следване на клас-памет, квалификатор-тип, спецификатор-тип и тип не е специфициран от стандарта.

тип

Може да бъде един от следните типове:

- Базов тип (`_BoolC99`, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури (`struct {...}`);
- `typedef`-име.

име-променлива

Име на променлива, зададено чрез идентификатор.

инициализатор_{опц}

Форматът на инициализатора зависи от типа и класа памет на променливата.

клас памет	тип	инициализатор	подразбиращ се инициализатор
static/ extern	скаларен	константен израз	0 / 0.0
static/ extern	масив	{списък-константни-изрази}	0 / 0.0
static/ extern	структура	{списък-константни-изрази}	0 / 0.0
static/ extern	обединение	константен израз	0 / 0.0
auto/ register	скаларен	всякакъв израз	недефиниран
auto/ register	масив	▪ {списък-константни-изрази} ▪ {списък-всякакви-изрази^{C99}}	недефиниран
auto/ register	структура	▪ {списък-константни-изрази} ▪ {списък-всякакви-изрази^{C99}} ▪ struct-променлива от същия тип	недефиниран
auto/ register	обединение	▪ {списък-константни-изрази} ▪ {списък-всякакви-изрази^{C99}} ▪ union-променлива от същия тип	недефиниран

Дефинирането на променлива води до заделяне на памет за нея. Типът на една променлива носи следната информация:

- видът на данните, които тази променлива може да съхранява (цели числа, символи и т.н.), а оттук и размера памет, необходим за заделяне за тази променлива.
- видовете операции, които могат да се прилагат върху тази променлива (допустимите операции върху променлива от един тип може да не са позволени за променлива от друг тип).
- разполагане на променливата в паметта или подравняване в паметта (**memory alignment**). Тази характеристика не е изискване на езика C, а на конкретната хардуерна платформа. Различните хардуерни платформи имат различни изисквания как данните да се разполагат в паметта. Всяка имплементация на C компилатора, трябва да се съобразява с това изискване.

23.2 Категории типове

В Табл.87 са показани категориите типове в езика C.

С типове	категории		
<code>_Bool</code> ^{C99}	Целочислени типове	Аритметични типове	Скаларни типове
<code>char</code> (signed и unsigned)			
<code>short, int, long, long long</code> ^{C99} (signed и unsigned)			
<code>enum T{...}</code>			
<code>float, double, long double</code>	Типове с плаваща запетая		
<code>T*</code>	Указатели		
<code>T [...]</code>	Масиви		Агрегатни

struct T{...}	Структури	типове
union T{...}	Обединения	
T (...)	Функции	
void	тип без стойност	

Табл. 86 Категории типове данни в С

- Целочислени типове;

Както подсказва името, тези типове данни могат да представят само цели числа. За по-ефикасно използване на паметта езикът С включва няколко целочислени типа, всеки от които може да представи различен диапазон от стойности.

- Типове с плаваща запетая;

Тези типове се използват за представяне на реални числа (дробни числа). За по-ефикасно използване на паметта езикът С включва няколко типа с плаваща запетая, всеки от които може да представи различен диапазон от стойности.

Сумарно целочислените типове и типовете с плаваща запетая се наричат **аритметични типове**. Аритметичните типове, без enum-типовете, се наричат още **базови типове**. Това са типовете на базата, на които се получават останалите типове данни в С. Базовите типове са разгледани в [23.2.1 Базови типове данни](#).

- Изброявания;

Изброяването е съвкупност от целочислени константи, зададени чрез идентификатори. Изброяванията са описани в [23.2.2 Изброявания](#).

- Масиви;

Масивът описва съвкупност от променливи от един и същ тип и с общо име. Променливите принадлежащи на даден масив се наричат елементи на масива. Масивите са описани в [23.2.3 Масиви](#).

- Указатели;

Един указателен тип описва променлива, чиято стойност представлява адрес на друга променлива. Указателните типове са описани в [23.2.4 Указатели](#).

Сумарно аритметичните и указателните типове се наричат **скаларни типове**.

- Структури;

Структурата представлява съвкупност от променливи, които могат да са от различен тип. Променливите, принадлежащи на една структура, се наричат членове на структурата. Структурите са описани в [23.2.5 Структури](#).

Сумарно структурите и масивите се наричат **агрегатни типове**.

- Обединения;

Обединението представлява съвкупност от припокриващи се (частично или напълно) променливи, които могат да са от различен тип. Припокриващи се променливи означава, че те използват обща памет, която е достатъчно голяма да побере променливата с най-голям размер. Променливите, принадлежащи на едно обединение, се наричат членове на обединението. Обединенията са описани в [23.2.6 Обединения](#).

- Функции;

Функцията е група от оператори, които се изпълняват при извикването ѝ в някоя точка от програмата. Функциите могат да получават данни при извикването си и могат да връщат резултат. Типът на една функция се определя от типа на връщания резултат, броя и типа на параметрите на функцията. Функциите са описани в [28 Функции](#).

- тип void.

Тип void е непълен (незавършен) тип, който представлява тип без стойност. Той се използва за дефиниране на:

1. Обобщени указатели (указатели, които могат да сочат към всякакви типове данни) (вижте [23.2.4 Указатели](#)).
2. Функции, които не връщат стойност (вижте [28 Функции](#)).
3. Функции, които не приемат аргументи (вижте [28 Функции](#)).

Най-общо всички гореописани категории типове се разделят на 3 групи:

- **Обектни типове** – това са типове, описващи обекти (променливи);
- **Функционални типове** – това са типове, описващи функции;
- **Непълни типове** – това са типове, които описват обекти, но липсва информация необходима за определяне на техния размер. Непълен е тип void, също могат да се декларират непълни типове за масиви, структури и обединения. Всички непълни типове (без void) могат да станат пълни (завършени). Един пълен тип носи цялата информация, описваща този тип.

23.2.1 Базови типове данни

Езикът С дефинира няколко базови (вградени в самия език) типове данни (Табл.87). Базовите типове могат да се модифицират с помощта на т.нар. модификатори на типа (Табл.88).

тип	описание	пример
_Bool ^{C99}	Беззнаков целочислен тип, който може да съдържа само две стойности: 0 и 1. За удобство хедър-файлът <stdbool.h> дефинира макроса bool , който е синоним на _Bool , и макросите false и true , които имат стойност 0 и 1 съответно.	0 или 1
char	Използва се за дефиниране на променливи, които могат да съхраняват символи. Символите в С се затварят в единични кавички. Тип char заема 8 бита, и въпреки че се използва за съхраняване на символи, може да се използва и за съхраняване на цели 8-битови числа.	'z','s','#', 10, 38,127
int	Използва се за дефиниране на променливи, които могат да съхраняват цели числа със знак.	2330, -1325, 1000
float	Използва се за дефиниране на променливи, които могат да съхраняват дробни числа.	2.5, -0.123
double	Същото като float с тази разлика, че може да съхранява по-големи/малки числа и почти двойно по-голям брой значещи цифри.	125464.986537, -0.0000004574
void	Тип без стойност. Ключовата дума void има по-особено предназначение. Например не може да дефинирате променлива от този тип както с останалите базови типове. Ключовата дума void се използва за: ▪ Дефиниране на функции, които не връщат стойност; ▪ Дефиниране на функции без параметри; ▪ Дефиниране на обобщени указатели.	void foo(void); void *p;

Табл. 87 Базови типове данни

модификатор	описание	пример
unsigned	Може да се прилага към базовите типове char и int, за да укаже че тези типове могат да приемат само неотрицателни стойности.	unsigned char unsigned int

signed	Може да се прилага към базовите типове <code>char</code> и <code>int</code> , за да укаже, че тези типове могат да приемат знакови числа, т.е. . и положителни и отрицателни.	signed char signed int
long	Може да се прилага към базовите типове <code>double</code> и <code>int</code> , за да увеличи големината на стойностите, които тези типове могат да съхраняват. Този модификатор може да се използва в комбинация с модификаторите <code>unsigned</code> и <code>signed</code> към тип <code>int</code> .	long int unsigned long int signed long int long double
long long ^{C99}	Може да се прилага към базовия тип <code>int</code> , за да увеличи големината на стойностите, които този тип може да съхранява. Този модификатор може да се използва в комбинация с модификаторите <code>unsigned</code> и <code>signed</code> към тип <code>int</code> .	long long int unsigned long long int signed long long int
short	Може да се прилага към базовия тип <code>int</code> , за да намали големината на стойностите, които тези типове могат да съхраняват. Този модификатор може да се използва в комбинация с модификаторите <code>unsigned</code> и <code>signed</code> към тип <code>int</code> .	short int unsigned short int signed short int

Табл. 88 Модификатори на базовите типове

Не е задължително да укажете всички тези ключови думи, когато дефинирате променливи. Следните конструкции са еквивалентни:

short $\approx$ **signed short** $\approx$ **short int** $\approx$ **signed short int**

unsigned short $\approx$ **unsigned short int**

signed $\approx$ **signed int** $\approx$ **int**

unsigned $\approx$ **unsigned int**

long $\approx$ **long int** $\approx$ **signed long** $\approx$ **signed long int**

unsigned long $\approx$ **unsigned long int**

long long $\approx$ **long long int** $\approx$ **signed long long** $\approx$ **signed long long int**

unsigned long long $\approx$ **unsigned long long int**

Размерът (обемът заемана памет) и диапазонът от стойности на базовите типове данни е различен за различните процесори и компилатори. Стандартът C дефинира само минималните размери и стойности (Табл.89).

тип	минимален размер в байтове	числов диапазон	коментар
<code>_Bool</code> ^{C99}	1	0 или 1	За удобство хедър-файлът <code><stdbool.h></code> дефинира макроса <code>bool</code> , който е синоним на <code>_Bool</code> , и макросите <code>false</code> и <code>true</code> , които имат стойност 0 и 1 съответно.
<code>char</code>	1	-127 ÷ +127 или 0 ÷ 255	Стандартът не дефинира дали по подразбиране тип <code>char</code> е знаков или беззнаков. Това зависи от имплементацията на компилатора.
<code>signed char</code> ²	1	-127 ÷ +127	
<code>unsigned char</code>	1	0 ÷ 255	
<code>signed int</code> ^{1,2}	2	-32767 ÷ +32767	По-малък или равен на <code>long int</code>
<code>unsigned int</code>	2	0 ÷ 65535	
<code>signed short int</code> ²	2	-32767 ÷ +32767	По-малък или равен на <code>int</code>
<code>unsigned short int</code>	2	0 ÷ 65535	
<code>signed long int</code> ²	4	-2 147 483 647 ÷ +2 147 483 647	По-голям или равен на <code>int</code>
<code>unsigned long int</code>	4	0 ÷ +4 294 967 295	
<code>signed long long int</code> ^{C99,2}	8	-9 223 372 036 854 775 808 ÷ +9 223 372 036 854 775 807	
<code>unsigned long long int</code> ^{C99}	8	0 ÷ 18 446 744 073 709 551 615	
<code>float</code> ³	4	±1E-37 ÷ ±1E+37	По-малък или равен на <code>double</code>
<code>double</code> ³	4	±1E-37 ÷ ±1E+37	По-малък или равен на <code>long double</code>
<code>long double</code> ³	4	±1E-37 ÷ ±1E+37	По-голям или равен на <code>double</code>

Табл. 89 Минимални размери на базовите типове съгласно стандарта C

Забележка¹: Размерът на тип `int` типично съвпада с размера на машинната дума на компютъра, т.е. . за 16-битов компютър тип `int` е

16 бита, за 32-битов компютър тип `int` е 32 бита и т.н.

Забележка²: Стандартът C не дефинира метода на кодиране на отрицателните стойности на знаковите целочислени типове. Повечето C компилатори използват допълнителен код (**2's complement**).

Забележка³: Стандартът C не дефинира метода на кодиране на числата с плаваща запетая. Повечето C компилатори използват стандарта **IEEE 754-1985** и неговия наследник **IEEE 754-2008**.

Пример:

```
_Bool
char
signed char
unsigned char
signed int
unsigned int
signed short int
unsigned short int
signed long int
unsigned long int
signed long long int

errorFlag = 1;
firstNameLetter = 'D';
temperature = -15;
age = 36;
```

23.2.2 Изброявания

Изброяването е потребителски тип, който представлява набор от целочислени константи (enum-константи), зададени чрез идентификатори. Изброяване се декларира по следния начин:

Деклариране на изброяване

```
клас-паметопц квалификатор-типопц enum име-изброяванеопц
{
    име-enum-константа1 = инициализаторопц,
    име-enum-константа2 = инициализаторопц,
    име-enum-константа3 = инициализаторопц,
    име-enum-константа4 = инициализаторопц,
    ...
    име-enum-константаN = инициализаторопц
} клас-паметопц квалификатор-типопц списък-enum-променливиопц;
```

където

клас-памет_{опц} (**static**, **auto**, **register** или **typedef**)

квалификатор-тип_{опц} (**const**, **volatile** или **const volatile**)

Тези атрибути могат да се поставят или пред декларацията на изброяването:

клас-памет_{опц} **квалификатор-тип**_{опц} **enum** име-изброяване_{опц} {...}
списък-enum-променливи_{опц};

или след нея, но пред **списък-enum-променливи**:

enum име-изброяване_{опц} {...} **клас-памет**_{опц} **квалификатор-тип**_{опц}
списък-enum-променливи_{опц};

Използването им в декларацията на изброяването има смисъл само ако декларацията съдържа **списък-enum-променливи**.

име-изброяване_{опц}

Етикет (име) на изброяването, зададен чрез идентификатор.

име-enum-константа₁, ..., **име-enum-константа**_N

Целочислени константи, зададени чрез идентификатори. Наричат се още членове на изброяването.

инициализатор_{опц}

Целочислен литерален константен израз от тип **int**. Инициализаторите могат да имат еднакви стойности.

списък-enum-променливи_{опц}

име-enum-променлива₁ = **инициализатор**_{опц},

...,

име-enum-променлива_N = **инициализатор**_{опц}

където

име-enum-променлива₁, ..., **име-enum-променлива**_N

Име на enum-променлива, зададено чрез идентификатор.

инициализатор_{опц}

Може да бъде **име-enum-константа**_n ($n = 1 \div N$) или всеки целочислен израз. Инициализаторите не са позволени, ако за клас-памет се използва **typedef**.

Инициализаторите на enum-константите не са задължителни. Ако липсват инициализатори, по подразбиране първият идентификатор приема стойност 0, следващият 1 и т.н., т.е. всяка enum-константа е с единица по-голяма от предходната.

Ако enum-константа не е изрично инициализирана, нейната стойност е с 1 по-голяма от предходната. Enum-константа може да се използва като инициализатор на enum-константите, разположени след нея.

Пример:

```
enum rgb
{
    RED,    /* Компиляторът назначава стойност 0 */
    GREEN,  /* Компиляторът назначава стойност 1 */
    BLUE    /* Компиляторът назначава стойност 2 */
};
```

```
enum toggle { OFF = 0, ON = 1, NO = 0, YES = 1 };
```

или

```
enum toggle { OFF = 0, ON = 1, NO = OFF, YES = ON };
```

Щом едно изброяване е декларирано, може да дефинирате променливи от този тип. Това може да стане по време на декларацията на изброяването или по късно в кода, както е показано по-долу:

```
клас-паметопц квалификатор-типопц enum име-изброяване  
списък-enum-променливи;
```

или

```
enum име-изброяване клас-паметопц квалификатор-типопц  
списък-enum-променливи;
```

Пример:

```
/* Дефиниране на enum-променлива по време на декларацията */
const enum rgb                                enum rgb
{
    RED,                                     {
    GREEN,                                   RED,
    BLUE                                    GREEN,
} r = RED;                                BLUE
} cosnt r = RED;
```

```
/* Дефиниране на enum-променлива по-късно в кода */
const enum rgb g = GREEN; или enum rgb const r = RED;
```

Типът на enum-променливите е имплементационно зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове `char` (signed, unsigned), `short` (signed, unsigned) или `int`.

Тъй като enum-променливите по същество са целочислени типове, то на тях могат да им се присвояват и целочислени стойности извън списъка с enum-константи, без да е необходимо явно преобразуване. Подобно смесване на стойности може да е объркващо и не се препоръчва. Освен това, тъй като типът на enum-променливите зависи от имплементацията, подобни практики могат да доведат до препълване.



Типът на enum-променливите е имплементационно зависим. В зависимост от стойностите на enum-константите компилаторът може да избере най-подходящия от целочислените типове `char` (signed, unsigned), `short` (signed, unsigned) или `int`.



Използвайте само enum-константи за присвояване на enum-променливи, в противен случай рискувате да създадете логическа грешка, ако типът на изброяването не може да представи целочислената стойност.

Пример:

```
enum rgb g = 1000; /*!!!Възможно е препълване. Типът enum rgb съдържа три константи, автоматично инициализирани от компилатора с 0, 1 и 2 съответно. Много е вероятно компилаторът да използва тип unsigned char за представяне на тези стойности, при което стойността, записана в g ще бъде различна от очакваната: g = 232*/
```

Името на изброяването не е задължително. Ако го пропуснете обаче, не можете да дефинирате променливи в кода от този тип (но може да дефинирате променливи по време на декларацията).

Пример:

```
enum { black, blue, green, cyan, red, magenta, white };
enum { black, blue, green, cyan, red, magenta, white } color;
```

Декларацията на едно изброяване е видима от точката на деклариране до края на файла или блока, в който се намира декларацията.

23.2.3 Масиви

Едномерни масиви

Масивът е съвкупност от променливи от един и същ тип и с общо име. Променливите на масива се наричат елементи и се разполагат последователно в паметта, като първият елемент на масива се разполага на най-младшия адрес, вторият елемент на следващия адрес и т.н. Масивите могат да бъдат **едномерни** и **многомерни**.

Дефиниране на едномерен масив

```
клас-паметопц квалификатор-типопц тип име-масив[размеропц] =  
инициализаторопц;
```

където

клас-памет_{опц}
`static`, `auto`, `register` или `typedef`

квалификатор-тип_{опц}
`const`, `volatile` или `const volatile`

име-масив
Име на масива, зададен чрез идентификатор.

тип
Дефинира типа на елементите на масива. Елементите на масив могат да бъдат от всеки тип с изключение на битови полета и функции, т.е.:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);

- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Структури (`struct {...}`);
- Указатели (`T*`);
- `typedef`-име.

размер_{опц}

Определя броя на елементите на масива. Трябва да бъде литерален константен целочислен израз. Ако дефиницията на масива включва и инициализация, размерът може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

инициализатор_{опц}

Изрази (инициализатори)¹, разделени със запетая и затворени във фигурни скоби. Броят на инициализаторите трябва да е по-малък или равен на броя на елементите на масива, указан с **размер**. Ако броят на инициализаторите е по-малък от броя на елементите, елементите, за които не достигат инициализатори, се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (за указателни типове).

Забележка¹: В C89 инициализаторите трябва да са литерални константни изрази, независимо от класа памет. В C99 инициализаторите на масивите с клас памет `auto` или `register` могат да са всякакви изрази.

Ако за тип се използва `char` (`signed` или `unsigned`), инициализаторът може да е низова константа. Размерът на масива трябва да е с единица по-голям от броя на символите в низовата константа. Ако размерът се пропусне, той се определя автоматично от броя на символите в низа. В това число влиза и нулевия символ.

Инициализаторите не са позволени, ако за клас-памет се използва `typedef`.

Пример:

```
int a[10];
```

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

или

```
int a[] = {1,2,3,4,5,6,7,8,9,10};
```

```
char bookName[] = "Programming microcontrollers in C language."
```

Многомерни масиви

Ако елементите на едномерен масив са едномерни масиви, се получава двумерен масив. Ако елементите на едномерен масив са двумерни масиви, се получава тримерен масив и т.н. Обобщено казано, ако елементите на едномерен масив са масиви, се получава многомерен масив.

Дефиниране на многомерен масив

клас-памет_{опц} квалификатор-тип_{опц}
тип име-масив[размер1_{опц}][размер2]...[размерN] = инициализатор_{опц};

където

клас-памет_{опц}
`static`, `auto`, `register` или `typedef`

квалификатор-тип_{опц}
`const`, `volatile` или `const volatile`

име-масив

Име на масива, зададен чрез идентификатор.

тип

Дефинира типа на елементи на (N-(N-1))-мерните масиви. Може да бъде всеки тип без битово поле и функции, т.е.:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Структури (`struct {...}`);
- Указатели (`T*`);
- `typedef`-име.

размер1_{опц}

Задава броя на елементите на N-мерния масив. Всеки елемент представлява масив с размерност N-1, т.е. . (N-1)-мерни масиви. Трябва да бъде литерален константен целочислен израз.

размер2

Задава броя на елементите на (N-1)-мерните масиви. Всеки елемент представлява масив с размерност N-2, т.е. . (N-2)-мерни масиви. Трябва да бъде литерален константен целочислен израз.

...

размерN

Задава броя на елементите в (N-(N-1))-мерните масиви. Трябва да бъде литерален константен целочислен израз.

инициализатор_{опц}

Изрази (инициализатори)¹, разделени със запетая и затворени във фигурни скоби. Броят на инициализаторите трябва да е по-малък или равен на броя на елементарните-елементите на масива, получен от произведението на (размер1*размер2...*размерN). Ако броят на инициализаторите е по-малък от броя на елементарните-елементи, то елементарните-елементите, за които не достигат инициализатори се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL(указателни типове).

Забележка¹: В C89 инициализаторите трябва да са литерални константни изрази, независимо от класа памет. В C99 инициализаторите на масивите с клас памет `auto` или `register` могат да са всякакви изрази.

Когато масив се инициализира по време на дефинирането си, **размер1** може да се пропусне. Броят на елементите се определя от броя на инициализаторите.

Инициализаторите не са позволени, ако за клас-памет се използва `typedef`.

Пример:

```
int a[3][4];/* Дефиниране на двумерен масив a, от 3 елемента тип
int[4] (едномерен масив с 4 елемента тип int) */
```

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

или

```
int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

или

```
int a[3][4] =
{
    1,2,3,4, /* Инициализиране на едномерния масив a[0] */
    5,6,7,8, /* Инициализиране на едномерния масив a[1] */
    9,10,11,12 /* Инициализиране на едномерния масив a[2] */
};
```

Допълнително около всеки ред с инициализиращи стойности могат да се поставят фигурни скоби.

```
int a[3][4] =
{
    {1,2,3,4}, /* Инициализиране на едномерния масив a[0] */
    {5,6,7,8}, /* Инициализиране на едномерния масив a[1] */
    {9,10,11,12} /* Инициализиране на едномерния масив a[2] */
};
```

Пример:

int b[2][3][4]; /* Дефиниране на тримерен масив b, който се състои от 2 елемента двумерни масиви. Всеки елемент-двумерен-масив се състои от 3 елемента едномерни масиви. Всеки елемент-едномерен-масив се състои от 4 елемента от тип int */

```
int a[2][3][4] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
```

или

```
int a[][3][4] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};
```

или

```
int a[2][3][4] =
{
    /* Инициализиране на двумерния масив a[0] */
    1,2,3,4, /* Инициализиране на едномерния масив a[0][0] */
    5,6,7,8, /* Инициализиране на едномерния масив a[0][1] */
    9,10,11,12 /* Инициализиране на едномерния масив a[0][2] */
},
/* Инициализиране на двумерния масив a[1] */
13,14,15,16, /* Инициализиране на едномерния масив a[1][0] */

```

```

    17,18,19,20, /* Инициализиране на едномерния масив a[1][1] */
    21,22,23,24 /* Инициализиране на едномерния масив a[1][2] */
}
};

или

int a[2][3][4] =
{
    { /* Инициализиране на двумерния масив a[0] */
        {1,2,3,4}, /* Инициализиране на едномерния масив a[0][0] */
        {5,6,7,8}, /* Инициализиране на едномерния масив a[0][1] */
        {9,10,11,12} /* Инициализиране на едномерния масив a[0][2] */
    },
    { /* Инициализиране на двумерния масив a[1] */
        {13,14,15,16}, /* Инициализиране на едномерния масив a[1][0] */
        {17,18,19,20}, /* Инициализиране на едномерния масив a[1][1] */
        {21,22,23,24} /* Инициализиране на едномерния масив a[1][2] */
    }
};

```

Операции с масиви

Индексиране

Индексирането е начин за достъп до елементите на масива.

Индексиране на едномерен масив

име-масив[индекс]

където

индекс¹

Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

$$0 \leq \text{индекс} < \text{размер}, \text{ т.е. } \text{индекс} \in [0 \dots (\text{размер}-1)]$$

Индексиране на многомерен масив

име-масив[индекс1][индекс2]...[индексN]

където

индекс1¹

Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

$$0 \leq \text{индекс1} < \text{размер1} , \text{ т.е. } . \text{ индекс1} \in [0 \dots (\text{размер1}-1)]$$

индекс2¹

Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

$$0 \leq \text{индекс2} < \text{размер2} , \text{ т.е. } . \text{ индекс2} \in [0 \dots (\text{размер2}-1)]$$

...

индексN¹

Може да бъде всеки целочислен израз, чиято стойност изпълнява условието

$$0 \leq \text{индексN} < \text{размерN} , \text{ т.е. } . \text{ индексN} \in [0 \dots (\text{размерN}-1)]$$

Забележка¹: Отговорност на програмистта е да следи индексите да не излизат извън рамките на масива, в противен случай резултатът е недефиниран.



Езикът С не прави проверка дали индексът излиза извън границите на масива. Отговорност на програмистта е да следи това да не се случва.



Винаги проверявайте дали стойността на индекса е в границите на масива преди да го индексирате.

Указателни операции

Указателите и масивите са тясно свързани в езика С (вижте [Връзка между указатели и масиви](#)). Тази свързаност позволява масив да се дерекференцира подобно на указател.

***(име-масив + индекс)**

Горната конструкция е еквивалентна на **име-масив[индекс]**.

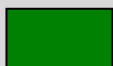
Името на масив в действителност, представлява адреса на първия елемент на масива, т.е. . името на масив е указател към първия елемент на масива. Всички операции, свързани с указатели са валидни и за масиви. Изключение са операциите присвояване на адрес, инкрементиране и декрементиране. Причината е, че името на масив е константен указател и не може да се променя да сочи към друго освен началото на масива. Компиляторът не заделя памет за името на масива, а го асоциира с адреса на първия елемент. Повече за указателните операции може да намерите в [Операции с указатели](#).



Името на масив е константен указател към първия елемент на масива, т.е. . ако a е име на масив, то следните конструкции са еквивалентни: $a \approx \&a[0]$.

Абстрактен изглед

Следващата фигура показва абстрактния изглед на едномерен, двумерен и тримерен масив.



Тип на елементите (елементарните-елементи):

- Базов тип (`_BoolC99`, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Структури (`struct {...}`);
- Указатели (`T*`);
- typedef-име.

Достъпът до елемент на многомерен масив става като името на масива се последва със съответния брой индекси в квадратни скоби. Индексите се разполагат, започвайки от

най-външния (погледнете двумерния и тримерния масив от Фиг.42). Например достъпът до елемента в червено се извършва чрез индексирание на масива по следния начин:

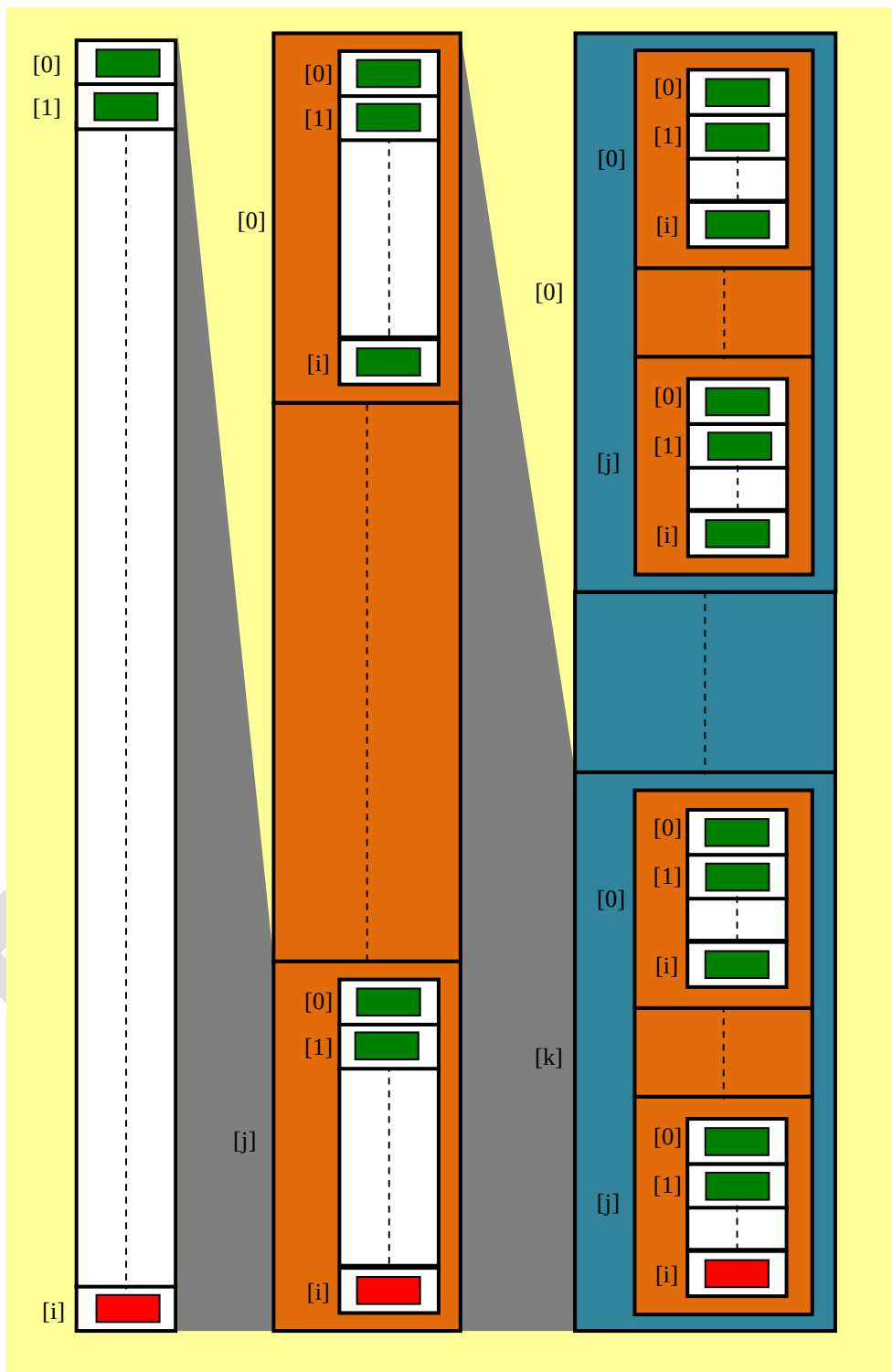
име-едномерен-масив[i]

име-двумерен-масив[j][i]

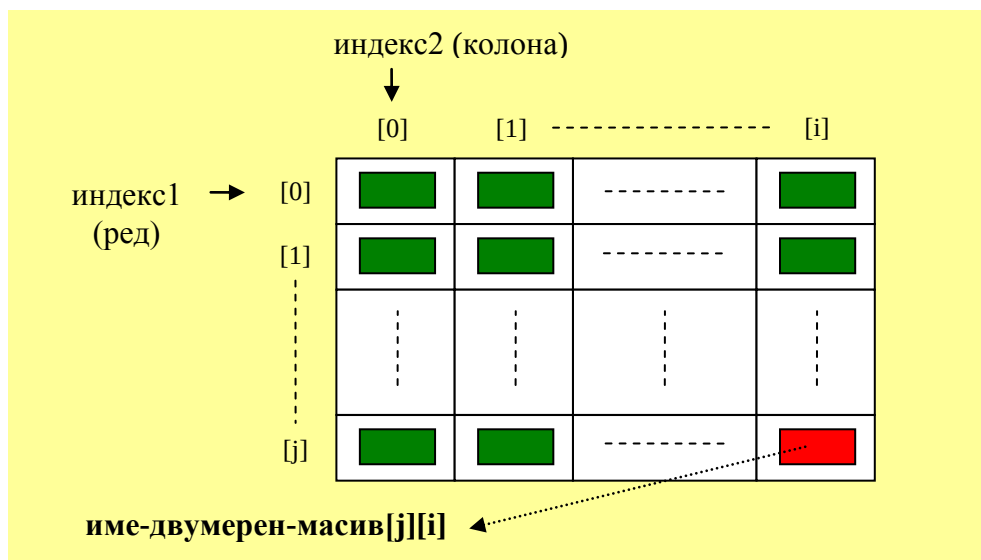
име-тримерен-масив[k][j][i]

Двумерният масив може да се представи още и като таблица. С **индекс1** се избира ред от таблицата, а с **индекс2** колона от таблицата – **име-двумерен-масив[ред][колона]** (Фиг.44).

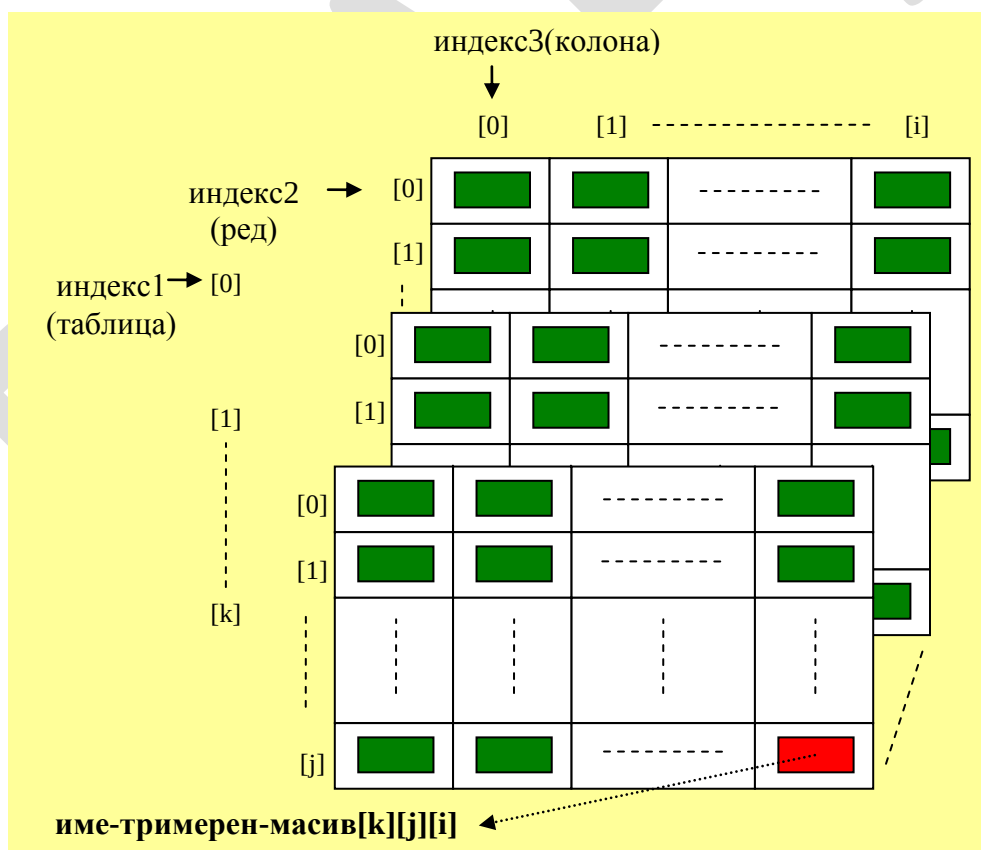
Тримерният масив може да се представи още и като съвкупност от таблици. С **индекс1** се избира таблица, с **индекс2** ред от таблицата, а с **индекс3** колона от таблицата – **име-тримерен-масив[таблица][ред][колона]** (Фиг.45).



Фиг. 43 Абстрактен изглед на масив - едномерен, двумерен и тримерен



Фиг. 44 Абстрактен изглед на двумерен масив -таблично представяне



Фиг. 45 Абстрактен изглед на тримерен масив - таблично представяне

23.2.4 Указатели

Указатели към обекти

Указателят към обект е променлива, която съдържа адрес на друга променлива.

Дефиниране на указател към обект

клас-памет_{опц} квалификатор-тип1_{опц} тип * квалификатор-тип2_{опц}
име-указател = инициализатор_{опц};

където

клас-памет_{опц}
`static`, `auto`, `register` или `typedef`

квалификатор-тип1_{опц} / квалификатор-тип2_{опц}
`const`, `volatile` или `const volatile`

име-указател

Име на указателя, зададено чрез идентификатор.

тип

Тип на обектите (променливите), към които указателя може да сочи. Може да бъде един от следните типове:

- Базов тип (`_Bool`^{c99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Структури (`struct {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- `void`¹;
- `typedef`-име.

Забележка¹: Указателите към тип `void` са по-специални и са разгледани в отделна точка (вижте [Обобщени Указатели](#)).

инициализатор_{опц}

Може да бъде всеки израз, който се свежда до адрес на променлива. В това число влизат:

- &име-променлива;
- име-масив;
- &име-масив[индекс];
- указател-към-обект;
- израз с указателна аритметика;
- функция, връщаща указател-към-обект като резултат;
- 0 (или NULL);
- Низова константа.

Инициализаторът не е позволен, ако за клас-памет се използва `typedef`.

Пример:

```
int    *pi; /* Указател към int */
enum   myenum *pe; /* Указател към enum myenum */
union  myunion *pu; /* Указател към union myunion */
struct mystruct *ps; /* Указател struct mystruct */
int (*pa)[10]; /* Указател към 1-мерен масив от 10 елемента int */
int** ppi; /* Указател към указател-към-int */
char *pstr = "Programming microcontrollers in C language."; /* Адресът на символа 'P' се присвоява на указателя pstr */
```

Указатели към функции

При дефинирането си функцията заема определен участък в паметта, където се разполага нейното тяло. В C е напълно допустимо да се дефинира указател към функция и да се инициализира с адреса на дадена функция. Адресът на функцията е началото на блока памет, където функцията се съхранява. Името на функцията в действителност представлява адреса на самата функция.

Дефиниране на указател към функция

клас-памет_{опц} тип (*квалификатор-тип_{опц} име-указател)(списък-параметри) = инициализатор_{опц};

където

клас-памет_{опц}
static, auto, register или **typedef**

квалификатор-тип_{опц}
const, volatile или **const volatile**

тип
Дефинира типа на резултата, връщан от функцията.

име-указател
Име на указателя, зададено чрез идентификатор.

списък-параметри
Дефинира списъка с параметри на функцията.

инициализатор_{опц}
Може да бъде всеки израз, който се свежда до адрес на функция. В това число влизат:

- &име-функция;
- име-функция¹;
- указател-към-функция;
- функция, връщаща указател-към-функция като резултат;
- 0 (или NULL²).

Забележка¹: Името на функция автоматично се преобразува в указател към функцията. По тази причина операцията & може да се изпусне пред името ѝ.

Забележка²: В контекста на указателите 0 или (void*)0 се нарича нулев константен указател. Езикът C дефинира специален макрос NULL в <stddef.h>, който представлява нулев константен указател.

Инициализаторът не е позволен, ако за клас-памет се използва **typedef**.

Пример:

```
void (*p)(int); /* Дефиниране на указател към функция, която не
връща резултат и има параметър тип int */
void (* const p)(int) = foo; /* Дефиниране на константен указател
към функция foo, която не връща резултат и има параметър тип int
*/
```

Обобщени указатели

Обобщеният указател е указател към тип `void`, затова се нарича още `void`-указател. `Void`-указател може да сочи към обекти от всякакви типове, но за да се достъпи (прочете или запише) обекта, `void`-указателят трябва да се преобразува в указател към типа на този обект, в противен случай компилаторът ще генерира грешка.

Дефиниране на void-указател

```
void* име-указател;
```

Пример:

```
int i; /* Дефиниране на променлива i от тип int */
double d; /* Дефиниране на променлива d от тип double */
void* vp; /* Дефиниране на void-указател vp */

vp = &i; /* vp сочи към i */
*((int*)vp) = 10; /* 10 → i */

vp = &d; /* vp сочи към d */
*((double*)vp) = 10.2; /* 10.2 → d */
```

Нулеви указатели

На всеки указател може да се присвои една специална стойност, наречена константен нулев указател `NULL`. Такъв указател се нарича нулев указател (**null-pointer**) и се счита, че не сочи към никакъв обект. В контекст, който изисква булев израз, стойността на нулевия указател се

интерпретира като ЛЪЖА (FALSE). Например следната конструкция

```
if (p) { i = *p;}
```

е еквивалентна на

```
if (p != NULL) { i = *p;}
```

Макросът NULL е дефиниран в хедър файла <stddef.h> както е показано по-долу.

```
#define NULL ((void*)0) или #define NULL (0)
```

Операции с указатели

Извличане на адрес

Извличане на адрес на променлива

&име-променлива

Тази конструкция връща като резултат адреса на променливата, към която е приложена операцията &.



Операцията за извличане на адрес & не може да се прилага върху register-променливи.

Пример:

```
int *p; /* Дефиниране на указател p към int променлива */
int var; /* Дефиниране на променлива var от тип int */
p = &var; /* Указателят p сочи към var */
```

Извличане на адрес на функция

&име-функция

Тази конструкция връща като резултат адреса на функцията, към която е приложена операцията &.

Пример:

```
void foo(void); /* Прототип на функцията foo */
void (*p)(void); /* Дефиниране на указател към функция, която не
връща резултат и няма параметри */
p = &foo; /* Инициализиране на указателя с адреса на foo */
p = foo; /* Същото като горната конструкция1 */
```

Забележка¹: Името на функция автоматично се преобразува в указател към функцията. По тази причина операцията & може да се изпусне пред името ѝ.

Присвояване на адрес

указател-към-обект = адрес-на-обект;

указател-към-функция = адрес-на-функция;

където

адрес-на-обект

Може да бъде всеки израз, който се свежда до адрес на променлива. В това число влизат:

- &име-променлива;
- име-масив;
- &име-масив[индекс];
- указател-към-обект;
- израз с указателна аритметика;
- функция, връщаща указател-към-обект като резултат
- $\emptyset$ (или NULL¹);
- Низова константа.

адрес-на-функция

Може да бъде всеки израз, който се свежда до адрес на функция. В това число влизат:

- &име-функция;
- име-функция¹;
- указател-към-функция;
- функция, връщаща указател-към-функция като резултат;
- 0 (или NULL²).

Забележка¹: Името на функция автоматично се преобразува в указател към функцията. По тази причина операцията & може да се изпусне пред името ѝ.

Забележка²: В контекста на указателите 0 или (void*)0 се нарича нулев константен указател. Езикът C дефинира специален макрос NULL в <stddef.h>, който представлява нулев константен указател.

Пример:

```
int* foo(void); /* Прототип на функцията foo */
int var; /* Дефиниране на променлива тип int */
int array[4]; /* Дефиниране на масив от 4 елемента тип int */

int* p1 = &var; /* &име-променлива */
int* p2 = array; /* име-масив */
int* p3 = &array[3]; /* &име-масив[индекс] */
int* p4 = p3; /* указател-към-обект */
int* p5 = p3 - 1; /* израз с указателна аритметика */
int* p6 = foo(); /* Функция, връщаща указател-към-обект
                  като резултат */
int* p7 = 0; /* 0 (или NULL) */
char* p8 = "Hello, World!"; /* Низова константа */
```

Пример:

```
int foo(void); /* Прототип на функцията foo */
int (*RetPtrToFoo(void))(void); /* Дефиниране на функция
RetPtrToFoo, без параметри и връщаща указател към функция, без
параметри, връщаща обект тип int */

int (*p1)(void) = &foo; /* &име-функция */
int (*p2)(void) = foo; /* име-функция */
```

```

int (*p3)(void) = p2;           /* указател-към-функция      */
int (*p4)(void) = RetPtrToFoo(); /* функция, връщаща указател-към-
                                функция като резултат      */
int (*p5)(void) = 0;           /* 0 (или NULL)           */

```

Дереференциране на указател

Дереференциране на указател към променлива

***име-указател-към-обект**

Тази конструкция връща като резултат обекта, сочен от указателят и може да стои както отляво така и отдясно на операцията за присвояване или да се използва в изрази. В първия случай променливата, сочена от указателя ще бъде записана, а в останалите ще бъде прочетена.

Пример:

```

int *p; /* Дефиниране на указател p към int променлива */
int var1; /* Дефиниране на променлива var1 от тип int */
int var2; /* Дефиниране на променлива var2 от тип int */
p = &var1; /* Указателят p сочи към var */
*p = 1000; /* На var1 се присвоява 1000 чрез указателя p */
var2 = *p; /* На var2 се присвоява стойността на var1 чрез
            указателя p */

```

Дереференциране на указател към функция

(*име-указател)(списък-аргументи)

или

име-указател(списък-аргументи)

Тези конструкции са еквивалентни и извършват извикване на функцията, сочена от указателя.



Когато функция се извиква чрез указател, за предпочитане е използването на формата **(*име-указател)(списък-аргументи)**, тъй като тя указва ясно, че извикването става чрез указател, а не чрез името на функцията. Това подобрява самодокументирането на кода.



Винаги проверявайте указател дали сочи към валиден адрес преди да го дереференцирате. В противен случай резултатът е недефиниран.

Пример:

```
void foo(void); /* Прототип на функцията foo */  
void (*p)(void) = foo; /* Дефиниране на указател към функция без  
параметри и невръщаща резултат, и инициализиране на указателя с  
адреса на функцията foo */
```

```
(*p)(); /* Извикване на функцията foo чрез указателя p */  
p(); /* Извикване на функцията foo чрез указателя p */
```

Събиране и изваждане на указател с цяло число

име-указател + цяло-число

име-указател - цяло-число

където

цяло-число

Може да бъде всеки целочислен израз.

При събиране/изваждане на указател с цяло число, указателят се увеличава/намалява, така че да сочи към следващия/предходния обект. С други думи казано при увеличение/намаление на указател с цяло-число, указателят се увеличава/намалява с цяло-число `*sizeof(тип-обект)` байта, т.е.. изразът `име-указател±цяло-число` се преобразува от компилатора в `име-указател±цяло-число*(sizeof(тип-обект))`.



Операциите събиране и изваждане на указател с цяло число имат смисъл само когато указателят сочи към обект от агрегатен тип (масив или структура).



Операциите събиране и изваждане на указател с цяло число не могат да се прилагат върху указатели към функции.

Пример:

```
int a[4]; /* Дефиниране на масив от 4 елемента тип int */
int* p = a; /* p сочи към първия елемент на масива a */
*(p + 2) = 10; /* 10 → a[2] */
```

Инкрементиране и декрементиране на указател

име-указател++ или ++име-указател

име-указател-- или --име-указател

Тези операции увеличават/намаляват указателя с 1¹.

Забележка¹: Указателят се увеличава/намалява с един обект, а не с един байт, т.е.. с `1*sizeof(тип-обект)` байта.

Операциите ++ и -- могат да се използват и в комбинация с операцията дереференциране на указател. Следващата таблица описва всички комбинации с кратко описание на ефекта от прилагането им върху указател.

$*p++ \approx *(p++)$ $*p-- \approx *(p--)$	Първо се извлича стойността на указателя (която е адрес). Операцията * се прилага върху тази стойност и след това указателят се увеличава/намалява.
$(*p)++$ $(*p)--$	Първо се извлича обектът, сочен от указателя, след което стойността на обекта се увеличава/намалява.
$++p \approx *(++p)$ $--p \approx * (--p)$	Първо указателят се увеличава/намалява, след това се извлича обектът, сочен от указателя.
$++*p \approx ++(*p)$ $--*p \approx --(*p)$	Първо се извлича обектът, сочен от указателя, след което стойността на обекта се увеличава/намалява.

Табл. 90 Операциите ++/-- и * върху указатели



Операциите инкрементиране и декрементиране на указател не могат да се прилагат върху указатели към функции.

Изваждане на указатели

име-указател1 - име-указател2

където

име-указател1, име-указател2

Трябва да са указатели към един и същ тип.

Изваждането на указатели дава като резултат броя на обектите между двата указателя.



Изваждането на указатели има смисъл само ако двата указателя сочат към общ обект от агрегатен тип (масив или структура).

Пример:

```
int a[4]; /* Дефиниране на масив от 4 елемента тип int */
int* p1 = &a[0]; /* p1 сочи към първия елемент на масива a */
int* p2 = &a[3]; /* p2 сочи към последния елемент на масива a */
int diff = p2 - p1; /* 3 → diff */
```

Сравнение на указател с цяло число

име-указател == 0

име-указател != 0

Единственото цяло число, с което указател може да се сравнява и което може директно да му се присвоява, е 0. В контекста на указателите 0 или (void*)0 се нарича нулев константен указател. Езикът C дефинира специален макрос **NULL** в `<stddef.h>`, който представлява нулев константен указател (вижте [Нулеви указатели](#)). Нулевият константен

указател автоматично се преобразува в типа на другия указател при сравнението.



Ако дефинирате указател без да го инициализирате с адрес, добра практика е да го инициализирате с NULL. Така по-късно в кода може да го проверите дали сочи към валиден адрес или не.

Пример:

```
#include <stddef.h>

int *p = NULL; /* Дефиниране на указател p към int променлива и */
               /* и инициализиране с NULL – нулев указател      */
...

if(p != NULL) /* Проверка за нулев указател */
{
}
else
{
    /* Указателят е невалиден */
}
```

Сравнение на указатели

```
име-указател1 == име-указател2
име-указател1 != име-указател2
име-указател1 >  име-указател2
име-указател1 >= име-указател2
име-указател1 <  име-указател2
име-указател1 <= име-указател2
```



Сравняването на указатели има смисъл само ако двата указателя сочат към общ обект от агрегатен тип (масив или структура).

Пример:

```
int a[4];      /* Дефиниране на масив от 4 елемента тип int */
int *p1, *p2; /* Дефиниране на указатели p1 и p2 към тип int */
p1 = &a[0];    /* p1 сочи към първия елемент на масива a */
p2 = &a[3];    /* p2 сочи към последния елемент на масива a */

if(p1 < p2)
{
    /*...*/
}
else
{
    /*...*/
}
```

Индексиране на указатели

Указателите и масивите са тясно свързани в езика C (вижте [Връзка между указатели и масиви](#)). Тази свързаност позволява указател да се индексира подобно на масив.

име-указател[индекс]

където

индекс

Може да бъде всеки целочислен израз.

Горната конструкция автоматично се преобразува от компилатора в следния израз: $*(\text{име-указател} + \text{индекс})$. Това означава, че ако имаме масив **a** и указател **p**, който сочи към него, елементите на масива могат да се достигат по следните начини:

- Индексиране на името на масива - **a[i]**;
- Индексиране на указателя - **p[i]**;
- Указателна аритметика, приложена към името на масива - $*(a + i)^1$;

Забележка¹: Името на масива е константен указател към първия елемент. По тази причина конструкцията $*(a - i)$ е неприложима, тъй като със сигурност ще доведе до излизане извън долната граница на масива.

- Указателна аритметика, приложена към указателя - $*(p \pm i)$.

Пример:

```
a[0]      = 10; /* 10 → a[0] */
*(a + 1)  = 20; /* 20 → a[1] */
p[2]      = 30; /* 30 → a[2] */
*(p + 3)  = 40; /* 40 → a[3] */
```



Индексирането на указател има смисъл само ако указателят сочи към масив.

Връзка между указатели и масиви

Името на масив в действителност, представлява адреса на първия елемент на масива, т.е. . името на масив е указател към първия елемент на масива. Всички операции, свързани с указатели са валидни и за масиви. Изключение са операциите присвояване, инкрементиране и декрементиране. Причината е, че името на масив е константен указател и не може да се променя да сочи към друго освен началото на масива. Компиляторът не заделя памет за името на масива, а го асоциира с адреса на първия елемент.



Името на масив е константен указател към първия елемент на масива, т.е. . ако a е име на масив, то следните две конструкции са еквивалентни: $a \approx \&a[0]$.

Как компилаторът интерпретира адреса, изразен чрез името на масив или операцията `&`, приложена към име на масив, зависи от това дали масивът е едномерен или многомерен.

```
тип име-1мерен-масив[p]  
тип име-2мерен-масив[p1][p2]  
тип име-3мерен-масив[p1][p2][p3]
```

```
име-1мерен-масив → явява се адрес на обект тип тип  
&име-1мерен-масив → явява се адрес на обект тип тип(*)[p]  
име-2мерен-масив → явява се адрес на обект тип тип(*)[p2]  
&име-2мерен-масив → явява се адрес на обект тип тип(*)[p1][p2]  
име-3мерен-масив → явява се адрес на обект тип тип(*)[p1][p2]  
&име-3мерен-масив → явява се адрес на обект тип тип(*)[p1][p2][p3]
```

В C всяка конструкция от вида `x[i]` се преобразува от компилатора в конструкция от вида `*(x + i)`, т.е. `x[i] ≈ *(x + i)`, където `x` е име на масив или указател, а `i` е индекс. Това означава, че указателите могат да се индексират подобно на масивите (вижте [Индексиране на указатели](#)).



В C всяка конструкция от вида `x[i]` се преобразува от компилатора в конструкция от вида `*(x + i)`, т.е. `x[i] ≈ *(x + i)`, където `x` е име на масив или указател, а `i` е индекс.



Индексирането на указател има смисъл само ако указателят сочи към масив.

Пример:

```
int a[4]; /* Дефиниране на масив от 4 елемента тип int */  
int *p = a; /* p сочи към първия елемент на масива a */  
p[0] = 2; /* 2 → a[0] */
```

23.2.5 Структури

Деклариране на структура

Структурата е потребителски тип данни, който позволява променливи от различен тип, но свързани логически, да се обединят в едно цяло. Променливите, принадлежащи на една структура, се наричат членове или полета. Декларирането на структура става по следния начин:

Деклариране на структура

```

клас-паметопц квалификатор-типопц struct име-структура
{
    квалификатор-членопц тип1 име-член1;
    квалификатор-членопц тип2 име-член2;
    ...
    квалификатор-членопц типN име-членN;
} клас-паметопц квалификатор-типопц списък-struct-променливиопц

```

където

клас-памет_{опц} (**static**, **auto**, **register** или **typedef**)

квалификатор-тип_{опц} (**const**, **volatile** или **const volatile**)

Тези атрибути могат да се поставят или пред декларацията на структурата:

```

клас-паметопц квалификатор-типопц struct име-структураопц {...}
списък-struct-променливиопц;

```

или след нея, но пред **списък-struct-променливи**:

```

struct име-структураопц {...} клас-паметопц квалификатор-типопц
списък-struct-променливиопц;

```

Използването им в декларацията на структурата има смисъл само ако декларацията съдържа **списък-struct-променливи**.

име-структура_{опц}

Етикет (име) на структурата, зададен чрез идентификатор.

квалификатор-член_{опц}

Същото като **квалификатор-тип**.

тип1, тип2, ..., типN

Може да бъде скаларен тип, агрегатен тип, обединение, т.е.:

- Базови типове (**_Bool**^{C99}, **char**, **int**, **float**, **double**);
- Изброявания (**enum** {...});

- Обединения (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури¹ (`struct {...}`);
- `typedef`-име.

Забележка¹: Типът на член на структурата не може да бъде структурата, към която принадлежи члена, но може да бъде указател към нея.

име-член1, име-член2..., име-членN

Име на член, зададено чрез идентификатор.

списък-struct-променливи_{опц}

име-struct-променлива1 = инициализатор_{опц},

...,

име-struct-променливаN = инициализатор_{опц}

където

име-struct-променлива1, ..., име-struct-променливаN

Име на struct-променлива, зададено чрез идентификатор.

инициализатор_{опц}

Изрази (инициализатори)¹, разделени със запетая и затворени във фигурни скоби. Броят на инициализаторите трябва да е по-малък или равен на броя на членовете на структурата. Ако броят на инициализаторите е по-малък от броя на членовете, членовете, за които не достигат инициализатори, се инициализират с 0 (за целочислени типове), 0.0 (за типове с плаваща запетая) или NULL (за указателни типове).

Забележка¹: В C89 инициализаторите трябва да са литерални константни изрази, независимо от класа памет. В C99 инициализаторите на структурни променливи с клас памет `auto` или `register` могат да са всякакви изрази.

Инициализаторите не са позволени, ако за клас-памет се използва `typedef`.

Пример:

```
struct StrType /* Деклариране на структура */
{
    const int i;
    long double ld;
    enum EnumType e;
    union UnionType u;
    char a[20];
    unsigned long * ptr;
    struct AnotherStructType s;
    unsigned BitField:8;1
};
```

Щом една структура е декларирана, може да дефинирате променливи от този тип. Това може да стане по време на декларацията на структурата или по късно в кода, както е показано по-долу:

клас-памят_{опц} квалификатор-тип_{опц} **struct** име-структура
списък-struct-променливи;

или

struct име-структура клас-памят_{опц} квалификатор-тип_{опц}
списък-struct-променливи;

Пример:

```
/* Дефиниране на struct-променлива по време на декларацията */
struct StrType
{
    const int i;
    long double ld;
    enum EnumType e;
    union UnionType u;
    char a[20];
    unsigned long * ptr;
    struct AnotherStructType s;
    unsigned BitField:8;1
} volatile s1 = { /*i*/10 };

/* Дефиниране на struct-променлива по-късно в кода */
struct StrType s2 = { /*i*/20 };
```

Забележка¹: Структурите могат да имат членове, наречени

битови полета. Битовите полета имат по-специални свойства и са разгледани в [23.2.7 Битови полета](#).



Декларацията на структура създава нов тип данни. Компиляторът заделя памет за членовете, само когато се дефинира променлива от тази структура.



Членовете на структурата не могат да се инициализират в тялото на структурата. Те могат да се инициализират само при дефиниране на променлива от тази структура.

Името на структурата не е задължително. Ако го пропуснете обаче, не можете да дефинирате променливи в кода от този тип (но може да дефинирате променливи по време на декларацията).

Пример:

```
struct /* Деклариране на структура без име*/  
{  
    const int i;  
    long double ld;  
    enum EnumType e;  
    union UnionType u;  
    char a[20];  
    unsigned long * ptr;  
    struct AnotherStructType s;  
    unsigned BitField:8;1  
} s3 = { /*i*/30};
```

Декларацията на една структура е видима от точката на деклариране до края на файла или блока, в който се намира декларацията.

Операции със структури

Директен достъп до член на структурна променлива

Директният достъпът до членовете на една структурна променлива става с помощта на името на структурната

променлива последвано от точка и името на члена.

Директен достъп до член на структурна променлива

име-структурна-променлива.име-член

Пример:

```
struct Cat
{
    char * name;
    char * color;
}myCat = {"Winnetou", "ginger-haired\n"};

printf("The name of my cat is %s\n", myCat.name);
printf("He is %s\n %s\n", myCat.color);
```

Косвен достъп до член на структурна променлива

Косвеният достъп до членовете на структурна променлива става с помощта на указател към структурата.

Дефиниране на указател към структура

struct име-структура * име-указател;

Достъпът до член на структурна променлива чрез указател към нея става с помощта на операцията ->.

Косвен достъп до член на структурна променлива

име-указател -> име-член

Пример:

```
struct Cat
{
    char * name;
    char * color;
}myCat = {"Winnetou", "ginger-haired"};

struct Cat * ptr;

ptr = &myCat; /* ptr сочи към myCat */
```

```
printf("The name of my cat is %s\n", ptr->name);  
printf("He is %s\n%s\n", ptr->color);
```

Присвояване на структури

Една структурна променлива може да се присвои директно на друга структурна променлива. Единственото условие, което трябва да е изпълнено е двете структурни променливи да са от един и същ тип.

Пример:

```
struct Cat  
{  
    char * name;  
    char * color;  
}myCat = {"Winnetou", "ginger-haired"}, yourCat;  
  
yourCat = myCat; /* myCat → yourCat */  
  
printf("The name of your cat is %s\n", yourCat.name);  
printf("He is %s\n%s\n", yourCat.color);
```

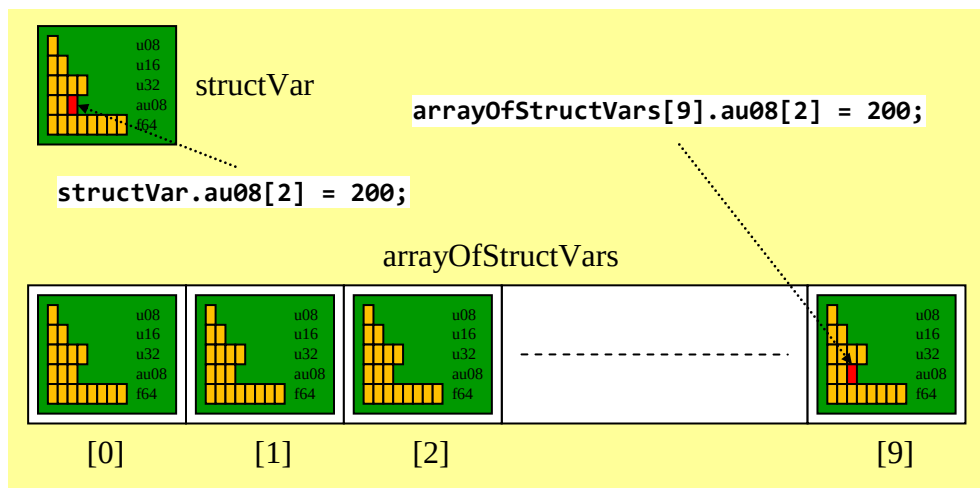
Абстрактен изглед

Нека допуснем, че имаме следната декларация:

```
struct AbstractView  
{  
    unsigned char u08; /* 1 байт */  
    unsigned short u16; /* 2 байта */  
    unsigned long u32; /* 4 байта */  
    unsigned char au08[3]; /* 3 байта */  
    long double f64; /* 8 байта */  
}structVar, arrayOfStructVars[10];
```

Следващата фигура показва абстрактния изглед на една структурна променлива **structVar** и масив от структурни променливи **arrayOfStructVars**. Структурната променлива

може да се представи визуално като единичен обект, състоящ се от променливи (членове). Променливите са представени на фигурата с жълти правоъгълници, отразяващи броя на байтовете в съответната променлива.



Фиг. 46 Абстрактен изглед на структурна променлива

Масивът може да се представи визуално като последователно разположени в паметта структурни променливи. Първата структурна променлива има индекс 0, следващата 1 и т.н. Достъпът до членовете става по описаните вече правила:

23.2.6 Обединения

Деклариране на обединение

Обединението е потребителски тип данни, който позволява една и съща област от паметта да се използва от различни типове променливи. Променливите, принадлежащи на едно обединение, се наричат членове или полета. Декларирането на обединение става по следния начин:

Деклариране на обединение

```
клас-паметопц квалификатор-типопц union име-обединение
{
    квалификатор-членопц тип1 име-член1;
    квалификатор-членопц тип2 име-член2;
    ...
    квалификатор-членопц типN име-членN;
} клас-паметопц квалификатор-типопц списък-union-променливиопц
```

където

клас-памет_{опц} (`static`, `auto`, `register` или `typedef`)

квалификатор-тип_{опц} (`const`, `volatile` или `const volatile`)

Тези атрибути могат да се поставят или пред декларацията на обединението:

```
клас-паметопц квалификатор-типопц union име-обединениеопц {...}
списък-union-променливиопц;
```

или след нея, но пред списък-union-променливи:

```
union име-обединениеопц {...} клас-паметопц квалификатор-типопц
списък-union-променливиопц;
```

Използването им в декларацията на обединението има смисъл само ако декларацията съдържа списък-union-променливи.

име-обединение_{опц}

Етикет (име) на обединението, зададен чрез идентификатор.

квалификатор-член_{опц}

Същото като квалификатор-тип.

тип1, тип2, ..., типN

Може да бъде скаларен тип, агрегатен тип, обединение, т.е.:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения¹ (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури (`struct {...}`);
- `typedef`-име.

Забележка¹: Типът на член на обединението не може да бъде обединението, към което принадлежи члена, но може да бъде указател към него.

име-член1, име-член2..., име-членN

Име на член, зададено чрез идентификатор.

списък-union-променливи_{опц}

име-union-променлива1 = инициализатор_{опц},

...,

име-union-променливаN = инициализатор_{опц}

където

име-union-променлива1,..., име-union-променливаN

Име на union-променлива, зададено чрез идентификатор.

инициализатор_{опц}

Израз (инициализатор), затворен във фигурни скоби.

Типът на инициализатора трябва да съвпада или да е съвместим с типа на първия член на обединението.

Забележка¹: В C89 инициализаторът трябва да е литерален константен израз, независимо от класа памет. В C99 инициализаторът на union-променливи с клас памет auto или register може да е всякакъв израз.

Инициализаторът не е позволен, ако за клас-памет се използва `typedef`.

Пример:

```
union UnionType /* Деклариране на обединение */
{
    int i;
    long double ld;
    enum EnumType e;
    union AnotherUnionType u;
    char a[20];
    unsigned long * ptr;
    struct StructType s;
    unsigned BitField : 8;1
};
```

Щом едно обединение е декларирано, може да дефинирате променливи от този тип. Това може да стане по време на декларацията на обединението или по късно в кода, както е показано по-долу:

```
клас-памятопц квалификатор-типопц union име-обединение  
списък-union-променливи;
```

или

```
union име-обединение клас-памятопц квалификатор-типопц  
списък-union-променливи;
```

Пример:

```
union UnionType /* Дефиниране на union-променлива по време на  
{  
    int i;  
    long double ld;  
    enum EnumType e;  
    union AnotherUnionType u;  
    char a[20];  
    unsigned long * ptr;  
    struct StructType s;  
    unsigned BitField : 8;1  
} u1;  
  
union UnionType u2; /* Дефиниране на union-променлива по-късно в  
кода */
```

Забележка¹: Обединенията могат да имат членове, наречени **битови полета**. Битовите полета имат по-специални свойства и са разгледани в [23.2.7 Битови полета](#).

При дефиниране на union-променлива, компилаторът заделя памет достатъчна да побере най-големия член на обединението. Всички членове на определението имат един и същ адрес.



Декларацията на обединение създава нов тип данни. Компиляторът заделя памет, достатъчна да побере най-големия член на обединението, само когато се дефинира променлива от това обединение.



Членовете на обединението не могат да се инициализират в тялото на обединението. Само първият член може да се инициализира при дефиниране на променлива от това обединение.

Пример:

```
union _Endiannes
{
    unsigned short u16;
    unsigned char a[2];
}endiannes = { /*u16*/0x55AA};

if (0xAA == endiannes.a[0])
    printf("The processor uses little-endian.\n");
else
    printf("The processor uses big-endian.\n");
```

Името на обединението не е задължително. Ако го пропуснете обаче, не можете да дефинирате променливи в кода от този тип (но може да дефинирате променливи по време на декларацията).

Пример:

```
union /* Деклариране на обединение без име */
{
    int i;
    long double ld;
    enum EnumType e;
    union AnotherUnionType u;
    char a[20];
    unsigned long * ptr;
    struct StructType s;
    unsigned BitField : 8;1
} u;
```

Обърнете внимание, че когато промените някой член на union-променлива, тази промяна се отразява на всички

останали членове.



Ако едно обединение се състои от членове, които не са логически свързани помежду си, уверете се, че логиката на програмата е такава, че докато някой от членовете е в употреба, никой от останалите членове не се използва. В противен случай рискувате да създадете логическа грешка.

Декларацията на едно обединение е видима от точката на деклариране до края на файла или блока, в който се намира декларацията.

Операции с обединения

Директен достъп до член на union-променлива

Директният достъпът до членовете на една union-променлива става с помощта на името на union-променливата последвано от точка и името на члена.

Директен достъп до член на union-променлива

име-union-променлива.име-член

Пример:

```
#include <stdio.h>
```

```
union Animal
```

```
{
```

```
    char* dogName;
```

```
    char* catName;
```

```
} animal;
```

```
int main(void)
```

```
{
```

```
    animal.catName = "Winnetou";
```

```
    printf("The name of my cat is %s\n", animal.catName);
```

```
    printf("The name of my dog is %s\n", animal.dogName);
```

```
    animal.dogName = "Benji";
```

```
    printf("The name of my dog is %s\n", animal.dogName);
```



```

    printf("The name of my cat is %s\n", animal.catName);

    return 0;
}

```

Косвен достъп до член на union-променлива

Косвеният достъп до членовете на union-променлива става с помощта на указател към обединението.

Дефиниране на указател към обединение

```
union име-обединение * име-указател;
```

Достъпът до член на union-променлива чрез указател към нея става с помощта на операцията ->.

Косвен достъп до член на union-променлива

```
име-указател -> име-член
```

Пример:

```

#include <stdio.h>

union Animal
{
    char* dogName;
    char* catName;
} animal, *ptr_to_animal = &animal;

int main(void)
{
    animal.catName = "Winnetou";
    printf("The name of my cat is %s\n",
        ptr_to_animal->catName);
    printf("The name of my dog is %s\n",
        ptr_to_animal->dogName);
    animal.dogName = "Benji";
    printf("The name of my dog is %s\n",
        ptr_to_animal->dogName);
    printf("The name of my cat is %s\n",
        ptr_to_animal->catName);

    return 0;
}

```

```
}
```

Присвояване на обединения

Една union-променлива може да се присвои директно на друга union-променлива. Единственото условие, което трябва да е изпълнено е двете union-променливи да са от един и същ тип.

Пример:

```
#include <stdio.h>

union Animal
{
    char* dogName;
    char* catName;
} animal1 = { "Winnetou" }, animal2;

int main(void)
{
    animal2 = animal1; /* animal1 -> animal2 */
    printf("The name of my cat is %s\n", animal2.catName);
    printf("The name of my dog is %s\n", animal2.dogName);

    return 0;
}
```

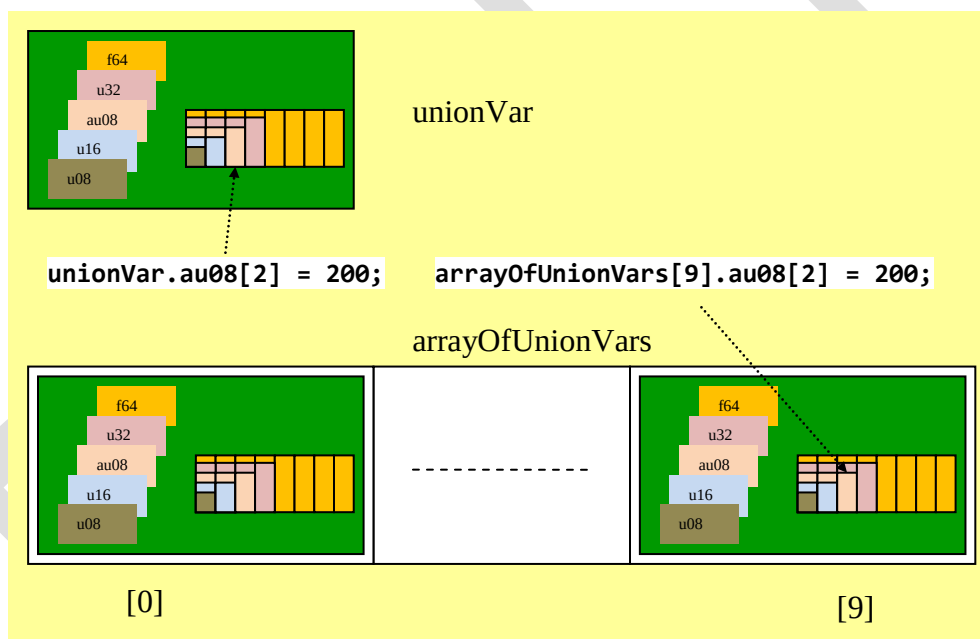
Абстрактен изглед

Нека допуснем, че имаме следната декларация:

```
union AbstractView
{
    unsigned char    u08;        /* 1 байт */
    unsigned short   u16;        /* 2 байта */
    unsigned long     u32;        /* 4 байта */
    unsigned char    au08[3];    /* 3 байта */
    long double       f64;        /* 8 байта */
}unionVar, arrayOfUnionVars[10];
```

Следващата фигура показва абстрактния изглед на една

union-променлива **unionVar** и масив от union-променливи **arrayOfUnionVars**. Union-променлива може да се представи визуално като единичен обект, състоящ се от променливи (членове), насложени една върху друга, което се налага от факта, че членовете използват обща памет. Променливите са представени на фигурата с насложени правоъгълници с различни цветове, отразяващи броя на байтовете в съответната променлива. Масивът може да се представи визуално като последователно разположени в паметта union-променливи. Първата union-променлива има индекс 0, следващата 1 и т.н. Достъпът до членовете става по описаните вече правила:



Фиг. 47 Абстрактен изглед на union-променлива

23.2.7 Битови полета

Дефиниране на битово поле

Битовото поле е целочислена променлива, която се състои от

определен брой битове. Едно битово поле може да бъде само член на структура или обединение. Извън този контекст дефинирането на битово поле не е позволено. Битово поле се дефинира по следния начин:

Дефиниране на битово поле

```

клас-паметопц квалификатор-типоп union-или-struct име-типопц
{
    квалификатор-членопц тип име-битово-поле1опц: широчина;
    квалификатор-членопц тип име-битово-поле2опц: широчина;
    ...
    квалификатор-членопц тип име-битово-полеNопц: широчина;
} клас-паметопц квалификатор-типоп списък-променливиопц;

```

където

клас-памет_{опц} (**static**, **auto**, **register** или **typedef**)

квалификатор-тип_{опц} (**const**, **volatile** или **const volatile**)

Тези атрибути могат да се поставят или пред декларацията на структурата или обединението:

```

клас-паметопц квалификатор-типопц union-или-struct
име-типопц {...} списък-променливиопц;

```

или след нея, но пред **списък-променливи**:

```

union-или-struct име-типопц {...}
клас-паметопц квалификатор-типопц списък-променливиопц;

```

Използването им в декларацията на структурата или обединението има смисъл само ако декларацията съдържа **списък-променливи**.

име-тип_{опц}

Етикет (име) на структурата или обединението, зададен чрез идентификатор.

квалификатор-член_{опц}

Същото като **квалификатор-тип**.

тип

Може да бъде **_Bool**^{c99}, **int**¹, **signed int** или **unsigned int**

Забележка¹: В контекста на битовите полета тип **int** може да бъде **signed** или **unsigned** в зависимост от компилатора.

име-битово-поле_{опц}1, **име-битово-поле**_{опц}2..., **име-битово-поле**_{опц}N

Име на битово поле, зададено чрез идентификатор. Ако **име-битово-поле** ($n=1\div N$) липсва и **широчина** $\neq 0$, битовото поле не може да се достъпи от кода и служи само за уплътнение. Ако **име-битово-поле** ($n=1\div N$) липсва и **широчина** = 0, компилаторът разполага следващото битово поле в началото на следващия сегмент за съхранение.

широчина

Определя броя на битовете в битовото поле. Може да бъде само литерален неотрицателен целочислен израз, чиято стойност не трябва да превишава броя на битовете на специфицирания тип.

`_Bool` → **широчина** $\in [0 \dots (8 * \text{sizeof}(\text{_Bool}))]$.

`int` (signed или unsigned) → **широчина** $\in [0 \dots (8 * \text{sizeof}(\text{int}))]$.

Ако **широчина** е 0, **име-тип**_{опц} трябва да се пропусне.

списък-променливи

име-променлива1 = **инициализатор**_{опц},

...,

име-променливаN = **инициализатор**_{опц}

където

име-променлива1, ..., **име-променлива**N

Име на struct- или union-променлива, зададено чрез идентификатор.

инициализатор

Ако типът е структура, инициализаторът се състои от изрази (инициализатори)¹, разделени със запетая и затворени във фигурни скоби. Броят на инициализаторите трябва да е по-малък или равен на броя на членовете на структурата. Ако броят на инициализаторите е по-малък от броя на членовете, членовете, за които не достигат инициализатори, се инициализират с 0.

Ако типът е обединение, инициализаторът се състои от един израз (инициализатор)¹, затворен във фигурни скоби. Типът на инициализатора трябва да съвпада или да е съвместим с типа на първия член на обединението.

Забележка¹: В C89 инициализаторът трябва да е литерален константен израз, независимо от класа памет. В C99 инициализаторите на struct- или union-променливи с клас памет `auto` или `register` могат да са всякакви изрази.

Едно n -битово поле може да представи 2^n различни стойности, от 0 до 2^n-1 ако е тип `unsigned`, и -2^{n-1} до $2^{n-1}-1$, ако е от тип `signed` и отрицателните числа се представят в допълнителен код (**2's complement**). Знаково битово поле с дължина 1 бит в допълнителен код може да има само две стойности: 0 и -1.

Езикът C дава голяма свобода на компилаторите по отношение на битовите полета. Следващите параметри на битовите полета са имплементационно-зависими:

- **Широчина на битовото поле** – зависи от големината на специфицирания тип:

За тип `_Bool` широчината може да се изменя от 0 до 8.

За тип `int` (`signed` или `unsigned`) широчината може да се изменя от 0 до 16 или 32 в зависимост от големината на тип `int`.

- **Големина на сегмента за съхранение на битовите полета** – стандартът дефинира, че компилаторът може да заделени всеки адресируем сегмент, достатъчен да съхрани битово поле. Сегментът може да бъде 1, 2, или 4 байта в зависимост от архитектурата на процесора.
- **Ред на разполагане на битовите полета в сегмента за съхранение** – компилаторът разполага първото битово поле в сегмента за съхранение. Ако има достатъчно място, следващото битово поле трябва да се разположи веднага след първото в същия сегмент. Ако няма достатъчно място, дали битовото поле ще се разположи веднага след предходното и ще продължи в

следващия сегмент или директно ще се разположи в следващия сегмент, зависи от имплементацията. Редът на разполагане на битовите полета в един сегмент за съхранение (от малдшите-към-старшите или от старшите-към-младшите битове) също зависи от имплементацията.

- **Подравняването в паметта на сегмента за съхранение** – стандартът не прави никакви изисквания по отношение на адресите в програмната памет, на които се разполагат сегментите за съхранение.

Всичко това прави битовите полета тясно свързани с архитектурата на процесора. Това означава, че не може да правите никакви допускания относно начина на пакетиране на битовите полета. Дори различни компилатори за един и същ процесор могат да имат различно поведение. По тази причина битовите полета имат много ниска преносимост.



Една декларация на тип (обединение или структура) ще бъде преносима, ако логиката на програмата не се интересува от начина на пакетиране на битовите полета и широчината на битовите полета не надвишава големината на машинната дума.

Битовите полета са членове на структури или обединения и като такива достъпът до тях се осъществява като всеки друг обичаен член:

име-struct-или-union-променлива . име-битово-поле
име-указател-към-структура-или-обединение -> име-битово-поле

Пример:

```
struct CommunicationStatus
{
    unsigned isTransmissionFinished : 1;
    unsigned isReceptionFinished   : 1;
```

```

}comStatus;
...

if (1 == comStatus.isTransmitionFinished)
{
    /*...*/
}
else
{
    /*...*/
}

```

Битовите полета могат да се използват навсякъде, където могат да се използват целочислени променливи. Компиляторът автоматично преобразува битовото поле в тип **int** или **unsigned int** съгласно правилата за автоматично повишаване на типовете (вижте [26 Преобразувания](#)).

Ограничения на битовите полета

Битовите полета имат следните ограничения:

- Битово поле не може да съществува самостоятелно извън тялото на структура или обединение;

```

int main(void)
{
    int b:3;    /* !!!Грешка, битово поле може да е само
                член на структура или обединение */
    return 0;
}

```

- Не може да се дефинира указател към битово и съответно не може да се извлича адреса на битово поле;

```

struct S
{
    int a : 3;
    int b : 5;
}bf;

```



```

int main(void)
{
    int:5 *p; /* !!!Грешка, това е невалидна конструкция в
               С */

    p = &bf.a; /* !!!Грешка, не може да се извлича адреса
               на битово поле */

    return 0;
}

```

- Не може да се дефинира масив от битови полета.

```

int:5 arr[10]; /* !!!Грешка, това е невалидна конструкция
               в С */

```

23.3 Квалифициращи на типа

C89 въвежда така наречените квалифициращи на типа - **const** и **volatile**. Тези квалифициращи могат да се прилагат по отделно или в комбинация върху:

- променливи;

volatile const тип име-променлива = инициализатор;

- параметри на функции;

```

тип-резултат име-функция(volatile const тип име-параметър)
{
    /* Функцията може само да чете const-параметър, но не
       може да го модифицира */
}

```

- членове на структури и обединения.

```

struct-или-union име-тип
{
    volatile const тип име-член;
    ...
};

```

Обекти, квалифицирани с **const**, не могат да се променят в програмата, но могат да се променят от хардуера. По тази причина const-обектите трябва да се инициализират по време на дефинирането си.

Пример:

```
const int limit = 1000;

struct Rectangle
{
    const int width;
    int lenght;
};

struct Rectangle square = {10, 10};
```

Ключовата дума **const**, приложена към параметър на функция, указва на компилатора, че параметърът не може да се променя в тялото на функцията.

Пример:

```
int square(const int x){ return x*x; }
```

Всеки опит да промените const-обект в кода ще доведе до грешка при компилиране.



Ако една функция само чете даден параметър, препоръчително е да използвате ключовата дума **const**.

тип-резултат име-функция(**const** тип име-параметър) { /*...*/ }



Ако една функция използва за параметър указател към обект, който не трябва да се променя, препоръчително е да дефинирате параметъра като указател към const-обект.

тип-резултат име-функция(**const** тип* име-указател) { /*...*/ }

Обект, квалифициран с **volatile**, може да бъде модифициран

по някакъв неявен начин за компилатора, например от хардуера или в подпрограма за обработка на прекъсване. Ключовата дума **volatile** инструктира компилатора да не прави никакви оптимизации и да изчита стойността на volatile-обекта всеки път, когато се използва, дори обектът да не е бил променен в кода от последното му четене.

Компилаторът е свободен да съхранява обекти, квалифицирани с **const**, но не с **volatile**, в ROM (Read Only Memory) паметта на компютъра.

23.4 Област на видимост на променливите

Областта на видимост на една променлива определя тази част от програмния код, в която променливата е видима и може да се използва. Областта на видимост на променлива се определя от мястото където тя е дефинирана. В С има 4 области на видимост:

- област на видимост в прототип на функция;

Облас на видимост в прототип на функция

```
тип-резултат име-функция(тип    име-параметър1опц,  
                             тип    име-параметър2опц,  
                             ...  
                             тип    име-параметърNопц );
```

където

име-параметър1_{опц}, име-параметър2_{опц}, ..., име-параметърN_{опц}

Име на параметър, зададено чрез идентификатор. Тези идентификатори са видими само в прототипа. В действителност компилаторът игнорира имената на параметрите. По тази причина името на параметър в прототип на функция може да не съвпада с името на параметъра в дефиницията на функцията или дори може да се изпусне.

Пример:

```
void foo(int x, double y);
```

или

```
void foo(int, double);
```

- област на видимост във функция;

Област на видимост във функция

```
тип-резултат име-функция( тип    име-параметър1,  
                           тип    име-параметър2,  
                           ...  
                           тип    име-параметърN )  
{  
    /* Дефиниране на променливи */  
    тип име-променлива1;  
    тип име-променлива2;  
    ...  
    тип име-променливаN;  
  
    /* С оператори */  
    оператор1;  
    оператор2;  
    ...  
    операторN;  
}
```

Област на видимост на параметрите и локалните променливи на функцията

Променливи, дефинирани във функция, са видими само в нея и не са достъпни за други функции. C89 изисква дефинирането на променливи да става в началото на функцията преди всички други C оператори. Такива променливи се наричат още **локални променливи**. Параметрите на функциите също се явяват локални променливи. Тяхната област на видимост също е тялото на функцията.

C99 премахва това ограничение и позволява променливите да се дефинират след C оператори. Областта на видимост се простира от точката на дефиниране до края на функцията.

Пример:

```
#include <stdio.h>
```

```

int main(void)
{
    int x = 1; // Дефиниране на променлива в начало на блок
    printf("x is %d\n", x);

    int y = 2; // Дефиниране на променлива в средата на блок
    printf("y is %d\n", y);

    return 0;
}

```

Област на видимост на x

Област на видимост на y^{C99}



Освен ако логиката на програмата не го изисква, за предпочитане е дефинициите на всички локални променливи да се разполагат в началото на функцията.

- област на видимост в блок с код;

Блок с код или още съставен оператор се нарича съвкупността от оператори затворена между отваряща и затваряща фигурна скоба.

Област на видимост в блок с код

```

{
    /* Дефиниране на променливи */
    тип име-променлива1;
    тип име-променлива2;
    .....
    тип име-променливаN;

    /* Изпълними оператори */
    оператор1;
    оператор2;
    .....
    операторN;
}

```

Област на видимост на локалните променливи на блок с код

C89 изисква дефинирането на променливи да става в началото на блока преди всички други C оператори.

C99 премахва това ограничение и позволява променливите да се дефинират след C оператори. Областта на видимост се

простира от точката на дефиниране до края на блока.

Блоковете могат да бъдат вложени един в друг. Имената на локални променливи, дефинирани във външния блок, могат да съвпадат с имената на локалните променливи, дефинирани във вложения блок. В този случай променливите на вложения блок "скриват" променливите на външния блок в рамките на вложения блок.

Пример:

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int i = 10; /* i е локална за main() */
```

```
    printf("main:i = %d\n", i);
```

main: i

```
    /* Блок 1 */
```

```
    int i = 20; /* i е локална за блок 1 */
```

```
    printf("block1:i = %d\n", i);
```

Блок1: i

```
    /* Блок 2 */
```

```
    int i = 30; /* i е локална за блок 2 */
```

```
    printf("block2:i = %d\n", i);
```

Блок2: i

```
    /* Край на Блок 2 */
```

```
    printf("block1:i = %d\n", i);
```

```
    /* Край на Блок 1 */
```

```
    printf("main:i = %d\n", i);
```

```
    return 0;
```

```
}
```



Избягвайте използването на еднакви имена на локални променливи във вложен и обхващащия го блок. Това води до объркване и влошава качеството на кода.

- област на видимост във файл.

Област на видимост във файл

```
/* Начало на файла */
```

```
тип име-променлива1; /* Видимост в име-функция1 ÷ име-функцияN */
```

```
тип-резултат име-функция1(списък-параметри)
```

```
{
```

```
}
```

```
тип име-променлива2; /* Видимост в име-функция2 ÷ име-функцияN */
```

```
тип-резултат име-функция2(списък-параметри)
```

```
{
```

```
}
```

```
...
```

```
тип име-променливаN; /* Видимост в име-функцияN */
```

```
тип-резултат име-функцияN(списък-параметри)
```

```
{
```

```
}
```

```
/* Край на файла */
```

Променливи, дефинирани извън всички функции, имат област на видимост от точката на дефиниране до края на файла. Такива променливи се наричат още **глобални променливи**. Глобалните променливи са достъпни за всички функции, които се намират след дефиницията ѝ.

Пример:

```
int i; /* Видима е в foo1() и foo2() */
```

```
int foo1(void)
```

```
{
```

```
    i = 10; /* i може да се използва в main() */
```

```
    j = 20; /* !!! j не може да се използва в main() */
```

```
    return 0;
```

```
}
```

```
int j; /* Видима е само в foo2() */
```

```
void foo2(void)
{
    i = 20; /* i може да се използва в foo() */
    j = 20; /* j може да се използва в foo() */
}
```

Стандартът допуска име на локална променлива да съвпада с име на глобална променлива. В този случай локалната променлива "скрива" глобалната променлива в рамките на блока, в който е дефинирана.



Избягвайте използването на еднакви имена на локална и глобална променлива. Това води до объркване и влошава качеството на кода.

23.5 Време на живот на променливите

Времето на живот на променлива представлява времето, през което променливата се съхранява в паметта на компютъра. Глобалните променливи се създават при стартиране на програмата и се съхраняват докато трае изпълнението ѝ. След завършване на програмата паметта, заделена за глобалните променливи, се освобождава. Локалните променливи се съхраняват само докато се изпълнява блока с код, в който са дефинирани. Паметта за локалните променливи се заделя при влизане в блока, в който са дефинирани и се освобождава при излизане от него. Времето на живот на локалните променливи може да се променя с помощта на т.нар. спецификатор за памет **static** (вижте [23.6 Клас памет на променливите](#)).

23.6 Клас памет на променливите

Табл.91 описва категориите променливи в C и класовете

памет, които могат да се прилагат към тях.

категория	клас памет
глобални	<code>static</code>
	<code>extern</code>
	<code>typedef</code>
локални	<code>auto</code>
	<code>register</code>
	<code>static</code>
	<code>typedef</code>

Табл. 91 Клас памет на променливите

- `auto`

Клас памет `auto` може да се прилага само към локални променливи.

Клас памет `auto`

`auto`_{опц} квалификатор-тип_{опц} спецификатор-тип_{опц} тип име-променлива = инициализатор_{опц};

където

квалификатор-тип_{опц} може да е:

`const`
`volatile`
`const volatile` или `volatile const`

спецификатор-тип_{опц} се прилага само върху някои базови типове и може да е:

<code>signed</code>	(<code>int</code> или <code>char</code>)
<code>unsigned</code>	(<code>int</code> или <code>char</code>)
<code>short</code>	(<code>int</code>)
<code>long</code>	(<code>int</code>)
<code>signed short</code>	(<code>int</code>)
<code>signed long</code>	(<code>int</code>)
<code>signed long long</code> ^{C99}	(<code>int</code>)
<code>unsigned short</code>	(<code>int</code>)
<code>unsigned long</code>	(<code>int</code>)
<code>unsigned long long</code> ^{C99}	(<code>int</code>)

тип

Може да бъде всеки обектен тип в езика C:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури (`struct {...}`);
- `typedef`-име.

име-променлива

Име на локална променлива, зададено чрез идентификатор.

инициализатор_{опц}

Форматът на инициализатора зависи от типа на променливата.

По подразбиране локалните променливи имат клас на съхранение `auto`. По тази причина ключовата дума `auto` е излишна и рядко се използва. Локалните променливи клас `auto` се наричат още автоматични променливи. Тези променливи се съхраняват в област от паметта, наречена стек. Когато програмата влезе в локална област на действие, каквато е тялото на функция или блок с код, компилаторът заделя памет за тях в стека. При излизане от функцията (или блока с код) паметта се освобождава и може да се използва от компилатора за заделяне на памет за други автоматични променливи.



По подразбиране локалните променливи имат клас на съхранение `auto`. По тази причина ключовата дума `auto` е излишна и може да не се използва въобще .

- `register`

Клас памет `register` може да се прилага само към локални променливи и параметри на функции.

Клас памет register

register_{опц} квалификатор-тип_{опц} спецификатор-тип_{опц} тип
име-променлива = инициализатор_{опц};

или

тип-резултат име-функция

```
(  
registerопц квалификатор-типопц спецификатор-типопц тип  
име-параметър1,  
registerопц квалификатор-типопц спецификатор-типопц тип  
име-параметър2,  
...  
registerопц квалификатор-типопц спецификатор-типопц тип  
име-параметърN,  
);
```

където

квалификатор-тип_{опц} може да е:

const
volatile
const volatile или **volatile const**

спецификатор-тип_{опц} се прилага само върху някои базови типове и може да е:

signed	(int или char)
unsigned	(int или char)
short	(int)
long	(int)
signed short	(int)
signed long	(int)
signed long long ^{C99}	(int)
unsigned short	(int)
unsigned long	(int)
unsigned long long ^{C99}	(int)

тип

Може да бъде всеки обектен тип в езика C:

- Базови типове (**_Bool**^{C99}, **char**, **int**, **float**, **double**);
- Изброявания (**enum** {...});
- Обединения (**union** {...});
- Масиви (T[]);
- Указатели (T*);

- Структури (`struct {...};`);
- `typedef`-име.

име-променлива

Име на локална променлива, зададено чрез идентификатор.

име-параметър

Име на параметър на функция, зададен чрез идентификатор.

инициализатор_{опц}

Форматът на инициализатора зависи от типа на променливата.

Когато локална променлива или параметър на функция се дефинира като `register`, това е заявка за компилатора да заделни памет за тази променлива така, че достъпът до нея да бъде максимално бърз, например във вътрешните регистри на процесора вместо в RAM паметта. Ако компилаторът не може да удовлетвори тази заявка, променливата се съхранява като обикновена `auto`-променлива. За разлика от `auto`-променливите обаче, адресът на `register`-променливите не може да се получи с операцията за вземане на адрес `&`.



Операцията за извличане на адрес `&` не може да се прилага към `register`-локални променливи.



Ключовата дума `register` на практика може да не се използва. Съвременните компилатори са достатъчно интелигентни сами да преценят дали да оптимизират достъпа до дадена променлива или не.

- `static`

Клас памет `static` може да се прилага към глобални и локални променливи (без параметри на функции).

Клас памет `static`

`static`_{опц} квалификатор-тип_{опц} спецификатор-тип_{опц} тип име-променлива
= инициализатор_{опц};

където

квалификатор-тип_{опц} може да е:

`const`
`volatile`
`const volatile` или `volatile const`

спецификатор-тип_{опц} се прилага само върху някои базови типове и може да е:

<code>signed</code>	(<code>int</code> или <code>char</code>)
<code>unsigned</code>	(<code>int</code> или <code>char</code>)
<code>short</code>	(<code>int</code>)
<code>long</code>	(<code>int</code>)
<code>signed short</code>	(<code>int</code>)
<code>signed long</code>	(<code>int</code>)
<code>signed long long</code> ^{C99}	(<code>int</code>)
<code>unsigned short</code>	(<code>int</code>)
<code>unsigned long</code>	(<code>int</code>)
<code>unsigned long long</code> ^{C99}	(<code>int</code>)

тип

Може да бъде всеки обектен тип в езика C:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури (`struct {...}`);
- typedef-име.

име-променлива

Име на локална или глобална променлива, зададено чрез идентификатор.

инициализатор_{опц}

Форматът на инициализатора зависи от типа на променливата.

Статичните променливи (глобални и локални) се съхраняват извън стека и съществуват през цялото време на изпълнение на програмата. Те се създават при стартиране на програма. Ако дефиницията на променливата е съпроводена и с инициализатор, то стойността на инициализатора се

присвоява на променливата също при стартирането на програмата.

Когато една локална променлива се дефинира като `static`, се променя времето ѝ на живот. Паметта за съхранение на статичните локални променливи се запазва през изпълнението на цялата програма. Въпреки това областта на видимост си остава блока, в който е дефинирана тази променлива. Това означава, че при излизане от блока променливата не е достъпна, но стойността ѝ се запазва.

Глобалните статични променливи са достъпни само във файла, в който са дефинирани. Имената на статични глобални променливи, дефинирани в различни файлове, могат да съвпадат. В този случай всяка променлива е достъпна само във файла, в който е дефинирана и няма нищо общо с другата променлива.



Ако една глобална променлива се използва само във файла, в който е дефинирана, направете я статична.

- `extern`

Глобалните променливи без никакъв спецификатор за клас-памет са достъпни за всички файлове на програмата съгласно следните правила. Глобалната променлива има област на видимост от точката на дефинирането си до края на файла. Ако тази променлива се използва преди точката на дефинирането си или в друг файл, тя трябва да се декларира. Декларацията изглежда като дефиницията само, че се добавя ключовата дума **`extern`**. Декларацията не води до заделяне на памет за променливата, а е просто указание за компилатора, че тази променлива вече е дефинирана на друго място в програмата.

Клас памет extern

extern квалификатор-тип_{опц} спецификатор-тип_{опц} тип
име-променлива;

където

квалификатор-тип_{опц} може да е:

const
volatile
const volatile или **volatile const**

спецификатор-тип_{опц} се прилага само върху някои базови типове и може да е:

signed	(int или char)
unsigned	(int или char)
short	(int)
long	(int)
signed short	(int)
signed long	(int)
signed long long ^{C99}	(int)
unsigned short	(int)
unsigned long	(int)
unsigned long long ^{C99}	(int)

тип

Може да бъде всеки обектен тип в езика C:

- Базови типове (**_Bool**^{C99}, **char**, **int**, **float**, **double**);
- Изброявания (**enum** {...});
- Обединения (**union** {...});
- Масиви (T[]);
- Указатели (T*);
- Структури (**struct** {...});
- typedef-име.

име-променлива

Име на външна променлива, зададено чрез идентификатор.

Глобалните нестатични променливи се наричат още външни променливи. Външните променливи се съхраняват извън стека и съществуват през цялото време на изпълнение на програмата. Те се създават при стартиране на програма. Ако

дефиницията на променливата е съпроводена и с инициализатор, то стойността на инициализатора се присвоява на променливата също при стартирането на програмата.



Въпреки, че глобалните променливи (статични или външни) могат да се дефинират навсякъде в един файл, стига да са извън функция, за предпочитане е дефинициите им да се разполагат в началото на файла преди всички функции.

▪ typedef

Клас памет `typedef`¹ се използва за създаване на други имена на вече съществуващи типове.

Забележка¹: Ключовата дума **`typedef`** в действителност не е спецификатор за клас памет. Тя се нарича така само от синтактична гледна точка, тъй като само една от ключови думи от Табл.91 може да присъства в декларацията на една променлива.

Декларация на typedef име

```
typedefопц квалификатор-типопц спецификатор-типопц тип  
ново-име-на-тип;
```

или

```
typedefопц тип-резултат ново-име-на-тип (списък-параметри);
```

където

квалификатор-тип_{опц} може да е:

```
const  
volatile  
const volatile или volatile const
```

спецификатор-тип_{опц} се прилага само върху някои базови типове и може да е:

```
signed (int или char)  
unsigned (int или char)
```


<code>short</code>	<code>(int)</code>
<code>long</code>	<code>(int)</code>
<code>signed short</code>	<code>(int)</code>
<code>signed long</code>	<code>(int)</code>
<code>signed long long</code> ^{C99}	<code>(int)</code>
<code>unsigned short</code>	<code>(int)</code>
<code>unsigned long</code>	<code>(int)</code>
<code>unsigned long long</code> ^{C99}	<code>(int)</code>

ТИП

Може да бъде всеки обектен тип в езика C:

- Базови типове (`_Bool`^{C99}, `char`, `int`, `float`, `double`);
- Изброявания (`enum {...}`);
- Обединения (`union {...}`);
- Масиви (`T[]`);
- Указатели (`T*`);
- Структури (`struct {...}`);
- `typedef`-име.

НОВО-ИМЕ-НА-ТИП

Синоним на типа `тип`, зададен чрез идентификатор.

Пример:

```
typedef unsigned char uint8;
typedef uint8 ubyte;
```

Идентификаторите `uint8` и `ubyte` са други имена на типа `unsigned char`. Може да ги използвате навсякъде, където може да използвате типа `unsigned char`.

```
uint8 x;
ubyte y;

uint8 foo(uint8 a, uint8 b) { /*...*/ }
```

Пример:

```
typedef struct
{
```

```
int x;  
int y;  
int z;  
}Coordinate;
```

```
Coordinate coordinate;
```

Ключовата дума `typedef` позволява свеждане на сложни дефиниции на променливи до класическата форма:

квалификатор-тип_{опц} спецификатор-тип_{опц} тип име-променлива;

Например сравнете следните конструкции:

```
int* (*ptrToFunction)(void); /* 1 */
```

и

```
typedef int* (*ptrToFunctionType)(void); /* 2 */  
ptrToFunctionType ptrToFunction; /* 3 */
```

Конструкция 1 се интерпретира като указател към функция, която няма параметри и връща указател към обекти тип `int`.

Конструкция 2 представлява декларация на тип `ptrToFunctionType`, който се интерпретира като указател към функция, която няма параметри и връща указател към обекти тип `int`.

Конструкция 3 представлява дефиниция на променлива `ptrToFunction` от тип `ptrToFunctionType`, т.е. `ptrToFunction` е указател към функция, която няма параметри и връща указател към обекти тип `int`.

Декларацията на `typedef`-име е видима от мястото на декларацията до края на блока или файла, в който се намира декларацията.



Ако в дефиницията на променлива или параметър на функция се използват квалификатори на типа и за тип се използва typedef-име, което съдържа вече в себе си същите квалификатори на типа, то компилаторът игнорира допълнителните квалификатори.

Пример:

```
typedef volatile int type;  
  
volatile type var;  
    ↓  
volatile volatile int var; → volatile int var;
```

23.7 Инициализация на променливите

Локалните променливи (auto и register) от скаларен тип могат да се инициализират с всеки израз, чийто тип е съвместим с типа на променливата.

Външните и статичните променливи (локални и глобални) от скаларен тип изискват инициализаторът да е КОНСТАНТЕН¹ израз, чийто тип е съвместим с типа на променливата.

Променливи от агрегатен тип и обединения изискват инициализаторът да е КОНСТАНТЕН¹ израз, чийто тип е съвместим с типа на променливата, независимо дали променливата е локална (auto и register)², външна или статична (локална и глобална).

Забележка¹: Константен е израз е израз, чиято стойност може да бъде изчислена по време на компилиране на кода.

Забележка²: C99 премахва това ограничение и позволява

инициализаторите на локални променливи с клас памет auto или register да бъдат всякакви изрази.

Ако липсва инициализатор, външните и статичните променливи се инициализират от компилатора в 0 за целочислени и указателни типове, и 0.0 за типове с плаваща запетая. Локалните променливи (auto и register) не се инициализират автоматично от компилатора и съдържат произволна стойност (Табл.92).

клас памет	тип	инициализатор	подразбиращ се инициализатор
static/ extern	скаларен	константен израз	0 / 0.0
static/ extern	масив	{списък-константни-изрази}	0 / 0.0
static/ extern	структура	{списък-константни-изрази}	0 / 0.0
static/ extern	обединение	константен израз	0 / 0.0
auto/ register	скаларен	всякакъв израз	недефиниран
auto/ register	масив	{списък-константни-изрази} {списък-всякакви-изрази^{C99}}	недефиниран
auto/ register	структура	{списък-константни-изрази} {списък-всякакви-изрази^{C99}} - struct-променлива от същия тип	недефиниран
auto/ register	обединение	{списък-константни-изрази} {списък-всякакви-изрази^{C99}} - union-променлива от същия тип	недефиниран

Табл. 92 Инициализация на променливи

24 Константи

Константата е фиксирана стойност, която не може да се изменя в програмата. Константите се наричат още литерали, за да се различават от обектите, квалифицирани с `const`, чиито стойности също не могат да се изменят в програмата. Всяка константа се характеризира със стойност и тип.

Езикът С дефинира следните константи:

- Целочислени константи;
- Константи с плаваща запетая;
- Символни константи;
- Низови константи.

24.1 Целочислени константи

Езикът С използва три формата за представяне на целочислените константи: десетични, осмични¹ и шестнайсетични.

Забележка¹: Осмичните константи на практика не се използват. Тук са споменати само за пълнота.

Десетичните константи се състоят от последователност от арабските цифри $0 \div 9$, като не могат да започват с 0.

Пример: 10, 123, 64674, -100 и т.н.

Шестнайсетичните константи се състоят от префикса 0x

(или 0X) последван от арабските цифри 0 ÷ 9 и латинските букви a ÷ f (A ÷ F).

Пример: 0x12, 0xa1 (или 0xA1), 0xFF (или 0xff) и т.н.

Типът на една целочислена константа е първият целочислен тип, започвайки от int, който може да представи стойността ѝ. Минималният тип на целочислените константи може изрично да бъде указан с помощта на следните суфикси.

- U или u – unsigned int;
- L или l – long int;
- UL или ul – unsigned long int;
- LL^{C99} или ll^{C99} – long long int;
- ULL^{C99} или ull^{C99} – unsigned long long int

Табл.93 описва в резюме типа, който компилатора ще назначи на една целочислена константа в зависимост от стойността, формата на представяне и наличието на суфикс.

суфикс	десетични		шестнайсетични	
	C89	C99	C89	C99
-	int long int unsigned long int	int long int long long int	int unsigned int long int unsigned long int	int unsigned int long int unsigned long int long long int unsigned long long int
u или U	unsigned int unsigned long int	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int	unsigned int unsigned long int unsigned long long int
l или L	long int unsigned long int	long int long long int	long unsigned long	long unsigned long int long long int unsigned long long int
ul	unsigned long int	unsigned long int	unsigned long	unsigned long int

или UL		unsigned long long int		unsigned long long int
ll ^{C99} или LL ^{C99}		long long int		long long int unsigned long long int
ull ^{C99} или ULL ^{C99}		unsigned long long int		unsigned long long int

Табл. 93 Тип на целочислените константи в C89 /C99

Ако стойността на константата е твърде голяма, за да се побере и в най-големия целочислен тип, резултатът е недефиниран.

Размерите на целочислените константи за всеки компилатор са описани в хедър-файла <limits.h>.

24.2 Константи с плаваща запетая

Константите с плаваща запетая имат цяла и дробна част. Те могат да се записват по следните начини:

дес-цяла-част.дес-дробна-частE(или e)±степен
0X(или x)шес-цяла-част.шес-дробна-частP(или p)±степен^{C99}

където

дес-цяла-част

дес-дробна-част

Последователност от десетични цифри 0÷9

шес-цяла-част

шес-дробна-част

Последователност от шестнайсетични цифри 0÷F(или f)

E(или e)

Десетична експонента (указва степен на 10)

P(или p)

Шестнайсетична експонента (указва степен на 2)

степен

Последователност от десетични цифри 0÷9

0X(или x)

Шестнайсетичен префикс, указващ шестнайсетична константа

В случай, че цялата или дробната част е 0, тя може да се изпусне, но разделителната точка задължително трябва да присъства. Именно, тя е указание за компилатора, че дадената константа трябва да се представи вътрешно като константа с плаваща запетая. Когато степента е положителна, знакът + пред нея може да се изпусне. При десетичните константи с плаваща запетая експоненциалната част (**E±степен**) може да се изпусне, докато при шестнайсетичните константи експоненциалната част е задължителна, дори степента да е 0.

Като разделител между цялата и дробната част се използва точка, а не запетая.



В С дробните числа използват точка, а не запетая, за разделяне на цялата и дробната част.

Пример: Десетични константи с плаваща запетая

123.093213

0.7124

123.093E+2

0.123E-2

0.123 или .123

123.0 или 123.

0.123E-2 или .123E-2

123.0E+1 или 123.E1

Пример: Шестнайсетични константи с плаваща запетая

0xabc.defP2 или 0xabc.defP+2

0x.defP2 или 0x0.defP+2
0xabc.P2 или 0xabc.0P+2
0xabc.defP0

По подразбиране типът на константите с плаваща запетая е **double**. Минималният тип може изрично да бъде зададен чрез следните суфикси:

F или f – float

L или l – long double

Пример:

23.3 – тип double

23.3f – тип float

23.3L – тип long double

Параметрите на типовете с плаваща запетая за всеки компилатор са указани в заглавния файл <float.h>.

24.3 Символни константи

Символната константа представлява един или повече символи затворени в апострофи. С стандартът дефинира два типа символни константи: нормални и широки. Широките символни константи се предхождат от буква L.

'последователност-символи'

L 'последователност-символи'

където

последователност-символи

Може да включва:

- Всеки символ от сорс символната таблица, с изключение на символите ' , \ и символ за нов ред;

▪ Escape-последователност;

\' – Апостроф
\" – Кавички¹
\? – Въпросителен знак¹
\ – Обратно наклонена черта
\a – Звуков сигнал
\b – Преместване на курсора една позиция назад
\f – Преместване на курсора на нова страница
\n – Преместване на курсора на нов ред
\r – Връщане на курсора в началото на текущия ред
\t – Хоризонтална табулация
\v – Вертикална табулация
\xHH..H – Символ, зададен чрез шестнайсетичен код. Стойността HH..H трябва да бъде в диапазона от стойности на тип **unsigned char** за нормални символни константи и в диапазона от стойности на беззнаковия тип, съответстващ на тип `wchar_t` за широки символни константи.

Забележка¹: Тези символи могат да се използват в символна константа и без обратно наклонената черта, т.е..

'\"' ≈ '\"' '\?' ≈ '?'

▪ Универсален символ^{C99}.

\uXXXX
\UXXXXXXXX

Нормалните символни константи имат тип **int**. Ако нормална символна константа съдържа един еднобайтов символ, компилаторът го заменя със съответния числов код от изпълняващата символна таблица. Например ако изпълняващата символна таблица е ASCII, компилаторът заменя символната константа 'a' с числото 97.



Нормалните символни константи имат тип **int.**

Числовата стойност на нормална символна константа, съдържаща повече от един символ (например 'ab'), или символ или escape-последователност, които нямат

еквивалент в изпълняващата символна таблица, зависи от имплементацията.

Пример:

```
char digits[]={ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
```

Широките символни константи имат тип **wchar_t**. Ако широка символната константа съдържа един многобайтов символ, компилаторът го заменя със стойността, която стандартната функция **mbtowc()** би върнала за този символ.

Числовата стойност на широка символна константа, съдържаща повече от един многобайтов символ (например L'ху'), или многобайтов символ или escape-последователност, които нямат еквивалент в изпълняващата символна таблица, зависи от имплементацията.



Освен ако не е абсолютно необходимо, избягвайте използването на многосимволни константи. Използването на такива константи намалява преносимостта на програмата.

Пример:

```
wchar_t warray[] = {L'a', L'12', L'\xF82', L'\u03B2'};
```

24.4 Низови константи

Низовата константа е последователност от нула или повече многобайтови символи затворени в кавички. ANSI C стандартът дефинира два типа низови константи: нормални и широки. Широките низови константи се предхождат от буква L.

"последователност-символи_{опц}"

L "последователност-символи_{опц}"

където

последователност-символи_{опц}

Може да включва:

- Всеки символ от сорс символната таблица, с изключение на символите `"`, `\` и символ за нов ред;
- Escape-последователност;

`\'` – Апостроф¹

`\"` – Кавички

`\?` – Въпросителен знак

`\\` – Обратно наклонена черта

`\a` – Звуков сигнал

`\b` – Преместване на курсора една позиция назад

`\f` – Преместване на курсора на нова страница

`\n` – Преместване на курсора на нов ред

`\r` – Връщане на курсора в началото на текущия ред

`\t` – Хоризонтална табулация

`\v` – Вертикална табулация

`\xHH..H` – Символ, зададен чрез шестнайсетичен код. Стойността HH..H трябва да бъде в диапазона от стойности на тип `unsigned char` за нормалните символни константи и в диапазона от стойности на беззнаковия тип, съответстващ на тип `wchar_t` за широки символни константи.

Забележка¹: Този символ може да се използва в низова константа и без обратно наклонената черта, т.е..

`"\"` $\approx$ `"\"`

- Универсален символ^{C99};

`\uXXXX`

`\UXXXXXXXX`

- Нищо – низова константа без символи се нарича празна низова константа.

Символите на низовата константа се съхраняват последователно в паметта на компютъра и компилаторът автоматично добавя нулев символ като последен символ, т.е. . низовата константа се съхранява в паметта като статичен

масив от символи. Нулевият символ е символ, чийто битове са нули и се означава с `\0`. Празна низова константа се състои от един символ - нулевия символ.

Пример:

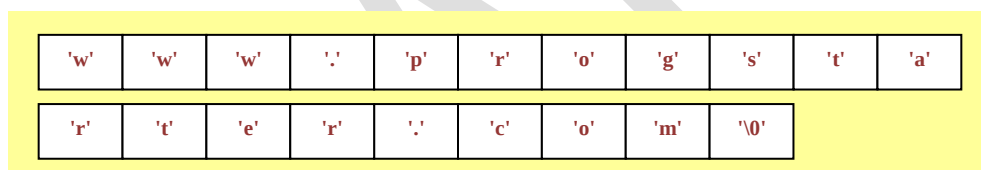
`""`



`"\""`



`"www.progstarter.com"`



Нормалната низова константа има тип `char[n+1]` (по-точно има тип `char (*)[n+1]`, който се интерпретира като указател към `char`-масив от `n+1` елемента, където `n` е броят на символите в низовата константа).

Широката низова константа има тип `wchar_t[n+1]` (по-точно има тип `wchar_t (*)[n+1]`, който се интерпретира като указател към `wchar_t`-масив от `n+1` елемента, където `n` е броят на символите в широката низовата константа).

Низовите константи могат да се използват за:

- Инициализатори на (широко)символни масиви;

Пример:

```
char docPath[] = "\\Program Files\\Progstarter\\doc";
wchar_t docPath[] = L"\\Program Files\\Progstarter\\doc";
```

- Инициализатори на (широко)символни указатели;

Пример:

```
char * docPath = "\\Program Files\\Progstarter\\doc";
wchar_t * docPath = L"\\Program Files\\Progstarter\\doc";
```

- Аргументи на библиотечни и потребителски функции.

Компилаторите са свободни да съхраняват низовете в ROM (Read Only Memory – Памет Само за Четене). Също така ANSI C стандартът позволява използването на една и съща памет за еднакви низове.



Не модифицирайте символи на низ. Това може да доведе до неочаквани резултати по време на изпълнение на програмата.

```
char * p = "house";
*p = 'm'; /*!!! Не е добра идея */
```

Твърде дълги низове могат да бъдат разположени на няколко реда с добавянето на обратно наклонена черта \ в края на всеки ред от низа без последния. Всеки интервал, добавен в началото на ред от низа, става част от него.

Пример:

```
char info[] = "This is a string literal broken up \
in several lines";
```

Ако компилаторът срещне две низови константи, разделени само от интервали, хоризонтални табулации или нови редове, той ги свързва в една низова константа. Тази особеност също може да се използва за разбиване на дълги низове на поднизове и разполагането им на отделни редове.

Пример:

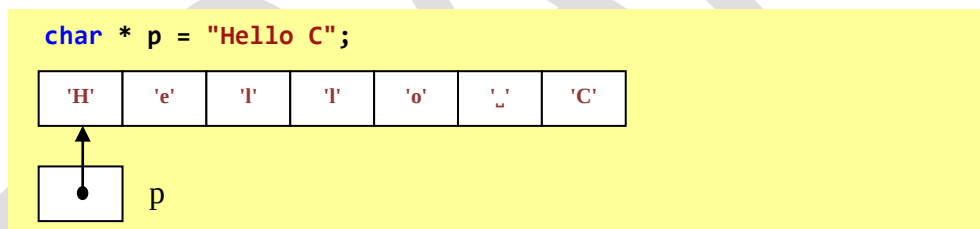
```
char info[] = "This is a string literal broken up "  
              "in several lines";
```

C99 позволява свързване на обикновена и широка низова константа. Резултатът е широка низова константа. В C89 подобна ситуация не е позволена и ще доведе до грешка при компилиране.

Пример:

```
wchar_t info[] = L"This is a string literal broken up "  
                "in several lines";
```

Ако низова константа се появи някъде, с изключение като аргумент на адресната операция `&`, `sizeof` или като инициализатор на (широко)символен масив, резултатът е адресът на първият символ на масива.



Операцията **sizeof(низова-константа)** връща като резултат броя на байтовете в низовата константа плюс нулевия символ. Ако низовата константа е нормална и съдържа само еднобайтови символи (каквито са символите от базовата символна таблица), броят на байтовете в низа ще отговаря на броя символите плюс нулевия символ. Във всички други случаи (нормална низова константа, съдържаща многобайтови символи, или широка низова константа) това няма да е вярно. Ако искате да определите само броя на символите в низовата константа (без нулевия символ), използвайте стандартната библиотечна функция **strlen()** за нормални низови константи и **wcslen()** за широки низови константи.

25 Операции

Главата ще бъде готова в следващо издание!!!

DRAFT

26 Преобразувания

Главата ще бъде готова в следващо издание!!!

DRAFT

27 C оператори

Главата ще бъде готова в следващо издание!!!

DRAFT

28 Функции

Главата ще бъде готова в следващо издание!!!

DRAFT

29 Предпроцесор

Табл.94 показва всички предпроцесорни команди и оператори.

№	Команда	Описание
1	#define	Дефиниране на макрос
2	#undef	Отдефиниране на макрос
3	#include	Вмъкване на текст от друг файл
4	#if	Директиви за условно компилиране
5	#ifdef	
6	#ifndef	
7	#else	
8	#endif	
9	#elif	
10	defined	Предпроцесорен оператор за условно компилиране
11	#line	Задаване на номер на ред
12	#error	Генериране на съобщение за грешка по време на компилиране
13	#pragma	Имплементационно-зависими директиви
14	#	Предпроцесорен оператор за създаване на низ
15	##	Предпроцесорен оператор за съединяване на предпроцесорни лексеми

Табл. 94 Предпроцесорни команди и оператори

30 Стандартна C библиотека

Езикът C предоставя голям набор от стандартни функции, наречени библиотечни, които можете да използвате наготово. Всяка библиотечна функция е декларирана в хедър, чието съдържание се прави налично в програмата чрез `#include` предпроцесорната директива.

Забележка¹: Хедърът не е задължително файл. На компилаторите е позволено да предефинират вътрешно съдържанието на хедърите. Въпреки това на практика C стандартните хедъри се съхраняват във файлове.

Освен декларации на стандартни библиотечни функции, хедър-файловете съдържат декларации на макроси и типове данни, дефинирани в самия стандарт.

По отношение на функционалността си библиотечните функции могат да се разделят на следните категории:

- Функции за вход-изход;
- Функции за обработка на низове и символи;
- Функции за преобразуване;
- Математически функции;
- Функции за управление на динамична памет;
- Управляващи функции;
- Функции за управление на многопоточност;
- Други функции.

ANSI C стандартът включва следните хедъри:

№	хедър-файл	кратко описание
1	<assert.h>	Дефинира макроса assert() за диагностични цели.
2	<complex.h> ^{C99}	Осигурява поддръжка за комплексни числа
3	<ctype.h>	Функции за обработка на символи
4	<errno.h>	Съобщения за грешки в библиотечни функции
5	<fenv.h> ^{C99}	Осигурява поддръжка на числа с плаваща запетая
6	<float.h>	Дефинира имплементационно-зависими граници за типове с плаваща запетая
7	<inttypes.h> ^{C99}	Форматни преобразувания на целочислени типове
8	<iso646.h> ^{C95}	Макроси-синоними на C логическите и битовите оператори
9	<limits.h>	Дефинира имплементационно-зависими граници за целочислени типове и други неща
10	<locale.h>	Дефинира средства за поддръжка на локализация
11	<math.h>	Функции за математически операции
12	<setjmp.h>	Дефинира средства за заобикаляне на нормалното извикване и връщане от функция
13	<signal.h>	Дефинира средства за обработка на сигнали
14	<stdalign.h> ^{C11}	Дефинира макроси за управление на подравняването
15	<stdarg.h>	Поддръжка на функции с променлив брой аргументи
16	<stdatomic.h> ^{C11}	Поддръжка на атомични типове
17	<stdbool.h> ^{C99}	Дефинира макроси за поддръжка на тип _Bool
18	<stddef.h>	Съдържа стандартни декларации на макроси и типове
19	<stdint.h> ^{C99}	Дефинира целочислени типове с фиксирана дължина
20	<stdio.h>	Функции за вход-изход
21	<stdlib.h>	Функции за обща употреба
22	<stdnoreturn.h> ^{C11}	Дефинира макрос noreturn
23	<string.h>	Функции за обработка на низове
24	<tgmath.h> ^{C99}	Дефинира поддръжка на математика с обобщени типове
25	<thread.h> ^{C11}	Функции за поддръжка на многопоточност

26	<time.h>	Функции за обработка на време и дата
27	<uchar.h> ^{C11}	Функции за поддръжка на Unicode символи
28	<wchar.h> ^{C95}	Функции за обработка на широки и многобайтови низове
29	<wctype.h> ^{C95}	Функции за обработка на широки символи

Табл. 95 Стандартни хедъри

Главата ще бъде готова в следващо издание!!!

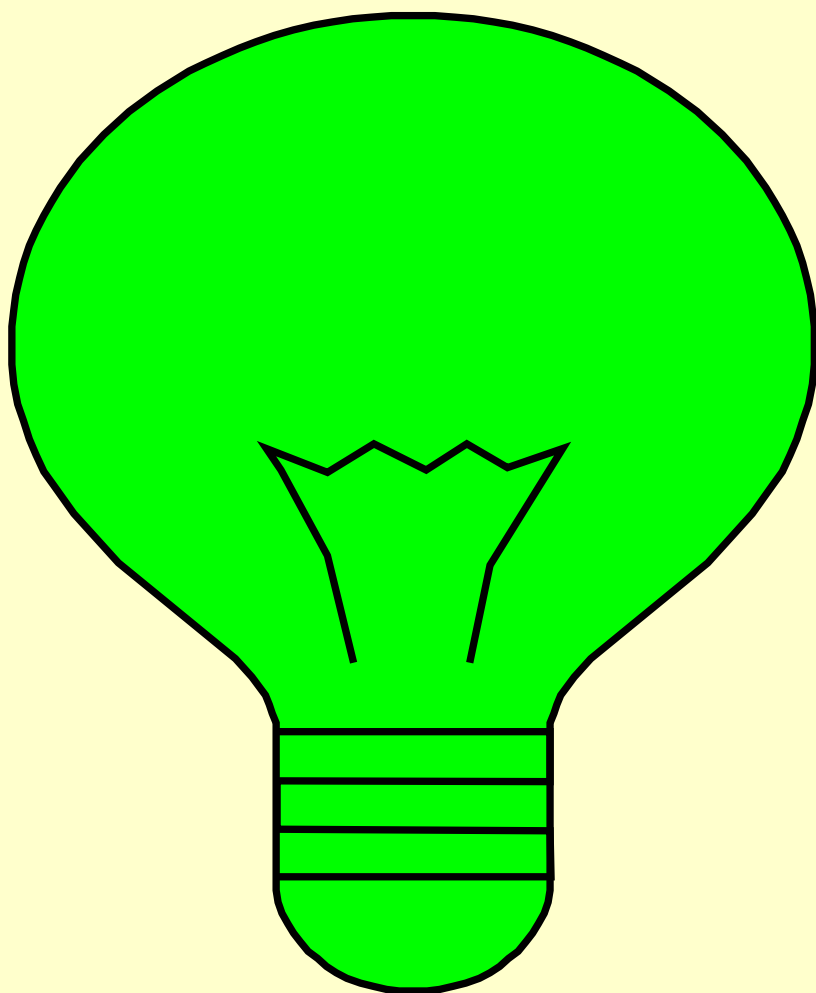
31 С програмни техники

Главата ще бъде готова в следващо издание!!!

DRAFT

Част III

Правила и съвети при писане на С-код



DRAFT

32 Въведение

В тази глава ще Ви запозная с примерен стил на писане на С-програми. Това включва:

- Организация на програмата;
- Организация на файл;
- Организация на функция;
- Коментари;
- Именуване на идентификатори;
- Форматиране на кода;
- Преносимост и производителност.

Ще Ви запозная също и с някои слабости в езика С, които да избягвате. Съществуват няколко стандарта, които дефинират набор от правила при използване на езика С (**MISRA-C**, **CERT-C**). За съжаление няма единен възприет стил на писане на С-програми. Различните компании имат различни разбирания и предпочитания. Въпреки това следването на определен стил и правила при писане на С-програми правят кода:

- Четлив;
- Лесен за поддръжка;
- Лесен за тестване, дебъгване и обновяване;
- Надежден.

Накрая ще завърша с представянето на различни софтуери за тестване на C-код.

Част III ще бъде готова в следващо издание!!!

DRAFT

33 Стил на писане на С-програми

DRAFT

34 Слабости в езика C

DRAFT

35 Тестване на C код

DRAFT

DRAFT

Част IV

Описание на развойните средства



DRAFT

36 Общи сведения

За да изпълните примерите, описани в тази книга, ще Ви е необходимо следното основно оборудване:

1. Персонален компютър;
2. Източник на постоянно напрежение $5V \div 20V$;
3. Основна платка **Z8Encore_MB01**;
4. Разширителни платки **Z8Encore_EBxx**;
5. Дебъгер **USB Smart Cable**;
6. Развойна среда **Zilog Developer Studio II**.

37 Описание на основната платка Z8Encore_MB01

Принципната електрическа схема на основната платка Z8Encore_MB01 е показана на Фиг.48. Тя съдържа следните основни компоненти:

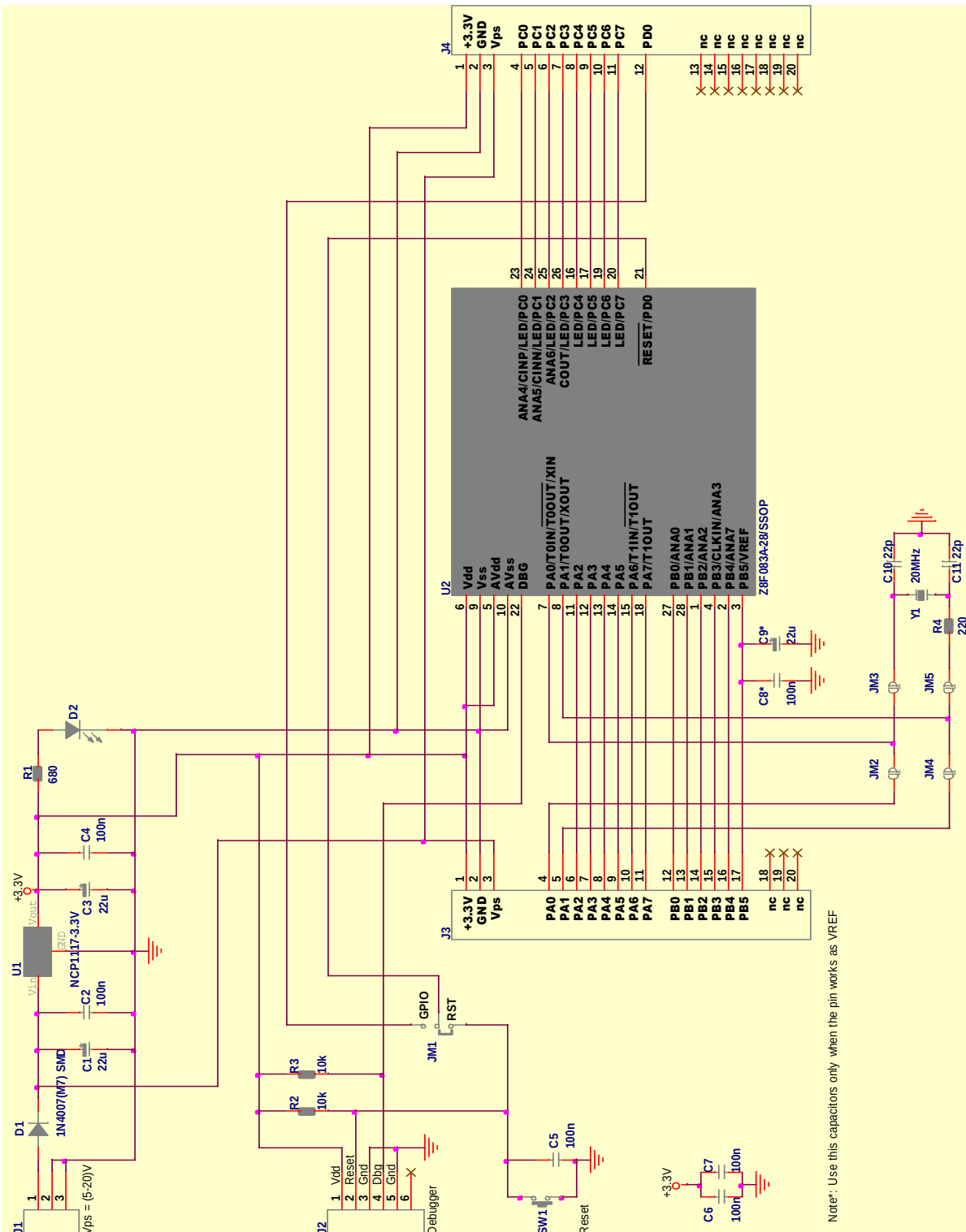
1. Микроконтролер U2 (**Z8F083AHJ020EG**) с изведени на рейки изводи.
2. Захранващ конектор J1. Захранването на микроконтролера е реализирано на базата на линейния стабилизатор на напрежение U1 (**NCP1117ST33T3G**).

	Източникът на входното напрежение може да бъде всеки мрежов адаптер с изходни напрежения в диапазона 5V÷20V (ø5.5мм x 2.1мм). 
--	--

3. Конектор J2 за свързване на дебъгера **USB SmartCable**.

	Когато използвате дебъгера, джъмперът JM1 трябва да е в позиция RST.
---	---

4. Бутон SW1 за начално установяване на микроконтролера.



Фиг. 48 Електрическа схема на основната платка Z8Encore_MB01

	Когато извод PD0 на микроконтролера се използва за начално установяване на микроконтролера, джъмперът JM1 трябва да е в позиция RST.
---	--

	Когато извод PD0 на микроконтролера се използва като цифров изход, след зареждане на фърмуера в микроконтролера с помощта на дебъгера, джъмперът JM1 трябва да е в позиция GPIO.
---	--

5. Рейка J3, на която са изведени следните сигнали и изводи на микроконтролера:

J3	Сигнал / Извод на микроконтролера
1	+3.3V
2	GND
3	Vps ¹
4	PA0
5	PA1
6	PA2
7	PA3
8	PA4
9	PA5
10	PA6
11	PA7
12	PB0
13	PB1
14	PB2

15	PB3
16	PB4
17	PB5
18	-
19	-
20	-

Изводите U2<PA0> и U2<PA1> се използват също и за свързване на кварцов/керамичен резонатор.

	Когато изводите U2<PA0> и U2<PA1> се използват като цифрови входове/изходи, е необходимо джъмперите JM2 и JM4 да бъдат затворени, а JM3 и JM5 отворени.
---	---

	Когато изводите U2<PA0> и U2<PA1> се използват за свързване на кварцов/керамичен резонатор, е необходимо джъмперите JM2 и JM4 да бъдат отворени, а JM3 и JM5 затворени.
---	---

	Кондензаторите C8 и C9 се използват само ако извод PB5 е конфигуриран като външен източник на опорно напрежение VREF за АЦП-то.
---	---

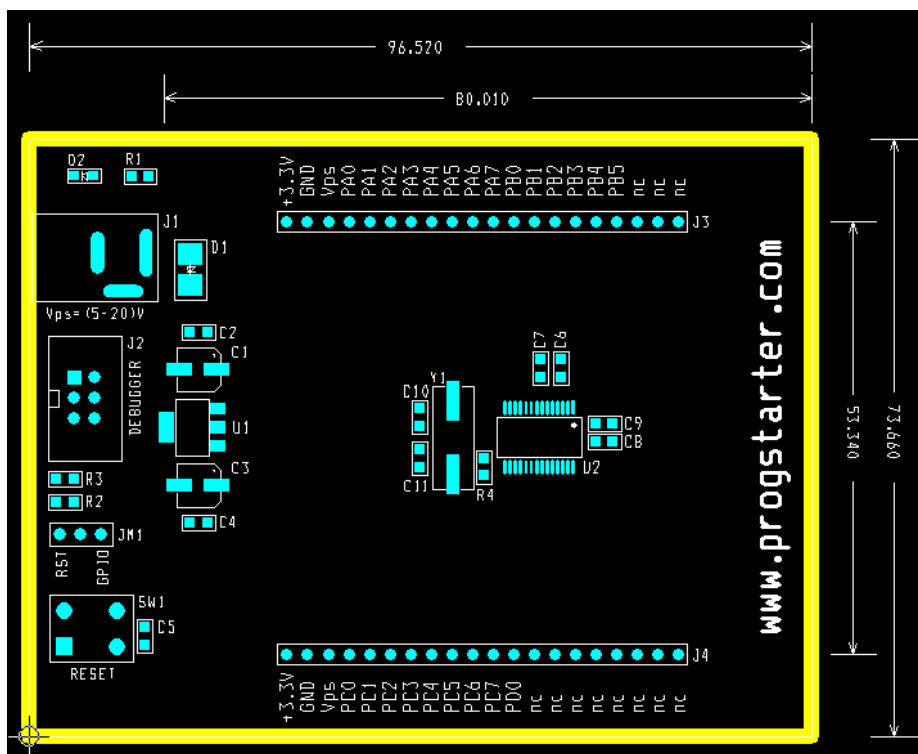
6. Рейка J4, на която са изведени следните сигнали и изводи на ми-кроконтролера:

J4	Сигнал / Извод на микроконтролера
1	+3.3V
2	GND
3	Vps ¹
4	PC0

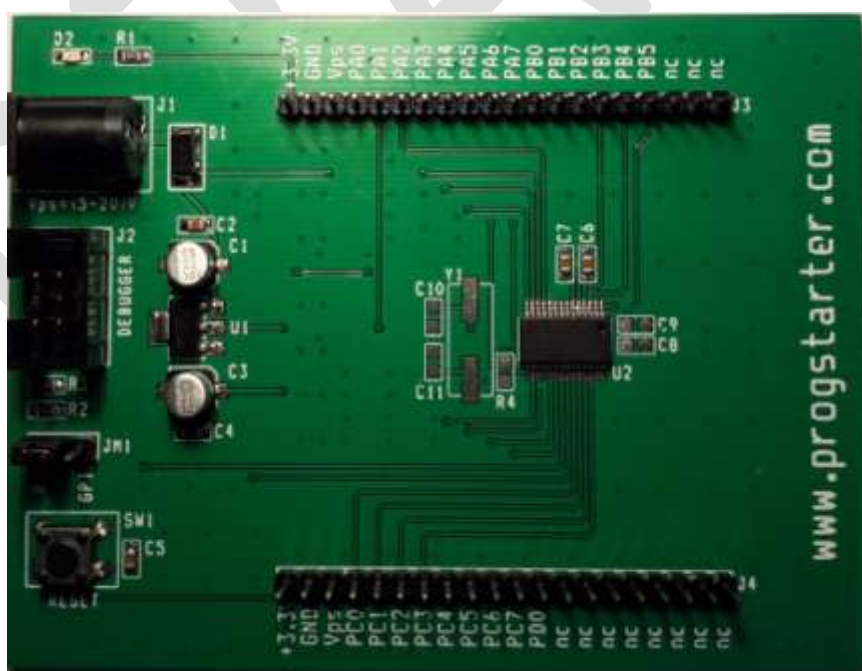
5	PC1
6	PC2
7	PC3
8	PC4
9	PC5
10	PC6
11	PC7
12	PD0
13	-
14	-
15	-
16	-
17	-
18	-
19	-
20	-

Забележка¹: Както може да видите от електрическата схема на Фиг.48, на изводи J3<3> и J4<3> е изведено входното захранващо напрежение (5V÷20V), подавано на захранващия конектор J1.

Фиг.49 и 50 показват разположението на елементите върху основната платка Z8Encore_MB01.



Фиг. 49 Разположение на елементите върху основната платка – поглед отгоре



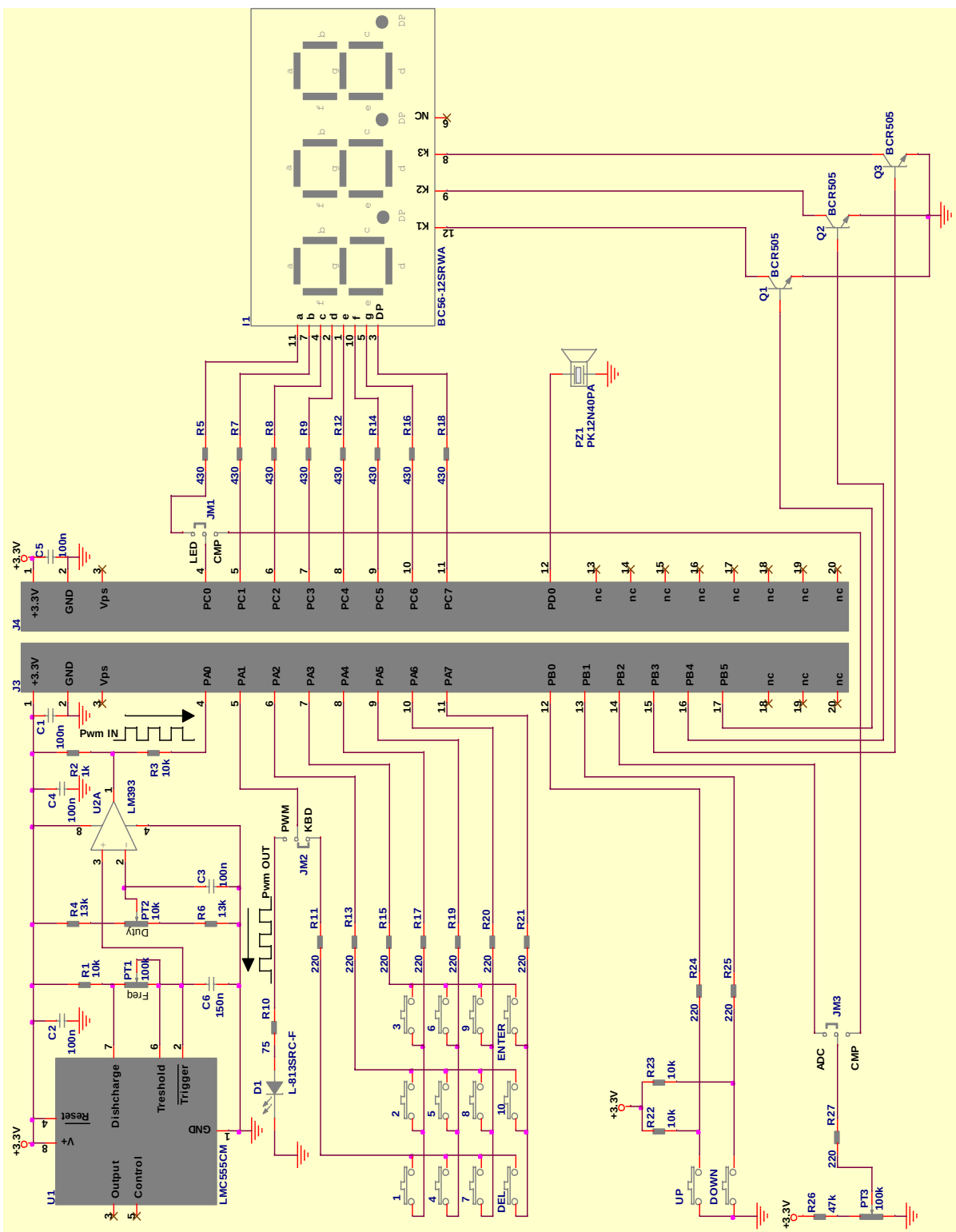
Фиг. 50 Основна платка Z8Encore_MB01 – поглед отгоре

38 Описание на разширителната платка Z8Encore_EB01(MB01)

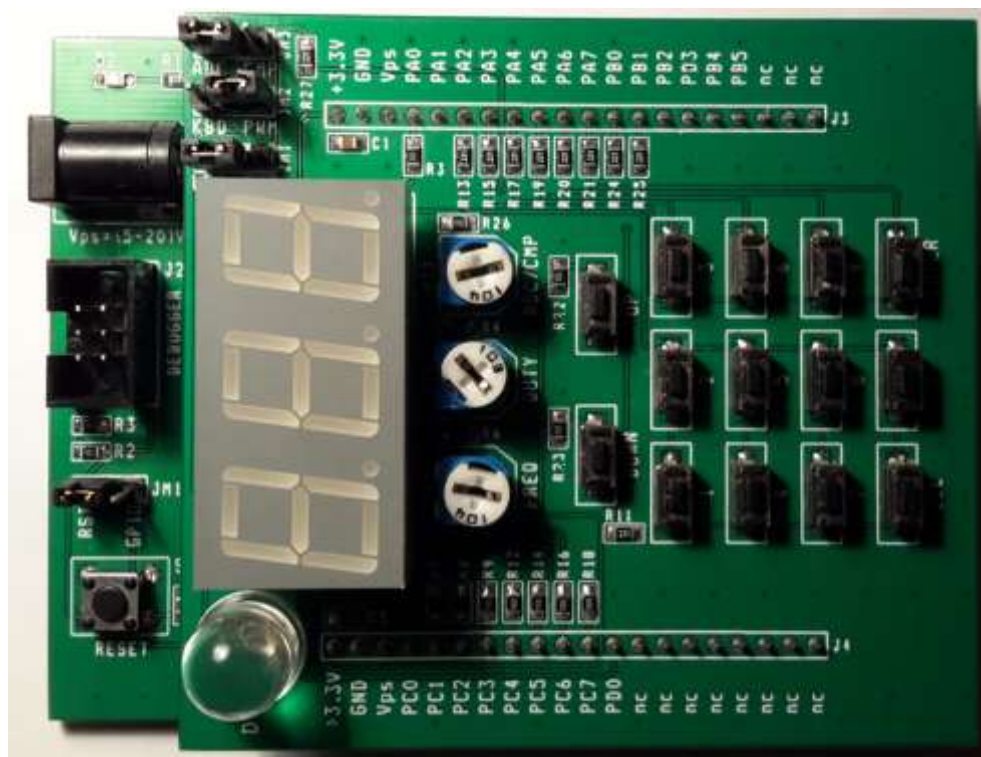
Принципната електрическа схема на разширителната платка Z8Encore_EB01(MB01) е показана на Фиг.51. Тя съдържа следните основни компоненти:

1. Генератор на правоъгълни импулси с регулируема честота $(50 \div 915)\text{Hz}$ и регулируем коефициент на запълване $(10 \div 90)\%$ на базата на интегралната схема **LCM555**;
2. Светодиод **D1**
3. Матрична клавиатура **4x3**;
4. Бутони **Up** и **Down**;
5. Източник на регулируемо напрежение $0 \div 2.1\text{V}$ с тример-потенциометър **PT3**.
6. Трицифрен седем-сегментен дисплей **BC56-12SRWA**;
7. Бъзер (звуков индикатор) **PZ1**.
8. Рейки **J3** и **J4**, предназначени за свързване на разширителната платка Z8Encore_EB01(MB01) към основната платка Z8Encore_MB01.

Фиг.52 и 53 показват разположението на елементите върху разширителната платка Z8Encore_EB01(MB01), а Фиг.54 показва свързването ѝ към основната платка Z8Encore_MB01.



Фиг. 51 Електрическа схема на разширителна платка Z8Encore_EB01(MB01)



Фиг. 54 Свързване на Z8Encore_EB01(MB01) към Z8Encore_MB01

39 Описание на развойната среда Zilog Developer Studio II

39.1 Общи сведения

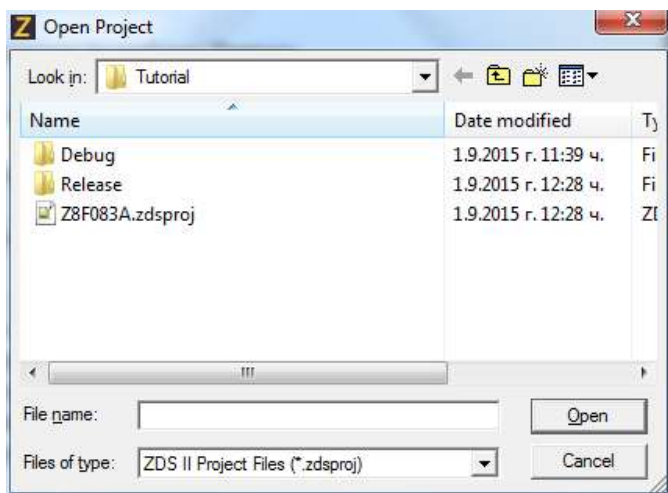
Zilog предоставя напълно безплатно развойната среда **Zilog Developer Studio II (ZDS II)**, която съдържа всичко необходимо за създаване на С-програми за микроконтролерите **Z8Encore!**. В това число влизат:

- Текстов редактор;
- Компилятор/Асемблер/Линкер;
- Симулатор;
- Дебъгер.

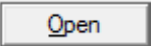
Преди да започнете със създаването на програма е необходимо да създадете проект. Следващите точки ще Ви дадат информация как да си създадете и конфигурирате проект, ще се запознаете с особеностите на С-компилятора и ще се научите как да използвате Симулатора и, за да тествате Вашите С-програми.

39.2 Отваряне на проект в ZDS II

1. Изберете **File→Open Project**. Отваря се диалоговата кутия **Open Project**.



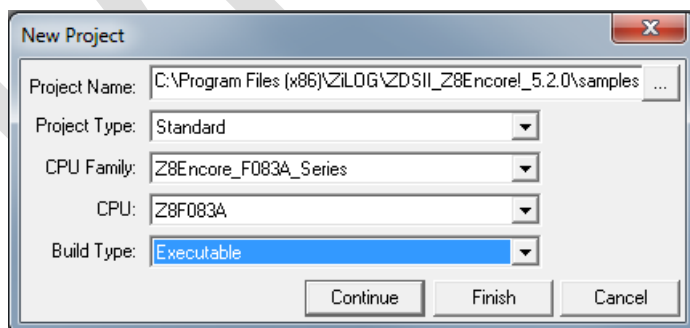
Фиг. 55 Open Project

- Изберете файла с разширение ***.zdsproj** и натиснете бутона .

39.3 Създаване на проект в ZDS II

39.3.1 Създаване на C проект

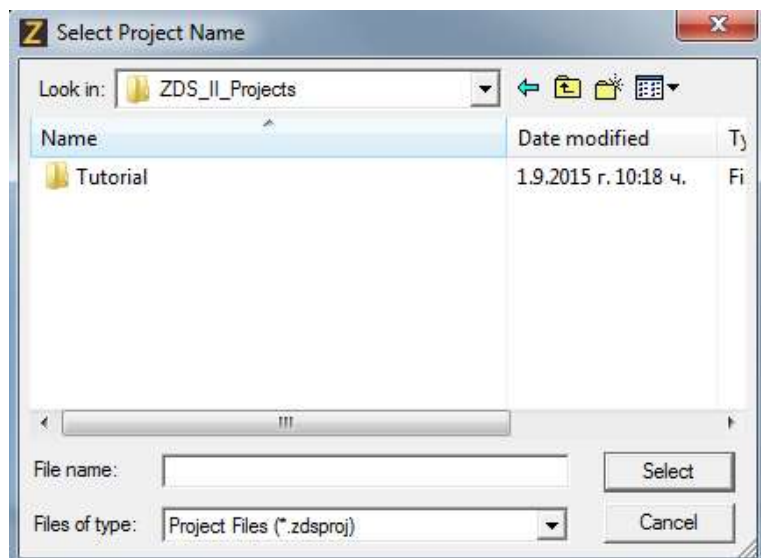
- Изберете **File** → **New Project**. Отваря се диалоговата кутия **New Project**.



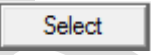
Фиг. 56 New Project

- Натиснете бутона . Отваря се диалоговата кутия

Select Project Name.



Фиг. 57 Select Project Name

3. Отидете в директорията, където искате да съхраните проекта.
4. В полето **File name** въведете името на проекта и натиснете бутона , за да се върнете в диалоговата кутия **New Project**.
5. В полето **Project Type** изберете **Standard**.
6. В полето **CPU Family** изберете фамилията микроконтролери.
7. В полето **CPU** изберете микроконтролер.
8. В полето **Build Type** изберете **Executable**.
9. Натиснете бутона .

Вече имате създаден проект. Следващата стъпка е да

добавите файл(ове) към проекта.

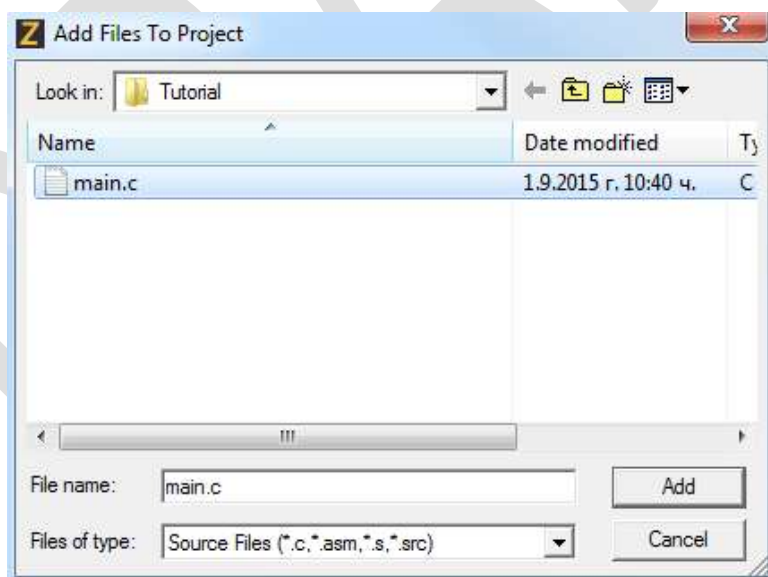
39.3.2 Добавяне на файл към проекта

Ако нямате все още създаден файл, може да го направите директно от средата ZDS II.

1. Изберете **File**→**New File**, за да си създадете файл.
2. Изберете **File**→**Save As** и съхранете в файла в желаната от Вас директория.

Добавянето на файл в проекта става по следния начин:

1. Изберете **Project**→**Add Files**. Отваря се диалоговата кутия **Add Files To Project**.



Фиг. 58 Add Files To Project

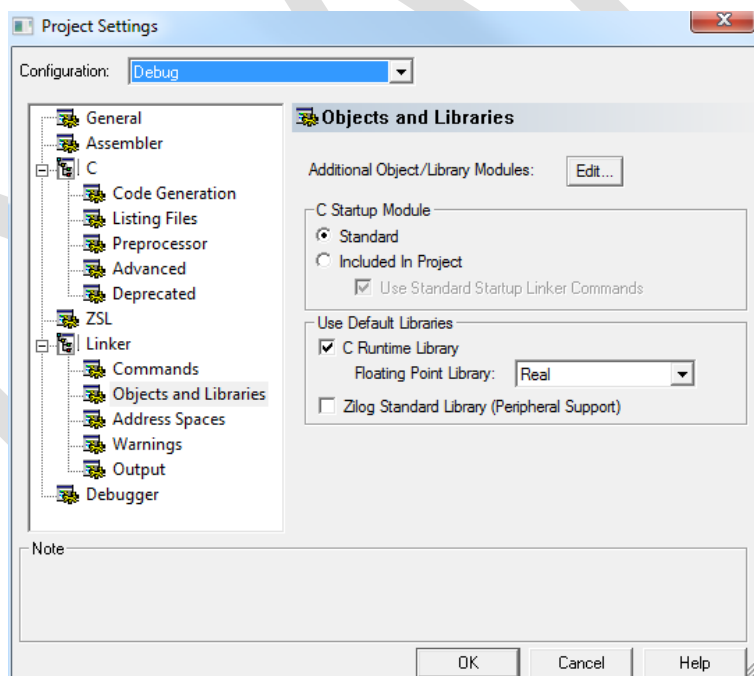
2. Отидете в директорията, където се намира файла, изберете го и натиснете бутона .

3. Повторете стъпки 1 и 2, за да добавите всички необходими файлове в проекта.

Следващата стъпка е да конфигурирате проекта.

39.3.3 Конфигуриране на проекта

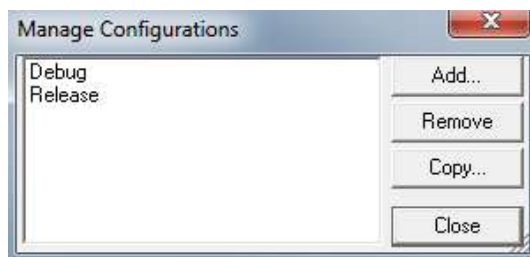
Конфигурирането на проекта става от **Project**→**Settings**. Отваря се диалоговата кутия **Project Settings**, която осигурява различни категории настройки, които са изброени вляво. Те включват Общи настройки (**General**), настройки на Асемблера (**Assembler**), настройки на Компилятора (**C**), настройки свързани със стандартната библиотека на Zilog (**ZSL**), настройки на Линкера (**Linker**) и настройки на Дебъгера (**Debugger**).



Фиг. 59 Project Settings

ZDS II позволява да създадете повече от една конфигурация.

Активната конфигурация се избира от полето **Configuration**. По подразбиране са предоставени две конфигурации – **Debug** и **Release**, но Вие може да добавяте и свои. Управлението на конфигурациите се осъществява от **Build→Manage Configurations**.



Фиг. 60 ManageConfigurations

Може изтривате конфигурации, да добавяте нови или да копирате настройките от една конфигурация в друга.

39.3.3.1 Общи настройки

Общите настройки са достъпни на **Project →Settings→General** (Фиг.61).

CPU Family

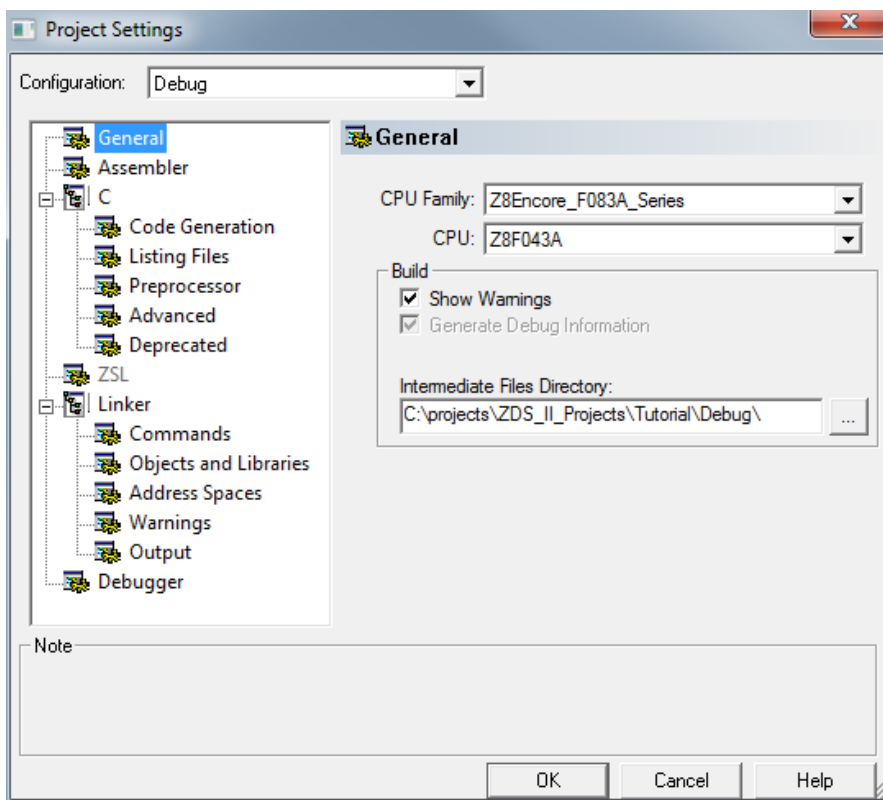
Падащ списък, изброяващ фамилияте микроконтролери.

CPU

Падащ списък, изброяващ микроконтролерите, принадлежащи на избраната в **CPU Family** фамилия.

Show Warnings

Контролира показването на предупредителни съобщения по



Фиг. 61 Project Settings → General

време всички фази на компилиране. Когато тази опция е избрана, всички предупредителните съобщения от Асемблера, Компилятора и Линкера се показват по време на компилиране на кода.

Generate Debug Information

Тази опция води до генериране на допълнителна информация, използвана от дебъгера. Разрешете я, ако смятате да използвате дебъгера, за да откривате и премахвате грешки във Вашия код. Обикновено разрешаването на тази опция води до умерено увеличаване на размера на кода.

Забележка: Опцията **Generate Debug Information** си взаимодейства с опцията **Limit Optimization for Easier Debugging** (вижте [Настройки](#)

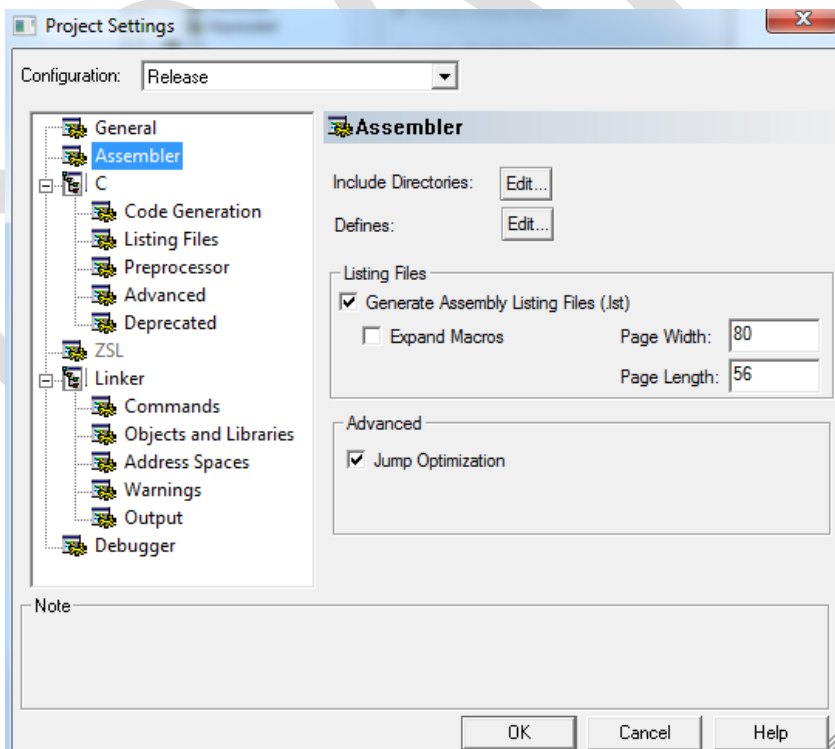
[на Компилятора](#)). Разрешаването на опцията **Limit Optimization for Easier Debugging** води до автоматично разрешаване и на опцията **Generate Debug Information** (в този случай опцията е затъмнена и не е достъпна за потребителя).

Intermediate Files Directory

Указва директорията, където се съхраняват междинните файлове (*.lst, *.obj, *.linkcmd, *.mak), генерирани в процеса на компилиране.

39.3.3.2 Настройки на Асемблера

Настройките на Асемблера са достъпни на **Project** → **Settings** → **Assembler**.



Фиг. 62 Project Settings → Assembler

Include Directories

Това поле позволява да укажете на Асемблера къде да търси файловете с разширение **.inc**¹.

Забележка: Това са файловете включени в асемблерните сорс-файлове с помощта на директивата INCLUDE. Например:

```
INCLUDE "eZ8.inc"

VECTOR RESET = startup

    ORG    100h
startup:

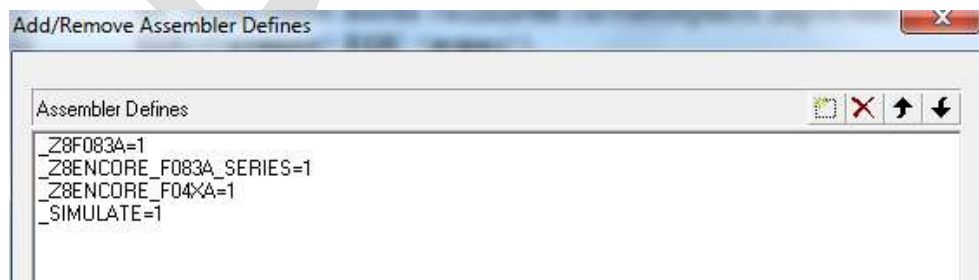
    ;асемблерен код

    JP     startup

END
```

Асемблерът първо проверява текущата директория, след това директориите, указани в полето **Include Directories**, и най-накрая в *<ZDS Installation Directory>\include\std*.

Defines



Фиг. 63 Project Settings → Assembler → Defines

Това поле позволява добавянето на символни имена на числови стойности в асемблерния код посредством следния синтаксис:

<символно-име> = <стойност>

Символните имена добавени в полето **Defines** са аналогични на символните имена създадени с асемблерната директива EQU (**<етикет> EQU <израз>**).

Generate Assembly Listing Files (.lst)

Когато тази опция е разрешена, Асемблерът генерира листинг файл (***.lst**) за всеки асемблерен сорс-файл. Този файл показва асемблерните инструкции и директиви, а също и съответните машинни инструкции и шестнайсетичните им адреси в Програмната Памет.

Expand Macros

Тази опция инструктира Асемблера да разгъва макросите в асемблерния листинг файл.

Page Width / Page Length

Дефинират максималния брой символи за ред и максималния брой редове в една страница на листинг файла.

Jump Optimization

Тази опция разрешава на Асемблера да замени инструкциите за относителен преход (**JR** и **DJNZ**) с инструкции за абсолютен преход, когато етикетът, към който се осъществява прехода е:

- извън обхвата -128 ÷ +127 на тези инструкции. На пример асемблерът ще замени инструкцията

DJNZ r0, lab

C

DJNZ r0, lab1

JR lab2

lab1:**JP** lab

lab2:

- външен за асемблерния файл. В този случай асемблерът винаги допуска, че целевият адрес е извън обхвата на тези инструкции.



Ако тази опция е забранена и възникне една от горните ситуации, ще получите съобщение за грешка в процеса на асемблиране на кода.



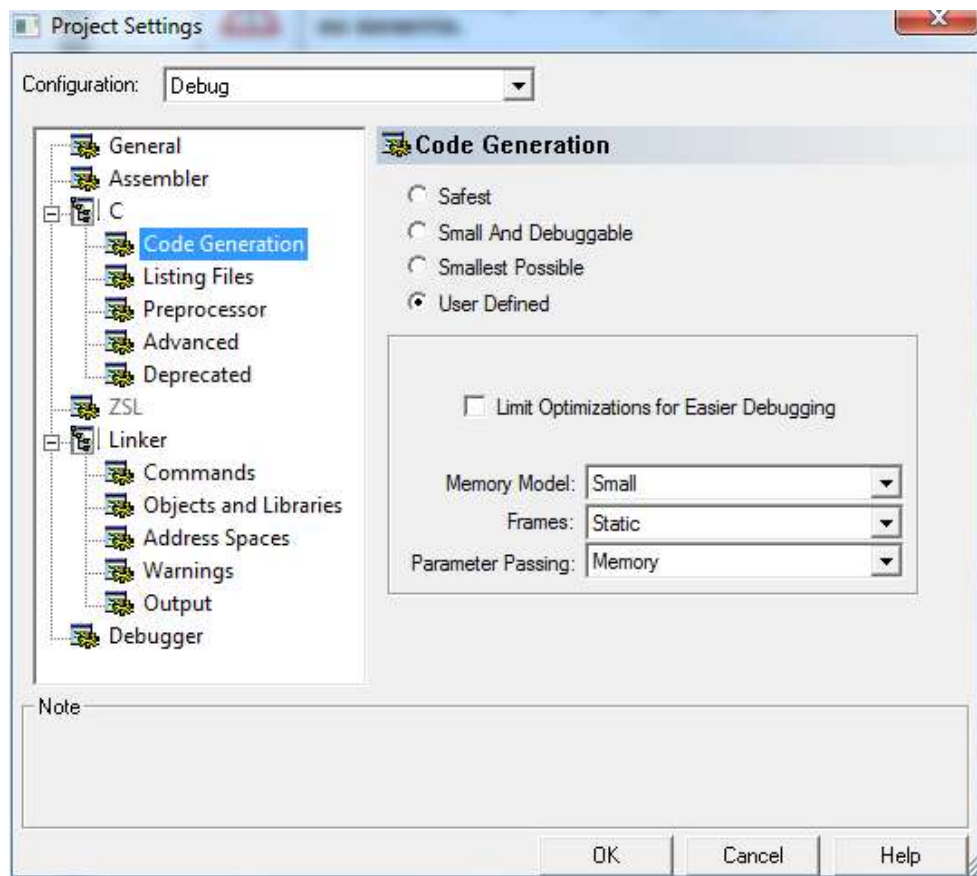
За предпочитане и винаги да разрешавате тази опция и да оставите асемблера да извършва тези замени.

39.3.3.3 Настройки на Компилятора

Настройките на Компилятора са достъпни на **Project** → **Settings** → **C**. Те са разделени на следните подкатегории:

- Code Generation;
- Listing Files;
- Preprocessor;
- Advanced;
- Deprecated.

CODE GENERATION



Фиг. 64 Project Settings → C → Code Generation

Limit Optimizations for Easier Debugging

Разрешаването на тази опция принуждава компилатора да изключи определени оптимизации. Тези оптимизации могат да доведат до объркване в процеса на дебъгване на кода.



Разрешаването на тази настройка води до генерирането на по-голям код и употреба на повече памет за данни.

Забележка: Опцията **Limit Optimization for Easier Debugging** си

взаимодейства с опцията **Generate Debug Information** (вижте [Общи настройки](#)). Разрешаването на опцията **Limit Optimization for Easier Debugging** води до автоматично разрешаване и на опцията **Generate Debug Information** (в този случай опцията е затъмнена и не е достъпна за потребителя).

Memory Model

Тази настройка позволява избирането на два модела на паметта – малък (**Small**) и голям (**Large**).



Не всички микроконтролери поддържат голям модел на паметта.

Пълна информация за моделите на паметта ще намерите в [Модели на паметта](#).

Frames

Тази настройка позволява избирането на два типа контекст на функциите – статичен (**Static**) и динамичен (**Dynamic**).

Пълна информация за контекстта на функциите ще намерите в [Контекст на функциите](#).

Parameter Passing

Тази настройка позволява избирането на два начина на съхраняване на параметрите на функция: **Memory** и **Register**.

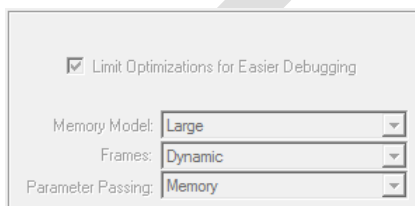
Ако изберете **Memory**, параметрите се съхраняват в стека при динамичен контекст (вижте [Механизъм на извикване на функции: Динамичен контекст](#)) и в статична памет (Регистрова Памет) при статичен контекст (вижте [Механи-](#)

зъм на извикване на функции: Статичен контекст).

Ако изберете **Register**, скаларните параметри се съхраняват в работните регистри R8÷R13, а всички останали се съхраняват в паметта (вижте Механизъм на извикване на функции: Съхраняване на параметри в регистри).

Safest

Изборът на тази опция води до следното:



☒ Limit Optimizations for Easier Debugging

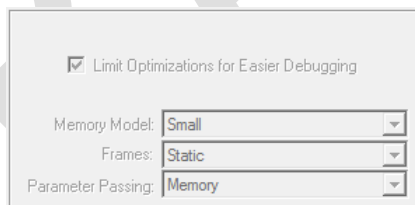
Memory Model: **Large**

Frames: **Dynamic**

Parameter Passing: **Memory**

Small And Debuggable

Изборът на тази опция води до следното:



☒ Limit Optimizations for Easier Debugging

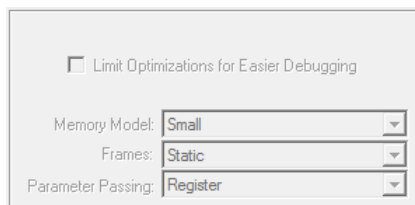
Memory Model: **Small**

Frames: **Static**

Parameter Passing: **Memory**

Smallest Possible

Изборът на тази опция води до следното:



☐ Limit Optimizations for Easier Debugging

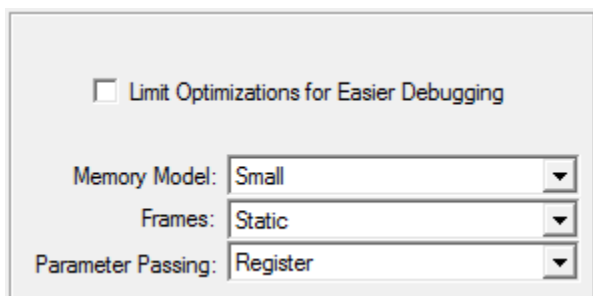
Memory Model: **Small**

Frames: **Static**

Parameter Passing: **Register**

User Defined

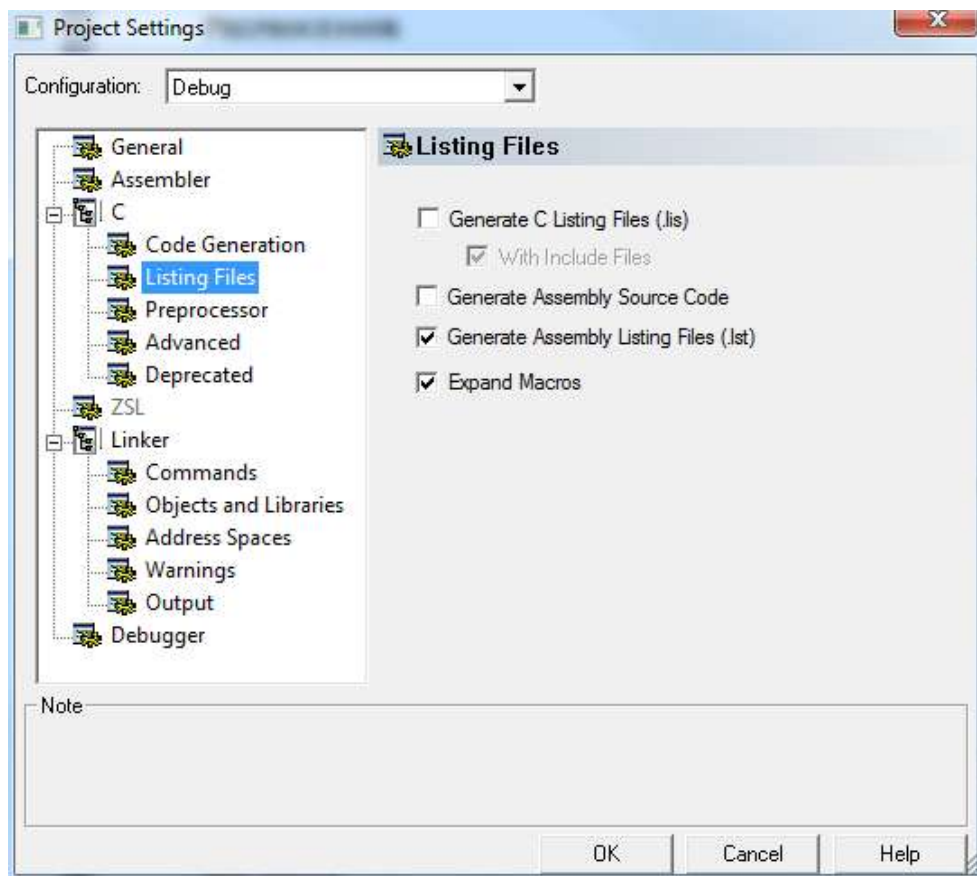
Когато тази опция е избрана, следните настройки са достъпни за конфигурация от потребителя:



A dialog box titled "User Defined" with a light gray background. It contains a checkbox labeled "Limit Optimizations for Easier Debugging" which is currently unchecked. Below the checkbox are three dropdown menus. The first is labeled "Memory Model:" and has "Small" selected. The second is labeled "Frames:" and has "Static" selected. The third is labeled "Parameter Passing:" and has "Register" selected. Each dropdown menu has a small downward arrow on its right side.

☐ Limit Optimizations for Easier Debugging
Memory Model: Small
Frames: Static
Parameter Passing: Register

LISTING FILES



Фиг. 65 Project Settings → C → Listing Files

Generate C Listing Files (.lis)

Ако тази опция е разрешена, Компиляторът генерира листинг файл (*.lis) за всеки C сорс-файл.

With Include Files

Ако тази опция е разрешена, Компиляторът вмъква съдържанието на всички хедър-файлове в листинг файла. Това може да е полезно за откриване на грешки в хедър-файловете.

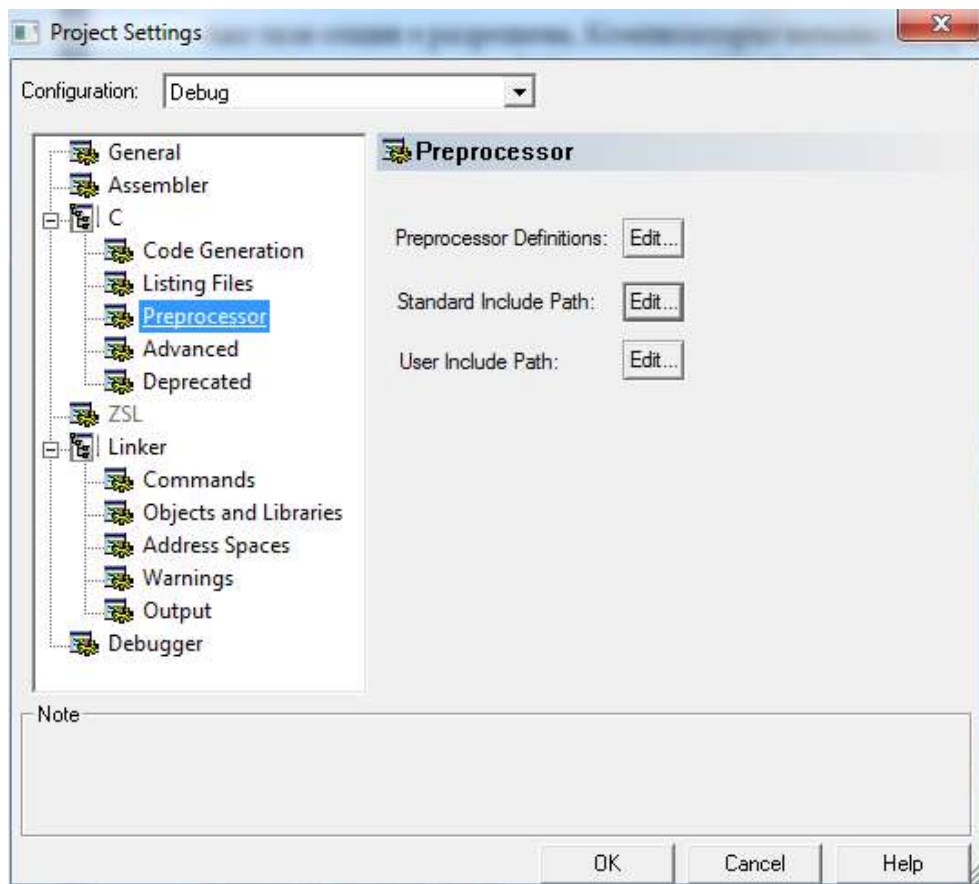
Generate Assembly Source Code

Ако тази опция е разрешена, Компиляторът генерира за всеки C сорс-файл, съответен асемблерен файл.

Generate Assembly Listing Files (.lst)

Ако тази опция е разрешена, Компиляторът генерира за всеки C сорс-файл, съответен асемблерен листинг файл. Този файл съдържа C сорс-кода и съответстващия му асемблерен код, генериран от Компилятора.

PREPROCESSOR



Фиг. 66 Project Settings →C→Preprocessor

Preprocessor Definitions

Това поле позволява дефинирането на макроси в началото на всеки файл преди всеки друг ред, аналогично на използването на предпроцесорната директива `#define`.

Standard Include Path

Това поле позволява да укажете на Компилятора къде да търси хедър-файловете **<хедър-файл.h>**.

Стандартните и нестандартните хедър-файлове, предоставе-

ни от Zilog се намират в

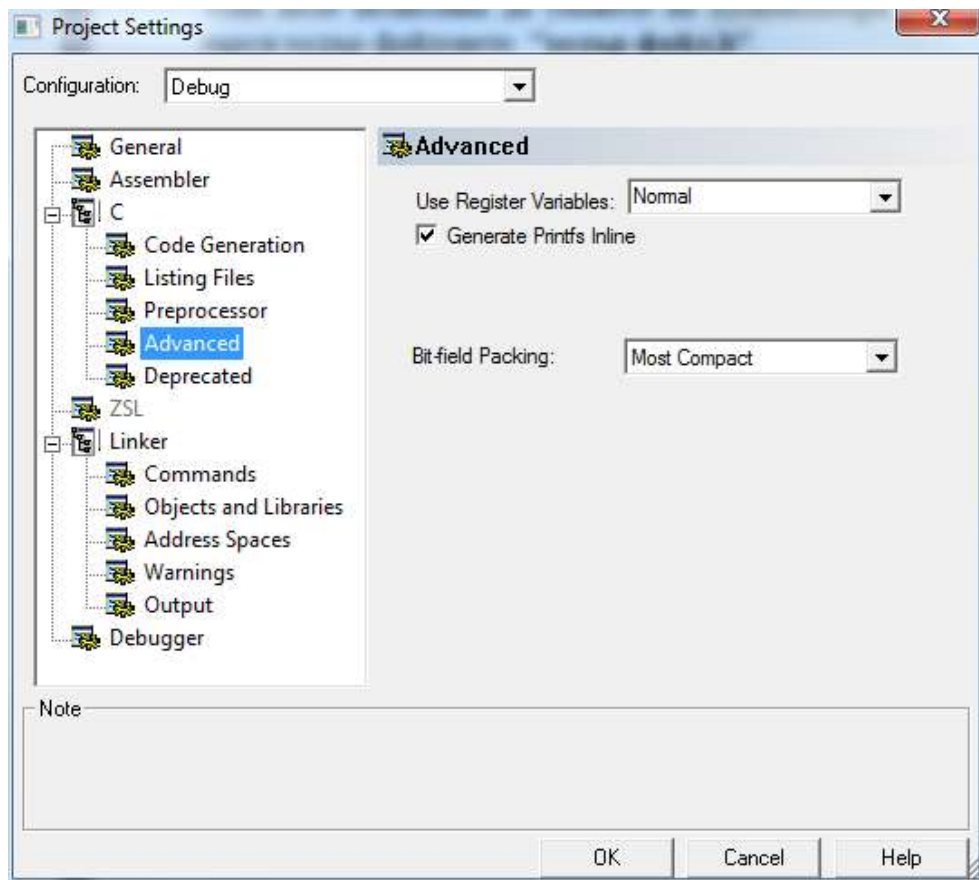
<ZDS Installation Directory> \include\std

<ZDS Installation Directory> \include\zilog

User Include Path

Това поле позволява да укажете на Компилятора къде да търси хедър-файловете **"хедър-файл.h"**.

ADVANCED



Фиг. 67 Project Settings → C → Preprocessor

Use Register Variables

Избирането на **Normal** или **Aggressive** позволява на Компилятора да съхрани локалните променливи в регистри, а не в стека, когато е възможно. Това обикновено води до по-малък и бърз код. Въпреки това в приложения с функции, съдържащи голям брой локални променливи, ефектът е противоположен.

Забележка: Опцията **Use Register Variables** си взаимодейства с опцията **Limit Optimization for Easier Debugging** (вижте [Настройки на](#)

[Компилятора](#)). Разрешаването на опцията **Limit Optimization for Easier Debugging** води до автоматично изключване на опцията **Use Register Variables** (в този случай опцията е затъмнена и не е достъпна за потребителя).

	Използването на регистрови променливи усложнява дебъгването на кода по два начина. Единият начин е, че регистровите променливи е вероятно да бъдат оптимизирани от Компилятора. За да наблюдавате оптимизирани променливи при дебъгване на кода, може да направите едно от следните неща: ▪ Разрешете опцията Limit Optimization for Easier Debugging; ▪ Изключете опцията Use Register Variables (избегнете Off от падащото меню); ▪ Променете кода, така че въпросните променливи да станат глобални вместо локални. Другият начин е, че един и същ регистър може да се използва за различни променливи или временни резултати в различни моменти от изпълнението на кода. Тъй като Дебъггерът не винаги „съзнава“ тези множествени употреби, понякога стойността на регистровата променлива, наблюдавана в Watch-прозореца, няма да съответства на действителната употреба променливата.
---	--

Generate Printfs Inline

Нормално, форматиращият низ на функциите `printf()` и `sprintf()` се анализира при тяхното извикване, т.е. по време на изпълнение на програмата. Когато опцията **Generate Printfs Inline** е разрешена, форматиращият низ се анализира по време на компилиране и се генерира код, който директно извиква помощните функции от по-ниско ниво. Това значи-

телно намалява кода, тъй като функциите, които се занимават с анализирането на форматиращия низ се изключват от кода.

Bit-field Packing

Ако е избран нормален режим (**Normal**) на пакетиране на битовите полета, се запазва декларирания тип на всичките индивидуални битови полета. Например в следната декларация

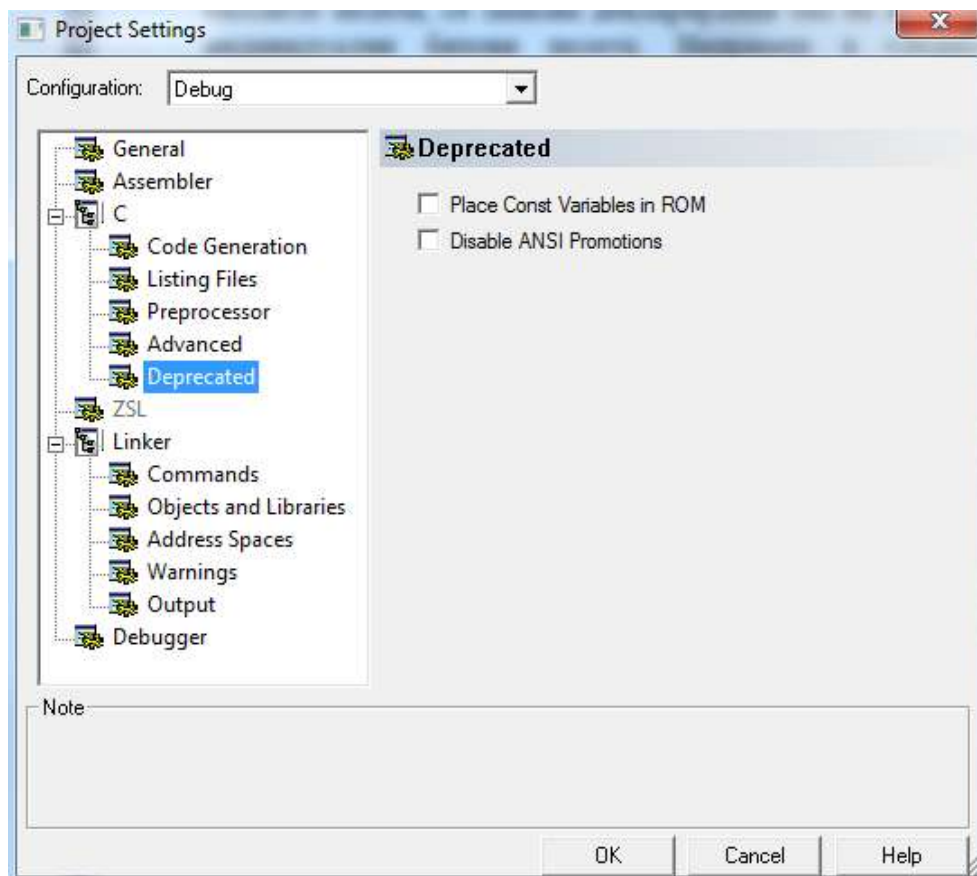
```
struct
{
    char    bf1 : 6;
    short   bf2 : 9;
    long    bf3 : 17;
} bf_struct;
```

Компиляторът ще задели 1 байт за **bf1**, 2 байта за **bf2** и 4 байта за **bf3**.

Ако е избран най-компактен режим (**Most Compact**) на пакетиране на битовите полета, Компиляторът ще пакетира битовите полета възможно най-сбито.

Режимът **Backward Compatible** използва по-сложна схема на пакетиране на битовите полета, използвана в по стари версии на Z8 Encore! Компилятора (преди версия 4.11.0).

DEPRECATED



Фиг. 68 Project Settings→C→Deprecated

Place Const Variable in ROM

Ако тази опция е разрешена, Компиляторът поставя променливите, дефинирани с квалификатора **const**, в ROM адресното пространство.



За предпочитане е тази опция да не се използва. Използвайте допълнителната ключова дума **rom**, преоставена от Zilog, когато искате да поставяте данни в ROM паметта (вижте [Нови ключови думи за спецификатори на клас-памет](#)).

Disable ANSI Promotions

Ако тази опция е разрешена, Компиляторът забранява автоматичното повишаване на типовете **char** и **short** до тип **int**. Това преобразуване е част от тъй нар. целочислено повишаване на типовете, дефинирано от ANSI C стандарта.

Ако тази опция е забранена, Компиляторът извършва целочислено повишаване на типовете само ако е необходимо. Например в следния израз, автоматичното повишаване на типовете не влияе на резултата:

```
signed char a, b, c;  
  
a = 126;  
b = 10;  
  
c = a + b; // Резултат -120  
  
c = (int)a + (int)b; // Резултат -120
```

В следващия пример обаче, се получават различни резултати в зависимост от това дали се извършва повишаване на типовете или не.

```
signed char a, b;  
signed int c;  
  
a = 126;  
b = 10;  
  
c = a + b; // Резултат -120  
  
c = (int)a + (int)b; // Резултат 136
```



За предпочитане и тази опция да не се използва, тъй като поведението на Компилятора се отклонява от това, дефинирано в ANSI C стандарта и е източник на неочаквани грешки.

39.3.3.4 Настройки на ZSL-библиотеката

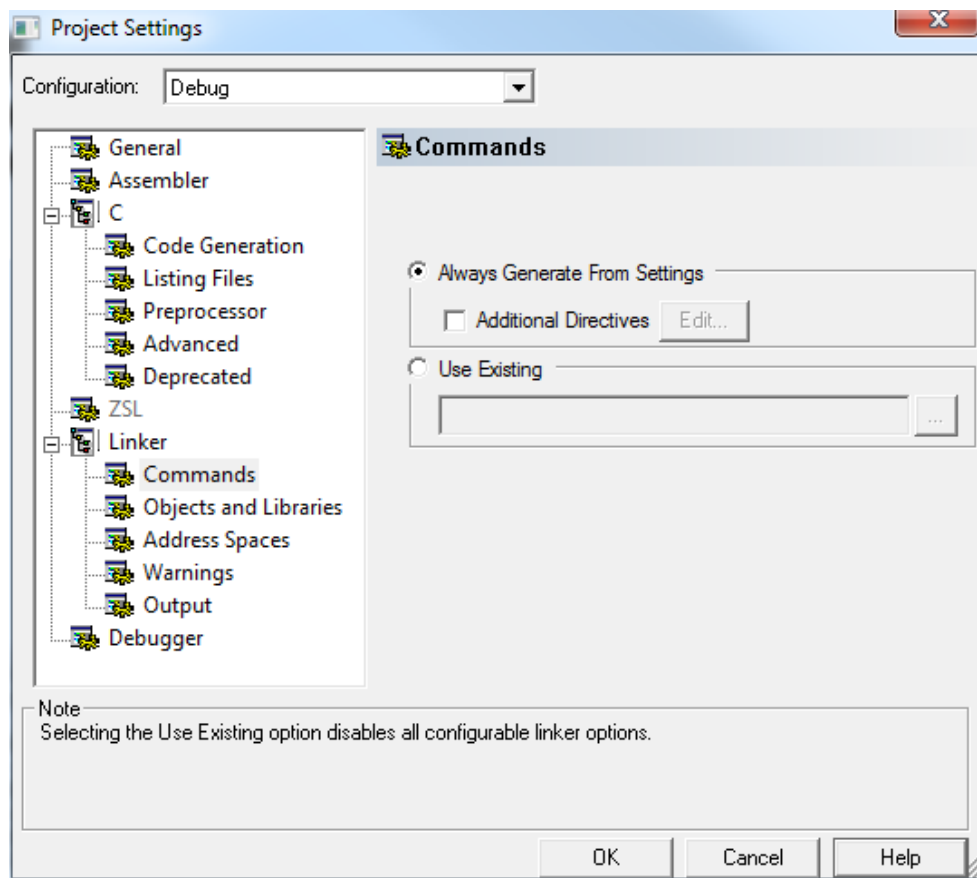
Тези настройки не са достъпни, когато е избран микроконтролера **Z8F083A**, затова няма да бъдат разглеждани в книгата. За повече информация се обърнете към вграденото помощно меню на средата **ZDS II (Help→Online Topics)**.

39.3.3.5 Настройки на Линкера

Настройките на Линкера са достъпни на **Project→Settings→Linker**. Те са разделени на следните подкатегории:

- Commands;
- Objects and Libraries;
- Address Spaces;
- Warnings;
- Output.

COMMANDS



Фиг. 69 Project Settings→Linker→Commands

Always Generate from Settings

Когато тази опция е избрана, линкер-командния файл¹ се генерира при всяко компилиране на проекта.

Забележка: Линкер-командният файл е специален файл, използван от Линкера в последната фаза на компилиране на проекта. Повече за Линкера и линкер-командния файл ще намерите в [Описание на Линкера](#).

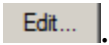
Това означава, че всички промени, които направите, такива

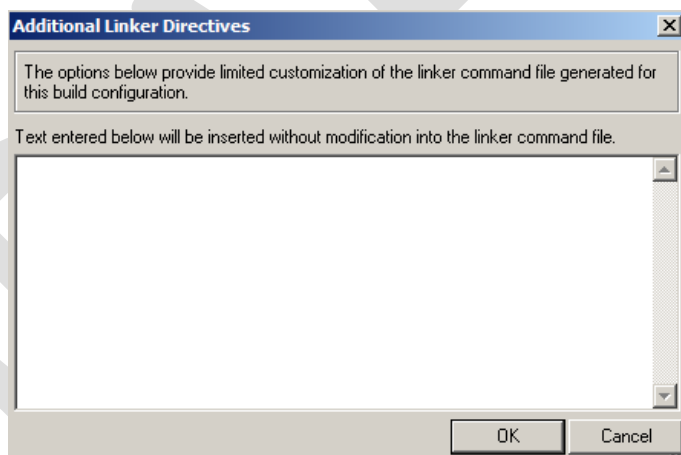
като добавяне на нови сорс-файлове в проекта или промяна на настройки, ще се отразят автоматично в линкер-командния файл.

Additional Directives

Тази опция Ви позволява да вмъквате допълнителни линкерни команди (директиви) към автоматично генерирания линкер-команден файл (вижте [Линкерни команди](#)).

1. Изберете **Additional Directives**.

2. Натиснете бутона .



3. Добавете нови директиви и натиснете **ОК**.

Ръчно добавените директиви винаги се поставят на едно и също място в линкер-командния файл: след повечето автоматично генерирани директиви и точно преди последната директива, която дава име на изпълнимия файл и модулите, които ще бъдат включени в процеса на компилация. По този начин ръчно вмъкнатите директиви отменят действието на автоматично добавените директиви, намиращи се преди тях. Изключение са директивите

CHANGE, RANGE и ORDER. Те не отменят действието на автоматично генерираните CHANGE, RANGE и ORDER порано във файла, а се комбинират с тях.

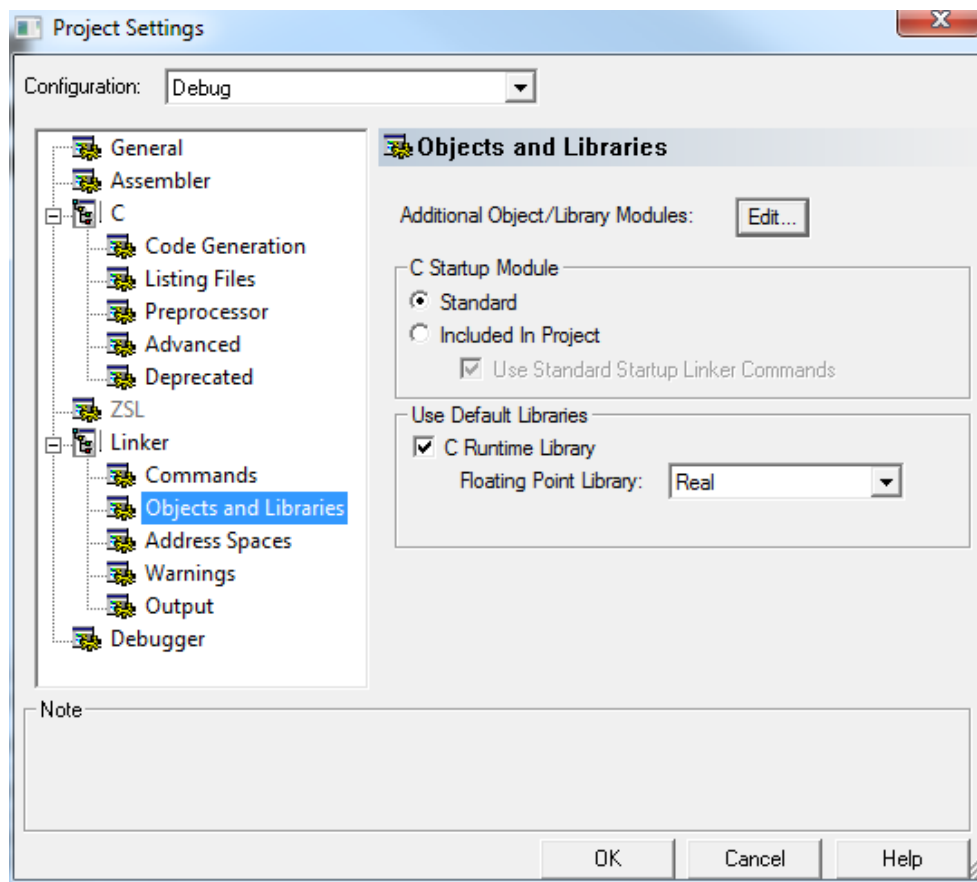
Use Existing

Тази опция Ви позволява да използвате Ваш собствен линкер-команден файл. Когато тази опция е избрана, много от проектните настройки се забраняват, тъй като нямат ефект в този случай.



Имайте предвид, че когато тази опция е избрана, някои промени в проекта, като добавяне на нови сорс-файлове, не се отразяват автоматично. В такива случаи тези промени трябва ръчно да бъдат добавени и в линкер-командия файл.

OBJECT AND LIBRARIES

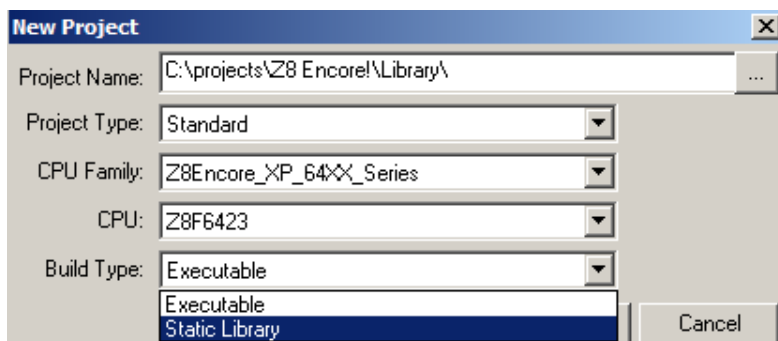


Фиг. 70 Project Settings→Linker→Objects and Libraries

Additional Object/Librari Modules

Използвайте тази опция, за да добавите обектни (*.obj) или библиотечни¹ (*.lib) файлове към Вашия проект.

Забележка: Библиотечният файл е съвкупност от един или повече обектни файлове. Когато създавате проект, Вие избирате какъв да бъде резултата от компилиране на проекта: изпълним или библиотечен файл. Изпълнимият файл е готов за зареждане в паметта на микроконтролера, докато библиотечният файл съдържа обектния код на функции, които могат да се преизползват и в други проекти.



Файлове, добавени чрез тази опция, се свързват преди библиотеките, осигурени от Zilog. Следователно ако има конфликт между символи в един от тези потребителски файлове и стандартната библиотека на Zilog, потребителските файлове са с по-висок приоритет и техните версии на символите ще бъдат използвани в процеса на свързване. Може да се възползвате от тази особеност в случай, че искате да използвате Ваша собствена версия на някоя стандартна библиотечна функция.

Standard

Когато тази опция е избрана, проектът използва предварително компилиран стандартен стартов файл, който идва заедно с развойната средата ZDS II. Ролята на стартовият файл е да извърши необходимите инициализации съгласно стандарта ANSI C плюс специфични за Z8Encore! конфигурации, като инициализиране на таблицата с векторите на прекъсване.

Средата предлага два стартови файлове в зависимост от това дали се използва малък или голям модел на паметта (вижте [Модели на паметта](#)). Стартовите файлове се намират в

<ZDS Installation Directory>\src\boot\common\startupX.asm.

където

X е s за малък модел и **l** за голям.

Included In Project

Когато тази опция е избрана, стандартният стартов модул не се добавя към проекта. В този случай Вие сте отговорни за добавянето на стартов код или чрез добавянето му в сорс кода или чрез предварително компилиран обектен файл, добавен в **Additional Object/Library Modules**.

Use Standard Startup Linker Commands

Когато тази опция е избрана, същите линкерни команди, които поддържат стандартния стартов файл, се вмъкват във Вашия линкер-команден файл, въпреки че сте избрали да включите в проекта Ваш собствен стартов файл.

C Runtime Library

Тази опция трябва да бъде избрана, ако във Вашия проект използвате извиквания към стандартни библиотечни функции.

Floating Point Library

- **Real**

Изберете тази опция в случай, че Вашия проект използва стандартни библиотечни функции с плаваща запетая.

- **Dummy**

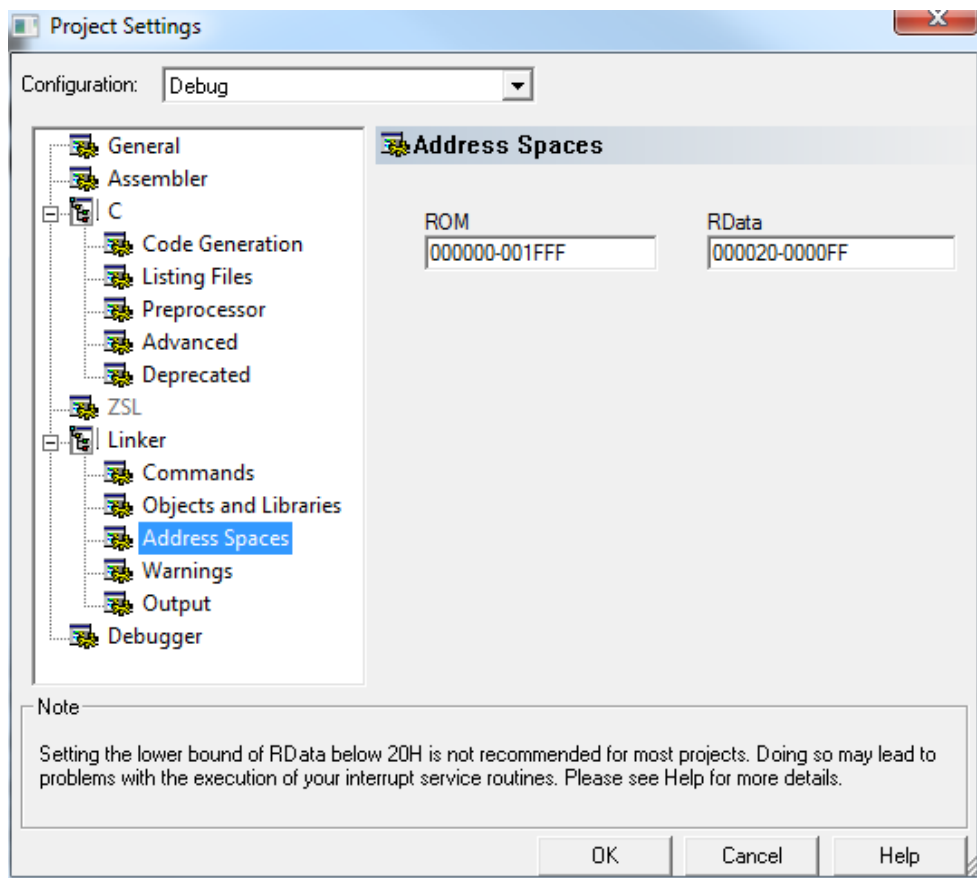
Когато тази опция е избрана, се използват „празни“ версии на стандартните библиотечни функции с плаваща запетая. Тези версии са създадени с цел да се намали „раздуването“ на кода, причинено от

извикванията на `printf()` и подобни на нея, като `sprintf()`. Те от своя страна извикват помощни функции с плаваща запетая в процеса на обработка на форматиращия низ. Тези извиквания не са необходими, освен ако наистина е необходимо да форматираме числа с плаваща запетая. За повечето ситуации този проблем е разрешен чрез опцията **Generate Printfs Inline. Dummy** опцията е необходимо да бъде избрана, ако трябва да забраните опцията **Generate Printfs Inline** и Вашият проект не използва операции с плаваща запетая.

- None

Когато тази опция е избрана, стандартните функции с плаваща запетая са изключени и всеки опит да използвате някоя от тях в кода ще доведе до грешка при компилиране.

ADDRESS SPACES



Фиг. 71 Project Settings→Linker→Address Spaces

ROM

Това поле показва размера на Програмната Памет налична в избрания микроконтролер.

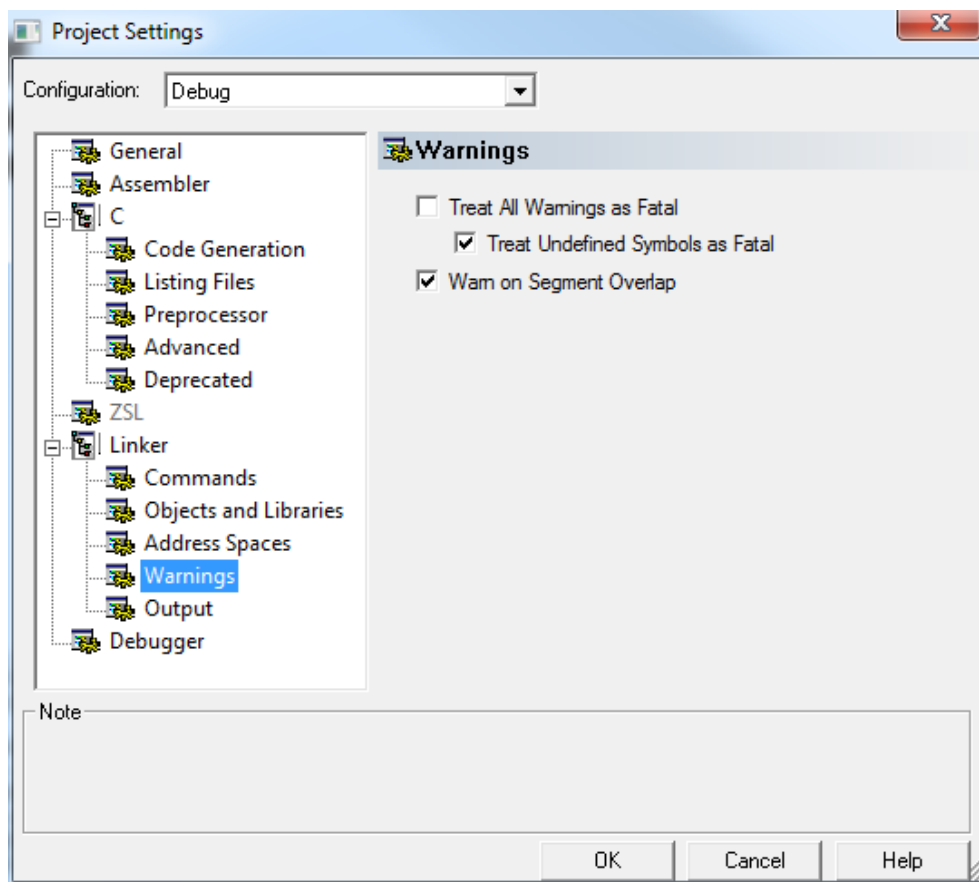
RData

RData паметта е разположена на адреси 00h÷FFh и се използва при малък модел на паметта. Долната граница по подразбиране е установена на 10h и трябва да бъде увеличена с 10h за всяко ниво на прекъсвания. Например за

невложени прекъсвания установете долната граница на 20h,
за две нива на прекъсвания на 30h и т.н.

DRAFT

WARNINGS



Фиг. 72 Project Settings→Linker→Warnings

Treat All Warnings as Fatal

Когато тази опция е избрана, всички предупреждения, които възникват в процеса на компилация се третират като грешки.

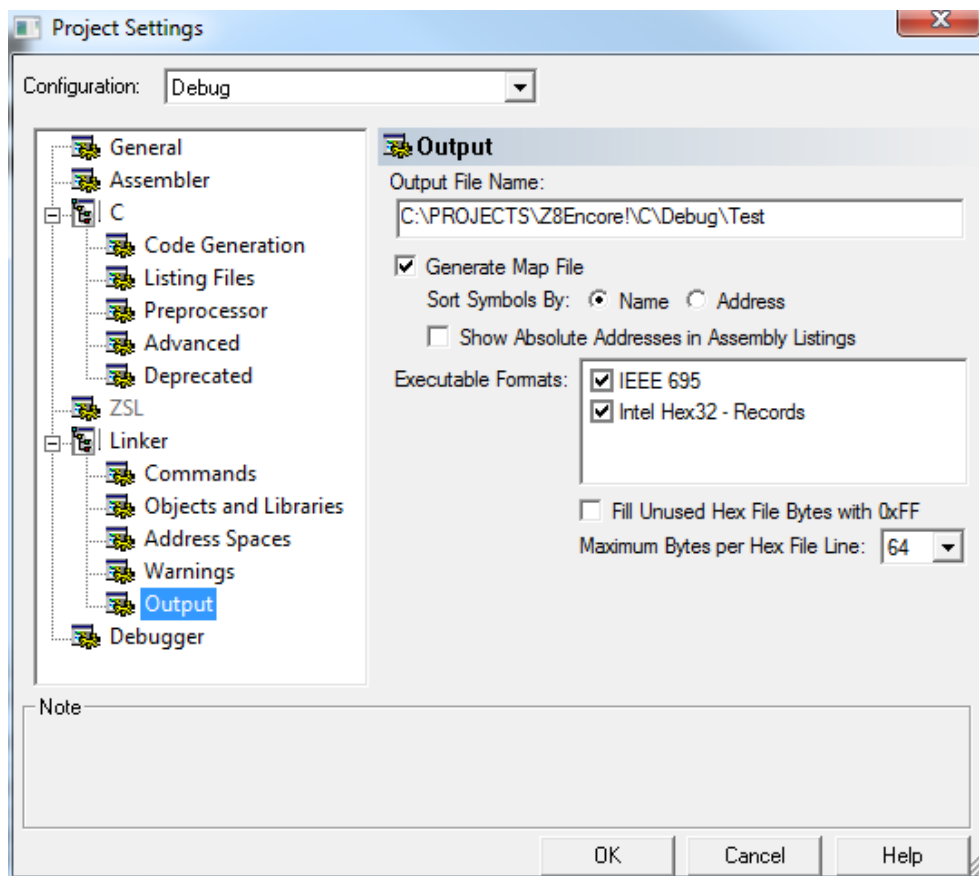
Treat Undefined Symbols as Fatal

Тази опция принуждава Линкерът да третира всички съобщения за недефинирани външни символи като грешки.

Warn on Segment Overlap

Тази опция разрешава предупрежденията при възникване на застъпване на сегменти. Обикновено такова предупреждение е индикация за грешка в конфигурацията на проекта и трябва да бъде коригирано.

OUTPUT



Фиг. 73 Project Settings→Linker→Output

Output File Name

Това поле Ви позволява да зададете името на изпълнимия файл, включително пътя към директорията, където да се генерира.

Generate Map File

Когато тази опция е избрана, Линкерът генерира ***.map** файл при всяко компилиране на проекта.

Sort Symbol By

Тази опция Ви позволява да конфигурирате начина на сортиране на символите в map-файла: по име или по адрес.

Show Absolute Addresses in Assembly Listings

Когато тази опция е избрана, всички асемблерни листинг файлове (*.lst), генерирани в процеса на компилиране на проекта, съдържат абсолютни адреси на асемблерните инструкции, в противен случай се използват относителни адреси, започващи от нула.

За да приложите тази опция към листинг файлове, генерирани от асемблерните сорс-файлове, изберете опцията **Assembler → Generate Assembly Listing Files (.lst)**.

За да приложите тази опция към листинг файлове, генерирани от C сорс-файловете, изберете опциите **C → Listing Files → Generate Assembly Source code** и **C → Listing Files → Generate Assembly Listing Files (.lst)**.

Executable Formats

Тази опция Ви позволява да изберете формата, в който изпълнимият файл ще се генерира. Линкерът поддържа два формата: IEEE 695 (.lod) и Intel Hex32 (.hex).

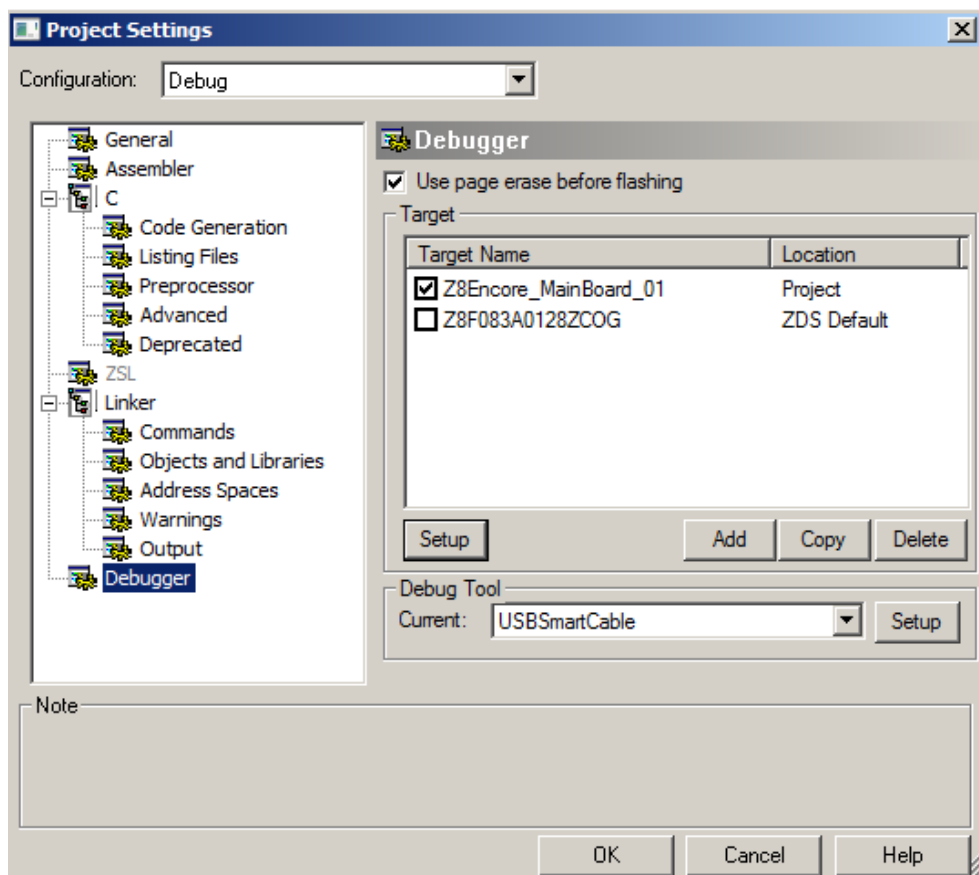
Fill Unused Hex File Bytes with 0xFF

Тази опция е налична само когато е избрана опцията **Intel Hex32 – Records**. Всички неизползвани байтове в .hex файл ще бъдат запълнени с 0xFF.

Maximum Bytes per Hex File Line

Тази опция е налична само когато е избрана опцията **Intel Hex32 – Records** и Ви позволява да изберете максималния брой байтове, които един ред в hex-файла може да съдържа.

39.3.3.6 Настройки на Дебъгера



Фиг. 74 Project Settings → Debugger

Use page erase before flashing

Изберете тази опция ако искате Програмната Памет да бъде изтривана на страници.

Target

Това поле представлява логическо представяне на целевата платка. Може да създавате (**Add**), копирате (**Copy**) и изтривате (**Delete**) имена на логически платки.

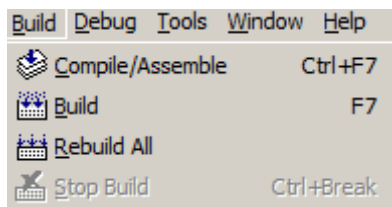
Debug Tool

Изберете инструмента, който ще използвате в процеса на тестване и търсене на грешки във Вашия код. Може да изберете Симулатор (**Simulator**) или някой от хардуерните интерфейси:

- Ethernet SmartCable;
- Serial SmartCable;
- USB SmartCable.

39.4 Компилиране на проекта

Менюто **Build** съдържа всички необходими команди за компилиране на отделни файлове или целия проект (Фиг.75).



Фиг. 75 Меню Build

Compile/Assemble – Компилира (Асемблира) текущия активен С (асемблерен) файл.

Build – Компилира (Асемблира) целия проект. Само файловете, които са били променени от последната компилация, се компилират (асемблират). Тази команда също извиква Линкера.

Rebuild All – Компилира (Асемблира) целия проект. Всички файлове в проекта се компилират (асемблират) независимо дали са били променени от последната компилация. Тази команда също извиква Линкера.

Stop Build – Спира процеса на компилиране (асемблиране).

Всички тези команди са изведени за удобство под формата на лента с бутони .

39.5 Описание на C-компилятора

39.5.1 Съответствие с ANSI C стандарта

Zilog Z8Encore! C-компиляторът (за по-кратко оттук нататък ще го наричам eZ8 C-компилятор) е фрийстендинг¹ C-компилятор, съответстващ на стандарта **ISO/IEC 9899:1990**, който е известен още като **ANSI X3.159-1989**, с някои отклонения.

Забележка¹: Фрийстендинг (**freestanding**) C-компилятор се нарича всеки компилатор, който поддържа езика C, без стандартната библиотека с изключение на средствата, осигурени в хедър-файловете **float.h**, **limits.h**, **stdarg.h**, **stddef.h**. По специално се предполага, че изпълняващата среда няма файлова система и не е под управлението на операционна система.

Отклонения от ANSI C стандарта:

- Прототип на функцията **main**

Съгласно ANSI C стандарта в една фрийстендинг среда името и типа на функцията, извиквана при стартиране на програмата, зависят от имплементацията. Също така, ефектът от завършване на програмата е имплементационно-зависим. eZ8 C-компиляторът използва името **main()**, като повечето имплементации, със следния прототип:

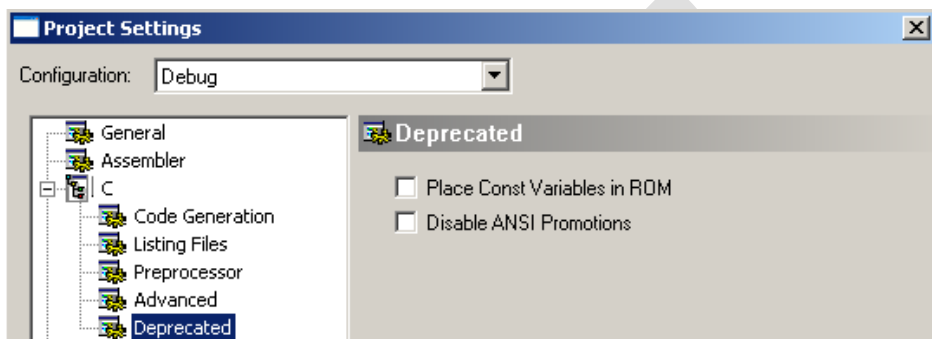
```
void main (void);
```

- Тип **double** се третира като **float**;

eZ8 C-компиляторът не поддържа числа с плаваща запетая с двойна точност. По тази причина тип **double** и **float** са еквивалентни.

- Ключовата дума **const** и **ROM**;

По подразбиране eZ8 C-компиляторът поставя const-променливите в RAM паметта. Въпреки това компилаторът предоставя опцията ☐ Place Const Variables in ROM за съхраняване на такива променливи в ROM паметта (Фиг.76).



Фиг. 76 const-променливи и ROM

Zilog обаче препоръчва тази опция да не се използва. Вместо това може да използвате ключовата дума **rom** (вижте [Нови ключови думи за спецификатори на клас-памет](#)).

Когато тази опция е избрана, ключовата дума **const** се третира като **rom**.

- Ключовата дума **const** и стандартните хедър-файлове;

ANSI C стандартът изисква следния прототип:

```
char* strcpy(char* dst, const char* src);
```

Ако обаче е избрана опцията ☐ Place Const Variables in ROM (Фиг.76), това ще доведе до грешка при компилиране. За да се избегне този проблем, ключовата дума **const** е махната от декларациите на такива параметри на стандартните C-функции, т.е. .

```
char* strcpy( char* dst, char* src);
```


- Повишаване на типовете.

Съгласно ANSI C стандартът променливи от тип `char` (signed и unsigned), `short` (signed и unsigned), enum-типове и битови полета автоматично се повишават до тип `int` (или unsigned int) преди всяко изчисление.

eZ8 C-компиляторът се придържа към това поведение. Компиляторът анализира и повишава типа на променливите само когато е необходимо да имитира поведението на ANSI C стандарта. Например в следния код

```
char ch1, ch2, ch3;
```

```
ch1 = ch1 * ch2;
```

компиляторът не повишава типа на `ch2` и `ch3`, тъй като резултатът отново се съхранява в `char`-променлива (повишаването на типовете в този случай е безсмислено).

Докато в следния пример

```
int ch1;
```

```
char ch2, ch3;
```

```
ch1 = ch1 * ch2;
```

компиляторът повишава типа на `ch2` и `ch3` до тип `int`, тъй като резултатът се съхранява в променлива от тип `int`.

eZ8 C-компиляторът предоставя опцията ☐ `Disable ANSI Promotions` за изключване на автоматичното повишване на типовете (Фиг.76). Zilog обаче препоръчва тази опция да не се използва.

39.5.2 Разширения спрямо ANSI C стандарта

39.5.2.1 Нови ключови думи за спецификатори на клас-памет

eZ8 C-компиляторът добавя следните нови ключови думи за клас-памет: **near**, **far** и **rom**. Те управляват разполагането на глобалните и статичните (локални и глобални) променливи от компилатора.

- **near**

Променливите, декларирани с клас памет **near**, се разполагат в 8-битовото адресно пространство RData (near RAM). Адресният обхват на тези променливи е 00h÷FFh.

Пример:

```
near int ni; /* ni е в адресното пространство RData */
```

- **far**

Променливите, декларирани с клас памет **far**, се разполагат в 12-битовото адресно пространство EData (far RAM). Адресният обхват на тези променливи е 100h÷FFFh.

Пример:

```
far int fi; /* fi е в адресното пространство EData */
```

Забележка: SFR-регистрите са разположени в EData адресното пространство (адреси F00h÷FFFh). Например:

```
#define T0CTL0 (*(unsigned char volatile far*)0xF06)
```

```
T0CTL0 = 0x12;
```

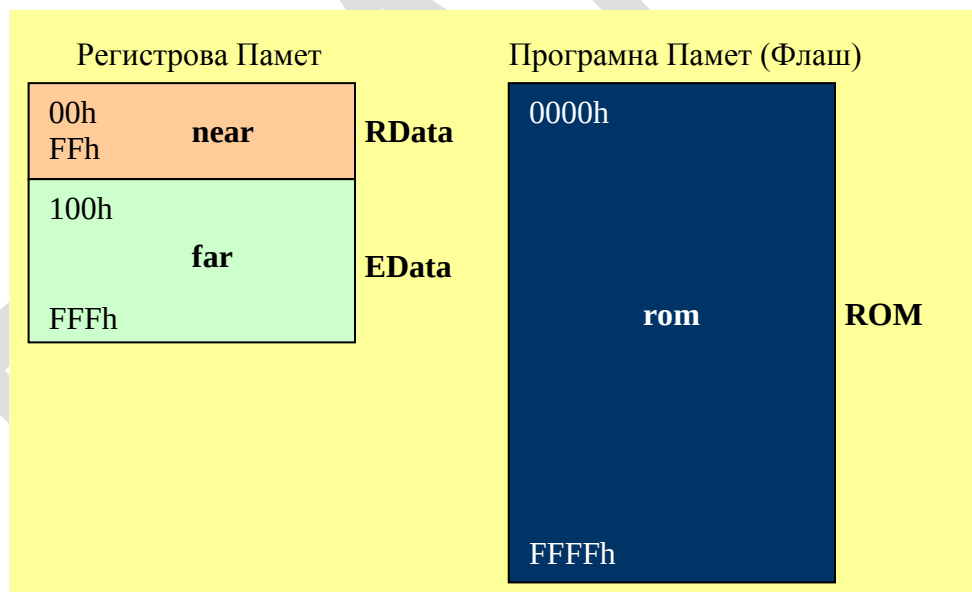
- rom

Променливите, декларирани с клас памет rom, се разполагат в 16-битовото адресно пространство ROM. Адресният обхват на тези променливи е 0000h÷FFFFh. Началото на това адресно пространство е резервирано за флаш-конфигурационните битове и таблицата с векторите на прекъсване.

Пример:

```
rom int ri; /* ri се поставя в адресното пространство ROM */
```

Фиг.77 показва нагледно връзката на спецификаторите за клас-памет и адресните пространства.



Фиг. 77 near, far и rom

Тези спецификатори за клас-памет могат да се прилагат и към указатели.

Пример:

```

char *p; /* p е указател към:
near char : при малък модел
far char : при голям модел */

char far *fp; /* fp е указател към far char, самият указател
fp се съхранява в:
near RAM : при малък модел
far RAM : при голям модел */

char far * far fpf; /* fpf е указател към far char, самият
указател fp се съхранява в far RAM */

char * near pn; /* pn е указател към:
near char : при малък модел
far char : при голям модел
самият указател pn се съхранява в near RAM */

char near * far npf; /* npf е указател към near char, самият
указател fp се съхранява в far RAM */

char rom * near crn; /* crn е указател към rom char, самият
указател crn се съхранява в near RAM */

```

Забележка: С цел по-ефективно използване, eZ8 C-компиляторът осигурява два модела на паметта: малък и голям. Вижте [Модели на паметта](#).

Табл.96 описва подразбиращите се спецификатори за клас-памет в зависимост от използвания модел памет.

	функции	глобални променливи	локални променливи	низ	const	параметри	указатели
Малък	rom	near	near	near	near	near	near
Голям	rom	far	far	far	far	far	far

Табл. 96 Спецификатори за клас-памет по подразбиране

Спецификатор за клас-памет, приложен върху структура/обединение, е с по-висок приоритет от спецификаторите, приложени върху членовете на структурата/обединението. Когато спецификаторът на даден член се различава от този на структурата/обединението, спецификаторът на члена се игнорира.

Пример:

```
struct{
    near char  num; /* спецификаторът near се игнорира */

    near char  * ptr; /* ptr сочи към near char, самият
                       указаел ptr се съхранява в
                       пространството, указано със
                       спецификатора на структурата,
                       който в случая е far */
} far mystruct; /* Всички членове на mystruct се съхраняват
                 в far RAM */
```

39.5.2.2 Модели на паметта

За по-ефективно използване на паметта на микроконтролерите eZ8 C-компиляторът осигурява два модела на паметта: малък (**small**) и голям (**large**). Тези модели Ви позволяват да управлявате разполагането на глобални и статични променливи, и съхраняването на контекста на функциите (локални променливи и параметри) при извикването им.

Малък модел на паметта

При този модел глобалните променливи се разполагат в RData адресното пространство. Адресът на тези променливи е 8-битов. Локалните променливи и параметрите се разполагат в стека, който също се разполага в RData. Адресът на тези променливи също е 8-битов. Глобалните променливи могат ръчно да бъдат поставени в EData или ROM адресното пространство чрез използване на спецификаторите `far` и `rom`.

Забележка: Микроконтролери, които имат 256 байта RAM памет или по-малко, нямат EData адресно пространство.

Локалните (нестатични) променливи и параметрите винаги се разполагат в RData адресното пространство и всякакви адресни спецификатори, използвани в техните декларации, се игнорират.

Голям модел на паметта

При този модел глобалните променливи се разполагат в EData адресното пространство. Адресът на тези променливи е 16-битов. Локалните променливи и параметрите се разполагат в стека, който също се разполага в EData. Адресът на тези променливи също е 16-битов. Глобалните променливи могат ръчно да бъдат поставени в RData или ROM адресното пространство чрез използване на спецификаторите `near` и `rom`.

Локалните (нестатични) променливи и параметрите винаги се разполагат в EData адресното пространство и всякакви адресни спецификатори, използвани в техните декларации, се игнорират.

Забележка: Не всички Z8Epcore! микроконтролери поддържат голям модел на паметта. Този модел изисква наличието на EData адресно пространство, което започва от адрес 100h. Следователно микроконтролери с 256 или по-малко байта RAM не поддържат голям модел на паметта.

39.5.2.3 Контекст на функции

Контекстът на една функция съдържа параметрите, локалните променливи и адресът на връщане. eZ8 C-компиляторът поддържа два типа контекст: **статичен** и **динамичен**.

Статичен контекст

При тази схема за всяка функция в програмата компилаторът заделя памет, където съхранява контекстът, съдържащ параметрите и локалните променливи на функцията.

Предимството на статичния контекст е, че компилаторът знае абсолютните адреси на параметрите на функцията и може да генерира по-компактен код за достъп до тях.

Недостатък на статичния контекст е, че не поддържа рекурсия и извикване на функции чрез указател. За избягване на този недостатък, eZ8 C-компилаторът предоставя ключовата дума **reentrant**, като друго разширение към ANSI C стандарта.

Динамичен контекст

Динамичният контекст се използва изключително в настолно-ориентираните компилатори. Когато една функция се извика, компилаторът заделя динамично памет в стека и съхранява адреса на връщане, параметрите на функцията и локалните променливи.

Ключовата дума reentrant

Ключовата дума **reentrant** се прилага само при използването на статичен контекст на функциите. Когато една функция е декларирана като reentrant, компилаторът използва динамичен контекст вместо статичен. Това позволява функцията да бъде рекурсивна и да се извиква с указател.

Пример:

```
reentrant int recursive_fn(int k)
{
    if (k == 0)
        return 1;
```

```

    return ( k * recursive_fn (k-1));
}

```

Очевидно е, че ако при използването на статичен контекст декларирате голям брой функции като reentrant, ползата от него намалява пропорционално и може да обмислите възможността да превключите към използването на динамичен контекст.

39.5.2.4 Прекъсвания

За поддръжка на прекъсванията eZ8 C-компиляторът използва ключовите думи **interrupt** и **_At** и вътрешната функция SET_VECTOR().

Използвайте следния синтаксис, за да укажете на компилатора, че една функция служи като обработчик на дадено прекъсване (ISR-функция):

```

void interrupt isr_timer0(void)
{
}

```

или

```

#pragma interrupt
void isr_timer0(void)
{
}

```

ISR-функциите не трябва да приемат аргументи и не трябва да връщат стойност.

Адресът на ISR-функцията може да се запише на съответния адрес в таблицата с векторите на прекъсване по два начина.

Първият начин е чрез извикване на вътрешната функция SET_VECTOR(). Тя има следния прототип:

```
intrinsic void SET_VECTOR(int vectnum, void (*hndlr)(void));
```

Пример:

```
#include <eZ8.h>
extern void interrupt isr_timer0(void);
void main(void)
{
    SET_VECTOR(TIMER0, isr_timer0);
}
```

Функцията SET_VECTOR() е описана в [SET_VECTOR](#).

Вторият начин е чрез използване на ключовата дума _At (вижте [Директиви за поставяне и подравняване](#)).

Пример:

```
#include <eZ8.h>
void interrupt isr_timer0(void) _At TIMER0
{
}
}
```

39.5.2.5 Разполагане на низове

Когато низови константи като "mystring" се използват в С-програми, те се съхраняват от компилатора в адресното пространство RData при малък модел на паметта и в EData при голям модел на паметта. eZ8 С-компиляторът добавя разширения, които дават контрол над разполагането на низове в паметта:

- N"mystring"

Низовата константа се съхранява в RData. Адресът на низа е near-указател.

- F"mystring"

Низовата константа се съхранява в EData. Адресът на низа е far-указател.

- R"mystring"

Низовата константа се съхранява в ROM. Адресът на низа е rom-указател.

Пример:

```
#include <sio.h>
void funcn(near char *str)
{
    while (*str)
        putchar (*str++);
    putchar ('\n');
}

void funcf(far char *str)
{
    while (*str)
        putchar (*str++);
    putchar ('\n');
}

void funcr(rom char *str)
{
    while (*str)
        putchar (*str++);
    putchar ('\n');
}

void main(void)
{
    funcn (N"nstr");
    funcf (F"fstr");
    funcr (R"rstr");
}
```

39.5.2.6 Вмъкване на асемблерен код

Има два начина на вмъкване на асемблерни инструкции в C код:

- използване на `#pragma` директива;

```
#pragma asm "асемблерна-инструкция"
```

Пример:

```
#include <ez8.h>
void main(void)
{
    #pragma asm "NOP"
}
```

- използване на оператора `asm`.

```
asm("асемблерна-инструкция");
```

Пример:

```
#include <ez8.h>
void main(void)
{
    asm ("NOP");
}
```

39.5.2.7 Директиви за поставяне и подравняване

Може да използвате следния синтаксис за разполагане на глобални и статични променливи на определен адрес:

```
char placed_char _At 0xff; /* placed_char се разполага на
                             адрес 0xFF */
far struct {
```

```

char ch;
int ii;
} ss _At 0xeff; /* ss се разполага на адрес 0xEFF */

rom char init_char _At 0xffff = 33; /* init_char се разполага
в Програмната Памет на адрес 0xFFFF и се инициализира с 33 */

```

Само променливи, поставени в ROM адресното пространство, могат да се инициализират, near и far променливите не могат да бъдат инициализирани и също така не се инициализират в 0 от компилатора.

Компилаторът също така осигурява синтаксис за последователно разполагане на променливи:

Пример:

```

char ch1 _At 0xef0; // ch1 е на адрес 0xEF0
char ch2 _At ...;   // ch2 е на адрес 0xEF1
char ch3 _At ...;   // ch3 е на адрес 0xEF2

```

Директивата **_At ...** може да се използва само след директивите **_At** или **Align**.

Може да използвате следния синтаксис, за да зададете подравняване на глобалните и статичните променливи по време на дефинирането им:

```

char ch2 _Align 2; // ch2 се разполага на адрес кратен на 2
char ch4 _Align 4; // ch4 се разполага на адрес кратен на 4

```

Само подравнени променливи, поставени в ROM адресното пространство, могат да се инициализират, near и far подравнените променливи не могат да бъдат инициализирани и също така не се инициализират в 0 от компилатора.

39.5.2.8 Изброявания тип char и short

Съгласно ANSI C стандартът типът на една изброяване е имплементационно-зависим. Компиляторът може да избере всеки един от целочислените типове char (signed, unsigned), short (signed, unsigned) или int, в зависимост от стойностите на enum-константите. eZ8 C-компиляторът предоставя възможност типът на едно изброяване да се задава изрично да бъде char или short.

Пример:

```
char enum
{
    RED = 0,
    YELLOW,
    BLUE,
    INVALID
} color; // color е тип char

short enum
{
    NEW= 0,
    OPEN,
    FIXED,
    VERIFIED,
    CLOSED
} status; // status е тип short

void main(void)
{
    if (color == RED)
        status = FIXED;
    else
        status = OPEN;
}
```

39.5.2.9 Конфигуриране на флаш-конфигурационните байтове в C

Флаш-конфигурационните байтове могат да бъдат конфигурирани по следния начин:

```
#include <eZ8.h>
```

```
FLASH_OPTION1 = val; /* Конфигурира байта, разположен на  
адрес 0000h в Програмната Памет */  
FLASH_OPTION2 = val; /* Конфигурира байта, разположен на  
адрес 0001h в Програмната Памет */
```

Пример:

```
#include <eZ8.h>
```

```
FLASH_OPTION1 = 0xFF;  
FLASH_OPTION2 = 0xFF;
```

```
void main(void)  
{  
  
}
```

39.5.2.10 Поддръжка на стандарта C99

eZ8 C-компиляторът поддържа следните свойства на стандарта C99, известен още като ISO/IEC 9899:1999:

- Коментари в стил C++;

Всичко разположено след `//` до края на реда, се игнорира от компилатора.

Пример:

```
// Коментар в стил C++
```

- Добавяне на завършваща запетая в enum декларация;

След последния член на изброяването е позволено да се поставя запетая.

Пример:

```
enum color {red, green, blue,} col;
```

- макро-функция без аргументи;

Един или повече аргументи при извикване на макро-функция могат да се пропуснат

Пример:

```
#define cat3(a,b,c) a b c  
printf("%s\n", cat3("Hello ", , "World"));
```

- тип long long.

Тип **long long** е позволен. eZ8 C-компиляторът третира този тип също като тип **long**.

39.5.3 Размер на типовете

еZ8 С-компиляторът поддържа следните размери на базовите типове данни:

тип	размер в битове	подравняване в байтове
int	16	1
short	16	1
char	8	1
long	32	1
float	32	1
double	32	1
near-указател	8	1
far-указател	16	1
rom-указател	16	1

Табл. 97 Размер на базовите типове

39.5.4 Предефинирани макроси

eZ8 С-компиляторът идва със следните стандартни предефинирани макроси:

- **__AUS_SIZED_BY_TYPE__**

Дефиниран във всички Zilog компилатори като 0 или 1 в зависимост дали размерът на едно битово поле зависи от типа (типовете) на битовите полета.

- **__BACKWARD_COMPATIBLE_BITFIELDS__**

Дефиниран във всички Zilog компилатори като 0 или 1 в зависимост дали имплементацията на битовите полета е съвместима с тази използвана преди януари 2007.

- **__BITFIELDS_OVERLAP_AUS__**

Дефиниран във всички Zilog компилатори като 0 или 1 в зависимост дали едно битово поле, което изисква повече битове, които остават в текущия байт, трябва да започне от нов байт (0 означава, че трябва).

- **__BITFIELDS_PACK_L2R__**

Дефиниран във всички Zilog компилатори като 0 или 1 в зависимост дали битовите полета се пакетират от ляво надясно, т.е. . от най-старшия бит към най-младшия.

- **__CONST_IN_RAM__**

Дефиниран ако const-обектите се поставят в RAM паметта.

- **__CONST_IN_ROM__**

Дефиниран ако const-обектите се поставят в ROM паметта.

- **__CPU_NAME__**

Дефиниран във всички Zilog компилатори и се разгъва до името на процесора.

- **__DATE__**

Разгъва се до низ, представляващ текущата дата, изразена във формата "mmm dd уууу", където имената на месеците са като тези генерирани от функцията asctime(), и първия символ на dd е интервал, ако стойността е по-малка от 10.

- **__FILE__**

Разгъва се до низ, представляващ текущото име на сорс-файла.

- **__LINE__**

Разгъва се до десетична константа, представляваща номера на текущия ред.

- **__NEW_AU_AT_TYPE_CHANGE__**

Дефиниран във всички Zilog компилатори като 0 или 1 в зависимост дали промяна в типа на битови полета изисква започване на нов байт при пакетирането им (1 означава, че изисква).

- **__STDC__**

Десетична константа 1, указваща, че компилаторът е ANIS/ISO C компилатор.

- **__TIME__**

Разгъва се до низ, представляващ времето на компилиране, изразено във формата "hh:mm:ss".

- **__UNSIGNED_CHARS__**

Този макрос е дефиниран, ако тип char е еквивалентен на unsigned char.

- **__ENCORE__**

Този макрос е дефиниран и се разгъва се до 1, ако компилаторът е Z8 Encore!, иначе е недефиниран.

- **__EZ8__**

Този макрос е дефиниран и се разгъва се до 1, ако процесорът е eZ8, иначе е недефиниран.

- **__FPLIB__**

Дефиниран във всички Zilog компилатори и указва дали е налична библиотека с плаваща запетая (0 – не е налична, 1 – налична е).

- **__MODEL__**

Този макрос указва моделът на паметта, използван от компилатора: 0 – малък модел; 3 – голям модел

- **__ZDATE__**

Този макрос се разгъва до датата на създаване на

компилятора във формата YYYYMMDD. Например ако компилаторът е създаден на 31 Май 2006, макросът се рагъва до 20060531.

- **__ZILOG__**

Този макрос е дефиниран във всички Zilog компилатори и се разгъва до 1, указвайки че компилаторът е осигурен от Zilog.

Никой от тези макроси не може да бъде отдефиниран с директивата #undef

Пример:

```
#include <stdio.h>

void main(void)
{
    #ifdef __ZILOG__
    printf("Zilog Compiler ");
    #endif
    #ifdef __ENCORE__
    printf("for Z8 Encore! ");
    #endif
    #ifdef __EZ8__
    printf("with eZ8 Cpu ");
    #endif
    #ifdef __ZDATE__
    printf("built on %d.\n", __ZDATE__);
    #endif
}
```

В допълнение към горните предефинирани макроси, развойната среда ZDS II генерира следните макроси:

- **_DEBUG;**

Генерира се ако кодът се компилира в DEBUG режим.

- **NDEBUG;**

Генерира се ако кодът се компилира в RELEASE режим.

- `_<cpu>`;

`<cpu>` е името на процесора, например `_Z8F1680`.

- `_<variant>`.

`<variant>` е специфичният вариант на процесора, например `_Z8F2480XX20XXSG`.

39.5.5 Конвенции при извикване на функции

39.5.5.1 Механизъм на извикване на функция: Динамичен контекст

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Calling Convention→Function Call Mechanism: Dynamic Frame).

39.5.5.2 Механизъм на извикване на функция: Статичен контекст

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Calling Convention→Function Call Mechanism: Static Frame).

39.5.5.3 Механизъм на извикване на функция: Съхраняване на параметри в регистри

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Calling Convention→Function Call Mechanism: Register Parameter Passing).

39.5.5.4 Връщана стойност

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Calling Convention→Return Value).

39.5.5.5 Специални случаи

Вижте в помощното меню на развойната среда ZDS II

(Help→Help Topics→Using the ANSI C-Compiler→Calling Convention→Special Cases).

DRAFT

39.5.6 Извикване на асемблерни функции от C

Вижте в помощното меню на развойната среда ZDS II (**Help→Help Topics→Using the ANSI C-Compiler→Calling Assembly Functions from C**).

39.5.7 Извикване на C-функции от Асемблера

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Calling C Functions from Assembly).

39.5.8 RTL-библиотека (Run-time Library)

39.5.8.1 Общи сведения

eZ8 C-компиляторът предоставя два типа библиотеки:

- Стандартна C-библиотека

Стандартната C-библиотека включва библиотечните функции и други средства, дефинирани от ANSI C стандарта. Пълни подробности за имплементацията на стандартната библиотека от Z8 C-компилятора може да намерите в помощното меню на развойната среда ZDS II (**Help**→**Help Topics**→**C Standard Library**).

- Zilog C-библиотека

Тази библиотека съдържа функции и средства, добавени от Zilog, и които не са част от ANSI C стандарта. Тя се състои от хедър-файловете <eZ8.h> и <sio.h>. Следващите подточки описват тези хедър-файлове и библиотечните функции, добавени от Zilog.

39.5.8.2 Zilog хедър-файлове

39.5.8.2.1 <eZ8.h>

Този хедър-файл съдържа:

- Прототипите на следните Zilog-функции;

```
intrinsic void EI(void);
```

```
intrinsic void DI(void);
```

```
intrinsic SET_VECTOR(int,void (* func) (void));
```

```

void reentrant INIT_FLASH(unsigned short freq);

char reentrant READ_FLASH(rom const void *addr);

void reentrant WRITE_FLASH(rom const void *addr,
                           char val);

char reentrant READ_NVDS(char addr);

int reentrant WRITE_NVDS(char value, char addr);

int reentrant READ_NVDS_GET_STATUS(char addr);

char reentrant WRITE_NVDS_GET_STATUS(char value,
                                     char addr);

intrinsic void RI(unsigned char istat);

intrinsic unsigned char TDI(void);

```

- Макро-дефинициите на всички SFR-регистри;
- Макро-дефинициите на флаш-конфигурационните байтове;

FLASH_OPTION1 → Разгъва се до адрес 0000h

FLASH_OPTION2 → Разгъва се до адрес 0001h

- Макроси, разгъващи се до номерата на векторите на прекъсване (вижте [SET VECTOR](#)).

39.5.8.2.2 <sio.h>

Този хедър-файл съдържа:

- Прототипите на следните Zilog-функции;

```

int getch(void);

int init_uart(int port,
              unsigned long freq,
              unsigned long baud);

unsigned char kbhit(void);

```

```
reentrant unsigned char putch(char);  
  
int select_port(int port);
```

- Макроси

```
_DEFFREQ  
_DEFBAUD  
_UART0  
_UART1
```

39.5.8.3 Описание на Zilog функциите

39.5.8.3.1 *DI*

```
#include <ez8.h>  
intrinsic void DI(void);
```

Тази функция забранява всички прекъсвания.

Пример:

```
#include <ez8.h>  
void main(void)  
{  
    DI(); /* Забрани прекъсванията */  
}
```

39.5.8.3.2 *EI*

```
#include <ez8.h>  
intrinsic void EI(void);
```

Тази функция разрешава всички прекъсвания.

Пример:

```
#include <eZ8.h>
void main(void)
{
    EI(); /* Разреши прекъсванията */
}
```

39.5.8.3.3 *getch*

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Run-Time Library→Zilog Functions→getch).

39.5.8.3.4 *INIT_FLASH*

```
#include <eZ8.h>
void reentrant INIT_FLASH(unsigned short freq);
```

Тази функция конфигурира тактовата честота при запис във флаш-паметта. Следните стойности се предефинирани за удобство:

```
FREQ20000 /* 20 MHz */
FREQ18432 /* 18.432 MHz */
FREQ16000 /* 16 MHz */
FREQ14000 /* 14 MHz */
FREQ12000 /* 12 MHz */
FREQ08000 /* 8 MHz */
FREQ04000 /* 4 MHz */
```

Пример:

```
#include <eZ8.h>
```

```

char x = 0x55;
void main(void)
{
    INIT_FLASH(FREQ18432); /* Конфигурирай тактовата честота
                             при запис във флаш-памятта */
    WRITE_FLASH((rom const char *)0x2f00,x); /* Запиши x във
                                                флаш-памятта */
    x = READ_FLASH((rom const char *)0x2f00); /* Прочети от
                                                флаш-памятта */
}

```

Забележка: Не записвайте във флаш-памятта повече от два пъти. За да запишете във флаш-памятта повече от два пъти е необходимо да изтриете страницата.

39.5.8.3.5 *init_uart*

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Run-Time Library→Zilog Functions→init_uart).

39.5.8.3.6 *kbhit*

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Run-Time Library→Zilog Functions→kbhit).

39.5.8.3.7 *putch*

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Run-Time Library→Zilog Functions→putch).

39.5.8.3.8 *READ_FLASH*

```
#include <ez8.h>
char reentrant READ_FLASH(rom const void *addr);
```

Тази функция чете стойността на специфицирания адрес на флаш-паметта и я връща като резултат.

Пример:

Вижте [INIT_FLASH](#).

39.5.8.3.9 *READ_NVDS*

```
#include <ez8.h>
reentrant char READ_NVDS(char address);
reentrant char nvds_read(char address);
```

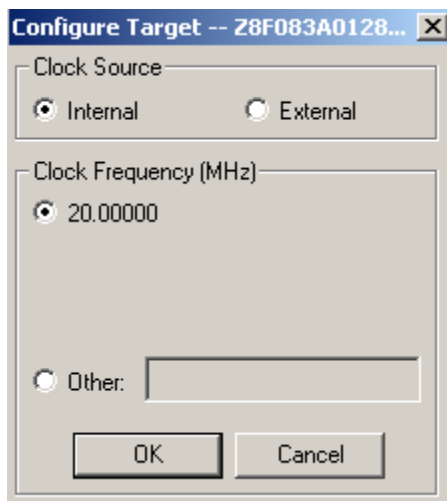
Тази функция чете стойността от специфицирания адрес на NVDS-паметта и я връща като резултат.

Пример:

```
#include <ez8.h>
char x = 0x55;
void main(void)
{
    INIT_FLASH(_DEFFREQ); /* Конфигурирай тактовата честота
                           при запис във флаш-паметта */
    WRITE_NVDS(x, 0x10); /* Запиши x на адрес 0x10 в NVDS */
    x = READ_NVDS(0x10); /* Прочети NVDS от адрес 0x10 */
}
```

Макросът `_DEFFREQ` се разгъва до стойността на тактовия сигнал по подразбиране така както е зададен в прозореца **Configure Target** в настройките на проекта (Фиг.78). За да отворите този прозорец изберете **Project** → **Settings...**→

Debugger → Setup.



Фиг. 78 Configure Target

39.5.8.3.10 *READ_NVDS_GET_STATUS*

```
#include <ez8.h>
int reentrant READ_NVDS_GET_STATUS(char addr);
int reentrant nvds_read_get_status(char addr);
```

Тази функция чете от специфицирания адрес на NVDS-паметта и връща като резултат прочетената стойност и състоянието от операцията четене. Старшият байт на върната стойност съдържа прочетената стойност, а младшият байт – състоянието.

Пример:

```
#include <ez8.h>
char x = 0x55, wstatus, rstatus;
unsigned int val;
```



```

void main(void)
{
    wstatus = WRITE_NVDS_GET_STATUS(x, 0x10); /* Запиши x в
                                                NVDS на адрес 0x10 и прочети състоянието */
    val = READ_NVDS_GET_STATUS(0x10); /* Прочети NVDS от
                                        адрес 0x10 и върни състоянието */
    x = (val >> 8) & 0xFF; /* Извлечи данните */
    rstatus = val & 0xFF; /* Извлечи състоянието */
}

```

39.5.8.3.11 RI

```

#include <eZ8.h>
intrinsic void RI(unsigned char istat);

```

Тази функция възстановява състоянието на прекъсванията. Тя е предназначена да се използва съвместно с функцията TDI(), която съхранява състоянието на прекъсванията.

Пример:

```

#include <eZ8.h>
void main(void)
{
    unsigned char istat;

    istat = TDI(); /* Тествай и забрани прекъсванията */
    /* ... */
    RI(istat); /* Възстанови прекъсванията */
}

```

39.5.8.3.12 select_port

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the ANSI C-Compiler→Run-Time Library→Zilog Functions→select_port).

39.5.8.3.13 SET_VECTOR

```
#include <ez8.h>
intrinsic void SET_VECTOR(int vectnum, void (*hndlr) (void));
```

където

vectnum е номер на вектор на прекъсване в таблицата с векторите на прекъсванията и може да бъде един от следните предефинирани макроси:

RESET	-	ресет вектор
WDT	-	WDT-таймер
POTRAP	-	Основен генератор
WOTRAP	-	RC генератор на WDT-таймера
TRAP	-	Грешна инструкция
TIMER1	-	Таймер1
TIMER0	-	Таймер0
ADC	-	АЦП
P0AD	-	Извод PA0
P1AD	-	Извод PA1
P2AD	-	Извод PA2
P3AD	-	Извод PA3
P4AD	-	Извод PA4
P5AD	-	Извод PA5
P6AD	-	Извод PA6
P7AD	-	Извод PA7
C3	-	Извод PC3
C2	-	Извод PC2
C1	-	Извод PC1
C0	-	Извод PC0

hndlr е името ISR-функцията

Тази функция се използва за конфигуриране на таблицата с векторите на прекъсване с адресите на ISR-функциите.

Пример:

```
#include <ez8.h>
extern void interrupt isr_timer0(void);
void main(void)
{
    SET_VECTOR(TIMER0, isr_timer0); /* Конфигурирай функцията
    isr_timer0 да бъде обработчик на прекъсването от Таймер 0 */
}
```

39.5.8.3.14 TDI

```
#include <ez8.h>
intrinsic unsigned char TDI(void);
```

Тази функция съхранява (връща като резултат) текущото състояние на прекъсванията и ги забранява. Тя е предназначена да работи съвместно с функцията RI(), която възстановява предишното състояние на прекъсванията.

Пример:

Вижте [RI](#).

39.5.8.3.15 WRITE_FLASH

```
#include <ez8.h>
void reentrant WRITE_FLASH(rom const void *addr, char val);
```

Тази функция записва стойност във флаш-паметта на специфициран адрес.

Пример:

Вижте [INIT_FLASH](#).

39.5.8.3.16 WRITE_NVDS

```
#include <ez8.h>
reentrant int WRITE_NVDS(char value, char address)
reentrant int nvds_write(char value, char address)
```

Тази функция записва стойност в NVDS-паметта на специфициран адрес. При успех функцията връща 0, иначе

връща ненулева стойност.

Пример:

Вижте [READ_NVDS](#)

39.5.8.3.17 WRITE_NVDS_GET_STATUS

```
#include <ez8.h>
char reentrant WRITE_NVDS_GET_STATUS(char value, char addr);
char reentrant nvds_write_get_status(char value, char addr);
```

Тази функция записва стойност в NVDS-паметта на специфициран адрес и връща състоянието от операцията запис.

Пример:

Вижте [READ_NVDS_GET_STATUS](#).

39.5.9 Стартови файлове

Стартовият файл е асемблерна програма, която извършва необходимите инициализации, след което извиква функцията `main()`, която е входна точка за всяка С програма. Инициализациите, които се извършват са:

- Инициализиране на Указателя към Регистрите `RP` и Указателя към Стека `SP`;
- Нулиране на `near` и `far` неиницилизирани променливи;
- Конфигуриране на инициализираните `near` и `far` променливи с техните начални стойности от `ROM`;
- Заделяне на памет за таблицата с векторите на прекъсване и флаш-конфигурационните байтове;
- Заделяне на памет за променливата **`errno`**, използвана от `RTL`-библиотеката.

Следващата таблица описва стартовите файлове, осигурени от `eZ8` С-компилятора.

име	описание
lib\zilog\startups.obj	Стартов обектен файл за малък модел
src\boot\common\startups.asm	Стартов сорс-файл за малък модел
lib\zilog\startupl.obj	Стартов обектен файл за голям модел
src\boot\common\startupl.asm	Стартов сорс-файл за голям модел

Табл. 98 Стартови файлове

39.5.10 Именуване на сегменти

Компиляторът поставя кода и данните в отделни сегменти в обектните файлове. Следващата таблица изброява имената на различните сегменти с кратко описание.

Сегмент	Описание
NEAR_DATA	near инициализирани глобални и статични данни
NEAR_BSS	near неинициализирани глобални и статични данни
NEAR_TEXT	near константни низове
FAR_DATA	far инициализирани глобални и статични данни
FAR_BSS	far неинициализирани глобални и статични данни
FAR_TEXT	far константни низове
ROM_DATA	rom глобални и статични данни
ROM_TEXT	rom константни низове
__VECTORS_nnn	rom вектор на прекъсване на адрес nnn
STARTUP	rom С стартов код

Табл. 99 Сегменти

39.6 Описание на Линкера

39.6.1 Общи сведения

Линкерът, както подсказва неговото име, свързва всички обектни файлове и в резултат генерира единичен изпълним файл. Процесът на свързване се управлява от линкерни команди, съхранявани в линкер-команден файл. Изпълнимият файл може да бъде зареден в микроконтролера и тестван посредством средата ZDS II и дебъгера USB SmartCable.

39.6.2 Линкер-командни файлове

Вижте в помощното меню на развойната среда ZDS II (**Help→Help Topics→Using the ANSI C-Compiler→Linker Command Files For C Programs**).

39.6.3 Линкерни команди

Вижте в помощното меню на развойната среда ZDS II (**Help→Help Topics→Using the Linker/Locator→Linker Commands**).

39.6.4 Линкерни изрази

Вижте в помощното меню на развойната среда ZDS II (**Help→Help Topics→Using the Linker/Locator→Linker Expressions**).

39.7 Описание на Дебъгера

39.7.1 Общи сведения

Дебъгерът Ви позволява да търсите и откривате грешки във Вашия код. Той се състои от две части. Софтуерна част, вградена в средата ZDS II, и хардуерна част, която служи за връзка (интерфейс) между софтуерната част и платката, на която е монтиран микроконтролера (Фиг.79).



Фиг. 79 Свързване на дебъгера към платката и компютъра

Zilog предоставят три варианта на интерфейса¹:

- USB SmartCable (ZUSBSC00100ZACG);
- Opto-Isolated SmartCable (ZUSBOPTSC01ZACG);
- Ethernet SmartCable (ZENETSC0100ZACG).

Забележка¹: Ще използвам Дебъгер, за да обозначавам софтуерната

39.7.2 Кодови индикатори на Дебъгера

Дебъгерът използва следните кодови индикатори:

- Жълта стрелка, указваща текущия ред, на който се намира програмния брояч;

```
148 void main (void)
149 - {
150     Adc_u16ResultType Adc_u16Result;
151
152     Mcu_vInit(&Mcu_sConfig);
153     Port_vInit(&Port_sConfig);
154     Adc_vInit(&Adc_sConfig);
```

- Синя точка, указваща валидни кодови редове. Това са редове, до които може да стартирате програмния брояч, на които може да поставите точки на прекъсване (**breakpoints**) и т.н. Редовете от кода, които не са компилирани или са били оптимизирани в процеса на компилация, нямат синя точка.

```
148 void main (void)
149 - {
150     Adc_u16ResultType Adc_u16Result;
151
152     Mcu_vInit(&Mcu_sConfig);
153     Port_vInit(&Port_sConfig);
154     Adc_vInit(&Adc_sConfig);
155
156     - #if 0
157     while(1)
158     {
159         Adc_vStartConversion(ADC_CH2);
160         while(ADC_BUSY == Adc_eGetStatus());
161         Adc_u16Result = Adc_u16GetResult();
162         LedInd_SendDataToLedIndicator(~(Adc_u16Result>>2));
163         Gnrl_vDelay(GNRL_DELAY_100_MSEC);
164     }
165     #endif
166 }
```

Редове 157÷168 нямат синя точка, тъй като са изключени от процеса на компилация с директивите #if-#endif.

- Червен осмоъгълник, указващ точка на прекъсване;

```

148 void main (void)
149  * - {
150      Adc_u16ResultType Adc_u16Result;
151
152      Mcu_vInit(&Mcu_sConfig);
153      Port_vInit(&Port_sConfig);
154      Adc_vInit(&Adc_sConfig);

```

Бял осмоъгълник, указва забранена точка на прекъсване.

Точките на прекъсване могат да се поставят само на редове, маркирани със синя точка. При достигане на ред, на който е поставена точка на прекъсване, програмата спира изпълнението си. Индикаторът на програмния брояч (жълтата стрелка) се установява на този ред в очакване на следваща команда.

```

148 void main (void)
149  * - {
150      Adc_u16ResultType Adc_u16Result;
151
152      *
153      *
154      *

```

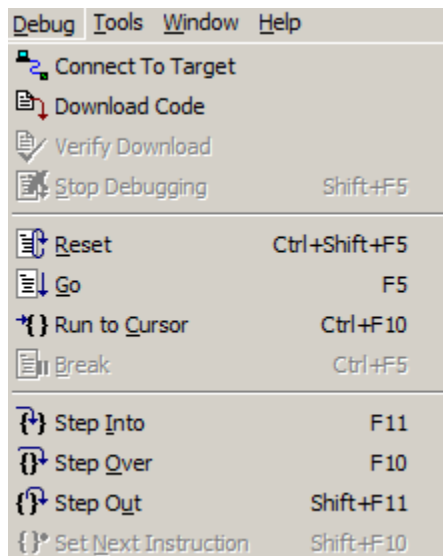
Повече за точките на прекъсване ще намерите в [Точки на прекъсване](#).

39.7.3 Описание на менюто Debug

Менюто **Debug** съдържа всички необходими команди за стартиране и използване на Дебъгера (Фиг. 80).

Connect To Target – Инициализира комуникацията с микроконтролера. Тази команда не зарежда програмата в микроконтролера, а само осъществява връзката му Дебъгера.

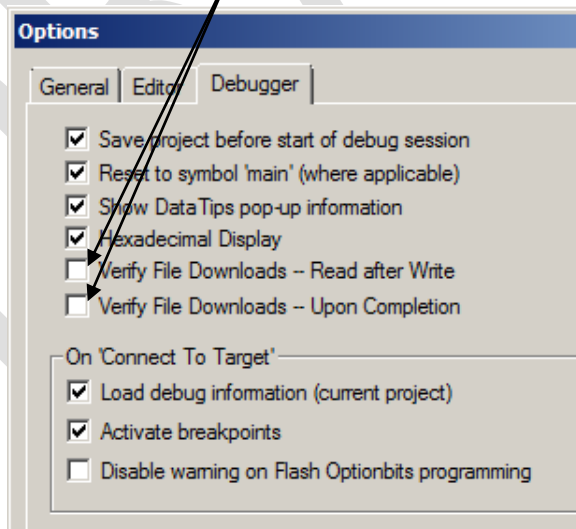
Download Code – Инициализира комуникацията с микроконтролера и зарежда програмата (изпълнимия файл).



Фиг. 80 Меню Debug

Verify Download – Сравнява съдържанието на записаната в Програмната Памет информация с изпълнимия файл.

Забележка: Тази команда може да се извършва автоматично, ако е избрана една от следните опции от **Tools**→**Options...**→**Debugger**:



Фиг. 81 Tools → Options... → Debugger

Stop Debugging – Спира Дебъгера.

Reset – Рестартира микроконтролера. Ако Дебъгерът е бил спрян, тази команда първо инициализира комуникацията с микроконтролера, зарежда програмата, след което рестартира микроконтролера. Програмният брояч се установява в началото на програмата¹.

Забележка¹: Ако е избрана опцията **Reset to symbol 'main' (where applicable)** от **Tools→Options...→Debugger** (Фиг.81), Програмният Брояч се установява на първия оператор във функцията main.

Go – Стартира изпълнението на програмата. Ако Дебъгерът е бил спрян, тази команда първо инициализира комуникацията с микроконтролера, зарежда програмата, рестартира микроконтролера, след което стартира изпълнението на програмата.

Run to Cursor – Изпълнява програмния код от текущата позиция на Програмния Брояч до реда, на който е поставен курсора. Курсорът трябва да бъде поставян на валидни редове (такива със синя точка).

Break – Спира изпълнението на програмата. Програмният Брояч се установява на реда, на който е бил в момента на изпълнение на тази команда.

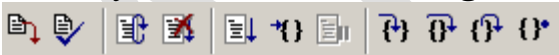
Step Into – Изпълнява оператора (инструкцията) текущо сочен от Програмния Брояч, след което Програмният Брояч се премества на следващата инструкция. Ако операторът е извикване на функция, Програмният Брояч се установява на първия оператор във функцията.

Step Over – Изпълнява оператора (инструкцията) текущо сочен от Програмния Брояч, след което Програмният Брояч

се премества на следващата инструкция. Ако операторът е извикване на функция, кодът на функцията се изпълнява и Програмният Брояч се установява на първия оператор след функцията.

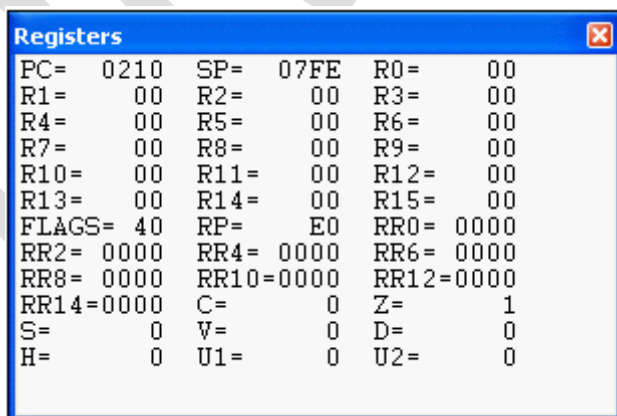
Step Out – Изпълнява всички оставащи оператори (инструкции) в текущата функция, започвайки от оператора (инструкцията) текущо сочен от Програмния Брояч, след което Програмният Брояч се установява на първия оператор след функцията.

Set Next Instruction – Установява Програмният Брояч на реда, на който се намира курсора. Курсорът трябва да бъде поставян на валидни редове (такива със синя точка).

За удобство всички бутони от меню **Debug** са изведени на лента с бутони .

39.7.4 Описание на прозорците на дебъгера

39.7.4.1 Registers



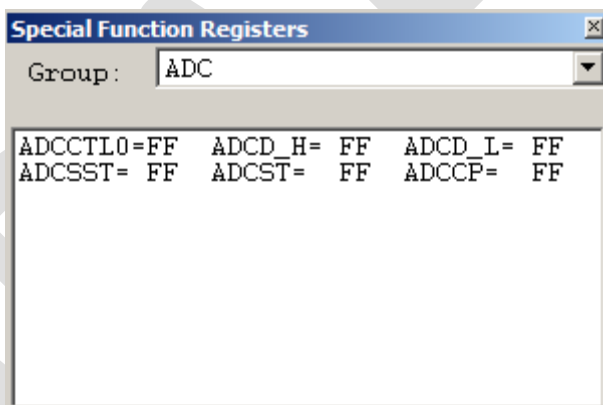
Фиг. 82 Прозорец Registers

Прозорецът **Registers** показва всички регистри на процесора eZ8. За да промените стойност на даден регистър, направете следното:

1. Изберете стойността, която искате да промените;
2. Въведете нова стойност;
3. Натиснете Enter.

Променената стойност се показва в червено.

39.7.4.2 Special Function Registers



Фиг. 83 Прозорец Special Function Registers

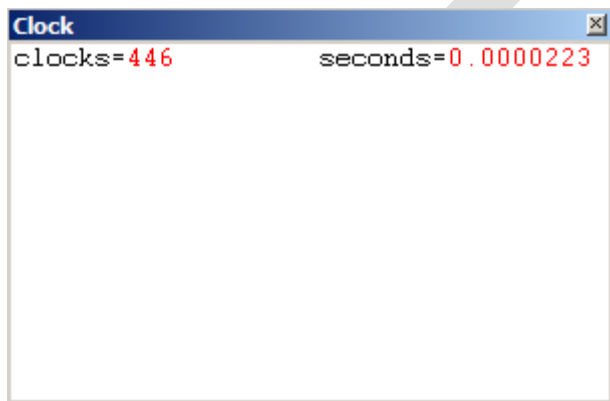
Този прозорец показва всички регистри със специално предназначение (адреси F00h÷FFFh в Регистровата Памет). Те са разделени на групи според функционалността си. За да промените стойност на даден регистър, направете следното:

1. Изберете стойността, която искате да промените;
2. Въведете нова стойност;
3. Натиснете Enter.

Променената стойност се показва в червено.

За да видите адреса на регистър, поставете курсора върху името на регистъра

39.7.4.3 Clock



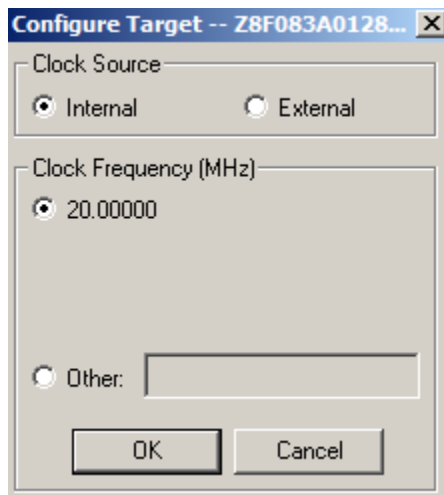
Фиг. 84 Прозорец Clock

Този прозорец показва броя на тактовите цикли¹ след последния ресет (параметъра **clocks** на фигурата).

Забележка¹: В случай че сте забравили, всяка асемблерна инструкция се изпълнява за определен брой периоди (цикли) на тактовия сигнал на процесора. Може да използвате този прозорец, за да определите времето необходимо за изпълнение на определени части от кода.

Може да нулирате съдържанието на този прозорец като изберете циклите (446 на фигурата), въведете 0 и натиснете Enter. Параметърът **seconds** показва изминалото време в секунди. Показаната стойност зависи от тактовата честота на процесора, зададена в прозореца **Configure Target**, достъпен от **Project**→**Settings...**→**Debugger**→**Setup** (Фиг.85).

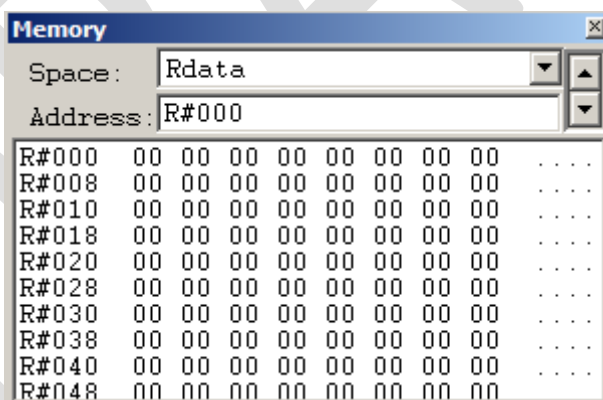
Прозорецът Clock е валиден само когато за тестващ инстру-



Фиг. 85 Прозорец Configure Target

мент е избран Симулатора (**Project**→**Settings...**→**Debugger**→**Debug Tool**→**Current = Simulator**).

39.7.4.4 Memory



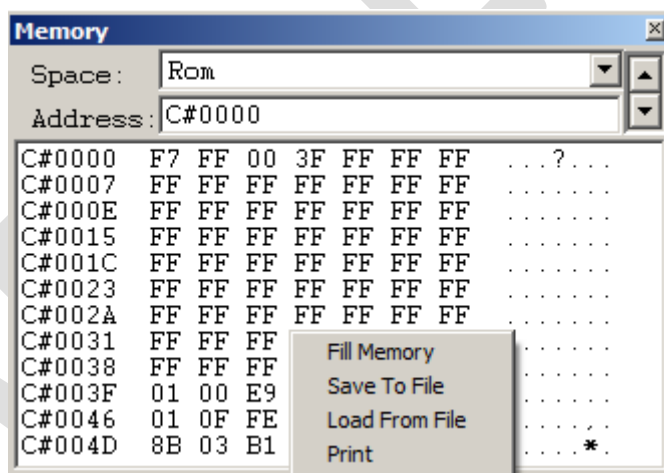
Фиг. 86 Прозорец Memory

Този прозорец осигурява достъп до всички видове памети в микроконтролера. Конкретния тип памет се избира от падащото меню **Space**. За да промените стойност на дадена клетка от паметта, направете следното:

1. Изберете паметта в полето Space;
2. Изберете стойността, която искате да промените (може да въведете конкретен адрес в полето **Address**);
3. Въведете нова стойност;
4. Натиснете Enter.

Променената стойност се показва в червено.

Фиг.87 показва допълнителните опции, достъпни с щракване с десен бутон в прозореца Memory.



Фиг. 87 Контекстно меню на прозореца Memory

39.7.4.5 Watch

Може да използвате този прозорец и да наблюдавате стойностите на променливите и адресите, на които са разположени. Една променлива може да бъде наблюдавана само когато е в областта си на видимост.

За да добавите променлива в прозореца Watch, изберете я в

Watch		
Expression	Value	Address
Adc_u16Result	Not valid for current scope	
u08Data	Not valid for current scope	
<new watch>		

Фиг. 88 Прозорец Watch

сорс-кода, влачете я и я пуснете в прозореца, или:

- Щракнете веднъж върху **<new watch>** в колоната **Expression**;
- Въведете името на променливата и натиснете Enter.

За да промените стойност на дадена променлива, направете следното:

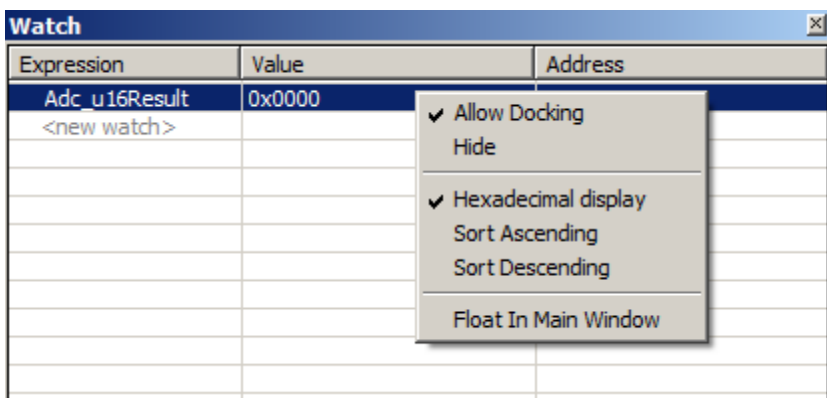
- Изберете стойността, която искате да промените;
- Въведете нова стойност;
- Натиснете Enter.

Променената стойност се показва в червено.

За да премахнете променлива от прозореца Watch, направете следното:

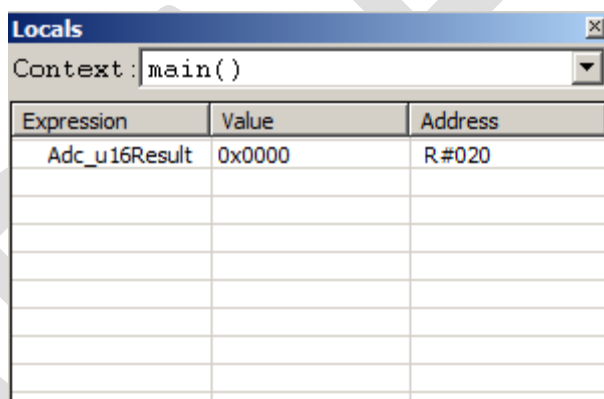
- Изберете променливата и натиснете Delete.

Фиг.89 показва допълнителните опции, достъпни с щракване с десен бутон в прозореца Watch.



Фиг. 89 Прозорец Watch

39.7.4.6 Locals

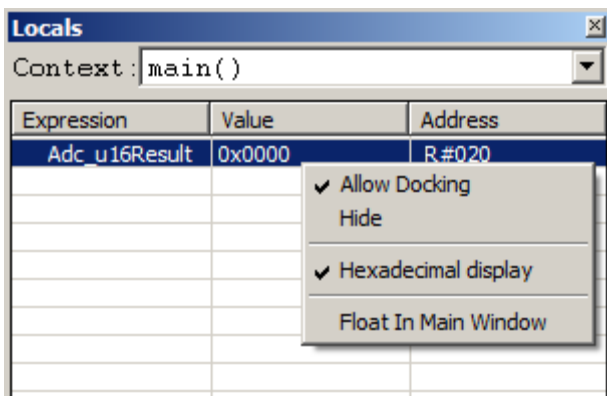


Фиг. 90 Прозорец Locals

Прозорецът **Locals** показва автоматично всички локални променливи, които са в областта си на видимост.

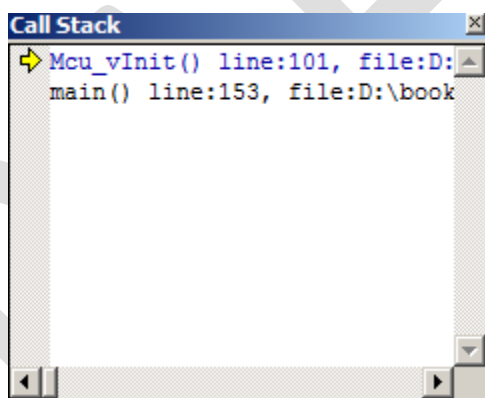
Фиг.91 показва допълнителните опции, достъпни с щракване с десен бутон в прозореца **Locals**.

Забележка: Ако прозорецът **Locals** показва неочаквани стойности, изключете опцията **Use Register Variables** от **Project→Settings→Advanced→Use Register Variables: Off**.



Фиг. 91 Контекстно меню на прозореца Locals

39.7.4.7 Call Stack



Фиг. 92 Прозорец Call Stack

Прозорецът **Call Stack** Ви позволява да видите извикването на функциите по време на изпълнение на програмата. Например Фиг.92 показва, че текущо извиканата функция е `Mcu_vInit()`, която от своя страна е извикана от функцията `main()`.

39.7.4.8 Symbols

Прозорецът **Symbols** показва всички символи и техните ад-

Symbols			
===== REGISTER VARIABLES =====			
REGISTER	VARIABLE	FILE	PROCEDURE
=====			
R8	psConfigPtr	mcu.c	McU_vInitG
R8	psConfigPtr	adc.c	Adc_vInitG
RR2	u16Return	adc.c	Adc_u16GetResultG
R8	psConfigPtr	port.c	Port_vInitG
R10	u08PortValue	port.c	Port_vWritePortG
RR8	eChannel	adc.c	Adc_vStartConversionG
RR8	ePort	port.c	Port_vWritePortG
===== SPACE: ROM =====			
LOCATION	VARIABLE	FILE	PROCEDURE
=====			
0000020e	_Gnrl_vDelay	general.c	
000001c3	_main	main.c	
000001b5	_LedInd_SendDataToLedIndicator	ledindicator.c	
00000190	_Adc_u16GetResult	adc.c	
00000179	_Adc_eGetStatus	adc.c	
00000172	_Adc_vStartConversion	adc.c	
0000014f	_Adc_vInit	adc.c	
00000139	_McU_vInit	mcu.c	

Фиг. 93 Прозорец Symbols

реси, използвани в програмата.

39.7.4.9 Disassembly

```

Disassembly
Address:
void main (void)
{
    Adc_u16ResultType Adc_u16Result;

    Mcu_vInit(&McU_sConfig);
01C3 8C39 LD    R8, #39
01C5 D60139 CALL #139
    Port_vInit(&Port_sConfig);
01C8 8C26 LD    R8, #26
01CA D60071 CALL #71
    Adc_vInit(&Adc_sConfig);
01CD 8C3B LD    R8, #3B
01CE D6014F CALL #14F

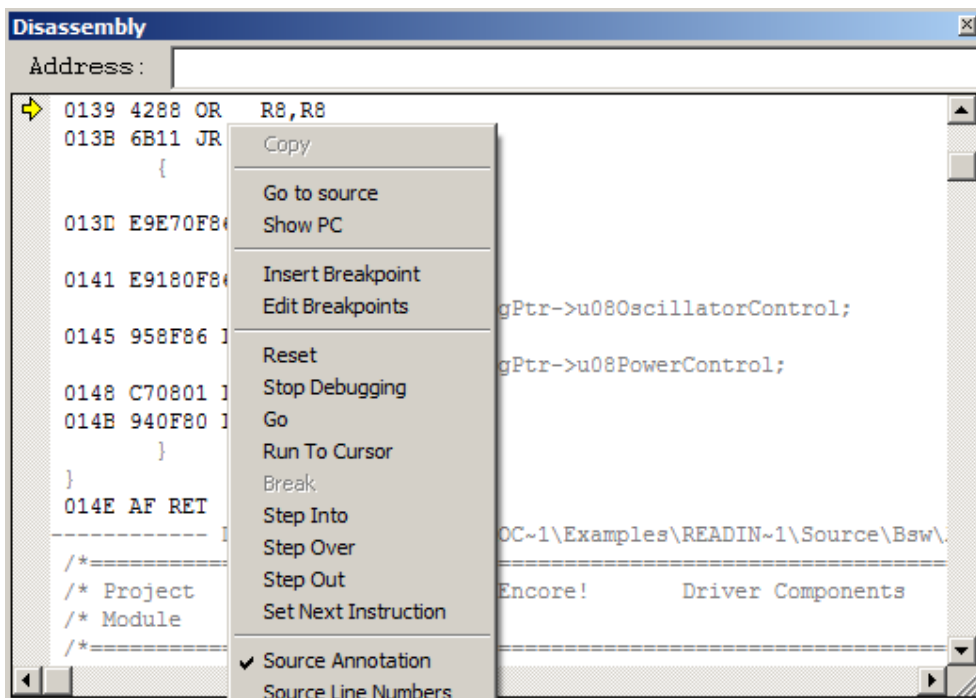
```

Фиг. 94 Прозорец Disassembly

Прозорецът **Disassembly** показва адресите, машинните кодове, асемблерните инструкции и техните операнди, съответстващи на всеки ред от С-кода. Забележете, че кодовите индикатори на Дебъгера също са активни в този

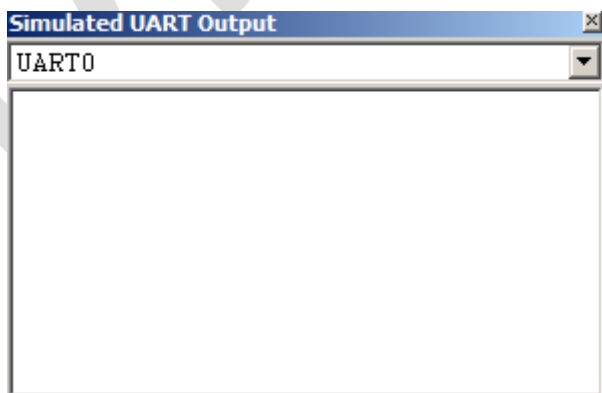
прозорец.

Фиг.95 показва допълнителните опции, достъпни с щракване с десен бутон в прозореца **Disassembly**.



Фиг. 95 Контекстно меню на прозореца Disassembly

39.7.4.10 Simulated UART Output



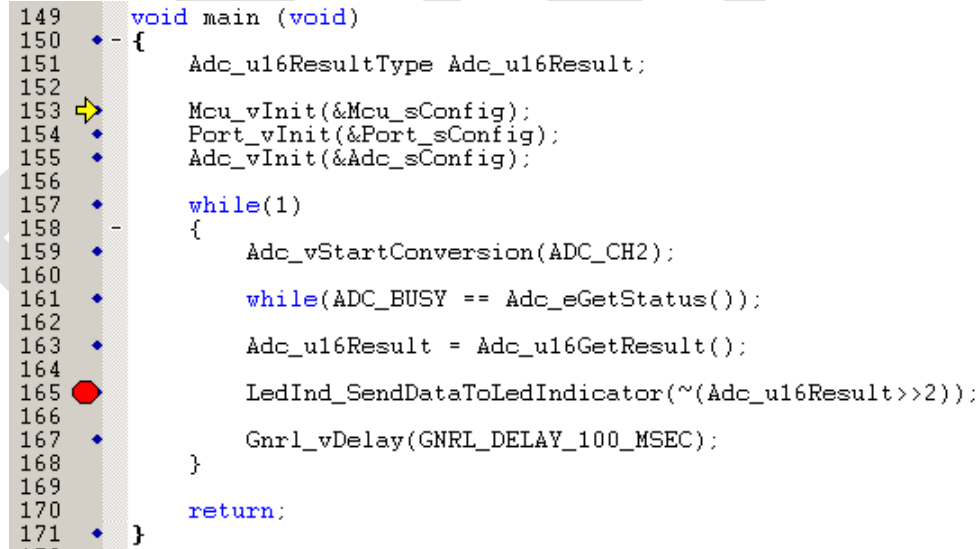
Фиг. 96 Прозорец Simulated UART Output

Вижте в помощното меню на развойната среда ZDS II (Help→Help Topics→Using the Debugger→Debug Windows→UART Output window).

39.7.5 Точки на прекъсване

39.7.5.1 Общи сведения

Точката на прекъсване е средство, чрез което програмата може да бъде временно спряна в определена точка от изпълнението си. Това позволява на програмиста да изследва стойностите на различни променливи и регистри, използвайки прозорците, предоставени от Дебъгера. Например на следващата фигура програмният брояч се намира на ред 153 (жълтата стрелка), а на ред 165 е разположена точка на прекъсване (червения осмоъгълник).



```
149 void main (void)
150 {
151     Adc_u16ResultType Adc_u16Result;
152
153     Mcu_vInit(&Mcu_sConfig);
154     Port_vInit(&Port_sConfig);
155     Adc_vInit(&Adc_sConfig);
156
157     while(1)
158     {
159         Adc_vStartConversion(ADC_CH2);
160
161         while(ADC_BUSY == Adc_eGetStatus());
162
163         Adc_u16Result = Adc_u16GetResult();
164
165         LedInd_SendDataToLedIndicator(~(Adc_u16Result>>2));
166
167         Gnrl_vDelay(GNRL_DELAY_100_MSEC);
168     }
169
170     return;
171 }
```

Фиг. 97 Поставяне на точка на прекъсване

Ако стартирате програмата с бутон , при достигане на ред 165 изпълнението ѝ спира (Фиг.98).

```

149 void main (void)
150 -{
151     Adc_u16ResultType Adc_u16Result;
152
153     Mcu_vInit(&Mcu_sConfig);
154     Port_vInit(&Port_sConfig);
155     Adc_vInit(&Adc_sConfig);
156
157     while(1)
158     {
159         Adc_vStartConversion(ADC_CH2);
160
161         while(ADC_BUSY == Adc_eGetStatus());
162
163         Adc_u16Result = Adc_u16GetResult();
164
165         LedInd_SendDataToLedIndicator(~(Adc_u16Result>>2));
166
167         Gnrl_vDelay(GNRL_DELAY_100_MSEC);
168     }
169
170     return;
171 }

```

Фиг. 98 Спиране на програмата в точка на прекъсване

39.7.5.2 Вмъкване на точка на прекъсване

Има няколко начина за вмъкване на точка на прекъсване:

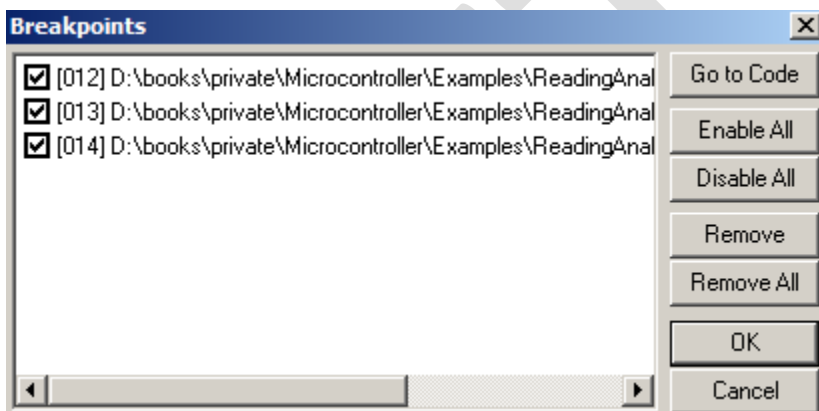
- Двукратно щракване в лентата вляво на реда, където искате да поставите точката на прекъсване.
- Поставете курсора на реда, където искате да поставите точката на прекъсване. Щракнете с десен бутон на мишката и от контекстното меню изберете **Insert Breakpoint** или натиснете бутона  от лентата с инструменти

Забележка: Точките на прекъсване могат да се поставят само на редове със синя точка. Сините точки се виждат само когато Дебъгерът е стартиран (вижте [Описание на менюто Debug](#)).

39.7.5.3 Преглед на всички точки на прекъсване

Има два начина да прегледате всички точки на прекъсване в проекта:

- От меню **Edit**→**Manage Breakpoints...**;
- С дясно щракване в произволно място в кода и избиране на **Edit Breakpoints** от контекстното меню.



Фиг. 99 Breakpoints

Прозорецът **Breakpoints** позволява преместване на курсора на ред с точка на прекъсване, забраняване, разрешаване и премахване на една или повече точки на прекъсване.

39.7.5.4 Преместване на курсора в точка на прекъсване

За да преместите курсора на ред с вече поставена точка на прекъсване, отворете прозореца **Breakpoints**, изберете точка на прекъсване и натиснете бутона **Go to Code**.

39.7.5.5 Разрешаване на точки на прекъсване

За да разрешите вече забранена точка на прекъсване, направете едно от следните неща:

- Щракнете два пъти върху забранената точка на прекъсване;
- Поставете курсора на ред с точка на прекъсване и натиснете бутона  от лентата с инструменти;
- Поставете курсора на ред с точка на прекъсване, щракнете с десен бутон на мишката и от контекстното меню изберете **Enable Breakpoint**;

За да разрешите всички забранени точки на прекъсване в кода, направете следното:

1. Отворете прозореца **Breakpoints** (**Edit→Manage Breakpoints...**);
2. Натиснете бутона **Enable All**;
3. Натиснете бутона **OK**.

39.7.5.6 Забраняване на точки на прекъсване

За да забраните разрешена точка на прекъсване, направете едно от следните неща:

- Поставете курсора на ред с точка на прекъсване и натиснете бутона  от лентата с инструменти;
- Поставете курсора на ред с точка на прекъсване,

щракнете с десен бутон на мишката и от контекстното меню изберете **Disable Breakpoint**;

За да забраните всички разрешени точки на прекъсване в кода, направете следното:

1. Отворете прозореца **Breakpoints (Edit→Manage Breakpoints...)**;
2. Натиснете бутона **Disable All**;
3. Натиснете бутона **OK**.

39.7.5.7 Премахване на точки на прекъсване

За да премахнете точка на прекъсване, направете едно от следните неща:

- Щракнете два пъти върху точката на прекъсване;
- Поставете курсора на ред с точка на прекъсване и натиснете бутона ;
- Поставете курсора на ред с точка на прекъсване, щракнете с десен бутон на мишката и от контекстното меню изберете **Remove Breakpoint**;

За да премахнете всички точки на прекъсване в кода, направете следното:

1. Отворете прозореца **Breakpoints (Edit→Manage Breakpoints...)**;
2. Натиснете бутона **Remove All**;

3. Натиснете бутона **ОК**.

DRAFT

Част V

Въведение в програмирането на микроконтролера Z8F083A на езика C

```
17 string sInput;  
18 int iLength, iN;  
19 double dbTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, sInput);  
26     system("cls");  
27     stringstream(sInput) >> dbTemp;  
28     iLength = sInput.length();  
29     if (iLength < 4) {  
30         again = true;  
31         continue;  
32     } else if (sInput[iLength - 3] != '.') {  
33         again = true;  
34         continue;  
35     } while (++iN < iLength) {  
36         if (isdigit(sInput[iN])) {  
37             continue;  
38         } else if (iN == (iLength - 3)) {  
39             continue;  
40         }  
41     }  
42 }
```

DRAFT

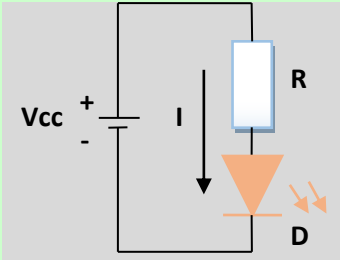
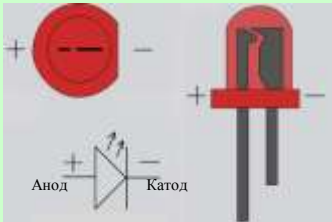
40 Предисловие

Може би вече сте нетърпеливи да напишете първите си редове код ☺. Ето че и този момент най-после дойде. В тази част ще започнем с няколко сравнително малки примери, които ще Ви помогнат да разберете по-добре как работят периферните модули на микроконтролера Z8F083A. Ще започнем по начин, който е по-подходящ за начинаещи, т.е. . в тази част няма да следваме правилата описани в част III, като структуриране на програмата, сорс-файловете и т.н., тъй като това ще направи старта по-труден. Целта на тази част е да разберете как работи микроконтролера на практика. В началото на всяко упражнение ще Ви се налага да се обръщате към съответната глава от описанието на микроконтролера. Периферните модули на Z8F083A са доста опростени, съответно описанието им е кратко, така че това няма да Ви отнеме много време, особено ако вече сте го чели. По моя преценка съм добавил допълнителни обяснения, свързани с работата на периферните модули, за да може по-бързо и лесно да се ориентирате в материята. За упражненията в тази част ще използваме основната платка [Z8Encore!_MB01](#) и разширителната платка [Z8Encore!_EB01\(MB01\)](#). Във всички примери ще използваме вътрешния прецизен генератор 20MHz, който е подразбиращият се източник на системен тактов сигнал на микроконтролера след ресет.

Съветвам Ви да разгледате примерите в реда, в който са представени, тъй като във всеки следващ пример ще използваме неща от предходните.

41 Управление на светодиода (вариант 1)

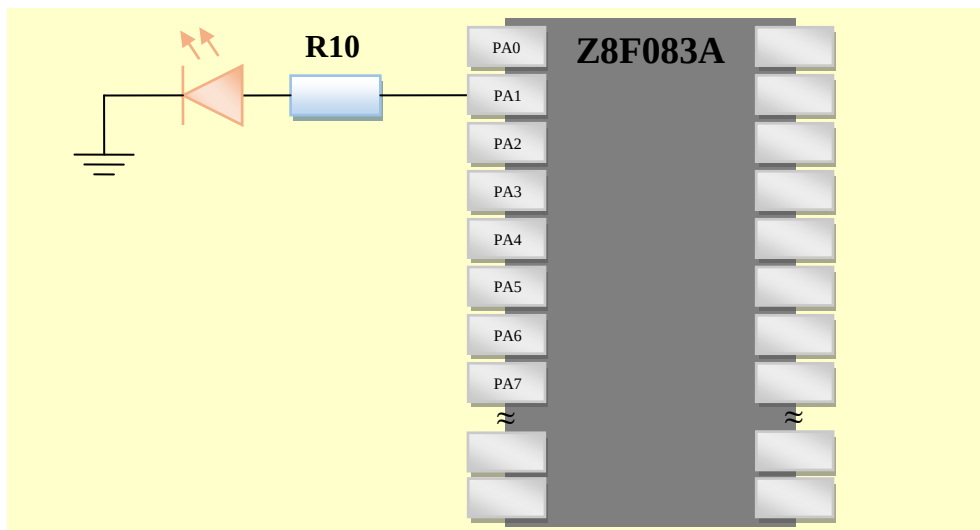
Първият пример, с който ще започнем, е да накараме светодиода D1 от разширителната платка **Z8Encore!_EB01 (MB01)** да светне. Като начало свържете платката към основната платка **Z8Encore!_MB01** и се уверете, че джъмперът JM2 е в позиция PWM.

	Какво представлява светодиода? Светодиодът (LED – Light Emitted Diode) е електронно устройство с два извода, което излъчва светлина при протичане на ток през него. Изводите на светодиода се обозначават като Анод (+) и Катод (-). Колкото е по-голям тока през светодиода, толкова по-ярко свети той. Типично големината на тока е в диапазона $I = 5 \div 20 \text{mA}$.  Фиг. 100 Светодиод
---	--

Следващата фигура представлява опростен изглед на част от принципната електрическа схема, която ще ни служи за ориентир.

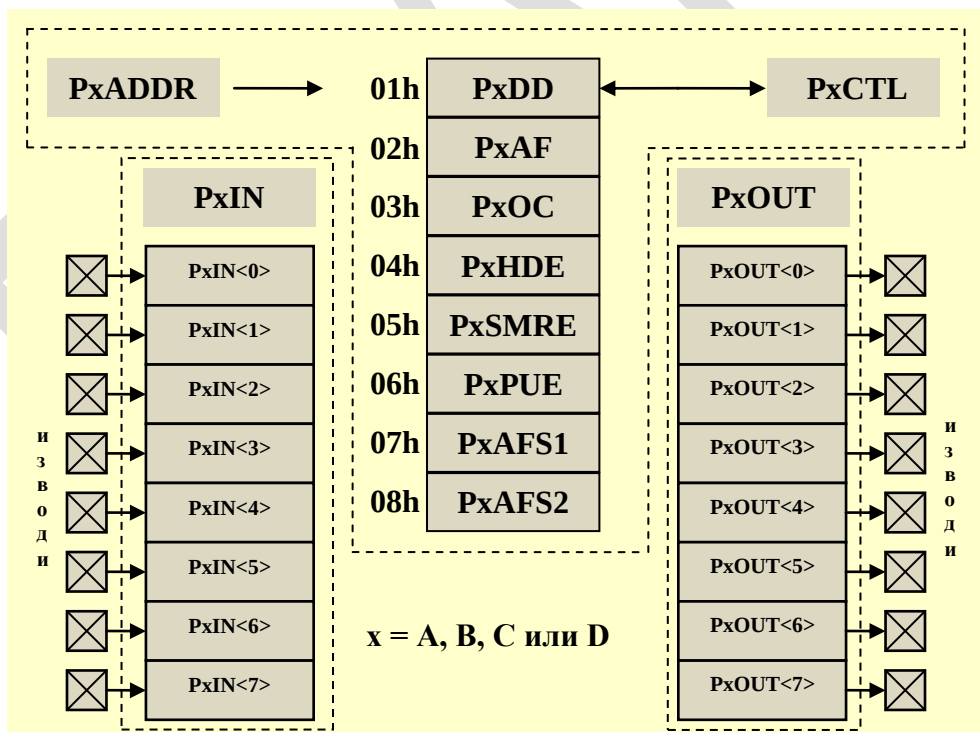
Светодиодът D1 е свързан през резистора R10 към извод PA1 на микроконтролера. За да накараме светодиода да светне, е необходимо да направим две неща:

1. Да конфигурираме извода като изход;
2. Да изведем лог.1 на извода.



Фиг. 101 Управление на светодиод – опростена схема

Фиг.102 ще Ви припомни управляващите регистри на портовете на микроконтролера Z8F083A.



Фиг. 102 Управляващи регистри на порт

Обърнете внимание, че част от регистрите на порта са вътрешни и не са директно достъпни, а се достигат чрез регистрите P_xADDR и P_xCTL (x = A, B, C и D).

За да запишете стойност в някой от вътрешните регистри, е необходимо първо да изберете регистъра, като запишете адреса му в P_xADDR, след което да запишете стойността в P_xCTL. Стойността, записана в P_xCTL, се копира в регистъра, сочен от P_xADDR.

За да прочетете стойността на някой от вътрешните регистри, е необходимо първо да изберете регистъра, като запишете адреса му в P_xADDR. Стойността на избрания регистър, се копира в P_xCTL. Това, което остава да направите е просто да прочетете стойността на P_xCTL.

Няма да разглеждам ролята на вътрешните регистри. Ако сте забравили кой регистър за какво служи, обърнете се към [10.4 Описание на управляващите регистри](#).

Когато даден извод на порт е конфигуриран като вход, стойността на извода се записва в съответния бит P_xIN<n> (вижте Фиг.102 вляво).

Когато даден извод на порт е конфигуриран като изход, стойността на съответния бит P_xOUT<n> се извежда на извода (вижте Фиг.102 вдясно).

След като си припомним как работят портовете, нека да се върнем към нашата задача. Както споменах по-рано, първо трябва да конфигурираме извод PA1 като изход. За целта е необходимо да нулираме бит 1 на регистър PADD. Това може да стане по следния начин:

```
PAADDR = 0x01; /* Избираме регистър PADD */  
PACTL &= 0xFD; /* Нулираме бит PADD<1> */
```

Обърнете внимание на стойността 0xFD. За по-ясно ще я разпиша в двоичен вид – 1111 1101. Забележете, че позицията, която искаме да нулираме, съдържа стойност 0, а всички останали битове има стойност 1, след което се извършва битовата операция И (&) между тази константа и съдържанието на регистъра PADD. Резултатът се записва отново в PADD. Тази операция ще нулира бит PADD<1>, а останалите битове на PADD ще запазят стойностите си непроменени.

Сега остава да изведем 1-ца на извод PA1. За целта трябва да запишем 1-ца в бит PAOUT<1>. Това може да стане по следния начин:

```
PAOUT |= 0x02; /* Установяваме бит PAOUT<1> */
```

Обърнете внимание на стойността 0x02. За по-ясно ще я разпиша в двоичен вид – 0000 0010. Забележете, че позицията, която искаме да установим, съдържа стойност 1, а всички останали битове имат стойност 0, след което се извършва битовата операция ИЛИ (|) между тази константа и съдържанието на регистъра PAOUT. Резултатът се записва отново в PAOUT. Тази операция ще установи бит PAOUT<1>, а останалите битове на PAOUT ще запазят стойностите си непроменени.

Ето и цялата програма:

```
void main(void)
{
    PAADDR = 0x01; /* Избираме регистър PADD */
    PACTL &= 0xFD; /* Нулираме бит PADD<1> */
    PAOUT |= 0x02; /* Установяваме бит PAOUT<1> */
    while(1); /* Зацикляме изпълнението на програмата тук */
}
```

Компилирайте кода, свържете дебъгера **USB SmartCable** към основната платка, подайте захранване, заредете програмата в микроконтролера и стартирайте изпълнението

й. Ако използвате бутона , за да заредите програмата, ще Ви се отвори файлът startup.asm.

```
34      ; Reset vector
35      ;
36      VECTOR    reset=_c_startup
37
38      DEFINE startup, _space-rom
39
40      SEGMENT startup
41
42      _c_startup:
43
44      SRP        #100                ; Set register pointers, Working registers 00-0f
45
46      IF ( _Z8F648=1)
47      FLLDONE     EQU    3
48                  ;check FLL done
49      $$:         LAX    R0, CLKCTL5
50                  STWZ   FLLDONE, R0, $B
51      ENDIF
```

Това е стандартен стартов файл, предоставен от С-компилятора, който извършва необходимите инициализации преди стартиране на програмата (преди извикване на функцията main). Може да използвате директно бутона . Програмата ще се зареди, инициализиращият код от startup.asm ще се изпълни и индикаторът на програмния брояч () ще се установи в началото на функцията main.

```
void main(void)
{
    PAADDR = 0x01; /* ***** PADD */
    PACTL &= 0xFD; /* ***** PADD<1> */
    PAOUT |= 0x02; /* ***** PAOUT<1> */
    while(1); /* ***** */
}
```

Като начало може да стартирате програмата с бутона . След това може да поекспериментирате с постъпково изпълнение на програмата, да сложите точка на прекъсване, да разгледате как се променят стойностите на регистрите чрез прозореца **Special Function Register** и т.н. Ролите на бутоните и различните прозорци на Дебъгера са описани в

39.7 Описание на Дебъгера.

Преди да приключим с примера искам да Ви наблегна на някои трикове, които ще Ви бъдат от полза.

Нулирането (Установяването) на бит може да стане по няколко начина. Единия вече го видяхте - чрез използване на константа, която съдържа нули (единици) в позициите, които искаме да нулираме (установим) и единици (нули) във всички останали. Прието е константите, които се използват за такива цели, да се наричат **маски**. Една маска може да се използва за нулиране (установяване) на един или няколко бита едновременно. Просто сложете нули (единици) в желаните позиции. Маската може обаче да се формира и по друг начин, който е самодокументиращ се и позволява по-лесно да се определи кои битове искаме да нулираме (установим).

Формиране на нулираща маска:

`~(1 << позиция)`

където

позиция указва позицията (номера на бита), в която искаме да преместим 1-цата (или с други думи, указва бита, който искаме да нулираме).

Горната конструкция премества 1-цата в позицията указана с **позиция**, след което резултатът се инвертира (нулите стават единици, а единицата - нула).

Например редът

```
РАСТL &= 0xFD; /* Нулираме бит PADD<1> */
```

може да се замени с

```
РАСТL &= ~(1 << 1); /* Нулираме бит PADD<1> */
```

Формиране на установяваща маска:

1 << позиция

където

позиция указва позицията (номера на бита), в която искаме да преместим 1-цата (или с други думи, указва бита, който искаме да установим).

Горната конструкция премества 1-цата в позицията указана с **позиция**.

Например редът

```
RAOUT |= 0x02; /* Установяваме бит RAOUT<1> */
```

може да се замени с

```
RAOUT |= 1 << 1; /* Установяваме бит RAOUT<1> */
```

По гореописания начин може да се формира маска за нулиране (установяване) само на един бит. За да формирате маска за нулиране (установяване) на няколко бита, създайте си маски за позициониране на 1-цата в съответните битове и ги свържете с операцията битово ИЛИ (`|`). Например:

`(1 << 0)` – маска за позициониране на 1-цата в бит 0

`(1 << 3)` – маска за позициониране на 1-цата в бит 3

`(1 << 7)` – маска за позициониране на 1-цата в бит 7

`~((1 << 0) | (1 << 3) | (1 << 7))` – маска за нулиране на битове 0, 3 и 7

`(1 << 0) | (1 << 3) | (1 << 7)` – маска за установяване на битове 0, 3 и 7

След всичко казано дотук, нашата програма може да бъде пренаписана по следния начин:

```
void main(void)
{
    PADDR = 0x01;          /* Избираме регистър PADD */
    PACTL &= ~(1 << 1);    /* Нулираме бит PADD<1> */
}
```

```

    PAOUT |= 1 << 1; /* Установяваме бит PAOUT<1> */
    while(1); /* Зацикляме изпълнението на програмата тук */
}

```



Обърнете внимание, че битовете в регистър (променлива) се номерират, започвайки от 0, т.е. . най-младшият бит има номер 0, а най-старшият бит има номер N-1, където N е броят на битовете в регистъра (променливата).

За по-лесен достъп до вътрешните регистри на портовете eZ8 C-компиляторът предоставя макросите

```

#define PxDD    PxADDR=0x01,PxCTL
#define PxAF    PxADDR=0x02,PxCTL
#define PxOC    PxADDR=0x03,PxCTL
#define PxHDE   PxADDR=0x04,PxCTL
#define PxSMRE  PxADDR=0x05,PxCTL
#define PxPUE   PxADDR=0x06,PxCTL
#define PxAFS1  PxADDR=0x07,PxCTL
#define PxAFS2  PxADDR=0x08,PxCTL

```

x = A, B, C или D

дефинирани в хедъра <ez8.h>. Това позволява да достъпвате тези регистри като всички останали, чрез имената им. Например тези два реда

```

PAADDR = 0x01; /* Избираме регистър PADD */
PACTL &= 0xFD; или PACTL &= ~(1 << 1); /* Нулираме бит PADD<1> */

```

могат да се заменят само с един ред

```

PADD &= 0xFD; или PADD &= ~(1 << 1); /* Нулираме бит PADD<1> */

```

Заменете макроса PADD с неговото тяло и ще видите, че се разгъва до конструкцията

```

PAADDR=0x01,PACTL &= 0xFD; или PAADDR=0x01,PACTL &= ~(1 << 1);

```

която е еквивалентна на горните два реда с тази разлика, че след първия операторен израз (PAADDR=0x01) се използва

запетая вместо точка и запетая. Ако този синтаксис Ви се струва странен, просто си припомнете как работи операцията за последователно изчисляване на изрази.

След тази промяна програмата ще изглежда по следния начин:

```
#include <ez8.h>

void main(void)
{
    PADD  &= ~(1 << 1); /* Конфигуриране на PA1 като изход */
    PAOUT |= 1 << 1; /* Включване на светодиода */
    while(1); /* Зацикляне на програмата */
}
```

И в заключение нека да упражним използването на функции. За целта ще поставим кода на програмата във функция, която ще си дефинираме сами и ще наречем примерно TurnOnLed.

```
void TurnOnLed(void)
{
    PADD  &= ~(1 << 1); /* Конфигуриране на PA1 като изход */
    PAOUT |= 1 << 1; /* Включване на светодиода */
}
```

Това, което остава да направим, е да извикаме функцията в тялото на main(). Ето и цялата програма.

```
#include <ez8.h>

extern void TurnOnLed(void);

void main(void)
{
    TurnOnLed(); /* Включване на светодиода */
    while(1); /* Зацикляне на програмата */
}

void TurnOnLed(void)
{
    PADD  &= ~(1 << 1); /* Конфигуриране на PA1 като изход */
    PAOUT |= 1 << 1; /* Включване на светодиода */
}
```



```
}
```

Тъй като функцията е дефинирана след извикването ѝ, е необходимо да добавим прототипа ѝ преди самото извикване. Ключовата дума `extern` не е задължителна, но добрите програмни практики изискват да се използва.

Програмата може да се напише и по следния начин:

```
#include <ez8.h>

void TurnOnLed(void)
{
    PADD  &= ~(1 << 1); /* Конфигуриране на PA1 като изход */
    PAOUT |= 1 << 1;    /* Включване на светодиода */
}

void main(void)
{
    TurnOnLed(); /* Включване на светодиода */
    while(1); /* Зацикляне на програмата */
}
```

В този случай дефиницията на функцията се намира преди извикването ѝ. При това положение прототипът може да се изпусне, тъй като дефиницията играе роля и на прототип. Файловете се компилират отгоре-надолу, така че компилаторът получава информация за параметрите и връщания резултат на функцията от дефиницията ѝ.

С това приключваме първия пример. Надявам се, че Ви е било забавано и интересно 😊.

42 Управление на светодиода (вариант 2)

В този пример ще разнообразим предходния, като накараме светодиода да мига. За целта е необходимо да изпълним следните стъпки:

1. Включваме светодиода;
2. Изчакаваме определено време (примерно 1 секунда);
3. Изключваме светодиода;
4. Изчакаваме определено време (примерно 1 секунда);
5. Връщаме се към т.1.

Вече знаем как да включим светодиода. Това което трябва да направим е да добавим изключване на светодиода и времезадръжка от 1 секунда, и да зациклим изпълнението на програмата.

За да изключим светодиода, просто трябва да изведем лог.0 на извод РА1. Това може да стане по следния начин:

```
RAOUT &= ~(1 << 1); /* Нулираме бит RAOUT<1> */
```

Относно времезадръжката най-простият вариант е софтуерна задръжка, реализирана чрез цикъл, чието изпълнение отнема желаното време. Софтуерната задръжка може да бъде генерирана по следния начин:

```
unsigned long delayCounter;
```

```
for (delayCounter = 505250; delayCounter > 0; delayCounter--);
```

Може би се чудите откъде взех стойността 505250. Просто използвах прозореца **Clock** и с негова помощ експеримен-

тално нагласих времето за изпълнението на цикъла for да е около 1 секунда. Спомнете си, че този прозорец може да се използва само в режим на Симулатор (**Project** → **Settings...** → **Debugger** → **DebugTool** → **Simulator**). Вместо обаче да използваме такива „магически числа“ с неясен смисъл, можем да заменим константата 505250 с име, което подсказва нейното предназначение. Това може да стане най-лесно с директивата #define:

```
#define DELAY_CONSTANT_1_SEC (505250)

unsigned long delayCounter;

for (delayCounter = DELAY_CONSTANT_1_SEC; delayCounter > 0;
delayCounter--);
```

Цялата програма изглежда по следния начин:

```
#include <ez8.h>

#define DELAY_CONSTANT_1_SEC (505250)

void main(void)
{
    unsigned long delayCounter;

    PADD &= ~(1 << 1); /* Извод PA1 е изход */

    while (1)
    {
        PAOUT |= 1 << 1; /* Установяване на извод PA1 */
        for (delayCounter = DELAY_CONSTANT_1_SEC; delayCounter > 0;
            delayCounter--); /* Изчакване 1 секунда */
        PAOUT &= ~(1 << 1); /* Нулиране на извод PA1 */
        for (delayCounter = DELAY_CONSTANT_1_SEC; delayCounter > 0;
            delayCounter--); /* Изчакване 1 секунда */
    }
}
```

Можем да подобрим програмата като разпределим отделните задачи във функции. Например:

PortConfig - конфигурира на портовете
TurnOnLed - включва светодиода

TurnOffLed - изключва светодиода

Delay_1s - осигурява времезадръжка от 1 секунда

```
#include <ez8.h>

#define DELAY_CONSTANT_1_SEC (505250)

extern void PortConfig(void);
extern void TurnOnLed(void);
extern void TurnOffLed(void);
extern void Delay_1s(void);

void main(void)
{
    PortConfig(); /* Конфигуриране на изходите на портовете */

    while (1)
    {
        TurnOnLed(); /* Включване на светодиода */
        Delay_1s(); /* Изчакване 1 секунда */
        TurnOffLed(); /* Изключване на светодиода */
        Delay_1s(); /* Изчакване 1 секунда */
    }
}

void PortConfig(void)
{
    PADD &= ~(1 << 1);
}

void TurnOnLed(void)
{
    PAOUT |= 1 << 1;
}

void TurnOffLed(void)
{
    PAOUT &= ~(1 << 1);
}

void Delay_1s(void)
{
    unsigned long delayCounter;

    for (delayCounter = DELAY_CONSTANT_1_SEC; delayCounter > 0;
        delayCounter--);
}
```

Можем да направим функцията Delay_1s() универсална

вместо да генерира фиксирано закъснение. Ето как може да стане това:

```
void Delay(unsigned long delayCounter)
{
    for (; delayCounter > 0; delayCounter--);
}
```

Сега можем да извикваме функцията с различни аргументи и съответно да генерираме закъснения с различна продължителност.

```
#include <ez8.h>

#define DELAY_CONSTANT_1_SEC    (505250)
#define DELAY_CONSTANT_500_MSEC (252625)

extern void PortConfig(void);
extern void TurnOnLed(void);
extern void TurnOffLed(void);
extern void Delay_1s(void);

void main(void)
{
    PortConfig(); /* Конфигуриране на изходите на портовете */

    while (1)
    {
        TurnOnLed(); /* Включване на светодиода */
        Delay(DELAY_CONSTANT_500_MSEC); /* Изчакване 500ms */
        TurnOffLed(); /* Изключване на светодиода */
        Delay(DELAY_CONSTANT_500_MSEC); /* Изчакване 500ms */
    }
}

void PortConfig(void)
{
    PADD &= ~(1 << 1);
}

void TurnOnLed(void)
{
    PAOUT |= 1 << 1;
}

void TurnOffLed(void)
{
    PAOUT &= ~(1 << 1);
}
```

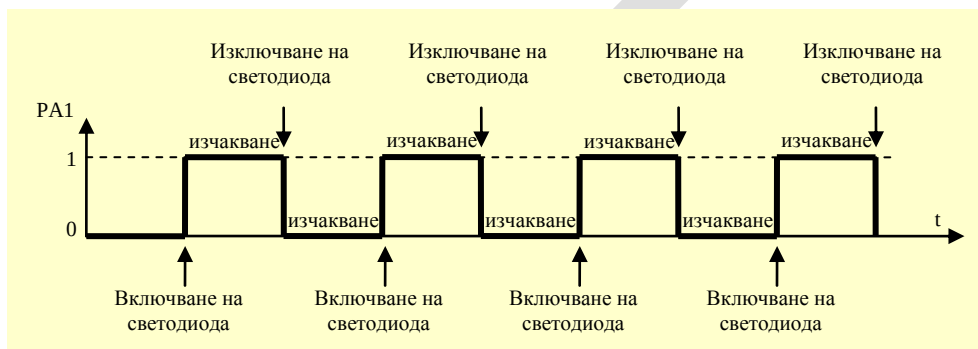
```

}

void Delay(unsigned long delayCounter)
{
    for (; delayCounter > 0; delayCounter--);
}

```

Всичко, което правят примерите по-горе е да генерират правоъгълни импулси на извод PA1 (Фиг.103).



Фиг. 103 Генериране на правоъгълни импулси

43 Управление на светодиода (вариант 3)

Примерът в тази точка отново ще генерира правоъгълни импулси на извод PA1, но с помощта на таймер. Спомнете си, че Z8F083A има два идентични таймери, които могат да работят в различни режими. Задачата може да бъде изпълнена и с двата таймера. Аз ще използвам примерно Таймер0. Целта на упражнението е да видим как работи таймера в „еднократен” и „непрекъснат” режим. С този пример ще направим и първия досег с прекъсванията. Необходимо е да изпълним следните стъпки:

1. Конфигурираме извод PA1 като изход;
2. Разрешаваме прекъсването от Таймер0;
3. Конфигурираме Таймера в „еднократен” или „непрекъснат” режим.
4. Превключваме светодиода в подпрограмата за обработка на прекъсването.
5. Връщаме се към т.4.

Първата стъпка е ясна, затова ще премина директно към втората. За да разрешим прекъсването от Таймер0, трябва да конфигурираме битове `IRQ0ENL<T0ENL>` и `IRQ0ENH<T0ENH>`. Тъй като имаме само един източник на прекъсване в програмата, е без значение какво ниво на приоритет ще изберем. Достатъчно е да установим единия от двата или и двата бита.

```
IRQ0ENL |= 1 << 5; /* IRQ0ENL<T0ENL> = 1 */  
IRQ0ENH |= 1 << 5; /* IRQ0ENH<T0ENH> = 1 */
```

За да се обработи прекъсването обаче, е необходимо да разрешим и глобалния бит `IRQE` (бит 7) в регистър `IRQCTL`.

IRQCTL |= 1 << 7; /* IRQCTL<IRQE> = 1 */

С това конфигурирането на прекъсванията приключва и преминаваме към конфигурирането на Таймер0. Ще направя един бърз преглед на работата на таймера. На всеки входен импулс на тактовия сигнал F_{TIMER} ($F_{\text{TIMER}} = F_{\text{sys}}/N$, където N е коефициент на деление) на входа на таймера, таймерните регистри T0H:T0L се увеличават с 1. Когато 16-битовата стойност в регистрите T0H:T0L достигне 16-битовата броячна стойност в регистрите T0RH:T0RL , таймерът се нулира (в случая терминът „нулира“ не е съвсем точен, тъй като дизайнерите са решили стойността в броячните регистри T0H:T0L да бъде 0001h, а не 0000h) и се генерира прекъсване. Ако таймерът е конфигуриран в „еднократен“ режим, след първото препълване той спира. Ако таймерът е конфигуриран в „непрекъснат“ режим, след всяко препълване таймерът автоматично продължава да се увеличава на всеки входен тактов импулс. За да можем да видим мигането на светодиода, периодът на правоъгълните импулси трябва да е над 20ms, т.е.. честотата на правоъгълните импулси да е под 50Hz.

Системната тактова честота $F_{\text{SYS}} = 20\text{MHz}$ е входният тактов сигнал за таймера. Таймерният модул, позволява входната честота да бъде разделена с коефициенти на деление 1, 2, 4, 8, 16, 32, 64 или 128. Ако изберем минимална входна честота на таймера, т.е.. максимален коефициент на деление 128, и максимална стойност на броене на таймера FFFFh (65535), то максималното време за препълване на таймера ще бъде:

$$T_{\text{OV}_{\text{max}}} [\mu\text{s}] = T_{\text{TIMER}_{\text{max}}} * T_{\text{T0RH:T0RL}_{\text{max}}},$$

където

$$T_{\text{TIMER}_{\text{max}}} = \frac{1}{F_{\text{TIMER}_{\text{min}}}} = \frac{N_{\text{max}}}{F_{\text{SYS}}} - \text{максимален период на входния тактов сигнал.}$$

$N_{max} = 128$ – максимален коефициент на деление на входния тактов сигнал на таймера.

$T_{TORH:TORL_{max}} = 65535$ – максималната броячна стойност на таймера.

$$T_{OV_{max}}[\mu s] = \frac{128}{20MHz} * 65535 = 419424$$

От горните изчисления излиза, че максималното време на препълване на таймера е 419424 μ s или приблизително 420 ms, т.е.. периодът на правоъгълните импулси ще бъде 420ms*2=840ms (1.2Hz).

След като имаме тази информация вече можем да преминем към конфигуриране на таймера.

```
/* Нулиране на таймерните регистри */
T0H = 0x00;
T0L = 0x00;

/* Зареждане на максималната броячна стойност */
T0RH = 0xFF;
T0RL = 0xFF;

/*-----*/
/* - Еднократен режим */
/* - Прекъсване при препълване */
/* - Коефициент на деление 128 */
/* - Стартиране на таймера */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0xB8;
```

Последното, което остана, е да превключваме светодиода в подпрограмата, която отговаря за препълване на таймера. Ще нарека тази подпрограма (функция) Timer0_ISR.

```
void interrupt Timer0_ISR(void) _At TIMER0
{
    PAOUT ^= 1 << 1; /* Превключване на нивото на извод PA1 */
    T0CTL1 |= 1 << 7; /* Стартиране на таймера отново */
}
```

```
}
```

Начините, по-които може да укажете, че една функция се използва за обработване на дадено прекъсване са описани в [39.5.2.4 Прекъсвания](#).

Тъй като сме конфигурирали таймера в „еднократен” режим, преди да излезем от подпрограмата за обработка на прекъсването е необходимо да стартираме таймера отново. Флагът на прекъсване автоматично се нулира при приемане на прекъсването за обработка, така че не е необходимо да се нулира в кода. Целият код на програмата изглежда по следния начин:

```
#include <ez8.h>

void interrupt Timer0_ISR(void) _At TIMER0
{
    PAOUT ^= 1 << 1; /* Превключване на светодиода:
                     0->1 или 1->0 */
    T0CTL1 |= 1 << 7; /* Стартиране на таймера отново */
}

void main(void)
{
    /* Конфигуриране на извод PA1 като цифров изход */
    PADD &= ~(1 << 1);

    /* Конфигуриране на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;

    /*-----*/
    /* Конфигуриране на Таймер0 */
    /*-----*/

    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xFF;
    T0RL = 0xFF;

    /*-----*/
}
```

```

/* - Еднократен режим */
/* - Прекъсване при препълване */
/* - Коефициент на деление 128 */
/* - Стартиране на таймера */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0xB8;

while(1);
}

```

Обърнете внимание на начина, по който се превключва светодиода. Използва се битовата операция изключващо ИЛИ (^) между стойността на бит PAOUT<1> и 1. При това положение ако PAOUT<1> е 0, след операцията ще стане 1 и обратно, ако е 1, ще стане 0, т.е. нивото на извод PA1 се превключва от 0 в 1 или от 1 в 0.

Ако конфигурираме таймера в „непрекъснат” режим, е достатъчно да го стартираме само един път.

```

#include <ez8.h>

void interrupt Timer0_ISR(void) _At TIMER0
{
    PAOUT ^= 1 << 1; /* Превключване на светодиода:
                       0->1 или 1->0 */
}

void main(void)
{
    /* Конфигуриране на извод PA1 като цифров изход */
    PADD &= ~(1 << 1);

    /* Конфигуриране на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;

    /*-----*/
    /* Конфигуриране на Таймер0 */
    /*-----*/

    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;
}

```

```

/* Зареждане на максималната броячна стойност */
T0RH = 0xFF;
T0RL = 0xFF;

/*-----*/
/* - Непрекъснат режим          */
/* - Прекъсване при препълване  */
/* - Коефициент на деление 128 */
/* - Стартиране на таймера      */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0xB9;

while(1);
}

```

Ако погледнете в описанието на изводите на микроконтролера, ще видите, че извод PA1 освен като цифров вход/изход може да работи и като изход на Таймер0. От описанието на таймера пък се вижда, че този изход се превключва при всяко препълване на таймера. Можем да се възползваме от това и да реализираме програмата по друг начин. За целта е необходимо да конфигурираме извод PA1 в режим на таймерен изход. Това може да стане по следния начин:

```

PAAF |= 1 << 1; /* PA1 е таймерен изход */

```

След като превключването на извод PA1 става автоматично, функцията за обработка на прекъсването не е необходима. Програмата придобива следния вид:

```

#include <ez8.h>

void main(void)
{
    /* Конфигуриране на извод PA1 като таймерен изход */
    PAAF |= 1 << 1;

    /*-----*/
    /* Конфигуриране на Таймер0          */
    /*-----*/

    /* Нулиране на таймерните регистри */
    T0N = 0x00;
}

```

```

T0L = 0x00;

/* Зареждане на максималната броячна стойност */
T0RH = 0xFF;
T0RL = 0xFF;

/*-----*/
/* - Непрекъснат режим */
/* - Прекъсване при препълване */
/* - Коефициент на деление 128 */
/* - Стартиране на таймера */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0xB9;

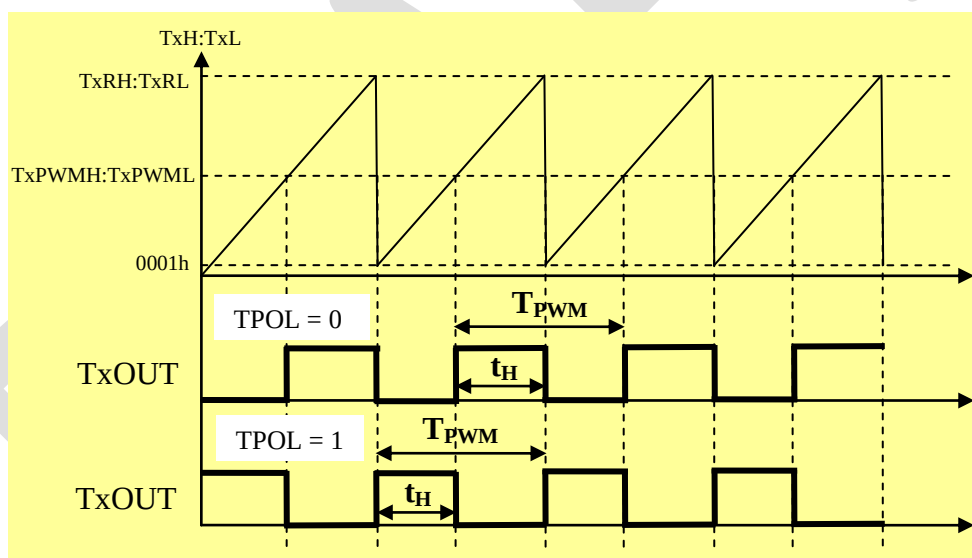
while(1);
}

```

Преди да преминете към следващия пример може да експериментирате с настройките на таймера. Например намалете броячната стойност наполовина (запишете стойност 32768 в T0RH:T0RL). Ще видите, че светодиодът ще започне да мига два пъти по-бързо от преди.

44 Управление на светодиода (вариант 4)

И отново продължаваме с нашия мигащ светодиод ☺. Пак ще използваме таймера, но този път ще го конфигурираме в „Едноизходен PWM” режим. Този режим позволява много лесно да генерираме правоъгълни импулси на таймерния изход с определена честота и определен коефициент на запълване. Припомнете си как работи този режим на [Едноизходен PWM режим](#). Най-общо казано периодът на генерирувания сигнал се определя от 16-битовата стойност в регистрите $T0RH:T0RL$, а коефициентът на запълване от 16-битовата стойност в регистрите $T0PWMH:T0PWML$. Бит $T0CTL1<TPOL>$ определя началното ниво на таймерния изход (Фиг.104).



Фиг. 104 Едноизходен PWM режим

Ще използваме информацията от предходната програма и ще накараме светодиода да мига с честота 2.4Hz (1/420ms) и коефициент на запълване 50%. Това означава, че честотата на тактовия сигнал на таймера ще бъде $F_{TIMER} = F_{SYS}/128$ и таймерът ще брои до максималната си стойност ($T0RH:T0RL = 65535$). За да осигурим коефициент на

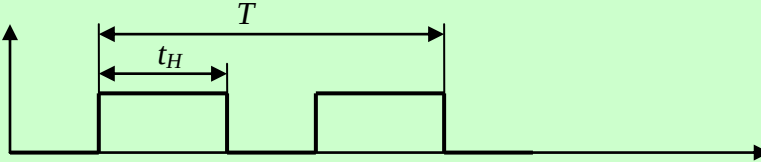
запълване 50%, е необходимо стойността в широчинните регистри TOPWMH:TOPWML да бъде половината от максималната броячна стойност на таймера или 32768 в нашия случай.

Какво е коефициент на запълване?



Коефициентът на запълване **k** на правоъгълен сигнал е отношението между продължителността на високото ниво **t_H** и периода **T** на сигнала.

$$k[\%] = \frac{t_H}{T} \times 100$$



Няма нищо магично в стойностите, които съм избрал. Идеята ми е когато е възможно да се избегне използването на специализирани уреди, като осцилоскоп, и резултатът от програмата да бъде видим с просто око.

Началното ниво на таймерния изход в случая е без значение. При изброените условия програмата изглежда по следния начин:

```
#include <ez8.h>

void main(void)
{
    /* Конфигуриране на извод PA1 като таймерен изход */
    PAAF |= 1 << 1;

    /*-----*/
    /* Конфигуриране на Таймер0 */
    /*-----*/

    /* Нулиране на таймерните регистри */
    T0H = 0x00;
```

```

T0L = 0x00;

/* Зареждане на броячната стойност */
TORH = 0xFF;
TORL = 0xFF;

/* Зареждане на широчинната стойност */
TOPWMH = 0x80;
TOPWML = 0x00;

/*-----*/
/* - Едноизходен PWM режим          */
/* - Ниско начално ниво (T0L = 0) */
/* - Коефициент на деление 128     */
/* - Стартиране на таймера          */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0xBB;

while (1);
}

```

Препоръчвам Ви да поекспериментирате с горния пример и да увеличите честотата на мигане на светодиода на 10, 20, 30, 40 и 50Hz. За целта променяйте стойността в броячните регистри TORH:TORL и коефициента на деление на тактовия сигнал на таймера. Не забравяйте да променят и стойността на широчинните регистри. За да поддържате коефициент на запълване 50%, стойността в тези регистри трябва да е наполовина на стойността на броячните регистри, т.е..

$$\kappa = 50\% \rightarrow TOPWMH:TOPWML = \frac{TORH:TORL}{2}$$

Забележка: Ако се затруднявате да изчислите стойността, която трябва да запишете в регистрите TORH:TORL, за да генерирате сигнал с желаната от Вас честота, използвайте вече позната формула:

$$T_{OV} = T_{TIMER} \times T_{TORH:TORL}$$

където

T_{OV} – време за препълване на таймера (период на генерирания сигнал)

$$T_{TIMER} = \frac{1}{F_{TIMER}} = \frac{N}{F_{SYS}} - \text{период на тактовия сигнал на таймера}$$

$T_{TORH:TORL}$ – броячна стойност на таймера

Следователно стойността на регистрите TORH:TORL се изчислява по следната формула:

$$T_{TORH:TORL} = \frac{T_{OV}}{T_{TIMER}} = T_{OV} \times F_{TIMER} = T_{OV} \times \frac{F_{SYS}}{N}$$

Пример: Генериране на сигнал с честота 50Hz (период 20ms)

$$T_{TORH:TORL} = 20ms \times \frac{20MHz}{1} = 400\,000$$

Получената стойност е твърде голяма за да се побере в TORH:TORL. Това означава, че трябва да увеличим коефициента на деление N. Не е трудно да се определи, че първият коефициент, който води до подходящ резултат, е 8.

$$T_{TORH:TORL} = 20ms \times \frac{20MHz}{8} = 50\,000 \text{ (C350h)}$$

Тази стойност вече е подходяща за запис в TORH:TORL. Промените, които трябва да направите в програмата са следните

```
/* Зареждане на броячната стойност 50 000 */
TORH = 0xC3;
TORL = 0x50;
```

```
/* Зареждане на широчинната стойност 25 000 */
TORWMH = 0x61;
TORWML = 0xA8;
```

```
/*-----*/
/* - Едноизходен PWM режим */
/* - Ниско начално ниво (TR0L = 0) */
/* - Коефициент на деление 8 */
/* - Стартиране на таймера */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0x9B;
```

Ако сте изпълнили съвета ми, ще забележите, че при генериране на честота 50Hz, мигането на светодиода вече не се забелязва. Причината се дължи на особеностите на човешкото зрение да възприема източници на светлина, мигащи с честота 48Hz или повече, като постоянно светещи.

Нека да се възползваме от тази особеност на човешкото зрение и да генерираме сигнал с честота примерно 200Hz (период 5ms) и да променяме коефициента на запълване с 10% през една секунда, като зациклим програмата да прави това постоянно. Изчисляването на стойностите на $T_{TORH:TORL}$, $T_{TOPWMH:TOPWML}$ и N вече е ясно:

$$T_{TORH:TORL} = 5ms \times \frac{20MHz}{2} = 50\,000 \text{ (C350h)}$$

$$T_{TOPWMH:TOPWML}[10\%] = \frac{50000}{10} = 5\,000 \text{ (1388h)}$$

След като ще променяме коефициента на запълване, началното ниво на сигнала е от значение. Погледнете отново фиг.104. Ако $TPOL = 0$, увеличаване на стойността на $T_{TOPWMH:TOPWML}$ води до намаляване на коефициента на запълване, докато ако $TPOL = 1$, е обратно. Аз ще избира $TPOL = 1$, тъй като поведението на сигнала в този случай е по-интуитивно за мен.

Вероятно се питате как ще става увеличаването на коефициента на запълване със стъпка 10%. Най-лесно е ако разрешим прекъсването при препълване на таймера. Тогава, при избраната честота, таймерът ще генерира прекъсване на всеки 5ms. Това означава, че на всеки 200 прекъсвания ($200 \times 5ms = 1000ms$) трябва да увеличаваме регистрите $T_{TOPWMH:TOPWML}$ с 5000. Вижте как е направено това в следващата програма. Обясненията се намират след нея.

[stdtypes.h](#)

```

#ifndef STDYPES_H
#define STDYPES_H

#define TRUE (1)
#define FALSE (0)

typedef unsigned char    boolean;
typedef signed char      sint8;
typedef signed short     sint16;
typedef signed long      sint32;
typedef unsigned char    uint8;
typedef unsigned short   uint16;
typedef unsigned long    uint32;
typedef float            float32;
typedef double           float64;

#endif

main.c

#include <ez8.h>
#include "stdtypes.h"

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint16 dutyCycle = 0; /* Съдържа стойността,
определяща коефициента на запълване k */

    static uint8 dutyCycleRefreshCounter = 0; /* Брояч,
определящ времето за обновяване на k */

    dutyCycleRefreshCounter++;

    if (200 == dutyCycleRefreshCounter) /* Изтече ли 1 секунда ?
*/
    { /* Да */
        dutyCycleRefreshCounter = 0; /* Нулиране на брояча */
        dutyCycle += 5000; /* Увеличаване на k с 10% */

        if (dutyCycle > 50000) /* k > 100% ? */
        { /* Да, повтаряне на процеса отначало */
            dutyCycle = 0;
            T0PWMH = 0x00;
            T0PWML = 0x01;
            T0H = 0x00;
            T0L = 0x00;
        }
        else
        { /* Не, обновяване на широчинните регистри */
            T0PWMH = (uint8)(dutyCycle >> 8);
            T0PWML = (uint8)dutyCycle;
        }
    }
}

```

```

        T0H = 0x00;
        T0L = 0x00;
    }
}

void main(void)
{
    /* Конфигуриране на извод PA1 като таймерен изход */
    PAAF |= 1 << 1;

    /*-----*/
    /* Конфигуриране на Таймер0 */
    /*-----*/

    /* Конфигуриране на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL |= 1 << 7;

    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /* Зареждане на широчинната стойност */
    T0PWMH = 0x00;
    T0PWML = 0x01;

    /*-----*/
    /* - Едноизходен PWM режим */
    /* - Високо начално ниво (TPOL = 1) */
    /* - Коефициент на деление 2 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0xCB;

    while (1);
}

```

Както ще забележите, ефектът от тази програма е промяна на яркостта на светене на светодиода. Колкото е по-голям коефициентът на запълване, толкова е по-голяма яркостта на светене. Първото нещо, което може би Ви прави

впечатление е файлът `stdtypes.h`, в който са добавени декларации на типове. Вместо да използваме стандартните имена за типове, като **unsigned char** или **unsigned int** и т.н., чийто размер зависи от компилатора и микроконтролера, по-удобно е да използваме унифицирани имена, като **uint8**, **uint16** и т.н., които ясно подсказват големината на типа. Ако програмата не Ви се компилира, уверете се че сте задали пътя към този файл в настройките на проекта (**Project**→**Settings...**→**C**→**Preprocessor**→**User Include Path**). Другият особен момент в програмата е увеличаването на регистрите `T0PWMH` и `T0PWML`. Тъй като това са два отделни 8-битови регистри, формиращи 16-битова стойност, определяща коефициента на запълване, директното им увеличаване със стъпка 5000 е неудобно. За целта в тялото на **Timer_ISR()** съм дефинирал 16-битова целочислена беззнакова локална статична променлива **dutyCycle**.

```
static uint16 dutyCycle = 0; /* Съдържа стойността, определяща
коефициента на запълване */
```

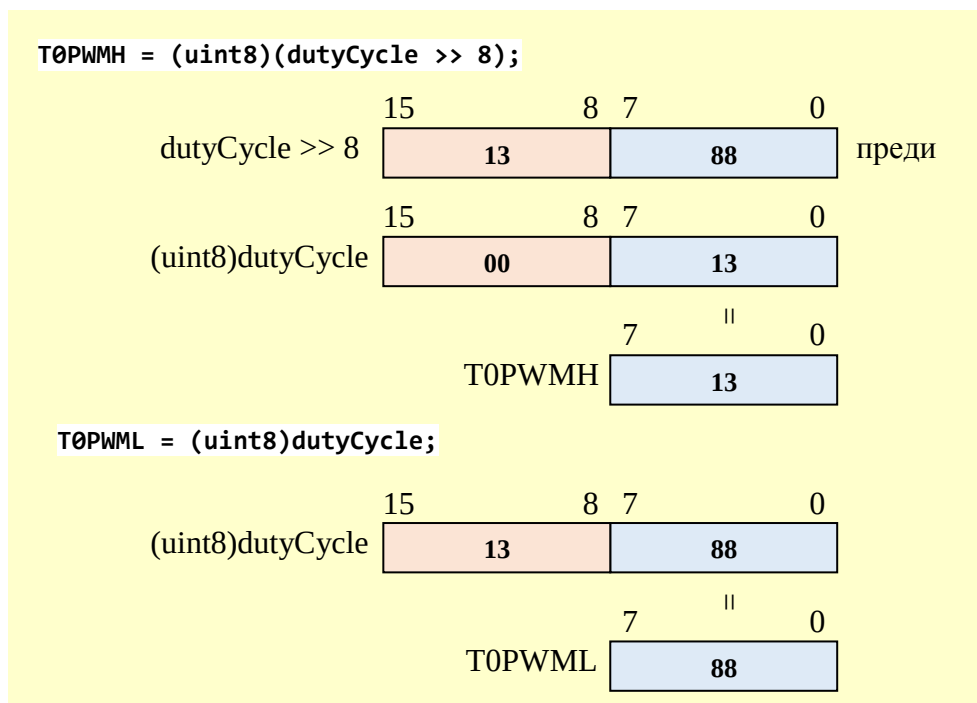
Когато изтече 1 секунда, променливата се увеличава с 5000, т.е.. с 10%.

```
dutyCycle += 5000; /* Увеличаване на k с 10% */
```

Ако получената стойност е под 50 000 (коефициентът на запълване е под 100%), тази стойност се присвоява на регистрите `T0PWMH` и `T0PWML`.

```
if (dutyCycle > 50000) /* k > 100% ? */
{ /* Да */
    ...
}
else
{ /* Не */
    T0PWMH = (uint8)(dutyCycle >> 8);
    T0PWML = (uint8)dutyCycle;
    ...
}
```

Следващата фигура онагледява процеса на присвояване на стойността на `dutyCycle` на регистрите `T0PWMH:T0PWL`.



Фиг. 105 T0PWMH:T0PWL = dutyCycle

Старшият байт на `dutyCycle` трябва да се присвои на `T0PWMH`, а младшият на `T0PWL`. В първия случай стойността на `dutyCycle` се премества с 8 бита надясно, след което се присвоява на `T0PWMH`. Тъй като регистърът `T0PWMH` е 8-битов, само младшият байт на 16-битовата променлива `dutyCycle` се копира в него. Не забравяйте, че операцията `>>` се прилага вътрешно върху временно копие, а не върху самата `dutyCycle`, т.е.. след операцията `dutyCycle` има същата стойност както и преди нея (1388h в нашия пример). По тази причина на следващия ред стойността на `dutyCycle` се присвоява директно на `T0PWL`, при което отново младшият байт се копира в регистъра. Преди двете присвоявания стойността на `dutyCycle` се преобразува в тип `uint8`. Тази операция по принцип е излишна, тъй като компилаторът ще извърши автоматично това преобразуване,

но добрите програмни практики изискват явното ѝ използване.

Ако получената стойност е над 50 000 (коефициентът на запълване е над 100%), целия процес се повтаря отначало.

```
if (dutyCycle > 50000) /* k > 100% ? */
{ /* Да */
    dutyCycle = 0;
    TOPWMH = 0x00;
    TOPWML = 0x01;
    T0H = 0x00;
    T0L = 0x00;
}
else
{ /* Не */
    ...
}
```

Обърнете внимание, че таймерът също се нулира (регистрите T0H и T0L). Това е необходимо, тъй като от момента на възникване на прекъсване до промяната на широчинните регистри таймерните регистри продължават да се увеличават и има вероятност стойността им да бъде по-висока от тази записана в широчинните регистри, което означава, че новият коефициент на запълване ще влезе в сила на следващото препълване на таймера. Това е вярно най-вече когато коефициентът на запълване се конфигурира да е 0%.

Относно отброяването на 1 секунда, мисля че няма нещо което да Ви затрудни в разбирането. За целта се използва 8-битова локална статична целочислена беззнакова променлива **dutyCycleRefereshCounter**, която се увеличава при всяко генериране на прекъсване от таймера. Когато dutyCycleRefereshCounter стане 200, означава че е време коефициентът на запълване да се увеличи с 10%. Забележете също, че променливите dutyCycle и dutyCycleRefereshCounter се използват само във функцията Timer_ISR(). По тази причина те са дефинирани вътре в нея.

Понеже е необходимо стойностите им да се запазват и след излизане от функцията, те са дефинирани като статични. Същият ефект може да се постигне ако променливите се дефинират като глобални, но това е лоша практика и не се препоръчва, освен ако не е абсолютно наложително.

И последно преди да приключим с примера, както вече съм Ви споменавал, използването на „магически” числа като 5000 в кода не е добра идея и трябва да се избягва. Използването на идентификатор с подходящо име е добра практика и е силно препоръчително.

```
#define DUTY_CYCLE_STEP    (5000)
```

```
...  
...
```

```
dutyCycle += DUTY_CYCLE_STEP;
```

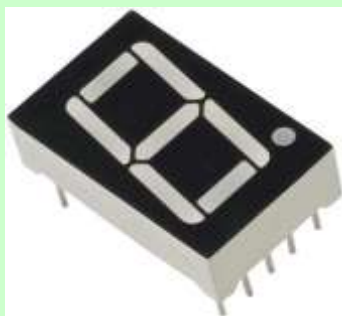
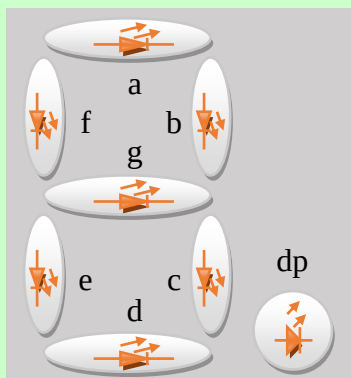
Друга полза от използването на макрос е, че ако константата се използва на повече от едно място в кода, то при промяна на стойността ѝ е необходимо да направите промяна само на едно място - в декларацията на макроса.

45 Управление на 7-сегментен дисплей

В тази точка ще Ви запозная с управлението на 7-сегментен дисплей. За целта свържете разширителната платка **Z8Encore!_EB01(MB01)** към основната платка **Z8Encore!_MB01** и се уверете, че джъмперът JM1 от разширителната платка е в позиция LED.

Какво представлява 7-сегментният индикатор?

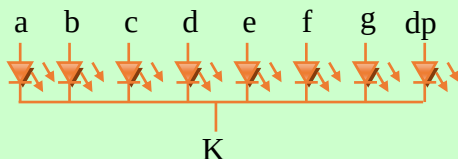
7-сегментният индикатор се състои от 7 светодиода в общ корпус, със специфична форма и разположение.



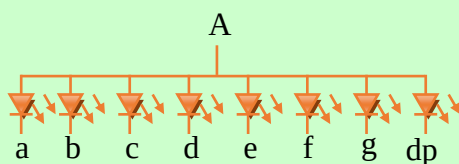
Фиг. 106 7-сегментен индикатор

Предлага се в два варианта:

- общ катод



- общ Анод



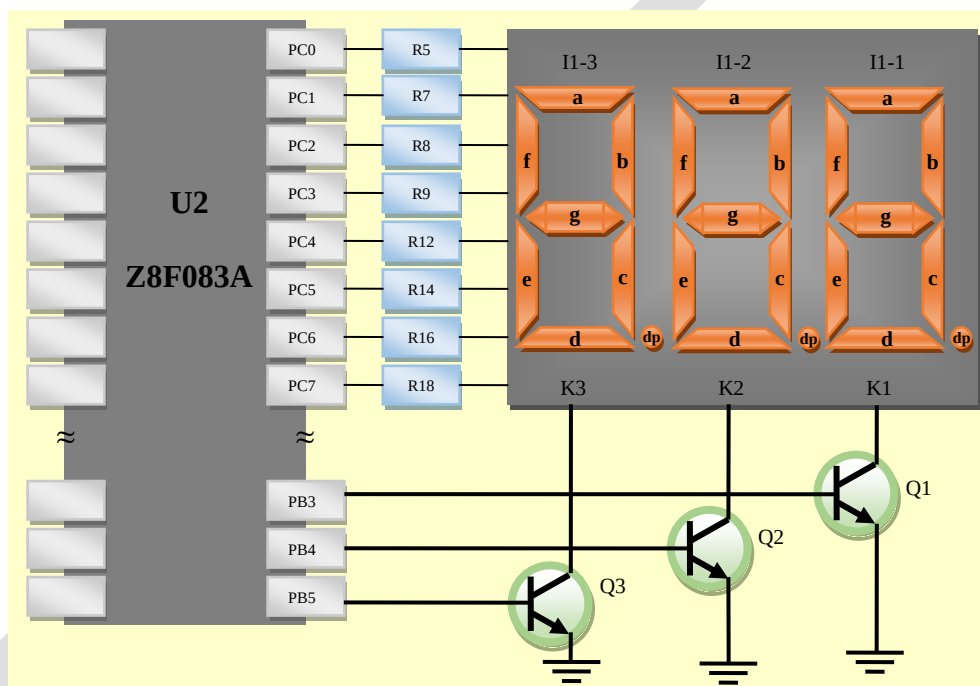
В долния десен ъгъл на индикатора често има допълнителен 8-ми сегмент, който представлява десетична запетая. Сегментите се именуват от **a** до **g** (**dp**). Общият катод (анод) и анодите (катодите) са изведени на изводи. В зависимост от това кои сегменти са активирани, на екрана се изписва съответната цифра (0÷9).

Произвеждат се поединично или по два, три и т.н. в корпус.



Фиг.107 представлява опростен изглед на част от принципната електрическа схема, която ще ни служи за ориентир. Ще използвам 7-сегментен дисплей **И1**, състоящ се от три 7-сегментни индикатори с общ катод. Анодите на едноимените сегментите на индикаторите вътрешно са

свързани помежду си и са изведени на извод. Ще номерирам условно индикаторите като индикатор1 (**I1-1**), индикатор2 (**I1-2**) и индикатор3 (**I1-3**), започвайки отдясно наляво. Съответно K1 е общият катод на индикатор1, K2 на индикатор2 и K3 на индикатор3. Катодите K1, K2 и K3 се свързват към маса, чрез цифровите транзистори Q1, Q2 и Q3 съответно.



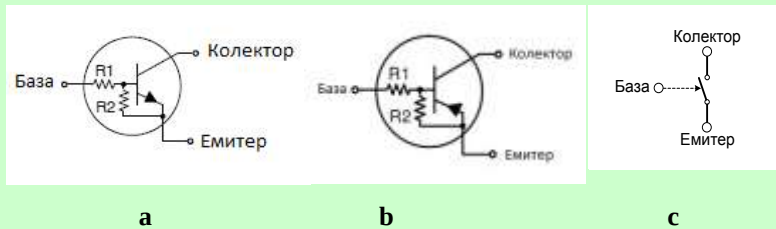
Фиг. 107 Управление на 7-сегментен дисплей – опростена схема

За простота първият пример, който ще разгледаме, ще използва само един индикатор от дисплея, примерно **I1-1**. Програмата ще изобразява циклично цифрите от 0 до 9 през една секунда.

Забележка: Всеки извод на микроконтролера може да отдава/приема ток с големина 25mA. Общият ток, обаче не трябва да надвишава 120mA. По тази причина ограничителните резистори R5, R7, R8, R9, R12, R14, R16, R18 са подбрани така, че всеки сегмент да консумира под 5mA, съответно един индикатор да консумира под 40mA. Така ако

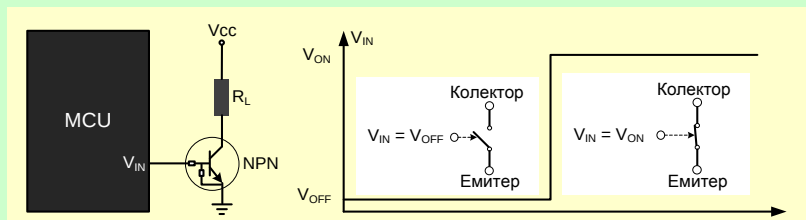
Какво представлява цифровият транзистор?

Цифровият транзистор е електронно устройство с три извода и играе ролята на ключ управляван с ток, т.е. . цифровият транзистор е електронен ключ. Предлага се в два варианта: NPN и PNP.

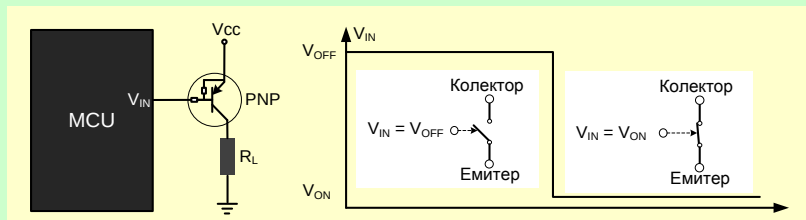


Фиг. 108 Цифров транзистор –а) NPN, б) PNP, в) еквивалентна схема

Следващата фигура описва типична употреба и принципа на работа на цифровите транзистори.



Фиг. 109 NPN цифров транзистор –принцип на работа



Фиг. 110 PNP цифров транзистор –принцип на работа

по погрешка активирате всички сегменти на трите индикатори едновременно, общият ток да е под 120mA.

Сегментите а ÷ g (dp) на дисплея са свързани към порт С, а

катодите към изводите PB3, PB4 и PB5 на микроконтролера. Това означава, че трябва да конфигурираме тези изводи като изходи. Преди това обаче е препоръчително да конфигурираме съответните битове за данни с подходящи стойности, така че след като конфигурираме изводите като изходи сегментите на дисплея да бъдат изгасени, т.е.. PCOUT = 0x00, PBOUT<3> = 0, PBOUT<4> = 0 и PBOUT<5> = 0. Добре е да отделим този код в отделна функция, например PortConfig().

```
void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD  = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD  &= ~((1 << 3) | (1 << 4) | (1 << 5));
}
```

Логиката на основната програма се състои от следното:

1. Изобразяваме поредната цифра на индикатор1;
2. Изчакваме 1 секунда;
3. Увеличаваме цифрата с 1;
4. Ако резултатът е 10, го нулираме;
5. Преминаваме към стъпка 1.

Бихме могли да превърнем горните стъпки в код по следния начин:

```
void main(void)
{
    uint8 digit = 0; /* Съдържа изобразяваната цифра */

    PortConfig();

    while (1)
    {
```

```

        SendToIndicator1(digit);
        Delay(DELAY_CONSTANT_1_SEC);
        digit++;
        if (10 == digit)
        {
            digit = 0;
        }
    }
}

```

Локалната променлива `digit` съдържа текущата цифра, която ще се изобразява на индикатор1. Функцията `SendToIndicator1()` изпраща тази цифра към индикатор1, след което изчакваме 1 секунда, увеличаваме `digit` с 1, ако стойността е 10, нулираме `digit`, в противен случай директно подаваме поредната цифра към индикатор1 и цикълът се повтаря отново.

Това което остана да добавим е кодът на функцията `SendToIndicator1()`. Следващата таблица показва цифрите от 0 до 9 и съответните им стойности, които трябва да изведем на изходите на порт C, за ги изобразим на индикатора.

цифра	PC<7>	PC<6>	PC<5>	PC<4>	PC<3>	PC<2>	PC<1>	PC<0>	PCOUT
	dp	g	f	e	d	c	b	a	
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F

Както изискват добрите програмни практики, ще представим всички тези стойности чрез макроси. Например:

```

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)

```

```
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
```

Това, което трябва да направим е в тялото на `SendToIndicator1()` да запишем една от тези стойности в регистъра `PCOUT` в зависимост стойността на променливата `digit`, т.е.. ако `digit` е 0, то:

```
PCOUT = LED_DIGIT_0;
```

Тази задача е идеален кандидат за нова функция, която да преобразува цифрите $0 \div 9$ в съответните им 7-сегментни индикаторни стойности. Можем да наречем тази функция например `ConvertTo7SegIndValue()`. Ето едно възможно решение:

```
#define LED_DIGIT_ERR      (0x79)

static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 retValue = LED_DIGIT_ERR;

    switch (digit)
    {
        case 0:
            retValue = LED_DIGIT_0;
            break;

        case 1:
            retValue = LED_DIGIT_1;
            break;

        case 2:
            retValue = LED_DIGIT_2;
            break;

        case 3:
            retValue = LED_DIGIT_3;
            break;

        case 4:
            retValue = LED_DIGIT_4;
            break;
    }
}
```

```

    case 5:
        retValue = LED_DIGIT_5;
        break;

    case 6:
        retValue = LED_DIGIT_6;
        break;

    case 7:
        retValue = LED_DIGIT_7;
        break;

    case 8:
        retValue = LED_DIGIT_8;
        break;

    case 9:
        retValue = LED_DIGIT_9;
        break;
}

return retValue;
}

```

Резултатът от тази функция може директно да се присвои на регистъра PCOUT.

```
PCOUT = ConvertTo7SegIndValue(digit);
```

Забележка: Дефинирал съм функцията ConvertTo7SegIndValue() като статична, тъй като нейната роля е да се използва само във файла, в който е дефинирана и никъде другаде. В случая това е без значение, защото нашата програма така или иначе се състои само от един сорс-файл (main.c), така че дефинирането на функциите с или без ключовата дума static, не променя поведението на програмата. Добре е обаче когато създавате по-големи и по-сложни програми да ограничавате областта на видимост на функции, които не трябва да се използват извън файла, в който са дефинирани.

Обърнете внимание, че ако по погрешка извикате функцията с аргумент по-голям от 9, върнатият резултат (стойността на макроса LED_DIGIT_ERR) ще доведе до изписване на буквата E (от Error) на индикатора. Вместо да

инициализирате локалната променлива `retValue` с код за грешка, може да добавите `default`-етикет и в него да присвоявате кода за грешка на `retValue`.

```
static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 retValue;

    switch (value)
    {
        case 0:
            retValue = LED_DIGIT_0;
            break;

        case 1:
            retValue = LED_DIGIT_1;
            break;

        case 2:
            retValue = LED_DIGIT_2;
            break;

        case 3:
            retValue = LED_DIGIT_3;
            break;

        case 4:
            retValue = LED_DIGIT_4;
            break;

        case 5:
            retValue = LED_DIGIT_5;
            break;

        case 6:
            retValue = LED_DIGIT_6;
            break;

        case 7:
            retValue = LED_DIGIT_7;
            break;

        case 8:
            retValue = LED_DIGIT_8;
            break;

        case 9:
            retValue = LED_DIGIT_9;
            break;
    }
}
```

```

default:
    retValue = LED_DIGIT_ERR;
    break;
}

return retValue;
}

```

За да се появи цифрата на индикатора обаче, е нужно да свържем катода K1 към маса. За целта трябва да включим цифровия ключ Q1. Вече знаете как да направите това. Просто изведете лог.1 на извод PB3.

```
PBOUT |= 1 << 3; /* Включване на Q1 */
```

За да сме сигурни обаче че другите два индикатора са изключени, добре е да добавим код, който да изключва съответните катода преди това.

```
PBOUT &= ~((1 << 4) | (1 << 5)); /* Изключване на Q2 и Q3 */
```

Въпреки че не е задължително, когато се използва само един индикатор, по-добре е преди да заредите новата стойност за изобразяване да го изключите т.е.:

```
PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5)); /* Изключване на Q1, Q2 и Q3 */
```

Функцията SendToIndicator1() придобива следния вид:

```

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

```

Като начинаещи C-програмисти, много е вероятно да напишете функцията ConvertTo7SegIndValue() по следния начин:

```
uint8 ConvertTo7SegIndValue(uint8 value)
```

```

{
    switch (digit)
    {
        case 0:
            return LED_DIGIT_0;

        case 1:
            return LED_DIGIT_1;

        case 2:
            return LED_DIGIT_2;

        case 3:
            return LED_DIGIT_3;

        case 4:
            return LED_DIGIT_4;

        case 5:
            return LED_DIGIT_5;

        case 6:
            return LED_DIGIT_6;

        case 7:
            return LED_DIGIT_7;

        case 8:
            return LED_DIGIT_8;

        case 9:
            return LED_DIGIT_9;

        default:
            return LED_DIGIT_ERR;
    }
}

```

Подобно използване на повече от един return във функция трябва да се избягва, т.е.. избягвайте да създавате функции с много изходи. Най-добре е функцията да бъде построена така, че да има един изход (един оператор return), който се намира в края на функцията преди затварящата скоба, както е направено в първата версия на функцията ConvertTo7SegIndValue(). Сега може и да не виждате проблема, който може да възникне, но в по-големи и сложни функции е много вероятно да пропуснете да добавите някъде

оператора return или да добавите return без израз. В последния случай върнатият резултат ще бъде недефиниран.

Функцията ConvertTo7SegIndValue() може да бъде написана и по следния компактен начин:

```
static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
}
```

Ако параметърът value е по-голям от 9, функцията връща код за грешка LED_DIGIT_ERR, в противен случай връща arr7SegIndValues[value].

Ето и целия код на програмата.

main.c

```
#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC    (505250)
#define DELAY_CONSTANT_500_MSEC (252625)

#define LED_DIGIT_0    (0x3F)
#define LED_DIGIT_1    (0x06)
#define LED_DIGIT_2    (0x5B)
#define LED_DIGIT_3    (0x4F)
#define LED_DIGIT_4    (0x66)
```

```

#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP     (0x80)
#define LED_DIGIT_ERR     (0x79)

extern void PortConfig(void);
extern void SendToIndicator1(uint8 digit);
extern void Delay(uint32 delay);
static uint8 ConvertTo7SegIndValue(uint8 value);

void main(void)
{
    uint8 digit = 0;

    PortConfig();

    while (1)
    {
        SendToIndicator1(digit);
        Delay(DELAY_CONSTANT_1_SEC);
        digit++;
        if (10 == digit)
        {
            digit = 0;
        }
    }
}

void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,

```

```

        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
}

void Delay(uint32 delay)
{
    for (;delay > 0; delay--);
}

```

Следващият пример ще бъде малко по-сложен. Ще направим десетичен кръгов брояч от 0 до 999, т.е.. ще използваме и другите два индикатори на дисплея. Точно в начина на изобразяване на повече от една цифра се крие тънкостта на управлението на 7-сегментния дисплей. Погледнете отново Фиг.107. Очевидно е, че ако активираме трите цифрови ключа едновременно, трите индикатори ще изобразят една и съща цифра. По тази причина, изобразяването на повече от една цифра се състои в последователно изобразяване на отделните цифри. Докато единият индикатор е активен, другите два трябва да са изключени (посредством катодите си). Времето, през което информацията на всеки индикатор се опреснява, трябва да е под 20ms (над 50Hz). Спомнете си примера със светодиода от предходната точка. Само при това условие, изображението на дисплея ще изглежда като постоянноосветещо, без трепкания.

Като начало, това което трябва да добавим са две функции, аналогични на SendToIndicator1().

```

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
}

```

```

        PBOUT |= 1 << 3;
    }

    void SendToIndicator2(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 4;
    }

    void SendToIndicator3(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 5;
    }

```

Всяка една от тези функции активира своя индикатор и изключва другите два, т.е.. ако е активиран например индикатор1, индикатор2 и индикатор3 са изключени. Следващата стъпка е да извикваме функциите циклично на интервали, не по-големи от 20ms. Например:

```

while (1)
{
    SendToIndicator1(digit1);
    SendToIndicator2(digit2);
    SendToIndicator3(digit3);
}

```

Нека да видим как ще изглежда програмата дотук в целия й вид и да изпишем числото 123 на дисплея:

main.c

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)

```

```

#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR      (0x79)

extern void PortConfig(void);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 value);

void main(void)
{
    PortConfig();

    while (1)
    {
        SendToIndicator1(3);
        SendToIndicator2(2);
        SendToIndicator3(1);
    }
}

void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));

```



```

        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 5;
    }

    static uint8 ConvertTo7SegIndValue(uint8 value)
    {
        uint8 arr7SegIndValues[] =
        {
            LED_DIGIT_0,
            LED_DIGIT_1,
            LED_DIGIT_2,
            LED_DIGIT_3,
            LED_DIGIT_4,
            LED_DIGIT_5,
            LED_DIGIT_6,
            LED_DIGIT_7,
            LED_DIGIT_8,
            LED_DIGIT_9
        };

        return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
    }

```

Ще забележите, че яркостта на светене на индикаторите е ниска. Причината, е че всеки индикатор се активира за твърде кратко време. Можем да отстраним това, като добавим закъснение примерно 5ms след всяко извикване на тези функции, т.е..

```

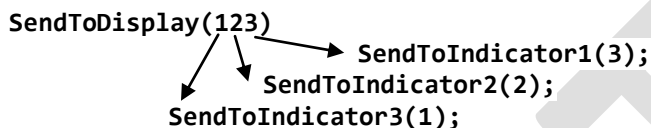
while (1)
{
    SendToIndicator1(3);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2(2);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3(1);
    Delay(DELAY_CONSTANT_5_MSEC);
}

```

По този начин всеки индикатор свети 5ms и се опреснява на всеки 15ms и проблемът с яркостта е отстранен.

Нашата задача изисква обаче да изобразяваме стойността на трицифрено число. Задачата, която трябва да решим, е да разделим изобразяваното число на съставните му цифри.

Например, за да изобразим числото 123, трябва да го разделим на съставните му цифри 1, 2 и 3, и да подадем всяка една от тях на съответната функция, както е показано по-горе. Най-добре е да си дефинираме функция, която приема числото като аргумент и тя да се грижи за отпечатване отделните цифри на дисплея. Например:



За да разделим едно десетично число на съставните му цифри, можем да използваме операцията деление по модул (%). Тази операция има следното свойство:

$$a \% b = a - (a / b) * b$$

Тоест операцията % е пряко свързана с операцията деление (/). Ако **a** е нашето число, а **b** е 10, то

$$123 \% 10 = 123 - (123/10) * 10 = 123 - 12 * 10 = 123 - 120 = 3$$

Не е необходимо да разбирате „как и защо“, просто запомнете, че ако разделите по модул едно десетично число на 10, резултатът е най-дясната цифра на числото. След като знаем това можем да разделим нашето число на съставните му цифри по следния начин:

$$123 \% 10 = 3$$

$$(123 / 10) \% 10 = 12 \% 10 = 2$$

$$(123 / 100) \% 10 = 1 \% 10 = 1$$

Нека сега да видим как ще изглежда нашата функция.

```
void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
```

```

        SendToIndicator2((value / 10) % 10);
        Delay(DELAY_CONSTANT_5_MSEC);
        SendToIndicator3((value / 100) % 10);
        Delay(DELAY_CONSTANT_5_MSEC);
    }

```

Основната програма придобива следния вид:

```

void main(void)
{
    PortConfig();

    while (1)
    {
        SendToDisplay(123);
    }
}

```

Добавете промените, които направихме, и стартирайте програмата. Ще видите, че резултатът е същия като предходния.

И последното, което остана да направим е да увеличаваме стойността на изобразяваното число през 1 секунда. Очевидно е, че трябва да използваме променлива. За целта ще се възползваме отново от Таймер 0 (може и Таймер 1). Ще конфигурираме таймера в „непрекъснат режим”. Вече сме конфигурирали таймера да се препълва за 5ms и можем да се възползваме от това. Предполагам, че вече се досещата какво трябва да се направи. На всеки 200 прекъсвания ($200 * 5ms = 1000ms$) да увеличаваме нашата променлива с 1. Не трябва да забравяме да нулираме променливата, ако след увеличението стойността ѝ е 1000, нали броячът ни е от 0 до 999. Ето и кодът, който изпълнява гореописаните действия.

```

void InterruptConfig(void)
{
    /* Разрешаване на прекъсване от Таймер 0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;
}

```

```

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /*-----*/
    /* - Непрекъснат режим */
    /* - Прекъсване при препълване */
    /* - Коефициент на деление 2 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x89;
}

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 timerInterruptCounter = 0;

    timerInterruptCounter++;

    if (200 == timerInterruptCounter)
    {
        timerInterruptCounter = 0;
        number++;
        if (1000 == number)
        {
            number = 0;
        }
    }
}

```

Сега вече може да сглобим цялата програма.

main.c

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC    (505250)
#define DELAY_CONSTANT_500_MSEC (252625)
#define DELAY_CONSTANT_5_MSEC  (2526)

#define LED_DIGIT_0            (0x3F)

```

```

#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR      (0x79)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
extern void Delay(uint32 delay);
static uint8 ConvertTo7SegIndValue(uint8 value);

uint16 number = 0;

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 timerInterruptCounter = 0;

    timerInterruptCounter++;

    if (200 == timerInterruptCounter)
    {
        timerInterruptCounter = 0;
        number++;
        if (1000 == number)
        {
            number = 0;
        }
    }
}

void main(void)
{
    uint16 val = 0;

    PortConfig();
    InterruptConfig();
    TimerConfig();

    while (1)
    {

```

```

        _di(); /* Забраняване на прекъсванията */
        val = number;
        _ei(); /* Разрешаване на прекъсванията */

        SendToDisplay(val);
    }
}

void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсване от Таймер 0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL |= 1 << 7;
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /*-----*/
    /* - Непрекъснат режим */
    /* - Прекъсване при препълване */
    /* - Коефициент на деление 2 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x89;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
}

```

```

        Delay(DelayConstant_5_Msec);
    }

    void SendToIndicator1(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 3;
    }

    void SendToIndicator2(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 4;
    }

    void SendToIndicator3(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 5;
    }

    static uint8 ConvertTo7SegIndValue(uint8 value)
    {
        uint8 arr7SegIndValues[] =
        {
            LED_DIGIT_0,
            LED_DIGIT_1,
            LED_DIGIT_2,
            LED_DIGIT_3,
            LED_DIGIT_4,
            LED_DIGIT_5,
            LED_DIGIT_6,
            LED_DIGIT_7,
            LED_DIGIT_8,
            LED_DIGIT_9
        };

        return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
    }

    void Delay(uint32 delay)
    {
        for (; delay > 0; delay--);
    }

```

Всичко в тази програма вече Ви е познато, с изключение на

тези редове:

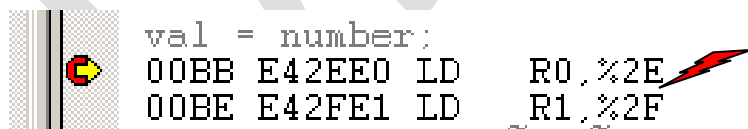
```
_di(); /* Забраняване на прекъсванията */  
val = number;  
_ei(); /* Разрешаване на прекъсванията */
```

Функциите `_di()` и `_ei()` са предоставени от компилатора. Функцията `_di()` забранява прекъсванията, като нулира бит `IRQCTL<IRQE>`, а `_ei()` разрешава прекъсванията, като установява бит `IRQCTL<IRQE>`. Вместо тези функции може да използвате вече познатите начини за нулиране и установяване на битове:

```
IRQCTL &= ~(1 << 7); /* Нулиране на бит IRQE */  
IRQCTL |= (1 << 7); /* Установяване на бит IRQE */
```

Както забелязвате, променливата `number` не се подава директно на функцията `SendToDisplay()`, а се присвоява на локалната променлива `val`, която се използва за аргумент на `SendToDisplay()`. Причината за това се крие във факта, че `number` също е достъпна и в `Timer0_ISR()`, освен това е 16-битова променлива. Нашият микроконтролер е 8-битов и съответно са необходими две операции, за да прочете стойността на такава променлива. Вижте как изглежда асемблерния код, генериран от компилатора за реда

```
val = number;
```



```
val = number;  
00BB E42EE0 LD R0,%2E  
00BE E42FE1 LD R1,%2F
```

Необходими са две инструкции за копиране на `number` във `val`. Какъв ще бъде резултатът обаче, ако възникне прекъсване от таймера преди втората инструкция да бъде изпълнена? Изпълнението на програмата временно ще се прехвърли в тялото на функцията `Timer0_ISR()`. Променливата `number` ще се увеличи с 1, след което

програмата ще се върне към изпълнението на втората инструкция, но съдържанието на `number` вече ще е променено. Същият ефект може да се получи ако извикаме `SendToDisplay()` директно с `number` за аргумент. Ще са нужни две операции за четене, за да се копира стойността на `number` в параметъра `value`. За да се избегне този проблем, текущата стойност на `number` се копира в друга променлива, като прекъсванията преди това се забраняват, а след копирането отново се разрешават.

Използването на софтуерна задръжка между превключването на отделните индикатори е лесна за реализация и върши работа в нашия малък пример, но този подход може да създаде проблеми в повечето реални приложения. В следващата точка ще научите малко повече за това. Вместо да използваме софтуерна задръжка можем да използваме хардуерна. Как може да стане това? Чрез използване на таймер, който да се препълва примерно на 5ms и във функцията за обработка на прекъсването да се извършва превключването между индикаторите на дисплея.

Ето как изглежда програмата:

```
#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
```

```

#define LED_DIGIT_ERR      (0x79)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
extern void Delay(uint32 delay);
static uint8 ConvertTo7SegIndValue(uint8 value);

uint16 number = 0;

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 displayCounter = 0;

    displayCounter++;

    switch (displayCounter)
    {
        case 1:
            SendToIndicator1(number % 10);
            break;

        case 2:
            SendToIndicator2((number / 10) % 10);
            break;

        case 3:
            SendToIndicator3((number / 100) % 10);
            displayCounter = 0;
            break;
    }
}

void main(void)
{
    PortConfig();
    InterruptConfig();
    TimerConfig();

    while (1)
    {
        Delay(LED_DIGIT_ERR);

        _di(); /* Забраняване на прекъсванията */
        ++number == 1000 ? number = 0 : number;
        _ei(); /* Разрешаване на прекъсванията */
    }
}

```

```

void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсване от Таймер 0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL |= 1 << 7;
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;
    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /*-----*/
    /* - Непрекъснат режим */
    /* - Прекъсване при препълване */
    /* - Коефициент на деление 2 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x89;
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{

```

```

        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 5;
    }

    static uint8 ConvertTo7SegIndValue(uint8 value)
    {
        uint8 arr7SegIndValues[] =
        {
            LED_DIGIT_0,
            LED_DIGIT_1,
            LED_DIGIT_2,
            LED_DIGIT_3,
            LED_DIGIT_4,
            LED_DIGIT_5,
            LED_DIGIT_6,
            LED_DIGIT_7,
            LED_DIGIT_8,
            LED_DIGIT_9
        };

        return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
    }

    void Delay(uint32 delay)
    {
        for (; delay > 0; delay--);
    }

```

Тази програма е реализирана противоположно на предходната. Променливата `number` се увеличава с 1 всяка секунда, но в основната програма, а на всеки 5ms функцията `Timer0_ISR()` обновява поредния индикатор на дисплея.

Можем да махнем едносекундната софтуерна задръжка и да променяме променливата `number` във функцията `Timer0_ISR()` както е показано в следващия пример.

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_5_MSEC              (2526)

#define LED_DIGIT_0               (0x3F)
#define LED_DIGIT_1               (0x06)

```

```

#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR      (0x79)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 value);

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint16 number      = 0;
    static uint8  numberCounter = 0;
    static uint8  displayCounter = 0;

    displayCounter++;

    if (200 == ++numberCounter)
    {
        numberCounter = 0;

        if (1000 == ++number)
        {
            number = 0;
        }
    }

    switch (displayCounter)
    {
    case 1:
        SendToIndicator1(number % 10);
        break;

    case 2:
        SendToIndicator2((number / 10) % 10);
        break;

    case 3:
        SendToIndicator3((number / 100) % 10);
        displayCounter = 0;
        break;
    }
}

```

```

    }
}

void main(void)
{
    PortConfig();
    InterruptConfig();
    TimerConfig();

    while (1);
}

void PortConfig(void)
{
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсване от Таймер 0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL |= 1 << 7;
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /*-----*/
    /* - Непрекъснат режим */
    /* - Прекъсване при препълване */
    /* - Коефициент на деление 2 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x89;
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));

```

```

        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 3;
    }

    void SendToIndicator2(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 4;
    }

    void SendToIndicator3(uint8 digit)
    {
        PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
        PCOUT = ConvertTo7SegIndValue(digit);
        PBOUT |= 1 << 5;
    }

    static uint8 ConvertTo7SegIndValue(uint8 value)
    {
        uint8 arr7SegIndValues[] =
        {
            LED_DIGIT_0,
            LED_DIGIT_1,
            LED_DIGIT_2,
            LED_DIGIT_3,
            LED_DIGIT_4,
            LED_DIGIT_5,
            LED_DIGIT_6,
            LED_DIGIT_7,
            LED_DIGIT_8,
            LED_DIGIT_9
        };

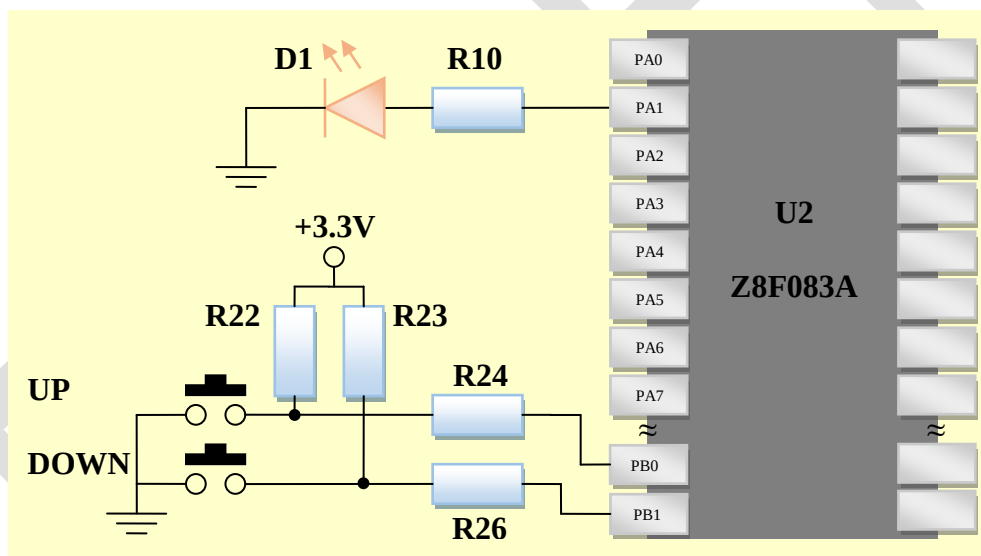
        return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
    }

```

Променливата `number` се използва само в `Timer0_ISR()`. Това позволява да я дефинираме като статична в тялото на тази функция. Забележете също, че работата по увеличаване на `number` и изобразяването на стойността ѝ на дисплея се извършва изцяло в прекъсването на таймера. Това „развързва ръцете“ на основната програма, позволявайки ѝ да извършва други дейности.

46 Четене на бутон.

В предходните примери само извеждахме информация на изводите на микроконтролера. В тази точка ще се запознаем и с въвеждането ѝ, като четем информацията, подавана чрез бутоните UP и DOWN. Задачата, която ще решим в това упражнение, е при натискане на бутона UP да светва светодиодът D1, а при натискане на бутона DOWN светодиодът да изгасва. Уверете, че джъмперът JM2 от разширителната платка Z8Encore!_EB01(MB01) е в позиция PWM. Фиг.111 представлява опростен изглед на част от принципна електрическа схема, която ще ни служи за ориентир.



Фиг. 111 Четене на бутон – опростена схема

Когато бутоните са отпуснати, изтеглящите резистори R22 и R23 поддържат високо ниво на изводите PB0 и PB1. Съответно при натискане на някой от бутоните нивото на извода става ниско. За да определим дали някой от бутоните е натиснат, трябва да прочетем нивото на съответните изводи. Необходимо е да изпълним следните основни стъпки:

1. Конфигурираме извод PA1 като изход.
2. Конфигурираме изводите PB0 и PB1 като входове.
3. Прочитаме нивото на извод PB0 .

3.1 Ако нивото е ниско, включваме светодиода D1.

4. Прочитаме нивото на извод PB1.

4.1 Ако нивото е ниско, изключваме светодиода D1.

5. Преминаваме към стъпка 3.

Първата стъпка вече сме я правили, затова преминаваме директно към втората. За да конфигурираме изводите PB0 и PB1 като входове, можем да използваме следния код:

```
PBDD |= ((1 << 0) | (1 << 1));
```

За да прочетем нивото на даден извод, трябва да прочетем целия входен регистър за данни и да отделим само този бит, който искаме да проверим, т.е. . трябва да използваме вече познатия ни метод с операцията битово И (&) и маска. Например, за да прочетем нивото на PB0 можем да използваме следния код:

```
PBIN & (1 << 0)
```

Ако бит PBIN<0> е 0 (бутонът е натиснат), то резултатът от цялата операция е 0, в противен случай резултатът е различен от 0. Можем да се възползваме от това по различни начини. Например можем да използваме горната конструкция в условието на if-оператор:

```
/* Бутон UP натиснат ? */
if (!(PBIN & (1 << 0))) ~ /* Бутон UP натиснат ? */
{ /* Да */ } if (0 == (PBIN & (1 << 0)))
{ /* Да */ }
```

<pre> TurnOnLed(); } /* Бутон DOWN натиснат ? */ if (!(PBIN & (1 << 1))) ~ { /* Да */ TurnOffLed(); } </pre>	<pre> TurnOnLed(); } /* Бутон DOWN натиснат ? */ if (0 == (PBIN & (1 << 1))) { /* Да */ TurnOffLed(); } </pre>
---	---

Обърнете внимание, че за да постигнем желаното поведение (да включваме светодиода, когато бутон UP е натиснат и да го изключим когато бутон DOWN е натиснат) е необходимо да инвертираме резултата с операцията лог. отрицание (!).

!(израз)

Ако (израз) е различен от 0, резултатът е 0, в противен случай е 1.

Можем да използваме горната конструкция и в условието на операцията ?: и да съхраняваме резултата в променлива, която след това да проверяваме в if-оператор:

```

typedef unsigned char boolean;

boolean buttonUp;
boolean buttonDown;

buttonUp = (PBIN & (1 << 0)) ? 1 : 0;

if (0 == buttonUp)
{
    TurnOnLed();
}

buttonDown = (PBIN & (1 << 1)) ? 1 : 0;

if (0 == buttonDown)
{
    TurnOffLed();
}

```

Ако бутонът е отпуснат, изразът (PBIN & (1 << 0)) ще бъде различен от 0 и на променливата buttonUp ще се присвои 1, в

противен случай 0. Същото важи и за buttonDown. Можем да подобрим четливостта на програмата, като си дефинираме подходящи имена за тези стойности, например:

```
#define BUTTON_PRESSED    (0)
#define BUTTON_RELEASED   (1)

typedef unsigned char boolean;

boolean buttonUp;
boolean buttonDown;

buttonUp = (PBIN & (1 << 0)) ? BUTTON_RELEASED : BUTTON_PRESSED;

if (BUTTON_PRESSED == buttonUp)
{
    TurnOnLed();
}

buttonDown = (PBIN & (1 << 1)) ? BUTTON_RELEASED : BUTTON_PRESSED;

if (BUTTON_PRESSED == buttonDown)
{
    TurnOffLed();
}
```

И накрая можем да дефинираме функции, които да връщат състоянието на тези изводи:

```
#define BUTTON_PRESSED    (0)
#define BUTTON_RELEASED   (1)

typedef unsigned char boolean;

boolean ReadButtonUp(void)
{
    return (PBIN & (1 << 0)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

boolean ReadButtonDown(void)
{
    return (PBIN & (1 << 1)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

if (BUTTON_PRESSED == ReadButtonUp())
{
    TurnOnLed();
}
```

```

}

if (BUTTON_PRESSED == ReadButtonDown())
{
    TurnOffLed();
}

```

Използвайки последния вариант, цялата програма придобива следния вид:

main.c

```

#include <ez8.h>
#include "stdtypes.h"

#define BUTTON_PRESSED    (0)
#define BUTTON_RELEASED   (1)

extern void PortConfig(void);
extern void TurnOnLed(void);
extern void TurnOffLed(void);
extern boolean ReadButtonUp(void);
extern boolean ReadButtonDown(void);

void main(void)
{
    PortConfig();

    while (1)
    {
        if (BUTTON_PRESSED == ReadButtonUp())
        {
            TurnOnLed();
        }

        if (BUTTON_PRESSED == ReadButtonDown())
        {
            TurnOffLed();
        }
    }
}

void PortConfig(void)
{
    PADD &= ~(1 << 1);
    PBDD |= ((1 << 0) | (1 << 1));
}

void TurnOnLed(void)

```

```

{
    PAOUT |= 1 << 1;
}

void TurnOffLed(void)
{
    PAOUT &= ~(1 << 1);
}

boolean ReadButtonUp(void)
{
    return (PBIN & (1 << 0)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

boolean ReadButtonDown(void)
{
    return (PBIN & (1 << 1)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

```

Натиснете бутона UP и го задръжете, след което натискайте и отпускате бутона DOWN. Ще забележите, че докато бутонът DOWN е натиснат яркостта на светене на светодиода намалява, тъй като светодиода се включва за много кратко време. Причината е в логиката на програмата, която проверява само, че даден бутон е натиснат, но не и дали е отпуснат. Можем да коригираме това, като добавим код, който да игнорира даден бутон, докато другият е вече натиснат. За целта можем да използваме цикъл, като този

```
while (BUTTON_PRESSED == ReadButtonUp());
```

Основната програма придобива вида:

```

void main(void)
{
    PortConfig();

    while (1)
    {
        if (BUTTON_PRESSED == ReadButtonUp())
        {
            TurnOnLed();

            while (BUTTON_PRESSED == ReadButtonUp());
        }
    }
}

```

```

        if (BUTTON_PRESSED == ReadButtonDown())
        {
            TurnOffLed();

            while (BUTTON_PRESSED == ReadButtonDown());
        }
    }
}

```

Сега ако натиснете първо бутон UP и го задържите, ще видите, че и да натискате бутона DOWN, това няма да има никакъв ефект. Обратното също е вярно.

Нека сега да променим малко програмата, така че при натискане на бутон UP например светодиодът да се превключва, т.е.. ако е бил включен, да се изключва, и обратно. Можем да си дефинираме функция ToggleLed(), която да извършва това.

```

void ToggleLed(void)
{
    PAOUT ^= (1 << 1);
}

```

Основната програма придобива вида:

```

void main(void)
{
    PortConfig();

    while (1)
    {
        if (BUTTON_PRESSED == ReadButtonUp())
        {
            ToggleLed();

            while (BUTTON_PRESSED == ReadButtonUp());
        }
    }
}

```

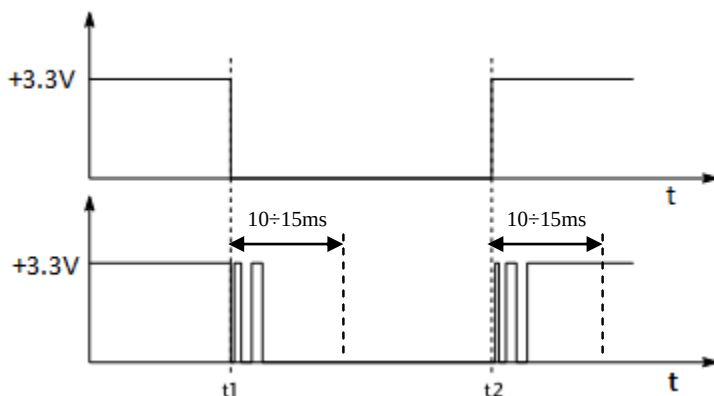
Добавете промените, компилирайте кода, стартирайте програмата и започнете да натискате бутона UP с различна скорост и сила. Ще забележите, че поведението на

светодиода не е точно това, което се очаква, въпреки че логиката на програмата изглежда вярна. И наистина проблемът не е в програмата, а в самия бутон. При натискането му вътрешната пластина на бутона започва да трепти, при което контактът с изводите на бутона ту се затваря, ту се отваря.



Този процес отнема известно време, обикновено няколко милисекунди, докато пластината се „успокои” и контактът с изводите на бутона стане стабилен. Същият ефект съществува и при отпускане на бутона. По тази причина нивото на извода е непостоянно за определено време (Фиг.113).

Продължителността на трептенията е различна за различните бутони, но зависи най-вече силата и скоростта на натискане. Ако не се вземат специални мерки, микроконтролерите ще детектират тези трептения и едно натискане/отпускане на бутона се възприема като многократно натискане/отпускане.



Фиг. 113 Трептения при натискане/отпускане на бутон

Като цяло повечето механични бутони и ключове (ако не и всички) страдат от този проблем. Той може да се избегне или чрез добавянето на външни „гасящи“ схеми или софтуерно. Най-лесният начин да се справим софтуерно с този проблем, е чрез добавяне на времезадръжка (10-15ms) след първото детектиране, че бутонът е натиснат. След изтичане на времезадръжката трябва отново да изчетем нивото на извода и ако пак детектираме, че нивото е ниско, можем с увереност да допуснем, че бутонът е натиснат, в противен случай приемаме, че бутонът е отпуснат.

Нека да добавим тези промени и да видим как ще изглежда цялата програма:

```
#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define BUTTON_PRESSED      (0)
#define BUTTON_RELEASED     (1)

extern void PortConfig(void);
extern void ToggleLed(void);
extern boolean ReadButtonUp(void);
extern void Delay(uint32 delay);
```



```

void main(void)
{
    PortConfig();

    while (1)
    {
        /* Бутон UP натиснат ли е ?*/
        if (BUTTON_PRESSED == ReadButtonUp())
        { /* Да */
            Delay(DELAY_CONSTANT_10_MSEC); /* Изчакаване 10ms */

            /* Бутон UP натиснат ли е ?*/
            if (BUTTON_PRESSED == ReadButtonUp())
            { /* Да */
                ToggleLed(); /* Превключване на светодиода */

                /* Стой тук докато бутонът е натиснат */
                while (BUTTON_PRESSED == ReadButtonUp());
            }
        }
    } /* Край на while(1)*/
} /* Край на main */

void PortConfig(void)
{
    PADD &= ~(1 << 1);
    PBDD |= ((1 << 0) | (1 << 1));
}

boolean ReadButtonUp(void)
{
    return (PBIN & (1 << 0)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

void ToggleLed(void)
{
    PAOUT ^= (1 << 1);
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

```

Нека сега да разнообразим нещата, като реализираме кръгов десетичен брояч от 0 до 999 и да изобразяваме резултата на седемсегментния дисплей (Фиг.114). При натискане на бутон UP броячът ще се увеличава, а при натискане на бутон

DOWN ще се намалява. Това, което трябва да спазваме е да извикваме функцията `SendToDisplay()` достатъчно често (на интервали не по-големи от 5ms). Тази функция отнема около 15ms за изпълнение. Можем да се възползваме от това и да я извикваме вместо `Delay(DELAY_CONSTANT_10_MSEC)`. За да не изгасне дисплея, докато сме натиснали някой от бутоните, можем да използваме следния код

```
do
{
    SendToDisplay(counter);
} while (BUTTON_PRESSED == ReadButtonDown());
```

ВМЕСТО

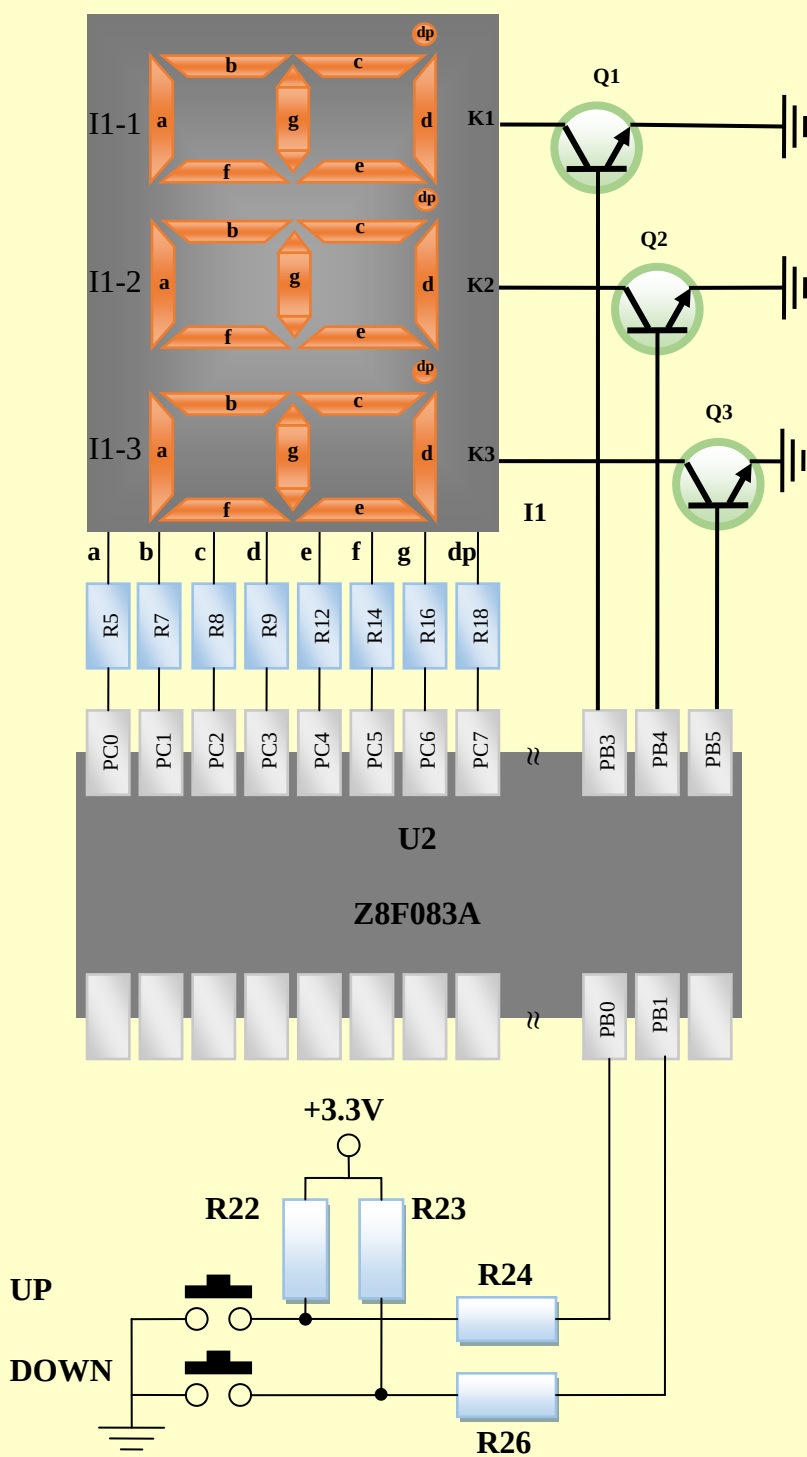
```
while (BUTTON_PRESSED == ReadButtonDown());
```

И последната особеност, която трябва да имате предвид е следната проверка на броячната променлива:

```
counter--;  
if (65535 == counter) counter = 999;
```

Тази проверка се възползва от особеността при отрицателно препълване на беззнаковите цели числа в C, а именно, че резултатът е най-голямото число, което може да се представи от типа на числото. В нашия случай `counter` е тип `uint16` с диапазон от стойности $0 \div 65535$. Ако `counter` е 0, то операцията `counter--` ще даде като резултат 65535. Тъй като нашата задача се състои в реализирането на кръгов десетичен брояч $0 \div 999$, то при възникването на такава ситуация е необходимо да заредим `counter` с 999.

Ето и цялата програма:



Фиг. 114 Кръгов десетичен брояч – опростена схема

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define BUTTON_PRESSED           (0)
#define BUTTON_RELEASED          (1)

#define LED_DIGIT_0              (0x3F)
#define LED_DIGIT_1              (0x06)
#define LED_DIGIT_2              (0x5B)
#define LED_DIGIT_3              (0x4F)
#define LED_DIGIT_4              (0x66)
#define LED_DIGIT_5              (0x6D)
#define LED_DIGIT_6              (0x7D)
#define LED_DIGIT_7              (0x07)
#define LED_DIGIT_8              (0x7F)
#define LED_DIGIT_9              (0x6F)
#define LED_DIGIT_DP             (0x80)
#define LED_DIGIT_ERR            (0x79)

extern void PortConfig(void);
extern void TurnOnLed(void);
extern void TurnOffLed(void);
extern boolean ReadButtonUp(void);
extern boolean ReadButtonDown(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 value);
extern void Delay(uint32 delay);

void main(void)
{
    uint16 counter = 0;

    PortConfig();

    while (1)
    {
        SendToDisplay(counter);

        if (BUTTON_PRESSED == ReadButtonUp())
        {
            SendToDisplay(counter);

            if (BUTTON_PRESSED == ReadButtonUp())

```

```

        {
            counter++;

            if (1000 == counter) counter = 0;

            do
            {
                SendToDisplay(counter);
            } while (BUTTON_PRESSED == ReadButtonUp());
        }

    }

    if (BUTTON_PRESSED == ReadButtonDown())
    {
        SendToDisplay(counter);

        if (BUTTON_PRESSED == ReadButtonDown())
        {
            counter--;

            if (65535 == counter) counter = 999;

            do
            {
                SendToDisplay(counter);
            } while (BUTTON_PRESSED == ReadButtonDown());
        }
    }
} /* Край на while(1) */
} /* Край на main */

void PortConfig(void)
{
    PADD  &= ~(1 << 1);
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD  &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD  |= ((1 << 0) | (1 << 1));
    PCOUT = 0x00;
    PCDD  = 0x00;
}

void TurnOnLed(void)
{
    PAOUT |= 1 << 1;
}

void TurnOffLed(void)
{
    PAOUT &= ~(1 << 1);
}

```

```

boolean ReadButtonUp(void)
{
    return (PBIN & (1 << 0)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

boolean ReadButtonDown(void)
{
    return (PBIN & (1 << 1)) ? BUTTON_RELEASED : BUTTON_PRESSED;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
    }
}

```

```

        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

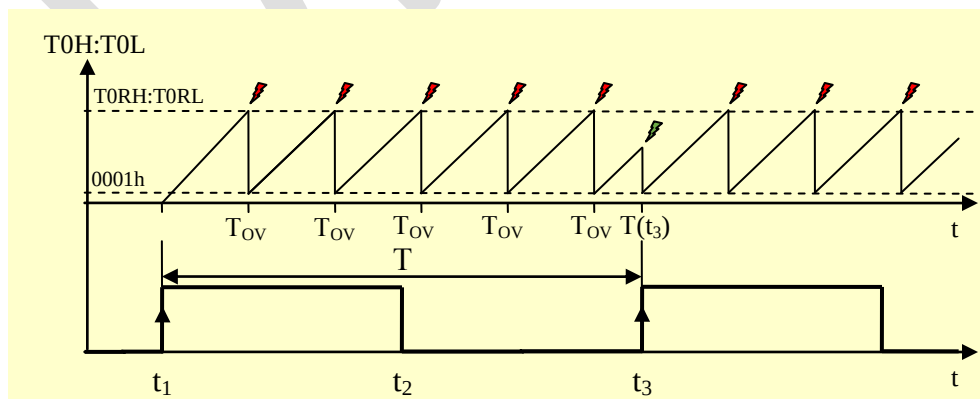
```

Забелязахте ли как използването на софтуерна задръжка между опресняването на индикаторите на дисплея, ни принуди да викаме многократно функцията `SendToDisplay()`? За упражнение Ви препоръчвам да промените програмата, така че опресняването на дисплея да се извършва във функцията `Timer0_ISR()`, както го направихме в предната точка.

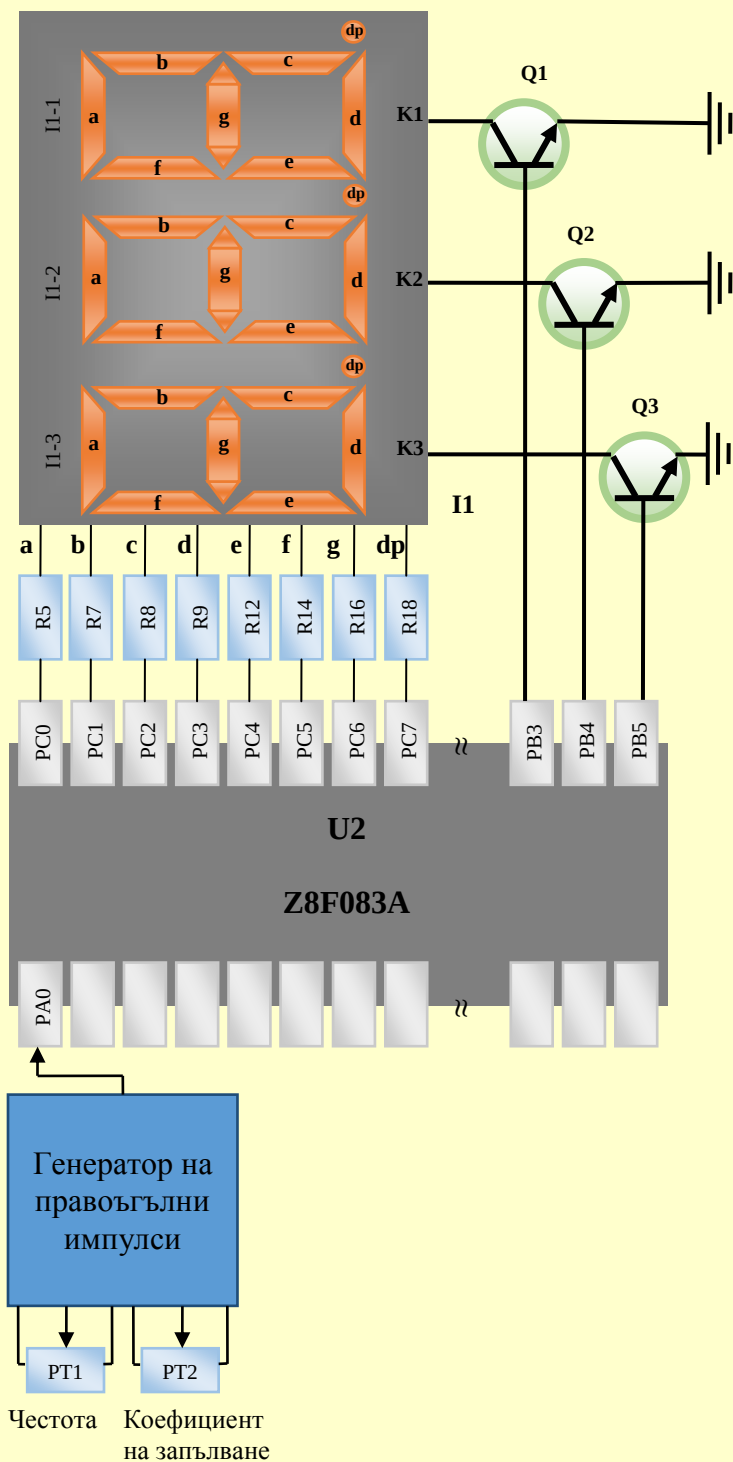
47 Измерване на период (честота)

В тази точка ще разгледаме начините за измерване на период (честота) на правоъгълен входен сигнал с помощта на таймер. За целта свържете разширителната платка **Z8Encore!_EB01(MB01)** към основната платка **Z8Encore!_MB01** и се уверете, че джъмперът JM1 от разширителната платка е в позиция LED. За измерване на период се използва т.нар. режим на прехващане (**capture**). Таймерите в Z8F083A имат три режима на прехващане – режим „прехващане/сравнение”, режим ”прехващане с рестартиране” и класически режим „прехващане”. И трите режима са подходящи за целта. Фиг.116 представлява опростен изглед на част от принципна електрическа схема, която ще ни служи за ориентир. Таймерният вход T0IN (извод PA0) е свързан към изхода на генератор на правоъгълни импулси. Честотата на импулсите може да се променя в диапазона (50÷915)Hz с помощта на потенциометъра PT1, а коефициентът на запълване в диапазона (10÷90)% с PT2.

Първият пример, който ще разгледаме ще използва режим „прехващане/сравнение”. Припомнете си как работи този режим на [Режим на прехващане/сравнение](#). Фиг.115 описва нагледно принципа на работа на този режим.



Фиг. 115 Измерване на период в режим „прехващане/сравнение”



Фиг. 116 Измерване на период – опростена схема

Таймерът започва да брои едва след като на извод РА0 постъпи активен фронт, в случая за активен е избран нарастващия фронт на сигнала. При постъпване на следващия активен фронт текущата стойност на таймерните регистри T0H:T0L се копира (прехваща) в регистрите TOPWMH:TOPWML (този момент е отбелязан с ⚡). За да се измери изтеклото време между двата нарастващи фронта обаче, трябва са вземат предвид препълванията на таймера (тези моменти са отбелязани с 🔥). При тези условия, периодът T ще бъде:

$$T = T(t_3) + n \times T_{OV}$$

където

$T(t_3)$ – прехванатата таймерна стойност.

n – брой препълвания на таймера между два активни фронта.

T_{OV} – време за препълване на таймера.

Очевидно е, че трябва да разрешим прекъсванията от таймера. При прекъсване от препълване (🔥) ще увеличаваме променлива, която ще съдържа броя на препълванията, а при прекъсване от прехващане (⚡) ще изчисляваме периода по горната формула и ще нулираме променливата на препълванията. Бит T0CTL0<INCAP> дава информация, за причината на прекъсването (1 – прекъсването се дължи на прехващащо събитие, 0 – прекъсването се дължи на препълване на таймера). Ще използваме тази информация във функцията, обработваща прекъсванията от Таймер0, която има следния вид:

```
uint32 T; /* Съдържа периода на входния сигнал */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера*/
```

```

uint16 Tov; /* Съдържа времето за препълване на таймера */

if (0 == (T0CTL0 & (1 << 0))) /* Прекъсване от препълване? */
{ /* Да */
    n++; /* Увеличаване на брояча на препълванията */
}
else
{ /* Не, прекъсване от прехващане */

    /* Копиране на прехванатата стойност →  $T = T(t_3)$  */
    T = (((uint32)T0PWMH << 8) | T0PWML);

    /* Вземане на броячната стойност на таймера, която всъщност
    определя времето му за препълване, т.е. . Tov */
    Tov = (((uint16)T0RHN << 8) | T0RL);

    /* Изчисляване на периода  $T += n \times \text{Tov}$  */
    T += (uint32)n*Tov;

    /* Нулиране на брояча на препълванията */
    n = 0;
}
}

```

Периодично в основната програма можем на базата на измерения в Timer_ISR() период да изчисляваме честотата и да я изобразяваме на дисплея. Основната програма изглежда по следния начин:

```

#define FSYS (20000000)
#define FREQ_VALUE_CALCULATION_TIME (25)

void main(void)
{
    uint16    F = 0; /* Честота */
    uint8     i = FREQ_VALUE_CALCULATION_TIME; /* Време за
    изчисляване на нова стойност на F */

    PortConfig();
    TimerConfig();
    InterruptConfig();

    while (1)
    {
        if (i != 0)
        {
            SendToDisplay(F);
            i--;
        }
    }
}

```

```

    }
    else
    {
        _di();/* Забраняване на прекъсванията */
        F = FSYS / T;/* Изчисляване на честотата в Hz */
        _ei();/* Разрешаване на прекъсванията */

        i = FREQ_VALUE_CALCULATION_TIME;/* Презареждане на
времето за изчисляване на нова стойност на F */
    }
}/* Край на while(1) */
}/* Край на main */

```

На всеки 25 извиквания на функцията SendToDisplay() се изчислява нова стойност на честотата F. Стойността на измерения период (която се съдържа в променливата T) е в брой периоди на входния тактов сигнал на таймера T_{TIMER}. За да преобразуваме тази стойност в секунди е необходимо да я умножим по периода на входния тактов сигнал на таймера, който е N/F_{SYs} (N – коефициент на деление на тактовия сигнал, F_{SYs} – честота на системния тактов сигнал), т.е..

$$T_{TIMER} = \frac{1}{F_{TIMER}} = \frac{1}{\frac{F_{SYS}}{N}} = \frac{N}{F_{SYS}}$$

$$T[s] = T \times T_{TIMER} = T \times \frac{N}{F_{SYS}}$$

$$F[Hz] = \frac{1}{T[s]} = \frac{1}{\frac{T \times N}{F_{SYS}}} = \frac{F_{SYS}}{T \times N}$$

В нашия пример коефициентът на деление е N = 1, т.е.. системният тактов сигнал F_{SYs} се използва директно за тактов сигнал на таймера.

Пълната програма придобива следния вид:

```

#include <ez8.h>
#include "stdtypes.h"

```

```

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR     (0x79)

#define FSYS (20000000)
#define FREQ_VALUE_CALCULATION_TIME (25)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
extern void Delay(uint32 u32Delay);
static uint8 ConvertTo7SegIndValue(uint8 digit);

uint32 T; /* Съдържа периода на входния сигнал */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    uint16 Tov; /* Съдържа времето за препълване на таймера */

    if (0 == (T0CTL0 & (1 << 0))) /* Прекъсване от препълване? */
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    { /* Не, прекъсване от прехващане */

        /* Копиране на прехванатата стойност → T = T(t3) */
        T = (((uint32)T0PWMH << 8) | T0PWML);

        /* Вземане на броячната стойност на таймера, която всъщност
        определя времето му за препълване, т.е. . Tov */
        Tov = (((uint16)T0RHL << 8) | T0RL);
    }
}

```

```

    /* Изчисляване на периода T += n x Tov */
    T += (uint32)n*Tov;

    /* Нулиране на брояча на препълванията*/
    n = 0;
}

}

void main(void)
{
    uint16    F = 0; /* Честота */
    uint8     i = FREQ_VALUE_CALCULATION_TIME; /* Време за
изчисляване на нова стойност на F */

    PortConfig();
    TimerConfig();
    InterruptConfig();

    while (1)
    {
        if (i != 0)
        {
            SendToDisplay(F);
            i--;
        }
        else
        {
            _di(); /* Забраняване на прекъсванията */
            F = FSYS / T; /* Изчисляване на честотата в Hz */
            _ei(); /* Разрешаване на прекъсванията */

            i = FREQ_VALUE_CALCULATION_TIME; /* Презареждане на
времето за изчисляване на нова стойност на F */
        }
    } /* Край на while(1) */
} /* Край на main */

void PortConfig(void)
{
    PADD |= (1 << 0);
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;
}

```

```

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xFF;
    T0RL = 0xFF;
    /*-----*/
    /* - Прехващане/сравнение          */
    /* - Активен нарастващ фронт        */
    /* - Коефициент на деление 1       */
    /* - Стартиране на таймера          */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x87;
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

```

```

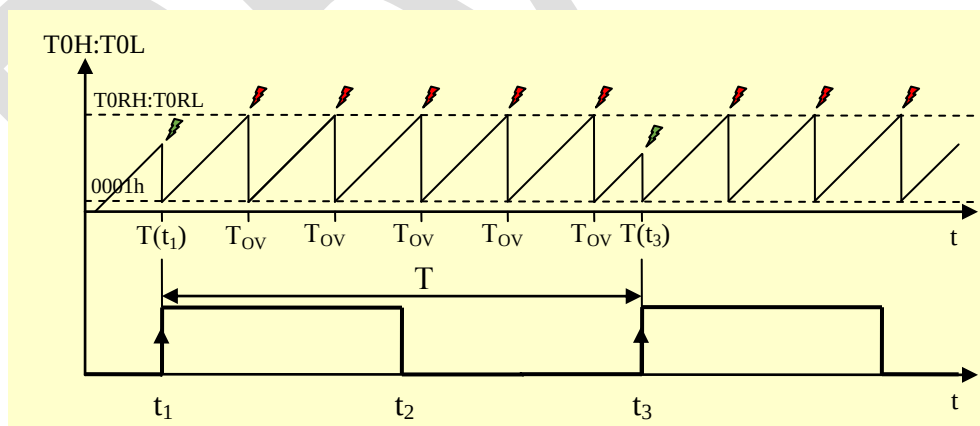
static uint8 ConvertTo7SegIndValue(uint8 digit)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (digit > 9) ? LED_DIGIT_ERR : arr7SegIndValues[digit];
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

```

В следващия пример ще конфигурираме таймера в режим „прехващане с рестартиране”. Припомнете си как работи този режим на [Режим на прехващане с рестартиране](#). Фиг.117 описва нагледно принципа на работа на този режим.

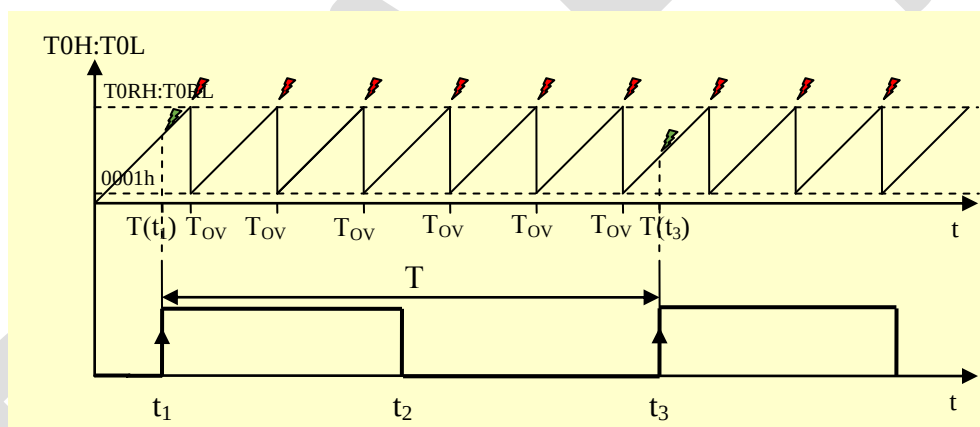


Фиг. 117 Измерване на период в режим „прехващане с рестартиране”

Основната разлика на този режим с предходния е, че таймерът започва да брои веднага след като е разрешен.

Можем да използваме предходната програма (с тази разлика, че таймерът ще бъде конфигуриран да работи в режим „прехващане с рестартиране“). Разликата е, че още първият активен фронт на сигнала ще предизвика прекъсване, но изчислената стойност няма да бъде вярна, тъй като все още нямаме изтекъл цял период. Това като цяло не е проблем. Ако искате, за упражнение може да си добавите код, който игнорира първото прекъсване по нарастващ фронт на таймера.

И третият последен пример ще използва режим „прехващане“. Припомнете си как работи този режим на [Режим на прехващане](#). Фиг.118 описва нагледно принципа на работа на този режим.



Фиг. 118 Измерване на период в режим „прехващане“

Основната разлика спрямо предходните два режима е, че таймерът работи изцяло асинхронно спрямо входния сигнал. При поява на активен фронт на входа текущата стойност на таймерните регистри се прехваща и таймерът продължава да брои. Периодът на входния сигнал в този случай се изчислява по следния начин:

$$T = T(t_3) + n \times T_{OV} - T(t_1)$$

Функцията Timer0_ISR() трябва да бъде модифицирана както е показано по-долу.

```
void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    static uint16 Tcap_old = 0; /* Съхранява последната прехваната
стойност */
    uint16 Tcap_temp; /* Съхранява текущата прехваната
стойност */
    uint16 Tov; /* Съдържа времето за препълване на таймера
*/

    if (0 == (T0CTL0 & (1 << 0))) /* Прекъсване от препълване? */
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    {
        /* Копиране на прехванатата стойност → T = T(t3) */
        T = (((uint32)T0PWML << 8) | T0PWML);

        /* Съхраняване на прехванатата стойност */
        Tcap_temp = T;

        /* Вземане на броячната стойност на таймера, която всъщност
определя времето му за препълване, т.е. . Tov */
        Tov = (((uint16)T0RNL << 8) | T0RNL);

        /* Изчисляване на периода T = T(t3) + n x Tov - T(t1) */
        T += (uint32)n*Tov - Tcap_old;

        /* Съхраняване на прехванатата стойност */
        Tcap_old = Tcap_temp;

        /* Нулиране на брояча на препълванията */
        n = 0;
    }
}
```

Останалият код е същият като в предходните примери (освен, че таймерът е конфигуриран да работи в режим ”прехващане“).

Пълната програма придобива следния вид:

```
#include <ez8.h>
```

```

#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR     (0x79)

#define FSYS (20000000)
#define FREQ_VALUE_CALCULATION_TIME (25)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
extern void Delay(uint32 u32Delay);
static uint8 ConvertTo7SegIndValue(uint8 digit);

uint32 T; /* Съдържа периода на входния сигнал */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на преплъванията на таймера */
    static uint16 Tcap_old = 0; /* Съхранява последната прехваната
стойност */
    uint16 Tcap_temp; /* Съхранява текущата прехваната
стойност */
    uint16 Tov; /* Съдържа времето за преплъване на таймера
*/

    if (0 == (T0CTL0 & (1 << 0))) /* Прекъсване от преплъване? */
    { /* Да */
        n++; /* Увеличаване на брояча на преплъванията */
    }
    else
    {

```

```

/* Копиране на прехванатата стойност →  $T = T(t3)$  */
T = (((uint32)T0PWMH << 8) | T0PWML);

/* Съхраняване на прехванатата стойност */
Tcap_temp = T;

/* Вземане на броячната стойност на таймера, която всъщност
определя времето му за препълване, т.е. . Tov */
Tov = (((uint16)T0RHH << 8) | T0RHL);

/* Изчисляване на периода  $T = T(t3) + n \times Tov - T(t1)$  */
T += (uint32)n*Tov - Tcap_old;

/* Съхраняване на прехванатата стойност */
Tcap_old = Tcap_temp;

/* Нулиране на брояча на препълванията */
n = 0;
}
}

void main(void)
{
    uint16    F = 0; /* Честота */
    uint8     i = FREQ_VALUE_CALCULATION_TIME; /* Време за
изчисляване на нова стойност на F */

    PortConfig();
    TimerConfig();
    InterruptConfig();

    while (1)
    {
        if (i != 0)
        {
            SendToDisplay(F);
            i--;
        }
        else
        {
            _di(); /* Забраняване на прекъсванията */
            F = FSYS / T; /* Изчисляване на честотата в Hz */
            _ei(); /* Разрешаване на прекъсванията */

            i = FREQ_VALUE_CALCULATION_TIME; /* Презареждане на
времето за изчисляване на нова стойност на F */
        }
    } /* Край на while(1) */
} /* Край на main */

void PortConfig(void)

```

```

{
    PADD |= (1 << 0);
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xFF;
    T0RL = 0xFF;

    /*-----*/
    /* - Прехвщане с рестартиране */
    /* - Активен нарастващ фронт */
    /* - Коефициент на деление 1 */
    /* - Стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x84;
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL |= 1 << 7;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

```

```

}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

static uint8 ConvertTo7SegIndValue(uint8 digit)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (digit > 9) ? LED_DIGIT_ERR : arr7SegIndValues[digit];
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

```

В заключение искам да Ви обърна внимание как да определяте типовете на променливите. Просто трябва да сте наясно със стойностите, които тези типове ще съхраняват. Нека да вземем следната дефиниция:

```
uint32 T; /* Съдържа периода на входния сигнал */
```

В зависимост от използвания режим стойността на тази променлива се изчислява по един от следните начини:

$$T = T(t_3) + n \times T_{OV} \text{ или } T = T(t_3) + n \times T_{OV} - T(t_1)$$

Променливата T_{OV} се използва да съхранява броячната стойност на таймера, която е 16-битова ($T0RH:T0RL$), т.е. . за T_{OV} можем да изберем тип `uint16`. Очевидно е, че резултатът от израза $n \times T_{OV}$ може да надхвърли максималната стойност на типа `uint16`, затова избираме тип `uint32` за T . За променливата n е достатъчен тип `uint8`, тъй като при максималния период на генератора на правоъгълни импулси 20ms (50Hz), таймерът ще се препълни най-много 6 пъти ($T_{TIMERmin} = 1/F_{SYS}$, $T_{OV}[ms] = T_{OVmax} * T_{TIMERmin} = 65535 / 20MHz = 3.3$, $20ms / 3.3ms \approx 6$). Обърнете отново внимание на израза $n \times T_{OV}$. Той представлява умножение на тип `uint8` (`unsigned char`) с тип `uint16` (`unsigned short`). Съгласно правилата за преобразуване, компилаторът ще повиши тези типове до `unsigned int` (за eZ8 C-компилятора типовете `unsigned int` и `unsigned short` са 16 бита) и типът на резултата също ще бъде `unsigned int`. Тъй като резултатът може да надхвърли максималната стойност на `unsigned int`, е необходимо да преобразуваме поне един от операндите до `uint32`. Тогава компилаторът автоматично ще преобразува и другия операнд до тип `uint32` и резултатът също ще бъде `uint32`. Ето защо преобразуван в C-код изразът $n \times T_{OV}$ ще бъде:

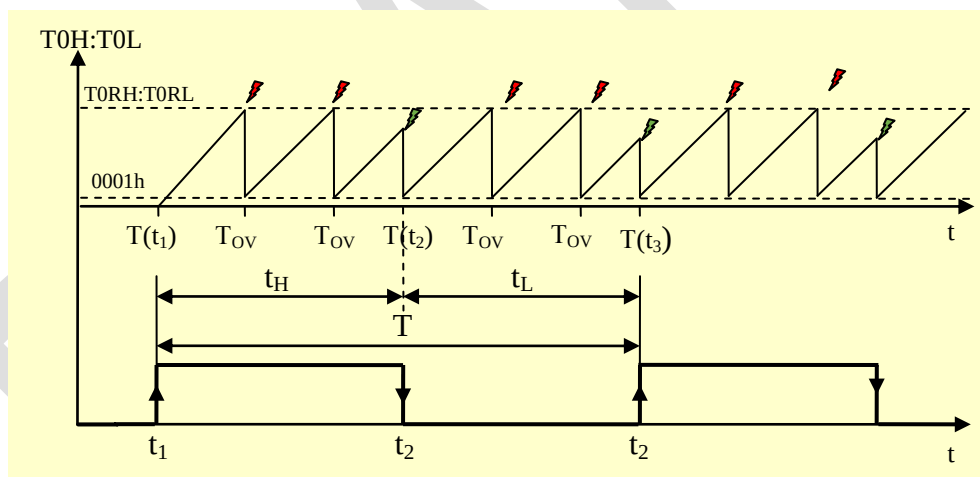
`(uint32)n*Tov`

Затова ако извършвате аритметични изчисления и получавате грешен резултат, въпреки че математически изразът Ви изглежда верен, проверете дали всъщност причината не е препълване.

48 Измерване на коефициент на запълване

Тази точка е продължение на предходната. В нея ще Ви запозная с измерването на коефициент на запълване на правоъгълен сигнал. Постановката е същата като предходната (Фиг.116). Коефициентът на запълване на измервания сигнал може да се променя в диапазона 10% ÷ 90% с помощта на потенциометъра РТ2. За да изчислим коефициента на запълване, освен периода на входния сигнал, трябва да измерим и продължителността на високото ниво. Отново ще използваме трите режима на таймера в същата последователност.

Фиг.119 описва принципа на измерване на коефициента на запълване в режим „прехващане/сравнение”.



Фиг. 119 Измерване на коефициент на запълване в режим „прехващане/сравнение”

$t_H = T(t_2) + n \times T_{OV}$ - високо ниво на сигнала

$t_L = T(t_3) + n \times T_{OV}$ - ниско ниво на сигнала

$T = t_H + t_L$ - период на сигнала

$k[\%] = \frac{t_H}{T} \times 100 = \frac{t_H}{t_H + t_L} \times 100$ - коефициент на запълване на сигнала

където

$T(t_2)$ - стойността на таймера по време на падащия фронт

$T(t_3)$ - стойността на таймера по време на нарастващия фронт

n - брой препълвания на таймера между два противоположни фронта

T_{OV} - време за препълване на таймера

Промените, които трябва да извършим спрямо измерването само на периода, се намират във функцията `Timer0_ISR()`.

Когато прекъсването се дължи на падащ фронт, е необходимо да изчислим t_H , да нулираме брояча на препълванията на таймера и да конфигурираме активния фронт да е нарастващ.

Когато прекъсването се дължи на нарастващ фронт, е необходимо да изчислим t_L , да нулираме брояча на препълванията на таймера и да конфигурираме активния фронт да е падащ. Също така можем да изчислим и коефициента на запълване, тъй като вече сме измерили t_H .

Когато прекъсването се дължи на препълване на таймера, е необходимо просто да увеличаваме брояча на тези препълвания.

Функцията `Timer0_ISR()` придобива следния вид:

```
uint8 k; /* Коефициент на запълване */
```

```

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    static uint32 t_h = 0; /* Високо ниво на сигнала */
    static uint32 t_l = 0; /* Ниско ниво на сигнала */

    uint32 T; /* Период на сигнала */
    uint32 Tov; /* Време за препълване на таймера */
    uint32 Tcap; /* Прехваната стойност */

    /* Прекъсване от препълване ? */
    if (0 == (T0CTL0 & (1 << 0)))
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    { /* Не */
        Tov = (((uint32)T0RH << 8) | T0RL); /* Tov */
        Tcap = (((uint32)T0PWH << 8) | T0PWL); /* Tcap */

        /* Прекъсване от нарастващ фронт ? */
        if (0 == (T0CTL1 & (1 << 6)))
        { /* Да */
            t_l = Tcap + n * Tov; /* Изчисляване на ниското ниво на
сигнала */
            T = t_h + t_l; /* Изчисляване на периода */
            k = (t_h * 100) / T; /* Изчисляване на коефициента на
запълване */
            n = 0; /* Нулиране на брояча на препълванията */
            T0CTL1 |= 1 << 6; /* Превключване към падащ фронт */
        }
        else
        { /* Не */
            t_h = Tcap + n * Tov; /* Изчисляване на високото ниво
*/
            n = 0; /* Нулиране на брояча на препълванията */
            T0CTL1 &= ~(1 << 6); /* Превключване към нарастващ
фронт */
        }
    }
}

```

В основната програма можем периодично да четем променливата k и да изобразяваме стойността ѝ на дисплея.

Пълната програма придобива следния вид.

```
#include <ez8.h>
```

```

#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR     (0x79)

#define DUTY_VALUE_READING_TIME (25)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 digit);
extern void Delay(uint32 u32Delay);

uint8 k; /* Коефициент на запълване */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на преплъванията на таймера */
    static uint32 t_h = 0; /* Високо ниво на сигнала */
    static uint32 t_l = 0; /* Ниско ниво на сигнала */

    uint32 T; /* Период на сигнала */
    uint32 Tov; /* Време за преплъване на таймера */
    uint32 Tcap; /* Прехваната стойност */

    /* Прекъсване от преплъване ? */
    if (0 == (T0CTL0 & (1 << 0)))
    { /* Да */
        n++; /* Увеличаване на брояча на преплъванията */
    }
    else
    { /* Не */

```

```

Tov = (((uint16)T0RH << 8) | T0RL);/* Tov */
Tcap = (((uint32)T0PWMH << 8) | T0PWML);/* Tcap */

/* Прекъсване от нарастващ фронт ? */
if (0 == (T0CTL1 & (1 << 6)))
{/* Да */
    t_l = Tcap + n* Tov; /* Изчисляване на ниското ниво на
сигнала */
    T = t_h + t_l; /* Изчисляване на периода */
    k = (t_h * 100) / T; /* Изчисляване на коефициента на
запълване */
    n = 0; /* Нулиране на брояча на препълванията */
    T0CTL1 |= 1 << 6; /* Превключване към падащ фронт */
}
else
{/* Не */
    t_h = Tcap + n* Tov; /* Изчисляване на високото ниво
*/
    n = 0; /* Нулиране на брояча на препълванията */
    T0CTL1 &= ~(1 << 6); /* Превключване към нарастващ
фронт */
}
}

void main(void)
{
    uint8    dutyCycle = 0;
    uint8    i = DUTY_VALUE_READING_TIME; /* Време за изчитане на
нова стойност на dutyCycle */

    PortConfig();
    InterruptConfig();
    TimerConfig();

    while (1)
    {
        if (i != 0)
        {
            SendToDisplay(dutyCycle);
            i--;
        }
        else
        {
            _di();
            dutyCycle = k; /* Четене на измерения
коефициент */
            _ei();

            i = DUTY_VALUE_READING_TIME; /* Презареждане на
времето за изчитане на нова стойност на dutyCycle */

```

```

    }
}

void PortConfig(void)
{
    PADD  |= (1 << 0);
    PCOUT  = 0x00;
    PCDD   = 0x00;
    PBOUT  &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD   &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xFF;
    T0RL = 0xFF;

    /*-----*/
    /* - Прехващане/сравнение           */
    /* - Активен нарастващ фронт         */
    /* - Коефициент на деление 1        */
    /* - Стартиране на таймера           */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x87;
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
}

```

```

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

static uint8 ConvertTo7SegIndValue(uint8 digit)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

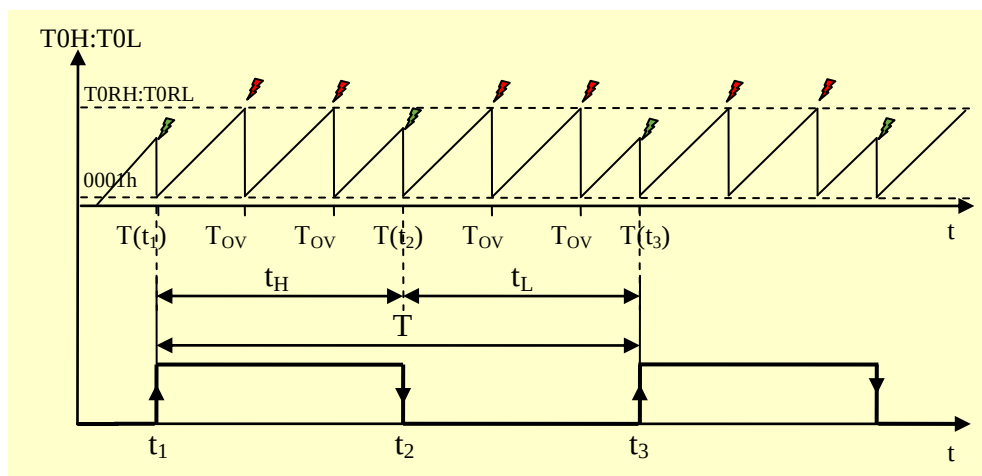
    return (digit > 9) ? LED_DIGIT_ERR : arr7SegIndValues[digit];
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

```

Първото изчисляване на коефициента на запълване няма да е вярно. Ако искате за упражнение, може да си добавите код, който игнорира първото прекъсване по нарастващ фронт на таймера.

В следващия пример ще конфигурираме таймера в режим „прехващане с рестартиране”. Фиг.120 описва принципа на измерване на коефициента на запълване в този режим.



Фиг. 120 Измерване на коефициент на запълване в режим „прехващане с рестартиране”

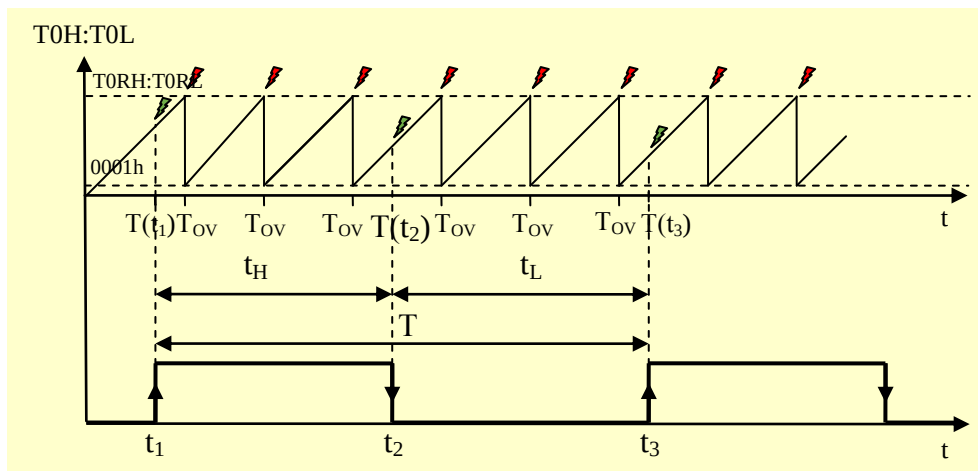
Единствената разлика спрямо предходния пример е, че трябва да конфигурираме таймера в режим „прехващане с рестартиране”, останалият код е същият. Отново първото изчисляване на коефициента на запълване няма да е вярно. Ако искате за упражнение, може да си добавите код, който игнорира първото прекъсване по нарастващ фронт на таймера.

И третият последен пример ще използва режим „прехващане”. Фиг.121 описва принципа на измерване на коефициента на запълване в този режим.

$$t_H = T(t_2) + n \times T_{OV} - T(t_1) \text{ - високо ниво на сигнала}$$

$$t_L = T(t_3) + n \times T_{OV} - T(t_2) \text{ - ниско ниво на сигнала}$$

$$T = t_H + t_L \text{ - период на сигнала}$$



Промените, които трябва да направим в `Timer0_ISR()` са показани в сиво каре. Простичко казано, трябва да запазваме всяка прехваната стойност, тъй като тя ще ни е нужна в изчисленията, извършвани при следващото прехващане.

```
void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    static uint32 t_h = 0; /* Високо ниво на сигнала */
    static uint32 t_l = 0; /* Ниско ниво на сигнала */
    static uint32 Tcap_old = 0; /* Съхранява прехванатите стойности
и по двата фронта */

    uint32 T; /* Период на сигнала */
    uint32 Tov; /* Време за препълване на таймера */
    uint32 Tcap; /* Прехванатата стойност */

    /* Прекъсване от препълване ? */
    if (0 == (T0CTL0 & (1 << 0)))
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    { /* Не */
        Tov = (((uint32)T0RH << 8) | T0RL); /* Tov */
        Tcap = (((uint32)T0PWMH << 8) | T0PWML); /* Tcap */

        /* Прекъсване от нарастващ фронт ? */
        if (0 == (T0CTL1 & (1 << 6)))
        { /* Да */
```



```

        t_l = Tcap + n* Tov - Tcap_old; /* Изчисляване на
ниското ниво на сигнала */
        T = t_h + t_l; /* Изчисляване на периода */
        k = (t_h * 100) / T; /* Изчисляване на коефициента на
запълване */
        n = 0; /* Нулиране на брояча на препълванията */
        T0CTL1 |= 1 << 6; /* Превключване към падащ фронт */
    }
    else
    { /* Не */
        t_h = Tcap + n* Tov - Tcap_old; /* Изчисляване на
високото ниво */
        n = 0; /* Нулиране на брояча на препълванията */
        T0CTL1 &= ~(1 << 6); /* Превключване към нарастващ
фронт */
    }

    Tcap_old = Tcap; /* Запазване на прехваната стойност */
}
}

```

Ето и кода на цялата програма:

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC      (505250)
#define DELAY_CONSTANT_500_MSEC  (252625)
#define DELAY_CONSTANT_10_MSEC   (5052)
#define DELAY_CONSTANT_5_MSEC    (2526)

#define LED_DIGIT_0      (0x3F)
#define LED_DIGIT_1      (0x06)
#define LED_DIGIT_2      (0x5B)
#define LED_DIGIT_3      (0x4F)
#define LED_DIGIT_4      (0x66)
#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP      (0x80)
#define LED_DIGIT_ERR      (0x79)

#define DUTY_VALUE_READING_TIME (25)

extern void PortConfig(void);
extern void InterruptConfig(void);
extern void TimerConfig(void);

```

```

extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 digit);
extern void Delay(uint32 u32Delay);

uint8 k; /* Коефициент на запълване */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    static uint32 t_h = 0; /* Високо ниво на сигнала */
    static uint32 t_l = 0; /* Ниско ниво на сигнала */
    static uint32 Tcap_old = 0; /* Съхранява прехванатите стойности
и по двата фронта */

    uint32 T; /* Период на сигнала */
    uint32 Tov; /* Време за препълване на таймера */
    uint32 Tcap; /* Прехванатата стойност */

    /* Прекъсване от препълване ? */
    if (0 == (T0CTL0 & (1 << 0)))
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    { /* Не */
        Tov = (((uint32)T0RH << 8) | T0RL); /* Tov */
        Tcap = (((uint32)T0PWMH << 8) | T0PWML); /* Tcap */

        /* Прекъсване от нарастващ фронт ? */
        if (0 == (T0CTL1 & (1 << 6)))
        { /* Да */
            t_l = Tcap + n* Tov - Tcap_old; /* Изчисляване на
ниското ниво на сигнала */
            T = t_h + t_l; /* Изчисляване на периода */
            k = (t_h * 100) / T; /* Изчисляване на коефициента на
запълване */
            n = 0; /* Нулиране на брояча на препълванията */
            T0CTL1 |= 1 << 6; /* Превключване към падащ фронт */
        }
        else
        { /* Не */
            t_h = Tcap + n* Tov - Tcap_old; /* Изчисляване на
високото ниво */
            n = 0; /* Нулиране на брояча на препълванията */
            T0CTL1 &= ~(1 << 6); /* Превключване към нарастващ
фронт */
        }
    }
}

```

```

        Tcap_old = Tcap; /* Запазване на прехваната стойност */
    }
}

void main(void)
{
    uint8 dutyCycle = 0;
    uint8 i = DUTY_VALUE_READING_TIME; /* Време за изчитане на
    нова стойност на dutyCycle */

    PortConfig();
    InterruptConfig();
    TimerConfig();

    while (1)
    {
        if (i != 0)
        {
            SendToDisplay(dutyCycle);
            i--;
        }
        else
        {
            _di();
            dutyCycle = k; /* Четене на измерения
            коефициент */
            _ei();

            i = DUTY_VALUE_READING_TIME; /* Презареждане на
            времето за изчитане на нова стойност на dutyCycle */
        }
    }
}

void PortConfig(void)
{
    PADD |= (1 << 0);
    PCOUT = 0x00;
    PCDD = 0x00;
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xFF;

```

```

T0RL = 0xFF;

/*-----*/
/* - Прехващане с рестартиране      */
/* - Активен нарастващ фронт          */
/* - Коефициент на деление 1         */
/* - Стартиране на таймера            */
/*-----*/
T0CTL0 = 0x00;
T0CTL1 = 0x84;
}

void InterruptConfig(void)
{
    /* Разрешаване на прекъсването от Таймер0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;
}

void SendToDisplay(uint16 value)
{
    SendToIndicator1(value % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator2((value / 10) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
    SendToIndicator3((value / 100) % 10);
    Delay(DELAY_CONSTANT_5_MSEC);
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

```

```

static uint8 ConvertTo7SegIndValue(uint8 digit)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (digit > 9) ? LED_DIGIT_ERR : arr7SegIndValues[digit];
}

void Delay(uint32 delay)
{
    for (; delay > 0; delay--);
}

```

Естеството на работа на таймера в режим „прехващане” позволява измерването на коефициента на запълване да се извърши по малко по-различен начин от предходния. Тъй като таймерът не се нулира при възникване на прехващащо събитие, вместо да измерваме ниското ниво на сигнала t_L , може директно да измерим периода T , както го направихме в предходната точка. За целта при възникване на прехващане по падащ фронт не трябва да нулираме брояча на препълванията на таймера и трябва да съхраняваме прехваната стойност само при нарастващ фронт. Това би спестило малко изчисления в програмата.

$t_H = T(t_2) + n \times T_{OV} - T(t_1)$ - високо ниво на сигнала

$T = T(t_3) + n \times T_{OV} - T(t_1)$ - период на сигнала

Промените, които трябва да направим във функцията `Timer0_ISR()` са показани в сиво кадре.

```

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8 n = 0; /* Брояч на препълванията на таймера */
    static uint32 t_h = 0; /* Високо ниво на сигнала */
    static uint32 t_l = 0; /* Ниско ниво на сигнала */
    static uint32 Tcap_old = 0; /* Съхранява прехванатите
стойности при нарастващ фронт */

    uint32 T; /* Период на сигнала */
    uint32 Tov; /* Време за препълване на таймера */
    uint32 Tcap; /* Прехваната стойност */

    /* Прекъсване от препълване ? */
    if (0 == (T0CTL0 & (1 << 0)))
    { /* Да */
        n++; /* Увеличаване на брояча на препълванията */
    }
    else
    { /* Не */
        Tov = (((uint32)T0RH << 8) | T0RL); /* Tov */
        Tcap = (((uint32)T0PWMH << 8) | T0PWML); /* Tcap */

        /* Прекъсване от нарастващ фронт ? */
        if (0 == (T0CTL1 & (1 << 6)))
        { /* Да */
            T = Tcap + n*Tov - Tcap_old; /* Изчисляване на
периода на сигнала */
            k = (t_h * 100) / T; /* Изчисляване на
коефициента на запълване */
            n = 0; /* Нулиране на брояча на препълванията */
            T0CTL1 |= 1 << 6; /* Превключване към падащ
фронт */
            Tcap_old = Tcap; /* Запазване на прехваната
стойност */
        }
        else
        { /* Не */
            t_h = Tcap + n*Tov - Tcap_old; /* Изчисляване на
високото ниво */
            T0CTL1 &= ~(1 << 6); /* Превключване към
нарастащ фронт */
        }
    }
}

```

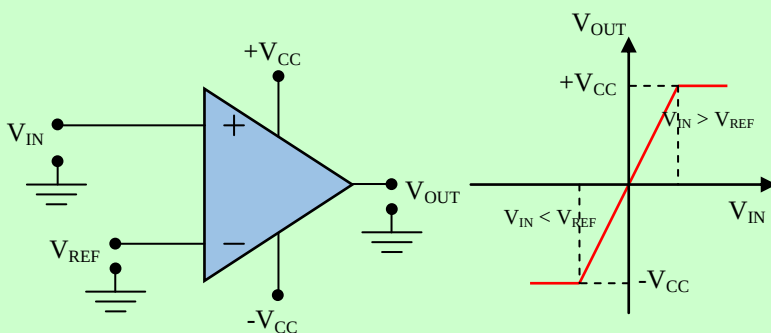
За да избегна необходимостта от явно преобразуване в изразите за изчисление на t_H , t_L и T , дефинирах променливите Tov , $Tcap$ и $Tcap_old$ да са тип `uint32`, въпреки че `uint16` е достатъчно голям да побере стойностите им.

49 Детектор на ниско напрежение

Микроконтролерът Z8F083A има вграден аналогов компаратор. В тази точка ще използваме този компаратор и ще реализираме с негова помощ прост детектор на ниско напрежение. За целта свържете разширителната платка **Z8Encore!_EB01(MB01)** към основната платка **Z8Encore!_MB01** и се уверете, че джъмперите JM1 и JM3 от разширителната платка са в позиция CMP.

Какво е аналогов компаратор?

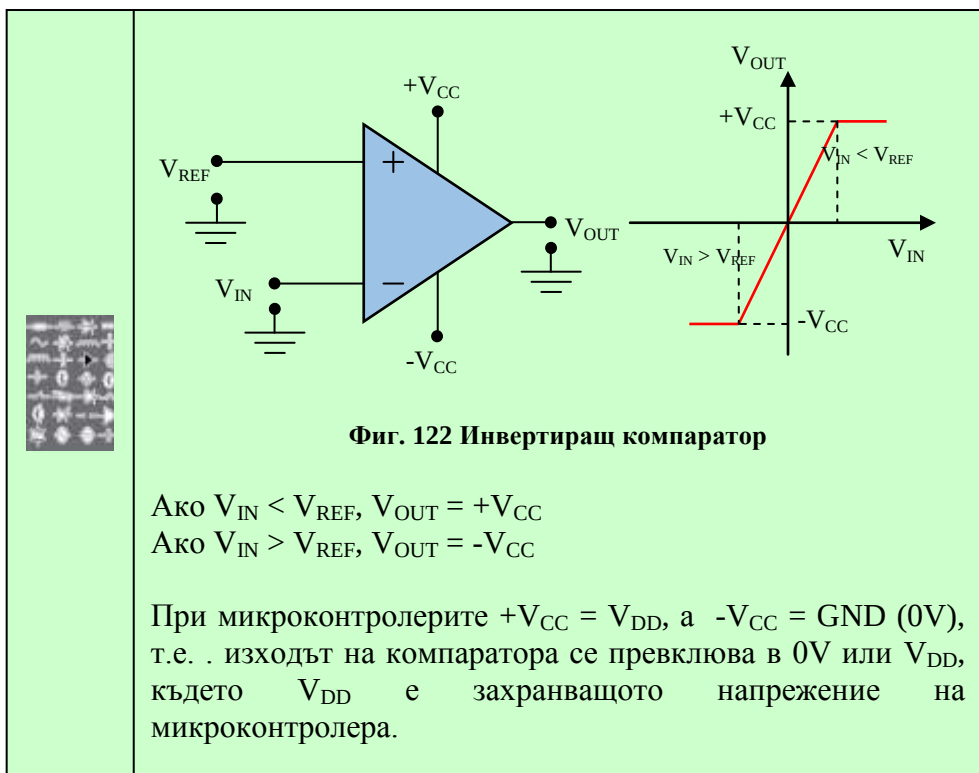
Аналоговият компаратор е електронно устройство, което сравнява две аналогови напрежения. Единият от входовете на компаратора служи за подаване на опорно напрежение, а другият за подаване на напрежението, което ще се сравнява. В зависимост това кой вход се използва за опорно напрежение, компараторите биват неинвертиращи и инвертиращи. В зависимост от входните напрежения, изходът на компараторът се превключва в едно от две възможни състояния. Фиг.121 показва неинвертиращ аналогов компаратор, а Фиг.122 – инвертиращ аналогов компаратор.



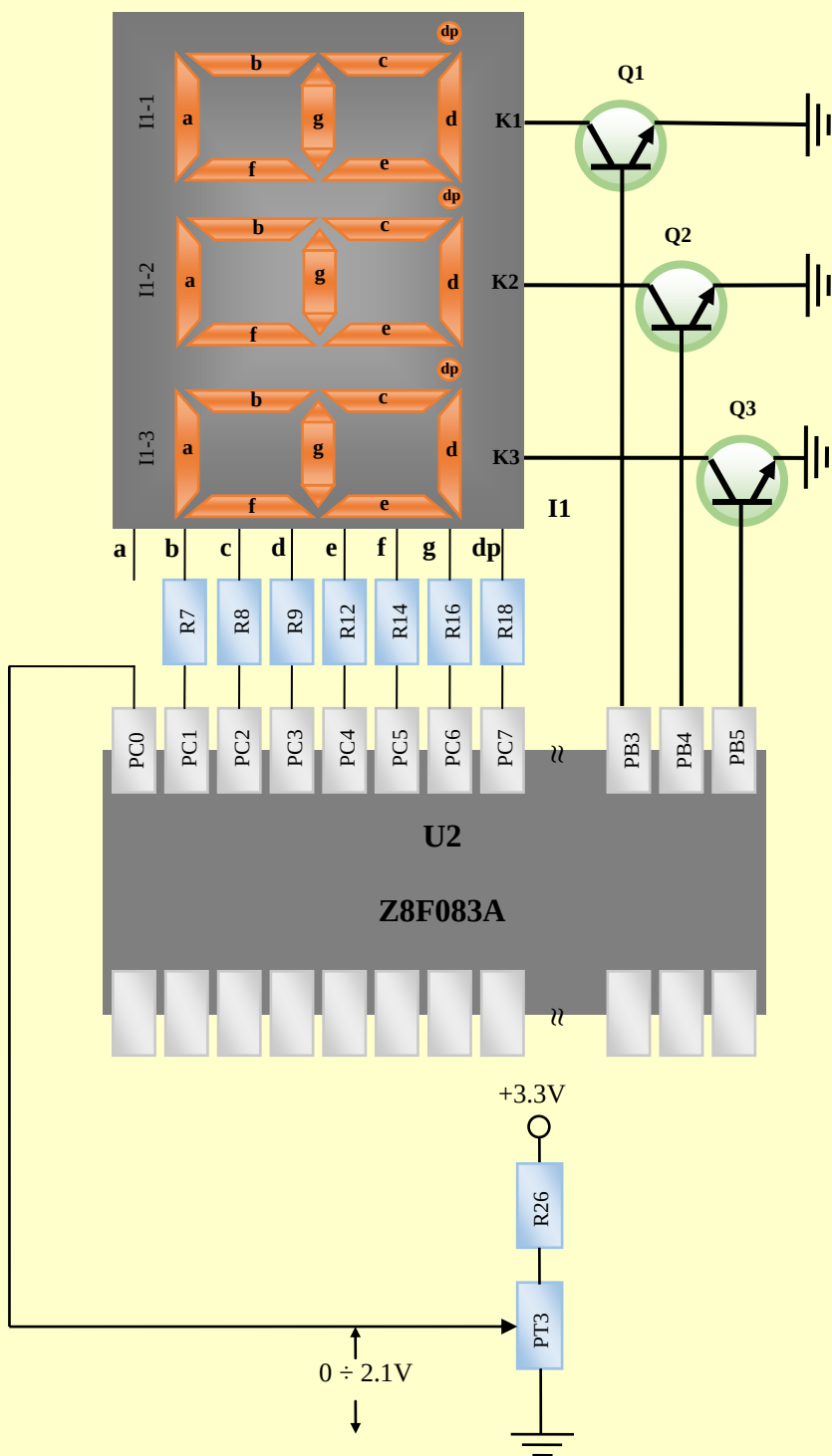
Фиг. 121 Неинвертиращ компаратор

Ако $V_{IN} < V_{REF}$, $V_{OUT} = -V_{CC}$

Ако $V_{IN} > V_{REF}$, $V_{OUT} = +V_{CC}$



Фиг.123 представлява опростен изглед на част от принципна електрическа схема, която ще ни служи за ориентир. Припомнете си как работи компаратора на [13 Компаратор](#). В нашият пример ще се възползваме от факта, че на отрицателния вход на компаратора може вътрешно да се подава опорно напрежение в диапазона $0V \div 1.8V$ със стъпка $0.2V$. Положителния вход на компаратора е изведен на извод PC0. С помощта на потенциометъра PT3 ще подаваме входно напрежение на този извод в диапазона $0V \div 2.1V$. Изходът на компаратора е изведен на извод PC3, който пък е свързан с d-сегментите на индикаторите на дисплея I1. Ще се възползваме от това и ще управляваме примерно d-сегментът на индикатора I1-1. Тъй като входното напрежение ще се подава на положителния вход на компаратора, ясно е че той ще работи в неинвертиращ режим, т.е.. докато напрежението на извод PC0 е по-голямо от вътрешното опорно напрежение, изходът на компаратора ще е във високо ниво и d-сегментът на I1-1 ще свети. Когато напре-



Фиг. 123 Детектор на ниско на напрежение – опростена схема

жението на PC0 падне под вътрешното опорно напрежение, изходът на компаратора ще се превключи в ниско ниво и d-сегментът на I1-1 ще изгасне.

Единствената новост в този пример е, че ще трябва да конфигурираме изводите PC0 и PC3 да използват алтернативните си режими – извод PC0 ще работи като положителен вход на компаратора (CINP), а PC3 като изход на компаратора (COUT).

Ето и цялата програма:

```
#include <ez8.h>

extern void PortConfig(void);
extern void ComparatorConfig(void);

void main(void)
{
    PortConfig();
    ComparatorConfig();

    while (1);
}

void PortConfig(void)
{
    PCAF |= ((1 << 0) | (1 << 3)); /* Разрешаване на
алтернативните функции на изводи PC0 и PC3 */
    PCAFS1 |= 1 << 0; /* PC0 работи като CINP */
    PCAFS1 &= ~(1 << 3); /* PC3 работи като COUT */
    PBDD &= ~(1 << 3); /* PB3 работи като цифров изход */
    PBOUT |= 1 << 3; /* Включване на Q1 */
}

void ComparatorConfig(void)
{
    CMP0 = 0x64; /* VREF = 1.8V */
}
```

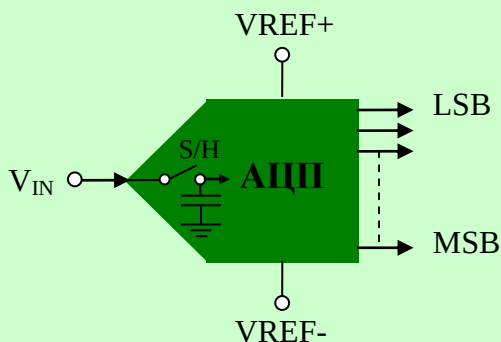
Завъртането на потенциометъра по посока на часовниковата стрелка води до намаляване на напрежението на извод PA0. Когато това напрежение падне по 1.8V, d-сегментът на I1-1 ще изгасне, в противен случай ще свети.

50 Измерване на напрежение

В тази точка ще разработим прост волтметър. За целта свържете разширителната платка **Z8Encore!_EB01(MB01)** към основната платка **Z8Encore!_MB01** и се уверете, че джъмперите JM1 и JM3 от разширителната платка са в позиция LED и ADC съответно. Фиг 127 представлява опростен изглед на част от принципна електрическа схема, която ще ни служи за ориентир. Измерването на напрежение се извършва с помощта на АЦП (Аналого-цифров преобразувател).

Какво е Аналого-цифров преобразувател ?

АЦП (ADC – **Analog-to-Digital Converter**) е електронно устройство, което преобразува аналогово напрежение в цифров сигнал. Резултатът е 8-, 10-, 12-битов и т.н.

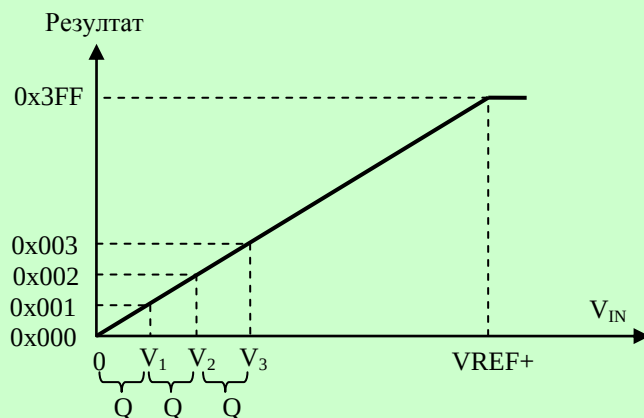


Фиг. 124 Аналого-цифров преобразувател

Обхватът на входното напрежение, което може да се измери от АЦП, се определя от опорните напрежения $VREF+$ и $VREF-$, т.е. $VREF- \leq V_{IN} \leq VREF+$.

При микроконтролерите $VREF-$ типично е свързан към GND, т.е.. входният диапазон от напрежение е в обхвата $0V \div VREF+$. Разделителната способност Q на АЦП-то е $VREF+ / 2^n$, където n е броят на битовете, с които се представя резултата. Например ако $VREF+ = 5V$ и $n = 10$, $Q = 5 / 2^{10} = 5 / 1024 = 0,0048828125V$. Фиг.126 показва

т.нар. трансферна характеристика на АЦП.



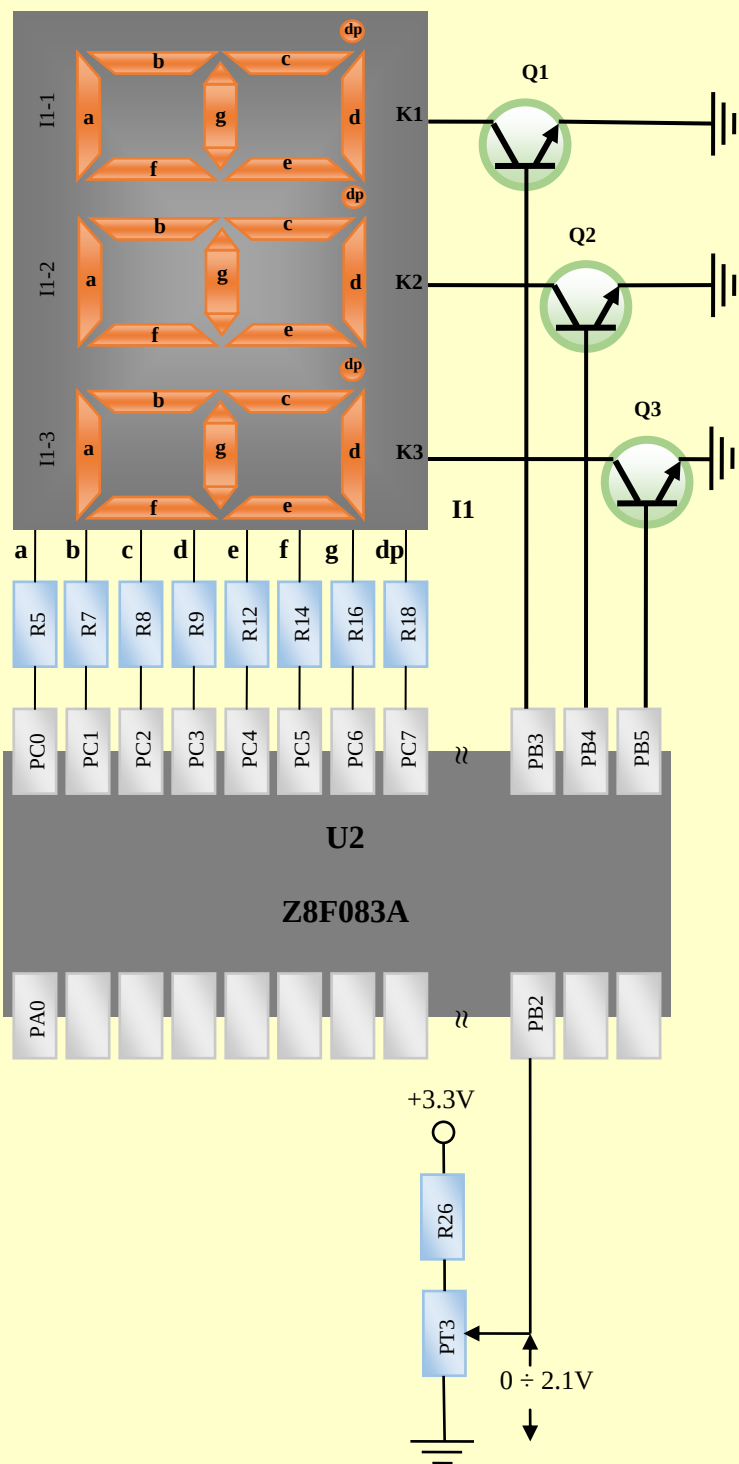
Фиг. 126 Трансферна характеристика на АЦП

Входното напрежение се подава на схема **Sample&Hold (S/H)**, която го „запомня”. Времето необходимо за запомняне на входното напрежение се нарича време за семплиране T_S . Времето за преобразуване на запомненото напрежение се нарича време на задръжка T_H . Сумарно двата параметъра определят бързодействието на АЦП-то.

Припомнете си как работи АЦП-то на [14 Аналого-цифров преобразувател](#). В нашият пример ще измерваме напрежението на аналоговия вход ANA2. Този вход е свързан към извод PB2 на микроконтролера. С помощта на потенциометъра PT3 можем да променяме входното напрежение в диапазона $0 \div 2.1V$. Ще изобразяваме измереното напрежение на дисплея във формат D.DD, т.е. показваното напрежение ще бъде в диапазона $0.00V \div 2.00V$. За опорно напрежение ще използваме вътрешния източник $2V$, предоставен от микроконтролер Z8F083A.

Новото в този пример е настройването на АЦП-то. Първата стъпка, която трябва да направим е да конфигурираме извод PB2 да работи като аналогов вход. Това става по следния начин:

PBAF |= (1 << 2);/* Разрешаване на алтернативната функция на



Фиг. 127 Измерване на напрежение – опростена схема

```
PB2 */
PCAFS1 |= 1 << 2; /* PB2 работи като аналогов вход */
```

Втората стъпка е да конфигурираме АЦП-то.

```
ADCCTL0 = 0x32; /* 1. VREFIN е източника на опорно напрежение */
               /* 2. Разрешаване на АЦП-то */
               /* 3. Преобразуване на напрежението на ANA2 */
ADCCP = 0x01; /* FADC = FSYS/2 = 10MHz */
ADCSST = 0x14; /* TS = (1/FSYS)*ADCST<3:0> = (1/20MHz)*20 = 1us */
ADCST = 0x28; /* TH = (1/FSYS)*ADCST<5:0> = (1/20MHz)*40 = 2us */
```

При тези настройки пълното време за преобразуване на канала ще бъде:

$$T_{\text{CONV}} = T_S + T_H + 13 \cdot (1/F_{\text{SYS}}) \cdot \text{DIV} = 1\mu\text{s} + 2\mu\text{s} + 13 \cdot (1/20\text{MHz}) \cdot 2 = 4.3\mu\text{s}$$

За да стартираме преобразуването, трябва да установим бит ADCCTL0<START>:

```
ADCCTL0 |= (1 << 7); /* Стартиране на преобразуването */
```

Същия бит се използва и за индикация, че преобразуването е завършило. 1 показва, че преобразуването е в прогрес, докато 0, че е завършило.

Можем да използваме това и в основната програма периодично да проверяваме дали преобразуването е завършило, и когато това стане да изчислим измереното напрежение, да го представим във формата D.DD, след което отново да стартираме преобразуването и да повторим целия процес.

```
#define Q (0.001953125) /* Разделителна способност на АЦП-то */

while (1)
{
    uint16 adcResult; /* Съхранява резултата от измерването */

    AdcStart(); /* Стартиране на преобразуването */
```

```

while (1 == (ADCCTL0 & (1 << 7)));/* Изчакване на завършване
на преобразуването */
adcResult = (((uint16)ADCD_H << 2)) | (ADCD_L >> 6));/*
Копиране на резултата */
_di();/* Забраняване на прекъсванията */
voltage = ((adcResult * Q) * 100);/* Изчисляване на стойността
до формат D.DD */
_ei();/* Разрешаване на прекъсванията */
}

```

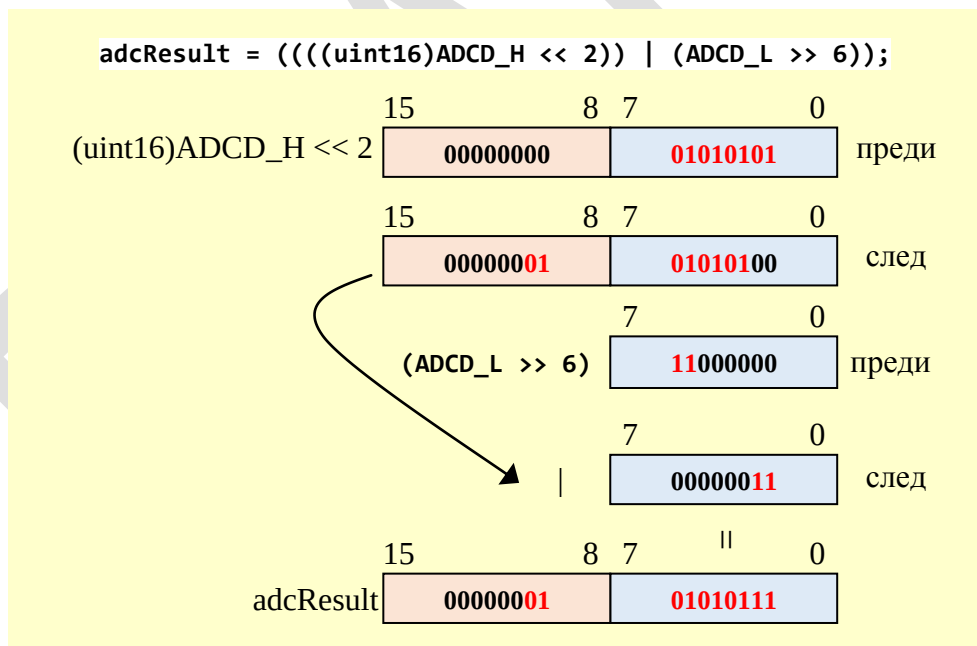
Две неща заслужават внимание в горните редове:

```

adcResult = (((uint16)ADCD_H << 2)) | (ADCD_L >> 6));/*
Копиране на резултата */

```

Следващата фигура описва нагледно копирането на резултата в 16-битовата променлива adcResult. Допуска се, че резултатът е 0x157 (010101011₂).



Фиг. 128 `adcResult = ADCD_H:ADCD_L`

```

voltage = ((adcResult * Q) * 100);/* Изчисляване на стойността до
формат D.DD */

```

Стойността, която може да бъде записана в `adcResult`, е в

диапазона $0 \div 1023$ в зависимост от входното напрежение на извод PB2 ($0V \div 2V$)¹.

Забележка: Обърнете внимание, че ако входното напрежение надвиши V_{REFIN} , АЦП-то се насища и измерената стойност винаги ще бъде 0x3FF (1023₁₀).

За да преобразуваме това напрежение във волтове, е необходимо да го умножим по разделителната способност на АЦП-то, която в нашия случай е $V_{REFINT} / 2^n = 2 / 2^{10} = 2 / 1024 = 0.001953125V$. Полученият резултат ще има формата $D_1.D_2D_3D_4...$. Умножавайки го по 100, ще преместим две цифри след десетичната запетая наляво, т.е.. $D_1D_2D_3.D_4...$. Така полученият резултат се съхранява в 8-битовата целочислена променлива, при което само цялата част $D_1D_2D_3$ ще се копира, а дробната част $.D_4...$ ще се изхвърли, т.е.. резултатът в voltage ще бъде десетично число в диапазона $000 \div 200$, като ние мислено знаем, че десетичната точка се намира между цифрите D_1 и D_2 .

Това което остана да направим е да изобразяваме изчисленото напрежение на дисплея. Тази задача вече сме я правили, така че кодът ще Ви бъде познат. Ще изобразяваме нови стойности на всеки 0.5 секунди например.

```
void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8  displayCounter = 0;
    static uint8  refreshCounter = 0;
    static uint8  value = 0;

    displayCounter++;
    refreshCounter++;

    if (100 == refreshCounter)/* 0.5 секунди ? */
    {/* Да */
        value = voltage;/* Копиране на последната изчислена
стойност */
        refreshCounter = 0;
    }
}
```



```

switch (displayCounter)
{
case 1:
    SendToIndicator1(value % 10);
    break;

case 2:
    SendToIndicator2((value / 10) % 10);
    break;

case 3:
    SendToIndicator3((value / 100) % 10);
    displayCounter = 0;
    break;
}
}

```

Единствената новост е леката модификация при изобразяването на най-дясната цифра (вижте сивото каре).

```

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = LED_DIGIT_DP | ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

```

Резултатът от функцията `ConvertTo7SegIndValue()` се комбинира със стойността `LED_DIGIT_DP`, която води до включване на сегмента DP на индикатора, така че резултатът да се изобразява във формата $D_1.D_2D_3$.

Ето и цялата програма:

```

#include <ez8.h>
#include "stdtypes.h"

#define DELAY_CONSTANT_1_SEC    (505250)
#define DELAY_CONSTANT_500_MSEC (252625)
#define DELAY_CONSTANT_5_MSEC  (2526)

#define LED_DIGIT_0    (0x3F)
#define LED_DIGIT_1    (0x06)
#define LED_DIGIT_2    (0x5B)
#define LED_DIGIT_3    (0x4F)
#define LED_DIGIT_4    (0x66)

```

```

#define LED_DIGIT_5      (0x6D)
#define LED_DIGIT_6      (0x7D)
#define LED_DIGIT_7      (0x07)
#define LED_DIGIT_8      (0x7F)
#define LED_DIGIT_9      (0x6F)
#define LED_DIGIT_DP     (0x80)
#define LED_DIGIT_ERR     (0x79)

#define ADC_RESOLUTION    (0.001953125)

extern void PortConfig(void);
extern void TimerConfig(void);
extern void InterruptConfig(void);
extern void AdcConfig(void);
extern void AdcStart(void);
extern void SendToDisplay(uint16 value);
extern void SendToIndicator1(uint8 digit);
extern void SendToIndicator2(uint8 digit);
extern void SendToIndicator3(uint8 digit);
static uint8 ConvertTo7SegIndValue(uint8 value);

uint8  voltage; /* Съдържа изчислената стойност на измереното
напрежение */

void interrupt Timer0_ISR(void) _At TIMER0
{
    static uint8  displayCounter = 0;
    static uint8  refreshCounter = 0;
    static uint8  value = 0;

    displayCounter++;
    refreshCounter++;

    if (100 == refreshCounter) /* 0.5 секунди ? */
    { /* Да */
        value = voltage; /* Копиране на последната изчислена
стойност */
        refreshCounter = 0;
    }

    switch (displayCounter)
    {
    case 1:
        SendToIndicator1(value % 10);
        break;

    case 2:
        SendToIndicator2((value / 10) % 10);
        break;

    case 3:

```

```

        SendToIndicator3((value / 100) % 10);
        displayCounter = 0;
        break;
    }
}

void main(void)
{
    PortConfig();
    InterruptConfig();
    TimerConfig();
    AdcConfig();

    while (1)
    {
        uint16 adcResult; /* Съхранява резултата от измерването */

        AdcStart(); /* Стартиране на преобразуването */
        while (1 == (ADCCTL0 & (1 << 7))); /* Изчакване на завършване
на преобразуването */
        adcResult = (((uint16)ADCD_H << 2) | (ADCD_L >> 6)); /*
Копиране на резултата */
        _di(); /* Забраняване на прекъсванията */
        voltage = ((adcResult * Q) * 100); /* Изчисляване на
стойността до формат D.DD */
        _ei(); /* Разрешаване на прекъсванията */
    }
}

void PortConfig(void)
{
    PBAF |= (1 << 2); /* Разрешаване на алтернативната функция
на PB2 */
    PCAFS1 |= 1 << 2; /* PB2 работи като аналогов вход */

    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PBDD &= ~((1 << 3) | (1 << 4) | (1 << 5));

    PCOUT = 0x00;
    PCDD = 0x00;
}

void AdcConfig(void)
{
    ADCCTL0 = 0x32; /* 1. VREFIN е източника на опорно напрежение */
                  /* 2. Разрешаване на АЦП-то */
                  /* 3. Преобразуване на напрежението на ANA2 */

    ADCCP = 0x01; /* FADC = FSYS/2 = 10MHz */
    ADCSST = 0x14; /* TS = (1/FSYS)*ADCSST<3:0> = (1/20MHz)*20 = 1us
*/

```

```

ADCST = 0x28; /*  $T_H = (1/F_{SYS}) * ADCST < 5:0 > = (1/20MHz) * 40 = 2\mu s$  */
}

void AdcStart(void)
{
    ADCCTL0 |= (1 << 7); /* Стартиране на АЦП-то */
}

void SendToIndicator1(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 3;
}

void SendToIndicator2(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 4;
}

void SendToIndicator3(uint8 digit)
{
    PBOUT &= ~((1 << 3) | (1 << 4) | (1 << 5));
    PCOUT = LED_DIGIT_DP | ConvertTo7SegIndValue(digit);
    PBOUT |= 1 << 5;
}

static uint8 ConvertTo7SegIndValue(uint8 value)
{
    uint8 arr7SegIndValues[] =
    {
        LED_DIGIT_0,
        LED_DIGIT_1,
        LED_DIGIT_2,
        LED_DIGIT_3,
        LED_DIGIT_4,
        LED_DIGIT_5,
        LED_DIGIT_6,
        LED_DIGIT_7,
        LED_DIGIT_8,
        LED_DIGIT_9
    };

    return (value > 9) ? LED_DIGIT_ERR : arr7SegIndValues[value];
}

void InterruptConfig(void)
{

```

```

    /* Разрешаване на прекъсванията от Таймер 0 */
    IRQ0ENL |= 1 << 5;
    IRQ0ENH |= 1 << 5;
    IRQCTL  |= 1 << 7;
}

void TimerConfig(void)
{
    /* Нулиране на таймерните регистри */
    T0H = 0x00;
    T0L = 0x00;

    /* Зареждане на максималната броячна стойност */
    T0RH = 0xC3;
    T0RL = 0x50;

    /*-----*/
    /* - Непрекъснат режим */
    /* - прекъсване при препълване */
    /* - коефициент на деление 2 */
    /* - стартиране на таймера */
    /*-----*/
    T0CTL0 = 0x00;
    T0CTL1 = 0x89;
}

```

Може да използвате волтметър и докато въртите потенциометъра РТЗ да измервате напрежението, подавано на извод РВ2 и да го сравните с това на дисплея.

За упражнение може да реализирате програмата по друг начин, като използвате прекъсването от АЦП-то.

51 Заключение

Общо взето разгледахме как работят повечето периферни модули. Вече сами можете да се справите и с другите части на микроконтролера, като WDT-таймера, генераторния блок и т.н.

Убедихте ли се, че микроконтролерът Z8F083A е лесен за използване ☺? Силно се надявам всеки един от Вас да отговори с ДА.

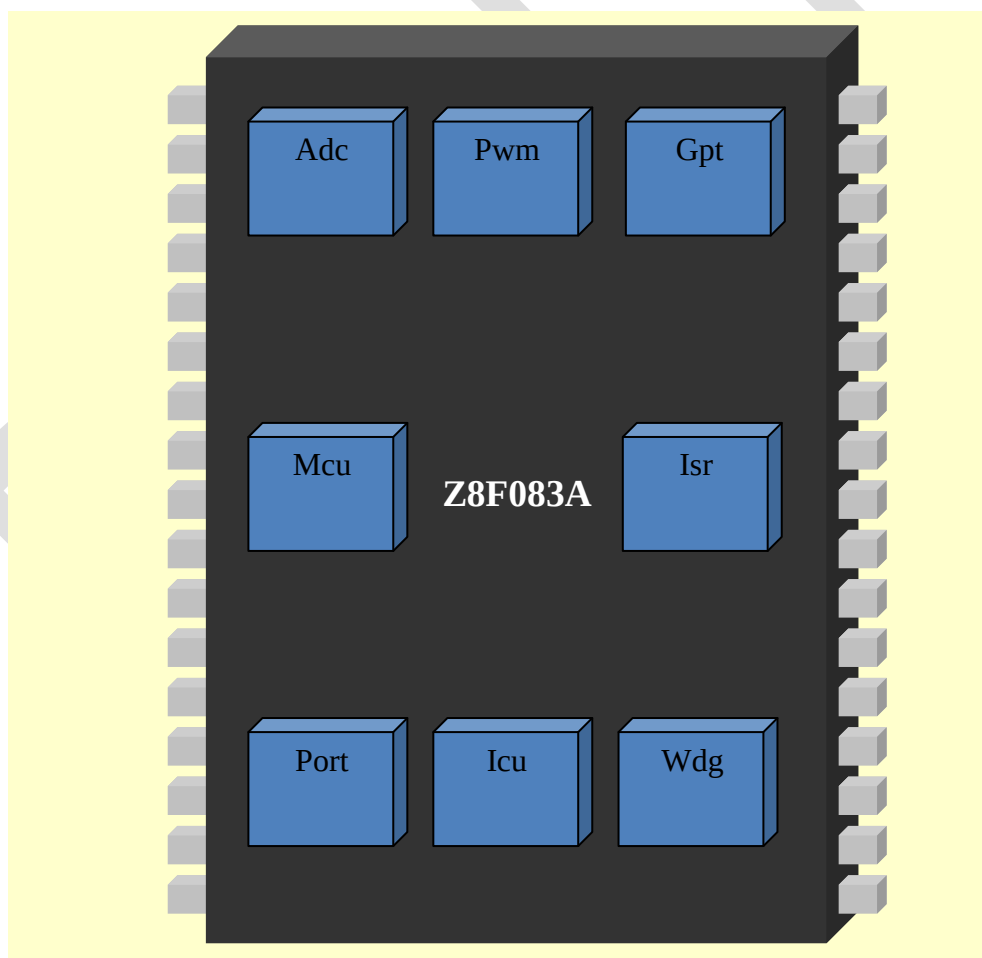
Имайте предвид, че наученото в тази част не е привързано изцяло и само за микроконтролера Z8F083A. Да, основната цел на тази част беше да се запознаете по-отблизо с този микроконтролер, но принципът на управление на 7-сегментен дисплей, четенето на бутон, или измерването на период и коефициент на запълване не зависят от него. Може да използвате тези принципи и да реализирате същите задачи с всеки друг микроконтролер.

Може да използвате платките **Z8Encore!_MB01** и **Z8Encore!_EB01(MB01)** и да си измислите допълнителни упражнения.

DRAFT

Част VI

Описание на драйверите за микроконтролера Z8F083A



DRAFT

Част VII

Практически примери



DRAFT

Пример1. Четене на матрична клавиатура. Реализация на кодова брава.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример2. Управление на буквено-цифров LCD-дисплей (Hitachi HD44780 LCD-контролер).

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример3. Управление на графичен LCD-дисплей (Nokia 5110 LCD-дисплей). Комуникация по SPI шина.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример4. Аналогов термометър с AD22103.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример5. Свързване на външна EEPROM-памет. Комуникация по I2C шина.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример6. Цифров термометър с DS18B20.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример7. Влагомер с DHT11.

Описание на програмата

Описание на хардуера

Описание на софтуера

Пример8. Инфрачервен термометър с MLX90614.

Описание на програмата

Описание на хардуера

Описание на софтуера

◆ Приложение А

Производитель	Микроконтролеры/CPU		
	8-битов	16-битов	32-битов
Analog Devices	8052	-	ARM7™ ARM Cortex™-M3 ADSP-CM4xx
Atmel (вече част от Microchip)	8051 AVR	-	AVR 32 ARM926™ ARM7™ ARM Cortex™-M0+ ARM Cortex™-M3 ARM Cortex™-M4
Cypress	M8C 8051	-	ARM Cortex™-M0 ARM Cortex™-M3
Elmos	EL3.5 EL8(68HC08)	S12 EL16(MSP430)	-
Epson	S1C88	S1C17	S1C33
Fairchild	FCM8531(8051)	-	-
Freescale (вече част от NXP)	HC08 HCS08 RS08	S12 MagniV HCS12 & S12X HC16 Digital Signal Controllers	MAC57Dxxx (ARM® Cortex®-M0+/M4/A5) ColdFire+ Qorivva (Power Architecture®) Kinetis (ARM® Cortex®-M0+/M4/M7)
Holtek	8051 Holtek 8-bits	-	Holtek 32-bits
Infinion	8051	C166	TriCore™ ARM® Cortex™-M0 ARM® Cortex™-M4
Innovasic			
Maxim Integrated	8051	MAXQ20 MAXQ20S	MAXQ30S ARM Cortex™-M3 ARM926EJ-S ARM 922T MIPS 4kSD
Microchip	PIC10 PIC12 PIC16 PIC18	PIC24F PIC24H/E dsPIC30F dsPIC33F/E	PIC32MZ EC PIC32MX
Nuvoton	8051		ARM Cortex™-M0/M4
NXP	8051	-	ARM7 ARM9 ARM Cortex™-M0/M0+/M3/M4 JN51xx wireless
ON Semiconductor	LC87	LC88	-
Renesas	78K0 78K0S H8/300 H8/300L	78K0R RL78 M16C/60 M32C/80 (16/32) R8C H8/300H	RX RZ V850 SuperH RH850 H8/300H-tiny

		H8S	H8SX R32C/100 M32R
ROHM Semiconductor (LAPIS)	U8	U16	ARM®7 TDMI
Silicon Labs	8051	-	ARM Cortex™-M0+/M3/M4
Sonix	SONiX 8-Bit	-	ARM Cortex™-M0
Spansion (вече част от Cypress)	F²MC-8FX	F²MC-16FX F²MC-16LX	FR60 FR81S ARM Cortex™-M0+/M3/M4/R4/R5
STMicroelectronics	STM8	-	ARM Cortex™-M0/M0+/M3/M4/M7
Texas Instrument	-	MSP430(CPU) MSP430X(CPUX) MSP430X(CPUXV2)	TMS320C28x™ ARM Cortex™-M3M4/R4 MSP432
Toshiba	TLC-870	TLCS-900 TLCS-900H TLCS-900L1	TLCS-900H1 ARM Cortex™-M0/M3/M4 ARM926EJ-S™
XMOS			xCORE xCORE-200 xCORE-AUDIO
Zilog	S3 Family Z8051 Z8 Encore!	ZNEO	ZNEO32

Табл. 100 Производители на микроконтролери

◆ Приложение В

Ще бъде готово в следващо издание!!!

DRAFT

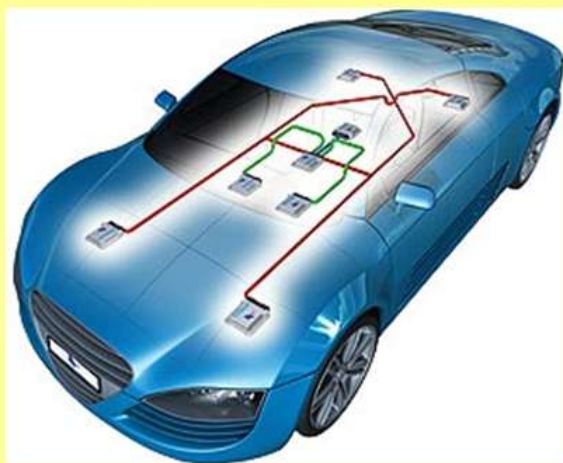
Заклучение

Поздравления! Изминахте дълъг път докато стигнете дотук. Вече може да се наречете програмисти на микроконтролери.

DRAFT

Очаквайте следващата
книга!

Controller Area Network (CAN)



Първо издание

Тази книга е предназначена за всеки електронен инженер, който иска да се запознае с един от най-известните серийни комуникационни интерфейси: **CAN (Controller Area Network)**.

DRAFT

