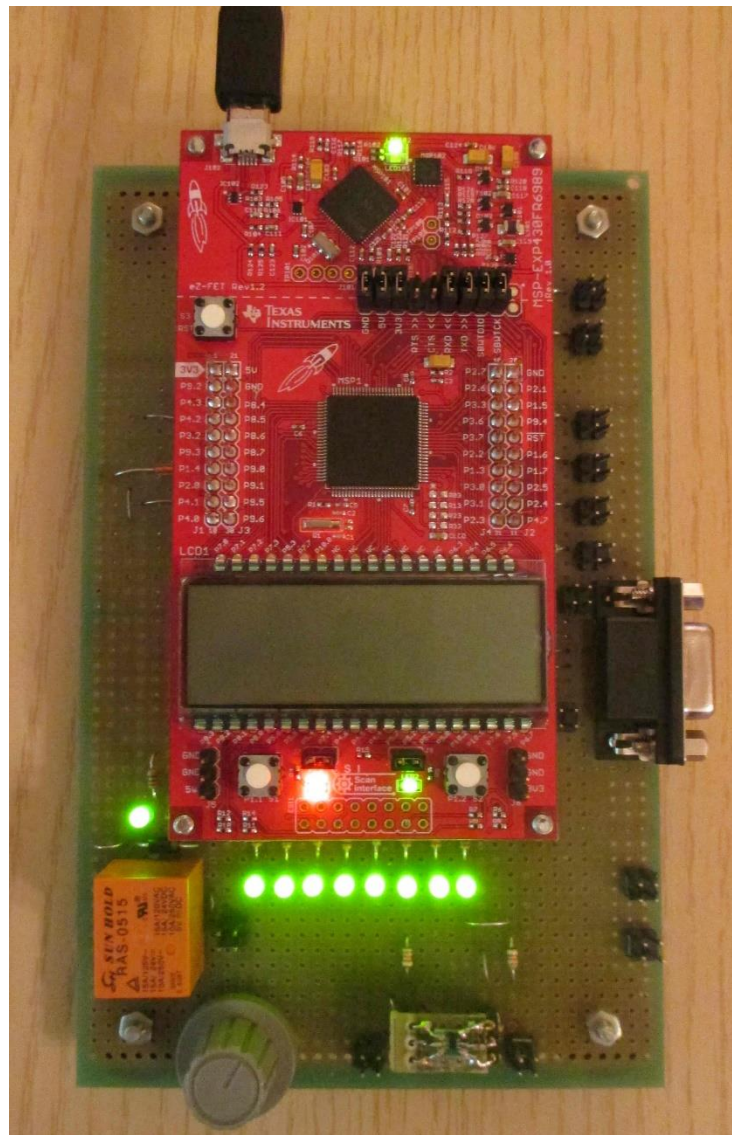


Лабораторни упражнения по Микропроцесорна схемотехника с микроконтролера MSP430FR6989

```
REG_NAME |= BIT_MASK;    //Лог. 1  
REG_NAME &= ~BIT_MASK;  //Лог. 0
```



СЪДЪРЖАНИЕ

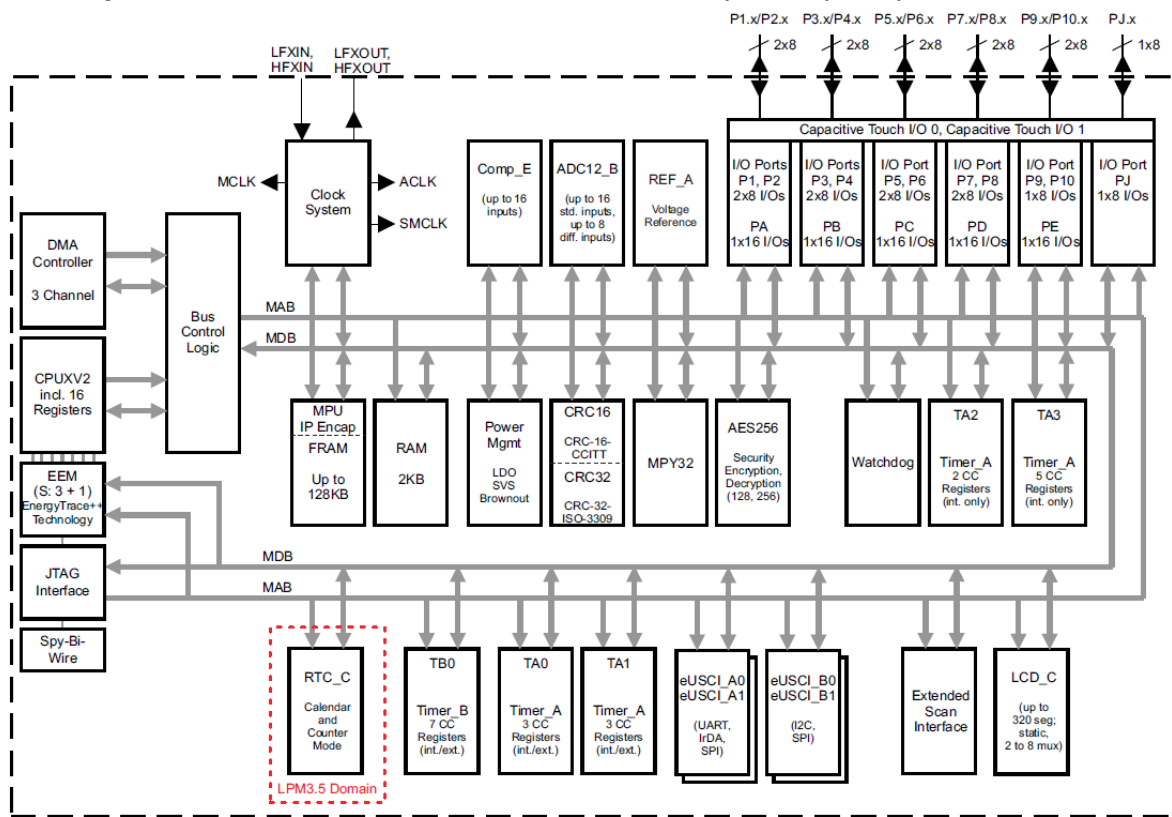
Лабораторно упражнение № 1	3
“Запознаване със средата IAR Embedded Workbench и записване на примерни програми в MSP430FR6989”	
Лабораторно упражнение № 2	17
“Видове адресации и инструкции на микроконтролер MSP430FR6989”	
Лабораторно упражнение № 3	29
“Управление на входно-изходните портове на микроконтролера MSP430FR6989”	
Лабораторно упражнение № 4	38
“Изследване на таймерен модул на микроконтролер MSP430FR6989”	
Лабораторно упражнение № 5	44
“Изследване на сериен интерфейс SPI на микроконтролера MSP430FR6989”	
Лабораторно упражнение № 6	49
“Изследване на сериен интерфейс I ² C на микроконтролера MSP430FR6989”	
Лабораторно упражнение № 7	58
“Аналогово-цифров преобразувател на MSP430FR6989 и комуникационен модул UART (интерфейс RS-232). Използване на printf() API.”	
Лабораторно упражнение № 8	67
“Безжичен LAN (WLAN) модул CC3100 и MSP430FR6989. Създаване на Web сървър.”	
Приложение	81
1. ASCII Таблица.....	81
2. Приоритети на операторите в C.....	82
3. Асемблер на MSP430.....	83
4. Intrinsic функции на MSP430.....	85
5. Разположение на периферията.....	86

Лабораторно упражнение №1

“Запознаване със средата IAR Embedded Workbench и записване на примерна програма в MSP430FR6989”

1.1. Въведение. MSP430FR6989(IPZ) е 16-битов, 100-изведен RISC микроконтролер на фирмата Texas Instruments. Той работи с максимална тактова честота от 16 MHz и има 2 kB SRAM памет за данни. Програмната памет е 128 kB и е реализирана по технология Ferroelectric RAM (FRAM), докато по-старите контролери използват Flash памет. Предимствата на FRAM спрямо Flash са: 100 000 000 000 000 цикъла за запис (срещу 100 000 при Flash), около 250 пъти по-малка консумация, около 100 пъти по-кратки времена за достъп и около 10 пъти по-ниски нива на захранващото напрежение при запис (1,5 V). Големият брой цикли за запис позволява тя да бъде използвана като програмна и даннова памет едновременно.

На **фиг. 1.1** е показана блокова схема на микроконтролера.

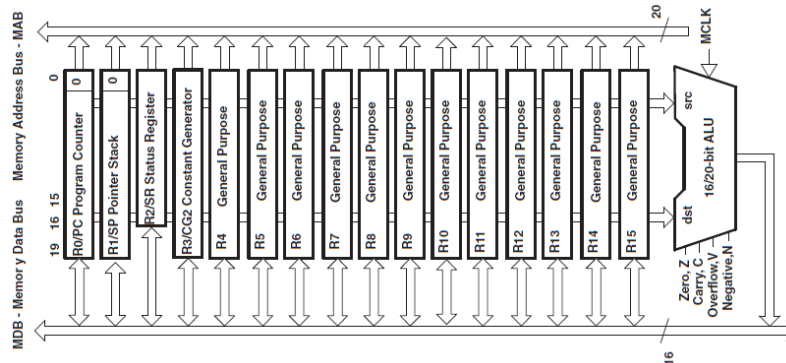


Фиг. 1.1 – Блокова схема на MSP430FR6989

Отделните модули са:

- CPU – 16-мегагерцов микропроцесор с 20-битова магистрала за адреси и 16-битова магистрала за данни (**фиг. 1.2**). Използват се 27 инструкции (+ 24 емулирани инструкции). Разполага с:
 - 12 регистъра с общо предназначение (R4 ÷ R15).
 - програмен брояч (PC, разположен в R0).
 - указател на стека (SP, разположен в R1).
 - регистър на състоянието (SR, разположен в R2).

- регистър за генериране на константи (CG, разположен в R3) – генерира 6 често използвани константи в зависимост от адресацията на изпълняваната инструкция. Тези константи могат да служат като „операнд по подразбиране” и така инструкция с един операнд може да се превърне в инструкция с два операнда (емулирана инструкция).



Фиг. 1.2 – Блокова схема на микропроцесора, използван от микроконтролера MSP430FR6989

- JTAG – интерфейс за програмиране на чипа и четене на вътрешните регистри.
- EEM – Embedded Emulation Module – хардуерен модул за реализация на “breakpoints”. Това са точки на прекъсване, в които изпълнението на програмата спира и чрез JTAG може да се прочете състоянието на даден регистър. Накратко – използва се за улеснение при дебъгването на програмата.
- System Clock – модул за генериране на тактови честоти. Генерират се шест тактови сигнала, използвани от различни периферни модули на чипа:
 - MCLK (Main Clock) – честота на микропроцесора
 - ACLK (Auxiliary Clock) – честота за периферните модули
 - SMCLK (Sub-Main Clock) – честота за периферните модули
 - VLOCLK (Very-LOW-Power Clock) – честота за периферните модули
 - LFXCLK (Low-Frequency Clock) – честота за периферните модули
 - MODCLK (Low-power Clock) – честота за периферните модули
- FRAM – програмна памет.
- RAM – даннова памет.
- Boot ROM – памет за съхранение на bootloader. Bootloader е специална програма, която помага за трансфер на потребителската програма (написана от програмиста) от персоналния компютър в програмната памет на микроконтролера.
- Power Management & SVS (Supply Voltage Supervisor) – модул за изработване на необходимите захранващи напрежения.
- SYS Watchdog – таймер, отброяващ интервал от време, при изтичането на което се подава сигнал за рестартиране на контролера. Програмистът трябва да заложи в програмата периодичното нулиране на този брояч. Така, ако някъде в програмата се получи „замръзване”, то броячът няма да се нулира и микроконтролерът ще се рестартира. В настоящите лабораторни упражнения този брояч е изключен.

- REF – еталонно напрежение, използвано от някое периферно устройство (ADC или компаратор в случая).
- I/O Ports – входно-изходни портове. Това са регистри, достъпни за потребителя и свързани с изводите на микроконтролера. В литературата се отбелязват още като GPIO – General Purpose Input Output pins. С тях може да се управляват външни устройства/схеми, приемащи CMOS нива.
- DMA (Direct Memory Access controller) – контролер за автоматично прехвърляне на данни (без намесата на микропроцесора) от един адрес на друг.
- Comp_E – аналогов компаратор.
- ADC12_B – 16-канално 12-битово АЦП.
- UART, IrDA, SPI, I2C – сериини синхронни (SPI, I2C) и асинхронни (UART, IrDA) интерфейси.
- CRC16 (Cyclic Redundancy Check) – модул за генериране на checksum.
- RTC_C – часовник за реално време.
- Timer_A, Timer_B – 16-битови таймери.
- MPY32 – 32-битов хардуерен умножител.
- AES256 - модул за хардуерно шифриране/дешифриране на информация.
- LCD_C - модул за управление на сегментни LCD дисплей.
- ESI (Extended Scan Interface) - модул за четене на оптични/магнитни ротационни енкодери и др.

Фърмуер (от англ. firmware) - софтуер, който се зарежда в постоянната памет на микроконтролера. Терминът се използва, за да не се бърка с приложния софтуер (от англ. application software), който се зарежда на персонален компютър. Микроконтролерните системи към днешна дата често се управляват от персонален компютър, което налага разграничаването фърмуер-приложен софтуер. В настоящите упражнения се разглежда само разработката на фърмуер.

Хардуерен дебъгер (от англ. hardware debugger) – устройство, с помощта на което развойната среда извършва следните действия (често обобщавани с израза дебъгване):

- зарежда фърмуера в паметта на микроконтролера;
- стартира изпълнението на фърмуера;
- изпълнява програмата стъпка по стъпка, т.е. асемблерна инструкция след асемблерна инструкция, или ред след ред от C програма;
- чете от или записва във всички регистри на микроконтролера;
- следи за точки на прекъсване (места в програмата, на които микропроцесорът спира изпълнението на инструкции);
- следи за регистри, които са си променили съдържанието.

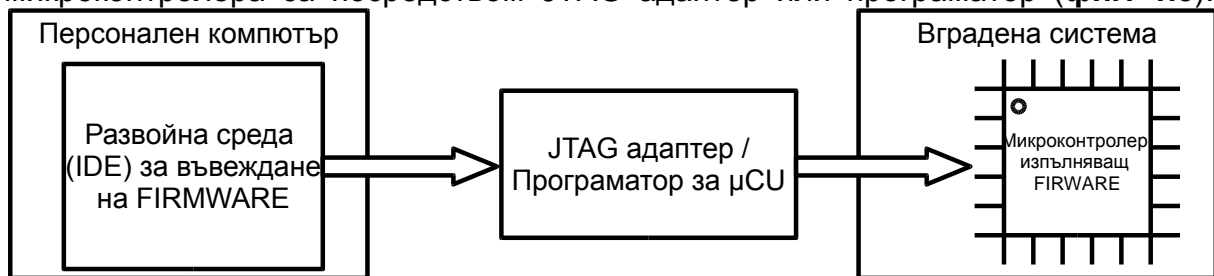
За да може дебъгерът да изпълнява всички тези функции, микроконтролерът трябва да съдържа хардуерни модули, които да изпълняват командите на дебъгера и да контролират микропроцесора. Такива са JTAG, SWD, Breakpoint, Watchpoint и др. модули. Повечето микроконтролери притежават поне JTAG или SWD.

Примери за хардуерни дебъгери са Keil ULINK, Texas Instruments ICDI, LPC Link и др. Някои от тях са реализирани като отделни устройства, други – включени в демо платка и неделими от нея.

Софтуерен дебъгер(от англ. software debugger) – програма, която следи действията на програмиста в развойната среда (натискане на бутони) и в зависимост от тях изпраща команди на хардуерния дебъгер. Той от своя страна изпраща команди на JTAG модула, който извършва желаното действие (например прочети даден регистър или спри микропроцесора).

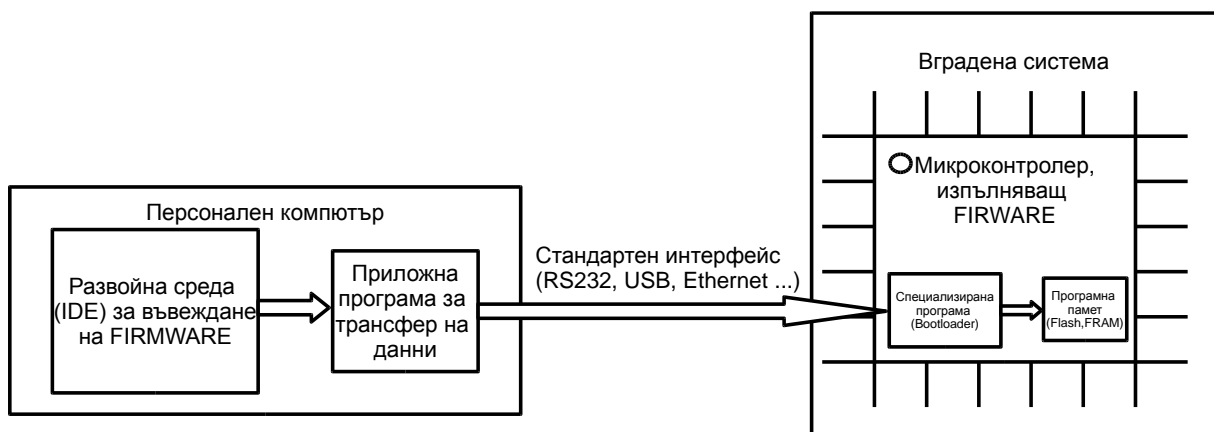
Пример за софтуерен дебъгер е програмата с отворен код GDB, която има версии за различни фамилии микропроцесори от Intel до ARM.

Използване на IAR Embedded Workbench IDE. Програмите за микроконтролери се разработват с помощта на развойни среди (IDE – Integrated Development Environment), работещи на персонални компютри. IDE представляват пакет от програмни продукти, обединени в един потребителски интерфейс (прозорец). Повечето IDE включват текстови редактор, компилатор, линкер, асемблер, дебъгер и програма, генерираща изпълнимият код, често наричан фърмуер. Хардуерните методи за зареждане на фърмуера в микроконтролера са посредством JTAG адаптер или програматор (**фиг. 1.3**).

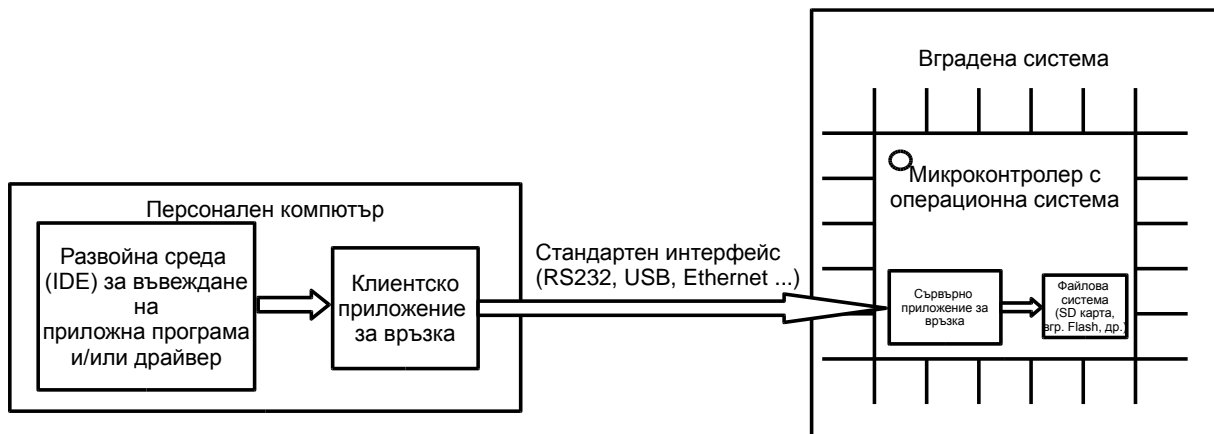


Фиг. 1.3 – Програмиране на микроконтролер чрез JTAG адаптер или програматор.

Съществуват и софтуерни методи за програмиране– чрез предварително зареден bootloader (**Фиг. 1.4 а**) или чрез копиране на файл във файловата система на микроконтролерно устройство с операционна система.



Фиг. 1.4 а)– Програмиране чрез bootloader.

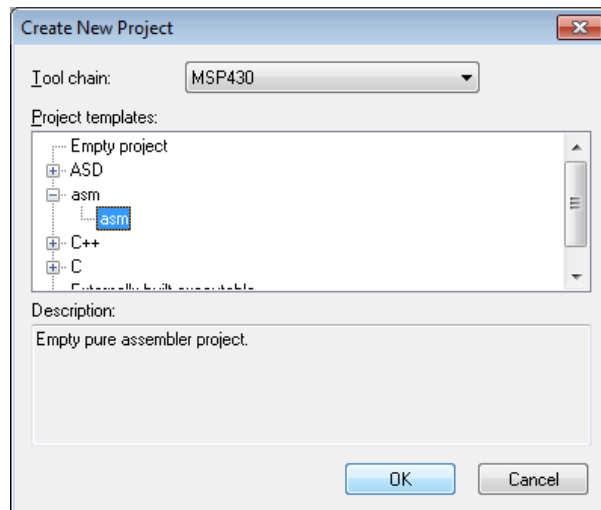


Фиг. 1.4 б) – Програмиране на устройство с операционна система.

В настоящото лабораторно упражнение ще се използва методът, показан на **фиг. 1.3**. Развойната платка е MSP-EXP430FR6989 (Experimenter's Board) на фирмата Texas Instruments и включва един микроконтролер MSP430FR6989, както и допълнителна периферия. Средата за програмиране е на фирмата IAR. Тя може да се стартира като се щракне два пъти върху иконката IAR Embedded Workbench, след което ще се отвори нейният прозорец.

За програмиране на *асемблер* трябва да се премине през следните стъпки:

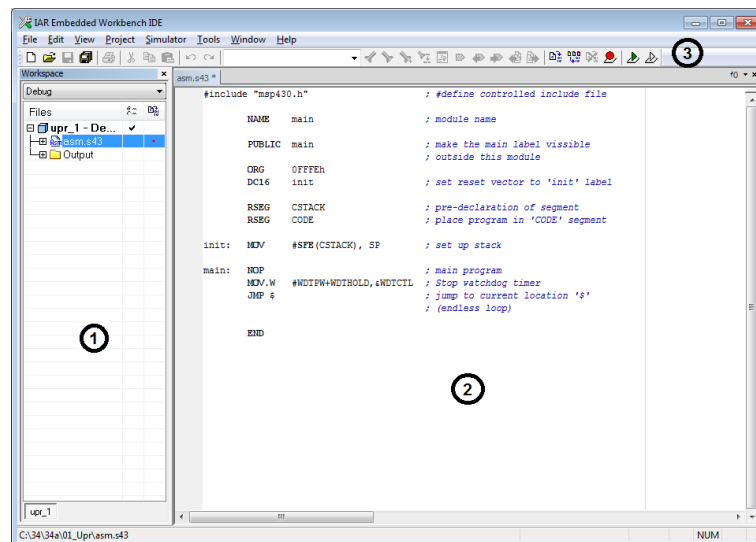
- Избира се File → New → Workspace
- Project → Create New Project...
- Tool chain → MSP430
- Project Templates → щраква се върху +asm → избира се asm
- Щраква се бутон ОК (**фиг.1.5**).



Фиг. 1.5

- Указва се мястото, където ще се намира проекта.
- Дава се име на проекта и се натиска Save.
- Избира се моделът на микроконтролера от Project → Options → Category: *General Options* → tab Target → Device → избира се MSP430FR6989.
- В Category: *Debugger* → tab Setup → Driver → FET Debugger.

- Избира се файлът asm.s43 от дървото на проекта вляво на екрана (позиция 1, **фиг. 1.6**).
- Програмата на асемблер се въвежда в текстовия редактор вдясно (позиция 2, **фиг. 1.6**). За целта трябва да се изтрие редът за безкраен цикъл JMP \$ и на негово място да се въведат новите инструкции.
- Програмата се асемблира с бутона Compile от toolbar-ът, показан на позиция 3 от **фиг. 1.6**. При първото натискане на този бутон средата IAR ще поиска въвеждане име на workspace (съдържа един или повече проекти, които имат някаква логическа връзка между тях). Указва се име по желание.
- Натиска се бутон Make. Ако програмата съдържа повече от един файл, то това е стъпката, в която останалите файлове ще се компилират/асемблират и свържат (link-нат) в един обектов файл. След това ще се генерира и изпълнимият файл, готов за зареждане в микроконтролера.




Фиг. 1.6


- Микроконтролерът се програмира с бутон Download and Debug. Преди натискането на този бутон макетът трябва да е включен с USB кабел към компютъра. При успешно зареждане на програмата ще се отвори нов toolbar с бутони, осигуряващи дебъг на програмата, докато се изпълнява в микроконтролера (**фиг. 1.7**).





Фиг. 1.7


Описанието на бутоните важи и при дебъгване на C/C++ програми.


Бутонът  служи за рестартиране на микроконтролера.


Бутонът  служи за спиране изпълнението на програмата.


Бутонът  служи за изпълняване на следваща инструкция (извикване на C/C++ функция) без да се влиза в подпрограми (в самата C/C++ функция).

Бутонът  служи за изпълняване на следваща инструкция (извикване на C/C++ функция) с влизане в подпрограми (в самата C/C++ функция).

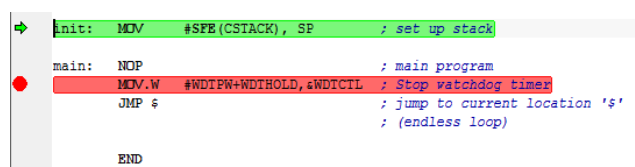
Бутонът  служи за излизане от C/C++ функцията, в която се намираме.

Бутонът  служи за придвижване изпълнението на програмата от настоящата инструкция (операция) до инструкция (операция), която е избрана чрез курсора на мишката.

Бутонът  служи за изпълнение на програмата от настоящото положение до края на програмата. Ако има поставени точки на прекъсване (breakpoints), то изпълнението ще спре при тях.

Бутонът  служи за изпълнение на програмата стъпка по стъпка, дори когато се изпълняват цикли с празно тяло (иначе дебъгерът ги прескача).

Поставяне на точка на прекъсване – щраква се с деснен бутон на реда, на който искаме изпълнението на програмата да спре. Избира се Toggle Breakpoint (Code). Срещу избрания ред ще се появи червена точка (фиг. 1.8).

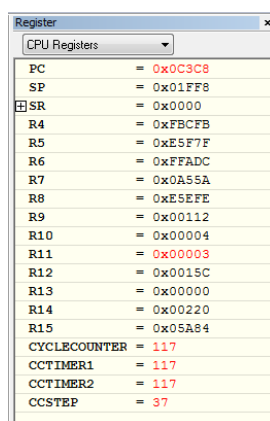


```
init:  MOV    #SPB(CSTACK), SP      ; set up stack
main:  NOP
      MOV.W  #WDTPW+WDTHOLD,&WDTCIL ; Stop watchdog timer
      JMP   $                       ; jump to current location '$'
                                     ; (endless loop)
      END
```

фиг. 1.8

Докато изпълнението на програмата е спряно може да се наблюдават регистрите на микроконтролера благодарение на JTAG адаптера. В дебъг режим разнообразната информация може да се визуализира от менюто:

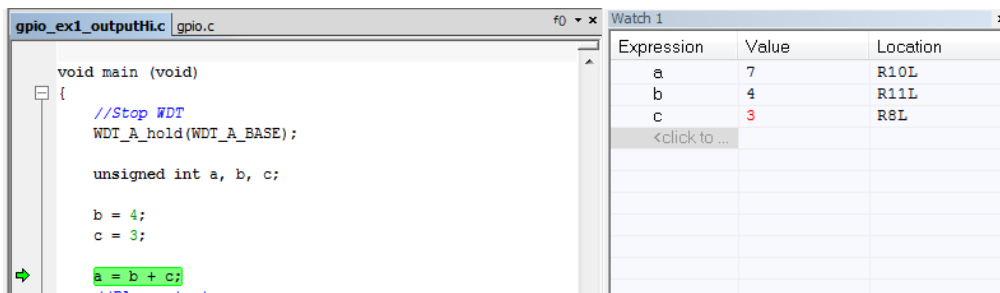
- View → Register – за наблюдение на всички регистри в микроконтролера. От падащото меню Register се избира периферен модул или микропроцесор. На **фиг. 1.9** са дадени моментните стойности на регистрите на микропроцесора от **фиг. 1.2**.



Register	Value
PC	= 0x0C3C8
SP	= 0x01FF8
SR	= 0x0000
R4	= 0xFBCFB
R5	= 0xE5F7F
R6	= 0xFFADC
R7	= 0x0A55A
R8	= 0xE5EFE
R9	= 0x00112
R10	= 0x00004
R11	= 0x00003
R12	= 0x0015C
R13	= 0x00000
R14	= 0x00220
R15	= 0x05A84
CYCLECOUNTER	= 117
CCTIMER1	= 117
CCTIMER2	= 117
CCSTEP	= 37

Фиг. 1.9

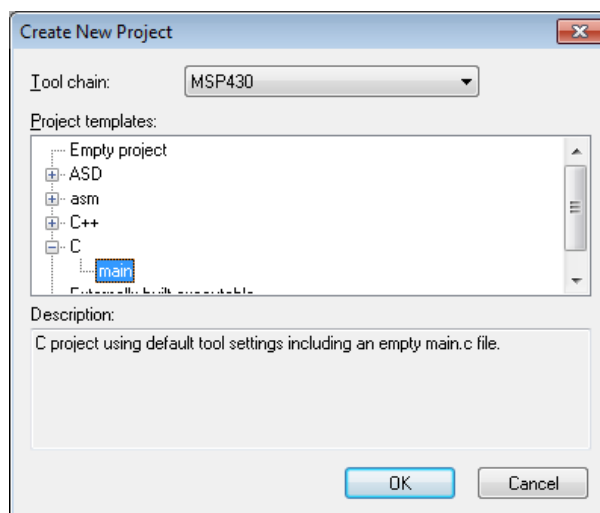
- View → Watch → Watch 1 → в полето Expression се записва имена променливата, която искаме да инспектираме. На **фиг. 1.10** е показана програма на C и наблюдаване на 3 променливиот целочислен тип a, b, c.



Фиг. 1.10

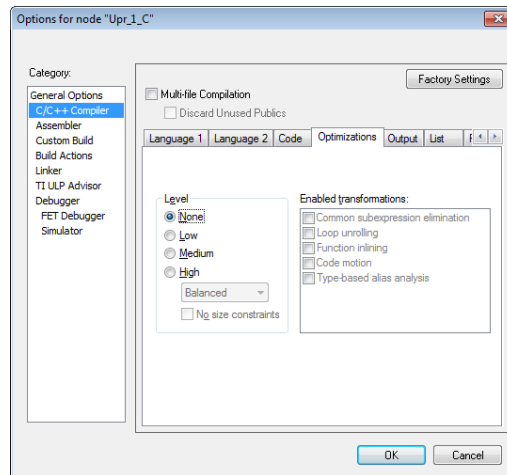
За програмиране на C без използване на библиотеки трябва да се премине през следните стъпки:

- Избира се File → New → Workspace
- Project → Create New Project...
- Tool chain → MSP430
- Project Templates → щраква се върху +C → избира се main
- Щраква се бутон ОК (фиг.1.11).



Фиг. 1.11

- Указва се мястото, където ще се намира проекта.
- Дава се име на проекта и се натиска Save.
- Избира се номерът на микроконтролера от Project → Options → Category: *General Options* → таб Target → Device → избира се MSP430FR6989.
- В Category: *Debugger* → таб Setup → Driver → Избира се FET Debugger.
- В Category: *C/C++ Compiler* → таб Optimizations → Level → None. Оптимизациите се изключват за лабораторните упражнения с цел по-ясно показване работата на микроконтролера. Натиска се ОК (фиг. 1.12).



Фиг. 1.12

- От дървото на проекта се избира файлът main.c и програмата на C се въвежда след реда WDTCTL = WDTPW + WDTNHOLD;
- Всички стъпки като компилирането, програмирането и дебъгването, описани за асемблер, също са приложими тук.

Компанията Texas Instruments предлага готови библиотеки на C, помагачи при инициализацията и използването на отделните периферни модули на MSP430FR6989. Благодарение на тях всяка опция се програмира чрез API функции, а не чрез директен регистров достъп. Последното означава, че нивото на абстракция в програмирането се увеличава, т.е. отдалечаваме се от програмирането на асемблер. Библиотеките са предоставени за свободно ползване и обединени в един пакет, наречен MSP430Ware. Редица микроконтролери от фамилията MSP430 може да ги използват. По подразбиране те се инсталират в директория driverlib_X_XX_XX_XX, където са включени и много полезни примери за първоначални тестове при разработката на фърмуер.

По-долу е разгледана програма, която спира watchdog таймера, инициализира порт J и установява изводи PJ.0, PJ.1, PJ.2 и PJ.3 във високо ниво. Представени са варианти на асемблер, на C без библиотеки и на C с използване на библиотеките MSP430Ware (start-up код не е показан).

Програма на асемблер:

```
main:  MOV.W      #WDTPW+WDTHOLD,&WDTCTL
      BIS.W      #0Fh, &PJDIR
      BIS.W      #0Fh, &PJOUT
      JMP $
      END
```

Програма на C без използване на библиотеки (директен регистров достъп)

```
void main( void )
{
    WDTCTL = WDTPW + WDTNHOLD;

    PJDIR |= 0x0F;
```

```
PJOUT = 0x0F;  
}
```

Програма на C с използване на библиотеките MSP430Ware

```
void main (void)  
{  
    WDT_A_hold(WDT_A_BASE);  
  
    GPIO_setAsOutputPin(GPIO_PORT_PJ, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3);  
    GPIO_setOutputHighOnPin(GPIO_PORT_PJ, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3);  
}
```

За програмиране на C с използване на библиотеки трябва да се премине през следните стъпки:

- Създава се C проект.
- Въвежда се програма на C, която използва API функции.
- API функциите, които могат да се използват са документирани във файла:

06_MSP430FR5xx_6xx_DriverLib_Users_Guide-X_XX_XX_XX.pdf

където е даден списък с имената на тези функции и параметрите, които те приемат.

- Project → Options → C/C++ Compiler → Preprocessor → Additional Include directories: (one per line) → натиска се бутон „...“ (Browse) → натиска се полето <Click to add> и се указва пътя до директорията със сорс файлове на библиотеката. Например:

MSHT_LAB \ 02_driverlib \ MSP430FR5xx_6xx

- От дървото на проекта се щраква с десен бутон върху top-level директорията, избира се Add → Add Group и се указва произволно име (например driverlib).
- Избира се новосъздадената група driverlib от дървото на проекта и с десен бутон Add → Add Files... → указва се пътя до сорс файловете на MSP430Ware библиотеката. Например:

MSHT_LAB \ 02_driverlib \ MSP430FR5xx_6xx

след което се избират всички файлове с CTRL + A → OK.

- Във файла main.c се включва #include директивата:

```
#include "inc/hw_memmap.h"
```

и допълнителните хедърни файлове за всяка една използвана периферия, например ако се използват Watchdog таймера и GPIO изводите:

```
#include "gpio.h"  
#include "wdt_a.h"
```

- Ако е включена директивата #include "io430.h", то тя трябва да се изтрие.

1.2. Задачи за изпълнение.

1.2.1. Да се създаде нов проект на Desktop-а в отделна директория с име 01_2_1_Lab, да се копира програмата на асемблер, която последователно включва само един от светодиодите LED0 ÷ 7 (ефект „бягаща точка”). Реализирано е софтуерно закъснение между всяко преместване на позицията, за да може да се наблюдава ефекта.

1.2.2. Да се създаде нов проект на Desktop-а в отделна директория с име 01_2_2_Lab, да се копира програмата на C, която извършва същите действия като програмата в предходната точка. Да се изпълни програмата стъпка по стъпка. Да се наблюдават регистрите на микропроцесора и изходните портове (P8 и P9). Да се постави точка на прекъсване. Да се наблюдава изменението на променливата i чрез View → Watch → Watch 1 → въвежда се i .

1.2.3. Да се създаде нов проект на Desktop-а в отделна директория с име 01_2_3_Lab, да се копира програмата на C, която извършва същите действия като програмата в предходната точка, но сега управлението на порта става чрез API функции от библиотеката MSP430Ware. Да се изпълни програмата стъпка по стъпка. По време на дебъгване да се влезе в сорс кода на библиотеката (за целта да се използва бутон „Step into”).

1.2.4. Да се покаже съответствието между инструкциите на асемблерната програма (от задача 1.2.1.) и операциите на C програмата (от задача 1.2.2. и 1.2.3.).

1.2.5. Според вас коя програма е с най-добро бързодействие?

Задача 1.2.1.

```

#include      "msp430.h"                ;#define controlled include file
NAME main                                     ; module name
PUBLIC main                                  ;make the main label visible
                                           ;outside this module

ORG 0FFFEh                                ;set reset vector to 'init' label
DC16 init                                   ;pre-declaration of segment
RSEG CSTACK                               ;place program in 'CODE'
RSEG CODE                                  ;segment

init:      MOV #SFE(CSTACK), SP          ;set up stack
main:      NOP                          ;main program
           MOV.W #WDTPW+WDTHOLD,&WDTCTL ;Stop watchdog timer

           BIC.W #LOCKLPM5,&PM5CTL0     ;Изключи високо-
                                           ;импедансното състояние на
                                           ;входно-изходните изводи

           BIS.B #0xF0, &P8DIR          ;Конфигуриране на изводите
           BIS.B #0x63, &P9DIR          ;като изходи

           BIC.B #0xF0, &P8OUT          ;Инициализиране на изходи-
           BIC.B #0x63, &P9OUT          ;те в логическа 0

L1         BIS.B #0x10, &P8OUT          ;Включване на D0
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x10, &P8OUT          ;Изключване на D0
           BIS.B #0x20, &P8OUT          ;Включване на D1
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x20, &P8OUT          ;Изключване на D1
           BIS.B #0x40, &P8OUT          ;Включване на D2
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x40, &P8OUT          ;Изключване на D2
           BIS.B #0x80, &P8OUT          ;Включване на D3
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x80, &P8OUT          ;Изключване на D3
           BIS.B #0x01, &P9OUT          ;Включване на D4
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x01, &P9OUT          ;Изключване на D4
           BIS.B #0x02, &P9OUT          ;Включване на D5
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x02, &P9OUT          ;Изключване на D5
           BIS.B #0x20, &P9OUT          ;Включване на D6
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x20, &P9OUT          ;Изключване на D6
           BIS.B #0x40, &P9OUT          ;Включване на D7
           CALL #Delay                  ;Софтуерно закъснение
           BIC.B #0x40, &P9OUT          ;Изключване на D7

           CALL L1                      ;Започни отначало

Delay:     MOV.W #65535, R15             ;Зареди регист. R15 с 65535
L2         DEC.W R15                    ;Намали R15 с 1
           JNZ L2                       ; Ако R15 не е 0, иди в L2
           RET                           ;Ако R15 е 0, излез

END                                              ;Край на асемблерния файл

```

Задача 1.2.2.

```
#include "io430.h"

void delay()
{
    volatile long i;
    for(i = 0; i < 65535; i++){ }
}

void main( void )
{
    WDTCTL = WDTPW + WDTHOLD;

    PM5CTL0 &= ~LOCKLPM5;

    P8DIR |= 0xF0;
    P9DIR |= 0x63;

    P8OUT &= ~0xF0;
    P9OUT &= ~0x63;

    while(1)
    {
        P8OUT |= 0x10;
        delay();
        P8OUT &= ~0x10;
        P8OUT |= 0x20;
        delay();
        P8OUT &= ~0x20;
        P8OUT |= 0x40;
        delay();
        P8OUT &= ~0x40;
        P8OUT |= 0x80;
        delay();
        P8OUT &= ~0x80;
        P9OUT |= 0x01;
        delay();
        P9OUT &= ~0x01;
        P9OUT |= 0x02;
        delay();
        P9OUT &= ~0x02;
        P9OUT |= 0x20;
        delay();
        P9OUT &= ~0x20;
        P9OUT |= 0x40;
        delay();
        P9OUT &= ~0x40;
    }
}
```

Задача 1.2.3.

```
#include "inc/hw_memmap.h"
#include "wdt_a.h"
#include "gpio.h"
#include "pmm.h"

void delay()
{
    volatile long i;
    for(i = 0; i < 65535; i++){ }
}

int main( void )
{
    WDT_A_hold(WDT_A_BASE);

    PMM_unlockLPM5();

    GPIO_setAsOutputPin(GPIO_PORT_P8, GPIO_PIN4 | GPIO_PIN5 | GPIO_PIN6 | GPIO_PIN7);
    GPIO_setAsOutputPin(GPIO_PORT_P9, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN5 | GPIO_PIN6);

    GPIO_setOutputLowOnPin(GPIO_PORT_P8, GPIO_PIN4 | GPIO_PIN5 | GPIO_PIN6 | GPIO_PIN7);
    GPIO_setOutputLowOnPin(GPIO_PORT_P9, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN5 | GPIO_PIN6);

    while(1){
        GPIO_setOutputHighOnPin(GPIO_PORT_P8, GPIO_PIN4);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P8, GPIO_PIN4);
        GPIO_setOutputHighOnPin(GPIO_PORT_P8, GPIO_PIN5);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P8, GPIO_PIN5);
        GPIO_setOutputHighOnPin(GPIO_PORT_P8, GPIO_PIN6);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P8, GPIO_PIN6);
        GPIO_setOutputHighOnPin(GPIO_PORT_P8, GPIO_PIN7);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P8, GPIO_PIN7);
        GPIO_setOutputHighOnPin(GPIO_PORT_P9, GPIO_PIN0);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P9, GPIO_PIN0);
        GPIO_setOutputHighOnPin(GPIO_PORT_P9, GPIO_PIN1);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P9, GPIO_PIN1);
        GPIO_setOutputHighOnPin(GPIO_PORT_P9, GPIO_PIN5);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P9, GPIO_PIN5);
        GPIO_setOutputHighOnPin(GPIO_PORT_P9, GPIO_PIN6);
        delay();
        GPIO_setOutputLowOnPin(GPIO_PORT_P9, GPIO_PIN6);
    }
}
```


Лабораторно упражнение №2

“Видове адресации и инструкции на микроконтролер MSP430FR6989”

2.1. Въведение. Всяка една програма на C за микроконтролери се преобразува в програма на Асемблер, а след това всяка една асемблерна инструкция се преобразува в двоичен код, който се зарежда в микроконтролера. Познаването на Асемблер за поне един микроконтролер е от особено значение в програмирането на вградени системи. Макар Асемблерите да са различни за различните микроконтролери, между тях има много общи черти. Ето и някои примери, в които познаването на Асемблер е от значение:

а) Понякога програмистът не може да предвиди как точно един ред от C програмата ще се „преведе“ на Асемблер. Това води до грешки, чието отстраняване (debug) изисква преглед на „преведената“ програма. Например празен for-цикъл от вида `for(i = 0; i < 20; i++){ }` може да се премахне от C-компилятора, който да счете тази конструкция за ненужна. Реално обаче програмистът може да я използва като софтуерно закъснение. Тогава единствено чрез поглеждане в „преведената“ на Асемблер програма може да се види липсата на този код.

б) Оптимизация от гледна точка на бързодействие и/или размер на програмата може да се постигне чрез програмиране на Асемблер. Микроконтролерите обикновено разполагат с минимален хардуерен ресурс (в сравнение с настолните компютри) и комбинирането C/Асемблер е често срещано. Пример → инициализиращ код на Linux за вградени системи.

2.2. Видове инструкции. Всяка една инструкция се състои от код на операцията и операнди. Фирмата Texas Instruments дефинира четири вида инструкции на микропроцесора MSP430:

Инструкция с два операнда (наричани още **Формат I инструкции**).

Пример: `add R3, R4` → Събери стойностите на регистри R3 и R4 и запиши резултата в R4 върху старата стойност.

Инструкция с един операнд (наричани още **Формат II инструкции**).

Пример: `RRA R3` → премести всички битове на числото в R3 с една позиция надясно като най-младшия бит запиши в STATUS регистъра на позиция Carry).

Инструкции за преход (наричани още **Формат III инструкции**).

Пример: `jmp 0x1000` → иди на абсолютен адрес 0x1000.

Инструкции, които не попадат в никоя категория (наричани още **Разни инструкции**)

Пример: Хендлер на прекъсване:

`add R3, R4`

`rra R3`

`reti`

→ Излез от хендлер на прекъсване

Прекъсванията и хендлерите на прекъсвания са обяснени в лабораторното упражнение за входно/изходните портове.

Формат I и II инструкциите имат наставка .b или .w, които означават работа със съответно 8-битови и 16-битови операнди. Така например инструкцията add може да се запише като add.b R3, R4 или add.w R3, R4. Ако не се укаже наставка, асемблерът по подразбиране слага .w наставка. Използването на наставки произлиза от факта, че част от периферните модули са разположени на адреси $0x00 \div 0xFF$, а друга част на $0x100 \div 0x1FF$. Първите изискват работа с „b” инструкции, а вторите – работа с „w” инструкции. Не може да се използва .w на периферия в обхвата $0x00 \div 0xFF$, както и .b на периферия в $0x100 \div 0x1FF$.

Микропроцесорът MSP430 има 16-битова адресна шина. Това води до ограничение на адресираната памет от 0 до 65535 адреса (или $0 \div 64\text{kB}$). За да може да се произведат микроконтролери с повече памет и периферия (периферията се адресира като памет), от Тексас Инструментс са проектирали разширен вариант на MSP430 и това е MSP430X. В него адресна шина е 20-битова, което позволява $0 \div 2^{20} = 0 \div 1048576$ (или $0 \div 1\text{ MB}$). MSP430FR5739 е с MSP430X микропроцесор. За да се адресира модул или памет, разположен на адрес по-голям от $0xFFFF$ се използва наставката A след мнемониката на инструкцията, например BR $0xFFFF$ (16-bit) и BRA $0x10000$ (20-bit).

2.3. Дисасемблер (Асемблерен листинг). Дисасемблер (disassembly), или още асемблерен листинг, е изходен файл, съдържащ програма на Асемблер, заедно с машинния код, съответстващ на всяка една инструкция и адреса, от който тя се изпълнява. На **фиг. 2.1** е показан дисасемблера на C програма.

e04:	b5f8	push	{r3, r4, r5, r6, r7, lr}
e06:	4e0b	ldr	r6, [pc, #44]; (e34 <UARTwrite+0x30>)
e08:	460d	mov	r5, r1
e0a:	4604	mov	r4, r0
e0c:	1847	adds	r7, r0, r1
e0e:	42bc	cmp	r4, r7
e10:	d00d	beq.n	e2e <UARTwrite+0x2a>
e12:	f814 3b01	ldrb.w	r3, [r4], #1
e16:	2b0a	cmp	r3, #10
e18:	d103	bne.n	e22 <UARTwrite+0x1e>
e1a:	6830	ldr	r0, [r6, #0]
e1c:	210d	movs	r1, #13
e1e:	f7ff ffb1	bl	d84 <UARTCharPut>

Адрес на инстр. Машинен код Програма на Асемблер

Фиг. 2.1

Байтовете, които се зареждат в паметта на контролера са показани в средната колонка. Сама по себе си тази информация не е достатъчна за успешното изпълнение на програмата от микропроцесора. Трябва да се знае всеки един байт на кой адрес се намира. Адресите са показани в най-лявата колонка. Микропроцесорът „разбира“ само от числата в тези две колонки. Всичко друго – програма на Асемблер, програма на C, програма на C++, програма на Java е абстракция за улеснение на програмиста.

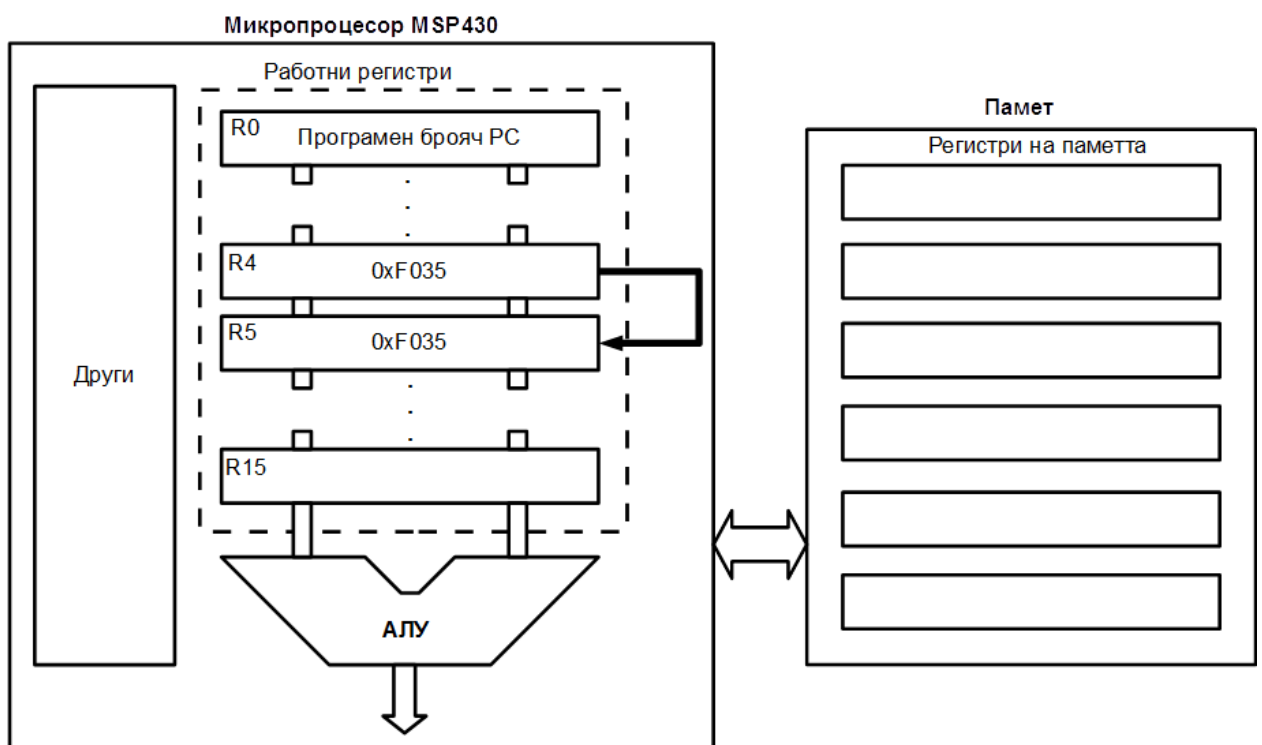
2.4. Видове адресации. Операндите на една инструкция могат да се извлекат по различен начин, от различни адреси в паметта или работните регистри на ядрото. Това обуславя различни видове адресации или още режими на адресиране (Addressing modes). Техният брой варира при различните микропроцесори. За MSP430 са 7 броя. Трябва да се прави разлика между работните регистри на ядрото и външната (за ядрото) памет.

2.4.1 Регистрова адресация (Register) – съдържанието на работен регистър от ядрото е операнд на инструкцията.

Пример: mov.w R4,R5

Преди: R4 = 0xF035, R5 = 0x5555

След: R4 = 0xF035, R5 = 0xF035

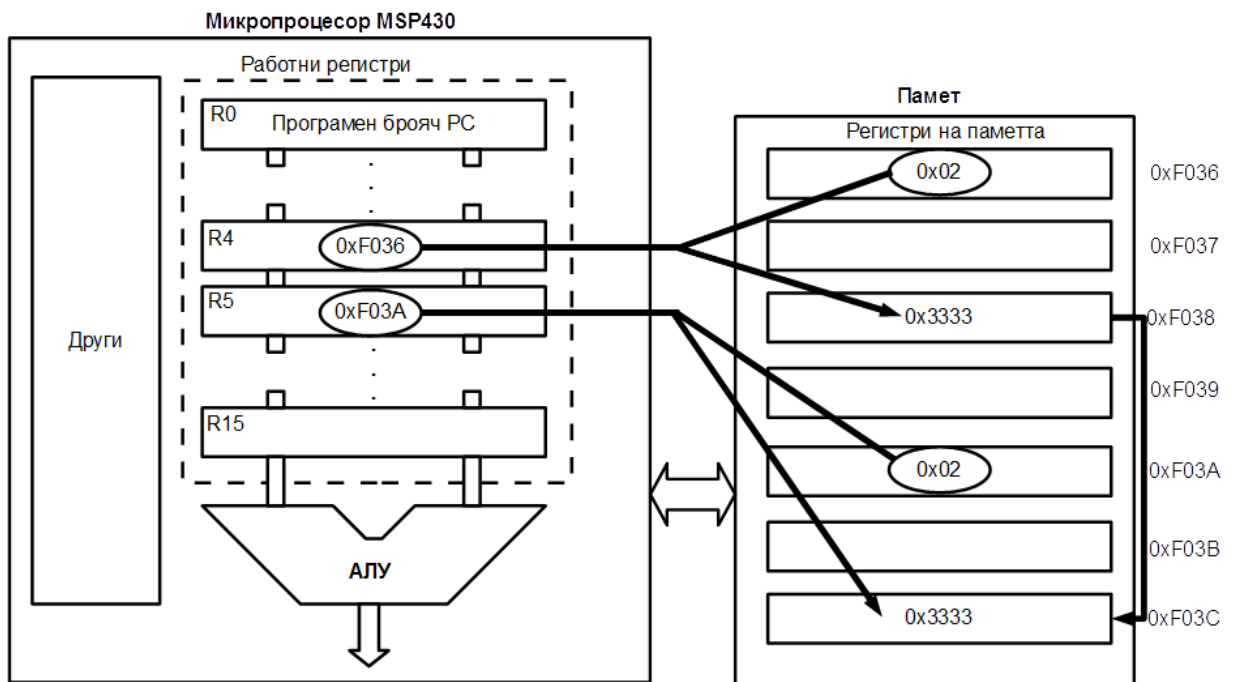


2.4.2. Индексна адресация (Indexed) – съдържанието на работен регистър от ядрото + съдържанието на регистър от програмната памет (отместване) указват адреса (някъде в паметта на микроконтролера) на операнда.

Пример: mov.w 0x02(R4), 0x02(R5)

Преди: R4 = 0xF036, R5 = 0xF03A, MEM(0xF038) = 0x3333, MEM(0xF03C) = 0x5555

След: R4 = 0xF036, R5 = 0xF03A, MEM(0xF038) = 0x3333, MEM(0xF03C) = 0x3333

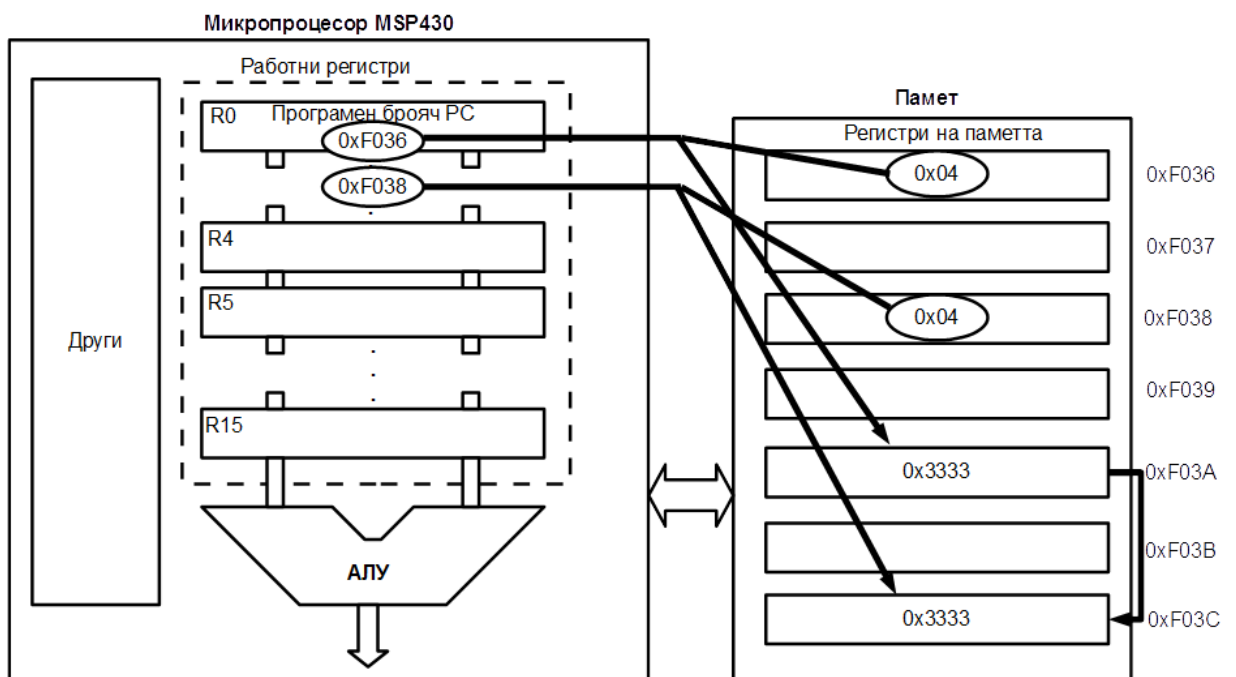


2.4.4. Относителна (Symbolic) адресация – съдържанието на регистър от програмната памет (отместване) + стойността на програмния брояч PC указват адреса на операнда. Тази адресация е аналогична на индексната, с тази разлика, че се използва не-кой да е регистър от ядрото, а точно регистър R0 (т.е. програмния брояч).

Пример: `mov.w Label1,Label2` (етикетите в Асемблер са относителни, или още условни адреси, които се заменят с абсолютни след линкването на програмата). Нека Label1 е с адрес 0xF036, а Label2 е с адрес 0xF038.

Преди: PC = 0xF036, MEM(0xF03A) = 0x3333, MEM(0xF03C) = 0x5555

След: PC = 0xF03A, MEM(0xF03A) = 0x3333, MEM(0xF03C) = 0x3333

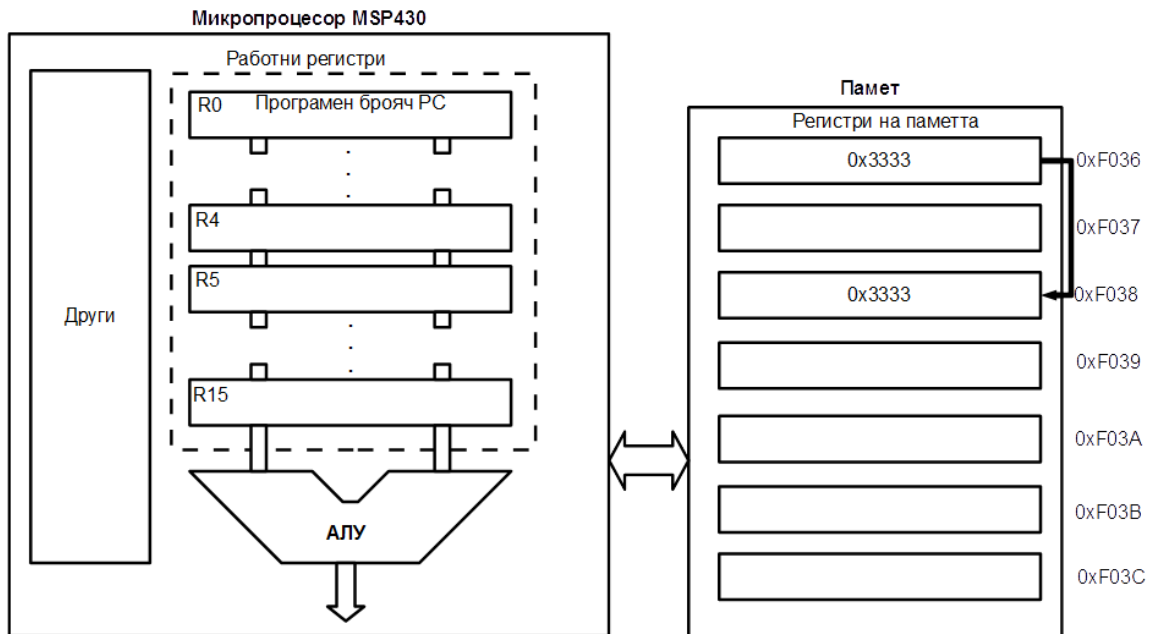


2.4.3. Абсолютна адресация (Absolute) – съдържанието на регистър от програмната памет указва адреса (някъде в паметта на микроконтролера) на операнда.

Пример: `mov.w &0xF036,&0xF038` (иди на адрес 0xF036 и копирай каквото има там на адрес 0xF038)

Преди: $MEM(0xF036) = 0x3333$, $MEM(0xF038) = 0x5555$

След: $MEM(0xF036) = 0x3333$, $MEM(0xF038) = 0x3333$

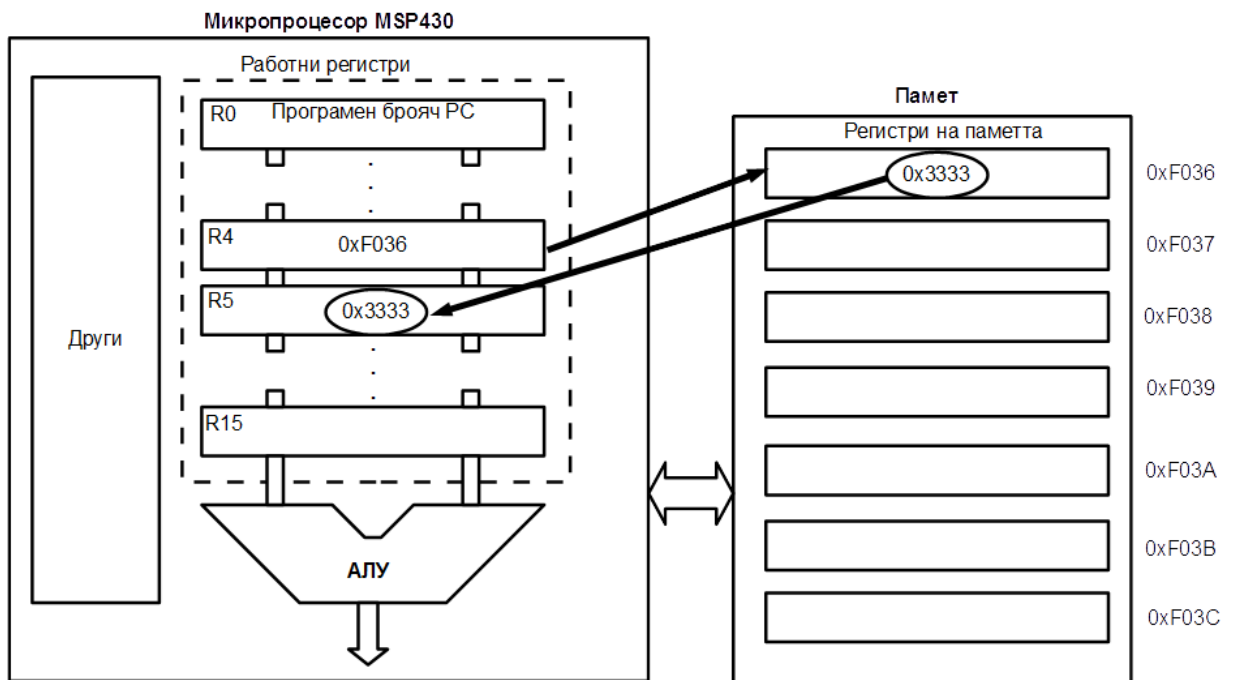


2.4.5. Индиректна регистрова адресация (Indirect Register) – аналогична на абсолютната. Разликата – съдържанието на работен регистър от ядрото указва абсолютен адрес от паметта, на който се намира операнда.

Пример: `mov.w @(R4),R5`

Преди: $MEM(0xF036) = 0x3333$, $R4 = 0xF036$, $R5 = 0x5555$

След: $MEM(0xF036) = 0x3333$, $R4 = 0xF036$, $R5 = 0x3333$



2.4.6. Индиректна автоинкрементираща адресация (Indirect Autoincrement) – съдържанието на работен регистър от ядрото указва абсолютен адрес от паметта, на който се намира операнда. След изпълнение на инструкцията, съдържанието на работния регистър се увеличава автоматично с 1, ако инструкцията е с наставка “.b” или се увеличава с 2, ако инструкцията е с наставка “.w”.

Пример: `mov.w @R4+,R5` (R4 се увеличава с 2)

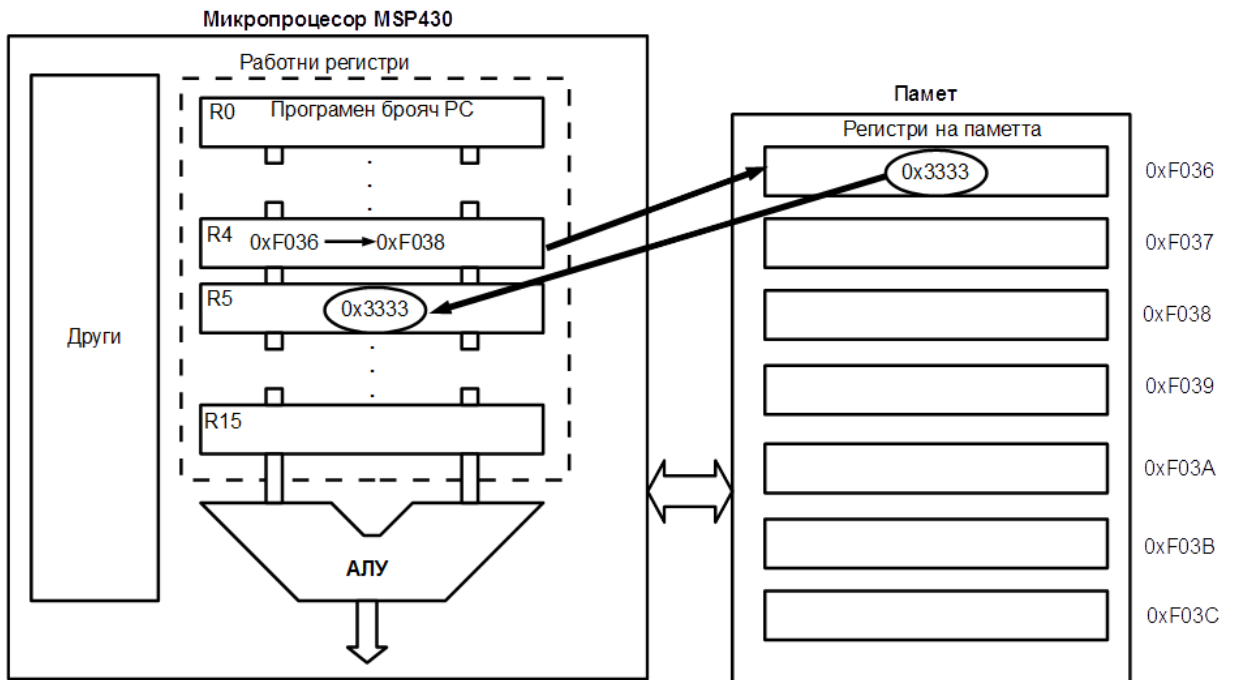
Преди: `MEM(0xF036) = 0x3333, R4 = 0xF036, R5 = 0x5555`

След: `MEM(0xF036) = 0x3333, R4 = 0xF038, R5 = 0x3333`

Пример: `mov.b @R4+,R5` (R4 се увеличава с 1)

Преди: `MEM(0xF036) = 0x3333, R4 = 0xF036, R5 = 0x5555`

След: `MEM(0xF036) = 0x3333, R4 = 0xF037, R5 = 0x0033` (старшата част се нулира заради „.b” наставката)

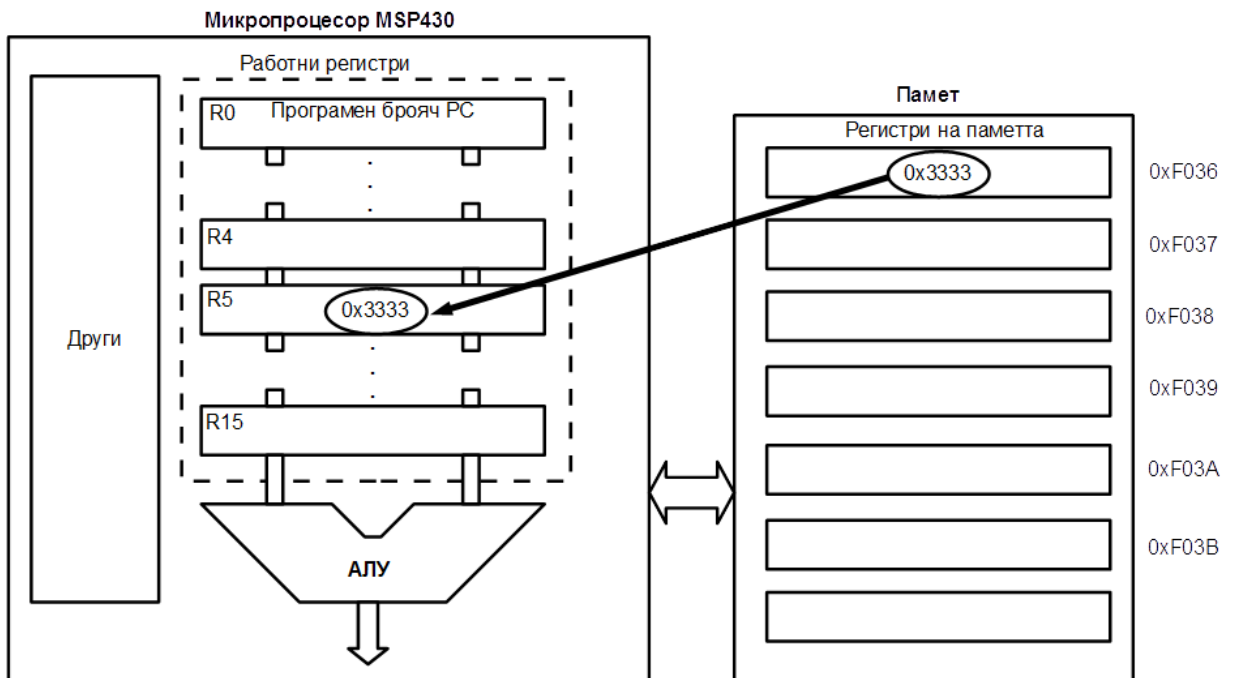


2.4.7. Непосредствена адресация (Immediate) – съдържанието на регистър от програмната памет (константа) е операнд на инструкцията.

Пример: mov.w #0x3333,R5

Преди: R5 = 0x5555

След: R5 = 0x3333



2.5. Задачи за изпълнение.

2.5.1. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на асемблер, която извършва примерни операции, демонстриращи видовете адресации. Използва се само една инструкция – MOV с цел опростяване, но адресациите са валидни за всички инструкции на MSP430.

2.5.2. Да се постави точка на прекъсване точно след реда:

```
MOV.W #WDTPW+WDTHOLD,&WDTCTL
```

Да се изпълнява стъпка по стъпка, като едновременно с това се попълва таблицата, показана по-долу. Програмата приключва на реда „JMP \$“.

1	2	3	4					5		6	7	8			
			Съдържание на регистрите					Вид Адресация (Reg, IX, Sym, Abs, Ind, Ind+, Imm)				Брой цикли	Брой думи	RAM	
№	Инструкция	Инструкция в шестнадесетичен вид	PC	SP	SR	R4	R5	SRC	DST					0x1C00	0x1C01
			.												
			.												
			.												
			Абсолютен адрес			Брой цикли общо:									
			L1	L2	L3										

Инструкцията в двоичен вид се взема от прозореца Disassembly.

Адресацията се записва за левия (SRC – source) и десния (DST – destination) операнд. Ако има само един операнд се пише само в DST. Ако няма операнди се поставя тире.

Брой цикли и брой думи се взимат след попълване на цялата таблица от приложението на следващата страница. Въпросното приложение е взето от документ с описание на микропроцесора MSP430 от фирмата производител Texas Instruments.

Абсолютният адрес на етикетите се взема от прозореца Disassembly. Асемблерът е дал адресите на тези етикети (не забравяйте, че микропроцесорът работи само с числени стойности). Етикетите се използват за улеснение на програмиста.

Общият брой на циклите се изчислява, като се сумират стойностите от всички редове на колонка „Брой цикли“. Това може да помогне на програмиста за оценка времето на изпълнение на програмта му, ако в нея няма условни преходи.

Показване на регистрите на ядрото: View → Register

Показване на регистрите на паметта: View → Memory. Изберете наблюдение на регион 0x1C00 (началото на SRAM).

Показване на дисасемблер View → Disassembly

Table 3-3. Source/Destination Operand Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/-	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/-	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/-	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

В таблиците по-долу TONY и EDE са етикети (условни адреси).

Rn – регистрова; @Rn – индиректна; @Rn+ – индиректна автоинкрементираща; #N – непосредствена, X(Rn) – индексна, EDE – относителна, &EDE – абсолютна.

4.5.1.5.2 Format II (Single-Operand) Instruction Cycles and Lengths

Table 4-9 lists the length and the CPU cycles for all addressing modes of the MSP430 single-operand instructions.

Table 4-9. MSP430 Format II Instruction Cycles and Length

Addressing Mode	No. of Cycles			Length of Instruction	Example
	RRA, RRC SWPB, SXT	PUSH	CALL		
Rn	1	3	4	1	SWPB R5
@Rn	3	3	4	1	RRC @R9
@Rn+	3	3	4	1	SWPB @R10+
#N	N/A	3	4	2	CALL #LABEL
X(Rn)	4	4	5	2	CALL 2(R7)
EDE	4	4	5	2	PUSH EDE
&EDE	4	4	6	2	SXT &EDE

Забележка: Колонката „RRA, RRC, SWPB, SXT” трябва да се чете като „Всички инструкции без PUSH и CALL”.

(Формат III)

4.5.1.5.3 Jump Instructions Cycles and Lengths

All jump instructions require one code word and take two CPU cycles to execute, regardless of whether the jump is taken or not.

NOP – 1 дума, 1 такт

4.5.1.5.4 Format I (Double-Operand) Instruction Cycles and Lengths

Table 4-10 lists the length and CPU cycles for all addressing modes of the MSP430 Format I instructions.

Table 4-10. MSP430 Format I Instructions Cycles and Length

Addressing Mode		No. of Cycles	Length of Instruction	Example
Source	Destination			
Rn	Rm	1	1	MOV R5, R8
	PC	3	1	BR R9
	x(Rm)	4 ⁽¹⁾	2	ADD R5, 4 (R6)
	EDE	4 ⁽¹⁾	2	XOR R8, EDE
	&EDE	4 ⁽¹⁾	2	MOV R5, &EDE
@Rn	Rm	2	1	AND @R4, R5
	PC	4	1	BR @R8
	x(Rm)	5 ⁽¹⁾	2	XOR @R5, 8 (R6)
	EDE	5 ⁽¹⁾	2	MOV @R5, EDE
	&EDE	5 ⁽¹⁾	2	XOR @R5, &EDE
@Rn+	Rm	2	1	ADD @R5+, R6
	PC	4	1	BR @R9+
	x(Rm)	5 ⁽¹⁾	2	XOR @R5, 8 (R6)
	EDE	5 ⁽¹⁾	2	MOV @R9+, EDE
	&EDE	5 ⁽¹⁾	2	MOV @R9+, &EDE
#N	Rm	2	2	MOV #20, R9
	PC	3	2	BR #2AEh
	x(Rm)	5 ⁽¹⁾	3	MOV #0300h, 0 (SP)
	EDE	5 ⁽¹⁾	3	ADD #33, EDE
	&EDE	5 ⁽¹⁾	3	ADD #33, &EDE
x(Rn)	Rm	3	2	MOV 2 (R5), R7
	PC	5	2	BR 2 (R6)
	TONI	6 ⁽¹⁾	3	MOV 4 (R7), TONI
	x(Rm)	6 ⁽¹⁾	3	ADD 4 (R4), 6 (R9)
	&TONI	6 ⁽¹⁾	3	MOV 2 (R4), &TONI
EDE	Rm	3	2	AND EDE, R6
	PC	5	2	BR EDE
	TONI	6 ⁽¹⁾	3	CMP EDE, TONI
	x(Rm)	6 ⁽¹⁾	3	MOV EDE, 0 (SP)
	&TONI	6 ⁽¹⁾	3	MOV EDE, &TONI
&EDE	Rm	3	2	MOV &EDE, R8
	PC	5	2	BR &EDE
	TONI	6 ⁽¹⁾	3	MOV &EDE, TONI
	x(Rm)	6 ⁽¹⁾	3	MOV &EDE, 0 (SP)
	&TONI	6 ⁽¹⁾	3	MOV &EDE, &TONI

⁽¹⁾ MOV, BIT, and CMP instructions execute in one fewer cycle.

Задача 2.5.1.

```

#include "msp430.h" ; #define controlled include file

NAME main ; module name
PUBLIC main ; make the main label visible
; outside this module

ORG 0FFFFh
DC16 init ; set reset vector to 'init' label

RSEG CSTACK ; pre-declaration of segment
RSEG CODE ; place program in 'CODE' segment

init: MOV #SFE(CSTACK), SP ; set up stack
MOV.W #0x00, R4 ; Нулирай работен регистър R5
MOV.W #0x00, R5 ; Нулирай работен регистър R6
MOV.W #0x0020, R7 ; Зареди числото 32 в R7
MOV.W #0x1C00, R8 ; Зареди началния адрес на SRAM в R8
MOV.W #0x1C00, R9 ; Зареди началния адрес на SRAM в R9
MOV.W #0x0000, &0x1C00 ; Инициализирай първите 2 байта от SRAM
;Нулирай първите 64 байта от SRAM-----
zero: MOV.W 0(R8), 0(R9)
ADD.W #2, R9
DEC.W R7
JNZ zero
;-----
CLR R2 ; Нулирай STATUS регистъра

main: NOP ; main program
MOV.W #WDTPW+WDTHOLD,&WDTCTL ; Stop watchdog timer

MOV.W #0x5555, R4 ; Поставете точка на прекъсване на този ред
MOV.W R4, R5

MOV.W #0x1BFF, R4 ; Демонстрация на Непосредствена адресация
MOV.W #0x1C00, R5
MOV.W #0x3412, &0x1C00
MOV.W #0x7856, &0x1C04

MOV.W 1(R4), 4(R5) ; Демонстрация на Индексна адресация

L1: MOV #0x05, R4
DEC R4
JNZ L1 ; Демонстрация на Символна адресация

NOP

JMP L2 ; Демонстрация на Символна адресация
NOP ; Обърнете внимание на инструкцията в двоичен вид
NOP ; и броя на NOP инструкциите
NOP
NOP

L2: JMP L3 ; Демонстрация на Символна адресация
NOP ; Обърнете внимание на инструкцията в двоичен вид
NOP ; и броя на NOP инструкциите
NOP

L3: MOV.W #0xcdab, &0x1C00
MOV.W &0x1C00, &0x1C04 ; Демонстрация на Абсолютна адресация

CLR.W &0x1C04
MOV.W #0x1C00, R4
MOV.W @R4, &0x1C04 ; Демонстрация на Индиректна адресация

MOV.W R4, &0x1C04

MOV.W @R4+, R5 ;Демонстрация на Индиректна автоинкрементираща
MOV.W @R4+, R5 ; адресация
MOV.W @R4+, R5
MOV.W @R4+, R5
MOV.W #0x1C00, R4

MOV.B @R4+, R5 ; Демонстрация на Индиректна автоинкрементираща

```

MOV.B	@R4+, R5	; адресация с байтов достъп
MOV.B	@R4+, R5	
MOV.B	@R4+, R5	
MOV.W	#0x1234, R4	
MOV.W	#0xabcd, R5	
PUSH.W	R4	; Демонстрация на PUSH и POP инструкции
PUSH.W	R5	; Обърнете внимание на реда на извикването им
MOV.W	#0x5555, R4	
MOV.W	#0x3333, R5	
POP.W	R5	
POP.W	R4	
JMP	\$	
END		

Лабораторно упражнение №3

“Управление на входно-изходните портове на микроконтролера MSP430FR6989”

3.1.Въведение. MSP430FR6989(PZ100) има 100 извода, от които 83 могат да се използват като логически входове/изходи за общо предназначение (GPIO – General Purpose Input Output). Конфигурацията им се променя в зависимост от фърмуера, който изпълнява микроконтролера. Texas Instruments номерират всеки извод по следната схема:

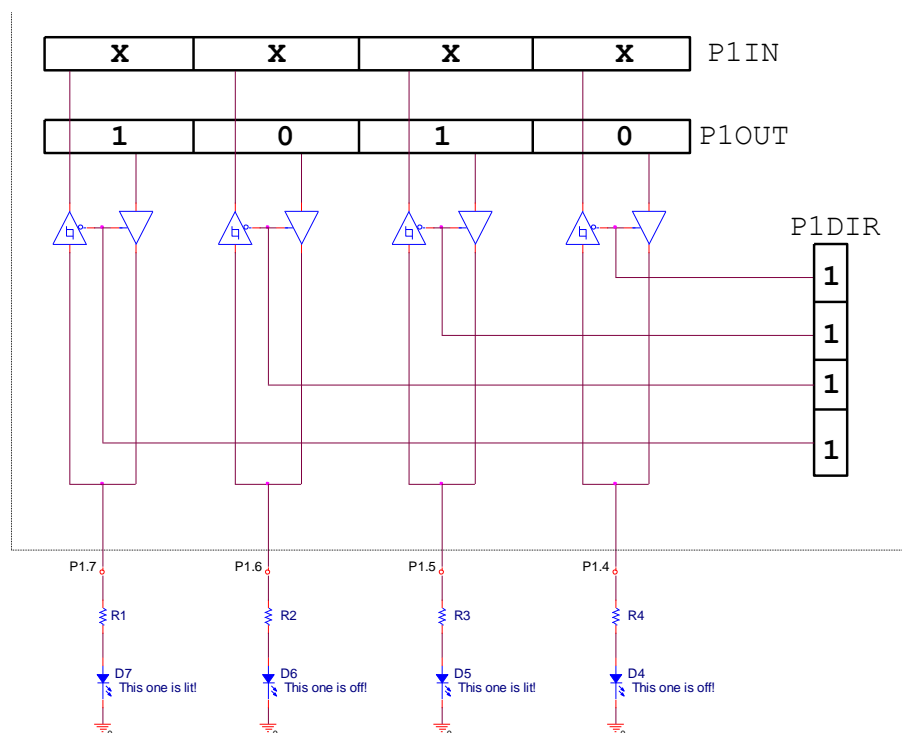
Px.y

където **x** е номера на входно-изходния порт, а **y** – номера на входно-изходния извод.

Портовете се номерират с цифра от 1 ÷ 9, като изключение прави PJ.y, който обозначава специален порт, към който е свързан JTAG интерфейсът. Ако JTAG не се използва, те могат да се използват като изводи с общо предназначение. Входовете са с тригер на Шмит. Номерът на извода може да приема стойности P1.(0 ÷ 7), P2.(0 ÷ 7), P3.(0 ÷ 7), P4.(0 ÷ 1), PJ.(0 ÷ 5) и т.н.

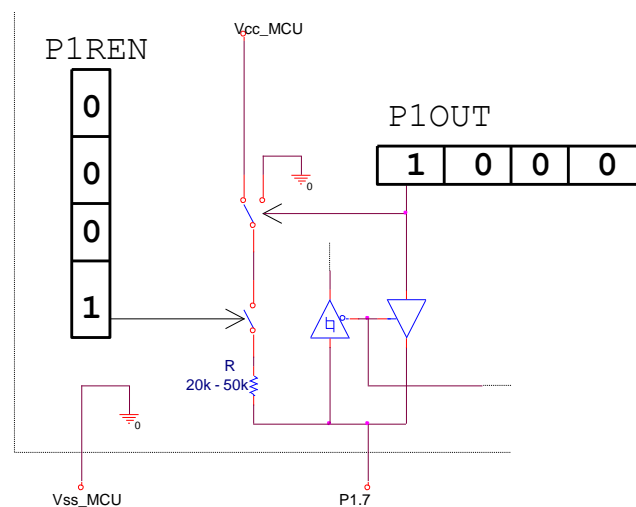
На всеки извод отговаря по един бит от даден регистър в микроконтролера. Най-важните регистри за управлението на портовете са:

- PxDIR – регистър, указващ типа на извода. Избира дали ще е вход или изход.
- PxIN – регистър, отразяващ логическото състояние на изводите, конфигурирани като входове.
- PxOUT – регистър, контролиращ логическото състояние на изводите, конфигурирани като изходи.



Фиг. 3.1

Ако конфигурираме изводите на микроконтролера като изходи и свържем към тях периферни устройства, например светодиоди, то чрез запис в P_xOUT регистъра ще можем да управляваме тези устройства посредством логическите нива на изводите. Такъв пример е показан на **фиг.3.1**. За простота е показан само най-старшият квартет от използваните регистри. При запис 10100000_(b) в P1OUT ще светят само светодиоди D7 и D5. Аналогично при конфигуриране на изводите като входове, ако подадем логическа 1 на P1.7 и P1.5, то бит 7 и 5 от P1IN ще се установят в 1. Тогава програмата може да прочете регистър P1IN, което ще върне резултат 10100000_(b) и така да разбере на кой извод какво ниво е подадено. Често в микроконтролерите се включват вътрешни резистори, издърпващи даден вход към логическа 0 или 1 (**фиг. 3.2**). Това позволява намаляване броя на външно-включваните елементи. Най-често това се използва при свързване на бутони и схеми с отворен колектор/дрейн.

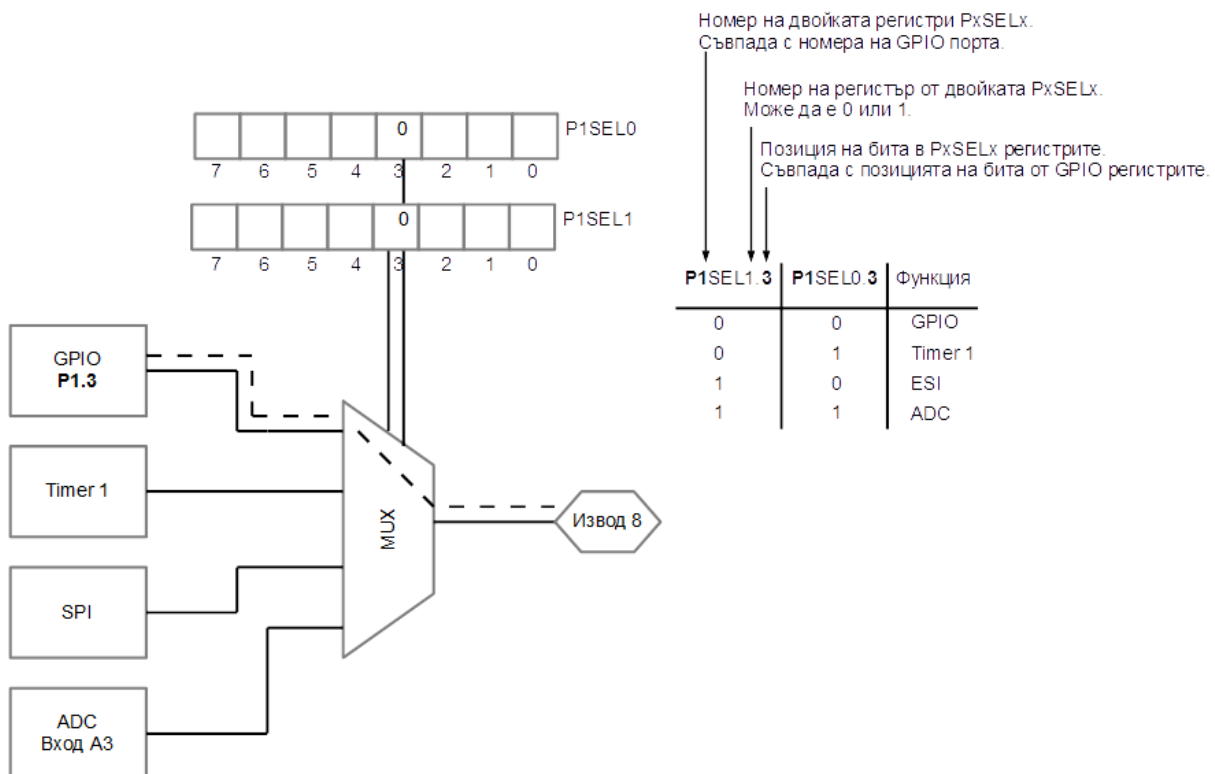


Фиг. 3.2

- P_xREN – регистър за включване/изключване на вътрешния издърпващ резистор (1 – включен, 0 – изключен). Когато дадения извод е конфигуриран като вход, P_xOUT задава вида на резистора (1 – pull-up, 0 – pull-down). Стойността му е без значение и варира от 20 ÷ 50 k (по каталожни данни).
- P_xIE – регистър, разрешаващ прекъсванията (interrupts). Ако даден извод е конфигуриран като вход, той може да се използва за генериране на прекъсване (1 – прекъсването от даден извод е разрешено, 0 – прекъсването е забранено). Прекъсването е процес на спиране изпълнението на главната програма и стартиране изпълнението на специална програма (interrupt handler), разположена на различен адрес в паметта. Този процес е асинхронен спрямо изпълнението на главната програма. Той позволява микроконтролерът свободно да извършва дадени операции и само при настъпване на събитие (прекъсване) да обслужва заявката чрез специален сорс код. Алтернативен метод на работа е с постоянна проверка (polling) за настъпило събитие, но докато тя се извършва, микроконтролерът не може да прави нищо друго.
- P1IFG – регистър с битове, отговарящи на източника на прекъсване. Специалната програма (interrupt handler) в повечето случаи е само една за GPIO модула. При работа с два или повече извода, програмата няма как да знае кой от тях е генерирал прекъсването. Затова регистър P1IFG

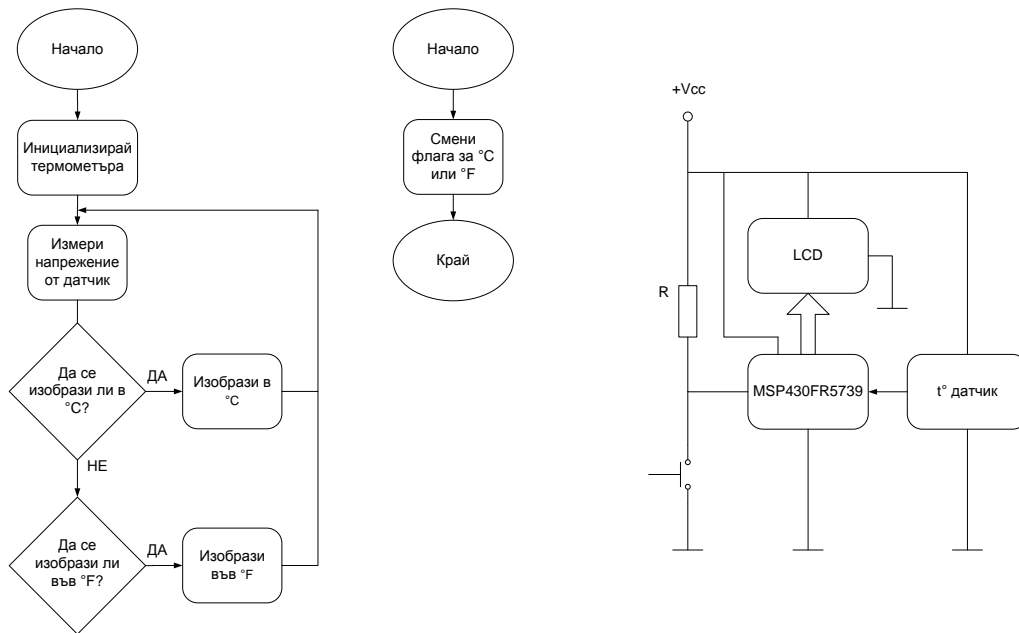
съдържа флагове, осигуряващи ни информацията кой точно извод е генерирал прекъсването. Позицията на флага (бита) носи тази информация.

- PxIES – регистър, определящ активния фронт на прекъсването. Прекъсването от извод, конфигуриран като вход, може да се осъществи по два начина – по нарастващ фронт (0 → 1) или по падащ фронт (1 → 0) на входния сигнал. 0 – конфигурира прекъсване по нарастващ фронт (rising edge), 1 – по падащ (falling edge). Други микроконтролери позволяват прекъсване по четири начина – по фронт (0 → 1 / 1 → 0) и по ниво (1 / 0).
- PxSEL0 и PxSEL1 – регистри, конфигуриращи функцията на извода. При всички микроконтролери даден извод понякога може да извършва повече от една функция. В настоящото упражнение разглеждаме функцията GPIO, но изводите могат да се използват и от ADC, компаратор, PWM, SPI и други модули. За да се превключи функцията на извода, производителят на ИС реализира мултиплексор, като всеки негов вход се избира от регистри PxSEL0 и PxSEL1 (за MSP430). Данни за функциите на всеки извод може да се намерят в каталога (datasheet) на микроконтролера. На **фиг. 3.3** е показан извод 63 на MSP430FR6989 (PZ100), който може да се използва от GPIO, Timer1, ESI, и ADC модула. Както се вижда от таблицата на фигурата, името на регистъра се формира от номера на GPIO порта, свързан към дадения извод. Това значи, че регистри P1SEL0 – P1SEL1 мултиплексират изводи P1.0 – P1.7, P2SEL0 – P2SEL1 мултиплексират P2.0 – P2.7 и т.н. Комбинацията от допълнителни периферни модули, свързани към изводите, зависи от конкретния микроконтролер.



Фиг. 3.3

Пример за програма, използваща прекъсвания е показана на **фиг. 3.4**. Представена е блоковата схема на устройството и управляващия алгоритъм. От тях е видно, че основната програма изпълнява измерването на температурата и опреснява дисплея. Когато потребителят натисне бутона, микропроцесорът получава прекъсване и обслужва по-малка програма, която променя начина на изобразяване на температурата. След това изпълнението на главната програма продължава там, откъдето е била прекъсната.



Фиг. 2.4

3.2 Задачи за изпълнение.

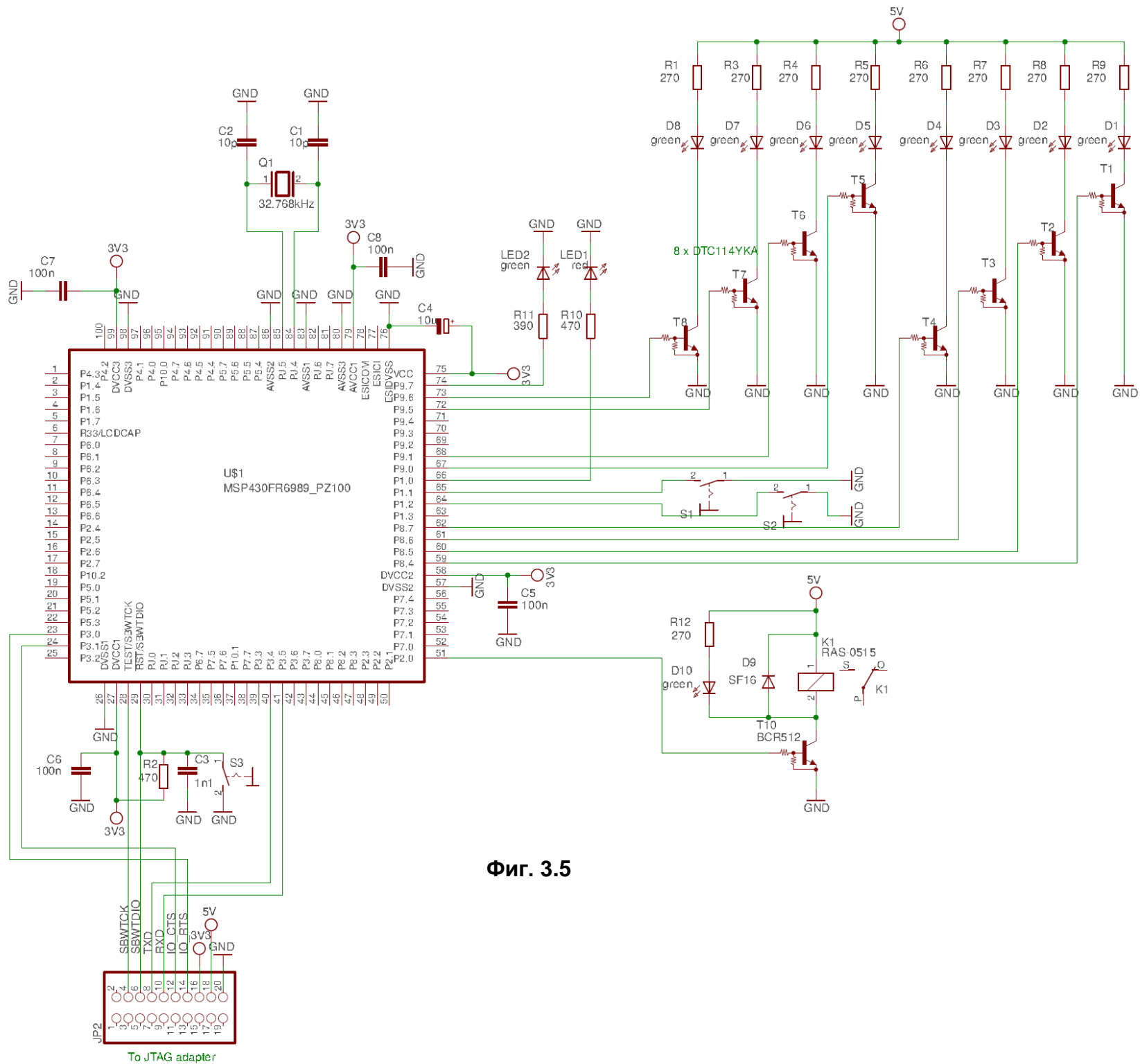
3.2.1. Да се създаде нов проект на Desktop-а в отделна директория с име 03_2_1_Lab, да се копира програмата на асемблер, която включва и изключва светодиод от демоплатката с MSP430FR6989. Принципната схема е показана на **фиг. 3.5**. С цел опростяване, част от елементите на макета са пропуснати. Да се изпълни програмата стъпка по стъпка. Да се наблюдават регистрите на микропроцесора и изходния порт. Да се постави точка на прекъсване.

3.2.2. Да се създаде нов проект на Desktop-а в отделна директория с име 03_2_2_Lab и да се реализира задача 3.2.1., но с програмния език C и друг (произволен) светодиод.

3.2.3. Да се създаде нов проект на Desktop-а в отделна директория с име 03_2_3_Lab и да се реализира програма на C, която да чете състоянието на бутон S1 и да превключва светодиод D8 от платката. Да се използва метода "polling". От View → Register да се наблюдава регистъра P1IN.

3.2.4. Да се създаде нов проект на Desktop-а в отделна директория с име 03_2_4_Lab и да се реализира програма на C, която да чете състоянието на бутон S1 и да превключва светодиод D8 от платката. Да се използват прекъсвания.

3.2.5. Да се обясни разликата в механизма на действие на задача 3.2.3. и 3.2.4.



Фиг. 3.5

Задача 3.2.1.

```
#include "msp430.h" ; #define controlled include file

NAME main ; module name

PUBLIC main ; make the main label visible
; outside this module

ORG 0FFFEh
DC16 init ; set reset vector to 'init' label

RSEG CSTACK ; pre-declaration of segment
RSEG CODE ; place program in 'CODE' segment

init: MOV #SFE(CSTACK), SP ; set up stack

main: NOP ; main program
MOV.W #WDTPW+WDTHOLD,&WDTCTL ; Stop watchdog timer
BIC.W #LOCKLPM5,&PM5CTL0 ;Изключи високоимпедансното състояние
;на входно-изходните изводи
;-----
BIS.B #???, &P8DIR ;Инициализация на порт P8, извод 4 – изход
;MSP430FR6989 Userguide →стр. 386
BIC.B #???, &P8OUT ;Установи извода в логическа 0
;MSP430FR6989 Userguide →стр. 386

L1 XOR.B #???, &P8OUT ;Смяна състоянието на извод P8.4
;MSP430FR6989 Userguide →стр. 386
CALL #Delay ;Извикай подпрограма за изчакване
JMP L1 ;Върни се при етикет L1

Delay: MOV.W #65535, R15 ;Зареди регистър R15 с числото 65535
L2 DEC.W R15 ;Намали регистър R15 с 1
JNZ L2 ;Ако R15 == 0, продължи. Ако R15 != 0,
;върни се при етикет L2
RET ;Върни се в main
;-----
END
```

Задача 3.2.2.

```
#include "io430.h"

void main( void )
{
    volatile unsigned long i;

    WDTCTL = WDTPW | WDTHOLD; //Stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5; //Изключи високоимпедансното състояние на GPIO

    ??? //Конфигурирай извод P8.5 като изход, MSP430FR6989 Userguide →стр. 386
    ??? //Инициализирай нивото на P8.5 в логическа нула, MSP430FR6989 Userguide →стр. 386

    for( ;; )
    {
        ???//Реализирай преобръщане на логическото ниво (toggle) чрез логически оператор от C

        ??? //Реализирай софтуерно закъснениечрез празен for( ;; ){ } цикъл
    }
}
```

Задача 3.2.3.

```
#include "io430.h"
```

```
void main( void )
```

```
{  
    unsigned long i;
```

```
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer to prevent time out reset  
    PM5CTL0 &= ~LOCKLPM5; //Изключи високоимпедансното състояние на GPIO
```

```
    P8DIR |= 0x30; //Конфигурирай изводи P8.4 и P8.5 като изходи  
    P8OUT &= ~0x30; //Инициализирай изводи P8.4 и P8.5 с лог. 0  
    P9DIR |= 0x40; //Конфигурирай извод P9.6 като изход  
    P9OUT &= ~0x40; //Инициализирай извод P9.6 с лог. 0
```

```
    ??? //Конфигурирай извод P1.1 като вход, MSP430FR6989 Userguide →стр. 386
```

```
    ???//Включи издърпващ резистор на извод P1.1, MSP430FR6989 Userguide →стр. 387
```

```
    ???//Свържи издърпващия резистор към захранване Vdd, MSP430FR6989 Userguide  
        //→стр. 386
```

```
    while(1)
```

```
    {  
        P8OUT |= 0x10;  
        P8OUT &= ~0x20;  
        for(i = 0; i < 40000; i++){  
            P8OUT &= ~0x10;  
            P8OUT |= 0x20;  
            for(i = 0; i < 40000; i++){
```

```
                if((P1IN & 0x02) == 0)
```

```
                {  
                    P9OUT ^= 0x40;
```

```
                }
```

```
            }
```

```
    }
```

Задача 3.2.4.

```
#include "io430.h"

void main( void )
{
    unsigned long i;
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer to prevent time out reset
    PM5CTL0 &= ~LOCKLPM5; //Изключи високоимпедансното състояние на GPIO

    P8DIR |= 0x30; //Конфигурирай изводи PJ.0 и PJ.1 като изходи.
    P8OUT &= ~0x30; //Инициализирай изводи P8.4 и P8.5 с лог. 0
    P9DIR |= 0x40; //Конфигурирай извод P3.7 като изход.
    P9OUT &= ~0x40; //Инициализирай извод P3.7 с лог. 0.

    ??? //Конфигурирай извод P1.1 като вход, MSP430FR6989 Userguide →стр. 386
    ??? //Включи издърпващ резистор на извод P1.1, MSP430FR6989 Userguide →стр. 387
    ??? //Свържи издърпващия резистор към захранване Vdd, MSP430FR6989 Userguide →стр. 386

    ??? //Избери прекъсване по падащ фронт, MSP430FR6989 Userguide →стр. 388
    ??? //Нулирай всички флагове на прекъсване за порта, MSP430FR6989 Userguide →стр. 389
    ??? //Разреши прекъсването от извод P1.1, MSP430FR6989 Userguide →стр. 388

    __enable_interrupt( ); //Разреши глобално прекъсванията към микропроцесора MSP430.

    while(1)
    {
        P8OUT |= 0x10;
        P8OUT &= ~0x20;
        for(i = 0; i < 40000; i++){ }
        P8OUT &= ~0x10;
        P8OUT |= 0x20;
        for(i = 0; i < 40000; i++){ }
    }
}

#pragma vector=PORT1_VECTOR //Хендлер на прекъсване за GPIO модул 1.
__interrupt void port1_handler(void)
{
    unsigned int interrupt_flag;
    unsigned long i;

    interrupt_flag = P1IFG; //Прочети регистъра с флаговете.

    switch(interrupt_flag)
    {
        case 0x01: //Прекъсване от извод P1.0.
            break;
        case 0x02: //Прекъсване от извод P1.1.
            ??? //Преобърни извод P9.6, чрез логически оператор от C
            break;
        default: //Неразпознат източник на прекъсване.
            break;
    }

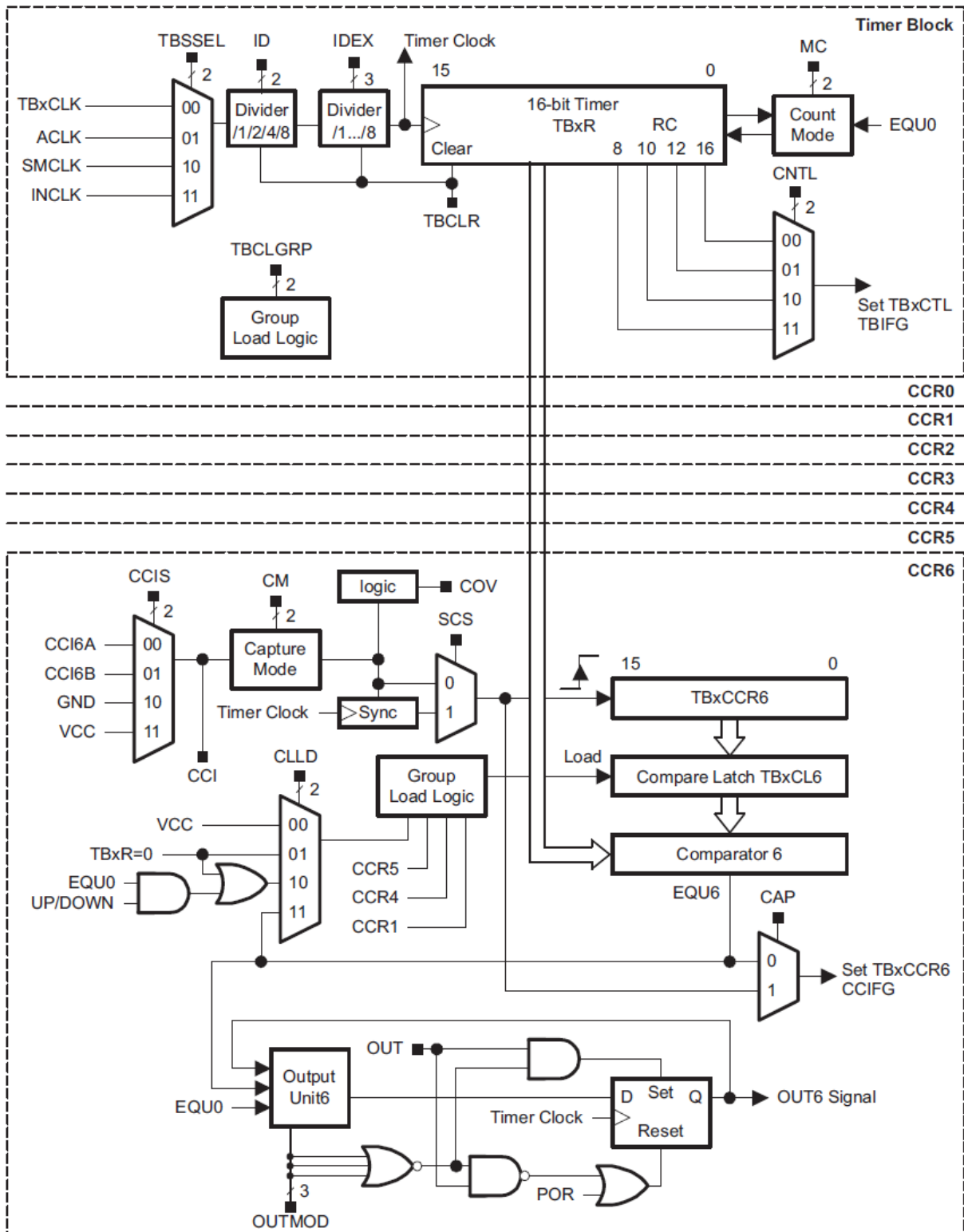
    for(i = 0; i < 30000; i++){ } //Закъснение срещу притрепване на контакта.
    P1IFG = 0x00; //Нулирай флаговете на прекъсване.
}
}
```

Лабораторно упражнение №4

“Изследване на таймерен модул на микроконтролер MSP430FR6989”

4.1. Въведение. В практиката се налага микроконтролерните системи да генерират или да отмерват точно определени времеинтервали. Ако програмиста знае тактовата честота на контролера и времето за изпълнение на асемблерните инструкции, то той би могъл да реализира софтуерно времеинтервал. Това обаче почти никога не се прави, защото тактовата честота може да се изменя динамично, изпълнението на програмата не е последователно (обслужват се прекъсвания, които биха удължили софтуерното закъснение), съществуват инструкции с условие, които променят хода на програмата. Затова във всички микроконтролери се вграждат хардуерни модули за отмерване на време, сърцето на които е един брояч. Тези модули се наричат таймери. Те се конфигурират от микропроцесора, чрез запис в съответните им регистри. Броячът се увеличава/намалява автономно, независимо от останалата система. При препълване той се нулира и сигнализира за това на микропроцесорното ядро чрез сигнал за прекъсване. Когато увеличаването/намаляването на брояча става в интервал от време, определен от извод на микроконтролера се казва, че таймерът работи в режим *Capture*. Когато текущата стойност на брояча се сравнява с друг регистър и при съвпадение на двете стойности се генерира прекъсване се казва, че таймерът работи в режим *Compare*. Когато стойността на брояча се сравнява с други два регистъра и автоматично се запуска отначало след достигане на съвпадение се казва, че таймера работи в режим на генерация на ШИМ (Широчинно Импулсна Модулация) сигнал, или още режим *PWM*. В последния случай единия регистър определя периода на ШИМ, а другия – коефициента на запълване. Таймерите понякога се наричат *Capture/Compare/PWM* или *CCP* модули.

4.2. Таймерен модул на MSP430FR6989. На **фиг. 4.1** е показана блокова схема на таймерния модул. От нея се вижда, че 16-битовият брояч за един таймерен модул е само един, докато *CCP* регистрите може да достигнат до 7 броя. За конкретния модел (6989) те са 7 (т.е. максимумът). Всеки един *CCP* регистър е поместен заедно с допълнителна логика в таймерен подмодул. От друга страна микроконтролера има общо три таймерни модула (*Timer_A0*, *Timer_A1* и *Timer_B0*) с общо 14 *CCP* регистъра. Тактовата честота, която се подава на брояча идва от един от източниците *TBxCLK*, *ACLK*, *SMCLK* или *INCLK*. Следват два делителя на честота, които позволяват гъвкава настройка на времеинтервалите. Бит *CAP* от съответния *TB(x)CCTL(n)* регистър определя работата на всеки един подмодул в режим на *Capture* или *Compare*.



Фиг. 4.1

4.3. Задачи за изпълнение.

4.3.1. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която мига светодиода D1 от демо платката с помощта на таймерния модул. Задайте 100 милисекунди времеинтервал. Нека тактовата честота на DCO генератора е 8 MHz. Нека тактовата честота на ядрото и всички тактови сигнали на микроконтролера да се конфигурират на 8 MHz (ACLK = VLOCLK, SMCLK = MCLK = DCO).

4.3.2. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която генерира PWM сигнал на извод P2.6 с честота 1 kHz и коефициент на запълване 50 %. Да се свърже осцилоскоп на този извод.

4.3.3. Да се зададе коефициент на запълване 5%, 25 % и 75 %. Да се снимат осцилограмите.

4.3.4. Да се промени честотата на PWM сигнала.

4.3.5. Да се генерират два PWM сигнала в противофаза (на P2.6 и P2.7). Да се снимат осцилограмите.

Задача 4.3.1.

```
#include "io430.h"

// Timer A0 interrupt service routine
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    TA0CTL &= ~TAIFG;
    P8OUT ^= 0x10;
}

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    P8DIR |= 0x10;

    // Тактов генератор DCO = 8MHz, Userguide -> стр. 104
    CSCTL0_H = 0xA5;
    CSCTL1 |= DCORSEL; //DCORSEL = 1
    CSCTL1 &= ~0x0E; //Занули DCOFSEL битовете
    CSCTL1 |= DCOFSEL_3; //Задай DCOFSEL битовете = 011b = 3h
    CSCTL2 = 0x00; //Занули всички битове от регистъра CSCTL2
    CSCTL2 |= (SELA_1 | SELS_3 | SELM_3); //ACLK = VLOCLK; SMCLK = DCO; MCLK = DCO
    CSCTL3 = 0x00; //Занули всички битове от регистъра CSCTL3
    CSCTL3 |= (DIVA_0 | DIVS_0 | DIVM_0); //Всички делители на тактовите сигнали ÷ 1

    TA0CTL = TASSEL_2 | ID_3 | ?????; //SMCLK | делител=8 | режим UP/DOWN, Userguide -> стр. 641
    TA0CCTL0 |= ?????; //Прекъсване при съвпадение с Capture/Compare рег., Userguide -> стр. 643
    TA0CCR0 = ?????; // Compare стойност за сравнение с рег. на таймера, Userguide -> стр. 319

    __enable_interrupt( );

    while(1){
        __bis_SR_register(CPUOFF | GIE); //Изключи µPU за понижаване на консумацията
    }
}
```

Задача 4.3.2.

```
#include "io430.h"

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    // Тактов генератор DCO = 8MHz, Userguide -> стр. 104
    CSCTL0_H = 0xA5;
    CSCTL1 |= DCORSEL; //DCORSEL = 1
    CSCTL1 &= ~0x0E; //Занули DCOFSEL битове
    CSCTL1 |= DCOFSEL_3; //Задай DCOFSEL битове = 011b = 3h
    CSCTL2 = 0x00; //Занули всички битове от регистъра CSCTL2
    CSCTL2 |= (SELA_1 | SELS_3 | SELM_3); //ACLK = VLOCLK; SMCLK = DCO; MCLK = DCO
    CSCTL3 = 0x00; //Занули всички битове от регистъра CSCTL3
    CSCTL3 |= (DIVA_0 | DIVS_0 | DIVM_0); //Всички делители на тактовите сигнали ÷ 1

    //Превключи мултиплектора на извод P2.6 към таймерния модул
    P2DIR |= 0x40; // P2.6 изход
    P2SEL0 |= 0x40; // P2.6 изход на
    P2SEL1 &= ~0x40; // таймерния модул TB0, capture/compare submodule №5
    LCDPCCTL2 &= ~0x200; //Изключи сигналът S41 на LCD модула

    TB0CTL |= (TBSSEL_2 | ID_3 | MC_1 | TBCLR); // SMCLK | SMCLK/8 | сумиращ брояч (up mode) |
    //изчисти TBR
    TB0CCTL5 |= OUTMOD_7; //Включи логиката на изходния блок от submodule №5
    TB0CCR0 = 0x0000; // PWM период, Userguide -> стр. 669
    TB0CCR5 = 0x0000; // PWM коэффициент на запълване, Userguide -> стр. 669

    while(1){
        __bis_SR_register(CPUOFF | GIE); //Изключи µPU за понижаване на консумацията
    }
}
```

Задача 4.3.5.

```
#include "io430.h"

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    // Тактов генератор DCO = 8MHz, Userguide -> стр. 104
    CSCTL0_H = 0xA5;
    CSCTL1 |= DCORSEL; //DCORSEL = 1
    CSCTL1 &= ~0x0E; //Занули DCOFSEL битовете
    CSCTL1 |= DCOFSEL_3; //Задай DCOFSEL битовете = 011b = 3h
    CSCTL2 = 0x00; //Занули всички битове от регистъра CSCTL2
    CSCTL2 |= (SELA_1 | SELS_3 | SELM_3); //ACLK = VLOCLK; SMCLK = DCO; MCLK = DCO
    CSCTL3 = 0x00; //Занули всички битове от регистъра CSCTL3
    CSCTL3 |= (DIVA_0 | DIVS_0 | DIVM_0); //Всички делители на тактовите сигнали ÷ 1

    //Превключи мултиплектора на извод P2.6 към таймерния модул
    P2DIR |= 0x40; // P2.6 изход
    P2SEL0 |= 0x40; // P2.6 изход на
    P2SEL1 &= ~0x40; // таймерния модул TB0, capture/compare субмодул №5
    LCDCPCTL2 &= ~0x200; //Изключи сигналът S41 на LCD модула

    //Превключи мултиплектора на извод P2.7 към таймерния модул
    P2DIR |= 0x80; // P2.6 изход
    P2SEL0 |= 0x80; // P2.6 изход на
    P2SEL1 &= ~0x80; // таймерния модул TB0, capture/compare субмодул №5
    LCDCPCTL2 &= ~0x100; //Изключи сигналът S40 на LCD модула

    TB0CTL |= (TBSSSEL_2 | ID_3 | MC_1 | TBCLR); // SMCLK | SMCLK/8 | сумиращ брояч (up mode) |
    //изчисти TBR
    TB0CCTL5 |= OUTMOD_7; //Включи логиката на изходния блок от субмодул №5
    TB0CCTL6 |= ???; //Включи логиката на изходния блок от субмодул №6 в противофаза
    //на субмодул №5, Userguide -> стр. 667

    TB0CCR0 = ???; // PWM период, Userguide -> стр. 669
    TB0CCR5 = ???; // PWM_TB0.5 коефициент на запълване, Userguide -> стр. 669
    TB0CCR6 = ???; // PWM_TB0.6 коефициент на запълване, Userguide -> стр. 669

    while(1){
        __bis_SR_register(CPUOFF | GIE); //Изключи µPU за понижаване на консумацията
    }
}
```

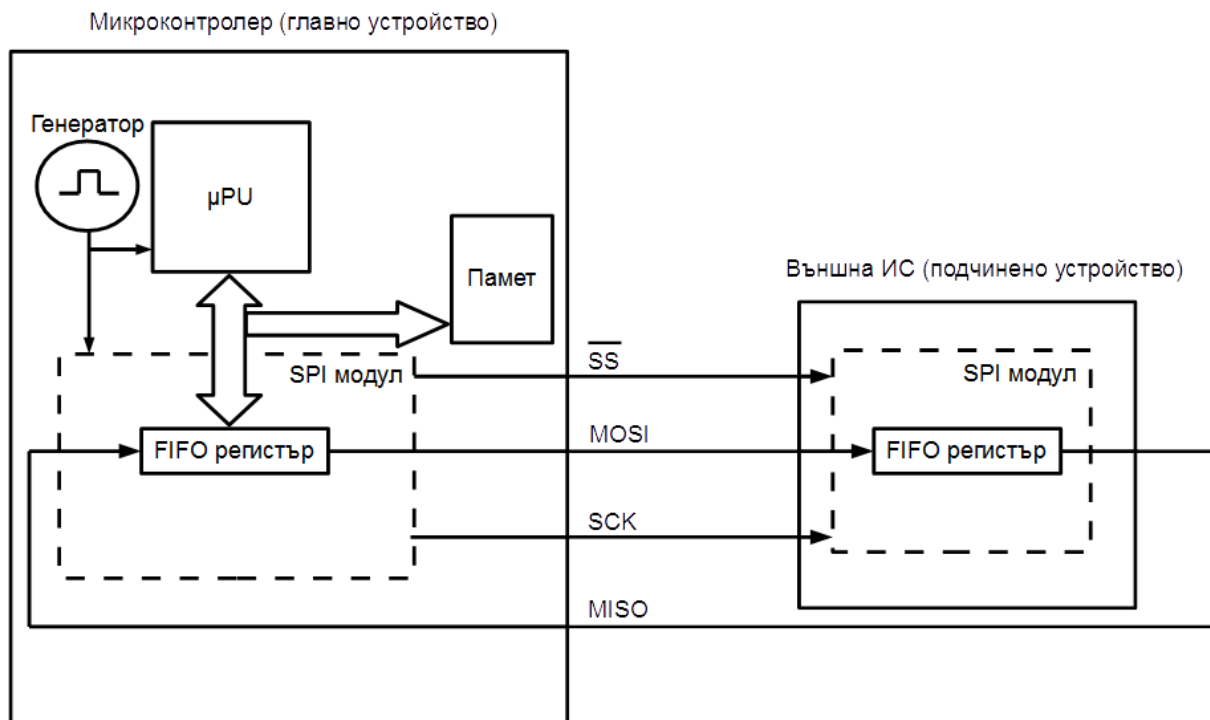
Лабораторно упражнение №5

“Изследване на сериен интерфейс SPI на микроконтролера MSP430FR6989”

5.1. Въведение. Често използван интерфейс за комуникация между две и повече ИС в рамките на една вградена система е SPI интерфейсът, създаден от фирмата Motorola. Това е сериен, синхронен интерфейс, използващ до 4 проводника за обмен на данни (и една маса). Те са:

- MOSI (Master output/slave input) – изход на главното устройство/вход на подчиненото.
- MISO (Master input/slave output) – вход на главното устройство/изход на подчиненото.
- SCK (SlaveClock) – синхронизиращ тактов сигнал, изработван от главното и подаван към подчиненото устройство. Този сигнал определя интерфейса като „синхронен“.
- SS (SlaveSelect) – сигнал за избор на подчинено устройство.

При SPI главното (или още – master) устройство задава синхронизиращия тактов сигнал, спрямо който се предават данните. Със всеки тактов импулс се предава един бит информация в даден момент от времето. Това определя интерфейса като „сериен“. Подчиненото устройство (или още – slave) приема правилно данните благодарение на този сигнал. Генератор на такт има само в главното устройство. На **фиг. 5.1** е дадено едно типично свързване на две ИС по SPI интерфейс. От нея се вижда, че предаването на данни става посредством преместващи регистри свързани в една обща верига (chain). Зареждането на данните в преместващия регистър се извършва от микропроцесора и е паралелно.



Фиг. 5.1

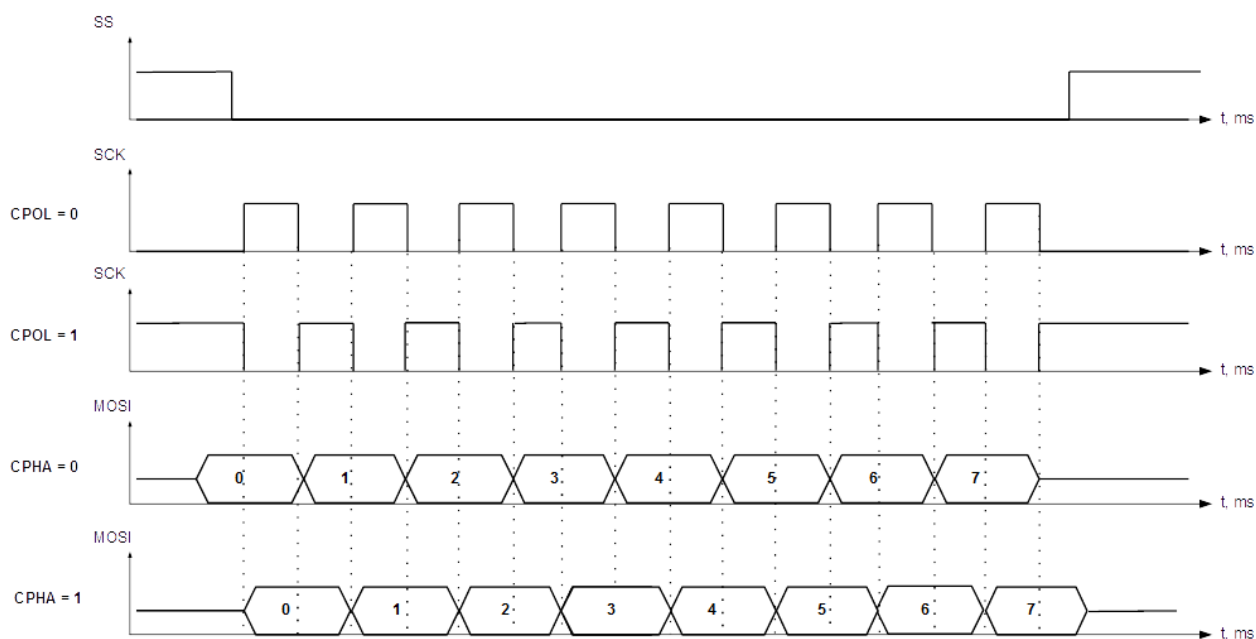
Протоколът е сравнително прост – реализира 4 режима (SPI modes) на обмен на данни. В Таблица 1 са дадени тези режими в зависимост от полярността (CPOL) и фронта на тактовия сигнал (или още – фазата на данните спрямо тактовия сигнал, CPHA).

CPOL	CPHA	Режим
0	0	0
0	1	1
1	0	2
1	1	3

Таблица 1

CPOL – определя логическото състояние на проводника, осигуряващ тактов сигнал (SCK), когато по интерфейса няма обмен на данни.

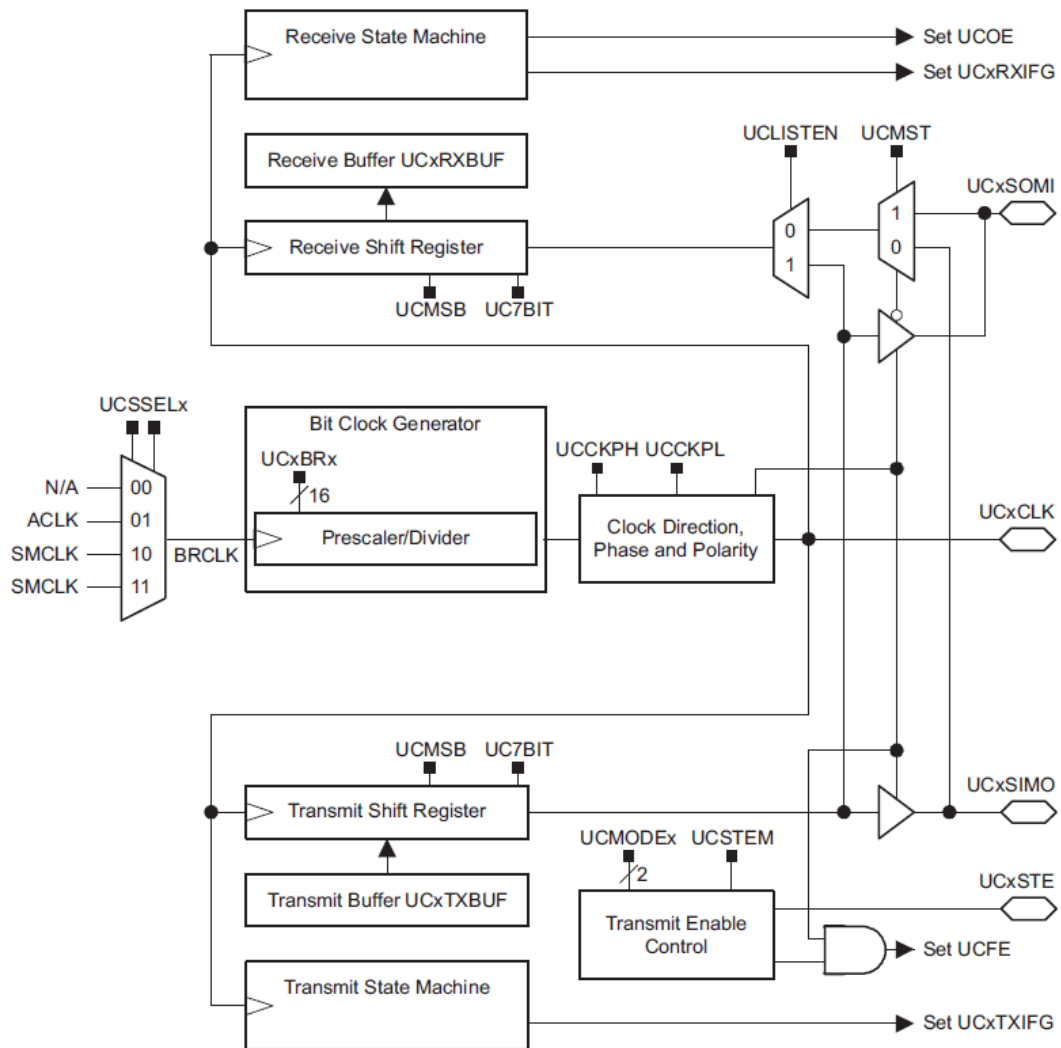
CPHA – определя фронта на тактовия сигнал, по който ще се предават данните. Оригинално тези означения идват от имената на битовете на конфигуриращи регистри за SPI модул на микроконтролера MC68HC11 на Motorola.



5.2. SPI модул на MSP430FR6989. На фиг. 5.2 е дадена блокова схема на SPI модула. От нея е видно, че данните се записват от микропроцесора в паралелния регистър UCxTXBUF, след което се прехвърлят в преместващия регистър „Transmit Shift Register”. Тогава данните се изместват бит по бит извън микроконтролера, синхронно спрямо сигнала UCxCLK, на UCxSIMO (или още MOSI) извода. Тактовият сигнал на модула и на интерфейса може да се вземе от системните тактови сигнали ACLK или SMCLK. Той се подава на делител на честота. Приемането на данни от външни схеми става на извод UCxSOMI (или още MISO), на който е свързан входа на приемащия преместващ регистър „Receive Shift Register”. Той прехвърля приетите данни в паралелния регистър UCxRXBUF, който е достъпен за четене от микропроцесора. Интересен факт за MSP430FR6989 е, че SPI, I2C и UART всъщност са един модул (обобщен с

названието eUSCI – enhanced Universal Serial Communication Interface, от англ. ез. усложнен универсален сериен комуникационен интерфейс), който може да се конфигурира според нуждите на един от трите интерфейса.

Забележете, че модулът има два FIFO регистъра, т.е. четенето и записването на данни става на различни адреси от SPI модула.



Фиг. 5.2

5.3. Задачи за изпълнение.

Внимание! Уверете се, че синият джъмпер на носещата платка е премахнат преди да продължите с упражнението!

5.3.1. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която изпраща числото 0x75 по SPI интерфейса в режим 0 (CPOL = 0 и CPHA = 0). Да се снимат осцилограмите на изводи MOSI (P3.1), SCK (P3.0) и SS (P4.3).

Забележка: SPI модулът на MSP430FR6989 притежава SS извод, който обаче е наименуван STE. Неговото активно ниво може да се променя с битово поле UCMODEx от UCAxCTLW0.

5.3.2. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която изпраща числото 0x75 по SPI интерфейса в режим 1 (CPOL = 0 и CPHA = 1). Да се снимат осцилограмите на изводи MOSI (P3.1), SCK (P3.0) и SS (P4.3).

5.3.3. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която изпраща числото 0x75 по SPI интерфейса в режим 2 (CPOL = 1 и CPHA = 0). Да се снимат осцилограмите на изводи MOSI (P3.1), SCK (P3.0) и SS (P4.3).

5.3.4. Да се създаде нов проект на Desktop-а в отделна директория и да се въведе програмата на C, която изпраща числото 0x75 по SPI интерфейса в режим 3 (CPOL = 1 и CPHA = 1). Да се снимат осцилограмите на изводи MOSI (P3.1), SCK (P3.0) и SS (P4.3).

Задача 5.3.1. / 5.3.2. / 5.3.3. / 5.3.4.

```
#include "io430.h"

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    //Конфигурирай мултиплексора за работа с извод P4.3 като UCB1STE (SS)
    P4SEL1 |= 0x08;
    P4SEL0 &= ~0x08;

    //Конфигурирай мултиплексора за работа с извод P3.0 като UCB1CLK (SCK)
    P3SEL1 &= ~0x01;
    P3SEL0 |= 0x01;
    LCDCPCTL2 &= ~0x04; //Изключи сигналът S34 на LCD модула

    //Конфигурирай мултиплексора за работа с извод P3.1 като UCB1SIMO (MOSI)
    P3SEL1 &=~ 0x02;
    P3SEL0 |= 0x02;
    LCDCPCTL2 &= ~0x02; //Изключи сигналът S33 на LCD модула

    UCB1CTLW0 |= UCSWRST;           //Задръж SPI модула в ресет

    // SPI master | режим - синхронен интерфейс | формат на данните MSB първи |
    // 4-изводен SPI (MISO не се използва в л. упр.) със SS активно ниво - ниско | STE извод
    //използван като SS сигнал | такт на SPI модул -> от SMCLK
    UCB1CTLW0 |= (UCMST | UCSYNC | UCMSB | UCMODE_2 | UCSTEM | UCSSEL_3);

    UCB1BRW = 0x02;                // Раздели входната честота за SPI модула на 2

    UCB1CTLW0 ??? = ???;           //Задай полярност на SCK
    //стр. 799 от Userguide
    UCB1CTLW0 ??? = ???;           //Задай фаза на MOSI
    //стр. 799 от Userguide

    UCB1CTLW0 &= ~UCSWRST;         //Пусни SPI модула от ресет

    while(1){
        ??? = ???; //Запиши числото, което трябва да се изпрати
        //в съответния изходен (TX) регистър
        //стр. 801 от Userguide
        __delay_cycles(50);
    }
}
```

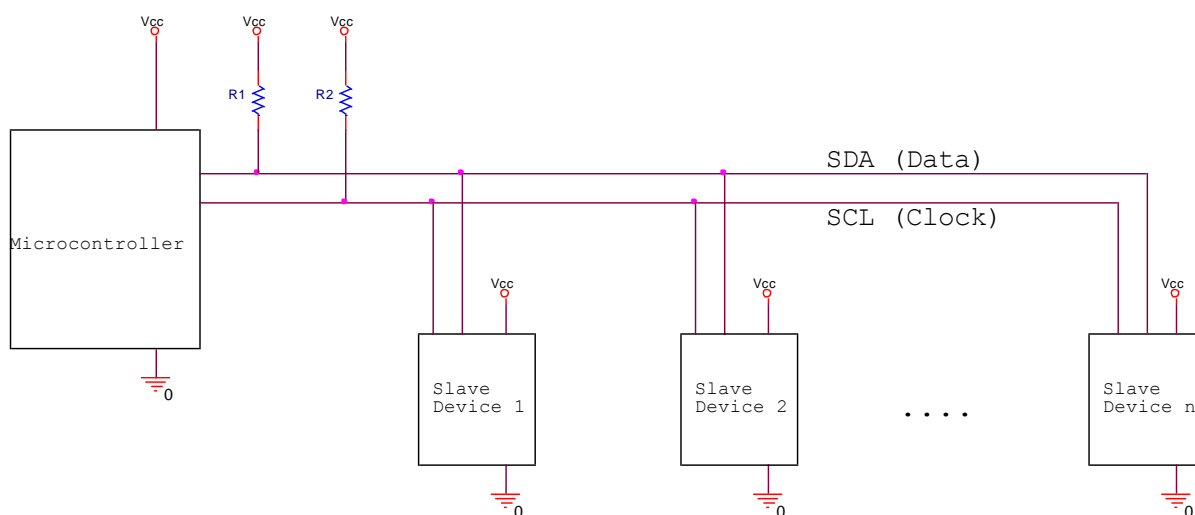

Лабораторно упражнение №6

“Изследване на сериен интерфейс I²C на микроконтролера MSP430FR6989”

6.1 Въведение. I²C е синхронен сериен интерфейс, който използва два (+ 1 маса) проводника за предаване на информация между два и повече чипа. Името му идва от IIC – Inter-Integrated Circuit и е създаден от Philips през 1982 г. От 2006 година използването на протокола за този интерфейс е безплатно. Платено е единствено запазването на уникален адрес за чип от даден вид, използващ I²C (това засяга само фирмите-производителки на чипа).

Връзката между два чипа се реализира на принципа master-slave, т.е. има една главна интегрална схема (master) и една подчинена (slave) такава. Обикновено master чипа е микроконтролерът в системата, а slave е периферен чип, с който ще се осъществява връзката. Съществува и режим multi-master, при който към I²C интерфейса са свързани два или повече главни (master) чипа.

На **фиг. 6.1** е показана блокова схема на система, използваща I²C интер-



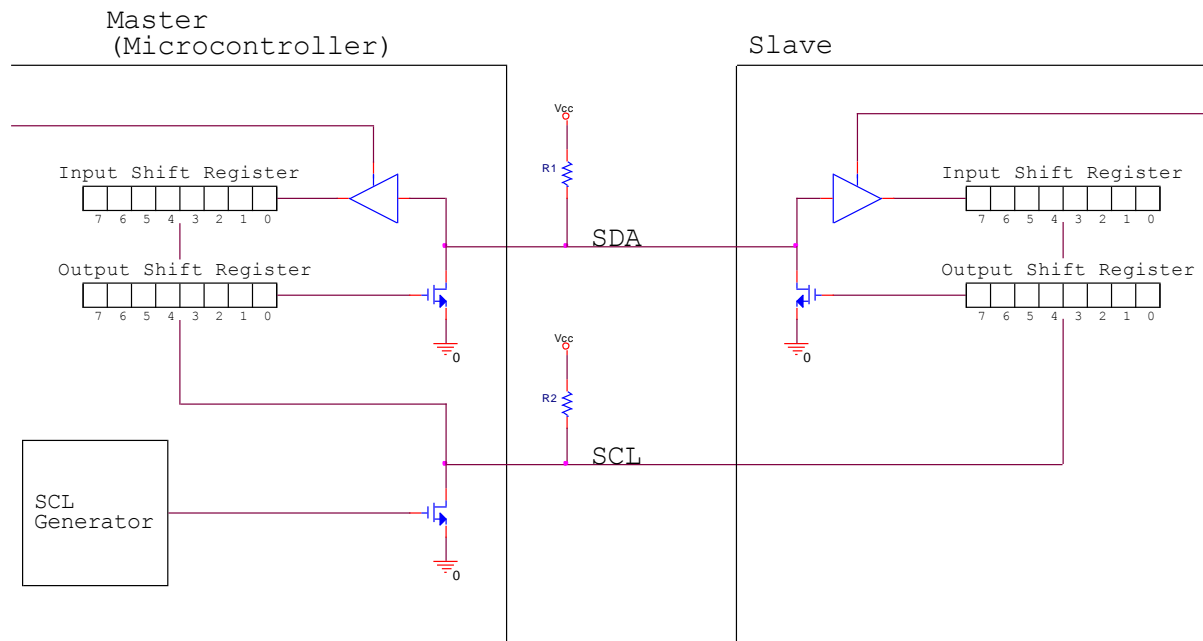
Фиг. 6.1

фейса. Предаването на информацията се осъществява посредством сигналите:

- SDA – данните, предавани серийно по този проводник.
- SCL – тактовият сигнал, по който е синхронизирано изпращането бит по бит на данните.

Към двата проводника са свързани изтеглящи (pull-up) резистори към захранването на системата Vcc. Те са необходими, защото предаването на информация е двупосочно – от микроконтролера към подчинената схема и от подчинената схема към микроконтролера. Това означава, че микроконтролерът ще използва изводът си, свързан към SDA, веднъж като вход и веднъж като изход. Аналогично – подчиненото устройство ще използва SDA като вход, когато иска да приеме информация и след това като изход, когато иска да предаде информация (**фиг. 6.2**). Тази конфигурация води до една теоретична конфликтна ситуация, в която ако и двата чипа са установили изводите си SDA като изходи и единият е в логическа 1, а другият – в логическа 0, то ще се получи късо съединение между захранващия извод Vcc и масата GND. За да се избегне тази

възможност се използват изходи с отворен дрейн (или колектор), които изискват режимен pull-up резистор. SCL също включва такъв резистор, защото протокола I²C дефинира ситуация, в която някой от чиповете задържа тактовия сигнал в логическа 0.



Фиг. 6.2

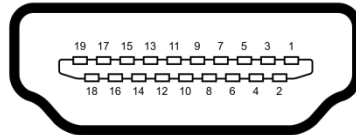
Аналогични на I²C са интерфейсите:

- SMBus – създаден от Intel през 1995. Дефинира по-строго изискванията за ниво и времеви параметри (timings) на сигнала.
- I²S – използва се за предаване на цифров звуков сигнал. Съдържа един допълнителен проводник (Word Select), указващ изпращаният байт на кой канал принадлежи (при стерео предаване – ляв или десен).

I²C интерфейса се характеризира със следните скорости на предаване:

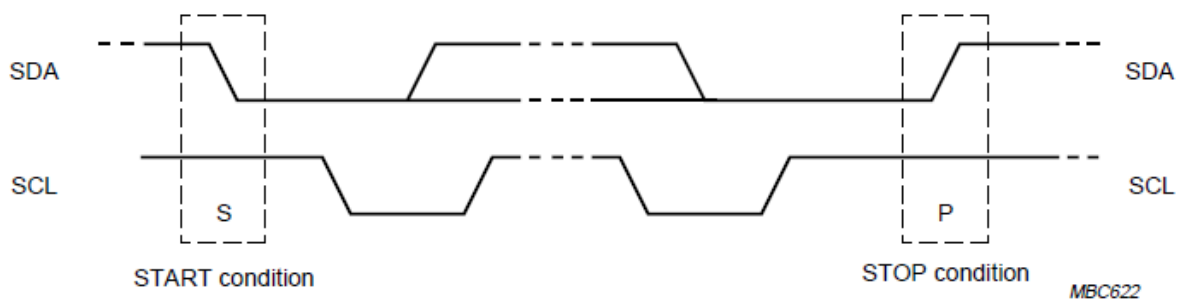
- Оригинална версия от 1982 – 100 kbit/s.
- Версия 1 (1992) – 400 kbit/s (Fast mode).
- Версия 2 (1998) – 3.4 Mbit/s (High-speed mode).
- Версия 3 (2007) – 1 Mbit/s (Fast mode plus).
- Версия 4 (2012) – 5 Mbit/s (Ultra Fast mode).

I²C интерфейса се включва в интегрални схеми с различни функции във вградените системи. АЦП, ЦАП, EEPROM, цифрови потенциометри, температурни датчици, хол датчици и т.н. са само малка част от приложението му. Въпреки че този интерфейс се използва главно за комуникация между ИС в рамките на една вградена система, то не липсват примери за приложения и в комуникацията между две отделни устройства. Например всеки персонален компютър използва I²C като част от HDMI интерфейса **фиг. 6.3**. С негова помощ се „опознават“ поддържаните видео формати от изобразяващото устройство (монитор, телевизор, прожектор и други).



Фиг. 6.3-HDMI куплунг. I²C на извод 15 – SCL и извод 16 – SDA.

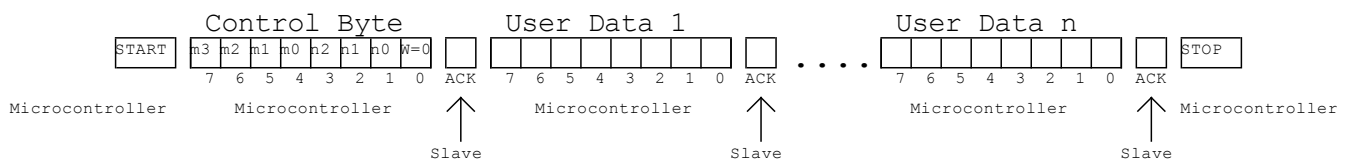
I²C комуникацията се осъществява по стандартизиран протокол. Най-общо казано обменът на данни започва с условие СТАРТ, продължава с трансфер на данни и завършва с условие СТОП. Условието старт и стоп са специални събития, реализирани чрез комбинации на логически нива и фронтите на SCL и SDA линиите, които са уникални и се различават от данните. На **фиг. 6.4** са показани едно старт и стоп условие. Старт условие се генерира, когато SDA



Фиг. 6.4

линията премине в ниско ниво, докато SCL линията е във високо. Стоп условие се генерира, когато SDA линията премине във високо ниво, докато SCL линията е във високо. Между старт и стоп условията се изпращат потребителските данни. Всеки бит от тях трябва да запази своето състояние, докато SCL е във високо ниво (в противен случай ще се генерира старт или стоп условие и трансферът ще се наруши), т.е. трансферът на данни се осъществява по ниво.

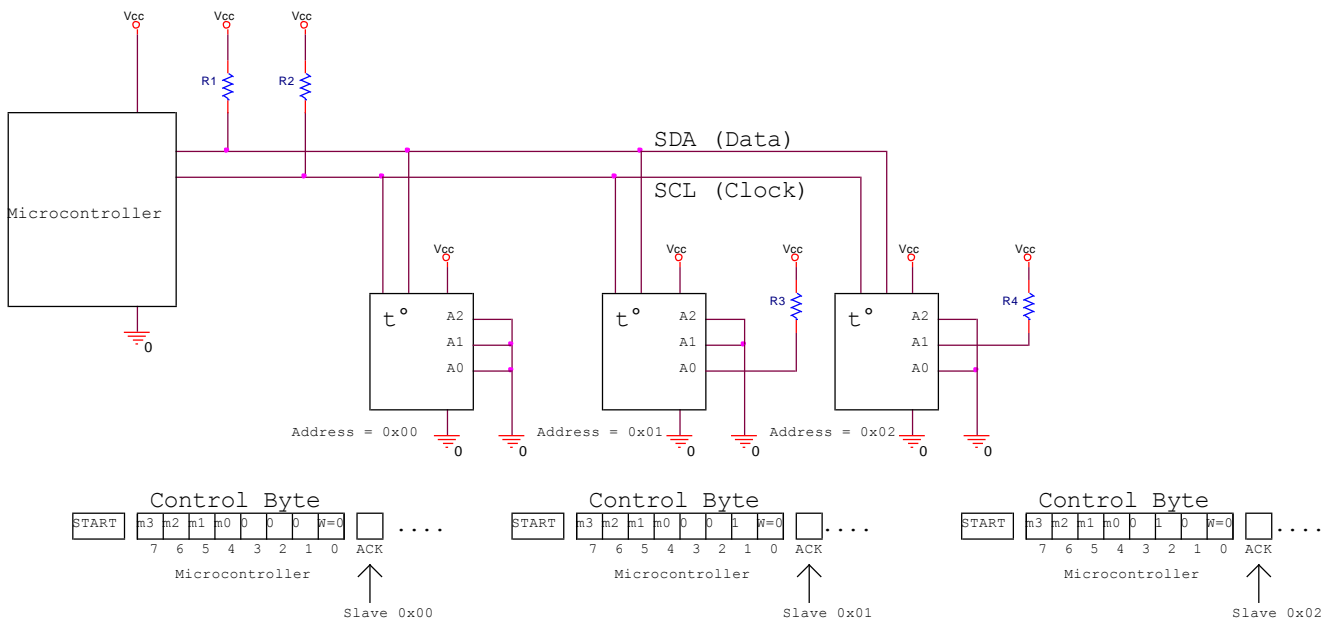
Форматът на данните при **запис** от микроконтролер в подчинена ИС е показан на **фиг. 6.5**.



Фиг. 6.5 – Запис в подчинена ИС.

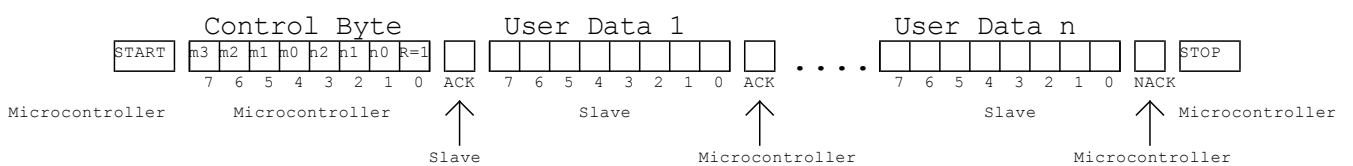
Микроконтролерът генерира СТАРТ условие, след което изпраща контролен байт. Този байт достига до всички подчинени устройства. Старшата тетрада (битове m0 ÷ m3) съдържа контролен код, част от 7-битов адрес, който е различен за различните подчинени чипове – трябва да се провери datasheet-а им. Следват три бита, които указват младшата част на 7-битовия адрес на подчиненото устройство (битове n0 ÷ n2). Благодарение на тях към една I²C шина могат да се свържат повече от един чипове от един и същи вид (на **фиг. 6.6** е показано свързване на три еднакви температурни датчика, чиито адреси са зададени хардуерно с изводи 1, 2 и 3). Последният бит от контролния байт е

четене/запис (R / !W) и при запис трябва да е 0. Ако на I²C шината има устройство със зададения адрес (битове m0 ÷ m3 и n0 ÷ n2), то трябва да отговори с изпращане на един бит, наречен ACK (acknowledge) или още - потвърждение. Неговото ниво е логическа 0. След това се изпращат потребителските данни User Data 1 ... User Data n и най-накрая се генерира условие стоп, с което трансферът приключва.



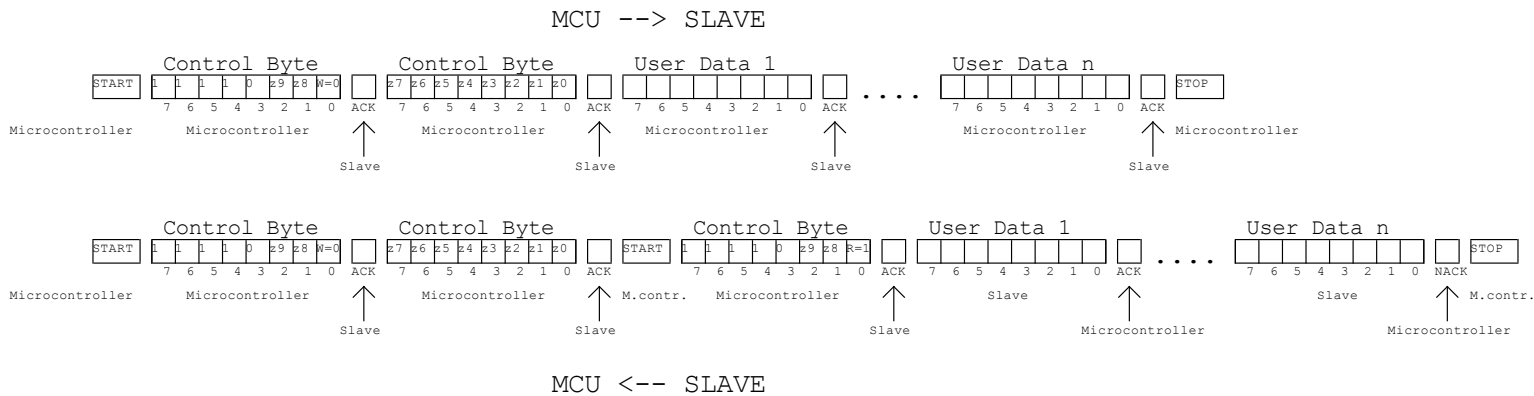
Фиг. 6.6

Трансферът на данните от подчинена ИС към микроконтролера се определя като **четене** и форматът е показан на **фиг. 6.7**. Да се обърне внимание на предавателя и приемника под всеки байт на **фиг. 6.5** и **фиг. 6.7**! Ако микроконтролерът иска да спре приемането на данни, той трябва да генерира NACK бит (No Acknowledge, логическа 1) и след това условие СТОП.



Фиг. 6.7

Отделените седем бита m0 ÷ m3 и n0 ÷ n2 позволяват на една I²C шина да се свържат до 2⁷ = 128 чипа (минус един бродкаст адрес и няколко за бъдещо ползване). Версия 1 дефинира някои леки промени в протокола, които позволяват адресиране на 2¹⁰ = 1024 (т.е. 10-битов адрес) чипа. Те са показани на **фиг. 6.8**. Новото тук е фиксираното число 11110 в контролния байт. То указва, че ще се използва 10-битово адресиране. Следват битове z9 ÷ z0, които са 10-битовия адрес на подчиненото устройство. Да се обърне внимание на двойното изпращане на условие СТАРТ и редуването на W=0 и R=1 при четене (MCU ← SLAVE).

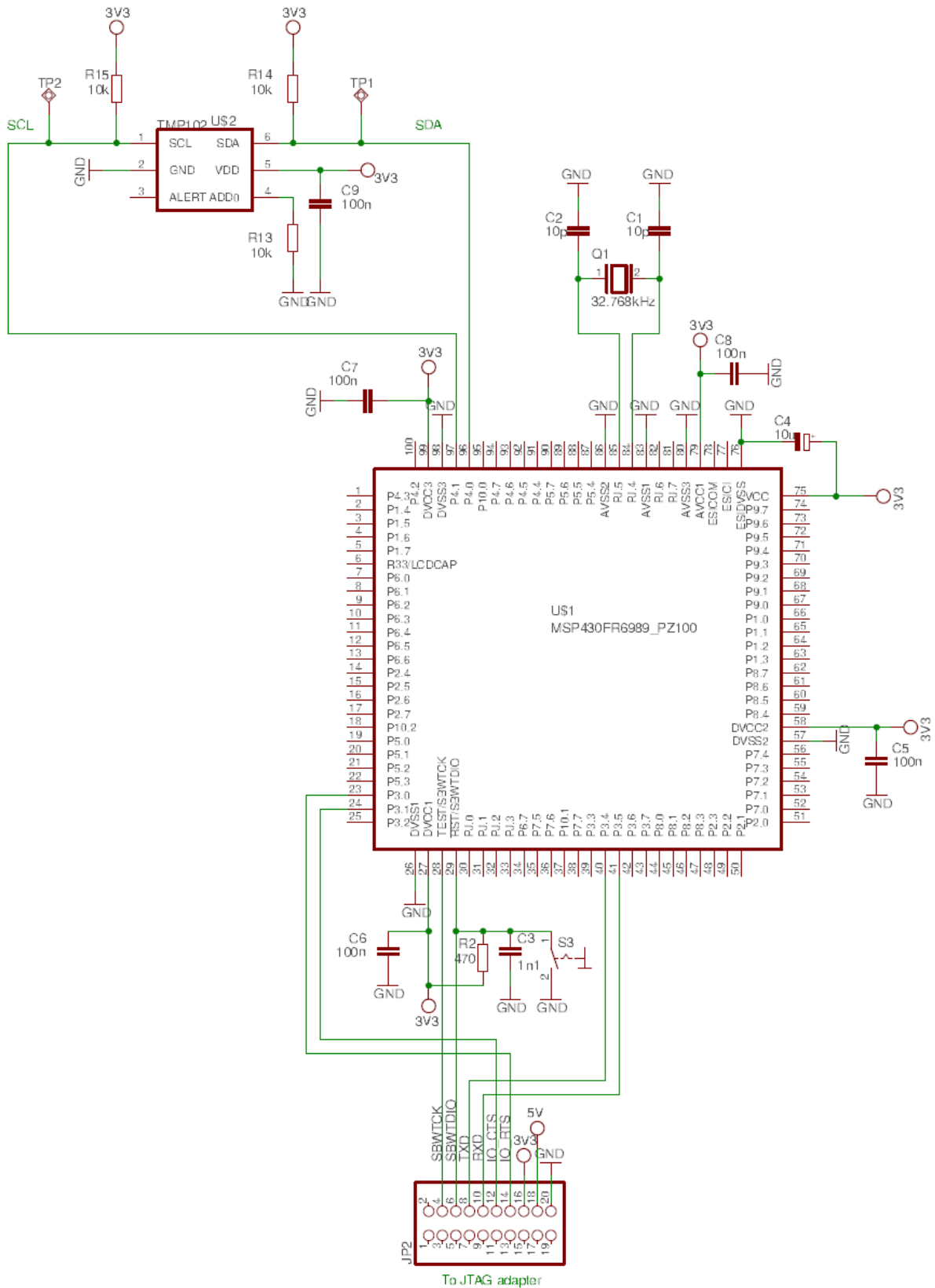


Фиг. 6.8

6.2 Задачи за изпълнение.

6.2.1. Да се създаде нов C проект на Desktop-а в отделна директория и да се напише програма, която чете данни от температурния датчик TMP102 посредством I²C интерфейс. Принципната схема е показана на **фиг. 6.9**. Да се наблюдават SCL и SDA линиите с осцилоскоп и да се свалят осцилограмите.

6.2.2. Да се създаде нов C проект на Desktop-а в отделна директория, която реализира запис по I²C интерфейса. За целта да се запише произволно число в един от регистрите T_{LOW} или T_{HIGH} на температурния датчик TMP102. Да се снимат осцилограмите. Да се сравнят с осцилограмите от предишната задача. Да се обяснят разликите.



Фиг. 6.9

Задача 6.2.1

```
#include <stdint.h>
#include "io430.h"

#define TMP102ADDR ??? //Дефиниране на TMP102 адреса, Datasheet на TMP102 →стр. 10

// Тактов генератор DCO = 8MHz, Userguide -> стр. 104
void init()
{
    CSCTL0_H = 0xA5;
    CSCTL1 |= DCORSEL; //DCORSEL = 1
    CSCTL1 &= ~0x0E; //Занули DCOFSEL битовете
    CSCTL1 |= DCOFSEL_3; //Задай DCOFSEL битовете = 011b = 3h
    CSCTL2 = 0x00; //Занули всички битове от регистъра CSCTL2
    CSCTL2 |= (SELA_1 | SELS_3 | SELM_3); //ACLK = VLOCLK; SMCLK = DCO; MCLK = DCO
    CSCTL3 = 0x00; //Занули всички битове от регистъра CSCTL3
    CSCTL3 |= (DIVA_0 | DIVS_0 | DIVM_0); //Всички делители на тактовите сигнали ÷ 1
}

void init_I2C()
{
    //Превключи мултиплексора от P4.0 и P4.1 към I2C модула
    P4SEL0 ???= ???; //MSP430FR6989 Datasheet стр. 103
    P4SEL1 ???= ???; //MSP430FR6989 Datasheet стр. 103
    LCDCPCTL0 ???= ???; //S3(P4.0) = 0, S2(P4.1) = 0; MSP430FR6989 Userguide стр. 945

    UCB1CTLW0 |= UCSWRST; //Задръж I2C модула в reset, докато се конфигурира
    UCB1CTLW0 |= (UCMODE_3 | UCMST); //Избери режим I2C | роля - master
    UCB1CTLW0 |= UCSSEL_3; //Източник на тактов сигнал за I2C модула е SMCLK
    UCB1BRW = 0x50; //SMCLK / 80 = 100 kbps
    UCB1I2CSA = TMP102ADDR; //Укажи адреса на TMP102 датчика (slave устройството)
    UCB1CTLW0 &= ~UCSWRST; //Изведи модула от reset
}

void init_TMP102()
{
    UCB1CTLW0 |= (UCTXSTT | UCTR); //Генерирай start условие, режим предавател
    __delay_cycles(500); //Изчакай предаването на start условието
    UCB1TXBUF = 0x00; //Запиши данните, които ще се изпратят
    while(!(UCB1IFG & UCTXIFG0)){ } //Изчакай данните от UCB1TXBUF да се изпратят
    UCB1CTLW0 |= UCTXSTP; //Генерирай stop условие
    __delay_cycles(1000);
}

void main( void )
{
    uint8_t lsb;
    uint16_t msb;
    volatile uint16_t result;

    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    init();
    init_I2C();
    init_TMP102();

    while(1)
    {
```

UCB1CTLW0 &= ~????; //Режим приемник, Userguide → стр. 828

```
UCB1CTLW0 |= UCTXSTT; //Генерирай start условие
while(!(UCB1IFG & UCRXIFG0)){ //Изчакай първия байт да пристигне в UCB1RXBUF
lsb = UCB1RXBUF; //Запиши първия байт в променливата lsb
while(!(UCB1IFG & UCRXIFG0)){ //Изчакай втория байт да пристигне в UCB0RXBUF
msb = UCB1RXBUF; //Запиши втория байт в променливата msb
UCB1CTLW0 |= UCTXSTP; //Генерирай stop условие
```

```
result = (msb<<8) | lsb; //Обедини младшата и старшата част на резултата в една променлива
result >>= 4; //Младшите 4 бита са винаги 0, затова ги премахни
__delay_cycles(8000); //Сложете точка на прекъсване тук, ако искате да прочетете резултата
//като целочислена стойност
```

```
}
}
```

Задача 6.2.2

```
#include <stdint.h>
#include "io430.h"
```

```
#define TMP102ADDR ??? //Дефиниране на TMP102 адреса, Datasheet на TMP102 →стр. 10
```

```
// Тактов генератор DCO = 8MHz, Userguide -> стр. 104
```

```
void init()
{
    CSCTL0_H = 0xA5;
    CSCTL1 |= DCORSEL; //DCORSEL = 1
    CSCTL1 &= ~0x0E; //Занули DCOFSEL битовете
    CSCTL1 |= DCOFSEL_3; //Задай DCOFSEL битовете = 011b = 3h
    CSCTL2 = 0x00; //Занули всички битове от регистъра CSCTL2
    CSCTL2 |= (SELA_1 | SELS_3 | SELM_3); //ACLK = VLOCLK; SMCLK = DCO; MCLK = DCO
    CSCTL3 = 0x00; //Занули всички битове от регистъра CSCTL3
    CSCTL3 |= (DIVA_0 | DIVS_0 | DIVM_0); //Всички делители на тактовите сигнали ÷ 1
}
```

```
void init_I2C()
```

```
{
    //Превключи мултиплектора от P4.0 и P4.1 към I2C модула
    P4SEL0 ???= ???; //MSP430FR6989 Datasheet стр. 103
    P4SEL1 ???= ???; //MSP430FR6989 Datasheet стр. 103
    LCDCPCTL0 ???= ???; //S3(P4.0) = 0, S2(P4.1) = 0; MSP430FR6989 Userguide стр. 945
```

```
UCB1CTLW0 |= UCSWRST; //Задръж I2C модула в reset, докато се конфигурира
UCB1CTLW0 |= (UCMODE_3 | ???); //Избери режим I2C | роля - master
UCB1CTLW0 |= UCSSEL_3; //Източник на тактов сигнал за I2C модула е SMCLK
UCB1BRW = 0x50; //SMCLK / 80 = 100 kbps
UCB1I2CSA = TMP102ADDR; //Укажи адреса на TMP102 датчика (slave устройството)
UCB1CTLW0 &= ~UCSWRST; //Изведи модула от reset
}
```

```
void init_TMP102()
```

```
{
    UCB1CTLW0 |= (UCTXSTT | UCTR); //Генерирай start условие, режим предавател
    __delay_cycles(500); //Изчакай предаването на start условието
    UCB1TXBUF = 0x03; //Указател към регистър THIGH на датчика TMP102
    while(!(UCB1IFG & UCTXIFG0)){ //Изчакай данните от UCB1TXBUF да се изпратят
    UCB1CTLW0 |= UCTXSTP; //Генерирай stop условие
}
```



```

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    init();
    init_I2C();
    init_TMP102();

    while(1)
    {
        UCB1CTLW0 |= (UCTXSTT | ???); //Генерирай start, режим предавател Userguide → стр. 828
        __delay_cycles(500);          //Изчакай предаването на start условието
        UCB1TXBUF = 0x03;             //Задай адрес 0x03 на TMP102 вътрешния указател
        while(!(UCB1IFG & UCTXIFG0)){ } //Изчакай данните от UCB1TXBUF да се изпратят
        UCB1TXBUF = 0xF0;             //Произволно число в MSB на THIGH регистъра
        while(!(UCB1IFG & UCTXIFG0)){ } //Изчакай данните от UCB1TXBUF да се изпратят
        UCB1TXBUF = 0x0F;             //Произволно число в LSB на THIGH регистъра
        while(!(UCB1IFG & UCTXIFG0)){ } //Изчакай данните от UCB1TXBUF да се изпратят

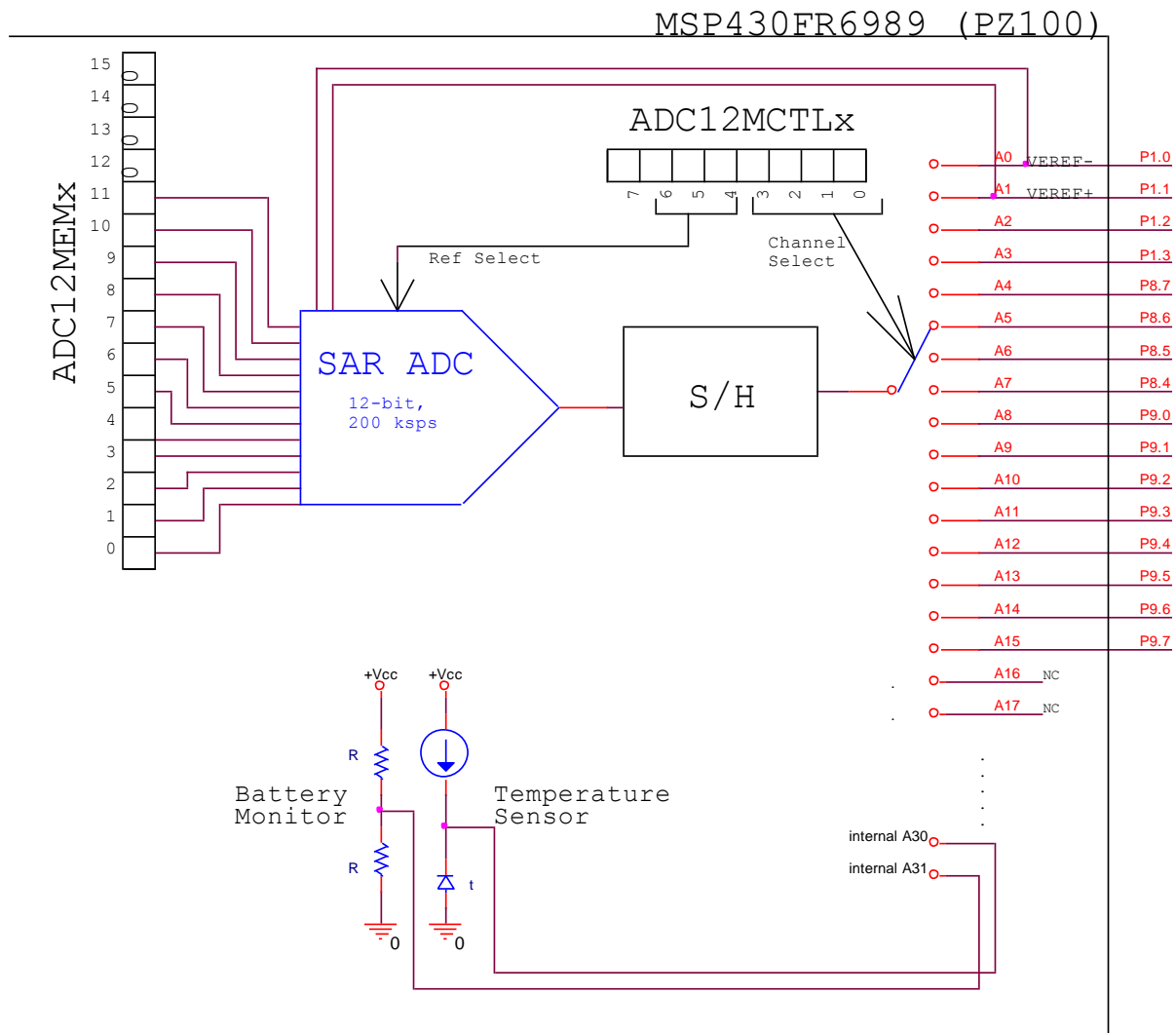
        UCB1CTLW0 |= UCTXSTP;         //Генерирай stop условие
        __delay_cycles(8000);
    }
}

```

Лабораторно упражнение №7

“Аналогово-цифров преобразувател на MSP430FR6989 и комуникационен модул UART (RS-232)”

7.1. Въведение. Микроконтролерът MSP430FR6989 притежава вграден аналогово-цифров SAR преобразувател с 12-битова разредност, бързодействие 200 ksp/s и 16 входни канала (фиг. 7.1).



Фиг. 7.1

Потребителят може да измерва напрежение с до 16 извода на микроконтролера. Други 2 канала са вътрешни и са за измерване температурата на чипа (с помощта на интегриран диод), за измерване на захранващото напрежение (с помощта на интегриран резисторен делител) и останалите не се използват. При включване на външни еталонни източници за горен и долен праг на измерването потребителят ще може да измерва напрежение само на 14 извода.

Преди даден извод да бъде използван от АЦП-то, трябва да се конфигурира *мултиплексорът*, превключващ извода измежду различните периферни модули (всеки извод може да бъде използван от много периферни модули, но в даден

момент само един е включен на него). Това става чрез регистър PxSEL0 и PxSEL1.

АЦП може да работи в 4 режима, задавани от битовете ADC12CONSEQx на регистъра ADC12CTL1:

- един канал, едно измерване;
- много канали, по едно измерване на всеки;
- един канал, много измервания;
- много канали, много измервания на всеки.

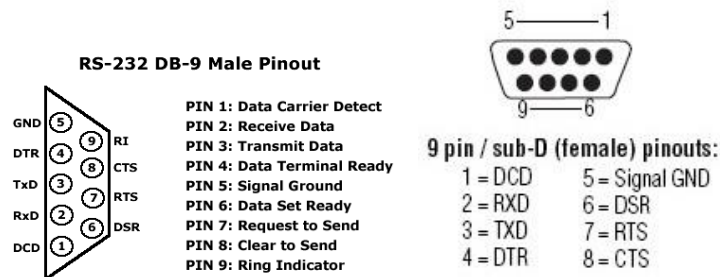
За всеки един канал има по един MCTL регистър, който указва източниците на еталонно напрежение за конкретния канал, както и дали той е първи или последен при многоканално измерване.

АЦП-то може да използва *еталонния източник* на напрежение, вграден в микроконтролера. Неговата стойност може да се избере от напреженията: 1.2 V, 2.0V, 2.5 V. Това може да стане, като се установи бит 0 от REFCTL0 в единица и с REFVSEL битовете се избере една от трите стойности. Да се разгледа MSP430FR6989 User's Guide-а за желаната комбинация на REFVSEL.

Тактовата честота на модула е един от източниците SMCLK, MCLK, ACLK, MODCLK и може да се избере чрез битовете ADC12SSELx от регистъра ADC12CTL1.

Тази честота допълнително може да бъде *разделена* $1 \div 512$ пъти чрез ADC12DIVx битове от регистъра ADC12CTL1 и ADC12PDIVx битове от регистъра ADC12CTL2.

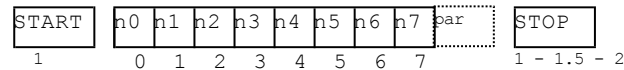
В настоящото упражнение се използва RS232 интерфейс, наречен още UART, за осъществяване на връзка между микроконтролера и персонален компютър. Това е напрежителен, сериен, асинхронен интерфейс. На **фиг. 4.2** са показани най-често използваните куплунги с 9 извода (има и с 25 извода) - мъжки и женски DB9.



Фиг. 7.2

При повечето приложения микроконтролерите използват само три проводника от този куплунг – това са TxD (извод 3) за предаване, RxD (извод 2) за приемане и маса (извод 5). Логическите нива се представят с отрицателни и положителни напрежения за разлика от TTL нивата, където са само положителни. Логическата 0 (нарочана Space) е дефинирана като $\geq +5$ V, а логическата 1 (нарочана Mark) като ≤ -5 V. Максималните нива на напреженията трябва да са в границите ± 15 V, но чиповете, използващи този интерфейс трябва да са способни да издържат до ± 25 V. Приемниците трябва да регистрират и затихнали сигнали с амплитуда не по-малка от +3V и не по-голяма от -3 V (т.е. имат минимален $5 - 3 = 2$ V запас за шумоустойчивост).

Предаването на данни се осъществява байт по байт, всеки от който съдържа един стартов бит, даннови битове, бит проверка за грешки и един или повече стопови бита (**фиг. 7.3**). Изпращането на данните се извършва от LSB към MSB. Повечето инженери са свикнали да представят байтовете в MSB към LSB формат, затова може да се каже, че при наблюдение на осцилоскопа данните от UART изглеждат обърнати. Когато не се предават данни линиите се държат в логическа 1.



Фиг. 7.3

Битът „проверка за грешки“ може да бъде 4 вида:

- проверка по четност – битът се установява в 1 или 0, така че сумата от всички единици в изпращания байт да е четно число;
- проверка по нечетност – битът се установява в 1 или 0, така че сумата от всички единици в изпращания байт да е нечетно число;
- проверка с единица (mark parity) – битът е винаги единица;
- проверка с нула (space parity) – битът е винаги нула.

Пример: ако се изпраща числото 00010010 с проверка по четност, деветият бит ще е нула. Аналогично при проверката по нечетност числото 0011 0011 ще има единица за деветия бит.

Стоповите битове може да са :

- един;
- един и половина;
- два.

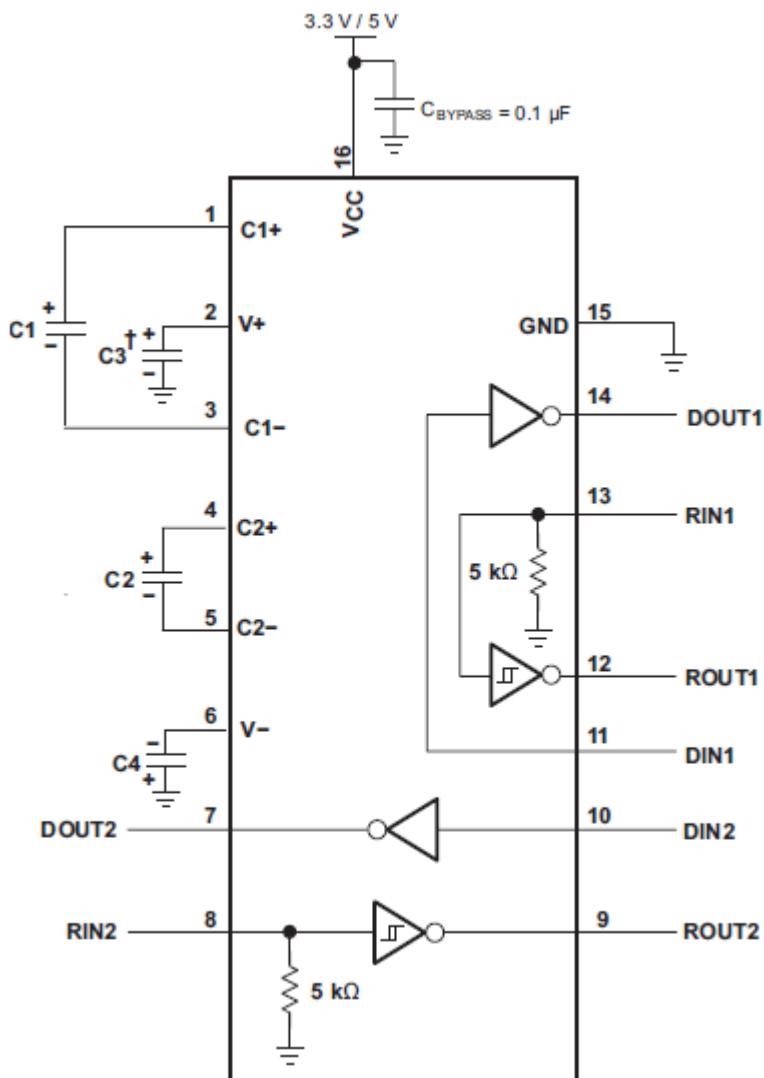
Един и половина и два стоп бита се използват при по-бавни приемачи устройства, които имат нужда от малко по-дълъг (във времето) стоп бит, за да обработят приетите данни.

Данновите битове може да са 5 ÷ 8. Изпращащото устройство предава данните на всеки свой нарастващ фронт на вътрешния му генератор, а приемащото устройство регистрира тези данни на всеки падащ фронт на вътрешния му генератор. Синхронизацията на вътрешните генератори на приемника и предавателя се осъществява по спадания фронт на старт бита (т.е. при първия преход $-15\text{ V} \rightarrow +15\text{ V}$). Така интерфейсът не се нуждае от допълнителен проводник за тактов сигнал и затова наричаме UART-а асинхронен интерфейс – разчита се на собствените генератори, вградени в предавателя и в приемника.

Скоростта на предаване се дава в бодове и 1 бод = 1 бит информация. В практиката се използват стандартизирани скорости, някои от които – 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200. По-високи скорости на обмен се осъществяват с USB-UART емулатори. Максималната дължина на кабела между две устройства е 10 m.

Понеже микроконтролерите се хранят с напрежения до 5 V, то те не могат да изработят със сигнали с амплитуда $\pm 5\text{ V} \div \pm 15\text{ V}$. Също така не биха могли да приемат такива сигнали без да се повредят. Затова се използват специални транслаторни схеми, които изработват необходимите сигнали и за интерфейса, и за микроконтролера. Една примерна такава ИС е показана на **фиг.**

7.4. Използван е MAX3232, а на схемата е дадена и вътрешната му структура. Инверторите са всъщност специални и осигуряват трансляцията на нивата. Използват се charge-pump постоянноточови преобразуватели за повишаване на положителните и изработване на отрицателните напрежения. Тези преобразуватели използват кондензаторите C1, C2, C3 и C4.



Фиг. 7.4

7.2 Задачи за изпълнение.

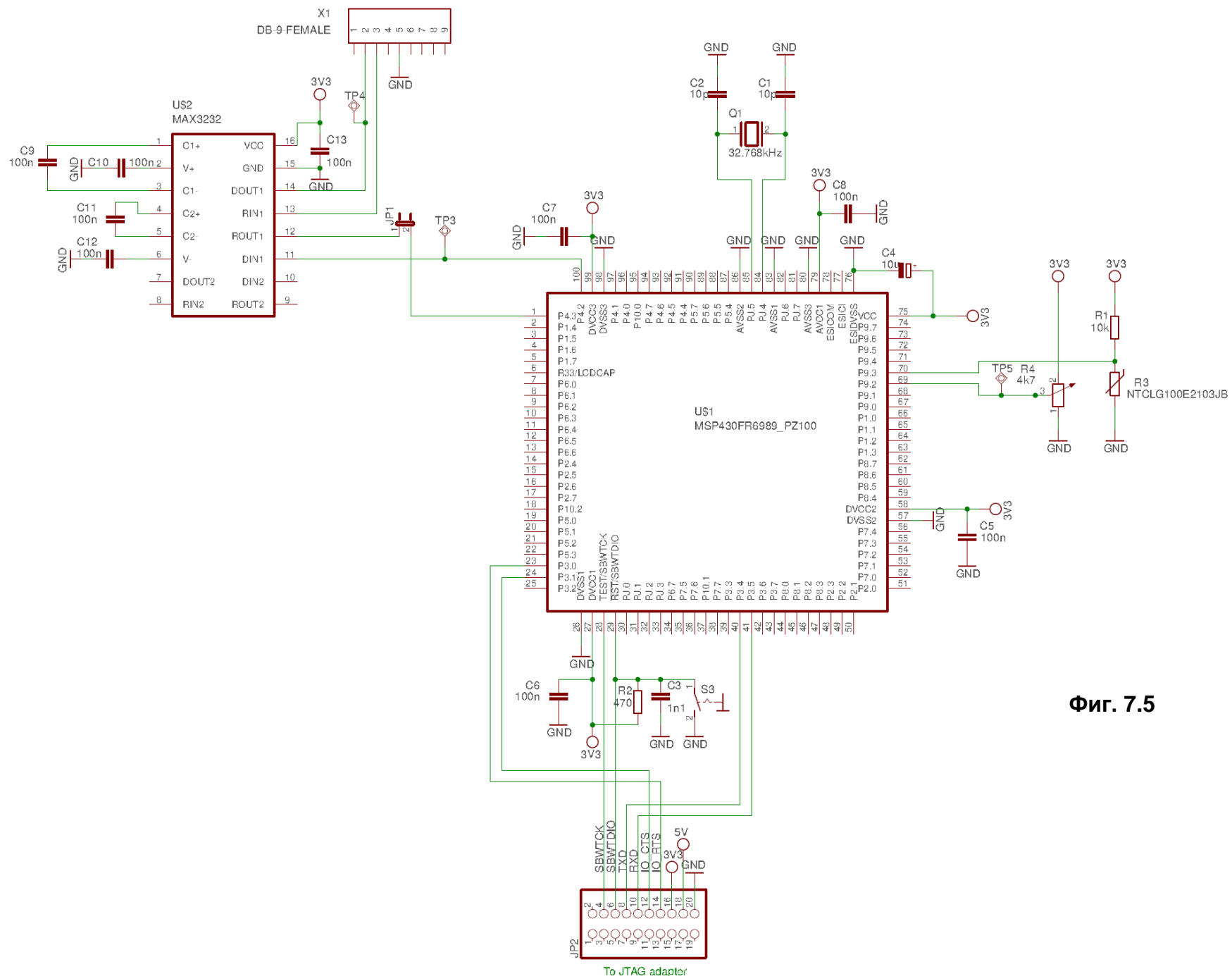
7.2.1. Да се създаде нов C проект на Desktop-а. Да се напише програма, която изпраща „Hello, World!” съобщение по UART (RS232) интерфейса.

За целта трябва да се копират файловете `uart.c`, `uart.h`, `uartstdio.c` и `uartstdio.h` (предоставени от ръководителя на упражнението) в директорията на проекта. След това в IAR → десен бутон върху top-level директорията → Add Files → указва се пътя до току-що добавените файлове → Open. В `main.c` се включва хедърния файл `uartstdio.h`. Използвайки API функциите от този файл реализирайте програмата си. За да видите резултата стартирайте терминална програма (Hyper Terminal, Tera Term, Real Term и т.н.) и настройте връзка **9600 бода, 8 даннови бита, без проверка по четност, 1 стопов бит**. Честотата на микропроцесора се настройва от библиотеката на 16 MHz.

»RS232 може да бъде виртуален и емулиран от USB дебъгера на демо платката. В <uartstdio.h> библиотеката се използва реалният (физическият) интерфейс.

7.2.2. Да се създаде нов C проект на Desktop-а в отделна директория. Да се свърже потенциометър към демо платката (**фиг.7.5**). Да се напише програма, която измерва напрежението в средната точка на потенциометъра. Да се наблюдава резултата от измерването посредством JTAG дебъгера. Ако за променливата `ADCResult` пише, че е `<unavailable>`, изключете оптимизациите на компилатора от Project → Options → Category: C/C++ Compiler → tab Optimizations → Level: сложете радио бутона на None. Изберете Rebuild на проекта.

7.3.3. Да се създаде нов C проект на Desktop-а в отделна директория и да се добави UARTStdio, както беше направено в задача 7.2.1. Да се напише програма, която измерва напреженията от потенциометъра, от термистора и от вградения резисторен делител на MSP430R6989 (който е свързан към захранване, маса и има коефициент на предаване 0,5). Данните да се изпращат от микроконтролера към компютъра по RS232. Нека данните от измерванията на трите канала се представят в необработен вид като цели числа в интервала 0 ÷ 4095.



Фиг. 7.5

Задача 7.2.1

```
#include "io430.h"
??? //Включи хедърния файл на UARTStdio, който сте добавили към проекта

void main( void )
{
    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    ???//Инициализирай UARTStdio функцията → отвори хедърния файл и виж прототипите на
        //достъпните функции

    while(1)
    {
        ??? //Извикай printf за MSP430 с произволен стринг → отвори хедърния файл и виж
            //прототипите на достъпните функции, извикайте printf( ) функцията, както бихте я
            //извикали в конзолна C програма на вашия PC

        __delay_cycles(16000000);
    }
}
```

Задача 7.2.2

```
#include <stdint.h>
#include "io430.h"

void initADC( )
{
    //Избери функцията на извод P9.2 като вход на АЦП (A10)
    P9SEL0 ???= ???;
    P9SEL1 ???= ???;

    ADC12CTL0 &= ~ADC12ENC; //Забрани преобразуването, ENC = 0
    ADC12CTL0 |= ADC12ON; //Включи АЦП
    ADC12CTL0 |= ADC12SHT1_15; //Избери Sample-and-hold време = 512 такта на АЦП-то за
        //каналы 8 - 23

    ADC12CTL1 |= ADC12SSEL_2; //Избери MCLK за тактов сигнал
    ADC12CTL1 |= ADC12SHP; //Използвай sampling таймера
    ADC12CTL1 |= ADC12CONSEQ_0; //Избери режим "един канал - едно измерване"
    ADC12CTL1 |= ADC12PDIV_0; //Раздели тактовия сигнал на 1
    ADC12CTL1 |= ADC12DIV_1; //Раздели тактовия сигнал на 2 (от втория делител)
    ADC12CTL1 &= ~ADC12ISSH; //Не инвертирай sample-and-hold сигнала
    ADC12CTL1 |= ADC12SHS_0; //Използвай ADC12SC бита като сигнал за начало на sample-
        //and-hold

    ADC12CTL2 &= ~ADC12DF; //Формат на резултата - прав код
    ADC12CTL2 &= ~ADC12RES_3; //Занули битовете за разредността
    ADC12CTL2 |= ???; //Избери 12-битово преобразуване, Userguide стр. 880
//-----MCTL конфигурация-----
    ADC12MCTL10 &= ~ADC12DIF; //Избери single-ended преобразуване
    ADC12MCTL10 |= ADC12VRSEL_0; //Използвай Vdd за горен праг, Vss - за долен.
    ADC12MCTL10 |= ADC12INCH_10; //Избери A10 за входен канал
    ADC12CTL3 |= ADC12CSTARTADD_10; //Избери регистър ADC12MEM10 да е приемник на
        //резултата от преобразуването
    ADC12MCTL10 |= ADC12EOS; //Канал A10 е крайният канал за преобразуване
//-----

    ADC12CTL0 |= ADC12ENC; //Установи ENC в 1
}
```



```

void main( void )
{
//Тук ще се съхранява резултата от измерването
volatile uint16_t ADCResult;

WDTCTL = WDTPW + WDT HOLD;
PM5CTL0 &= ~LOCKLPM5;

//Инициализирай АЦП-то
initADC();

while(1){
ADC12CTL0 |= ???; // Започни измерване, Userguide →стр. 877

while(ADC12CTL1 & ???){} // Изчакай, докато приключи измерването, Userguide →стр. 878
ADCResult = ADC12MEM10; //Запиши резултата в променливата ADCResult, сложете тук точка
//на прекъсване
}
}

```

Задача 7.2.3

```

#include <stdint.h>
#include "io430.h"
#include "uartstdio.h"

```

```

void initADC( )
{
//Избери функцията на извод Р9.2 като вход на АЦП (А10)
P9SEL0 ???= ???;
P9SEL1 ???= ???;

//Избери функцията на извод Р9.3 като вход на АЦП (А11)
P9SEL0 ???= ???;
P9SEL1 ???= ???;

ADC12CTL0 &= ~ADC12ENC; //Забрани преобразуването, ENC = 0
ADC12CTL0 |= ADC12ON; //Включи АЦП
ADC12CTL0 |= ADC12SHT1_15; //Избери Sample-and-hold време = 512 такта на АЦП-то за
//каналы 8 - 23
ADC12CTL0 |= ADC12MSC; //Автоматично ресетиране на SH таймера след всяко
//преобразуване
ADC12CTL1 |= ADC12SSEL_2; //Избери MCLK за тактов сигнал
ADC12CTL1 |= ADC12SHP; //Използвай sampling таймера
ADC12CTL1 |= ADC12CONSEQ_1; //Избери режим "много канали, по едно измерване на всеки"
ADC12CTL1 |= ADC12PDIV_0; //Раздели тактовия сигнал на 1
ADC12CTL1 |= ADC12DIV_1; //Раздели тактовия сигнал на 2 (от втория делител)
ADC12CTL1 &= ~ADC12ISSH; //Не инвертирай sample-and-hold сигнала
ADC12CTL1 |= ADC12SHS_0; //Използвай ADC12SC бита като сигнал за начало на sample-
//and-hold
ADC12CTL2 &= ~ADC12DF; //Формат на резултата - прав код
ADC12CTL2 &= ~ADC12RES_3; //Занули битове за разредността
ADC12CTL2 |= ???; //Избери 12-битово преобразуване, Userguide стр. 880
//-----MCTL конфигурация-----

ADC12CTL3 |= ADC12CSTARTADD_10; //Избери регистър ADC12MEM10 да е приемник на
//резултата от първото преобразуване

//Конфигурирай измерването на потенциометъра
ADC12MCTL10 &= ~ADC12DIF; //Избери single-ended преобразуване

```

```

ADC12MCTL10 |= ADC12VRSEL_0; //Използвай Vdd за горен праг, Vss - за долен.
ADC12MCTL10 |= ADC12INCH_10; //Избери A10 за входен канал
ADC12MCTL10 &= ~ADC12EOS; //Канал A10 не е крайният канал за преобразуване

//Конфигурирай измерването на NTC резистора (термистора)
while(REFCTL0 & REFGENBUSY); //Провери дали REFGENBUSY е 1, ако ДА - изчакай
REFCTL0 |= ( ??? | REFON); //Задай 2.5 V | Включи вътрешния еталон
ADC12MCTL11 &= ~ADC12DIF; //Избери single-ended преобразуване
ADC12MCTL11 |= ???; //Използвай Vref(буфериран) за горен праг, Vss - за долен.
ADC12MCTL11 |= ADC12INCH_11; //Избери A30 за входен канал
ADC12MCTL11 &= ~ADC12EOS; //Канал A30 не е крайният канал за преобразуване

//Конфигурирай измерването на вътрешния резисторен делител (A31)
ADC12CTL3 |= ADC12BATMAP; //Избери вътрешния източник за канал A31
ADC12MCTL31 &= ~ADC12DIF; //Избери single-ended преобразуване
ADC12MCTL31 |= ADC12VRSEL_0; //Използвай Vdd за горен праг, Vss - за долен.
ADC12MCTL31 |= ADC12INCH_31; //Избери A31 за входен канал
ADC12MCTL31 |= ADC12EOS; //Канал A31 е крайният канал за преобразуване
//-----

ADC12CTL0 |= ADC12ENC; //Установи ENC в 1
}

void main( void )
{
    unsigned long adc_samples[3]; //Тук ще се съхраняват резултатите от измерванията

    WDTCTL = WDTPW | WDTHOLD;
    PM5CTL0 &= ~LOCKLPM5;

    initADC(); //Инициализирай АЦП-то
    UARTStdioInit(); //Инициализирай UART модула и UARTprintf функцията

    while(1){
        ADC12CTL0 |= ???; // Започни измерване, Userguide стр. 877
        while(ADC12CTL1 & ???){ }

        adc_samples[0] = ADC12MEM10;
        adc_samples[1] = ADC12MEM11;
        adc_samples[2] = ADC12MEM31;

        UARTprintf("Pot: %04d Temp: %04d Vdiv: %04d \r", adc_samples[0], adc_samples[1],
adc_samples[2]);

        __delay_cycles(1600000);
    }
}

```

Лабораторно упражнение №8

“Безжичен LAN (WLAN) модул CC3100 и MSP430FR6989. Създаване на Web сървър.”

8.1. Въведение. Към демо платката MSP-EXP430FR6989 може да се свърже комуникационна демо платка SimpleLink Wi-Fi CC3100BOOST (произведена от фирмата Texas Instruments), позволяваща реализиране на безжична LAN мрежа (или още Wireless LAN, WLAN, но да не се бърка с WAN) по стандарта IEEE 802.11 b/g. Такъв вид мрежа се поддържа от безжичните домашни рутери, което позволява да се създаде вградена система на базата на MSP430FR6989 с връзка към Интернет (**фиг. 8.1**). Ядрото на SimpleLink платката е интегралната схема CC3100 на фирмата Texas Instruments. Към него трябва да се свърже единствено външна антена и няколко кондензатора. Този вид чипове често се наричат мрежови процесори и служат за добавяне на Интернет свързаност към съществуващи вградени системи със собствен микроконтролер. От друга страна чипът CC3200 включва освен мрежови процесор, също и микроконтролер с ARM Cortex-M4 микропроцесор и периферия, т.е. може да се разглежда като сложен микроконтролер. Връзката MSP430FR6989 ← → CC3100 се осъществява чрез SPI интерфейс. SimpleLink използва SMD антена (отбелязана е като E1), която реализира безжичния интерфейс CC3100 ← → Wifi рутер. Максималната скорост на Интернет връзката е 16 Mbps (на носеща честота 2.4 GHz), което е много над възможностите на MSP430FR6989 (макс. тактова честота 16 MHz) и реално обменът на данни става на по-ниска скорост.

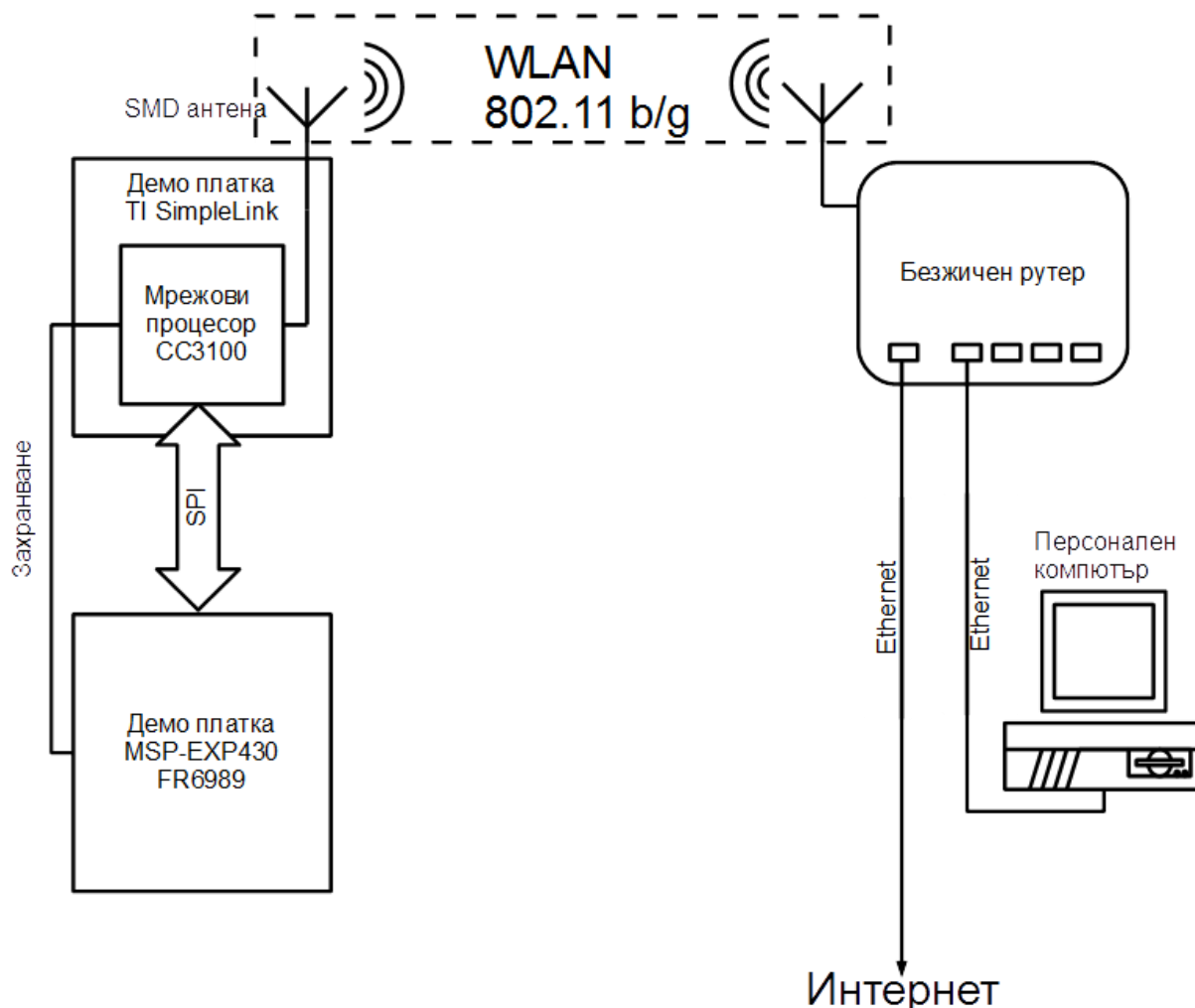
8.2. Основни понятия за Интернет. Някои от по-важните термини за Интернет комуникацията са:

- **LAN** (Local Area Network) - мрежа, обхващаща различен брой компютри, разположени в област не по-голяма от 10 km. Броят на компютрите варира много и зависи от мрежовата структура и използваните кабели. LAN може да се нарече мрежа от три компютъра в три съседни блока, както и 1000 компютъра в един квартал.

- **WLAN** (Wireless Local Area Network) - класическа LAN мрежа, но използваща ефира за преносна среда. Комуникацията се осъществява в определен радиоканал.

- **WAN** (Wide Area Network) – международни мрежи, обхващащи голяма географска област. Най-известният пример е Интернет. Друг пример са корпоративните мрежи (Интранет) на фирми, имащи клонове по цял свят и които са си изградили WAN за спомагане дейността на фирмата. Такива мрежи осигуряват обмен от около 1 ÷ 6Mbit/s, което е значително по-малко от скоростите, използвани в LAN.

- **Gateway** (на български се използва термина „шлюз“) - активни мрежови устройства, наречени още рутери (router), които представляват компютри или специализирани вградени системи, конфигурирани да управляват маршрута на данните. За да се осъществи връзка от една LAN в друга LAN трябва да се премине през gateway.



Фиг. 8.1

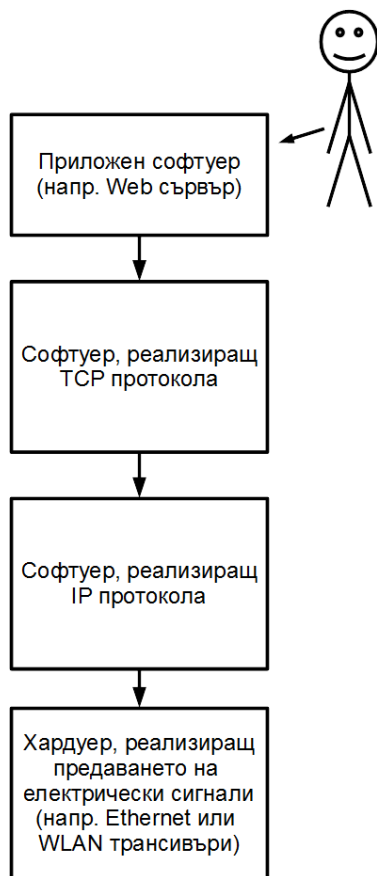
- **IP адрес** – уникално 32-битово цяло число, присвоено на дадено устройство в Интернет. Такъв адрес е необходим при предаването на информация между две и повече устройства. Изобразява се в десетичен вид, като всеки байт е разделен с точки (например 192.168.1.10, 10.1.1.23, 81.150.230.33 и други). Тези адреси използват версия на IP протокола номер 4 (IPv4).

С 32-битово число може да се адресират до $2^{32} = 4\,294\,967\,296$ устройства. Поради широката употреба на световната мрежа Интернет този брой се оказва недостатъчен, затова съществуват и 128-битови адреси (IPv6).

- **IP протокол** (Internet Protocol) – стандартизиран механизъм за предаване на информация в световната мрежа Интернет. Той дефинира комуникация, която се осъществява чрез разделяне на данните на пакети в подателя и събирането им в получателя. Всеки пакет освен полезните данни съдържа и IP адреса на подателя/получателя, заедно с различни служебни данни. При този протокол доставката на информация не се гарантира, т.е. някои от пакетите могат да бъдат изгубени.

- **DHCP протокол** (Dynamic Host Configuration Protocol) – аналогичен на IP протокола, но разликата е, че при първоначалното включване на устройството параметрите на IP протокола (IP адрес, Mask, Gateway и др.) се получават автоматично от DHCP сървър. Това означава, че потребителят ще получи автоматично, без негова намеса, всички необходими настройки, за да се свърже към Интернет. Характерното за този протокол е, че при всяко стартиране на компютъра си, потребителят получава IP адрес, който може да не съвпада с IP адреса от предишна сесия (предишното включване) на компютъра. Нещо повече – този IP адрес може да се промени по време на настоящата сесия.

- **TCP протокол** (Transmission Control Protocol) – протоколът осигурява пренасянето на непрекъсната поредица от данни, като за разлика от повечето Интернет протоколи, работи с изграждане на връзка и гарантиране на доставката на данните. Софтуерът, реализиращ този протокол е на най-високото ниво на абстракция в TCP/IP комуникацията (**фиг. 8.2**). Гарантиране предаването на данните означава, че ако някой пакет се изгуби от IP протокола, TCP слоя ще детектира това и ще го изпрати наново.



Фиг. 8.2

8.3. Връзка клиент-сървър. Връзката между две устройства в Интернет се основава на принципа клиент-сървър, при която:

- **клиент** се нарича компютърна програма, изискваща създаването на връзка и използване на някакъв компютърен ресурс;

- **сървърно приложение** е компютърна програма, която предоставя даден ресурс на една или повече клиентски програми. Сървърното приложение „слуша“ за идващи заявки от клиенти, обслужва ги и продължава да „слуша“ за следващи клиенти.

Типичен пример за комуникация клиент-сървър е качването на файл в Google Drive. Тук Интернет браузърът (Mozilla, Opera, Google Chrome) се нарича клиент, а Google Drive – сървърно приложение. Предоставеният ресурс е дисково пространство.

За да може клиент да се свърже към сървър е необходимо да се знаят поне два параметъра на сървъра:

- IP адрес;
- Порт, на който сървърното приложение очаква заявки от клиентите.

Под порт се разбира софтуерен механизъм за връзка, а не физически порт (куплунг). Софтуерните портове са необходими, за да се реализира достъп до сървъра от много потребители към много сървърни програми, а физически конекторът за връзка може да е само един. Благодарение на

портовете един и същ сървър може да се използва например за веб-страница и дисково пространство за документи едновременно. По подразбиране веб-страниците се обслужват на порт 80, т.е. когато напишем в браузъра `www.mysite.com`, всъщност ще се преведе като `www.mysite.com:80`. С двуточие се указва номера на порта. Друг пример е клиентската програма, осигуряваща достъп до файловата система на сървър – SSH. Връзката на SSH се осъществява винаги на порт 22. Валидни числа за портове са $0 \div 65535$.

От гледна точка на програмиста, връзката клиент-сървър се осъществява посредством т.нар. сокети (socket), които наподобяват файлове. Сокетът е крайна точка за комуникация (endpoint) в една компютърна мрежа. За да започне комуникацията, потребителската програма трябва да отвори сокет, след което да извърши всички операции по комуникацията с помощта на указател към този сокет (точно както се използват файлови указатели) и накрая да затвори сокета. Този вид достъп наподобява файловия достъп (отваряне, четене/запис, затваряне).

В настоящото упражнение се използва библиотека на Texas Instruments, която предлага множество API функции за работа със сокети. Най-често използваните от тях са показани в **Таблица 8.1**.

Име на функцията	Кратко описание
<code>sl_Socket()</code>	Създава крайна точка за комуникация, сокет, и връща указател към нея.
<code>sl_Bind()</code>	Присвоява на сокета статичния IP адрес на мрежовата карта. Ако тя използва DHCP, като адрес на функцията трябва да се подаде нула. Присвоява на сокета порт, на който ще се осъществяват връзките.
<code>sl_Listen()</code>	Използва се от сървърни програми. Отваря се присвоеният порт на присвоения IP адрес. От този момент нататък системата очаква („слуша“) за пристигащи връзки от клиенти.
<code>sl_Accept()</code>	Приема връзка с клиент. Тази функция връща указател към сокета на клиента (с негова помощ може да разберем например IP адреса на клиента). Блокираща функция.
<code>sl_Connect()</code>	Използва се от клиентски програми. На тази функция се предава IP адреса на сървъра и неговия порт, към който ще се свързваме.

sl_Recv()	Връща приетите байтове от сървъра/клиента. Блокираща функция.
sl_Send()	Изпраща определен брой байтове към сървъра/клиента.

Таблица 8.1

8.4. Развойна среда Code Composer Studio. Фирмата Texas Instruments разработва своя развойна среда за микроконтролерите си, въпреки че много от тях могат да се програмират със софтуер с отворен код (от компилатори до програми за зареждане във флаш паметта). При инсталацията на средата потребителят може да избере комерсиалните компилатори на TI (със същото име) и/или отворени такива (GCC). След това при създаването на нов проект може да се избере един от двата.

CCS е базирана на отворената развойна среда Eclipse. Визуално двете среди си приличат много. Вляво на екрана е разположено дървото на проекта, централно - текстов редактор, долу – терминал на компилатора. При дебъгване на програма допълнително се отварят прозорци за дисасембли на програмата и организацията на стека. Преминаването от редактиращ в дебъгващ изглед (perspective) става с два бутона горе, вдясно на екрана.

8.5. Задачи за изпълнение.

8.5.1. Да се разучи създаването на TCP/IP връзка. За целта се стартират два прозореца на програмата Sock, намираща се в MSHT_LAB/tools. В първия прозорец се избира:

- Socket type → Server
- В полето Server Name → 127.0.0.1
- В полето Server Port → 80 (пример, може и друго число)
- Натиска се бутон Listen.

Във втория прозорец се избира:

- Socket type → Client
- В полето Server Name → 127.0.0.1
- В полето Server Port → 80 (пример, може и друго число)
- Натиска се бутон Connect
- В полето Message се въвежда произволно текстово съобщение
- Натиска се бутон Send.

Ако всичко е минало наред, в прозореца на сървъра трябва да се покаже същото съобщение.

Пояснение: с този експеримент тествахме TCP/IP комуникацията по „виртуалната LAN карта“ на персоналния компютър. Linux и Windows

операционните системи поддържат такъв виртуален интерфейс, чийто виртуален IP адрес е 127.0.0.1 и се нарича localhost.

Допълнителен експеримент: научете IP адреса на вашия компютър, като въведете в терминал (за Windows 7 → Startmenu → Search → cmd) командата ipconfig. Попитайте ваши колеги за тяхното IP и опитайте да изпратите текстово съобщение до техния компютър.

8.5.2. Да се тества работата на MSP430FR6989 и CC3000 с помощта на демо програмата CC3000_FRAM_Sensor_Application. За целта се стартира средата Code Composer Studio на Texas Instruments. От главното меню се избира File → Switch workspace → Other → Browse → указва се пътя до работното поле (workspace) CC3000_workspace → OK → OK. Вляво на средата, в дървото с директории, се намира директорията на проекта SensorApplication. Щраква се два пъти върху нея, за да се покажат C файловете. От тях се щраква два пъти върху main.c, където се намира main() функцията на проекта. Избира се Project → Build Project. Компилацията на проекта трябва да премине без грешки. След това се зарежда фърмуерът в микроконтролера чрез Run → Debug. В CCS ще се отвори Debug сесия, което е нормално. Изпълнението на програмата ще спре в началото на main().

Следва да се отвори терминална програма за RS232 интерфейс (Hyper Terminal, TeraTerm, Real Term, Conect или др.) с 9600 бода, 8-N-1 и COMx, където x е номера на виртуалния COM порт на компютъра, към който е свързана MSP430FR6989 платката.

Когато се направи това, обратно в CCS се натиска бутон Resume или клавиш F8 от клавиатурата. Сървърът използва безжична LAN връзка, затова преди да стартираме клиента трябва да свържем MSP430FR6989 към безжичен рутер. Ако връзката е успешно осъществена, червеният светодиод LED1 ще свети постоянно.

!Важно → Wi-Fi мрежата трябва да използва WPA2 (криптиран) протокол.

В рамките на няколко секунди трябва да излязат данните на връзката в терминала, **фиг. 8.3**.

```
*****
Starting Saturn server application ...
WLAN connection established!
----INFO----
Joined SSID: MSHT_PPMK
Board IP: 192.168.0.101
Gateway: 192.168.0.1
DNS: 192.168.0.1
Waiting for a client ...
```

Фиг. 8.3

Фърмуерът на MSP430FR6989 реализира сървър, към който ще се свързва клиентът (компютърната програма) CC3000_FRAM_Sensor_Application.

Клиента CC3000_FRAM_Sensor_Application се намира в MSHT_LAB/tools/CC3000_FRAM_Sensor_Application. За да се стартира трябва да се влезе в директорията със същото име и да се щракне два пъти върху batch скрипта:

```
run_gui.bat
```

което ще покаже GUI програма, разработена от Texas Instruments за тест на Wi-Fi връзката.

В първия прозорец, който ще се покаже (с име Configure Connection Type) се натиска бутона Manual, който ще отвори нов прозорец (с име Network Interface Selection), в който трябва да изберем LAN картата на компютъра, която го свързва към рутера. Така ще реализираме сценария, показан на **фиг. 8.1**. Натиска се ОК.

Ако всичко е минало без грешки, ще се отвори прозорец със слънчева система и планетата Сатурн (**фиг. 8.4**). Позицията на пръстените на Сатурн всъщност се контролира от потенциометъра на демо платката. Температурата се измерва от NTC, разположен също на демо платката. Цветът на Сатурн се изменя в червено или синьо, в зависимост от околната температура. Светодиод LED2 (мигащ) показва изпращането на съобщение към клиента (всяко съобщение съдържа X, Y, Z координати, стойността Vdd на демо платката и околната температура).

8.5.3. Да се реализира изпращане на низ от символи от демо платката с MSP430FR6989 към програмата Sock. За целта да се отвори файлът main.c от проекта sendstr в работното поле, отворено по-рано (CC3100_workspace). Да се разучи C кода в този файл. Под коментара:

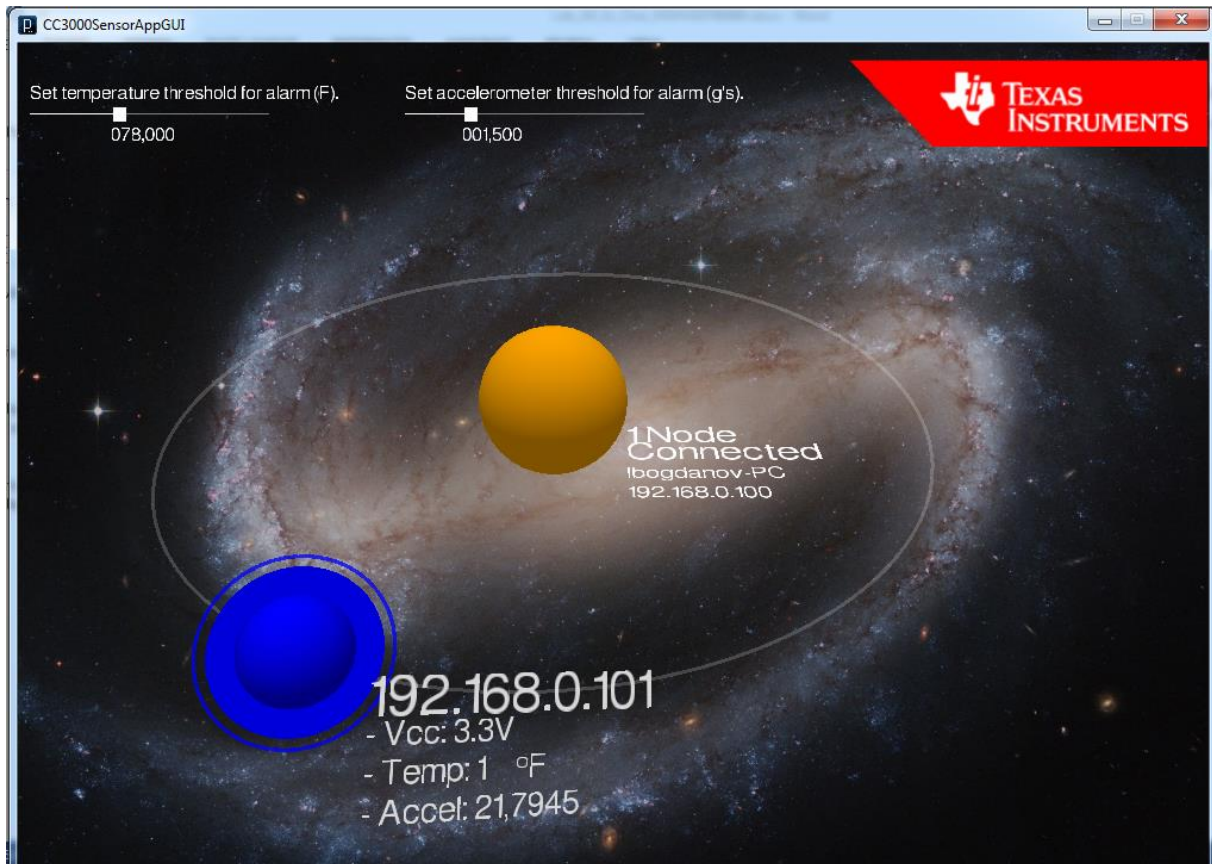
```
//Send string in TCP/IP terminal
```

да се извика функцията send() с необходимите параметри. Описание на функциите от библиотеката за CC3100 може да бъде намерено в MSHT_LAB/Documents/simplelink_doxygen_api/programmers_guide.html → след което се избира Modules → Socket_api → функция sl_Send().

След това се стартира програмата Sock в режим server и се въвежда IP-то на компютъра, на който върви то. За порт се избира числото, което е записано в макросът #define TCP_PORT от sendstr проекта (виж main.c). Натиска се бутона Listen.

Да се компилира и зареди фърмуерът sendstr. Ако свързването с рутера е минало успешно в терминала на COM порта трябва да се видят съобщенията от **фиг. 8.5**. Малко след това в сървърното приложение трябва

да се види низът Hello, World! Изпратете низ към MSP430FR6989 и вие от полето Message: на програмата Sock.



Фиг. 8.4

```
*****
Starting sendstr client application ...
WLAN connection established!
----INFO----
Joined SSID: MSHT_PPMK
Board IP: 192.168.0.101
Gateway: 192.168.0.1
DNS: 192.168.0.1
Establishing connection with TCP server... █
```

Фиг. 8.5

8.5.4. Да се реализира уеб сървър, съдържащ елементарна уеб страница, която изобразява текущите стойности на АЦП от измерването на потенциометъра, терморезистора и делителя на напрежение. Проектът се казва webserv. Да се разучи файлът html_sensors.h. Забележете как в код на С се вгражда HTML – чрез низ от символи (в случая char *index_sens_html). За да е по-прегледен кодът, низът е разположен на няколко реда и в няколко променливи в хедърния файл. Към HTML кода са добавени символи за нов

ред \n и преди кавичките е сложен символът \ (което е изискване на C-низовете). Тоест HTML кодът трябва да претърпи известна трансформация, преди да бъде зареден в контролер, на който няма файлова система. В практиката се използват програми, които генерират хедърен (.h) C файл от HTML (.html) файл.

Да се извика функция от sensors.c, която попълва масива data_sens_html_dynamic с валиден HTML код, съдържащ стойностите на АЦП като низ. Това трябва да стане под коментара:

```
//Invoke the sensor reading function here
```

След това да се изпрати информацията, поместена в масива data_sens_html_dynamic, подобно на предходното упражнение – с функцията sl_Send(), под коментара:

```
//Send the sensor data in HTML format here
```

Да се компилира и зареди програмата. Да се отвори уеб-браузър и в полето за име на сайт да се напише IP-то на демо платката. Натиснете Enter. Трябва да се появи уеб-страницата, показана на **фиг. 8.6**.



Фиг. 8.6

Пояснение: HTML кодът на програмата е следния:

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
  <TITLE>
    MSP430FR6989
  </TITLE>
</HEAD>
<BODY>
  <meta http-equiv="refresh" content="2" />
  <H1>SENSORS</H1>
  <P>POT: 0000 TEMP: 0000 VDD: 0000</P>
</BODY>
</HTML>
```

Задача 8.5.3.

```
#include "simplelink.h"
#include "sl_common.h"
#include "simple_link_func.h"
#include "uartstdio.h"
#include "leds.h"

#define SSID_NAME      "MSHT_PPMK"           // Access point name to connect to.
#define SEC_TYPE      SL_SEC_TYPE_WPA_WPA2 // Security type of the Access piont
#define PASSKEY       "arm-cortex"         // Password in case of secure AP
#define PASSKEY_LEN   pal_Strlen(PASSKEY)  // Password length in case of secure AP
#define TCP_PORT      3333

int main(int argc, char** argv)
{
  SISockAddrIn_t sock_addr;
  SISecParams_t secParams = {0};
  _u32 server_ip_addr[4] = {192, 168, 0, 100};
  _u32 server_ip_converted = 0;
  _u8 *tcp_send_msg = "Hello, World!";
  _u8 tcp_recv_msg[128];
  _i16 tcp_recv_msg_size;
  _i16 client_socket = 0;

  stopWDT();
  initClk();
  CLI_Configure();
  init_leds();
  UARTprintf("\n\r*****\n\r");
  UARTprintf("Starting sendstr client application ... \n\r");

  configureSimpleLinkToDefaultState();

  sl_Start(0, 0, 0);

  secParams.Key = (_i8 *)PASSKEY;
  secParams.KeyLen = pal_Strlen(PASSKEY);
```

```

secParams.Type = SEC_TYPE;
sl_WlanConnect((_i8 *)SSID_NAME, pal_Strlen(SSID_NAME), 0, &secParams, 0);

while(!IS_CONNECTED(g_Status) || (!IS_IP_ACQUIRED(g_Status)))
{ _SINonOsMainLoopTask(); }

set_led(LED_RED, 1);

UARTprintf("Establishing connection with TCP server... ");

server_ip_converted = (server_ip_addr[0]<<24) | (server_ip_addr[1]<<16) | (server_ip_addr[2]<<8) |
(server_ip_addr[3]);
sock_addr.sin_family = SL_AF_INET;
sock_addr.sin_port = sl_Htons((_u16)TCP_PORT);
sock_addr.sin_addr.s_addr = sl_Htonl(server_ip_converted);

client_socket = sl_Socket(SL_AF_INET,SL_SOCKET_STREAM, 0);

sl_Connect(client_socket, ( SISockAddr_t *)&sock_addr, sizeof(SISockAddrIn_t));

set_led(LED_GREEN, 1);

UARTprintf("done!\n\r");

//Send string in TCP/IP terminal
??

while(IS_CONNECTED(g_Status)){
tcp_rcv_msg_size = sl_Recv(client_socket, tcp_rcv_msg, 128, 0);

if(tcp_rcv_msg_size <= 0){
break;
}

tcp_rcv_msg[tcp_rcv_msg_size] = '\0';

UARTprintf("MSG from server: %s\n\r", tcp_rcv_msg);

blink_led(LED_GREEN);
}

sl_Close(client_socket);

sl_WlanDisconnect();

sl_Stop(SL_STOP_TIMEOUT);

set_led(LED_RED, 0);
set_led(LED_GREEN, 0);

while(1){
__bis_SR_register(LPM0_bits + GIE); //Enter sleep mode
}
}

```

Задача 8.5.4.

```
#include "html_sensors.h"
#include "simplelink.h"
#include "sl_common.h"
#include "simple_link_func.h"
#include "uartstdio.h"
#include "leds.h"
#include "html.h"
#include "sensors.h"

#define SSID_NAME      "MSHT_PPMK"          // Access point name to connect to.
#define SEC_TYPE      SL_SEC_TYPE_WPA_WPA2 // Security type of the Access point
#define PASSKEY       "arm-cortex"         // Password in case of secure AP
#define PASSKEY_LEN   pal_Strlen(PASSKEY)  // Password length in case of secure AP
#define MAX_MSG_SIZE  20
#define TCP_PORT      80

int main(int argc, char** argv)
{
    uint8_t data_sens_html_dynamic[40];
    SISecParams_t secParams = {0};
    SISockAddrIn_t server_sock_addr;
    _u16 server_sock_addr_size = sizeof(SISockAddrIn_t);
    SISockAddrIn_t client_sock_addr;
    _u16 client_sock_addr_size = sizeof(SISockAddrIn_t);
    _i16 server_socket = 0;
    _i16 client_socket = -1;
    char recv_buff[255];
    unsigned long html_size;

    stopWDT();
    initClk();
    CLI_Configure();
    init_leds();
    init_sensors();
    UARTprintf("\n\r*****\n\r");
    UARTprintf("Starting web server application ...\n\r");

    configureSimpleLinkToDefaultState();

    sl_Start(0, 0, 0);

    sl_NetAppStop(SL_NET_APP_HTTP_SERVER_ID); //Stop internal SimpleLink web page on port 80

    secParams.Key = (_i8 *)PASSKEY;
    secParams.KeyLen = pal_Strlen(PASSKEY);
    secParams.Type = SEC_TYPE;
    sl_WlanConnect((_i8 *)SSID_NAME, pal_Strlen(SSID_NAME), 0, &secParams, 0);

    while(!IIS_CONNECTED(g_Status) || (!IIS_IP_ACQUIRED(g_Status)))
    { _SINonOsMainLoopTask(); }

    set_led(LED_RED, 1);

    server_sock_addr.sin_family = SL_AF_INET;
    server_sock_addr.sin_port = sl_Htons((_u16)TCP_PORT);
    server_sock_addr.sin_addr.s_addr = 0;

    server_socket = sl_Socket(SL_AF_INET,SL_SOCKET_STREAM, SL_IPPROTO_TCP);
```

```

sl_Bind(server_socket, (SISockAddr_t *)&server_sock_addr, server_sock_addr_size);

sl_Listen(server_socket, 1);

while(1){

    while(client_socket < 0){
        client_socket = sl_Accept(server_socket, ( struct  SIOckAddr_t  *)&client_sock_addr,
(SIOcklen_t*)&client_sock_addr_size);
        _SINonOsMainLoopTask();
    }

    //Wait for HTTP GET request
    sl_Recv(client_socket, recv_buff, 255, 0);
    UARTprintf("MSG from client: %s \n\r", recv_buff);

    //Static Hello, WORLD web page-----
    //unsigned char *html_msg = "<HTML><HEAD></HEAD><BODY>Hello,
WORLD!</BODY></HTML>";
    //unsigned char *html_msg = "HTTP/1.0 200 OK\r\nContent-Type:
text/html\r\n\r\n<HTML>\n<BODY>\n\t<H1>Hello, WORLD!</H1>\n</BODY>\n</HTML>\n";
    //sl_Send(client_socket, html_msg, strlen(html_msg), 0);
    //-----

    //Invoke the sensor reading function here
    ???

    html_size = strlen(index_sens_html);
    sl_Send(client_socket, index_sens_html, html_size, 0);

    //Send the sensor data in HTML format here
    ???

    html_size = strlen(close_sens_html);
    sl_Send(client_socket, close_sens_html, html_size, 0);

    sl_Close(client_socket);
    client_socket = -1;

    _SINonOsMainLoopTask();

    blink_led(LED_GREEN);
}
}

```


Приложение

П1. ASCII Таблица

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

128	Ç	144	É	160	á	176	☒	192	Ł	208	⌚	224	α	240	≡
129	ü	145	æ	161	í	177	☓	193	ł	209	⌛	225	β	241	±
130	é	146	Æ	162	ó	178	☑	194	ṽ	210	⌜	226	Γ	242	≥
131	â	147	ô	163	ú	179		195	ı	211	⌝	227	π	243	≤
132	ã	148	õ	164	ñ	180	†	196	–	212	⌞	228	Σ	244	∫
133	ä	149	ö	165	Ñ	181	‡	197	+	213	⌟	229	σ	245	∫
134	å	150	ù	166	ª	182	‡	198	†	214	⌠	230	μ	246	+
135	ç	151	û	167	º	183	¶	199	‡	215	‡	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	¶	200	⌚	216	‡	232	Φ	248	°
137	ë	153	ÿ	169	ƒ	185	¶	201	⌛	217	‡	233	Θ	249	·
138	è	154	ÿ	170	ƒ	186	¶	202	⌚	218	⌛	234	Ω	250	·
139	ì	155	◊	171	½	187	¶	203	⌛	219	■	235	δ	251	√
140	í	156	£	172	¼	188	¶	204	‡	220	■	236	∞	252	∞
141	î	157	¥	173	¡	189	¶	205	=	221	■	237	φ	253	²
142	ÿ	158	€	174	«	190	¶	206	‡	222	■	238	ε	254	■
143	ÿ	159	ƒ	175	»	191	¶	207	±	223	■	239	∩	255	

Source: www.LookupTables.com

П2. Приоритети на операторите в C

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (<i>type</i>) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change <i>type</i>) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

П3. Асемблер на MSP430

				Status Bits			
				V	N	Z	C
*	ADC(.B)	dst	dst + C → dst	x	x	x	x
	ADD(.B)	src,dst	src + dst → dst	x	x	x	x
	ADDC(.B)	src,dst	src + dst + C → dst	x	x	x	x
	AND(.B)	src,dst	src .and. dst → dst	0	x	x	x
	BIC(.B)	src,dst	.not.src .and. dst → dst	-	-	-	-
	BIS(.B)	src,dst	src .or. dst → dst	-	-	-	-
	BIT(.B)	src,dst	src .and. dst	0	x	x	x
*	BR	dst	Branch to	-	-	-	-
	CALL	dst	PC+2 → stack, dst → PC	-	-	-	-
*	CLR(.B)	dst	Clear destination	-	-	-	-
*	CLRC		Clear carry bit	-	-	-	0
*	CLRN		Clear negative bit	-	0	-	-
*	CLRZ		Clear zero bit	-	-	0	-
	CMP(.B)	src,dst	dst - src	x	x	x	x
*	DADC(.B)	dst	dst + C → dst (decimal)	x	x	x	x
	DADD(.B)	src,dst	src + dst + C → dst (decimal)	x	x	x	x
*	DEC(.B)	dst	dst - 1 → dst	x	x	x	x
*	DECD(.B)	dst	dst - 2 → dst	x	x	x	x
*	DINT		Disable interrupt	-	-	-	-
*	EINT		Enable interrupt	-	-	-	-
*	INC(.B)	dst	Increment destination, dst +1 → dst	x	x	x	x
*	INCD(.B)	dst	Double-Increment destination, dst+2→dst	x	x	x	x
*	INV(.B)	dst	Invert destination	x	x	x	x
	JC/JHS	Label	Jump to Label if Carry-bit is set	-	-	-	-
	JEQ/JZ	Label	Jump to Label if Zero-bit is set	-	-	-	-
	JGE	Label	Jump to Label if (N .XOR. V) = 0	-	-	-	-
	JL	Label	Jump to Label if (N .XOR. V) = 1	-	-	-	-
	JMP	Label	Jump to Label unconditionally	-	-	-	-
	JN	Label	Jump to Label if Negative-bit is set	-	-	-	-

Legend:	0	Status bit always cleared	1	Status bit always set
	x	Status bit cleared or set on results	-	Status bit not affected
	*	Emulated Instructions		

			Status Bits			
			V	N	Z	C
JNC/JLO	Label	Jump to Label if Carry-bit is reset	-	-	-	-
JNE/JNZ	Label	Jump to Label if Zero-bit is reset	-	-	-	-
MOV(.B)	src,dst	src → dst	-	-	-	-
* NOP		No operation	-	-	-	-
* POP(.B)	dst	Item from stack, SP+2 → SP	-	-	-	-
PUSH(.B)	src	SP - 2 → SP, src → @SP	-	-	-	-
RETI		Return from interrupt	x	x	x	x
		TOS → SR, SP + 2 → SP				
		TOS → PC, SP + 2 → SZP				
* RET		Return from subroutine	-	-	-	-
		TOS → PC, SP + 2 → SP				
* RLA(.B)	dst	Rotate left arithmetically	x	x	x	x
* RLC(.B)	dst	Rotate left through carry	x	x	x	x
RRA(.B)	dst	MSB → MSBLSB → C	0	x	x	x
RRC(.B)	dst	C → MSBLSB → C	x	x	x	x
* SBC(.B)	dst	Subtract carry from destination	x	x	x	x
* SETC		Set carry bit	-	-	-	1
* SETN		Set negative bit	-	1	-	-
* SETZ		Set zero bit	-	-	1	-
SUB(.B)	src,dst	dst + .not.src + 1 → dst	x	x	x	x
SUBC(.B)	src,dst	dst + .not.src + C → dst	x	x	x	x
SWPB	dst	swap bytes	-	-	-	-
SXT	dst	Bit7 → Bit8 Bit15	0	x	x	x
* TST(.B)	dst	Test destination	x	x	x	x
XOR(.B)	src,dst	src .xor. dst → dst	x	x	x	x

Legend: 0 The Status Bit is cleared 1 The Status Bit is set
 x The Status Bit is affected - The Status Bit is not affected
 * Emulated Instructions

П4. Intrinsic функции на MSP430

MSP430 Ininsics

[Table 6-3](#) lists all of the intrinsic operators in the MSP430 C/C++ compiler. A function-like prototype is presented for each intrinsic that shows the expected type for each parameter. If the argument type does not match the parameter, type conversions are performed on the argument.

For more information on the resulting assembly language mnemonics, see the *MSP430x1xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, and the *MSP430x4xx Family User's Guide*.

Table 6-5. MSP430 Ininsics

Intrinsic	Generated Assembly
unsigned short __bcd_add_short(unsigned short op1, unsigned short op2);	MOV op1, dst CLRC DADD op2, dst
unsigned long __bcd_add_long(unsigned long op1, unsigned long op2);	MOV op1_low, dst_low MOV op1_hi, dst_hi CLRC DADD op2_low, dst_low DADD op2_hi, dst_hi
unsigned short __bic_SR_register(unsigned short mask);	BIC mask, SR
unsigned short __bic_SR_register_on_exit(unsigned short mask);	BIC mask, saved_SR
unsigned short __bis_SR_register(unsigned short mask);	BIS mask, SR
unsigned short __bis_SR_register_on_exit(unsigned short mask);	BIS mask, saved_SR
unsigned long __data16_read_addr(unsigned short addr);	MOV.W addr, Rx MOVA 0(Rx), dst
void __data16_write_addr(unsigned short addr, unsigned long src);	MOV.W addr, Rx MOVA src, 0(Rx)
unsigned char __data20_read_char(unsigned long addr); ⁽¹⁾	MOVA addr, Rx MOVX.B 0(Rx), dst
unsigned long __data20_read_long(unsigned long addr); ⁽¹⁾	MOVA addr, Rx MOVX.W 0(Rx), dst_lo MOVX.W 2(Rx), dst_hi
unsigned short __data20_read_short(unsigned long addr); ⁽¹⁾	MOVA addr, Rx MOVX.W 0(Rx), dst
void __data20_write_char(unsigned long addr, unsigned char src); ⁽¹⁾	MOVA addr, Rx MOVX.B src, 0(Rx)
void __data20_write_long(unsigned long addr, unsigned long src); ⁽¹⁾	MOVA addr, Rx MOVX.W src_lo, 0(Rx) MOVX.W src_hi, 2(Rx)
void __data20_write_short(unsigned long addr, unsigned short src); ⁽¹⁾	MOVA addr, Rx MOVX.W src, 0(Rx)
void __delay_cycles(unsigned long);	See Section 6.8.2 .
void __disable_interrupt(void); OR __disable_interrupts(void);	DINT
void __enable_interrupt(void); OR __enable_interrupt(void); OR __enable_interrupts(void);	EINT
unsigned int __even_in_range(unsigned int, unsigned int);	See Section 5.8.13 .
unsigned short __get_interrupt_state(void);	MOV SR, dst
unsigned short __get_R4_register(void);	MOV.W R4, dst
unsigned short __get_R5_register(void);	MOV.W R5, dst
unsigned short __get_SP_register(void);	MOV SP, dst
unsigned short __get_SR_register(void);	MOV SR, dst
unsigned short __get_SR_register_on_exit(void);	MOV saved_SR, dst
void __low_power_mode_0(void);	BIS.W #0x18, SR
void __low_power_mode_1(void);	BIS.W #0x58, SR
void __low_power_mode_2(void);	BIS.W #0x98, SR
void __low_power_mode_3(void);	BIS.W #0xD8, SR
void __low_power_mode_4(void);	BIS.W #0xF8, SR
void __low_power_mode_off_on_exit(void);	BIC.W #0xF0, saved_SR
void __never_executed(void);	See Section 6.8.3 .
void __no_operation(void);	NOOP
void __op_code(unsigned short);	Encodes whatever instruction corresponds to the argument.
void __set_interrupt_state(unsigned short src);	MOV src, SR
void __set_R4_register(unsigned short src);	MOV.W src, R4
void __set_R5_register(unsigned short src);	MOV.W src, R5
void __set_SP_register(unsigned short src);	MOV src, SP
unsigned short __swap_bytes(unsigned short src);	MOV src, dst SWPB dst

П5. Разположение на периферията

