

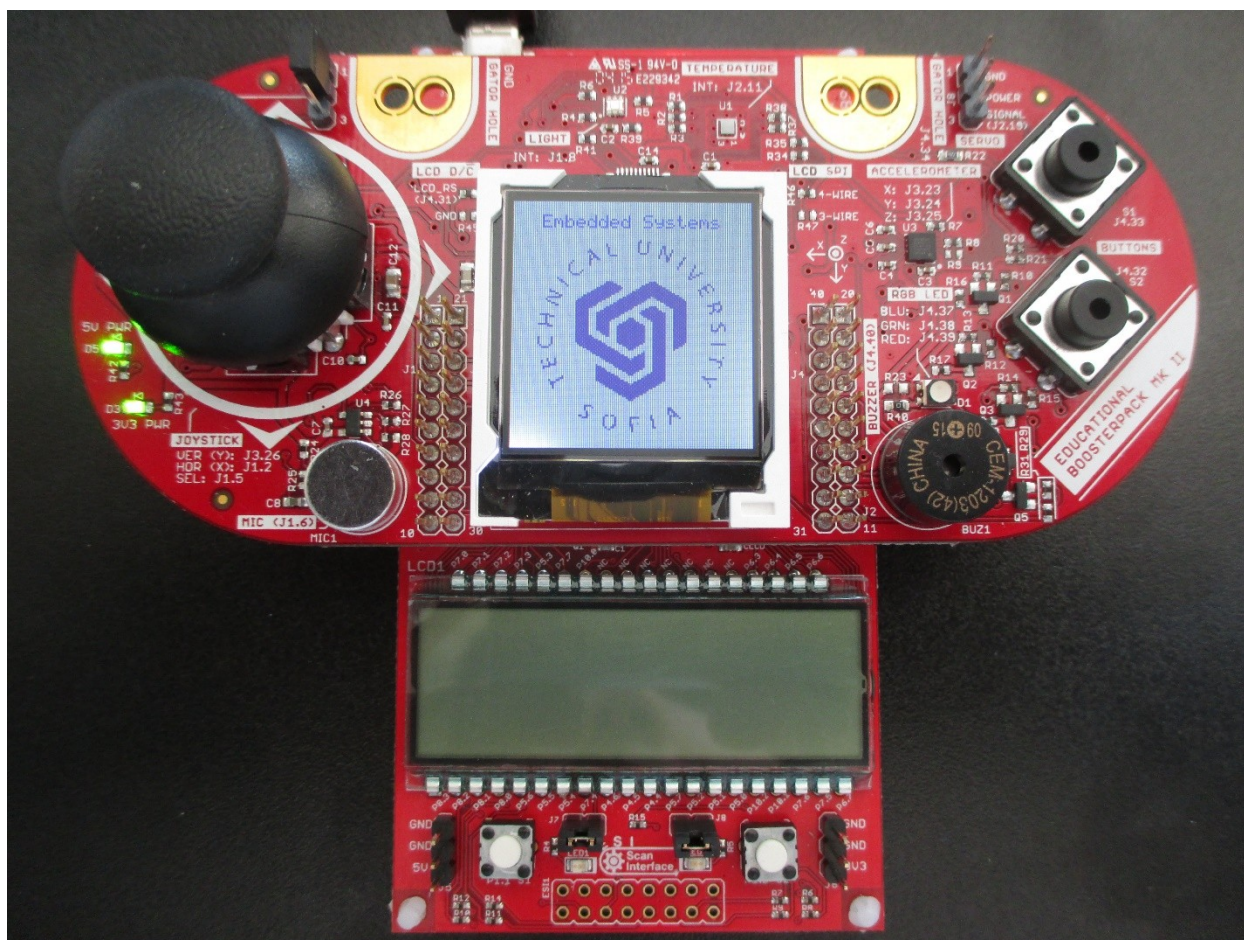
# РЪКОВОДСТВО

## за

### „ПРАКТИКУМ ПО ПРОГРАМИРАНЕ НА МИКРОКОНТРОЛЕРИ“

#### КЪМ

#### ФЕТТ, ТУ-София

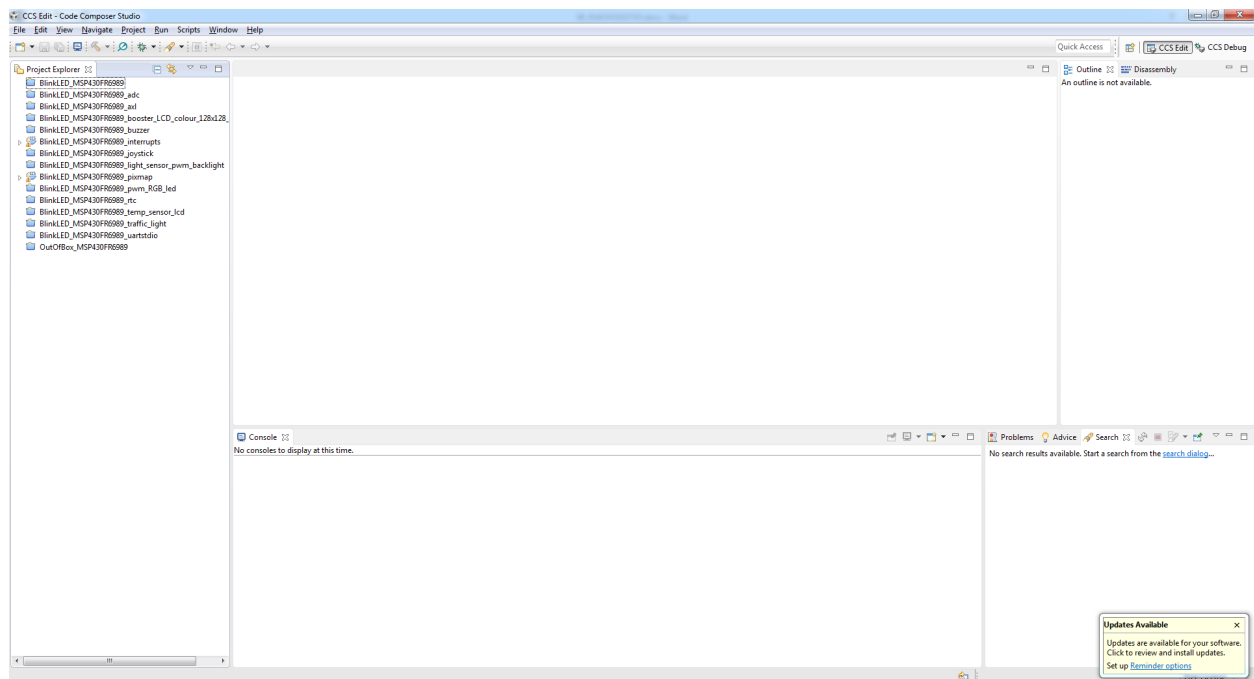


# I. Създаване на нов проект в средата Code Composer Studio

1. Стартирайте средата Code Composer Studio на фирмата Texas Instruments от иконката:



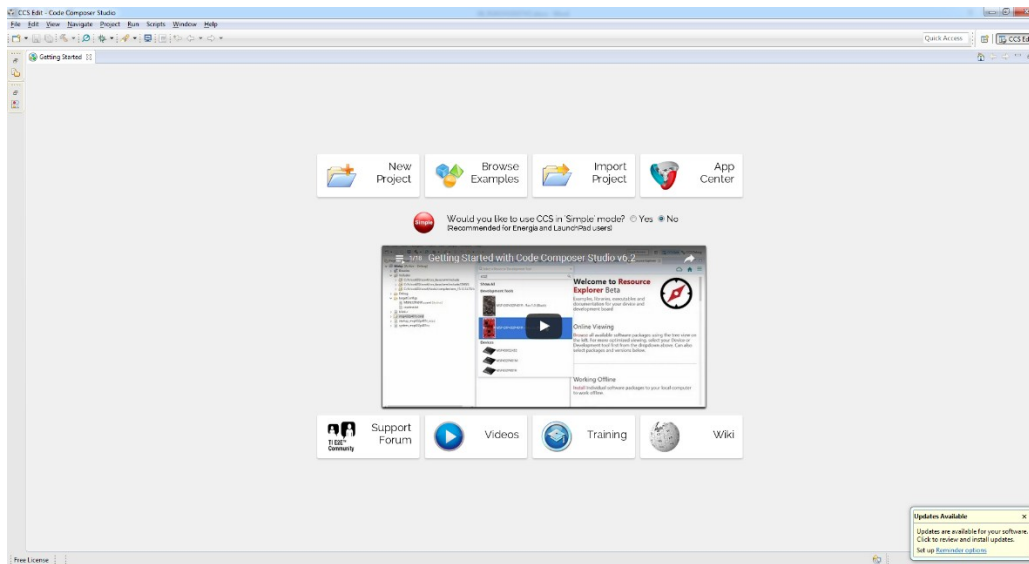
2. Ще се отвори основния прозорец на средата, който изглежда подобно на:



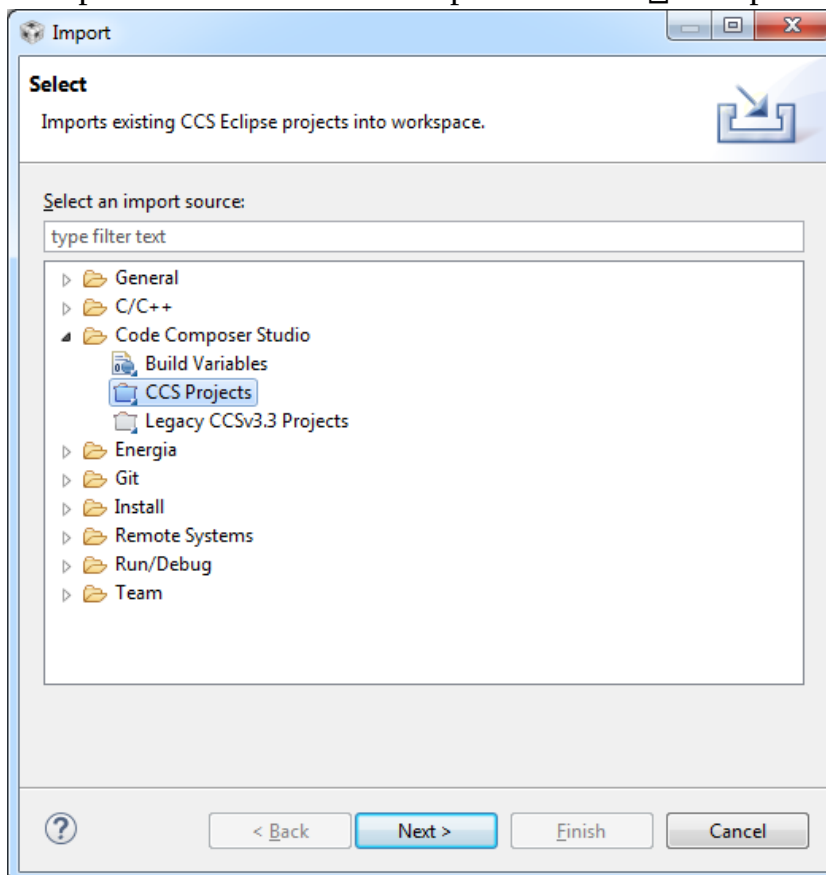
3. Създайте си нов Workspace (Workspace е главна директория, в която ще се съдържат директориите на проектите ви). За целта:

Изберете File ☐ Switch Workspace ☐ Other ☐ Направете директория за Workspace на Desktop-а и я кръстете с вашето име ☐ OK.

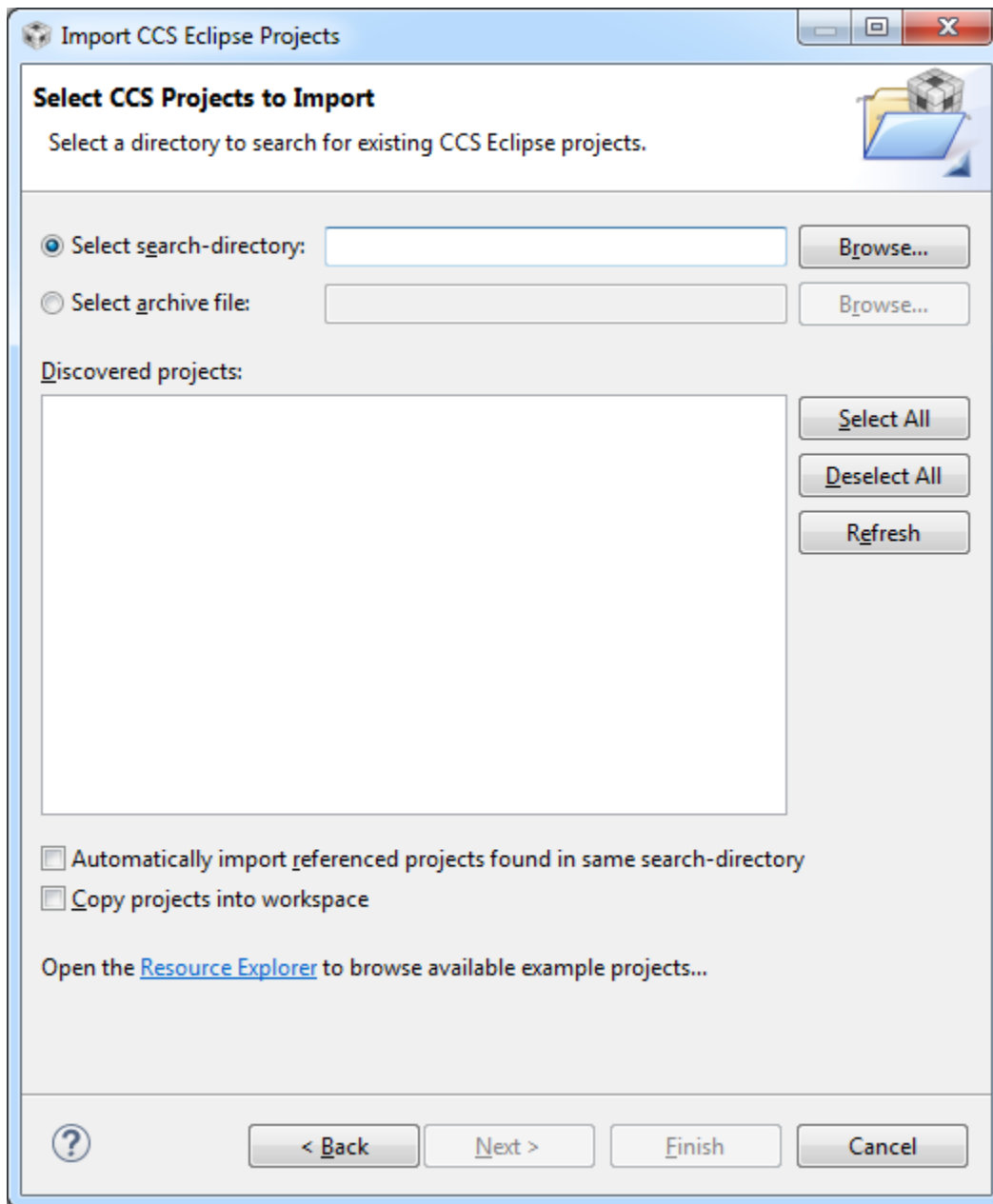
4. Ако създаването е било успешно ще се отвори средата със Getting started таб:



5. Затворете Getting started таб-а с хикса. Изберете File → Import → отворете папката Code Composer Studio → изберете CCS Projects → Next



6. Поставете радио-бутона на “Select search-directory:” и натиснете Browse:



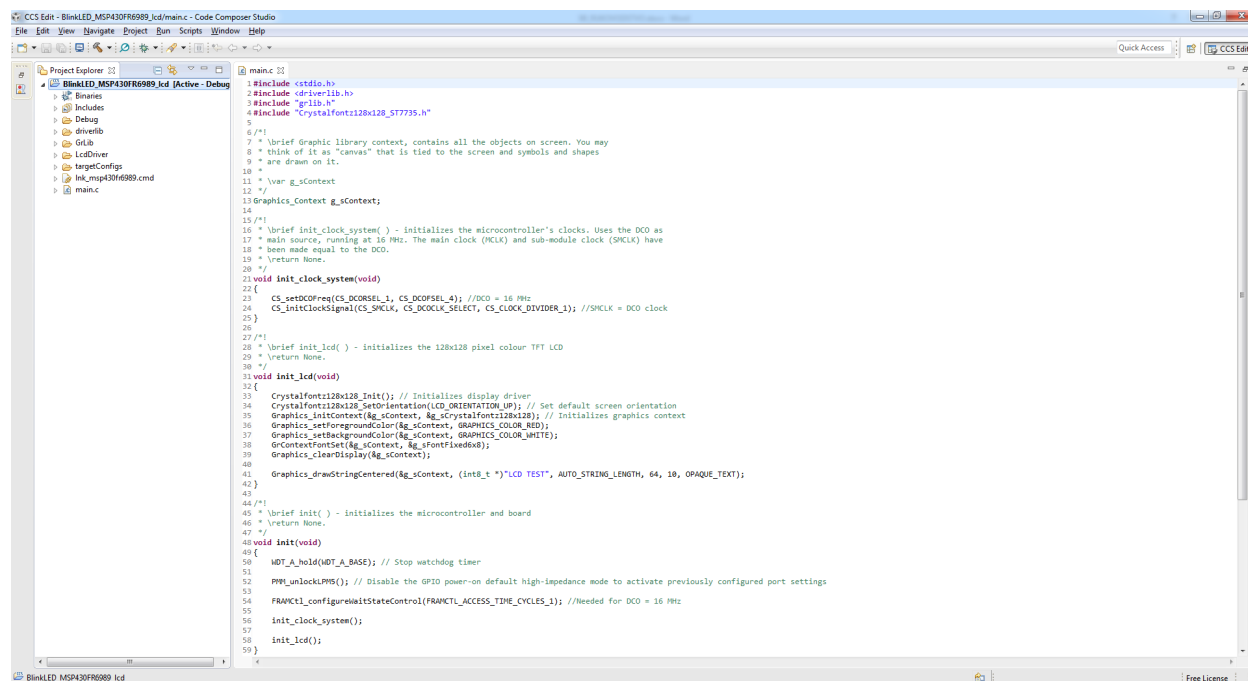
7. Посочете проекта:

MSHT\_PRAKT\examples\BlinkLED\_MSP430FR6989\_lcd

където MSHT\_PRAKT е директория, предоставена от ръководителя на практикума. Поставете отметка на “Copy projects into workspace” □ Finish.

Този проект е template проект, който инициализира микроконтролера MSP430FR6989 за работа с тактова честота 16 MHz, инициализира цветен LCD дисплей и изобразява съобщението LCD TEST.

## 8. Ако всичко е преминало успешно, ще видите прозореца:



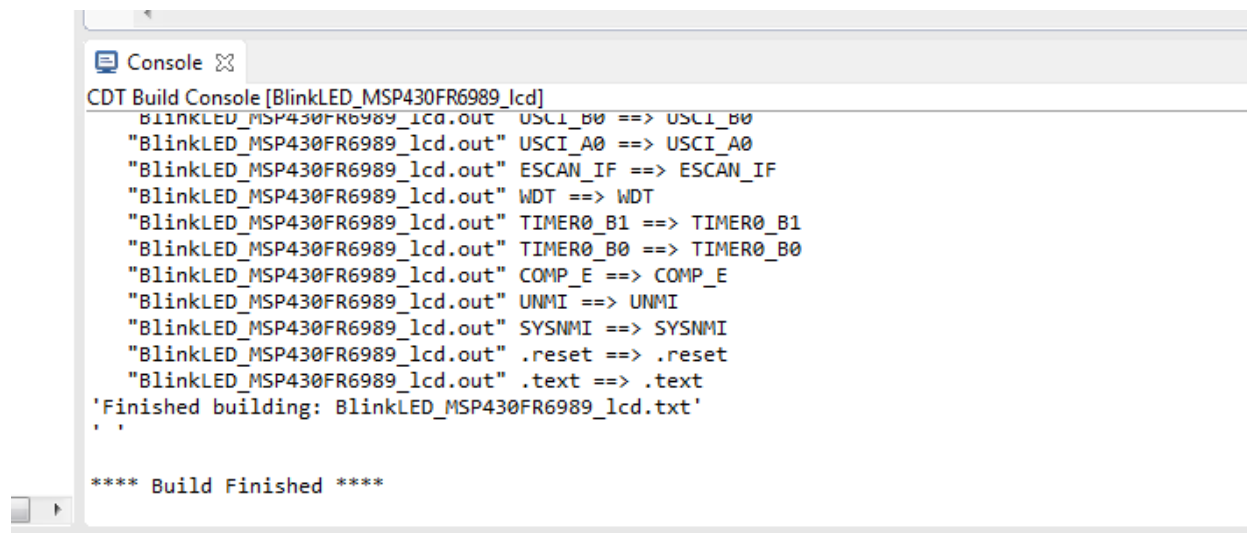
## 9. Натиснете бутона Build (компилиране, асемблиране и линкване на проекта):



По подразбиране средата използва комерсиален (платен) компилатор на Texas Instruments, който е наречен cl430. Той позволява да се тества демо програма с размер до 16 kB (валидно към 2016 г.), която не бива да се използва за производство. 16 kB са достатъчни за нуждите на практикума.

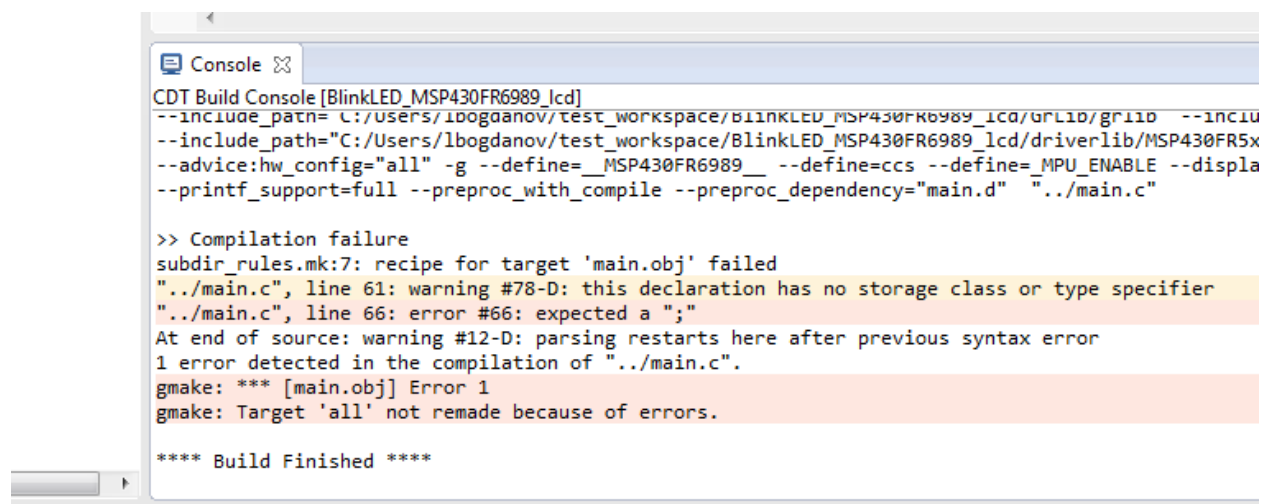
*Средата позволява и използването на безплатен крос-компилатор за MSP430, базиран на известния GCC. Той обаче генерира по-голям размер на програмите и синтаксисът на хардуерно-зависимите части в кода леко се различава от комерсиалния вариант.*

10. Ако Build-ването е преминало успешно, трябва да се изпише в новопоявилата се конзола Build Finished.



```
CDT Build Console [BlinkLED_MSP430FR6989_lcd]
BlinkLED_MSP430FR6989_lcd.out USC1_B0 ==> USC1_B0
"BlinkLED_MSP430FR6989_lcd.out" USC1_A0 ==> USC1_A0
"BlinkLED_MSP430FR6989_lcd.out" ESCAN_IF ==> ESCAN_IF
"BlinkLED_MSP430FR6989_lcd.out" WDT ==> WDT
"BlinkLED_MSP430FR6989_lcd.out" TIMER0_B1 ==> TIMER0_B1
"BlinkLED_MSP430FR6989_lcd.out" TIMER0_B0 ==> TIMER0_B0
"BlinkLED_MSP430FR6989_lcd.out" COMP_E ==> COMP_E
"BlinkLED_MSP430FR6989_lcd.out" UNMI ==> UNMI
"BlinkLED_MSP430FR6989_lcd.out" SYSNMI ==> SYSNMI
"BlinkLED_MSP430FR6989_lcd.out" .reset ==> .reset
"BlinkLED_MSP430FR6989_lcd.out" .text ==> .text
'Finished building: BlinkLED_MSP430FR6989_lcd.txt'
**** Build Finished ****
```

Ако има грешки в програмата, в конзолата се появяват червени редове с мястото на грешката и текст „gmake: Target 'all' not remade because of errors.“



```
CDT Build Console [BlinkLED_MSP430FR6989_lcd]
--include_path= C:/Users/lbogdanov/test_workspace/BlinkLED_MSP430FR6989_lcd/glib/glib --inciu
--include_path="C:/Users/lbogdanov/test_workspace/BlinkLED_MSP430FR6989_lcd/driverlib/MSP430FR5x
--advice:hw_config="all" -g --define= _MSP430FR6989_ --define=ccs --define= MPU_ENABLE --displa
--printf_support=full --preproc_with_compile --preproc_dependency="main.d"  "../main.c"

>> Compilation failure
subdir_rules.mk:7: recipe for target 'main.obj' failed
"../main.c", line 61: warning #78-D: this declaration has no storage class or type specifier
"../main.c", line 66: error #66: expected a ";"
At end of source: warning #12-D: parsing restarts here after previous syntax error
1 error detected in the compilation of "../main.c".
gmake: *** [main.obj] Error 1
gmake: Target 'all' not remade because of errors.

**** Build Finished ****
```

В случая с template-а не би трябвало да има грешки.

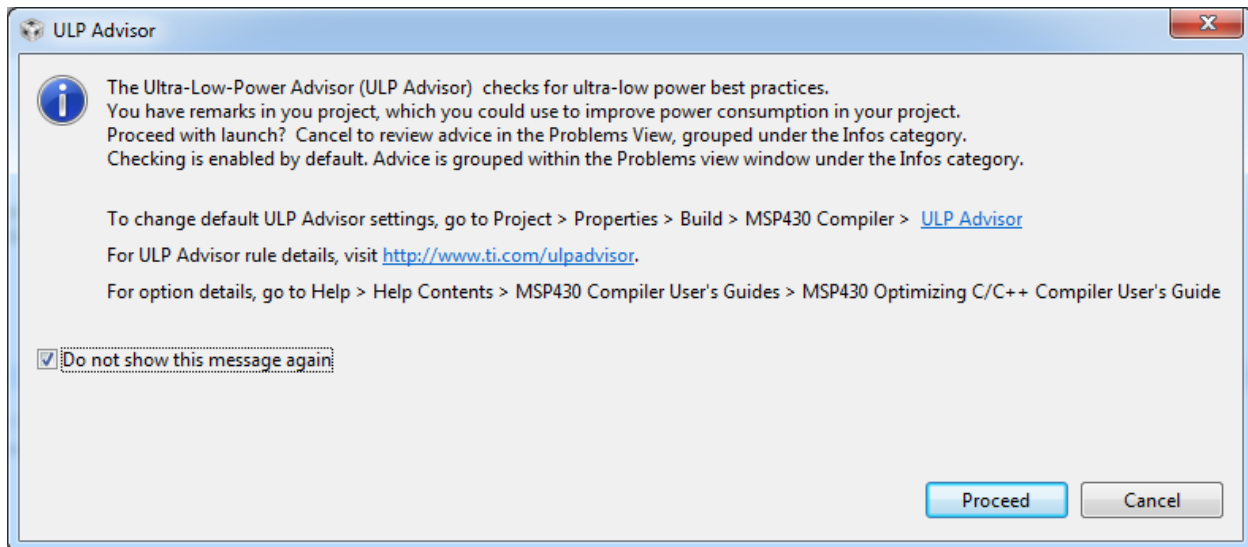
11. Включете макета към USB порт на вашия компютър. Натиснете бутон за зареждане и дебъгване на програмата в микроконтролера:



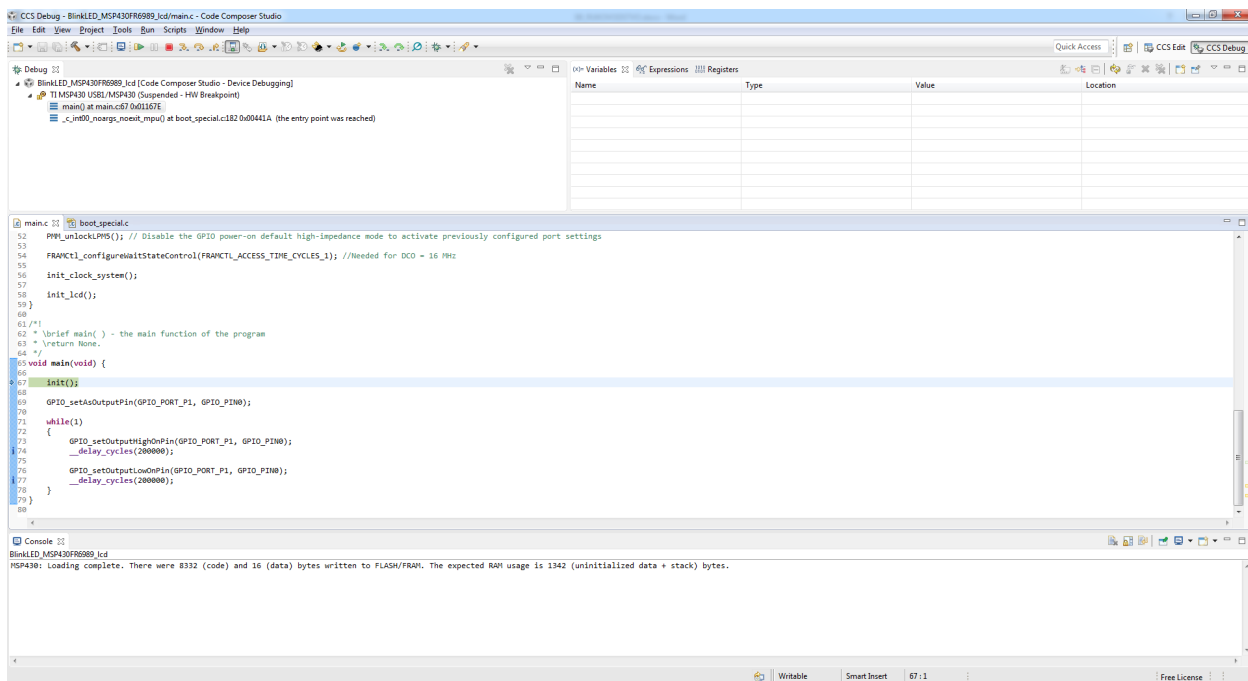
12. Първоначално ще излезе прозорец от компилатора със съвети за намаляване на консумацията на енергия на вашата програма. Към момента



този аспект от програмирането може да го пренебрегнем, затова поставете отметка на „Do not show this message again” и натиснете бутона Proceed.



13. Ще се отвори нов изглед (Debug perspective) на средата, пригоден за дебъгване на вашата програма:



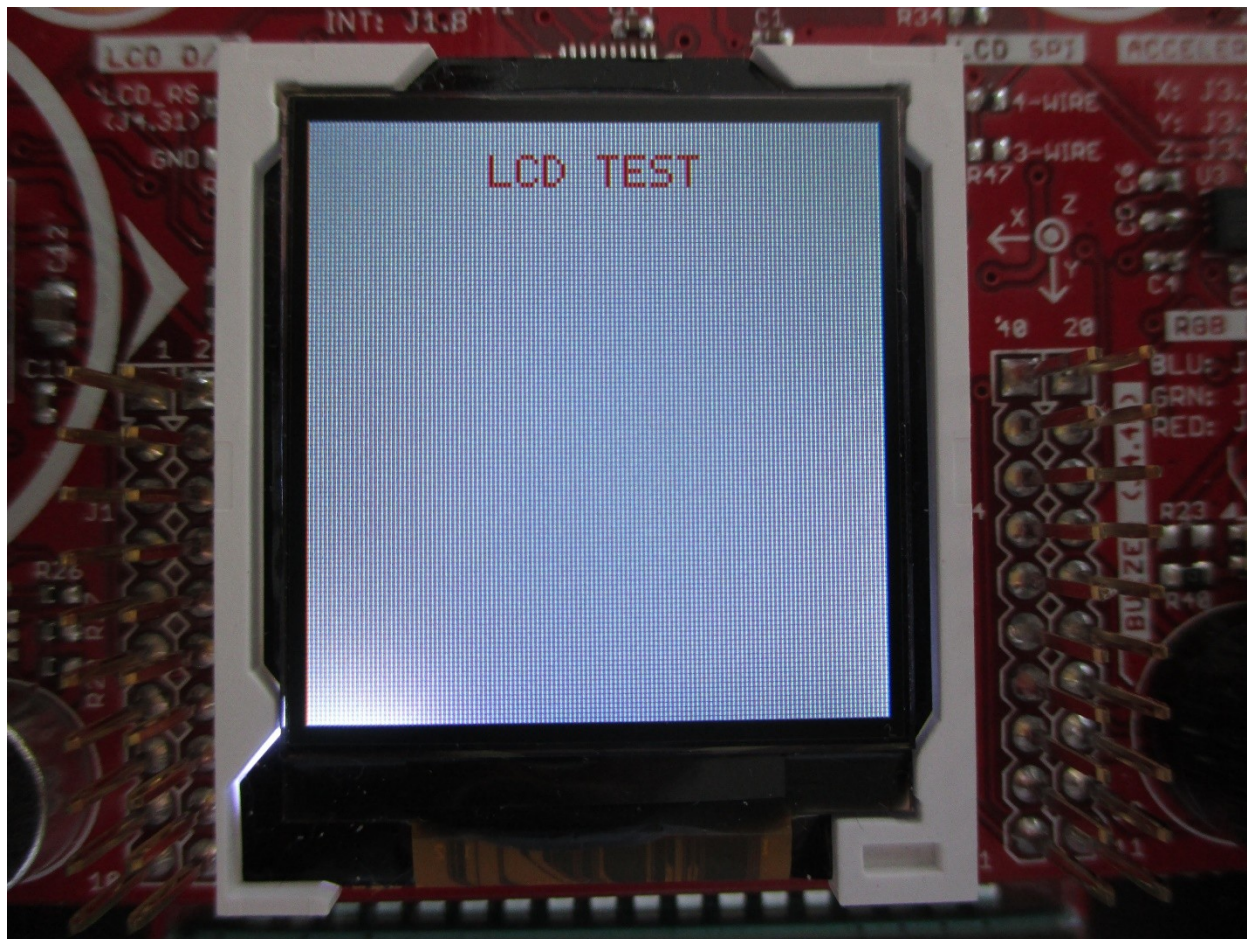
Смяната на изгледите става автоматично при стартиране и спиране на дебъг сесия, но ако това не стане винаги можете ръчно да го направите чрез бутоните “CCS Edit” и “CCS Debug” в горния десен ъгъл на средата:



14. След зареждане на програмата, хардуерният дебъгер на демо борд-а държи микроконтролера в състояние пауза на пърия ред от вашата програма (засветен със зелено). За да пуснете програмата да се изпълни докрай натиснете бутона Resume или бутон F8 от клавиатурата:



Скоро след това върху цветния LCD дисплей трябва да се изпише LCD TEST с червени букви.





## II. Отстраняване на грешки (дебъг) в средата Code Composer Studio

### 1. Използват се бутоните от Debug менюто:



**Бутон 1** – build на проекта, включващ компилиране, асемблира и линкване на програмата.

**Бутон 2** – пускане на изпълнение на програмата до край. Обикновено микроконтролерните програми никога не излизат от `main( )`, защото преди края ѝ е поставен безкраен цикъл, например `while(1) { }`.

**Бутон 3** – спиране на изпълнението на програмата, докдето е стигнала.

**Бутон 4** – термилиране на дебъг сесия. Натиска се, когато трябва да се правят корекции по програмата. След това трябва отново да се натисне Бутон 1 или Бутон 12.

**Бутон 5** – изпълнение на програмата ред по ред, влизайки в тялото на всяка функция, която се среща.

**Бутон 6** – изпълнение на програмата ред по ред, без да се влиза в тялото на функциите, които ще се срещнат. Функциите обаче ще бъдат изпълнени.

**Бутон 7** – излизане от тялото на функцията, в която се намираме.

**Бутон 8** – рестартиране на микроконтролера. Може да бъде софтуерно (чрез запис на стойност от дебъгера в специален регистър на микроконтролера) или хардуерно (дебъгерът установява свой извод в логическа нула, която е подадена на `reset` извода на програмирания микроконтролер). Регистрите на микроконтролера се установяват в default стойностите си.

**Бутон 9** – рестартиране хода на програмата от началото на `main( )`. Регистрите на микроконтролера запазват данните си.

**Бутон 10** – изпълнение на програмата асемблерна инструкция по асемблерна инструкция. Зад всеки ред на `C`, компилаторът е генерирал по един или повече редове на асемблер, затова натискането на този бутон понякога трябва да бъде няколко пъти преди да се продължи към следващия ред от `C` програмата. Използва се за много задълбочено дебъгване, в повечето случаи не се налага да се използва. Ако се срещне асемблерна подпрограма се влиза в нея.

**Бутон 11** – аналогичен на Бутон 10, но ако се срещне подпрограма не се влиза в нея, но все пак тя ще бъде изпълнена.

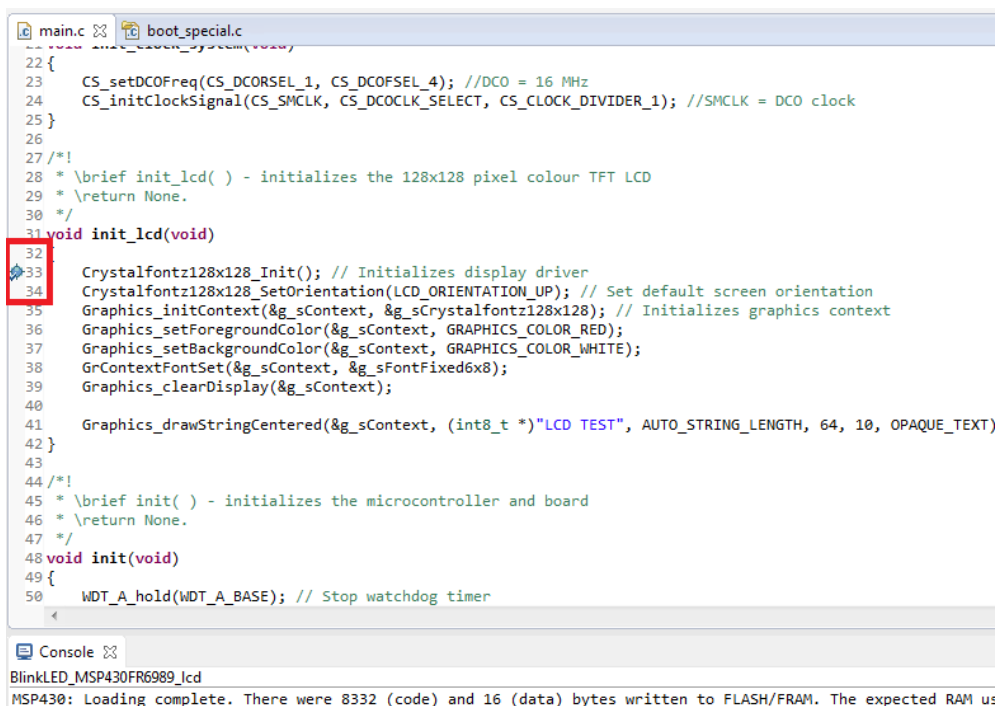
**Бутон 12** – компилиране, асемблиране, линкване и зареждане на програмата в микроконтролера. Този бутон стартира дебъг сесия.

**Внимание** – някои бутони са активни само след като се натисне Бутон 3.

Ако при изпълнението на програмата стъпка по стъпка срещнете цикъл с много голям брой итерации, не е нужно да минавате през всички тях. За целта сложете курсора в текстовия редактор на ред, след проблемния цикъл и изберете менюто Run □ Run to line (или натиснете Ctrl + R), което ще принуди микроконтролера да изпълни цикъла и да спре след него.

## 2. Поставят се точки на прекъсване в програмата.

Понякога има нужда дадена програма да бъде спряна точно на определен ред. Това се постига чрез точки на прекъсване. Точка на прекъсване се слага като в текстовия редактор, където се намира програмата, се натисне два пъти с мишката отстрани на програмата. Ако поставянето на прекъсването е било успешно, ще се изобрази синя точка:



The screenshot shows an IDE with two tabs: 'main.c' and 'boot\_special.c'. The 'main.c' tab is active, displaying a C program. A blue arrow cursor is positioned on line 34, which is highlighted with a red box. The code in 'main.c' includes initialization for a 128x128 pixel TFT LCD and a microcontroller. The console window at the bottom shows the output of the program, indicating that the loading is complete and the expected RAM usage is 16 bytes.

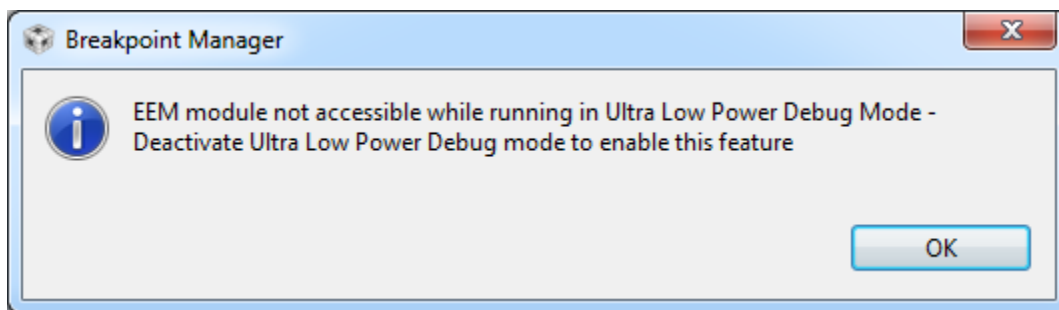
```
22 {
23     CS_setDCOFreq(CS_DCORSEL_1, CS_DCOFSEL_4); //DCO = 16 MHz
24     CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1); //SMCLK = DCO clock
25 }
26
27 /*!
28 * \brief init_lcd( ) - initializes the 128x128 pixel colour TFT LCD
29 * \return None.
30 */
31 void init_lcd(void)
32 {
33     Crystalfontz128x128_Init(); // Initializes display driver
34     Crystalfontz128x128_SetOrientation(LCD_ORIENTATION_UP); // Set default screen orientation
35     Graphics_initContext(&g_sContext, &g_sCrystalfontz128x128); // Initializes graphics context
36     Graphics_setForegroundColor(&g_sContext, GRAPHICS_COLOR_RED);
37     Graphics_setBackgroundColor(&g_sContext, GRAPHICS_COLOR_WHITE);
38     GrContextFontSet(&g_sContext, &g_sFontFixed6x8);
39     Graphics_clearDisplay(&g_sContext);
40
41     Graphics_drawStringCentered(&g_sContext, (int8_t *)"LCD TEST", AUTO_STRING_LENGTH, 64, 10, OPAQUE_TEXT)
42 }
43
44 /*!
45 * \brief init( ) - initializes the microcontroller and board
46 * \return None.
47 */
48 void init(void)
49 {
50     WDT_A_hold(WDT_A_BASE); // Stop watchdog timer
```

Console □

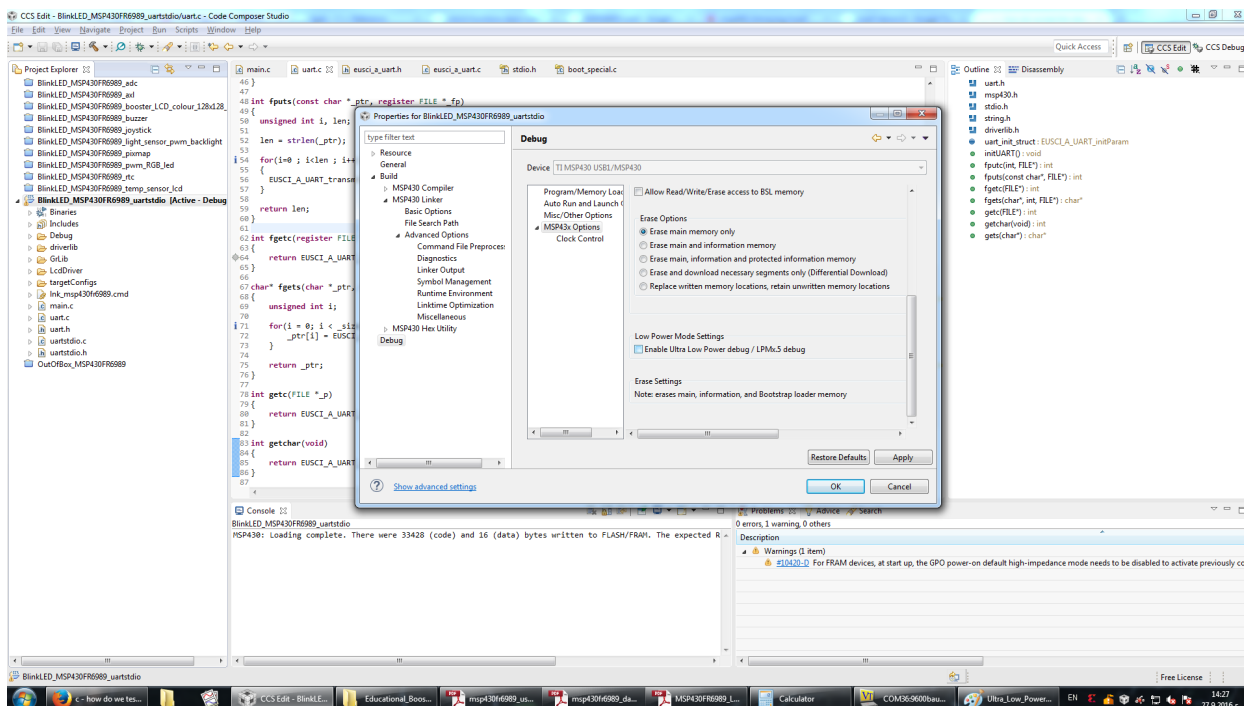
BlinkLED\_MSP430FR6989\_lcd

MSP430: Loading complete. There were 8332 (code) and 16 (data) bytes written to FLASH/FRAM. The expected RAM us

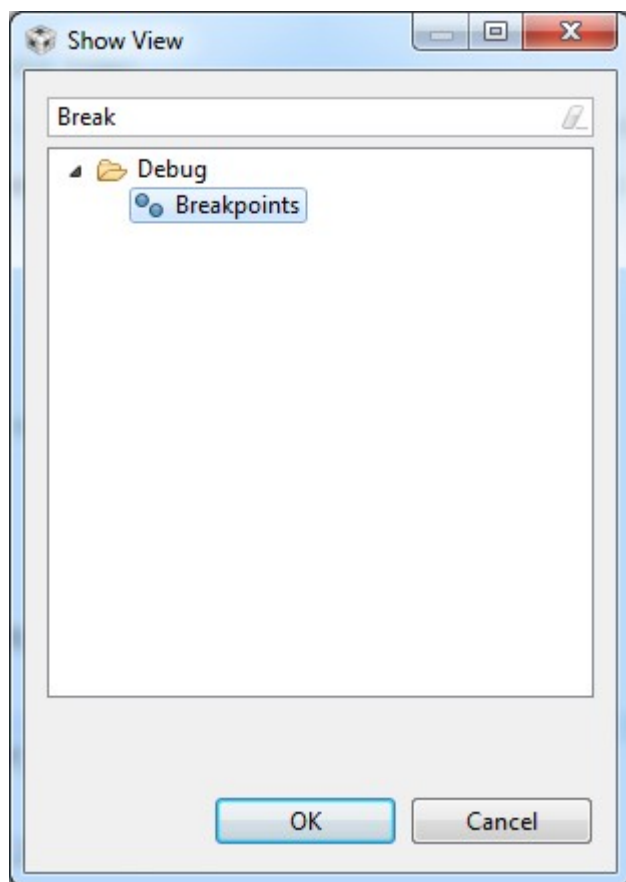
Микроконтролерите от фамилията MSP430 притежават режими на много дълбока хибернация и тогава дебъгърът не може да ги събуди, за да продължи дебъга. Това става видно от съобщението за грешка, която се появява при опит за поставяне на точка на прекъсване:



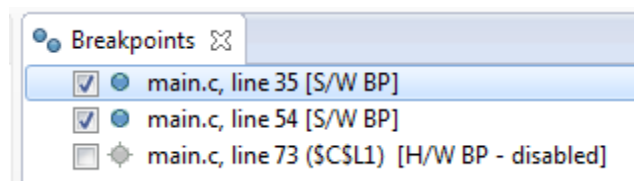
За да стане възможен дебъгът, трябва да се натисне десен бутон върху директорията на проекта (вляво на средата, в таба с дървото на проекта) ▢ Properties ▢ Debug ▢ MSP43x Options ▢ махнете отметката „Enable Ultra Low Power debug / LPMx.5 debug“ в раздела „Low power mode settings“.



При поставяне на много точки на прекъсване е възможно да забравите къде в кода се намират те. Затова CCS ви позволява да изведете прозорец със списък с всички точки на прекъсване. За целта изберете Window → Show view → Other → на реда за търсене въведете „Breakpoints” → изберете категорията Breakpoints → OK



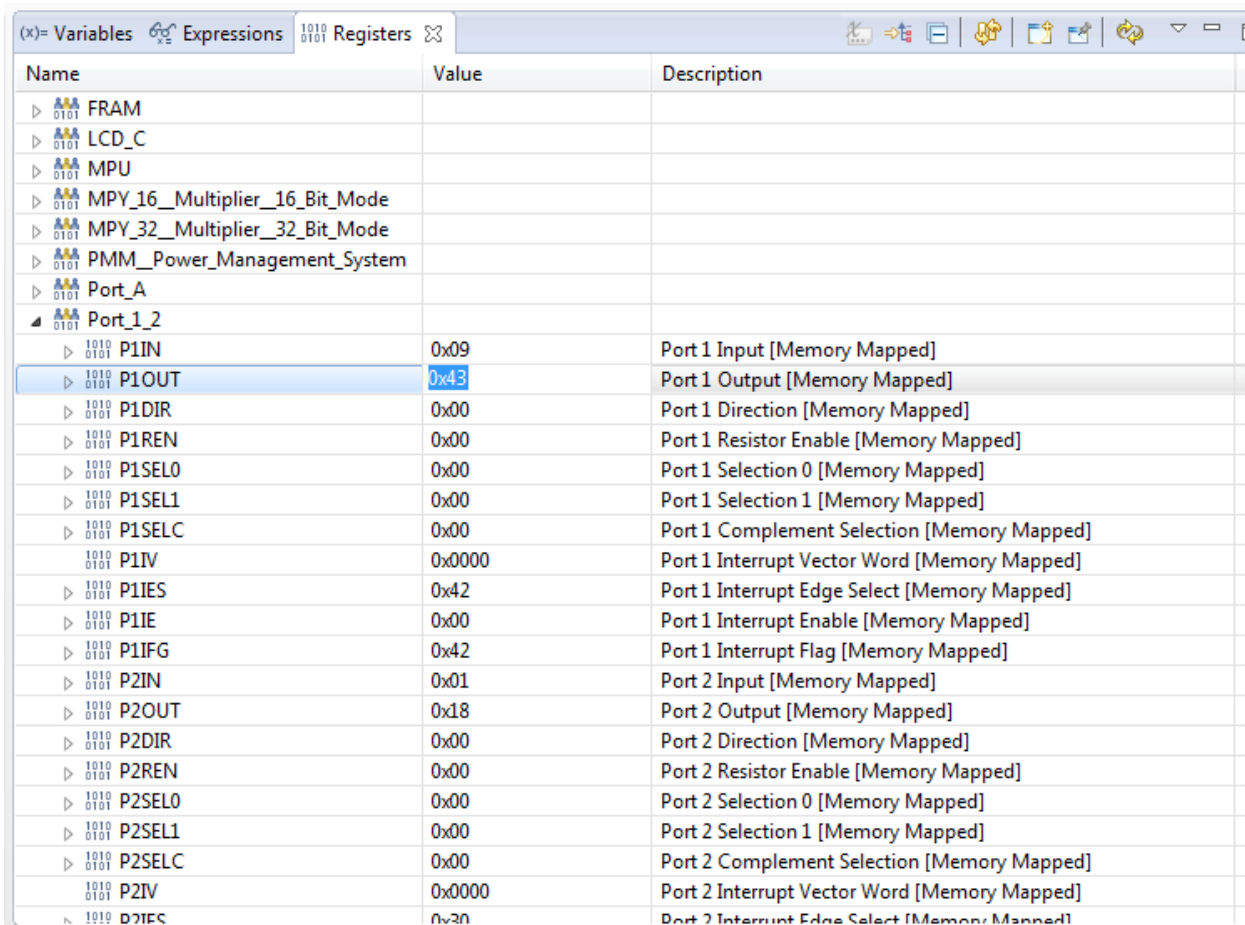
Ще се появи таб, показващ всички точки на прекъсване, както и редът на който се намират те. Ако махнете отметката от дадена точка на прекъсване това ще я забрани и тя няма да е активна.



### 3. Наблюдаване на регистрите.

Всеки един регистър на микроконтролера е достъпен за четене/запис от дебъгера. Това позволява разработчика на софтуер да наблюдава регистрите със специално предназначение и да види дали записаните стойности в тях са очакваните. За целта изберете таба горе вдясно на CCS с име Registers. Там ще видите списък с всички периферни модули, вградени в микроконтролера.

Ако искате например да наблюдавате регистрите на GPIO модул, изберете го от списъка и под него ще се отвори подписък с регистрите от дадения модул. Вие можете както да ги наблюдавате, така и да ги променяте след зареждането на програмата в микроконтролера. Единствено обаче трябва да сте натиснали първо бутон за пауза (Бутон 3).

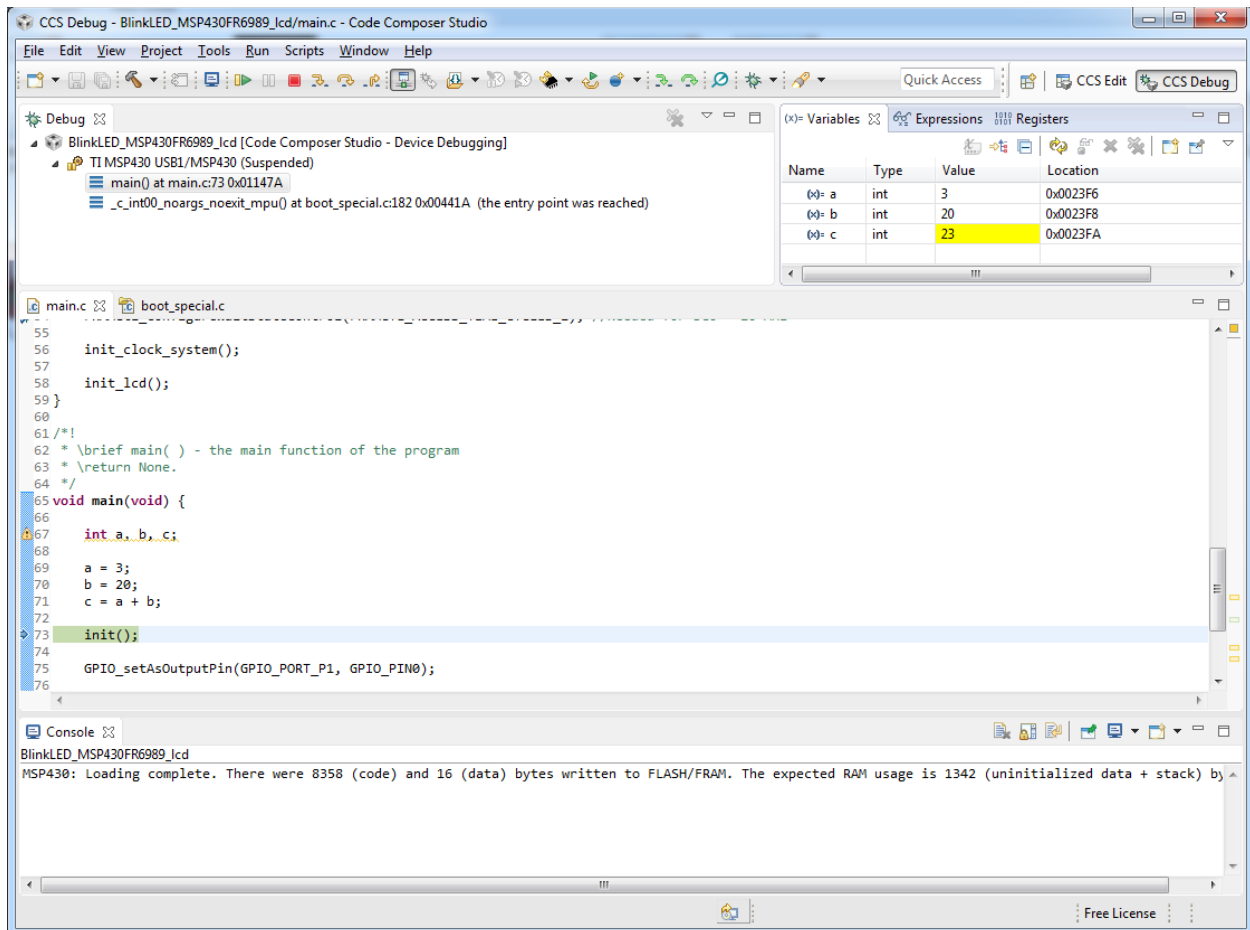


Name	Value	Description
▶ 1010 0101 FRAM		
▶ 1010 0101 LCD_C		
▶ 1010 0101 MPU		
▶ 1010 0101 MPY_16_Multiplier_16_Bit_Mode		
▶ 1010 0101 MPY_32_Multiplier_32_Bit_Mode		
▶ 1010 0101 PMM_Power_Management_System		
▶ 1010 0101 Port_A		
▶ 1010 0101 Port_1_2		
▶ 1010 0101 P1IN	0x09	Port 1 Input [Memory Mapped]
▶ 1010 0101 P1OUT	0x43	Port 1 Output [Memory Mapped]
▶ 1010 0101 P1DIR	0x00	Port 1 Direction [Memory Mapped]
▶ 1010 0101 P1REN	0x00	Port 1 Resistor Enable [Memory Mapped]
▶ 1010 0101 P1SEL0	0x00	Port 1 Selection 0 [Memory Mapped]
▶ 1010 0101 P1SEL1	0x00	Port 1 Selection 1 [Memory Mapped]
▶ 1010 0101 P1SELC	0x00	Port 1 Complement Selection [Memory Mapped]
1010 0101 P1IV	0x0000	Port 1 Interrupt Vector Word [Memory Mapped]
▶ 1010 0101 P1IES	0x42	Port 1 Interrupt Edge Select [Memory Mapped]
▶ 1010 0101 P1IE	0x00	Port 1 Interrupt Enable [Memory Mapped]
▶ 1010 0101 P1IFG	0x42	Port 1 Interrupt Flag [Memory Mapped]
▶ 1010 0101 P2IN	0x01	Port 2 Input [Memory Mapped]
▶ 1010 0101 P2OUT	0x18	Port 2 Output [Memory Mapped]
▶ 1010 0101 P2DIR	0x00	Port 2 Direction [Memory Mapped]
▶ 1010 0101 P2REN	0x00	Port 2 Resistor Enable [Memory Mapped]
▶ 1010 0101 P2SEL0	0x00	Port 2 Selection 0 [Memory Mapped]
▶ 1010 0101 P2SEL1	0x00	Port 2 Selection 1 [Memory Mapped]
▶ 1010 0101 P2SELC	0x00	Port 2 Complement Selection [Memory Mapped]
1010 0101 P2IV	0x0000	Port 2 Interrupt Vector Word [Memory Mapped]
▶ 1010 0101 P2IES	0x30	Port 2 Interrupt Edge Select [Memory Mapped]

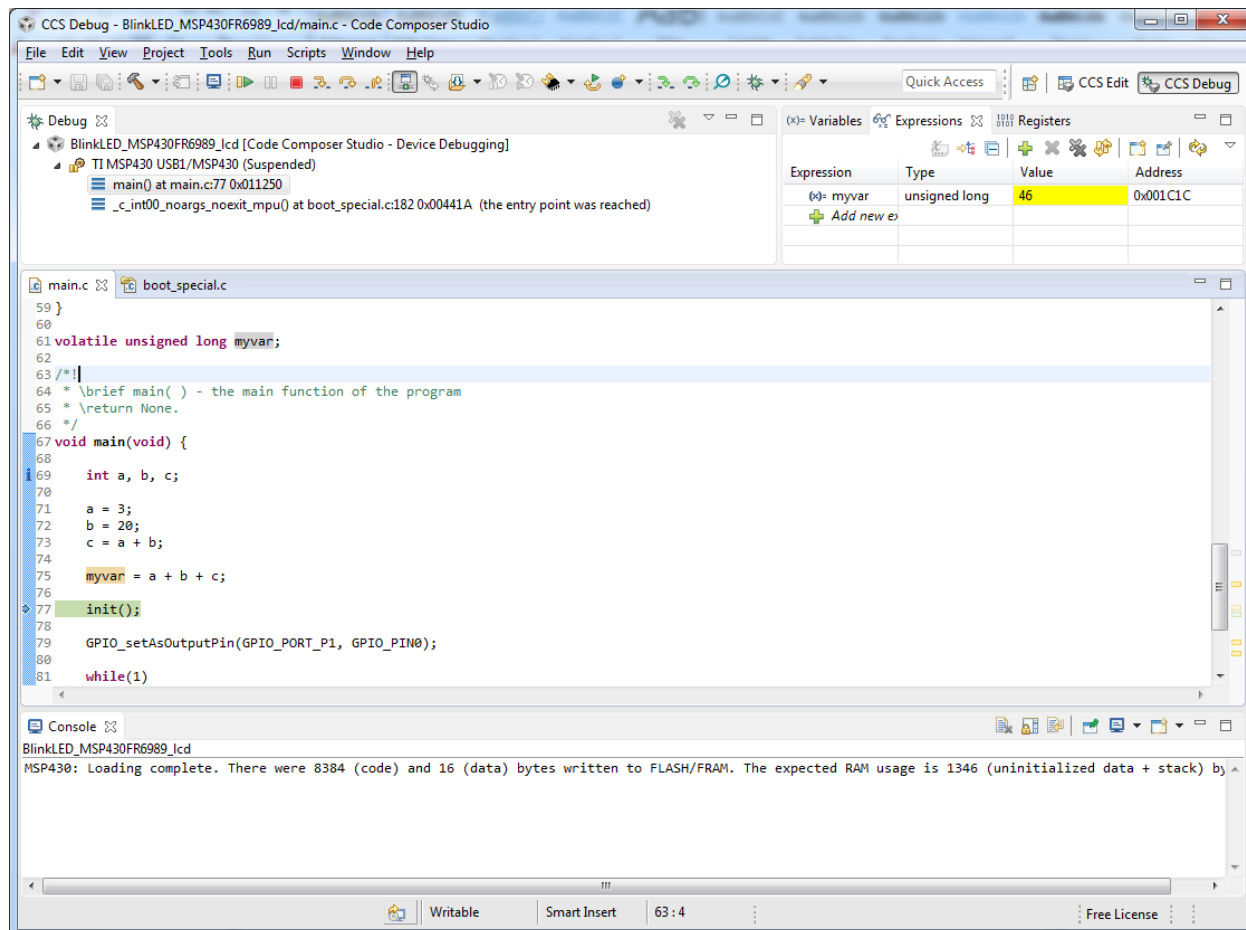


#### 4. Наблюдаване на променливи.

Аналогично на регистрите и променливите във вашата програма също могат да бъдат наблюдавани (променливите всъщност са поместени в регистри от паметта на микроконтролера). За да направите това, изберете таба Variables. Табът Variables показва **САМО** локалните променливи.



За да наблюдавате глобални променливи трябва да изберете таба Expressions и да въведете там тяхното име.

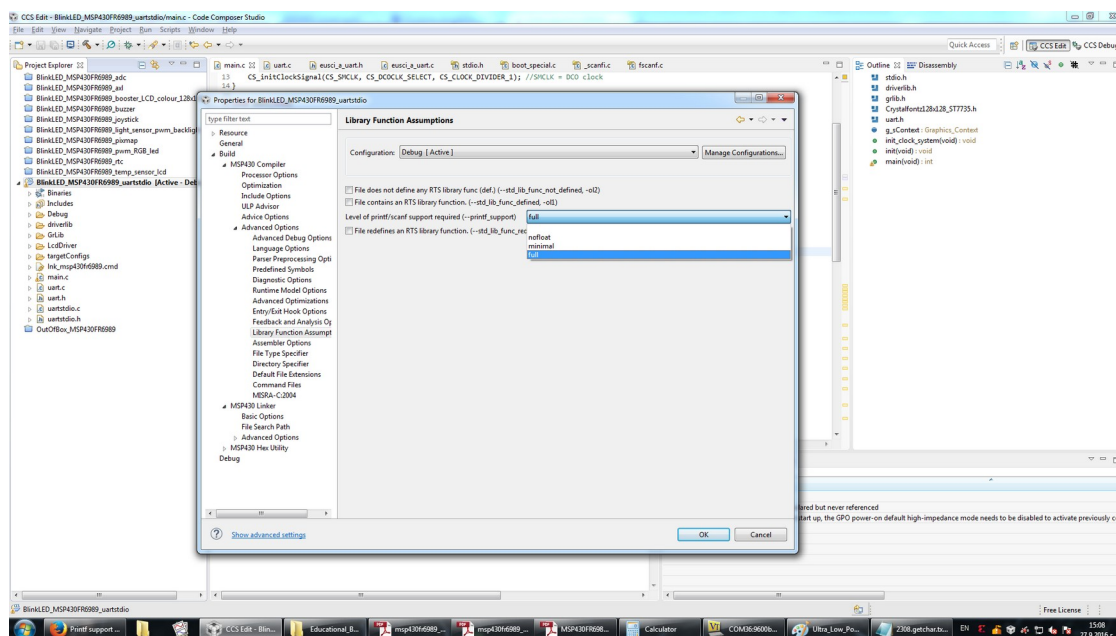


## 5. Printf, scanf, sprintf и стандартни библиотеки.

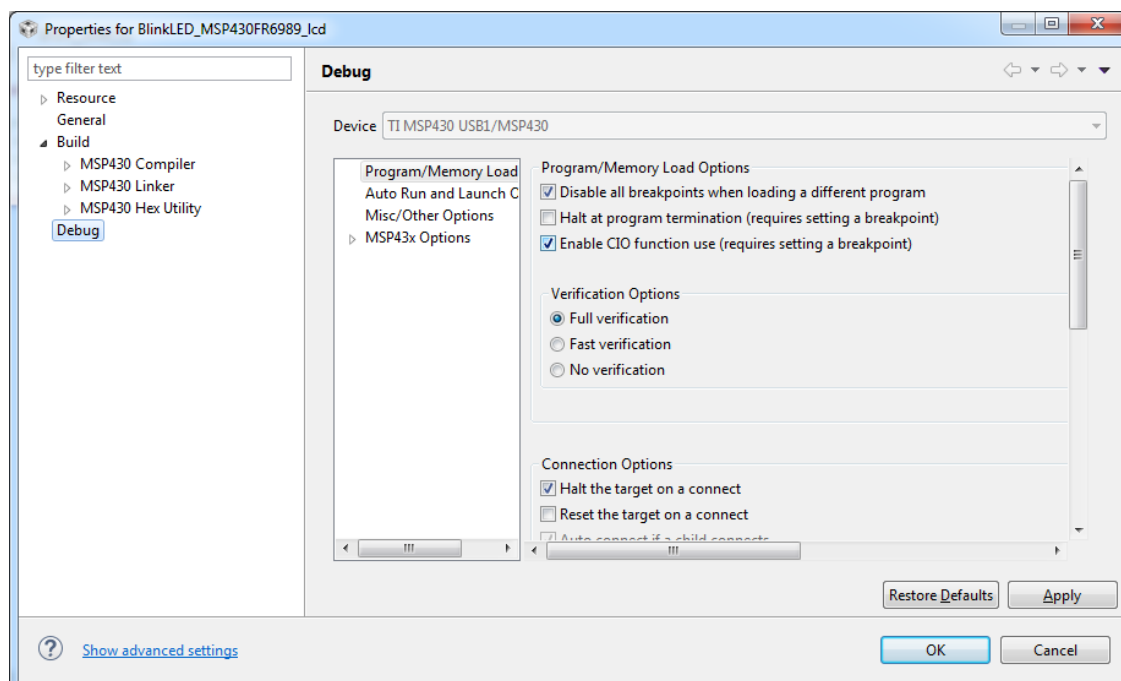
Понякога на програмистите се налага да проследят изпълнението на дадена програма в много голям период от време. Ако тя замръзне поради някаква причина, те могат да видят последните няколко редове код, които са изпълнени. Този процес се нарича трасиране на програмата. Трасирането се води комплексна операция и обикновено фирмите предлагат тази опция само като платена. Тогава на програмиста остава единствено да заложи принтиращи съобщения във най-важните точки на програмата с `printf( )`. Повечето модерни микроконтролери поддържат тази функция, като данните се изобразяват в терминал на компютъра, с който е свързан микроконтролера по някакъв интерфейс.

За да се използва printf в CCS трябва да се изпълнят следните стъпки:

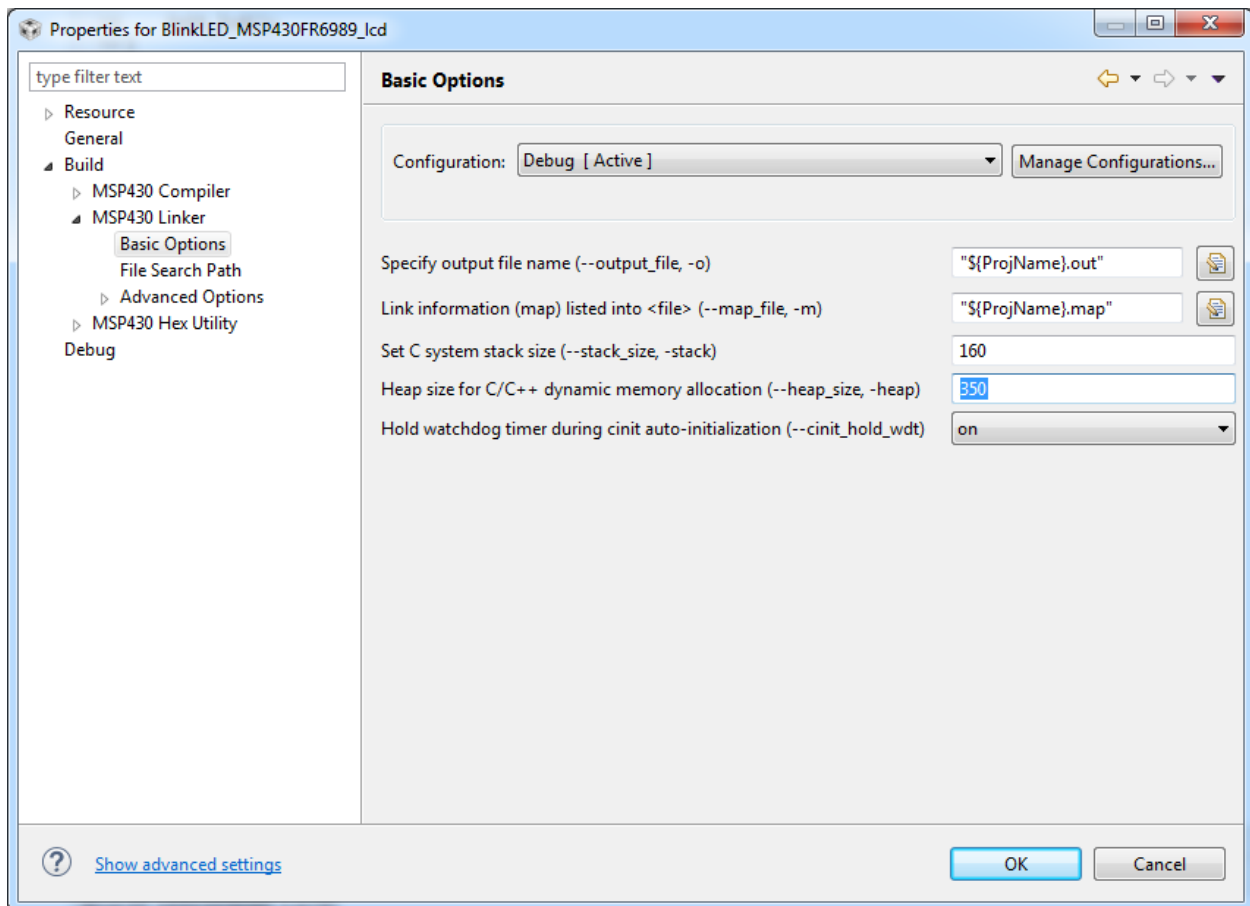
а) Десен бутон върху директорията на проекта ▢ Properties ▢ Build ▢ MSP430 Compiler ▢ Advanced options ▢ Library function assumptions ▢ Level of printf/scanf support required ▢ избира се Full от падащото меню.



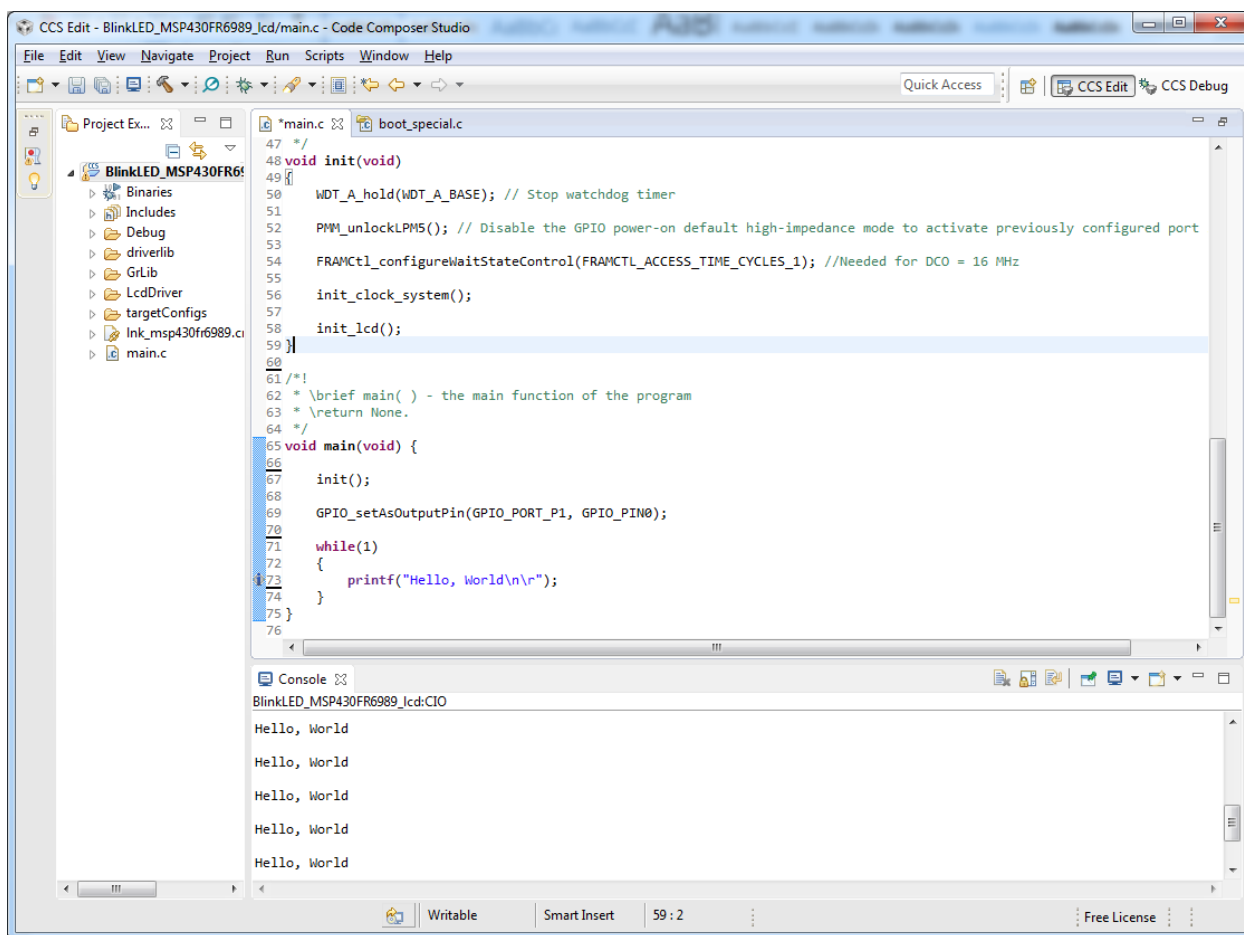
б) Десен бутон върху директорията на проекта ▢ Properties ▢ Build ▢ Debug ▢ Program Memory Load Options ▢ поставете отметка върху “Enable CIO function use (requires setting a breakpoint)”



в) Printf е функция, която използва много динамично заделена памет. По подразбиране сегментът от паметта наречен Heap за MSP430FR6989 е 160 байта. Той трябва да е поне 300 – 350 при използване на printf( ). За да се промени: десен бутон върху директорията на проекта □ Properties □ Build □ MSP430 Linker □ Basic Options □ Heap size for C/C++ dynamic memory allocation □ въвежда се числото 350.



Резултатите от този `printf( )` са видими в терминала на CCS:



Към момента на създаването на това ръководство **`scanf( )`** функцията не работи!

Функцията **`sprintf( )`** е напълно работоспособна.

Функциите за обработка на низове (`strcpy`, `strlen`, `strcat`, и т.н.) са напълно работоспособни. Включете хедърния файл `<string.h>`.

## 6. Оптимизиран `printf`.

Стандартните библиотеки заемат много от програмната и даннова памет на микроконтролера. В по-сложните проекти може това да се окаже проблем. Тогава може да се използва оптимизираната версия `uartstdio`, точно както



беше направено в лабораторните упражнения по Микропроцесорна схемотехника. За целта копирайте файловете:

*uart.h*

*uart.c*

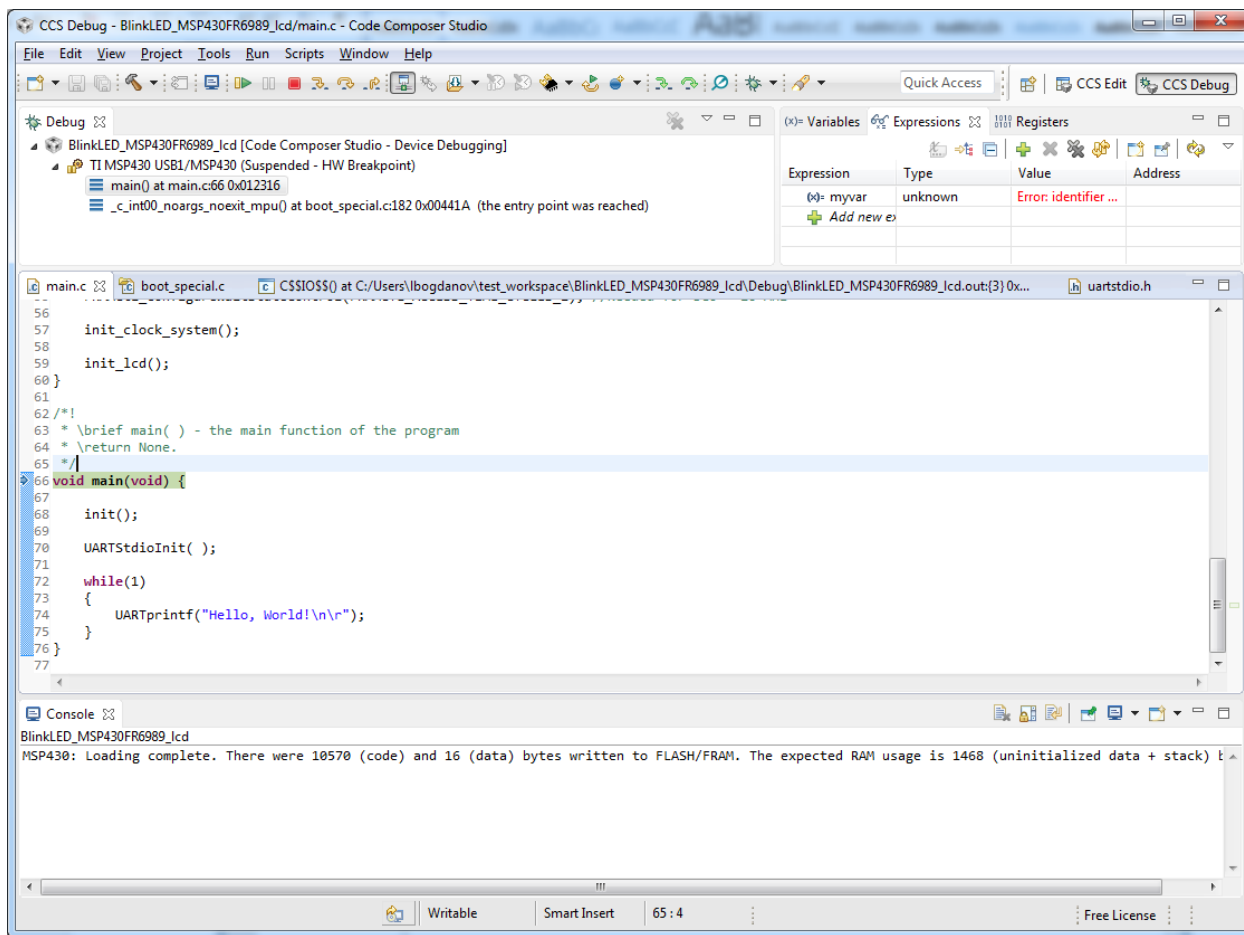
*uartstdio.h*

*uartstdio.c*

от директорията MSHT\_PRAKT\src\_drivers\uartstdio в директорията на вашия проект. Те автоматично ще се появят в дървото на проекта в CCS (ако не – натиснете F5). След това включете хедърния файл

*#include "uartstdio.h"*

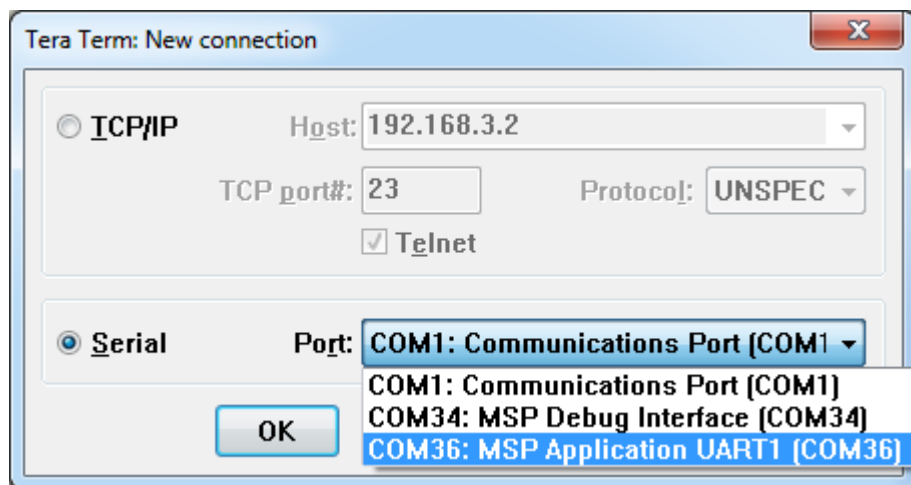
в началото на вашия *main.c*. Във функцията *main( )* извикайте инициализиращата функция *UARTStdioInit( )* и след това *UARTprintf( )*.



За да видите резултата трябва да отворите отделна терминална програма, която изобразява данните от RS232 интерфейса. При макета MSP-

EXP430FR6989 изходът и входът на UART1 на микроконтролера са свързани към дебъгера, който осигурява виртуален RS232.

Примерна такава терминална програма е Tera Term. При стартирането ѝ се посочва приложния виртуален COM порт на дебъгера:



Настройката на интерфейса трябва да е:

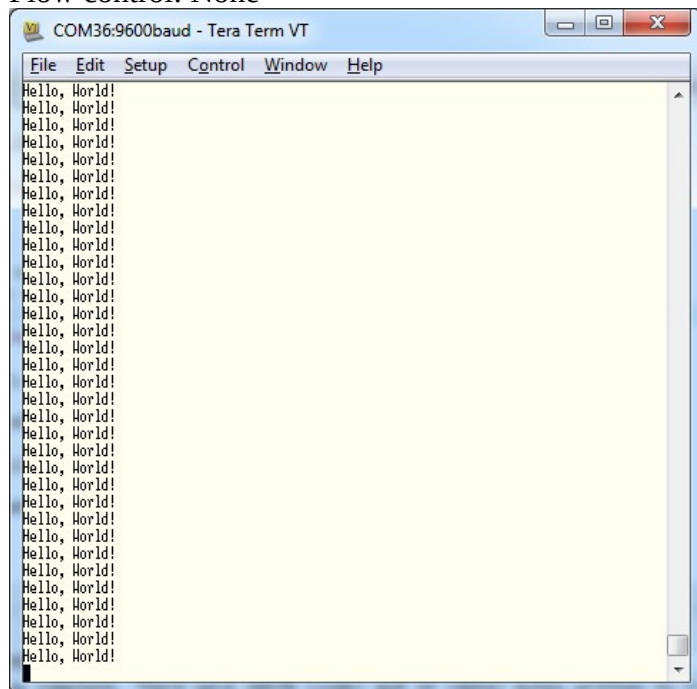
Baud rate: 9600

Data bits: 8

Parity: None

Stop bits: 1

Flow control: None



## 7. Размер на програмата.

MSP430FR6989 е микроконтролер със 128 kB програмна и 2 kB даннова памет. Потребителската програма трябва да е с размер по-малък или равен на тези стойности. След натискането на бутона за зареждане на програмата в терминала на CCS се изписва съобщението:

```
MSP430: Loading complete. There were 10570 (code) and 16 (data) bytes written to FLASH/FRAM. The expected RAM usage is 1468 (uninitialized data + stack) bytes.
```

Сумата на числата от сегментите *code* и *data* е размера на програмата, която ще се запише в програмната памет. Сумата от числата на *uninitialized data*, *stack* и *heap* дават размера на паметта, която ще се записва/чете в данновата памет. Неар-ът расте динамично и ако неговата стойност превиши предварително зададена стойност, програмата ви ще започне да замръзва без причина. Затова е добре „the expected RAM usage” да е възможно по-малко от 2048 байта.

### III. Работа с MSPWare

На лабораторните упражнения по Микропроцесорна схемотехника бяха използвани програми, които работят чрез т.нар. директен регистров достъп. Например, за да се направи един извод като GPIO изход и стойността му да се промени на нула, след това на единица трябваше да се запише:

```
P1DIR |= 0x02; //Извод P1.1 изход
```

```
P1OUT &= ~0x02; //Извод P1.1 в логическа нула
```

```
P1OUT |= 0x02; //Извод P1.1 в логическа единица
```

Това е най-бързият код на C, който може да се напише за реализирането на тази функция. В практиката обаче програмистите използват *библиотечни функции*, които правят регистровия достъп вместо нас. Това помага да се повиши нивото на абстракция и следователно даден код лесно да може да се пренесе на друг микроконтролер от същата фамилия. Аналогична програма може да се запише като:

```
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN1);
```

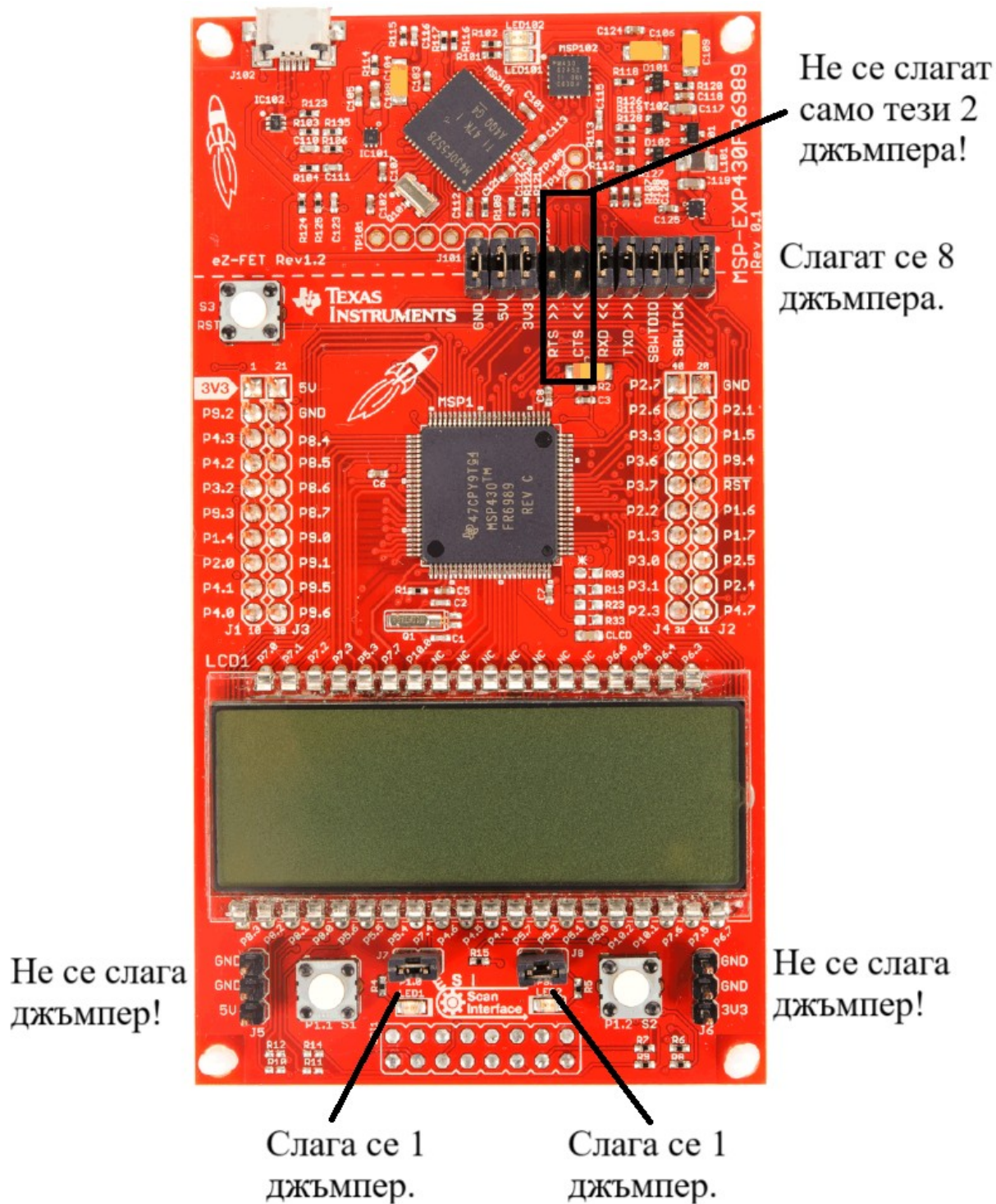
```
GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN1);
```

```
GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN1);
```

Тези API (Application Programmer Interface) функции, освен че достъпват до регистрите, правят и проверка на подаваните параметри. Като резултат програмата леко се забавя (заради извикването на функцията, проверката за грешки и излизането от функцията), но програмирането се улеснява.

За да използвате библиотеката MSPWare разгледайте директорията **driverlib\MSP430FR5xx\_6xx** в template проекта. Всички периферни модули на MSP430FR6989 имат функции за работа с тях. Например за работа с АЦП отворете `adc12_b.h` и вижте кои функции може да извикате. Аналогично за работа с таймер - `timer_b.h` и така нататък.

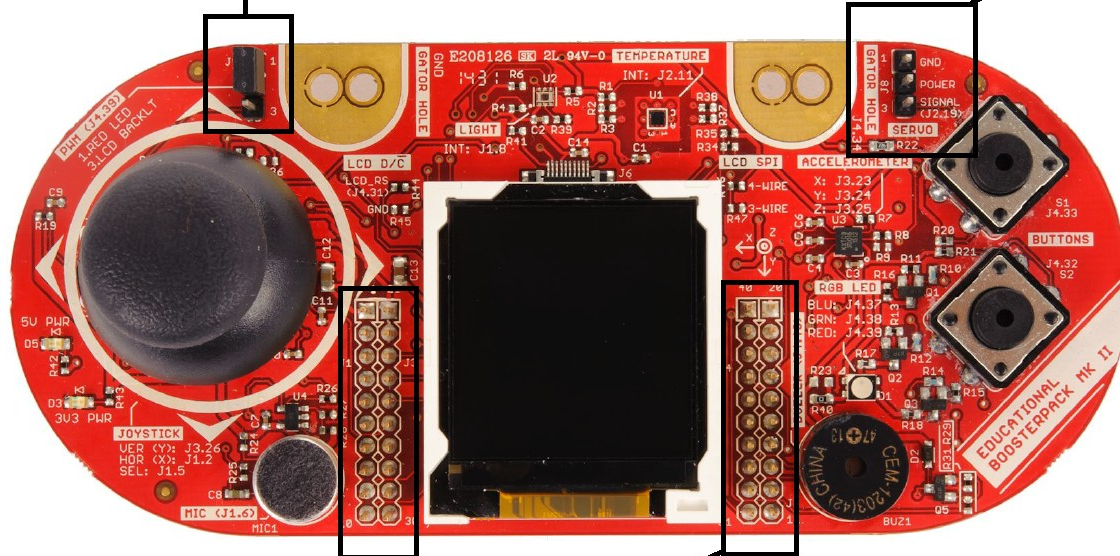
## IV. Конфигурация на джъмперите





Слага се джъмпер.  
 Позиция 1-2 = червено на LED D1.  
 Позиция 2-3 = подсветка на LCD  
 матричен дисплей 128x128.

Не се слагат  
 джъмperi!



Не се слагат  
 джъмperi!

\*

\*

\*

доц. д-р инж. Любомир Богданов, 2025 г.