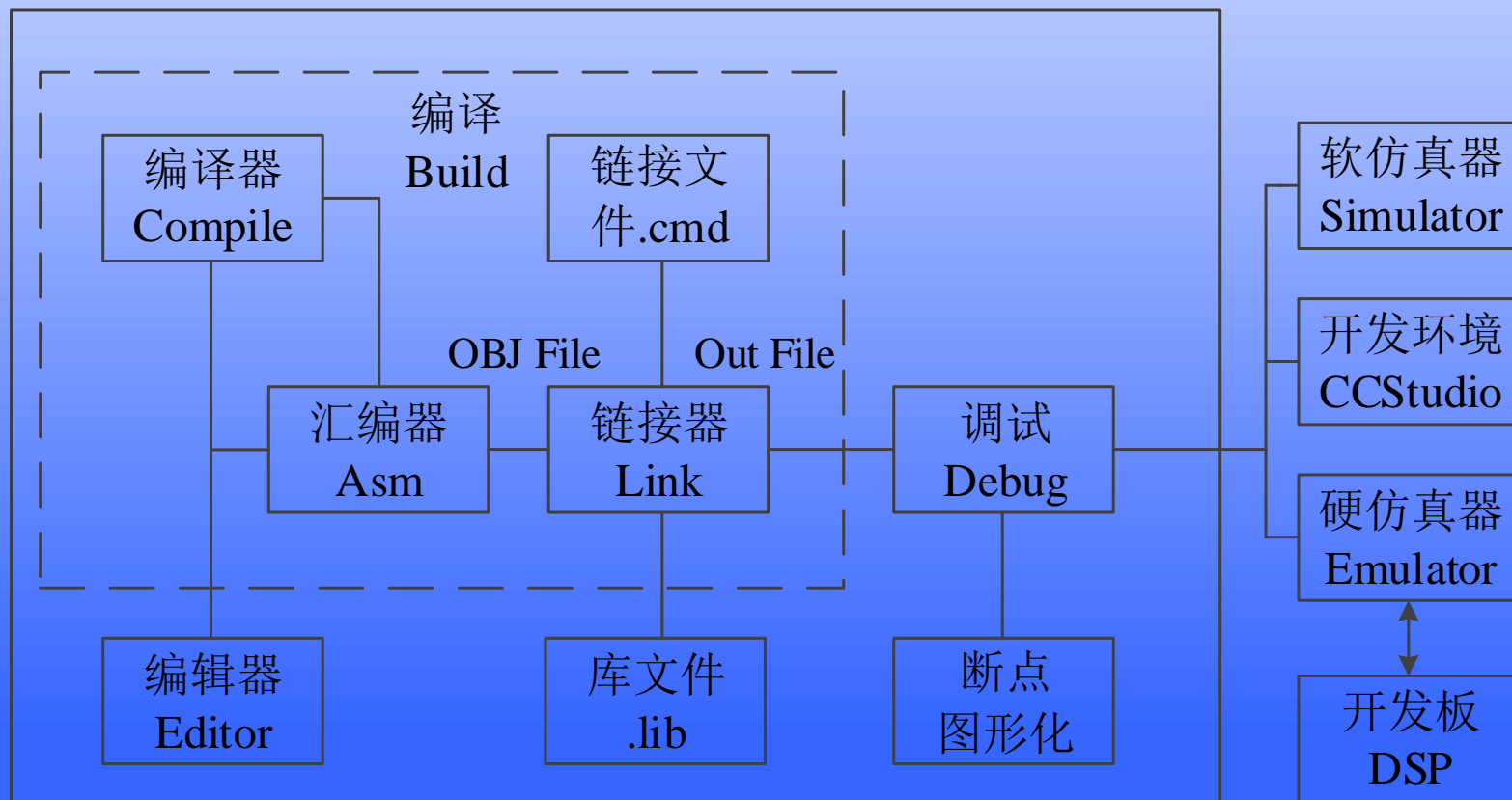


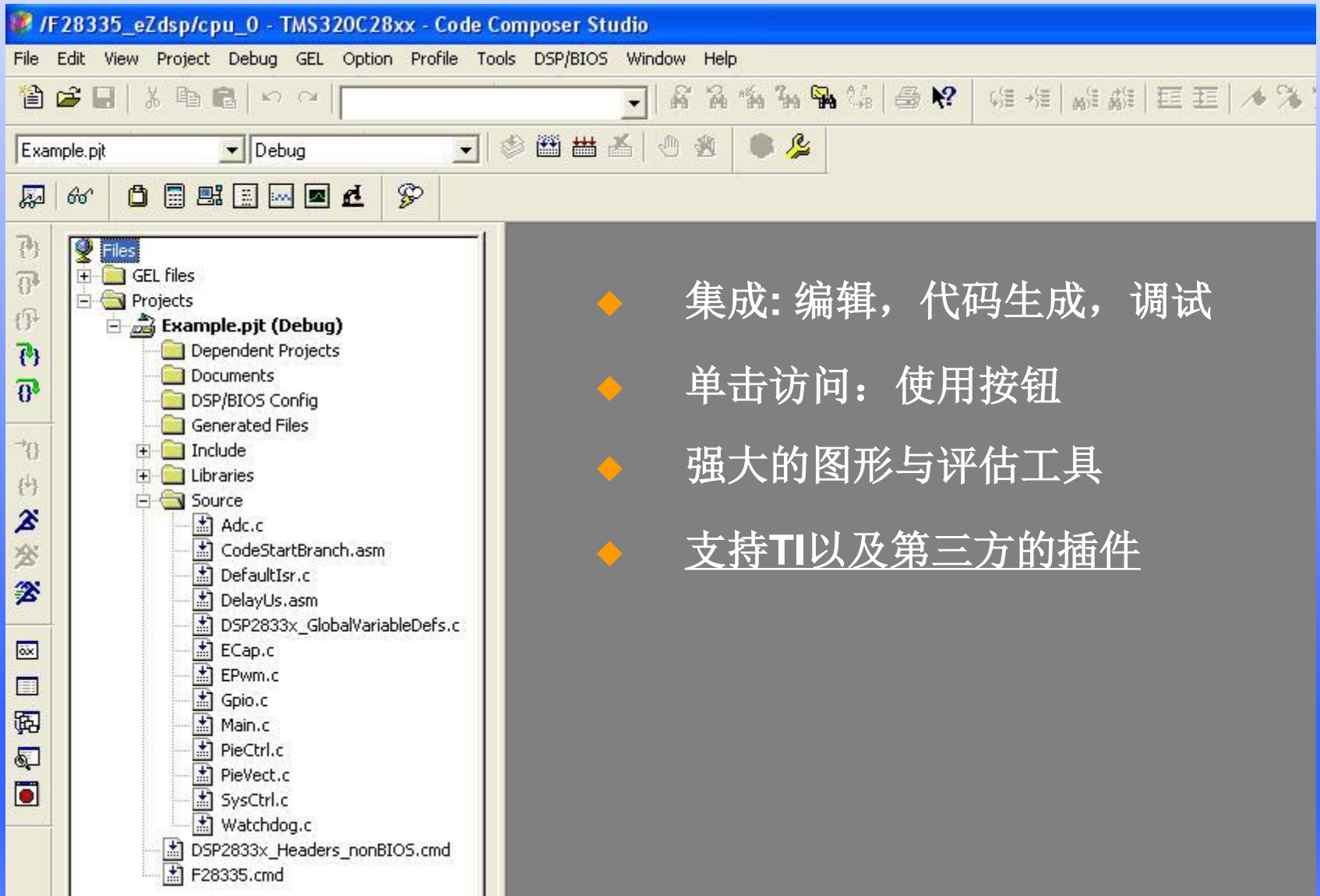
学习目标

- ◆ **Code Composer Studio**软件的基本使用方法
- ◆ 如何创建链接命令文件**cmd**
- ◆ 理解外设寄存器的头文件

CCS开发环境的基本框架



Code Composer Studio: IDE

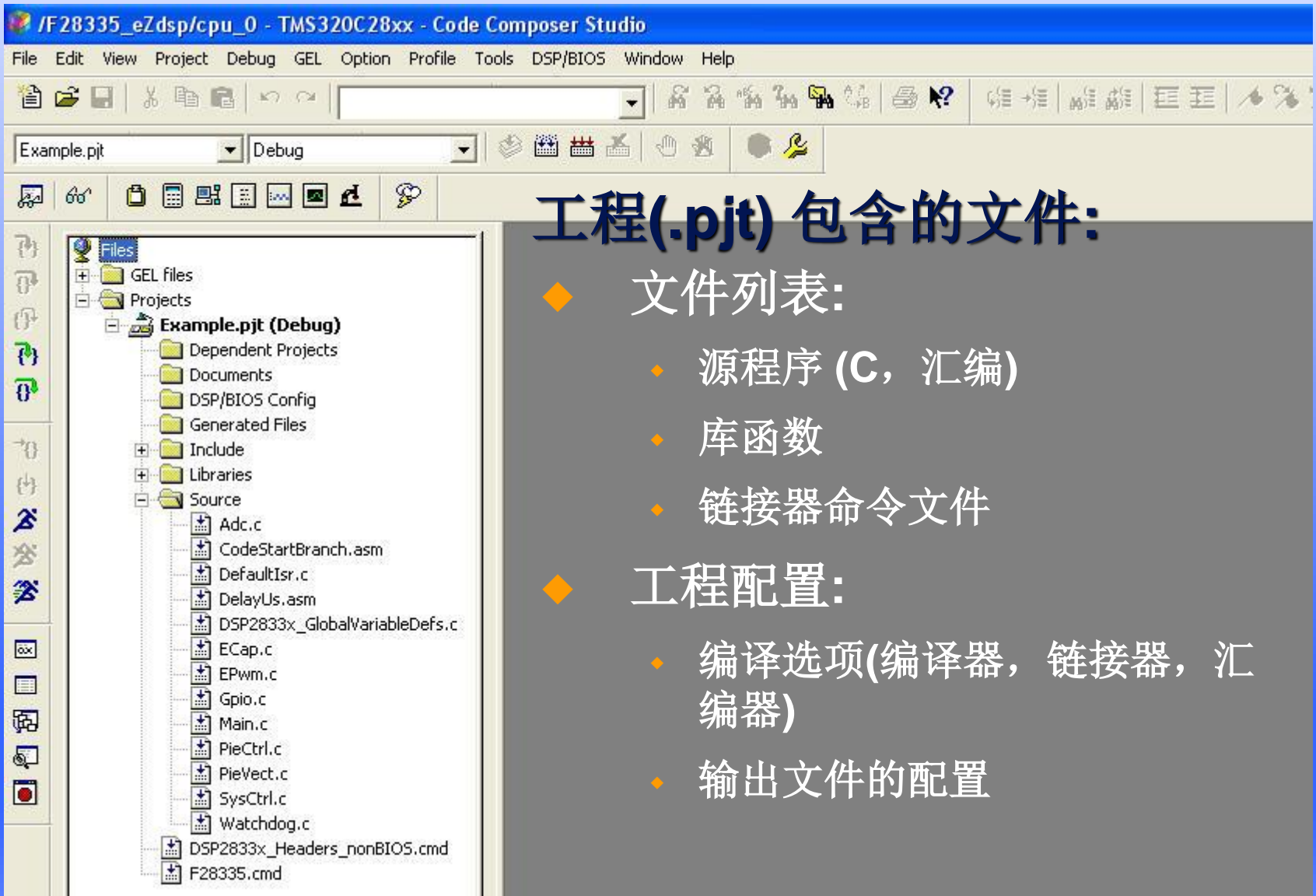


The screenshot displays the Code Composer Studio IDE interface. The title bar reads "/F28335_eZdsp/cpu_0 - TMS320C28xx - Code Composer Studio". The menu bar includes File, Edit, View, Project, Debug, GEL, Option, Profile, Tools, DSP/BIOS, Window, and Help. The toolbar contains various icons for file operations, editing, and debugging. The main workspace shows a project tree for "Example.pjt (Debug)". The tree structure is as follows:

- Files
 - GEL files
 - Projects
 - Example.pjt (Debug)
 - Dependent Projects
 - Documents
 - DSP/BIOS Config
 - Generated Files
 - Include
 - Libraries
 - Source
 - Adc.c
 - CodeStartBranch.asm
 - DefaultIsr.c
 - DelayUs.asm
 - DSP2833x_GlobalVariableDefs.c
 - ECap.c
 - EPwm.c
 - Gpio.c
 - Main.c
 - PieCtrl.c
 - PieVect.c
 - SysCtrl.c
 - Watchdog.c
 - DSP2833x_Headers_nonBIOS.cmd
 - F28335.cmd

- ◆ 集成: 编辑, 代码生成, 调试
- ◆ 单击访问: 使用按钮
- ◆ 强大的图形与评估工具
- ◆ 支持TI以及第三方的插件

CCS的工程结构



The screenshot shows the Code Composer Studio interface for a project named 'Example.pjt' in 'Debug' mode. The file explorer on the left displays the following structure:

- Files
 - GEL files
 - Projects
 - Example.pjt (Debug)**
 - Dependent Projects
 - Documents
 - DSP/BIOS Config
 - Generated Files
 - Include
 - Libraries
 - Source
 - Adc.c
 - CodeStartBranch.asm
 - DefaultIsr.c
 - DelayUs.asm
 - DSP2833x_GlobalVariableDefs.c
 - ECap.c
 - EPwm.c
 - Gpio.c
 - Main.c
 - PieCtrl.c
 - PieVect.c
 - SysCtrl.c
 - Watchdog.c
 - DSP2833x_Headers_nonBIOS.cmd
 - F28335.cmd

工程(.pjt) 包含的文件:

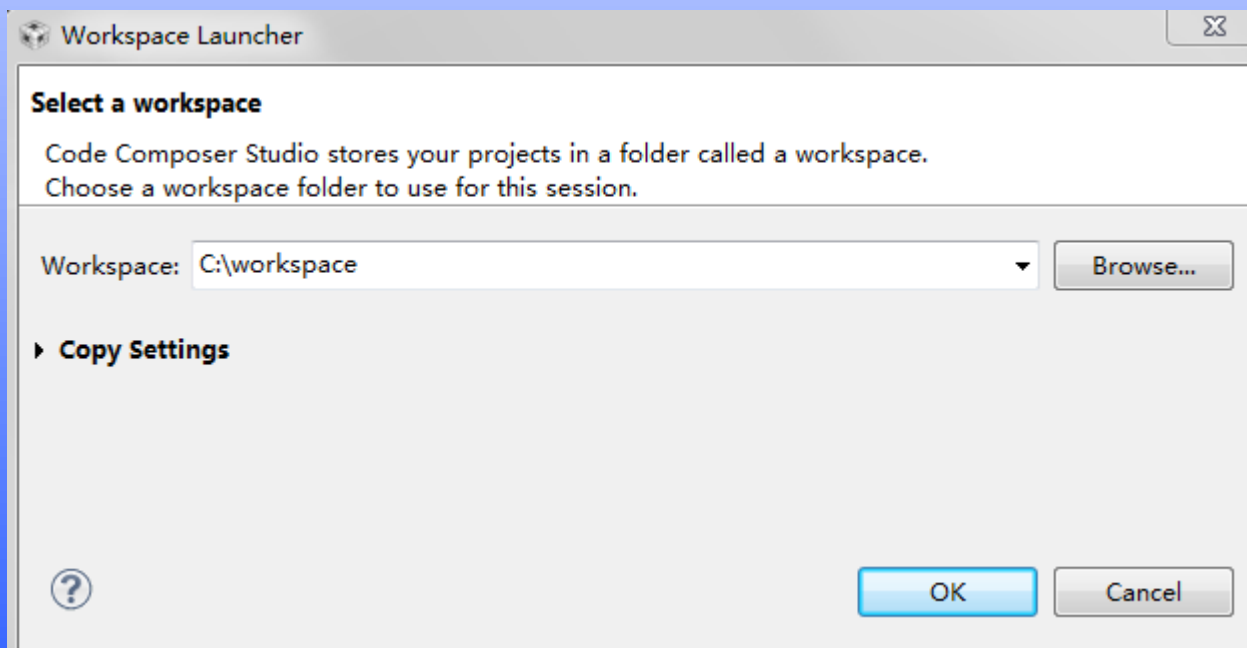
◆ 文件列表:

- ◆ 源程序 (C, 汇编)
- ◆ 库函数
- ◆ 链接器命令文件

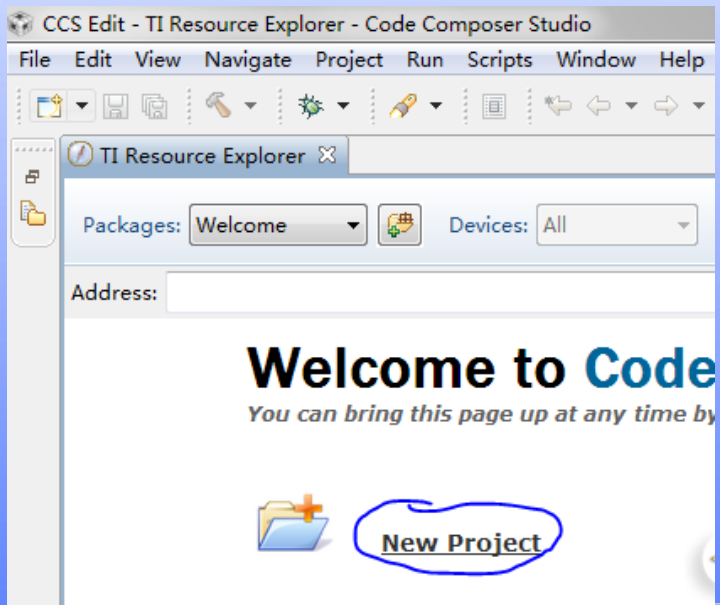
◆ 工程配置:

- ◆ 编译选项(编译器, 链接器, 汇编器)
- ◆ 输出文件的配置

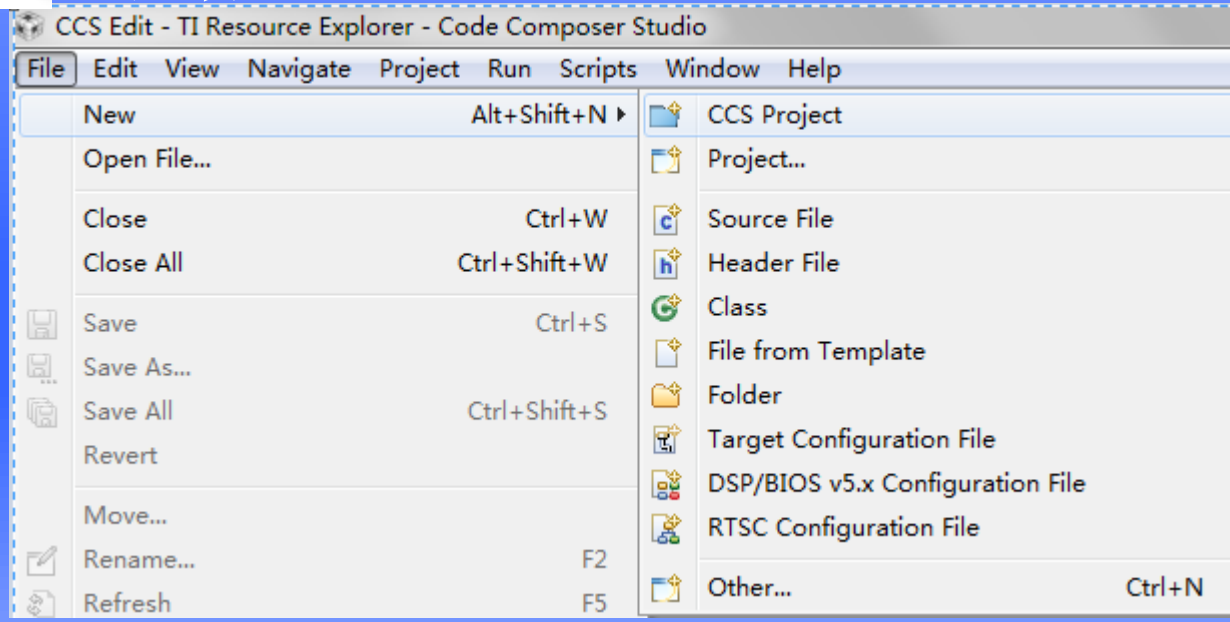
CCS 4.x/5.x/6.x的基本使用-创建工作空间



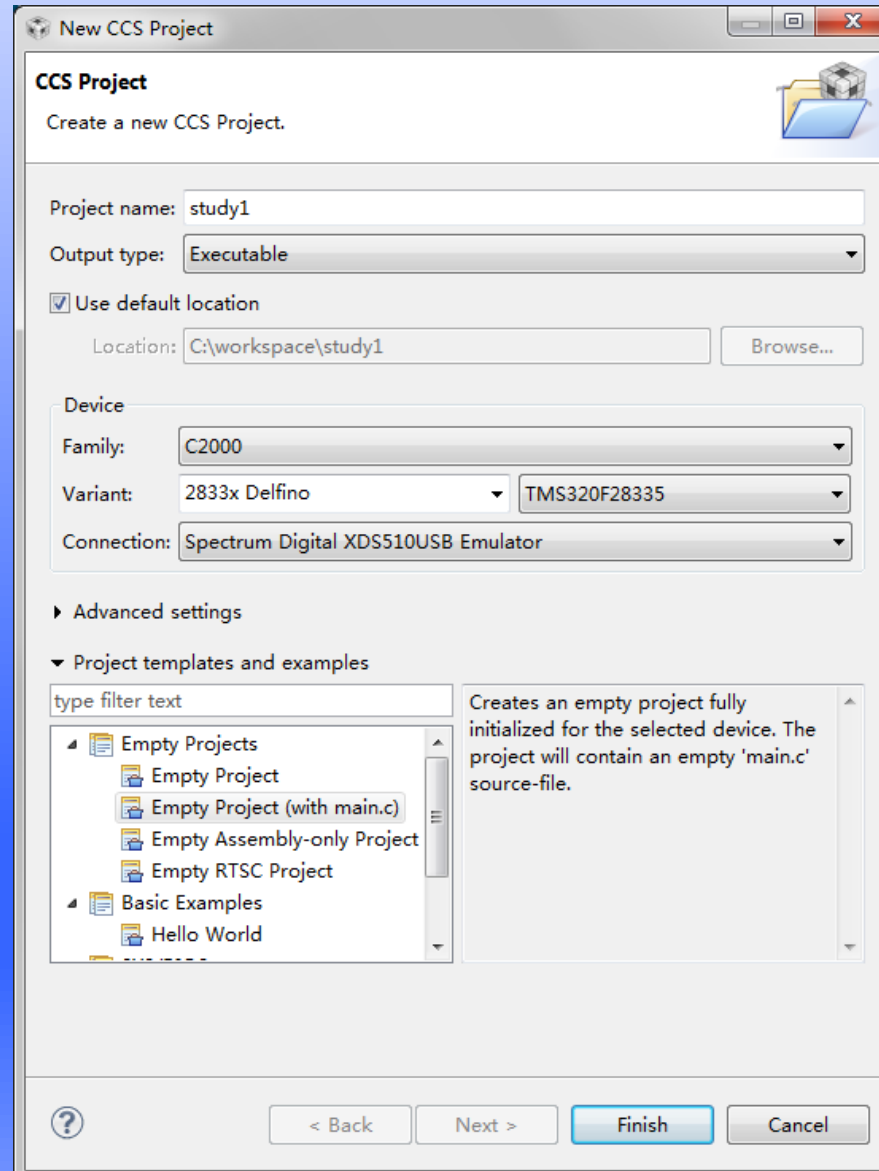
CCS 4.x/5.x/6.x的基本使用-新建工程



或者



CCS 4.x/5.x/6.x的基本使用-选择器件



CCS 4.x/5.x/6.x的基本使用-测试硬件连接

The screenshot displays the CCS IDE interface. On the left, the Project Explorer shows a project named 'study1' in 'Active - Debug' mode. The project structure includes 'Includes', 'targetConfigs' (containing 'README.txt' and 'TMS320F28335.ccxml [Active/Default]'), '28335_RAM_Ink.cmd', and 'main.c' (containing 'main(void) : int').

The main window is titled 'TI Resource Explorer' and shows the 'Basic' configuration page for the target. The 'General Setup' section describes the general configuration about the target. The 'Connection' is set to 'Spectrum Digital XDS510USB Emulator'. The 'Board or Device' section contains a list of TI devices with checkboxes:

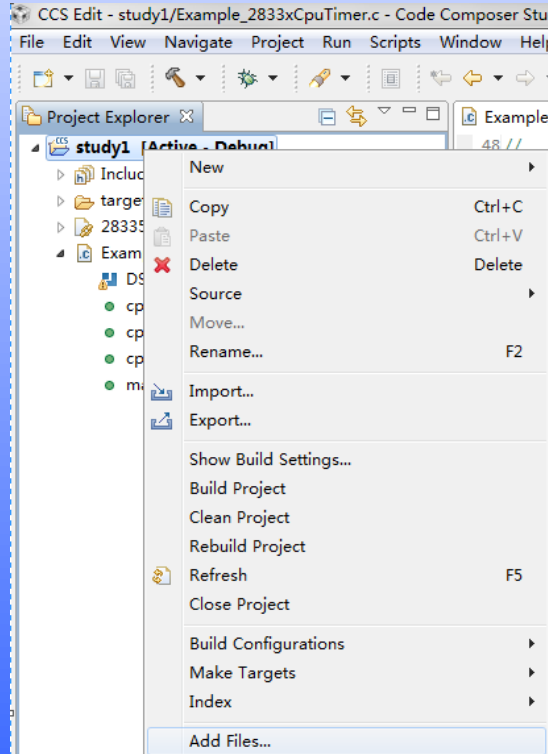
- TMS320F2809
- TMS320F2810
- TMS320F2811
- TMS320F2812
- TMS320F28232
- TMS320F28234
- TMS320F28235
- TMS320F28332
- TMS320F28334
- TMS320F28335

Test Connection

To test a connection, all changes must have been saved, the configuration file contains no errors and the connection type supports this function.

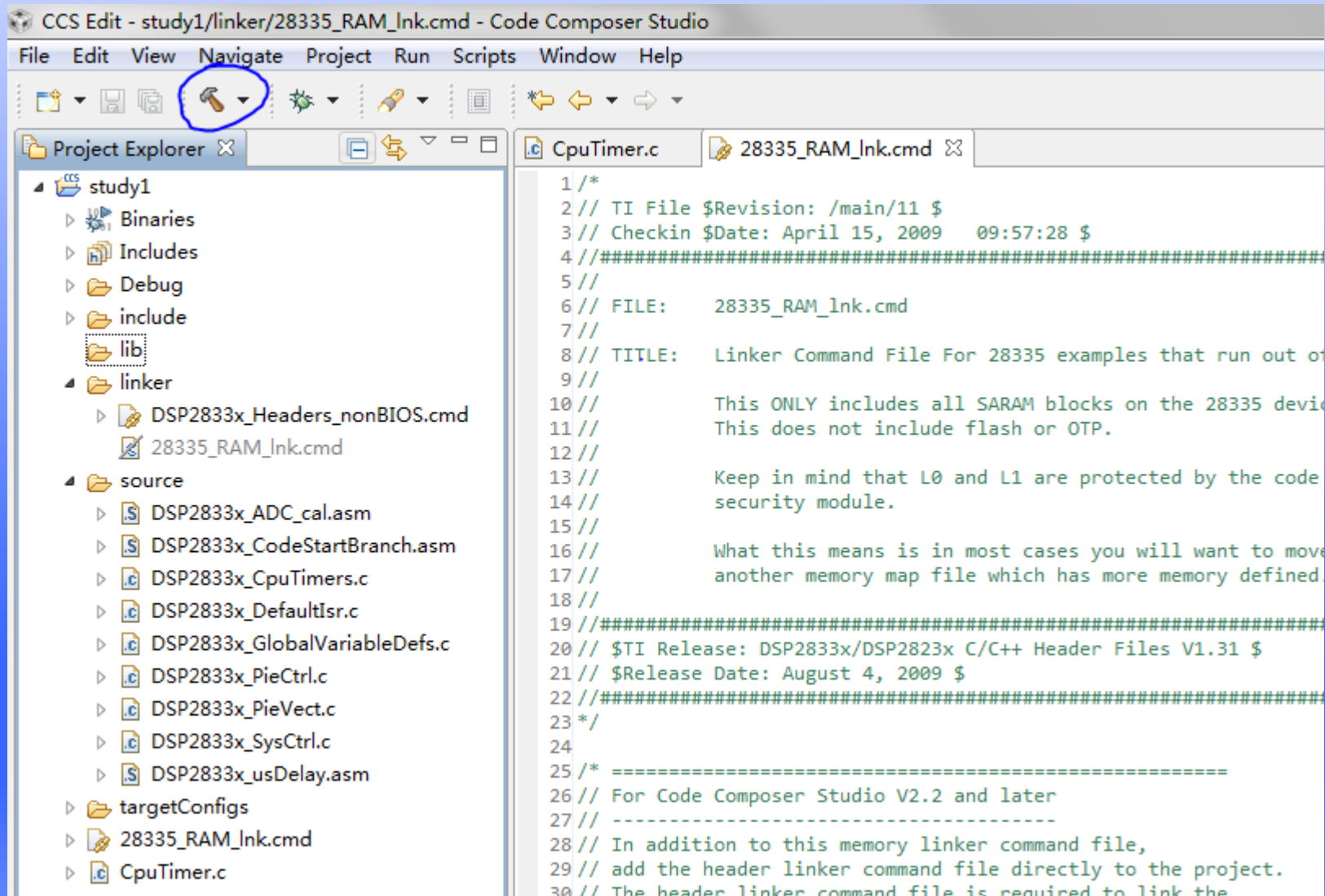
Test Connection

CCS 4.x/5.x/6.x的基本使用-添加文件

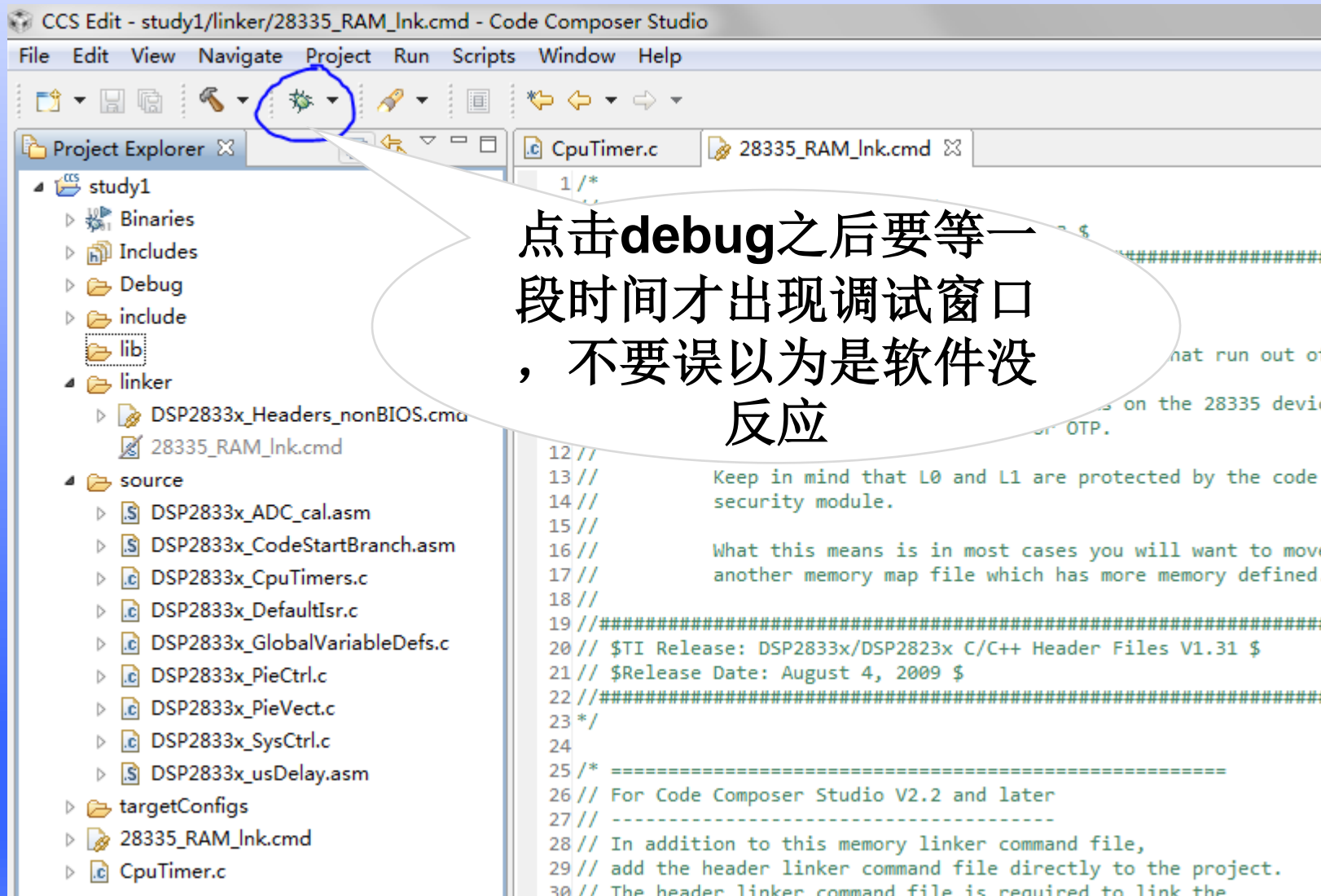


添加几个文件

CCS 4.x/5.x/6.x的基本使用-编译工程

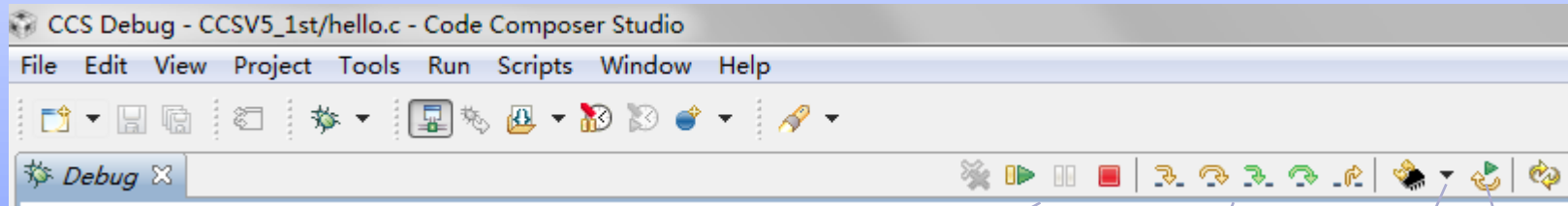


CCS 4.x/5.x/6.x的基本使用-调试工程



The screenshot shows the Code Composer Studio (CCS) interface. The title bar reads "CCS Edit - study1/linker/28335_RAM_Ink.cmd - Code Composer Studio". The menu bar includes "File", "Edit", "View", "Navigate", "Project", "Run", "Scripts", "Window", and "Help". The toolbar contains various icons, with the "Debug" icon (a green bug) circled in blue. The Project Explorer on the left shows a project named "study1" with folders for "Binaries", "Includes", "Debug", "include", "lib", "linker", "source", "targetConfigs", and files like "28335_RAM_Ink.cmd" and "CpuTimer.c". The main editor window displays the content of "28335_RAM_Ink.cmd", which includes comments about code security and memory linker command files. A callout box with a white background and black border points to the debug button, containing the text: "点击debug之后要等一段时间才出现调试窗口，不要误以为是软件没反应" (After clicking debug, you have to wait for a period of time before the debug window appears, do not mistakenly think the software is not responding).

CCS 4.x/5.x6.x的基本使用-调试工程



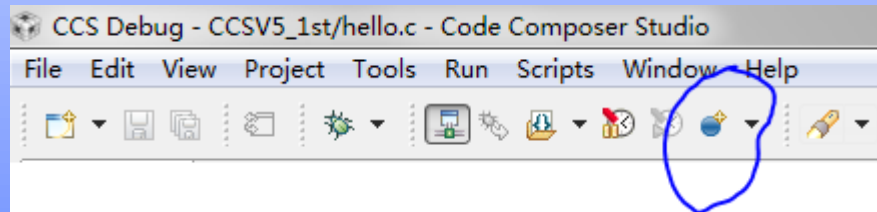
运行、暂停、终止

单步调试相关

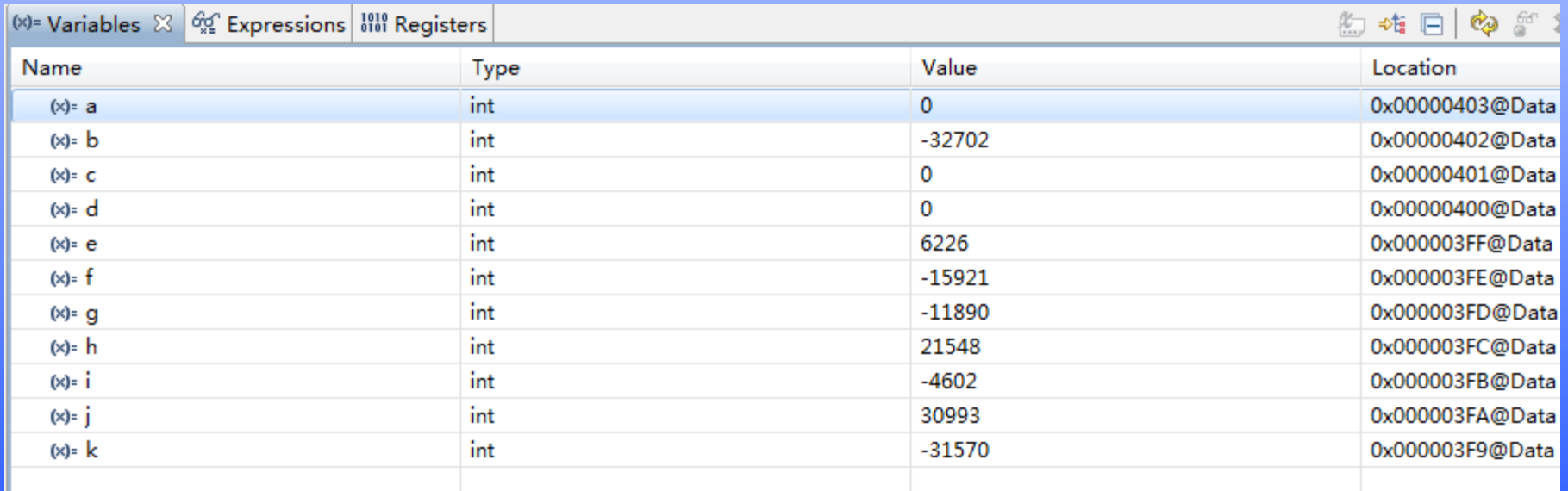
Reset
CPU

重新开始

CCS 4.x/5.x/6.x的基本使用-放置断点



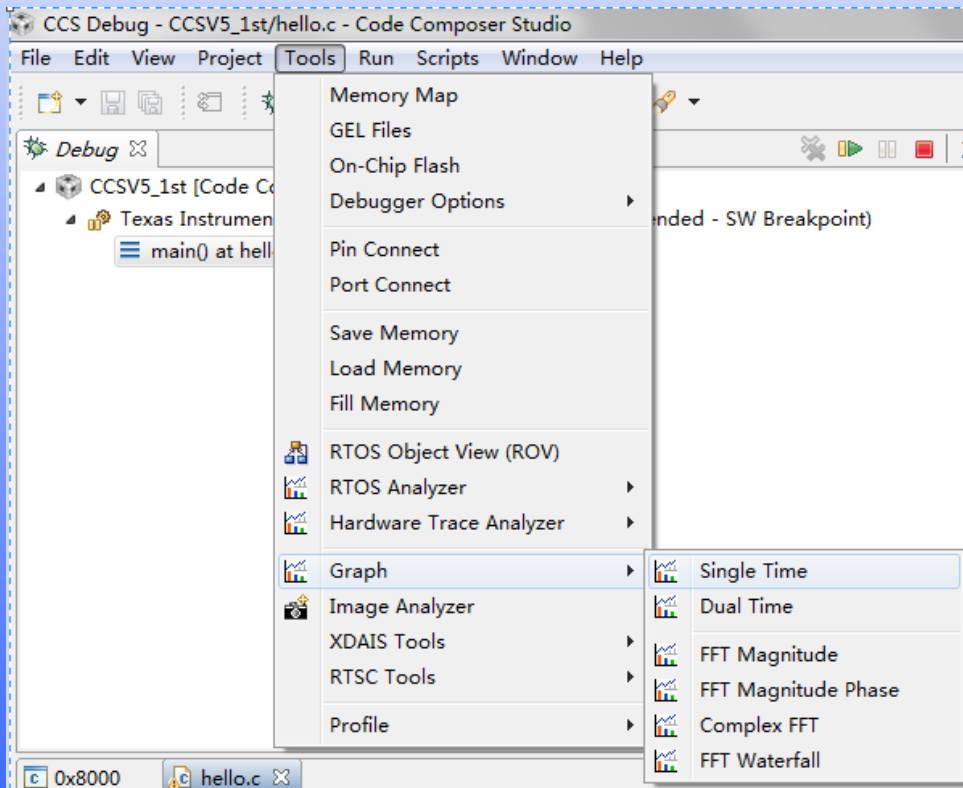
CCS 4.x/5.x/6.x的基本使用-观测变量



The screenshot shows the 'Variables' tab in the CCS IDE. The window title is '(x)- Variables'. The tabs are 'Variables', 'Expressions', and 'Registers'. The table below lists the variables and their values.

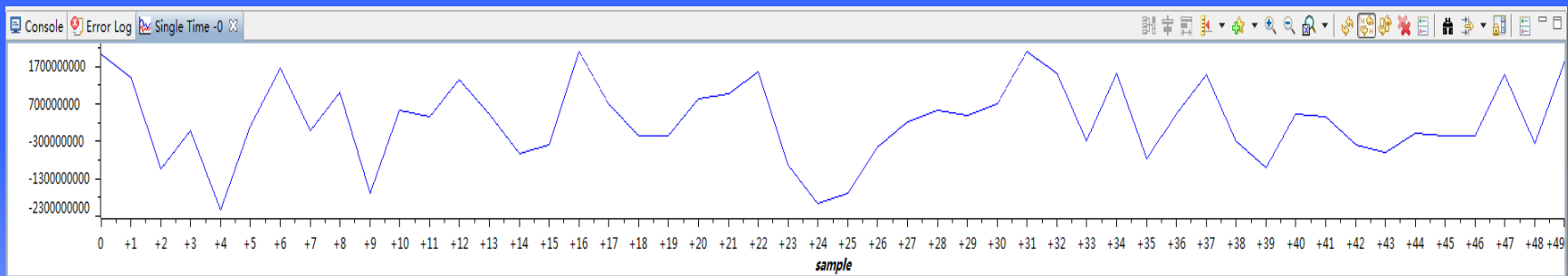
Name	Type	Value	Location
(x)- a	int	0	0x00000403@Data
(x)- b	int	-32702	0x00000402@Data
(x)- c	int	0	0x00000401@Data
(x)- d	int	0	0x00000400@Data
(x)- e	int	6226	0x000003FF@Data
(x)- f	int	-15921	0x000003FE@Data
(x)- g	int	-11890	0x000003FD@Data
(x)- h	int	21548	0x000003FC@Data
(x)- i	int	-4602	0x000003FB@Data
(x)- j	int	30993	0x000003FA@Data
(x)- k	int	-31570	0x000003F9@Data

CCS 4.x/5.x/6.x的基本使用-图形显示

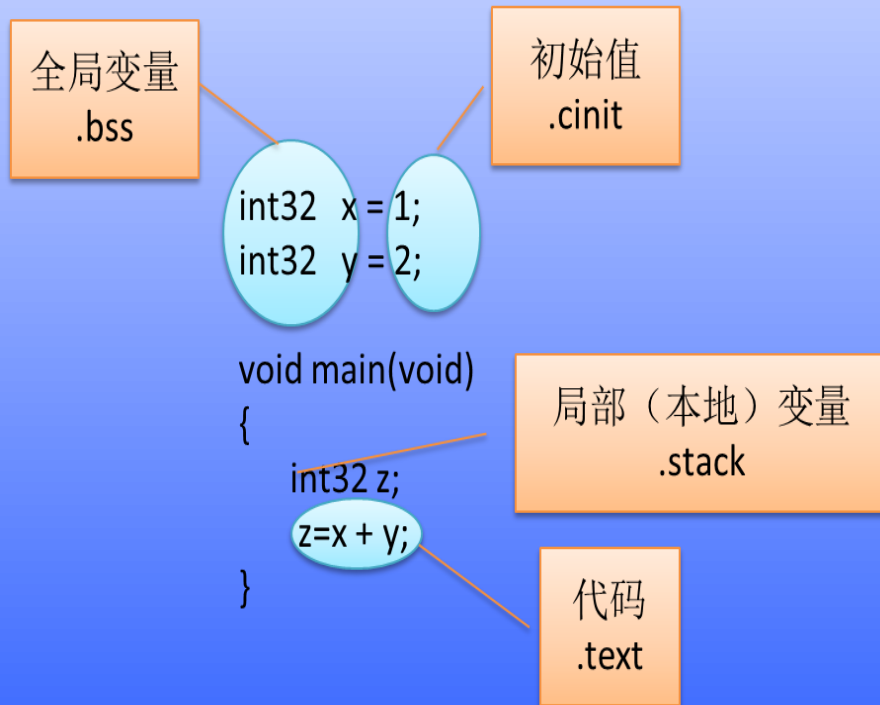


Graph Properties

Property	Value
▲ Data Properties	
Acquisition Buffer Size	50
Dsp Data Type	32 bit signed integer
Index Increment	1
Q_Value	0
Sampling Rate Hz	1000
Start Address	&k
▲ Display Properties	
Axis Display	<input checked="" type="checkbox"/> true
Data Plot Style	Line
Display Data Size	200
Grid Style	No Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
Use Dc Value For Graph	<input type="checkbox"/> false



段Sections



- ◆ 代码按照不同的类型保存，即‘段’
- ◆ 每个段都以“.”开头
- ◆ 段有默认的名字

cmd文件中各个段的含义

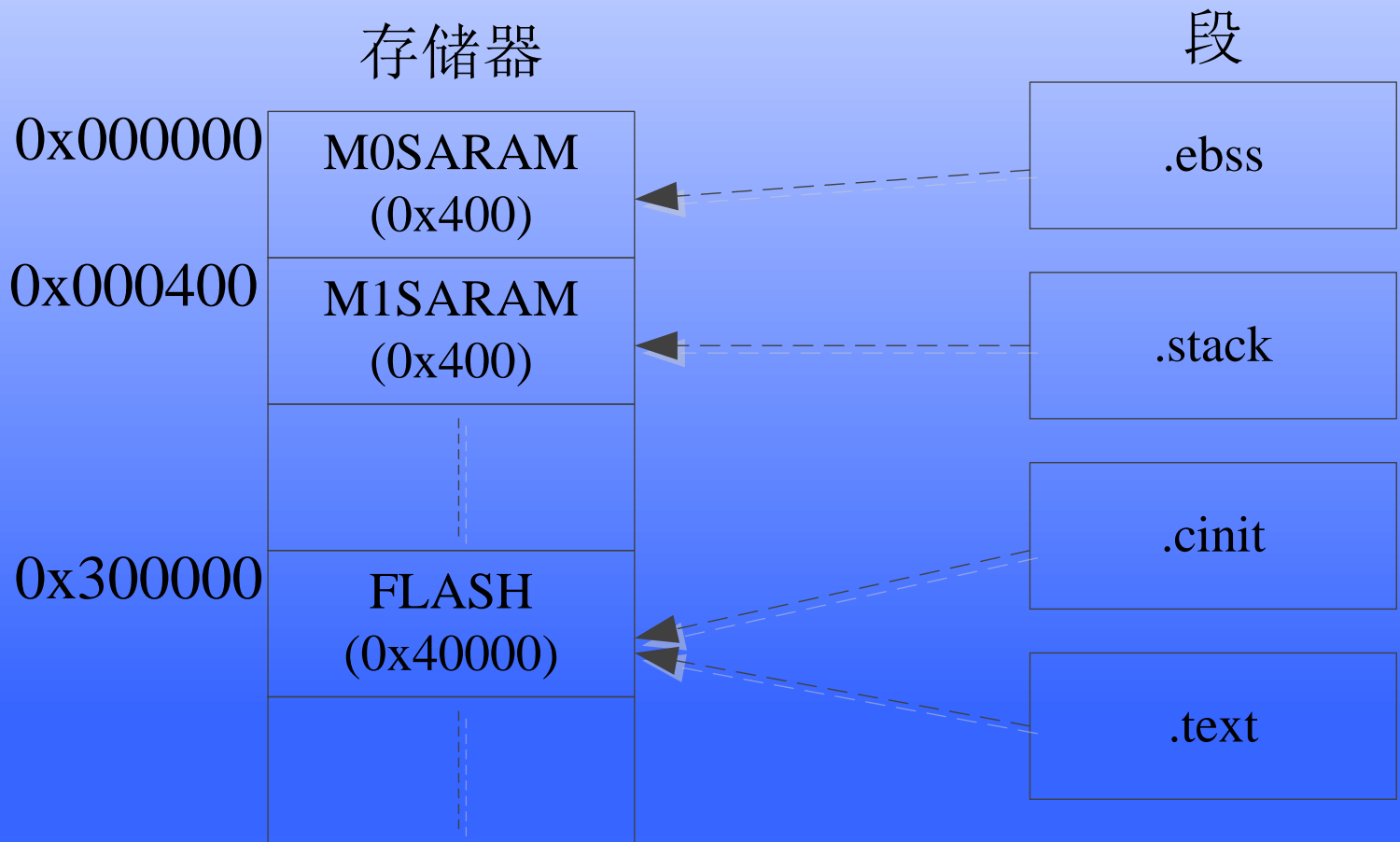
初始化的段

名字	描述	链接位置
<code>.text</code>	代码	FLASH
<code>.cinit</code>	全局与静态变量的初始值	FLASH
<code>.econst</code>	常数	FLASH
<code>.switch</code>	Switch表达式的表格	FLASH
<code>.pinit</code>	全局构造函数表（C++里面的constructor）	FLASH

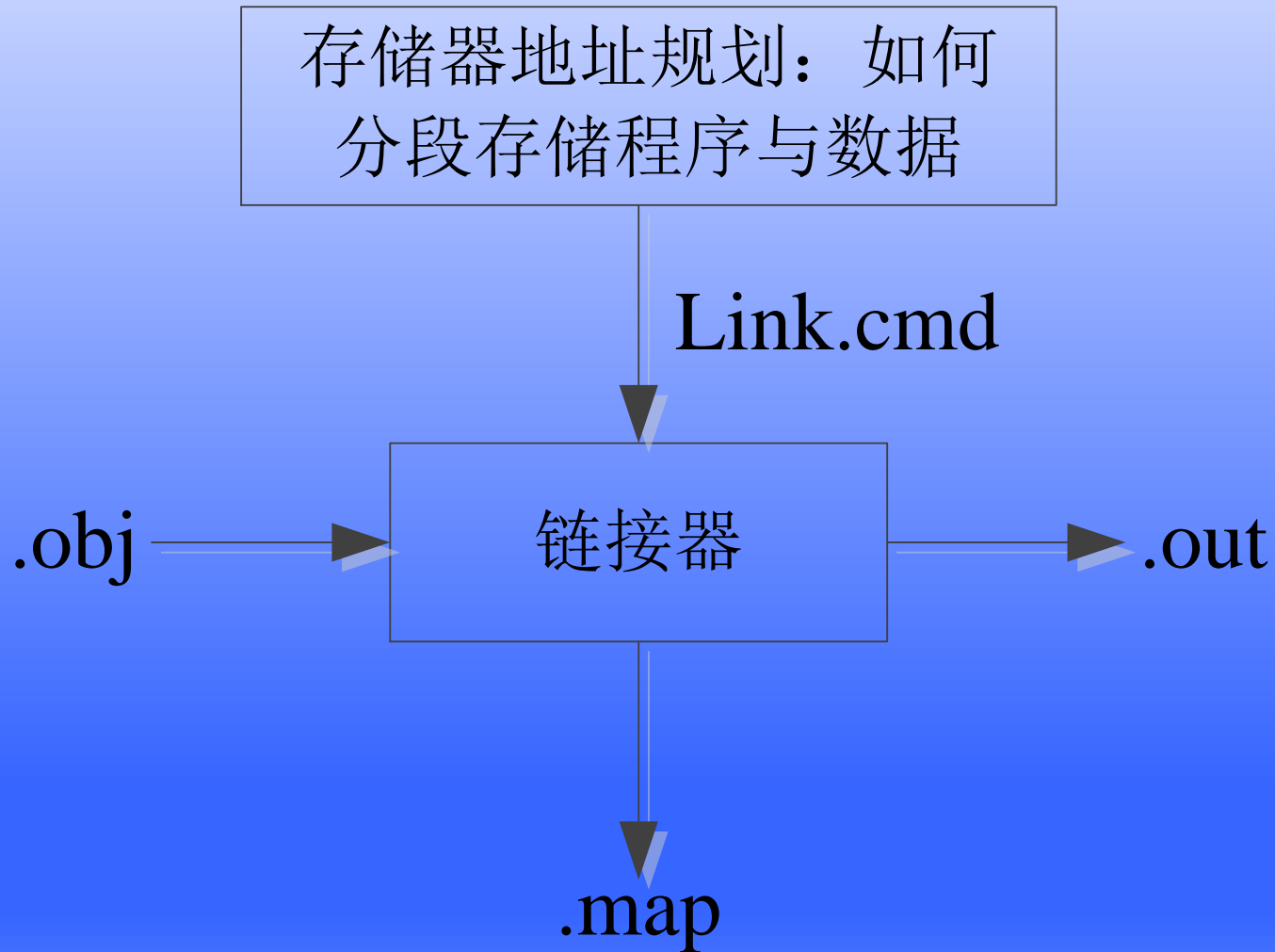
未初始化的段

名字	描述	链接位置
<code>.ebss</code>	全局与静态变量	RAM
<code>.stack</code>	堆栈空间	低64K字的RAM
<code>.esysmem</code>	Far malloc函数的存储空间	RAM

链接代码与存储单元



链接的示意图



cmd文件的格式

```
MEMORY
```

```
{
```

```
    PAGE 0:          /* Program Memory */
```

```
        FLASH:      origin = 0x300000, length = 0x40000
```

```
    PAGE 1:          /* Data Memory */
```

```
        M0SARAM:    origin = 0x000000, length = 0x400
```

```
        M1SARAM:    origin = 0x000400, length = 0x400
```

```
}
```

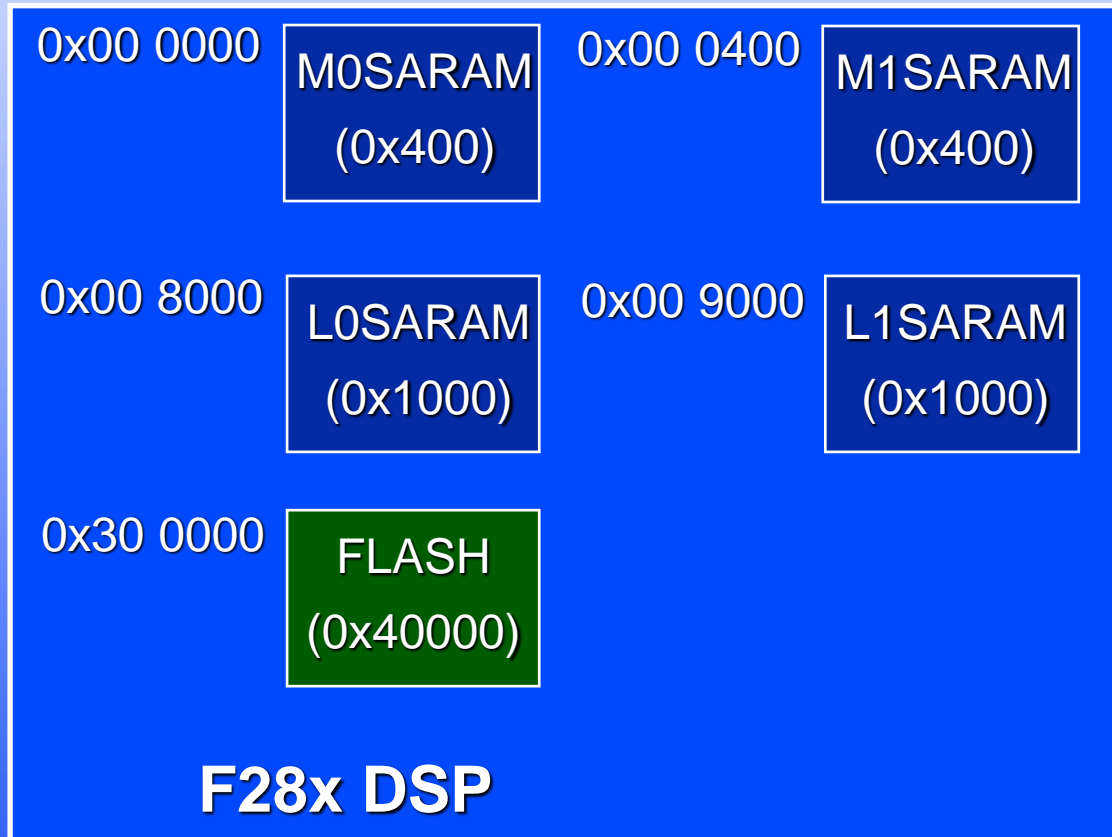
完整的cmd文件

```
MEMORY
{
    PAGE 0:          /* Program Memory */
    FLASH:          origin = 0x300000, length = 0x40000

    PAGE 1:          /* Data Memory */
    M0SARAM:        origin = 0x000000, length = 0x400
    M1SARAM:        origin = 0x000400, length = 0x400
}

SECTIONS
{
    .text:>          FLASH          PAGE = 0
    .ebss:>         M0SARAM        PAGE = 1
    .cinit:>        FLASH          PAGE = 0
    .stack:>        M1SARAM        PAGE = 1
}
```


练习1



把段分配到上面的RAM中

练习1

MEMORY

```
{  
    PAGE__ :          /* Program Memory */  
    _____ :    origin = _____,    length = _____  
    _____ :          /* Data Memory */  
    _____ :    origin = _____,    length = _____  
    _____ :    origin = _____,    length = _____  
    _____ :    origin = _____,    length = _____  
    _____ :    origin = _____,    length = _____  
}
```

SECTIONS

```
{  
    .text:    >    FLASH        PAGE = 0  
    .ebss:    >    M0SARAM      PAGE = 1  
    .cinit:   >    FLASH        PAGE = 0  
    .stack:   >    M1SARAM      PAGE = 1  
}
```

练习1-结果

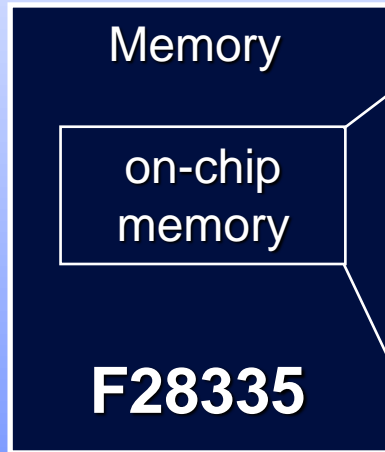
MEMORY

```
{  
    PAGE_0:          /* Program Memory */  
    FLASH:          origin = 0x300000,    length = 0x40000  
    PAGE_1:          /* Data Memory */  
    M0SARAM:        origin = 0x000000,    length = 0x400  
    M1SARAM:        origin = 0x000400,    length = 0x400  
    L0SARAM:        origin = 0x008000,    length = 0x1000  
    L1SARAM:        origin = 0x009000,    length = 0x1000  
}
```

SECTIONS

```
{  
    .text:          >    FLASH          PAGE = 0  
    .ebss:          >    M0SARAM        PAGE = 1  
    .cinit:         >    FLASH          PAGE = 0  
    .stack:         >    M1SARAM        PAGE = 1  
}
```

练习2 cmd文件的示例



0x00 0000	M0SARAM (0x400)	0x00 B000	L3SARAM (0x1000)
0x00 0400	M1SARAM (0x400)	0x00 C000	L4SARAM (0x1000)
0x00 8000	L0SARAM (0x1000)	0x00 D000	L5SARAM (0x1000)
0x00 9000	L1SARAM (0x1000)	0x00 E000	L6SARAM (0x1000)
0x00 A000	L2SARAM (0x1000)	0x00 F000	L7SARAM (0x1000)

•描述

•TMS320F28335

•分配所有RAM

段的分配方案:

- .text保存到RAM的L0123SARAM，位于PAGE 0 (程序空间)
- .cinit保存到RAM的L0123SARAM，位于PAGE 0 (程序空间)
- .ebss保存到RAM的L4SARAM，位于PAGE 1 (数据空间)
- .stack保存到RAM的M1SARAM，位于PAGE 1 (数据空间)

练习2 cmd分配的结果

MEMORY

```
{
    PAGE 0:          /* Program Memory */
    L0123SARAM:     origin = 0x008000, length = 0x4000
    PAGE 1:          /* Data Memory */
    M0SARAM:        origin = 0x000000, length = 0x0400
    M1SARAM:        origin = 0x000400, length = 0x0400
    L4SARAM:        origin = 0x00C000, length = 0x1000
    L5SARAM:        origin = 0x00D000, length = 0x1000
    L6SARAM:        origin = 0x00E000, length = 0x1000
    L7SARAM:        origin = 0x00F000, length = 0x1000
}
```

SECTIONS

```
{
    .text:          > L0123SARAM    PAGE = 0
    .ebss:          > L4SARAM      PAGE = 1
    .cinit:         > L0123SARAM    PAGE = 0
    .stack:         > M1SARAM      PAGE = 1
    .reset:         > L0123SARAM    PAGE = 0, TYPE = DSECT
}
```


传统C代码访问寄存器的方法

```
#define ADCTRL1      (volatile unsigned int *)0x00007100
#define ADCTRL2      (volatile unsigned int *)0x00007101
...

void main(void)
{
    *ADCTRL1 = 0x1234; //write entire register
    *ADCTRL2 |= 0x4000; //reset sequencer #1
}
```

0x1234=1001000110100

优点

- 简单，方便，易于输入
- 变量名与寄存器名一致，方便记忆

0x4000=1000000000000000

不足

- 无法直接操作单独的位
- 在调试时无法显示单独的位
- 效率不高

结构化的寄存器访问方法

```
void main(void)
{
    AdcRegs.ADCTRL1.all = 0x1234;        //write entire register
    AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;    //reset sequencer #1
}
```

优点

- 容易操作单独的位.
- 调试功能更强大
- 可产生更高效的代码

不足

- 不容易记忆
- 输入更多的字符

编译器的自动完成与提示功能可解决这些问题)

传统方法-在CCS的Watch窗口中观察寄存器

The screenshot displays the Code Composer Studio (CCS) interface for a TMS320C28xx DSP. The main window shows the source code for `Main.c`, with a menu open for selecting registers to watch. The menu path is: `Watch ADC Registers` > `All_ADC_Regs`. The `Watch Window` is open, showing a list of registers and their current values. The registers listed include `ADCTRL1`, `ADCTRL2`, `ADCMAXCONV`, `ADCCHSELSEQ1`, `ADCCHSELSEQ2`, `ADCCHSELSEQ3`, `ADCCHSELSEQ4`, `ADCASEQSR`, `ADCRESULT0`, `ADCRESULT1`, `ADCRESULT2`, `ADCRESULT3`, `ADCRESULT4`, `ADCRESULT5`, `ADCRESULT6`, `ADCRESULT7`, `ADCRESULT8`, `ADCRESULT9`, `ADCRESULT10`, `ADCRESULT11`, `ADCRESULT12`, `ADCRESULT13`, `ADCRESULT14`, `ADCRESULT15`, `ADCTRL3`, `ADCST`, `ADCREFSEL`, `ADCOFFTRIM`, `ADCRESULT0 Mirror`, `ADCRESULT1 Mirror`, `ADCRESULT2 Mirror`, `ADCRESULT3 Mirror`, `ADCRESULT4 Mirror`, `ADCRESULT5 Mirror`, `ADCRESULT6 Mirror`, `ADCRESULT7 Mirror`, and `ADCRESULT8 Mirror`. The values for all these registers are `0x0000`. The status bar at the bottom indicates `Build Complete, 0 Errors, 0 Warnings, 0 Remarks.` and the target is `HALTED`.

Watch Window

Name	Value	Type	Radix
ADCTRL1	0x0000	int	hex
ADCTRL2	0x0000	int	hex
ADCMAXCONV	0x0000	int	hex
ADCCHSELSEQ1	0x0000	int	hex
ADCCHSELSEQ2	0x0000	int	hex
ADCCHSELSEQ3	0x0000	int	hex
ADCCHSELSEQ4	0x0000	int	hex
ADCASEQSR	0x0000	int	hex
ADCRESULT0	0x0000	int	hex
ADCRESULT1	0x0000	int	hex
ADCRESULT2	0x0000	int	hex
ADCRESULT3	0x0000	int	hex
ADCRESULT4	0x0000	int	hex
ADCRESULT5	0x0000	int	hex
ADCRESULT6	0x0000	int	hex
ADCRESULT7	0x0000	int	hex
ADCRESULT8	0x0000	int	hex
ADCRESULT9	0x0000	int	hex
ADCRESULT10	0x0000	int	hex
ADCRESULT11	0x0000	int	hex
ADCRESULT12	0x0000	int	hex
ADCRESULT13	0x0000	int	hex
ADCRESULT14	0x0000	int	hex
ADCRESULT15	0x0000	int	hex
ADCTRL3	0x0000	int	hex
ADCST	0x0000	int	hex
ADCREFSEL	0x0000	int	hex
ADCOFFTRIM	0x0000	int	hex
ADCRESULT0 Mirror	0x0000	int	hex
ADCRESULT1 Mirror	0x0000	int	hex
ADCRESULT2 Mirror	0x0000	int	hex
ADCRESULT3 Mirror	0x0000	int	hex
ADCRESULT4 Mirror	0x0000	int	hex
ADCRESULT5 Mirror	0x0000	int	hex
ADCRESULT6 Mirror	0x0000	int	hex
ADCRESULT7 Mirror	0x0000	int	hex
ADCRESULT8 Mirror	0x0000	int	hex

使用结构化方法观察寄存器的每个位

The screenshot shows the Code Composer Studio interface for a TMS320C28xx processor. The main window displays the source code for `Main.c`, with the `main` function visible. The `main` function includes several initialization and enablement calls for the ADC, such as `ADCRegs.bit.rsvd1 = 0x0000;`, `ADCRegs.bit.SEQ_CASC = 0;`, `ADCRegs.bit.SEQ_OVRD = 0;`, `ADCRegs.bit.CONT_RUN = 0;`, `ADCRegs.bit.CPS = 0;`, `ADCRegs.bit.ACQ_PS = 0x0000;`, `ADCRegs.bit.SUSMOD = 0;`, and `ADCRegs.bit.RESET = 0;`. The `main` function ends with `return 0;`.

The Watch Window is open, displaying a list of variables and their values. The variables are organized into a tree structure under `ADCRegs`. The variables are:

Name	Value	Type	Radix
ADCRegs	{...}	stru...	hex
ADCTRL1	{...}	uni...	hex
all	0	Uin...	unsig
bit	{...}	stru...	hex
rsvd1	0000	(un...	bin
SEQ_CASC	0	(un...	bin
SEQ_OVRD	0	(un...	bin
CONT_RUN	0	(un...	bin
CPS	0	(un...	bin
ACQ_PS	0000	(un...	bin
SUSMOD	00	(un...	bin
RESET	0	(un...	bin
rsvd2	0	(un...	bin
ADCTRL2	{...}	uni...	hex
ADCMAXCONV	{...}	uni...	hex
ADCCHSELSEQ1	{...}	uni...	hex
ADCCHSELSEQ2	{...}	uni...	hex
ADCCHSELSEQ3	{...}	uni...	hex
ADCCHSELSEQ4	{...}	uni...	hex
ADCASEQSR	{...}	uni...	hex
ADCRESULT0	0	Uin...	unsig
ADCRESULT1	0	Uin...	unsig
ADCRESULT2	0	Uin...	unsig
ADCRESULT3	0	Uin...	unsig
ADCRESULT4	0	Uin...	unsig
ADCRESULT5	0	Uin...	unsig
ADCRESULT6	0	Uin...	unsig
ADCRESULT7	0	Uin...	unsig
ADCRESULT8	0	Uin...	unsig
ADCRESULT9	0	Uin...	unsig
ADCRESULT10	0	Uin...	unsig
ADCRESULT11	0	Uin...	unsig
ADCRESULT12	0	Uin...	unsig
ADCRESULT13	0	Uin...	unsig
ADCRESULT14	0	Uin...	unsig
ADCRESULT15	0	Uin...	unsig
ADCTRL3	{...}	uni...	hex

The Watch Window also shows a `Watch Locals` tab with `Watch 1` selected. The status bar at the bottom indicates `Ln 1, Col 1`.

结构化方法是否更高效？

结构化方法可使编译器更高效地寻址并使用原子指令

C源代码

```
// Stop CPU Timer0
CpuTimer0Regs.TCR.bit.TSS = 1;

// Load new 32-bit period value
CpuTimer0Regs.PRD.all = 0x00010000;

// Start CPU Timer0
CpuTimer0Regs.TCR.bit.TSS = 0;
```

编译之后的汇编代码

```
MOVW    DP, #0030
OR      @4, #0x0010

MOVL   XAR4, #0x010000
MOVL   @2, XAR4

AND    @4, #0xFFEF
```

- 代码易读
- 位操作方便

5 words, 5个指令周期

传统方法编程的对比

传统方法需要大量的指针操作与随机存储器访问，也无法更高效地使用原子指令

C源代码

```
// Stop CPU Timer0
*TIMER0TCR |= 0x0010;

// Load new 32-bit period value
*TIMER0TPRD32 = 0x00010000;

// Start CPU Timer0
*TIMER0TCR &= 0xFFEF;
```

0x0010=10000

0xFFEF=1111111111101111

- 不直观
- 对位的操作较为麻烦

编译之后的汇编代码

```
MOV    @AL, *(0:0x0C04)
ORB    AL, #0x10
MOV    *(0:0x0C04), @AL

MOVL   XAR5, #0x010000
MOVL   XAR4, #0x000C0A
MOVL   *+XAR4[0], XAR5

MOV    @AL, *(0:0x0C04)
AND    @AL, #0xFFEF
MOV    *(0:0x0C04), @AL
```

9 words, 9个指令周期

结构化的命名规范

- ◆ DSP2833x头文件中定义了:
 - ◆ 所有的外设结构体
 - ◆ 所有的寄存器名称
 - ◆ 所有寄存器的位的含义
 - ◆ 所有的寄存器的地址

<code>PeripheralName.RegisterName.all</code>	<code>// Access full 16 or 32-bit register</code>
<code>PeripheralName.RegisterName.half.LSW</code>	<code>// Access low 16-bits of 32-bit register</code>
<code>PeripheralName.RegisterName.half.MSW</code>	<code>// Access high 16-bits of 32-bit register</code>
<code>PeripheralName.RegisterName.bit.FieldName</code>	<code>// Access specified bit fields of register</code>

注解: [1] “PeripheralName” 外设名称(例如CpuTimer0Regs).

[2] “RegisterName” 寄存器名称(例如TCR, TIM, PRD,...).

[3] “FieldName” 位的名称(i.e. POL, TOG, TSS,...).

DSP2833x的外设与头文件包

(<http://www.ti.com>, 搜索编号SPRC530的文件包)

- ◆ 包含所有使用结构化编程方法所需的头文件和示例
- ◆ 定义了外设寄存器、位的信息和地址
- ◆ 内容包括

- ◆ \DSP2833x_headers\include → .h files
- ◆ \DSP2833x_headers\cmd → linker .cmd files
- ◆ \DSP2833x_headers\gel → .gel files for CCS
- ◆ \DSP2833x_examples → '2833x examples
- ◆ \DSP2823x_examples → '2823x examples
- ◆ \doc → documentation

外设结构的.h文件 (1)

- ◆ 包含了每个外设寄存器的位的信息。以ADC为例：

你的C代码(例如Adc.c)

```
#include "DSP2833x_Device.h"

Void InitAdc(void)
{
    /* Reset the ADC module */
    AdcRegs.ADCTRL1.bit.RESET = 1;

    /* configure the ADC register */
    AdcRegs.ADCTRL1.all = 0x0710;
};
```

DSP2833x Adc.h

```
/* ADC Individual Register Bit Definitions */
struct ADCTRL1_BITS { // bits description
    Uint16  rsvd1:4;    // 3:0 reserved
    Uint16  SEQ_CASC:1; // 4 Cascaded sequencer mode
    Uint16  SEQ_OVRD:1; // 5 Sequencer override
    Uint16  CONT_RUN:1; // 6 Continuous run
    Uint16  CPS:1;     // 7 ADC core clock prescaler
    Uint16  ACQ_PS:4;  // 11:8 Acquisition window size
    Uint16  SUSMOD:2;  // 13:12 Emulation suspend mode
    Uint16  RESET:1;   // 14 ADC reset
    Uint16  rsvd2:1;   // 15 reserved
};

/* Allow access to the bit fields or entire register */
union ADCTRL1_REG {
    Uint16  all;
    struct ADCTRL1_BITS bit;
};

// ADC External References & Function Declarations:
extern volatile struct ADC_REGS AdcRegs;
```

外设结构的.h文件 (2)

- ◆ 每个外设都对应一个头文件

DSP2833x_Device.h	DSP2833x_DevEmu.h	DSP2833x_SysCtrl.h
DSP2833x_PieCtrl.h	DSP2833x_Adc.h	DSP2833x_CpuTimers.h
DSP2833x_ECan.h	DSP2833x_ECap.h	DSP2833x_EPwm.h
DSP2833x_EQep.h	DSP2833x_Gpio.h	DSP2833x_I2c.h
DSP2833x_Sci.h	DSP2833x_Spi.h	DSP2833x_XIntrupt.h
DSP2833x_PieVect.h	DSP2833x_DefaultIsr.h	DSP2833x_DMA.h
DSP2833x_Mcbsp.h	DSP2833x_Xintf.h	

- ◆ ***DSP2833x_Device.h***

- ◆ 主头文件(for '2833x and '2823x devices)
- ◆ 用来引用其它所有的头文件
- ◆ 使用方法:

```
#include "DSP2833x_Device.h"
```


全局变量定义文件

DSP2833x_GlobalVariableDefs.c

- ◆ 为每个外设建立了一个全局变量以方便使用
- ◆ 每个外设的结构体都使用**DATA_SECTION**指令来链接到**cmd**文件中的特定段

DSP2833x_GlobalVariableDefs.c

```
#include "DSP2833x_Device.h"  
...  
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");  
volatile struct ADC_REGS AdcRegs;  
...
```

- ◆ 把此文件添加到自己的工程中:

DSP2833x_GlobalVariableDefs.c

全局变量定义文件

DSP2833x_GlobalVariableDefs.c

```
DSP2833x_GlobalVariableDefs.c
1 // TI File $Revision: /main/2 $
2 // Checkin $Date: March 5, 2007 12:20:33 $
3 //#####
4 //
5 // FILE:    DSP2833x_GlobalVariableDefs.c
6 //
7 // TITLE:   DSP2833x Global Variables and Data Section Pragmas.
8 //
9 //#####
10 // $TI Release: DSP2833x Header Files V1.01 $
11 // $Release Date: September 26, 2007 $
12 //#####
13
14 #include "DSP2833x_Device.h"    // DSP2833x Headerfile Include File
15
16 //-----
17 // Define Global Peripheral Variables:
18 //
19 //-----
20 #ifdef __cplusplus
21 #pragma DATA_SECTION("AdcRegsFile")
22 #else
23 #pragma DATA_SECTION(AdcRegs, "AdcRegsFile");
24 #endif
25 volatile struct ADC_REGS AdcRegs;
26
27 //-----
28 #ifdef __cplusplus
29 #pragma DATA_SECTION("AdcMirrorFile")
30 #else
31 #pragma DATA_SECTION(AdcMirror, "AdcMirrorFile");
32 #endif
33 volatile struct ADC_RESULT_MIRROR_REGS AdcMirror;
34
35 //-----
36 #ifdef __cplusplus
37 #pragma DATA_SECTION("CpuTimer0RegsFile")
38 #else
39 #pragma DATA_SECTION(CpuTimer0Regs, "CpuTimer0RegsFile");
40 #endif
41 volatile struct CPUTIMER_REGS CpuTimer0Regs;
42
43 //-----
44 #ifdef colusolus
```

外设结构体与cmd文件的对应

DSP2833x_nonBIOS.cmd

DSP2833x_GlobalVariableDefs.c

```
#include "DSP2833x_Device.h"  
...  
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");  
volatile struct ADC_REGS AdcRegs;  
...
```

DSP2833x-Headers_nonBIOS.cmd

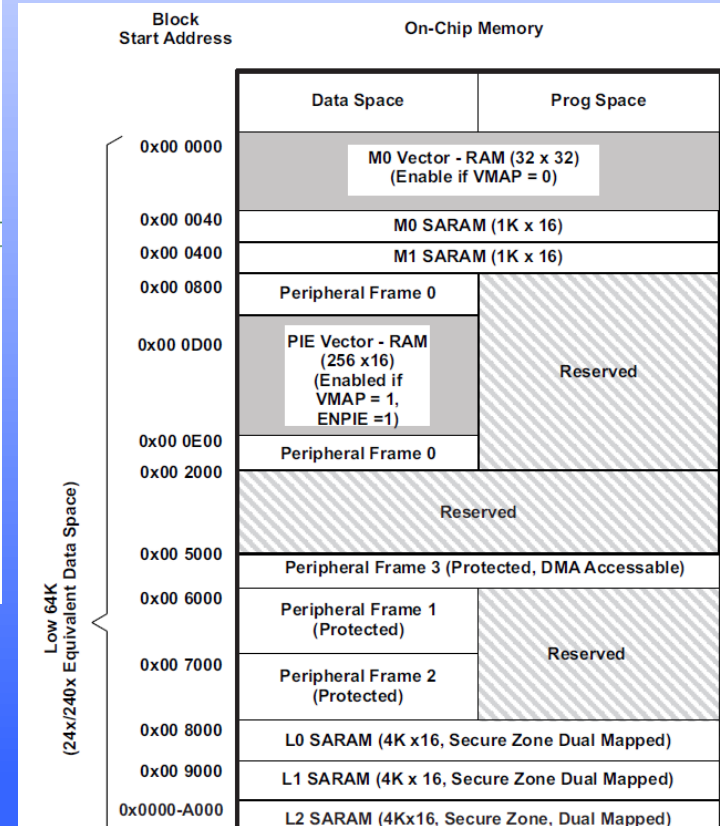
```
MEMORY  
{  
  PAGE1:  
    ADC:  origin=0x007100, length=0x000020  
    ...  
}  
SECTIONS  
{  
  AdcRegsFile:  >  ADC          PAGE = 1  
  ...  
}
```

外设结构体与cmd文件的对应

DSP2833x_nonBIOS.cmd

```
25 MEMORY
26 {
27   PAGE 0:      /* Program Memory */
28
29   PAGE 1:      /* Data Memory */
30
31   DEV_EMU      : origin = 0x000880, length = 0x000180 /* device emulation registers */
32   FLASH_REGS   : origin = 0x000A80, length = 0x000060 /* FLASH registers */
33   CSM          : origin = 0x000AE0, length = 0x000010 /* code security module registers */
34
35   ADC_MIRROR   : origin = 0x000B00, length = 0x000010 /* ADC Results register mirror */
36
37   XINTF        : origin = 0x000B20, length = 0x000020 /* external interface registers */
38
39   CPU_TIMER0   : origin = 0x000C00, length = 0x000008 /* CPU Timer0 registers */
40   CPU_TIMER1   : origin = 0x000C08, length = 0x000008 /* CPU Timer0 registers (CPU Timer1 & Timer2) */
41   CPU_TIMER2   : origin = 0x000C10, length = 0x000008 /* CPU Timer0 registers (CPU Timer1 & Timer2) */
42
43   PIE_CTRL     : origin = 0x000CE0, length = 0x000020 /* PIE control registers */
44   PIE_VECT     : origin = 0x000D00, length = 0x000100 /* PIE Vector Table */
45
46   DMA          : origin = 0x001000, length = 0x000200 /* DMA Rev 0 registers */
47
48   MCBSPA       : origin = 0x005000, length = 0x000040 /* McBSP-A registers */
49   MCBSPB       : origin = 0x005040, length = 0x000040 /* McBSP-B registers */
50
51   ECANA        : origin = 0x006000, length = 0x000040 /* eCAN-A control and status registers */
52   ECANA_LAM    : origin = 0x006040, length = 0x000040 /* eCAN-A local acceptance masks */
53   ECANA_MOTS   : origin = 0x006080, length = 0x000040 /* eCAN-A message object time stamps */
54   ECANA_MOTO   : origin = 0x0060C0, length = 0x000040 /* eCAN-A object time-out registers */
55   ECANA_MBOX   : origin = 0x006100, length = 0x000100 /* eCAN-A mailboxes */
56
```

```
100
101 SECTIONS
102 {
103   PieVectTableFile : > PIE_VECT, PAGE = 1
104
105   /** Peripheral Frame 0 Register Structures **/
106   DevEmuRegsFile   : > DEV_EMU, PAGE = 1
107   FlashRegsFile    : > FLASH_REGS, PAGE = 1
108   CsmRegsFile      : > CSM, PAGE = 1
109   AdcMirrorFile    : > ADC_MIRROR, PAGE = 1
110   XintfRegsFile    : > XINTF, PAGE = 1
111   CpuTimer0RegsFile : > CPU_TIMER0, PAGE = 1
112   CpuTimer1RegsFile : > CPU_TIMER1, PAGE = 1
113   CpuTimer2RegsFile : > CPU_TIMER2, PAGE = 1
114   PieCtrlRegsFile  : > PIE_CTRL, PAGE = 1
115   DmaRegsFile      : > DMA, PAGE = 1
116
117   /** Peripheral Frame 3 Register Structures **/
118   McbspaRegsFile   : > MCBSPA, PAGE = 1
119   McbspbRegsFile   : > MCBSPB, PAGE = 1
120
```



教材25页

每个外设都有专门的示例

