前言

MSP430 是德州仪器（TI）一款性能卓越的超低功耗 16 位单片机，自问世以来，MSP430 单片机一直是业内公认的功耗最低的单片机。除采用先进的制造工艺使芯片的静态电流尽可能降低外，MSP430 的独立可配置的时钟系统是其低功耗的基石之一。在追求绿色能源的今天，MSP430 超低功耗微控制器正以其超低功耗的特性，以及丰富多样化的外设受到越来越多设计者们的青睐。

MSP430 发展到今天已经行成了非常丰富的产品体系：从最初通用型的 F1 和 F2 系列，到集成有段式 LCD 驱动的 F4 系列（比较广泛地应用于水电表中），到集成有 USB 驱动的 F5/F6 系列，到集成有 1GHz 射频模块的 CC430 系列，再到近期的超高性价比的 G2，再到采用新存储技术的 FRAM 系列。MSP430 产品已经被广泛地应用到工业生活的各个领域，从水电表到烟雾探测，从电动牙刷到便携式血糖仪，从遥控器到平板触摸家电，430 正潜移默化地改善用户体验，使得生活更加安全与简单。

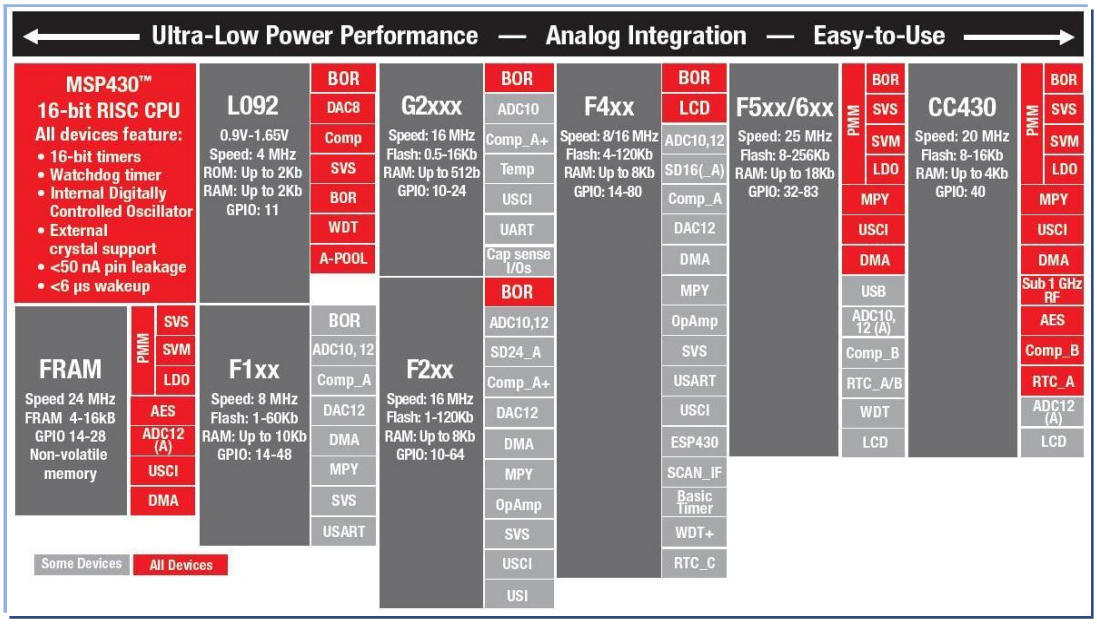


图 MSP430 产品家族

同时，MSP430 有着开发简单，容易上手的优势，这为新手进行单片机学习提供了很大的便利于帮助。为方便使用者的学习，TI 推出了多款基于 430 的开发学习板，供初学者学习以及项目的开发，例如 MSP430Launchpad，以及基于 F5529 的开发板等。TI 也为 MSP430 学习者和开发者提供了丰富的资源，包括硬件原理设计，软件开发，应用笔记等。

德州仪器中国大学计划也一直致力于将业内先进的技术引入到中国高校中，让学生能够学以致用。在 MSP430 的推广上更是不遗余力，在硬件开发平台和软件技术支持上都全力帮助老师和同学。在与高校老师和同学交流的过程中发现很多是刚开始接触单片机或者

430，对 430 的开发流程不是很清楚，在过程中往往会产生很过困惑。鉴于此，我们希望能够通过这本小册子帮助刚接触 MSP430 的开发者，解答在开发过程中的一些问题，使其能够尽快开始 430 的开发工作。本书一共包括 6 个小节，从实践的角度分别讲解了：

- CCS5.1 的安装和基本使用方法；
- 430Ware 软件的介绍和使用说明；
- Grace 软件的介绍和使用说明；
- CCS 软件开发流程；
- CCS 工程结构解析；
- TI 官方典型例程解析。

通过概念结合实例的方法希望能够帮助读者尽快地开始 MSP430 的开发。附录中通过一个开发案例更为直接地向读者介绍如何利用 CCS 以及其他资源进行 MSP430 的开发。在常见问题中列出了学校同学在 430 学习过程中可能会碰到的问题。本书由高校和 TI 大学计划部的工程师通力合作来完成，其中第一章节 CCS 的安装和使用方法由合肥工业大学电气与自动化工程学院 DSP 实验室的老师和同学编写，第三章节 Grace 软件的介绍和使用说明由西安电子科技大学 MSP430 单片机联合实验室的老师和同学编写，其余章节则由德州仪器大学计划部 MCU 工程师崔萌，王沁编写完成。在此对合肥工业大学和西安电子科技大学的老师和同学表示感谢。

由于时间仓促，本书中不免有错误和漏洞存在，希望大家能积极反馈，在使用过程中帮忙查漏补缺，完善该书，以期更好地帮助初学者，谢谢。

德州仪器中国大学计划

2012 年 10 月

MSP430 软件开发指南

目录

1. 软件开发环境 CCSV5.1

CCS（Code Composer Studio）是 TI 公司研发的一款具有环境配置、源文件编辑、程序调试、跟踪和分析等功能的集成开发环境，能够帮助用户在一个软件环境下完成编辑、编译、链接、调试和数据分析等工作。CCSv5.1 为 CCS 软件的最新版本，功能更强大、性能更稳定、可用性更高，是 MSP430 软件开发的理想工具。在本节中将具体介绍 CCSV5 的安装方法，并以 MSP430F5529 为例介绍 CCS 的一般使用方法。

1.1. CCSv5.1 的安装

（1）运行下载的安装程序 ccs_setup_5.1.1.00031.exe，当运行到如图 1.1 处时，选择 Custom 选项，进入手动选择安装通道。

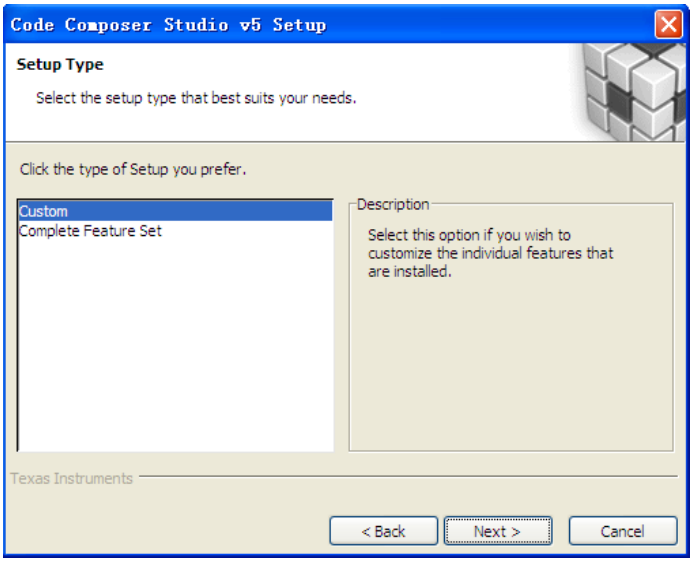


图 1.1 安装过程 1

（2）单击 Next 得到如图 1.2 所示的窗口，为了安装快捷，在此只选择支持 MSP430 Low Power MCUs 的选项。单击 Next，保持默认配置，继续安装。

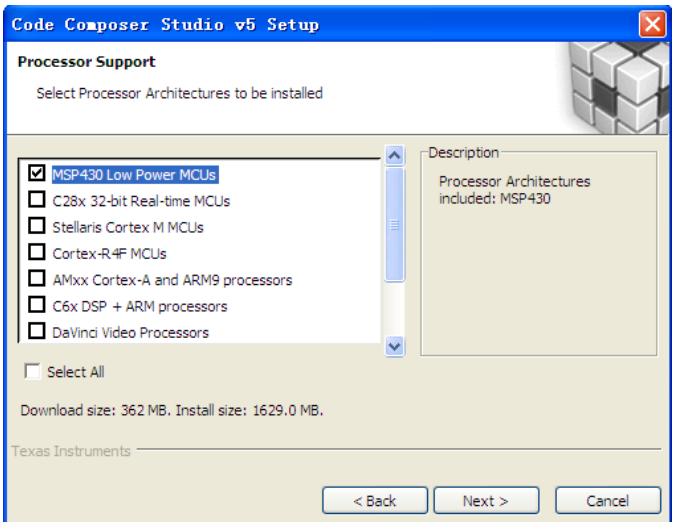


图 1.2 安装过程 2

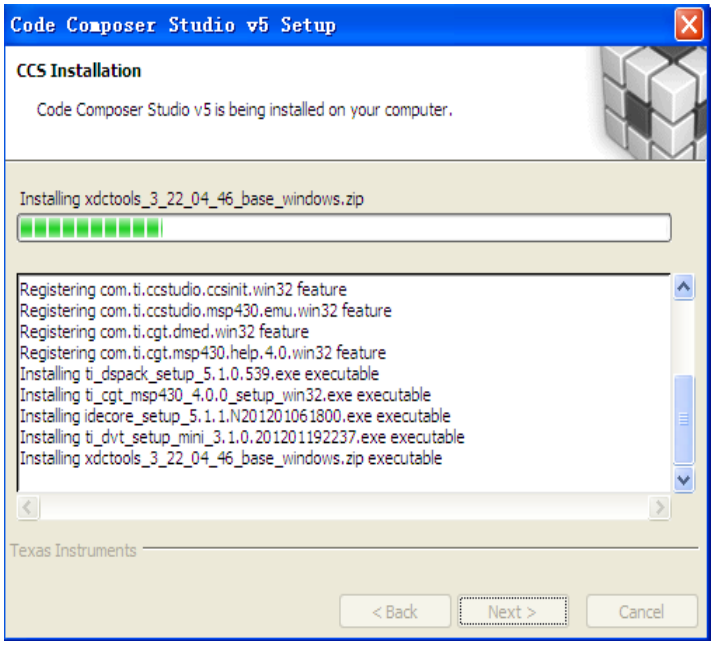


图 1.3 软件安装中

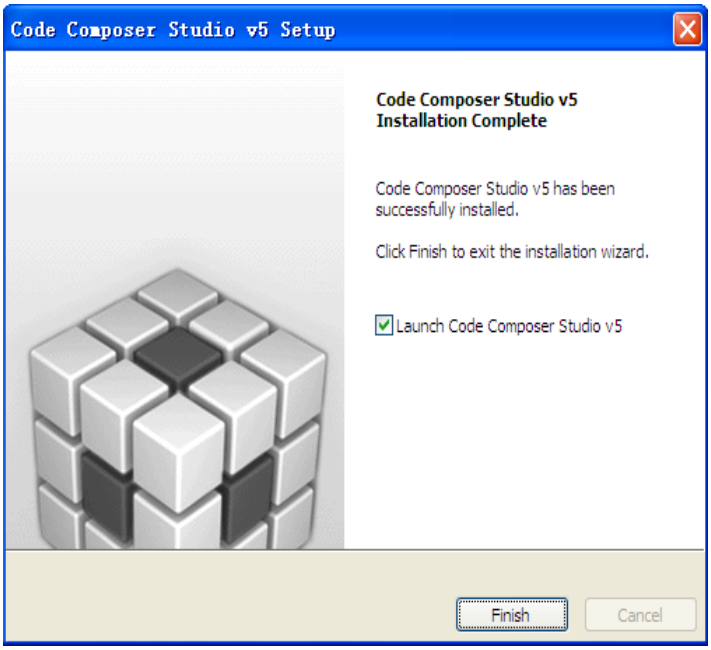


图 1.4 软件安装完成

（3）单击 Finish，将运行 CCS，弹出如图 1.5 所示窗口，打开“我的电脑”，在某一磁盘下，创建以下文件夹路径：-\MSP-EXP430F5529\Workspace，单击 Browse，将工作区间链接到所建文件夹，不勾选"Use this as the default and do not ask again"。

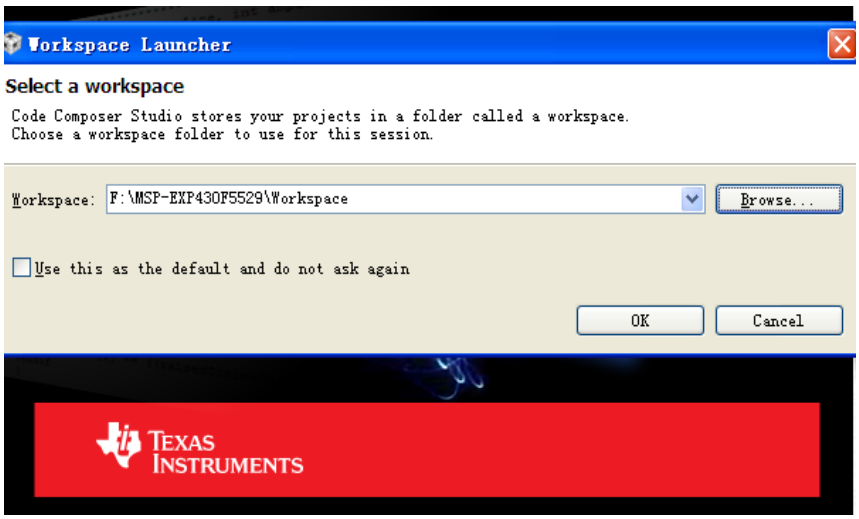


图 1.5 Workspace 选择窗口

（4）单击 OK，第一次运行 CCS 需进行软件许可的选择，如图 1.6 所示。

在此，选择 CODE SIZE LIMITED(MSP430)选项，在该选项下，对于 MSP430，CCS 免费开放 16KB 的程序空间；若您有软件许可，可以参考以下链接进行软件许可的认证：

http://processors.wiki.ti.com/index.php/GSG:CCSv5_Running_for_the_first_time，单击 Finish 即可进入 CCSv5.1 软件开发集成环境，如图 1.7 所示。

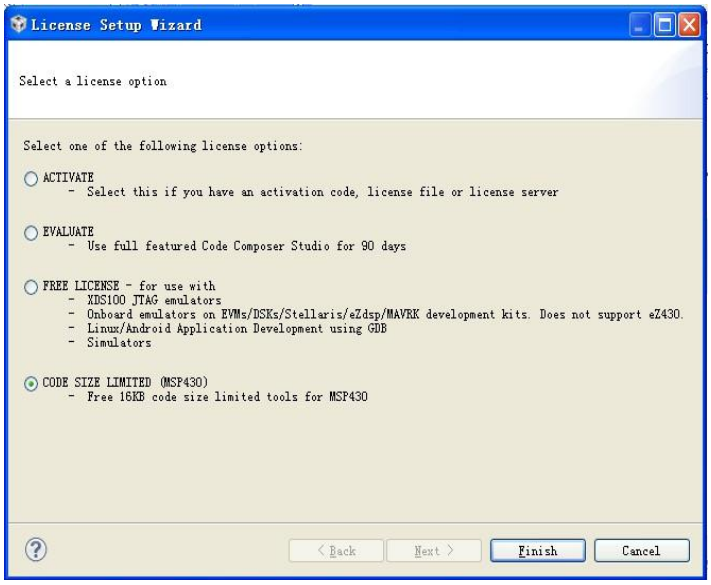


图 1.6 软件许可选择窗口

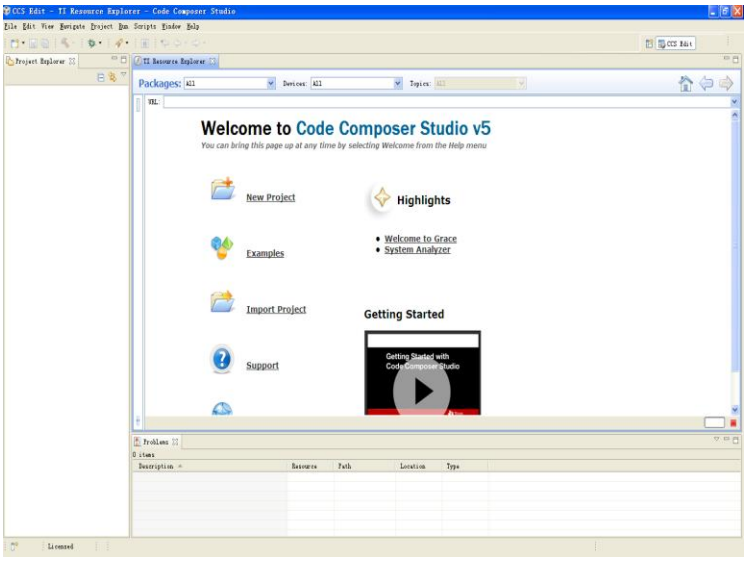


图 1.7 CCSv5 软件开发集成环境界面

1.2. 利用 CCSv5.1 导入已有工程

（1）在此以实验一的工程为例进行讲解，首先打开 CCSv5.1 并确定工作区间：F\MSP-EXP430F5529\Workspace，选择 File-->Import 弹出图 1.8 对话框，展开 Code Composer Studio 选择 Existing CCS/CCE Eclipse Projects。

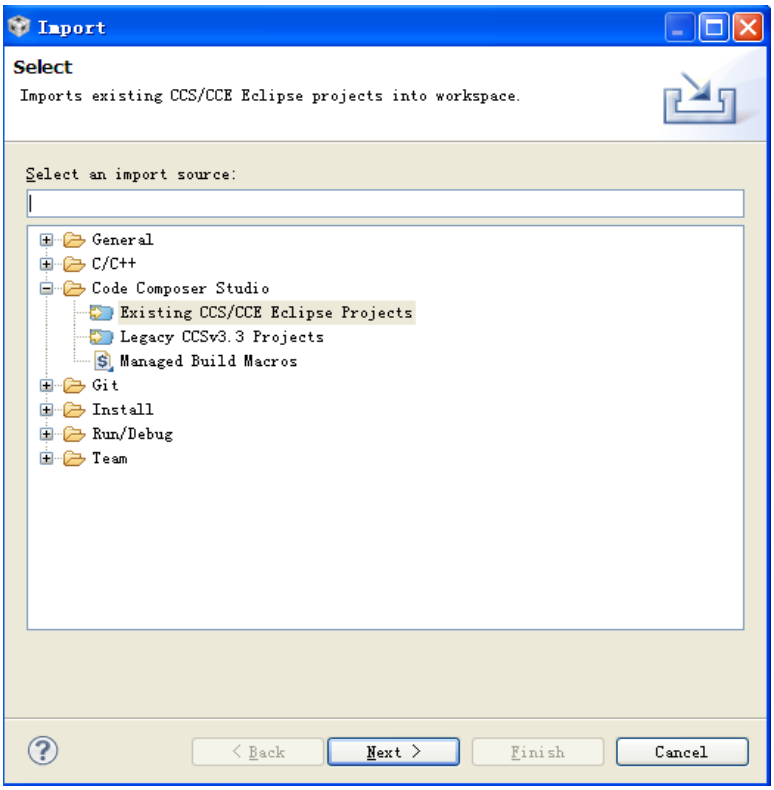


图 1.8 导入新的 CCSv5 工程文件

（2）单击 Next 得到图 1.9 对话框。

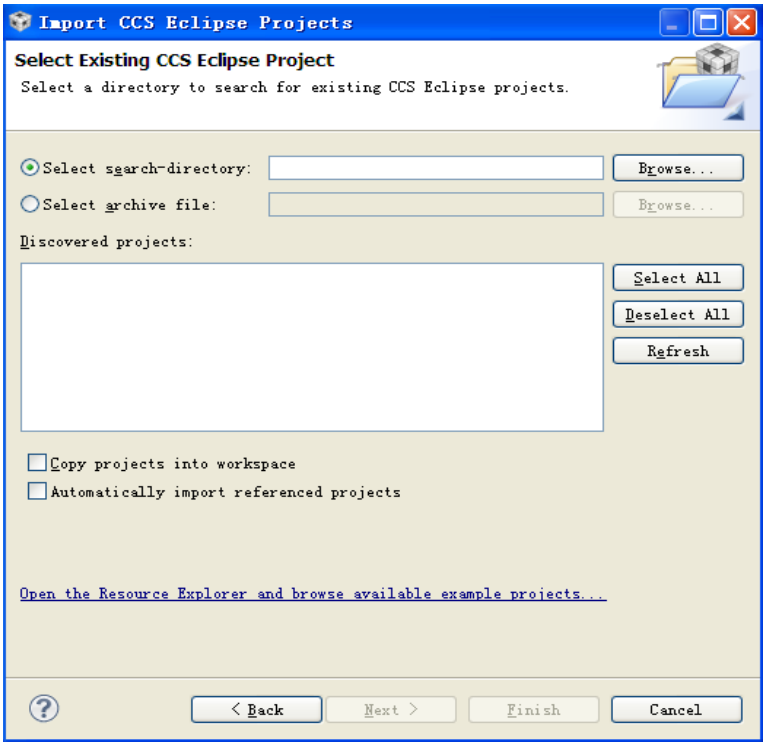


图 1.9 选择导入工程目录

（3）单击 Browse 选择需导入的工程所在目录，在此，我们选择：F\MSP-EXP430F5529\

Workspace\MSP-EXP430F5529 LAB CODE\LAB1（需在此之前，将实验代码复制到工作区间下），得到图 1.10。

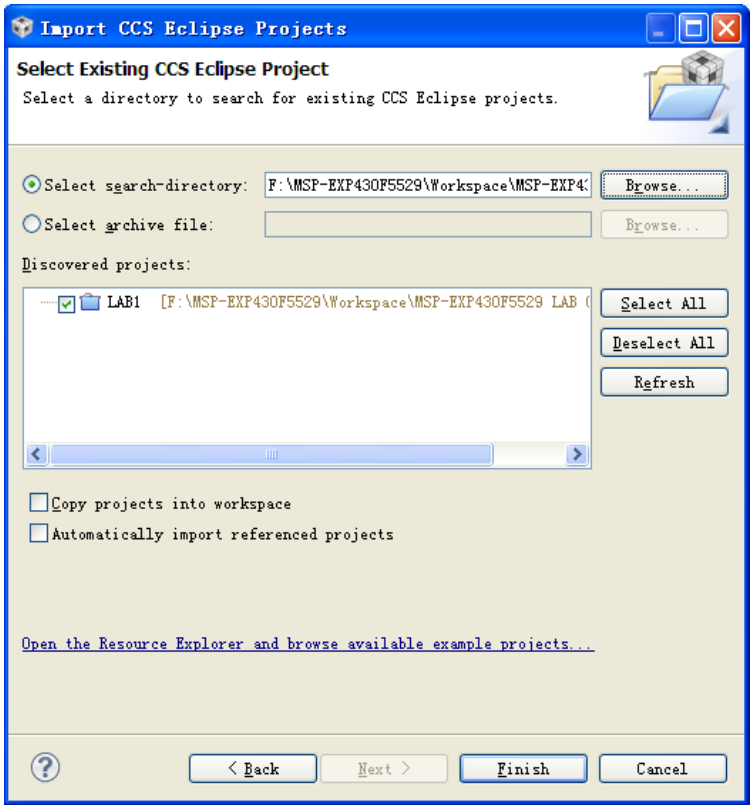


图 1.10 选择导入工程

（4）单击 Finish，即可完成既有工程的导入。

1.3. 利用 CCSv5.1 新建工程

（1）首先打开 CCSv5.1 并确定工作区间，然后选择 File-->New-->CCS Project 弹出图 1.11 对话框。

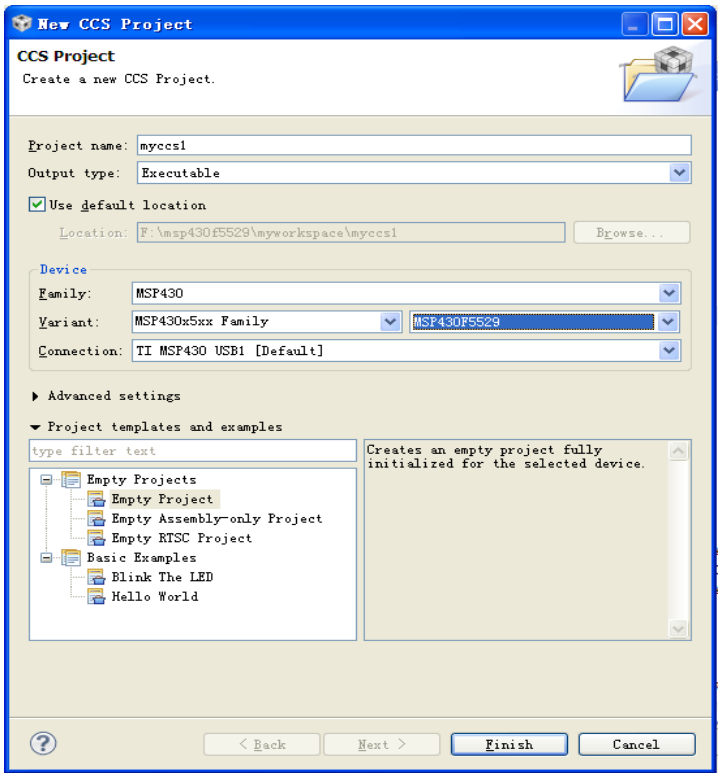


图 1.11 新建 CCS 工程对话框

（2）在 Project name 中输入新建工程的名称，在此输入 myccs1。

（3）在 Output type 中有两个选项：Executable 和 Static library，前者为构建一个完整的可执行程序，后者为静态库。在此保留：Executable。

（4）在 Device 部分选择器件的型号:在此 Family 选择 MSP430；Variant 选择 MSP430X5XX family，芯片选择 MSP430F5529；Connection 保持默认。

（5）选择空工程，然后单击 Finish 完成新工程的创建。

（6）创建的工程将显示在 Project Explorer 中，如图 1.12 所示。

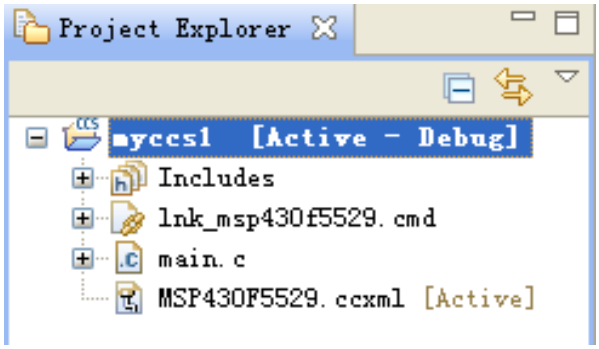


图 1.12 初步创建的新工程

特别提示：若要新建或导入已有.h 或.c 文件，步骤如下：

（7）新建.h 文件：在工程名上右键点击，选择 New-->Header File 得到图 1.13 对话框。

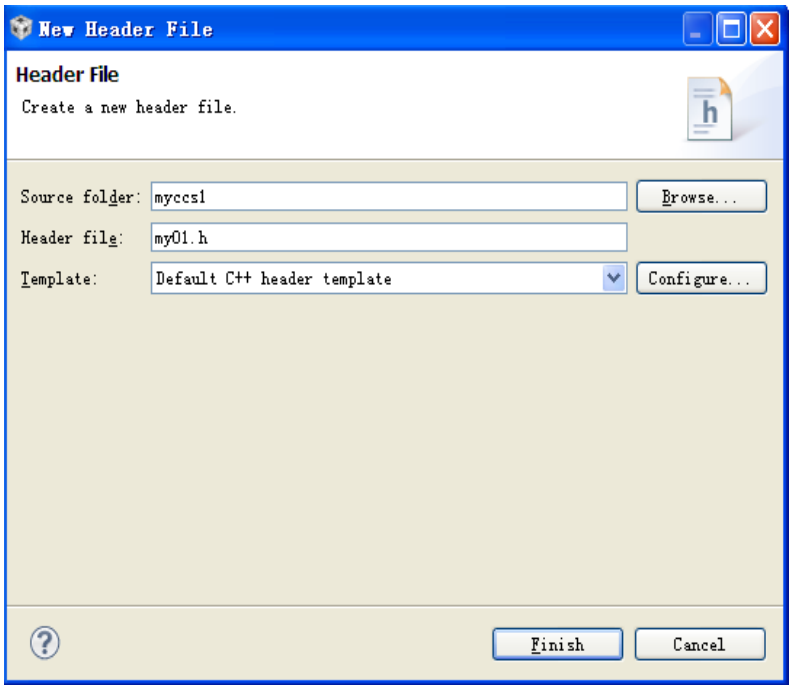


图 1.13 新建.h 文件对话框

在 Header file 中输入头文件的名称，注意必须以.h 结尾，在此输入 myo1.h。

（8）新建.c 文件：在工程名上右键单击，选择 New-->source file 得到图 1.14 对话框。

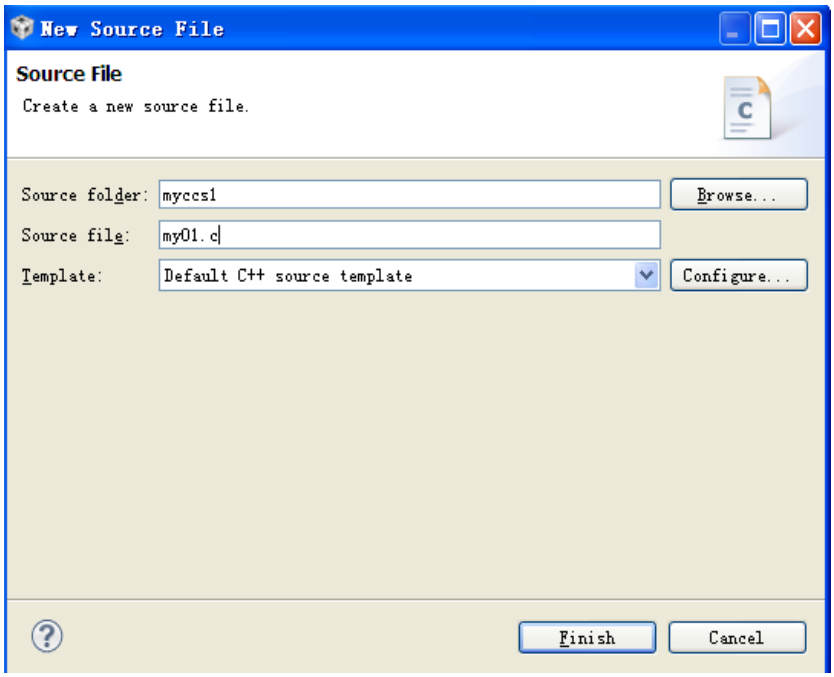


图 1.14 新建.c 文件对话框

在 Source file 中输入 c 文件的名称，注意必须以.c 结尾，在此输入 my01.c。

（9）导入已有.h 或.c 文件：在工程名上右键单击，选择 Add Files 得到如 1.15 对话框。

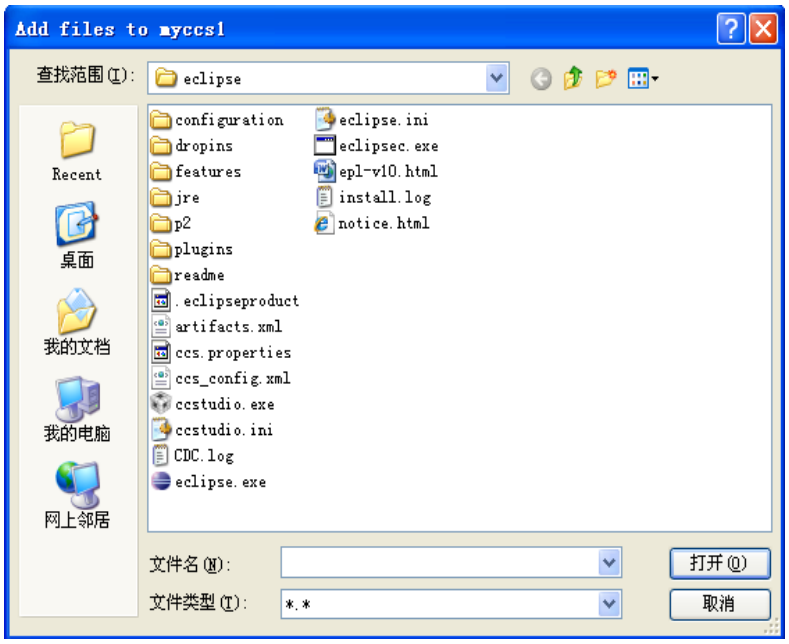


图 1.15 导入已有文件对话框

找到所需导入的文件位置，单击打开，得到图 1.16 对话框。

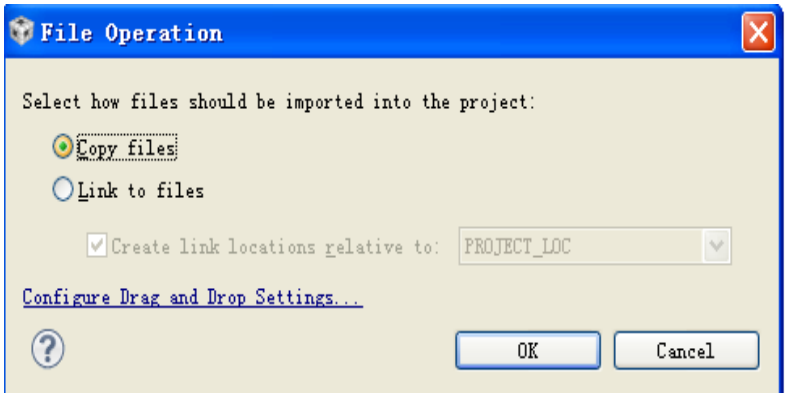


图 1.16 添加或连接现有文件

选择 Copy files，单击 OK，即可将已有文件导入到工程中。

若已用其它编程软件（例如 IAR），完成了整个工程的开发，该工程无法直接移植入 CCSv5，但可以通过在 CCSv5 中新建工程，并根据步骤（7）、（8）和（9）新建或导入已有.h 和.c 文件，从而完成整个工程的移植。

1.4. 利用 CCSv5.1 调试工程

1.4.1. 创建目标配置文件

（1）在开始调试之前，有必要确认目标配置文件是否已经创建并配置正确。在此以实验一为例进行讲解：首先导入实验一的工程，导入步骤请参考 1.2 节，如图 1.17 所示，其中 MSP430F5529.ccxml 目标配置文件已经正确创建，即可以进行编译调试，无需重新创建；若目标配置文件未创建或创建错误，则需进行创建。为了讲解目标配置文件创建过程，在此对 LAB1 的工程再次创建目标配置文件。

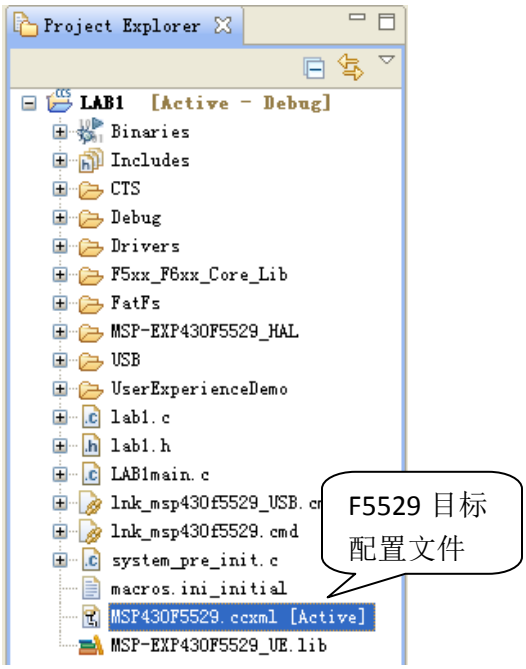


图 1.17 LAB1 工程浏览器

（2）创建目标配置文件步骤如下：右键单击项目名称，并选择 NEW --> Target Configuration File。

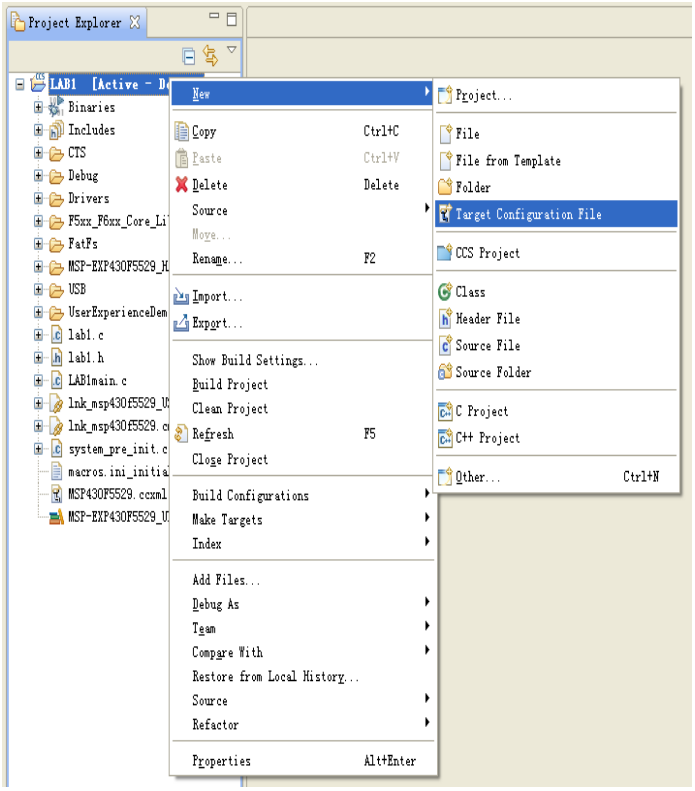


图 1.18 创建新的目标

（3）在 File name 中键入后缀为.ccxml 的配置文件名，由于创建 F5529 开发板的目标配置文件，因此，将配置文件命名为 MSP-EXP430F5529.ccxml，如图 1.19 所示。

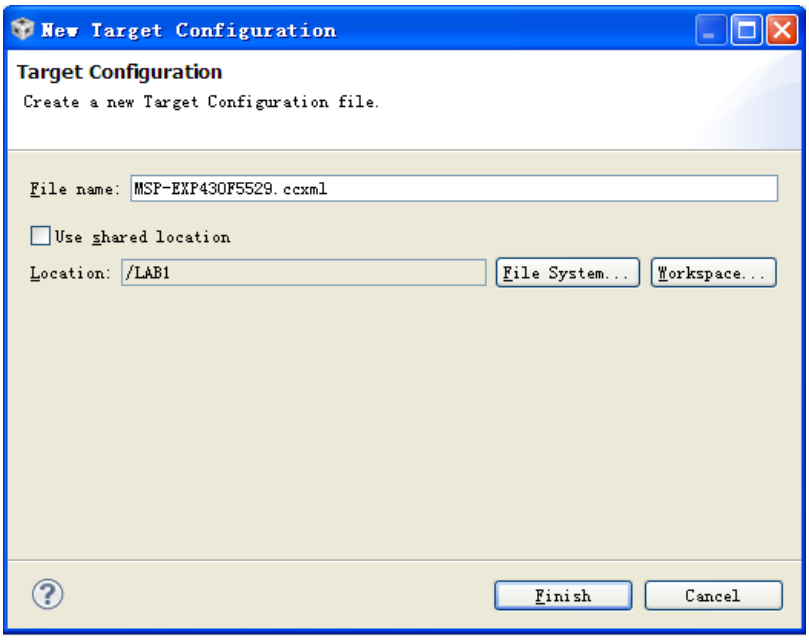


图 1.19 目标配置文件名

（4）单击 Finish，将打开目标配置编辑器，如图 1.20 所示。

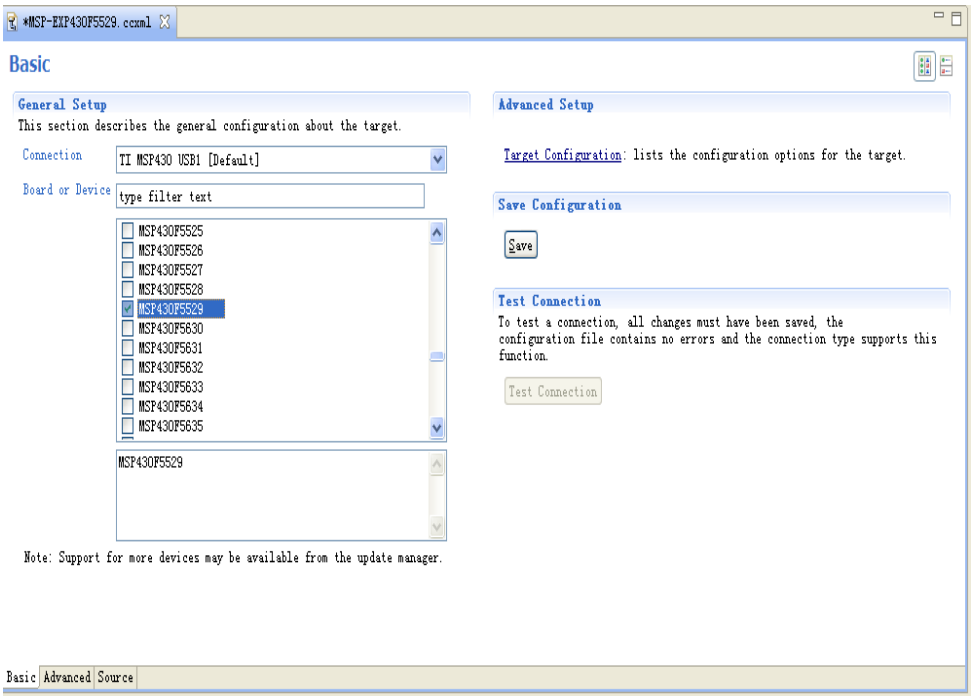


图 1.20 目标配置编辑器

（5）将 Connection 选项保持默认：TI MSP430 USB1 (Default)，在 Board or Device 菜单中选择单片机型号，在此选择 MSP430F5529。配置完成之后，单击 Save，配置将自动设为活动模式。如图 1.21 所示， 一个项目可以有多个目标配置，但只有一个目标配置在活动模式。要查看系统上所有现有目标配置，只需要去 View --> Target Configurations 查看。

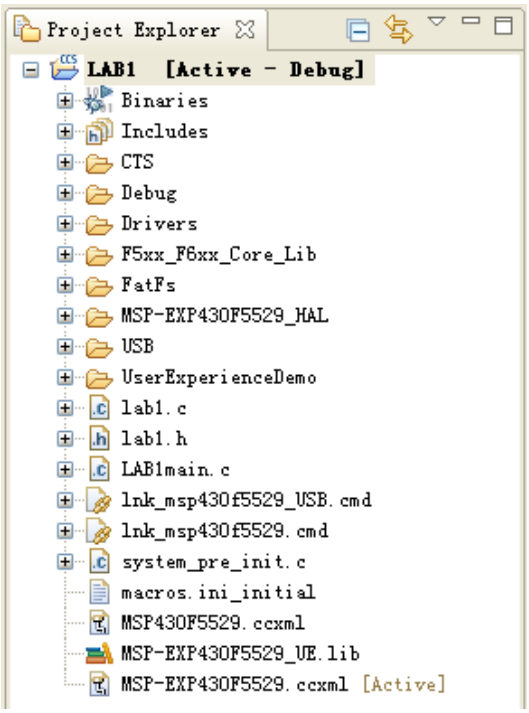


图 1.21 项目与配置后的目标文件

1.4.2. 启动调试器

（1）首先将 LAB1 工程进行编译通过：选择 Project-->Build Project，编译目标工程。在第一次编译实验工程时，系统会提示自动创建 rts430xl.lib 库文件，您可以选择等待创建完成，但可能会花费较长的时间。或者，为了方便，推荐在编译之前将本实验文件夹内的 rts430xl.lib 库文件复制到 CCSV5.1 的库资源文件夹内，其复制路径为：----\tools\compiler\msp430\lib(----为 CCSv5.1 的安装路径)。

编译结果，如图 1.22 所示，表示编译没有错误产生，可以进行下载调试；如果程序有错误，将会在 Problems 窗口显示，根据显示的错误修改程序，并重新编译，直到无错误提示。

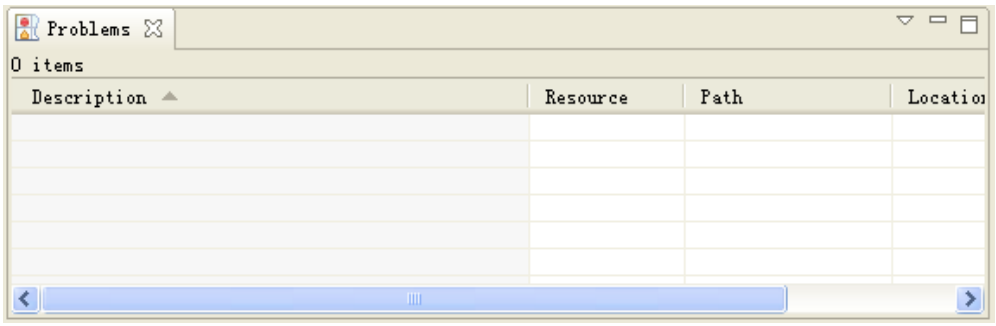


图 1.22 LAB1 工程调试结果

（2）单击绿色的 Debug 按钮 进行下载调试，得到图 1.23 所示的界面。

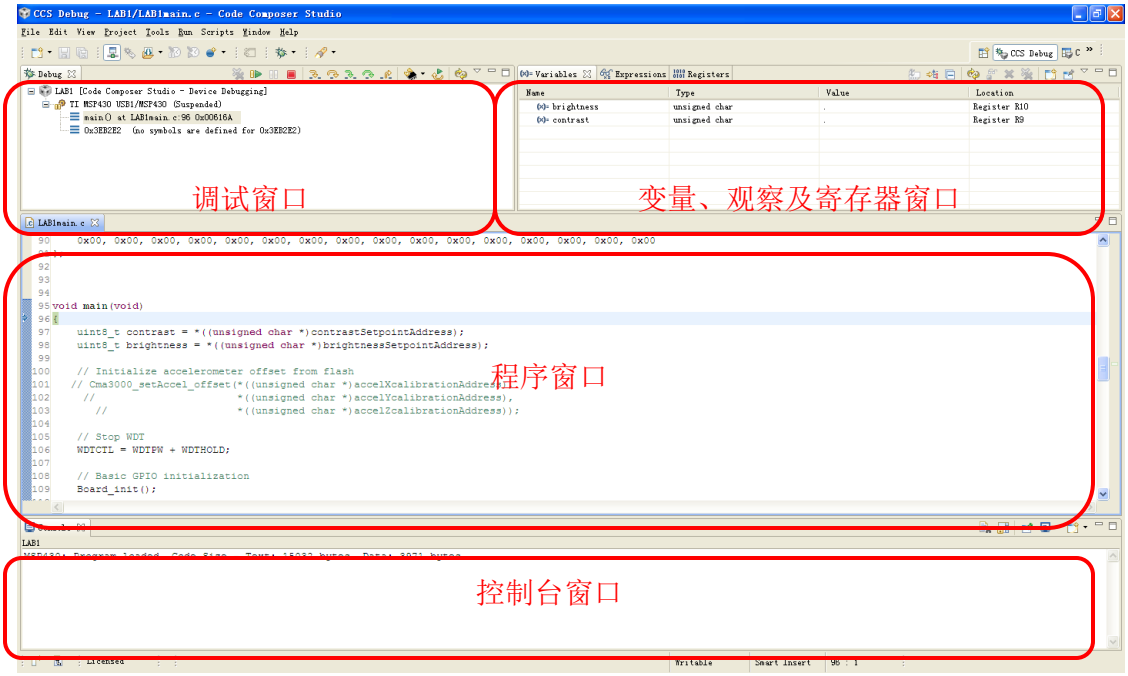


图 1.23 调试窗口界面

（3）单击运行图标 运行程序，观察显示的结果。在程序调试的过程中，可通过设置断点来调试程序：选择需要设置断点的位置，右击鼠标选择 Breakpoints→Breakpoint，断点设置成功后将显示图标，可以通过双击该图标来取消该断点。程序运行的过程中可以通过单步调试按钮 配合断点单步的调试程序，单击重新开始图标 定位到 main()函数，单击复位按钮 复位。可通过中止按钮 返回到编辑界面。

（4）在程序调试的过程中，可以通过 CCSV5.1 查看变量、寄存器、汇编程序或者是 Memory 等的信息 显示出程序运行的结果，以和预期的结果进行比较，从而顺利地调试程序。单击菜单 View→Variables，可以查看到变量的值，如图 1.24 所示。

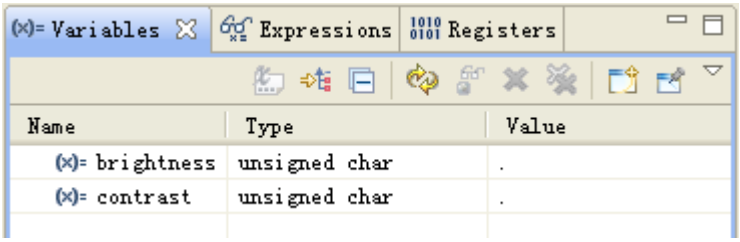


图 1.24 变量查看窗口

（5）点击菜单 View→Registers，可以查看到寄存器的值，如图 1.25 所示。

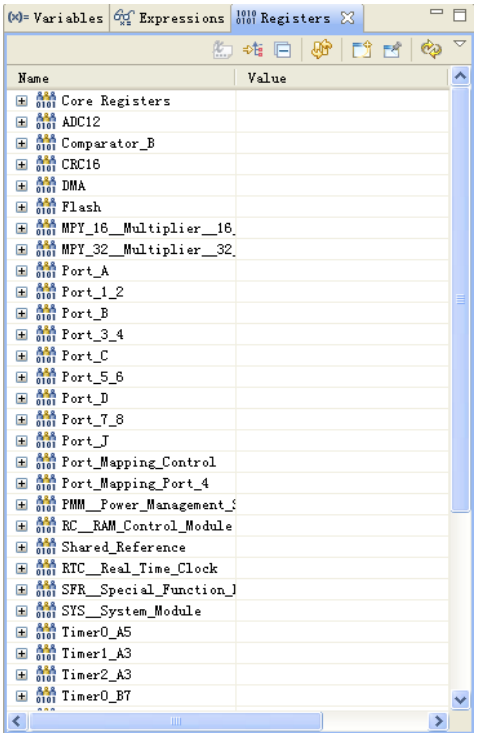


图 1.25 寄存器查看窗口

（6）点击菜单 View→Expressions，可以得到观察窗口，如图 1.26 所示。可以通过 Add new 添加观察变量，或者在所需观察的变量上右击，选择 Add Watch Expression 添加到观察窗口。

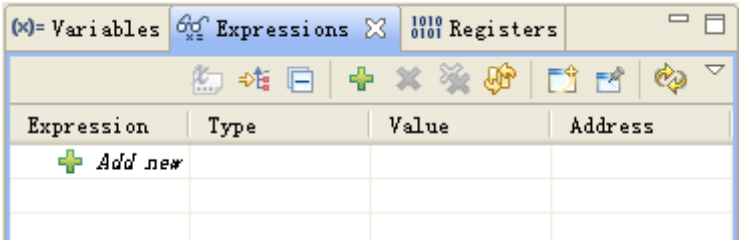


图 1.26 观察窗口

（7）点击菜单 View→Disassembly，可以得到汇编程序观察窗口，如图 1.27 所示。

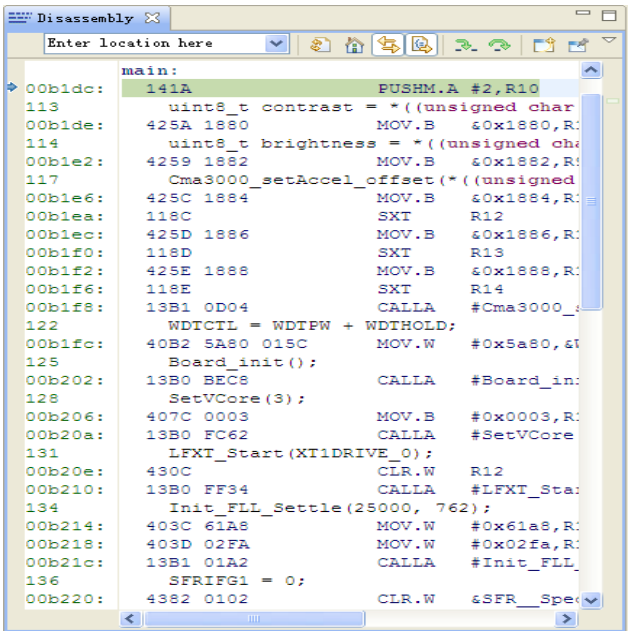


图 1.27 汇编程序观察窗口

（8）点击菜单 View→Memory Browser，可以得到内存查看窗口，如图 1.28 所示。

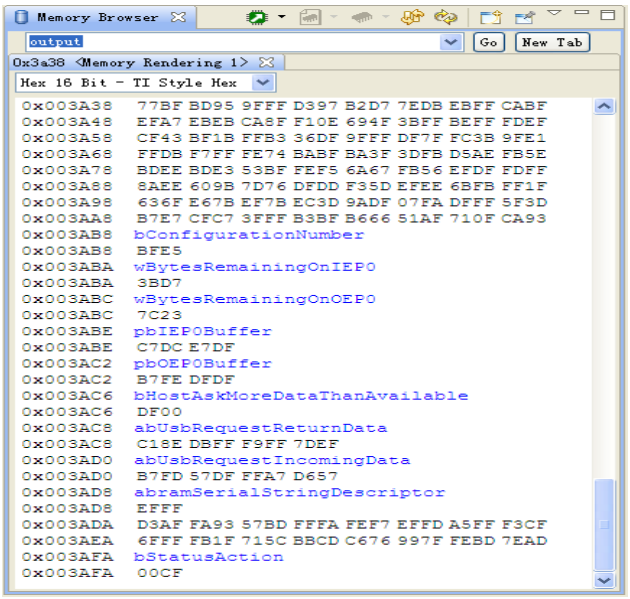


图 1.28 内存查看窗口

（9）点击菜单 View→Break points，可以得到断点查看窗口，如图 1.29 所示。

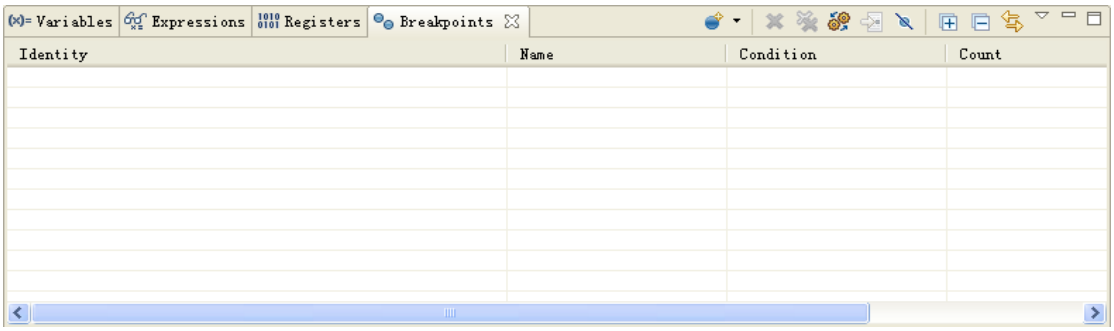

Variables
Expressions
Registers
Breakpoints
Identity
Name
Condition
Count

图 1.29 断点查看窗口

2. 430Ware 使用指南

430Ware 是 CCS 中的一个附带应用软件，在安装 CCSV5 的时候可选择同时安装 430Ware，在 TI 官网上也提供单独的 430Ware 安装程序下载。（http://www.ti.com/tool/msp430ware）

在 430Ware 中可以容易地找到 MSP430 所有系列型号的 Datasheet，User's guide 以及参考例程，此外 430Ware 还提供了大多数 TI 开发板（持续更新中）的用户指南，硬件设计文档以及参考例程。针对 F5 和 F6 系列还提供了驱动库文件，以方便用户进行上层软件的开发。

2.1.430Ware 使用说明

在 CCS 中单击“view”->“TI Resource Explorer”，在主窗口体会显示如图所示的界面。其中，Package 右侧的下拉窗口中可以观察目前 CCS 中安装的所有附加软件。在 package 旁的下拉菜单中选择 MSP430Ware，进入 430Ware 的界面。

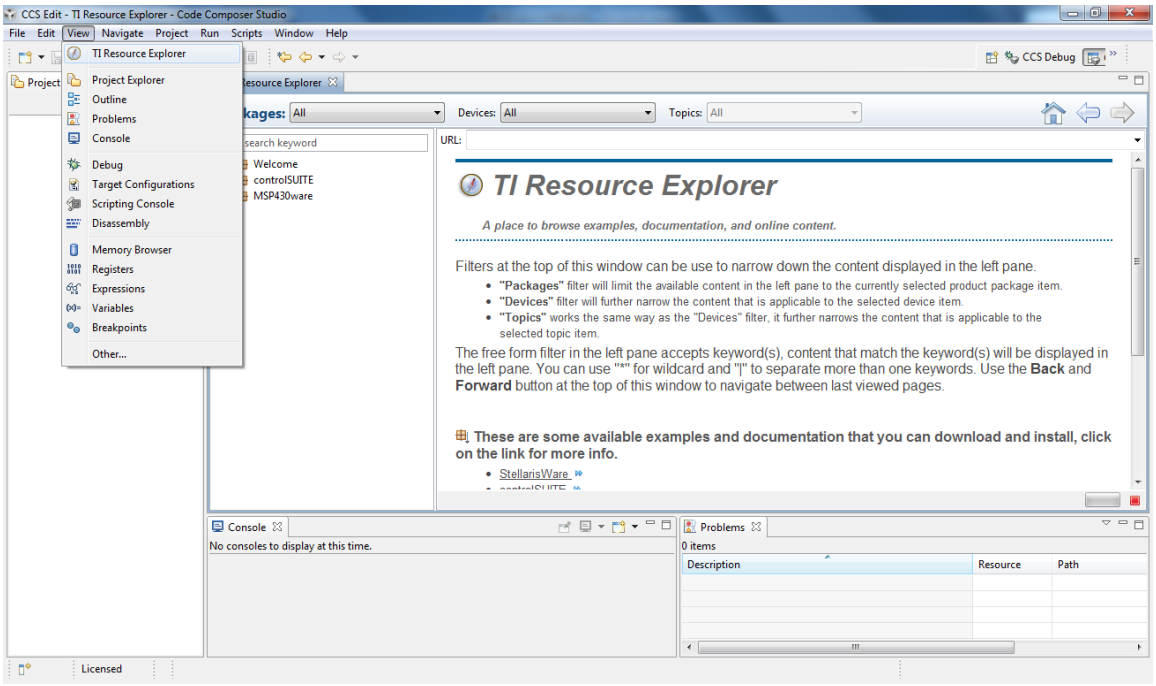


图 2.1 TI resource explorer 界面

在 430Ware 的界面左侧可以看到 3 个子菜单，分别是 Device，里面包含 MSP430 所有的系列型号；Development Tools，里面包括 TIMSP430 较新的一些开发套件的资料；和 Libaries，包含了可用于 F5 和 F6 系列的驱动库函数以及 USB 的驱动函数。

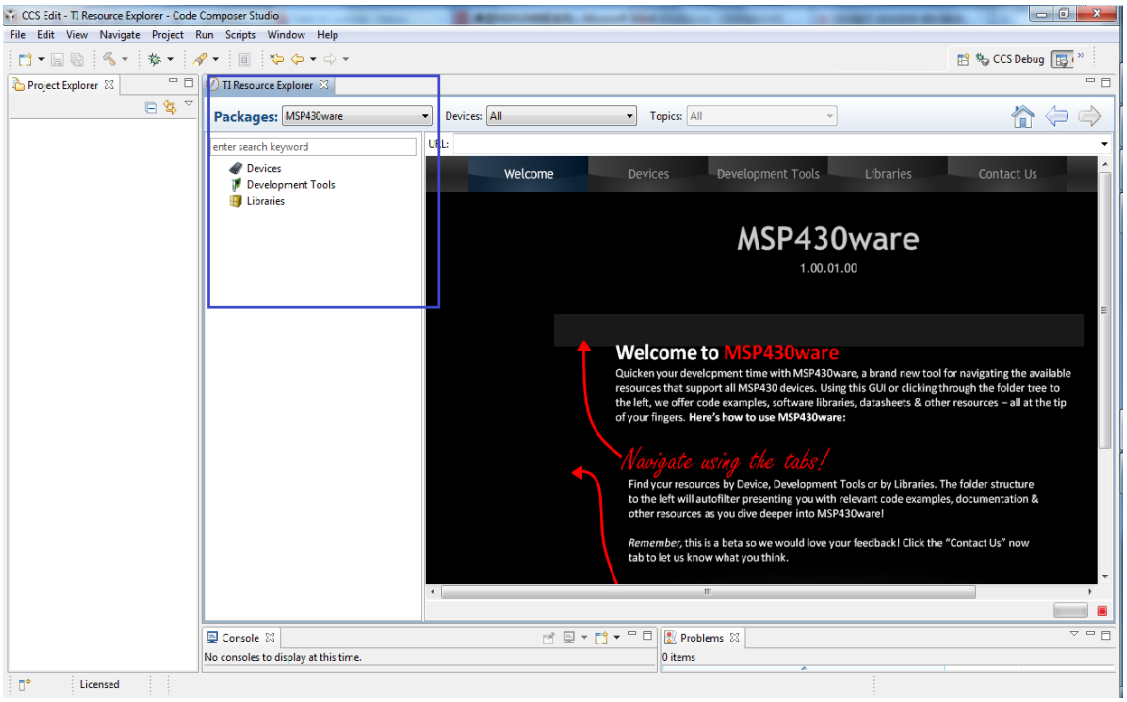


图 2.2 TI MSP430Ware 界面

单击菜单前的三角下拉键，查看下级菜单，可以看到在 Devices 的子目录下有目前所有的 MSP430 的型号，找到正在使用的型号，例如 MSP430G2xx，同样单击文字前的三角下拉键，在子目录可以找到该系列的 User's Guide，在用户指南中有对该系列 MSP430 的 CPU 以及外围模块，包括寄存器配置，工作模式的详细介绍和使用说明；同时可以找到的是该系列的 Datasheet，数据手册是与具体的型号相关，所以在 datasheet 的子目录中会看到具体不同型号的数据手册；最后在这里还可以找到的是参考代码。

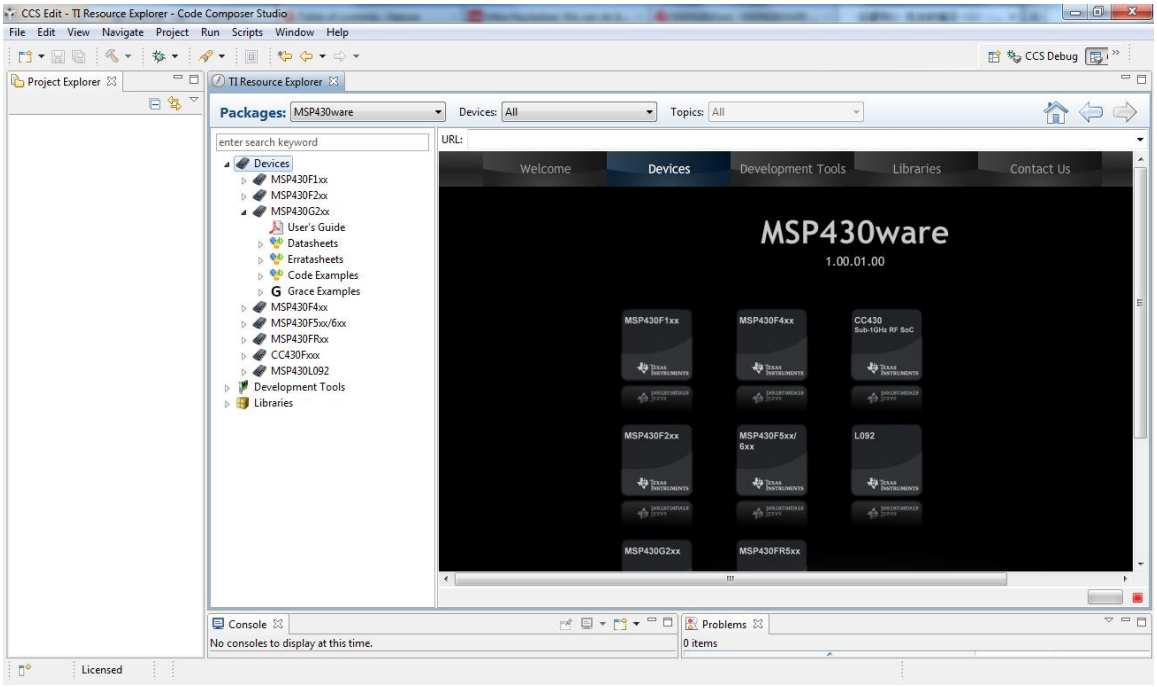


图 2.3 MSP430Ware 覆盖 430 全系列产品

在 430Ware 中提供不同型号的 CCS 示例程序，以及基于 Grace 的示例程序供开发者参考。选择具体型号后，在右侧窗口中看到提供到的参考示例程序。为更好地帮助用户了解 MSP430 的外设，430Ware 中提供了基于所有外设的参考例程，从示例程序的名字中可以看出该示例程序涉及的外设，同时在窗口中还可以看到关于该例程的简单描述，帮助用户更快地找到最合适的参考程序。

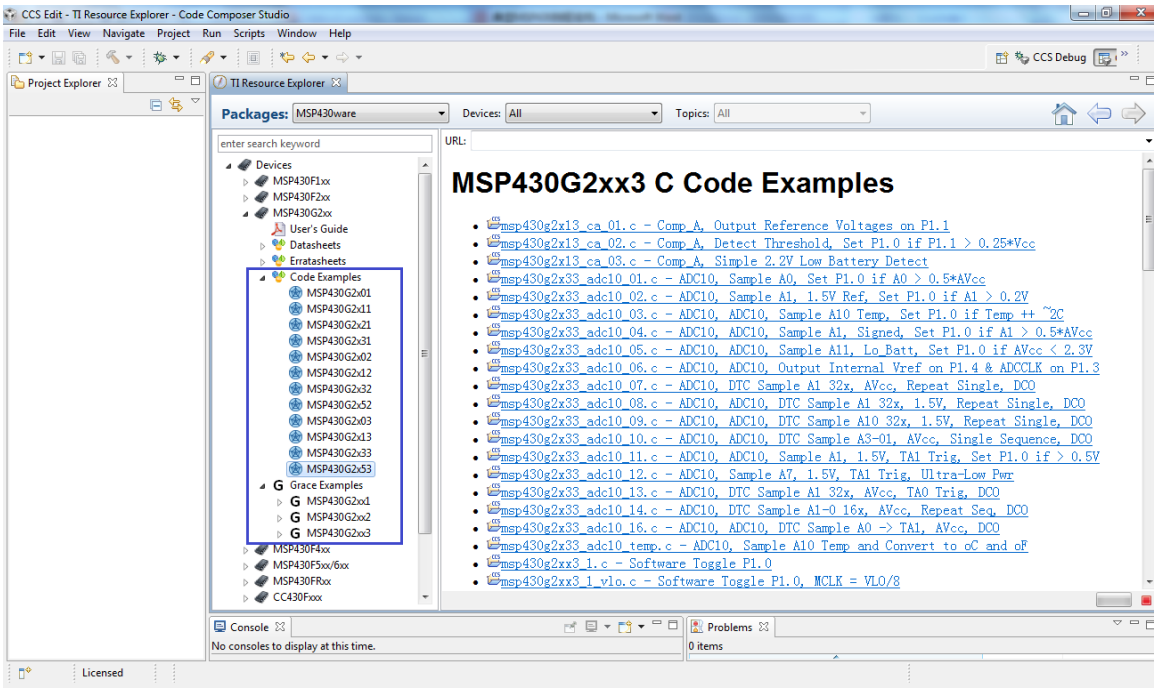


图 2.4 430Ware 中的示例程序

单击选中的参考例程，在弹出的对话框中选择连接的目标芯片型号。

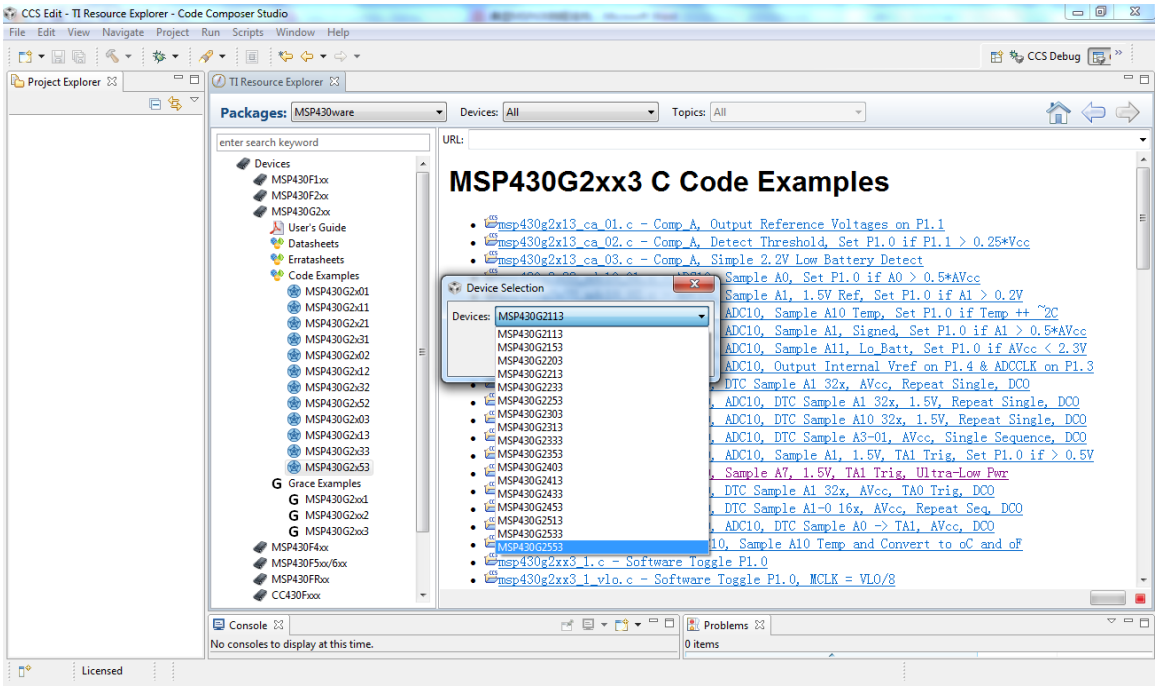


图 2.5 导入 430Ware 中的示例程序

经过这一步后，CCS 会自动生成一个包含该示例程序的工程，用户可以直接进行编译，下载和调试。

在 Development Tools 的子目录中可以找到 TI 基于 MSP430 的开发板，部分资源已经整合在软件中，另外还有部分型号在 430Ware 中也给出了链接以方便用户的查找和使用。在该目录下可以方便地找到相应型号的开发板的用户指南，硬件电路图以及参考例程。

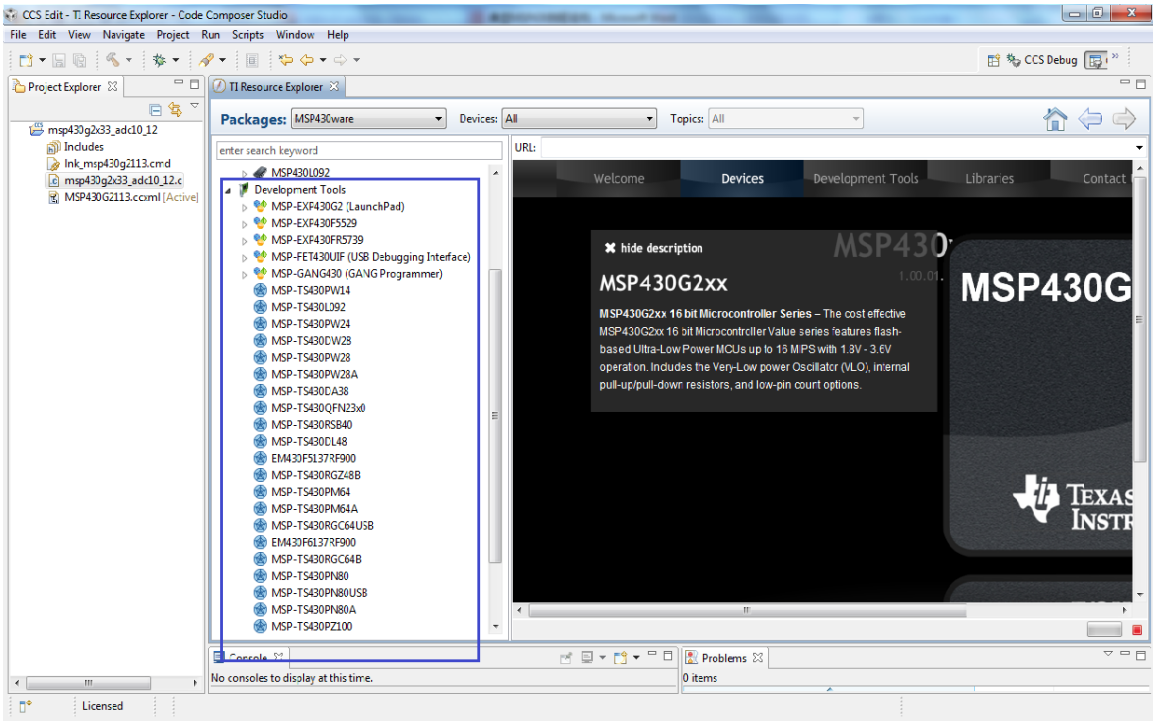


图 2.6 430Ware 包含 TI430 开发板资料

为简化用户上层软件开发，TI 给出了 MSP430 外围模块的驱动库函数，这样用户可以不用过多地去考虑底层寄存器的配置。这些支持可以在 430Ware 的 Libraries 子目录中方便地找到。目前对 DriverLib 的支持仅限于 MSP430F5 和 F6 系列。

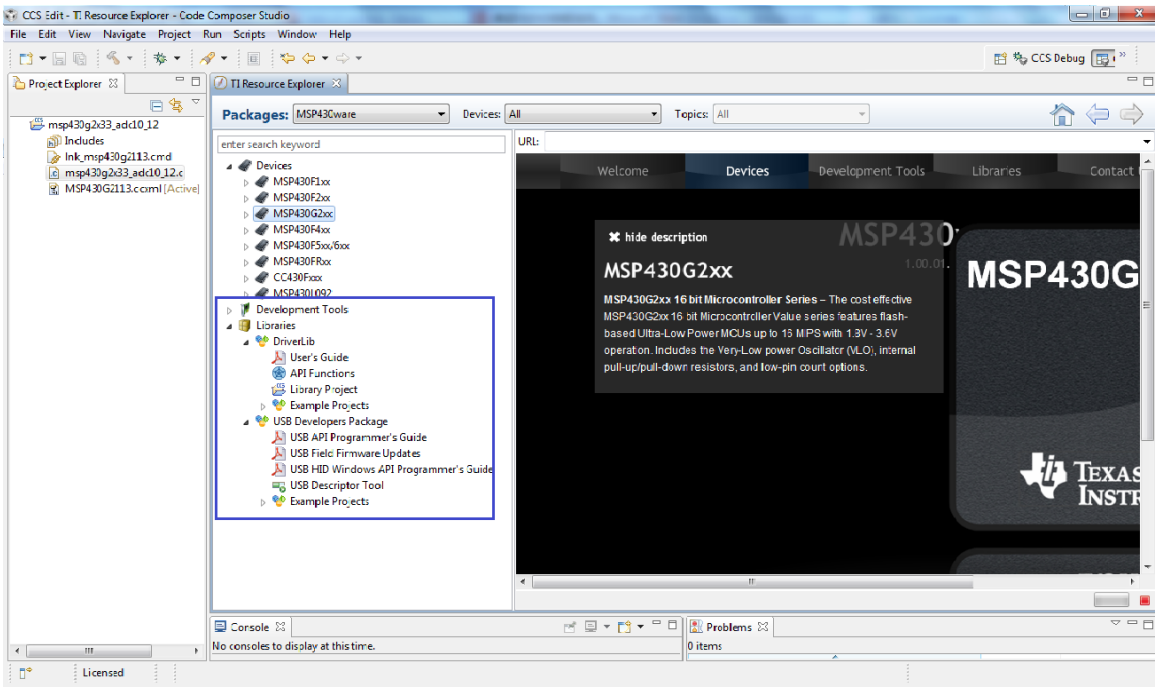


图 2.7 430Ware 包含 430 开发驱动库

通过上述描述可以看出，430Ware 是一个非常有用的工具，利用 430Ware 可以很方便地找到进行 430 开发所需要的一些帮助，包括用户指南，数据手册和参考例程。

3. Grace

3.1.Grace 软件介绍

刚接触单片机开发时最大的困惑之一可能就是对单片机中寄存器的理解和配置。C 语言对寄存器的配置相对来说会比较抽象，尤其对于初学者而言。CCS 中集成的 Grace 软件则可以帮助初学者更快地理解寄存器，同时也提供了一种全新的图形化的编程方式。Grace 是 TI 推出的集成在 CCS 中的一款简单易用的图形化 I/O 与外设配置软件，是基于 GUI 的 I/O 与外设配置的软件工具，目前支持的型号包括 MSP430F2XX/G2XX 器件，并且即将支持 F5 和 F6 系列。

Grace 使得开发人员能够简单生成包括全面注释的简单易读的 C 代码并快速完成外设的配置，利用 Grace 代码的生成，快速启动开发工作，使开发人员可以在数分钟内完成 MSP430 单片机的外设模块的配置，并可以生成 C 语言代码，极大的缩减了开发者花在配置外设上的时间，缩短了开发周期。

3.2.Grace 安装

Grace 软件有两种形式：一是作为 Code Composer Studio(CCS)的一个插件，在 CCSV5 版本中，Grace 作为一个插件已经自动安装到其中，如果使用的 CCS 版本号较低，没有自动安装 Grace 插件，也可以在 TI 官网上下载 Grace 插件，安装后即可使用；二是 Standalone 版本，在这种情况下 Grace 独立于任何编译环境运行，可以在电脑上独立运行，开发人员需要的做的只是把最终生成的代码复制到工程中即可。

3.3.Grace 开发实例

利用 Grace 实现 LED 闪烁，闪烁频率为每秒两灭一次。

3.3.1. 创建 Grace 工程

如图所示，打开 CCS，Project->New CCS Project，弹出如下对话框，输入创建新工程的名称（避免中文），继续在 Device 框中选择连接的器件，Family 选择 MSP430，Variant 选择您所使用的芯片型号，如 MSP430G2553，可以在前面的过滤文字输入框中输入所用型号的关键词进行过滤。在 Project templates and examples 框中选择 Empty Projects->Empty Grace(MSP430) Project. 其余项保持默认即可，点击 Finish 完成 Grace 工程的创建。

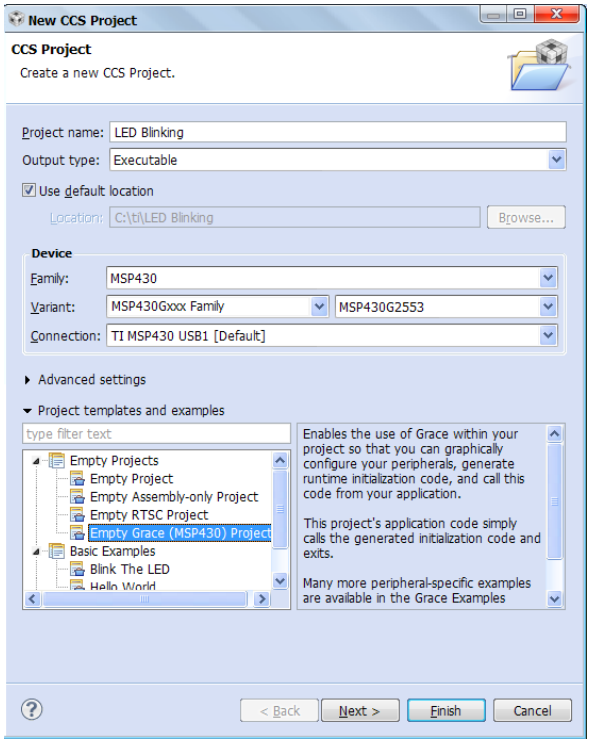


图 3.1 新建 grace 工程

3.3.2. 使用 Grace 配置 I/O 口及外设

上述操作后，CCS 会新建一个空白的基于 Grace 的工程，相较于一般 CCS 工程，Grace 的工程中会有一个后缀为.cfg 的配置文件。通过这个文件，用户可以对 MSP430 的外设进行配置。

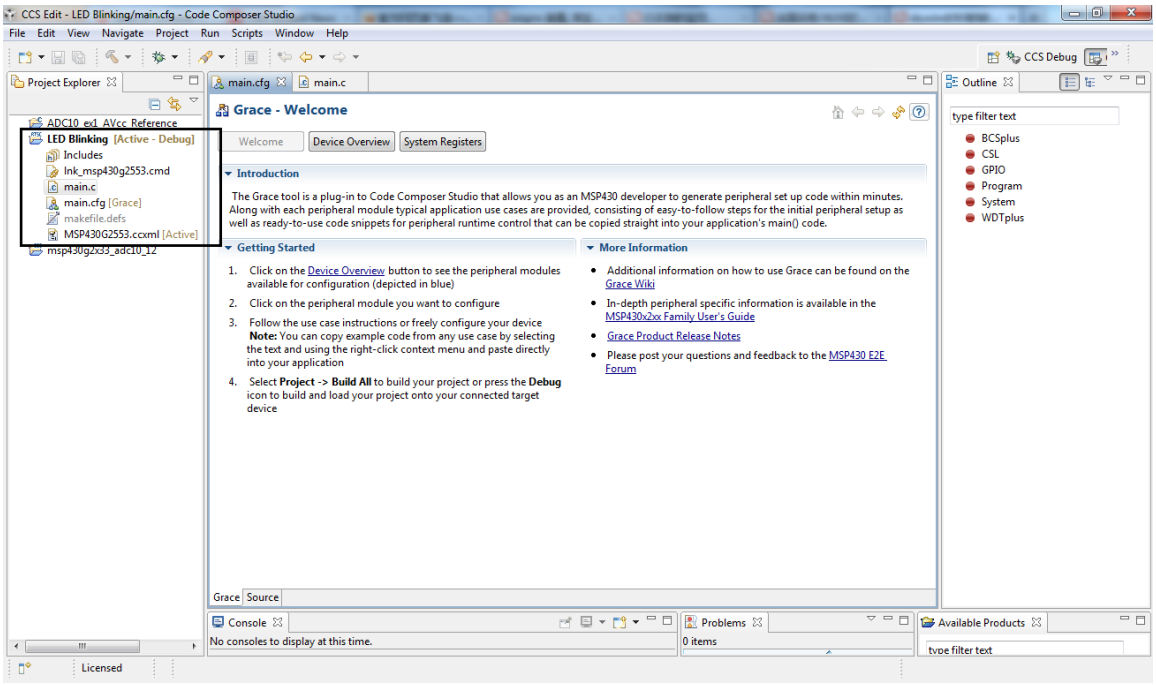


图 3.2 空白 Grace 工程界面

在配置文件上方可以看到三个按键：Welcome，Device Overview 和 System Register，如图所示，Welcome 按键是灰色显示，表明当前显示的内容即时 Welcome 下的，此时可以看到 Grace 使用过程中的一些帮助和操作指南。

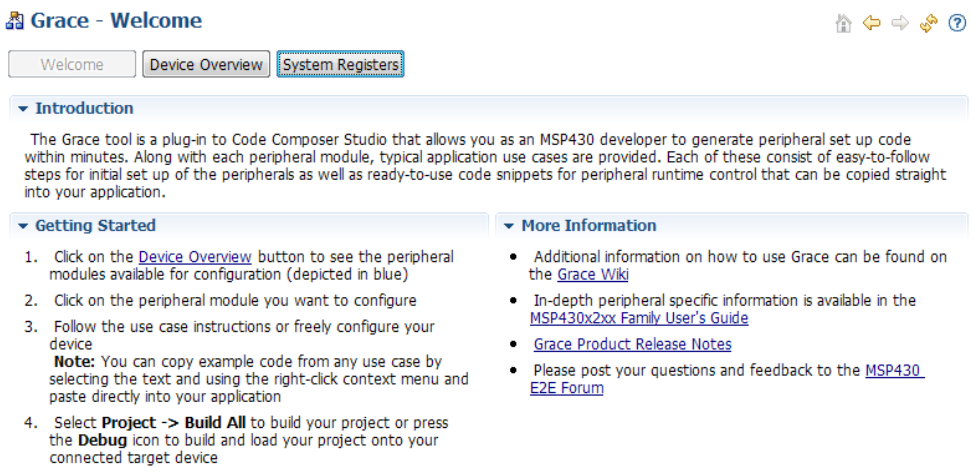


图 3.3 Grace 欢迎界面

- 点击 Device Overview 按钮，窗口显示当前片上的所有外设模块，其中蓝色的为可以通过编程配置的模块。有些模块的左下角有绿色的标记，表明当前模块正在被使用，可以通过在模块上右键，选择 Use xxxx 来实现模块的使用。当然在正在使用的模块上右键可以选择 Stop xxxx 来停止该模块的使用。

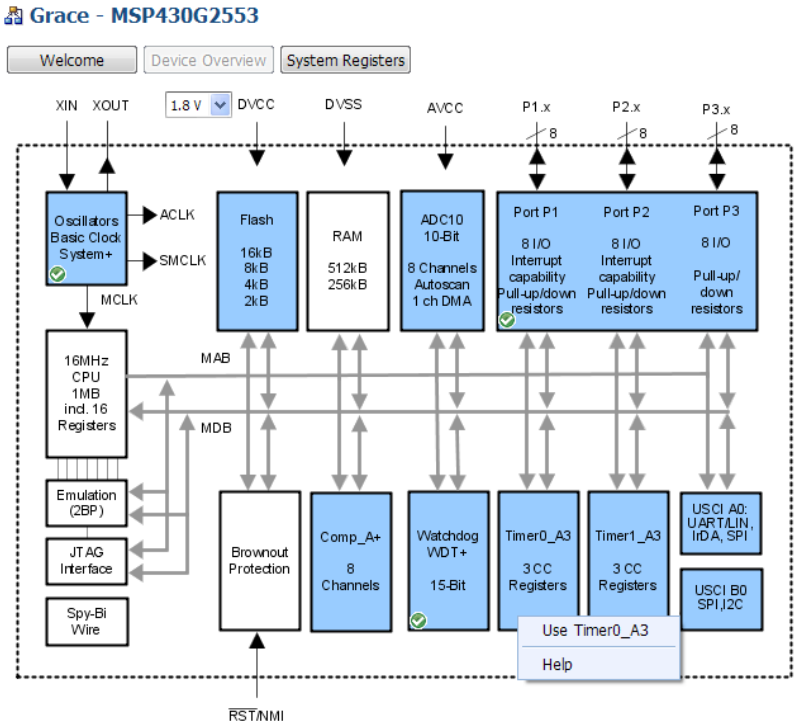


图 3.4 Grace 寄存器配置概览界面

- 当将鼠标移到某个模块上方，并点击该模块时，会出现如图所示的界面（这里点击 Timer0_A3），该模块并没有被使用，可以通过选中 Enable Timer0_A3 in my configuration 前的复选框来使用该模块：

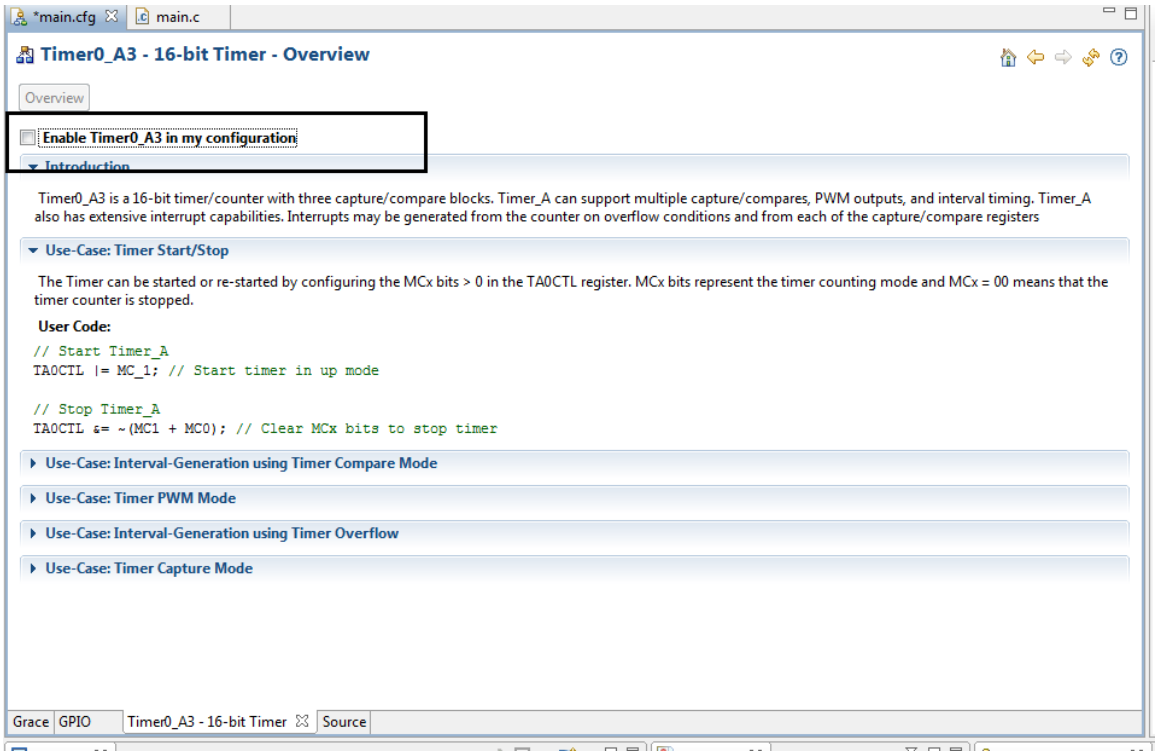


图 3.5 定时器模式配置界面

- 选中后会出现如图所示界面。选择 Basic User，可以对该模块进行一些简单的设置，如果需要使用更复杂、更高级的功能，可以点击 Power User 进行配置。

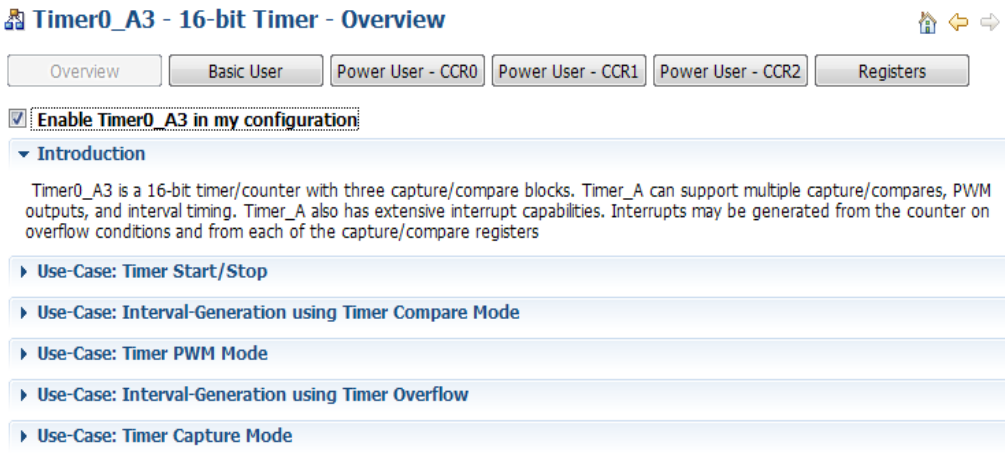


图 3.6 定时器配置界面

- 需要编程实现 LED 每秒钟闪一次，即需要产生一个频率为 1Hz 的中断，不妨选择 ACLK 为时钟源，发现这里 ACLK 默认频率为 12kHz，这意味着 Grace 软件在时钟系统模块中已经默认 ACLK 的时钟源自 VLO。在分频上不妨选用 8 分频，这样就产生了一个 1.5kHz 的时钟，该时钟将作为定时器的时钟被使用。选择计数模式为增计数模式，时间间隔在 Capture Register 中输入设定：这里的时间间隔为 1Hz，时钟频率为 1.5k，所以设置计数次数为 1500，实现 1s 的中断间隔。由于只需要产生一个定时间隔，所以选择输出比较模式，无需输出引脚。注意 CCR0 寄存器不能设为模式 2、3、6 和 7，如果设置了其中任何一个，Grace 会自动报错并提示出错的原因。这里会发现左边的 Input Selection 和 Capture Mode 两栏是灰色，即不能进行选择，因为在输出捕获模式下这两栏的配置是没有意义的，只有选择输入捕获模式，灰色部分才会被激活。
- 再在下面使能捕获/比较中断，这个中断即为自定义的定时器每隔 1s 所产生的一个中断。填写中断句柄，该句柄相当于是自定义的一个中断服务函数，在中断函数中被调用。最后选择中断之后的状态，根据实际时钟使用的状况，选择在中断结束后进入的低功耗模式。

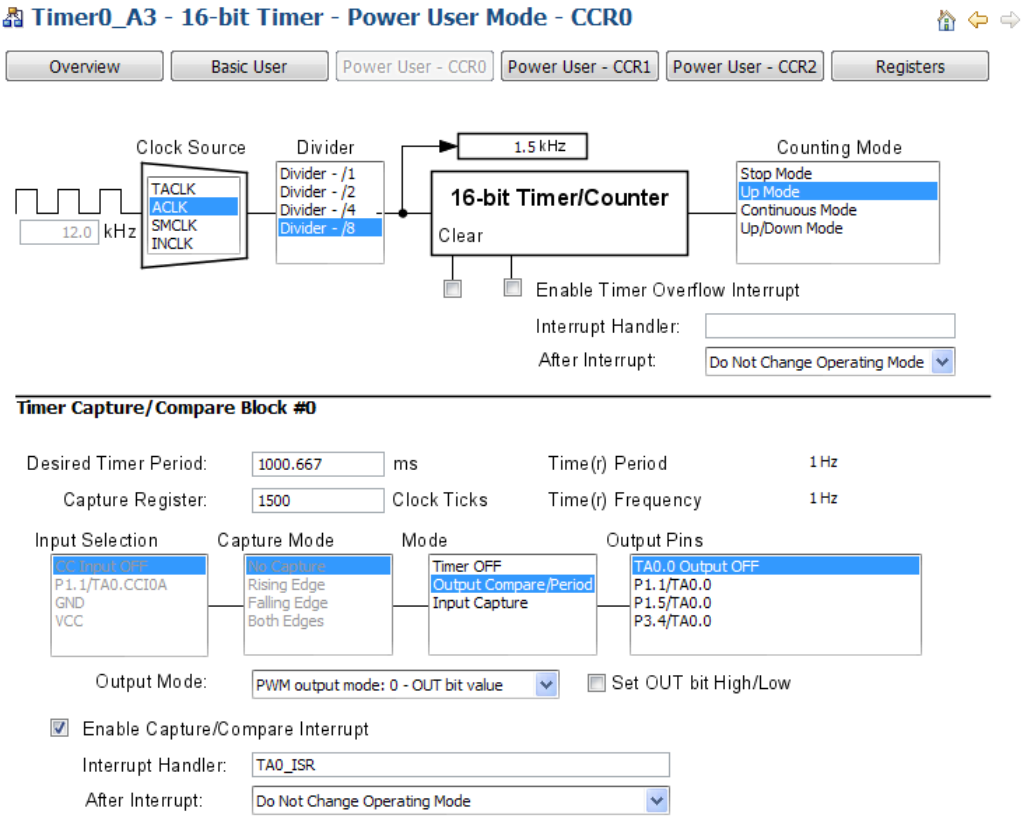


图 3.7 定时器详细配置界面

- 在配置之后可以点击 Register 来查看当前寄存器每一位的值。将鼠标移到某个寄存器位时，会显示该位的定义和功能。

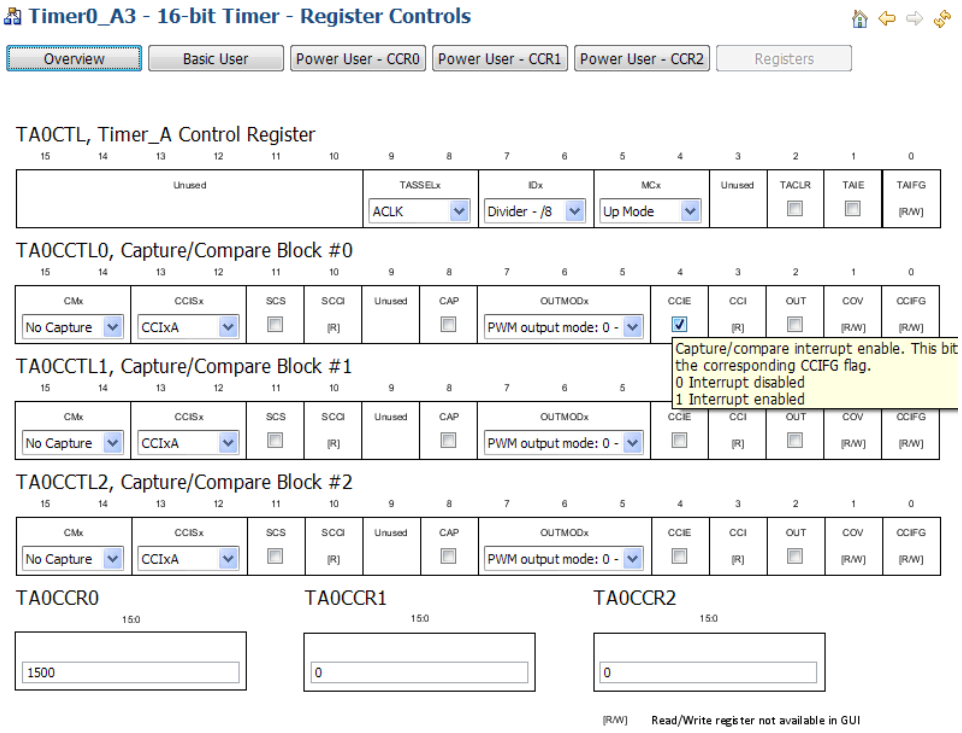


图 3.8 Grace 定时器寄存器列表界面

- 以上，完成了对定时器外设的配置，下面点击左下角的 Grace 标签或右上角的 图标，回到 **Error! Reference source not found.**主窗口进行 I/O 口的配置。点击 I/O 口模块，根据所使用的芯片封装点击相应的按钮。在此选择 Pinout 20-TSSOP/20-TDIP，一个 MSP430G2553 芯片就出现了，点击蓝色向下箭头，对相应引脚的功能进行选择和配置。不妨选择配置 P1.0 为 GPIO Output，用来驱动 LED。

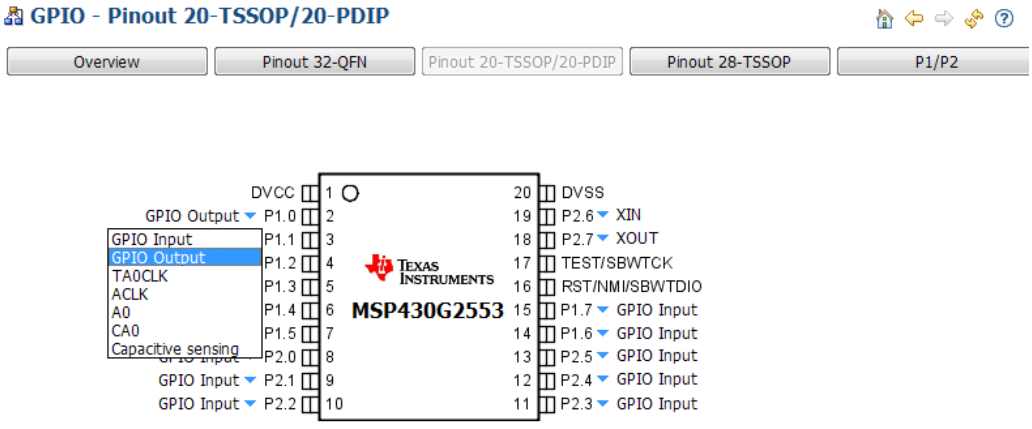


图 3.9 Grace 引脚功能配置界面

3.3.3. 生成可编译文件

经过上述步骤，程序中涉及到的两个外设：定时器和 I/O 口已经配置完成了，接下来可以进行编译和最后的程序完善。单击工程，然后单击 CCS 工具栏上的  图标，进行编译，这时会提示有错误出现，这是因为刚才定义的中服务函数没有定义。右键单击 main 函数里面的 CSL_init()函数，选择 Open Declaration，打开 CSL_init.c 文件，或者在 Project Explorer 中找到并打开 CSL_init.c 文件。

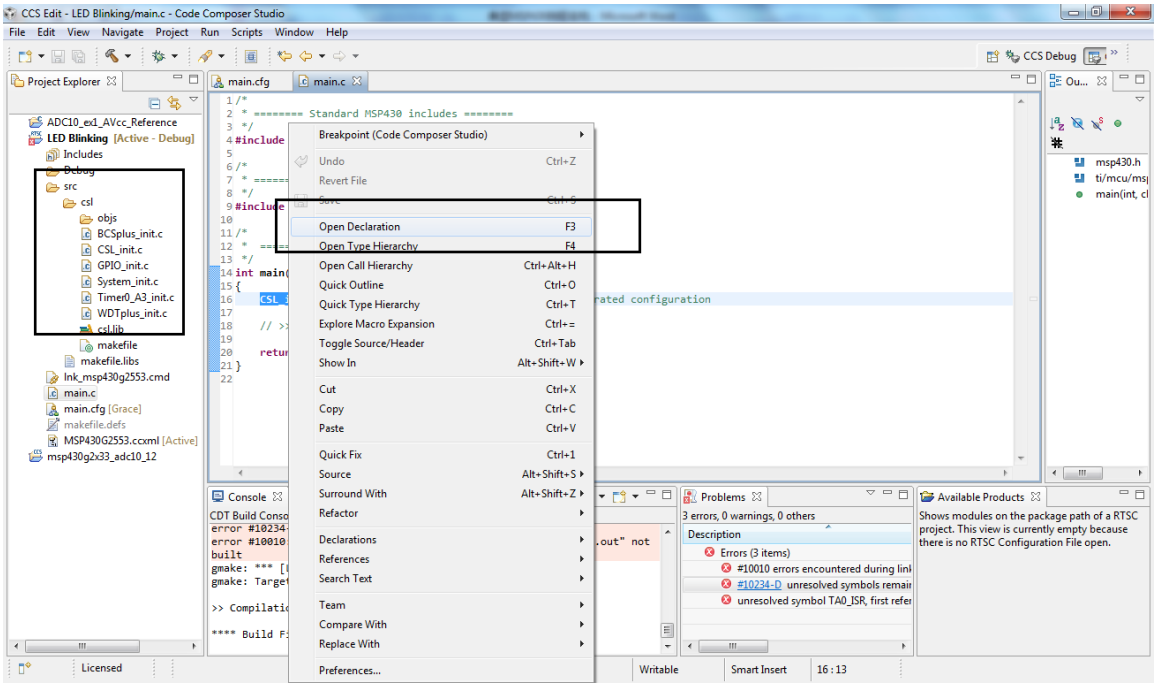


图 3.10 Grace 代码添加

在该文件中发现有对之前定义的中断服务子程序 TA0_ISR 的声明，但却没有定义。在如图所示位置（蓝色部分）添加代码，实现 LED 的状态切换，然后编译，点击  图标即可进行下载仿真。

```
/*
 *  ======== Interrupt Function Definitions ========
 */

/* Interrupt Function Prototypes */
extern void TA0_ISR(void);

void TA0_ISR(void)
{
    P1OUT ^= BIT0;
}


/*
 *  ======== Timer0_A3 Interrupt Service Routine ========
 */
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR_HOOK(void)
{

    /* Capture Compare Register 0 ISR Hook Function Name */
    TA0_ISR();

    /* No change in operating mode on exit */
}
```

图 3.11 代码添加位置

综上，就实现了利用 Grace 软件实现了定时器控制 LED 闪烁的实验。

4. MSP430 软件开发编程介绍

4.1.MSP430 软件开发流程

在 MSP430 的软件编程中，较常见的是使用 C 语言进行开发。CCS 一个主要的功能即是将 C 代码转换成机器可以识别的机器代码，从而实现程序功能。如图所示为 MSP430 的软件开发流程图。其中阴影部分为对 430 进行 C 语言编程软件的一个开发流程，其余部分则为一些增强开发过程的外围功能。用户编写的 C/C++源代码由 C/C++编译器编译生成汇编代码，之后由汇编工具生成对象文件，最后由连接器生成可执行的对象文件，在单片机上运行。编程过程中，用户面对的是 C/C++编程这一块，所以在后文对 C/C++编译部分进行展开介绍。

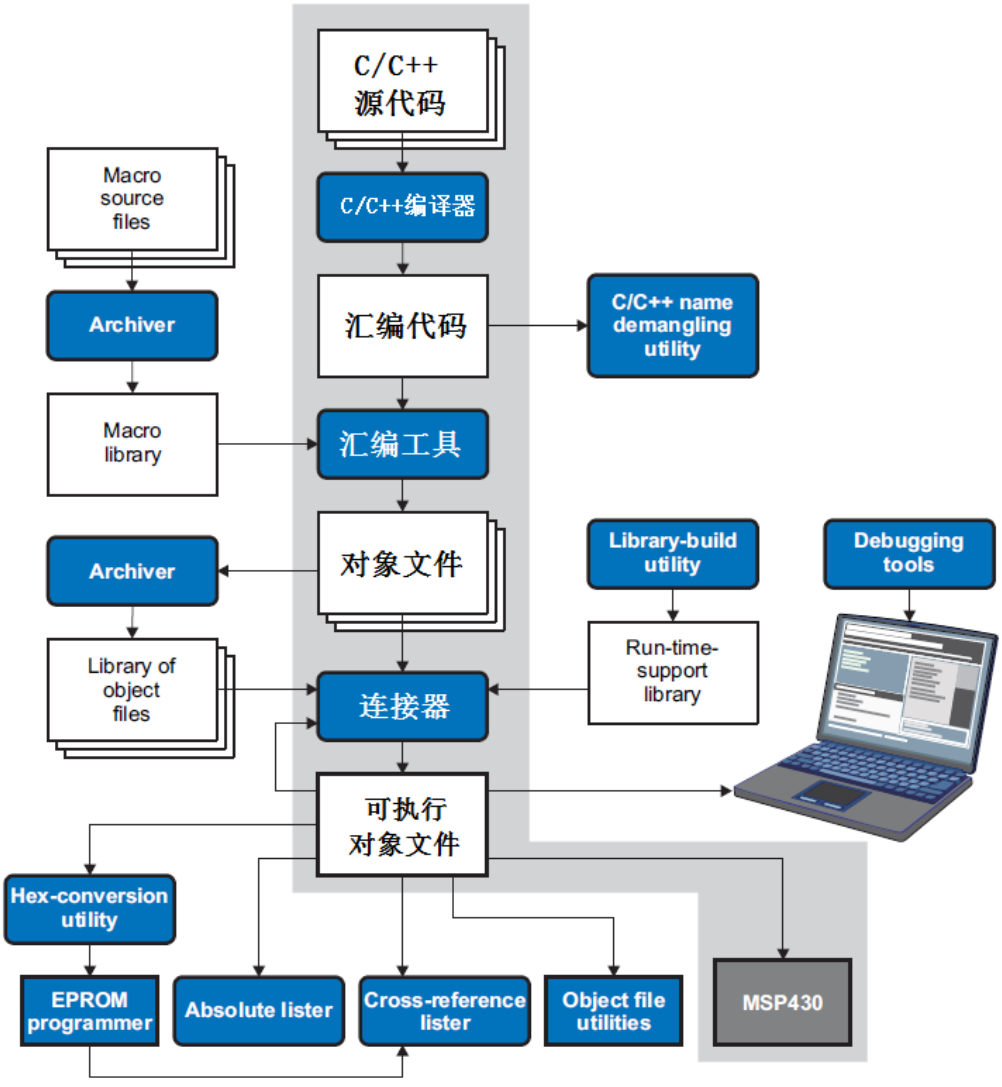


图 4.1 MSP430 软件流程图

4.1.1. C/C++编译器

一般来说，C/C++语言编译器中功能的实现遵循以下 ISO 标准：

- ISO 标准 C 语言

C/C++编译器中 C 语言部分遵循 C 语言标准 ISO/IEC 9889:1990，相当于美国国家信息系统编程语言标准定义的 cX3.159-1989 标准，俗称 C89，此定义是由美国国家标准协会出版的。当然 ISO 标准也发布过 1999 年版，但是 TI 公司的编译器（CCS）仅支持 1990 年而非 1999 年版 ISO，这也是为什么在进行不同编译环境之间的软件移植时，在程序编译过程中可能会出现语法不兼容的情况。

- ISO 标准 C++

C/C++编译器中 C++语言部分遵循 C++语言标准 ISO/IEC14882:1998。编译器还支持嵌入式 C++语言，但对某些特定的 C++类型不支持，详情可参考 slau132G。

- ISO 标准的实时支持

编译器工具自带有庞大的实时运行数据库。所有的库函数都符合 C/C++标准。该数据库涵盖的功能有标准输入与输出，字符串操作，动态内存分配，数据转换，计时，三角函数、指数函数以及双曲函数分析等。

ANSI、ISO 标准定义了 C 语言中那些受目标处理器特点、实时运行环境或主机环境影响因素的一些特征。出于实效性考虑，这一功能在不同编辑器之间存在一定差异。

C 语言库中所不支持的功能有：

- 实时运行库对大长度和多字节的数据仅提供最小程度的支持。若使用 wchar_t 类型将仍然被用作 int 类型来操作。宽字符集相当于 char 型，库文件中包含了头文件<wchar.h>和<wctype.h>，但并不支持其中所有功能。
- 实时运行库包含一个头文件<locale.h>，但同样仅提供最小程度的支持，唯一支持的语言环境是 C 语言环境。

4.2.MSP430 C 语言简介

4.2.1. 数据类型

MSP430 的数据类型定义如下所示。

			Range	
Type	Size	Representation	Minimum	Maximum
char, signed char	8 bits	ASCII	-128	-127
unsigned char, bool	8 bits	ASCII	0	255
short, signed short	16 bits	2s complement	-32 768	32 767
unsigned short, wchar_t	16 bits	Binary	0	65 535
int, signed int	16 bits	2s complement	-32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	16 bits	2s complement	-32 768	32 767
float	32 bits	IEEE 32-bit	1.175 495e-38	3.40 282 35e+38
double	32 bits	IEEE 32-bit	1.175 495e-38	3.40 282 35e+38
long double	32 bits	IEEE 32-bit	1.175 495e-38	3.40 282 35e+38
pointers, references, pointer to data members	16 bits	Binary	0	0xFFFF
MSP430X large-data model pointers, references, pointer to data members	20 bits	Binary	0	0xFFFFF
MSP430 function pointers	16 bits	Binary	0	0xFFFF
MSP430X function pointers	20 bits	Binary	0	0xFFFFF

4.2.2. 变量种类

在程序中需要定义和使用一些变量，一般来说可以在以下几个位置进行变量的声明：1 函数内部；2 函数的参数定义；以及 3 所有函数的外部。这样，根据声明位置的不同，可以将变量分为局部变量，形式参数和全局变量。

```
#include "msp430g2553.h"

int add (int x, int y);

int z = 9;  //global vars        全局变量

void main()
{
    int a = 2;
    int b = 4;  //local vars & init values

    z = add(a,b);                局部变量

    while(1)
    {
        _NOP();
    }
}

int add(int x, int y)   形参
{
    return (x+y);
}
```

如图所示，变量 z 在函数外外部进行声明，为全局变量。全局变量，顾名思义该变量可以被程序中所有函数使用。在运行过程中，无论执行哪个函数都会保留全局变量的值。在 CCS 默认的 cmd 配置文件中，全局变量分配在内存 RAM 空间。相对于在函数外定义的全局变量，局部变量则是指在函数内部定义的变量，例如图中的整型 a 和 b，和全局变量不同的是，局部变量只能被当前函数使用，且只有在函数调用时局部变量才会生成，同时当函数调用完成后，该变量空间也被释放，直至函数再次调用，改变量才会重新生成，重新赋值。还有一种变量类型为形式参数，如图中定义的子函数 add，括号中的整型数 x 和 y 为形参，在 add 子函数中不需要对 x 和 y 进行声明就可以直接使用。

4.2.3. 变量存储类型

- Static 静态变量

在前文中提到局部变量只有在函数内有效，在离开函数时，内存空间被释放，变量值也会清除，待到再次进入函数时重新生成变量，执行变量的赋值。而 static 静态变量和一般的局部变量的差别在于，在离开函数时，静态变量的当前值会被保留，可在下次进入函数时使用。下图中给出了两段程序，定义 add（）子函数，实现的是整型 a 的累加，可以通过全局变量 z 来观察程序的运行状况。两段代码的差别在于右边将 add 子函数中的变量 a 定义为静态变量。通过断点调试，观察到 z 的变化分别如图所示。很容易理解这样的结果产生的原因：在调用完 add 函数后，局部变量 a 的空间被释放，在再次进入 add 函数时，重新生成变量 a，并初始化为 1，所以 z 的值总是 2。而将 a 定义为静态变量后，初次调用

add 后，a 的值变为 2，根据静态变量的定义，此时 a 的值会被保留，当再次调用 add 函数时，a 不会被再次初始化而是使用上次的值 2，所以会观察到 z 的值依次递增。

```
#include "msp430g2553.h"

int add ();

int z;  //global vars

void main()
{

       while(1)
       {
              z = add();
              _NOP();
       }
}

int add()
{
       int a=1;

       return (a++);
}
 }
```

```
#include "msp430g2553.h"

int add ();

int z;  //global vars

void main()
{

       while(1)
       {
              z = add();
              _NOP();
       }
}

int add()    将a声明为静态变量
{
       static int a=1;

       return (a++);
}
 }
```

运行结果

```
z=2;
z=2;
z=2;
...
```

```
z=2;
z=3;
z=4;
...
```

- extern 外部变量

在未作特殊说明的情况下，在某个文件下定义的变量只能被当前文件，甚至是特定函数（局部变量）所使用，这样当工程中包含多个文件时，变量无法被所有文件使用。而 extern 变量则解决了不同文件之间变量的调用问题。将变量声明为 extern 型，则该变量不仅可以在当前文件中使用，同时也可以被工程中其他文件中的函数调用。如图所示，file1 和 file2 为同一工程中的两个源文件，file1 中定义了变量 z，在 file2.c 中通过语句 extern int z，使得 file1 中的 z 变量同样可以在 file2 中使用。

file1.c

```c
#include "msp430g2553.h"

int add ();

int z;  //global vars

void main()
{

       while(1)
       {
              z = add();
              _NOP();
       }
}

int add()
{
       int a=1;

       return (a++);
}
 }
```

file2.c

```c
#include "msp430g2553.h"

extern int z;

 ...

 ...
```

4.2.4. 运算符

MSP430 编程中使用的运算符和其他 C 语言中使用的类似，但是在 MSP430 编程学习中会经常看到逻辑运算符和按位运算符，尤其是在对寄存器进行配置的时候。在这里对这些运算符进行简要的阐述和说明。

操作符	说明
&&	逻辑与
||	逻辑或
!	逻辑非
位操作	
&	逻辑与
|	逻辑或
^	逻辑异或
>>	右移
<<	左移

- 逻辑与和逻辑或比较容易理解，逻辑分为逻辑 0 和逻辑 1，其中逻辑 1 可以是任何非 0 的值。与逻辑 0 相与的结果都为逻辑 0，与逻辑 1 相与则保持不变；与逻辑 0 相或保持不变，与逻辑 1 相与则结果都为 1。注意到表格中逻辑与和逻辑或分别都有两种，其中单个符号的为按位操作，返回值为按位逻辑运算后的结果，而两个符号的逻辑运算的返回值只有逻辑 0 和逻辑 1 两种，例如：

5&&8	5&8
因为 5 和 8 都为逻辑 1，所以返回值为逻辑 1，即真值	需按位进行与操作：0101&1000，结果为 0000

或运算与之相似。一般来说逻辑与（&&），逻辑或（||）和逻辑非（！）在判断语句中比较常见，作为 if…else…或 while（）语句的判断条件出现，而按位操作在 MSP430 编程中常出现在寄存器的配置或数字计算中。

```
#include "msp430g2553.h"

void main()
{
      ...

      P1OUT = BIT0 | BIT1;
      P2OUT|= BIT0;
      P3OUT|= BIT0 + BIT1;
      ...

}
```

```
/************************************************************
* STANDARD BITS
************************************************************/

#define BIT0                (0x0001)
#define BIT1                (0x0002)
#define BIT2                (0x0004)
#define BIT3                (0x0008)
```

如图示例程序所示，对通用 IO 口的输出值（PxOUT）进行配置，右图为头文件中对 BIT0 和 BIT1 的定义。上图中给出了三种对寄存器赋值的方法，都可以实现对寄存器的赋值。第一种，通过按位或操作，P1OUT 被配置为（0x0001|0x0002），这是 16 位的写法，转为 2 进制后，后 8 位为 0000 0011，即 P1OUT 寄存器的最后两位被置 1，也就是说将 P1.1 和 P1.0 的输出置为高电平。其实在过程中会发现这段代码等效于：

P1OUT = 0x0003;

上述代码虽然可以实现同样的功能，但显然相比较于用按位或操作的书写方式，后者的可读性较强。在 MSP430 的寄存器配置中经常会看到如图第二种所示的配置方法，这种方法也是在 MSP430 编程中比较推荐的。

P2OUT|=BIT0; 等同于 P2OUT = P2OUT | BIT0; 我们知道“|”操作不改变原先的值，这样和对寄存器直接赋值一个最大的差别在于保留了该寄存器内原来的值，也就是说在赋新值的同时不影响寄存器之前的状态。单片机在初始化的时候寄存器也会有不同的初始化值，大多数情况下在不使用的时候无需对寄存器初始值进行更改，而直接赋值的方法其实是对寄存器的所有位，无论是否使用到都进行的了重新定义，如果对寄存器其他位没有考虑全面的话，可能会造成不良的影响，所以建议在进行寄存器配置的时候尽量使用第二种方式。此外，有时会看到第三种方式，乍一看会困惑于代码中的“+”，其实在明白 BIT0 和 BIT1 真正代表的数值含义后很容易理解第三种用“+”取代“|”的道理所在。

- 逻辑异或（^）比较常见于依赖前次状态的状态改变，比如说 LED 闪烁，LED 闪烁的实现其实是不断变换电平的高低状态，如果前次为高则改为低，相反如果前次为低则改为高。这种情况下就可以用逻辑异或（^）来实现：

```
While(1)
{
     PxOUT ^=BIT0;
    _delay();
}
```

该段代码简单地实现了 P1.0 端口高低电平的改变，当 P1.0 与一个 LED 相连，即实现了 LED 的闪烁。可以通过下面的表格更为清晰地观察是如何利用逻辑异或（^）简单地实现上述功能，不妨假设 PxOUT 寄存器 8 位初始值为 0100 1110：

执行次数	PxOUT	BIT0
0	0100 1110	0000 0001
1	0100 1111	0000 0001
2	0100 1110	0000 0001
… …		

从表格中很清楚地看到，合理地使用逻辑异或可以很容易地实现某特定位的状态改变，同时不影响其他位的状态。

- 左移（<<）和右移（>>）操作则比较多地用于数值计算。对于这两个操作从数值上理解为：
  x = x<<n; 等价于 x = x *2^n; 左移 n 位，相当于乘以 2 的 n 次方；同理，
  x = x>>n; 等价于 x = x /2^n; 右移 n 位，相当于除以以 2 的 n 次方。移位操作是基于位操作进行的，所以移位操作不限于进行数值计算，巧妙地应用移位操作可以进行一些基于判断的循环操作。

5. CCS MSP430 工程结构解析

在 CCS 使用指南一章中对如何在 CCS 中新建一个工程做了详细的介绍，这里就一个完整的 MSP430 工程中包含的文件的作用做简单的介绍和说明。

如图所示，从 CCS 窗口左侧的 Explorer 导航栏中观察工程，发现工程中的文件分为 4 种，由上至下分别为 1. Includes；2. Cmd 配置文件；3. 源文件；4. Ccxml 配置文件。

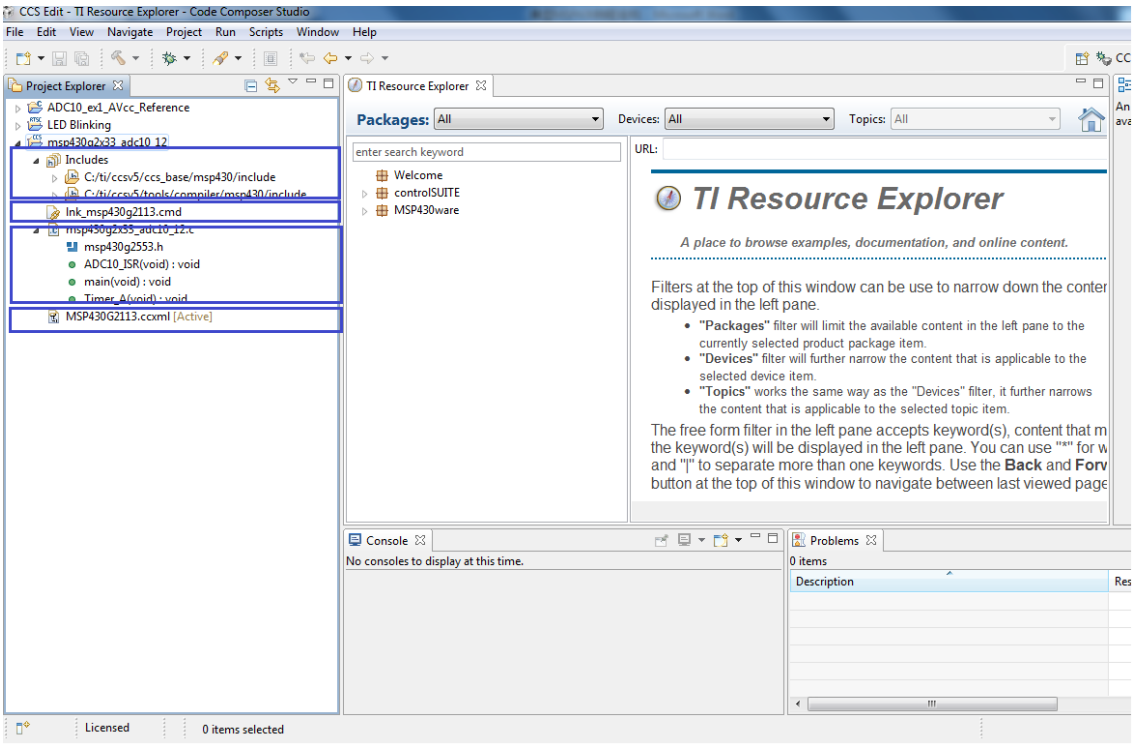


图 5.1 CCS 标准工程组成

5.1.includes

在该目录下包含了用户设置的头文件路径下的所有头文件，如图所示为 CCS 默认的两个头文件路径，分别为 MSP430 的头文件和 C 语言相关的头文件。前者提供了不同型号的 MSP430 的头文件定义，包括寄存器定义，常用位定义等，这部分内容是与编译平台相关的，这意味着不同的编译软件提供的头文件可能略有不同，所以在做平台间的移植的时候，注意要同时考虑到头文件间的差异，可以在源文件中修改，更方便的做法是将头文件覆盖。

那如何在工程中添加自定义的头文件呢？在工程名上右击，选择“properties”，在出现的属性对话框中选择“Build”→“MSP430 Complier”→“include options”，如图所示，有两种添加方式，一种是向其中添加头文件，另一种是添加一个目录，包含该路径下所有的头文件。可以在框边上的添加和删除按键进行相应的操作。如图可以看到现在添加的 include 路径已经有两个，即 CCS 默认的两个头文件路径。

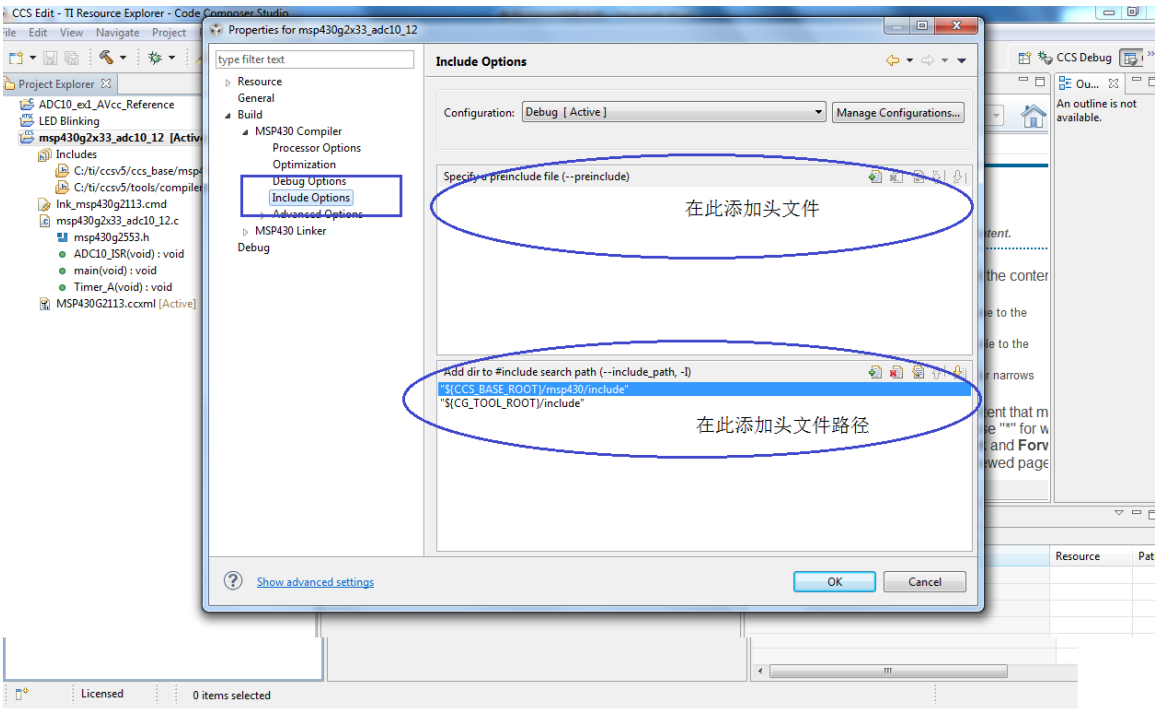


图 5.2 CCS 工程头文件添加方法

5.2.Cmd 配置文件

如图所示为 MSP430G2553 的默认 cmd 配置文件，该文件主要用来分配 430 内部的 FLASH 和 RAM 空间，在 link 过程中告诉链接器怎样进行地址的计算和空间的分配。文件的开始，在 MEMORY 这一段，会对选择型号的芯片（MSP430G2553）的存储单元映射进行定义，这部分是器件相关的，不同型号的器件 FLASH，RAM 的大小以及映射关系都不尽相同，所以对于不同型号的 430 会分别有不同的 cmd 文件一般而言，用户不会对该部分的内容进行修改操作。另外还有就是对 SECTIONS 的定义，在这部分主要是对程序的内容进行地址空间的分配。每个代码程序中都会包含有不同的段（section），默认对每个段的定义格式都以“.”开头，编译器对段的名称和定义有规定值。如下图的一个实例程序所示：定义的全局变量会储存在.bss 段，在程序中初始化的值会存储在.cinit 段，.stack 段中则为程序中定义的局部变量，而书写的指令代码则会存储在.text 段中。

```
/****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                            */
/****************************************************************************/

MEMORY
{
    SFR                     : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT        : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT       : origin = 0x0100, length = 0x0100
    RAM                     : origin = 0x0200, length = 0x0200
    INFOA                   : origin = 0x10C0, length = 0x0040
    INFOB                   : origin = 0x1080, length = 0x0040
    INFOC                   : origin = 0x1040, length = 0x0040
    INFOD                   : origin = 0x1000, length = 0x0040
    FLASH                   : origin = 0xC000, length = 0x3FE0
    INT00                   : origin = 0xFFE0, length = 0x0002
    INT01                   : origin = 0xFFE2, length = 0x0002
     ……
     ……
    INT13                   : origin = 0xFFFA, length = 0x0002
    INT14                   : origin = 0xFFFC, length = 0x0002
    RESET                   : origin = 0xFFFE, length = 0x0002
}

/****************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                              */
/****************************************************************************/

SECTIONS
{
    .bss       : {} > RAM               /* GLOBAL & STATIC VARS              */
    .sysmem    : {} > FLASH                /* DYNAMIC MEMORY ALLOCATION AREA    */
    .stack     : {} > RAM (HIGH)         /* SOFTWARE SYSTEM STACK             */

    .text      : {} > FLASH              /* CODE                              */
    .cinit     : {} > FLASH              /* INITIALIZATION TABLES             */
    .const     : {} > FLASH              /* CONSTANT DATA                     */
    .cio       : {} > FLASH               /* C I/O BUFFER                      */

    .pinit     : {} > FLASH              /* C++ CONSTRUCTOR TABLES            */

    .infoA     : {} > INFOA              /* MSP430 INFO FLASH MEMORY SEGMENTS */
    .infoB     : {} > INFOB
    .infoC     : {} > INFOC
    .infoD     : {} > INFOD

    .int00   : {} > INT00                /* MSP430 INTERRUPT VECTORS          */
    .int01   : {} > INT01
     ……
     ……
    .int13   : {} > INT13
    .int14   : {} > INT14
    .reset   : {} > RESET              /* MSP430 RESET VECTOR               */
}

/****************************************************************************/
/* INCLUDE PERIPHERALS MEMORY MAP                                           */
/****************************************************************************/

-l msp430g2553.cmd
```

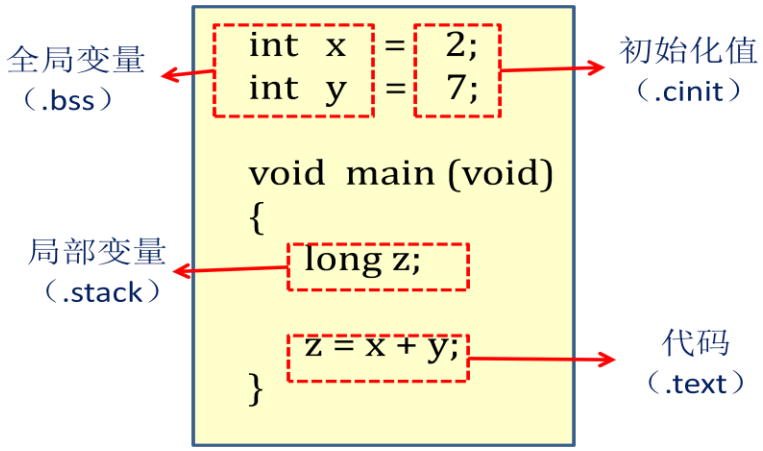


图 5.3 变量类型及存储段

MSP430 的存储空间有 FLASH 和 RAM 两种，具体上述的代码中出现的各段如何在内存空间中进行分配则在 cmd 文件中配置。例如说在图中的 cmd 文件中：

```
.bss        : {} > RAM                  /* GLOBAL & STATIC VARS                */
```

比较容易读懂该语句实现的功能：将程序中未初始化的全局变量和静态变量，即 bss 段，保存在 RAM 中。同样在进行 430 编程时，该部分内容在大多数情况下也不需要进行修改。但同时，需要格外注意所选用型号器件的 RAM 和 FLASH 空间，有时编译中会出现 cmd 文件相关的错误，有很多是因为定义的变量空间或者程序需要的空间超出了 RAM 范围，所以在编程时要对内存空间的安排有一定的考虑，尤其是一些 RAM 空间有限的型号。例如，我们用 MSP430G2 系列进行点阵 LCD 的图形显示，要求有比较大的空间进行图形数据变量的存储，我们一般会将其定义为 const 常量，这时就要去注意程序链接时.const 段指向的位置空间是否足够大，在默认的 cmd 配置文件中发现，.const 段是默认指向 FLASH 空间的，而 FLASH 空间一般比较足够。如果在进行变量定义时，不慎将其落在 RAM 空间的段中，在编译时可能会出现错误提示。

在 cmd 文件中可以对堆栈的大小进行定义：

```
-c
-stack  0x0100
-heap   0x0100
```

上述语句定义了 stack 大小为 0x100，heap 大小为 0x100。对堆栈大小的定义也可以在 CCS 的设置中进行定义，打开工程的属性对话框，在 MSP430 Linker 的 Basic Options 中有对系统堆栈大小的定义。在使用到需要大量占用堆栈资源的程序代码时要对堆栈大小重新进行定义，例如 sprintf()语句。

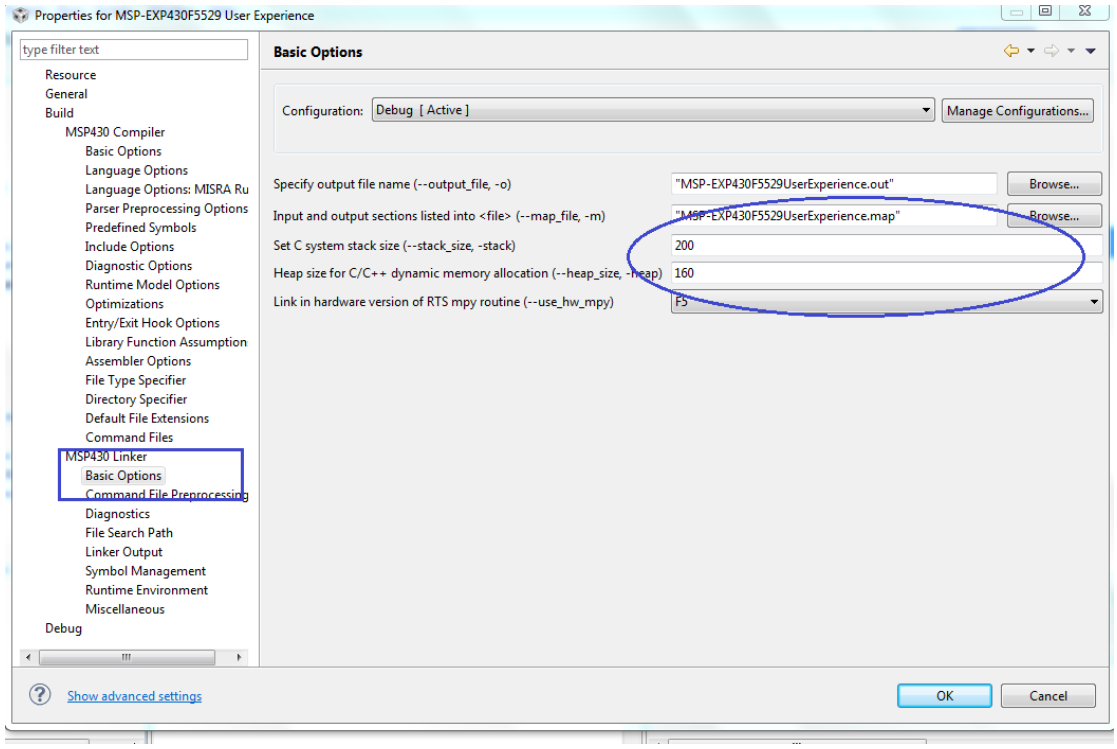


图5.4 堆栈空间修改

5.3.源文件

在工程名上右击选择“add files...”，可以向工程中添加文件，包括源文件。源文件的类型可以是 c 文件也可以是汇编文件。单击文件前的三角下拉菜单可以看到该文件中包含的头文件，全局变量和函数。关于 TI 例程的典型结构在后文中有详细的介绍。

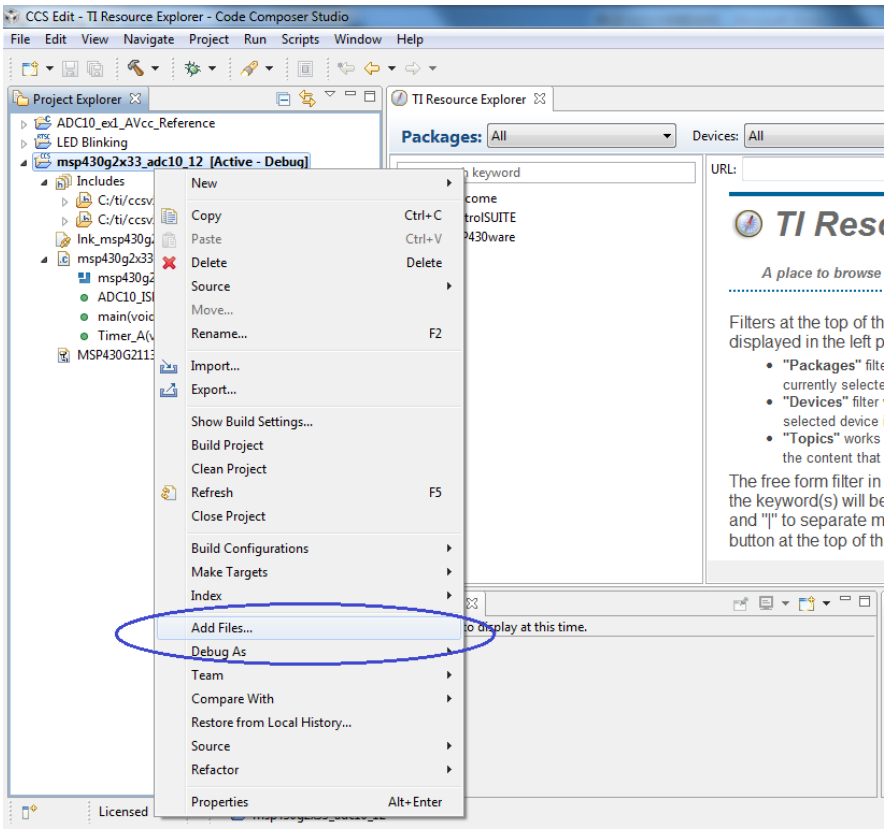
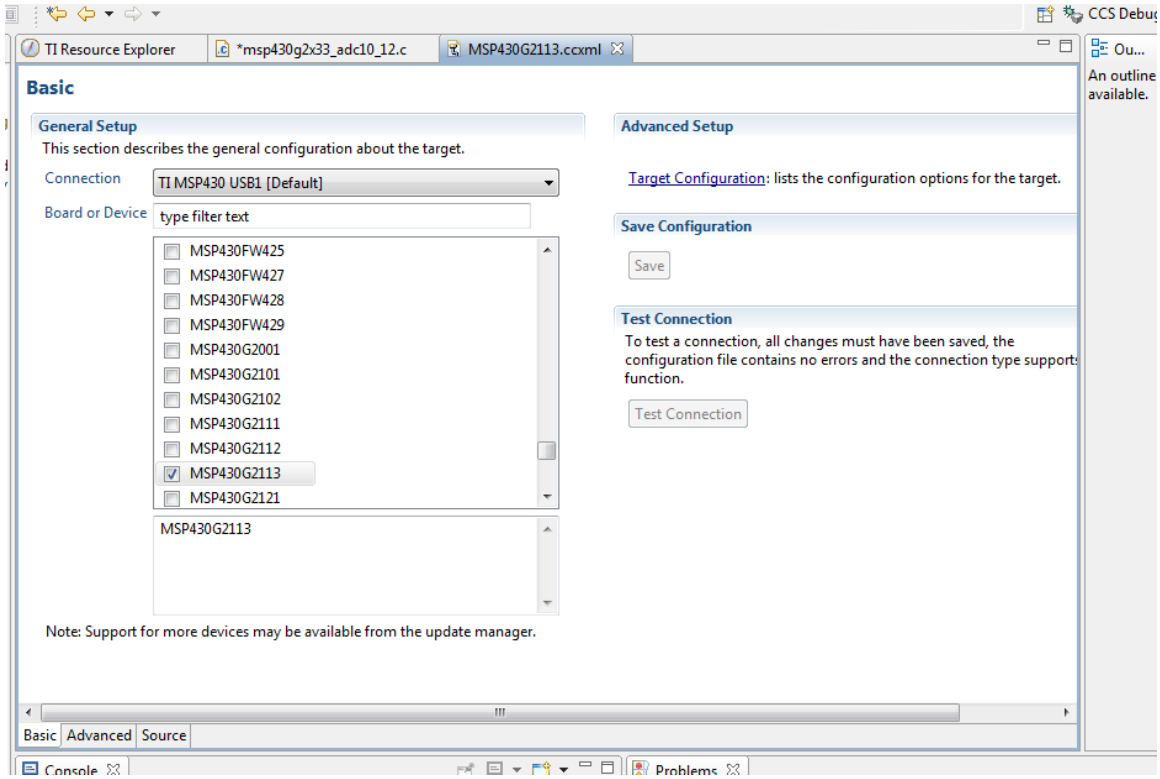


图 5.5 源文件添加

5.4.ccxml 配置文件

如图所示为目标板配置文件，在该文件中实现了对链接目标的定义和设置。一般该文件会在创建工程时自动新建，其中大部分内容在新建工程时已经进行了配置，包括连接器的选择，连接器件的选择等。



5.6 配置文件

如果在工程创建时该文件没有自动生成，用户也可以进行新建操作创建一个目标配置文件。选择“file”→“new”→“target configuration file”新建配置文件，输入文件名称和保存路径。

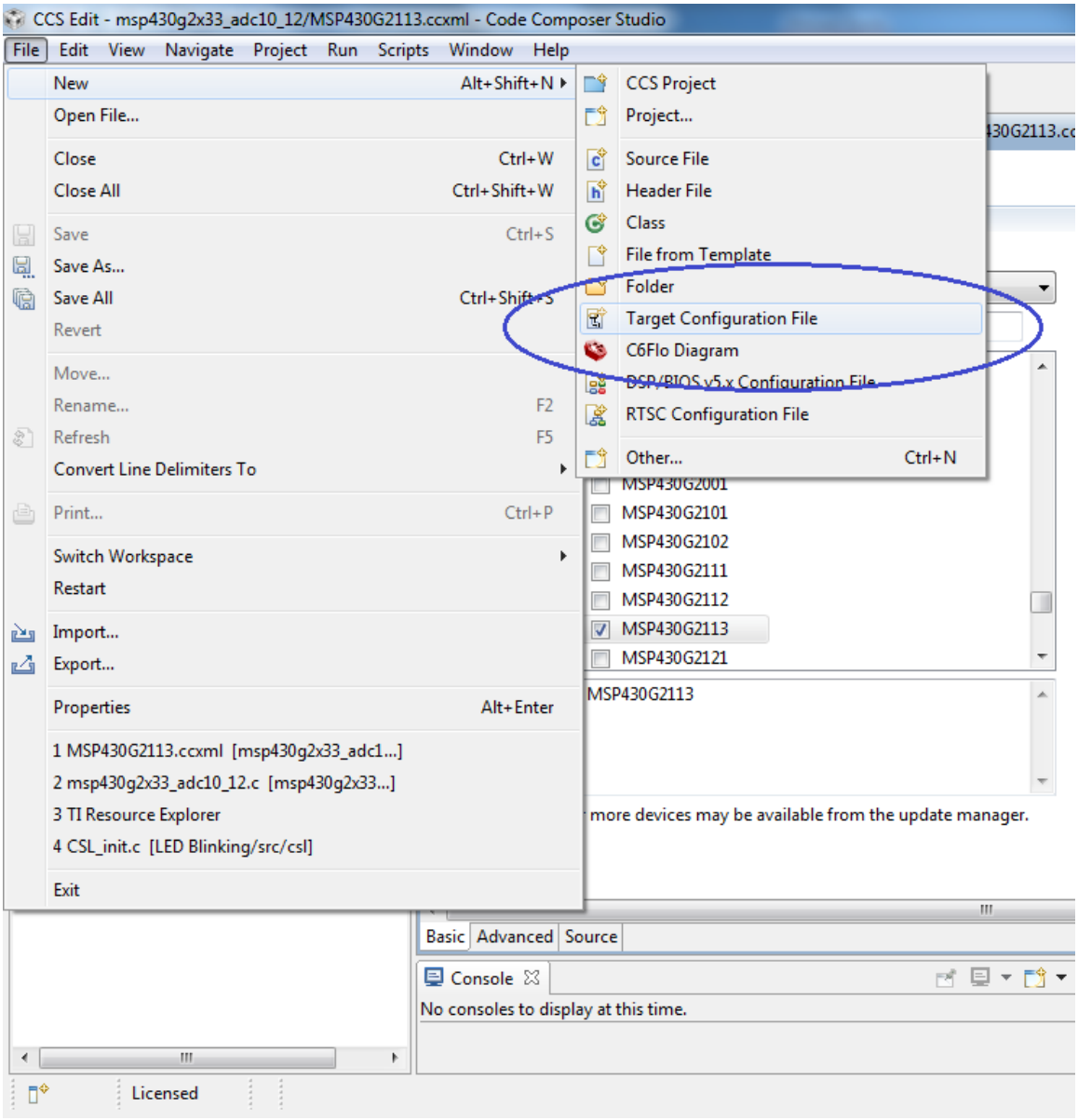


图 5.7 新建配置文件

在新建完成之后，单击“view”→“target configuration”查看新建的配置文件，如图所示在 target configuration 标签页中看到有一个 user defined 文件夹，下面包含用户自定义的目标配置文件。双击配置文件在窗口中打开，这样用户可以进行定义和配置，结束后点击“save”按键保存配置。

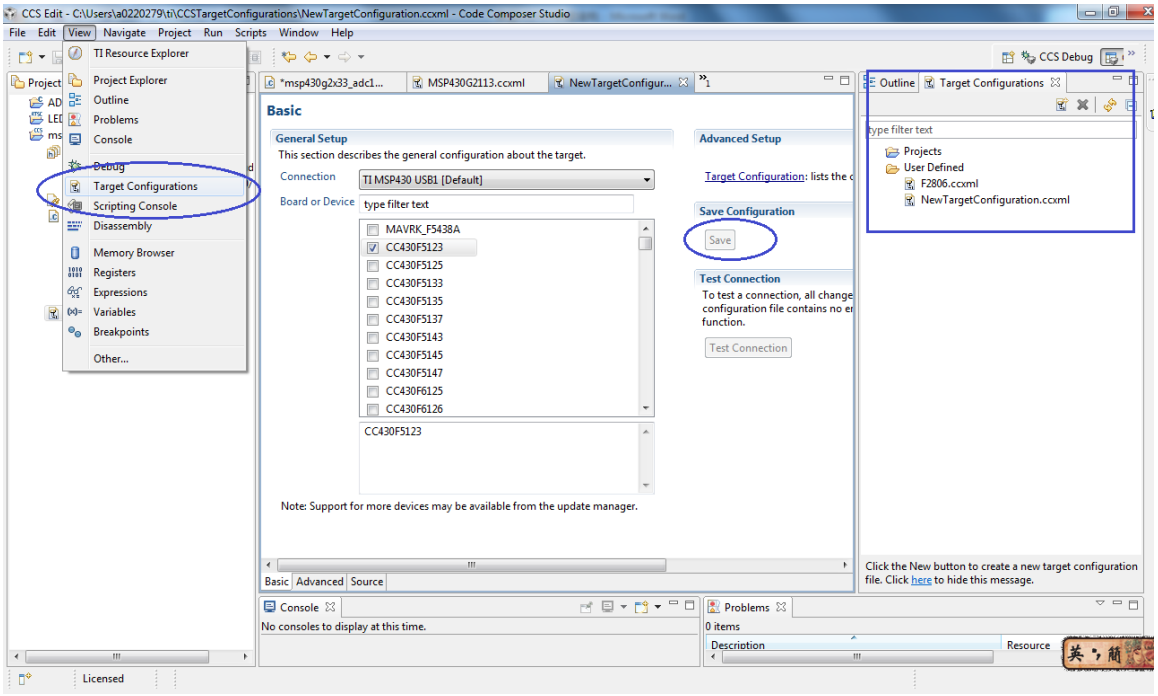


图 5.8 查看配置文件

以上完成了目标配置文件的新建和设置，接下来需要做一个连接的操作。在需要使用的配置文件上右击可以看到两个选项：“set as default”和“link file to project”，这两个选项都可以实现配置文件的使用，前者表明在默认情况下都将使用该配置文件，而后者则可以将该目标配置文件与特定的工程连接。用户可以根据实际的需求选择合适的方式。

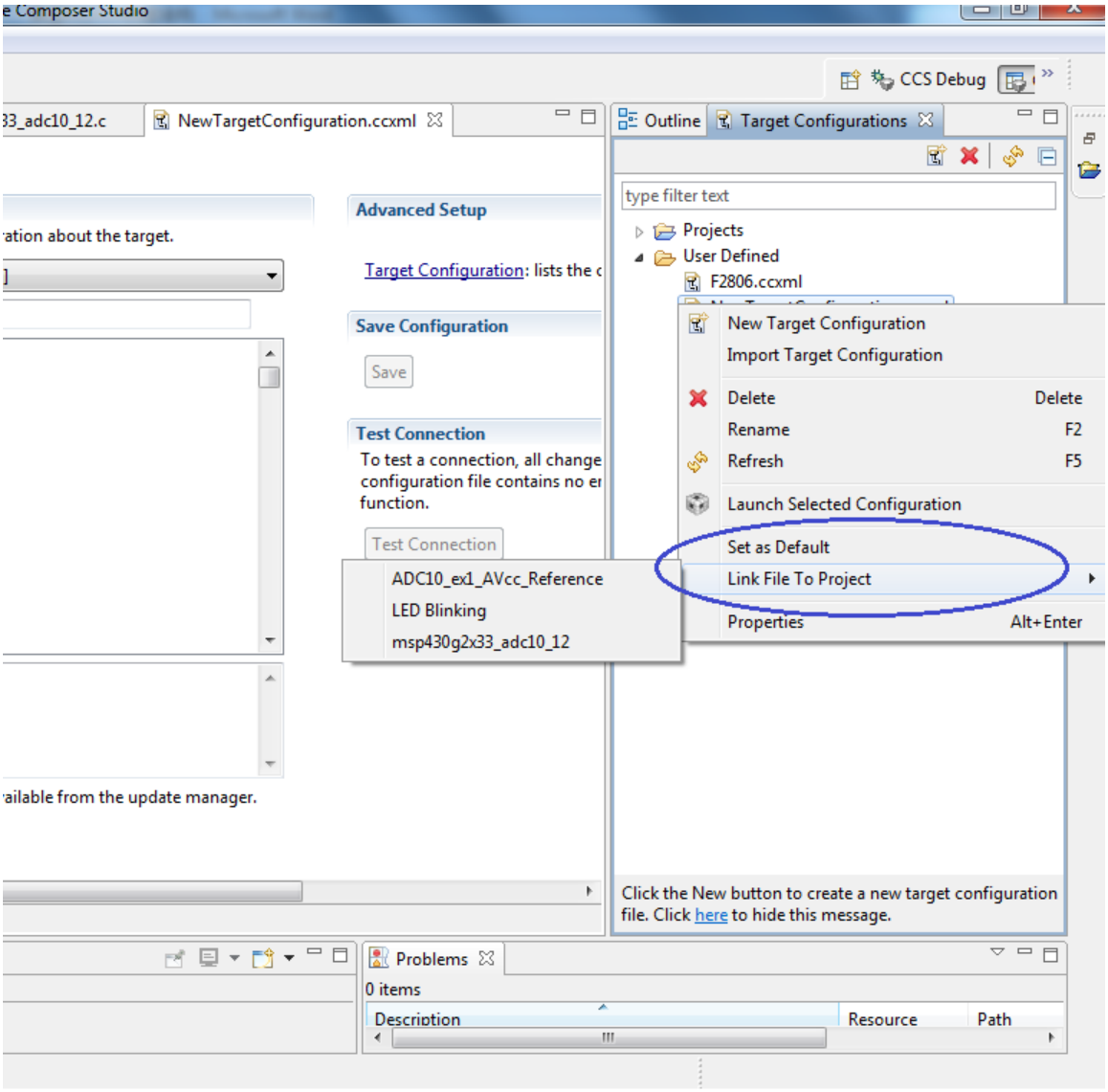

e Composer Studio
CCS Debug
33_adc10_12.c
NewTargetConfiguration.ccxml
Outline
Target Configurations
type filter text
Projects
User Defined
F2806.ccxml
ration about the target.
Advanced Setup
Target Configuration: lists the c
Save Configuration
Save
Test Connection
To test a connection, all change
configuration file contains no er
function.
Test Connection
New Target Configuration
Import Target Configuration
Delete
Delete
Rename
F2
Refresh
F5
Launch Selected Configuration
Set as Default
Link File To Project
Properties
Alt+Enter
ADC10_ex1_AVcc_Reference
LED Blinking
msp430g2x33_adc10_12
vailable from the update manager.
Click the New button to create a new target configuration
file. Click here to hide this message.
Problems
0 items
Description
Resource
Path

图 5.9 配置文件匹配

6. 典型 MSP430 例程结构

如前文所讲，进行 MSP430 开发所需的例程可以通过 TI 官网下载，430Ware 下载或者互联网搜索寻得。示例程序是我们进行板卡学习和项目开发必不可少的一个辅助工具，德州仪器提供的 MSP430 例程非常齐全，每个型号的 MSP430 都可以方便地找到相应的示例程序，对单独型号的 MSP430 又针对其每个外设的不同功能分别有相应的示例程序。此外，德州仪器提供的示例程序结构清晰明了，可以帮助用户了解该例程的内容，进行快速使用和开发。在本章节对德州仪器提供的典型 MSP430，尤其是用于 launchpad 的例程结构进行介绍和说明。

简单的例程包括一个.C 文件，即源代码（src），按照前文 CCS 新建工程，添加源代码的方法即可实现该例程的使用和调试。我们来一起看看这个 C 文件的结构。

```
/*******************************************************************************
*
 *                 MSP-EXP430G2-LaunchPad User Experience Application
 *
 * 1. Device starts up in LPM3 + blinking LED to indicate device is alive
 *    + Upon first button press, device transitions to application mode
 * 2. Application Mode
 *    + Continuously sample ADC Temp Sensor channel, compare result against
 *      initial value
 *    + Set PWM based on measured ADC offset: Red LED for positive offset,
Green
 *      LED for negative offset
 *    + Transmit temperature value via TimerA UART to PC
 *    + Button Press --> Calibrate using current temperature
 *                       Send character '°' via UART, notifying PC
*
 * Texas Instruments, Inc.                                  程序说明部分

*******************************************************************************/
```

```
#include  "msp430g2231.h"                                  头文件声明
```

```
#define     LED1                  BIT0                     常量定义
…
```

```
unsigned char applicationMode = APP_STANDBY_MODE;          变量声明
…
```

```
void InitializeLeds(void);                                 子函数声明
…
```

```
void main(void)
{                                                          主函数定义
…
}
```

```
void PreApplicationMode(void)
{                                                          子函数定义
…
}
…
```

```
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  …                                                        中断服务子程序定义
}

…
```

6.1.示例程序注释说明

在文件开头，一般会有一段注释的文字说明，有时我们会忽略这段文字，但实际上这段文字往往是帮助我们理解示例程序最好的切入点（TI 的版权声明除外）。如图所示的是 Launchpad 的 User experience application 的例程，在例程开始的一段文字对该程序实现的功能做了清晰的描述，包括在程序运行过程中可观测的现象描述。通过该段文字，我们可

以发现在启动后首先板上的 led 会闪烁，当按下板上按键后进入应用模式，采集温度，对温度变化进行检测，同时将温度通过 TimerA UART 传输至 PC。同时，我们看到后面有 release 的版本信息，不同版本做的修改动作，这些都对我们理解程序很有帮助。

```
/*******************************************************************************
 *                  MSP-EXP430G2-LaunchPad User Experience Application
 *
 * 1. Device starts up in LPM3 + blinking LED to indicate device is alive
 *    + Upon first button press, device transitions to application mode
 * 2. Application Mode
 *    + Continuously sample ADC Temp Sensor channel, compare result against
 *      initial value
 *    + Set PWM based on measured ADC offset: Red LED for positive offset, Green
 *      LED for negative offset
 *    + Transmit temperature value via TimerA UART to PC
 *    + Button Press --> Calibrate using current temperature
 *                       Send character '°' via UART, notifying PC
 *
 * Changes:
 * 1.1  + LED1 & LED2 labels changed so that Green LED(LED2) indicates sampled
 *        temperature colder than calibrated temperature and vice versa
 *        with Red LED (LED1).
 *      + Turn off peripheral function of TXD after transmitting byte to
 *        eliminate the extra glitch at the end of UART transmission
 * 1.0  Initial Release Version
 *
 * Texas Instruments, Inc.
 ******************************************************************************/
```

此外，可能还会看到如图所示的文字说明，这一般是针对某款 430 的某个外设应用的参考代码。在这里我们除了可以找到对 CPU 以及外设的一些配置信息之外，还会发现有一个框图（图中蓝框所示），该部分信息告诉我们实现本例程的全部功能，硬件电路上的连接要求。如图所示的例程功能是演示利用 G2xx3 的 USCI_A0 模块实现 3 线 SPI 通讯，本机作为主设备，那么在硬件上，SPI 的几个必要界限，如数据，时钟等必须按图示连接好，此外程序中还涉及到 LED 的操作，且该 LED 是与 P1.0 相连接的。

```
//******************************************************************************
//   MSP430G2xx3 Demo - USCI_A0, SPI 3-Wire Master Incremented Data
//
//   Description: SPI master talks to SPI slave using 3-wire mode. Incrementing
//   data is sent by the master starting at 0x01. Received data is expected to
//   be same as the previous transmission.  USCI RX ISR is used to handle
//   communication with the CPU, normally in LPM0. If high, P1.0 indicates
//   valid data reception.
//   ACLK = n/a, MCLK = SMCLK = DCO ~1.2MHz, BRCLK = SMCLK/2
//
//   Use with SPI Slave Data Echo code example. If slave is in debug mode, P3.6
//   slave reset signal conflicts with slave's JTAG; to work around, use IAR's
//   "Release JTAG on Go" on slave device.  If breakpoints are set in
//   slave RX ISR, master must stopped also to avoid overrunning slave
//   RXBUF.
//
//                   MSP430G2xx3
//                -----------------
//            /|\|              XIN|-
//             | |                 |
//             --|RST          XOUT|-
//               |                 |
//               |             P1.2|-> Data Out (UCA0SIMO)
//               |                 |
//         LED <-|P1.0         P1.1|<- Data In (UCA0SOMI)
//               |                 |
// Slave reset <-|P1.5         P1.4|-> Serial Clock Out (UCA0CLK)
//
//
//   D. Dang
//   Texas Instruments Inc.
//   February 2011
//   Built with CCS Version 4.2.0 and IAR Embedded Workbench Version: 5.10
//******************************************************************************
```

由此可见，通过对例程开始的文件的阅读，我们可以无需阅读每行代码对程序实现的功能有一个直观的了解，从而对这段代码有了初步的认识，也可以根据此对代码的参考价值进行评估。此外，简洁明了的连接框图也使得我们清楚明白硬件连接的要求，帮助我们成功运行示例程序，实现程序功能。

6.2.声明头文件

在正式代码的开头必不可少的是下面的语句：

```
#include  "msp430g2553.h"
```

这段语句声明了该文件中使用的头文件。从使用的角度可以简单地将头文件分为两大类，一类是公共的标准化的头文件，例如 CCS 中提供的型号相关的头文件，像这里的”msp430g2553.h”，再如如果涉及科学计算需要包含的”math.h”，一般是 C 通用的；还有一类就是用户自定义的。在 CCS 中，第一类头文件中的 430 相关以及标准 C 头文件在创建工程时已经自动添加在工程中，我们可以在 Project Explorer 中工程目录下的 Includes 目录中找到包含的头文件目录，其中上方的为 430 器件相关的头文件，下方的为通用 C 头文件。在这里可以通过浏览的方式找到相应的头文件，打开对具体内容进行查看，但记住，不要轻易进行修改。

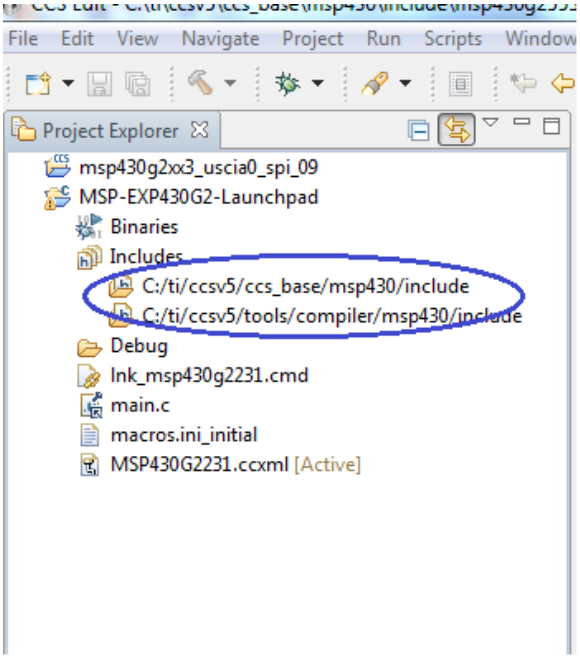


图 6.1 CCS 中默认包含头文件路径

所有 430 相关的头文件里面包含的都是 MSP430 的寄存器以及位的定义。通过这些定义，在对 430 寄存器配置时，不需要再去查找寄存器的位置，而是使用头文件中定义的可读性较强的文字进行程序的配置，这也是为什么 430 的源文件中大家会发现相当多的文字内容。关于这部分在后面会有详细的介绍。因为对 430 的编程很大程度上其实是对 CPU 或者外设寄存器的配置，所以在我们进行 430 编程时，记得在程序的开始一定要添加包含的头文件信息。此外，不同的编译环境对头文件的定义不尽相同，例如 CCS 和 IAR 的头文件名也许是一样，但里面具体的寄存器定义则略有差别，这时候就会出现编译出错的问题，所以在不同编译环境上进行程序的移植时一定要注意这个问题。

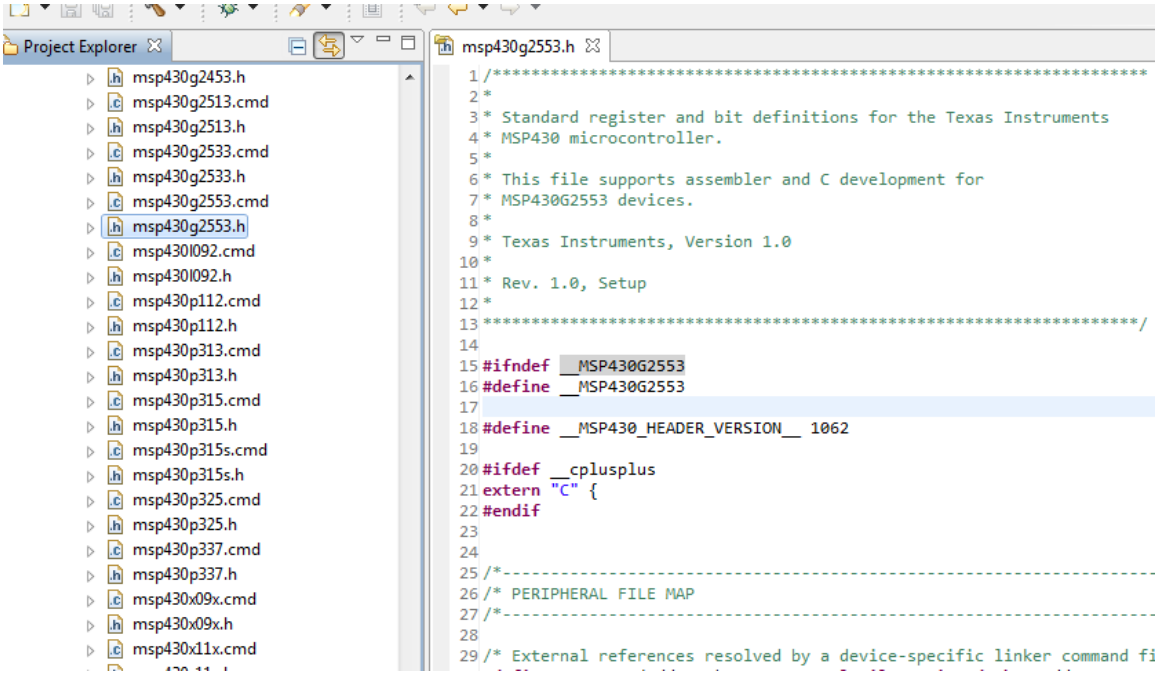


图 6.2 头文件内容

6.3.常量定义，变量声明及函数声明

6.3.1. 常量定义

在 C 文件中可以定义一些常量，使用”#define”进行定义。在以下几种情况下可以进行常量定义使编写的代码更加清晰易于修改。

- 与硬件连接相关

```
#define     LED1                BIT0
#define     LED2                BIT6
#define     LED_DIR             P1DIR
#define     LED_OUT             P1OUT
```

上面的语句，定义了 LED1 和 LED2，在板上分别与 P1.0 和 P1.6 相连，根据硬件连接做如图的配置。这样在程序中对涉及到 LED1 和 LED2 的 IO 端口配置可以使用 LED1，LED2，LED_DIR 和 LED_OUT。这样做一方面增强了程序的可读性（如果使用 P1DIR，P1OUT 在程序阅读过程中不能清楚表明这个 IO 口的功能）；另一方面，当硬件连接发生改变，如其中 LED1 不再和 P1.0 相连，而是和 P1.2 相连，用户需要做的修改仅仅是在该处将常量定义做修改，即”#define LED1 BIT2”即可，在后面的具体函数中无需做任何改变，大大减轻了因硬件改动带来的程序调整的工作量。当然，涉及板上的硬件端口定义比较多，用户可以自己定义一个头文件，如 board_hardware.h，在这里面进行板级常量的定义，然后在 C 文件的开始将这个.h 文件包含进去就可以了。

- 用户自定义常量

```
#define     TIMER_PWM_MODE        0
#define     TIMER_UART_MODE       1
#define     TIMER_PWM_PERIOD      2000
#define     TIMER_PWM_OFFSET      20

#define     TEMP_SAME             0
#define     TEMP_HOT              1
#define     TEMP_COLD             2

#define     TEMP_THRESHOLD        5
```

该部分和常规 C 编程中使用的一样。在进行编程过程中往往会碰到一些固定值的常量，可以在该处对其进行定义，同样这样做的好处增强了程序的可读性，对后续程序的修改也提供了便利。例如，如果程序中用到了圆周率π，可以在开始”#define PI 3.14”，这样在程序中如果用到π，则可以用 PI 代替，后续如果对精度进行调整，只需在 define 处对 3.14 进行修改即可。

6.3.2. 变量声明

对在程序中出现的变量，在使用前需要声明。在程序的开始对变量进行声明是一个比较好的编程习惯，可以方便对程序变量的查看和修改。对变量的命名，在符合 C 语言规范要求之外，为增加程序的可读性，其名称与功能最好能有对应关系。在这里也可以对声明的变量直接进行赋值操作。

```
unsigned char tempMode;
unsigned char calibrateUpdate = 0;
```

6.3.3. 函数声明

最后是子函数的声明。在有子函数的程序书写上一般说来有两种合法的方式：一种是像这样在 main 函数之前先对子函数进行声明，具体的定义则在 main 函数之后；还有一种则是将被调用的函数定义直接书写在程序的开始。建议使用前者，这样书写的程序规范，条理清晰。

```
void InitializeLeds(void);
void InitializeButton(void);
void PreApplicationMode(void);
```

6.4.主函数定义

在上述声明和定义之后就是主函数，作为程序的入口，每个工程必须也只能有一个 main 函数（注：在 CCSV5 中，为简化操作会在工程创建时自动新建一个 main.c 的文件，注意如果是自己导入源文件需将这个自动生成的文件删除）。在 MSP430 微控制器编程中，main 函数主要实现了包括时钟在内的外设初始化，程序运行的控制以及必要的数字信号处理（很多时候这部分会放在中断服务子程序中完成）。

如图所示为一个比较典型的 main 主函数组成。

- 局部变量声明

在开始可以声明一些主函数中用到的变量，需要注意的是在函数中声明的变量一般为局部变量，只能在该函数中使用。

- 功能性代码部分

作为调试程序，一般在开始会有关闭看门狗的一句语句。当然如果需要使用看门狗定时器功能时无需添加该语句。

接下来一般是外设的初始化语句。这些初始化语句可以直接在 main 函数中书写，也可以以子函数调用的方式实现。通常如果初始化的外设比较多，可以使用子函数的方式分类书写，如果初始化比较简单，例如只需对通用 IO 端口进行配置的话，可以直接写在 main 函数的开始。当然初始化的风格会因为程序员的编程习惯略有不同，如图示中的初始化则根据功能的不同，对 LED 和按键分别进行初始化。

其他的处理程序也会出现在 main 函数中。往往在主函数中会发现一个 Loop 循环，在循环中让处理器进入低功耗模式（如图中 Main Application Loop 所示）。这是一个比较典型的 430 低功耗控制方式，对所有外设初始化之后，将软件功能交由外设实现，从而让设备进入相应的低功耗模式，实现 MSP430

```
void main(void)
{
  unsigned int uartUpdateTimer = UART_UPDATE_INTERVAL;
  unsigned char i;

  WDTCTL = WDTPW + WDTHOLD;

  InitializeClocks();
  InitializeButton();
  InitializeLeds();

  PreApplicationMode();

   …

  /* Main Application Loop */
  while(1)
  {
     …
     __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled
     …
  }
}
```

局部变量声明

关看门狗

设备初始化

其他处理程序

6.5.子函数定义

在主函数之后可以对之前声明的子函数依次进行定义，中断服务子程序也在这里进行定义。

```
void PreApplicationMode(void)
{
…
}
…
```

子函数定义

```
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  …
}

…
```

中断服务子程序

如图所示为中断服务子程序的结构。需要用户自定义的有中断向量名称，这会在 430 器件相关的头文件中定义，在使用的时候可以去参考头文件中的具体定义或者直接在参考例程中查看；中断函数名，用户根据函数的功能进行函数名的自定义；以及具体的程序代码。

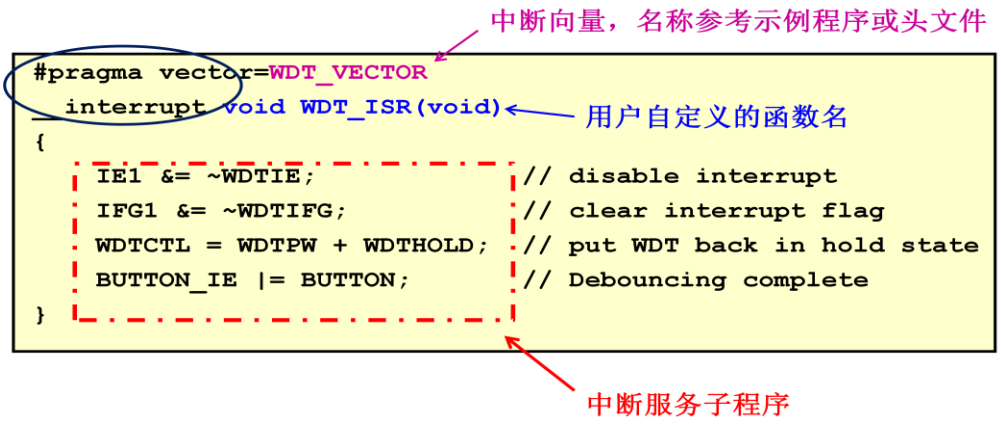


图 6.3 典型中断服务子程序组成

附录 A

开发实例

利用定时器在 Launchpad 上实现占空比为 50%的 PWM 波的输出，要求周期为 1s。

1. 在 CCS 中打开 430Ware，找到 Launchpad 中的 MSP430 型号——G2 系列的用户指南以及数据手册，作为编程开发中的参考资料。

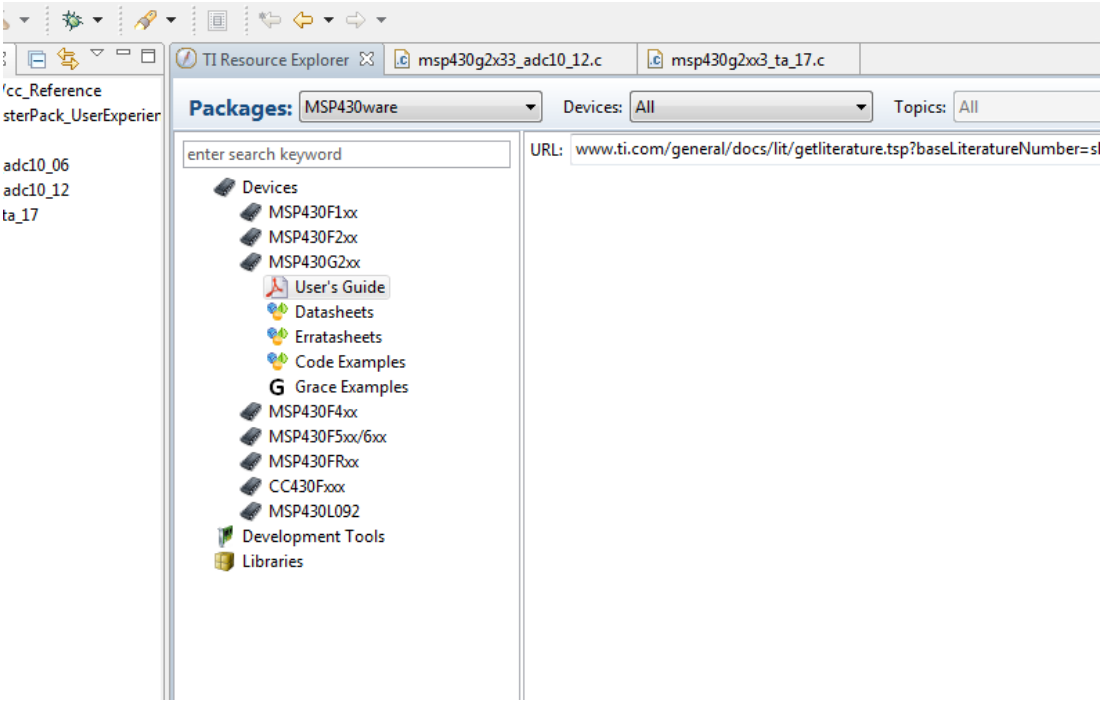


图 A.1 430Ware 中查看用户指南及数据手册

2. 可以通过多种不同的方法产生 PWM 波。通过对 MSP430 各模块的学习和了解后，发现采用定时器模块可以较为简单地实现占空比可调的 PWM 波形的输出。
3. 在 430Ware 中找到如图所示的示例程序。在关于定时器模块中专门列出了多达四种利用 TimerA 生成 PWM 的代码。
4. 根据实际需求选择最接近的一个示例代码，在此处不妨选用 TA 的第 17 个例程，在该例程中，定时器工作在 UP 模式下，采用外接 32KHz 晶振作为时钟源。

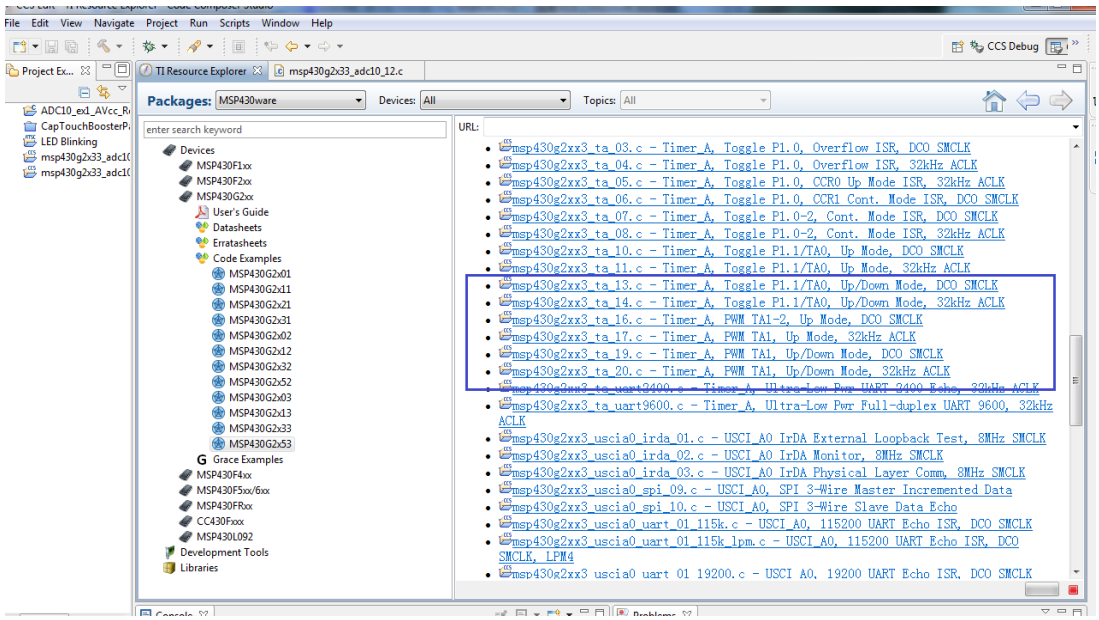


图 A.2 定时器示例代码

5. 导入工程之后，打开 C 文件，在注释部分对该例程进行了详细的解释和说明，可知当前示例程序为在 P1.2 端口输出占空比为 75%的 PWM 波（如图划线部分）。而在具体的程序部分，通过注释可以清楚看出图中框中代码明确了该 PWM 的周期和占空比的配置。
6. CCR0 决定了 PWM 的周期：当前示例程序中定时器的时钟为 32.768kHz，而 CCR0 的值为 256-1，所以当前输出 PWM 的周期为：

$$T = 2\times(CCR0+1)\times\frac{1}{f} = \frac{2\times256}{32768}s = 15.625ms$$

符合注释中对该例程的描述。根据上述公式，将 T=1s 代入，计算 CCR0 的值，并将值写入相应的代码处。

7. CCR1 与 CCR0 的比值则决定了占空比。在 CCR0 值的基础上计算出 CCR1 的值并写入相应的代码处。

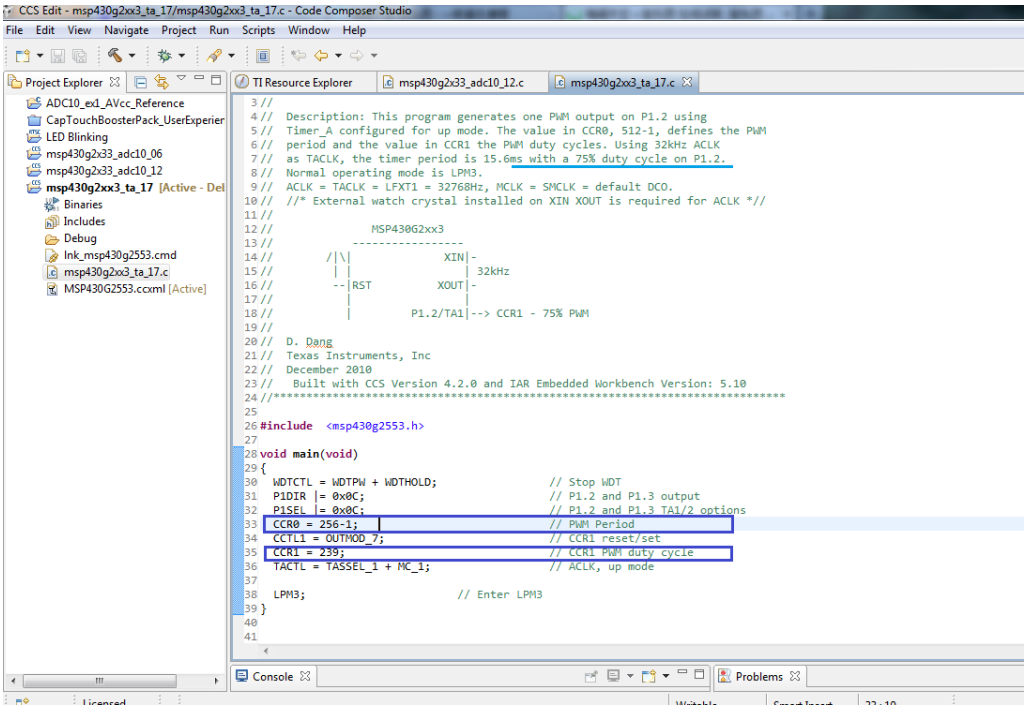


图 A.3 示例程序代码

8. 至此，已经完成了题目中要求输出的 PWM 波形。
9. 实际情况可能会比这个复杂很多，有时不一定能够恰好完全利用示例程序，而需要在示例程序的基础上对某些寄存器进行修改，这时候用户指南（User's Guide）就会显得很重要。
10. 如在本例中如果对定时器的时钟源另有要求，就需要对 TACTL 这个寄存器进行重新赋值，在用户指南中可以找到寄存器每个位的定义。如图所示为用户指南中定时器这一章节中对 TACTL 寄存器的定义，从中可以看到定时器 A 的时钟源选择是由 TASSELx 这一位决定的，在原程序中这里配置为 TASSEL_1，即选择定时器的时钟源为 ACLK。

12.3.1 TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted. 01 Up mode: the timer counts up to TACCR0. 10 Continuous mode: the timer counts up to 0FFFFh. 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

图 A.4 User's Guide 中对寄存器定义有详细的说明

11. 需要清楚的就是这里 TASSEL_1，之所以可以被编译器识别为 ACLK，即 01，是因为在头文件中对 TASSEL_1 进行了 define 的定义。可以通过双击 TASSEL_1 选中，然后右击，在弹出的菜单中选择“open declaration”查看原始的定义。

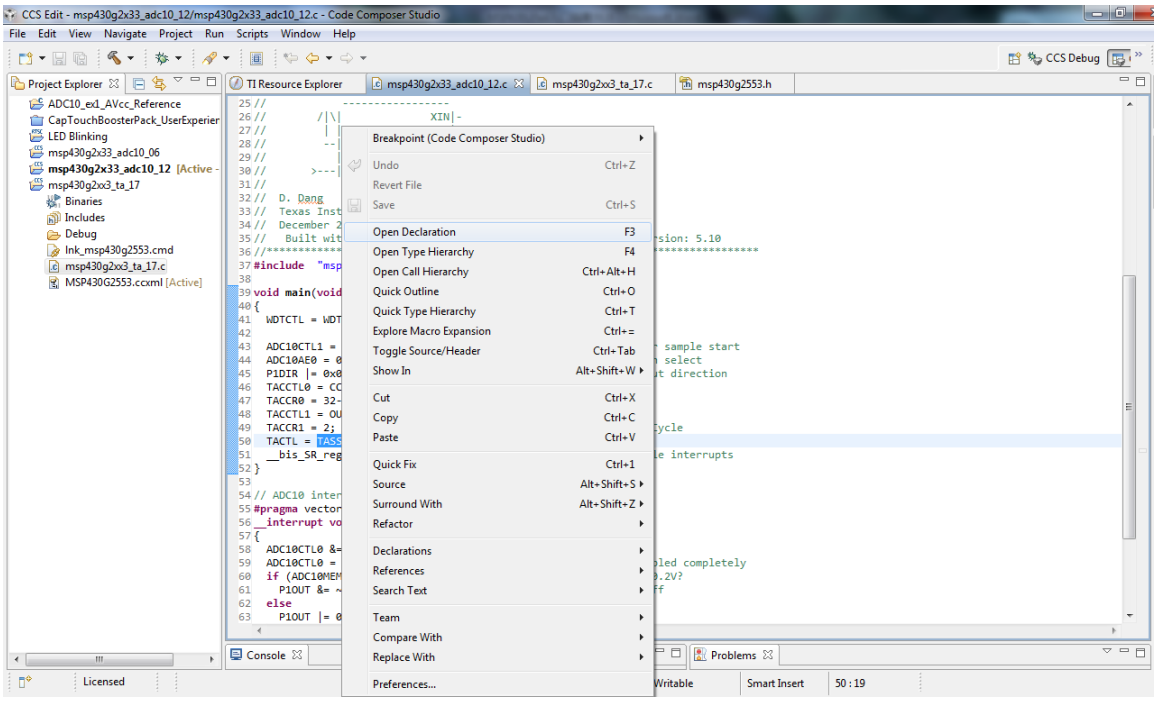


图 A.5 变量值查看

12. 通过变量值的追溯，可以看到在头文件中，对 TASSEL_1 进行了定义，同时可以看到头文件同样对 TASSEL_0，TASSEL_2 和 TASSEL_3 进行了定义，通过注释可以清楚地了解这三个值分别代表将定时器 A 的时钟源选择为 TACLK, SMCLK 和 INCLK。

```
516 #define ID_0                (0*0x40u)       /* Timer A input divider: 0 - /1 */
517 #define ID_1                (1*0x40u)       /* Timer A input divider: 1 - /2 */
518 #define ID_2                (2*0x40u)       /* Timer A input divider: 2 - /4 */
519 #define ID_3                (3*0x40u)       /* Timer A input divider: 3 - /8 */
520 #define TASSEL_0            (0*0x100u)      /* Timer A clock source select: 0 - TACLK */
521 #define TASSEL_1            (1*0x100u)      /* Timer A clock source select: 1 - ACLK  */
522 #define TASSEL_2            (2*0x100u)      /* Timer A clock source select: 2 - SMCLK */
523 #define TASSEL_3            (3*0x100u)      /* Timer A clock source select: 3 - INCLK */
524
525 #define CM1                 (0x8000)        /* Capture mode 1 */
526 #define CM0                 (0x4000)        /* Capture mode 0 */
527 #define CCIS1               (0x2000)        /* Capture input select 1 */
528 #define CCIS0               (0x1000)        /* Capture input select 0 */
529 #define SCS                 (0x0800)        /* Capture sychronize */
530 #define SCCI                (0x0400)        /* Latched capture signal (read) */
531 #define CAP                 (0x0100)        /* Capture mode: 1 /Compare mode : 0 */
532 #define OUTMOD2             (0x0080)        /* Output mode 2 */
533 #define OUTMOD1             (0x0040)        /* Output mode 1 */
```

图 A.6 头文件中定义

附录 B

FAQ

1. 使用 Launchpad 进行开发工作时，运行示例程序却发现外设不工作？

在 Launchpad 开发板中，默认的外接晶振并没有焊接，而通过阅读 User's Guide 可以了解到，芯片启动后默认的时钟分配为外设使用外接晶振为时钟源。在这种情况下，如果示例程序中恰好使用外接晶振为外设的时钟源，或者没有做特别的设定（时钟源使用默认配置，仍为外接晶振），则会表现为外设没有正常工作。可以通过焊接外接晶振来解决，或者在程序端更改外设的时钟源。

2. 在调试过程中断点单步调试查看局部变量，发现在 watch 窗口中没有观察到预期的结果。

在 CCS 新的版本中对局部变量的调试进行了优化，用户会发现，如果该定义了一个局部变量，如果在后续程序中没有使用到该变量，则该变量在 watch 窗口中显示的值是错误的。但实际上该局部变量的值在 memory 中是正确的，这个可以通过赋值的方法验证。所以如果需要实时查看变量的变化情况，尤其是该变量实际上没有其他用途时，建议将其定义为全局变量。

3. 如何通过引导加载（BSL）来对 MSP430 编程？

BSL 是另一种程序下载的方式，Launchpad 可以用于 BSL，具体方法可参阅以下应用笔记：

- MSP430 Programming via the Bootstrap Loader SLAU319B;
- Launchpad-Based MSP430 UART BSL Interface SLAA535A;
- Application of Bootstrap Loader in MSP430 with Flash Hardware and Software Proposal SLAA096;
- Features of the MSP430 Bootstrap Loader SLAA089.

上述应用笔记均可在 TI 官网下载（ www.ti.com.cn ）

4. MSP430 的 UART 无法达到 UART 控制寄存器所设置的预期的效果。

一般来说 MSP430 中每次 USART 的配置被重新设置后，必须复位才有效。在用户指南中有关于这个的说明，正确有效的配置需要通过设置 UCTL 寄存器中的 SWRST 位的设置/复位顺序来实现。在上电复位时，SWRST 位是默认置位的。如果上电复位后第一次通过配置控制寄存器来定义 USART 模块参数，那么，必须最后配置 UCTL 寄存器，这样，SWRST 就被复位。也就是说如果 UART 模块被重配置，SWRST 位的置位/复位顺序必须在重配置之后进行，以达到预期的效果。

5. MSP430 Flash 中的数据可以保持多长时间？

MSP430Flash 数据保持率至少是 100 年。 在数据手册的 JTAG、程序存储器和熔丝特性部分都可以查找到这个数据。

6. MSP430 Flash 的写入/擦除周期数的最大值可达到多少？

MSP430Flash 器件正常的写入/擦除周期是 100,000 次。可以在器件的数据手册的 JATG、程序存储器和熔丝特性部分找到这个数据的说明。

7. 在 MSP430 中，哪些端口引脚具有中断能力?

MSP430 的通用 IO 口具有中断的功能，这为系统设计提供了很大的便利，用户可以方便地扩展按键等其他外设。在 G2 系列中 P1 和 P2 都具有中断的功能，但需要注意的是并不是所有的 GPIO 都有中断功能的。一般来说在所有的端口中 P1 和 P2 具有中断能力。P1 和 P2 的每个中断可以独立地被使能且配置成上升沿或下降沿中断。所有的 P1 口中断源共享一个中断向量，所有的 P2 口中断源共享一个不同于 P1 口的中断向量。

8. 怎样降低 MSP430 的功耗?

降低功耗的最重要的途径是使用 MSP430 的时钟系统来最大限度地提高 MSP430 处于低功耗模式（建议为 LPM3）的时间。在实时钟和所有的中断都处于活跃状态的情况下，LPM3 的功耗低于 2uA。。

以下是其他的一些减小功耗的原则：

- 使用中断来唤醒处理器，控制程序流向。
- 外围模块仅当在需要时将其打开。
- 使用低功耗的集成外围模块来取代软件驱动。例如 Timer_A 和 Timer_B 可以自动产生 PWM 波、捕获外部定时而不占用 CPU 资源。
- 使用计算分支和快速查找表来取代标记的设置和大量的软件计算。
- 避免频繁的子程序和函数调用以降低软件开销。
- 对于较长的软件程序，最好用单周期 CPU 寄存器。
- 确保所有未使用的端口引脚是开路的，并且设置成输出。

9. MSP430 DCO 的频率会有抖动吗?

DCO 模块混有两个 DCO 频率，fDCO 和 fDCO+1，用以产生介于 fDCO 和 fDCO+1 之间的频率。这样就可得到带有所需的平均频率的调制时钟。调制的影响表现形式就是频率的抖动。本质上来说，这种调制将时钟能量扩散到一个宽带中，减小了电磁干扰(EMI)。

DCO 频率会随着温度和电压的变化而有所波动。请参阅器件数据手册关于 DCO 的具体说明。注意任何供电电源的不稳定也会造成 DCO 频率的抖动。

10. 在 MSP430 中可以使用嵌套中断吗?

如果在中断处理函数中 GIE 位被置位，那么中断嵌套将被激活。正常情况下，在中断服务程序中，GIE 位是被复位的。因此，如果你想在一个中断中需要嵌套另一个中断时，就必须在中断处理函数中将 GIE 位置位。可参阅用户指南的`System Resets, Interrupts, and Operating Modes`这个章节。

11. MSP430 的端口引脚中断时边沿有效还是电平有效？

端口引脚中断是边沿有效并且可以单独设置。用户可以为每一个引脚选择上升沿或下降沿中断。注意在 MSP430x3xx 器件中仅仅是有专门中断向量的 P0.0 和 P0.1 的中断标志会被自动清零。在其他具有中断能力的端口引脚上，中断标志不会自动清零，用户必须软件清零。对于任何一个需要服务的中断，除了那些独立的中断使能位，在状态寄存器中的全局中断使能位(GIE)也必须被置位。更多信息，请参阅用户指南中的数字 I/O 的相关章节。

12. 除了 32.768kHz 的晶振频率，MSP430 还可以与多大频率的晶振协同工作？

MSP430x3xx 器件被设计成专门使用 32KHz 的晶振，然后再从一个独立的内部数字控制振荡器（DCO）中产生一个内部的主时钟（MCLK）。MSP430x3xx 器件使用 FLL 电路使 MCLK 稳定到用户所需的值。

MSP430x1xx 和 MSP430x4xx 器件具有一个支持 32KHz 或者更高速度的晶振。有些 MSP430x1xx 和 MSP430x4xx 还有另一个晶振，这个晶振仅支持高速的晶振。这样，就允许在同一时刻有一个或两个晶振同时连接于器件上且在需要时可以只使用其中一个。

MSP430x1xx 和 MSP430x4xx 器件还具有可编程的内部 DCO，DCO 可以在独立于任何晶振的情况下产生高速时钟。类似于 MSP430x3xx，MSP430x4xx 器件也使用 FLL 来使 DCO 稳定多种不同于外部的 32KHz 的时钟。对于不同的时钟电路和器件的特点和性能，请参阅数据手册和用户指南。

13. MSP430 的静电效应值是多大？

MSP430 符合 TI 标准静电效应规格，并且达到静电效应测试的标准，包括外围模块和端口引脚。TI 使用标准的静电效应对 MSP430 器件进行测试 (Human Body Model=1.5KV, Charged Device Model=500V and Machine Model=200V)。

14. MSP430 器件工作和贮存温度的范围是什么？

请务必参阅器件详尽的数据手册来了解器件的工作温度范围。MSP430 器件专门设计工作的工业温度范围为- 40C 至+85℃。数据手册不推荐，不保证或描述芯片工作在这个温度范围外。请参阅设备详细的数据手册来了解可编程和不可编程的设备的贮存温度范围。

15. 如何使用 MSP430 进行数据存储器的扩展？

任何 MSP430 器件都没有外部数据和地址线。然而，扩展外部数据存储器可以使用 I/O。或者外部的 I2C 或串行存储器 EEPROM 可用于数据存储器扩展。

16. 在 MSP430 上有没有静电保护二极管？

每个引脚端都有静电保护二极管，可以被认为是连接到电源电压的钳位二极管。静电保护的等效电路可以认为是两个二极管共同连接到输入信号，而二极管的另一端一个连接到 Vcc，另一个连接到 Vss。二极管的最大绝对额定电流范围是+ / - 2mA。请参阅设备数据手册上的“最大绝对值范围”章节。

17. 在哪里可以找到 430 开发过程中的帮助？

一般来说 430 的开发必不可少的是：用户指南（user’s guide），包含了芯片各外设以及 CPU 模块的详细功能介绍以及使用方法；数据手册（datasheet），包含了芯片的物理特性，如电流电压等；以及参考代码（code examples），可以帮助我们最快地开始开发工作。所以对于已经有一定开发经验的使用者来说，大部分的困难都可以通过上述三类文档找到解答。对于初学者，可以尝试去阅读这些文档寻求帮助。这些内容都可以在 TI 官网上找到，或者直接在 430Ware 上找到。

另外网络上有不少不错的经验论坛，在上面也可以找到帮助，TI 有自己的官方论坛：www.deyisupport.com，可以在论坛上搜索或者求助。