SN8 C STUDIO

USER'S MANUAL

V 1.0

AMENDMENT HISTORY

	Date	Description
VER 1.0	14/02/2007	V1.0 first issue

Table of Content

第一部分 集成开发环境

1. 简介

1.1 系统概览

SN8 C Studio是一个基于Windows的软件开发平台。它包含有功能强大的编辑器和工程管理器，且方便易用。本文将讲述如何建立一个SN8 芯片开发应用的操作环境。
SN8 C STUDIO是针对SONIX 8-bit MCU设计的一个高性能且完整的开发环境。该系统由方便用户快速应用SONIX 8-bit MCU系列芯片所必需的软硬件工具合并组成。其中最主要的部分是SN8 C STUDIO，它提供了强大的工程编译和调试功能；其次是SN8 ICE，提供在线仿真。

就软件方面而言，SN8 C Studio为用户提供了良好的工作平台使用户在使用过程中更轻松，方便。此软件平台结合了所有软件工具如编辑器、汇编、（目标代码）连接器、库函数及带符号的调试器，是一个很好的基于Windows的用户环境。SN8-ICE硬件的所有基本功能对仿真器可用。本手册中包含了更多的SN8 C Studio的详细信息。

SONiX提供了版本更新的服务包，可以在SONiX的网站进行下载。

以下为SN8 C STUDIO的一些特性：

仿真
程序指令实时仿真

硬件
容易安装和使用
断点设置机制

软件
基于Windows的软件
源程序调试
多个源程序文件的工作平台（C语言源程序文件或汇编源程序文件）
包含开发、调试、仿真、并实现最终应用的所有工具

1.2 系统配置需求

- 具有 Pentium-II 以上或兼容的处理器的 PC
- Windows-98, Windows 2000, Windows XP 操作系统
- 内存最小为 16M
- 具有 40M 的空闲磁盘空间

2. 安装

2.1 硬件安装

将电源接头插入ICE的电源接口；
用扁平电缆将目标板与SN8-ICE连到一起；
用并口线将ICE连到主机上；
此时ICE上的LED应该被点亮，如果没有被点亮则说明有问题，请与经销商联系。

2.2 软件安装

从经销商处或SONiX公司网站都可以得到SN8 C Studio的软件安装包，解压缩后你可以得到一个如下图标所示的可执行安装文件：

Figure 2-1

双击安装文件，弹出一个对话框提醒启动 SN8 C Studio 的安装向导：

Figure 2-2

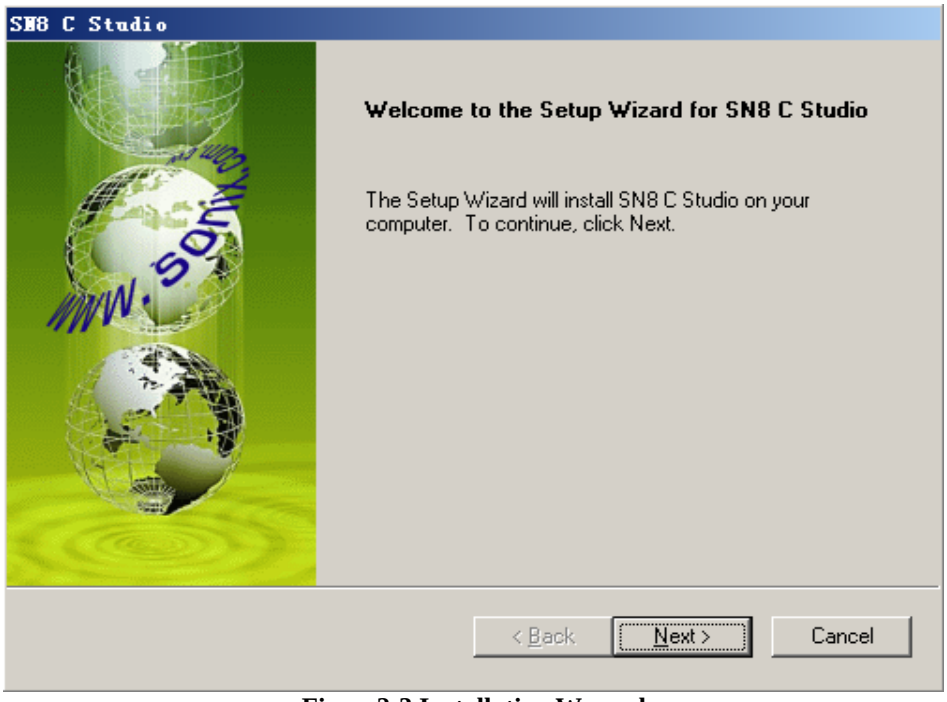


Figure2-3 Installation Wazard

按Next按钮继续安装，按Cancel按钮退出安装。

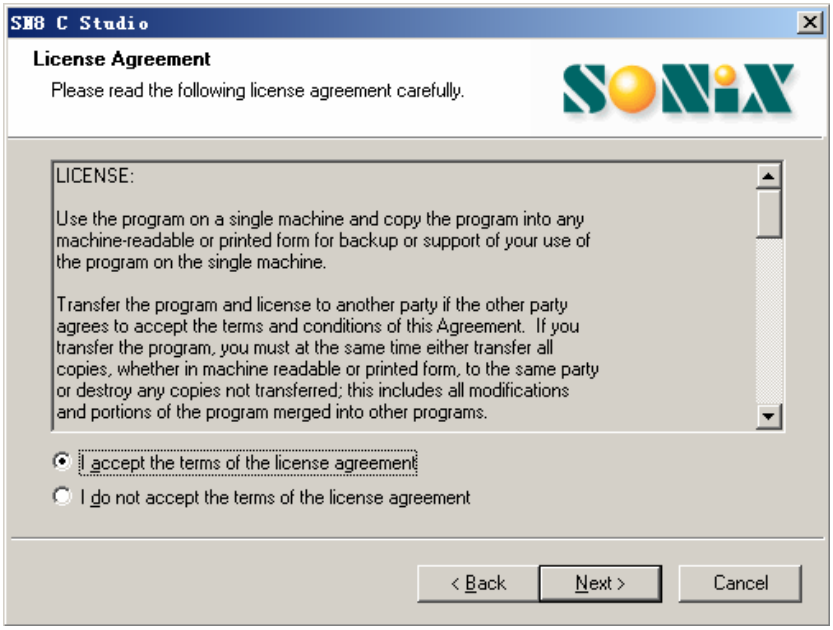


Figure2-4

请仔细阅读通行协议，你必须选择上面那条选项接受通行协议中的条款才能继续安装，若要继续安装请单击Next按钮。

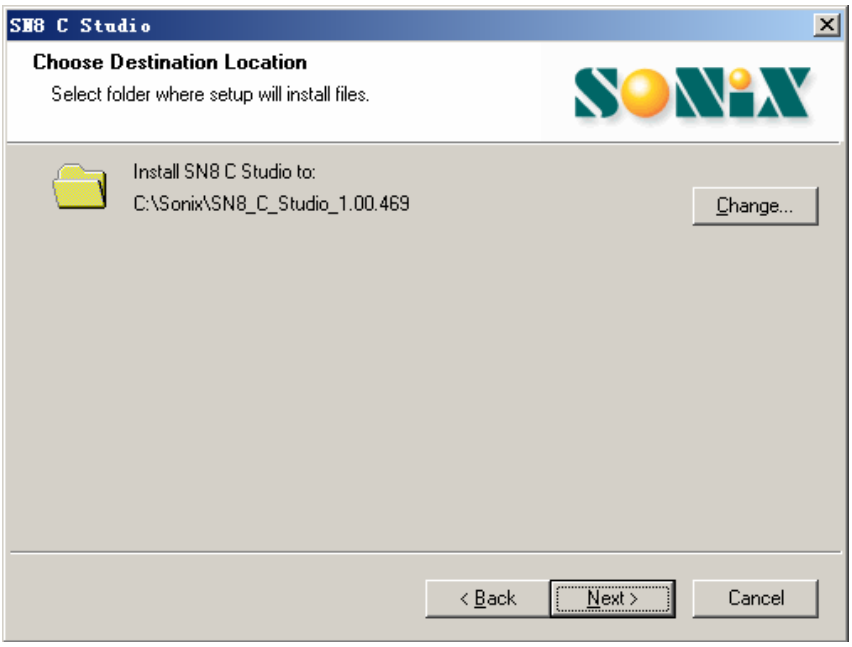


Figure 2-5

系统默认在系统安装盘下（这里是C：\）创建一个新的文件夹sonix\，软件将安装在新的文件夹下面。当然你也可以通过单击Change按钮来指定你想要安装此开发系统软件的路径，并单击Next按钮继续。

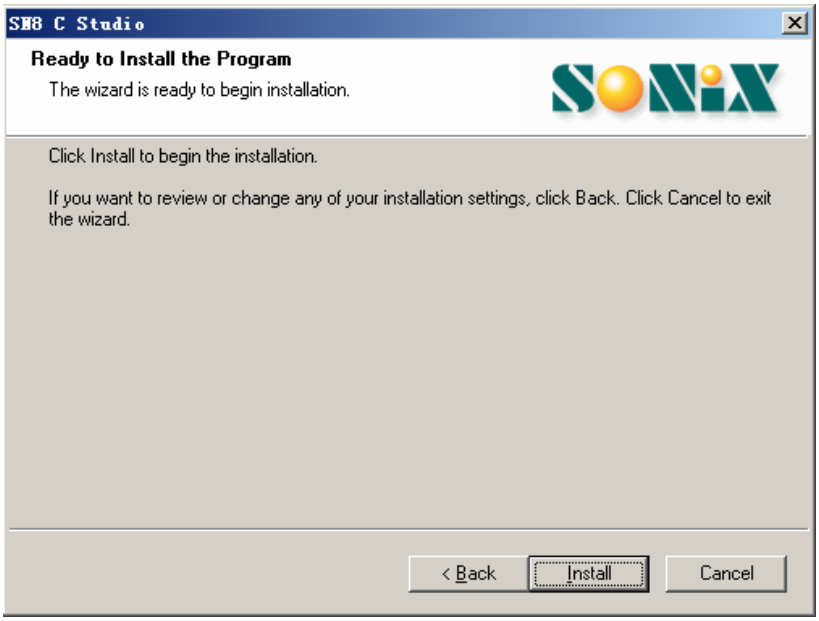


Figure 2-6

单击Install按钮将开始软件安装，软件会自动安装到系统当中。

在安装过程中，在对话框当中会显示出当前的安装状态。

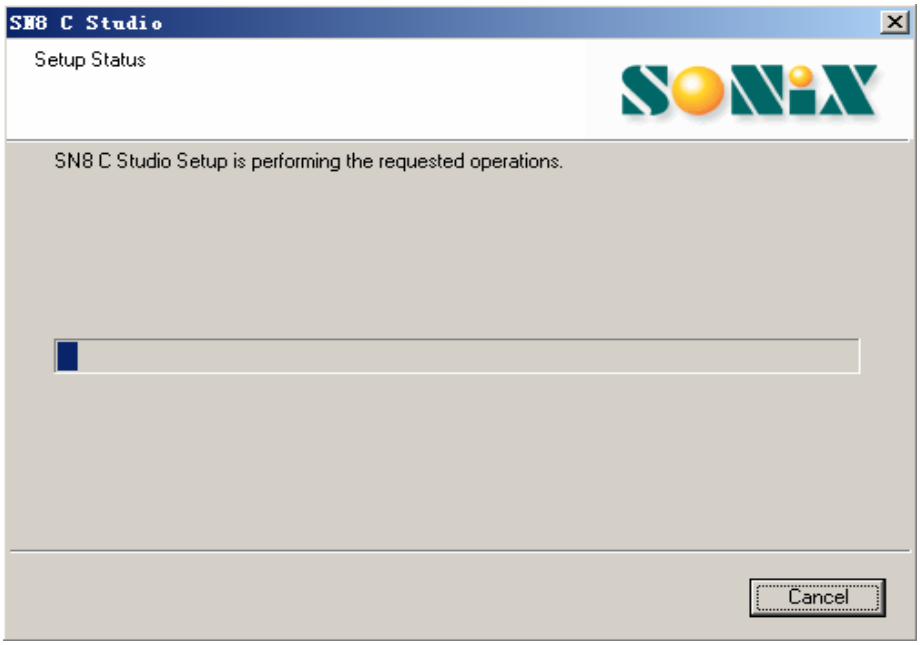


Figure 2-7

最后，SN8 C Studio安装成功后，请在复选框中选择是否将快捷方式建立在桌面上，单击Finish按钮结束安装。

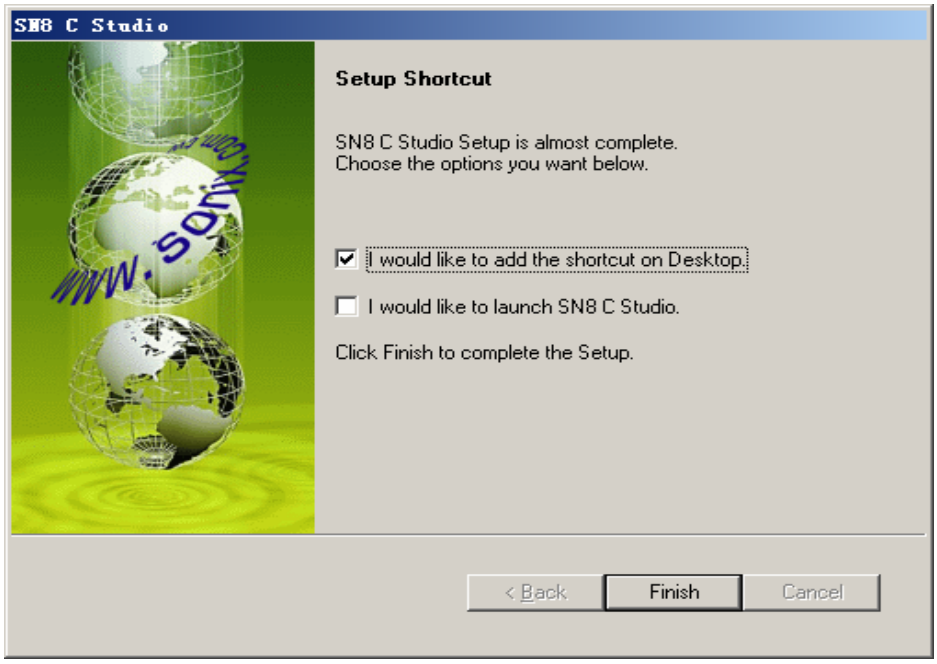


Figure 2-8

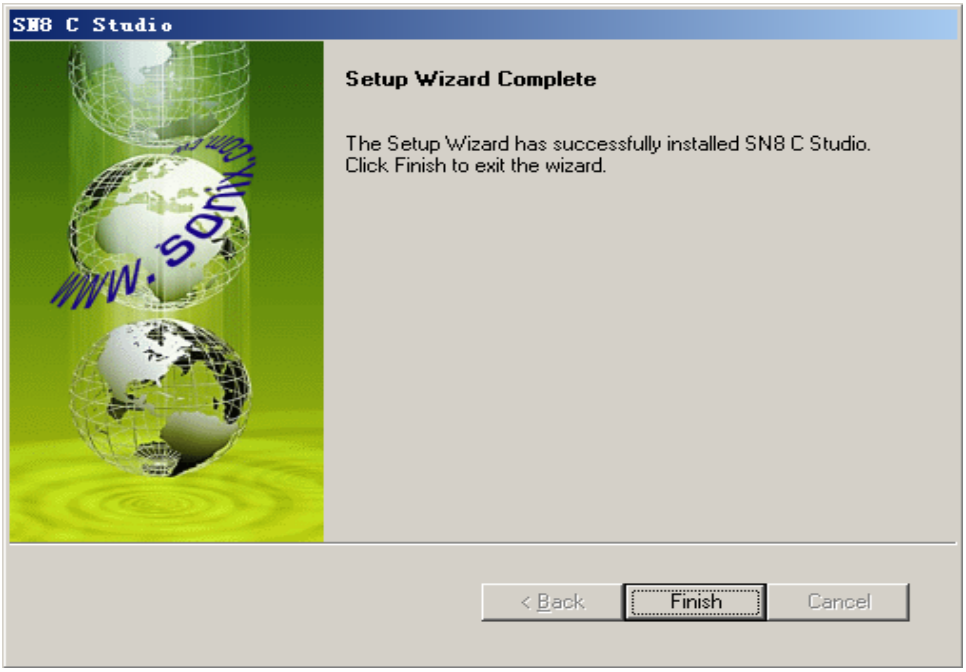


Figure 2-9

安装完成后将在你先前指定的安装目录下创建8个子目录：BIN、LIB、SAMPLE、C、CHIP、MANUAL、TEMPLATE和TOOLS。

3. 菜单与工具栏

菜单栏中包括文件、编辑、工程维护、设定开发工具选项、调试程序、窗口选择和窗口操作等菜单。应用工具栏按钮可以更加快捷地执行命令。快捷键也可以快捷地执行SN8 C STUDIO命令。

本节将逐一介绍SN8 C Studio的各项菜单。

3.1 File 菜单和工具栏

File菜单和工具栏提供SN8 C Studio所有相关文件的菜单项。为了和其他应用程序菜单一致，此菜单也包括一个参数选项和退出菜单选项。

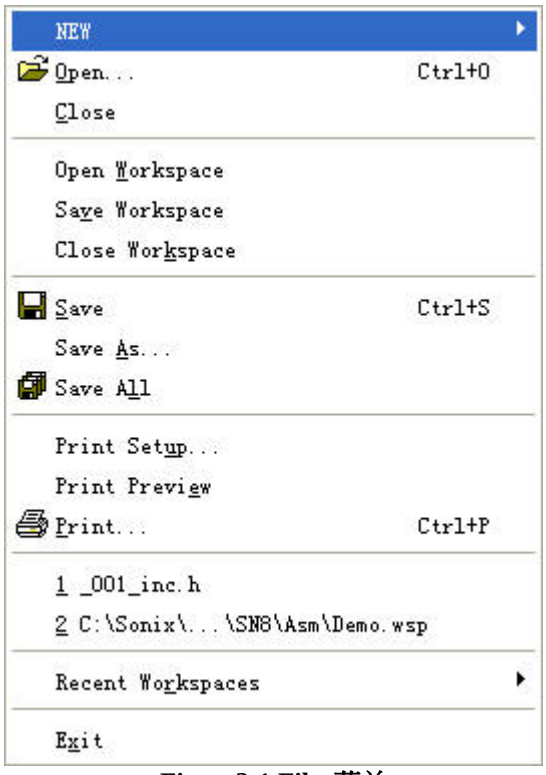


Figure3-1 File 菜单

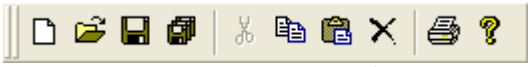


Figure 3-2 File 工具栏

新建

创建一个新文件。单击File\New菜单项，将出现一个新的对话框供用户选择新文件型。

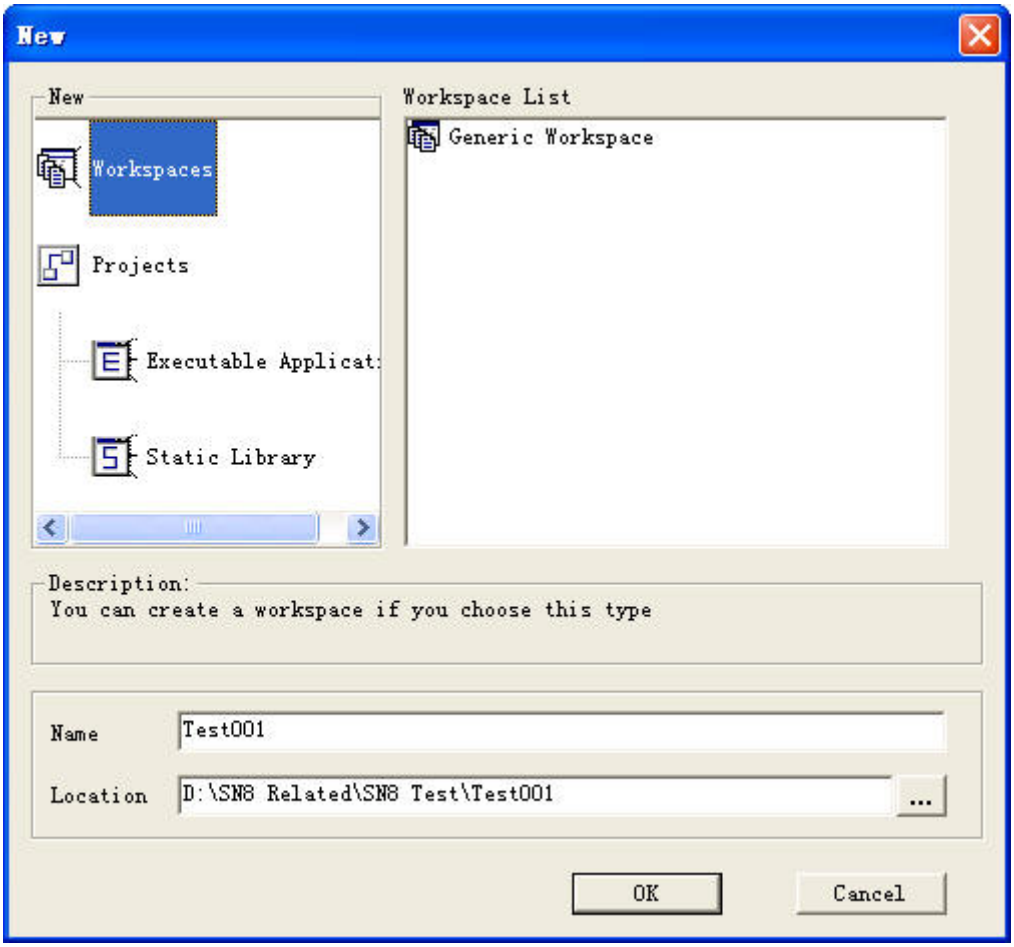
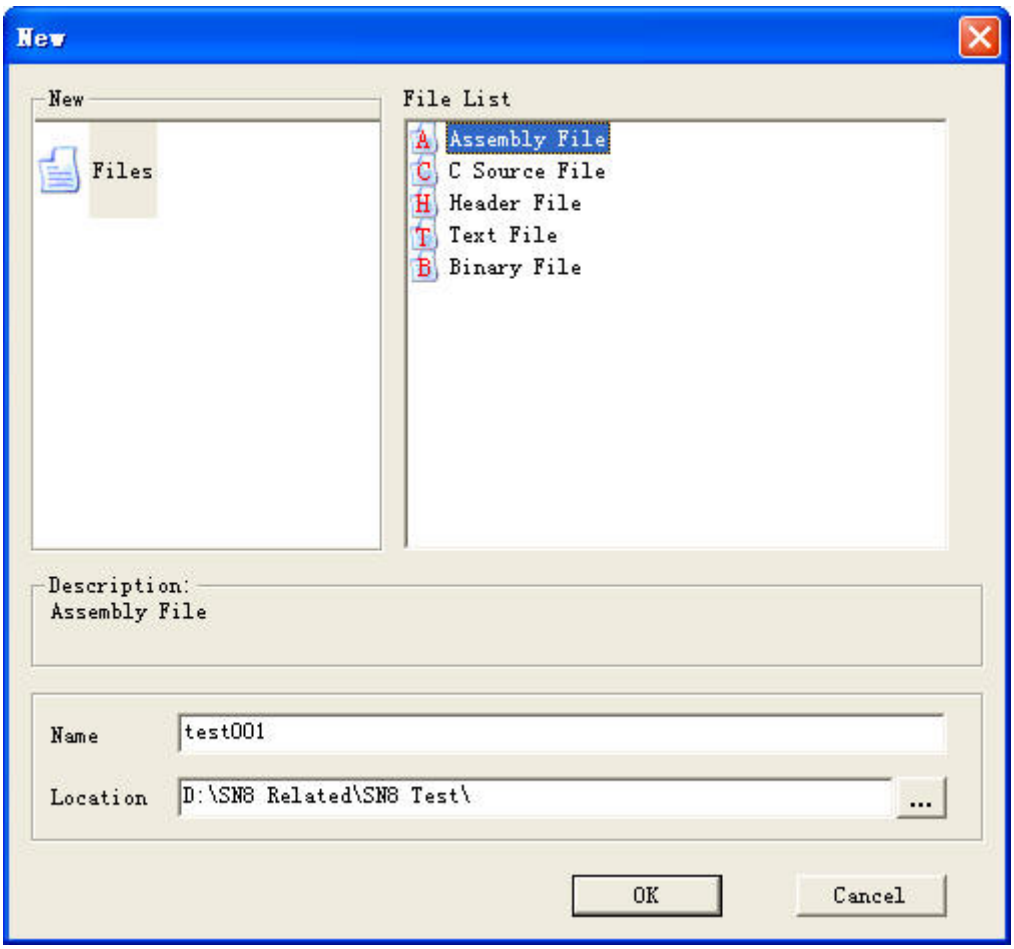


Figure 3-3 New 对话框



打开

打开文件对话框选择一个文件，在编辑器里直接打开。

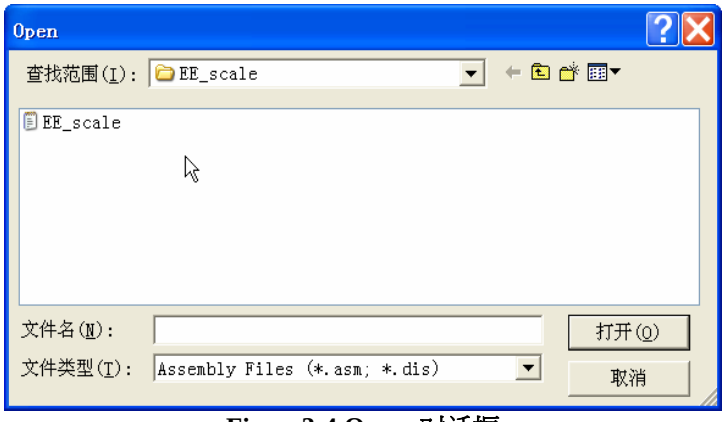


Figure3-4 Open 对话框

关闭
关闭当前正在显示的源文件。

打开工作区
打开一个已经存在的工作区。

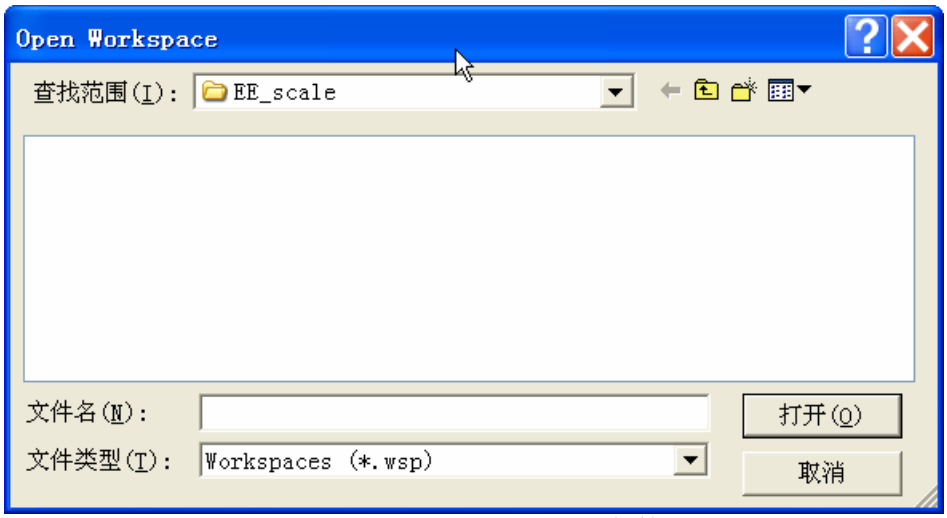


Figure3-5 Open workspace 对话框

保存工作区
保存当前工作区。

关闭工作区
关闭当前工作区。

保存
将选中窗口的数据写入选中的文件中。

全部保存
将所有的窗口数据改动保存到相应文件中。

另存为
打开一个文件保存对话框，允许键入新的文件名来保存当前活动文件，这样将不会对打开的文件重新命名，而是得到一个新的文件。

打印
显示打印编辑文件的对话框。

3.2 Edit 菜单和工具栏

Figure 3-6 Edit 菜单

Figure 3-7 Edit 工具栏

撤销
取消先前的编辑操作。

恢复
取消先前的撤销动作。

剪切
从文件中删除选中文本并放入剪切板中。

复制
将选中文本放入剪切板。

粘贴
将剪切板中的内容粘贴到当前插入点处。

删除
删除选中文本。

查找
在当前编辑缓冲器中搜索指定文字。

在文件中查找

在指定目录下查找指定文字。

替换
在当前编辑器中用目标文字替换指定的源文字。

全选
选中当前文件中所有文本。

设置
对当前编辑的显示等进行设置。

3.3 View 菜单和工具栏

View菜单提供以下一些控制窗口的命令：

Figure 3-8 View 菜单

工具栏
显示工具栏信息。工具栏中有5个下拉菜单，他们的功能分别如相应菜单项所列出的命令所示。把鼠标箭头放在其中的一个工具按钮处，相应功能的名字将显示在按钮旁。单击鼠标，执行此命令。

状态栏
显示状态栏信息。

工作区
打开或关闭工作区信息窗口。

消息
打开或关闭输出窗口。

调试
打开调试窗口。

3.4 Build 菜单和工具栏

Build菜单提供创建可执行文件等所需的命令，Build工具栏实现快速创建。

Figure 3-9 Build 工具栏

Figure 3-10 Build 菜单

编译当前文件
编译所执行工程中的当前文件。

创建工程
创建执行工程。

重建工程
删除所有输出文件，再生成工作工程。

全清
删除执行工程中的所有中间文件和输出文件。

全部创建
在工作区创建所有工程。

全部重新创建
在工作区重新创建所有工程。

清除全部工程
删除工作区中的所有工程。

停止
停止对当前工程的编译，生成或重新生成。

3.5 Debug 菜单和工具栏

在开发过程中修改和测试源程序是必不可少的过程。系统所提供的工具不仅可以方便调试，还可以缩短开发周期。其中包括的功能有单步执行，设置断点，自动单步执行和跟踪触发条件等等。

Figure 3-11 Debug 菜单

Figure 3-12 Debug 工具栏

调试（Begin Debug，F5）
开始调试当前工程。

退出调试（Exit Debug，Shift+F5）
停止调试。

运行（Run，F5）
运行可执行文件。

重启（Restart，Ctrl+ F5）
寄存器复位，准备运行可执行文件。
暂停（Pause，F5）

暂停后继续运行程序。

单步执行（Step into，F11）
Step Into命令每次只执行一条指令。即使遇到CALL子程序，PC也只是进入子程序停在第一条指令处。

单步跳过（Step Over，F10）
Step Over每次只执行一条指令。遇到CALL子程序，它仅执行CALL指令代替程序，并且PC指向CALL指令后的下一条指令。程序里的所有指令被执行且寄存器的内容和状态也可能随之改变。

单步跳出（Step Out，Shift+F11）
Step Out命令仅用于执行到子程序内部时。执行当前PC所指处到RET指令（包括RET指令）之间的所有指令，并且PC指向CALL指令后面的下一条指令。

运行到光标处（Run to Cursor，Ctrl+ F12）
运行到光标所在行停止。

自动单步执行（Animate step into）
自动单步运行程序，子程序中也一样。

自动单步跳过（Animate step out）
自动单步运行程序，但不进入子程序。

停止自动运行（Animate stop）
停止单步运行程序。

断点（breakpoints，Alt+ F9）
选择调试菜单中的断点命令可以在指定的地方设置断点。每设一个断点，可以通过单击Condition按钮设置跳过次数。

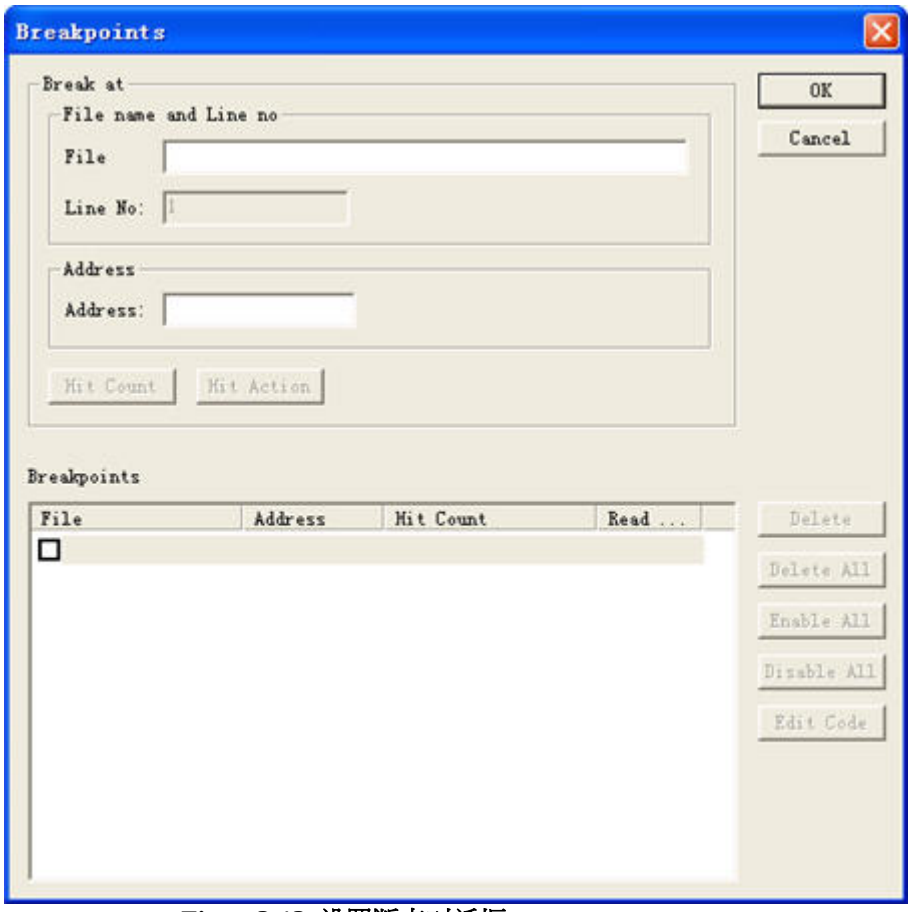


Figure3-13 设置断点对话框

3.6 Tool 菜单

用户设定

用户设定工具。可以设定显示的工具栏、用户工具及快捷键。

参数选择

提供混合设置和格式化设置

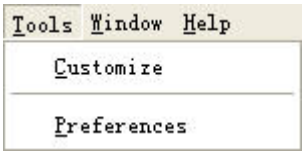


Figure 3-14 Tool 菜单

3.7 Window 菜单

窗口菜单提供如何排列当前打开的编辑窗的方法。

3.8 Help 菜单

帮助菜单提供关于本软件的信息及其用法。

4. 界面预览

本章介绍SN8 C Studio主窗口结构和使用方法。当没有工程下载到SN8 C Studio时，窗口中心的工程区是空白的，只有菜单、工具栏和状态栏是可用的，且控制工程的菜单和工具栏按钮不可用。下图4-1为典型的没有下载工程的SN8 C Studio视窗。

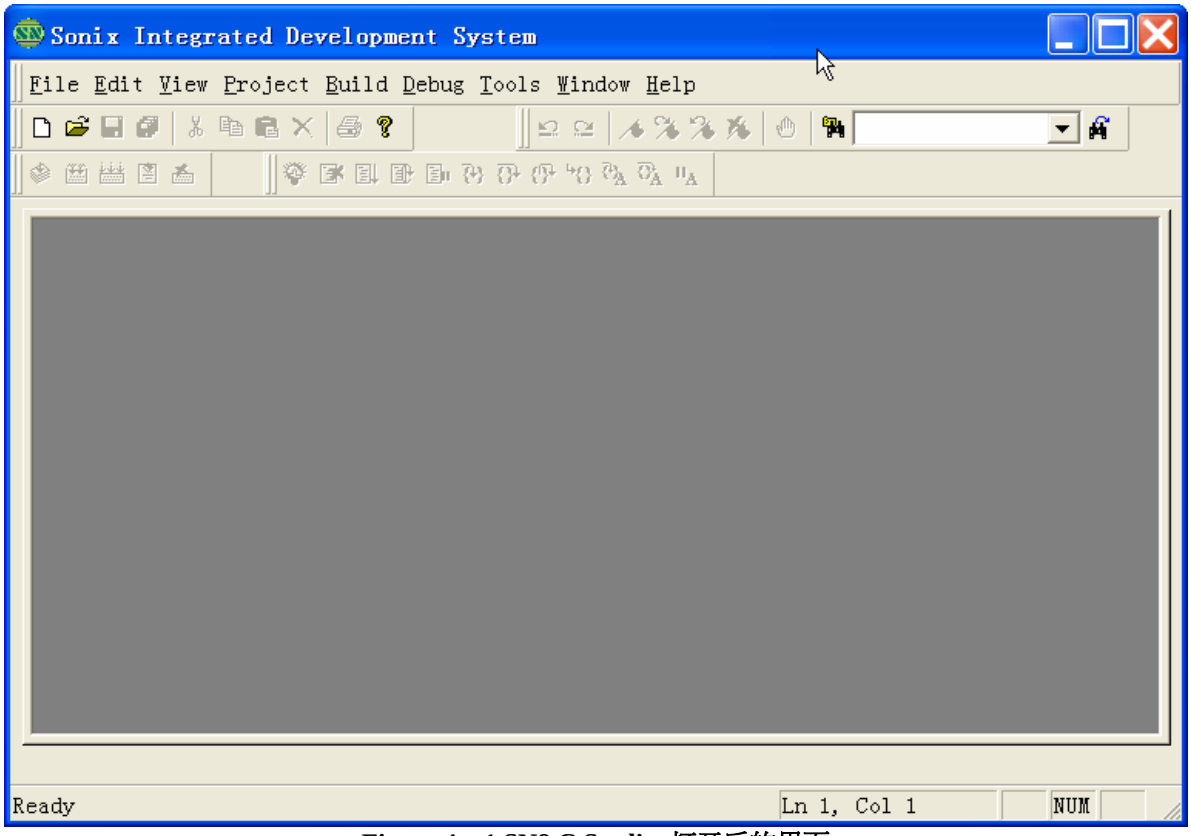


Figure 4 – 1 SN8 C Studio 打开后的界面

一个工程下载到工程区后将填入工程窗口。图4-2为Windows下运行的一个典型的SN8 C Studio窗口界面，有一个工程被打开。所有窗口给出了相应的信息。

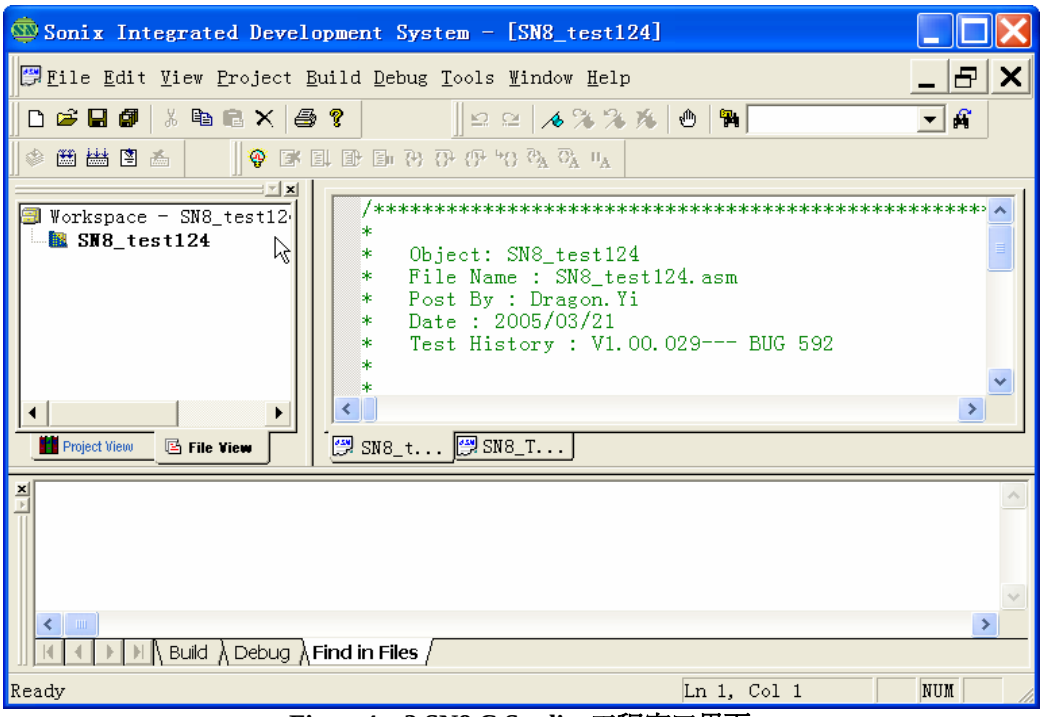


Figure4 – 2 SN8 C Studio 工程窗口界面

4.1 工作区窗口

如果没有下载任何工程，工作区窗口是空的。工作区窗口包括两个关于源文件的窗口：工程视窗和文件视窗。可以分别点击窗口底部它们的名字对两个窗口进行切换。它们的大小可以调整，也可以将它们隐藏。点击并拖动隔离条可以把它们从工作区分开。隐藏工程视窗：既可以通过点击视窗右上角的close (“x”)按钮，也可以通过取消视图菜单中对workspace的选中来实现。选择视图菜单中的workspace显示工程视窗。

4.1.1 Project View

工程视窗里列出与正在开发的应用程序相关的不同工程的工程名。工程视窗在工程工作区的左侧。

Figure 4 - 3 工程视图

右击工程名，将弹出菜单列出各个选项：

设置工作工程
将此工程设置为当前有效工程，或将其激活。

删除工程
从工作区列表中删除选中的工程。

保存工程
保存对选中工程的更改。

属性
显示所选工程的相关信息。

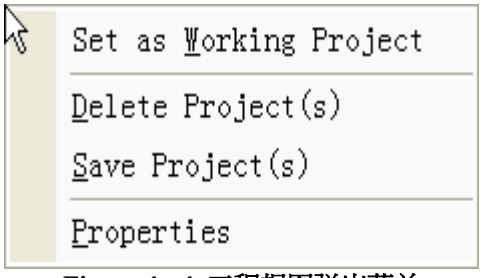


Figure 4 - 4 工程视图弹出菜单

4.1.2 File View

文件视图窗口中以目录树结构列出源文件和头文件的名字。文件夹在文件视窗下工作，SN8 C Studio不会在磁盘中创建物理文件夹。双击标签中的文件名就可以打开文件并显示其内容。目录结构的顶部（根部）为输出文件，顶部的下一级的文件夹中包含与工程相关的源文件和头文件。

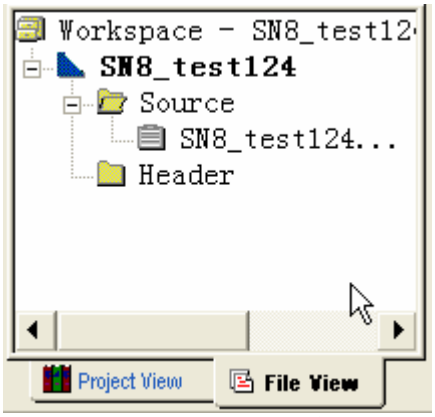


Figure 4-5 文件视图窗口

右击文件或文件夹，将弹出一个菜单列出各个选项。包括像添加文件到工程、新建文件、在编辑器中打开文件等功能。下面详细介绍这些选项。

- 工作区中弹出菜单

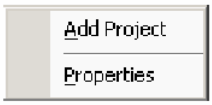


Figure 4-6工作区弹出菜单

添加工程
在现在的工作区下加入新的工程。

属性
显示工作区信息。

- 工程项弹出菜单

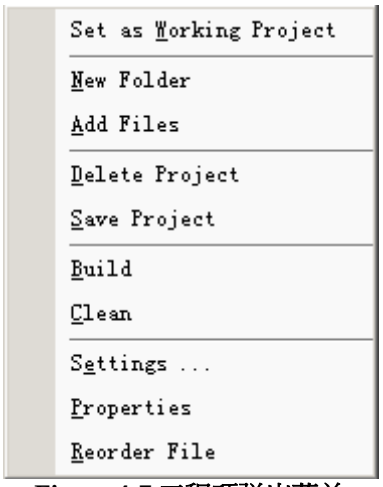


Figure 4-7 工程项弹出菜单

建立工作工程
将所选工程设为有效工程。

新建文件夹
创建一个新文件夹。

添加文件
将现有的文件添加到所选工程中并准确规类到文件夹。

删除工程
删除选中工程。

保存工程
保存对所选工程的更改。

生成
生成此工程。

清空
清空此工程。

设置
显示对话框设置工程。

属性
显示此工程的信息。

重排文件
改变工程中文件的排序。

4.2 输出窗口

输出窗口包括building消息窗口、debug消息窗口和find in files消息窗口。

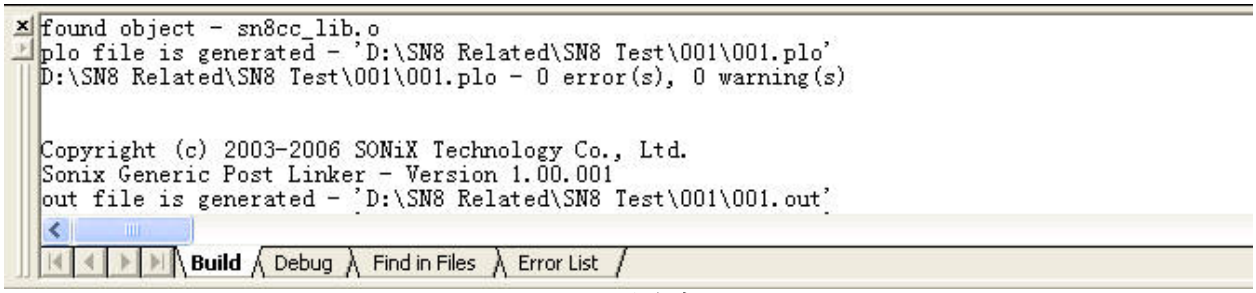


Figure 4-8 输出窗口

4.2.1 Build 消息窗口

显示在创建过程中的详细信息和结果。

当被编译连接的程序文件中存在错误，窗口就会显示出错或警告信息，提示问题产生的原因或位置以及简单提示问题处理的方法。双击此行信息，系统便引导指针自动查询并指向发生错误所在的行。

消息窗口中的信息详细地记录了创建过程中编译系统中各个部分所产生的信息，每一步骤都有显示版本号，和产生的文件名称和位置，详细阅读这些信息对Debug会有很大的帮助。

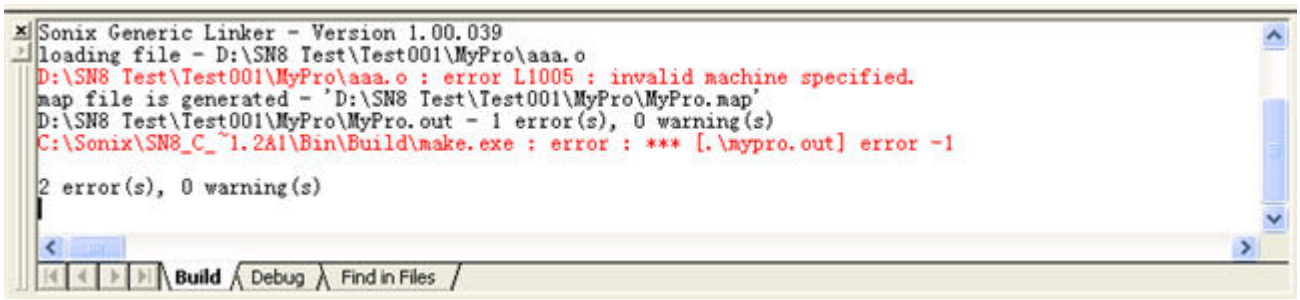


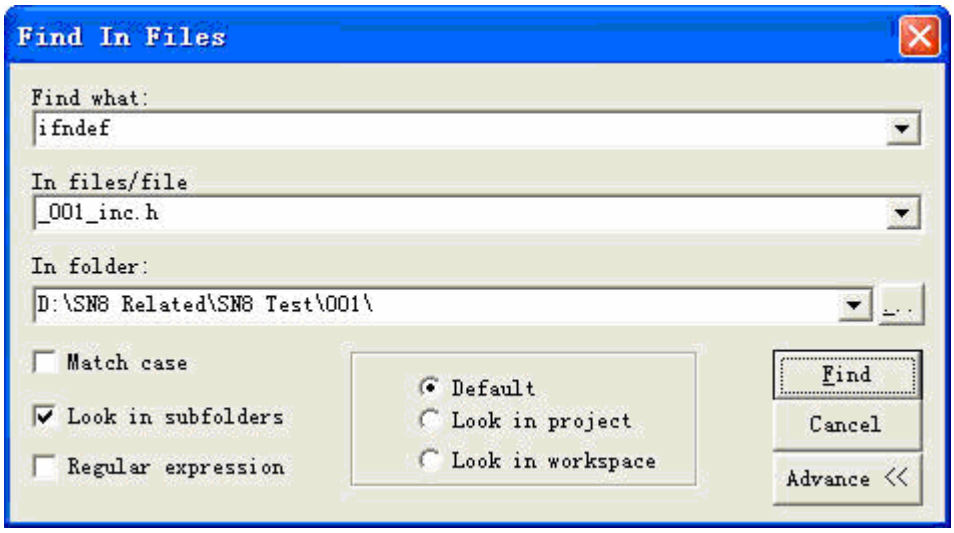
Figure 4-9 Build消息窗口

4.2.2 调试消息窗口

此窗口显示调试过程中的信息。

4.2.3 在文件中查找窗口

当执行了编辑菜单中的文件中查找命令后，此窗口将显示出对应的查找结果。此菜单可以在指定文件中找到需要的文字。



4.2.4 消息窗口弹出菜单

撤销
取消先前的编辑操作。

恢复
恢复先前被撤销的操作。

剪切
从文件中删除选中的文本并放入剪贴板中。

复制
将选中文本复制到剪贴板中。

粘贴
将剪贴板中的信息粘贴到当前插入点。

全选
选中当前文件中所有文本。

另存为
将内容保存到日志文件。

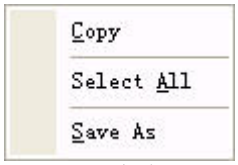


Figure 4-10 消息窗口弹出菜单

4.3 调试视图（Debug view）

成功编译后，就可以开始调试程序。调试窗口提供了如下图中所示的良好调试界面环境以方便调试。在调试模式下成功建立应用程序后，PC指针将指向源程序的第一行执行指令。此时，SN8 C STUDIO已经准备好接受并执行调试命令。

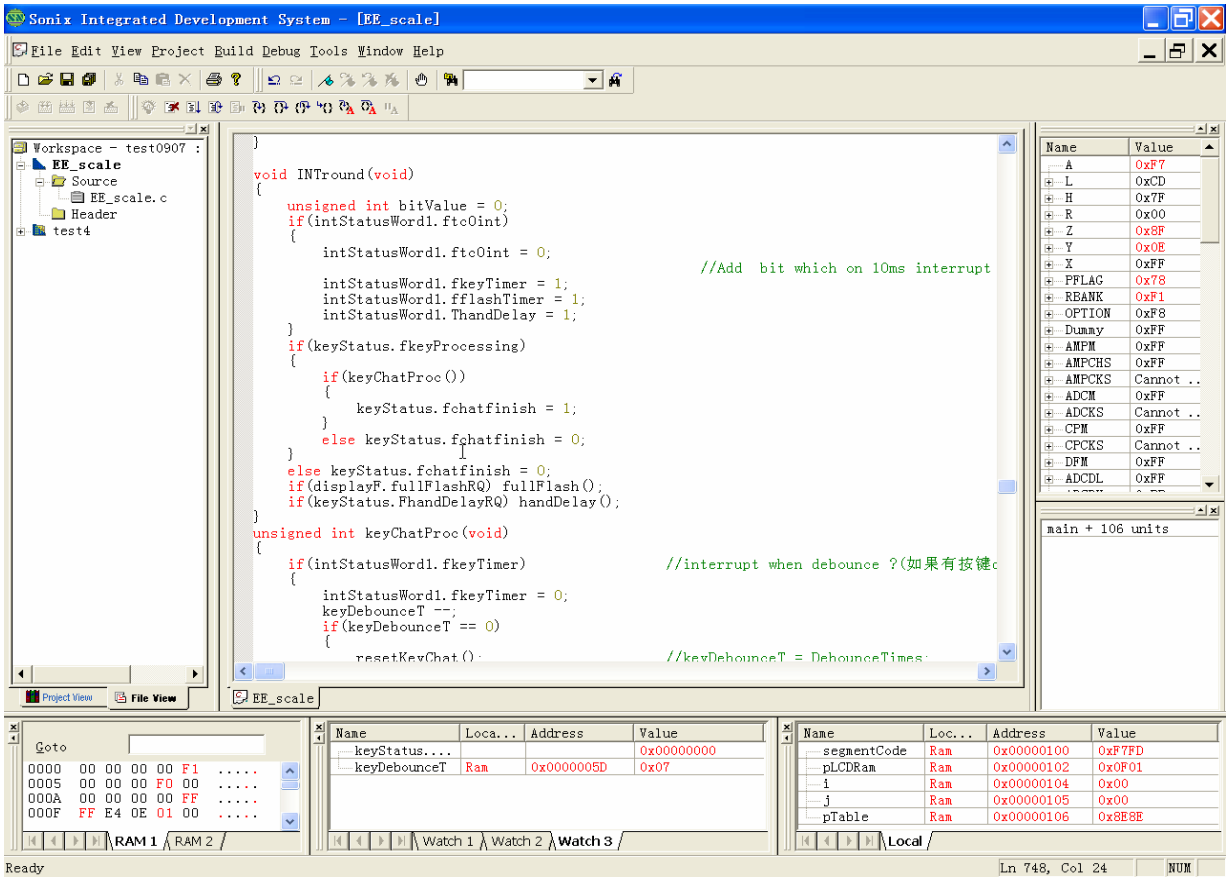


Figure 4-11 调试窗口

4.3.1 监视窗口（Watch Window）

观察窗口显示所选变量的值或观察表的值。观察窗口被分为3个部分来放置不同的标签，有足够的空间显示变量。这样当程序执行到断点处停下来或发生意外时观察窗口才

会被更新，且最后发生变化的值被加亮显示。选择并拖动一个变量到观察窗口就可以把变量加入观察窗，并且其变量名、位置、地址以及值都将被显示出来。在radix的弹出菜单中可以更改数据的基数。

Figure 4 -12 监视窗口和弹出菜单

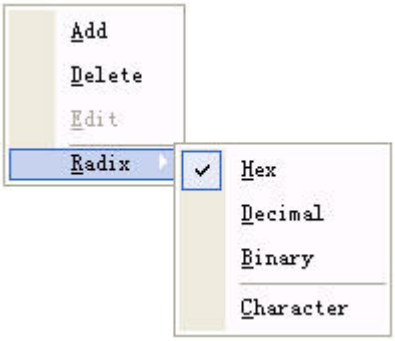


Figure 3 -12 弹出菜单

添加

添加新变量到监视（Watch）窗口。

删除

删除存在项。

编辑

编辑"名字"栏或"数值"栏中的值。

基数

根据不同的基数显示数值。

4.3.2 变量窗口（Variable Window）

变量窗口将自动显示当前正在使用的局部变量的值。这在C源程序中是十分有意义的。显示基数可以在弹出的菜单中更改。

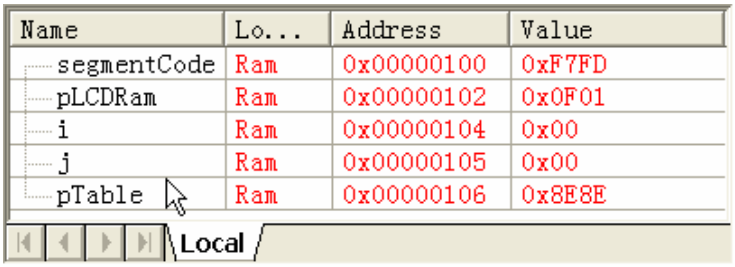


Figure 4 -13 变量窗口

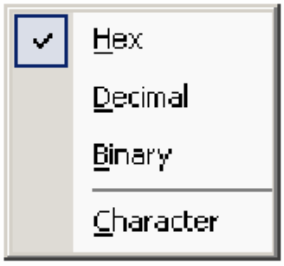


Figure 4 -14 弹出菜单

十六进制
以十六进制形式显示。

十进制
以十进制形式显示。

二进制
以二进制形式显示。

特征
如果是整型数据则显示其特性。

4.3.3 内存跟踪显示窗口（Memory Window）

内存窗中显示程序数据存储器空间的内容。调试时可以直接修改RAM窗口的数据，所有数值以十六进制形式显示。可以直接从'Goto'准确的进入存储器空间地址（可以为十进制数或十六进制数。）

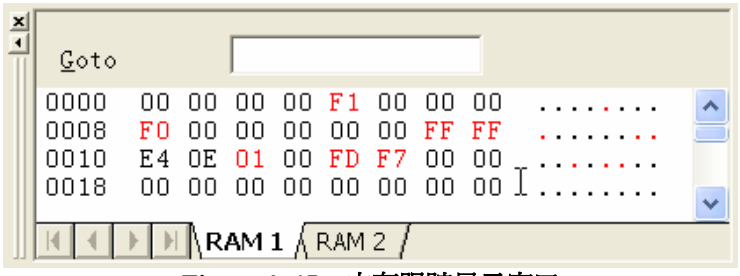


Figure 4 -15　内存跟踪显示窗口

Figure 4 -16 弹出菜单

ASCII码
显示ASCII码。

单位
以不同的单位显示数据。

Columns
根据不同的待选项显示。

4.3.4 寄存器显示窗口（Register Window）

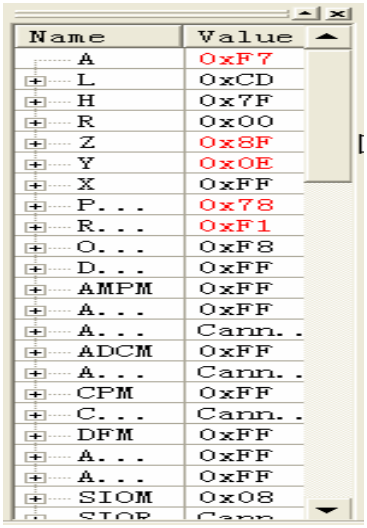


Figure 4-17　寄存器窗口

寄存器窗口显示出在工程中所选的不同芯片中定义的所有寄存器。上图4-17为一个寄存器窗口的举例。在调试阶段可以修改寄存器窗口中的内容，显示基数可以在弹出菜单中修改。

Figure 4 -16　弹出菜单

编辑

编辑寄存器的值。

基数

以不同的基数显示。

5. 工程设计

本章介绍用SN8 C Studio管理工程的主要步骤。同时也将给出正确设置系统选项的说明。

5.1 工程管理器（Project Manager）

5.1.1 建立一个新项目

SN8 C Studio包含一个工程管理器使创建一个新项目变得更加简单。通过下列步骤可创建新的工程：

第一步：

选择文件菜单或工具栏中的新建命令，将弹出一个对话框

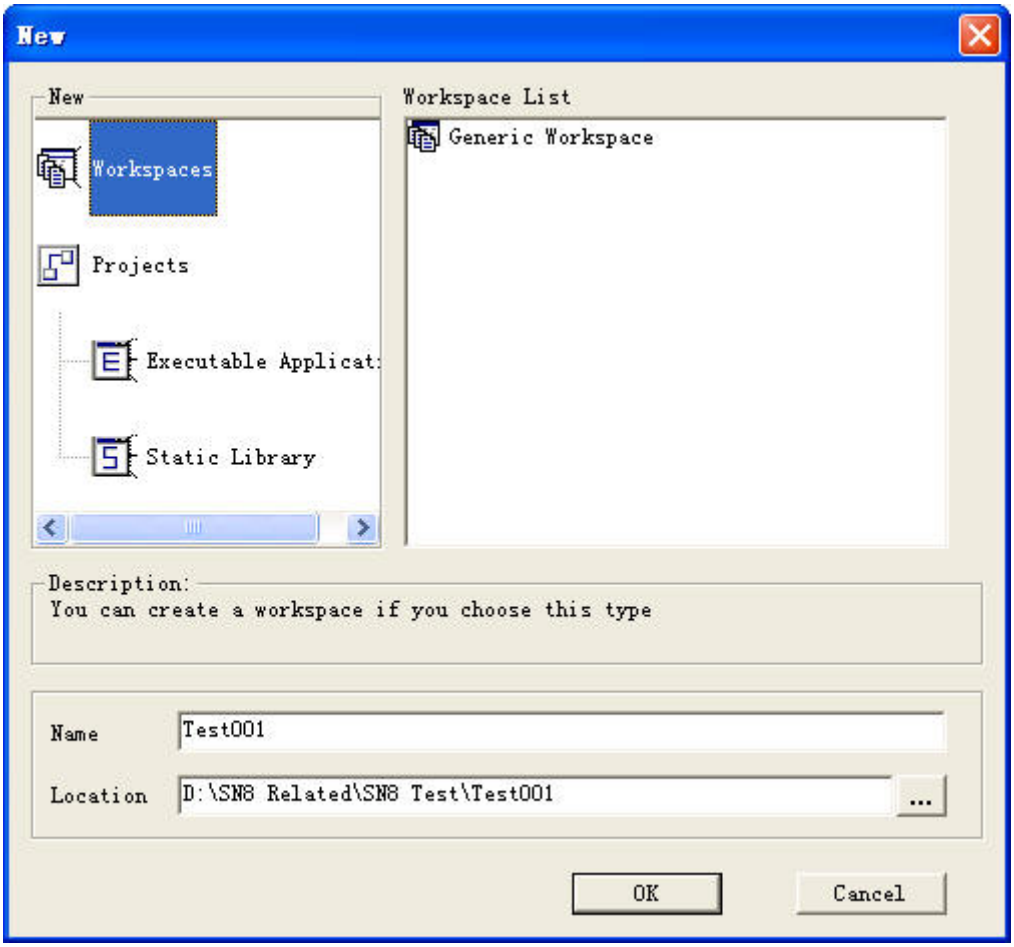


Figure5-1 建立一个新工作区

第二步：

单击Workspace按钮创建新的工作区文件，并对其命名，即成功创建了新的工作区。Studio将显示出工作区窗口和输出窗口；

第三步：

重复第一步；

第四步：

单击Project按钮创建一个工程文件，并选择工程中所用到的芯片类型。不要忘记对工程文件进行命名并正确存放。成功创建后此工程属于当前工作区。

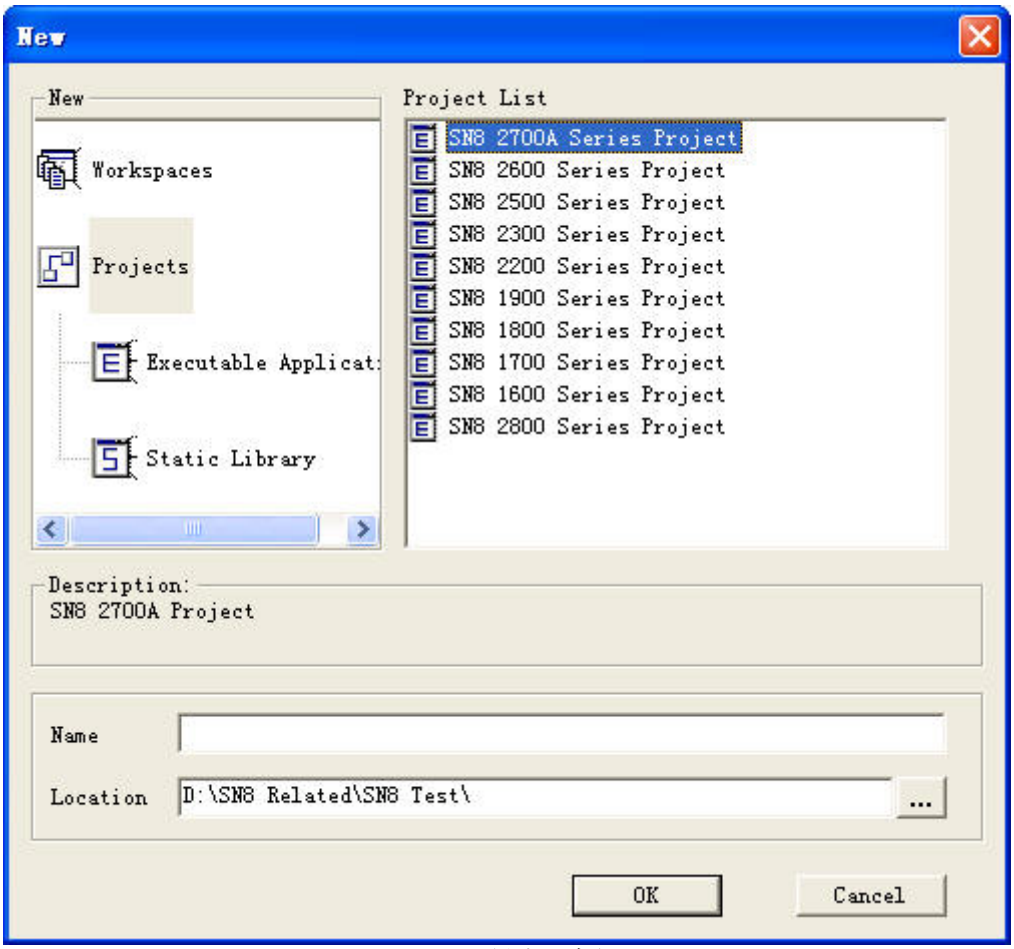


Figure 5-2　创建一个新项目

命名

键入你想要的工程名，后缀名为".PRJ"

路径

单击'…'按钮选择一个路径

最后，完成工程的建立。下来应该进一步完善当前所建工程。

5.1.2 Open and Close a Project

在任何时候，SN8 C Studio一次只能运行一个打开的工程。如果要执行一个工程，首先用Open命令将它打开。然后直接输入工程名或从浏览器的目录中选择一个工程名。使用Close命令关闭工程。

5.2 管理源文件

用户可以使用命令New，Edit分别对已经打开的工程源程序文件进行添加和删除。源文件按次序显示在列表框中，这些次序为输入（目标代码）连接器中的文件次序。（目标代码）连接器按照列表框中文件的次序对输入文件进行处理。
以下步骤将阐明如何管理源文件。

5.2.1 创建一个新的源文件

在文件菜单或工程菜单中选择新建命令。
单击Files按钮创建源文件，此文件将归入活动工程中。
软件提供了多种源文件类型，用户可选择合适的类型创建文件。

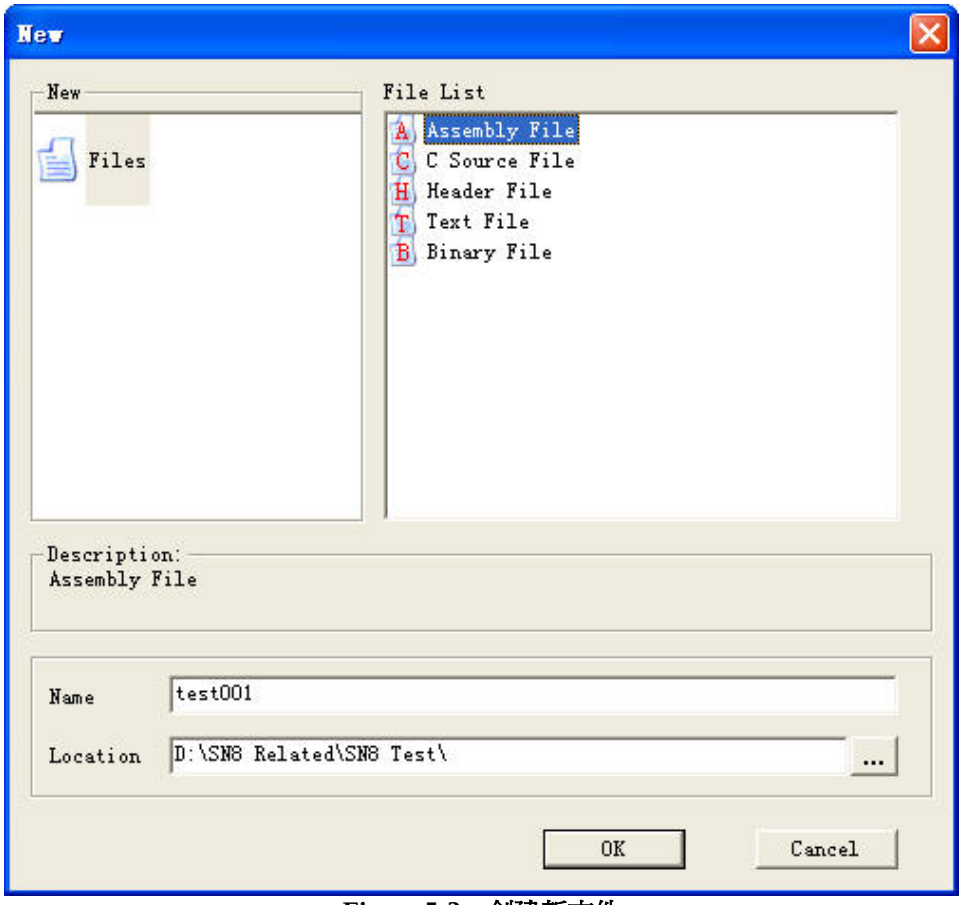


Figure 5-3　创建新文件

汇编文件（Assembly File）
汇编语言指令的集合。

C源文件（C Source File）
C语言指令的集合。

头文件（Header File）
与汇编源文件共同使用或包含在汇编源文件中的符号声明。

文本文件（Text File）
纯文本内容。

二进制文件（Binary File）
二进制数据内容
点击对话框中你想要的源文件类型，然后键入文件名。选择源文件正确存放的目录，双击OK按钮将文件添加到工程。
当所选源文件成功被添加，其文件名将显示在工程文件的列表框中。

5.2.2 添加源文件

用户也可以右击工作区窗口的活动工程选择'Add Files'将已经存在的文件添加到选中的工程中。新文件将自动添加到活动工程当中。

5.2.3 删除原文件

如果要从工程中删除源文件，右击此文件名选择'删除文件'项。事实上，从工程中删除源文件并不会把文件删除掉，而是将文件信息移出工程。

Studio 还支持另外两种文本文件：
链接文件：（目标代码）连接器使用链接文件依指定顺序放置文件。
工具定义文件：文件中对各工具定义。

5.3 设置工程

选择工程菜单中的设置命令，出现工程设置窗口。

5.3.1 常规设置

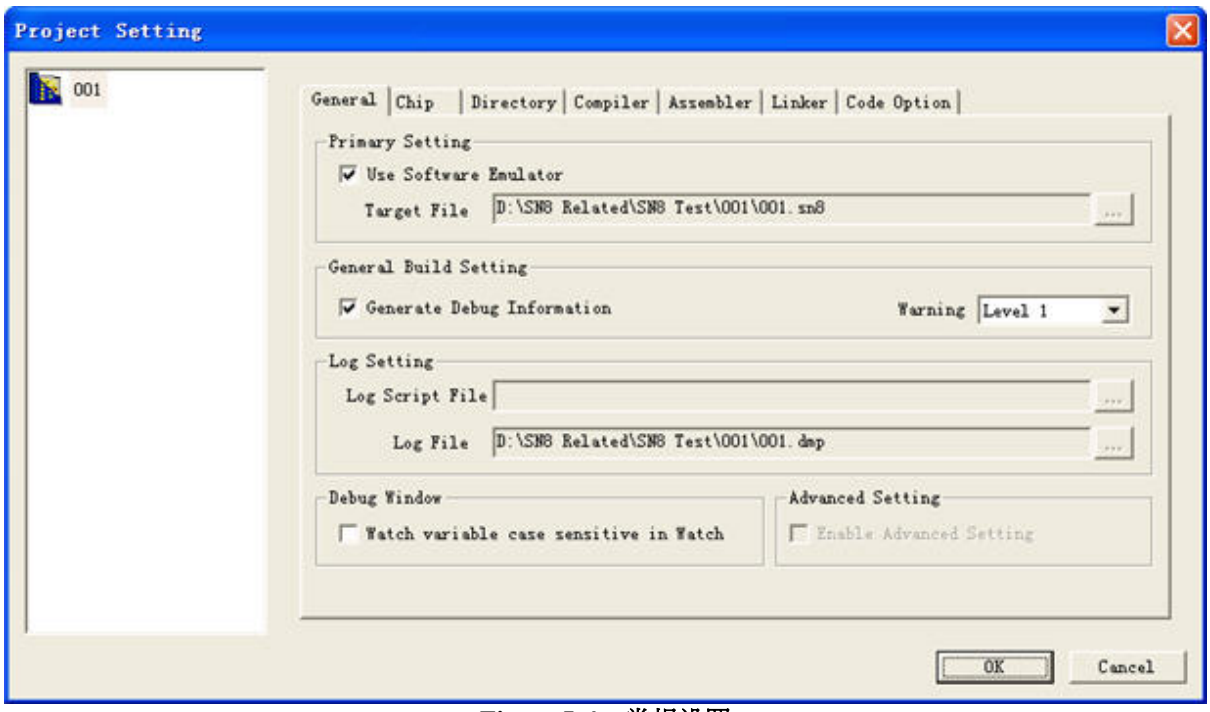


Figure 5-4 常规设置

基本设置

选择则使用软件仿真否则使用ICE仿真；

选择设置生成的目标文件名（系统默认设置）。在本文中没有选择此项。

常规Build设置

选择是否在输出文件中产生调试信息；

选择设置代码段保护特性；

设置编译警告级别；

记录设置

把记录原文件（由连接器生成）和记录文件设置为LOG指示，仿真器（ICE）将把内存寄存器的值保存到运行记录文件中。本文中没有向用户开放文件名设置。

调试窗口

选择在观察窗口中灵活的对各变量是否区分大小写进行设置。

高级设置

现在还无此功能。

5.3.2 芯片设置

用户可根据各芯片的定义正确的选择所需芯片。Chip List 中给出了所选系列中所有支持的IC。Description窗口显示被选中芯片的描述。

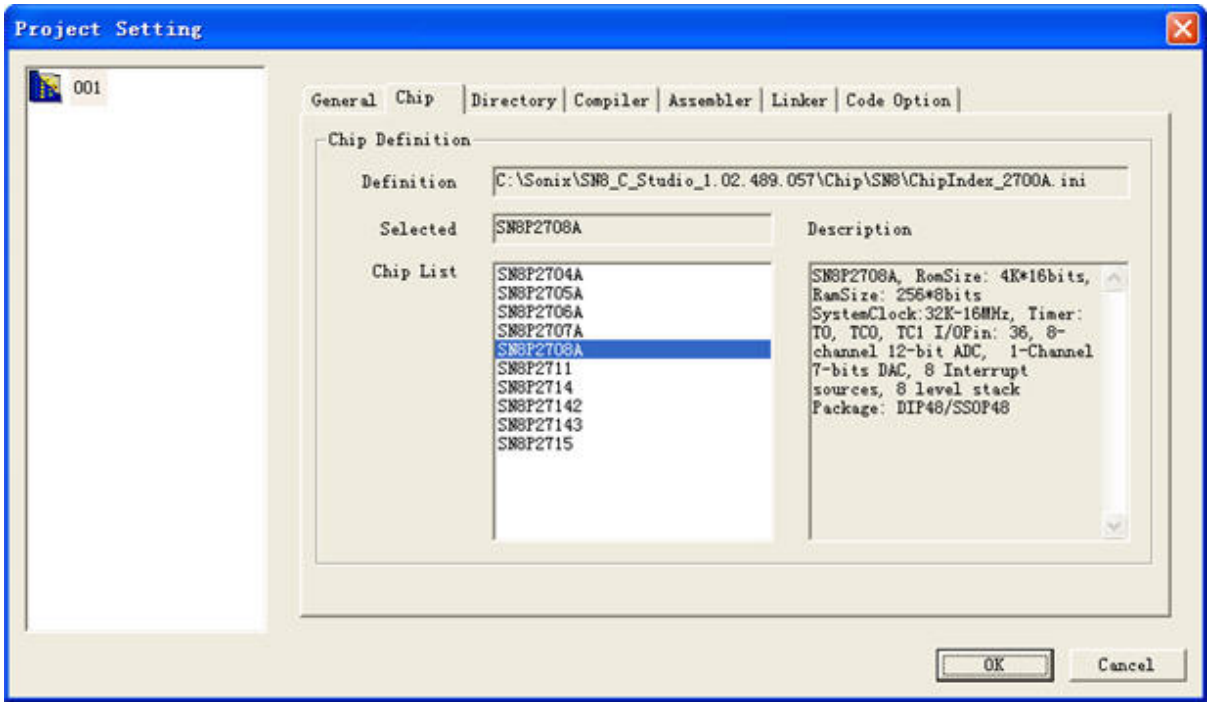


Figure 5-5 芯片设置

5.3.3 目录设置

标准库

设置标准（系统定义）库的版本和路径。如果没有指定，编译器将自动使用INI文件中定义的版本和路径。

附加路径和库

设定将要在编译连接过程中包含进来的文件的路径、库的路径、库链接的路径。

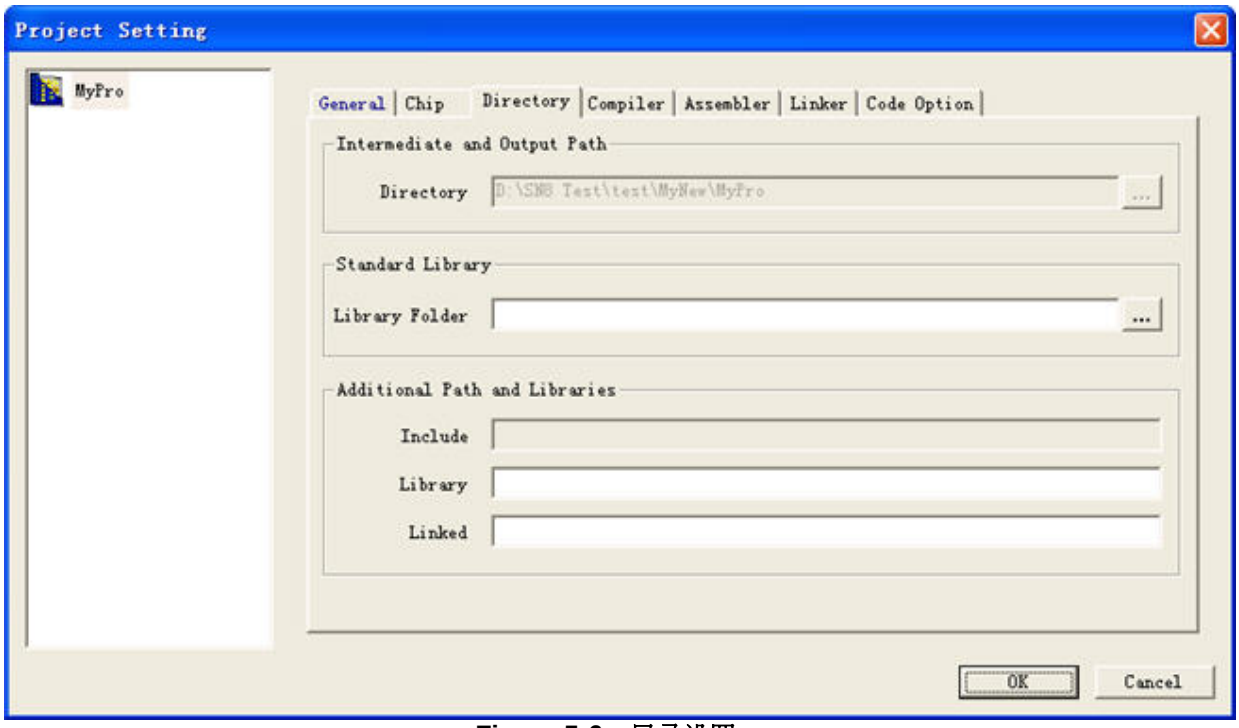


Figure 5-6 目录设置

5.3.4 汇编器设置

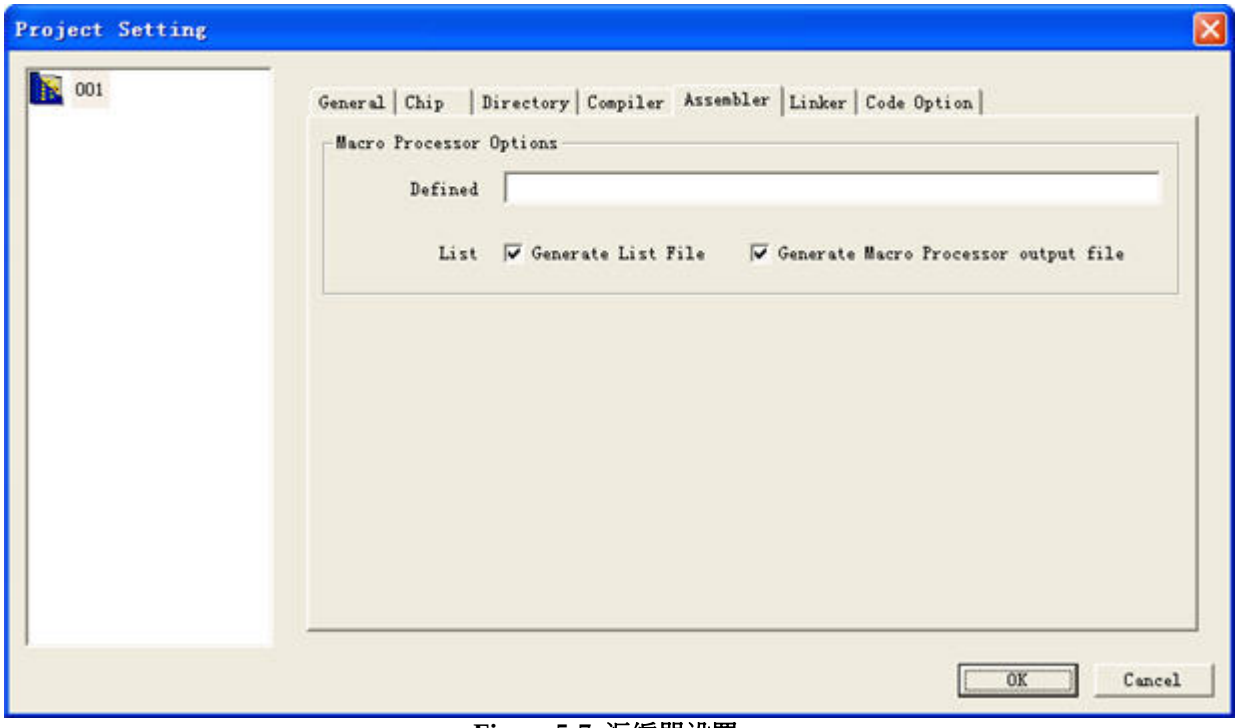


Figure 5-7 汇编器设置

定义
定义宏处理程序，使工程能够识别并按其定义对它们进行操作。

列表文件
设定创建列表文件信息

5.3.5 连接设置

输出
显示相应输出文件的详细路径。

MAP文件
设置创建普通的map文件并显示其准确路径。SN8 C STUDIO将给此map文件定义与工程相同的扩展名.map file。每个输出文件的默认文件名与源文件的相同，但文件扩展名不同。

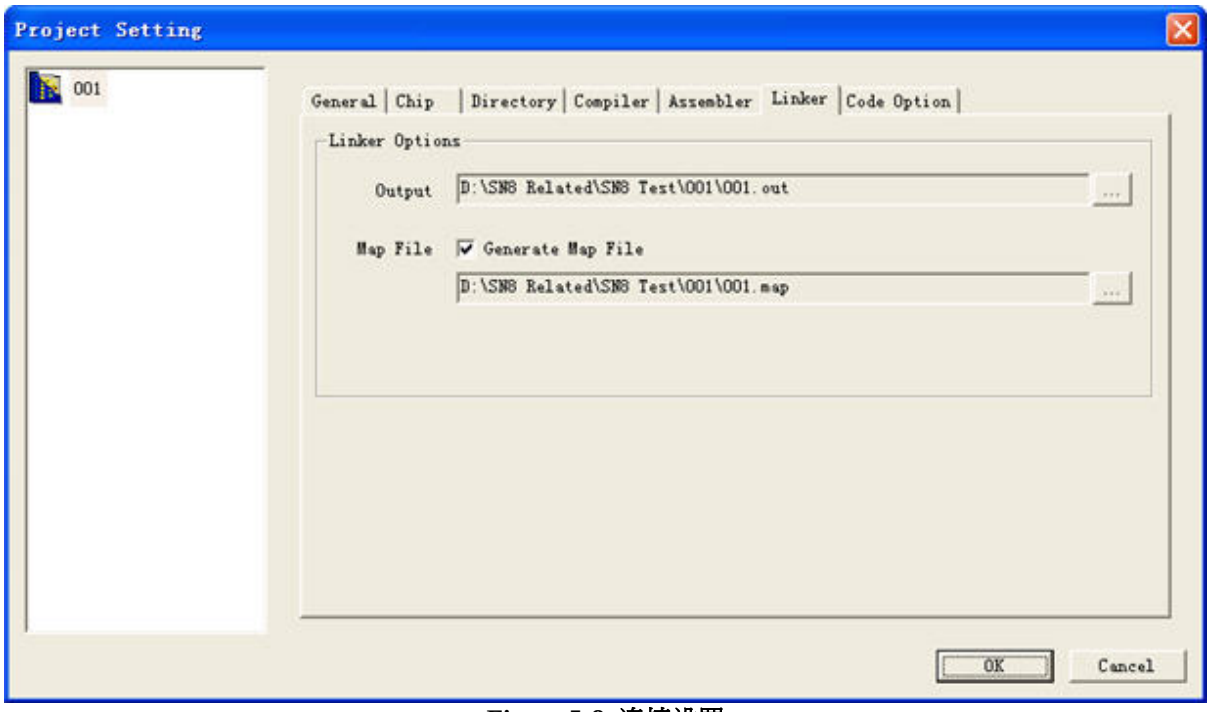


Figure 5-8 连接设置

5.3.6 Code Option 设置

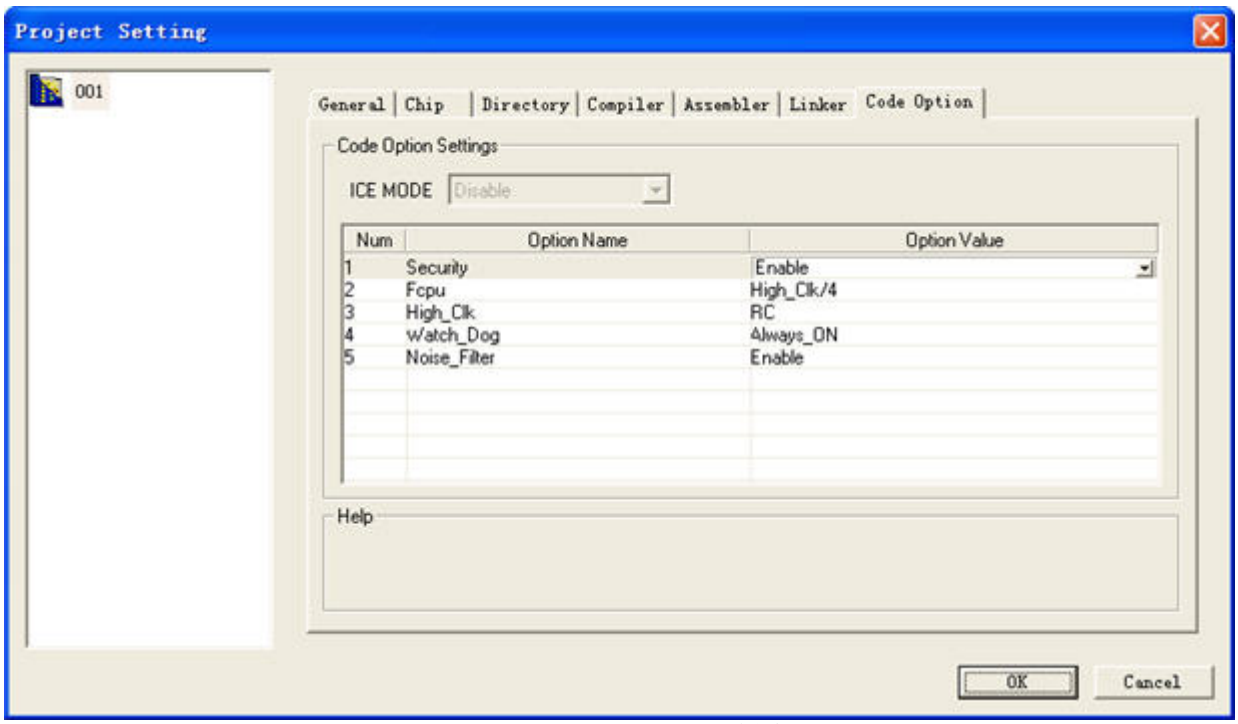


Figure5-9 Code Option 设置

编译选项对话框包括两部分。最上面的是ICE模式设定。如果仅对工程进行仿真选择"启用"，否则选择"禁用"。文本列表框内罗列了包括程序加密、CPU频率、高速时钟、看门狗、复位引脚和杂讯滤波器等等所选芯片具有Code Option选项（注：依不同芯片会显示不同的选项）。单击相应的选项就可以在下拉菜单中选择想要的值进行设置。

5.4 编译项目

下面几节介绍如何编译一个打开的工程。要编译的工程中必须最少有一个可以被编译的文件。

5.4.1 编译源文件

编译一个源文件，即将一个C文件和指令设置表文件当作输入文件在编译器中运行，产生一个可变地址的目标文件和列表文件。生成的可变地址目标文件直接放置在工程中。

编译器检测并列出当前被编译文件中的错误和警告，此错误和警告信息显示在输出窗口。双击那些信息可以在源文件中找到它们的位置，且相关行被加亮显示。此时要做的是，修改源文件重新编译，直到不出现任何错误。

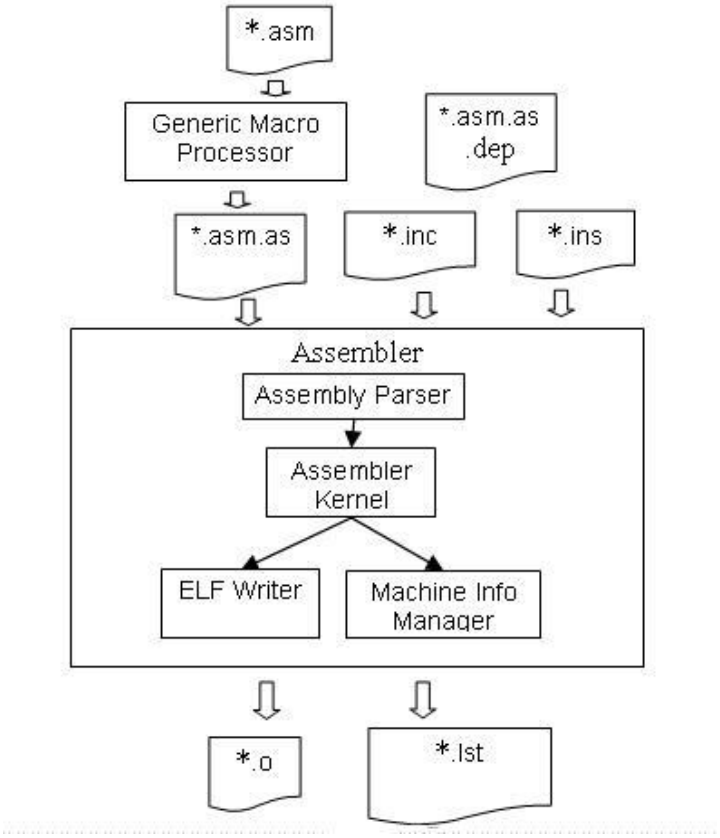


Figure5-10 编译过程描述

5.4.2 链接

链接就是将上述可变地址目标文件和库文件当作输入文件在编译器中运行的动作，这将在工程目录下生成一个或多个由工程指定的输出文件。

在编译过程中最多只运行一次链接。

5.4.3 Make

执行Make命令将编译那些执行从属检验功能的工程文件，从而只有旧的源文件被重新编译且在必要时只进行链接。

执行Make前将检查正在被编辑的工程源文件是否已经被修改。当源文件已经被修改，则在下面对话框中指定是否执行此动作。通常，选择"Y"重新编译工程。

在Make过程中如果没有出现任何错误则认为成功Make，即编译器可以生成一个新的make.exe文件。
"警告"并不影响编译的成功与否，但在开发过程中可能对应用程序的调试很重要。

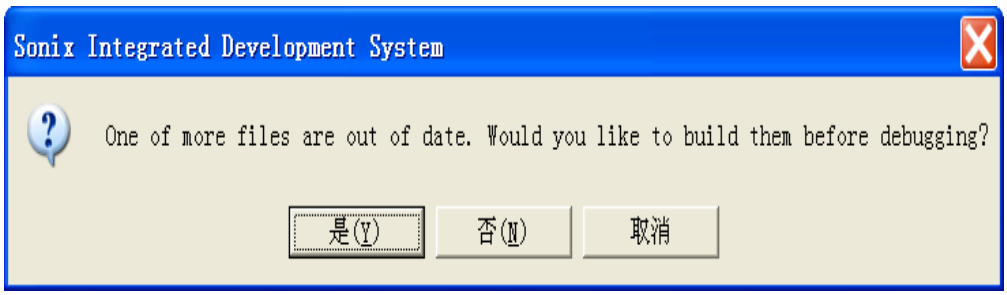


Figure 5-11 文件被修改的提示对话框

5.5 创建（Build）项目

确保下列任务在编译当前活动工程前被完成：正确设置所有工程选项，特别是选择与源文件声明相匹配的芯片。
可以分几步来执行编译命令：

- 调用SONiX普通宏处理程序和汇编程序对当前工程的源文件进行汇编
- 链接当前生成的目标文件并生成一个map文件和一个输出文件
- 将目标文件下载到ICE
- 在创建窗口显示生成结果

有两个与创建工程相关的命令：创建（Build）和重新创建（Rebuild）。

5.6 调试项目

创建工程后，将生成几个目标文件。扩展名为'SN8'的文件为最后的可执行文件，扩展名为'map'的文件描述ROM中字段和标号的位置。

输出文件
用于调试。

二进制文件
原始的可执行二进制文件。

Map 文件
由链接生成，描述符号地址和字段地址。

列表文件
由汇编程序生成。

- **开始调试**

单击主菜单'Debug'中的'Begin Debug'开启调试器，Studio将在进程窗口中显示下载进度。下载完毕后，将显示所有调试窗口（和仿真窗口）且所有被打开的文件进入只读状态。

- **程序跟踪**

开始调试后编辑窗口的左边将出现一个指针 ,指针代替当前程序计数器指向源程序执行点。单击调试命令，如'Step Into'、'Step Over'、'Run'可以跟踪程序。每次跟踪命令结束后，指针指向下一条将被执行的指令且所有调试窗口将更新其结果。

- **设置断点**

SN8 C STUDIO为有条件停止目标程序的运行提供断点。 在调试过程中设置断点是很平常的，在这些断点处可以获得所需的指定寄存器和IO口的信息。设置断点有两种方法：一种在文本编辑器里直接设置，另一种在断点对话框中进行设置。

在一条指令处设置有效断点，仿真器或ICE将在执行此条指令之前停止运行。也就是说，这条指令为程序再次开始运行时第一条被执行的指令。

- **退出调试**

完成程序调试后，单击'Exit Debug'退出调试器，随后所有被打开的文件只读释放。

6. 应用实例

本章将详细介绍如何利用SN8 C STUDIO建立和执行工程，也适用于初学用户熟悉此工程开发环境。

6.1 创建一个新工作区

SN8 C STUDIO 利用工作区模式管理工程。每次对工程操作，都应该首先为工程创建一个新的工作区。

选择文件菜单中的新建命令，将出现如下新建对话框：

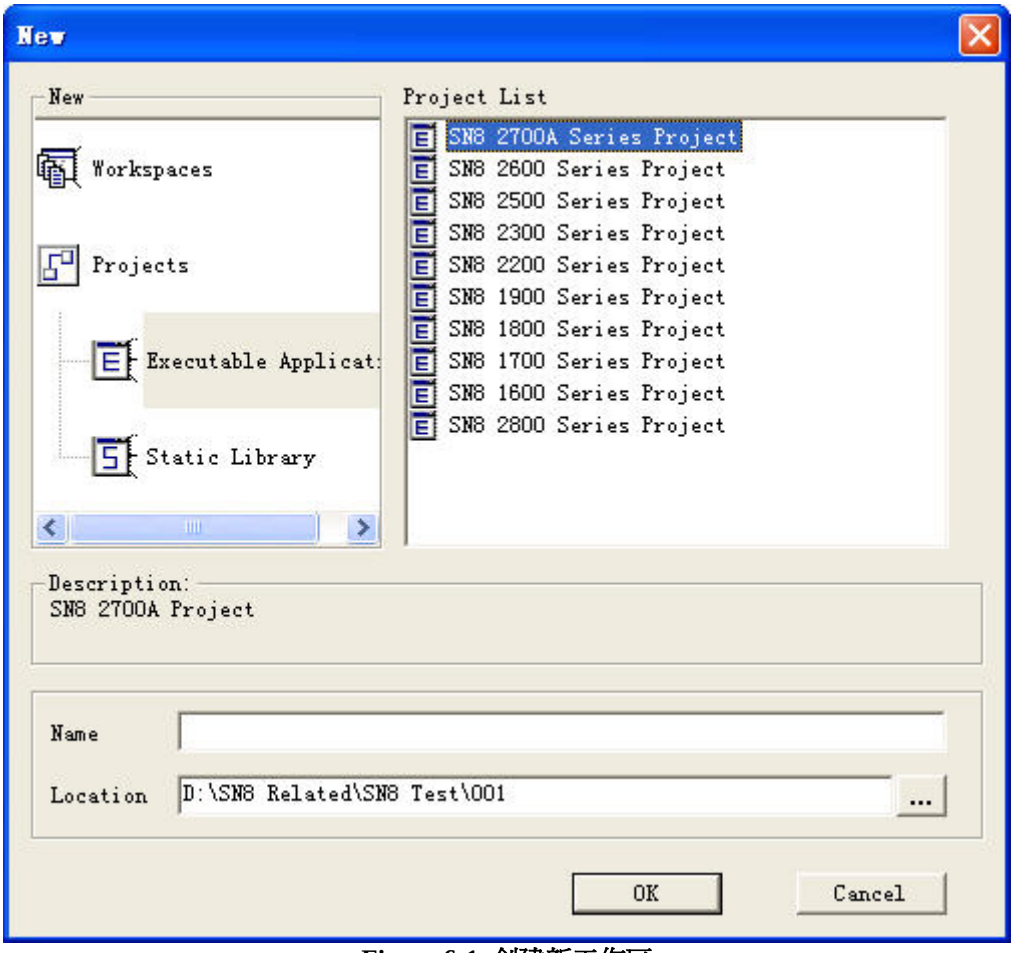


Figure 6-1 创建新工作区

单击工作区（Workspace）图标，在工程列表窗口中列出可用芯片类型供用户选择适合于工程的芯片。然后为工作区命名并指定详细存储路径。

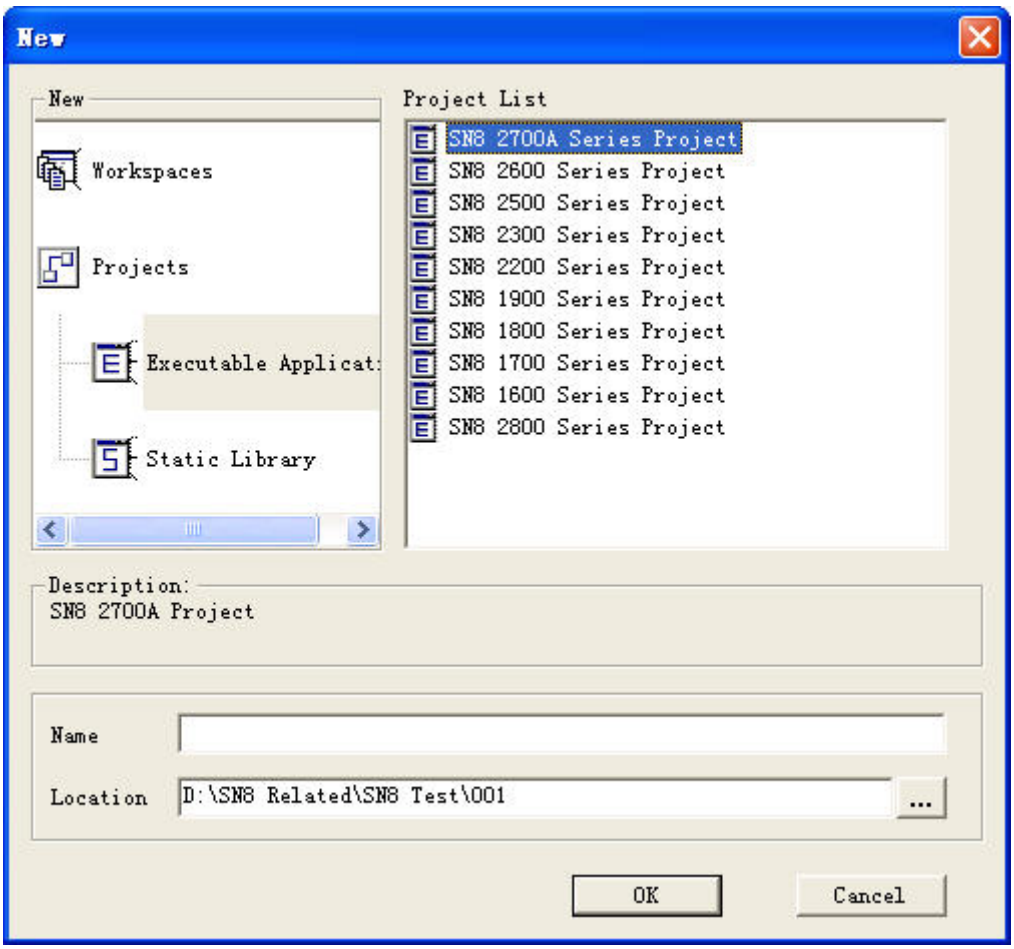


Figure 6-2

按 OK 按钮，出现工作区窗口和输出窗口。打开目标文件夹，你会发现刚才被定义的子文件夹，其中包含一个新创建的.wsp 配置文件。

Figure6- 3

6.2 创建一个新项目

成功创建一个工作区后，其窗口内的工程数为零。接着根据工程所用芯片来创建一个工程。

在文件菜单中选择新建命令，SN8 C Studio 将默认创建一个新工程。在弹出的新建对话框中的工程列表窗口中选择合适的芯片母体。SN8 C Studio 将自动在地址栏内显示当前创建的工作区目录。通常，不对默认路径做任何修改。给新建工程区一个有意义的名字，通常取与工作区相同的名字。

选择新建命令，点击合适的芯片母体；

单击 OK 按钮；设置工程选项。

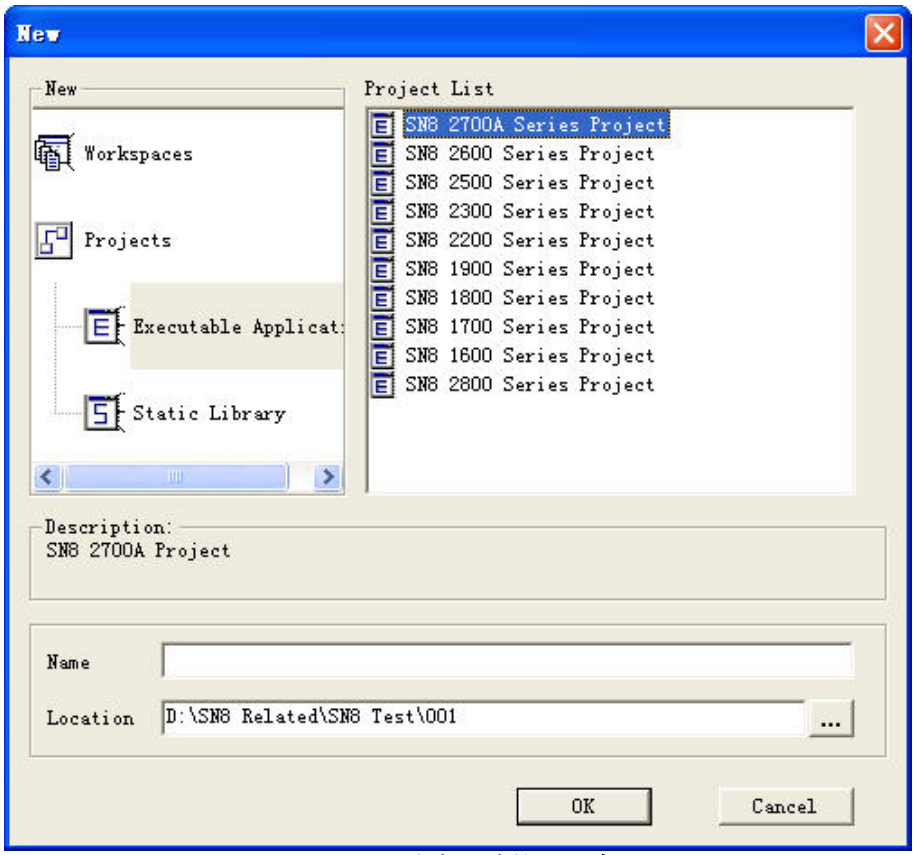


Figure 6-4 建立正确的项目类型

显示弹出的工程设置对话框设置工程选项，左边窗口为当前创建工程的工程名。

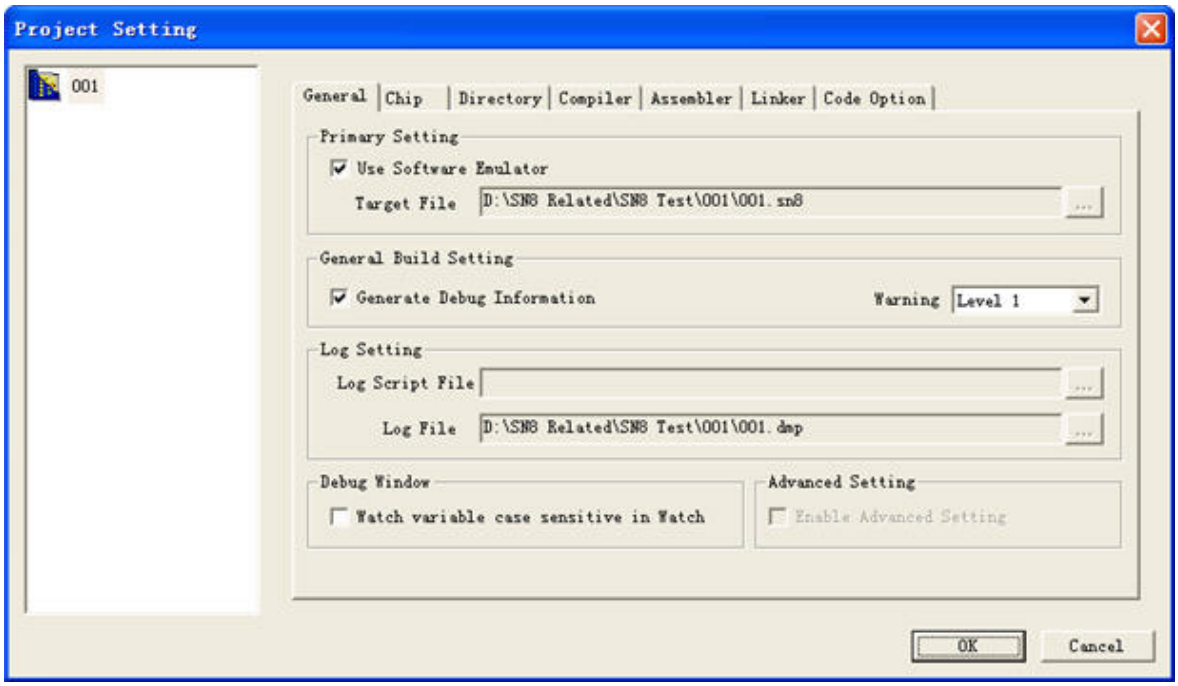


Figure 6-5 项目设置

选择合适的芯片类型：

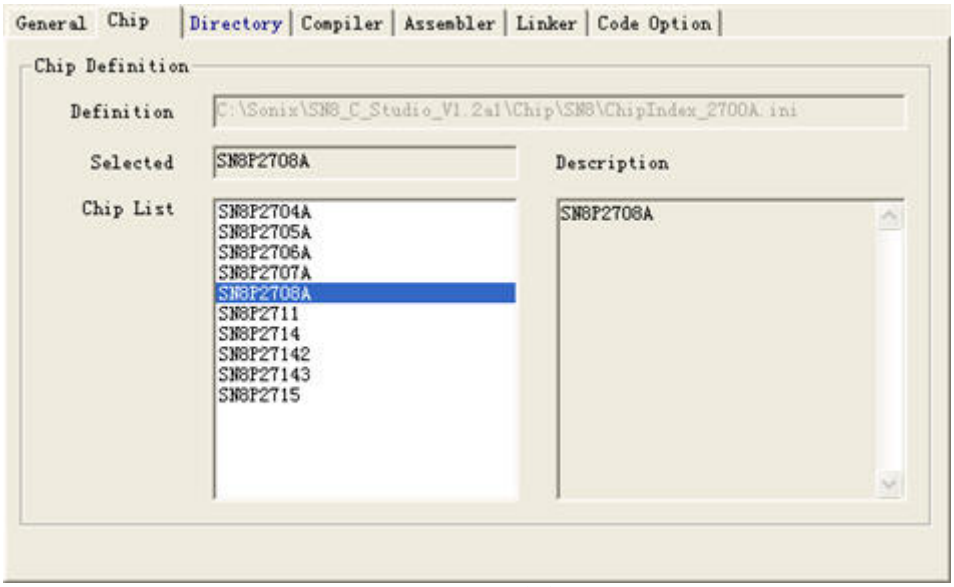


Figure6- 6 芯片设置

设定编译选项和 ICE 模式：

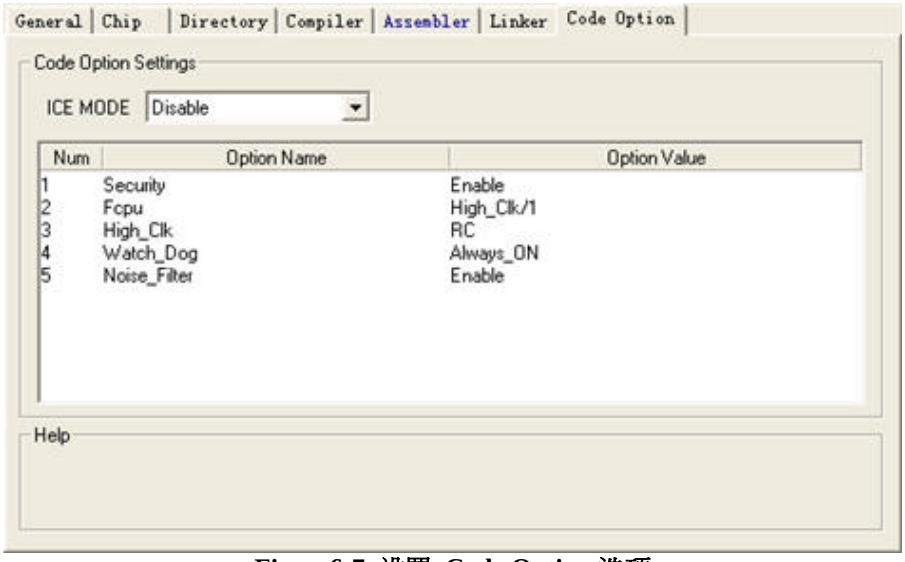


Figure6-7 设置 Code Option 选项

单击 OK 按钮，生成一个没有文件的工程。在 Workspace 管理窗口中出现当前创建的工程，处在被激活状态（工程名为加粗字体）。

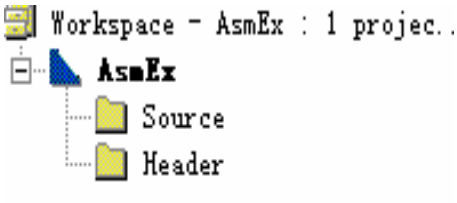


Figure6-8 工程管理窗口

打开相关的文件夹将发现 SN8 C STUDIO 生成了一些新文件。其中.pfj 文件为工程文件，其它的为工程的配置文件和头文件。

6.3 创建一个新文件

选择文件菜单中的新建命令；
单击新建对话框中的 Files 图标，在文件列表中选择汇编文件会 C 源文件；
最后必须在底部的空白栏内输入文件名和指定存储的地址；
单击 OK 按钮，SN8 C Studio 将显示当前创建源文件的活动编辑窗口。

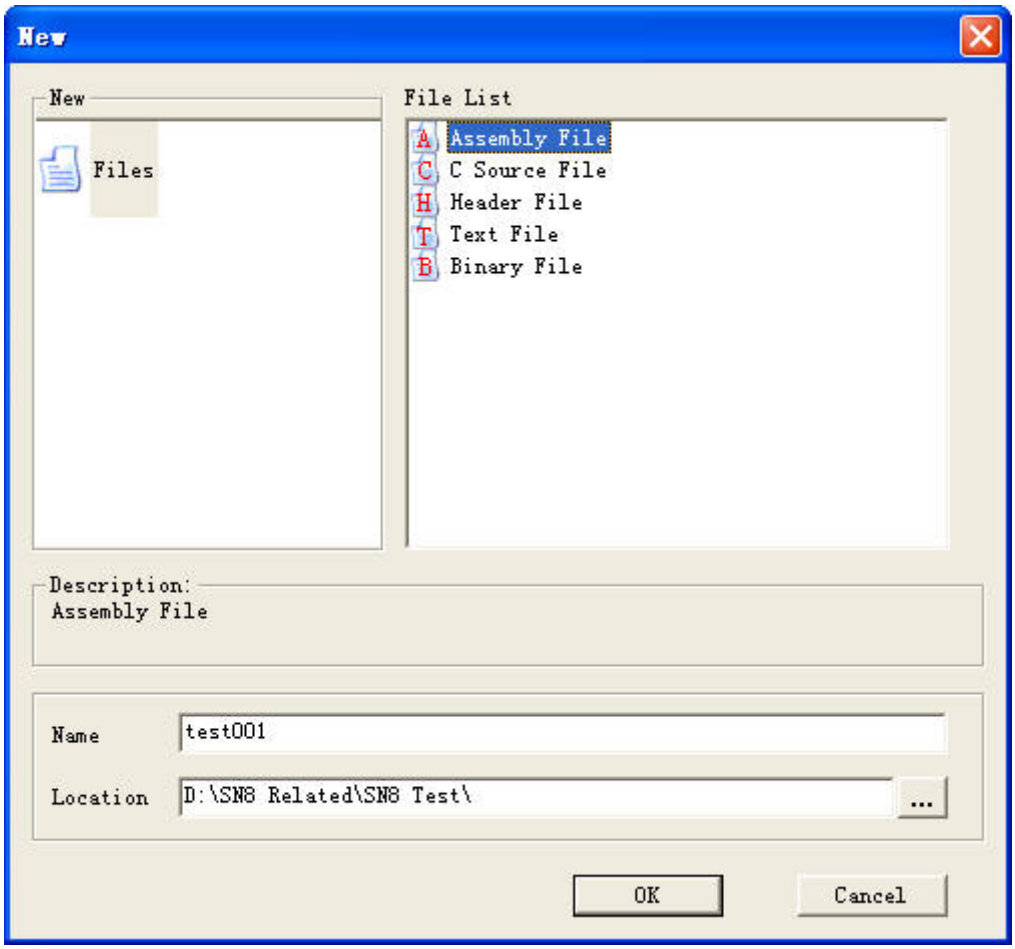


Figure 6-9 Crate a Source File

6.4 编辑代码

现在我们编写如下汇编文件。主控部分 Main.asm 绝对必需，它包含调用功能函数部分和全局变量。

Example:

```
////////////////////////////////////////////////////////////
//                      s_a_pro.asm                       //
////////////////////////////////////////////////////////////
;************************************************************
;                        main
;************************************************************
 chip    SN8P2604
 extern      code      reset
 extern      code      mnkey
 extern      code      int_rs
 extern      code      mnintgnd
 .nolist
 includestd      macro1.h
 includestd      macro2.h
 includestd      macro3.h
 .list
 .data
 include         Tsn8p2604.inc
 include         custom.h
 .code
 user_seg    segment  code at   0x00
 jmp         main00
 org         8
 jmp         int_rs
 org         10
 main00:
        mov      a,#0fh
        b0mov    stkp,a
        mov      A,#0c0h
        b0mov    pflag,a
        call  reset          ;initial cpu register  初始化 cpu 寄存器
        b0bset   fgie
 main10:
        @rst_wdt
            .
            .
            .
        call  mnintgnd       ;interface between interrupt and main  中断与主程序接口
        call  mnkey
 main90:
```

```
        jmp     main10

//////////////////////////////////////////
//          tb_key                      //
//////////////////////////////////////////
chip    sn8p2604
extern      data    keychat
extern      data    aplcode
extern      code    relaysw
public      mnkey

.nolist
includestd      macro1.h
includestd      macro2.h
includestd      macro3.h
include         custom.h

.list
.data
include         Tsn8p2604.INC
;*************************************
keyinbuf0   ds  1       ; bit0 > key 1      第 0 位，按键 1
                        ; bit1 > key 2      第 1 位，按键 2
                        ; bit2 > key 3      第 2 位，按键 3
                        ; bit3 > key 4      第 3 位，按键 4
                        ; bit4 > key 5      第 4 位，按键 5
                        ; bit5 > key 6      第 5 位，按键 6
                        ; bit6 > key 7      第 6 位，按键 7
                        ; bit7 > key 8      第 7 位，按键 8

keyinbuf        ds  1       ; bit0 > run_key
keychkbuf   ds  1
keycvtbuf       ds  1
keyoldbuf       ds  1
keystat     ds  1           ; bit0 > key processing
                            ; bit1 > pin processing
                            ; bit7 > clean key buffer
K1t             equ     keystat.2
//keychat       ds      1
keycode     ds      1
;*********************************************
.code

;*********************************************
;       keyBoard scan
;*********************************************
mnkey:                      ;scan key loop          按键扫描回圈
```

```
mnkey10:
        call  keyin             ;read into in buf
        call  keychk            ;read into check buf
        call  keycvt            ;read into convert buf

mnkey90:
        ret

;***********************************************
;   keyoutport              ;scan output
;   keyinport               ;scan input
;***********************************************
keyin:
        clr     keyinbuf            ;clear the scan content
            b0bts1  key1_p          ;inspect the key station

        b0bset  keyinbuf.0
        b0bts1  key2_p
        b0bset  keyinbuf.1
        b0bts1  key3_p
        b0bset  keyinbuf.2
        b0bts1  key4_p
        b0bset  keyinbuf.3
ifdef           key5_p
        b0bts1  key5_p
        b0bset  keyinbuf.4
endif


keyin90:
        ret

;***********************************************
;   check keyinbuf AND keychkbuf        按键防抖动
;***********************************************
;
keychk:
        b0mov   a,keyinbuf          ;confirm the depressing of keys
        xor     a,keychkbuf
        jnz     keychk10            ;if same
        b0bts1  keystat.0           ;process with a key, quit
        jmp     keychk90            ;
                                    ;wait chatter
        b0mov   a,keychat           ;check keychat
        jnz     keychk90            ;whether have finished eliminating flutter
                                    ;key bounce time=0,copy chk buf into cuv buf
        b0mov   a,keychkbuf         ; finished eliminating flutter and store the key
```

```
station
        b0mov   keycvtbuf,a
        b0bclr  keystat.0           ;clear the flag of key processing
        jmp     keychk90
keychk10:
        b0mov   a,keyinbuf          ;deal with eliminating flutter
        b0mov   keychkbuf,a
        b0bset  keystat.0           ;set key processing
        mov     a,#7                ;constant of eliminating flutter
        b0mov   keychat,a
keychk90:
        ret

;***********************************************
;          compare keycvt and keyold          *
;    and find the different.                  *
;***********************************************
keycvt:
        b0mov   a,keycvtbuf
        xor     a,keyoldbuf
        jnz     keycvt0                 ;
        jmp     keycvt90                ;quit
                                        ; Check the new pressed key from table
keycvt0:
        b0mov   a,keycvtbuf
        b0mov   keyoldbuf,a
        b0bts1  keycvtbuf.0             ;judge the key1_p station
        jmp     keycvt10
        call  relaysw                   ;run the relay
        jmp     keycvt90

keycvt10:
        b0bts1  keycvtbuf.1
        jmp     keycvt20
        mov     a,#1
        b0mov   aplcode,a
        jmp     keycvt90

keycvt20:
        b0bts1  keycvtbuf.2
        jmp     keycvt30
        mov     a,#2
        b0mov   aplcode,a
        jmp     keycvt90
keycvt30:
        b0bts1  keycvtbuf.3
        ifdef   key5_p
        jmp     keycvt40
```

```
    else
        jmp     keycvt90
    endif
        mov     a,#3
        b0mov   aplcode,a
        jmp     keycvt90
keycvt40:
        b0bts1  keycvtbuf.4
        jmp     keycvt90
        mov     a,#4
        b0mov   aplcode,a
        jmp     keycvt90

keycvt90:
        ret

;*********************************************
```

C source file:

```
/*****************************************************
*
*   File Name : SN8C_Ex.c
*   Test History : V1.00.220
*   describe:test 2708 interrupt
*
*****************************************************************/
#include <sn8p2708a.h>
struct word{
    unsigned fint:1;
    unsigned :7;
}intword;
unsigned int tc0cvalue=0x64;
unsigned int accbuf = 0x00;
unsigned int pflagbuf = 0;
__interrupt intserv(void)
{                                       //The data will auto store!
    _bCLR(&INTRQ,5);
    TC0C = tc0cvalue;
    intword.fint = 1;
}

void initIO(void);
void initINT(void);

void main(void)
    {
        STKP=0x07;
```

```
        initIO();
        initINT();
        while(1)
        {
            if(intword.fint!=0)
            {
                P1=0x00;
                P2=0x00;
                P3=0x00;
                P4=0x00;
                P5=0x00;
                P0=0x00;
            }
            else
            {
                P0=0xff;
                P1=0xff;
                P2=0xff;
                P3=0xff;
                P4=0xff;
                P5=0xff;
            }
        }
    }
void initIO(void)
{
    P0M=0xFF;
    P1M=0xFF;
    P2M=0xFF;
    P3M=0xFF;
    P4M=0xFF;
    P5M=0xFF;
}
void initINT(void)
{
    INTRQ=0x00;
    INTEN=0x00;
    TC0M=0x00;
    TC0M=0x20;
    TC0C=0x64;
    _bCLR(&INTRQ,5);
    _bSET(&INTEN,5);
    _bSET(&TC0M,7);
    _bSET(&STKP,7);
}
```

注意：以上程序只为程序架构提供参考。

6.5 编译链接（Compiling and Builing）

选择 Build 菜单中的 Compile current file 命令，单击工具栏中的 Compile 按钮或使用快捷键 Ctrl+F7 开始编译工程。如果出错或者有警告系统将会在输出窗口显示出来，用户可根据显示出的信息对程序进行修改：如果是语法错误，双击此错误信息光标将移动相应的代码行。

成功编译后，下一步为创建。创建过程及生成一些必要的配置文件，同时系统检查硬件配置。

6.6 调试

在'Debug'菜单中选择'Begin Debug'命令或直接点击工具栏中的  图标，系统界面变为如下图所示，且在调试过程中内存窗口、监视窗口、变量窗口、寄存器窗口、堆栈窗口和汇编窗口都可以使用。

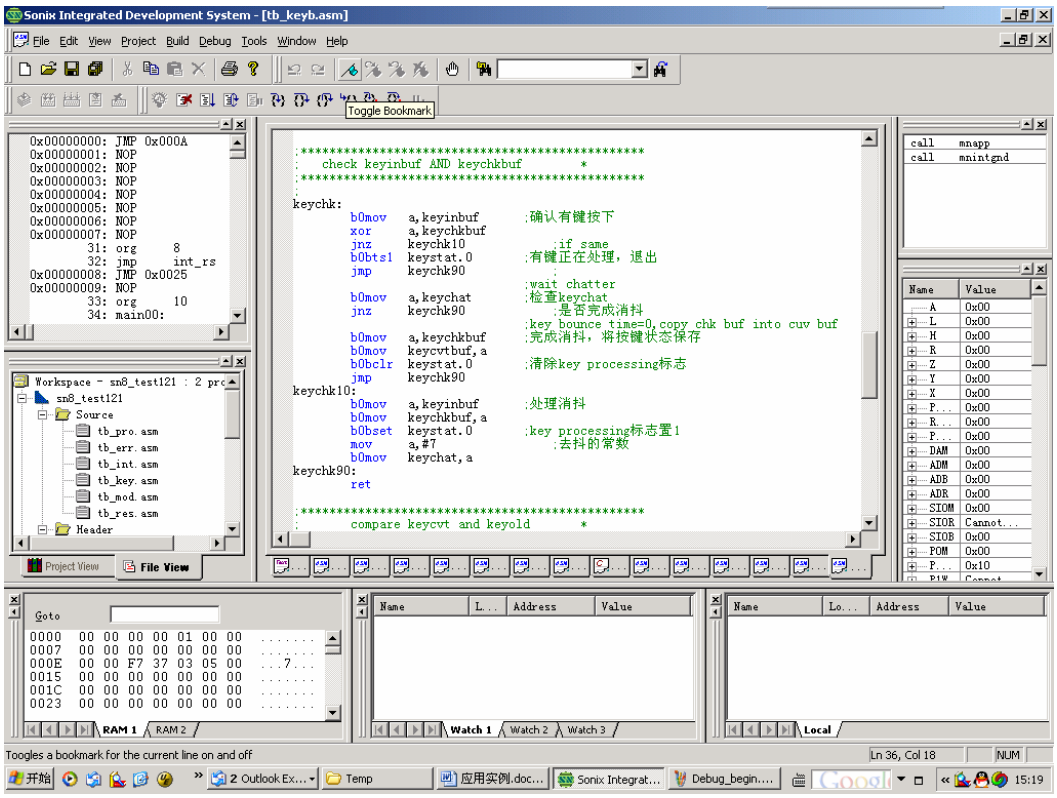


Figure 6-10

监视窗口（Watch）

监视窗口用来观察指定变量。双击想要观察的变量并将其拖动到观察窗口，就可以观察它的变化。SN8 C Studio 把发生变化的值显示为红色。

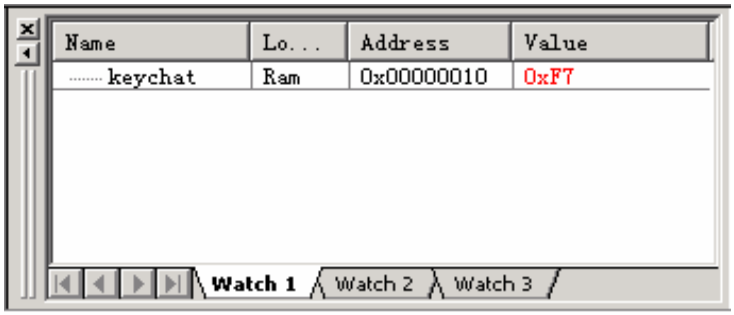


Figure6-11

为了方便观察，可以将观察窗口分为 3 个部分。

变量窗口

变量窗口显示由用户自行改变的变量形式和内容数值。此窗口仅自动显示局部变量的变化。

寄存器窗口

寄存器窗口显示 RAM 空间中位于 0X80~0XFF 的专用寄存器的当前值。单击前面的加号'+'，将展开对每一个有效位进行显示。系统用红色显示标示当前已经发生变化的值。

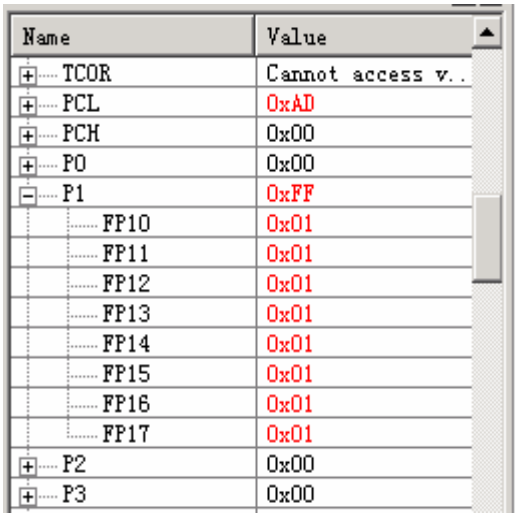


Figure 6-12

堆栈窗口

堆栈窗口显示堆栈的使用情况和入栈函数，在此可以判断程序调用是否正确。

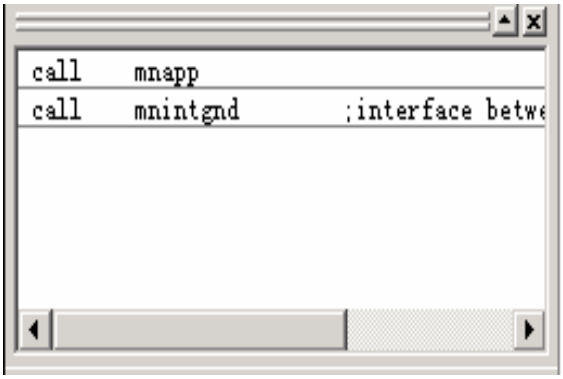


Figure 6-13

内存窗口

在程序运行过程中，如果想知道指定 RAM 寄存器的值，在 Go to 后面的框中输入地址点击 Enter，内存窗口中将显示指定寄存器的值。

汇编分析窗口

汇编分析用来在运行 C 源代码是，观察生成的对应汇编代码。

6.7 设置断点

将光标移到有效行并选择 Breakpoint 命令（F9）就可以在程序中设置断点，程序将运行到断点处停止并且不执行此行指令。

```
int_rs:
                                    ;push
        xch     a,accbuf
        b0mov   a,pflag
        b0mov   pflagbuf,a
                                    ;test irq
        b0bts0  fTimerirq
        jmp     inttc0

int_rs90:                           ;pop
        b0mov   a,pflagbuf
        b0mov   pflag,a
        xch     a,accbuf
        reti
```

6.8 跟踪程序执行

用户可以利用系统的调试工具跟踪程序，保证对正在执行的命令进行跟踪。根据实际情况可以选择单步执行或越过一段程序往下执行。系统提供三种方式跟踪程序：调试菜单、调试工具栏按钮和快捷键。

6.9 输出文件

下面主要讨论程序的输出文件：
仿真结果：
正确的将仿真器与 PC 连接起来，清除工程设置中的用户仿真选项并与需要仿真的目标硬件连起来。按下 Debug 按钮或快捷键进入调试状态，即可在所连硬件上看到程序运行结果。如果没有出现预期的结果，可以修改程序直到出现你想要的结果。最后得到一个真正可以运行的程序。

输出烧录文档
SN8 C Studio 将自动在默认目录下生成烧录文档.sn8，此文档包含在当前活动工程中。

第二部分　开发语言和开发工具

7. Assembler

7.1 SN8 汇编语言

一般而言，汇编的指令包含四部分，它们之间用空格或TAB键隔开。如下所示：
　标号　操作助记符　操作码　　注释

Example:

```
START:  MOV     A,#0X35         ;A = 0X35
        ADD     A,#36H          ;A = 0X6B
        DAA                     ;A = 0X71
        JMP     START
```

下面章节将介绍标号，操作助记符，操作数和注释的书写限制和规则。接下来详细介绍这四部分。

7.1.1 标号

首先，标号必须以字母a~z，@，或下划线为开头；
除了最后一个字符的其他字符可以是a~z，@，下划线或0~9；
标号的最后一个字符必须为冒号“：”；
标号的长度没有限制但不可以重复命名。

为了避免使用太多不同的标号，以下符号可以用来表示不同的标号名称：
首先，使用符号“@@:”作为临时标号名，可以从前面和后面对它进行访问。
其次，使用“@B”指向前面离它最近的“@@:”标号；使用“@F”指向下一个（离它最近的）“@@:”标号。

Example:

```
        JMP     @F              ; jump to the next @@
@@:
        …       …
        JMP     @B              ; jump to the previous@@
```

7.1.2 操作数

如果有两个操作数，应该用符号‘，’将它们隔开。

Example:

```
        MOV     A , #43h
or
        MOV     A, #'C'         // block the Operand by ' '
```

如果是位操作数，用‘.’隔开。

Example:

```
        B0BSET  0X86.2          // To set bit2 of 0x86 as 1.
or
        B0BSET  FC              // To default the constant value by system.
```

如果操作数是内存单元，可以用数值来代替它的地址。如果操作数为常量，应在此操作数前加‘#’。

Example:

```
        B0MOV   0x80, #3        // RAM[0x80] = #3
```

另外，符号$表示编译器当前运行程序地址（PC值）。

Example:

```
        JMP     $               // To represent unlimited loop
        JMP     $+1             // Equivalent to two NOPs
```

符号$还可以用来获取标号地址的高、中、低字节。

Example :

```
        B0MOV   X, #DATA1$H     // X = 0X12
        B0MOV   Y, #DATA1$M     // Y = 0X34
        B0MOV   Z, #DARA1$L     // Z = 0X56
        MOVC                    // ACC = 0X90，R = 0X78
        …
ORG     0X123456
        Data1   DW      7890H
```

最后，符号$还可以用来将表的14~17位定义为高位元（注：这一表示方法2系芯片不支持）。

Example:

```
            B0MOV       PFLAG, #Far_Lab$J       ;=B0MOV PFLAG, #30h
            JMP         Far_Lab
            …
ORG     0XC000
Far_Lab:
            …
```

7.1.3 注释

通常，符号“；”和“//”均可表示注释声明，连续的文字说明写在符号“；”或“//”后面。

Example:

```
            ; This is an example of demo code.
//    This is an example of demo code.
```

另外，使用符号/*….*/可以框住一行或多行注释。

Example:

```
            /* this is an example of demo code*/.
```

7.1.4 系统操作数

系统默认一部分操作数供用户方便使用。但也有一些操作数不被默认，这取决于所选芯片。确定这些默认操作数是否存在，可以在程序中判断默认常量@??_existas为1或0来检查系统是否提供这些操作数。另外，可以通过检查@p?_bits设定操作数的有效值来确定引脚数量。

7.1.5 芯片保留字

每款SONIX 8_bit 系列MCU母体都拥有自己的系统寄存器，且已经在汇编程序中定义为芯片的保留字。例如，H/L寄存器不需要声明就可直接在程序中使用，对应位也一样。如果程序中用到位寄存器，则应在寄存器名前加前缀“F”。例如将“GIE”置为1，正确代码为“b0bset FGIE”。可以参阅相关的数据手册了解详细的系统寄存器名称和芯片资料。

7.1.6 数值表达式

数值的表示方法有十进制、十六进制、二进制几种，但不管哪一种表示方法，第一个数字必须是0~9。

Example:

```
255          ；Decimal expression.
0xFF         ；Hexadecimal expression.
0FFh         ；Hexadecimal expression
11111111b.   ；Binary Expression
```

7.1.7 算术运算符

用户可使用+、-、*、/、%、&、|、^、~、()等符号进行算术运算。

Example:

```
2 + 3 – 4     1
2 + 3 * 4     14
```

以下按优先级顺序列出这些符号的数学涵义。

( )	**= sub-expression**	括号
+	**= plus**	正的
-	**= minus**	负的
~	**= not**	位逻辑反
!	**= logical not**	逻辑非
*****	**= multiplication**	乘
/	**= division**	除
%	**= modulo**	取模（取余）
+	**= addition**	加
-	**= subtraction**	减
<<	**= shl; logical shift left**	左移
>>	**= shr; logical shift right**	右移
>	**= greater than**	大于
<	**= less than**	小于
>=	**= greater than or equal to**	大于等于
<=	**= less than or equal to**	小于等于
==	**= is equal**	等于
!=	**= is not equal**	不等于
&	**= and**	位逻辑与
^	**= xor**	位逻辑异或
|	**= or**	位逻辑或
&&	**= logical and**	逻辑与

|| = logical or 逻辑或

7.2 伪指令

这里列举了交叉编译器进行编译产生项目代码时的风格样式的伪指令，根据不同的功能按以下分类介绍：

7.2.1 程序开始和结束

Syntax: CHIP SN8XXXX
说明：选择所用IC型号。用户可以通过菜单[option]->[chip info]了解当前所用的芯片。此命令必须定义在汇编程序的最前面，且只能定义一次。

Example:

Chip sn8p2708a

Syntax: ENDP
说明：强制结束程序，且位于此命令后的程序将不会被编译。

Example:

Endp

7.2.2 用户定义标题

Syntax: TITLE descritpion statment
说明：TITLE后面为标题的说明部分。

Example:

TITLE This is a demo code.

7.2.3 变量

EQU
语法：　　VARIABLE EQU VALUE or BIT
说明：固定变量声明后不能被修改。

Example:

```
TRUE    EQU    1
FALSE   EQU    0
PIN1    EQU    P0.0
NUM1    EQU    0X20+0X3
```

=
Syntax:　　VARIABLE = VALUE | BIT
说明：可变变量在编译过程中可以被修改。

Example:

```
TEMP = 0
TEMP = TEMP + 1 // TEMP = 1
TPIN = TEMP.7
```

TEXTEQU
Syntax:

```
STRING    TEXTEQU    <STRING>
STRING    TEXTEQU    TEXT MACRO
STRING    TEXTEQU    % VARIABLE
STRING    TEXTEQU    % (ARITHMETIC)
```

说明：文本宏能再次被更改，一般用于字符串的替换。符号"%"后的变量或算术表达式的值将被转换为字符串。Catstr、substr、sizestr和instr可以用来表示以上四种不同的字符串。

Example 1:

```
<STRING>
    CLRA      TEXTEQU    <MOV A,#0>
```

Example 2:

```
TEXT MACRO
CLRALU    TEXTEQU    CLRA
```

Example 3:

```
% VARIABLE
TEMP = 30
STR1   TEXTEQU    %TEMP          ; TEXTEQU <0x1E>
STR1   TEXTEQU    %0d:TEMP       ; TEXTEQU <30>
STR1   TEXTEQU    %0x:TEMP       ; TEXTEQU <1e>
```

Example 4:

```
 % (ARITHMETIC)
TEMP = 20
STR1   TEXTEQU    %(TEMP+6)      ; TEXTEQU <0x1A>
```

CATSTR

```
Syntax:    STRING    CATSTR    <STRING1>, <STRING2>
           STRING    CATSTR    TEXT MACRO 1, TEXT MACRO 2
           STRING    CATSTR    %VARIABLE1, %VARIABLE2
```

说明：执行CATSTR命令将两个字符串/文本宏结合起来生成一个新的文本宏。字符串的格式请参考TEXTEQU。

Example:

```
S1    TEXTEQU     <12>
S2    CATSTR      <0x>, S1          // S2 equivalent to "0x12"
```

SUBSTR

```
Syntax:    STRING    SUBSTR    <STRING>, START, [LENGTH]
           STRING    SUBSTR    TEXT MACRO, START, [LENGTH]
           STRING    SUBSTR    % VARIABLE, START, [LENGTH]
           STRING    SUBSTR    % (ARITHMETIC), START, [LENGTH]
```

说明：执行SUBSTR命令从STRING中抽取一个子字符串。Start为要取的字符的开始位置（原字符串的第一个字母为位置1）。Length表示要抽取出字符串的长度。如果省略长度，则一直取到字符串的最后一位。字符串的格式请参考TEXTEQU。

Example:

```
S1    TEXTEQU     <123456>
S2    SUBSTR      S1, 4, 2          // s2 equivalent to "45"
S3    SUBSTR      S1, 3             // s3 equivalent to"3456"
```

SIZESTR

```
Syntax:    VALUE    SIZESTR    <STRING>
```

```
VALUE   SIZESTR   TEXT MACRO
VALUE   SIZESTR   % VARIABLE
VALUE   SIZESTR   % (ARITHMETIC)
```

说明：执行SIZESTR命令可以从STRING得到字符串的长度。字符串的格式请参考TEXTEQU。

Example:

```
V1   SIZESTR   <123456>          // v1 Equivalent to 6
```

INSTR

Syntax:

```
VALUE   INSTR   START, <STRING>, <SUBSTRING>
VALUE   INSTR   START, TEXT MACRO, <SUBSTRING>
VALUE   INSTR   START, % VARIABLE, <SUBSTRING>
VALUE   INSTR   START, % (ARITHMETIC), <SUBSTRING>
```

说明：从STRING中得到子字符串的位置。STRING的第一个字符的位置为1。如果找不到子字符串，其值为0。格式请参阅TEXTEQU。

Example:

```
S1   TEXTEQU   <12,34,56>
V1   INSTR     1, S1, <,>          // v1 equivalent to 3
V2   INSTR     V1+1, S1, <,>       // v2 equivalent to 6
```

V1 在<12，34，56>字串的第1个字符开始查找< , >，结果为3；
V2 在<12，34，56>字串的第V1+1个字符开始查找< , >，结果为6；

7.2.4 段定义

Syntax:

```
.CODE
.DATA
.CONST
```

说明：设置当前位置为代码段(.code)、数据段(.data)或常数段(.const)。代码段的大小取决于ROM区的大小，数据段的大小取决于RAM区的大小，但常数段没有大小限制，且各段之间可以相互替换。系统默认的程序段起始地址为0。

Example:

```
    .CODE
        MOV     A, #0
         …
.DATA
        RAM0    DS   1          // equivalent to ram0 equ 0
        RAM1    DS   1          // equivalent to ram1 equ 1
.CODE
        …
.DATA
        BUF0    DS   2          // equivalent to buf0 equ 2
        BUF1    DS   1          // equivalent to buf1 equ 4
.CODE
        …
```

ORG
Syntax: ORG NEW ADDRESS

通常，用户可以重新设置程序中的地址。如果在程序开头没有指定程序地址，系统将默认其为0。

Example:

```
        MOV     A,  #0FH
        B0MOV   STKP,  A        // disable inter rupt, set stack to bottom
        B0MOV   PFLAG, #0       // at the first 16k rom
        JMP     START
ORG     8                       // t0, t0c, t1c, p0 … interruption's starting ddress.
        CLR     INTRQ           // clear all interrupt request
        RETI
START:
        …
        ORG     _TMP = $
        ORG     ($+15) & 0X3FF0     // re-position 16 times alignment
        DW      …
        ORG     ORG_TMP + 0X100     // re-position 0x100 apart…
```

.ALIGN
Syntax: .ALIGN NUMBER

说明：.ALIGN指令用来调整下一条指令或变量的边界值。Number必须是2的乘方，如2、16、256，最大为65536（0x10000）。

Example:

```
// IF $ == 7
. ALIGN          16
//SAME AS ABOVE ORG ($+15) & 0X3FF0
// THEN $ == 16
```

7.2.5 数据定义

DB

语法: [LABEL] DB D1 [, D2 , …]

说明：在程序中定义数据。数值必须在范围0~0XFF之间，字符串必须用符号“”框起来。两个字节组成一个字：第一个字节为低字节，第二个字节为高字节。如果不够一个字，高字节设为0。

Example:

```
DB   12, 34, 30, “ABCD”
equivalent to
DW 0X220C, 0X411E, 0X4342, 0X0044
```

DW

语法: [LABEL] DB D1 [, D2 , …]
[LABEL] DW “STRING” [, “STRING”, …]

说明：在程序中定义数据。数值必须在0~0xffff之间，字符串必须用符号“”括起来。两个字节组成一个字：第一个字节为低字节，第二个字节为高字节。如果不够一个字，高字节设为0。

Example:

```
TEMP = 45
DW      0X1234, 5678H, TEMP+3, 2*5
DW      “ABCDEFG”, 23H
HERE    DW      “HERE”, “SONIX”
 …
B0MOV   X,  #HERE$H
B0MOV   Y,  #HERE$M
B0MOV   Z,  #HERE$L
MOVC                        // ACC = ‘H’, R = ‘E’
```

DD

语法: [LABEL] DD D1 [, D2 , …]
[LABEL] DW “STRING” [, “STRING”, …]

说明：在程序中定义数据。数值必须在0~0xffffffff之间，字符串必须用符号“”括起来。两个字节组成一个双字，用来保存标号（因为标号的内容经常超过64K）。

Example:

```
TABLE    DD     L1, L2, L3
 …
 L1      DW     “HELLO”
 L2      DW     “GOOD”
 L3      DW      “SONIX”
```

DS

语法: **[LABEL] DS SIZE**

说明：用DS在RAM中定义数据。Size指数据在RAM中占用空间的的大小。

Example:

```
    BUFFER1   DS   4
    XBUF      DS   0          // XBUF EQUIVALENT TO BUFFER2
    BUFFER2   DS   8
.CODE
    ….
    B0MOV     H, #BUFFER1$M
    B0MOV     L, #BUFFER1$L
    MOV       A,   DP0X       // ACC = [BUFFER]
```

7.2.6 位运算函数

语法: **@BIT(parameter);**
@INT(parameter);
@FIELD(parameter)

说明：位操作数，用@bit()截取位部分，用@int()截取整数部分，用@fied()截取位的字段。

Example:

```
P_XOR    EQU     P1.4
…
B0BSET @INT(P_XOR)-0X10.@BIT(P_XOR)
// b0bset p1m.4设置成为输出模式
…
MOV     A,    #@FIELD(P_XOR)      // MOV A, #10H; (BIT4)
XOR     @INT(P_XOR),   A          // XOR P1, A
```

注：@INT(P_XOR)即截取P_XOR的整数部分，即为P1(0xD1)。@INT(P_XOR)-0x10结果则为0xC1(即为P1M)。而@BIT(P_XOR)即为4，而@INT(P_XOR)-0x10.@BIT(P_XOR)则为P1M.4。@FIELD(P_XOR)为Bit4，Bit4为1，即为10H。

7.3 编译伪指令

- 条件编译伪指令

使用条件编译指令可以检测一些特定条件，且当条件为真时执行命令 IF 和 ENDIF 之间的指令。可选择的 ELSE 命令跟在 IF 命令后面，并支持嵌套。

语法: **IF expression_1**
statements_1
{ ELSEIF expression_2
statements_2}
{ ELSE
statements_3}
ENDIF

说明：指令 IF 和 ELSE 之间的程序指令可以是任意有效指令或命令。如果相应的条件表达式为真，处理器将执行 IF 命令后的指令；如果表达式为假，处理器只执行 ELSE 命令后的指令（如果程序中有 ELSE 命令），否则处理器没有任何动作。下表为其他条件编译命令。

Directive 指令	Condition 条件
IF expression	表达式为真
IFE expression	表达式不为真
IFDEF symbol	被定义为真
IFNDEF symbol	未被定义为真
IFB argument	为空时为真
IFNB argument	不为空时为真
IFIDN arg1, arg2	arg1==arg2 时为真
IFIDNI arg1, arg2	arg1==arg2（不分大小）时为真
IFDIF arg1, arg2	arg1<>arg2 时为真
IFDIFI arg1, arg2	arg1<>arg2（不分大小）时为真

- **Include Directives**

语法: **[INCLUDE] filename**
[INCLUDE] "long file path name"
[INCLUDE] <long file path name>

这里的[INCLUDE]定义如下：

INCLUDE, INCLUDESTD, INCLUDEOS, INCLUDEBIN,
INCLUDEWAV, INCLUDEPCM (For SNC8X only)
INCLUDEMLD, INCLUDEMD4 (For DSP only)

说明：INCLUDE 指令中的文件名必须完整指定，指定的文件类型可以是 DBT（*.dbt）格式。DBT 文件是被内部二进制加密程序加密的源文件。

一个包含文件可以指定其它包含文件。处理器先执行里层的包含文件，再执行外层的包含文件。程序中可以容纳剩余内存允许的最多层的包含文件。如果指定的包含文件不在源文件目录下，处理器将自动根据命令行选项或 SN8 C STUDIO 选项对话框中列出的路径寻找此包含文件。

- **File Control Directives**

语法: **.LIST**
.NOLIST

说明：命令**.LIST** 和**.NOLIST** 决定是否将源文件包含在程序列表文件中。 **.NOLIST**：禁止将后面的源文件包含在程序列表文件中。**.LIST**：将后面的源文件恢复在程序列表文件中。

语法: **.LISTMACRO**
.NOLISTMACRO

说明：执行命令**.LISTMACRO**，交叉编译器将列出一个宏文件中包括注释的所有宏指令。命令**.NOLISTMACRO** 禁止列出任何宏指令，这是默认设定值。

语法: **.LISTIF**
.NOLISTIF

说明：命令**.LISTIF** 列出包含的条件命令，**.NOLISTIF** 禁止列出条件命令。

8. SN8 C Language

8.1 Overview

8.1.1 C 源程序的结构特点

1. 一个 C 语言源程序可以由一个或多个源文件组成。
2. 每个源文件可由一个或多个函数组成。
3. 一个源程序不论由多少个文件组成，都有一个且只能有一个 main 函数，即主函数。
4. 源程序中可以有预处理命令(include 命令仅为其中的一种)，预处理命令通常应放在源文件或源程序的最前面。
5. 每一个说明，每一个语句都必须以分号结尾。但预处理命令，函数头和花括号"}"之后不能加分号。
6. 标识符，关键字之间必须至少加一个空格以示间隔。若已有明显的间隔符，也可不再加空格来间隔。

书写程序时应遵循从书写清晰，便于阅读，理解，维护的角度出发，在书写程序时应遵循以下规则：

1. 个说明或一个语句占一行。
2. {} 括起来的部分，通常表示了程序的某一层次结构。{}一般与该结构语句的第一个字母对齐，并单独占一行。
3. 一层次的语句或说明可比高一层次的语句或说明缩进若干格后书写。以便看起来更加清晰，增加程序的可读性。在编程时应力求遵循这些规则，以养成良好的编程风格。

8.1.2 C 语言的字符集

字符是组成语言的最基本的元素。C 语言字符集由字母，数字，空格，标点和特殊字符组成。在字符常量，字符串常量和注释中还可以使用汉字或其它可表示的图形符号。

1. 字母
   小写字母 a～z 共 26 个，大写字母 A～Z 共 26 个
2. 数字
   0～9 共 10 个
3. 白符
   空格符、制表符、换行符等统称为空白符。空白符只在字符常量和字符串常量中起作用。在其它地方出现时，只起间隔作用， 编译程序对它们忽略。因此在程序中使用空白符与否，对程序的编译不发生影响，但在程序中适当的地方使用空白符将增加程序的清晰性和可读性。
4. 点和特殊字符

8.1.3 C 语言词汇

在 C 语言中使用的词汇分为六类：标识符，关键字，运算符，分隔符，常量，注释符等。

1. 标识符

在程序中使用的变量名、函数名、标号等统称为标识符。除库函数的函数名由系统定义外，其余都由用户自定义。C 规定，标识符只能是字母(A～Z，a～z)、数字(0～9)、下划线()组成的字符串，并且其第一个字符必须是字母或下划线。

以下标识符是合法的：

a,x, 3x,BOOK 1,sum5

以下标识符是非法的：

3s 以数字开头
s*T 出现非法字符*
-3x 以减号开头
bowy-1 出现非法字符-(减号)

在使用标识符时还必须注意以下几点：

(1). 标准 C 不限制标识符的长度，但它受各种版本的 C 语言编译系统限制，同时也受到具体机器的限制。例如在某版本 C 中规定标识符前八位有效，当两个标识符前八位相同时，则被认为是同一个标识符。
(2). 在标识符中，大小写是有区别的。例如 BOOK 和 book 是两个不同的标识符。
(3). 标识符虽然可由程序员随意定义，但标识符是用于标识某个量的符号。因此，命名应尽量有相应的意义，以便阅读理解，作到"顾名思义"。

2.关键字

关键字是由 C 语言规定的具有特定意义的字符串，通常也称为保留字。用户定义的标识符不应与关键字相同。C 语言的关键字分为以下几类：

类型说明符

用于定义、说明变量、函数或其它数据结构的类型。如前面例题中用到的 int,double 等
语句定义符

用于表示一个语句的功能。如例 1.3 中用到的 if else 就是条件语句的语句定义符。

预处理命令字
用于表示一个预处理命令。如前面各例中用到的 include。

Keyword	Related Chapter
__interrupt	Interrupt Functions
__RAM	Memory Qualifier
__ROM	Memory Qualifier
#progma bank	Bank Configuration

Figure 7-7

3.运算符

C 语言中含有相当丰富的运算符。运算符与变量，函数一起组成表达式，表示各种运算功能。运算符由一个或多个字符组成。

4.分隔符

在 C 语言中采用的分隔符有逗号和空格两种。逗号主要用在类型说明和函数参数表中，分隔各个变量。空格多用于语句各单词之间，作间隔符。在关键字，标识符之间必须要有一个以上的空格符作间隔， 否则将会出现语法错误，例如把 int a;写成 inta;C 编译器会把 inta 当成一个标识符处理，其结果必然出错。

5.常量

C 语言中使用的常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等多种。

6.注释符

块注释 :C 语言的注释符是以"/*"开头并以"*/"结尾的串。在"/*"和"*/"之间的即为注释。程序编译时，不对注释作任何处理。注释可出现在程序中的任何位置。注释用来向用户提示或解释程序的意义。在调试程序中对暂不使用的语句也可用注释符括起来，使翻译跳过不作处理，待调试结束后再去掉注释符。
行注释：行注释以"//"开头，本行"//"后面的部分即为注释。

8.2 数据类型

虽然 SN8 C 支持 C 的所有数据类型，但是由于它面对的是 8-bit 单片机，所以必然会考

虑数据类型的定义方法和长度。在这些方面，SN8 C 有它自己的专有的定义特征和数据长度，在使用时一定要加于区分。请看下表：

Data Type	Size（Byte）	Range of the data
Signed char（short、int）	1	-128~+127
Unsigned char(short、int)	1	0~255
Signed long	2	-32768~+32767
Unsigned long	2	0~65535
float、double	4	
Pointer	2	
enum	1	

表 3-1 数据类型长度定义表

8.2.1 常量与变量

在程序设计的过程中，我们有些量可能是参考值，也可能是预设的值。总之，我们希望它在整个程序中保持不变，并且在程序的任何地方可以提供我们调用，用来比对某些条件是否成立等等。

对于这样的值，我们可以不去定义它，而直接参考数值，这在汇编编程过程中，经常会有一些缺乏经验的程序员这样用。但这就需要程序员把程序中的这些值都牢牢记住，并且要将他们的值前后统一，这是一个非常容易出错的过程，而且也会影响了程序的可读性，这样，程序的修改和维护都非常艰难，必须处处小心翼翼，一处不慎就全盘错误！这是任何人不想看到的后果。因此，建议对于程序中用到的不会改变的参考值或其他预设的值都进行一个预先的定义，给它取一个有意义的或与其相关的名字。这个过程就是常量定义，自然，这个不变的量就称为常量。

在标准 C 中，由于面对的是功能强大的 CPU 系统和大内存，用户可以不去管它放置的地方。但是，面对一个单片机系统，它的 Ram 非常小，有时会显得很紧张。所以系统为了节省空间会将一些表格等放在系统的 ROM 中。而我们直接命名的常量，则由编译器自动将其替换为所需要的值，这些工作就由计算机来完成好了。

我们先来看看我们用汇编编写程序时是怎么来定义常量的：

```
door_service_c      equ     #80     ;80ms
t0int_c             equ     #224    ;t0 interupt time
segment_c           equ     #3      ;3 cooks at most
```

注：上面数值前的#号，是 SN8 Assembler 的符号，用于提示后面的是立即数。

上面定义了 3 个程序中会用到的参考值，顺便提一下，在定义的时候加上注释是有必要的，要不然时间久了你就又不知道你定义的到底是什么了。在上面的定义中，用的是汇编 EQU 关键字，在编译过程中，程序里但凡出现了 EQU 前面的字段都会被其后面的值直接代替，因此，很方便地减轻了程序员的工作。

我们再来看看用 SN8 C 是如何定义这些相同的常量的：

```
#define   door_service_c    80      //80ms
#define   t0int_c           224     //t0 interupt time
#define   segment_c         3       //3 cooks at most
```

上面定义的 3 个量是与前面汇编当中定义的 3 个常量是完全相同的。在进行编译预处理时，这些量就会被数值代替。

还有一个特殊的地方，那就是一些数值列表，在汇编当中，查表项都是放在 Code 当中的，作为 Code 来处理，其实这些值也是常量，只不过他们的处理不同于一般常量而已，并且它们共用一个入口。下面是一个汇编的表：

```
disp_automenu:      ;Table
   dw   0000h
   dw   0ae1fh      ;A-1
   dw   0ae2fh      ;
   dw   0ae3fh      ;
   dw   0ae4fh      ;
   dw   0ae5fh      ;
   dw   0ae6fh      ;
   dw   0ae7fh      ;
```

我们可以看到，汇编的表是用 DW 关键字定义一个 word，它是存放在.code 段里面的，通过表头地址来得到每一个相对应的值。

那么在 SN8 C 里面又如何来处理这些表呢？

在讲到表的处理之前，必须先提一提变量定义关键字的问题。SN8 C 定义一个变量时，可以指明它所放置的地方（RAM 或 ROM），分别用关键字__RAM 和__ROM 来指定存放的地点，如：

```
Unsigned int __RAM ramVariable;      //将变量存放在 RAM 中 Store the variables in RAM
```

```
__RAM unsigned int ramVariable2;
Unsigned int __ROM romVariable;      //将变量存放在 ROM 中 Store the variables in ROM
__ROM unsigned int romVariable2;
```

我们可以知道，当一个量放到了 ROM 当中就没法改变它的值了，其实就是我们说的常量。

在 C 当中，可以通过一个头名称来访问的变量类型比较多，其中数组是比较方便的一种，我们可以通过定义一个数组来存储这些表的数值，然后通过对数组的访问来查询对应的值。

```
unsigned long __ROM disp_automenu[]=                    //Table
{
    0x0000,0x0ae1f,0x0ae2f,0x0ae3f,0x0ae4f,
    0x0ae5f,0x0ae6f,0x0ae7f
};
```

这是一个与上面的汇编表完全相同的表，我们将它存放在__ROM 中，通过调用数组来查表，这在后面将详细介绍。

而在程序中还有另外一类的量是会在程序中不断被改变的，比如程序中的计数器，状态寄存器等等都会随着程序的运行而改变。我们将这一类量称之为变量。

我们先来看看汇编的定义变量的方法：

```
.DATA
    org     0h
    temp1           ds   1
    temp2           ds   1
    temp3           ds   1
    temp4           ds   1
    led_dp          ds   1
    step            ds   1     ; current status
```

上面的代码定义了 temp1、temp2、temp3、temp4、led_dp、step6 个变量，它们分别占用一个 Byte 的 RAM 空间，那么程序当中就可以通过变量名对该变量的空间进行读写。当然在汇编中你也可以用一个变量名来访问两个或多个 RAM 空间，这类似于查表的操作，其定义如下：

```
Job_mode        ds   2
Power_mode      ds   4
```

对上面的 job_mode 变量可以通过 job_mode 和 job_mode+1 来读写定义的两个存储单元，以此类推 power_mode 或其他多个 RAM 空间定义的变量可以通过相同的方法来定义。可见，SN8 Assembler 定义变量的方法主要是通过 DS 关键字来申请需要的变量空

间，空间一旦被申请，就在整个程序流程里面被占用，也就是说定义一个变量就少一个空间，这对于 RAM 本身就很少的单片机而言，不能不说是一种浪费。

那么 SN8 C Compiler 又是如何来定义变量的呢？

要说到 C 的变量，就不能不提一提变量的有效作用域。

我们知道，汇编当中定义的变量都是在整个程序中有效的，在程序中任何地方都可以改变变量的值，这样程序员就会经常遇到这样的情况：在调试程序时发现变量的值有误，却无法判断变量到底在什么地方被错误赋值或被赋予了错误的值，从而不得不在整个程序中到处设置断点，运行多次才找出问题的所在。这就是汇编当中变量定义造成不方便的地方！

而 C 的变量的定义区分了作用域，分为全局变量和局部变量，而区分定义这两种变量的方法很简单，只是在不同的位置定义就行了，这与标准 C 所规定的方法是一样的，后面再详细讨论。

变量说明的一般形式为： 类型说明符 变量名标识符，变量名标识符，...; 例如：

下面是 SN8 C 定义的几个变量：

```
unsigned int temp1;
unsigned int temp2;
unsigned int temp3;
unsigned int temp4;
unsigned int led_dp;
unsigned int step;                  //Current status
unsigned long job_mode;
unsigned long power_mode1;
float powerValue;
int temp1_1;                        //Signed int type
long temp2_2;
```

在书写变量说明时，应注意以下几点：

1. 允许在一个类型说明符后，说明多个相同类型的变量。各变量名之间用逗号间隔。类型说明符与变量名之间至少用一个空格间隔。
2. 最后一个变量名之后必须以“；”号结尾。
3. 变量说明必须放在变量使用之前。一般放在函数体的开头部分。

从上面的定义可以看到 C 的变量不仅区分了作用域，还有不同长度的变量类型，这样就方便了程序员的使用。

与面向数学运算的计算机相比，单片机的编程对变量类型或数据类型的选择更具有关键性意义。SN8 系列单片机是 8-bit 处理器的单片机，只有 Byte 型的数据是处理器直接支持的。对于 C 这样的高级语言，不管使用何种数据类型，虽然某一行程序从字面上看，其操作十分简单，然而，实际上系统的 C 编译器需要用一系列机器指令对其复杂的变量类型、数据类型的进行处理。相同的一行语句，变量选择的类型不同，处理时就会产生很大的差别，产生的代码更是差别很多。特别是当使用浮点变量时，将明显地赠加运算时间和程序的长度。当程序必须保证运算精度时，C 编译器将调用相应的子程序库，把它们加到程序中。然而许多不熟练的程序员，在编写 C 程序时往往会使用大量的、不必要的变量类型。这就导致 C 编译器相应地增加所调用的库函数以处理大量增加的变量类型，并最终导致程序变得过于庞大，运行速度减慢，甚至因此会在 Link 时，出现因程序过大而装不进 ROM 的情况。所以必须特别慎重的选择变量和数据类型。

而对于有符号与无符号的变量类型。在编写程序时，如果使用 signed 和 unsigned 两种数据类型，那么就得使用两种格式类型的库函数。这将使得占用的存储空间成倍增长。因此在编程时，如果只强调程序的运算速度而又不进行负数运算时，最好采用无符号（unsigned）格式。

Note：
1. Please choose the data types which need less memory if they can accomplish the functions all right.This will bring us profit when we use the RAM memory and improve the efficiency of code generating.
2. The unsigned types are preferred, and this can avoid some errors, as the data inside the chip is handled as unsigned type.
3. Attention!C language is case-sensitive, and a rule for naming the variables should be followed at the beginning. Hump notation is a good choice, but assembler programmers may not be accustomed to it at the beginning.

当然，在标准 C 编程当中，经常会有为书写方便，对数据类型进行缩写定义，这在 SN8 C 也是允许的。如：

```
#define uchar unsigned char
#define uint unsigned int
```

这样，在以后的编程中，就可以用 uchar 代替 unsigned char，用 uint 代替 unsigned int 来定义变量。

8.2.2 数据的存储类型与存储结构

在前面分析查表类型数据定义的时候已经提到了数据在单片机里的存储，会分为 ROM 和 RAM 两部分，我们分别称之为程序存储器（ROM）和数据存储器（RAM）。它们在

我们编写汇编的时候，会分别用关键字.code 和.data 来预先声明。

通常 SN8 系列单片机不用扩展存储器，它提供种类丰富的机型，你完全可以根据你的需求来选择合适的机型。因此，数据只存储在片内，寻址也不存在片内片外的区分。

SN8P 单片机的数据存储器（RAM）的结构如下：

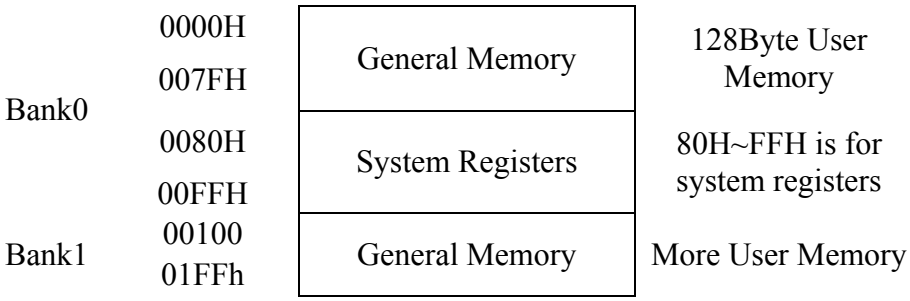


图 3-2、RAM structure

SN8P 系列单片机的通用 RAM 区的大小依不同的芯片而不同，但都是按 BANK 来划分，每一个 BANK 内的地址为 00H~FFH。而所有芯片的 Bank0 的 80H~ FFH 的空间都是留给系统专用的系统寄存器区。

其实我们上面定义的变量都是放在 RAM 区的数据结构，在定义一个变量时，我们前面已经提到可以用__RAM 和__ROM 关键字来指定存储的地方，而变量定义一般都放在 RAM 当中，因而__RAM 关键字是缺省项。看下面的例子：

```
unsigned int temp1;
unsigned int __RAM temp2;
__RAM unsigned int temp3;
unsigned long job_mode;
unsigned long __RAM job_mode2;
__RAM unsigned long job_mode3;
float powerValue;
float __RAM powerValue;
__RAM float powerValue;
int temp1_1;                    //Signed int
int __RAM temp1-2;
__RAM temp1-3;
long temp2_2;
```

其实上面的定义都是相同的效果，就是定义变量，并将其放置在数据存储区（RAM）内。也就是说，我们在定义一个变量，按我们 C 常用的方法定义就自动地放到 RAM 里了，缺省项给我们带来了很大的方便。

上面已经说明了，在 RAM 存储区内，80H~FFH 是系统寄存器区。

根据不同的芯片的资源，寄存器的内容会随之改变，但是他们定义的区域都不变。这些寄存器都分别对应了芯片内部的资源，SN8 C 针对这部分的系统资源，对这些寄存器进行了定义，其定义的形式如下：

```
#define   L     (*((__RAM unsigned int*)0x80))
#define   H     (*((__RAM unsigned int*)0x81))
#define   R     (*((__RAM unsigned int*)0x82))
#define   Z     (*((__RAM unsigned int*)0x83))
#define   Y     (*((__RAM unsigned int*)0x84))
#define   X     (*((__RAM unsigned int*)0x85))
#define   PFLAG   (*((__RAM unsigned int*)0x86))
```

这些定义都被包含在相应芯片的头文档里（.h），因此并不需要用户自己去定义寄存器的相应名称，只需要在程序的开头包含相应的头文档就行了。如下所示：

```
#include <sn8p2708a.h>
```

Note：
这些系统寄存器都是以大写字母进行定义的，在编写程序时要注意这一点。

8.2.3 Bank Configuration

SN8P 支持 RAM BANK 配置，准确地把变量存放到区域中。

RAMBANK n Model
这个模式供用户对指定 RAM BANK 中的全局变量和静态变量的地址。例如：

```
//following variables are all static or global.
//where the "n" is the bank number (0~255)
//"default" means decided by linker.
#pragma rambank 3
int X = 123; //X is in bank 3
#pragma rambank 27
static int Y = 123; //Y is in bank 27
#pragma rambank off
//from now on，all new defined static or global variables will be
//located by linker.
```

SN8 C COMPILER 每次在访问指定的区中的变量前这种区切换指令。编译器具有删除无用区切换指令这一优化功能。

8.3 基本运算符和表达式

C 语言中运算符和表达式数量之多， 在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言功能十分完善。 这也是 C 语言的主要特点之一。

C 语言的运算符不仅具有不同的优先级， 而且还有一个特点，就是它的结合性。在表达式中， 各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定，还要受运算符结合性的制约， 以便确定是自左向右进行运算还是自右向左进行运算。 这种结合性是其它高级语言的运算符所没有的，因此也增加了 C 语言的复杂性。

运算符的种类 C 语言的运算符可分为以下几类：

8.3.1 算术运算符与算术表达式

用于各类数值运算。包括加(+)、减(-)、乘(*)、除(/)、求余(或称模运算，%)、自增(++)、自减(--)共七种。

算术表达式是由算术运算符和括号连接起来的式子， 以下是算术表达式的例子：

a+b (a*2)／c (x+r)*8-(a+b)／7 ++i sin(x)+sin(y) (++i)-(j++)+(k--)

8.3.2 关系运算符与关系表达式

用于比较运算。包括大于(>)、小于(<)、等于(==)、 大于等于(>=)、小于等于(<=)和不等于(!=)六种。

关系表达式的一般形式为： 表达式 关系运算符 表达式 例如：a+b>c-d,x>3/2,'a'+1(b>c),a!=(c==d)等。关系表达式的值是“真”和“假”，用“1”和“0”表示。

如： 5>0 的值为“真”，即为 1。(a=3)>(b=5)由于 3>5 不成立，故其值为假，即为 0。

字符变量是以它对应的 ASCII 码参与运算的。对于含多个关系运算符的表达式，如 k==j==i+5,根据运算符的左结合性 ,先计算 k==j，该式不成立 ,其值为 0 ,再计算 0==i+5，也不成立，故表达式值为 0。

8.3.3 辑运算符与逻辑表达式

逻辑运算符 C 语言中提供了三种逻辑运算符 && 与运算 || 或运算 ！非运算 与运算符&&和或运算符||均为双目运算符。具有左结合性。 非运算符!为单目运算符，具有右结合性。

逻辑运算的值

逻辑运算的值也为“真”和“假”两种，用“1”和“0 ”来表示。其求值规则如下：

1. 运算&&参与运算的两个量都为真时，结果才为真，否则为假。例如，5>0 && 4>2, 由于 5>0 为真，4>2 也为真，相与的结果也为真。
2. 运算||参与运算的两个量只要有一个为真，结果就为真。 两个量都为假时，结果为假。例如：5>0||5>8,由于 5>0 为真，相或的结果也就为真
3. 运算!参与运算量为真时，结果为假；参与运算量为假时，结果为真。
   例如：!(5>0)的结果为假。

虽然 C 编译在给出逻辑运算值时，以“1”代表“真”，“0 ”代表“假”。 但反过来在判断一个量是为“真”还是为“假”时，以“0”代表“假”，以非“0”的数值作为“真”。例如：由于 5 和 3 均为非“0”因此 5&&3 的值为“真”，即为 1。
又如：5||0 的值为“真”，即为 1。

逻辑表达式的一般形式为：表达式 逻辑运算符 表达式

其中的表达式可以又是逻辑表达式，从而组成了嵌套的情形。例如：(a&&b)&&c 根据逻辑运算符的左结合性，上式也可写为： a&&b&&c 逻辑表达式的值是式中各种逻辑运算的最后值，以“1”和“0”分别代表“真”和“假”。

8.3.4 位操作运算符

参与运算的量，按二进制位进行运算。包括位与(&)、位或(|)、位非(~)、位异或(^)、左移(<<)、右移(>>)六种。

8.3.5 赋值运算符与赋值表达式

赋值运算符用于赋值运算，分为简单赋值(=)、复合算术赋值(+= , -= , *= , /= , %=)和复合位运算赋值(&= , |= , ^= , >>= , <<=)三类共十一种。

简单赋值运算符和表达式

简单赋值运算符记为"="。由"= "连接的式子称为赋值表达式。其一般形式为： 变量= 表达式 例如：

```
x=a+b
w=sin(a)+sin(b)
y=i+++--j
```

赋值表达式的功能是计算表达式的值再赋予左边的变量。
赋值运算符具有右结合性。因此：

```
a=b=c=5
```

可理解为 a=(b=(c=5))

在其它高级语言中，赋值构成了一个语句，称为赋值语句。 而在 C 中，把"="定义为运算符，从而组成赋值表达式。凡是表达式可以出现的地方均可出现赋值表达式。例如，式子 x=(a=5)+(b=8)是合法的。它的意义是把 5 赋予 a，8 赋予 b，再把 a,b 相加，和赋予 x ，故 x 应等于 13。

在 C 语言中也可以组成赋值语句，按照 C 语言规定， 任何表达式在其末尾加上分号就构成为语句。因此如 x=8;a=b=c=5；都是赋值语句，在前面各例中我们已大量使用过了。

如果赋值运算符两边的数据类型不相同， 系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

1. 型赋予整型，舍去小数部分。前面的例 2.9 已经说明了这种情况。
2. 型赋予实型，数值不变，但将以浮点形式存放， 即增加小数部分(小数部分的值为 0)。
3. 符型赋予整型，由于字符型为一个字节， 而整型为二个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。
4. 型赋予字符型，只把低八位赋予字符量。

复合赋值符及表达式

在赋值符"="之前加上其它二目运算符可构成复合赋值符。如 +=、-=、*=、／=、%=、<<=、>>=、&=、^=、|=。 构成复合赋值表达式的一般形式为： 变量 双目运算符=表达式 它等效于 变量=变量 运算符 表达式 例如：

```
a+=5      equals    a=a+5
x*=y+7    equals    x=x*(y+7)
r%=p      equals    r=r%p
```

复合赋值符这种写法，对初学者可能不习惯， 但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

8.3.6 条件运算符

条件运算符为?和：，它是一个三目运算符，即有三个参与运算的量。由条件运算符组成条件表达式的一般形式为：

表达式 1? 表达式 2： 表达式 3

其求值规则为：如果表达式 1 的值为真，则以表达式 2 的值作为条件表达式的值，否则以表达式 3 的值作为整个条件表达式的值。 条件表达式通常用于赋值语句之中。

例如条件语句：

```
if ( a > b) max=a;
else        max=b;
```

可用条件表达式写为 max = (a>b)?a:b; 执行该语句的语义是：如 a > b 为真，则把 a 赋予 max，否则把 b 赋予 max。

```
One can write
max = (a > b) ? a : b ;
```

使用条件表达式时，还应注意以下几点：

1. 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。因此 max = (a > b) ? a : b 可以去掉括号而写为 max = a > b ? a : b
2. 条件运算符?和：是一对运算符，不能分开单独使用。
3. 条件运算符的结合方向是自右至左。

8.3.7 逗号运算符

用于把若干表达式组合成一个表达式(，)。

C 语言中逗号“，”也是一种运算符，称为逗号运算符。 其功能是把两个表达式连接起来组成一个表达式， 称为逗号表达式。

其一般形式为： 表达式 1，表达式 2 其求值过程是分别求两个表达式的值，并以表达式 2 的值作为整个逗号表达式的值。

```
void main()
```

```
{
  int a=2 , b=4 , c=6 , x , y;
  y = (x = a+b),(b+c);
  printf ("y=%d , x=%d" , y , x);
 }
```

a<--2 , b<--4 , c<--6 , x<--0 , y<--0
x<--a+b , y<---b+c

本例中，y 等于整个逗号表达式的值，也就是表达式 2 的值，x 是第一个表达式的值。对于逗号表达式还要说明两点：

1. 逗号表达式一般形式中的表达式 1 和表达式 2 也可以又是逗号表达式。例如： 表达式 1，(表达式 2，表达式 3) 形成了嵌套情形。因此可以把逗号表达式扩展为以下形式： 表达式 1，表达式 2，…表达式 n 整个逗号表达式的值等于表达式 n 的值。
2. 程序中使用逗号表达式，通常是要分别求逗号表达式内各表达式的值，并不一定要求整个逗号表达式的值。
3. 并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明中，函数参数表中逗号只是用作各变量之间的间隔符。

8.3.8 指针运算符

用于取内容(*)和取地址(&)二种运算。

8.3.9 求字节数运算符

用于计算数据类型所占的字节数(sizeof)。

8.3.10 特殊运算符

有括号()，下标[]，成员(→，.)等几种。

8.3.11 优先级和结合性

C 语言中，运算符的运算优先级共分为 15 级。1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。而在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。C 语言中各运算符的结合性分为两种，即左结合性(自左至右)和右结合性(自右至左)。例如算术运算符的结合性是自左至右，即先左后右。如有表达式 x-y+z 则 y 应先与"-"号结合， 执行 x-y 运算，然后再执行+z 的运算。这种自左至右的结合方向就称为"左结合性"。而自右至左的结合方向称为"右结合性"。 最典型的右结合性运算符是赋值运算符。如 x=y=z,由于"="的右结合性，应先

执行 y=z 再执行 x=(y=z)运算。 C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。

Precedence	Operators	Association
From High to low	( ) [ ] -> .	left to right
	! ~ ++ -- （类型） sizeof + - * &	**right to left**
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	**right to left**
	?:	**right to left**
	= += -= *= /= %= &= ^= |= <<= >>=	left to right

8.4 程序流程控制

程序流程控制是程序的精华所在，正确的程序流程才能实现正确的程序功能。安排精巧的程序流程控制才可能使程序具有高效率。程序流程控制不管是在 C 还是在汇编中都是程序设计者最值得思考的地方。

8.4.1 顺序结构

顺序结构是程序最基本的流程，其语句顺序执行，PC 指针依次下移，是 CPU 内部处理指令的初始方法。这也是编程思维的最初级方式，也是程序的最基本方式和流程。

顺序结构流程：

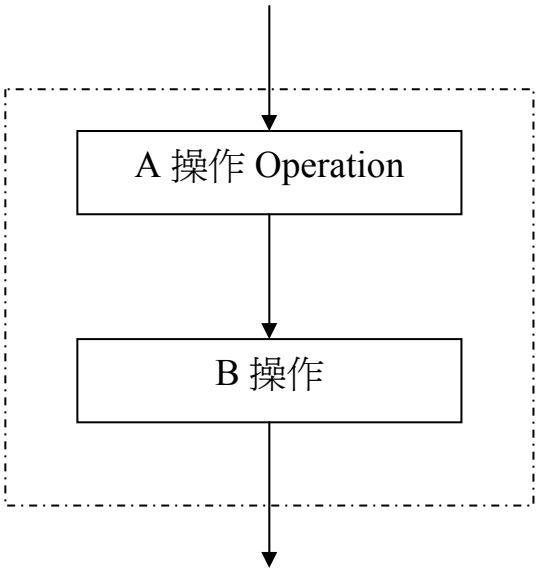


图 Figure 4-1 Sequential Program Flowing Chart

其实，其描述的是一个向量方式发展的问题，是所有问题发展和描述的基本方法，具有明确的方向性和时间性，如图中的 A 操作的发展方向只有一个，即接下来就是 B 操作，中间既没有反复也没有曲折变化，B 一定发生在 A 的后面。

我们平时可以看到的 C 语言的实现：（微波炉开机初始化）

```
key_bibi_f = 1;              // Beep once when electrified
menu_disp_h = 0xf0U;         //Display "0:00" when electrified
menu_disp_l = 0;
disp5 = 0xffU;
```

上述简短的几条语句分别完成一定目的的初始化，虽然在这里没有强制要求什么样的顺序，但是执行过程中是严格按语句的先后来开关目标的。

SN8 Assembler 的实现：

```
b0bset    key_bibi_f          ; Beep once when electrified

mov       a,#0f0h
mov       menu_disp_h,a
mov       a,#00h              ; Display "0:00" when electrified
mov       menu_disp_l,a

mov       a,#11111111b
mov       disp5,a
```

这是一段功能完全与上面这段 C 相同的汇编实现，我们可以看见在顺序结构的汇编语句当中没有任何的跳转和判断，都是按原功能的顺序来描述。

从对比当中我们完全可以找到它们的逻辑上的一一对应关系。

8.4.2 选择结构

事实上，完全顺序结构的流程的程序很少，因为很多事情并不是一帆风顺，顺流而下。很多情况下，事件的发生都需要具备一定的条件，只有一定条件下才有可能实现。这就得使用选择结构来描述。

if 语句

用 if 语句可以构成分支结构。它根据给定的条件进行判断， 以决定执行某个分支程序段。C 语言的 if 语句有三种基本形式。

1. 第一种形式为基本形式

```
if(表达式)  语句；
```

其语义是：如果表达式的值为真，则执行其后的语句， 否则不执行该语句。

2. 第二种形式为 if-else 形式

```
if(表达式)
   语句 1；
else
   语句 2；
```

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2 。其过程可表示为下图

选择结构流程图：

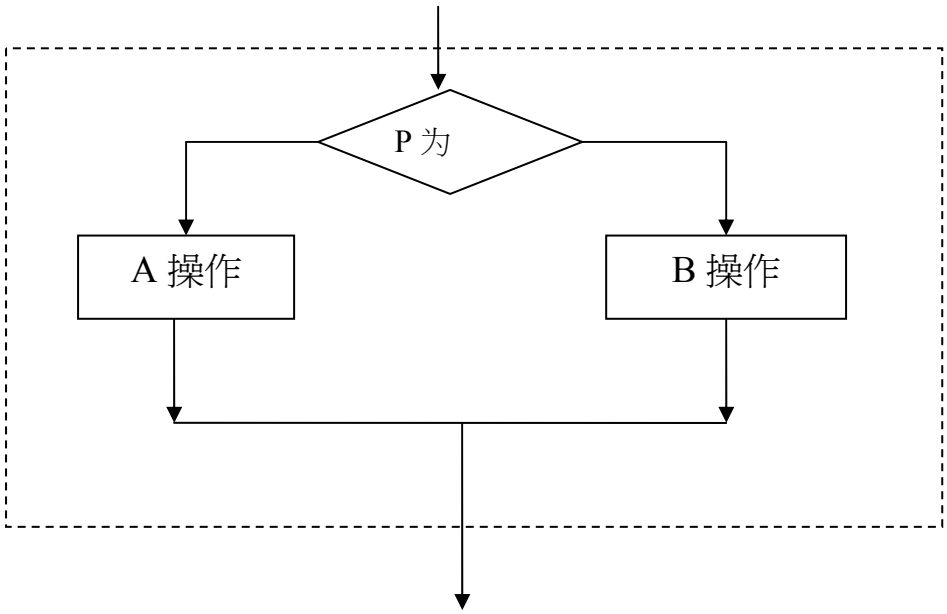


图 Figure4-2 Conditional Program Flowing Chart

简单选择结构只判断一个条件的真假来决定下面的需要执行的操作，这是一个最简单的判断。

C 语言实现：（BCD 码调整）

```
If (result_buf > 0x0a)
{
    result_buf = input + 6;
}
else
{
    result_buf = input;
}
```

我们看见在 C 语言当中，我们用判断语句来实现这样一个判断逻辑，不管是什么样的条件，我们只计算其真假，做出判断。If……else……是一个非常好的判断组合，很容易用它来完成下列我们用汇编来实现的功能。

SN8 汇编实现：

```
cmprs    a,#0ah
nop
b0bts0   fc
jmp      $+3
b0mov    a,y
```

```
        ret
        b0mov   a,y
        add     a,#6h               ;调整后的数放在 a
```

事实上，在汇编程序中，我们会遇到几种情况的条件选择：

一是当条件是一个位变量时，我们可以直接用 b0bts 指令来进行判断。
一是当条件是是否满足一个预定的值，我们可以用 CMPRS 指令来进行判断，当然也可以将其转化为标志为来判断，比如用 SUB 指令将其转化成 FC 或 FZ 标志位进行判断

A、 串行多分支结构流程：

事实上大多数时候，单个的条件无法分析判断复杂的问题，有时候一个结果的出现需要很多个条件同时成立，而每一个条件成立，又都有不同的结果产生，这样就会有一系列的有层次的判断！
前二种形式的 if 语句一般都用于两个分支的情况。 当有多个分支选择时，可采用 if-else-if 语句，其一般形式为：

```
if(condition1)
 statement1；
 else if(condition 2)
 statement 2；
 else if(condition 3)
 statement 3；
 …
 else if(condition m)
 statement m；
 else
 statement n；
```

其语义是：依次判断表达式的值，当出现某个值为真时， 则执行其对应的语句。然后跳到整个 if 语句之外继续执行程序。 如果所有的表达式均为假，则执行语句 n 。 然后继续执行后续程序。流程如图所示：

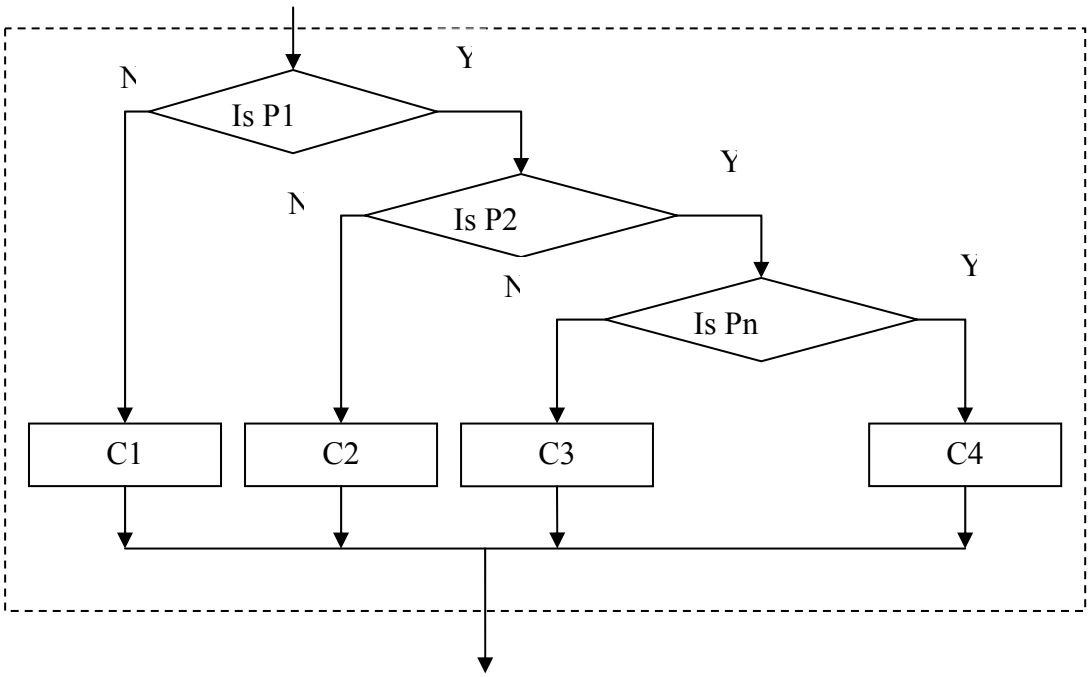


图 Figure4-3: Serial multiplied branches flowing chart

C 用 if，else if 嵌套来实现：

```
if(key_bibi_f)
{
    buzzer_time = d_buzzer_time1;       // Length of beeps, 200ms.
    buzzer_not = d_buzzer_not3;         // Leep times, #1
}
else if(end_bibi_f)                     //buzzer10
{
    buzzer_time = d_buzzer_time2;       // Length of beeps, 500ms
    buzzer_not = d_buzzer_not1;         // Beep times, #10
}
else if(segment_bibi_f)                 //buzzer20
{
    buzzer_time = d_buzzer_time2;       // length of beeps, 500ms
    buzzer_not = d_buzzer_not1;         // Beep times, #4
}
```

Explanation：

```
key_bibi_f          ;   Key-press beep request flag.
end_bibi_f          ;   Cook-finished beep request flag.
sgment_bibi_f       ; Segment shift beep request flag
key-press beeps:200ms/once
cook-finished beeps:500ms/5 times
segment shift beeps:500ms/twice
```

问：SN8 ASM 怎么实现？

The implementation of SN8 ASM is as following：

```
Buzzer00:
    b0bts1    key_bibi_f
    jmp  buzzer10
    mov a,@buzzer_time1        ;200ms.
    mov buzzer_time,a          ; Length of beeps.
    mov a,@buzzer_not3         ;#1
    mov buzzer_not,a           ; Beep times.
    jmp  buzzer40
buzzer10:
    b0bts1    end_bibi_f
    jmp  buzzer20
    mov a,@buzzer_time2        ;500ms.
    mov buzzer_time,a          ; Length of beeps
    mov a,@buzzer_not1         ;#10
    mov buzzer_not,a           ; Sign-reverse times.
    jmp  buzzer40
buzzer20:
    b0bts1    segment_bibi_f
    jmp  buzzer30
    mov a,@buzzer_time2        ;500ms.
    mov buzzer_time,a          ; Length of beeps.
    mov a,@buzzer_not2         ; #4.
    mov buzzer_not,a           ; Sign-reverse times.
    jmp  buzzer40
```

在使用 if 语句中还应注意以下问题：

1. 在三种形式的 if 语句中，在 if 关键字之后均为表达式。 该表达式通常是逻辑表达式或关系表达式， 但也可以是其它表达式，如赋值表达式等，甚至也可以是一个变量。例如： if(a=5) 语句；if(b) 语句； 都是允许的。只要表达式的值为非 0，即为“真”。如在 if(a=5)…；中表达式的值永远为非 0，所以其后的语句总是要执行的，当然这种情况在程序中不一定会出现，但在语法上是合法的。 又如，有程序段：

```
if(a=b)
printf("%d",a);
else
printf("a=0");
```

   本语句的语义是，把 b 值赋予 a，如为非 0 则输出该值，否则输出“a=0”字符串。这种用法在程序中是经常出现的。
2. 在 if 语句中，条件判断表达式必须用括号括起来， 在语句之后必须加分号。
3. 在 if 语句的三种形式中，所有的语句应为单个语句，如果要想在满足条件时执行一

组(多个)语句，则必须把这一组语句用{} 括起来组成一个复合语句。但要注意的是在}之后不能再加分号。

例如：

```
if(a>b){
   a++;
   b++;
}
else{ a=0;
   b=10;
}
```

if 语句的嵌套

当 if 语句中的执行语句又是 if 语句时，则构成了 if 语句嵌套的情形。其一般形式可表示如下：

```
if(表达式)
   if 语句；
```

或者为

```
if(表达式)
   if 语句；
else
   if 语句；
```

在嵌套内的 if 语句可能又是 if-else 型的，这将会出现多个 if 和多个 else 重叠的情况，这时要特别注意 if 和 else 的配对问题。例如：

```
if(condition1)
if(contition2)
   statement1；
else
   statement2；
```

其中的 else 究竟是与哪一个 if 配对呢?

```
Is it:                    or:
if(condition1)            if(condition1)
   if(condition2)                 if(condition 2)
      statement1；                        statement1；
   else                    else
```

statement2；　　statement2；

为了避免这种二义性，C 语言规定，else 总是与它前面最近的 if 配对，因此对上述例子应按前一种情况理解。

B、 并行多分支结构流程

C 语言还提供了另一种用于多分支选择的 switch 语句， 其一般形式为：

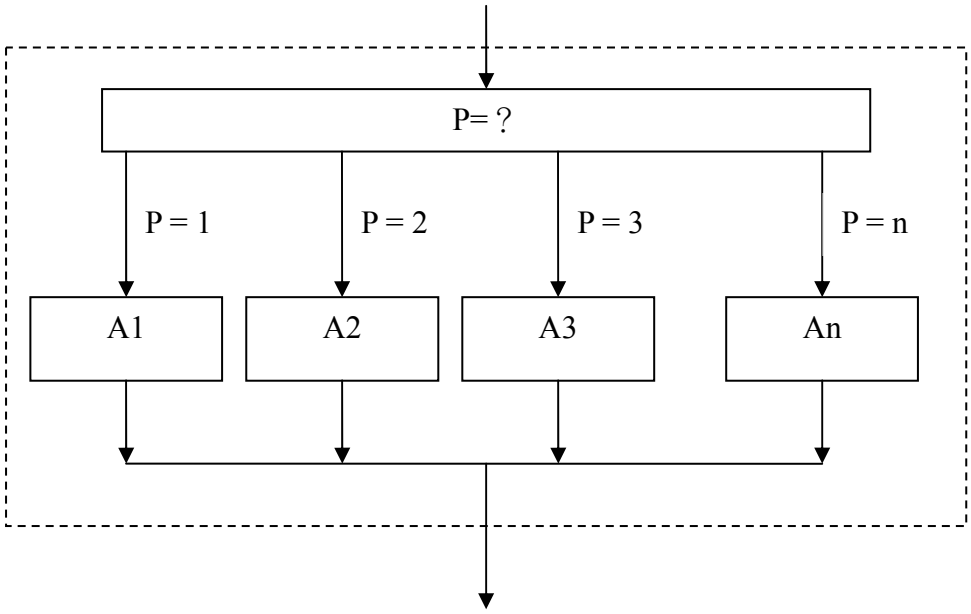


图 Figure 4-4 Parallel multiplied branches flowing chart

C 可以用 switch…Case 来实现：

```
    :
  switch(expression){
    case constant expression1: statement1;
    case constant expression 2: statement2;
    …
    case constant expression n: statement n;
    default : statement n+1;
}
```

其语义是：计算表达式的值。 并逐个与其后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时， 即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句。 如表达式的值与所有 case 后的常量表达式均不相同时，则执行 default 后的语句。例：

```
switch(step)
{
```

```
                case 0:                    //Press button once to set time
                    ks81();break;
                case ONE_PRESS_CLOCK_KEY_C: //Press twice to set the
minutes
                    ks82();break;
                case TWO_PRESS_CLOCK_KEY_C : //Press for the third time to
end time setting
                    ks83();break;
                case BESPOKE_ING_C :        //Press button to query
pre-set time
                    ks84();break;
                case SELECT_TIME_C :        //Set flag bespoke_f=1
                    if(job_mode1== DEFROST_MODE_C) break;
                    else ks85();break;
                case START_ING_C:           //Query for the running time
                    if(job_mode1 == DEFROST_MODE_C) ks86();break;
            }
```

在使用 switch 语句时还应注意以下几点：

1. 在 case 后的各常量表达式的值不能相同，否则会出现错误。
2. 在 case 后，允许有多个语句，可以不用{}括起来。
3. 各 case 和 default 子句的先后顺序可以变动，而不会影响程序执行结果。
4. default 子句可以省略不用。

汇编的实现如下：

```
mov a,step
b0bts1    fz                        ;=0 press button once to set time.
jmp  ks82
cmprs     a,ONE_PRESS_CLOCK_KEY_C
jmp  ks83
cmprs     a,TWO_PRESS_CLOCK_KEY_C
jmp  ks84                           ; end time setting
cmprs     a,BESPOKE_ING_C           ; Query time.
jmp  ks85
cmprs     a,SELECT_TIME_C           ;Press button to reserve
jmp  ks86
key81：
          jmp       key89
key82:
          jmp       key89
key83:
          jmp       key89
key84:
          jmp       key89
key85:
```

```
            jmp     key89
key86:
            jmp     key89
key89:      ret
```

8.4.3 循环结构

循环结构是程序中一种很重要的结构。其特点是， 在给定条件成立时，反复执行某程序段，直到条件不成立为止。 给定的条件称为循环条件，反复执行的程序段称为循环体。 C 语言提供了多种循环语句，可以组成各种不同形式的循环结构。

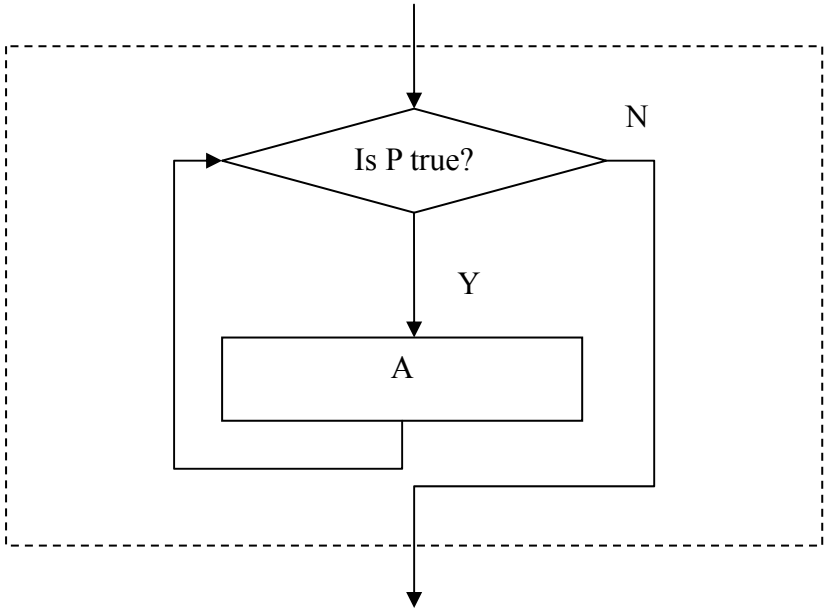


图 Figure4-5 "While" loop flowing chart

The while Statement

while 语句的一般形式为：

while(表达式)语句；

其中表达式是循环条件，语句为循环体。

while 语句的语义是：计算表达式的值，当值为真(非 0)时， 执行循环体语句。其执行过程可用图 3—4 表示。

While 型循环的 C 语言实现：

```
tempbuf = 0;
while(tempbuf==15){++tempbuf;}        //Delay
```

While 型循环的汇编实现：

```
            Clr   y
Loop:
            B0mov   a,y
            Cmprs   a,#15
            Jmp     $+2
            Jmp     loop90
            decms   y                   ; Delay
            jmp     loop
loop90:
            RET
```

使用 while 语句应注意以下几点：

1. while 语句中的表达式一般是关系表达或逻辑表达式，只要表达式的值为真(非 0)即可继续循环。
2. 循环体如包括有一个以上的语句，则必须用{}括起来， 组成复合语句。
3. 应注意循环条件的选择以避免死循环。

```
void main(){
   int a,n=0;
   while (a=5)
     {
      printf("%d ",n++);
     }
}
```

本例中 while 语句的循环条件为赋值表达式 a=5， 因此该表达式的值永远为真，而循环体中又没有其它中止循环的手段， 因此该循环将无休止地进行下去，形成死循环。
4.允许 while 语句的循环体又是 while 语句，从而形成双重循环。

do-while 语句

do-while 语句的一般形式为：

```
do
      statement；
while(condition)；
```

其中语句是循环体，表达式是循环条件。

do-while 语句的语义是：

先执行循环体语句一次， 再判别表达式的值，若为真(非 0)则继续循环，否则终止循环。

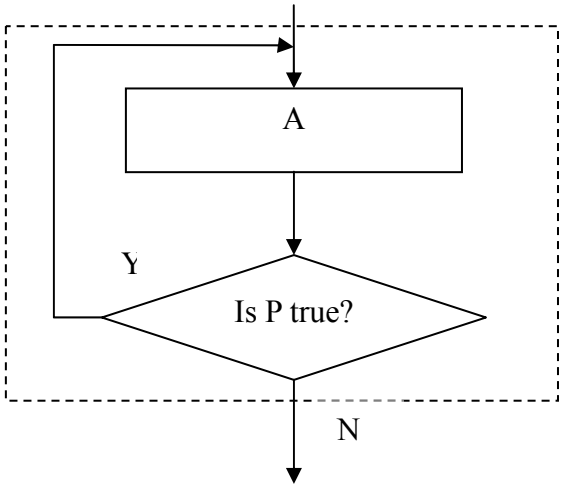


图 4-6 "Do…while" loop flowing chart

Do…while 循环的 C 实现：

```
unsigned int * pyz = (unsigned int *)0x7f;
do{
    *pyz =   0x00;
    --i;
}while(i);
```

Do…while 循环的汇编实现：

```
ClrRAM:
    clr   Y                 ;Select bank 0.
    b0mov   Z,#0x7f         ;Set @YZ address from 7fh.

ClrRAM10:
    clr   @YZ               ;Clear @YZ content.
    decms   Z               ; z= z - 1 , skip next if   z=0.
    jmp  ClrRAM10
    clr   @YZ               ;Clear address 0x00.

    mov a,#00H
```

对于 do-while 语句还应注意以下几点：

1. 在 if 语句，while 语句中， 表达式后面都不能加分号， 而在 do-while 语句的表达式后面则必须加分号。
2. do-while 语句也可以组成多重循环，而且也可以和 while 语句相互嵌套。
3. 在 do 和 while 之间的循环体由多个语句组成时，也必须用{}括起来组成一个复合语句。
4. do-while 和 while 语句相互替换时，要注意修改循环控制条件。

do-while 语句和 while 语句的区别在于 do-while 是先执行后判断，因此 do-while 至少要执行一次循环体。而 while 是先判断后执行，如果条件不满足，则一次循环体语句也不执行。

while 语句和 do-while 语句一般都可以相互改写。

for 语句

for 语句是 C 语言所提供的功能更强，使用更广泛的一种循环语句。其一般形式为：

```
for(init-statement; condition; expression)
  statement
```

表达式 1　通常用来给循环变量赋初值，一般是赋值表达式。也允许在 for 语句外给循环变量赋初值，此时可以省略该表达式。

表达式 2　通常是循环条件，一般为关系表达式或逻辑表达式。

表达式 3　通常可用来修改循环变量的值，一般是赋值语句。

这三个表达式都可以是逗号表达式， 即每个表达式都可由多个表达式组成。三个表达式都是任选项，都可以省略。

一般形式中的"语句"即为循环体语句。for 语句的语义是：

1. 首先计算表达式 1 的值。
2. 再计算表达式 2 的值，若值为真(非 0)则执行循环体一次， 否则跳出循环。
3. 然后再计算表达式 3 的值，转回第 2 步重复执行。在整个 for 循环过程中，表达式 1 只计算一次，表达式 2 和表达式，3 则可能计算多次。循环体可能多次执行，也可能一次都不执行。

在使用 for 语句中要注意以下几点

(1). for 语句中的各表达式都可省略，但分号间隔符不能少。如：for(；表达式；表达式)省去了表达式 1。for(表达式；；表达式)省去了表达式 2。

for(表达式；表达式；)省去了表达式 3。for(；；)省去了全部表达式。

(2). 在循环变量已赋初值时，可省去表达式 1。如省去表达式 2 或表达式 3 则将造成无限循环， 这时应在循环体内设法结束循环。

(3). 循环体可以是空语句。

4. for 语句也可与 while,do-while 语句相互嵌套，构成多重循环。以下形式都是合法的

嵌套。

```
(1) for(){…
      while()
        {…}
      …
          }

(2) do{
        …
      for()
        {…}
      …
      }while();

(3) while(){
                …
                for()
                  {…}
                …
            }

(4) for(){
            …
            for(){
            …
              }
            }
```

转移语句

程序中的语句通常总是按顺序方向， 或按语句功能所定义的方向执行的。如果需要改变程序的正常流向， 可以使用转移语句。在 C 语言中提供了 4 种转移语句：

goto,break, continue and return。

其中的 return 语句只能出现在被调函数中， 用于返回主调函数。

1. goto 语句

goto 语句也称为无条件转移语句，其一般格式如下： goto 语句标号； 其中语句标号是按标识符规定书写的符号， 放在某一语句行的前面，标号后加冒号(：)。语句标号起标识语句的作用，与 goto 语句配合使用。

```
goto label;
```

如：

```
label: i++;
 loop: while(x<7)
      {
          ...
        goto (label);
      };
```

C 语言不限制程序中使用标号的次数，但各标号不得重名。goto 语句的语义是改变程序流向， 转去执行语句标号所标识的语句。

goto 语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。

但是，在结构化程序设计中一般不主张使用 goto 语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

2. break 语句

break 语句只能用在 switch 语句或循环语句中， 其作用是跳出 switch 语句或跳出本层循环，转去执行后面的程序。由于 break 语句的转移方向是明确的，所以不需要语句标号与之配合。break 语句的一般形式为： break; 上面例题中分别在 switch 语句和 for 语句中使用了 break 语句作为跳转。使用 break 语句可以使循环语句有多个出口，在一些场合下使编程更加灵活、方便。

3. continue 语句

continue 语句只能用在循环体中，其一般格式是：

```
continue;
```

其语义是：结束本次循环，即不再执行循环体中 continue 语句之后的语句，转入下一次循环条件的判断与执行。应注意的是， 本语句只结束本层本次的循环，并不跳出循环。

8.5 数组

数组在程序设计中，为了处理方便， 把具有相同类型的若干变量按有序的形式组织起来。这些按序排列的同类数据元素的集合称为数组。在 C 语言中， 数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或是构造类型。因此按数组元素的类型不同，数组又可分为数值数组、字符数组、指针数组、结

构数组等各种类别。

8.5.1 数组类型说明

在 C 语言中使用数组必须先进行类型说明。 数组说明的一般形式为：

类型说明符 数组名 [常量表达式]，……；

其中，类型说明符是任一种基本数据类型或构造数据类型。 数组名是用户定义的数组标识符。 方括号中的常量表达式表示数据元素的个数，也称为数组的长度。

例如：

```
int a[10];          说明整型数组 a，有 10 个元素。
float b[10],c[20];  说明实型数组 b，有 10 个元素，实型数组 c，有 20 个元素。
char ch[20];        说明字符数组 ch，有 20 个元素。
```

对于数组类型说明应注意以下几点：

1. 数组的类型实际上是指数组元素的取值类型。对于同一个数组，其所有元素的数据类型都是相同的。
2. 数组名的书写规则应符合标识符的书写规定。
3. 数组名不能与其它变量名相同，例如：

```
void main()
{
  int a;
  float a[10];
  ……
}
```

4. 方括号中常量表达式表示数组元素的个数，如 a[5]表示数组 a 有 5 个元素。但是其下标从 0 开始计算。因此 5 个元素分别为 a[0],a[1],a[2],a[3],a[4]。
5. 不能在方括号中用变量来表示元素的个数， 但是可以是符号常数或常量表达式。例如：

```
#define FD 5
void main()
{
  int a[3+2],b[7+FD];
  ……
}
```

是合法的。但是下述说明方式是错误的。

```
void main()
```

```
{
   int n=5;
   int a[n];
   ……
}
```

6. 允许在同一个类型说明中，说明多个数组和多个变量。
   例如： int a,b,c,d,k1[10],k2[20];

8.5.2 数组元素的表示方法

数组元素是组成数组的基本单元。数组元素也是一种变量， 其标识方法为数组名后跟一个下标。 下标表示了元素在数组中的顺序号。数组元素的一般形式为： 数组名[下标] 其中的下标只能为整型常量或整型表达式。如为小数时，C 编译将自动取整。例如，a[5],a[i+j],a[i++]都是合法的数组元素。 数组元素通常也称为下标变量。必须先定义数组， 才能使用下标变量。在 C 语言中只能逐个地使用下标变量， 而不能一次引用整个数组。

arrayName [subscript]

数组的赋值

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外， 还可采用初始化赋值和动态赋值的方法。数组初始化赋值是指在数组说明时给数组元素赋予初值。 数组初始化是在编译阶段进行的。这样将减少运行时间，提高效率。

初始化赋值的一般形式为： static 类型说明符 数组名[常量表达式]={值，值……值}；其中 static 表示是静态存储类型，C 语言规定只有静态存储数组和外部存储数组才可作初始化赋值。在{ }中的各数据值即为各元素的初值，各值之间用逗号间隔。例如：static int a[10]={ 0,1,2,3,4,5,6,7,8,9 }; 相当于 a[0]=0;a[1]=1...a[9]=9;

C 语言对数组的初始赋值还有以下几点规定：

1. 可以只给部分元素赋初值。当{ }中值的个数少于元素个数时，只给前面部分元素赋值。例如： static int a[10]={0,1,2,3,4};表示只给 a[0]～a[4]5 个元素赋值，而后 5 个元素自动赋 0 值。

2. 只能给元素逐个赋值，不能给数组整体赋值。 例如给十个元素全部赋 1 值，只能写为：

   static int a[10]={1,1,1,1,1,1,1,1,1,1};

   而不能写为：

static int a[10]=1;

3. 如不给可初始化的数组赋初值，则全部元素均为 0 值。
4. 如给全部元素赋值，则在数组说明中， 可以不给出数组元素的个数。例如：

static int a[5]={1,2,3,4,5};

can be written as:

static int a[]={1,2,3,4,5};

动态赋值可以在程序执行过程中，对数组作动态赋值。

二维数组

前面介绍的数组只有一个下标，称为一维数组， 其数组元素也称为单下标变量。在实际问题中有很多量是二维的或多维的， 因此 C 语言允许构造多维数组。多维数组元素有多个下标， 以标识它在数组中的位置，所以也称为多下标变量。 本小节只介绍二维数组，多维数组可由二维数组类推而得到。二维数组类型说明二维数组类型说明的一般形式是：

类型说明符 数组名[常量表达式 1][常量表达式 2]；

其中常量表达式 1 表示第一维下标的长度，常量表达式 2 表示第二维下标的长度。例如： int a[3][4];

8.6 指针

指针是 C 一种应用非常广泛的数据类型，指针的应用使得 C 可以对底层进行灵活操作。同时，由于指针的引入使 C 的数据处理能力空前强大。SN8 C 在支持标准 C 的基础上，又进一步加强了指针的功能，使其更适合于 SN8P 系列芯片资源。

8.6.1 RAM/ROM 指针

可变指针（有两个主要特性）不仅要表示它自己的指针变量类型（即指针本身存放的地点），也要表示指向它所指的变量（即变量存放的地点）。在 location 选项中它被设置为指向 RAM 或 ROM。表 8-3 列出 SN8 C Compiler 支持的四种指针变量：

指针类型	指向的区域	指针声明的语法
__RAM	__RAM	TypeObject **__RAM** * **__RAM**

		PtrInRamToObjInRam;
__RAM	__ROM	TypeObject **__ROM** * **__RAM** PtrInRamToObjInRom;
__ROM	__RAM	TypeObject **__RAM** * **__ROM** PtrInRomToObjInRam;
__ROM	__ROM	TypeObject **__ROM** * **__ROM** PtrInRomToObjInRom;

表 8-3 指针类型表

这里 TypeObject 指出指针所指数据的类型。这些可以是支持的基本类型，比如结构体，指针，或联合体。'*'右边的限制语'__RAM'、'__ROM'指出将变量存放在 RAM 区还是 ROM 区。 '*'左边的限制语'__RAM'、'__ROM'表示所指变量在 RAM 区还是在 ROM 区。所有限制语__RAM 均可省略。

为了更好的说明指针行为，__RAM 指针所指目标在 RAM 区，__ROM 指针所指目标在 ROM 区。__GENERIC 指针所指目标在 RAM 或 ROM 区中。一般，指针指向 RAM 区地址而且是可变的，除非预先定义过。指向 ROM 区地址的指针不可变。

```
int data;
int __ROM * romptr;
int __RAM * ramptr;
data = *romptr;                 // O.K. get data from ROM memory。
data = *ramptr;                 // O.K. get data from RAM memory。
ramptr = romptr;            // error。Ram pointer can't address ROM。
romptr = ramptr;            // error。Rom pointer can't address RAM。
*romptr = data;                 // error，can't write data into ROM。
*ramptr = data;                 // O.K. RAM memory is un-changeable。
*ramptr = *romptr;          // O.K. passing data from ROM to RAM via pointer。
```

8.6.2 Generic 指针

Generic 指针用来访问数据而不管数据存储。例如：

```
typedef struct { int x; int y; } TypeObject;
static TypeObject __RAM RamData = { 1，2 };
static TypeObject __ROM RomData = { 3，4 };
TypeObject value;
TypeObject __RAM * ramptr = &RamData;
TypeObject __ROM * romptr = &RomData;
TypeObject * ptr;
ptr = &RamData;         // O.K.
value = *ptr;           // access RAM Data
ptr = &RomData;         // O.K.
```

```
value = *ptr;          // access ROM data
ptr = ramptr;          // O.K.
value = *ptr;          // get the value of    *ramptr (i.e. RomData)
ptr = romptr;          // O.K.
value = *ptr;          // get the value of *romptr (i.e. RomData)
```

8.7 函数

函数在 C 当中占有相当重要的地位，它是组成程序的元素。函数的定义、函数的调用构成 C 程序的主要结构。而以单片机(Single Chip Micyoco)为对象的编程又与通用 C 的编程略有不同，主要表现在全局变量与函数参数的选择上以及参数类型的选择上。

8.7.1 函数定义

函数是完成一定的功能的模块，函数的出现使代码可以得以复用，可以大大减少产生的代码量，这是使用函数最为充分的理由。但是，从另一方面来看，函数基本都涉及参数的传递，若是处理不好就很容易产生参数传递的冗余代码，还有就是调用函数就一定涉及到入栈和出栈的处理，这些都占用 CPU 的资源，影响程序的实时性，若函数调用过于频繁，势必影响程序的执行效率。所以函数定义之前要考虑函数的划分，函数功能既不能覆盖过大，也不能覆盖过小，使函数的划分太细。

所有的函数定义，包括主函数 main 在内，都是平行的。也就是说，在一个函数的函数体内， 不能再定义另一个函数， 即不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。习惯上把调用者称为主调函数。 main 函数是主函数，它可以调用其它函数，而不允许被其它函数调用。 因此，C 程序的执行总是从 main 函数开始， 完成对其它函数的调用后再返回到 main 函数，最后由 main 函数结束整个程序。一个 C 源程序必须有，也只能有一个主函数 main。

函数定义的一般形式

1.无参函数的一般形式

```
类型说明符  函数名()
{
  类型说明
  语句
}
```

其中类型说明符和函数名称为函数头。 类型说明符指明了本函数的类型，函数的类型实际上是函数返回值的类型。 该类型说明符与第二章介绍的各种说明符相同。 函数名是由用户定义的标识符，函数名后有一个空括号，其中无参数，但括号不可少。{} 中的内容称为函数体。在函数体中也有类型说明， 这是对函数体内部所用到的变量的类型说明。在很多情况下都不要求无参函数有返回值， 此时函数类型符可以写为 void。

2.有参函数的一般形式

```
类型说明符  函数名(形式参数表)
型式参数类型说明
{
  类型说明
  语句
}
```

有参函数比无参函数多了两个内容，其一是形式参数表， 其二是形式参数类型说明。在形参表中给出的参数称为形式参数， 它们可以是各种类型的变量， 各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。 形参既然是变量，当然必须给以类型说明。

函数说明在主调函数中调用某函数之前应对该被调函数进行说明， 这与使用变量之前要先进行变量说明是一样的。 在主调函数中对被调函数作说明的目的是使编译系统知道被调函数返回值的类型， 以便在主调函数中按此种类型对返回值作相应的处理。

对被调函数的说明也有两种格式，一种为传统格式，其一般格式为：

```
类型说明符  被调函数名()；
```

这种格式只给出函数返回值的类型，被调函数名及一个空括号。这种格式由于在括号中没有任何参数信息， 因此不便于编译系统进行错误检查，易于发生错误。

另一种为现代格式，其一般形式为：

```
类型说明符  被调函数名(类型  形参，类型  形参…)；
```

或为：

```
类型说明符  被调函数名(类型，类型…)；
```

现代格式的括号内给出了形参的类型和形参名， 或只给出形参类型。这便于编译系统进行检错，以防止可能出现的错误。

SN8 C 支持现代格式。

> **Note：**
>
> Functions without return value should be declared as "void" to prevent memory reservation.

在 C 当中，函数都应该先声明，后调用，标准 C 的写法一般是先在程序的开头声明要用到的函数，然后在程序中编写完整的函数，我们先来看看 C 的函数。

如：

```
unsigned int bcd(unsigned int);
.
.
clock_min = bcd(clock_min);
.
.
.
unsigned int bcd(unsigned int input)
{
    unsigned int result_buf;
    result_buf = input & 0x0f;
    if(result_buf > 0x0a)
    {
        result_buf = input + 6;
    }
    else
    {
        result_buf = input;
    }
    return(result_buf);
}
```

上面的函数完成一个固定的功能，能提供给程序中任意一个地方来调用并完成所需的功能，在 C 中都是通过给予不同的参数来得到需要的计算结果，这样就可以节省代码量，使程序得到复用。

C 语言中又规定在以下几种情况时可以省去主调函数中对被调函数的函数说明。

1. 如果被调函数的返回值是整型或字符型时，可以不对被调函数作说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。
2. 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再作说明而直接调用。
3. 如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数作说明。例如：

```
char str(int a);
float f(float b);
 main()
 {
 ……
 }
  char str(int a)
```

```
 {
……
}
 float f(float b)
 {
……
}
```

其中第一，二行对 str 函数和 f 函数预先作了说明。 因此在以后各函数中无须对 str 和 f 函数再作说明就可直接调用。

4. 对库函数的调用不需要再作说明， 但必须把该函数的头文件用 include 命令包含在源文件前部。

其实汇编一样可以写成函数形式，这样就可以起到和 C 一样的效果，我们来看看汇编是怎么来完成的。

如：
调用点的汇编代码：

```
mov     a,clock_min
b0mov   y,a
call    bcd
mov     clock_min,a
```

功能函数代码：

```
bcd:
    b0mov   a,y             ; Move the initial number to "Y".
    and     a,#0fh
    cmprs   a,#0ah
    nop
    b0bts0  fc
    jmp     $+3
    b0mov   a,y
    ret
    b0mov   a,y
    add     a,#6h           ; Move the changed number to "a".
    ret
```

从上面的程序中我们可以看出，汇编是通过用户规定的一个或几个寄存器来将参数传递进去！又用同样的方法，将结果传递出来。可以想到，所需的结果在函数返回后可以在 A 中得到，其应用方式如下：

```
mov     clock_min,a
cmprs   a,#60h
```

8.7.2 函数参数传递

在汇编程序中，用户必须自己定义寄存器用来存放往函数内传递的参数数据和返回的结果。在 C 程序中函数参数的传递都是通过形参和实参的对应来传递的，我们通过函数调用来将参数传递给函数，在完成函数功能得到结果后，又通过返回值将结果传回给调用的地方。如图：

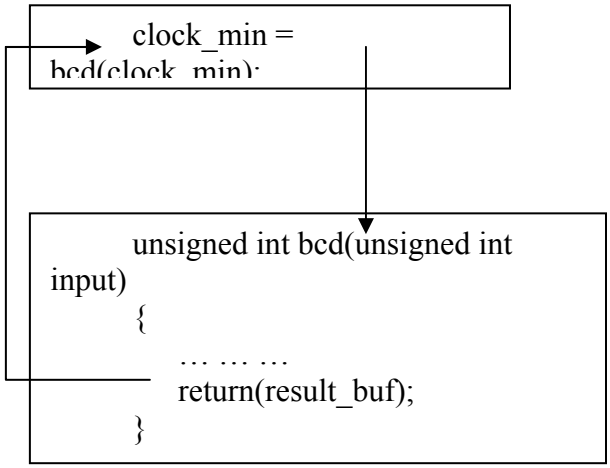


Figure5-1: The Argument passing and returning

函数的形参和实参具有以下特点：

1. 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。 函数调用结束返回主调函数后则不能再使用该形参变量。
2. 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值。
3. 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生"类型不匹配"的错误。
4. 函数调用中发生的数据传送是单向的。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

函数的值是指函数被调用之后， 执行函数体中的程序段所取得的并返回给主调函数的值。

1. 函数的值只能通过 return 语句返回主调函数。return 语句的一般形式为：

return 表达式；
或者为：
return (表达式)；

该语句的功能是计算表达式的值，并返回给主调函数。 在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行， 因此只能返回一个函数值。

2. 函数值的类型和函数定义中函数的类型应保持一致。 如果两者不一致，则以函数类型为准，自动进行类型转换。
3. 如函数值为整型，在函数定义时可以省去类型说明。
4. 不返回函数值的函数，可以明确定义为"空类型"， 类型说明符为"void"。

```
void s(int n)
{ ……
}
```

一旦函数被定义为空类型后， 就不能在主调函数中使用被调函数的函数值了。例如，在定义 s 为空类型后，在主函数中写下述语句 sum=s(n); 就是错误的。为了使程序有良好的可读性并减少出错， 凡不要求返回值的函数都应定义为空类型。

但是，我们都知道 C 是可以通过转成汇编来实现的，那么 SN8 C 是如何来实现 C 的函数传递的呢？虽然这是一个编译器内部的问题，但是面对单片机硬件的编程，了解其原理会有助于用户写出更为精简合理的程序。
我们来看看 C 函数调用时，编译器生成的中间汇编代码，假设于 caller 函数内调用 callee 函式. callee 的参数名称为

```
_callee_arg ?                          ; ? 为参数个数。

   int foo(int a, int b, long c)   will generate:

_foo_data SEGMENT DATA INBANK   ;OVERLAYABLE
_foo_arg0 DS 1                  ; int a
_foo_arg1 DS 1                  ; int b
_foo_arg2 DS 2                  ; long c

  MOV          A, (_clock_min)
  __SelectBANK _bcd_arg0
  MOV          _bcd_arg0, A
                                ;End push arg....
  call  _bcd
  __SelectBANK (_clock_min)
  MOV      _clock_min, A
```

通过 C 与汇编程序的对应，我们很容易知道参数的传递是这样的：实际参数 clock_min 被直接赋值给函数的形参_bcd_arg0,然后在调用函数中参加运算。再来看看返回值，从

程序中我们发现最后返回值是在 A 中得到的，那么在这里是由系统生成的，是不是有一定的规律呢？还是随意的？当然不可能是随意的！那么系统是如何规定的呢？

若返回值是基本的数据类型，一般都是放到固定的寄存器或虚拟寄存器里面，如下表所示：

返回值类型	寄存器
unsigned/signed char	A
unsigned/signed short	A
unsigned/signed int	A
unsigned/signed long	A，R
float	A，R，Y，Z

表 5-1 不同返回值的存放列表

从上边的列表可以知道，对于不同返回值类型，编译器都会有固定的返回值存放点，当程序从函数中返回的时候，这些对应的寄存器里面就存放着返回值，这就从规则上印证了上述 bcd 函数的转换结果。

而对于那些返回值是数据结构的，系统则会在调用函数的时候增加一个隐含参数(地址)传递给被调函数，被调函数在完成功能运算后将结果放到指定的位置上，程序从函数里返回后就可以从该地址读取返回值。

尽管从原理上看起来，函数的参数传递与返回值的 C 与汇编的转换的形式是一样的，但是编译器产生代码时依然可能产生一些冗余代码。所以，在面对单片机这样的资源的硬件编程，过多的参数传递及过于复杂的返回值往往造成代码转换得低效率，这是用户需要注意的。

8.7.3 变量的作用域

在讨论函数的形参变量时曾经提到， 形参变量只在被调用期间才分配内存单元，调用结束立即释放。 这一点表明形参变量只有在函数内才是有效的， 离开该函数就不能再使用了。这种变量有效性的范围称变量的作用域。不仅对于形参变量， C 语言中所有的量都有自己的作用域。变量说明的方式不同，其作用域也不同。 C 语言中的变量，按作用域范围可分为两种， 即局部变量和全局变量。

一、局部变量

局部变量也称为内部变量。局部变量是在函数内作定义说明的。其作用域仅限于函数内，离开该函数后再使用这种变量是非法的。

例如：

```
int f1(int a) /*函数 f1*/
{
int b,c;
……
}a,b,c 作用域

int f2(int x) /*函数 f2*/
{
int y,z;
}x,y,z 作用域

main()
{
int m,n;
}
```

m,n 作用域 在函数 f1 内定义了三个变量，a 为形参，b,c 为一般变量。在 f1 的范围内 a,b,c 有效，或者说 a,b,c 变量的作用域限于 f1 内。同理，x,y,z 的作用域限于 f2 内。 m,n 的作用域限于 main 函数内。关于局部变量的作用域还要说明以下几点：

1. 主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。同时，主函数中也不能使用其它函数中定义的变量。因为主函数也是一个函数，它与其它函数是平行关系。这一点是与其它语言不同的，应予以注意。
2. 形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。
3. 允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。如在例 5.3 中，形参和实参的变量名都为 n，是完全允许的。
4. 在复合语句中也可定义变量，其作用域只在复合语句范围内。例如：

```
main()
{
int s,a;
……
{
int b;
s=a+b;
……b 作用域
}
……s,a 作用域
}
```

二、全局变量

全局变量也称为外部变量，它是在函数外部定义的变量。 它不属于哪一个函数，它属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量，一般应作全局变量说明。 只有在函数内经过说明的全局变量才能使用。全局变量的说明符为 extern。 但在一个函数之前定义的全局变量，在该函数内使用可不再加以说明。 例如：

```
int a,b; /*外部变量*/
void f1() /*函数 f1*/
{
……
}
float x,y; /*外部变量*/
int fz() /*函数 fz*/
{
……
}
main() /*主函数*/
{
……
}/*全局变量 x,y 作用域  全局变量 a,b 作用域*/
```

从上例可以看出 a、b、x、y 都是在函数外部定义的外部变量，都是全局变量。但 x,y 定义在函数 f1 之后，而在 f1 内又无对 x,y 的说明，所以它们在 f1 内无效。 a,b 定义在源程序最前面，因此在 f1,f2 及 main 内不加说明也可使用。因此外部变量是实现函数之间数据通讯的有效手段。

对于全局变量还有以下几点说明：

1. 对于局部变量的定义和说明，可以不加区分。而对于外部变量则不然，外部变量的定义和外部变量的说明并不是一回事。外部变量定义必须在所有的函数之外，且只能定义一次。其一般形式为： [extern] 类型说明符 变量名，变量名… 其中方括号内的 extern 可以省去不写。

   例如：

   ```
   int a,b;
   ```

   等效于：

   ```
   extern int a,b;
   ```

   而外部变量说明出现在要使用该外部变量的各个函数内， 在整个程序内，可能出现多次，外部变量说明的一般形式为： extern 类型说明符 变量名，变量名，…；外部变量在定义时就已分配了内存单元， 外部变量定义可作初始赋值，外部变量说明不能再赋初始值， 只是表明在函数内要使用某外部变量。

2. 外部变量可加强函数模块之间的数据联系， 但是又使函数要依赖这些变量，因而使得函数的独立性降低。从模块化程序设计的观点来看这是不利的， 因此在不必要时尽量不要使用全局变量。
3. 在同一源文件中，允许全局变量和局部变量同名。在局部变量的作用域内，全局变量不起作用。

```
int vs(int l,int w)
{
extern int h;
int v;
v=l*w*h;
return v;
}
main()
{
extern int w,h;
int l=5;
printf("v=%d",vs(l,w));
}
int l=3,w=4,h=5;
```

本例程序中，外部变量在最后定义， 因此在前面函数中对要用的外部变量必须进行说明。外部变量 l，w 和 vs 函数的形参 l，w 同名。外部变量都作了初始赋值，mian 函数中也对 l 作了初始化赋值。执行程序时，在 printf 语句中调用 vs 函数，实参 l 的值应为 main 中定义的 l 值，等于 5，外部变量 l 在 main 内不起作用；实参 w 的值为外部变量 w 的值为 4，进入 vs 后这两个值传送给形参 l，wvs 函数中使用的 h 为外部变量，其值为 5，因此 v 的计算结果为 100，返回主函数后输出。

变量的存储类型

各种变量的作用域不同， 就其本质来说是因变量的存储类型不同。所谓存储类型是指变量占用内存空间的方式， 也称为存储方式。

变量的存储方式可分为“静态存储”和“动态存储”两种。

静态存储变量通常是在变量定义时就分定存储单元并一直保持不变， 直至整个程序结束。全局变量即属于此类存储方式。动态存储变量是在程序执行过程中，使用它时才分配存储单元， 使用完毕立即释放。 典型的例子是函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时，才予以分配， 调用函数完毕立即释放。如果一个函数被多次调用，则反复地分配、 释放形参变量的存储单元。从以上分析可知， 静态存储变量是一直存在的， 而动态存储变量则时而存在时而消失。我们又把这种由于变量存储方式不同而产生的特性称变量的生存期。 生存期表示了变量存在的时间。 生存期和作用域是从时间和空间这两个不同的角度来描述变量的特性，这两者既

有联系，又有区别。一个变量究竟属于哪一种存储方式，并不能仅从其作用域来判断，还应有明确的存储类型说明。

在 C 语言中，对变量的存储类型说明有以下四种：

auto	自动变量
register	寄存器变量
extern	外部变量
static	静态变量

自动变量和寄存器变量属于动态存储方式，外部变量和静态变量属于静态存储方式。在介绍了变量的存储类型之后，可以知道对一个变量的说明不仅应说明其数据类型，还应说明其存储类型。

因此变量说明的完整形式应为：

存储类型说明符 数据类型说明符 变量名，变量名…；

例如：

static int a,b;	说明 a,b 为静态类型变量
auto char c1,c2;	说明 c1,c2 为自动字符变量
static int a[5]={1,2,3,4,5};	说明 a 为静整型数组
extern int x,y;	说明 x,y 为外部整型变量

下面分别介绍以上四种存储类型：

一、自动变量的类型说明符为 auto

这种存储类型是 C 语言程序中使用最广泛的一种类型。C 语言规定，函数内凡未加存储类型说明的变量均视为自动变量，也就是说自动变量可省去说明符 auto。

自动变量具有以下特点：

1. 自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量，只在该函数内有效。在复合语句中定义的自动变量只在该复合语句中有效。
2. 自动变量属于动态存储方式，只有在使用它，即定义该变量的函数被调用时才给它分配存储单元，开始它的生存期。函数调用结束，释放存储单元，结束生存期。因此函数调用结束之后，自动变量的值不能保留。在复合语句中定义的自动变量，在退出复合语句后也不能再使用，否则将引起错误。
3. 由于自动变量的作用域和生存期都局限于定义它的个体内(函数或复合语句内)，因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名。

4. 对构造类型的自动变量如数组等，不可作初始化赋值。

二、外部变量的类型说明符为 extern

在前面介绍全局变量时已介绍过外部变量。这里再补充说明外部变量的几个特点：

1. 外部变量和全局变量是对同一类变量的两种不同角度的提法。全局变是是从它的作用域提出的，外部变量从它的存储方式提出的，表示了它的生存期。
2. 当一个源程序由若干个源文件组成时，在一个源文件中定义的外部变量在其它的源文件中也有效。例如有一个源程序由源文件 F1.C 和 F2.C 组成：

F1.C

```
int a,b; /*外部变量定义*/
char c; /*外部变量定义*/
main()
{
……
}
```

F2.C

```
extern int a,b; /*外部变量说明*/
extern char c; /*外部变量说明*/
func (int x,y)
{
……
}
```

在 F1.C 和 F2.C 两个文件中都要使用 a,b,c 三个变量。在 F1.C 文件中把 a,b,c 都定义为外部变量。在 F2.C 文件中用 extern 把三个变量说明为外部变量，表示这些变量已在其它文件中定义，并把这些变量的类型和变量名，编译系统不再为它们分配内存空间。 对构造类型的外部变量， 如数组等可以在说明时作初始化赋值，若不赋初值，则系统自动定义它们的初值为 0。

三、静态变量

静态变量的类型说明符是 static。 静态变量当然是属于静态存储方式，但是属于静态存储方式的量不一定就是静态变量， 例如外部变量虽属于静态存储方式，但不一定是静态变量，必须由 static 加以定义后才能成为静态外部变量，或称静态全局变量。 对于自动变量，前面已经介绍它属于动态存储方式。 但是也可以用 static 定义它为静态自

动变量，或称静态局部变量，从而成为静态存储方式。

由此看来， 一个变量可由 static 进行再说明，并改变其原有的存储方式。

1. 静态局部变量

在局部变量的说明前再加上 static 说明符就构成静态局部变量。
例如：

```
static int a,b;
static float array[5]={1,2,3,4,5}；
```

静态局部变量属于静态存储方式，它具有以下特点：

(1). 静态局部变量在函数内定义，但不象自动变量那样，当调用时就存在，退出函数时就消失。静态局部变量始终存在着，也就是说它的生存期为整个源程序。
(2). 静态局部变量的生存期虽然为整个源程序，但是其作用域仍与自动变量相同，即只能在定义该变量的函数内使用该变量。退出该函数后， 尽管该变量还继续存在，但不能使用它。
(3). 允许对构造类静态局部量赋初值。若未赋以初值，则由系统自动赋以 0 值。
(4). 对基本类型的静态局部变量若在说明时未赋以初值，则系统自动赋予 0 值。而对自动变量不赋初值，则其值是不定的。

根据静态局部变量的特点， 可以看出它是一种生存期为整个源程序的量。虽然离开定义它的函数后不能使用，但如再次调用定义它的函数时，它又可继续使用， 而且保存了前次被调用后留下的值。 因此，当多次调用一个函数且要求在调用之间保留某些变量的值时，可考虑采用静态局部变量。虽然用全局变量也可以达到上述目的，但全局变量有时会造成意外的副作用，因此仍以采用局部静态变量为宜。

2. 静态全局变量

全局变量(外部变量)的说明之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。 这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误。从以上分析可以看出， 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域， 限制了它的使用范围。因此 static 这个说明符在不同的地方所起的作用是不同的。应予以注意。

四、寄存器变量

8.7.4 函数参数与全局变量

在通用计算机上编写 C 程序的时候，为了尽量使程序模块化，尽量减小模块之间的耦合度，往往会将函数的所有与外界的联系都通过参数传递来进行，这样就使得函数完全被功能化，同时具有很强的复用性。这些正是 C 所具有的优势所在，正是由于 C 具有这样的特征才使得 C 具有很好的封装性，让这些函数很容易被封装成函数库，方便地提供任意的程序中调用。也正是有这么多的优势，才让我们选择了 C。

但是，SN8 C 是面对运算能力并不是很强的 8-bit 单片机处理核心，我们在关注程序的模块化，可维护性，封装性的优势的同时又不得不考虑程序的转换效率和运行效率。

由于 C 的封装特性，使得 C 程序非常简洁易懂，一个简单的加法赋值运算在汇编中会有很多条语句，而在 C 中则用一条非常简洁易懂的表达式就可以准确表达。从而使得 C 也显得更加简单易用。但是，忽略单片机的特征的 C 程序就不会是一个好的程序。

面对 C 的结构特征，用户必须定义适当数量的函数用于完成相应功能，这是符合 C 的编程思想，也符合整体程序编程思想的。而如何来兼顾单片机的硬件特性呢？正因为如此，就要求 SN8 C 的用户对硬件和对 SN8 ASM 应该有一定的了解。

我们在前面已经看到过汇编是如何来传递参数的，事实上它是借助了汇编当中的变量的全局特性来完成的。因为全局变量的值不会因为程序的跳转而发生改变，所以在跳转到子程序内部执行的时候，依然可以得到所需变量的值。同样，当在子程序中对确定的全局变量进行改变，当返回到主调程序时依然可以从相应地址得到变量的值。

SN8 C 的参数传递也采取类似的方法，编译器会为相应的函数参数定义相应个数的全局变量（可覆盖的）用于传递参数，尽管这样做已经是一种很优化的处理方案，但相比于汇编的全局变量来看，始终存在数据传递的环节（将调用函数时的实际参数传递给定义的全局变量），不如汇编高效。而当有多个参数需要传递时更增加了转换的代码量！

再从返回值方面来考虑，我们先来看看得到返回值的转换代码。

如：

```
keyPress = keyConvert(keyPress);
```

这条 keyFinishCHK 中的语句调用 keyConvert ( ) 函数，并将返回值存放到 keyPress 中。转换得代码如下：

```
;push arg....
__SelectBANK _keyFinishCHK_arg0
MOV A, (_keyFinishCHK_arg0)
__SelectBANK _keyConvert_arg0
MOV _keyConvert_arg0, A
```

```
;End push arg....
call _keyConvert
__SelectBANK (_keyFinishCHK_arg0)
MOV _keyFinishCHK_arg0, A
;end of function call
```

从中我们也可以看到返回的时候，都会有一个中间寄存器的转换(上例中的 A) ，当返回值只有一个 Byte 时，这个中间寄存器是 A，我们看不出差距。当用到其它寄存器时，就会有明显的代码消耗。

C 有了全局变量和局部变量的区分，从而可以将有限的 RAM 进行复用，从而达到节省资源的目的。同时由于与底层地址的剥离，使变量的命名更趋向人性化，这些都为用户带来了很大的方便。但在考虑使用局部变量的时候我们又不得不考虑代码空间的消耗。这样，通用计算机编程时的"尽量少用全局变量"的准则在单片机编程中就受到了挑战。在函数定义上，我们就需要权衡，在全局变量和参数之间进行取舍。对于面对硬件的单片机编程，我们还是建议多用全局变量来传递数值，而不是用参数。这样在产生代码时就会越贴近汇编的形式，产生代码的效率会越高。

当然，必要时例外！

8.8 结构体、联合在 SN8 C 程序中的应用

结构体、联合都是属于 C 的数据封装形式，它们在通用 C 程序中起着举足轻重的作用。同样，在编写 C 程序时，用好这些数据封装形式可以非常方便用户实现面对硬件的编程。

8.8.1 结构体

"结构"是一种构造类型，它是由若干"成员"组成的。 每一个成员可以是一个基本数据类型或者又是一个构造类型。 结构既是一种"构造"而成的数据类型， 那么在说明和使用之前必须先定义它，也就是构造它。如同在说明和调用函数之前要先定义函数一样。

结构的定义

定义一个结构的一般形式为：

```
struct 结构名
{
成员表列
};
```

成员表由若干个成员组成， 每个成员都是该结构的一个组成部分。对每个成员也必须作类型说明，其形式为：

类型说明符 成员名;

成员名的命名应符合标识符的书写规定。例如：

```
struct stu
{
   int num;
   char name[20];
   char sex;
   float score;
};
```

在这个结构定义中，结构名为 stu，该结构由 4 个成员组成。 第一个成员为 num，整型变量；第二个成员为 name，字符数组；第三个成员为 sex，字符变量；第四个成员为 score，实型变量。 应注意在括号后的分号是不可少的。结构定义之后，即可进行变量说明。 凡说明为结构 stu 的变量都由上述 4 个成员组成。由此可见， 结构是一种复杂的数据类型，是数目固定，类型不同的若干有序变量的集合。

结构类型变量的说明

说明结构变量有以下三种方法。以上面定义的 stu 为例来加以说明。

1. 先定义结构，再说明结构变量。如：

```
struct stu
{
   int num;
   char name[20];
   char sex;
   float score;
};
struct stu boy1,boy2;
```

说明了两个变量 boy1 和 boy2 为 stu 结构类型。也可以用宏定义使一个符号常量来表示一个结构类型，例如：

```
#define STU struct stu
STU
{
   int num;
   char name[20];
   char sex;
   float score;
};
STU boy1,boy2;
```

2. 在定义结构类型的同时说明结构变量。例如：

```
struct stu
{
   int num;
   char name[20];
   char sex;
   float score;
}boy1,boy2;
```

3. 直接说明结构变量。例如：

```
struct
{
   int num;
   char name[20];
   char sex;
   float score;
}boy1,boy2;
```

第三种方法与第二种方法的区别在于第三种方法中省去了结构名，而直接给出结构变量。说明了 boy1,boy2 变量为 stu 类型后，即可向这两个变量中的各个成员赋值。在上述 stu 结构定义中，所有的成员都是基本数据类型或数组类型。成员也可以又是一个结构， 即构成了嵌套的结构。

在 ANSI C 中除了允许具有相同类型的结构变量相互赋值以外， 一般对结构变量的使用，包括赋值、输入、输出、 运算等都是通过结构变量的成员来实现的。

表示结构变量成员的一般形式是： 结构变量名.成员名 例如：boy1.num 即第一个人的学号 boy2.sex 即第二个人的性别 如果成员本身又是一个结构则必须逐级找到最低级的成员才能使用。例如：boy1.birthday.month 即第一个人出生的月份成员可以在程序中单独使用，与普通变量完全相同。

结构变量的赋值

前面已经介绍，结构变量的赋值就是给各成员赋值。 可用输入语句或赋值语句来完成。

结构变量的初始化

如果结构变量是全局变量或为静态变量， 则可对它作初始化赋值。对局部或自动结构变量不能作初始化赋值。

结构体在存储形式上是按定义形式的先后，将成员逐个存储摆放形成一个数据块。这样就为程序的数据块操作提供了一个方便的途径。例如在显示屏上要依次序显示[0，5]，

[1，10]，[2，25]，[3，50]这样格式的数，定义一个数据结构如下：

```
Struct disvalue{
        Unsigned int number;
        Unsigned int Weight1;
        Unsigned int weight2;
            };
Disvalue  disarray[] = {{0,0,5},{1,1,0},{2,2,5},{3,5,0}};
```

在程序中我们就可以根据从程序操作过程中获得的 num 来进行显示。

```
Void diplayFun(unsigned int num)
{
    Disvalue display;
    Unsigned int disbuff[5]；
    Display = disarray[num];
    disbuff[0] = display.number;
    disbuff[4] = display.weight1;
    disbuff[5] = display.weight2;
}
```

结构体在硬件编程中还有一个常用的功能就是定义位域，其定义的方法如下：

```
Struct  结构体名称{
        Unsigned bit0:1;
        Unsigned bit1:2;
        Unsigned bit2:1;
        Unsigned bit3:1;
        Unsigned num:4;
      };
```

这个结构体中的成员数据类型都是 Unsigned，这是必须的。成员名称同样要符合变量命名规则，它与其他结构体不同的是成员名称后面有一个冒号，后面跟一个数据。冒号后面的数据是指占用位的数量，如成员 bit1 会占用 2 个 bit，而 num 会占用 3 个 bit，其他分别占用 1 个 bit。这样这个结构体就占用 1 个 Byte 的空间。在后面的应用我们会更详细地分析这一数据类型的应用。

同样，与通用 C 语言的编程相比，在 SN8 C 中有一些限制，对于 struct，我们只能对其实例安排存放的空间，加__RAM 和__ROM 限制字加于限制。而对于 struct 的成员我们就无法对其进行限制，这个原因是非常明显的，若是都允许对存放空间进行限制的话，就会造成冲突。如下定义是系统不允许的：

```
struct StuType {
        int __ROM data;                    // 错误
};
```

对结构体内部的成员变量，其存储的位置由整个结构体来决定。

8.8.2 联合体

"联合"与"结构"有一些相似之处。但两者有本质上的不同。在结构中各成员有各自的内存空间， 一个结构变量的总长度是各成员长度之和。而在"联合"中，各成员共享一段内存空间， 一个联合变量的长度等于各成员中最长的长度。联合体的特点在于其成员的存储空间相同，即联合体内部成员指向同一个空间，事实上是同一个空间通过不同的名称调用。如同时定义几种数据类型 char、int、long 组成一个联合体。那么，这就意味着修改三个成员中的任意一个都会影响到另外两个的值。

一、联合的定义

定义一个联合类型的一般形式为：

```
union  联合名
  {
        成员表
};
```

成员表中含有若干成员，成员的一般形式为： 类型说明符 成员名 成员名的命名应符合标识符的规定。
例如：

```
union perdata
{     int class;
   char office[10];
};
```

定义了一个名为 perdata 的联合类型，它含有两个成员，一个为整型，成员名为 class；另一个为字符数组，数组名为 office。联合定义之后，即可进行联合变量说明，被说明为 perdata 类型的变量，可以存放整型量 class 或存放字符数组 office。
联合在硬件编程中的应用最大的好处在于对存储空间的控制 ,我们来看看下面的这段汇编代码：

```
.DATA
    flag7               ds    1
        …
    micro_on_off_f      equ   flag7.0
    grill_on_off_f      equ   flag7.1
    motor_on_off_f      equ   flag7.2
    uv_on_off_f         equ   flag7.3
```

```
        …
.CODE
        …
    b0bset   motor_on_off_f          ;没有回到原来位置电机继续转

        …
    b0bts1   motor_on_off_f
    b0bclr   motor_port              ;关炉灯
    b0bts0   motor_on_off_f
    b0bset   motor_port


    b0bts0   grill_on_off_f
    jmp  $+3
    b0bclr   grill_port              ;关烧烤
             …
    clr  flag7                       ;关闭所有负载
```

这段代码中定义了一个变量 flag7，占一个 Byte。同时又对 Flag7 的前面 4 个位进行定义。因为汇编对硬件操作的灵活性，所以能很容易的对一个 Byte 进行整体操作，也很容易对这个 Byte 的每个位进行操作。我们用 struct 定义位域的方法，在 C 中也能对位进行单独的操作而不用通过与或等运算。我曾想用下面的方法来定义：

```
Unsigned int Flag7；
Struct bitFlag{
    Unsigned bit0：1；
    Unsigned bit1：1；
    Unsigned bit2：1；
    Unsigned bit3：1；
    Unsigned null：4；
        }pFlag7;
```

我想用一个结构体指针指向 Flag7，但是我发现并不能达到我的需求目的，那么该怎么来实现这上面的汇编所需实现的功能呢？当然，如果细想的话会有很多种方法，其中定义联合体就是一个很好的方法。我们在 C 中为了能对位进行直接的操作，一般都会定义一个以下形式的结构体：

```
Struct bitDefine{
    Unsigned bit0:1;
    Unsigned bit1:1;
    Unsigned bit2:1;
    Unsigned bit3:1;
    Unsigned bit4:1;
    Unsigned bit5:1;
    Unsigned bit6:1;
    Unsigned bit7:1;
```

```
};
```

那我们就可以利用这个定义来声明一个结构体放到一个联合体里，让它与一个 int 型变量共存。如：

```
union flagWord
{
 unsigned int flagByte;
 bitdefine flagBit;
};
```

我们就可以通过 union 来对相同地址的 Byte 和 Bit 进行操作。

对于上面的这些功能我们得到下面的 C 程序：

```
FlagWord flag7；
                Flag7.flagBit.bit2 = 1;          ;没有回到原来位置电机继续转
    …
If(flag7.flagBit.bit2) motor_port =1;
Else   motor_port = 0;                           ;关炉灯
    …
If(!flag7.flagBit.bit1) grill_port = 0;          ;关烧烤
Else
    …
Flag7.flagByte = 0;                              ;关闭所有负载
```

这段 C 程序可以实现上面汇编一模一样的功能。

同样的，将一个 Long 型数据与两个 int 型数据的结构体组成 Union，如：

```
union longtype
    {
        unsigned long longV;
        struct inttype
            {
                unsigned int int_l,int_h;
            }intV;
    };
```

通过定义 longType 的实例，我们就可以既对一个 long 型数据进行操作，又可以对数据的高低字节进行操作，而不用通过与或运算或移位就可以实现。这在我们的 8-bit 单片机的编程中，会用的比较频繁。这种的实现既兼顾了 C 的方便特性，又能有汇编的灵活性。这大概是 union 的最大优势了。

8.9 中断

在实时性程序中，中断功能是一个非常重要的功能，中断的实现状况会直接影响程序中的计时计数，更可能影响程序功能的正确实现。因此如何来实现中断是一个很有价值的问题。那么在 SN8 C 中又如何来实现中断？

8.9.1 中断函数的定义三

在芯片资源当中，大多是通过向量列表跳转的方法来实现程序的中断。在标准 C 中，所有的功能都是通过函数来实现，但是都有固定的中断函数，如 int86（）等，它们也都指向一个固定的中断向量，如 0x80 等系统固定的资源。

在 8-bit Sonix 芯片当中，都有一个中断向量 0x08，我们来看看汇编的中断程序实现：

```
.CODE
        org   00h

        jmp  Main_ST

        org   08H
        jmp  int_ser
             …

int_ser:
int_ser0:
        b0xch     a,accbuf
        mov       a,pflag
        b0mov     pflagbuf,a

        nop
int_ser10:
        b0bts1    fp00ien
        jmp  int_ser11
        b0bts0    fp00irq
        jmp  int_ser20            ;p00 中断
int_ser11:
        b0bts1    ft0ien
        jmp  int_ser19
        b0bts0    ft0irq
        jmp  int_ser40            ;t0 中断
int_ser19:
        jmp  int_ser9
int_ser20:                        ;p00 中断
        b0bclr    fp00irq
        ;activation
```

```
            jmp  int_ser9
int_ser40:                          ;T0 中断
            b0bclr     ft0irq
            b0bset     t0int        ;activation
            mov_       t0c,#64h
int_ser9:
            b0mov      a,pflagbuf
            mov pflag,a
            b0xch      a,accbuf
            reti
            …
```

上面的这段程序完成了从 0x08 的地址跳转并且根据设定的优先顺序判断中断类别（int_ser10—int_ser19），然后进行处理(int_ser20—int_ser40)的任务。其中中断开始和结束分别进行了寄存器的数据的 Push 和 pop 实现。

在 SN8 C 中，通过一个特殊函数来完成相同功能，这是一个 SN8 C 专有的函数，用关键字__interrupt 来声明。__interrupt 关键字(以两个底线开头)指示函数是要作中断向量的处理函数。中断向量函数是一个无参数, 也无返回值的特殊函数。

其声明方式如下:

```
__interrupt MyHandler () { .... }
```

我们来看一个 SN8 C 的中断函数：

```
__interrupt ints (void)                 //中断程序入口 ;1ms
{
    if(INTRQ&0x10) t0ints();
    else if(INTRQ&0x01) p00ints();      //过零点中断
}
void t0ints(void)
{
    _bCLR(&INTRQ,4);
    t_loop_f = 1;
    if(int_f) ++cnt11;                  //有外部中断触发开始计时继电器开关的时间

    T0C+=t0int_val+1;                   //T0 中断数值重装
    _bCLR(&INTRQ,4);
}
void p00ints(void)                      //20ms
{
    _bCLR(&INTRQ,0);
    int_f = 1;
    if(cnt11 >= 18)
    {
        cnt11 = 0;
```

```
            flag2.flagByte = 0xffU;
            ++t_ms2;
            if(t_ms2 == 50)
            {
                t_ms2= 0;
                flag10.flagByte = 0xffU;
            }
        }
    }
```

我们看到__interrupt 函数非常简单，只是按顺序判断中断的类别，调用相应的处理函数。这里与汇编的区别就是没有 0x08 的跳转和对寄存器的 push 和 pop。原来中断向量进入点会备份所有寄存器，中断向量结束前, 前述备份的项目均会被还原。这些都是由编译器内部完成的。

事实上，这正是 C 语言的优势所在，只有跨过了硬件关联才使得 C 有了跨平台特性！

用户若有其它额外的备份需求, 需要自行撰写 inline assemble (__asm("*asmcode*")或 __asm { … }),来完成或修改 sn8cc_macro.asm 中的 __pushInterruptSavedRegs 与 __popInterruptSavedRegs 。这里不对此进行详述！

8.9.2 中断过程的分析

中断产生后，系统都完成了哪些动作，对于单片机程序编写者来讲还是需要进了解才能使编写的程序达到最佳的效率。

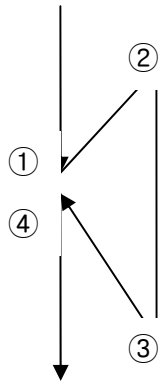


图 7-1 中断的过程

上图所描绘的是一个中断的过程。程序在主循环中运行到□的时候，中断条件成立，系统产生中断，此时，系统将会去执行中断程序。而为了能从中断中正确返回，在进入中断程序之前，系统会对当前的状态（ACC，PC 等等）进行保存。然后，在□处程序运行进入中断程序，□处中断程序结束，系统又需要将原来的运行状态还原，然后在□处继续运行主循环程序。一次中断完成。

8.9.3 中断函数的结构

在了解中断的过程后我们再来看看中断函数应该用一个什么样的结构。由于我们的中断往往涉及我们的计时，同时它还是占用主循环的时间来运行的。那么，要保证主循环的实时性，就要求中断占用的时间越短越好。那么，就要求我们在中断里不能做太多事情！

同时我们可能有几个中断源，我们在系统产生中断的时候就必须去判断是哪个中断源产生的中断。
鉴于以上要求，我们可以对中断程序作这样的安排：

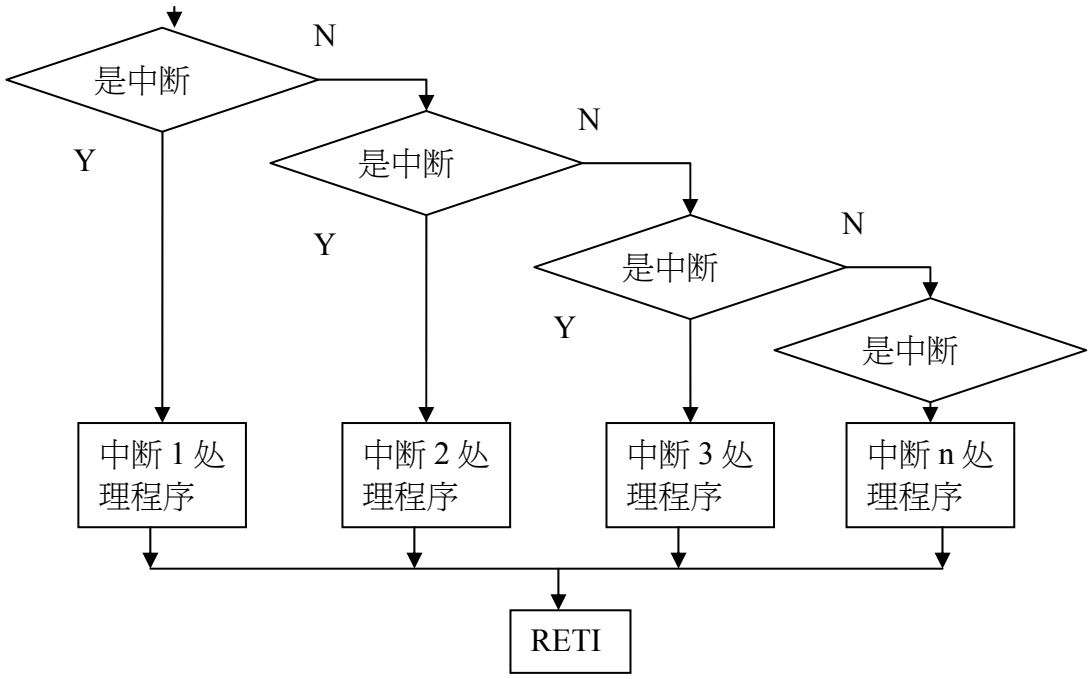


图 7-2　中断处理程序结构图

为了使中断资源不被长时间占用，我们的中断程序内不能运行任何长时间占用系统时间的程序！那么，我们怎么安排中断中的程序呢？其实占用中断资源的任务，我们完全可以安排到中断外去完成!我们只需要告诉主控程序发生了中断以及是哪个中断就可以了！

如下图：

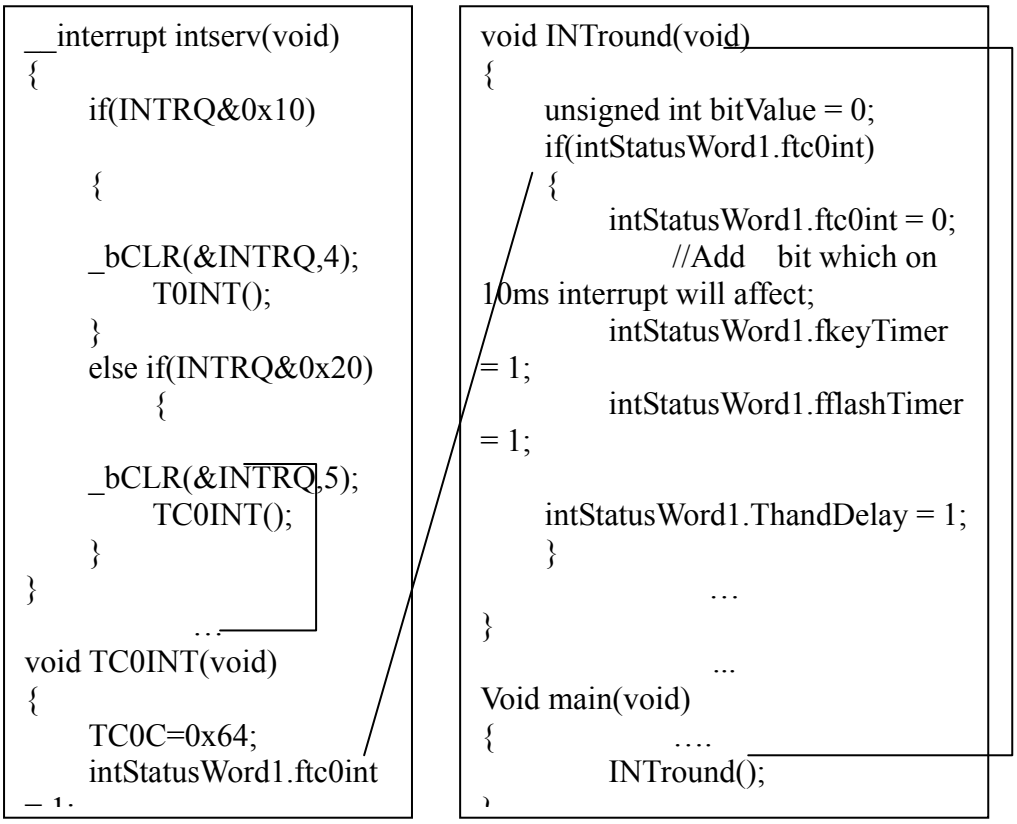


图 7-3 中断程序与主循环的关联

在上面图中的程序里面，我们看到中断程序只是判断是由哪个中断源引发的中断，然后调用中断处理程序，而中断处理程序只是将中断的状态恢复，然后将相应中断的自定义标志位置 1。而主循环设置中断任务安排程序 INTround（），由它通过检测相应的中断标志 ftc0int 去检测中断的发生，并且安排任务，然后在主循环中调用执行它。这样既能及时发现中断产生的请求，并及时处理，又不会过多地占用中断资源。

8.10 位操作

位是计算机的最基本存储单位，与其相对应的数值表示是二进制。事实上，硬件的所有数据的处理都是通过二进制来处理的，并且 SN8 系列的单片机的 RAM 的全部空间都是可位操作的。如此，位操作在我们的单片机程序中的重要性便不言而喻了。

8.10.1 位的定义

由于对位操作在单片机程序有着举足轻重的地位，所以，有很多类型的单片机 C 语言提供了 bit 型变量的定义，如 C51。而 SN8 C 则和 ANSI C 一样，并没有 bit 型变量的定义方法。为了更好地标识我们所需要的每一个标志位，方便我们区分并进行操作，增强程序的可读性。我们都会寻求一种办法来对位进行定义，并给我们需要的位赋予一个我

们容易理解的名称如 keyPress，int10ms 等等。

我们在前面已经提到过位域的定义，我们通过 struct 来定义一个 Byte 当中的每一个位，如：

```
Struct bitDefine{
        Unsigned bit0:1;
        Unsigned bit1:1;
        Unsigned bit2:1;
        Unsigned bit3:1;
        Unsigned bit4:1;
        Unsigned bit5:1;
        Unsigned bit6:1;
        Unsigned bit7:1;
            };
```

我们就可以定义一些具体的结构体实例：

```
Struct bitDefine flag1,flag2,flag3;
```

然后我们用宏定义的方法将我们需要的位名称赋予相对应的位。如：

```
#define fkeypress (flag1.bit1)          //确认有一个键按下的标志
#define fchatfinish (flag1.bit2)        //按键 Debounce 完成标志
#define fkeyProcessing (flag1.bit3)     //有键正在处理的标志
#define FhandDelay (flag1.bit4)
#define FhandDelayRQ (flag1.bit5)
#define FreleaseKey (flag1.bit6)        //按键释放标志位
#define FfirstAck (flag1.bit7)
```

通过这样的转换，我们就可以在后续的程序中直接用我们自己定义的名称来对每一个定义过的位进行处理了！

上面所说明的处理方法都是对用户自己定义的位进行的操作，其实还有一个重要的存储器部分是系统定义的。这就是芯片的系统寄存器（system register），这也是我们使用最频繁的寄存器。系统已经对需要用到的资源中的寄存器进行了定义，并被赋予了固定的名称，为其编写了固定的操作函数。用户只管拿来用就好了。

对于位域的定义尚有以下几点说明：

1. 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
   unsigned a:4
   unsigned :0 /*空域*/
   unsigned b:4 /*从下一单元开始存放*/
   unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

2. 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。
3. 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
{
   int a:1
   int :2 /*该 2 位不能使用*/
   int b:3
   int c:2
};
```

从以上分析可以看出，位域在本质上就是一种结构类型， 不过其成员是按二进位分配的。

类型定义符 typedef

C 语言不仅提供了丰富的数据类型，而且还允许由用户自己定义类型说明符，也就是说允许由用户为数据类型取"别名"。 类型定义符 typedef 即可用来完成此功能。例如，有整型量 a,b,其说明如下： int aa,b; 其中 int 是整型变量的类型说明符。int 的完整写法为 integer.

为了增加程序的可读性，可把整型说明符用 typedef 定义为： typedef int INTEGER 这以后就可用 INTEGER 来代替 int 作整型变量的类型说明了。 例如： INTEGER a,b;它等效于： int a,b; 用 typedef 定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单而且使意义更为明确，因而增强了可读性。例如：

```
typedef struct stu{ char name[20];
   int age;
   char sex;
} STU;
```

定义 STU 表示 stu 的结构类型，然后可用 STU 来说明结构变量： STU body1,body2;

typedef 定义的一般形式为：

typedef 原类型名 新类型名

其中原类型名中含有定义部分，新类型名一般用大写表示， 以便于区别。在有时也可用宏定义来代替 typedef 的功能，但是宏定义是由预处理完成的，而 typedef 则是在编译时完成的，后者更为灵活方便。

8.10.2 位的运算

对位的操作主要包含以下几种：置位，清除，位与(&)，位或(|)，位非(~)，位异或(^)，左移(<<)，右移(>>)。

对于自定义的位，对于上面的这些操作举例如下：

```
Fkeypress = 1;      //置位
Fkeypress = 0;      //清除
```

而位与(&)，位或(|)，位异或(^)，左移(<<)，右移(>>)基本上是基于基本数据类型上的位操作，如：

```
keyinbuf <<=2;
tempbuf = P0&0x03;
keyinbuf |= tempbuf;
keyinbuf = ~keyinbuf;
```

这些运算目的都是改变或者获得基本数据类型里面的位的值。这就是我们常用的逻辑尺等等东西。这些操作在生成.asm 是转换为逻辑运算指令，上面的语句会被转换为：

```
L209:
    ;_keyinbuf = _keyinbuf << 2
    ;__SelectBANK _keyinbuf
    RLCM (_keyinbuf)
    RLCM (_keyinbuf)
    MOV A, #0xfc
    AND (_keyinbuf), A
L210:
    B0MOV A, 0xd0
    AND A, #(0xFF & 3)
    __SelectBANK _ee_scale_c_keyscan_LOCAL
    MOV _ee_scale_c_keyscan_LOCAL+1, A
L211:
    ; R3 = _keyinbuf | _ee_scale_c_keyscan_LOCAL+1
    __SelectBANK _keyinbuf
```

```
        MOV A, (_keyinbuf+0)
        __SelectBANK _ee_scale_c_keyscan_LOCAL
        OR A, (_ee_scale_c_keyscan_LOCAL+1)
        B0MOV R3, A
        B0MOV A, R3
        __SelectBANK _keyinbuf
        MOV _keyinbuf, A
L212:
        __SelectBANK (_keyinbuf)
        MOV A, (_keyinbuf)
        B0MOV R3, A
        B0MOV A, R3
        XOR A, #0xff
        B0MOV R3, A
        B0MOV A, R3
        __SelectBANK _keyinbuf
        MOV _keyinbuf, A
```

而如果我们操作的对象是系统寄存器中的位，系统给我们提供了专门的位操作函数其原型如下：

```
void _bSET(unsigned long address, unsigned int bitOffset);
void _bCLR(unsigned long address, unsigned int bitOffset);
int _bTest0 (unsigned long address, unsigned int bitOffset);
int _bTest1(unsigned long address, unsigned int bitOffset);
```

其中传入参数 address 与 bitOffset 必须为常数，不可为变量。

BitOffset 有效值为 0 ~7
address 的高位为 bank number

前面两个是置位和清除函数，专门用于对系统寄存器进行位的置位和清除。
如我们要启动 TC0，那么我们就将其 TC0ENB（TC0M.bit7）位标志置 1：

```
_bSET（&TC0M,7）;
```

而如果要停止 TC0，那么我们就清除其 TC0ENB（TC0M.bit7）位标志：

```
_bCLR（&TC0M,7）;
```

这两个函数也具有很高的转化效率，其转化代码分别为：

```
PreB0SET 218 7 0
PreB0CLR 218 7 0
```

后两个是位判断函数，其对给定的位判断为 1 或者为零，并返回一个 int 型数值。

8.10.3 位比较在程序流程控制中的应用

在程序中往往都通过判断一个或几个条件是否成立来控制程序的执行。而条件的成立就会用标志位来标示，这样位的判断比较在程序中就非常重要了!例如下面例子中的判断条件：

```
if(globalSW.tareRQ)                    //进行零位处理的话，不进行下面的处理
{
    captrueZero(stabledata);
}
```

上面的判断是对自定义位 globalSW.tareRQ 进行的判断，若为 1 则执行函数调用，若为 0 则跳过函数调用，转而执行下面的语句。产生的汇编码如下：

```
    __SelectBANK _globalSW
    BTS1 _globalSW.3
    JMP L430
L450:
    ;push arg....
    __SelectBANK _stabledata
    MOV A, _stabledata+1
    ;Select BANK
    __SelectBANK _captrueZero_arg0
    MOV (_captrueZero_arg0+1), A
    __SelectBANK _stabledata
    MOV A, _stabledata
    __SelectBANK _captrueZero_arg0
    MOV _captrueZero_arg0, A
    ;End push arg....
    call _captrueZero
    ;end of function call
L451:
.stabn 0x44,0,587,L452-_procADC
L452:
    jmp L431
```

而对系统寄存器的位判断则允许有下面三种方式：

一是用上面提到的专用函数：

```
if(_bTest1(&INTRQ,4))          //FT0IRQ)
{
    _bCLR(&INTRQ,4);           //FT0IRQ=0;
```

```
        T0INT();
}
```

一是用系统定义的位名称：

```
if(FT0IRQ)                      //(_bTest1(&INTRQ,4))
{
    _bCLR(&INTRQ,4);            //FT0IRQ=0;
    T0INT();
}
```

用位运算的方法：

```
if(INTRQ&0x10)                  //FT0IRQ
{
    _bCLR(&INTRQ,4);            //FT0IRQ=0;
    T0INT();
}
```

但是，上面 3 种方式产生代码的效率并不一样。
当然，它们的运行效果是一样的，都能起到分支判断的作用。

8.11 预处理

8.11.1 概述

在前面已多次使用过以"#"号开头的预处理命令。如包含命令# include，宏定义命令# define 等。在源程序中这些命令都放在函数之外， 而且一般都放在源文件的前面，它们称为预处理部分。

所谓预处理是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所作的工作。预处理是 C 语言的一个重要功能， 它由预处理程序负责完成。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分作处理， 处理完毕自动进入对源程序的编译。

C 语言提供了多种预处理功能，如宏定义、文件包含、 条件编译等。合理地使用预处理功能编写的程序便于阅读、修改、 移植和调试，也有利于模块化程序设计。

8.11.2 宏定义

在 C 语言源程序中允许用一个标识符来表示一个字符串， 称为“宏”。被定义为“宏”的标识符称为“宏名”。在编译预处理时，对程序中所有出现的“宏名”，都用宏定义中的字符串去代换， 这称为“宏代换”或“宏展开”。

宏定义是由源程序中的宏定义命令完成的。 宏代换是由预处理程序自动完成的。在 C 语言中，“宏”分为有参数和无参数两种。 下面分别讨论这两种“宏”的定义和调用。

无参宏定义

无参宏的宏名后不带参数。其定义的一般形式为：

#define 标识符 字符串

其中的“#”表示这是一条预处理命令。凡是以“#”开头的均为预处理命令。“define”为宏定义命令。 “标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。在前面介绍过的符号常量的定义就是一种无参宏定义。 此外，常对程序中反复使用的表达式进行宏定义。例如： # define M (y*y+3*y) 定义 M 表达式(y*y+3*y)。在编写源程序时，所有的(y*y+3*y)都可由 M 代替，而对源程序作编译时，将先由预处理程序进行宏代换，即用(y*y+3*y)表达式去置换所有的宏名 M，然后再进行编译。程序中 s=3*M+4*M+5* M 中作了宏调用。在预处理时经宏展开后该语句变为：s=3*(y*y+3*y)+4(y*y+3*y)+5(y*y+3*y);但要注意的是，在宏定义中表达式(y*y+3*y)两边的括号不能少。否则会发生错误。 在作宏定义时必须十分注意。应保证在宏代换之后不发生错误。

对于宏定义还要说明以下几点：

1. 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单的代换，字符串中可以含任何字符，可以是常数，也可以是表达式，预处理程序对它不作任何检查。如有错误，只能在编译已被宏展开后的源程序时发现。
2. 宏定义不是说明或语句，在行末不必加分号，如加上分号则连分号也一起置换。
3. 宏定义必须写在函数之外，其作用域为宏定义命令起到源程序结 束。如要终止其作用域可使用# undef 命令。
4. 宏名在源程序中若用引号括起来，则预处理程序不对其作宏代换。
5. 宏定义允许嵌套，在宏定义的字符串中可以使用已经定义的宏名。在宏展开时由预处理程序层层代换。
6. 习惯上宏名用大写字母表示，以便于与变量区别。但也允许用小写字母。
7. 可用宏定义表示数据类型，使书写方便。

应注意用宏定义表示数据类型和用 typedef 定义数据说明符的区别。宏定义只是简单的字符串代换，是在预处理完成的，而 typedef 是在编译时处理的，它不是作简单的代换，

而是对类型说明符重新命名。被命名的标识符具有类型定义说明的功能。宏定义虽然也可表示数据类型， 但毕竟是作字符代换。在使用时要分外小心，以避出错。

带参宏定义

C 语言允许宏带有参数。在宏定义中的参数称为形式参数， 在宏调用中的参数称为实际参数。对带参数的宏，在调用中，不仅要宏展开， 而且要用实参去代换形参。

带参宏定义的一般形式为：

#define 宏名(形参表) 字符串

在字符串中含有各个形参。

带参宏调用的一般形式为：

宏名(实参表)；

例如：

```
#define M(y) y*y+3*y /*宏定义*/
k=M(5); /*宏调用*/
```

在宏调用时，用实参 5 去代替形参 y， 经预处理宏展开后的语句为： k=5*5+3*5

对于带参的宏定义有以下问题需要说明：

1. 带参宏定义中，宏名和形参表之间不能有空格出现。
2. 在带参宏定义中，形式参数不分配内存单元，因此不必作类型定义。而宏调用中的实参有具体的值。要用它们去代换形参，因此必须作类型说明。这是与函数中的情况不同的。在函数中，形参和实参是两个不同的量，各有自己的作用域，调用时要把实参值赋予形参，进行"值传递"。而在带参宏中，只是符号代换，不存在值传递的问题。
3. 在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。这与函数的调用是不同的， 函数调用时要把实参表达式的值求出来再赋予形参。 而宏代换中对实参表达式不作计算直接地照原样代换。
4. 在宏定义中，字符串内的形参通常要用括号括起来以避免出错。对于宏定义不仅应在参数两侧加括号， 也应在整个字符串外加括号。
5. 带参的宏和带参函数很相似，但有本质上的不同，除上面已谈到的各点外，把同一表达式用函数处理与用宏处理两者的结果有可能是不同的。函数调用和宏调用二者在形式上相似， 在本质上是完全不同的。
6. 宏定义也可用来定义多个语句，在宏调用时，把这些语句又代换到源程序内。

8.11.3 文件包含

文件包含是 C 预处理程序的另一个重要功能。

文件包含命令行的一般形式为：

#include"文件名"

在前面我们已多次用此命令包含过库函数的头文件。
文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行， 从而把指定的文件和当前的源程序文件连成一个源文件。在程序设计中，文件包含是很有用的。 一个大的程序可以分为多个模块，由多个程序员分别编程。 有些公用的符号常量或宏定义等可单独组成一个文件， 在其它文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写那些公用量， 从而节省时间，并减少出错。

对文件包含命令还要说明以下几点：

1. 包含命令中的文件名可以用双引号括起来，也可以用尖括号括起来。但是这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找(包含目录是由用户在设置环境时设置的)， 而不在源文件目录去查找； 使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。 用户编程时可根据自己文件所在的目录来选择某一种命令形式。
2. 一个 include 命令只能指定一个被包含文件， 若有多个文件要包含，则需用多个 include 命令。
3. 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

8.11.4 条件编译

预处理程序提供了条件编译的功能。 可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。 这对于程序的移植和调试是很有用的。 条件编译有三种形式，下面分别介绍：

第一种形式：

```
#ifdef identifier
Program 1
#else
Program 2
#endif
```

它的功能是，如果标识符已被 #define 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(它为空)，本格式中的#else 可以没有。

第二种形式：

The second form:
#ifndef identifier
Program 1
#else
Program 2
#endif

与第一种形式的区别是将"ifdef"改为"ifndef"。它的功能是，如果标识符未被#define 命令定义过则对程序段 1 进行编译， 否则对程序段 2 进行编译。这与第一种形式的功能正相反。

第三种形式：

#if constant-expression
Program 1
#else
Program 2
#endif

它的功能是，如常量表达式的值为真(非 0)，则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可以使程序在不同条件下，完成不同的功能

上面介绍的条件编译当然也可以用条件语句来实现。 但是用条件语句将会对整个源程序进行编译，生成的目标代码程序很长，而采用条件编译，则根据条件只编译其中的程序段 1 或程序段 2， 生成的目标程序较短。如果条件选择的程序段很长， 采用条件编译的方法是十分必要的。

8.12 内嵌汇编

C 语言和别的语言一样，虽然具有非常多的优点，但也免不了存在一些局限性。有的时候，对硬件操作的功能无法实现，或者实现的代码所花费的时间空间代价太大。这时，我们还是不得不考虑使用汇编，毕竟其具有高效率和紧贴硬件的特性。正出于这样的考虑，SN8 C 支持在 C 程序中内嵌汇编。

8.12.1 如何内嵌汇编

和一般的 C 语言一样，SN8 C 提供专门的关键字__asm（两个下划线）用于在 C 的源代码内嵌入汇编。

__asm 关键词有以下两种用法:

__asm(“*code***\n”)**

__asm { *asm_text* **}**

这两种方式都可以在 C 中内嵌汇编，在 C 中是作为一条语句来处理，当它处在顶层时（也就是说没有被任何函数所包含）就可以内嵌任意的汇编码到输出的汇编码中。

在 asm_text 中的 Macros 和其他的 C 指示性语句在预处理中不被处理，除非选择 **cpp_noskip_asmblock** 编译选项。**cpp_noskip_asmblock** 选项会对插入到 asm_text 中的 cpp 指示性语句和 Macros 进行处理。在 asm_text 中的 C 和 C++ 类型的注释仍然会被忽略。在 asm_text 里面，不能带有右括符(i.e., “)”)，除非这个符号是字符串附带的。

我们来看看在 C 中嵌入汇编的例子：

```
if(step == ONE_PRESS_CLOCK_KEY_C)
{                   //调节时钟时时间闪烁
    disp_blink ^= 0x03;                 //闪烁小时位
    disp_blink &=0x03;
    __asm
    {
        mov     a,0x01
        mov     _led_dp,a
    };
    //led_dp = 1;               //两点常亮
}
```

我们来看看其产生的汇编码，看看内嵌汇编语句是怎么被处理的：

```
L125:
    __SelectBANK _step
    MOV A, (_step+0)
    SUB A, #0x10
    JNZ L93
L128:
    __SelectBANK _disp_blink
    MOV A, #3
    XOR   _disp_blink, A
L129:
    ;__SelectBANK _disp_blink
    MOV A, #3
    AND   _disp_blink, A
L130:
    mov     a,0x01
L131:
    mov     _led_dp,a
L132:
L133:
```

```
        jmp L94
```

我们看到内嵌的汇编被原样转换嵌入生成的汇编码当中（L130，L131），那么其运行的结果自然就是我们所需要的了。

8.12.2 内嵌汇编时变量的传递

我们知道，在汇编当中与在 C 当中定义变量的方法是不同的。但是，我们在内嵌汇编当中也需要对由 C 定义的变量进行操作。该如何来处理这样的一个矛盾呢？

如下 C 程序代码：

```
void func(void)
{
    int x;

     __asm {
…
;; 存取局部变量 x
…
    }
    return;
}
```

经编译的结果:

```
C:\sn8cc>sn8cc +w prog.c
SN8CCWARN@(`prog.c' 3): source_warning: local `int x' is not referenced
prog.c: 0 error(s), 1 warning(s)
```

上例中, 因为 SN8 C Compiler 认为变量 x 没有用到(SN8 C Compiler 不检查内嵌式汇编码), +w 编译选项会让 SN8 C Compiler 报告警告讯息。且不会为变量 x 配置空间。这样内嵌式组语码内所存取的位置是错误的。

为了解决这样的问题，SN8 C 专门引入"**#pragma ref** *id*"预处理指令，上述问题可用"**#pragma ref** *id*"通知 SN8 C Compiler, 变数 x 是有用到的:

```
void func(void)
{
    int x;

#pragma ref x
     __asm {
…
;; 存取局部变量 x.
```

```
…
        }
        return;
}
```

这样 SN8 C Compiler 才不会报告警告讯息。

而我们还会面对的一种情况是在内嵌汇编当中要对全局变量进行读写操作，这又是一个什么情况呢？

假如我们直接应用 C 定义的名称，如：

```
Unsigned int Ver1；
Void func(void)
{
        …
        __asm{
                Mov     ver1,#0x5a;
                …
            };
        …
}
```

编译则系统会提示出错！警告用户没有 Ver1 这样一个变量！这时候很奇怪了，明明有定义这样一个变量，可是系统就是提示 Error！

其实，这就是由于在编译当中，编译器会对内嵌汇编进行原样转换所引发的错误。我们来看前面嵌入汇编的例子中的全局变量在产生的.Asm 档中，对全局变量是怎么转换的。

C 源程序中对全局变量的定义如下：

```
union flagWord2
{
  unsigned int flagByte;
  struct bitdefine2
  {
        unsigned bit0:1;
        unsigned bit1:1;
        unsigned bit2:6;
  }flagBit;
}led_dp;
unsigned int door_cnt;
unsigned int door_cnt1;
unsigned int door_cnt2;
```

而这些定义在生成的汇编中被转换成如下形式：

```
.stabs "led_dp:G43",32,0,0,_led_dp
.stabs "door_cnt2:G14",32,0,0,_door_cnt2
.stabs "door_cnt1:G14",32,0,0,_door_cnt1
.stabs "door_cnt:G14",32,0,0,_door_cnt
```

原来它们将每一全局变量的前面都增加了一个"_"来标识它们！

那么我们就可以找到解决的方法了，我们在嵌入的汇编中，如果要读写全局变量，就在它们的前面加上一个"_"，其实我们提到的例子中的 led_dp 就是这样子的！回过头去看看这条语句：

```
mov     _led_dp,a
```

上面的 led_dp 就是一个全局变量，而在内嵌汇编码中被增加了一条下划线。

需要提醒用户的是，变量转换的语法并不属于程序员掌握的范畴，这里也没有详细介绍，使用当中尽量少用这方面的转换，以免引起程序查错上的麻烦。这些应用必须是对生成.asm 档有一定的了解才行。

8.13 其它

以下项说明关于 SN8 C COMPILER 的规定：

1. 标识符最多为1022个字符；
2. 字符串最多为4095个字符；
3. 此编译器支持指针功能，使用"call @HL"。如果目标板不支持"@HL"，程序将报错，编译器不再检查目标板是否支持"@HL"；
4. 函数指针不能带参数；

Ex.

```
int (*fptr)(int);
int foo(int);
int main()
{
    fptr = foo;
    fptr(1);    // error，fun pointer can not pass argument
    return 0
}
```

8.14 SN8 C 自定义库

Character Class Tests: <ctype.h>

```
isalnum(c);
isalpha(c);
iscntrl(c);
isdigit(c);
isgraph(c);
islower(c);
isprint(c);
ispunct(c);
isspace(c);
isupper(c);
isxdigit(c);
int tolower(int c);
int toupper(int c);
```

String Functions: <string.h>

```
char *strcpy(char *，const char *);
char *strncpy(char *，const char *，size_t);
char *strcat(char *，const char *);
char *strncat(char *，const char *，size_t);
int strcmp(const char *，const char *);
int strncmp(const char *，const char *，size_t);
char *strchr(const char *，int);
char *strrchr(const char *，int);
size_t strspn(const char *，const char *);
size_t strcspn(const char *，const char *);
char *strpbrk(const char *，const char *);
char *strstr(const char *，const char *);
size_t strlen(const char *);
char *strerror(int);
char *strtok(char *，const char *);
void *memcpy(void *，const void *，size_t);
void *memmove(void *，const void *，size_t);
int    memcmp(const void *，const void *，size_t);
void *memchr(const void *，int，size_t);
void *memset(void *，int，size_t);
```

Mathematical Functions: <math.h>

```
float sin(float);
float cos(float);
float tan(float);
float asin(float);
float acos(float);
float atan(float);
float atan2(float，float);
float sinh(float);
float cosh(float);
float tanh(float);
float exp(float);
float log(float);
float log10(float);
float pow(float，float);
float sqrt(float);
float ceil(float);
float floor(float);
float fabs(float);
float ldexp(float，int);
float frexp(float，int *);
float modf(float，float *);
float fmod(float，float);
```

Utility Functions: <stdlib.h>

```
int atoi(const char *);
long atol(const char *);
long strtol(const char *，char **，int);
long strtoul(const char *，char **，int);
int rand(void);
void srand(unsigned);
int abs(int);
long labs(long);
div_t div(int，int);
ldiv_t ldiv(long，long);
```

Variable Argument Lists: <stdarg.h>

```
Type: va_list
Macro    : va_start(va_list，lastarg)
Macro    : type va_arg(va_list，type)
Macro    : void va_end(va_list)
```

Limits: <limits.h>

```
CHAR_BIT : 8
CHAR_MAX : 127
CHAR_MIN : -128
INT_MAX : 127
INT_MIN : -128
LONG_MAX : 32767
LONG_MIN : -32768
SCHAR_MAX : 127
SCHAR_MIN : -128
SHRT_MAX : 127
SHRT_MIN : -128
UCHAR_MAX : 255
UINT_MAX : 255
ULONG_MAX : 65535
USHRT_MAX : 255
```

<float.h>

```
FLT_RADIX
FLT_ROUNDS
FLT_DIG
FLT_EPSILON
FLT_MANT_DIG
FLT_MAX
FLT_MAX_EXP
FLT_MIN
FLT_MIN_EXP
DBL_DIG
DBL_EPSILON
DBL_MANT_DIG
DBL_MAX
DBL_MAX_EXP
DBL_MIN
DBL_MIN_EXP
```

Standard defined types: <stddef.h>

```
ptrdiff_t : unsigned int
size_t : unsigned int
Input Output data: <stdio.h>
printf(char*，...)
sprintf(char*，const char*，...)
```

在调用“printf”函数时，必须用“putchar”函数预先定义“printf”函数的输出类型。
例如：void putchar(char ch)。

9. 链接和调试

9.1 链接程序做了什么？

目标代码连接器接受大量的ELF目标和档案文件，一个描述程序段地址的链接文件将随一个可选的map文件（记录修改或变化的地址信息）生成可执行的二进制和（或）ELF文件。连接器在目标文件中结合了代码和数据，并寻找定义了的库函数把外部参量转化为执行程序和变量。它也将代码和数据段放置在指定的内存地址，如果没有明确的地址则放置在默认地址。最后，连接器将程序代码和其他信息复制到目标文件中。这就是所要调试的目标文件。Holtek库管理器生成交叉连接器所包含的库函数。

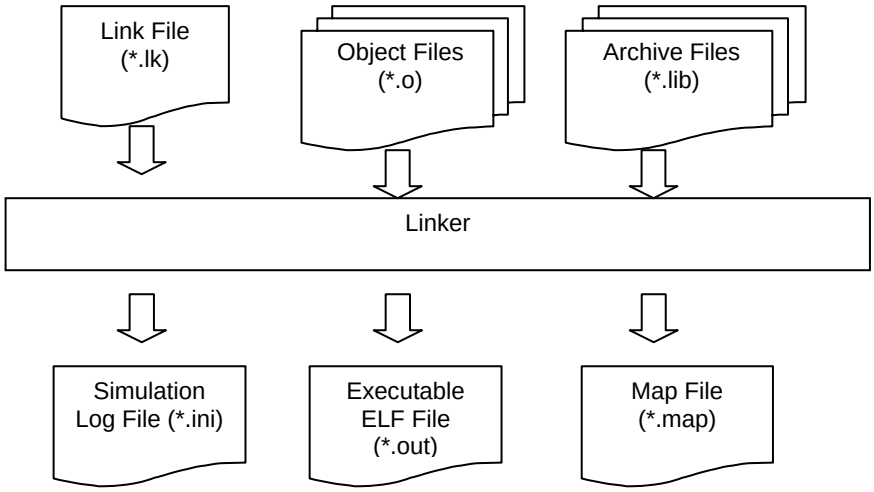


Figure9-1 Linking processing

链接过程：

- 分析整个程序符号在库函数中输入和寻找所包含的模块；
- 段分配；
- 分解局部变量和全局变量；
- 修改再定义的地址；
- 生成输出文件；
- 生成运行记录文件供仿真寄存器转存。

9.2 链接选项

连接器通过这些选项指定并控制目标文件。工程管理器提供一个 Linker Options 对话框指定连接器的这些选项：

输出

此显示可执行 ELF 文件的指定路径。

Map 文件

此选项指定是否生成 map 文件和其相关路径。map 文件应该记录下列信息：

- 与 map 文件有关的可执行文件路径名；
- 时间标记记录 map 文件生成的时间和日期。全局符号表提供了全局符号的地址、类型及存储文件的信息；
- 修改记录；
- 段内存的规划和他们的基本信息；
- 调用树状信息。

堆栈

由用户指定堆栈层数。

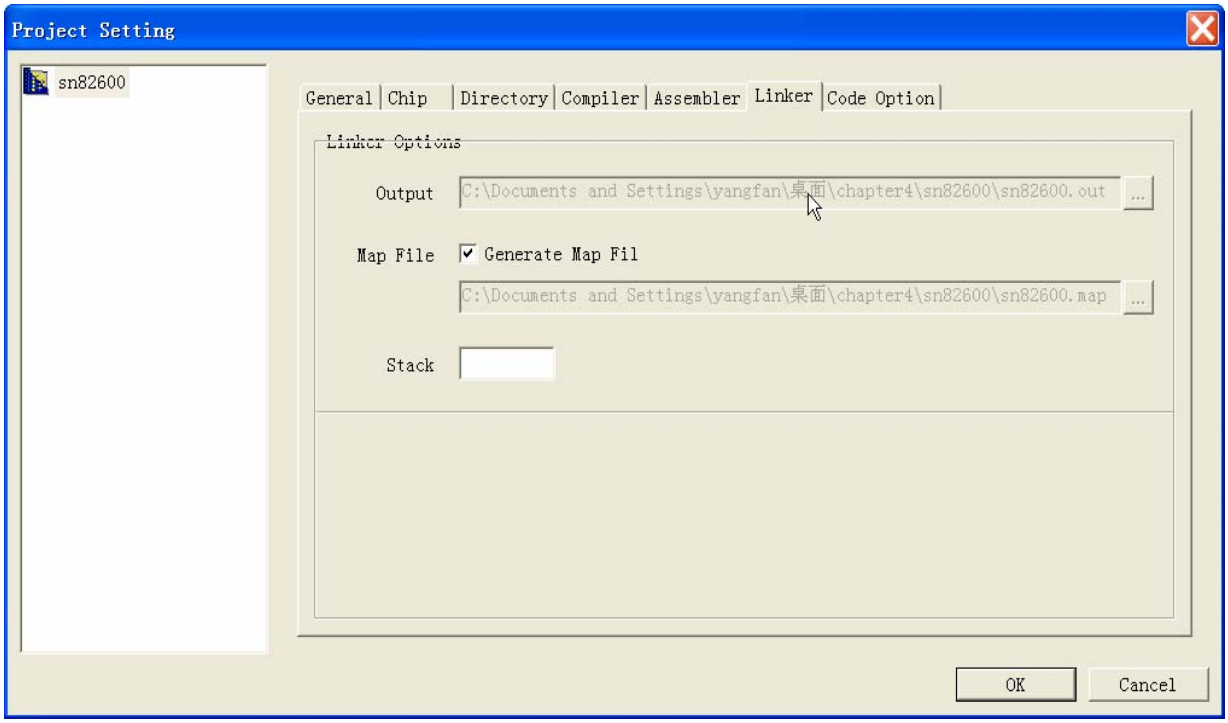


Figure 9-2 Linker Options

9.3 功能介绍

9.3.1 Linker

slink [options] [files]

Options: (case insensitive)
1. Help - /H or /HELP or /?
2. Target machine - /MACHINE:*id*
3. Link file - /LINKFILE:"file"
4. Library search path - /PATH:*"path"*
5. Init file - /INI:"*file_name*"
6. Map file - /MAP.
7. Output file - /OUTPUTFILE:*filename*
8. Stack - /STACK:*size*
9. Case sensitive - /CASE
10. Standard Library Version – /STDLIB:*version*
11. *Output Path - /OUTPUTPATH:path*
12. *C Type - /CSource*
13. *Disable Data Overlaying - /NOOVERLAY*

Examples:

slink /MACHINE:SNC745 /PATH:"c:\library" a.o b.o c.o

文件:
1. ELF 格式的目标文件；
2. 文档文件。

链接文件:
1. 连接命令文件（请查阅连接文件格式规格书）；
2. 如果命令行选项和连接文件中的命令之间有冲突，连接器应选择连接文件中的命令，忽略命令行选项。

9.3.2 Librarian

slib [Options] [files]

Options:
1. Help - /H or /HELP or /?
2. Target machine - /MACHINE:*id*
3. List objects - /LIST
4. Output file name - /OUTPUT:*filename*
5. Search Path - /PATH:*path*
6. Extraction - /EXTRACT:*object*

7. Insertion - /INSERT:*object*
8. Deletion - /REMOVE:*object*
9. Case sensitive - /CASE
10. Init file - /INI:"*file_name*"
11. Output Path - /OUTPUTPATH:*path*

Examples:

slib /MACHINE:SN735 /OUTPUT:ar.lib a.o b.o c.o

文件：
1. ELF 格式的目标文件；
2. 文档文件。

9.3.3 Dump 用法

sdump [Options] [file]

Options:
1. All information - /ALL
2. All Header information - /HEADERS
3. Section - /SECTIOIN:*sectionname*
4. Symbol table - /SYMBOLS
5. Relocations - /RELOCATIONS
6. Debug information - /DEBUG
7. Ouput file path name - /OUTPUT:*filename* or
   /OUTPUT:*"full path name with space in it"*

Example :

sdump /all /output:a.out.dmp a.out
sdump /debug /output:"c:\path\project 1\a.out.dmp" a.out

文件:
1. ELF formatted object and executable file.
2. Archive file.5.4 File Format

9.4 Map File 格式

Executable File = .\Output/SonixDemo.out
Timestamp is Mon Jun 14 11:06:57 2004

Start	Length	Type	Name	Module
0X000000	0X0004	Code	zeroaddr	SONIXDEMO.o
0X010000	0X009E	Code	.code1	SONIXDEMO.o
Start	Length	Type	Name	Module

0X000000	0X020E	Data	.data	SONIXDEMO.o
Address	Type	Size	Publics	Module
Address	Type	Size	Locals	Module
0X000000	Section	0X000	.DATA	SONIXDEMO.o
0X010000	Section	0X000	.CODE1	SONIXDEMO.o
0X010036	No type	0X000	@@@test_start	SONIXDEMO.o
0X000100	Object	0X001	LAST_KEY0	SONIXDEMO.o

Log Configuration File Format (INI)：

运行记录文件做为仿真器的输入转存仿真时运行时间寄存器的值。运行记录文件记录地址和寄存器，或记录在文件中被记录的内存信息。连接器的责任就是生成名为"xxx.out.slg"的运行记录文件。

```
[General]
  StopAt=0x10000 ; LOG STOP_PROGRAM for ges.exe
  [Item0]
  DumpAt=0x3f
  Register=x0, y0, x1, y1
  Ram=0x0-0x10, 0x200
  [Item1]
  DumpAt=0x4b
  Register=ALL_REGISTERS
  Ram=0x10, 0x20, 0x30, 0x40
  [Item2]
  DumpAt=0x400000
  Register=
  Ram=0x0-0x10, 0x20
  [Item3]
  DumpAt=0x4000a4
  Register=ALL_REGISTERS
  Ram=0x0-0x10
```

9.5 Error and Warning 消息

1. 描述程序源文件中的错误消息，以支持未来多种语言的开发。
2. 请遵照下面所示语法。SN8 C STUDIO 将解析其语法并在消息窗显示消息的正确信息，用户可以在编辑器中双击此消息找到源文件所在行。

[Error | Warning] error_number : error message description

For example:

Error *2001: multiply defined global symbol ("symbol").*
Warning 2001: multiply defined global symbol ("symbol"), second definition is ignored.

注: 更多 Error/Warning 消息，请開啓 SN8 C Studio 環境按鍵盤 F1 参阅在"SONiX SN8 C Studio Tool Help"文件中"Tool chains Error/Warning Messages"一章節。

9.6 Debugger

在开发过程中，重复的修改和测试源程序是必不可少的过程。SN8 C STUDIO 为用户提供了一个调试环境，包括单步执行、设置断点、自动单步执行和跟踪触发条件等等。

成功汇编和生成应用程序后，可以使用调试命令对应用程序进行仿真。需要注意的是，要仿真一个应用程序，应该打开相应的工程。

选择调试菜单中的 Begin Debug 命令或按下工具栏的调试按钮就可以开始仿真。如果遇到断点，系统将自动停止仿真，否则，仿真一直持续到应用程序结束。End Debug 按钮在 ICE 仿真时可用，按下此键停止仿真。

9.7 Single Step

用户可能会想查看某些指令的执行结果。而且，每次只可能查看一条指令的执行结果，也就是以单步的方式。SN8 C STUDIO 提供两种单步模式：手动模式和自动模式。手动模式下，每执行一次 single-step 命令就执行一条指令；在自动模式下，系统持续执行单步命令直到执行了 stop 命令。在自动模式下，所以指定的断点均不可用。这里有3条步进指令：Step Into、Step Over 和 tep Out。

Step Into 命令每次只执行一条指令。执行此命令将进入程序进程并且当遇到 CALL 指令时在第一条指令处就停止。
Step Over 命令每次只执行一条指令，即使遇到调用程序，也将在执行完调用子程序后停在 CALL 指令后面那条指令处。调用子程序中的所有指令将被执行且寄存器内容和状态也可能会改变。

Step Out 命令只用在子程序中，将执行当前点到 RET 指令之间的所有指令（包括 RET 指令），且 PC 将停在 CALL 指令后的那条指令处。

9.8 Breakpoints

SN8 C COMPILER 有强大的断点机制，接受各种条件包括程序地址、源文件行数和标志断点等等。

当在一条指令处设置有效断点，SN8 C COMPILER 将不执行这条指令且停在此处。当 SN8 C COMPILER 在循环或条件跳转时不管指令是否设置了有效断点也不会在此指令

处停止执行。如果有效断点在数据区（RAM），且指令和条件断点值相匹配时仍然会执行这条指令，PC 停在下一条指令处。

选择调试菜单中的 Breakpoint 命令，弹出设置断点对话框供用户指定对断点的描述。设置要设断点的文件和准确行数，选择断点类型，单击 OK 按钮确定。

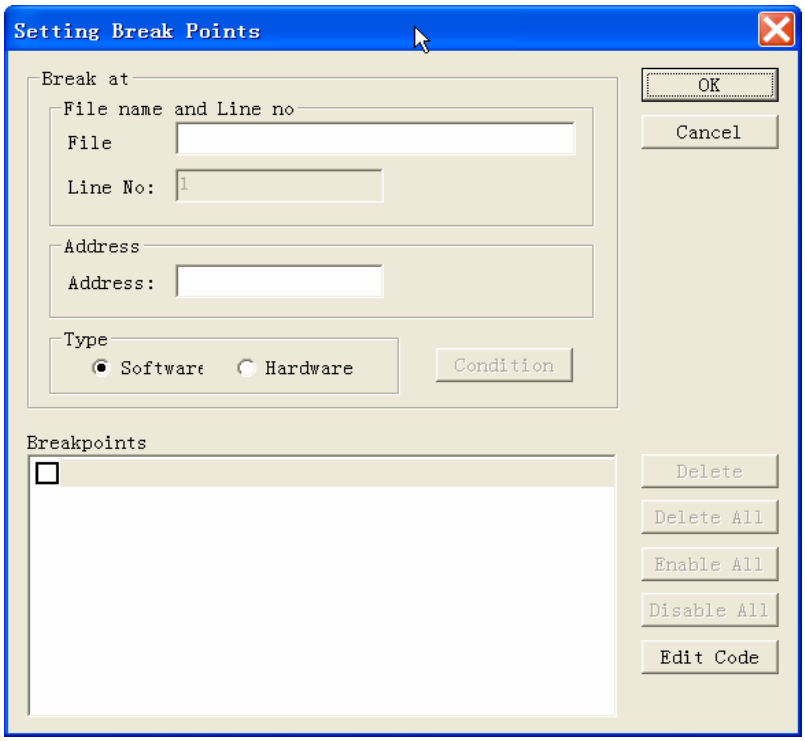


Figure 9-3 Breakpoint Setting

附录 A FAQ

本章列出用户在使用 SN8 C STUDIO 时可能遇到的问题和解决办法。合理使用这部分内容能很好的提高用户的开发效率。

注: 关于FAQs的详细信息 请開啓SN8 C Studio環境按鍵盤F1参阅在"SONiX SN8 C Studio Tool Help" 文件中"FAQs" 一章節。

Main Office:
Address: 9F, NO. 8, Hsien Cheng 5th St,Chupei City, Hsinchu, Taiwan R.O.C.
Tel: 886-3-551 0520
Fax: 886-3-551 0523

Taipei Office:
Address: 15F-2, NO. 171, Song Ted Road, Taipei, Taiwan R.O.C.
Tel: 886-2-2759 1980
Fax: 886-2-2759 8180

Hong Kong Office:
Address: Flat 3 9/F Energy Plaza 92 Granville Road, Tsimshatsui East Kowloon.
Tel: 852-2723 8086
Fax: 852-2723 9179

Technical Support by Email:
Sn8fae@sonix.com.tw