WicePlus C Compiler

for EM78 Series Microcontrollers

USER'S GUIDE

Doc. Version 2.0

ELAN MICROELECTRONICS CORP.

Feb 2007

Trademark Acknowledgments

IBM is a registered trademark and PS/2 is a trademark of IBM.
Windows is a trademark of Microsoft Corporation.
ELAN and ELAN logo are trademarks of ELAN Microelectronics Corporation.

The contents of this User's Guide (publication) are subject to change without further notice. ELAN Microelectronics assumes no responsibility concerning the accuracy, adequacy, or completeness of this publication. ELAN Microelectronics makes no commitment to update, or to keep current the information and material contained in this publication. Such information and material may change to conform to each confirmed order.

In no event shall ELAN Microelectronics be made responsible for any claims attributed to errors, omissions, or other inaccuracies in the information or material contained in this publication. ELAN Microelectronics shall not be liable for direct, indirect, special incidental, or consequential damages arising from the use of such information or material.

The software (WicePlus) described in this publication is furnished under a license or nondisclosure agreement, and may be used or copied only in accordance with the terms of such agreement.

ELAN Microelectronics products are not intended for use in life support appliances, devices, or systems. Use of ELAN Microelectronics product in such applications is not supported and is prohibited.
NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS WITHOUT THE EXPRESSED WRITTEN PERMISSION OF ELAN MICROELECTRONICS.

ELAN MICROELECTRONICS CORPORATION

Headquarters:

No. 12, Innovation Road 1
Hsinchu Science Park
Hsinchu, Taiwan 30077
Tel: +886 3 563-9977
Fax: +886 3 563-9966
http://www.emc.com.tw

Hong Kong:

Elan (HK) Microelectronics Corporation, Ltd.

Rm. 1005B, 10/F Empire Centre
68 Mody Road, Tsimshatsui
Kowloon , HONG KONG
Tel: +852 2723-3376
Fax: +852 2723-7780
elanhk@emc.com.hk

USA:

Elan Information Technology Group

1821 Saratoga Ave., Suite 250
Saratoga, CA 95070
USA
Tel: +1 408 366-8223
Fax: +1 408 366-8220

Shenzhen:

Elan Microelectronics Shenzhen, Ltd.

SSMEC Bldg., 3F, Gaoxin S. Ave.
Shenzhen Hi-Tech Industrial Park
Shenzhen, Guandong, CHINA
Tel: +86 755 2601-0565
Fax: +86 755 2601-0500

Shanghai:

Elan Microelectronics Shanghai Corporation, Ltd.

23/Bldg. #115 Lane 572, Bibo Road
Zhangjiang Hi-Tech Park
Shanghai, CHINA
Tel: +86 021 5080-3866
Fax: +86 021 5080-4600

进版说明

用户如何在WicePlus2的平台上使用C编译器？首先，新旧版本的C编译器差别不大，操作基本雷同，就是说在这两种版本中用户都可以建立一个新的工程，但要注意这两个版本的不同之处：

1. 卸载WicePlus1.XXX。在安装WicePlus2之前，用户必须先完全卸载以前版本的WicePlus，也就是说在卸载的过程中，用户必须选择remove的选项。卸载以后，用户必须安装WicePlus2.，因为新版C编译器和WicePlus2相关联。
2. 在用户建立的工程里删除system.inc 和 sysdef.inc这两个文件。假如用户在D:\develop\的路径下建立了旧版本的工程prg1.c，这样在同一个文件夹里就会有两个C编译器的系统文件system.inc 和 sysdef.inc，用户必须先删除D:\develop\下面的这两个文件。
3. 明确分配rpage , iopage, bank。在旧版本里，rpage 0, iopage 0 和 bank 0可以被省略，如果用户在这些寄存器定义变量时不用特别申明。但在新版本中，用户不能再省略rpage 0, iopage 0 和 bank 0，这些“ 0 ”状态的变量也必须明确申明。对于只有一个rpage , iopage 或 bank的微控制器，在定义变量时也必须明确申明是哪一个page 或 bank。
4. 新版本的C编译器编译效率较高，用户可以参考本文件的第57页的转换表。
5. 编译器将不定地占用通用寄存器，它将提醒用户在中断服务程序里哪些通用寄存器需要保存和备份，请参考5.10.3章节。

我们希望能提供给用户一个理想的开发工具，如果您在使用 C 编译器的过程中有任何的问题，请随时联系我们，您可以发邮件到下面的地址：

myjian@emc.com.tw
john.cheng@emc.com.tw

目录

附录

用户手册版本历史		
版本	版本描述	日期
1.0	用户手册最初版本。	2005/07/27
1.1	在 4.5 章节数据类型里增加了长型数据类型的使用方法到注释里。	2005/08/31
1.2	增加了整型数据和无符号整型数据类型。	2006/07/27
2.0	用户必需明确申明寄存器所在的 rpage、iopage 及 bank。	2006/02/12
2.1	1. 在中断保护程式里增加了“PAGE @0X0”语句。 2. 在 5.10 章节的范例与注释里详细描述了中断备份与恢复的用法。 3. EM78569, EM78367 与 EM78369 等 IC 增加了嵌入汇编的乘法指令“MUL”。 4. 优化了 c=(a+b) << 1; unsigned int a, b, c. 5. 在 4.3 章节增加了关于#include “xxx.c”使用的注释。 6. 在第 26 页举例说明了多源文档编程的应用。 7. 编译器将不定地占用通用寄存器，WicePlus 将告诉用户在中断服务程式里哪些寄存器（0x10~0x1F）需要保存和备份，请参考 5.10.3 章节。	2006/03/

第一章
介绍

1.1 概述

EM78系列的C编译器是允许用户使用C语言来实现他们应用的一个辅助代码转换器。用户的源代码可以通过此编译器编译成汇编代码而生成机器代码。

注意

- *请注意WICEPLUS只能安装在预先设好的目录底下（C:\EMC\WicePlus），该限制是为了避免用户分配一个在编译时会产生比较严重错误的包含空格字符的安装路径。*
- *在文件路径**(.cpj , *.c or *.h)**内不能包含空格字符,如果有空格字符,在编译时将出现问题。*

1.2 系统要求

1.2.1 主机要求

安装EM78系列C编译器的计算机需满足如下要求：

- IBM PC (Pentium 100 或以上)或相兼容机器
- Win2000, WinME, NT, or WinXP
- 至少 10M 以上的硬盘空间
- 至少 16M 的 RAM，推荐 30M 或更高
- 强烈推荐有鼠标和 USB 接口

1.3 软件安装

编译器包含在WICEPLUS,EM78系列的开发环境内。当安装WICEPLUS时,编译器也将被安装进去了。

1.4 ANSI 兼容性

遵从ANSI标准受限于独立的C，以适应EM78系列微控制器独特的设计特征。

第二章
WicePlus 界面

2.1 概述

WICEPLUS是一个集成开发系统的工具,此系统是用来编辑用户应用程序和产生义隆EM78系列8位控制器的仿真文件。

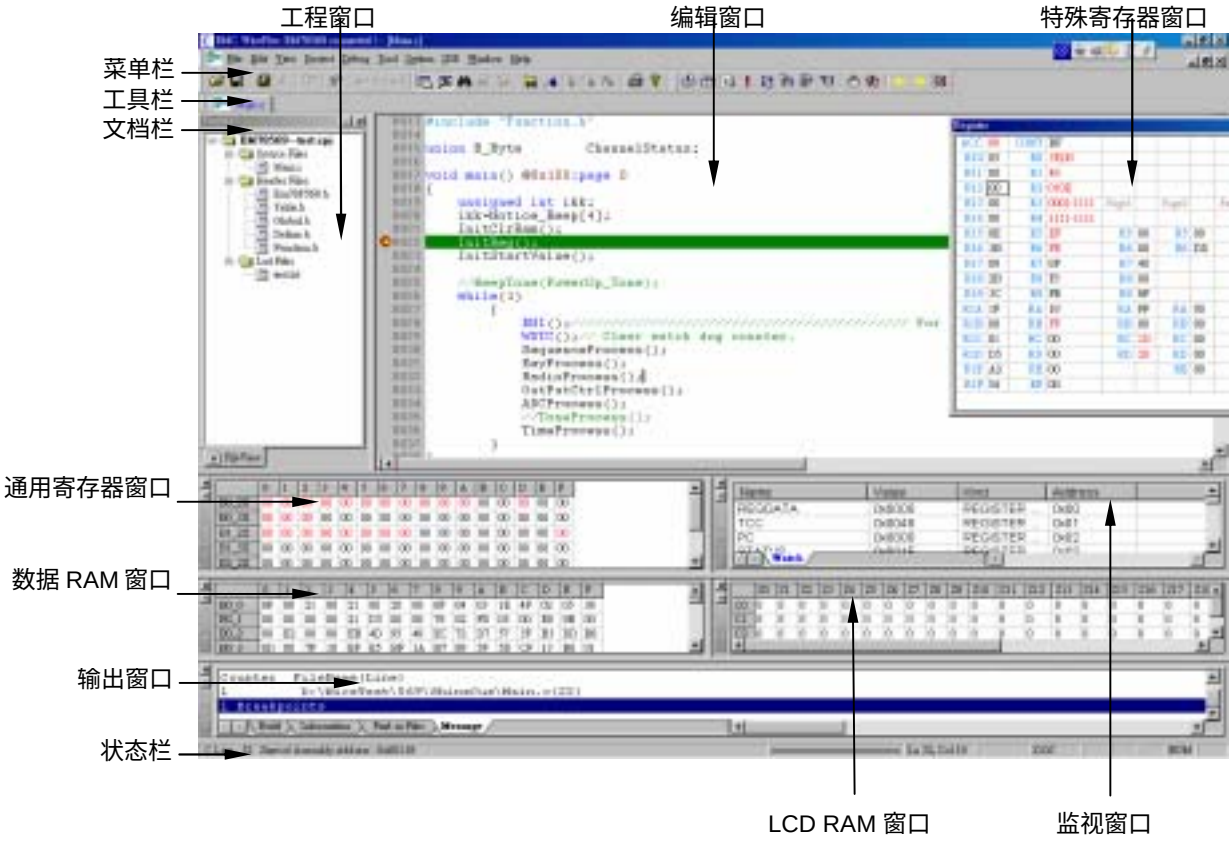
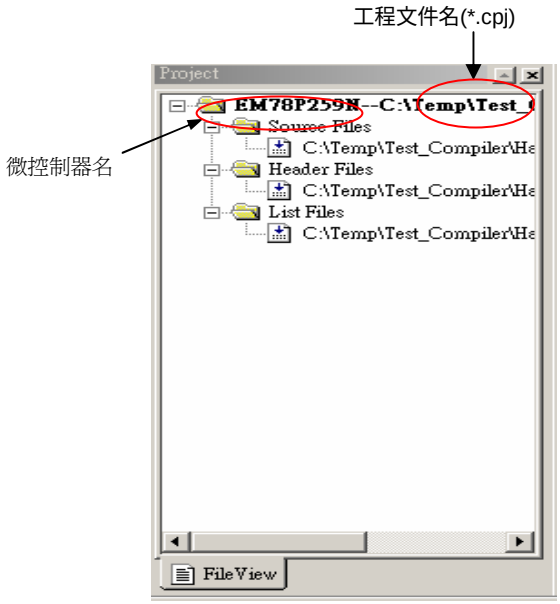


Fig. 2-1 WicePlus 主窗口布局

2.2 WicePlus 子窗口

子窗口可通过点击视图菜单内相关的窗口命令来显示或隐藏. (见4.3.3.3 部分)。

2.2.1 工程窗口



工程窗口包含源文档、头文件、LIST文件和MAP文件。

Where:

源文档 (*.c) – 加载到当前工程文件内需要编译的源文件；

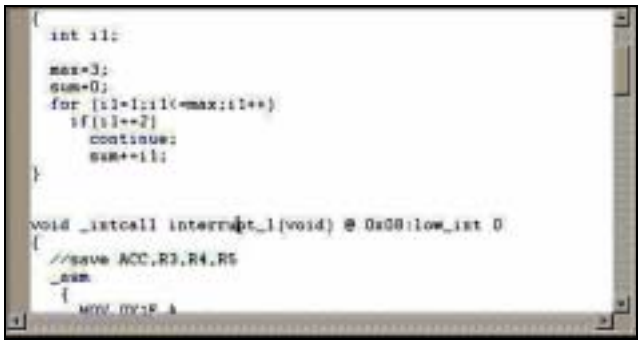
头文件 (*.h)– 源程序所需的参数文件；

List 文件 (*.lst) – 列表文件。

Fig. 2-2 工程窗口

工程窗口的标题栏显示了当前微控制器和工程文件名。

2.2.2 编辑窗口



编辑窗口是一个创建、查看和调试源程序的多窗口编辑工具。

编辑窗口的主要特点是：

- 不限制文件大小
- 同一时间可以打开和显示多个文件

Fig. 2-3 编辑窗口

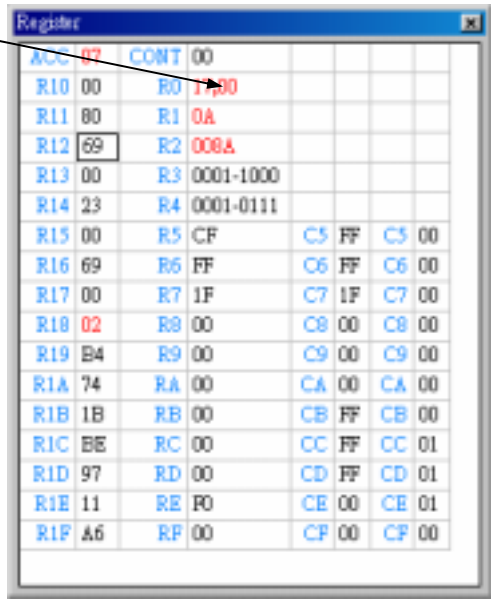
- 插入模式以编辑
- 撤销/重做

支持剪切板（用按键可将文本剪切、拷贝、移动和粘贴到剪切板）

- 拖放文本操作（选定的文本能够在任何开发环境窗口内拖放）

2.2.3 特殊寄存器窗口

当寄存器的值改变时，对应的数字将变为红色



特殊寄存器窗口显示寄存器的最新内容和相对应型号的 MCU 的 I/O 控制寄存器。

Fig. 2-4 特殊寄存器窗口

2.2.4 通用寄存器（RAM区）窗口

Ram Bank

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B0_2X	E2	6D	31	E7	7E	F8	BD	6D	70	C7	4B	9F	14	3E	6F	EE
B0_3X	10	77	C7	FD	B5	27	38	7F	6B	66	6E	BE	D1	47	45	9A
B1_2X	0B	34	4D	16	4C	E4	51	7E	32	F4	70	B7	03	E1	50	4F
B1_3X	D6	77	18	39	80	FB	F4	8F	FE	65	DD	FC	B3	FF	28	4B
B2_2X	DD	D7	59	C1	F1	6F	8A	E7	B2	79	3B	FC	F2	76	89	B6
B2_3X	C5	23	4D	FF	5D	54	FC	FC	D2	7B	69	E0	0E	BF	2B	C3
B3_2X	95	C4	F8	F6	8F	E7	C3	FF	EF	5F	AE	E1	86	6B	22	59
B3_3X	61	AB	74	F5	19	43	CE	ED	18	3C	64	DC	AA	DB	3F	3F

Fig. 2-5 通用寄存器（Ram区）窗口

通用寄存器窗口显示最新的通用RAM内的数据。

2.2.5 监视窗口

Fig. 2-6 监视窗口

用户可以往监视窗口内增加在C里面申明过的变量。监视窗口会显示定义的C变量的信息，比如名称、内容、寄存器所在的Bank及地址。

往监视窗口增加变量的步骤：

1. 反白变量(例如aa)；
2. 点击鼠标右键，即弹出对话框；
3. 选择“ Add to Watch ”项。

2.2.6 “ 数据RAM ”窗口

Call Ram Window

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B0_0	2A	6A	1B	4B	0B	9F	4C	CB	C2	D9	30	34	5B	EC	46	02
B0_1	60	D8	48	B7	51	FE	46	66	10	F5	39	D3	B0	9D	9F	93
B0_2	7E	9F	E3	EF	C9	DF	64	55	81	DB	36	58	7C	75	89	FF
B0_3	60	2B	35	BE	71	C4	81	F5	89	82	28	BB	9F	6E	F1	ED
B0_4	D5	FB	7B	76	B2	6A	A3	4D	6A	ED	1D	1C	08	5E	06	7A
B0_5	1E	2F	8B	E3	32	D8	59	FC	A2	5B	FC	BF	EA	B6	58	E6
B0_6	3F	2D	C1	06	04	AA	6D	9E	52	FF	15	DD	7E	EF	94	B2
B0_7	A3	C6	AD	D6	33	C4	36	C7	89	B3	18	5D	62	96	CA	FD
B0_8	B2	7D	DA	77	E4	9D	AE	7F	62	B3	53	35	10	31	2A	EF

Fig. 2-7 数据RAM窗口

数据RAM窗口仅仅对当前使用的微控制器可以使用，数据RAM窗口显示数据RAM的内容。

2.2.7 “LCD RAM” 窗口

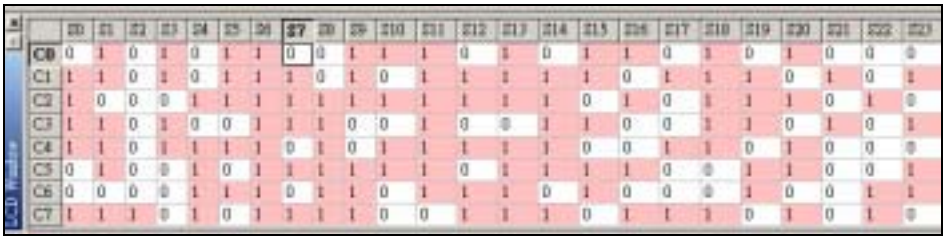


*Fig. 2-8 **LCD RAM** 窗口*

当前使用的微控制器如果支持LCD RAM，那么LCD RAM窗口将会显示LCD RAM的内容。“Cx”表示LCD的“COMx”信号，而“Sx”则表示“SEGMENTx”。

双击被选择的部分（方格），可以修改LCD RAM的内容，内容的颜色会从无色(0)变成粉红色(1)。任何相关的信息都将显示在输出窗口。

2.2.8 “输出”窗口

Fig. 2-9 输出窗口

输出窗口会显示当前正在编译的工程结果(包含错误)的信息，例如汇编信息、连接信息、跟踪和调试等的过程。这个窗口包含了四个子窗口，即：编译、ROM&RAM信息、在文件里寻找和LCD RAM调试信息，详述如下：

编译 – 显示汇编程序/连接器的相关信息和跟踪日志，双击错误信息会链接到相应源程序错误信息所在的行。若相关的源文档当前没有被激活，那么它将会在编辑窗口内被自动打开。

ROM&RAM信息 – 显示正在调试的相关ROM和RAM的使用信息。

文件里寻找 – 允许在文件夹内其它激活的或者没有激活的文件里寻找同一字符串(从激活的文件里选择)。包含相同字符串的文件，连同它的源文件名和目录都会显示在输出窗口内。

LCD RAM信息 – 显示正在调试的对LCD RAM的相关改变的信息。

2.3 WicePlus 菜单栏与它的命令

Fig. 2-10 菜单栏

2.3.1 文件菜单

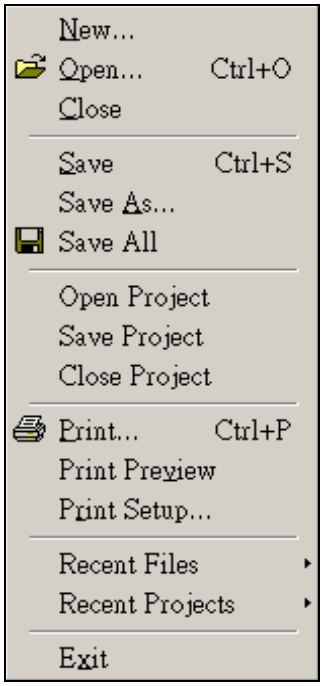


Fig. 2-11 菜单栏

New…	创建一新的工程或源文件
Open…	打开已有的文档或工程
Close	关闭激活的文档或工程
Save	保存当前激活文档
Save As	用一新文件名保存当前激活的文档
Save All	保存所有打开的文档
Open/Save/Close Project	打开/保存/关闭激活的工程
Print	打印激活的文件
Print Preview	预览激活文件的打印格式
Print Setup…	打印机设定
Recent Files	查看最新使用文件的记录
Recent Projects	查看最新使用工程的记录
Exit	退出WicePlus程序

2.3.2 编辑菜单

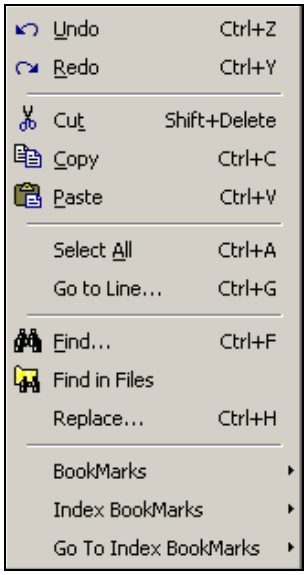
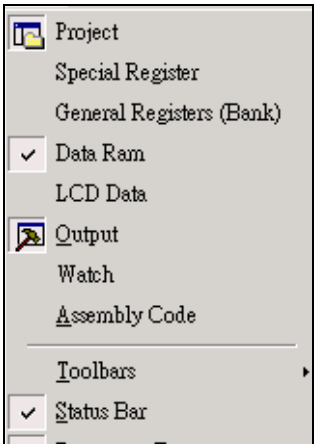


Fig. 2-12 编辑菜单

Undo	撤销编辑
Redo	重复编辑
Cut/Copy/Paste	使用标准剪贴板功能
Select ALL	选择激活窗口的所有内容
Go to Line…	移动光标到激活窗口的指定行
Find…	在激活的窗口内查询定义过的字符串
Find in Files	在激活与非激活的文件里查询定义过的字符串
Replace…	查找与替换功能
Bookmarks	在光标所在位置的行置一标记
Index Bookmarks	清除掉所有的标识或者分配一标识值(0~9)，以便使用下面的“**Go to Index Bookmarks**”命令比较容易找到它们
Go to Index Bookmarks	跳到指定值“x”的标识处

2.3.3 查看菜单



Project 显示/隐藏工程窗口

Special Registers 显示/隐藏特殊寄存器窗口

General Registers (Bank) 显示/隐藏通用寄存器窗口

Data RAM 显示/隐藏数据RAM窗口(如果该IC支持)

LCD Data 显示/隐藏LCD RAM窗口(如果该IC支持)

Output 显示/隐藏输出窗口

Watch 显示/隐藏监视窗口

Assembly Code 显示/隐藏编辑窗口内的编译代码

Toolbars 显示/隐藏标准、编译或两者的工具栏

Status Bar 显示/隐藏状态栏

Document Bar 显示/隐藏文档栏

2.3.4 工程菜单

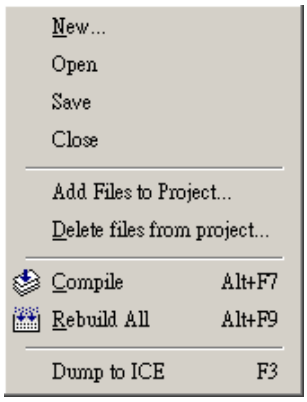


Fig.2-14 工程菜单

New… 新建工程

Open 打开已有工程

Save 保存所激活的工程文件及相关的文件

Close 关掉激活的工程窗口

Add Files to Project… 增加已有的源文件到工程文件内

Delete Files from Project… 从工程文件内删除源文件

Compile 编译编辑窗口内激活的文件

Rebuild All 编译所有文件

Dump to ICE 下载程序代码到ICE

2.3.5 调试菜单

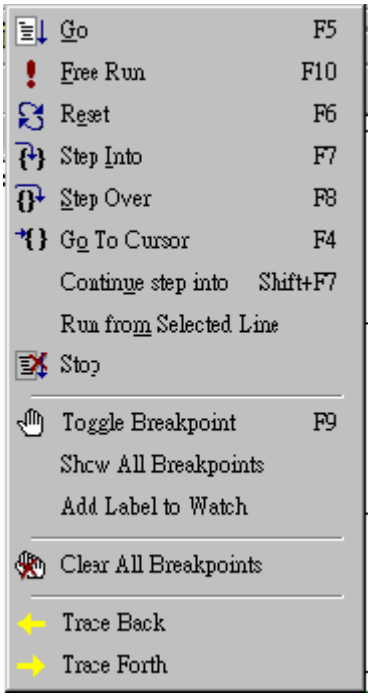


Fig.2-15 调试菜单

Go	从当前指针开始运行直到遇到断点为止。
Free Run	从当前指针开始运行，直到按“STOP Running”对话框的OK按钮停止。
Reset	ICE复位(寄存器内容显示初始值)。
Step Into	单步运行并进入CALL子程序（寄存器的内容同时被更新）
Step Over	单步运行，但若遇到CALL子程序会跳过，不会进入CALL子程序 。
Go to Cursor	从当前指针运行到光标所在位置(只使用于调试模式)。
Continue step into	连续单步调试，用户可看到寄存器的变化，不同于“Go”。
Run from Select Line	从光标所在行开始运行程序。
Stop	停止运行程序。
Toggle Breakpoint	设置或清除断点。
Show All Breakpoints	在输出窗口内显示所有断点的数据信息。
Clear All Breakpoints	清除所有断点。

2.3.6 工具菜单

Connect	设置打印端口地址以连接ICE(默认为378H)。

Check ICE Memory	检测ICE内有效的存储器。
Get Checksum from Project	获取校验和。
Get Code=1FFF size	获取程序ROM大小以及空的ROM的大小。
Compute Execution Time	计算两个断点间的运行时间。

2.3.7 选项菜单

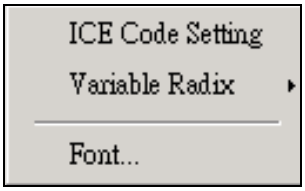


Fig. 2-17 选项菜单

ICE Code Setting 设置微控制器的代码选项

Variable Radix 选择10进制或者16进制

Font… 设置编辑窗口的文本字体(其他窗口的字体是固定的)

Environment Setting 设置WicePlus环境，例如，是否激活list文档，是否激活map文档以及设置可同时开启编辑窗口的数目。

Debug Option Setting 设定调试功能的参数。

View Setting 可视界面的设定，例如，编辑器显示或不显示行数。

2.3.8 IDE 菜单

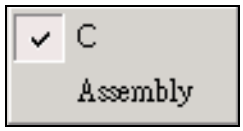


*Fig. 2-18 **IDE 菜单***

C 选择C语言作为编辑语言

Assembly 选择汇编语言作为编辑语言

2.3.9 窗口菜单

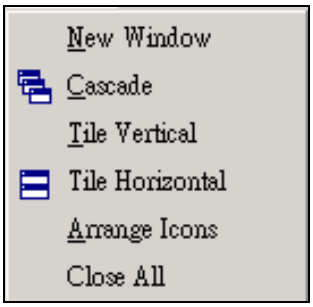


Fig. 2-19 窗口菜单

New Window 打开新的编辑窗口

Cascade 重叠排列所有激活文件的编辑窗口。

Tile Vertical 垂直排列所有打开的编辑窗口

Tile Horizontal 水平排列所有打开的编辑窗口

Arrange Icons 在编辑窗口的底部同一行内排列所有打开的文件名（最小化为若干文件图标）。

> **注意**
> *该命令仅适用于WicePlus窗口最小化的情况下。*

Close All 关闭所有文件

2.3.10 帮助窗口

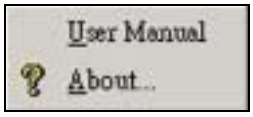


Fig. 2-20a 帮助窗口

User's Manual 打开WICEPLUS的用户手册。

About… 显示WICEPLUS的当前版本，以及包含"READ ME"文件的其他信息，其中"READ ME"包含了WICEPLUS的最近的变动

*Fig. 2-20b **About** 对话框*

2.4 工具栏

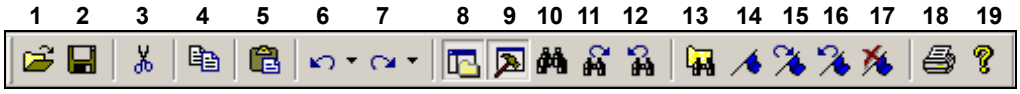


Fig. 2-21a WicePlus 主窗口的(标准)工具栏

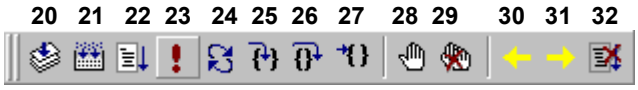


Fig. 2-21b WicePlus 主窗口的(编译)工具栏

2.4.1 工具栏图标和功能

括号内是相应的快捷键：

1  **打开**：打开已有的文件(Ctrl + O)

2  **存储**：存储当前文档(Ctrl + S)

3  **剪切**：移动所选的字符到剪贴板(Shift + Del)

4  **拷贝**：拷贝字符到剪贴板 (Ctrl + C)

5  **粘贴**：从剪贴板粘贴字符 (Ctrl + V)

6  **撤消**：取消上次的编辑行为 (Alt + Backspace)

7  **重做**：取消最近的撤消动作 (恢复撤消编辑行为)

8  显示/隐藏工程窗口

9  打开/隐藏输出窗口

10  **查找**：在整个激活的文件内查找字符 (Ctrl + F)

11  **向下找**：从光标位置到文档最后之内查找字符

12  **向上找**：从光标位置到文档最前面之内查找字符

13  **从文件内查找**：从没有激活的文件内查找字符

14  **标识**：在光标的位置设置一个标识

15  **向下跳至标识处**：跳到从光标处到文本最后之内的下一个标识处

16  **向上跳至标识处**：跳到从光标处到文档最前面之内的下一个标示处

17  **清除标识**：清除所有标识

18  **打印**：打印激活的文件 (Ctrl + P)

19  **关于**：关于编译器版本和其他信息

20  **编译**：编译编辑窗口内激活的文件(Alt + F7)

21  **重建所有**：编译工程内的所有文件(Alt + F9)

22 **自由运行：**自动下载并忽略断点执行程序 (F10)

23 **运行：**自动下载并运行程序，但遇断点停止(F5)

24 **运行到光标处：**自动下载并运行程序，在光标处停止，并且忽略断点 (F4)

25 **不进入子程序单步运行：**自动下载并单步运行程序，而不进入子程序(F8)

26 **进入子程序单步运行：**自动下载并单步运行，遇CALL指令进入子程序(F7)

27 **复位：**复位ICE (F6)

28 **插入/清除断点：**插入/清除断点标识

29 **清除所有断点**

30 停止：停止运行

2.5 文档栏

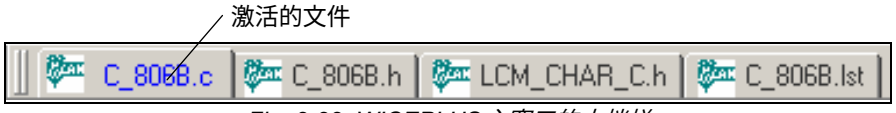


Fig. 2-22 WICEPLUS主窗口的文档栏

文档栏显示在编辑窗口内打开的每个文件的图标，点击你希望激活的相关文件的图标(放在正在编辑的编辑窗口的前面)，高亮的文件名是激活的文件(功能类似窗口下面的任务栏按钮)。

2.6 状态栏

当你的工程正在进行编译时，WICCEPLUS运行进度将会显示在工具栏内。

在文本编辑窗口内光标位置处指示当前光标行列数。

R/W标志表示当前文件的读写状态，如果为只读，将会显示读，否则显示为空。

键盘模式显示的是下列按键的状态：

- **插入键** – 当插入模式关时，不显示 OVR，否则显示。
- **字母锁定键** – 当大写字母关时，不显示 CAP，否则显示。
- **数字锁定键** – 当数字键盘模式关时，不显示 NUM，否则显示。
- **Scroll Lock** key – 当滚动键模式关时，不显示 SCRL，否则显示。

第三章
开始

3.1 硬体上电

E8-ICE正确连接到目标板、PC和电源，打开ICE开关，并看到红色电源指示灯亮起。如果目标板使用的是ICE电源，那么黄色灯也将亮起。

当ICE和目标板检测证实功能正常，那么即可开始使用软件开发环境了。

3.2 开始使用WicePlus应用程序

从桌面或者WINDOWS的开始菜单内点击WICEPLUS图标来打开WICEPLUS软件。当从开始菜单启动时，点击程序，然后选择WicePlus组并点击WicePlus图标。

3.2.1 连接对话框

一旦程序启动，程序主窗口最初会显示“CONNECT”对话框，以提示你正确选择现有的微控制器和打印接口(默认值378H)。

你也可以使能“CHECK ICE MEMORY”来检查ICE存储器的状况。“I/O wait time”描述地是I/O响应的速度。增大它的值表示减慢速度，减小表示加快速度。

设置完成之后，点击OK按钮。

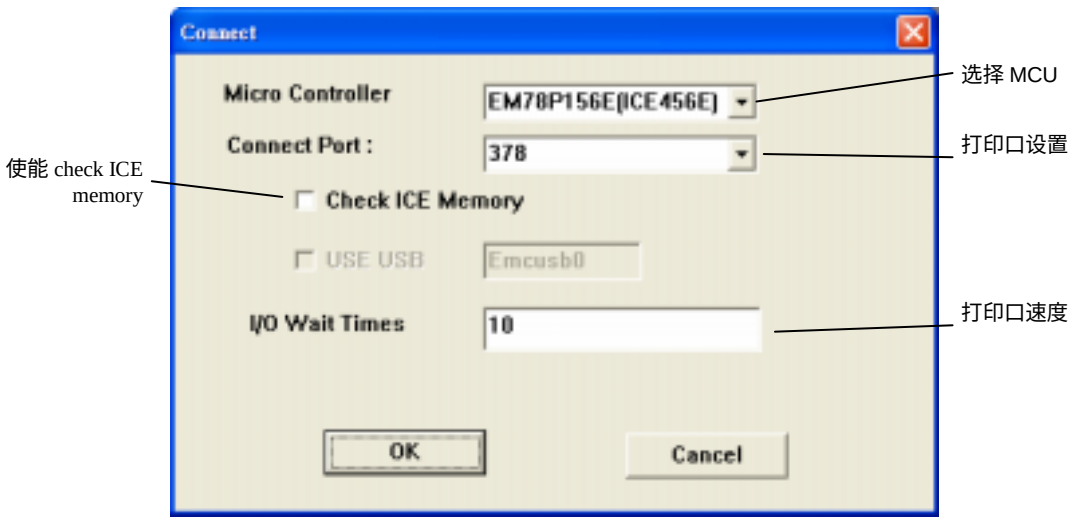


Fig. 3-1 WicePlus 连接对话框

3.2.1 代码选项对话框

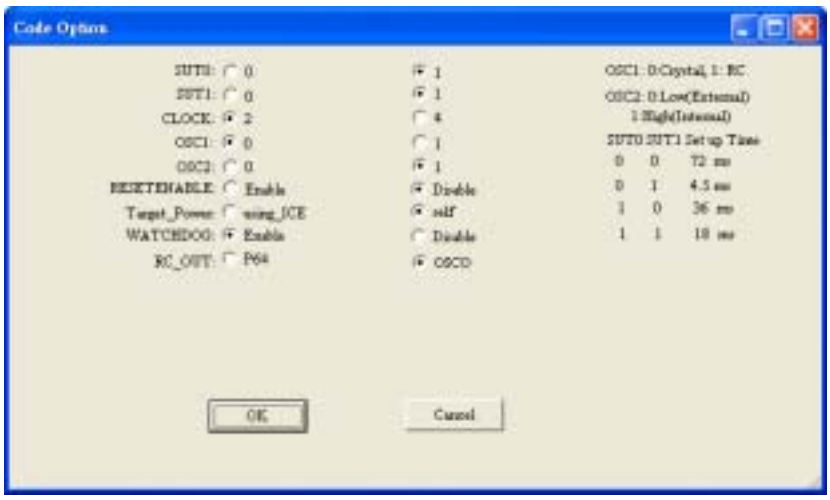


Fig. 3-2 WicePlus 代码选项对话框

接下来显示代码选项对话框。根据所需的要求选择适合ICE实际状态的所有选项，然后点击OK按钮。

3.3 创建新的工程

需要根据以下步骤来创建新的工程：

1. 点击菜单栏的FILE菜单或PROJECT菜单并从下拉菜单选择NEW命令。
2. 如果从FILE菜单选择了NEW命令，那么NEW对话框(显示如下)将会显示，否则，如果NEW命令是从Project菜单选择的话，NEW Project对话框(Fig. 3-5)将会显示。

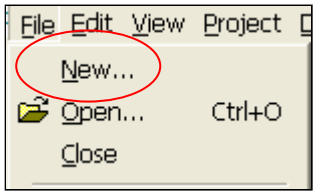


Fig. 3-3 文件菜单

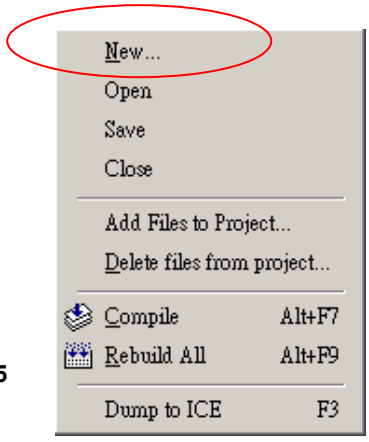


Fig. 3-4 工程菜单

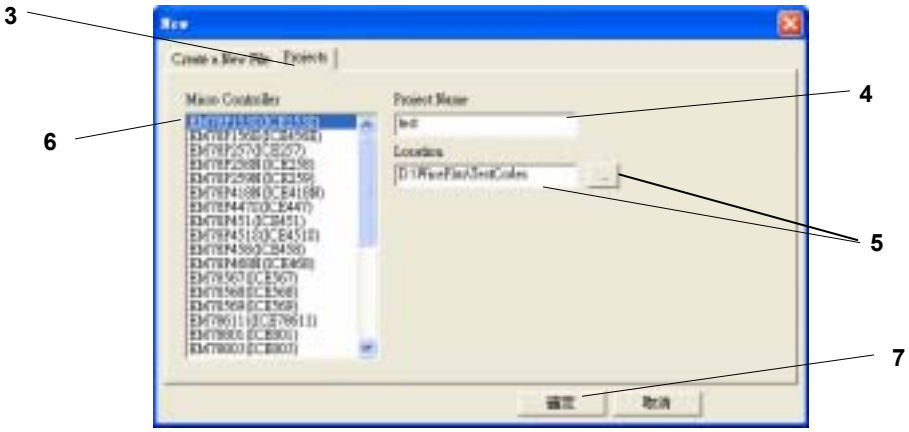


Fig. 3-5 “NEW”对话框显示创建新工程的工程界面
(从 File 菜单打开)

3. 从NEW对话框选择PROJECTS表。
4. 在PROJECT NAME内为新的工程分配一个文件名(后缀.prj 将会自动生成)。
5. 找出你想要存放文件的文件夹，可以使用浏览图标查找合适的文件夹。
6. 在微控制器列表为新的工程选择合适的微控制器。
7. 确认所有的选择和输入后点击OK按钮。

新工程创建完成了，并且工程文件名与所选择的微控制器即显示在PROJECT窗口的顶部。

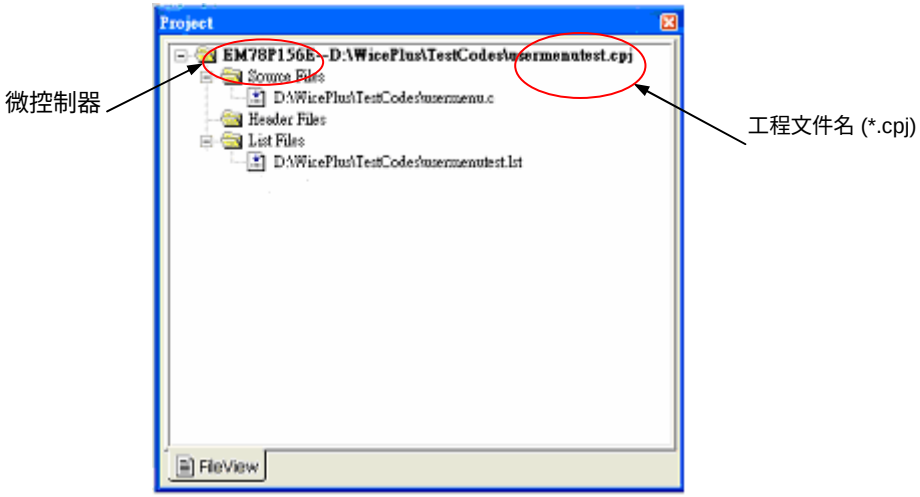


Fig. 3-6 "Project"窗口

3.4 将源文档加到工程内或从工程内删除源文档

要么往新的工程或已有的工程内增加已有的源文档，要么用WICEPLUS文本编辑器创建一个新的源文档到工程内。

3.4.1 给工程创建和增加一个新源文档

如果您的源文档还没有创建的话，那么可以优先使用NEW对话框的方法(通过从FILE菜单点击NEW命令的方式)去创建新的源文档并用WICEPLUS文本编辑器去创建它的内容。

1. 点击NEW对话框内的FILE格，选择你想要从EMC源文件列表内创建的源文件的类型，如用于编译的*.C(默认)文件，*.H头文件。

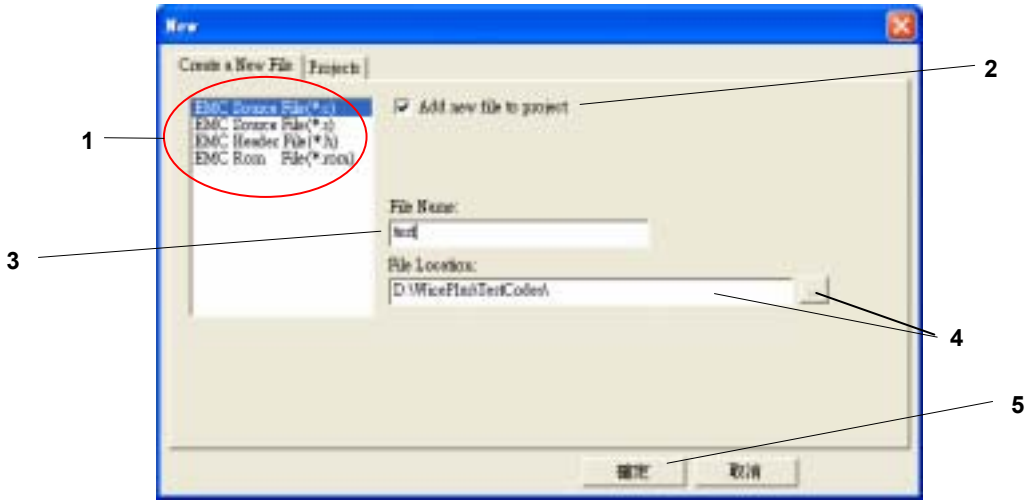


Fig. 3-7 "New" 对话框显示创建新的源文件的工程项目

2. 如果想自动往工程里增加新的文件，检查**Add to Project** 检测框(默认)，否则清除检测框。
3. 在**File Name** 框内给新的源文档分配一个文件名。
4. 选择存放新源文件的文件夹，可以使用**Browse**图标去查找合适的文件夹。
5. 确认您的输入后点击OK按钮，您即可以在编辑窗口内开始写程序了。

3.4.2 往新的工程内添加已有的源文档

如果已经准备好了源文件，您可以立刻将其增加到新工程内。

1. 点击菜单栏的PROJECT菜单，从下拉菜单中选择**Add Files to Project**命令，则OPEN对话框显示如下。

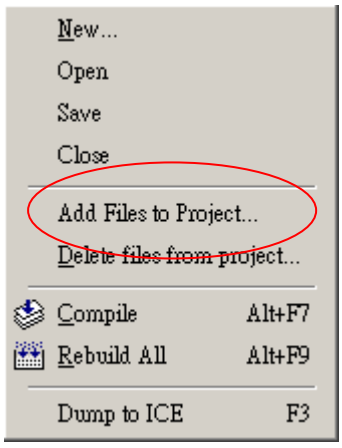


Fig. 3-8a "Add Files to Project" 命令

Fig. 3-8b "Open" 对话框

2. 浏览并选择文件(或者多个文件)加入新工程内。
3. 确认后点击OK按钮。

3.4.3 从工程内删除源文档

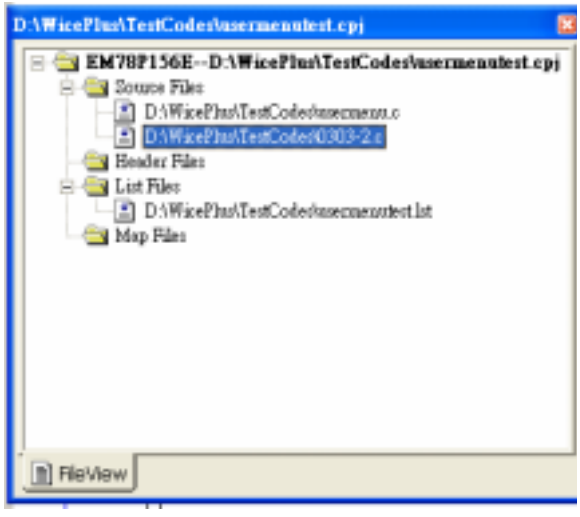


Fig. 3-9a直接从 "Project" 窗口删除工程文件

1. 从PROJECT窗口选择要删除的文件，按键盘上的DELETE键即可删除文件。

2. 也可以从PROJECT下拉菜单中点击**Delete Files from Project...** 命令来删除工程内的文件。

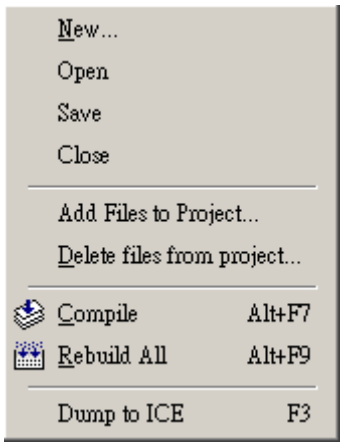


Fig. 3-9b从工程菜单删除工程文件

3.5 从文件夹或工程内编辑源文档

3.5.1 从文件夹内打开源文档

您也可以在增加新的工程之前在编辑窗口内打开已有的源文件，可以按如下方法执行：

1. 点击菜单栏的FILE或PROJECT菜单，从下拉菜单中选择Open指令；
2. 从打开对话框(上文Fig. 3.8b)点击源文件，文件就会自动在编辑窗口内打开。

编辑已经增加到工程内的源文档，可参考以下部分。

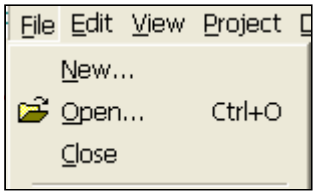


Fig. 3-10 从文件菜单里打开&编辑源文档

3.5.2工程文件内打开源文档

您可以编辑工程内的源文档。从工程窗口双击源文档，文件即会在编辑窗口打开。

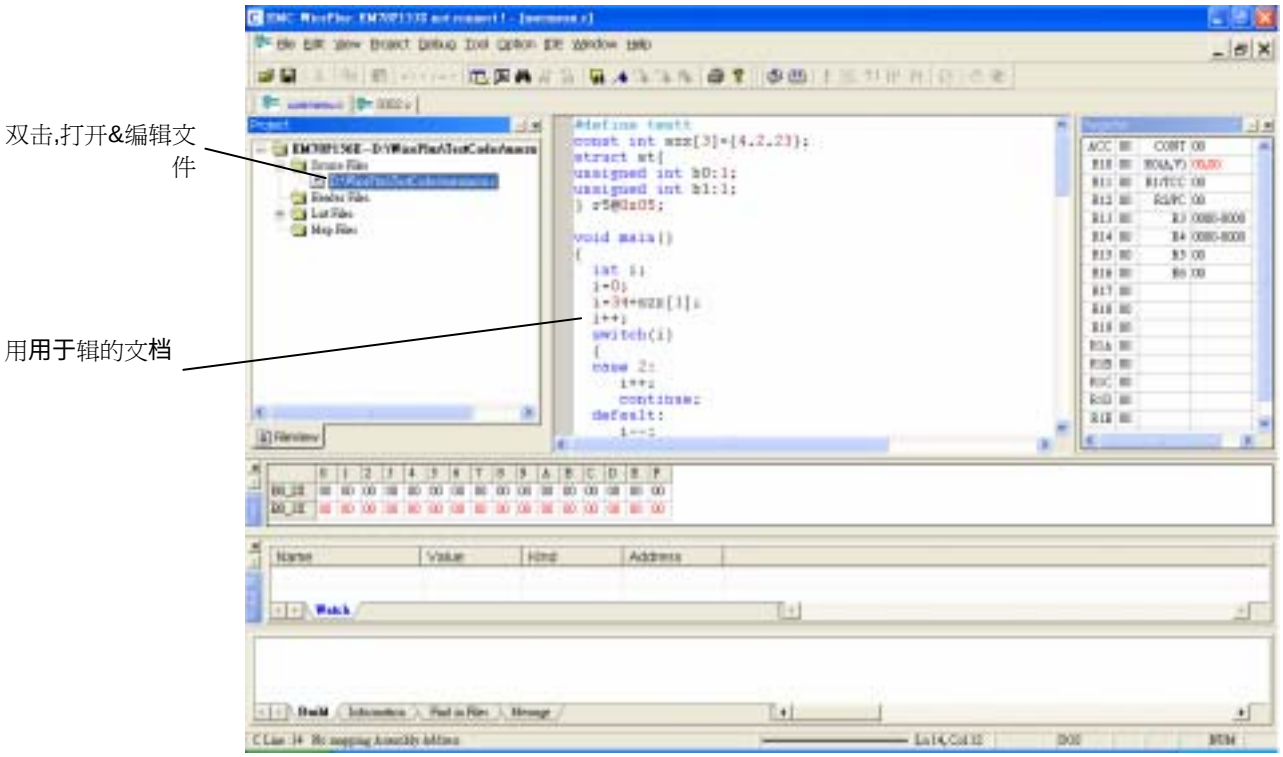


Fig. 3-11 直接从工程窗口编辑文件

3.6 编译工程

源文件嵌入后，您就可以用PROJECT菜单里的命令编译您的工程了。

- 点击 **Compile** 命令仅仅编译激活的文件(产生*.asm)。
- 点击 **Rebuild All** 命令编辑工程内的所有文件。

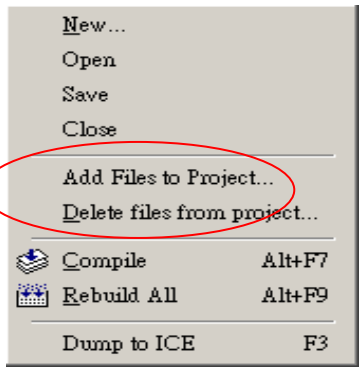


Fig. 3-12 Assemble & Link Commands

Rebuild All将会产生目标文件(*.bbj)、列表文件(*.list)、CDS文件(*.cds)。

被编译的文件会自动存储到存放源文件的文件夹内，编译状态可在如下的输出窗口内监视到。

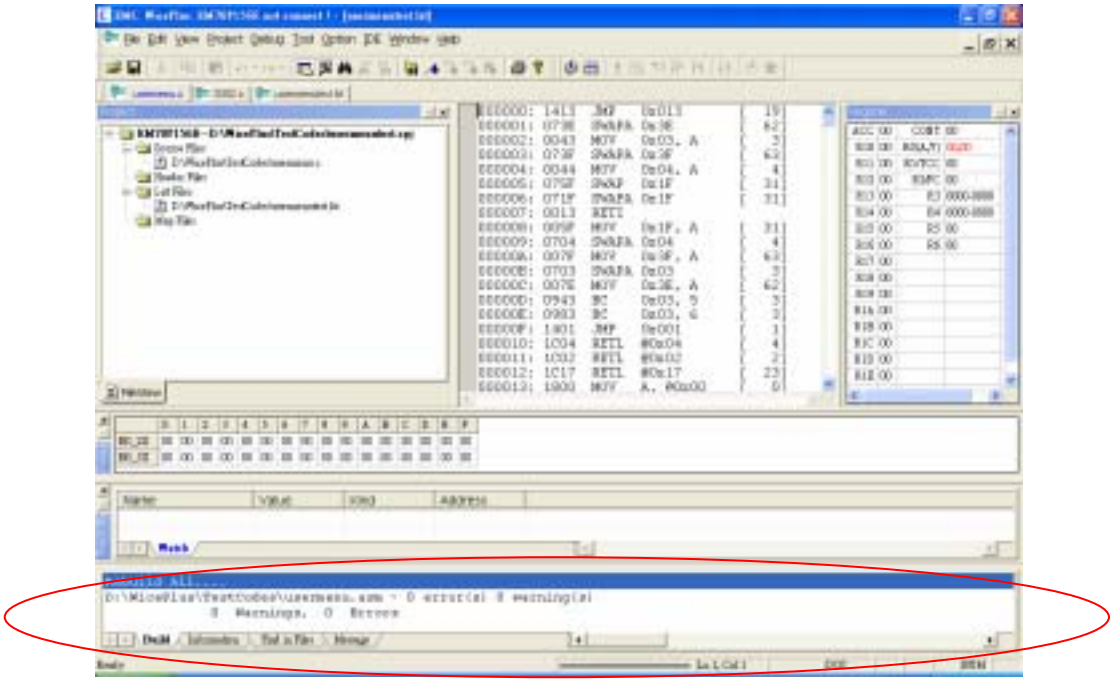


Fig. 3-13 输出窗口显示编辑成功

在编译过程中如果发现了错误，相关的错误信息会显示在**Output**窗口。双击错误信息即可连接到编辑窗口源文档对应的错误的位置，如果相应的源文档没打开，它将自动打开。

双击连接到相应的错误源

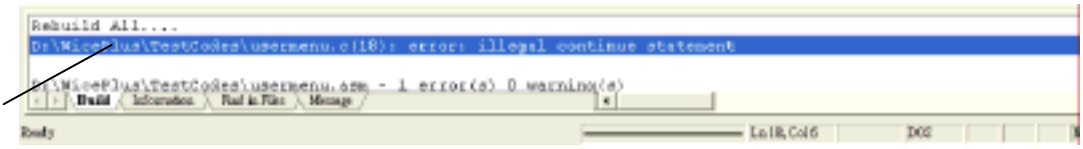


Fig. 3-14 OUTPUT窗口显示编辑错误

修改源文件纠正错误，重新编译与连接操作。

3.7 下载编译好的程序到ICE

修改完错误并编译通过后，使用PROJECT下拉菜单里的DUMP TO ICE指令下载到ICE，或者用快捷键(F3)，如下图所示。

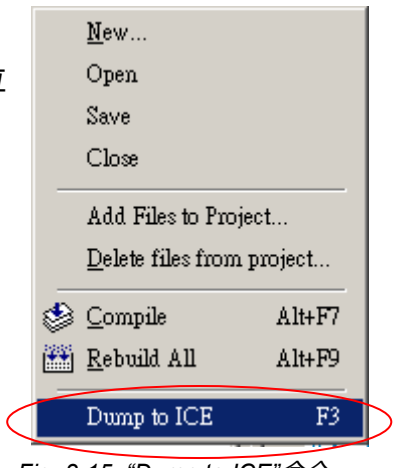


Fig. 3-15 "Dump to ICE"命令

3.8 调试工程

成功下载后，可以准备调试程序了，确定您的ICE正确连接到您的计算机上。

所有的调试命令在DEBUG菜单内(右图是下拉菜单内相应的快捷键)。许多在WICEPLUS工具栏内使用频繁的调试图标，也是有效的。

Fig. 3-16b 调试命令下拉菜单

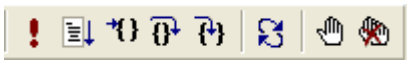


Fig. 3-16a 工具栏内调试图标

Toggle Breakpoint – 设置光标处的断点和移除断点。

Clear All Breakpoints – 清除所有断点。

F5

Go – 从当前指针开始运行程序，遇断点停止。

F6

Reset – 硬件复位 (寄存器内容显示初值)，ICE处于初始状态。

F7

Step Into – 单步运行，遇CALL子程序进入子程序(寄存器内容同时改变)。

F8

Step Over – 单步运行，遇CALL子程序不进入(寄存器内容同时改变)。

Free Run – 从当前指针运行，直到按"STOP"按钮停止运行，程序运行时，所有断点将被忽略。

F10

Stop – 停止运行

在调试期间，PC指针、寄存器、RAM的内容实时显示和可读，以便程序停止后可以提供有用的信息。

3.8.1 断点设置

选择要设置断点的地方，然后双击，观察棕色点亮部分。

您也可以使用工具栏内的**Insert/ Remove Breakpoint**图标(手的形状)来设置断点。

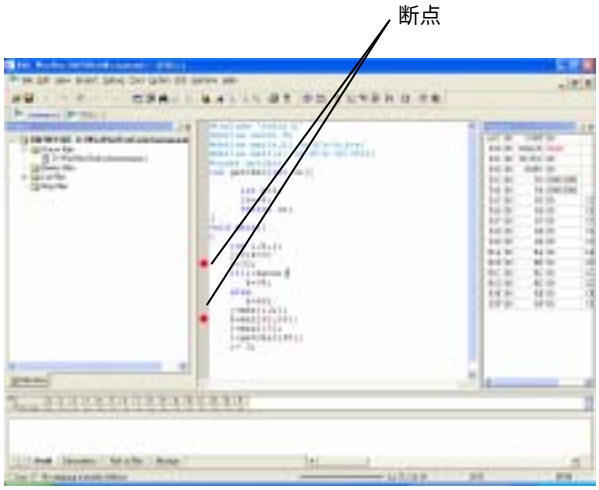


Fig. 3-17 设置断点的源文档

同样，如果再在刚才的位置双击，设置的断点就被清除或者将光标置于此处再次点击手形图标也可清除断点。清除所有的断点，点击DEBUG菜单内的**Clear All Breakpoints**命令。

第四章
C 基础理论

4.1 注释

单行注释：

//	在双斜线后面所有的数据为注释数据，会被忽略。

多行注释：

/ ... **/*	在如左边所示内的所有语句为注释语句，会被忽略。

注释用来帮助您容易理解程序，可放在源程序的所有位置。编译器会从源代码忽略掉注释部分，因此不会占用程序空间。

例如：

```
// 这是单行数据

/*
这是第一行注释
这是第二行注释 */
```

4.2 保留字

WICEPLUS C编译器的保留字是由ANSI C保留字和EM78系列特有的保留字组成。下表是此编译器的保留字。

ANSI C Conformity Words					
const	Default	goto	switch	typedef	sizeof
break	Do	if	short	union	extern
case	Else	int	signed	unsigned	
char	Enum	long	static	void	
continue	For	return	struct	while	

EM78 Series Unique Words				
Indir	ind	page	on	off
Io	iopage	_intcall	rpage	
low_int	_asm	bit	bank	

注意

- *EM78系列不支持**Double**和**float** 。*
- *_asm增加到EM78系列的C编译器*
- *indir, ind, io, iopage, rpage为MCU硬件定义和宣告。*

4.3 预处理命令

预处理命令通常都是以“#”符号开始。这些命令都是可以被预处理器识别，以便正确的编译源代码。

4.3.1 #include

#include “file_name”:	预处理器会寻找文件的工作目录。
#include <file_name>:	预处理器首先会搜寻工作目录去寻找文件。如果在工作目录找不到文件，它会从EMC_INCLUDE指定的目录中搜索。

#include 让预处理器增加头文件的内容到源程序。

范例1：

```
#include <EM78.h>
#include “project.h”
```

范例2:

```
unsigned int uaa;    //在headfile.h文档里
…
main ()              //在testcode.c文档里
{
   uaa=0x21;
…
}

extern unsigned int uaa;  //在kkdr.c文档里
void ()
{
   uaa=0x38;
….
   Uaa=0x43;
}

void ()
{
   uaa=0x29;
}
```

4.3.2 #define

#define identifier
#define identifier token_list
#define identifier (parameter_list) token_list

#define identifier( ) token_list

#define指令用于定义字符常数代替源代码，这使得源程序更具可读性。

注意

多行宏定义在行之间应该用"\"包含起来, 因此当在宏内使用汇编代码, 可以只用一行。

范例：

```
#define MAXVAUE 10
#define sqr2(x, y) x * x + y * y
```

4.3.3 #if, #else, #elif, #endif

```
#if constant_expression
#else
#elif constant_expression
#endif
```

#if指令用来作为条件判断，它将被#endif终止。#else可以用来提供多种选择，如果需要，程序可以用#elif作为多种编辑的选择，但是必须是有效的表达式。

例如：

```
#define RAM 30
#if (RAM < 10)
        #define MAXVALUE 0
#elif (RAM < 30)
        #define MAXVALUE 10
#else
        #define MAXVALUE 30
#endif
```

4.3.4 #ifdef, #ifndef

```
#ifdef identifier
#ifndef identifier
```

#ifdef指令用来作为符号定义的一种条件编译。当条件编译的代码没有定义为特征字符，可以用#ifndef指令。这两条指令都须以#endif指令终止，并且可以用#else指令来选择。

范例：

```
#define DEBUG   1
#ifdef (DEBUG)
        #define MAXVALUE 10
#else
        #define MAXVALUE 1000
#endif
```

4.4 常量

4.4.1 数字常量

十进制:	默认
十六进制:	前缀为 “0x”

数据常量可以以十进制和十六进制的形式定义，可通过前缀来确定，不支持二进制和八进制。

范例：

```
12, 34       // 十进制
0x5A, 0xB2   // 十六进制
```

4.4.2 字符常量

‘字符’

字符常量通过单引号表示，如下显示的ANSI C换码顺序都作为单字符的。

ANSI C Escape Sequence		
Escape Character	**Meaning**	**Hexadecimal**
\0	空	00
\a	报警	07
\b	回车	08
\f	换页	0C
\n	换行	0A
\r	带返回	0D
\t	水平制表符	09
\v	垂直制表符	0B
\\	斜线	5C
\?	问号	3F
\’	单引号	27
\”	双引号	22

范例：

```
‘a’, ‘b’,’c’, /x00
```

4.4.3 字符串常量

“character_list”

字符串常量是用双引号标识，并且在最后一个字符有一个暗含的空值。

注意
它会多占用一个以上字符常量的空间来存储空值。

范例：

```
“Hello World”
“Elan Micro”
```

4.5 数据类型

基本数据类型的数据的大小和范围 (最大和最小值) 显示如下：

类型	范围	存储大小（字节）
void	N/A	None
单字符	–128 ~ 127	1
无符号字符	0 ~ 255	1
整型	-128 ~ 127	1
无符号整型	0 ~ 255	1
短型数	–32768 ~ 32767	2
无符号短型数	0 ~ 65535	2
长型数	–2147483648 ~ 2147483647	4
无符号长型数	0 ~ 4294967295	4
位	0 ~ 1	1 (Bit)

NOTE
*1. 不支持 **Floating**与**Double**类型。*
2. 更详细的关于“Bit Data Type.”可参照第五章5.4 节。
3. 如果用户用长整形进行乘法、除法、取模与比较运算，bank 0的0x20~0x24(5字节)会被编译器使用，因此，当要进行那些运算时，请勿使用这些地址。

当算术运算， 例如“*”、“/” 和“%”用于不同的数据类型，在操作数有效之前，必须完成变量到数据类型的转换，建议用户使用相同的数据类型编写。

范例：

```
Int I1, I2;
Short S1, S2, S3;
Long L1, L2;
I1 = 0x11;
I2 = 0x22;
S1 = I1 * I2;  ➔  change to S1 = (short) I1 * (short) I2;
          // 如果在I1和I2之前忘记加”(short)”,那么最后的结果
          // (在S1)将会只有一个字节
S1 = 0x1111;
S2 = 0x02;
L1 = S1 / S2;  ➔  change to L1 = (long) S1 * (long) S2;
          // 如果在I1和I2之前忘记加”(long)”，那么最后的结果
          // (在L1)将会只有2个字节
```

4.6 枚举类型

enum identifier
enum idenftifier {enumeration-list [=int_value]...}
enum {enumeration-list}

枚举类型定义一系列的整型常数，定义后所有的整型常数会以一组有效的名字放在一起。而对每个常量，您可以定义一个互不相同的名字。

范例：

```
enum tagLedGroup {LedOff, LedOn} LEDStatus;
```

4.7 结构和联合类型

结构(联合)类型名:
结构体(联合体)成员名
结构体(联合体)成员名{成员列表}
结构(联合)成员列表

成员名列表:
成员名
成员名列表 成员名

成员名:
成员名特定列表

所有特定成员名:
特定成员名
所有特定成员名 特定成员名

在结构类型内的每个数据和相关的数据可以赋予一个相同的名字，联合类型内的变量则是分享相同的存储空间。

注意

- *不要在结构和联合内使用位数据类型，用位元栏代替。*
- *结构和联合类型不能用做函数参数*

范例1:

```
struct st
{
   unsigned int b0:1;
   unsigned int b1:1;
   unsigned int b2:1;
   unsigned int b3:1;
   unsigned int b4:1;
   unsigned int b5:1;
   unsigned int b6:1;
   unsigned int b7:1;
};
struct st R5@0x05 ;  //结构类型R5与0x05相对应
```

范例2:

```
struct tagSpeechInfo{
   short rate;
   long size;
} SpeechInfo;

union tagTest{
   char Test[2];
   long RWport;
} Test;
```

4.8 数组

类型申明:
数组申明:
数组名
[常量表达式]
数组名 [常数表达式]

数组是相同类型的数据的集合，并且可以用相同的名字命名。

注意

- *如果"const"用来申明一组数组，那么数据将会放在ROM内。*
- *数组的最大位是32个字节(RAM 区)。*

范例：

```
int array1 [3] [10]
char port [4]
const int myarr [2] = {0x11, 0x22};
            // 0x11, 0x22 将会放在ROM内
```

4.9 指针

Declarator
type-qualifier-list * declarator

指针是其它数据或NULL常量的地址的索引。所有的指针类型占用1个字节。

注意

不支持指针函数。

范例：

```
int *pt;
```

4.10 操作运算

4.10.1 支持操作运算的类型

C表达式支持的操作运算表达式如下：

- 算术运算
- 增量减量运算
- 赋值运算
- 逻辑运算
- 位运算
- 等式和关系运算
- 混合运算

下表是每种运算的详细描述：

算术运算		
符号	功能	表达式
+	加	expr1 + expr2
–	减	expr1 – expr2
*	乘	expr1 * expr2
/	除	expr1 / expr2
%	取模	expr1 % expr2

增减运算		
符号	功能	表达式
++	加 1	expr ++
--	减 1	expr --

赋值运算		
符号	功能	表达式
=	等于	expr1 = expr2

位运算		
符号	功能	表达式
&	与	expr1 & epxr2
|	或	expr1 | expr2
~	非	~expr
>>	右移	expr1 >> expr2
<<	左移	expr1 << expr2
^	位异或	expr1^expr2

等式、关系和逻辑运算			
符号	功能	表达式	示例
<	小于	expr < expr	x < y
<=	小于等于	expr <= expr	x <= y
>	大于	expr > expr	x > y
>=	大于等于	expr >= expr	x >= y
==	等式	expr == expr	x == y
!=	不等	expr != expr	x != y
&&	逻辑与	expr && expr	x && y
||	逻辑或	expr || expr	x || y
!	逻辑非	!expr	!x

混合运算		
符号	功能	示例
+=	y=y + x	x += y
-=	y = y - x	x -= y
<<=	y = y << x	y<<=x
>>=	y= y >> x	y>>=x
&=	y= y & x	y&=x
^=	y= y ^ x	y^=x
|=	y= y | x	y |= x

4.10.2 运算符的优先级

优先级	同级别的操作数，从左到右
最高	( ) [ ] - >
↑	! ~ ++ -- -(unary) +(unary) (type_cast) *(indirection) & (address) sizeof
↑	* / %
↑	+ -
↑	<< >>
↑	< <= > >=
↑	== !=
↑	&
↑	^
↑	|
↑	&&
↑	||
↑	?:
↑	= += -= *= /= %= >>= <<= &= |= ^=
最低	,

4.11 If-else 语句

```
if (表达式)  语句1
else  语句2
```

当假设的条件成立，执行语句1。如果假设的条件不成立，那么执行语句2 。

范例：

```
if (flag == 1)
{
      timeout=1;
      flag=0;
}
else timeout=0;
```

4.12 Switch 语句

```
Switch (表达式)
{
        case 常数表达式1:语句1
        case 常数表达式2:语句2
        default: 语句3
}
```

当表达式与常数表达式的值一致时，执行常数表达式后面的语句。

注意

表达式将被设为INT类型，因此在SWITCH结构内只能有256种情形。

范例：

```
switch (I)
{
        case 0: function0(); break;
        case 1: function1(); break;
        case 2: function2(); break;
        default: funerror();
}
```

4.13 While 语句

while (表达式) 语句

“While”语句将会先检测表达式，如果表达式为真值，将会执行后面的语句。

范例：

```
while (value != 0)
{
      value--;
      count++;
}
```

4.14 Do-while 语句

do
{
语句
} while (表达式);

“Do-while”将会先执行语句然后检查表达式，如果表达式为真值，那么继续执行语句，直到表达式的值为FALSE。

范例：

```
do {
         value --;
         count++;
} while (value != 0);
```

4.15 For 语句

for (表达式1; 表达式2; 表达式3) 语句;

“For”语句相当于下面的语句：

```
表达式1;
while (表达式2)
{
        语句;
        表达式3;
}
```

先执行表达式1，正常情况下表达式1为初值，“While”语句以相同的方式执行。

范例：

```
for (i = 0; i < 10; i++)
{
         value = value + i;
}
```

4.16 Break and Continue 语句

break;
continue;

“break”适用于switch及循环语句，遇“break”会跳出，而“continue”语句则会跳过循环的剩余部分并且直接跳到下次循环，“continue”适用于循环语句，但不能在swicth语句内使用。

范例：

```
break exampl see switch.

for (i = 0; i < 10; i++)
{
         flag = indata(port);
         if (flag == 0) continue;
         outdata(port);
}
```

4.17 Goto 语句

```
goto label;
    …
label: …
```

“goto”语句是用来跳到任意地址的功能。 它在深循环内是很有用的。

范例：

```
for (i = 0; i < 10; i++)
        for (j = 0; j < 100; j++)
                for (k = 0; k < 100; k++)
                {
                flag = crccheck(buffer);
                if (flag != 0) goto error;
                outbuf(buffer);
                }
error:
        //clear up buffer;
```

4.18 函数

函数是C语言的基本模块，它包含了库函数和定义函数。

4.18.1 库函数

```
<返回值类型> <函数名> (<参数列表t>);
```

函数必须在被调用之前申明，它包含返回值、函数名和参数类型。

注意

- *所有传递给函数的参数应该是一个固定的数，编译器不支持不确定的参数。*
- *编译器不支持递归函数。*
- *不要使用”struct”或者“union”作为函数的参数。*
- *不支持函数指针。*
- *为了减少RAM区的使用，建议用户在函数里使用全局变量代替局部变量。*

范例 (库函数)：

```
unsigned char sum(unsigned char a,unsigned char b);
…
```

4.18.2 定义函数

```
<返回类型> <函数名> (<参数列表>)
{
    语句
}
```

范例 (函数内容)：

```
unsigned char sum(unsigned char a,unsigned char b)
{
        return (a+b);
}
```

第五章
硬件相关编程

5.1 寄存器页 (rpage)

<变量名> @<地址>[: rpage <寄存器页数 >];

数据类型用来申明某一寄存器页的一个变量，用户不得不申明寄存器的页，包括rpage 0。

注意

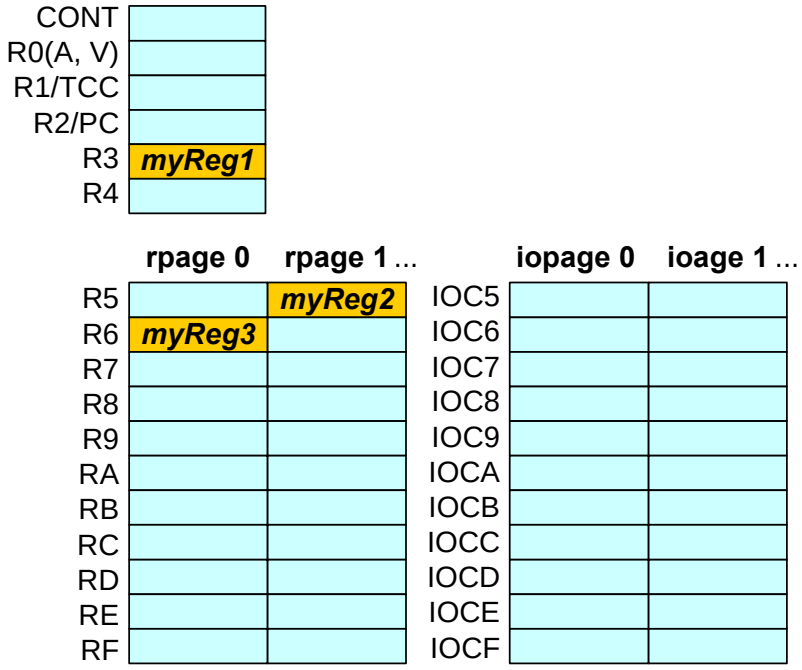
- *如果变量申明了"rpage"，那么不能同时申明为"bank"、"iopage"或者"indir"。*
- *只有全局变量才能定义为"rpage"的数据类型。*
- *虽然MCU只有rpage0，但<寄存器页数 >必须分配。*

范例：

```
unsigned int myReg1 @0x03: rpage 0;
        // myReg1在寄存器page 0的0x03地址
        // 虽然特殊寄存器只有一个寄存器页，
        // 但寄存器页数是不能被忽视的。

unsigned int myReg2 @0x05: rpage 1;
        // myTest2在寄存器page 1的0X05地址。
        // 如果特殊寄存器不止一页，那么用户需要指出寄存器所在的页

struct st
{
    unsigned int b0:1;
    unsigned int b1:1;
    unsigned int b2:1;
    unsigned int b3:1;
    unsigned int b4:1;
    unsigned int b5:1;
    unsigned int b6:1;
    unsigned int b7:1;
};
struct st myReg3@0x06: rpage 0;
```



5.2 I/O 控制寄存器页 (iopage)

io <变量名 > [@<地址>[: iopage <io控制页数r>]];

申明变量所在的寄存器页，用户不得不申明io变量的iopage，哪怕只有一个io控制页。

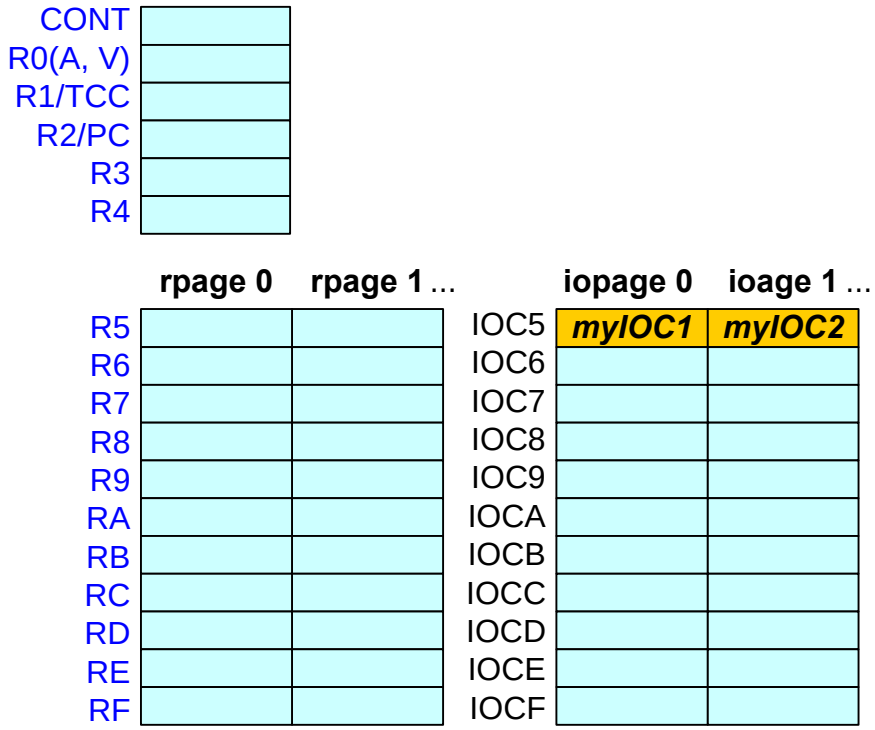
注意

- *如果变量被申明为"iopage"了，那么它不能同时申明为"bank"、"rpage"或者"IND"。*
- *只有全局变量才可以定义为"iopage"。*
- *虽然MCU只有iopag0，但<io控制页数>必须分配。*

范例：

```
io unsigned int myIOC1 @0x05: iopage 0;
        // myIOC1在io控制寄存器PAGE0的0x05地址

io unsigned int myIOC2 @0x05: iopage 1;
        // myIOC2在控制寄存器PAGE1的0x05地址
```



5.3 Ram 区

<变量名> [@<地址>[:bank <bank 数值>]];

定义RAM BANK的变量，<bank 数>需给出，包括变量定义在Bank 0。

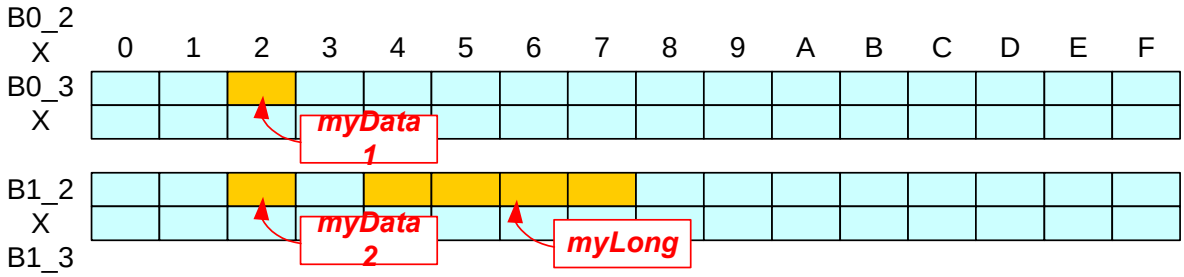
注意

- *如果变量已被申明为"rpage"了，那么它不能同时申明为"bank"、 "iopage"或者"indir"。*
- *只有全局变量才可以定义为"bank"数据类型。*

范例：

```
unsigned int myData1 @0x22: bank 0;
        // myData1是bank 0的变量
unsigned int myData2 @0x22: bank 1;
        // myData2是bank 1的变量
unsigned short myshort @0x20: bank 2;
        // myshort是bank 2的0x20与0x21地址的变量
unsigned long myLong @0x24: bank 1;
        // myLong是bank 1的0x24~0x27地址的变量
```

RAM Bank:



5.4 位数据类型

位 <变量名 > [@<地址> [@系列位] [: bank <bank 号> / rpage <寄存器页数 >]];

位数据类型只占用1个位。

注意

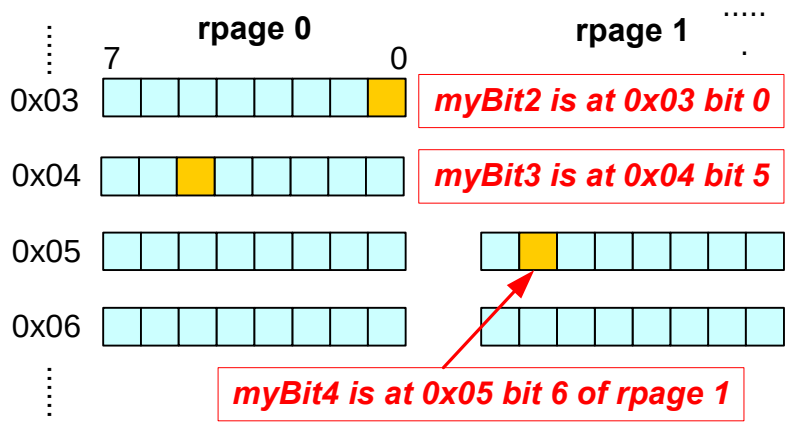
- *位数据类型不能用于结构和联合内，推荐用于位元栏，例如：*

```
union mybit {
        unsigned int b0:1
        unsigned int b1:1
        unsigned int b2:1
        unsigned int b3:1
        unsigned int b4:1
        unsigned int b5:1
        unsigned int b6:1
        unsigned int b7:1
};
```

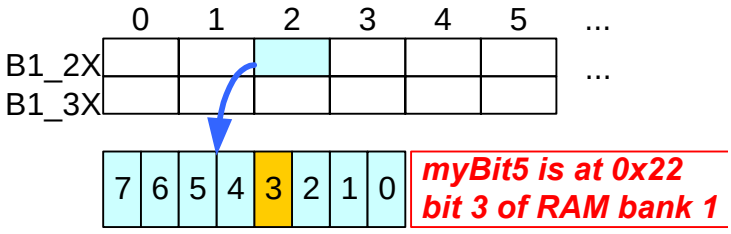
- *位数据类型不能用于函数参数。*
- *位数据类型不能与其他的数据类型用于算术运算。*
- *在IO控制寄存器r内不支持位数据类型。*
- *位是保留字，因此不要把它命名为"结构"或者"联合"。*
- *只有全局变量才可以定义为位数据类型。*

范例：

```
bit myBit1;                          // myBit1的位址由编译器分配
bit myBit2 @0x03 :rpage 0;
                                     // 如果没有定义位,那么默认位第0位
                                     // 因此myBit2在0X03的第0位
bit myBit3 @0x04 @5: rpage 1; // myBit3在rpage 1的0X04的第5位
bit myBit4 @0x05 @6: rpage 1; // myBit4在rpage 1的0x05的第6位
bit myBit5 @0x22 @3: bank 1;  // myBit5在bank 1的0x22的第3位
```



RAM Bank:



5.5 数据/LCD RAM 间接寻址

Indir <变量名> [@<地址>[: ind <ind数字>]];

定义间接寻址的变量，若要分配地址，<ind 数字>需给出。

如果MCU有数据RAM，使用“ind 0”(间接RAM 0)。

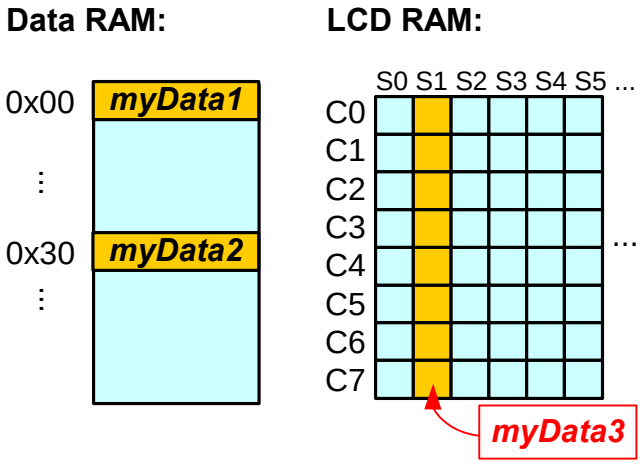
如果MCU有LCD RAM，使用“ind 1”(间接RAM 1)。

注意

- *如果指定的MCU不支持IND区，那么编译器将产生错误信息，例如：“符号 ‘WriteIND’ 没有定义”。*
- *只有全局变量才能定义为”indir”类型。*
- *indir类型不支持数组或者是指针变量。*

范例：

```
indir int nData1;
                  //默认值为“ind 0”，因此nData1在数据Ram区
indir int nData2 @0x30: ind 0;
                  //因为使用了”ind 0”，nData2 位于数据RAM区
indir int nData3 @0x01: ind 1;
                  //因为使用了”ind 1”，nData3位于LCD Ram区
```



5.6 在ROM内定位C函数

```
<返回值> <函数名>(<参数列表>) @<地址> [: page <页数>]
{
      ……
}
```

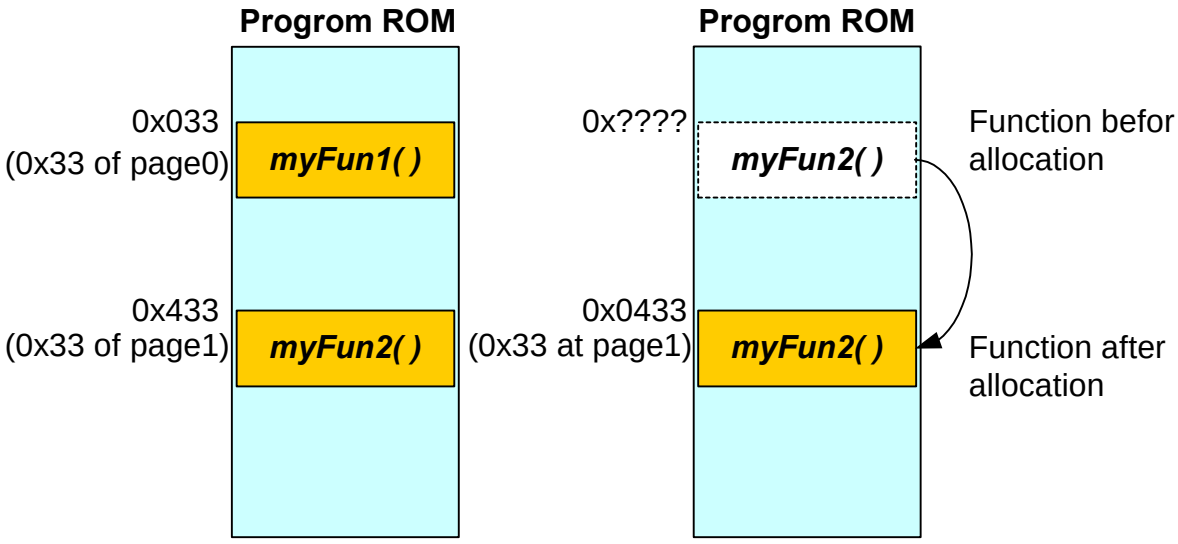
您可以将函数置于ROM特定的地址，并且可以使用“page”指令分配您希望分配的ROM区。

注意

- *只有函数才可以分配页。*
- *不要在ROM内给中断保护程序或中断服务程序分配页。*

范例：

```
void myFun1(int x, int y) @0x33
          // myFun1()放在page 0的0X33处(默认的page)
{
          ……
}
void myFun2(int x, int y) @0x33: page 1
          //myFun2()放在page 1的0X33处
{
          ……
}
```



5.7 放数据到ROM

const <变量名 >;

在程序执行的时候一些数据不能被改变，因此，您需要把这样的数据存放在ROM内以便利用有限的RAM空间。编译器用TBL指令将这种数据放于ROM内。

注意

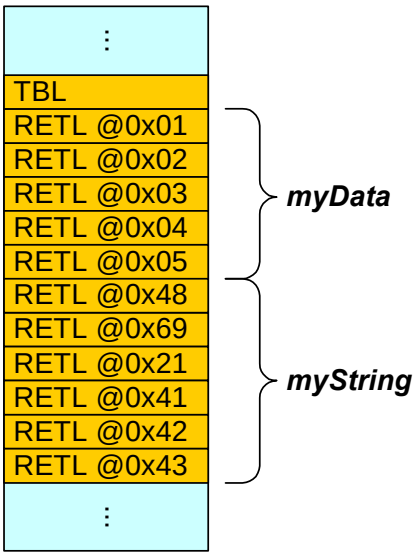
- *使用constant数据类型存放数据到ROM内。*
- *只有全局变量才能定义为“const”数据类型。*
- *常量数组的最大有效值为255字节。*

范例：

```
const int myData[] = {1, 2, 3, 4, 5};

const char myString[2][3] = {
         “Hi!”,
         “ABC”
};
```

Program ROM:



注意

如果指定的MCU不支持TBL指令，一页只有一个ROM数据区(0X100以下)，否则，1页有两个ROM数据区。

5.8 嵌入汇编

编译器有嵌入汇编语句的功能，以增强程序的灵活性。

5.8.1 保留字

嵌入汇编语句的保留字是：

```
_asm
{
      …… //写汇编语句区域
}
```

EM78系列所有的汇编指令(高级或者低级的)都支持。

注意

- *0X10—0X1F寄存器为C编译器保留，不建议使用这些保留寄存器。*
- *如果用户不得不在内嵌的汇编语句内切换 “rpage”、“iopage” 或者 “bank”，那么原始的“rpage”、“iopage”或者 “bank”必须在一开始就进行保存并且在汇编语句的结尾进行恢复。参考下面部分的例子 (5.8.2)。*

5.8.2 嵌入汇编语句里C变量的使用

编译器允许按照如下的方法在嵌入的汇编里使用C变量：

```
mov a, %<variable name>   //将变量值给ACC
mov a, @%<variable name>  //将变量的地址给ACC
```

范例1:

```
_asm
{
//保存rpage，iopage和 bank 寄存器的步骤
     mov a,0x0
     mov %nbuf, a
     mov a, 0x04
     mov %nbuf+1, a
     bs 0x03, 7
     bs 0x03, 6    //切换到其他页(rpages)
     ……
                   //恢复rpage，iopage和bank寄存器的步骤
     mov a, %nbuf
     mov 0x03, a
     mov a, %nbuf + 1
     mov 0x04, a
}
```

范例2:

```
int temp;
temp=0x03;                //假设temp位于bank 0的0x21地址
_asm {mov a, %temp}       //将值0X03给ACC
_asm {mov a, @%temp}      //将地址0x21给ACC
```

范例3:

```
unsigned int temp_a @0x20: bank 0;
unsigned int temp_s @0x21: bank 0;
#define status 0x03;
void main()
{
_    asm
     {
        mov %temp_a, a   // ➔ mov 0x20, a
        mov a, status    // ➔ mov a, 0x03
        mov %temp_s, a   // ➔ mov 0x21, a
     }
}
```

5.9 使用宏指令

您可以使用宏指令来控制MCU并且减小程序长度。

注意

- *使用"#define"来定义宏。*
- *使用"\"来加入多行汇编代码。*
- *不要在"\"后使用任何字符(即使一个字长的字符集也是不允许的)，否则将会出现错误。*

范例：

```
#define SetIO(portnum, value)   _asm {mov a, @value} \
                                _asm {iow portnum}
```

5.10 中断程序

处理中断，两件事情必须考虑：

1. **中断保护：**必须在执行中断服务程序前进行一些寄存器的保护，例如EM78P458的A与R3在中断出现的时候应该被保护起来，在中断保护程序下，只有汇编程序才允许这么做。
2. **中断服务程序：**中断的执行。

注意

- *在WICEPLUS执行任务时，您或许会忽略设置中断的细节。然而，您需要在中断服务上倾注更多的注意力。*
- *“page”指令不能分配相同的ROM空间给中断服务程序(见3.6 .)。*

5.10.1 中断保护程序

```
void _intcall <函数名>_I(void) @<中断向量地址>: low_int <中断向量数>
```

需注意的是“_1”(低级中断)必须加在函数名的后面。

5.10.2 中断服务程序

```
void _intcall <函数名>(void) @int <中断向量数>
```

<中断向量数>是指如果MCU有几个中断向量，那么0,1,2,3….将每种中断向量分开。

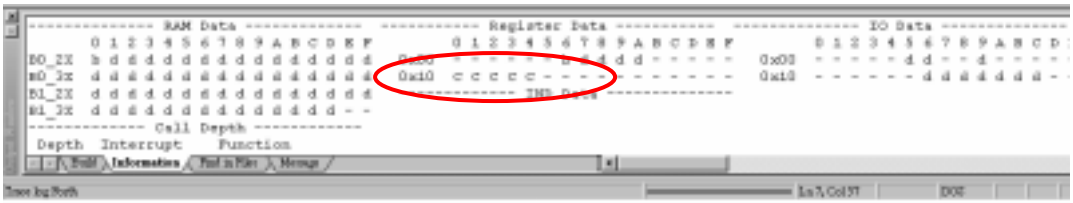
编译器将会自动在<中断向量数>内将中断保护程序和中断服务程序结合起来。

注意

- *中断保护程序和中断服务程序不能分配参数，否则编译器会出错。*
- *中断程序只支持单个字节数据类型的操作(例如 “int”，“char”),否则编译器会出错。*
- *在中断程序下，你可以调用其他的函数，但是长型数据类型 *、/和%不允许出现在函数中，参考下面的例3。*
- *每款MCU都必须在中断保护程序内嵌入汇编程序来保存一些寄存器，例如，当中断发生时，EM78P806B的ACC、R3、R4和R5寄存器需保存。如果MCU支持硬件备份,您可以忽略在中断保护程序内保护寄存器。*
- *在C编译器内，地址为0X10~0X1F的16个通用寄存器是给某些操作数保留的。当中断发生时，强烈推荐在中断服务程序之前备份所有的寄存器（0X10—0X1F）。否则运行结果可能出错。*
- *希望只存储少量寄存器的熟练用户须注意地是执行了什么操作，（例如，如果在中断服务程序里不用“*”、“/”或者 ”%”，您可以不用存储0X1D和0X1E寄存器），下表列出使用特殊寄存器的某些操作。在这个表的基础上，你可以确定哪些寄存器需要存储或哪些不需要存储。*
- *用户不可使用那些用来备份ACC, R3, R4, R5及其它通用寄存器0x10~0x1F的RAM空间。*

5.10.3 保留的通用寄存器的操作

十六个通用寄存器(0x10~0x1f)是为某些运算而保留的，当中断发生时，强烈建议用户备份一些通用寄存器，编译后WicePlus将从信息窗口和输出窗口内提醒用户哪些寄存器必需备份。在下图里，用户可看到0x10所在的行有五个C字符，这些位置是0x10, 0x11, 0x12, 0x13, 0x14地址，因此，在中断服务子程式里用户必需保存和恢复这五个通用寄存器。也就是说，用户必需编译一下以获得这些信息，0x10所在行的C字符即表示C编译器在其它函数里占用了的地址，WicePlus 2版本的编译器将不定地使用到这些寄存器。



范例1:

```
Void _intcall INTERRUPT1_l(void) @0x08: low_int 0
{
     // backup ACC, R3, R4, R5
     _asm
     {
          MOV 0X1F, A
          SWAPA 0X4
          BS 0X4, 6//切换到RAM bank 3
          BS 0X4, 7
          MOV 0X3F, A
          SWAPA 0X3
          MOV 0X3E, A
          SWAPA 0X5
          MOV 0X3D, A
          PAGE @0X0   //若MCU不支持page指令，用BC指令将程序切换到第0页
     }
}

void _intcall INTTERRUPT1(void) @int 0
{
     // backup C system
     _asm
     {
          MOV A, 0X10 //使用2字节C数据类型，C系统备份
          MOV 0X3C, A  //将0x10~0x19保存到bank 3的0x3C,0X3B, 0X3A,
          MOV A, 0X11 //0X39, 0X38, 0X37，因为在intcall INTERRUPT1_l
          MOV 0X3B, A //子程式里已将ram切换到了bank 3
          MOV A, 0X14
          MOV 0X3A, A
          MOV A, 0X15
          MOV 0X39, A
          MOV A, 0X18
          MOV 0X38, A
          MOV A, 0X19
          MOV 0X37, A
     }
```

```
        // Write your code (inline assembly or C) here
        ……
        // restore C system
        _asm
        {
             BS 0X04, 6//切换到bank 3 恢复寄存器的内容
             BS 0X04, 7
             MOV A, 0X3C //使用2字节C类型数据，C系统恢复
             MOV 0X10, A
             MOV A, 0X3B
             MOV 0X11, A
             MOV A, 0X3A
             MOV 0X14, A
             MOV A, 0X39
             MOV 0X15, A
             MOV A, 0X38
             MOV 0X18, A
             MOV A, 0X37
             MOV 0X19, A
        }
        // restore ACC, R3, R4, R5 following backup C system
        _asm
        {
             SWAPA 0X3D //用户必需确认是否在RAM bank 3
             MOV 0X5, A  //如果不是，必需切换到RAM bank 3进行正确恢复
             SWAPA 0X3E //
             MOV 0X3, A
             SWAPA 0X3F
             MOV 0X4, A
             SWAP 0X1F
             SWAPA 0X1F
        }
}
```

范例2:

```
int nBuf[5];
void _intcall INTERRUPT2_l(void) @0x08: low_int 0
{
     // 备份ACC, R3, R4, R5
     _asm
     {
          ……
     }
}
```

```
void _intcall INTERRUPT2(void) @int 0
{
     _asm //保存寄存器
     {
          mov a, 0x10
          mov %nBuf, a
          mov a, 0x14
          mov %nBuf + 1, a
          mov a, 0x18
          mov %nBuf + 2, a
          mov a, 0x1B
          mov %nBuf + 3, a
          mov a, 0x1C
          mov %nBuf + 4, a
     }
     // 中断发生时做你想做的事情。
     ……
     _asm //恢复寄存器
     {
          mov a, %Buf
          mov 0x10, a
          mov a, %nBuf + 1
          mov 0x14, a
          mov a, %nBuf+2
          mov 0x18, a
          mov a, %nBuf + 3
          mov 0x1B, a
          mov a, %nBuf + 4
          mov 0x1C, a
     }
}
```

范例3:

```
void _intcall INTERRUPT3_l(void) @0x08: low_int 0
{
     // 备份ACC, R3, R4, R5
     _asm
     {
          ……
     }
}
void _intcall INTERRUPT3(void)@int 0
{
     long ans;
     ……
     ans = LongMult(0x1234, 0x5678 );
     ……
}
long LongMult(long a, long b)
{
     return (a * b);
          //长型数据类型的乘法操作是不允许的！
```

附录 A
转换表

A-1 C和汇编代码间的转换对应表

由WICEPLUS产生汇编代码

Description	C Statement Example	Assembly Code	Conversion Rate (Compiler's Code Size / General User's Code Size * 100%)
整型变量	intVar1 = 0xFF;	MOV A, @0xFF MOV %intVar1, A	100% (2 / 2 * 100)
	intVar2 = intVar1;	MOV A, %intVar1 MOV %intVar2, A	100% (2 / 2 * 100)
字符变量	charVar1 = 0xFF;	MOV A, @0xff MOV %charVar1, A	100% (2 / 2 * 100)
	charVar2 = intVar1;	MOV A, %charVar1 MOV %charVar2, A	100% (2 / 2 * 100)

短型变量	shortVar1 = 0x1234;	MOV A, @0x34
		MOV %shortVar1, A
		MOV a, @0x12
		MOV %shortVar1+1, A
			100% (4 / 4 * 100)

	shortVar2 = shortVar1;	MOV A, %shortVar1
		MOV %shortVar2, A
		MOV A, %shortVar1+1
		MOV %shortVar2+1, A
			100% (4 / 4 * 100)

长型变量	longVar1 = 0x123456;	MOV A, @0x56
		MOV %longVar1, A
		MOV A, @0x34
		MOV %longVar1+1, A
		MOV A, @0x12
		MOV %longVar1+2, A
			100% (6 / 6 * 100)

	longVar2 = longVar1	MOV A, %longVar1
		MOV %longVar2, A
		MOV A, %longVar1+1
		MOV %longVar2+1, A
		MOV A, %longVar1+2
		MOV %longVar2+2, A
			100% (6 / 6 * 100)

For 循环	for (i = 0; i < 5; i++)
	{
	……
	}
		CLR %i
		JMP L2
		L1:
		……
		L2:
		INC %i
		MOV A, @0x05
		SUB A, 0x14
		JBS 0x03, 0
		JMP L1
			100% (7 / 7 * 100)

While 语句	while (cnt != 1)
	{
	……
	}
		L1:
		…
		
		MOV A, %cnt
		XOR A, @0X01
		JBS 0X03,2
		JMP L1
			100% (4 / 4 * 100)

Do-while 语句	do
	{
	……
	} while (cnt != 1);
		L1:
		……
		MOV A, %cnt
		MOV A, @0x01
		XOR A, @0x01
		JBS 0x03, 2
		JMP L1
			100% (4 / 4 * 100)

Do-while 语句	do { Var_c2++; }while(-- var_c1);	L1: INC %var_c2; DJZ %var_c1; JMP L1	100%(3/3*100)

If-else 语句	unsigned int cnt;
	if (cnt == 0)
	{
	……
	}
	else if (cnt < 5)
	{
	……
	}
	else
	{
	……
	}
		MOV A, %cnt
		JBS 0x03, 2
		JMP L1
		……
		JMP ENDIF
		L1:
		MOV A,@0X05
		SUB A, %cnt
		JBC 0x03, 0
		JMP ENDIF
		……
		JMP L2
		L2:
		……
		ENDIF:
			100% (10 / 10 * 100)

Switch 语句	unsigned int cnt;
	switch(cnt)
	{
	case 1:
	……
	break;
	case 2:
	……
	break;
	case 3:
	……
	break;
	default:
	……
	break;
	}
		MOV A,%cnt
		MOV 0X14,A
		MOV A,0X14
		XOR A,@0x01
		JBC 0X03,2
		JMP case 2
		MOV A,0X14
		XOR A,@0X02
		JBC 0X03,2
		JMP case 2
		MOV A,0X14
		XOR A,@0X3
		JBC 0X03,2
		JMP case 3
		JMP default
		
		Case 1:
		JMP …ENDSWITCH
		Case 2:
		JMP ENDSWITCH
		Case 3:
		JMP ENDSWITCH
		Case 4:
		default
		
		ENDSWITCH
			106% (18 / 17 * 100)

函数	main()
	{
	int i;
	i = fun(3);
	
	return;
	}
	
	int fun(int in)
	{
	return in+1;
	}
		; using EM78806B
		MOV A, @0x03
		BANK @0
		MOV %in, A
		CALL FUN
		MOV A, 0x10
		BANK @0
		MOV %i, A
		RET
		
		FUN:
		MOV A, 0x14
		BANK @0
		MOV %temp1, A
		MOV A,%in
		MOV 0x14,A
		MOV 0X10,A
		MOV A,@0X01
		ADD 0X10, A
		MOV A,%temp1
		MOV 0X14,A
		RET
			136% (19 / 14 * 100)

Const 数据	const int myConst[5] = {1, 2, 3, 4, 5};
	
	main()
	{
	int i;
	
	i = myConst[3];
	
	return;
	}
		; using EM78569
		
		MOV A, @0x3D
		MOV 0x19, A
		MOV A, @0x1F
		MOV 0x1A, A
		PAGE @0x0F
		CALL 0x280
		PAGE @0x00
		BC 0x04, 6
		BC 0x04, 7
		MOV 0x20, A
		…
		BC 0X03,0
		RLCA 0x1A
		TBL
		…
		PAGE @0x0F
		JMP 0x2FE
		MOV A, 0x19
		TBL
		….
		
		RETL @0x01
		RETL @0x02
		RETL @0x03
		RETL @0x04
		RETL @0x05
			162% (21 / 13 * 100)

寄存器页	unsigned int myR5P0 @0x05: rpage 0;
	unsigned int myR5P1 @0x05: rpage 1;
	unsigned int myR5P2 @0x05: rpage 2;
	
	myR5P0 = 0x12;
	myR5P1 = 0x34;
	myR5P2 = 0x56;
		; using EM78P468N
		MOV A, @0x12
		BS 0X03,6
		MOV 0x05, A
		
		MOV A, @0x34
		BS 0x03, 6
		BC 0x03, 7
		MOV 0x05, A
		
		MOV A, @0x56
		BC 0x03, 6
		BS 0x03, 7
		MOV 0x05, A
			100% (11 / 11 * 100)

I/O 控制寄存器页	io unsigned int myIO6P0 @0x06: rpage 0;
	io unsigned int myIO6P1 @0x06: rpage 1;
	io unsigned int myIO7P1 @0x07: rpage 1;
	
	myIO6P0 = 0x00;
	myIO6P1 = 0xFF;
	myIO7P1 = 0x55;
		; using EM78569
		MOV A, @0x00
		BC 0x03, 5
		IOW 0x6
		
		MOV A, @0xFF
		BS 0x03, 5
		IOW 0x6
		
		MOV A, @0x55
		IOW 0x7
			100% (8 / 8 * 100)

RAM 区	unsigned int myData1 @0x20: bank 0;
	unsigned int myData2 @0x21: bank 0;
	unsigned int myData3 @0x21: bank 1;
	
	myData1 = 1;
	myData2 = 2;
	myData3 = 3;
		; using EM78569
		MOV A, @0x01
		BC 0x04, 6
		BC 0x04, 7
		MOV 0x20, A
		MOV A, @0x02
		MOV 0x21, A
		MOV A, @0x03
		BS 0x04, 6
		BC 0x04, 7
		MOV 0x21, A
			100% (10 / 10 * 100)

Bit 数据类型	bit myB0R6P0 @0x06@0x00: rpage 0;
	bit myB2R6P0 @0x06@0x02: rpage 0;
	
	myB0R6P0 = 1;
	myB2R6P0 = myB0R6P0;
		BS 0x06, 0
		MOV A, @0x00
		JBC 0x06, 0
		ADD A, @0x01
		MOV 0x14, A
		BC 0x06, 2
		JBC 0x14, 0
		BS 0x06, 2
			200% (8 / 4 * 100)

间接寻址	indir unsigned myData1 @0x30: ind 0;
	indir unsigned myData2 @0x05: ind 1;
	
	myData1 = 0x55;
	myData2 = 0xAA;
		; using EM78806B
		MOV A, @0x55
		MOV 0x1B, A
		MOV A, @0x30
		MOV 0x18, A
		MOV A, @0x00
		MOV 0x19, A
		MOV A, @0x00
		MOV 0x1A, A
		MOV A, 0x1B
		CALL INDIR
		MOV A, @0xAA
		MOV 0x1B, A
		MOV A, @0x05
		MOV 0x18, A
		MOV A, @0x00
		MOV 0x19, A
		MOV A, @0x01
		MOV 0x1A, A
		MOV A, 0x1B
		CALL INDIR
		……
			146% (35 / 24 * 100)

		INDIR:
		BC 0x05, 0
		MOV 0x1B, A
		MOV A, 0x1A
		JBS 0x03, 2
		JMP 0x081
		MOV A, 0x18
		IOW 0x9
		MOV A, 0x1B
		IOW 0xA
		RET
		
		LCDRAM:
		MOV A, 0x18
		MOV 0x0A, A
		MOV A, 0x1B
		MOV 0x0B, A
		RET	

位操作
(所有的变量都是整型数据类型)
	f = e & d;
	(f = e ^ d;)
	(f = e | d;)
		MOV A, %e
		AND A, %d
		(XOR A, %d)
		(OR A, %d)
		MOV %f, A
			100% (3 / 3 * 100)

	f=~e;	COMA %e MOV %f, A	100%(2/2*100)
	f &=e; (f ^=e;) (f |= e)	MOV A, %e AND %f , A (XOR %f, A) (OR %f, A)	100%(2/2*100)

	f = e >> 1;	MOV A, %e
		MOV 0x14, A
		BC 0x03, 0
		RRCA 0x14
		MOV %f, A
			167% (5 / 3 * 100)

	f = e << 1;	MOV A, %e
		MOV 0x14, A
		BC 0x03, 0
		RLCA 0x14
		MOV %f, A
			167% (5 / 3*100)

	f>>=3;	BC 0x03, 0 RRC %f BC 0x03, 0 RRC %f BC 0x03, 0 RRC %f	100%(6/6*100)
	f<<=3	BC 0x03, 0 RLC %f BC 0x03, 0 RLC %f BC 0x03, 0 RLC %f	100%(6/6*100)

	f>>=4	SWAPA 0x06 AND A, @0x0F MOV 0x06, A	100%(3/3*100)
	f<<=4	SWAPA 0x06 AND A, @0xF0 MOV 0x06, A	100%(3/3*100)
	f>>=6;	SWAP 0x06 RRC 0x06 RRCA 0x06 AND A, @0x03 MOV 0x06, A	100%(5/5*100)
	f<<=6;	SWAP 0x06 RLC 0x06 RLCA 0x06 AND A, @0xC0 MOV 0x06, A	100%(5/5*100)

	f=(e<<5) | d;	MOV A, %e
		MOV 0x14, A
		SWAP 0x14
		RLCA 0x14
		AND A, @0xE0
		OR A, %d
		MOV %f, A
			100%(7/7*100)

	f=(f & const.1) | const. 2	MOV A, 0x06
		AND A, const. 1
		OR A, const. 2
		MOV 0x06, A
			100%(4/4*100)

算术表达式
(所有的变量都是整型数据类型)
	f = e + d;	MOV A, %e
		ADD A, %d
		MOV %f, A
			100% (3 / 3 * 100)

	f = e – d;	MOV A, %d
		SUB A, %e
		MOV %f, A
			100%(3/3*100)

	f++;	INC %f	100% (1 / 1 * 100%)
	f—;	DEC %f	100% (1 / 1 * 100%)

	c = a * b;	MOV A, %a
		MOV 0X1C,A
		MOV A, %b
		MOV 0X18, A
		CLRA
		L1:
		ADD A, 0X1C
		DJZ 0X18
		JMP L1
		MOV %c, A
			100%(9/9*100%)

	c = a / b;	MOV A, %a
		MOV 0x1C, A
		MOV A, %b
		CLR 0x18
		L1:
		SUB 0x1C, A
		JBC 0x03, 0
		INC 0x18
		JBC 0x03, 0
		JMP 0x3BB
		MOV A, 0x18
		MOV %c, A
			100%(11/11*100%)

混合运算
(所有的变量都是整型数据类型)
	f += e;
	(f -= e;)
	(f &= e)
	(f ^= e)
	(f |= e)
		MOV A, %e
		ADD %f, A
		(SUB %f, A)
		(AND %f, A)
		(XOR %f, A)
		(OR %f, A)
			100% (2 / 2 * 100%)

	f >>= 1;	BC 0x03,0 RRC %f	100% (2 / 2 * 100%)
	f <<= 1	BC 0x03,0 RLC %f	100% (2 / 2 * 100%)

附录B
常见问题 (FAQ)

Q: 函数参数最多是多少?

A: 依RAM区的大小而定 (约 32或31字节)。

Q: 在函数内，CALL函数最深可到多少?

A: 依据硬件堆栈大小而定。

Q: 最大数组是多少?

A: 依据RAM区的大小而定 (约 32或31字节)。

Q: 如果代码超出了ROM范围，会提示错误信息吗?

A: 会，连接器将报告错误信息。

Q: 在高级别的中断子程序内，用户可以在ROM内分配地址吗？ (例如，使用“page”数据类型，在子程序前放置“_asm” ｛org xxx｝等)。

A: 不可以，这样会产生不可预知的错误！

Q: “static”的使用方法与ANSI C相同吗？

A: 是的。

Q: 万一用户用“const”定义了很多的常数，而且超过了ROM空间，是否会有错误信息?

A: 是的，连接器会报告分配错误。

Q: 如何在*.h文件里定义变量，而不只在.c文件里使用。

A: 例如，在*.h文件里定义如下：

```
extern io unsigned int DIRPORT6;
```

但在*.c文件里，需定义成如下形式：

```
io unsigned int DIRPORT6   @0x06: iopage 0;
```

Q: 可以修改程序页或RAM区吗？

A: 如果只是用C语言编写程式，不需要切换任何程序页、寄存器页、RAM区等，但如果在程序里嵌入汇编，你必需保存或恢复程序页或RAM区。

Q: 可否知道有多少级堆栈可以调用？

A: 可以，在C语言开发环境里编译完成后，用户可以从输出窗口获得函数调用级别的信息。

Q: C编译器仅占用0x10~0x1F通用RAM吗?

A: 对，C编译器基本上只占用0x10~0x1F 的通用寄存器，但在调用函数里若有一些局部变量，编译器将会用到bank 0 ~ bank 3的0x20~0x3F的一些地址。 因此，我们建议用户在函数里使用全局变量代替局部变量。

用户必需注意地是在中断保护程序和中断服务程序里已用了一些RAM空间，因此用户不能再使用这些RAM空间。