第 八 章

HT-IDE 2000

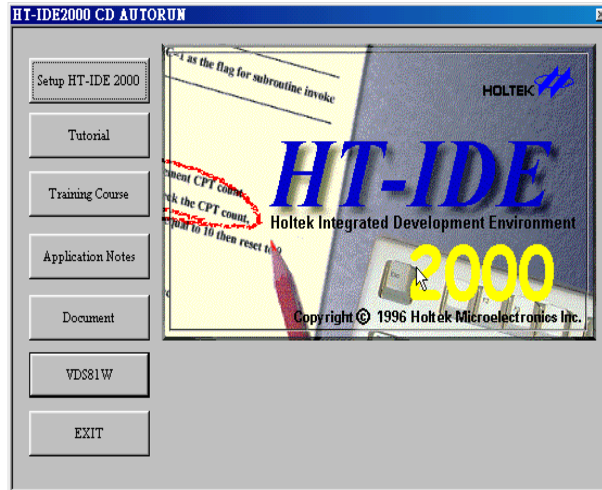
8

HT-IDE 2000 是盛群公司于 2000 年最新推出的八位元单片机发展系统软件工具，支持的工作平台包括 Windows 95/98/NT/2000，它继承了原 HT-IDE 完善易操作的用户界面，系统综合了文字编辑器(Editor)，编译器(Assembler)，C 编译器 (Compiler)，连结器 (Linker)，载入器 (Loader) 等，大大地缩短了单片机应用程序的开发时间，而其强大的纠错功能，更是用户不可缺少的有利工具，另外，它所提供的一些应用程序，例如程序库管理器(Library Manager)，虚拟外围元器件管理器 (VPM)，LCD 模拟器 (HT-LCDS) 以及单次烧录芯片的烧录工具 (HT-OTP 与 HandyWriter) 等等，都使得整个系统的功能显得更加完备。
本章将说明 HT-IDE2000 的安装程序，随后用一个简单的范例来说明使用发展系统开发一个项目的流程，让读者对整个 HT-IDE 2000 有个基本的认识。

安装

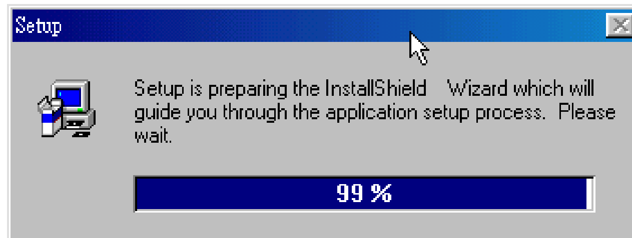
HT-IDE 2000 目前是以 CD 的方式发行，你也可以随时从盛群公司的网站 (http://www.holtek.com.tw/)下载最新版的 HT-IDE 2000，以下介绍用 CD 安装 HT-IDE 2000 的流程。

将 HT-IDE 2000 CD 放入 PC 的光驱中，屏幕会出现以下的画面：

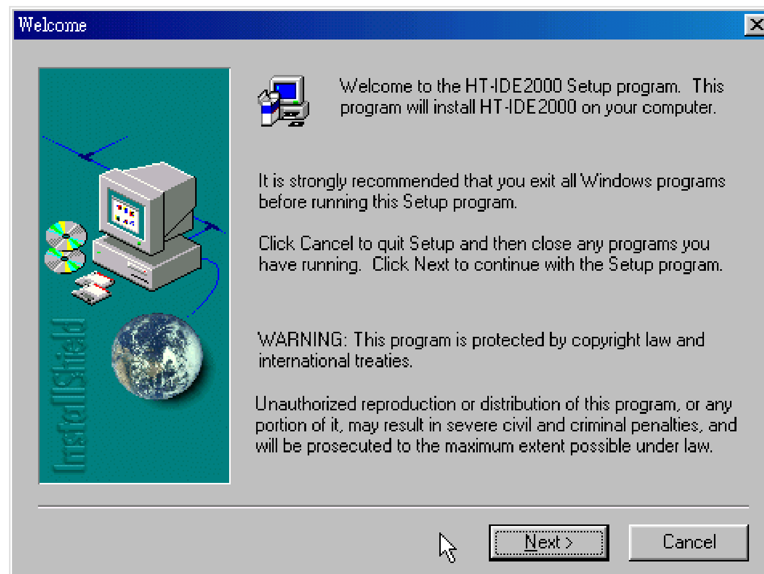


请按下 Setup HT-IDE 2000 选项，若你是第一次安装 HT-IDE 2000 v5.0，请选择 INSTALL HT-IDE 2000，若你的电脑已安装过 HT-IDE2000 v5.0，则你可以选择 INSTALL HT-IDE 2000 SERVICE PACK，此时系统只会安装修正过的模组与新母体资料，以节省安装时间。

在选择 INSTALL HT-IDE 2000 之后，会出现以下的 InstallShield 画面：

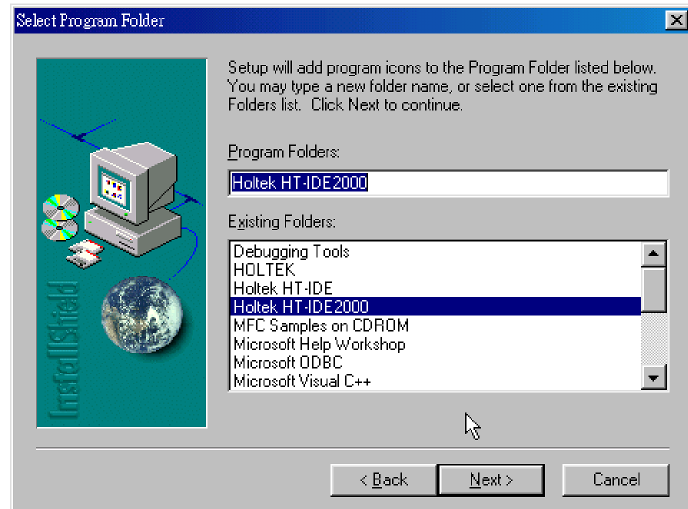


随后并出现以下的 Setup 画面：

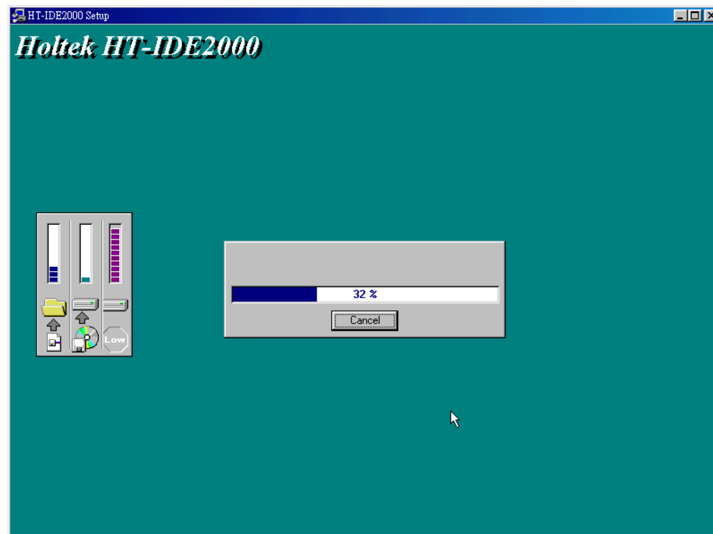


请依照指示输入或选定安装路径。

与程序组名称。



Setup 会自动将文件复制到 PC 的指定目录中。



在安装完毕之后，Setup 会出现以下的画面，提醒你要重新开机以让新安装的 HT-IDE 2000 的设定生效，并建议你选择立即重新开机。

在完成安装并重新开机让设定生效之后，你就可以开始使用 HT-IDE2000 开发你的项目，在启动 HT-IDE2000 之后，系统的启始画面如下：

接下来用一个简单、完整的例子来介绍使用 HT-IDE 2000 发展单片机应用系统的流程，让读者有个整体发展的认识。

一个简单的例子

基本上，HT-IDE 2000 将每一个应用系统看为一独立的项目（Project），并以 Project 为管理系统的基本单位，所以开发每一个应用系统之初都必须先建立一个 project（实际上是一个供 HT-IDE 2000 管理用的文件），此后 IDE 便可辅助管理所有 project 相关的文件和设定。

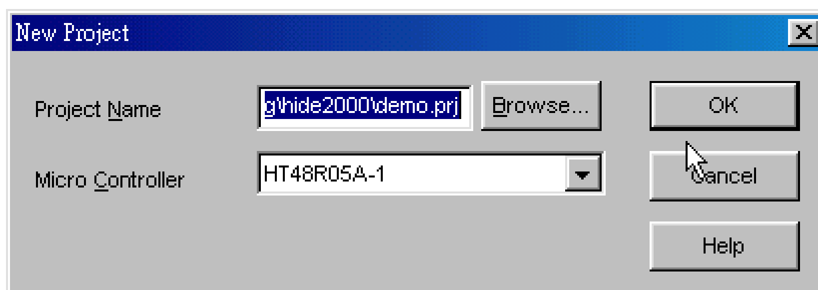
→ 应用程序开发的主要操作步骤如下

- 建立 Project
- 填写 mask option
- 编辑、加入或移除原始文件
- 建立 (Build)
- 除错
- 展示、生产

以下就上述操作步骤逐一介绍：

建立 Project

由 Project 菜单下选用 New 命令（以下用 [Project/New] 表示类似的动作），便会出现 New Project 对话窗，提示你输入 Project 最基本的资料。



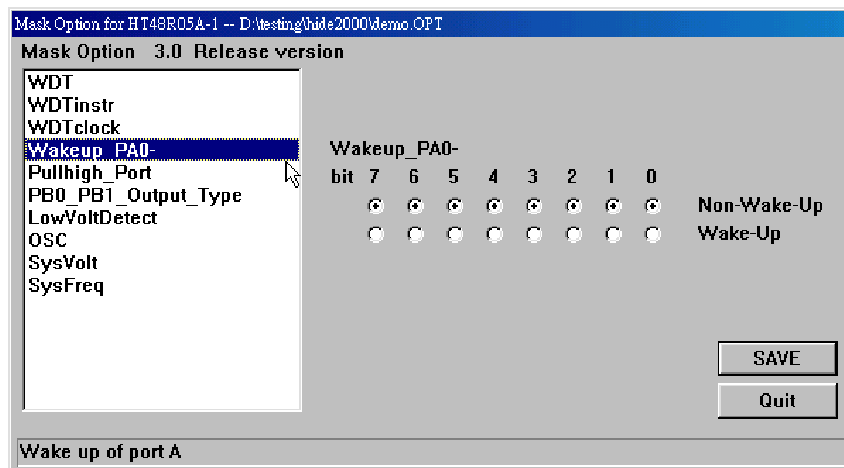
其中，你可以在 Micro Controller 选项中选择适当的单片机，并在 Project Name 对话框中输入该 Project 的路径及名称，Browse 按钮可以帮助您搜寻适当的 Project 的保存路径。确定后按 OK 按钮即可建立一个新的 Project。

这个例子，我们选择 HT48R05A-1 为单片机母体，并在 D:\Testing\HT-IDE 2000 目录下建立一个叫做 demo 的 project。

注意　选用的单片机母体，以后仍可使用 [Options/Project] 命令来更换。

填写 Mask option

新的 Project 建完后 HT-IDE 2000 会自动启动 Mask option 编辑器（如下图），供用户做修改。



所有的 Mask option 所代表的意义，你可以在 用户手册的其他章节找到。

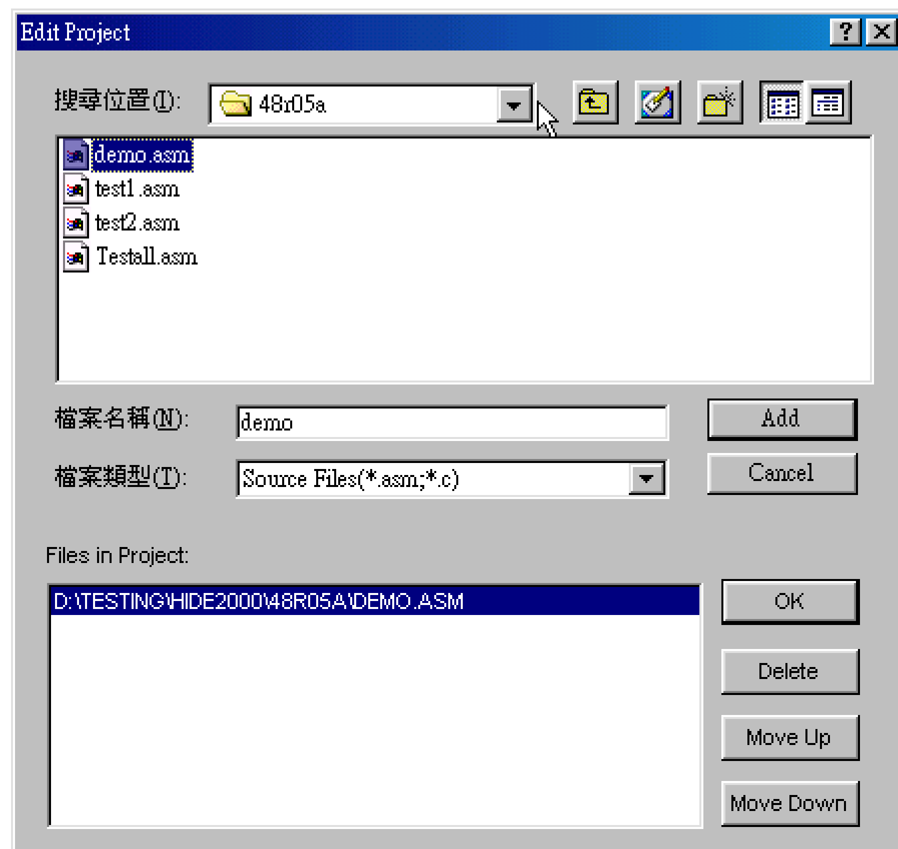
其中特别要注意的是：SysFreq (System Frequency)，系统频率的来源有两种模式：ICE 内部时钟，外部时钟。如选用外部时钟，用户必须自己在 ICE 的 I/O 接口卡上外接振汤器，或调整可变电阻来产生系统频率。

选择完后，按 Save 按钮以保存所做的设定。以后，你仍可选 [Tools/Mask option] 命令来更改 mask option 设定。现在我们先选用内部频率，其余选项都用初始值，并按 Save 存档。

编辑、加入、删除原始档

HT-IDE 2000 内含了一个文字编辑器，允许同时编辑多个文件，你可以用 [File/New] 命令来建立新的文字档或用 [File/Open] 命令来开启愿有的文字档，编辑的键盘指令和大多数视窗系统下的文字编辑器类似，在此不再赘述。

HT-IDE 2000 的 Project 支持多个文件，所以你必须将原始文件「加入」Project，来告知 HT-IDE 2000 目前的 Project 包含了哪些原始文件。选用 [Project/Edit] 命令可以启动 Edit Project 对话窗（如下图），它提供了加入或移除目前 Project 的原始文件的用户界面。

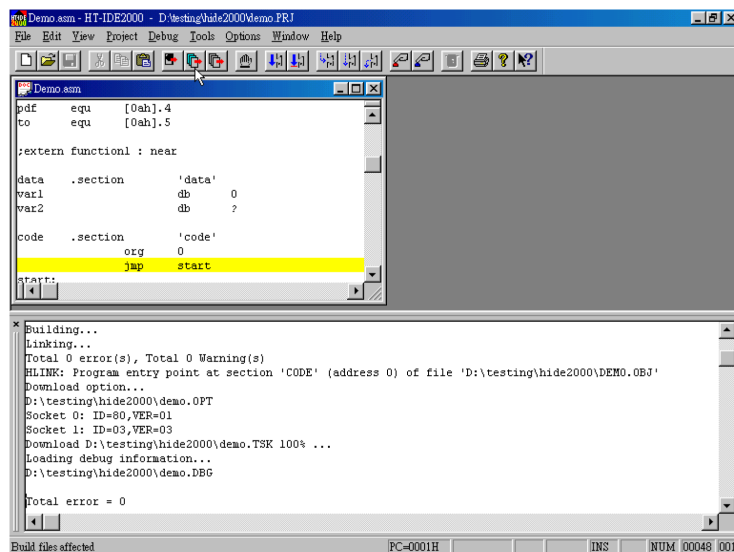


在这范例中，我们将 demo.asm 加入目前的 Project 中。

注意，虽然原始文件可以保存在与 Project 不同的子目录，但是建议你将所有相关的文件（包括原始文件）都放在相同的子目录下，以方便管理 Project。

建立 (Build)

所有的原始文件加入 Project 后，只要选用建立 [Project/Build] 命令就可以进入纠错模式（如果没有语法或其他错误的话）。为方便执行建立命令，HT-IDE 2000 主视窗的工具栏上设计了一个快捷按钮，你可以用鼠标直接点击此快捷按钮来执行建立命令。



所有建立过程与结果会显示在 Output 视窗，如果发现有任何语法上的错误，你可以在 Output 视窗内双击错误行，HT-IDE 2000 便会自动调出错误的原始行，并显示在最上层，以方便你更正错误。

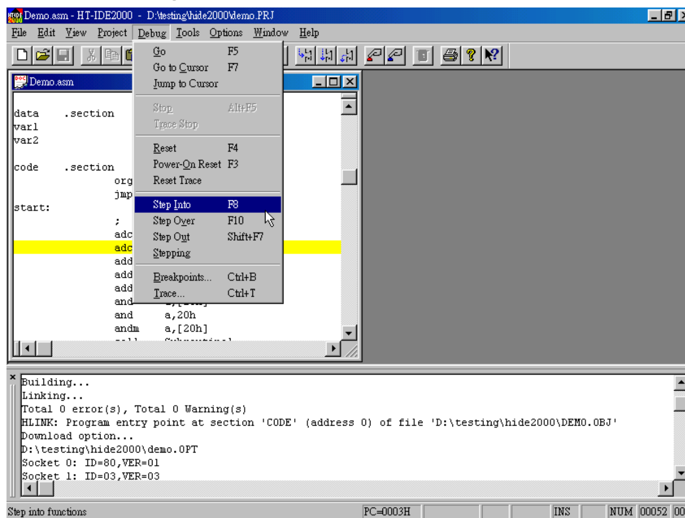
Build 命令实际上包括了编译、连结、设定 ICE 等工作，HT-IDE 2000 会自动决定该做哪些动作、哪些动作可以省略以减少所花费的时间。

注意　　假如 HT-ICE 序号是在 S/N：4800050 之前的号码，硬件是先前的版本会引起 (Latch-up) 死机的问题，这也许会引起 "download failure."，另外，此时 Windows 上不要执行太多任务。

纠错

正确建立完后，便进入纠错模式，HT-IDE 2000 会将程序执行起始行显示在最上层，也就是 Program Counter=0000H 的位置，这时你就可以做所有的纠错命令。

纠错命令都安排在 Debug 选单下（如下图）。

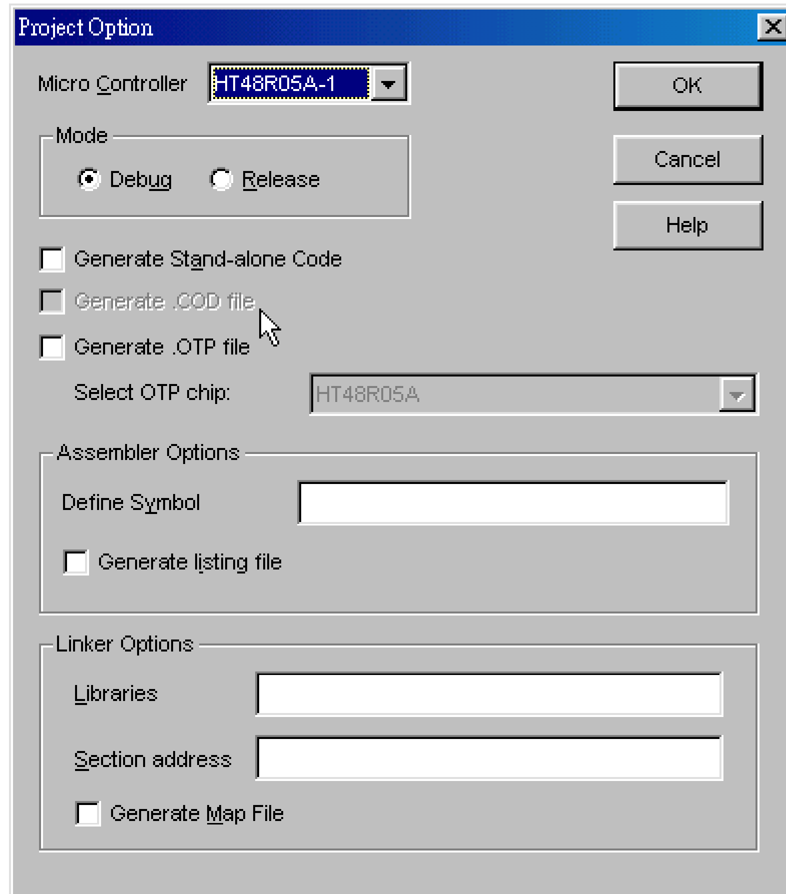


你可以试著选 Step Into 命令来做单步执行，或选 Go 命令做即时模拟执行。纠错命令的意义与使用方法会在后面的章节做较详细的叙述。

展示、生产

纠错模式下，HT-IDE 2000 借由断点、追踪等功能提供用户快速找到逻辑错误的方法，如果你认为程序已经没有问题了，接下来你可以利用 HT-ICE 允许单独执行的特性，来展示你的应用系统。

欲利用 HT-ICE 单独执行应用系统，你必须在 Project Option 对话框（如下图）选中 Generate Stand-alone Code，并重新 Build 一次(选[Options/Project] 命令以启动 Project Option 对话框)。成功后，HT-ICE 便记录了这个应用程序，此时你可以单独展示你的应用程序无需发展系统，若你的 HT-ICE 上的 ROM 是 Flash 版本，你更可以带著它到别的地方，无须电脑，展示发展完成的应用系统。



当你验证完成你的应用程序后，若是要大量生产 mask-type chip，你必须产生 .COD 档并把它连同 mask option 确认单送回盛群公司下单，产生 .COD 档的方法是在 Project Option 对话窗内，选中 Generate .COD file，再重新 Build 一次即可。至于确认单则是用 Project/Print Option Table 的命令将其打印出。若你是要烧录 OTP chip，则你必须先产生 .OTP 档，然后利用 HT-OTP Writer 或 HT-HandyWriter 烧录，产生 .OTP 档的方法也是在 Project Option 对话窗内，选中 Generate .OTP file，再重新 Build 一次即可。

除错的基本技巧

本节介绍一些 HT-IDE 2000 的基本除错技巧，你可以学习到如何设定简单的断点、如何在程序间移动、如何观察单片机的状态，以及模拟 Reset、Powe-on Reset 的方法等。

设断点

HT-IDE 2000 允许用户设定复杂状况的断点，你可以根据程序的地址，资料的地址，甚至外部讯号等作为断点的条件。这里仅介绍设定程序断点的方法。

在工具拦上有个 toggle breakpoints 的快捷按钮，在除错模式下，将光标放置在欲设定断点的程序码行上，按这个快捷按钮即可设定（或取消）断点。成功后，该行会变为以棕色为底，表示该行为一断点。

注意，光标必须放在有指令的程序码行才能设定为断点。

在程序间移动

除错模式下，最主要观察的就是目前 Program Counter 所在的位置，你可以在 HT-IDE 2000 主画面的状态列看到目前的 Program Counter，HT-IDE 2000 也会将所在的原始文件显示在最上层并用黄的底色来指出目前即将执行的指令。

依据除错当时的需求，HT-IDE 2000 提供了数个不同移动 program counter 的方法，分述如后：

→ **Go**

Go 命令启动 ICE 自目前所在的位置开始执行。Go 命令执行后，除非遇到断点或用户下 Stop 命令，才可能使得 ICE 停住。
你可以在 [Debug/Stop] 找到 Stop 命令。

→ Goto Cursor

Goto Cursor 启动 ICE 开始执行直到当前光标所在的行为止。实用上，通常先将光标置放在目标程序的某行上，再执行此命令，便可令 ICE 执行至光标所在行，再继续做后续的除错动作。

注意，光标所在行必须含有指令，否则无法使用此命令。此外，若执行当中遇到断点或用户下 Stop 命令仍会促使 ICE 停止执行。

→ **Jump to Cursor**

Jump to Cursor 与 Goto Cursor 有点类似但并不相同，主要的差异在 ICE 是否执行指令：Goto Cursor 会执行指令直到光标所在行，但 Jump to Cursor 并不执行指令而只是重设 Program Counter 为光标所在行而已。

→ **Step Into**

单步执行是经常使用的除错功能，HT-IDE 2000 将单步执行依据即将执行的指令是否为调用子程序，区分为两种：Step Into 和 Step Over。Into 是指进入子程序，Over 是指跨越子程序。

如使用汇编语言 Step Into 就是传统的单步执行 它会真正的执行单一指令，并自动反应目前 ICE 的状态。如果即将执行的指令为 CALL，执行 Step Into 后，系统会自动调出子程序 (无论它是否在同一个程序) ，并显示在最上层。

→ **Step Over**

当即将执行的指令为子程序调用时，Step Over 会执行完整个子程序后，停在子程序调用后的下一条指令。

实用上，通常在你觉得调用的子程序应该正确无误，或不想细看子程序调用的细节时，可以选用 Step Over 命令。

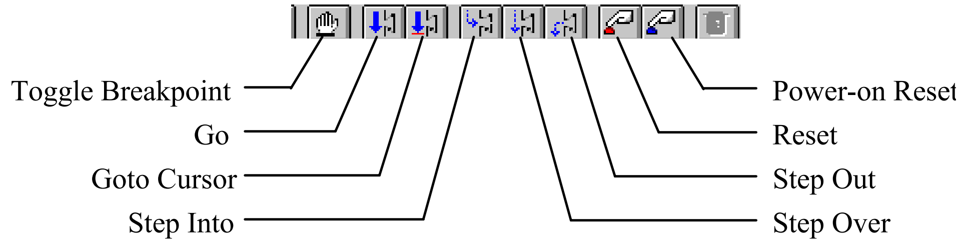
若即将执行的指令不为子程序调用，则 Step Over 与 Step Into 完全相同，都是执行该指令。

→ **Step Out**

若是目前已在某个子程序内，你不想要再继续单步执行下去，希望直接执行完目前的子程序时，可以用 Stet Out 命令。Step Out 表示跳出目前子程序，它会促使 ICE 执行完目前的子程序。

注意 Step Out 一定要在某个子程序内使用，否则会有不可预期的结果，你可以在 Stack 视窗中观察到目前堆栈的层次。

以上移动 program counter 的命令都可以在 Debug 选单内找到，此外，除了 Jump to Cursor 之外， 都在工具列提供了快捷键方便执行命令。

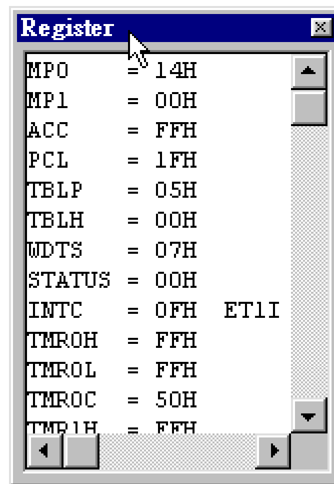


观察单片机的状态

HT-IDE 2000 提供了许多视窗供用户观察、修改单片机的状态，包括：寄存器、变量、RAM、ROM、Trace、Stack、Disassembled code 等。这些视窗都安排在 Windows 菜单下。

Register 视窗

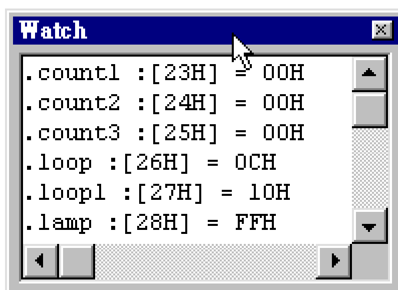
Register 视窗提供一个供用户观察及修改寄存器的场所。HT-IDE 2000 会依据你所选的单片机显示其所有的寄存器。欲修改寄存器，只要直接在画面上覆盖原值，再按 ENTER 即可。注意，部份寄存器为只读型，例如：TBLH、TMR。你无法修改只读型寄存器。



Watch 视窗

Watch 视窗提供一个供用户观察及修改变量的地方。所谓「变量」，是指程序内 Data Section 所定义的变量（注意，不是 EQU 所定义的符号）。

欲观察某个变量，你必须在 Watch 视窗输入变量名称才可以看到。例如：若欲观察变量 count1 的值，你可以输入 .count1 再按 ENTER，屏幕上就会显示 count1 的在 RAM 区的实际地址以及当前值（如下图）。欲修改变量，只要直接在画面上覆盖原值，再按 ENTER 即可。

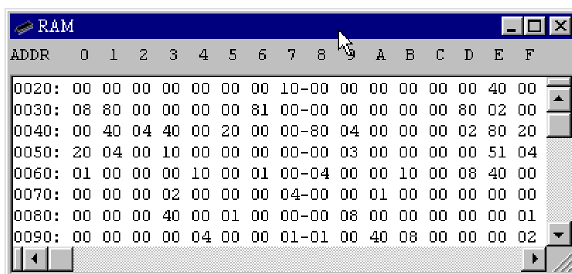


如果变量并非定义在目前 Program Counter 所在的文件，你必须指定变量定义的文件名，系统才能找得到。例如：变量 dtmp 定义在 t2.asm 内，你可以输入 t2.asm!.dtmp 再按 ENTER。

除了变量之外，你也可以利用 Watch 视窗来观看寄存器，只要直接输入寄存器名称（前面不要有'.'），屏幕就会显示寄存器的地址及目前值，当然你也可以修改它。

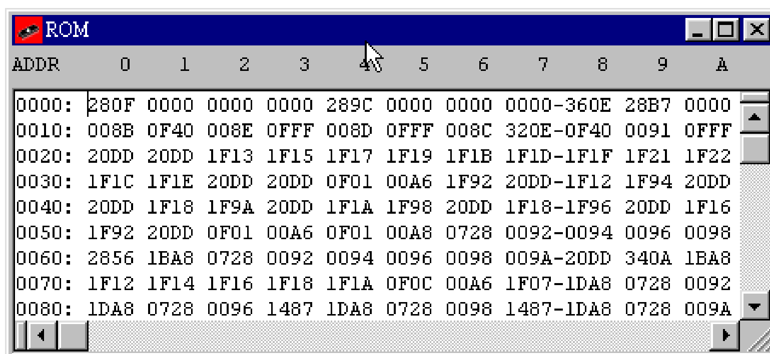
RAM 视窗

RAM 视窗以十六进制位显示所有的 RAM 空间。直接覆盖原值，即表示修改该地址的值。



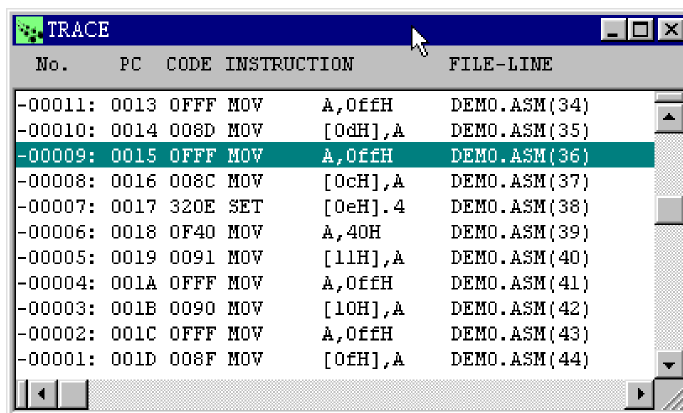
ROM 视窗

ROM 视窗显示 ROM 空间。ROM 空间的大小依 Project 的单片机母体而定。ROM 视窗皆为只读，用户无法修改。



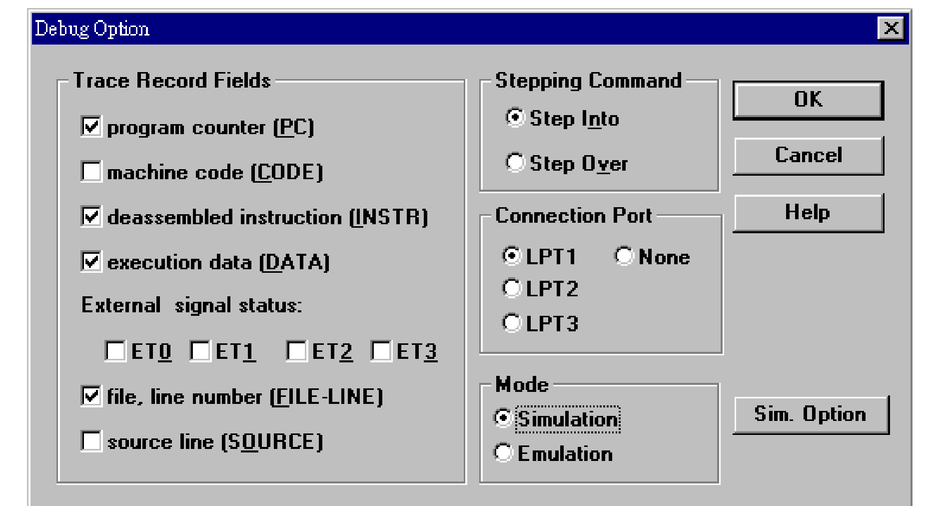
Trace List 视窗

Trace List 视窗记录了最近执行的指令，用户可以由 Trace List 视窗观察已执行的指令。初设的状况下，只要 ICE 执行指令，Trace List 便会在最后增加一笔记录，HT-IDE 2000 也会自动更新 Trace List 的内容，以符合目前 ICE 的状态。



如果你想找出某笔记录的原始程序，你可以用鼠标双击含该笔记录的行，HT-IDE 2000 便会自动调出这笔记录所在的原始档。

Trace List 视窗中的栏位是可以选择的，欲选择显示栏位，执行[Options/Debug]命令引起 Debug Option 对话窗（如下图），Trace Record Fields 内包含所有可选择显示的栏位。



秘诀　整个追踪的记录中，很可能会记录一些你并不感兴趣的程序片段，为了充分应用最大 8K 可记录的空间，你可以使用限定条件列举，来记录你想知道的资料，如下例：

假设 'func' 的执行内容是你不感兴趣的程序片段，而你只想观察变量'data1'& 'data2' 的变化，就可以在追踪设定下，<code space/line number> 部份，使用限定条件的方式，将 MOV 指令的行号加入即可。

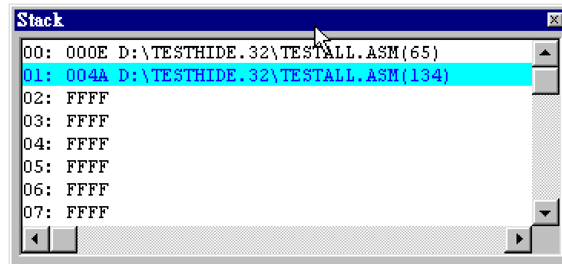
```
loop:
MOV   A, data1
CALL  func
MOV  data2,A
JMP  loop
```

此一结果下，只有两个 MOV 的行号会被记录在整个追踪的记录中，从[Options/Debug] 选择追踪 'execution data'，你就能清楚的看到变量'data1'& 'data2' 的变化。

Stack 视窗

Stack 视窗（如下图）可以让用户看到 stack 深度及目前使用的状况。

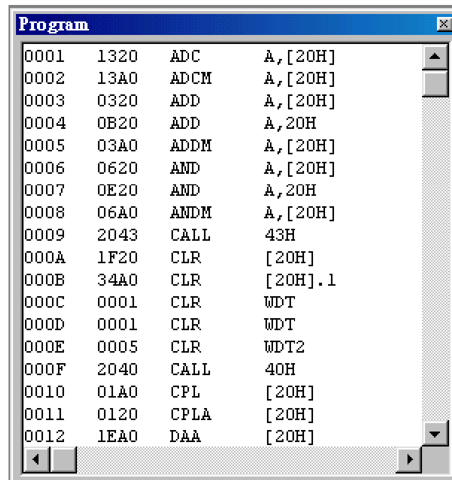
其中最左边的序号表示 stack 的阶层，接下来是 stack 的内容，也就是 return 地址，最右边则显示 return 地址所在的原始程序和行号。



其中，蓝色行表示目前 stack 最上层（top of stack）。你可以用鼠标双击含档名、行号的行，HT-IDE 2000 便可调出这个原始档并将光标移至该行的起始位置。

Program 视窗

Program 视窗让用户以另外一种形式来观察 ROM 空间，除十六进制码外，亦可以看到反汇编的程序代码。



用鼠标双击某行可以调出该指令所在的程序档，并会将光标移至原始程序行的起始位置。

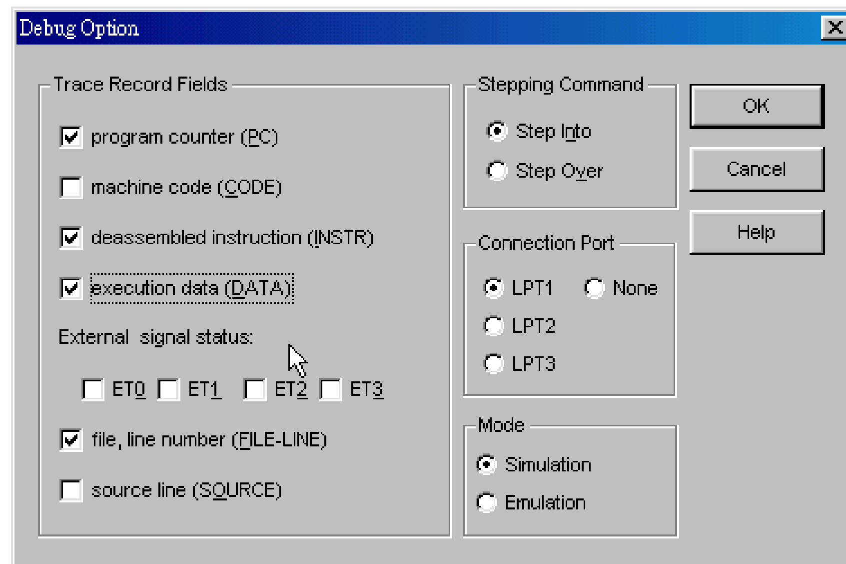
复位 (Reset)

HT-IDE 2000 提供两种 Reset 模式分别模拟单片机的 Reset 和 Power-on reset，这两个命令可以在 Debug 菜单下找到。

软件模拟

HT-IDE 2000 V5.0 之后的版本新增了软件模拟的功能，方便用户在没有 ICE 的状态下也能够学习、熟悉这套系统的强大除错功能。因此，整个软件模拟的操作方式与结果，除了一些与实际输出（入）脚位相关的部份外，所有的除错功能包含断点及 Trace 都力求与 HT-ICE 模拟一致。

首先，欲使用软件模拟，必须执行 [Options/Debug] 命令，启动 Debug Option 对话窗 (如下图)，选用 Simulation Mode 后按 OK 便可切换至软件模拟状态。



另外，HT-IDE 2000 提供了外围元件模拟的功能，请参考第九章 VPM 的使用。

第 九 章

VPM

9

简介

VPM（Virtual Peripheral Manager）的功能是用来模拟外围设备，它提供了许多外部元件，帮助用户建构虚拟电路板，再配合 HT-IDE 2000 与单片机的模拟器 (simulator)，让用户更方便的发展其应用程序。

开发程序的流程为，先确定 HT-IDE 2000 在 simulation 模式下，然后写好程序，编(组)译过后，由 HT-IDE 2000 菜单上的 TOOL 选项启动 VPM，在 VPM 上建构所需的电路。

图-1 显示了 VPM 的一个 project。每个 project 可看成一虚拟电路板，可与 HT-IDE 2000 配合动作。图中分成三个区域，工具栏上列出 VPM 的功能，状态栏则显示一些资料，而在中间有许多元件的区域则可视为电路板。元件可加入此区域，或由此区域删除。按鼠标左键在元件上可选择此元件，被选到的元件会有个框，而且在状态栏的元件区会列出被选到的元件的名称。

状态栏分成四个区域，模式表示目前 VPM 处于何种模式下，有 Running 模式 与 Configure 模式。元件则显示目前被选到元件的名称，执行时间及执行周期则分别显示在 HT-IDE 2000 上的程序执行的时间与周期。当 VPM 处于 Running 模式时，表示虚拟电路正在配合 HT-IDE 2000 动作中，此时执行时间与执行周期则会随著程序执行的时间与周期而增加。

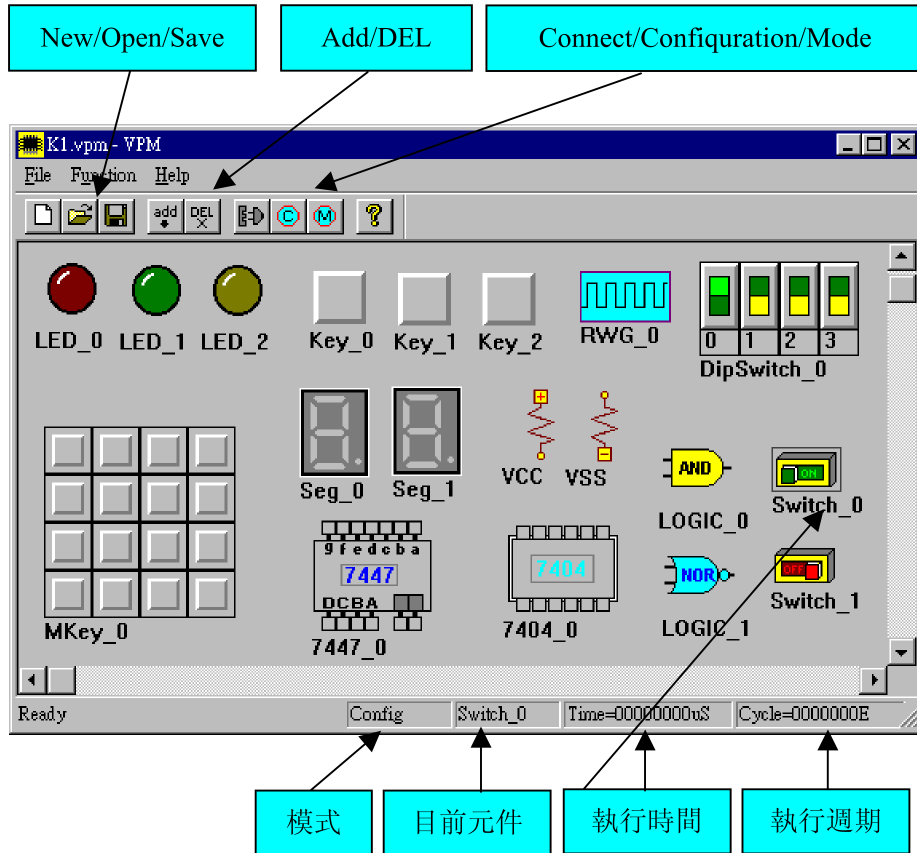


图-1

VPM 菜单

File Menu

文件菜单上有五个功能 (如图-2)，在工具栏上也有相对应的快捷按钮 (如图-3)。

图-2

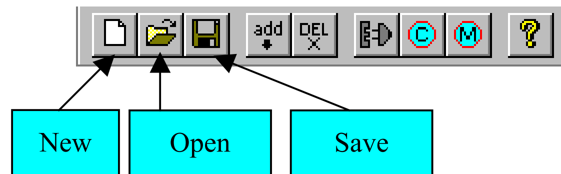


图-3

- New
  打开一新的 VPM 档，每次 VPM 启动时，都会自己打开新的 VPM 档。
- Open
  打开一已存在的 VPM 文件。
- Save
  储存目前的电路。
- Save As
  储存目前的电路至其他不同的名称的文件。
- Exit
  结束 VPM。

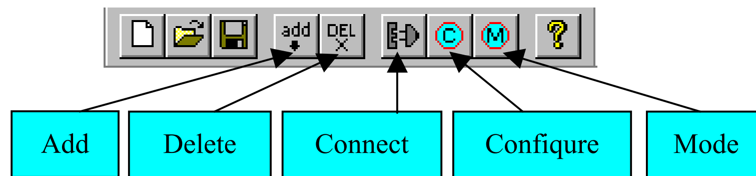
Function Menu

在功能菜单上有五个项目 (如图-4)。

而在工具栏上也可找到相对应的按钮 (如图-5)。

图-4

图-5



- Add

  加入元件至虚拟电路中。

  按工具栏上的 Add 钮，则会出现 Add Peripheral 对话框 (如图-6)。选择要加入的元件再按 OK 钮，或是直接在所要元件上按鼠标左键两下即可。

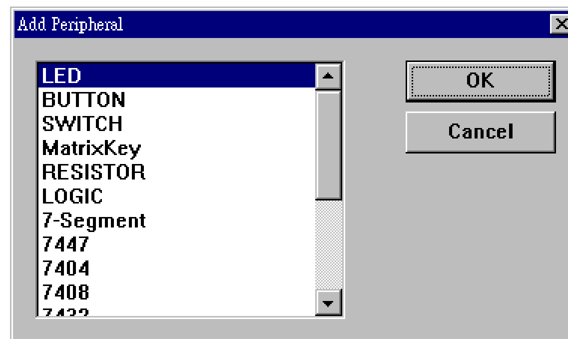


图-6

- Delete

  从虚拟电路中删除一元件。

  先选择所要删除的元件 (在元件上按鼠标左键)，然后按工具栏上的 Del 钮，则被选择的元件会被删除。

- Connect

  此功能为将所加入的元件连结起来。先选择所要连结的元件，然后按工工具栏上的 Connect 钮，会出现 Connect 对话框，如图-7. 对话框中的 connect status list box 显示目前此元件的连结状态，用户可用 connect /disconnect 钮来进行连结或不连结。

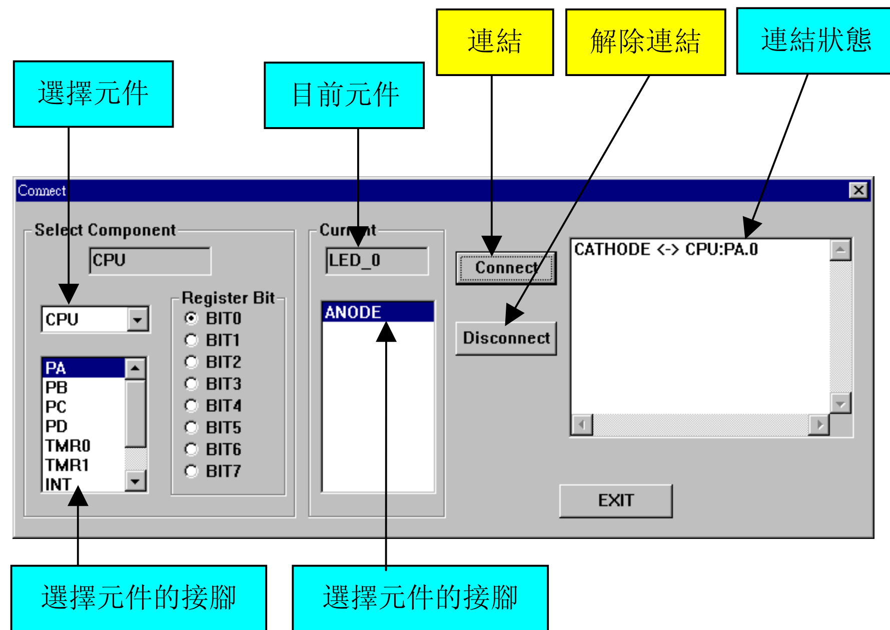


图-7

以 图-7 为例，上图显示名称为 LED_0 的 LED 元件的 Connect 对话框。以此例来说，目前的元件为 LED_0. 选择元件显示所有在虚拟电路板上的元件，LED_0 可与这些元件连结。选择元件的接脚列出了被选择的元件的所的脚位。Register Bit 则显示 pin 脚的细节。

LED 元件有两根 pin 脚：ANODE 与 CATHODE. 上图的连线状态显示 LED_0 的 CATHODE 与 CPU 中的 Port A 的 bit0 连接。

- Configure
  某些元件有选项可设定。选择所要的元件，再按工具栏上的 Configure 钮，若此元件有选项可设定，则会出现Configuration对话框。图-8显示了LED 的 configuration 对话框，由此对话框用户可选择 LED 的颜色与改变 LED 元件的名称。

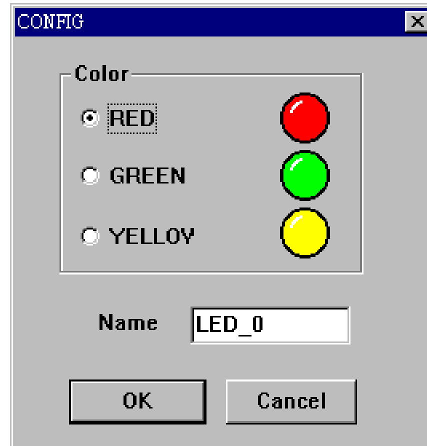


图-8

- Mode
  VPM 有两个模式，Running 模式与 Configuration 模式。
  按工具栏上的 mode 钮会使 VPM 在这两种模式中切换。在 configuration 模式之中，使用者可用 add/delete/configure 来编辑虚拟电路。在 running 模式中，VPM 会随著 HT-IDE 2000 中的单片机 simulator 而动作，也就是说，当 HT-IDE 2000 在 simulation 模式下执行程序时，若 VPM 也在 Running 模式时，虚拟电路会跟著程序而反应。

VPM 元件描述

LED

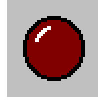


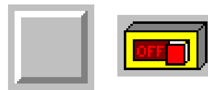
图-9

LED 有两根脚，分别为 CATHODE 与 ANODE

当 CATHODE=0 而且 ANODE=1 时，LED 则会被点亮，LED 可由选项对话框中设定其颜色。

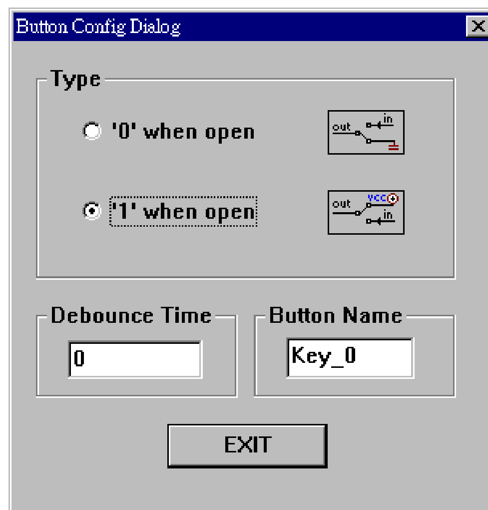
Button/Switch

图-10

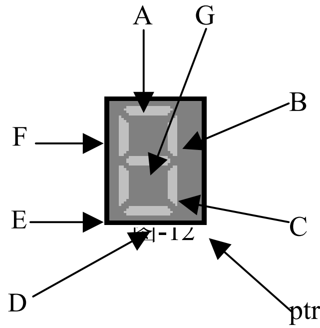


Button/Switch 有三个选项可设定，如下图所示。

图-11



7-Segment display



七段显示器 个亮灯所组成，分 ，B，C，D，E，F，G 与 ptr 。每一根亮灯 输入脚位，输出脚则为共用，输出脚位可由选项设定成为共阴极 (CATHODE) 或共阳极 (ANODE)。

Resistor

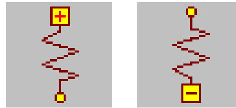


图-13

此元件主要是用来提供 VCC 与 VSS。
用户可由选项设定对话框中设定。

Logic gate

Logic gate 提供六种 logic function (如下图)。

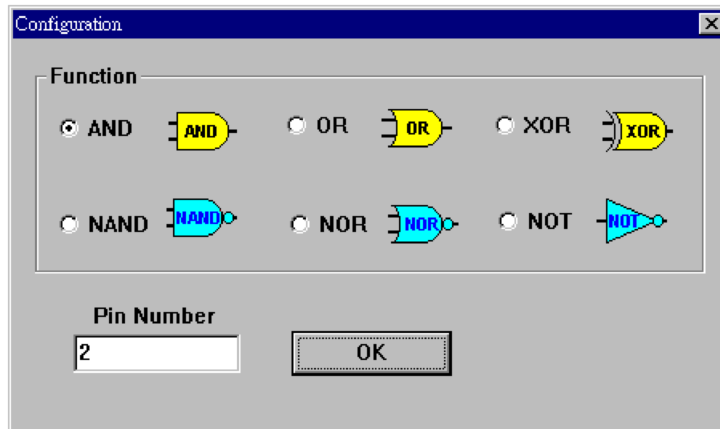


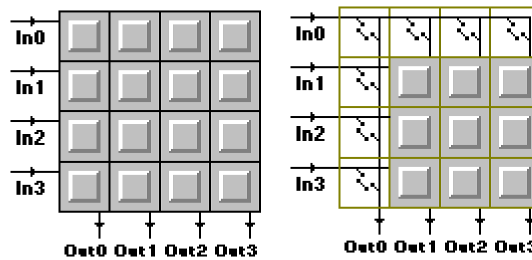
图-14

PinNumber 用来设定 logic gate，的输入脚数，除了 NOT gate 之外 (Not gate 只能有一根输入脚位)。

Matrix key

Matrix key 是很好用的外部元件，用户不需自行由 Push Button 建构 Matrix key，用户可由 configuration dialog box 设定 Matrix key 的大小。

图-15



当用户设定列为 4，行为 4 时 (如图-15)，这表示 Matrix key 有 4 个输入脚(列为 4)，有 4 个输出脚 (行为 4)。

Rectangle Wave Generator

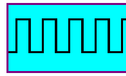


图-16

方波产生器可产生方波给需要的元件，选项对话框中的频率 (frequency) 是以单片机的周期为单位。

范例

HT-IDE 2000 手册中有几个范例，此处选择几个例子来示范如何建构虚拟电路板。

霹雳灯

→ 在 **HT-IDE 2000** 之中

- 新建一 project，body 选择 HT48C10 (Project/New)
- 1.2 将 scanning.asm 加入 project (Project/Edit)
  (此档案可在置于 HT-IDE 2000\SAMPLE\CHAP15)
- 将 HT-IDE 2000 设成 simulation mode。(Options/Debug/Mode)
- 编译此 project。(Project/Build)
- 启动 VPM (Tools/Virtual Peripheral)

→ 在 **VPM** 中

- 建立一新的 VPM project。
- 加入 8 个 LED 到 project 中 (重复按工具列上的 Add 钮，再选 LED，共做八次)
- 加入 Resistor (按 Add 钮，再选 Resistor)
  在刚加入的 Resistor 上按鼠标左键二下，将 Resistor 的名字改为 VCC。

- 将所有的 LED 的 ANODE 接到 VCC，连接 LED_n 的 CATHODE 脚到 CPU 的 PA 的 bit n (n=0~7)。

  此处示范如何将 LED_0 的 ANODE 连接到 VCC 与 LED_0 的 CATHODE 连到 CPU 中的 PA 的 bit n。

  — 在 LED_0 按下鼠标左键，将 LED_0 选成目前元件。

  — 在 LED_0 按下鼠标右键，会出现连结对话框 (如图-18)。

  — 将 LED_0 的 CATHODE 连接到 CPU PA 的 bit0。

  — 重覆 2.4.1 至 2.4.3 的步骤，将所有的 LED_n 连到 CPU 的 PA 的 bit_n。

  — 按下工具栏上的 mode 钮将 VPM 由 configuration 模式切换到 running 模式。

→ **在 HT-IDE 2000 上，开始进行除错的步骤 (step into 或 go 指令) ，此时 VPM 将会反应程序的输出结果。**

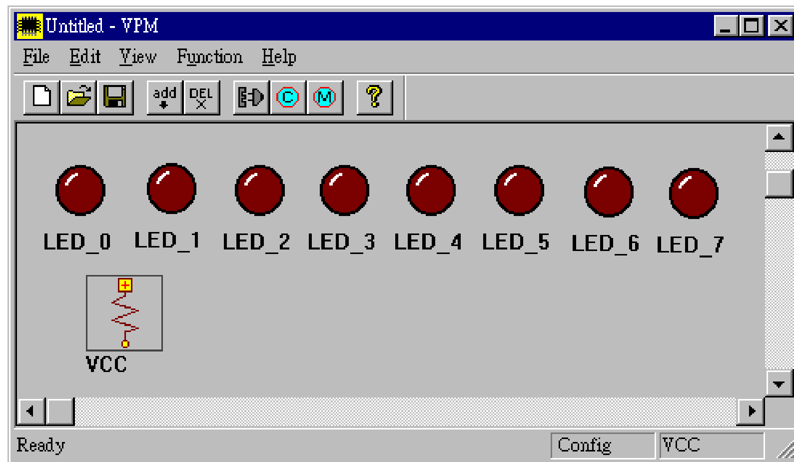


图-17

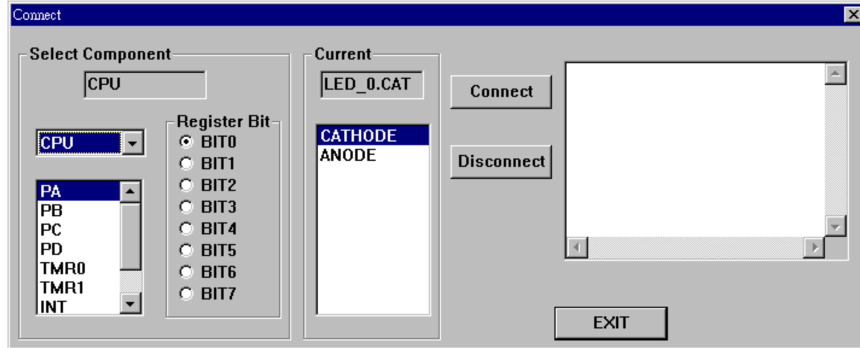


图-18

第 十 章

八位元单片机 C 编译器

10

Holtek C 编译器是仿 ANSI C 标准的编译器，但是受限于本身单片机硬件的结构限制，因此只有部份相容，相信读者能够谅解，本章主要用来说明 Holtek C 编译器及其程序语言的架构和语法。

Holtek C 程序语言的架构

C 的叙述 (Statement)

叙述可以分为宣告 (Declaration)、变量定义 (Variable Setting)、运算式 (Expression) 和函数调用 (Function) 等四类，每个叙述的最后由分号‘;’做为结束符号，并且可以执行以下的行为：

1. 宣告程序变量和程序结构
2. 定义程序空间
3. 执行数学和逻辑运算
4. 执行程序的流程控制

然而程序的进入点是由 main() 这个函数所宣告，也就是说一个程序的来源档 (Source File) 可以很多，但是只能有一个含有 main() 这个函数，而且程序是由 main() 底下的括弧 {} 的内容开始执行的。

```
例: void main()
     {   宣告资料变量;
         叙述;          /*注解*/
     }
```

C 的注解 (Comment)

Holtek C 编译器支持两种注解符号，如下所述：

1. /*……..*/

   凡介于 /* 符号与 */ 符号间的资料或文字，编译器会把它们当成是程序的注解而不予编译。

   例：/* this is a block comment */

2. 双斜线注解 //

   以双斜线//开始之后的本行字元，编译器都以注解视之。

   例： //this is a line comment

标识符 (Identifier)

定义一个变量名称必须由字符 (characters)，数字 (digits) 与底线 (underscore) 等组合而成，不过必须符合下述规则：

1. 第一个字元不可为数字
2. 代名最长只能由 31 个字符组成
3. 大小写是有区别的 (case-sensitive)
4. 不可以使用保留字 (reserved words)

保留字 (Reserved words)

下列为 Holtek C 编译器的保留字，注意要小写

```
auto       bits       break      case       char
const      continue   default    do         else
enum       extern     for        goto       if
int        long       return     short      signed
static     struct     switch     typedef    union
unsigned   void       volatile   while
```

数据类型 (data type)，修饰词 (qualifier) 和大小

char 字符类型，占一个 byte
int 整数类型，占一个 byte
void 无返回值 (函数返回类型)

修饰词	可用数据类型	作用
Const	Any	放数据在 ROM 区
Long	Int	配置一 16 位元整数
Short	Int	配置一 8 位元整数
Signed	Char，int	配置一带符号变量
Unsigned	Char，int	配置一无符号变量

下列为数据类型的大小和范围

Data Type	Size(bits)	Range
Char	8	-128~127
unsigned char	8	0~255
Int	8	-128~127
Unsigned	8	0~255
short int	8	-128~127
unsigned short int	8	0~255
Long	16	-32768~32767
unsigned long	16	0~65536

变量的定义 (declaration)

变量必须经过定义才能使用，其语法如下：

data_type variable_name [，variable_name….];

若变量在一个函数里定义，则此变量为局部变量 (local variable)，即其他函数不可直接存取此变量，只能由定义它的函数所使用，当变量是在所有函数外定义时，则此变量为全局变量 (global variable)，即所有函数都能存取它。

修饰词 const 可以修饰任何全局变量而且指定此变量的值是不可改变的，也就是定义它存放在 ROM 区域。目前局部变量不可以使用 const 。 符号 "@" 可以指定一个变量被配置在 RAM 的某个地址。

例： int lcd @ 0x20 ; /* 变量 lcd 被定义放在 0x20 的地址*/

一个数组 (array) 也可被定义在一特定地址

例：
```
int port[8] @ 0x20; /* array port takes memory location
                       0x20 through 0x27*/
```

除非变量被定义为外部变量 (external variable) 否则编译器预设变量为静态变量 (static variable)，无论静态或外部变量，编译器皆不会设定其初始值。

常数 (constant)

一个常数为任何文数字 (alphanumeric) 或单一字符或字符串 (char string) 。

整数常数 (integer constant)

整数常数被设成 int 类型，长常数 (long constant) 以 l 或 L 表示，无符号常数 (unsigned constant) 以 u 或 U 表示，ul 或 UL 则表示 unsigned long 。

→ 整数常数可以下列型式指定

- 二进制常数：以 0b 或 0B 为首的数字，其他数字只能是 0 或 1
- 八进制常数：以 0 (zero) 为首的数字，有效数字从 0 到 7
- 十六进制常数：以 0x 或 0X 为首的数字，有效数字从 0 到 9，以及 A 到 F 或 a 到 f
- 十进位常数：非以上为首的数字

字符常数 (character constant)

一个字符常数被视为是一个整数的类型，以单引号 ' ' 框住表示之，如 'a'。 每个字符皆有其对应的字符代码，如 A 的字符代码为 65。

字符串常数 (character string constant)

一个字符串是一连串的字符，也就是 char 类型的数组，以双引号 " " 框起来。如 "hello!" 就是一个字符串常数。每个字符占用一个位元组，而字符串最后一个字符为 '\0'。所有字符串都必须以空字符 '\0' 为结束，此字符的值为 0。

列举常数 (enum)

列举集合是另一种整数常数的表示法。如：

```
enum { PORTA，PORTB，PORTC}；
```

就等于下列的定义

```
const PORTA=0；
const PORTB=1；
const PORTC=2；
```

因为 enum 在内部是把列举名称以 0, 1, 2, …，的整数来表示。

运算符 (operator) 与运算表达式 (expression)

运算符是对变量所存数据加以运算的符号，而运算表达式是运算操作符与数据、变量、函数、运算表达式的合法表示式。下面说明几种型态的运算操作符：

数学运算符 (arithmetic operator)

+	加法运算符
−	减法运算符
*	乘法运算符
/	除法运算符
%	余数运算符

注意 余数运算符的运算操作数 (operand) 必须是整数。

比较运算符 (comparison operator)

比较运算操作符是将两个运算操作数做比较，返回比较后的结果为真 (非零) 或是假(零)。

>	大于
>=	大于或等于
<	小于
<=	小于或等于
==	等于
!=	不等于

逻辑运算符 (logical operator)

逻辑运算符是取运算操作数的真假值来做运算，运算结果返回真或假。

&&	逻辑 AND
||	逻辑 OR
!	逻辑 NOT

位操作逻辑运算符 (bitwise logical operator)

提供六种位元运算的逻辑能力。

&	位元 AND
		位元 OR
^	位元 XOR
~	位元取反
>>	向右移位元
<<	向左移位元

设定算符 (assignment operator)

<var>	+=<expr>	变量加上 expr 的值，将结果存入变量
<var>	-=<expr>	变量减去 expr 的值，将结果存入变量
<var>	*=<expr>	变量乘以 expr 的值，将结果存入变量
<var>	/=<expr>	变量除以 expr 的值，将商数存入变量
<var>	%=<expr>	变量除以 expr 的值，将余数存入变量
<var>	&=<expr>	变量与 expr 的值做位元 AND 后再存入变量

<var> |=<expr> 变量与 expr 的值做位元 OR 后再存入变量
<var> ^=<expr> 变量与 expr 的值做位元 XOR 后再存入变量
<var> >>=<expr> 变量向右移 expr 个 bits，将结果存入变量
<var> <<=<expr> 变量向左移 expr 个 bits，将结果存入变量

递增 (increment) 和递减 (decrement) 运算符

++<var> 变量先加 1，再做运算
<var>++ 运算后，变量再加 1
--<var> 变量先减 1，再做运算
<var>-- 运算后，变量再减 1

条件运算符 (conditional operator): ?

<expr> ? <statement1> : <statement2>

若 <expr> 为真，则执行 <statement1> ，否则就执行 <statement2>

运算符的优先性 (priority) 与结合性 (associativity)

当一个运算式中包含有多个运算符的时候，就必须注意运算符的执行顺序，以免造成错误的结果而不自知。另一方面执行的结果也有方向性，称之为结合性。现将所有运算符依优先级的高低列表如下：

运算符	说明	结合性
[]	数组元素	由左到右
()	函数或运算式中的括号	
->	结构成员的间接存取	
sizeof	数据所占 byte 数	
++	增 1	由右到左
--	减 1	
~	取 1 补数	
!	否定	
-	负号	
+	正号	
&	取变量地址	
*	存取指针所指地址的内容	
/	除法运算	由左到右

运算符	说明	结合性
*	乘法运算	
%	余数运算	
+	加法运算	由左到右
-	减法运算	
<<	左移运算	由左到右
>>	右移运算	
<	小于	
<=	小于或等于	
>	大于	
>=	大于或等于	
==	相等	由左到右
!=	不相等	
&	位元 AND	
^	位元 XOR	
|	位元 OR	
&&	逻辑 AND	
||	逻辑 OR	
?:	条件运算	
=	设定	由右到左
*=	相乘再存入变量	
/=	相除再存入变量	
%=	取余数再存入变量	
+=	相加再存入变量	
-=	相减再存入变量	
<<=	左移后再存入变量	
>>=	右移后再存入变量	
&=	位元 AND 后再存入变量	
|=	位元 OR 后再存入变量	
^=	位元 XOR 后再存入变量	
,	分隔变量或算式	由左到右

类型转换 (type conversion)

若运算式中混合不同的数据类型，编译器会自动做类型转换。也可使用强定型别 (type cast) 的方式，如下：

```
(type_name) expression
```

运算式的结果将会被转为 type_name 所指定的类型。

程序流程控制 (program control flow)

程序流程控制主宰程序执行的顺序，下面说明几种方式：

if-else 叙述

语法：

```
if(expression)
    statement1;
[else
    statement2;
]
```

说明：

这是一个条件判断控制式，当条件式 expression 成立时，则执行 statement1，然后略过 **else** 部份 (statement2)，当条件式不成立时，则略过 statement1 ，而执行 statement2。

例：

```
if(word_count>128)
{
        word_count=1;
        line++;
}
else
        word_count++;
```

for 叙述

语法：

for(initial-exp;condition-exp;update-exp)

说明：

initial-exp: 起始值的设定。

condition-exp: 条件判断式，如果条件成立时则执行循环动作否则离开 **for** 循环。执行 **for** 循环后的叙述。

update-exp: 循环动作做完，必须到此做运算，也可以没有任何叙述。

例：

```
for(I=0；I<10；I++)
        a[I]=b[I]; //copy elements from an array to
        another array
```

while 叙述

语法：

while(condition-exp)

    statement;

说明：

while 为先判断再执行的一种条件式循环。即先判断条件是否成立，若成立则执行循环动作，然后再回条件式做下一次判断，直到条件不成立，才离开循环。

例：

```
I=0;
while(b[I]!=0)
{
      a[I]=b[I];
      I++;
}
```

do-while 叙述

语法：

do

    statement;

while(*condition-exp*);

说明：

do-while 是先做动作再判断条件式的循环，所以不论条件式结果为何，至少会执行动作一次。

例：

```
I=0;
do
{
        a[I]=b[I];
        I++;
}while(I<10);
```

goto 叙述

语法：

goto 标名

说明：

goto 是一种强制性的控制流程。使用 **goto** 可以一次跳过数个循环而将程序移转到标名 (label) 的地方。标名必须和 **goto** 在同一个程序内，即不能跳至其它函数里。

switch 叙述

语法：

```
switch(variable)
{
    case    constant1:
                        statement1;
                        break;
    case    constant2:
                        statement2;
                        goto Label;
    default:
                    statement;
Label:
                    statement4;
                    break;
}
```

说明：
switch 的变量值将会拿来和 **case** 后的常数做比较，如果符合则执行该 **case** 后的动作。若不符合，则执行 **default** 后的叙述。在执行 **switch-case** 的动作时，一直遇到 **break** 才会停止。
例：

```
for(I=j=0;I<10;I++)
{
        switch(b[I])
        {
              case 0: goto outloop;
              case 0x20: break;
              default:
                 a[j]=b[I];
                 j++;
                 break;
        }
}
outloop:
```

break 和 continue 叙述

语法：
break;
continue;
说明：
break 是跳出循环的指令，不过一次只能跳出一层循环。
continue 是跳过 **continue** 后的叙述，然后从下一次循环继续执行，并非跳离循环。

函数 (function)

函数就是叙述的集合。要使用函数前，此函数必须定义或宣告，否则编译器会发出警告信号。Holtek C 编译器支持两种函数宣告或定义方式，分别称为古典式和现代式。

古典式

return-type function-name(*arg1,arg2,*…)
var-type arg1;

var-type arg2;

现代式

return-type function-name(*var-type* arg1,*var-type* arg2,...)

在这两种型式中， return-type 是函数传回值的数据类型。如果函数没有传回值，则必须宣告为 void 型别。 function-name 是函数的名称，亦相当于一个全局变量。参数 arg1，arg2 是函数使用的参数，必须指定其数据类型 (var-type)。

函数宣告

古典式：

return-type   function-name(arg1,arg2,…);

现代式：

return-type function-name(var-type arg1,var-type arg2,…)

函数定义

古典形式：

return-type   function-name(arg1,arg2,…)

var-type arg1;

var-type arg2;

{

    statements;

}

现代形式：

return-type   function-name(*var-type* arg1, *var-type* arg2,..)

{

    statements;

}

函数参数的传递

传参数到函数有两种方式

- 传值 (pass by value)

  此方法是将参数的值传给函数内相对应的形式参数(formal parameter)，所以改变函数的形式参数值并不会影响到原来的调用时的变量值。

- 传地址 (pass by reference)

  此方法为参数的地址被复制给函数形式参数。因此在函数里若改变此形式参数的值，亦同时改变原调用的变量值。

函数返回值 (return value from functions)

return 是用来使函数结束并返回原调用程序的指令。除了返回原调用调用程序还可将函数的数据返回给调用程序。

指针和数组 (pointer and array)

指针

指针是一个指向另一个变量地址的变量。

语法：

data-type *var_name;

data-type 是指针指向的变量的数据类型。var_name 则是指向变量的指位器 (pointer variable) 。 *是告诉编译器 var_name 是一个指针。

指针常用到的两种运算子为 * 和 & 。

→ *的用法

主要用来存取指位器所指的变量值。

例：

```
*px = 5;
x = *px;
```

上例是把 5 存入 px 所指的变量内，然后再取 px 所指变量的值放入 x，所以 x 会等于 5。

→ & 的用法

例：

```
int x,*px;
px=&x;
```

上例是以&取得 x 变量的地址，然后存入 px 指位器。& 只能对变量做运算。

数组

数组是由相同类型的变量排列而成。

例：

char a[30];

char 为数组元素的类型 ，a 为数组名，30 为元素个数。

结构体 (structure) 和等位 (union)

结构

语法：

struct struct-name
{
    data-type member1;
    data-type member2;
    ……
    data-type membern;
}[*variable-list*];

说明：

结构是一个变量或多个变量的集合，这些变量可以有不同的数据类型，可以将之整合在一个结构名称下使用之。并称此种复合的数据类型为一个结构。

结构本身是一种类型而不是变量，所以结构的宣告是建立一个新的类型，而不是宣告变量。

例：

```
struct person_id
{
        char id_num[6];
        char name[3];
        unsigned long birth_date;
}mark;
```

等位 (union)

语法：

union union-name
{
    data-type member1;

data-type member2;
…….
data-type membern;
}[*variable-list*];
说明：
等位和结构的宣告或定义，基本上是相同的。唯一不同的是其记忆体空间的分配方式。也就是利用一段共用的空间来存放不同数据类型的变量。为了容纳不同类型的资料，我们必须宣告可能存入的类型，而等位的大小刚好是这些型别中占最大 byte 数的类型长度。
例：

```
union common_area
{
        char name[3];
        int id;
        long   date;
}cdata;
```

上例中, 等位 common_area 的大小是 3 个 bytes

前置处理指令 (preprocessor directive)

前置处理指令是以 # 字为字首的文字串，程序中所有前置处理指令的叙述会最先被前置处器 (preprocessor) 处理，前置处理器主要有三大功能：宏指令 (macro)、含括文件 (include file)、条件式编译 (conditional compiler)。
底下说明所支持的前置处理指令：

#define

语法：
#define name replaced-text
#define name[(parameter-list)]replaced-text
说明：
#define 是定义字串常数，在程序被执行前，前置处理器就已经将之安放在原始文件适当的位置。其目的是提高程序的可读性而且维护容易。

例：
```
#define TOTAL_COUNT		40
#define USERNAME		"Henry"
```

#error

语法：

#error "message-string"

说明：

#error 命令编译器将除错讯息显示出。

例：
```
#if	TOTAL_COUNT>100
#error	"Too many count."
#endif
```

条件式前置处理指令

#if #else #endif

语法：

#if *expression*
    source codes
[**#else**
    source codes]
#endif

说明：

条件式前置处理命令是让编译器有条件地编译程序，也就是说如果 **#if** 后的运算式为真，就编译其后的程序码。 **#else** 的部份可以有，也可以没有，视情况而定。如果有 **#else** 部份，则当 **#if** 的运算式不为真的情况时， **#if** 之后一直到 **#else** 之前的叙述会被省略，不编译。而将 #else 之后到 **#endif** 之前的叙述编译。

例：
```
#define MODE 2
#if MODE>0
	#define DISP_MODE MODE
#else
	#define	DISP_MODE 7
#endif
```

#ifdef

语法：

#ifdef *symbol*
source codes
[**#else**
source codes]
#endif

说明：

这个前置处理命令有点类似 #if 。不同的是它检查指定的符号 symbol 是否之前已被定义。如果有定义，则编译后面的程序码。

例：

```
#ifdef DEBUG_MODE
#define TOTAL_COUNT 100
#endif
```

#ifndef

语法：

#ifndef *symbol*
source codes
[**#else**
source codes]
#endif

说明：

这个前置处理命令刚好跟 **#ifdef** 相反动作。

例：

```
#ifndef DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

#elif

语法：

#if *expression1*
source codes
#elif *expression2*
source codes
[**#else**
source codes]
#endif

说明：

#elif 前置处理命令必须与 **#if** 同时存在。如果 *expression1* 的结果为真，则在 **#if** 之后到 **#elif** 之前的程序会被编译。如果 *expression1* 的结果为假，则再检查 *expression2* 看是否为真，若为真，则 **#elif** 之后到 **#else** 或 **#endif** 之前的程序会被编译。

例：

```
#if        MODE==1
#define    DISP_MODE 1
#elif    MODE==2
#define        DISP_MODE 7
#endif
```

defined

语法：

#if **defined** *symbol*

source codes

[**#else**

source codes]

#endif

说明：

defined 可以用在 **#if** 或 **#elif** 。 如下列写法

#ifdef symbol 和 **#if defined** symbol 是相同。

例：

```
#if defined    DEBUG_MODE
#define    TOTAL_COUNT    50
#endif
```

#undef

语法：

#undef symbol

说明：

如果 symbol 之前已被定义，可用 **#undef** 将之取消定义。

例：
#define TOTAL_COUNT 100
……………….
#undef TOTAL_COUNT
#define TOTAL_COUNT 50

文件含括 #include

语法：
#include *<file-name>*
or
#include "*file-name*"
说明：
#include 用来把指定的文件含括到主程序里。
有两种使用格式：
#include "档名"：指示编译器到工作目录找寻文件，若找不到则到目前的目录寻找。
#include <文件>：只到环境参数所设定的目录去寻找。
例：
#include <ht48c10.inc>
#include "ht8270.inc"

内嵌汇编语言 (inline assembly)

#asm #endasm
语法：
#asm
<[*label:*]*opcode*[*operands*]>;
……
#endasm
说明：
#asm，**#endasm** 为线上汇编语言前置处理命令，在 **#asm** 之后到 **#endasm** 之前，可以直接写入汇编语言格式 (汇编语言的指令须要用大写) 。

例：

```
#asm
    AND A,0FH
    SUB A,09H
    SZ  C
    ADD A,40H-30H-9
    ADD A,30H+9
#endasm
```

#pragma

语法:

#pragma *vector symbol @ address*

说明：

此前置处理指令可以定义向量地址，如果定义的 symbol 是程序中某个函数的名字，则当发生复位或中断时，会跳往相对应的函数执行。 address 可以定义为单片机的向量地址。

例:

```
#pragma vector _RESET @ 0x0000
#pragma vector _INT @  0x0004
void _RESET(void){
….
}
void _INT(void){
…….
}
```

编译器预先定义的前置处理符号

下列的特殊字已经被 Holtek C compiler 定义为特殊之用，在编译时将被展开为指定的数值或字串，以取代之。

__LINE__	正在处理的程序的行号，以十进位的数值取代。
__FILE__	目前正被处理的来源档名，以档名的字串取代。
__DATE__	前置处理器处理文件的日期。以日期的字串取代， 其格式为 "Mmm dd yyyy"，例如：Jul 21 1998
__TIME__	前置处理器处理来源档的时间。以时间的字串取代。

其格式为 "hh mm ss"， 例如：11 23 50

__STDC__ 前置处理器使用数值 1 取代此特殊字

Holtek C 编译器提供的专属功能

C 编译器提供以下的专属功能，以便支持 Holtek 单片机发挥其特有的功能：

1. 使用多个来源档 (multiple source files)
2. I/O 口的系统调用 (Input/Output ports system calls)
3. 重复位和中断 (Reset and Interrupts)

多个源文件 (multiple source files)

C 编译器支持多个源文件，但是其中只能有一个含有 main() 函数。编译器会将各源文件编译成目的文件 (object file)，再将所有的目的档联结 (link) 成一个可以被执行的执行文件 (executable file)。

I/O 口的系统调用 (system calls)

对于基本输入输出口，C 编译器提供一些基本的系统调用函数，这些系统调用并不占用 stack。

→ 基本输入/输出口函数

- unsigned char peekPX()

  从 port 读回资料，其中 X=A,B,C,D,E,F,G

  例：

  ```
  unsigned char i;
  i = peekPA();   //从 port A 读入外部状态，并存入变量 i
  ```

- void pokePX(unsigned char)

  将数据写到 port X，X=A,B,C,D,E,F,G

  例：

  ```
  pokePC(0x00); //将 0 写到 port C
  ```

→ **读写控制寄存器 (control registers)**

- unsigned char peekPXC();

  从控制寄存器读回数据,其中 X=A,B,C,D,E,F,G

  例：

  ```
  unsigned char i;
  ```

i = peekPEC(); //从控制寄存器 PEC 读回数据并存入变量 I

●void pokePXC(unsigned char)

写数据到控制寄存器，其中 X=A,B,C,D,E,F,G

例:

pokePBC(0x20); //将 0x20 写至 port B 的控制寄存器

→ 设定/清除 端口的位元

●void setPX()

设定 port X 所有的位元为 1，其中 X=A,B,C,D,E,F,G

例:

setPD(); // 设定 port D

●void setPXi()

设定 port X 的位元 i 为 1，其中 I=0,1,2,3,4,5,6,7 。X＝A,B,C,D,E,F,G

例:

setPD3(); // 设定 port D 位元 3

●void clrPX()

清除 port X 的所有位元为 0，其中 X=A,B,C,D,E,F,G

例：

clrPC(); // 清除 port C

●void clrPXi()

清除 port X 的位元 i 为 0，其中 i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G

例：

clrPB7(); // 清除 port B 的位元 7

→ 设定/清除 控制寄存器口的位元

●void setPXC()

设定 port X 控制寄存器，其中 X=A,B,C,D,E,F,G

例：

setPDC(); // 设定 port D 控制寄存器

●void setPXCi()

设定 port X 控制寄存器的位元 I，其中 i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G

例：

setPEC3(); // 设定 port E 控制寄存器的位元 3

● void clrPXC()

清除 port X 控制寄存器，其中 X=A,B,C,D,E,F,G

例:

```
clrPCC();   // 清除 port C 控制寄存器
```

● void clrPXCi()

清除 port X 控制寄存器位元 I，其中 i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G

例：

```
clrPBC7();   // 清除 port B 控制寄存器的位元 7
```

复位(reset) 和中断 (interrupt)

透过 #pragma 前置处理指令，可以建立复位向量和中断向量，只要函数名称和定义的向量名称相同，程序会自动找到相对应的进入点。

例：

```
#pragma vector _RESET @ 0x00
#pragma vector _INT @ 0x0004
void _RESET(void){
…..
}
void _INT(void){
…..
}
```

Holtek C 与 ANSI C 编译器的差异

保留字 (reserved words)

Holtek C compiler 不支持下列保留字和修饰词

关键字: float double

修饰词: auto register static

变量

所有的变量皆存在 RAM 中。 "@"可以用来指定该变量存在 RAM 中的地址。

语法为：

data_type *variable_name @ memory_location*

例如：

```
unsigned char flag @ 0x25; /*宣告 flag 在 0x25 记忆体地址*/
```

常数

Holtek C 编译器支持二进制位常数。凡是以 0b 或 0B 为开头的数值将被视为二进制位常数。

例如：

```
0b101 = 5
0b1110 = 14
```

函数

不支持重覆进入 (reentrant) 与循环调用 (recursive code) 。

例如：

```
void foo(){
        foo();
    }
```

阵列

Holtek C 编译器只支持一维数组 (one dimension array) ，一个数组会被配置在一块连续的记忆体上，其长度最大为 256 位元组 (bytes)。

常数变量 (constant variables)

常数变量，必须宣告为全局变量 (global variable) 且需要设定初始值 (initial value) 。其地址是被配置在 ROM 上的。所有的常数变量的大小不可超过 255 位元组。

例如：

```
const int    val ;         /* 错误，必须设定初始值 */
const int    new =1 ;   /* 正确 */
const char    array[]= "abcde" ;     /* 正确 */
```

初始值 (initial value)

全局变量 (global variable) 不可设定初始值，局部变量 (local variable) 则无此限制，常数变量必须设定初始值。

乘/除/余数

Holtek 单片机不提供硬体的乘、除、余数 (*, /, %) 运算，因此 C 编译器是采用系统函数来实作。当使用者需要用到这些运算时，必须要连结 math.lib 函数库。连结 math.lib 的方法：在主选单上选 [Options]，再选 [project…]，将 math.lib 写入 [libraries] 栏位。

堆栈 (stack)

因为 HT48CX0 单片机只有 2 到 8 层堆栈，程序设计者需要考虑函数的使用，以避免堆栈溢出 (overflow)。

运算子 / 函数	所需堆栈
main()	0
*	1
/	1
%	1
peekPX(), X=A,B,C,D,E,F,G	0
peekPXC(), X=A,B,C,D,E,F,G	0
pokePX(), X=A,B,C,D,E,F,G	0
pokePXC(), X=A,B,C,D,E,F,G	0
setPX(), X=A,B,C,D,E,F,G	0
setPXC(), X=A,B,C,D,E,F,G	0
setPXi(), i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G	0
setPXCi(), i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G	0
clrPX(), X=A,B,C,D,E,F,G	0
clrPXC(), X=A,B,C,D,E,F,G	0
clrPXi(), i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G	0
clrPXCi(), i=0,1,2,3,4,5,6,7 X=A,B,C,D,E,F,G	0
数组	1

Holtek C 编译器

Holtek C 编译器是采用 little-endian 的编译器，假设变量 v 是配置在 0×20 地址的两个位元组的长变量，其值为 0×1234 则安置方式为：

0×1f	
0×20	0×34
0×21	0×12
0×22	

命令列参数 (command line arguments)

- /? 或 /h
  显示使用讯息
- /errlog=log_file_name
  将编译过程的错误讯息写入 log_file_name 文件
- /nologo
  不显示编译器的版本讯息

中英对照表

英文	中译
Alphanumeric	文数字
Comment	注解
Constant	常数
Declaration	宣告
Definition	定义
Enumeration	列举
Expression	运算式
external variable	外部变量
Function	函数
global variable	全局变量
Identifier	代名
local variable	局部变量
NULL char '\0'	空字元

英文	中译
Operator	运算操作数
Pointer	指针
preprocessor directives	前置处理指令
Qualifier	修饰词
Statement	叙述
static variable	静态变量
type cast	强定类型
Union	等位
Variable	变量

第 十一 章

编译、连结软件 (Cross-Tools)

11

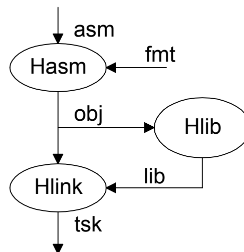
本章的目的在告诉使用者 cross-tools 的功能以及如何使用 cross-tools 以节省开发时间和人力。首先让我们回顾一下 cross-tools 的发展过程。

在 Holtek 以前的开发系统中，cross-tools 的部分都只含有 cross-assembler。所以使用者必需在一个文件中完成所有的功能。这一方面是由于程序空间 (ROM) 本来就不大，并不需要分为许多个程序来完成；另一方面也是为了节省成本，使用者必需在有限的程序空间中完成所有的功能，所以程序的结构化便不是很好，使用者很难把他自己的程序再细分成许多个文件。

在 Holtek 八位单片机问世之后，由于程序空间加大了许多 (HT48C70 的有 8K，而八位单片机的结构更可以达到 64K)，可以处理的功能也愈来愈多，程序的复杂度也提高了，而为了加速完成项目并抢市场，便希望由许多人同时进行一个项目，最后再由软件把所有的程序段都连结在一起。因此我们提供 linker，那么一个项目便可以依功能划分并交由许多人同时设计，最后再把各个程序连结起来即可。

如此一来，便可以加速项目的开发，而如果有另一个项目需用到相同的功能，则这一部分的程序码就可以重复使用，节省人力。

在目前 Holtek 八位元开发系统的 cross-tools 部分即提供了 assembler，linker 以及 library 管理程序三种软件。Assembler 读入使用者所键入的原始文件，产生 OBJ 文件；Linker 则是将许多的 OBJ 文件以及 LIB 文件连结在一起并产生 TSK 文件；Library 管理程序则是用来管理 OBJ 文件。整个流程如下：



节区 (section) 的概念

在 cross-tools 中是把整个项目看成是一段段的程序段的总和，每一个程序段就是一个 section。每个 section 都有参数来指明这个 section 将要放到程序空间中的那一个位置或是如何和其他 section 组合在一起。Section 的语法如下：

SectionName .section [alignment] [combination] 'ClassType'

其中

SectionName 这个 section 的名称

Alignment	指明这个 section 要放到程序空间中的那一个位置 byte，该 section 可以放到程序空间中的任一位置 word，该 section 必需放在程序空间中的偶数位置 para，该 section 必需放在程序空间中地址可以被 16 整除的位置 page，该 section 必需放在程序空间中地址可以被 256 整除的位置
combination	说明这个 section 将要如何和其他 section 组合在一起 at addr，该 section 必需被放在 addr 的位置 common，该 section 可以和其他 section 重叠

ClassType　　这个 section 是定义 ROM 或是 RAM 中的数据
　　　　　　code，该 section 为 ROM 数据
　　　　　　data，该 section 为 RAM 数据

在 section 的所有参数中，SectionName 以及 ClassType 是一定要指明的，至於像 alignment 及 combination 这两个参数则是可有可无。如果不特别指明这两个参数的话，那么这个 section 将有可能被放在程序空间中的任何位置。以下面的程序为例：

```
Func    .section   'code'
ToAscii:
        and     a, 0fh
        sub     a, 09h
        sz      c
        add     a, 40h - 30h - 9
        add     a, 30h + 9
        ret
```

在这个例子中定义一个 section，名称是 Func。他的 ClassType 是 code，表示这个 section 定义的数据是要被放到程序空间中。这个 section 中含有一个函数 ToAscii，由于这个 ToAscii 并没有用到特别的指令也不需要将他放到特别的程序空间位置中，因此 alignment 和 combination 这两个参数并没有用到。

程序节区 (Program Section)

对於一般的程序而言，如果并不限定必需放在程序空间中的特定位置，则只要指明 SectionName 以及 ClassType 即可，linker 会把这个 section 安置到程序空间中的某一位置。然而在某些情况下，我们有必要把程序段放置在程序空间中的特定位置。一个明显的例子就是：在 Holtek 八位单片机中有许多位置是有特别意义的，例如：执行点是在程序空间中的位置 0，而外部中断是在位置 4。在以前如果我们必需在特定的程序空间位置放入特定的程序码时，便要用到 org 伪指令，就像以下的例子：

```
org     0
jmp     start
org     4
; external    int
reti
```

```
start:
       clr    pac
       clr    pbc
       …
       …
```

其中第一行指明程序的进入点，所以 jmp start 这个指令会被放入程序空间中位址 0 的位置；第三行指明外部中断的进入点，reti 这个指令会被放入程序空间中位址 4 的位置。把以上的程序改写成一段 section 的方式如下：

```
main   .section  at  0  'code'
org  0
jmp  start
org  4
;external int
reti
start:
       clr  pac
       clr  pbc
       …
       …
```

与原先的程序相比，我们增加了第一行 section 伪指令，并且把这个 section 的 combination 参数设为 at 0，指明这个 section 是要被放到程序空间的位址 0。在 linker 连结所有的 obj 文件时，会自动把这个 section 放到正确的位置。

在加入 section 伪指令之后，org 的意义也要做一点修改：在以前 org 都是相对於 ROM 的起始位置，而现在 org 则是相对於该 section 的起始位置。

combination 参数除了上面所提到的绝对地址之外，还有 common。使用到 common 这个参数而且是相同 section 名称、ClassType 的 section 会被重叠放到相同的位置。这个参数的原本用意是让使用者可以定义一个以上的程序存储区块，或是定义 RAM mapping 的 LCD 数据，可是使用者也可以使用 equ 的方式定义这类的数据，所以在此就不多做解释了。

section 语法的使用上还有一点要注意的。Holtek 单片机的查表指令可以查目前页以及最后一页两种，一页有 256 个字节。如果要查最后一页的数据时，这些数据可以用前述的 at addr 参数将其放置到最后一页中。而如果要目前页的数据，则指令 tabrdc 必需和所要查的数据在同一页中。如下例：

```
add    a, offset Table
mov    tblp, a
```

```
tabrdc LowByte
…
…
Table: dc 0123H, 4567H
```

其中第三行 tabrdc LowByte 必需和第六行的数据 1234H，4567H 在同一页中。在这一页中所有被 tblp 指到的位置都可以被 tabrdc 指令读入，也就是说所有在这一页的数据都有可能是要查的数据，因此 cross-tools 并没有办法区别哪一部分是要查的数据，而哪一部分是要执行的指令；更何况要执行的指令也可以是要查的数据。像这种情形只有使用者了解哪一个是指令，哪一个是数据。使用者可以透过 Cross-tools 提供 alignment 参数决定查表指令和数据放置的位址。alignment 参数共有 byte，word，para，page 四种值。如果某一个 section 的 alignment 参数是 page，那么这个 section 将会自程序空间中的某一页的起始位置放起。改写上面的例子：

```
Func    .section   page   'code'
lookup proc
        add     a, offset Table
        mov     tblp, a
        tabrdc LowByte
        …
        …
    Table: dc 0123H, 4567H
lookup endp
```

最后 cross-tools 会把这一段程序放入以页起始的位置，如下：

X00	add a, offset Table
X01	mov tblp, a
X02	tabrdc LowByte
	...
	...
X14	0123
X15	4567

这样，使用者更能掌握函数的存放方式，当然也更能确定程序执行的正确性。这种方式并不是十分完美的，毕竟在写完这个函数之后使用者还要去了解查表指令的数据是否超过范围。

alignment 参数主要是用在这类情况。除了 page 之后，alignment 参数还可以指定 byte，word，para 等。这个参数的内定值是 byte，所以如果省略了这个参数，那么 cross-tools 可以将这个程序段安置於程序空间中的任一位址。

数据节区 (Data Section)

以前使用 RAM 的数据都是直接定义的，如下例：

```
Count  equ    40H
       …
       …
       mov    a, [Count]
       …
```

对於多人共同开发一个项目的时候，这种方式需要每个人都协调出所需要的程序空间位置，而且开发完成的程序码也没有办法留待以后再使用。因此希望每个人各别定义他所要用到的程序空间位置，最后再由 cross-tools 来安排所有使用到的程序空间。

在定义程序空间时，使用到 DB，DW，DBIT 三个伪指令。

```
1 Data  .section     'data'
2 Count    DB  ?
3 Flag     DBIT
4 Buffer   DW  ?
5       …
6 Func  .section 'code'
7          mov    a, 1
8          addm   a, Count
9          set    Flag
10         mov    a, offset Buffer
11         mov    mp0, a
12         mov    a, r0
           …
           …
```

第 2 行定义 Count，一个字节的数据；第 3 行定义 Flag，一个位的数据；第 4 行定义 Buffer，这是两个字节的数据。至於在使用这些数据上，在第 7，8 行把 Count 这个变量加一，在第 9 行设定 Flag 这个变量 ；由于目前并没有直接对 word 这一类数据进行处理的指令，因此用 DW 定义出来的变量可以做为 array 使用，第 10 到 12 行取得 Buffer 这个双字节变量中的第一个字节的内容。

在定义数据 section 的时候可以使用 DBIT，DB，DW 这三种伪指令，而在定义程序 section 时可以使用 DC 或是 DW：

```
Data       .section   'data'
Buffer     DW  4 dup  (?)
           …
           …
Func       .section   'code'
           …
Table: DC  0123H,  4567H
       DW 89abH
       …
       …
```

上例在程序 section 定义了 3 个数据，0123H，4567H，89abH。对於 cross-tools 而言使用 DC 或 DW 定义出程序 section 的数据都是一样的。但在此建议使用者在定义程序 section 的数据时，使用 DC 伪指令。使用 DW 定义数据 section 的数据。

数据共享的方式 **Public** 和 **Extern**

在 Holtek 汇编语言中，同一个文件所定义的所有变量以及函数都可在这个文件的任何一个位置参考到，也就是说：所有变量以及函数对於这个文件而言是全局的。然而其他文件如果要使用到的话，必需使用 public 以及 extern 这两个伪指令来说明。在这个文件所定义的数据，不论是标名或是字节、位、双字节等。若要允许其他文件也可以使用的话，就要用 public 伪指令说明为全局变量，public 的语法如下：

```
public  name [,name]
```

目前只有标名(label)、位、字节、双字节这四种数据可以说明为全局变量。对程序 section 而言，只有函数(也就是标名)可以说明为全局；对於数据 section 而言，用 DBIT，DB，DW 定义出来的位、字节、双字节数据都可以说明为全局的。如下例：

```
public Count, Flag, Array
Data   .section 'data'
Flag   dbit
```

```
Count   db ?
Array   dw ?
        …
        …
public ToAscii
Func    .section  'code'
ToAscii:
        and a, 0fh
        …
        …
```

第 3 行定义 Flag 这个位变量，第 4 行定义 Count 这个字节变量，第 5 行定义 Array 这个双字节变量，在第 10 行定义一个函数 ToAscii。同时在第 1 行说明 Flag，Count，Array 这三个变量是全局的；第 8 行说明 ToAscii 这个函数是全局的函数。在定义变量的文件中说明变量为全局变量后，其他文件就可以用 extern 说明所使用的变量是定义在其他文件，然后就可以使用到这些变量了。extern 的语法如下：

```
        extern name : type [, name : type]
```

相对应於标名、位、字节、双字节等数据，extern 所说明的变量类型可以是 near，bit，byte，word 等。以下面的程序段为例，以下的程序段说明如何运用前面所定义到的全局变量。

```
1   extern Flag : bit, Count : byte, Array : word
2   extern ToAscii : near
3   code    .section   'code'
4           mov     a, 1
5           addm    a, Count
6           set     Flag
7           mov     a, offset Array
8           mov     mp0, a
9           mov     a, r0
10          mov     a, Value
11          call    ToAscii
            ……
            ……
```

在第 1 和第 2 行说明前段程序所定义的变量，Flag，Count，Array，ToAscii 等。在第 4，5 行把 Count 加一，第 6 行设定变量 Flag，第 7，8，9 行读取 Array 这个变量的第 0 位，第 10，11 行则是调用 ToAscii 函数 。

要注意在说明某一个变量是外界定义时一定要正确指明这个变量的数据类型，否则 cross-tools 将无法正确连结到数据。

宏、条件编译、打印文件以及其他功能

宏、条件编译 以及打印功能等等是 cross-tools 提供的诸多功能之一，也是使用者常常会用到的。分别说明如下：

宏 (Macro)

宏定义一段程序以完成一个功能，程序中调用宏，在编译的时候 cross-tools 会把这一段宏展开出来。由于在程序中可能不只一次呼叫同一个宏，如果宏中含有标名的话，那么这个标名将会重复定义某多次，所以在宏中会指明这个宏所使用到的标名，cross-tools 也会对这个标名做特别处理。

宏的语法如下：

```
    name    MACRO  param1, param2
            LOCAL  loc1, loc2
            …
            …
            ENDM
```

name 是宏的名称，在保留字 **macro** 之后是这个宏的参数，保留字 **local** 定义出这个宏所使用到的标名；在保留字 **local** 之后一直到保留字 **endm** 之前是宏的程序部分。

看以下例子：

```
1   #include ht48c10.inc
2   .listmacro
3
4   max     MACRO  n1, n2
5           LOCAL  loc1, end_max
6           mov a, n1
7           sub a, n2
8           sz c
9           jmp loc1          ;carry if n1 > n2
10          mov a, n2
11          jmp end_max
12   loc1:
13          mov a, n1
14  end_max:
15          ENDM
16
17          max 3, 9
18          end
```

这个例子定义了一个宏 max，有二个参数，n1 和 n2。其功能是用来找出这两个参数中的最大值。这个宏使用到两个标名，loc1 和 end_max。在编译之后整个宏会被插入在第 17 行的位置。

以下是编译后的打印文件：

```
File: macro.asm Holtek  Cross-Assembler Version 1.44 Page 1
    0000      #include ht48c10.inc
    0000      .listmacro
    0000
    0000      max    MACRO  n1, n2
    0000             LOCAL  loc1, end_max
    0000             mov a, n1
    0000             sub a, n2
    0000             sz c
    0000          jmp loc1          ;carry if n1>n2
    0000          mov a, n2
    0000             jmp end_max
    0000   loc1:
    0000      mov a, n1
    0000      end_max:
    0000         ENDM
    0000
    0000      max 3, 9
    0000  0F03  1   mov a, 3
    0001  0A09  1   sub a, 9
    0002  3C0A  1   sz [0ah].0
    0003  2800  R1   jmp ??0000 ;carry if n1>n2
    0004  0F09  1   mov   a, 9
    0005  2800  R1  jmp ??0001
    0006       1 ??0000:
    0006  0F03  1   mov a, 3
    0007     1   ??0001:
    0007       end
        0 Errors
```

条件编译

条件编译是使用者用来控制 cross-assembler 哪一部分应该编译，哪一部分不必编译的一种方式。有以下伪指令：

if
ife
ifdef
ifndef
else

endif

if 是依其后的式子的结果是否为零来决定编译的部分：

```
if 0
   mov a, 3
else
   mov a, 4
endif
```

上面的例子会编译第 4 行，也就是 a 的值会是 4。条件编译可以用来协助侦错。

```
#define   debug
…
…
ifdef  debug
       mov    a, Value
       mov    pd, a
endif
```

像上面的例子，在需要侦错的时候，就用第一行定义出 debug 这个名称，然后在执行的时候，Value 的值就会被显示在 pd。然而当程序开发完成后，只要把定义 debug 这个名称取消即可。

打印文件

打印文件是由 cross-tools 中的 cross-assembler 所产生。在 HT-IDE 系统中可以选择 Options 的 Project 的 Generate listing file 以产生打印文件。

cross-tools 提供了许多伪指令来控制列印表的输出方式。page 这个伪指令是用来控制打印文件中每一页的行数，如下例：

```
       page   55
```

在打印文件中每 55 行将会跳页。内定值是 60，也就是打印文件每 60 行会跳页。

在打印文件中，使用者可以用以下的参数来决定要打印出来的指令部分：

.list

.nolist

.listmacro

.nolistmacro

.listinclude

.nolistinclude

在所有**.list** 以下的指令都会被列入打印文件，而**.nolist** 以下的指令则不会被列入打印文件。**.listmacro** 和**.nolistmacro** 是用来控制在打印文件中是否要展开 macro 指令，在**.listmacro** 伪指令之后的所有的 macro 都会在打印文件中展开，而**.nolistmacro** 以下的所有 macro 都将不展开。同理，.listinclude 和**.nolistinclude** 是用来控制 include 文件是否要出现在文件中，在**.listinclude** 以下的含入文件中的指令都会被列入打印文件中，而在.nolistinclude 以下的含入文件中的指令都将不会出现在打印文件中。

对於说明为 public 的变量或是位置，使用者可以藉由 map 文件来查看最后 cross-tools 将他放到什么位置去了。MAP 文件是由 cross-linker 所产生的，在

HT-IDE 系统中使用者可以由 Option 的 Project 的 Generate map file 来产生。

其他功能

cross-tools 提供包含文件以及 message 功能。使用者可以把各个文件都会用到的固定值部分定义到一个文件中，同时在各个使用到这些固定值的文件中包含到这个文件即可。包含文件的使用方式如下：

include file_name

message 这项功能主要是定义出一个字符串，然后在 cross-assembler 编译到这个位置的时候，它就显示这个字符串。message 的语法如下：

Message 'show this message'

注意事项及建议

目前的 cross-tools 虽然提供了不少功能，对使用者而言也更方便了，但还有许多事情是可以再加强的。比如提供暂时变量，改善 table lookup 的侦测等。

在使用查表功能时，如果要查当前页的数据，则该 section 的 alignment 参数要注明是 page，然后由打印文件注意查表指令和要查的数据是否在同一页中。如果要查的数据是在最后一页，则要注意数据是放於该页中的。

在 Holtek 八位单片机的结构上，可以寻址到 32 个数据存储区块，由 0 到 31。HT48C10、HT48C30、HT48C50、HT48C70 等等都只有一个数据存储区块。如果有多於两个数据存储区块时，那么数据传送指令只能移动第 0 个区块的数据，其他由第 1 到第 31 个记忆体区块的数据都必需通过存储器指针寄存器 mp1 间接寻址的方式取得(在数据存储区中的 2h、3h)。存储器指针寄存器 mp0(在程序存储区的地址 0h、1h)只能指向第 0 个区块。

对於提供暂时变量而言，由于 Holtek 8-bit μC 的结构十分有弹性，因此如果要提供暂时变量，那么 cross-tools 要做许多的分析工作，才可能让各个函数间能有效地共用暂时变量而又不致浪费有限的 RAM。但是在目前 cross-tools 尚未提供暂时变量时，使用者还是可以运用程序的处理技巧以达到函数间共用暂时变量的。

虽然目前的 cross-tools 并未提供暂时变量，函数所使用到的暂时变量在函数结束时便无法释放出来供其他函数使用。但是我们也提供了一些伪指令以便在未来的版本中加入此功能。例如使用者可以用 proc 以及 endp 包含一个完整的函数，改写本章一开始的 ToAscii 函数如下：

```
1   Func    .section  'code'
2   ToAscii proc
3           and    a, 0fh
4           sub    a, 09h
5           sz     c
6           add    a, 40h - 30h - 9
7           add    a, 30h + 9
8           ret
9   ToAscii endp
```

以后 cross-tools 就可以将 proc 及 endp 所包含起来的区段视为一个完整的函数，提供此函数暂时变量并对此函数作较为严谨的检查。同时使用者也应尽量把程序分为许多完整而独立的函数的总合。

在目前，如果需要使用到暂时变量时，可以使用间接暂存器，并彷照 stack 的方式来存取暂时变量。以下程序计算一个串列的总和就是用到这个观念。

```
1   include ht48c30.inc
2   data  .section  'data'
3   TmpAcc db  ?
4   TmpPtr db  ?
5   TmpBuf db  20 dup(?)
6   Times  db  ?
7
8   main   .section  at 0  'code'
9          org 0
10         jmp begin
11         org 4
12         reti
13  begin:
14         mov a, offset TmpBuf - 1
15         mov TmpPtr, a
16  again:
17         mov a, 0
18         mov Times, a
19         mov a, 10
20         call   Series
21         jmp again
22
23  Func   .section 'code'
24  Series proc
25         inc    Times
26         mov    TmpAcc, a
27         mov    a, TmpPtr
28         mov    mp0, a
29         inc    mp0
30         inc    TmpPtr
31         mov    a, TmpAcc
32         mov    r0, a
33         dec    acc
34         sz     acc
35         jmp    Series
36  Adding:
37         add    a, r0
38         dec    mp0
39         dec    TmpPtr
40         sdz    Times
41         jmp    Adding
42         ret
```

```
43  Series endp
44              end
```

你是否注意到以上的 Series 函数其实就是一个 recursive 函数。事实上，你也可以把你想 jmp 的位址直接移入 pcl 并使用 RAM 来记录该要 return 的位址。但是这种方式的限制是所有要 jmp 的位址是在同一页中。