Mini-C 使用手册

*第 **1.03** 版*

***2023** 年 **2** 月 **8** 日*

6F-6, No.1, Sec. 3, Gongdao 5th Rd., Hsinchu City 30069, Taiwan, R.O.C.

TEL: 886-3-572-8688 www.padauk.com.tw

重要声明

应广科技保留权利在任何时候变更或终止产品，建议客户在使用或下单前与应广科技或代理商联系以取得最新、最正确的产品信息。

应广科技不担保本产品适用于保障生命安全或紧急安全的应用，应广科技不为此类应用产品承担任何责任。关键应用产品包括，但不仅限于可能涉及的潜在风险之死亡、人身伤害、火灾或严重财产损失。

应广科技不承担任何责任来自于因客户的产品设计所造成的任何损失。在应广科技所保障的规格范围内，客户应设计和验证他们的产品。为了尽量减少风险，客户设计产品时，应保留适当的产品工作范围安全保障。

提供本文档的中文简体版是为了便于了解，请勿忽视文中英文的部份，因为其中提供有关产品性能以及产品使用的有用信息，应广科技暨代理商对于文中可能存在的差错不承担任何责任。

目　录

修订历史:

修 订	日 期	作 者	描 述
V1.00	2013/04/10	Kent	初版
V1.01	2013/05/10	Charley	更新 UI 图片
V1.02	2019/08/23	Kent	1. 更新图片、部分内容调整 2. 原文件拆分成 IDE_UM 和 Mini-C_UM
V1.03	2023/02/01	Freya	1. Mini-C_UM 更新

1. Mini.C 介绍

1.1. 一般规则

1.1.1. 变量名称

(1). 变量名称不分大小写，变量开头为英文字母 `A~Z`, `a~z` , ` _`
(2). @不能做变量名称之开头
(3). 非变量之名称开头则可以为 `A~Z`，`a~z`，`_`，`0~9`
(4). 变量长度不限
(5). 在使用 Mini-C 的项目时，变量名称一样是不分大小写
(6). 范例：
```
BYTE abcd,  _123;(O)
BYTE @ABC;          (X) => @不能为变量开头
```

1.1.2. 数值表示

(1). 十六进制： 例如： 0xAB, 0CDh
(2). 十进制： 例如： 12, 34d
(3). 二进制： 例如： 0B11_00_1100, 01011010B
(4). 字符 ： 例如： ‘A’, ‘Z’
(5). 在二进制中，可以增加 `_`，以提高程序撰写的可读性
(6). 范例：
```
MOV     A, 31h
MOV     A, 49
MOV     A, 0B11_0001
MOV     A, ‘1’
```

1.1.3. 注释的表示法

(1). 利用 /* … */ 来作整个区块的注解
(2). 利用 // 来作单行的注解
(3). 范例：
```
/* 这是区块注解
……
…… */
```

```
MOV         A, 12         // 这是单行注解
```

1.1.4. 符号“；”的定义

(1). 在 Asm 模式的项目中，`；`可用来作单行的注解。

范例：

```
MOV         A, 34         ；这是 Asm 项目的单行注解
```

虽然很多使用者，在指令的结束，习惯不加`;`，也能组译成功，

但是底下的 C 语法，少了`;`，语意完全不同。

范例：

a)
```
while (PA.0)
    count++;
```
当 PA.0 为 High 时，count 一直递增。

b)
```
while (PA.0);
    count++;
```
要等到 PA.0 为 Low 时，count 才加一。

c)
```
while (! ADCC.6);
    count++;
```
要等到 ADC 转完后，count 才加一。

d)
```
while (! ADCC.6)
    count++;
```
计算 ADC 转完，须要 count 几次。

(2). 在 Mini-C 模式的项目中，`；`代表的是指令的分隔元。而且所有语法都必须以`；`作为结束。

范例：

```
A = 56 ;      A += M   ;    // 这是两条件令。
```

(3). 在 IDE 的编辑器中，符号`；`也是有相同的特性。

如果编辑一个副档名为．ASM / ．INC 的程序，`；`后面的文字会被当作注解来显示。

如果编辑一个副档名为．C / ．H 的程序，`；`只是单纯的指令分隔元。

(4). 在 Asm 的项目中，预设是使用副档名为．ASM / ．INC 的程序；

在 Mini-C 的项目中，预设是使用副档名为．C / ．H 的程序；

副档名的不同，只能影响编辑器的显示规则，无法影响编译器的编译方法。

编译方法的决定，仍是以 Asm 或 Mini-C 的项目型式来决定。

1.1.5. 标记的表示法

在所有标记名称的后面，皆有一个 ： 号，后面可以继续跟着指令码，以下为例：

```
Label:      mov       A, 12;
            …
            goto      Label;
```

另外，为了减少过多的标记名称，支援简易的标记名称 @@:，如下例：

```
            goto    @F          //  跳跃到后面的 @@标记 1
            …
@@:                             //  标记 1
            …
            goto    @F          //  跳跃到后面的 @@标记 2
            …
            goto    @B          //  跳跃到前面的 @@标记 1
            …
@@:                             //  标记 2
            …
            goto    @B          //  跳跃到前面的 @@标记 2
```

你也可以为简易的标记命名，那么跳越的距离就更有弹性了，如下例：

```
@@.XX:                          //  标记 XX
            goto    @F          //  跳跃到后面的 @@
            …
@@:         …
            goto    @B.XX       //  跳跃到前面的 @@标记 XX
            …
            goto    @F.XX       //  跳跃到后面的 @@标记 XX
            …
@@:         …
            goto    @B          //  跳跃到前面的 @@
            …
@@.XX:                          //  标记 XX
```

1.1.6. EQU 的使用

(1). EQU 可用来宣告固定常数，而且宣告后不允许再次改变，对于不变的常数，建议用这种方式宣告。

```
范例：      Const       EQU         20
```

(2). EQU 可用来取代固定字符串，如下例。

```
范例：      GET_HI      EQU         mov     a, hb@
            WORD        point;
            GET_HI      point               //      等于 mov    a, hb@point
```

(3). EQU 也可用于宣告成变量的别名。

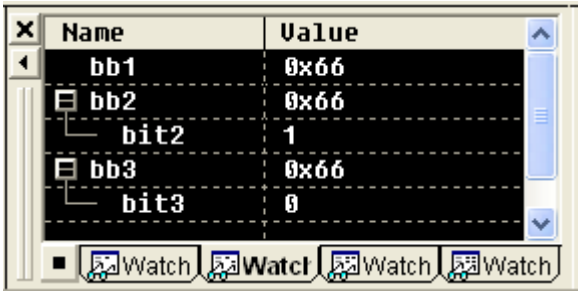
```
范例：      BYTE        bb1;
```

```
bb2         EQU     bb1;    //      bb2 / bb3 是 bb1 的别名，这是习惯语法
bb3         EQU     bb1;    //
```

(4).EQU 也可用于 BIT 的宣告。

```
范例：      BIT     bit2    :   bb2.2; //    BIT 宣告的标准语法
            bit3    EQU         bb3.3; //    大部份人的习惯语法
```

从附图，可以看到彼此的关系。



1.1.7. 符号 `=>` 的使用

(1).对于常须改变的变量，可以利用`=>`来宣告暂时变量，而且宣告后允许再次更改。

(2).范例：

```
Temp        =>      0;
. REPEAT    10                  //      关于.REPEAT 的语法，请参考后面的巨集介绍。
DC                  Temp
Temp  =>    Temp + 1;
. ENDM
```

此语法类似 DC 0, 1, 2, 3, 4, 5, 6, 7, 8, 9。

(3).在大部份的组译器中，常使用`=`来代表此功能，但显然会与 C 的`=`相冲突，所以改用`=>`较为妥当。

1.1.8. #DEFINE 与#UNDEF 的使用

(1).#DEFINE 用来宣告文字取代，#UNDEF 来取消定义

(2).如果想再次定义成不同文字，必须用#UNDEF 来取消前次定义

(3).善用#DEFINE 的文字取代功能，可使程序更有弹性

```
范例:   #DEFINE     EINT        ENGINT
        #DEFINE Goto_Val(x) goto    label_#x
        tmp =>  0;
```

```
_pcadd
{
    .repeat 2                   //  这相当于产生
    Goto_Val(%tmp);         //  goto    label_0;
    tmp =>  tmp + 1;        //  goto    label_1;
    .endm
}
```

(4). 你可以使用 % 字符，将传递的参数，由字符串变成数值，如上例，关于 % 转换的更进一步运用，请参考 ECHO 的应用。

1.1.9. ．ENUM 的使用

(1). 语法： enum { name0 [= CONST] {, name1 [= CONST] } … };
范例： enum { nameA, nameB, nameC };
说明： 定义 nameA EQU 0，nameB EQU 1，nameC EQU 2。
范例： enum { nameX = 0xAA, nameY, nameZ = 0xCC };
说明： 定义 nameX EQU 0xAA，nameY EQU 0xAB，nameZ EQU 0xCC。

(2). 使用 enum 宣告的常数，用鼠标移过时，会显示内容。
而使用文字取代功能 (#DEFINE) 宣告的常数，则无法用鼠标显示内容。
而且 enum 内宣告的常数，预设值会自动加一，比 EQU 宣告好用。

1.1.10. ．CHIP 的宣告

(1). 语法： ．CHIP xxxx

(2). 说明：用来选择所使用的 Chip，通常这个指令放在程序的起始位置。
范例： ．CHIP PDK82C10

1.1.11. 用 ．INCLUDE 来包含其它档案

(1). 语法： ．INCLUDE filename 或 #INCLUDE filename

(2). 说明： 将所选择的 filename 包含进来， 使用 ．INCLUDE 与 #INCLUDE 的结果是相同的

范例：
```
．INCLUDE  Src1.asm
．INCLUDE  “Src  2.asm”        //      “包住空白”
# INCLUDE <Std.h>              //      从系统目录包含档案
```

1.2. 宏

1.2.1. MACRO 宏

(1). 语法：
```
macro_name MACRO    param
           ENDM
```

(2). 说明：巨集的使用可让程序更具弹性，也是汇编语言中最富变化的一环。可与 #define 指令互相比较。

范例一：
```
MOV_ MACRO    x,  y
     .IFB  <y>
        .ERR
            .ECHO       param2 is blank
        EXITM
     .ENDIF
     MOV      A, y
     MOV      x, A
     ENDM
```

范例一之说明：.IFB 是在测试这个角括号内的巨集参数是否为空白？如果是的话，.ERR 将强制产生一个错误，并由 .ECHO 将讯息送给组译器，而 EXITM 则会离开这个巨集。常用这个条件组合，将让程序更具防止错误的能力。

范例二：
```
MOV_    MACRO    x,  y
        .IFB     <x>
            .ERROR      param is blank
            EXITM
        .ELSEIFB <y>
            .ERROR      param2 is blank
            EXITM
        .ENDIF
        MOV      A, y
        MOV      x, A
        ENDM
```

范例二之说明：.ERROR 是 .ERR 与 .ECHO 的合并语法。

范例三：
```
MOV_    var1,   3
MOV_    var2,   < 1 + 2 >
```

范例三之说明：　使用巨集时，如果参数中有' '、','等特殊字符，你可以用 < > 将该参数含括起来，就可以被视为同一个参数了。

范例四：

(1). 使用：巨集最常被用来当作条件跳跃的指令代换。

(2). 范例：
```
JAE   Var1, Var2, label    //  If  Var1 ≧ Var2, then jump to label
JBE   Var1, Var2, label    //  If  Var1 ≦ Var2, then jump to label
JA    Var1, Var2, label    //  If  Var1 > Var2, then jump to label
JB    Var1, Var2, label    //  If  Var1 < Var2, then jump to label
```

(3). 巨集 JAE 之程序内容，可写成如下：
```
JAE     MACRO   v1,  v2,  lab
        MOV     A,   v1
        COMP    A,   v2        //  if  v1 >= v2,  then CF = 0
        T1SN    CF
        GOTO    lab            //  jump to lab, at CF == 0
        ENDM
```

(4). 如果 ACC 也要能当作参数，（例如：　JAE　A, Var1, label），巨集程序内容可修改如下：

Step 1：
```
JAE     MACRO     v1,  v2,  lab
        .IFIDNI   <A>, <v1>      //  if  v1 == "A"
            COMP  A,   v2
        .ELSEIFIDNI <A>, <v2>    //  else if v2 == "A"
            COMP  v1,  A
        .ELSE
            MOV   A,   v1
            COMP  A,   v2
        .ENDIF
        T1SN    CF
        GOTO    lab              //  jump to lab, at CF == 0
        ENDM
```

Problem：上面程序仍然有误，以 JAE 20, A, label 为例：
```
COMP    20,  A      //  没有这条指令
```

```
        T1SN    CF
        GOTO    lab
```

Step 2： 巨集内容修改如下：

```
JAE     MACRO   v1,     v2,     lab
        .IFIDNI         <A>, <v1>
            COMP    A,      v2
            T1SN    CF
        .ELSEIFIDNI     <A>, <v2>
            CEQSN   A,      v1
            T0SN    CF
        .ELSE
            MOV     A,      v1
            COMP    A,      v2
            T1SN    CF
        .ENDIF
        GOTO    lab             //      jump to lab, at CF == 0
        ENDM
```

(5). 在本发展系统请舍去这种过时的写法，让编译器搞定这种繁琐小事，直接写成：

```
IF      ( 20 >= A)      GOTO        label;
```

1.2.2. .EXPAND 巨集

语法： macro_name EXPAND param

…

ENDM

说明： EXPAND 与 MACRO 的功能大同小异，在使用时机上，当巨集内容有误时，建议使用 EXPAND，展开巨集，这样可以方便知道巨集内容错误的行号。

当巨集完成时，建议使用 MACRO，将整个巨集视为一条指令，

这样在单步执行时，可以一次执行完整个巨集内容。

1.2.3. .REPEAT 巨集

语法： .REPEAT 常数

说明： 巨集方块内的指令。

范例：
```
Temp            =>      0;
. REPEAT 10
    DC                  Temp
    Temp  =>    Temp + 1;
. ENDM
```

此语法类似 DC 0, 1, 2, 3, 4, 5, 6, 7, 8, 9。

1.2.4. .FOR 参数, <参数 1, 参数 2, …>

语法： .FOR 参数, <参数 1, 参数 2, …>

使用： 重复巨集方块内的指令，并将参数以<>内的参数逐一取代。

范例：
```
. FOR       ww,   <PA, PB, PC>
    ww    =     0;     ww#C =     0;
. ENDM
```

此语法类似 PA = 0; PAC = 0; PB = 0; PBC = 0; PC = 0; PCC = 0。

为了将参数与巨集方块内的其他指令相连接，可以用 # 字符，当做连接符号。

1.2.5. .FORC 参数, <字符串>

语法： .FORC 参数, <字符串>

使用： 重复巨集方块内的指令，并将参数以<>内的字符串逐一取代。

范例：
```
. FORC      cc,   <ABC>
    P#cc  =     0;     P#cc#C     =     0;
. ENDM
```

此语法类似 PA = 0; PAC = 0; PB = 0; PBC = 0; PC = 0; PCC = 0。

1.3. 传统组译指令

1.3.1. 条件式组译指令

(1). 在 Mini-C 与 ASM 中，都支援条件式组译指令，以#以及 . 开头的指令格式，本发展系统都支援。支援指令如下：

#IF #IFZ #IFB #IFNB
#IFIDN #IFIDNI #IFDIF #IFDIFI
#IFDEF #IFNDEF
#ELSEIF #ELSEIFZ #ELSEIFB #ELSEIFNB
#ELSEIFIDN #ELSEIFIDNI #ELSEIFDIF #ELSEIFDIFI
#ELSEIFDEF #ELSEIFNDEF
#ELSE #ENDIF

.IF .IFZ .IFB .IFNB
.IFIDN .IFIDNI .IFDIF .IFDIFI
.IFDEF .IFNDEF
.ELSEIF .ELSEIFZ .ELSEIFB .ELSEIFNB
.ELSEIFIDN .ELSEIFIDNI .ELSEIFDIF .ELSEIFDIFI
.ELSEIFDEF .ELSEIFNDEF
.ELSE .ENDIF

(2). 指令说明：

第一群组是以 IF 为开头，指令包括如下：

- #IF， . IF
  格式： #IF 数值
  说明:： 当数值非 0 时，为真
- #IFZ， . IFZ
  格式： #IFZ 数值
  说明： 当数值是 0 时，为真
- #IFB， . IFB
  格式： #IFB <参数>
  说明： 当没有参数时，为真
- #IFNB， . IFNB
  格式： #IFNB <参数>
  说明： 当存在参数时，为真

- #IFDEF， . IFDEF

  格式： #IFDEF <参数>

  说明： 当参数有定义时，为真

- #IFNDEF， . IFNDEF

  格式： #IFNDEF <参数>

  说明： 当参数未定义时，为真

- #IFIDN， . IFIDN

  格式： #IFIDN <参数 1>, <参数 2>

  说明： 当参数 1 大小写完全相等于参数 2 时，为真

- #IFIDNI， . IFIDNI

  格式： #IFIDNI <参数 1>, <参数 2>

  说明： 不管大小写，当参数 1 等于参数 2 时，为真

- #IFDIF， . IFDIF

  格式： #IFDIF <参数 1>, <参数 2>

  说明： 须管大小写，当参数 1 不等于参数 2 时，为真

- #IFIDIFI， . IFDIFI

  格式： #IFDIFI <参数 1>, <参数 2>

  说明： 不管大小写，当参数 1 不等于参数 2 时，为真

第二群组为 #ELSEIFxx / . ELSEIFxx，是搭配在 #IFxx / . IFxx 或 #ELSEIFxx / . ELSEIFxx 之后，当作另一种条件式的判断为开头，指令包括如下：

- #ELSEIF， . ELSEIF

  格式： #ELSEIF 数值

  说明： 当数值非 0 时，为真

- #ELSEIFZ， . ELSEIFZ

  格式： #ELSEIFZ 数值

  说明： 当数值为 0 时，为真

- #ELSEIFB， . ELSEIFB

  格式： #ELSEIFB <参数>

  说明： 当没有参数时，为真

- #ELSEIFNB， . ELSEIFNB

  格式： #ELSEIFNB <参数>

  说明： 当存在参数时，为真

- #ELSEIFDEF， . ELSEIFDEF

  格式： #ELSEIFDEF <参数>

  说明： 当参数有定义时，为真

- #ELSEIFNDEF， . ELSEIFNDEF

  格式： #ELSEIFNDEF <参数>

说明： 当参数未定义时，为真

- #ELSEIFIDN， . ELSEIFIDN

  格式： #ELSEIFIDN <参数 1>, <参数 2>

  说明： 当参数 1 大小写完全相等于参数 2 时，为真

- #ELSEIFIDNI， . ELSEIFIDNI

  格式： #ELSEIFIDNI <参数 1>, <参数 2>

  说明： 不管大小写，当参数 1 等于参数 2 时，为真

- #ELSEIFDIF， . ELSEIFDIF

  格式： #ELSEIFDIF <参数 1>, <参数 2>

  说明： 须管大小写，当参数 1 不等于参数 2 时，为真

- #ELSEIFDIFI， . ELSEIFDIFI

  格式： #ELSEIFDIFI<参数 1>, <参数 2>

  说明： 不管大小写，当参数 1 不等于参数 2 时，为真

#ELSE / . ELSE 是搭配在 #IFxx / . IFxx 或 #ELSEIFxx / . ELSEIFxx 之后，只要前面的条件判断皆不符合的话，则以下为真。#ENDIF / . ENDIF 是所有条件组译的结束指令，只要有一个 #IFxx / . IFxx 当起始，就必有成对的 #ENDIF / . ENDIF 来作结束。

(3). 范例暨说明：

- **例一：**

```
value       =>    1;
#IF         value                 //    这区域会被组译
    …..
#ENDIF
```

- **例二：**

```
Value  =>   0;
#IF         value == 1            //    这区域不会被组译
    …..
#ELSEIFZ    value                 //    这区域会被组译
    …..
#ENDEF
```

- **例三：**

```
DELAY_      macro x
                #IFB        <x>             //    如果参数 x 是空白
                    . ERROR         param is blank
                #ENDIF
            #IFIDNI     <x>, <A>        //    如果参数 x 是 A 或 a
```

```
        DELAY       A
#ELSEIFZ            x           //      如果参数 x 是 0
#ELSEIF     x   == 1            //      如果参数 x == 1
        nop
#ELSEIF     x   > 256           //      如果参数 x > 256
                .ERROR          param is too large
#ELSE
        DELAY               x - 1
#ENDIF
endm
```

由于 delay N 这条指令，实际是暂停 N+1 条指令。所以最小的 delay time 是 1，相当于暂停 2T，最大的 delay time 是 delay 0，暂停 257T，不过为了可读性，通常用 delay 0xFF，暂停 256 条指令，当作最大值。不过在实际应用上，例如 UART 的接收发送，如果 Baud Rate 太低而须暂停超过 256 个指令周期，以达到 delay time 的需求。我们可以设计如下的巨集 Easy_Delay。

◆ **例四：**

```
//      Easy_Delay  需要暂停的次数
        Easy_Delay  1000        //  须要 delay 1000 次。
```

巨集宣告如下：

```
Easy_Delaymacro temp
    .repeat     (temp >> 8)     //  超过 256 次的暂停
        delay       0xFF;       //  以 delay 0xFF 取代
    endm
    temp    =>  temp & 0xFF;    //  得到不足 256 次的暂停

#IFZ        temp                    //  0       :   不用产生码
#ELSEIF     (temp == 1)     //  1       :   NOP
            nop;
#ELSE                       //  2 ~ 255 :   DELAY
            delay   temp - 1;
#ENDIF
    endm
```

(4). 补充语法：

在 Mini-C 中，支援巨集语法 .DELAY 常数，自动产生最简化的指令码，而且常数允许超过 64K，所以，上述的巨集语法，纯粹只有介绍指令的功能，请改用如下语法作延迟功能。

```
. DELAY     0;          //      不产生指令
. DELAY     1;          //      产生 NOP 指令
. DELAY     2;          //      产生 DELAY 1 指令
. DELAY     1000; //    请自行参考反组译的结果
```

1.3.2. 条件式错误指令

想让编译器产生错误，有以下两种方法。

```
. ERR                               //    (1)    强制产生错误。
. ERROR     Error_Message   //    (2)    强制产生错误，并提供说明文字。
```

如果想要在特定条件下，产生错误指令，也可以有下两种方法。

```
(1)  . IFxxx                        //      利用 5-12 节所述的条件式组译指令
                . ERROR             Error_Message
        . ENDIF
```

(2) 利用 . ERRxxx 指令群，当特定条件下符合时，产生错误指令。
详细规则，请参考上节所述的条件式组译指令。

```
. ERRZ          . ERRNZ         . ERRB          . ERRNB
. ERRDEF  . ERRNDEF  . ERRIDN   . ERRIDNI
. ERRDIF        . ERRDIFI
```

底下是一些范例。

```
. ERRZ      val         //      当 val== 0 时，产生错误指令。
. ERRNZ     val == 5    //      当 val== 5 时，产生错误指令。
. ERRB      <x>         //      如果参数 x 是空白，产生错误指令。
```

另外， . ERRE 同义于 . ERRZ， . ERRNE 同义于 . ERRNZ，不再另外说明。

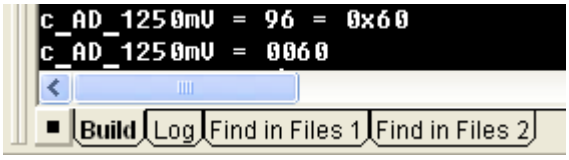
1.3.3. . ECHO 的应用

想让编译器将一些编译过程的参数，显示在 Build View，你可以参考以下的范例。

假设 VCC = 3.3V，当一个 AD Input 为 1.25V 时，ADC 的值该为多少？

```
c_AD_1250mV                 =>              255 * 1250 / 3300;
 . echo  c_AD_1250mV = %c_AD_1250mV = 0x%X：c_AD_1250mV
 . echo  c_AD_1250mV = %04X：c_AD_1250mV
```

第一个 % 变量，显示变量的 10 进制值，第二个 %X：变量，显示变量的 16 进制值，第三个 %04X：变量，显示变量的 16 进制值，长度 4 码，不足 4 码的地方，可以补 0。



再来看另一个例子，在工作电压为 5V 时，想利用分压电阻，测得高压 20V 与低压 3V，于是利用 6.8KΩ 与 1.3KΩ 作分压设计，使用者想知道此时的 AD 值，与经过分压电阻后的电压值。

如下范例，定义了 c_VDD，R_Voltage_Up，R_Voltage_Dn 等常数。

利用巨集 Calu_AD_Value，传入参数一，当作储存 AD 常数的名称，传入参数二，则是高低压值。

第一个巨集，算出了 AD 值与分压后的电压值，但是输出格式完全不整齐。

第二个巨集，利用一些格式化指令，将字符串设成 20 个字，数字也都对齐了，但看起来还不够工整。

到第三个巨集，将字符串靠左对齐，将 mV 转成 V，看起来就整齐明了多了。

```
c_VDD               EQU     5000    //  mV
R_Voltage_Up        EQU     6800    //  Ω
R_Voltage_Dn        EQU     1300    //  Ω

Calu_AD_Value   macro   cc, vv
        v2      =>      vv * R_Voltage_Dn / (R_Voltage_Up + R_Voltage_Dn)
        cc      EQU     256 * v2 / c_VDD
        .echo   cc : AD = %cc , Vol = vv -> %v2
                endm

Calu_AD_Value   c_Volt1_Low         3000    //  mV
Calu_AD_Value   c_Volt1_High        20000   //  mV

Calu_AD_Value   macro   cc, vv
        v2      =>      vv * R_Voltage_Dn / (R_Voltage_Up + R_Voltage_Dn)
        cc      EQU     256 * v2 / c_VDD
        .echo   %20s:cc : AD = %02X:cc , Vol = %5d:vv mV -> %04d:v2 mV
                endm

Calu_AD_Value   c_Volt2_Low         3000    //  mV
Calu_AD_Value   c_Volt2_High        20000   //  mV

Calu_AD_Value   macro   cc, vv
        v2      =>      vv * R_Voltage_Dn / (R_Voltage_Up + R_Voltage_Dn)
        cc      EQU     256 * v2 / c_VDD
        .echo   %-20s:cc : AD = 0x%02X:cc , Vol = %5d.3:vv V -> %04d.3:v2 V
                endm

Calu_AD_Value   c_Volt3_Low         3000    //  mV
```

```
c_Volt1_Low : AD = 24 , Vol = 3000 -> 481
c_Volt1_High : AD = 164 , Vol = 20000 -> 3209
        c_Volt2_Low : AD = 18 , Vol =  3000 mV -> 0481 mV
       c_Volt2_High : AD = A4 , Vol = 20000 mV -> 3209 mV
c_Volt3_Low         : AD = 0x18 , Vol =  3.000 V -> 0.481 V
c_Volt3_High        : AD = 0xA4 , Vol = 20.000 V -> 3.209 V
```

Build | Log | Find in Files 1 | Find in Files 2

1.3.4. 重新组译

在有些时候，你可能须要将一个项目，同时组译出不同的编码，并储存成不同的 PDK 档案。

如范例项目 [Menu]->[File]->[Demo Project]->[P201A_PA1&4_PWM.PRJ]，
想要在 P201A 的 PA4 输出 PWM，但 ICE PDK3S-I 只能选择从 PA1 输出 PWM，
所以要重新组译，第一次组译出 Real IC 的编码，第二次组译出 ICE 的编码，
并使用专用的转接版，将 PA4 导向 PA1。

使用了指令：.ReAssembly -> 须要再重新组译。
.OutFile -> 设定输出档名。
_SYS(ASM_LOOP) -> 获得已经重新组译的次数。
.ICE_CMD Change_PA4_to_PA1
-> 组译时，接受 PA1 的使用。

并将这些指令放在 extern.h，让其它的档案参考，组合如下。

```
NOW_SEL     =>      _SYS(ASM_LOOP)

#if     NOW_SEL == 0
    BIT PWM_Pin :   PA.4;   //  组译 Real IC 编码。
    //  .OutFile    ...     //  就用预设档名。
    .REASSEMBLY             //  还要再重新组译。
#elseif NOW_SEL == 1
    .ICE_CMD    Change_PA4_to_PA1
                            //  只限 P201A 使用。
    BIT PWM_Pin :   PA.1;   //  组译 ICE 编码。
    .OutFile        "Only_ICE.PDK"
                            //  另外取新档名，以免覆盖。
#endif
```

数据型态

1.4.1. 变量的资料型态/常数型态

(1). 提供 BYTE，WORD，EWORD，DWORD 等资料型态。
(2). 所有资料型态皆为无号数。
(3). BYTE 是 8-BIT 资料型态，WORD 为 16 BIT，EWORD 为 24 BIT，DWORD 为 32 BIT。
(4). INT 与 BYTE 皆是相同的无号数 8-BIT 资料型态，以支援往前兼容。
(5). 范例：

```
BYTE  BB1, BB2;
WORD    WW1, WW2;
EWORD   BB3;
DWORD   DW1, DW2;
```

```
mov     A, BB1;
A   =   BB2;
```

(6). 常数型态的定义如下范例

```
CONST   EQU     12
Value   =>      34h

mov     A, CONST
mov     A, value;
A   =   55h;
```

(附注) 系统会自动区分资料型态或常数型态。

1.4.2. BIT 的资料型态

(1). 语法： BIT name [: 变量 . bit_num]

(2). 范例：

```
BIT     BB1_2     :   BB1.2;
BIT     WW1_12    :   WW1.12
BIT     Flag;         //  由系统自行定义
```

(3). 说明：对于 BYTE 的变量，bit_num 的范围为 0~7，对于 WORD 的变量，bit_num 的范围为 0~15，依此类推 …，未指定位址的变量(如上例的 Flag)，位址由系统安排。

1.4.3. REG 的 BIT 宣告

(1). 范例：

```
PA_1    EQU     PA.1   或
PA_1    BIT     PA.1
```

(2). 说明： 宣告 PA_1 等于 PA 的 BIT 1。

(附注) 对于 BIT 的宣告，你可以使用以下的所有格式，

```
a)  BIT   name ：   变量 . bit_num
b)  name  BIT       变量 . bit_num
c)  name  EQU       变量 . bit_num
```

1.4.4. 简易的资料型态

对一个 WORD 的资料而言，只存取其中的一个 Byte，是不太方便的。

如下例 ：

```
WORD    WW;
WW  =   (WW & 0xFF00)  |  ((WW + 1) & 0xFF);
    //  将 WW 的 Low Byte 加 1，WW 的 High Byte 保持不变
```

虽然只是想将 Low Byte 加 1，大部份编译器却得写出一长串指令码来。

因此，Mini-C 提供了以下的新语法：

```
WW $ 0      =   WW + 1;         // the low byte of WW <= low byte of (WW+1)
                                // the high byte of WW keeps unchanged
```

依此类推, $ 2 就是第 2 Byte, $ 3 就是第 3 Byte

```
WW $ 0      =   WW $ 1;         // WW 的 Low Byte <= WW 的 High Byte.
DWORD           DD;
A       =   DD $ 0 ^ DD $ 1 ^ DD $ 2 ^ DD $ 3;      // 将 DD 的四个 Byte 作 XOR 运算.
```

在 6.5 节中，WORD 的资料常被当作 Point，如果 Point 不超过 256 的边界，在 Point 每次加一时，可以只改变 Low Byte，以节省程序空间。

```
WORD    point;
point++;            改写成          point$0++;
```

附带一提，虽然在上例中，

```
WW  =   (WW & 0xFF00)  |  ((WW + 1) & 0xFF);
```

只是想将 WW 的 Low Byte 加 1；但很多编译器却会编译出一些无用的程序码来，让人认为用 C 写的程序没效率，其实，是编译器的最佳化没作好而已。

1.4.5. Point 的使用

在汇编语言中 IDXM，LDTABL，LDTABH 可对应到 Mini-C 的语法，查 RAM 语法，可用*point 表示，为了查 ROM 语法的方便，本系统定了一个方法，* point $ L 与 * point $ H 用来代表 ROM 的 low byte 与 high byte，* point $ W 则代表 ROM 的 word。在 Mini-C 的语法中，所有的 WORD 变量，皆可以当 Point 使用。

使用范例如下：

```
WORDpoint, data;
```

```
BYTE        buffer[10];
Label：     DC          1234h, 5678h;

point       =       buffer;                     // point 指向 buffer 的位址
//          mov         A, la@buffer[0]
//          mov         lb@point, A
//          mov         A, ha@buffer[0]
//          mov         hb@point, A

* point =   * point ^ 0x55;                     // 使用 IDXM 更改 RAM
//          idxm        A, point
//          xor         A, 0x55
//          idxm        point, A

point       =       Label;                      // point 指向 Label 的位址
//          mov         A, la@Label
//          mov         lb@point, A
//          mov         A, ha@Label
//          mov         hb@point, A

A           =       * point $ L;                // 使用 LDTABL 查 ROM
//          ldtabl      point

data$1      =       * point $ H;                // 使用 LDTABH 查 ROM
//          ldtabh point
//          mov         hb@data, A

data        =       * point $ W;                // 使用 LDTABL / H 查 ROM
//          ldtabl      point
//          mov         lb@bb, A
//          ldtabh point
//          mov         hb@bb, A
```

1.4.6. WORD 的存放住址

WORD 的存放位址，一定是在偶数边界。这是因为目前 WORD 也可以当作 Point，而且是硬件的限制，Point 并须在偶数边界上，所以连带所有的 WORD，都是在偶数边界上。当使用 Point ++ / -- 时，指标只会加减一，如果指向的变

量是 RAM 的 WORD、EWORD、DWORD，请自行改用 Point += 2/3/4 或是 Point -= 2/3/4。

1.4.7. [ ] 阵列的使用

阵列是以 0 为基底，而且长度不能为 0，长度也不能超过所宣告的最大值。

```
#define         BUF_SIZE        0x10
BYTE        Buffer [BUF_SIZE];

Buffer[0x0]         =       0;          //      第一个
Buffer[0xF]     =       0xF;        //      最后一个
Buffer[0x10]    =       0x10;   //      错误，超过范围
```

1.4.8. 利用 & 读取变量的位址

如果我们想用上例的阵列作一个 Queue，可以用如下的方法。

```
WORD            Point;
Point       =       Buffer;
//      …
while (1)
{
    //      …       calculate ACC
    *Point =        A;
    Point ++;                                   //      Point plus one each time
    If      ( Point == & Buffer [BUF_SIZE] )    //      until (Point == Buffer), bottom of buffer
    {
        Point       =       Buffer;             //      set Point to the initial value of Buffer
    }
}
```

1.4.9. 利用 & 产生变量的参考名称

如果我们想要在同一个变量位址上，拥有不同的变量名称，可以用如下的方法。

```
WORD            Point, Data[2];
BYTE &      Var1        =       Point$0;        //      Var1 跟 Point 的 Low Byte 是同一位址。
```

```
BYTE &    Var2    =    Point$1;        //    Var2 跟 Point 的 High Byte 是同一位址。
BYTE &    Var3    =    Data[0]$1;      //    Var3 跟 Data[0] 的 Low Byte 是同一位址。
WORD&     Var4    =    Data[1];        //    Var4 跟 Data[1]是同一位址。
```

1.4.10. RAM 的定址

另一个在同一个变量位址上，拥有不同的变量名称，是自定变量位址。

```
.RAMADR 0x00                    //  从 RAM 0 的地方开始放资料
    BYTE    Array[0x10];        //  共占用了 Adr 0x0 ~ Adr 0xF

.RAMADR 0x00                    //  从 RAM 0 的地方开始放资料
    WORD    Word_0, Word_2;     //  共占用了 Adr 0x0 ~ Adr 0x3

.RAMADR SYSTEM                  //  在 "Mini-C" 项目中才有，
                                //  回复系统自动安排变量位址。
    BYTE    Mem_0;              //  因为 Adr 0x0 ~ 0xF 已被占用
                                //  系统可能会从 0x10 以后定址
```

1.4.11. 常数资料

在 “ASM” 项目，常数资料可以宣告如下：

```
. ROMADR    0x3F0               //      绝对定址到 0x3F0
Labe1：     DC    1234h, 5678h, 0xABCD
```

在 Mini-C 项目，常数资料可以宣告如下：

```
CONST   WORD   array_name   [ ]  =   { 0x1234, 0x5678, 0xABCD. };
```

在 Mini-C 项目， . ROMADR 只用来指定相对位址，例子如下所示：

```
. ROMADR        $ + 2           //      program address <= (current + 2)
```

在 Mini-C 项目，不再支援绝对定址了。

```
所以            Error  :        . ROMADR  0x3F0;
```

假如想将常数宣告成字符串，要用 DC 或者 DB 命令，例子如下所示：

```
Label2：        DC    “ABC”, 55h, 0AAh    //      0x4241, 0x0043, 0x0055, 0x00AA
```

```
Label3：        DB    “ABC”, 55h, 0AAh   //      0x4241, 0x5543, 0x00AA
```

DC 命令是以 WORD 为单位，每笔资料不满 WORD 的部份填为 0。
DB 命令是以 BYTE 为单位，先放 Low Byte，再放 High Byte，全部资料填完后，不满 WORD 的部份填为 0。

关于字符串的表示法，支援一般 C 的格式，如下例：

```
Label4：        DC    “AB\0CD\377”       //      \nnn 为 8 进位表示法, \0 代表 0, \377 代表 255。
                                         //      所以放在 ROM 的资料将为 0x4241, 0x4300, 0xFF44。

Label5：        DC    “AB\tCD\n”   //      0x4241, 0x4309, 0x0A44。
```

以下是一个查表的范例 ：

```
WORD        Point;
Point       =       Label1;             //      Point 从 Label1 开始查表
While (1)
{
    A       =       *Point$L;
    If (A == 0)  break;         //      当资料为 0 时，查询才结束。
//  If (ZF) break;          //      错误：因为 LDTAB 指令不会影响 ZF。
    Point++;
}
```

1.4.12. 程序的宣告与实体

在程序的宣告上，Mini-C 与标准的 C 一样，以下面为例。

```
void    proc1 (void);               //      宣告  proc1
void    proc2 (void)                //      proc2 的实体
{
    proc1();                        //      呼叫  proc1
    …
    If (ZF) return;
    …
}
```

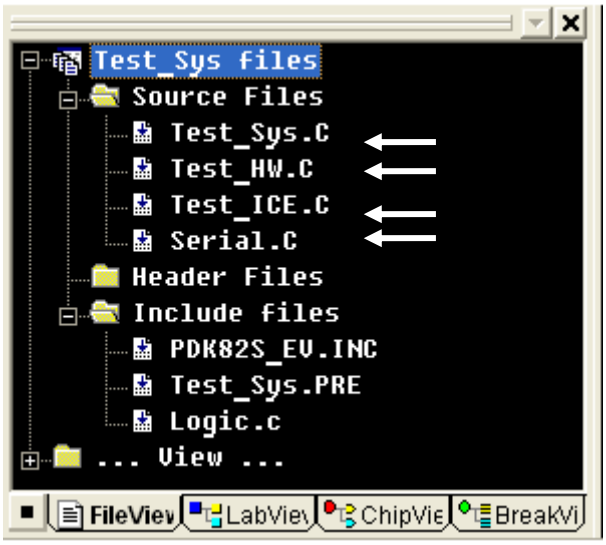
目前 Mini-C 的项目中，尚不支援参数传递与传回值。若中途离开程序，可以用 return 来表示。

而在 Asm 的项目中，不支援程序的宣告，全部皆需以标记来连结。如下范例：

```
proc1:      ….
proc2:      call        proc1  //      呼叫 proc1
            if (ZF) ret;
```

1.4.13. 程序的可见域

在 Mini-C 的项目中，可以有许多档案模块 (如下白色箭号所示) 来组成一个项目，而每个档案模块中，又由很多的程序组成。



所有的程序，预设皆是全域程序。在不同档案模块中，可宣告 extern 来使用其它模块的程序，以下为例：

```
// 档案模块 1
void        porc2 (void)  {…}

// 档案模块 2
void        proc3 (void)
{
        extern void    proc3 (void);
        proc3 ( );                  //      呼叫不同模块的 proc3
}
```

如果将设程序设为区域程序 (static)，则不同的档案模块允许有相同的程序名称。例子如下：

```
// 档案模块 1
```

```
static void proc4 (void)        {…}

// 档案模块 2
static void proc4 (void)        {…}
```

1.4.14. 变量的可见域

宣告在程序外面的变量，预设皆是全域变量。你可用 extern 来取得不同模块的变量。也可用 static 来定义函数外面的变量为区域变量，可见域是从宣告处开始，到模块结束为止。宣告在程序区块内的变量，则属于区块变量，可见域是从宣告处开始，到区块结束为止。假如使用者不想与其他的变量共享同一位址，使用者可以利用 “static” 来独立地定义程序区块的变量。以下为例：

```
BYTE BB1;                       //      定义全域变量
static      BYTE BB2;           //      定义区域变量
extern BYTE BB3;                //      不同模块的全域变量

void    proc1 (void)
{
        BB1   =     BB2;                //      全域变量 BB1 = 区域变量 BB2
        {
            BYTE BB11;              //      定义区块变量 (* 位址可能会与下面的 BB22 重叠)
            BB11  =     BB3;        //      区块变量 BB11 = 区域变量 BB3
        }
        {
            BYTE BB22;              //      定义区块变量 (*位址可能会与上面的 BB11 重叠)
            BB22  =     BB1;        //      区块变量 BB22 = 全域变量 BB1
        }
        {
            static      BYTE BB2;   //定义有独立位址之区块变量，其位址将不与其它变量重叠
            BB2++;                  // 独立的区块变量 + 1
        }
}
```

1.5. 表达式

1.5.1. 算术表达式

本发展系统已提供+，-，&，|，^，( )，=，+=，-=，&=，|=，^= 等操作数。

以下为例：

```
BB1     =     BB2;
WW1  +=     (WW2 + BY2) & 1234h;
```

不过以下两点请注意：

◆ 尚未提供连等的表达式，如下范例：

```
BB1   =     (BB2   =     BB3) + 2;
BB1   =     BB2   =     BB3;
```

◆ 对于++，--的操作数，暂时只允许如下格式。太复杂的表达式尚未被验証，请多包涵。

```
BB1++;          WW1--;
```

1.5.2. 逻辑表达式

本发展系统提供 IF，ELSE，WHILE，DO，BREAK，CONTINUE 等表达式，并提供 ==，!= ，>，>=，<，<=，！，&&，||，++，- -，( ) 等操作数。

范例一：

```
if      ( WW1 )                                 {…}
else if ( (BB1 >= BB2) && (BB1 > 12h) ) NULL;
else if ( (PA != 55h) || PB.2 )                 ;
else if ( ! BB1_2 )                             delay 2 ;
else
{
        if  (…)     …;                          // nesting if condition
        { …
        }
}
```

红色的符号`;`加在该处是所有 C 的规则。范例一的 NULL 是可以省略的，不过增加 NULL，可读性会好一点。

范例二:

```
while  ( A && BB2 )             { ...; continue ; … }
while  (1)                      { …; break ; …}
do          { … }               while (BB2 >= BB1);
```

```
while   ( BB1-- && BB2++ )        { ...; continue ; … }
do              { … }              while (--BB2);
```

本发展系统在使用表达式时，要特别注意下列事项：

(1). 在 If 的条件内，目前不支援包含算术表达式，如下例：

```
If      ( BB1 == (BB2 + 1) )      …          (X)
```

(2). 在上面的例子中，假如操作数是个标记，允许如下的特例

```
if      ( Point ==    Label + Const )      …
```

范例 ：

```
Label :          DC 12, 34, 56
WORD             Point;
if      (Point == Label + 3) Point = Label;
```

(3). 暂时不支援 for 语法。

1.5.3. do {…} while (..); 逻辑表达式

表达式 do {…} while (..) 是大家最常用的逻辑表达式，在底下的表达式中，会被组译成 ：

```
BYTE    count       =    10;
        do    { … }                  //    lab_do:    …
        while (--count);             //               dzsn       count ;
                                     //               goto       lab_do;
```

注意的是，--count 与 count--代表的意思不同。

```
BYTE    count       =    10;
        do    { … }                  //    lab_do:    …
        while (count--);             //               dec        count ;
                                     //               t1sn       CF;
                                     //               goto       lab_do;
```

但是如果循环中使用 WORD 当计数，在底下的表达式中，--count 会被组译成 ：

```
WORD    count       =    0x1234;
        do    { … }                  //    lab_do:    …
        while (--count);             //               dec        count$0 ;
                                     //               subc       count$1;
                                     //               mov        a, count$1;
                                     //               or         a, count$0;
                                     //               t1sn       ZF;
                                     //               goto       lab_do;
```

显然表达式变得很长，而且 ALU 值又被改变，如果改成 count-- 的表达式 ：

```
WORD    count       =   0x1234;
        do    { … }                 //    lab_do:    …
        while (--count);            //               dec        count$0 ;
                                    //               subc       count$1;
                                    //               t1sn       CF;
                                    //               goto       lab_do;
```

所以简单来说，在 BYTE 的运算中，--count 是比较能做最佳化，在 WORD 的运算中，count -- 却比较好。

1.5.4. SWITCH

IDE 支援 CASE、BREAK、DEFAULT 等语法。SWITCH 表达式中的 VAR，只支援 BYTE 的运算，所以操作数只能为 ACC，Register，宣告为 BYTE 的变量或小于 256 的常数。

范例一：

```
switch ( VAR )
{
case BB1 :    …;
case 12h :          case 34h :
                    …; break;
default :           …;
}
```

1.6. IO 的设定

1.6.1. REG 的设定

在设定 T16M、ADCC、ADCM、INTEN、INTRQ 等缓存器时，有些人使用如下语法。

```
$ ADCM   /16, 12BIT;          //   ADCM     =    100_0100_0b;
$ ADCC   Enable, PB0;         //   ADCC     =    1_0_000_00b;
$ T16M   EOSC, /1, BIT15;     //   T16M     =    0xA7;
```

显然，就方便性和可读性来看，$ IO xx, xx 的语法似乎也是不错的选择。

你可从 Include files:xxx.INC 中，得到更多语法的了解，如下例。

```
T16M            IO_RW           0x06
    $ 7 ~ 5 :    STOP, SYSCLK, X, PA4_F, IHRC, X, ILRC, PA0_F
    $ 4 ~ 3 :    /1, /4, /16, /64
    $ 2 ~ 0 :    BIT8, BIT9, BIT10, BIT11, BIT12, BIT13, BIT14, BIT15

ADCM            IO_RW           0x21
    $ 7 ~ 5 :    8BIT, 9BIT, 10BIT, 11BIT, 12BIT
    $ 3 ~ 1 :    /1, /2, /4, /8, /16, /32, /64, /128

INTEN           IO_RW           0x04
    $ 3 :    X, AD
    $ 2 :    X, T16
    $ 1 :    X, PB0
    $ 0 :    X, PA0
```

在 T16M 的定义中，可以知道 T16M 由 3 种字段组成，而关键词 STOP、SYSCLK、IHRC、PA4_F、ILRC、PA0_F 皆可用在 $ T16M 的语法中。而 X 表示不存在的状态，如 $ 7 ~ 5 ： STOP, SYSCLK, X, PA4_F, IHRC, X, ILRC, PA0_F，相同的，/1, /4, /16, /64，BIT8 ~ BIT15 等关键词，也皆可用在 $ T16M 的语法中。

在 INTEN 的定义中，有数种中断来源，如 AD、T16、PB0、PA0。你可以将想设定为 1 的中断来源填入，没有填入的中断来源，组译器将以 0 取代该字段。
如下例：

```
$ INTEN     PA0;        // INTEN = 0001B，只有 INTEN.PA0 = 1，其余为 0。
$ INTEN     PB0, AD;    // INTEN = 1010B，只有 INTEN.PB0/AD = 1，其余为 0。
```

当然，如果你只想改变 INTEN 中的某一项中断来源，也可以用如下语法：

```
INTEN.PA0   =   1;      //  等同于  set1    INTEN.0
INTRQ.T16   =   0;      //  等同于  set0    INTRQ.2
```

在 $ Reg xx, yy 的语法中，如果少了某些字段，组译器将以默认值 (通常是 0，但有些例外)设定该字段。如下例：

```
$ T16M      SYSCLK, /16;    // 等同于 $ T16M    SYSCLK, /16, BIT8;
$ T16M      STOP;           // T16M =   0; 或  $ T16M  STOP, /1, BIT8
```

请不要自行更改 Include files:xxx.INC 的内容，否则，会造成组译的错误。

在汇编语言中的 SET0/SET1 bit， 你可以改用 C 的 bit = 0/1 来表示，如下例。

```
WORD    ww;
BIT     Bit_1;
Bit_1   =   1;   //   等同于 SET1   Bit_1
Bit_1   =   0;   //   等同于 SET0   Bit_1
ww.15 =   1;     //   等同于 SET1   hb@ww.7
```

目前只支援这样的功能，以后再增加新语法。

在某些范例上，会有如下例子。

```
BIT         LED_Data    :   PA.0;
BIT         Key_In      :   PA.1;
$ LED_Data      Out, High;                  //   将 LED_Data 设为 Output High
$ Key_In        In, Pull;                   //   将  Key_In  设为 Input, Pull High
$ PA.2          Out, Low, High, Low;        //   将 PA.2 设为 Output, 并送出一个 High Pulse
```

它等同于如下范例。

```
BIT         LED_Ctrl    :   PAC.0;
BIT         LED_Data    :   PA.0;
LED_Ctrl        =   1;              //   设定为 Output
LED_Data        =   1;              //   输出为 High

BIT         Key_Ctrl    :   PAC.1;
BIT         Key_Pull    :   PAPH.1;
BIT         Key_In      :   PA.1;
Key_Ctrl        =   0;              //   设定为 Input
Key_Pull        =   1;              //   输出为 Pull High

PAC.2 =   1;   PA.2  =   0;   PA.2  =   1;   PA.2  =   0;
```

不过，就方便性和可读性来看，前者略胜一筹。

1.6.2. IO 的设定

在设定 T16M，ADCC，ADCM，INTEN，INTRQ 等 Register 时，有些人使用如下语法。

```
$ ADCM      /16, 12BIT;          //    ADCM=     100_0100_0;
$ ADCC      Enable, PB0;         //    ADCC=     1_0_000_00;
$ T16M EOSC, /1, BIT15;          //    T16M =    0xA7;
```

显然，就方便性和可读性来看，$ IO xx, xx 的语法似乎也是不错的选择。

你可从 PDKxxxx.INC 中，得到更多语法的了解，如下例。

```
T16M        IO_RW           0x06
        $ 7 ~ 5     :       STOP, SYSCLK, X, X, IHRC, EOSC, ILRC, PA0
        $ 4 ~ 3     :       /1, /4, /16, /64
        $ 2 ~ 0     :       BIT8, BIT9, BIT10, BIT11, BIT12, BIT13, BIT14, BIT15

ADCM        IO_RW           0x21
        $ 7 ~ 5     :       8BIT, 9BIT, 10BIT, 11BIT, 12BIT
        $ 3 ~ 1     :       /1, /2, /4, /8, /16, /32, /64, /128

INTEN           IO_RW           0x04
        $ 3     :       X, AD
        $ 2     :       X, T16
        $ 1     :       X, PB0
        $ 0     :       X, PA0
```

在 T16M 的定义中，可以知道 T16M 由 3 种栏位组成，而关键字 STOP, SYSCLK, IHRC, EOSC, ILRC, PA0 皆可用在 $ T16M 的语法中。而 X 表示不存在的状态，如 $ 7 ~ 5 : STOP, SYSCLK, X, X, IHRC, EOSC, ILRC, PA0，相同的，/1, /4, /16, /64，BIT8~BIT15 等关键字，也皆可用在 $ T16M 的语法中。

在 INTEN 的定义中，有数种中断来源，如 AD、T16、PB0、PA0。如果不想使用某种中断来源，只要不去宣告使用，缺少的栏位，组译器将以 0 取代该栏位。

如下例：

```
$ INTEN     PA0;            //    INTEN = 0001B，只有 INTEN.PA1 = 1，其余为 0。
$ INTEN     PB0, AD;        //    INTEN = 1010B，只有 INTEN.PA1/AD = 1，其余为 0。
```

当然，如果你只想改变 INTEN 中的某些栏位，也可以用如下语法：

```
INTEN.PA0   =   1;          //    等同于  set1 INTEN.0
INTRQ.T16   =   0;          //    等同于  set0 INTRQ.2
```

在 $ IO xx, xx 的语法中，如果少了某些栏位，组译器将以预设值(通常是 0)取代该栏位。如下例：

```
$ T16M SYSCLK, /16;     //    等同于 $    T16M SYSCLK, /16, BIT8;
$ T16M STOP;        //    T16M =      0;      // STOP, /1, BIT8
```

但是也有例外，以 P232 / P234 / P201A 中的 MISC 暂存器的 Bit 6 为例：

```
MISC        IO_WO               0x3B
    $ 6     :       EC_High_Drive, EC_Normal_Drive      :       MUST
                    // You can't loss the item, must select one item.
    $ 5     :       X, Fast_Wake_Up
    $ 4     :       X, En_LCD   //      PA0/3/4, PB0 support VDD/2
```

你一定要选择 MISC 的 Bit 6 为：外部 Crystal 要用加强推力(0)或正常推力(1)；组译器不使用预设值，因为一般来说，预设值 0 是代表 Off，但 MISC 的 Bit 6 为 0 是代表选用加强推力；为了避免使用上的误解，该栏位一定要由使用者作二选一的动作。如下例：

```
    $       MISC  EC_High_Drive, Fast_Wake_Up; //      MISC = 0010_0000B;
    $       MISC  EC_Normal_Drive;                      //      MISC = 0100_0000B;
```

请不要自行更改 PDKxxxx.INC 的内容，否则，会造成组译的错误。

1.6.3. INC 格式补充

在暂存器的定义中，常可以见到如下范例：

```
FPPEN       IO_RW       0x01        //      IO_RW = 可以读写。
ADCR        IO_RO       0x22        //      IO_RO = 只可以读取。
ADCDI       IO_WR       0x24        //      IO_WR = 只可以写入，但在仿真器可以看到内容。
PAOD        IO_WO       0x13        //      IO_WO = 只可以写入，仿真器也看不到内容。
PBOD        IO_XX                   //      IO_XX = 仿真器不支援该暂存器。
```

有时，定义后面会接一数值，如下范例：

```
    PAPH        IO_WR       0x12 : 0xDF  //      0xDF = 1101_1111B，少了 0010_0000B，
                                         //      所以只有 Bit 5 不支援 IO_WR 功能。
    PA          IO_RW       0x10 : 0xDF  //      0xDF = 只有 0x40 (Bit 5) 不支援 IO_RW 功能。
    PA5         IO_RO       PA.5         //      但是 PA5 可以被读回来
```

虽然 PDKxxxx.INC 的内容，总是有一些特殊关键字，但它的目的，主要在减少使用者的误用，所以不用太在意这些特殊字词。

1.6.4. 其它的转换指令

在一些应用程序中，常会看到一些特殊语法，如下范例，以下分别介绍它们的功能。

```
BIT     I2C_SDA         :       PA.0;       //      I2C 的 Data Pin
BIT     I2C_SCK         :       PA.1;       //      I2C 的 Clock Pin
BIT     p_Key_In        :       PA.2;       //      Direct Key
BIT     p_LED_Out   :       PA.3;       //      LED Output
```

写法一: 将 Output Pin (I2C_SDA, I2C_SCK, p_LED_Out) 设为 Output。

```
PAC     =       _FIELD (I2C_SDA, I2C_SCK, p_LED_Out);           //      PA.0 | PA.1 | PA.3 = 1101B
```

写法二: 将 Input Pin (p_Key_In) 以外的 Pin 设为 Output。

```
PAC     =       0xFF - _FIELD (p_Key_In);                       //      PA.2 = 0100B
```

_FIELD(变量 Bit)会将括号内所有 bit，转成相对应的栏位，如上例：I2C_SDA = PA.0 = 1<< 0 = 1，I2C_SCK=PA.1=1<<1 =2，p_LED_Out=PA.3=1<<3=8，依此类推，再全部集结成一个常数，传回程序中。

_FIELD(变量) 也有另一种用途，如下例：

```
BYTE            Var;
BIT             Bit1    :       Var.1;
BIT             Bit3    :       Var.3;
```

_FIELD (Var) 等于 0000_1010B，因为 Var 有两个 Bit 宣告(Bit1 & Bit3)，所以 _FIELD (Var) 会等于 (1 << 1) | (1 << 3) = 0000_1010B，在介绍 T16M 用法的应用须知中，也刚好用到。

在应用须知中，介绍 T16M 时，有一个参考程序 T16_Key_LED.C，裏面宣告一个 Key_Flag，记录 p_Key_In 的前次状态，因此程序中，利用 PA 与 Key_Flag 作 XOR 运算，再把用到的栏位作 AND 运算，就知道 p_Key_In 是否改变状态了。

```
BYTE Key_Flag;
…
A       =       (PA ^ Key_Flag) & _FIELD (p_Key_In);   //      only check the bit of p_Key_In.
…
if (! ZF)
        {       …
                Key_flag            ^=      _FIELD (p_Key_In);
```

当然，万一 p_Key_In 不在 PA 时，怎么办呢？你可以利用_VAR( )转换指令来求得 p_Key_In 的主变量。

```
A=      (_VAR (p_Key_In) ^ Key_Flag) & _FIELD (p_Key_In); //      only check the bit of p_Key_In.
```

我们可以宣告一个 f_Key_In 变量，它与 p_Key_In 变量，使用相同的 Bit，只要利用_BIT( )转换指令即可：

```
BIT             f_Key_In            :       Key_Flag._BIT(p_Key_In);        //      PA.2    =>  Key_Flag.2
```

再检查 f_Key_In，就知目前的按键状态。

当然，一个 Byte 的变量，只用了一个 Bit 是很浪费的，我们可以宣告一些 BIT 变量，放在尚未用到栏位中。

```
BIT         f_Key_Trig  :   Key_Flag.?;          //  Key_Flag 的其它 Bit，则作其它 BIT 变量用。
                            //  位置由系统自行决定。
```

在 I2C 范例中，I2C_SDA 是须要 Input 与 Output Low 互相切换，你可以利用_PXC( )转换指令，将 PA.0，转换成 PAC.0，达到控制 Input 与 Output，如下例所示。

```
do
{
    $   I2C_SCL     Low;
    data        <<=     1;
    swapc _PXC (I2C_SDA);                   //  if  (CF),   then Output Low
    $   I2C_SDA     Low;                    //  else            then Input

    $   I2C_SCL     High;
} while (--count);
```

在以前的一个 I2C 范例中，在 PDK22 系列都会组译失败，如下例。

```
$   I2C_SDA     Low,    Pull,   In;
```

这是因为在 PDK22 系列中，PAPH 是 Write Only，不允许作 SET1 PAPH.0 (请参考应用须知中，介绍 IO 的文章)，因此，程序须改成：

```
PAPH        =       _FIELD (I2C_SDA);
$   I2C_SDA     Low,    In;
```

当然，万一 I2C_SDA 不在 PA 时，怎么办呢？你可以利用_PXPH( ) 转换指令，将 PA.0 转换成 PAPH.0；再利用_VAR( )转换指令，将 PAPH.0 转换成 PAPH，于是得到 I2C_SDA 的 Pull High 暂存器，程序如下：

```
_VAR (_PXPH (I2C_SDA))=     _FIELD (I2C_SDA);
$   I2C_SDA     Low,    In;
```

你可能会想，如何自动区分 PAPH 是 Write Only 或可 Read+Write，好让程序更简易一点，因此，你会用到_IO_RW( )的转换指令，测试括号内的 IOBit 变量，是否支援 Read+Write，程序如下：

```
#if     _IO_RW (_PXPH (I2C_SDA))            //  PAPH 是可 Read+Write
    $   I2C_SDA     Low,    Pull,   In;
#else                                       //  PAPH 是 Write Only
```

```
    _VAR (_PXPH (I2C_SDA))=     _FIELD (I2C_SDA);
    $       I2C_SDA     Low,    In;
#endif
```

不过说了半天，似乎搞得太复杂了；在 I2C 的界面中，为了传输速度，外挂电阻是必须的，所以，内部的 Pull High 是可以省略的，因此，你只要写成如下就可以了。

```
    $       I2C_SDA     Low,    In;                     //      Pull High 是可省略的。
```

在类似语法中，底下的范例是在检查，PAPH 是否有 Read 的属性。

```
    #if     _SYS (IO_R：PAPH)
        …
    #endif
```

在 PDK82 与 PDK22 转换中，有些指令是 PDK22 不用的，如 PMODE 的设置，你可以用如下的语法区分它们。

```
    #if     _SYS (SERIAL) == 82                     //      如果现在使用的 CHIP 是 PDK82 系列
            pmode       6                           //      Use    1/4, 1/4, 1/4, 1/4 to debug
    #endif
```

你也可以认为，只有双核的 FPPA 不用 PMODE，如 PDK22 / P23X 系列。

```
    #if     _SYS (FPPA) > 2                         //      多核的 FPPA ( >= 3)，都需设定 PMODE
            pmode       6                           //      Use    1/4, 1/4, 1/4, 1/4 to debug
    #endif
```

当然，你也可以检查目前使用的 CHIP 名称：@NOW_CPU_NAME，来作语法区分，只是 CHIP 的名称，也可能是 PDK82C12 或 PDK82C10 或…，种类很多。

```
    #ifidni   <@NOW_CPU_NAME>, <PDK82C13>
            pmode       0x06                        //      Only use FPPA0 ~ FPPA3
    #endif
```

关于_SYS 语法，还有

```
    Var     =>      _SYS (RAM_SIZE);        //      得到目前使用 IC 的 RAM Size (单位为 Byte)
    Var     =>      _SYS (ROM_SIZE);        //      得到目前使用 IC 的 ROM Size (单位为 Word)
    Var     =>      _SYS (ADR_ROLL);        //      得到 Roll Code 的起始位址
    SP      =       _SYS (STACK:n);         //      重新设定第 n 个 FPPA, SP 的值为预设的配置起始位址。
                                            //      请参考 7-6-3 : 得到系统设定的堆栈位址
```

1.6.5. 混合语言的限制

基本上在 Mini-C 型式的项目中，C 与 ASM 语言是可以任意共享的，但是在下列情形还是需受到限制：

(1). 在 ASM 项目中的 call procedure，在 Mini-C 项目中必须写为 procedure()。

(2). 在 Mini-C 项目中的 goto label，其中的 label 必须在该程序内。

(3). 所有的变量不分大小写。

1.7. WORD 的特殊应用

1.7.1. 使用 PUSHW 搬移

(1) WORD 的搬移 ： 见如下范例：

```
WORD    ww1, ww2;
ww1     =   ww2;        // 将 ww2 搬移到 ww1。
```

在 Mini-C 中，它会组译成

```
mov     A,      ww2$0;
mov     ww1$0, A;
mov     A,      ww2$1;
mov     ww1$1, A;
```

不过用户如将它改成如下范例：

```
pushw   ww2;
popw    ww1;
```

虽然都需要 4 个指令时间。但却可节省两个脚本。

(2) WORD 的互换 ：

```
_swap ( ww1, ww2 );     // 在 Mini-C 中，暂不支持此函数。
```

用户可以用如下范例完成此功能：

```
pushw   ww2;
pushw   ww1;
```

```
popw        ww2;
popw        ww1;
```

(3) 在不同 FPPA 中互传 WORD 的资料 ：

在不同 FPPA 中互传 WORD 的数据，有可能因同步的问题，使第一个 FPPA 才接收完 Low Byte，还没来得及接收 High Byte，数据又被第二个 FPPA 修改了。

你可以利用 pushw / popw 的指令，让 WORD 的数据可以一次搬移完成，不会有 High / Low Byte 不同步的问题。

如下范例，想让 FPPA0 的 ww0 传送到 FPPA1 的 ww1。

```
WORD    ww_T;                   // 暂时传送用的 Buffer。
void    FPPA0   (void)
{
    // …
    WORD    ww0;
    ww0     =   …+-&|^…;    // 由使用者表达式，得到 ww0。
    pushw   ww0;
    popw    ww_T;           // ww0 传送到 ww_T。
    // …
}
void    FPPA1   (void)
{
    // …
    WORD    ww1;
    pushw   ww_T;           // 从 ww_T 传送到 ww1。
    popw    ww1;
    xx  =   ww1 …+-&|^…;    // 由使用者表达式，继续处理 ww1。
    // …
}
```

1.7.2. 不用 PUSHW 搬移

在双核心系列中，没有 PUSHW / POPW 指令，但双核会轮流执行指令，所以要作 WORD 的资料互传时，只要用下面的小技巧，就可以完成同步更新。

(a) 先完成 Low Byte 运算，如 Low Byte = A 或 A = Low Byte。

(b) 中间经过一个运算指令。

(c) 再完成 High Byte 运算，如 High Byte = A 或 A = High Byte。

范例： WORD 的互传 (一)，接收端 FPPA 会收到旧数据 0x1234

传送端 FPPA	接收端 FPPA
WORD transfer = 0x1234;	WORD receive;
A = 78h;	.
.	A = transfer$0; // 0x34
transfer$0 = A;	.
.	receive$0 = A;
A = 56h;	.
.	A = transfer$1; // 0x12
transfer$1 = A;	.
.	Receive$1 = A;

范例： WORD 的互传 (二)，接收端 FPPA 会收到新数据 0x5678

传送端 FPPA	接收端 FPPA
WORD transfer = 0x1234;	WORD receive;
A = 78h;	.
.	....
transfer$0 = A;	.
.	A = transfer$0; // 0x78
A = 56h;	.
,	receive$0 = A;
transfer$1 = A;	.
.	A = transfer$1; // 0x56
....	.
.	Receive$1 = A;

如果依照此范例，不管传送端或接收端的先后如何，都可以接收到正确的 WORD 信息。

不过也要小心，避免有中断中途发生，破坏了数据互传时的同步性。

1.7.3. 中断时的搬移

如果在 FPPA0 与中断之间，互传 WORD 的数据，除了使用 PUSHW/POPW 来传递数据外，你也可以使用 DISGINT 来同步中断，使修改 WORD 的动作不会受到干扰，如下范例。

```
WORD      ww;
void      FPPA0     (void)
{
    //  …
    DISGINT;
    ww   =    ww + …;
    ENGINT;
    // …
}
```

但有些时候，你希望中断程序不要受 DISGINT 干扰，而由中断程序产生的数据，又可被 FPPA0 正确的读取，你可改用如下范例。

```
WORD      ww;
void      Interrut      (void)
{
    //  …
    ww   =    …;                    //  修改数据
    //  …
}

void      FPPA0     (void)
{
    //  …
    WORD      ww_R;
    while (1)
    {
        ww_R      =    ww;
        if (ww_R == ww) break;   //  比对数据
    }
    //  …
}
```

如果连续检查 WORD 的数据一致，表示读到的资料是正确的了。

1.7.4. 指针

WORD 可以当 Point 使用，如下范例，利用 pnt1/pnt2，将 buf2 的数据搬移到 buf1。

```
WORD      pnt1, pnt2;
BYTE      buf1 [0x10],   buf2 [0x10];

pnt1      =    buf1;    pnt2      =    buf2;
while (1)
{
    //  …
    *pnt1    =    *pnt2;
    pnt1++;       //  inc       pnt1$0   需要两个指令
                  //  addc      pnt1$1
    pnt2$0++;     //  inc       pnt2$0   只需一个指令
    // …
}
```

在 RAM 空间小于 256 个大小的 IC 中，增加 point 的 High Byte 并没有意义，所以，请直接使用 point$0++ 取代 point++，将可节省 1 个指令空间。

如下例，你也可以用 WORD 当 Point 时，用来读取 ROM 空间中的 Rolling Code。

```
WORD      Point, L_Roll, H_Roll;

Point     =    _SYS(ADR_ROLL);       // 利用内建的常数 _SYS(ADR_ROLL) 来取得
                                     // 地址，一般来说，地址应该在 xxxA。
L_Roll    =    *Point$W;

Point$0++;                           //  所以，High Point 就不须要再加了。
H_Roll    =    *Point$W;
```

读取 ROM 的方法，一般有如下三种语法：

- *Point$L　　(取 Low Byte)。
- *Point$H　　(取 High Byte)。
- *Point$W　　(取 WORD)。

1.7.5. 清除 RAM

以下方法纯属介绍，更好的方法请参考 .ADJUST_IC 的特殊选项。

有些人习惯在程序初始的地方，将所有的 RAM 清为 0；因此，你必须要有一个 Point，而且这个 Point，必须寻址在 RAM 地址为 0 的地方，然后将 Point 的内容，设定为 RAM 的最高字节的地址，再用一个循环，逐步将 RAM 全清为 0。

如下范例，利用 .RAMADR 0 的语法，将 Point 变量的地址定为 0；至于其它变量的地址，就恢复成由系统自行决定，语法为 .RAMADR SYSTEM。

```
.RAMADR     0               //  RAM Address 0
WORD        Point;

.RAMADR     SYSTEM          //  后面变量的地址，由系统自行决定。
```

再由内建的常数 _SYS(SIZE.RAM)，可以得到 RAM 的最高字节的地址。

```
Point   =   _SYS(SIZE.RAM) - 1;
```

如果 RAM 的大小，小于 256，可以用如下的循环范例，组译出最短的 Code。

```
A       =   0;
do
{
    *Point  =   A;
} while (--Point$0);            //  循环被组译成：  DZSN    Point$0
                                //                  GOTO    $ - 2
```

如果 RAM 的大小，大于 256，以下的循环范例，可以将内容全清为 0。

```
do
{
    *Point  =   0;
} while (--Point);          //  循环被组译成：  DEC     Point$0
                            //                  SUBC    Point$1
                            //                  MOV     A, Point$1
```

```
//                  OR      A, Point$0
//                  T1SN    ZF
//                  GOTO    $ - 6
```

改用如下的循环范例，可以组译出最短的 Code (比上面范例少 2 WORD)。但 Point (RAM[0, 1]) 的内容最后为 0xFFFF，是唯一不为 0 的地方。

```
A       =   0;
do
{
    *Point  =   A;
} while (Point--);          //  循环被组译成：  DEC     Point$0
                            //                  SUBC    Point$1
                            //                  T1SN    CF
                            //                  GOTO    $ - 4
```

1.8. 其他讨论

1.8.1. #PRAGMA

使用 #PRAGMA 指令，可以做一些编译的特殊控制，如下介绍：

在 Mini-C 的项目中，有些指令会自动使用一些 Local Memory 当作运算用，

如

```
.DELAY  10000;
A   =   *Point$H + *Point$L;
```

但在作多次烧录时，如果你想自行分配所有 Memory 的资源，
你必须除能这种语法，以免变量的排序地址不如预期。

如果要除能这项检查，必须在 .PRE 档案中，加入以下指令才可。

```
#PRAGMA  DISABLE  SYS_LOCAL
```

1.8.2. _SYS

不同系列的 IC，所支持的指令集会有一点差异，你可以用底下的方法，判断 IC 是否有支持特定的指令。

语法：_SYS(OP:指令) :　如果指令存在，则回传 1。

指令的格式，采用各系列 IC 的 datasheet 的指令集命名方法。

```
#if _SYS (OP:SWAPC IO.n)                // 判断是否有 swapc IO.n 的指令
    swapc_   EQU      swapc             // 将 swapc_ 直接代换成 swapc 指令
#else
    swapc_   macro    iob               // 以宏指令取代 swapc 指令
        iob =    0;
        t0sn     CF;
        iob =    1;
    endm
#endif

swapc_   IO_Bit;                        //  swapc_ 就可变成通用命令。
```

其他类似命令，如
```
_SYS (OP:COMP A I)        //  A 跟立即值比较
_SYS (OP:PUSHW index)     //  pushw word_memory
_SYS (OP:SWAP M)          //  swap   byte_memory
_SYS (OP:XOR IO A)        //  xor    register, A
_SYS (OP:STOPEXE)         //
_SYS (OP:PMODE N)         //
```

都可以用来判断指令是否支持。

关于 _SYS 语法，还有

```
Var =>   _SYS (SIZE.RAM);      //  得到目前使用 IC 的 RAM Size (单位为 Byte)
Var =>   _SYS (SIZE.ROM);      //  得到目前使用 IC 的 ROM Size (单位为 Word)
Var =>   _SYS (ADR.ROLL);      //  得到 Roll Code 的起始地址
Var =>   _SYS (ADR.IHRC);      //  得到 IHRC 的校正地址
SP   =   _SYS (STACK:n);       //  重新设定第 n 个 FPPA 的 SP 的值为
                               //  默认的设定起始地址。
```

1.8.3. 其它的转换指令

在一些应用程序中，常会看到一些特殊语法，如下范例，以下分别介绍它们的功能。

```
BIT I2C_SDA     :   PA.0;       //  I2C 的 Data Pin
BIT I2C_SCK     :   PA.1;       //  I2C 的 Clock Pin
BIT p_Key_In    :   PA.2;       //  Direct Key
BIT p_LED_Out   :   PA.3;       //  LED Output
```

写法一: 将 Output Pin (I2C_SDA, I2C_SCK, p_LED_Out) 设为 Output。

```
PAC =   _FIELD (I2C_SDA, I2C_SCK, p_LED_Out);
        //  PA.0 | PA.1 | PA.3 = 1101B
```

写法二: 将 Input Pin (p_Key_In) 以外的 Pin 设为 Output。

```
PAC =   0xFF - _FIELD (p_Key_In);   //  PA.2 = 0100B
```

_FIELD (Bit 变量) 会将括号内所有 bit，转成相对应的数值，如上例：

I2C_SDA = PA.0 = 1 << 0 = 1，I2C_SCK = PA.1 = 1 << 1 = 2，
p_Key_In = PA.2 = 1 << 2 = 4，p_LED_Out = PA.3 = 1 << 3 = 8，

依此类推，再将全部数值加总为一个常数，即得结果。

_FIELD (非 Bit 变量) 也有另一种用途，如下例：

```
BYTE    Var;
BIT     Bit1    :   Var.1;
BIT     Bit3    :   Var.3;
```

_FIELD (Var) 等于 0000_1010B，因为 Var 已经宣告了两个 Bit 变量 (Bit1 & Bit3)，所以 _FIELD (Var) 会等于 _FIELD (Bit1, Bit3) 会等于 (1<<1) | (1<<3) = 0000_1010B，在介绍 T16M 用法的应用须知中，也刚好用到。

在应用须知中，介绍 T16M 时，有一个参考程序 T16_Key_LED.C，裏面宣告一个 Key_Flag，记录 p_Key_In 的前次状态，因此程序中，利用 PA 与 Key_Flag 作 XOR 运算，再把用到的 BIT 作 AND 运算，就知道 p_Key_In 是否改变状态了。

```
BIT       p_Key_In     :    PA.2;          //   Direct Key
BYTE      Key_Flag;                        //   Bit 2 for p_Key_In
…
A    =    (PA ^ Key_Flag) & _FIELD (p_Key_In);
//   only check the bit of p_Key_In.
…
if (! ZF)
{    …
     Key_flag      ^=   _FIELD (p_Key_In);
```

当然，p_Key_In 宣告可能在任意 IO Port，你可以利用 _VAR( ) 转换指令，来求得 p_Key_In 的真正 IO Port。

```
A    =    (_VAR (p_Key_In) ^ Key_Flag) & _FIELD (p_Key_In);
//   only check the bit of p_Key_In.
```

我们可以宣告一个 f_Key_In 变量，想要与 p_Key_In 变量，使用相同的 Bit，只要利用 _BIT( ) 转换命令即可知道该 Bit 的位置：

```
BIT       f_Key_In     :    Key_Flag._BIT(p_Key_In);
//        PA.2          =>   Key_Flag.2
```

当然，一个 Byte 的变量，只用了一个 Bit 是很浪费的，我们可以宣告一些 BIT 变量，放在该 Byte 的变数中。

```
BIT       f_Key_Trig   :    Key_Flag.?;
//   Key_Flag 的其它 Bit，则作其它 BIT 变数用。
//   位置由系统自行决定。
```

在 I2C 范例中，I2C_SDA 是须要 Input 与 Output Low 互相切换，你可以利用_PXC( ) 转换命令，将 PA.0，转换成 PAC.0，达到控制 Input 与 Output，如下例所示。

```
do
{
     $    I2C_SCL   Low;
     data      <<= 1;
     swapc     _PXC (I2C_SDA);       //   if (CF),      then Output Low
     $    I2C_SDA   Low;             //   else          then Input
     $    I2C_SCL   High;
```

```
    } while (--count);
```

类似转换命令：_PXPH ( )，可以将 PA.xx，转换成 PAPH.xx。

1.8.4. 重新组译 / 指定输出

在有些时候，你可能须要将一个项目，同时组译出不同的编码，并储存成不同的 .PDK 档案。

如范例项目 [Menu]->[File]->[Demo Project]->[OutFile.PRJ]，就可以组译出两种编码。

使用了指令：

```
.ReAssembly      ->   须要再重新组译。
.OutFile         ->   设定输出档名。
_SYS(ASM_LOOP)   ->   获得已经重新组译的次数。
```

组合如下：

```
NOW_SEL      =>      _SYS(ASM_LOOP)

#if      NOW_SEL == 0
    .OutFile     "Out_File_1.PDK"
    .REASSEMBLY                          //   还要再重新组译。
#elseif NOW_SEL == 1
    .OutFile     "Out_File_2.PDK"     //   最后一次组译。
#endif
```

.OutFile 格式如下：

```
.OutFile     Your_Name.PDK          //   自行决定输出名称

.OutFile     "Your_Name.PDK"      //   用 "..." 括住档名是好习惯，一样支持

.OutFile     XXX_%S.PDK             //   %S 会被置换成 .CHIP 宣告的明称

.OutFile     XXX_%X.PDK             //   %X 会被置换成 Check_Sum

.OutFile     XXX_%T.PDK             //   %T 会被置换成目前的 年月日 yymmdd
```

.OutFile XXX_%Tddmmyy.PDK // %Tddmmyy 会被置换成 日月年 ddmmyy

.OutFile XXX_%THHMMSS.PDK // %THHMMSSy 会被置换成 时分秒 HHMMSS

// %T 后的 yy mm dd MM HH SS 年月日时分秒 可自行组合

2. Asm & Mini-C 的讨论

在 Mini-C 型式的项目中，程序的进入点，是在程序 FPPA0；当打开反组译视窗，才能看到 FPPA0。从 CODE – 0 的位址 Goto FPPA0，再作一些初始化，才进入使用者程序。以下是 Asm 与 Mini-C 型式的项目比较表：

Asm 型式的项目	Mini-C 型式的项目
一个主档案	多个模块档案与一个前置档 (.PRE)
一次组译全部	组译各个档案，再全部连结
无最佳化	支援最佳化
变量位址从 0 递增	系统安排变量位址
使用者自订堆栈空间	系统安排堆栈空间
使用者自订 ROMADR 0~7，0x10	系统提供 FPPA0 ~ 7，Interrupt 等副程序
无程序要求	程序须撰写在程序内

2.1. Mini-C 的前置档(.PRE)

此档案记录着 .CHIP xxxx, FPPA0~7 以及 Interrupt 的宣告。使用者可以改选.CHIP，如 PDK22C13 或其它；也可以删除 CodeOption，系统在下次组译时会重新用选单要求使用者输入；至于其它是内定的标准格式，建议使用者不要任意更改内容，以免造成组译时的问题；以下是前置档 (.PRE)的内容：

```
. CHIP      PDK82C13

//{{PADAUK_CODE_OPTION
      .Code_Option      Security      Disable            // Security Disable
//}}PADAUK_CODE_OPTION

      . JMP      FPPA0      <?>, 0
      . JMP      FPPA1      <?>, 1
      . JMP      FPPA2      <?>, 2
      . JMP      FPPA3      <?>, 3
      . JMP      FPPA4      <?>, 4
      . JMP      FPPA5      <?>, 5
      . JMP      FPPA6      <?>, 6
      . JMP      FPPA7      <?>, 7

. ROMADR  0x10
      . PUT      Interrupt <reti>, 8
```

在 PDK22/ P234 / … 系列中，只有两个 FPPA 可以使用，所以系统会自动把多余的 .JMP FPPAx <?>, x 忽略，使用者可以不用理会。

2.2. Mini-C 的最佳化

```
范例一：        while  (PA.0)   NULL;
组译成 ：       wait0          PA.0
范例二：        if     (PA.0 && PA.1)     goto   xxx;
组译成 ：       bts1   PA.0
                bts0   PA.1
                goto   xxx
```

关于更多最佳化的例子，请参考 “最佳化的讨论”。

2.3. Mini-C 的变量位址

变量预设的位址是由系统决定，如果资料型态为 WORD，则自动对齐偶数边界。当使用者想自订位址时，可以使用下列方法，范例如下：

```
. RAMADR  100h                    //     进入使用者订址, 并订为 0x100
BYTE      Buffer [100h];
. RAMADR  SYSYTEM                 //     回复系统订址
```

以上例来说明，当 RAM 100h ~ 1FFh 已被使用者所定义，其余变量将不会再使用此区的 RAM。不过自订位址时， 请注意 WORD 须对齐偶数边界，这是硬件指令的限制，如 IDXM ww。

2.4. Asm 项目的暂时缓冲器

在 Asm 型式的项目，变量预设的位址是由 0 递增，或由 .RAMADR 命令自行订定位址。但是，在需要暂时缓冲器的表达式中，用 Asm 型式的项目，是会有问题的，举例如下：

例一: “A = PA + PB”，理论上该组译成如下汇编语言：

```
mov     a, pa
mov     暂时缓冲器, a
mov     a, pb
add     a, 暂时缓冲器
```

例二 ： “If (A >= PA)”，理论上该组译成如下汇编语言：

```
mov     暂时缓冲器, a
mov     a, pa
```

comp a, 暂时缓冲器

从上二例来看，暂时缓冲器的使用是必要；但又无法让使用者掌控，在 8 个 FPP 的环境中，这些暂时缓冲器可能互相干扰。**IDE** 对于如上的语法，都将给予错误讯息，高度建议使用者利用 Mini-C 来开发程序，上述的问题系统会自动帮使用者安排解决。

2.5. 自由的跳跃

在一般的 C 项目中，是不允许程序利用 GOTO 指令，从一个程序中跳跃到另一个程序里，如下范例：

```
void    Func1 (void)
{
        ....
Lab1:   ....
}

void    Func2 (void)
{
        ....
        goto   Lab1;
}
```

但是对某些应用来说，自由的跳跃是有其必要性，以下面程序为范例：

```
void    Func1 (void)
{
        A      =      1;
Lab1::                          //      利用 : : 符号，将 Lab1 变成全域 Label
        ....                    //      底下为共通的子程序码
}

void    Func2 (void)
{
        ....;
        goto   Func1;           //      虽然照理来说，应该使用 Func1( ); 才对。
}                               //      但是直接用 goto Func1;  却可以少一次堆栈使用。

void    Func3 (void)
{
        A      =      2;
        goto   Lab1;            //      虽然照理来说，应该将 Lab1 设计成一个副程序才对。
                                //      但直接跳到 Lab1，却可以少一个副程序的堆栈使用。
}

void    Func4 (void)
{
        A      =      3;
        call   Lab1;            //      既然可以 goto Lab1，那么 call Lab1 也必须要支援了。
        ....
}

Lab5:   A++;                    //      这是一个完全跳离 C 结构的语法
        ....
```

```
void    Func5 (void)
{
        ….;
        gotol  Lab5;        //    如果使用者跳离 C 结构的保护，那么所有存储器的分配都可能出错，
}                           //    请勿作此尝试。
```

在 IDE 0.37 版本以后，就支援设计者自由的程序跳跃。

2.6. 堆栈的计算

2.6.1. Mini-C 之堆栈处理

在 ASM 型式的项目中，程序设计师必须定义堆栈的深度，以下面程序为范例：

```
    . ROMADR  0
    GOTO      FPPA0
    GOTO      FPPA1
    GOTO      FPPA2
    ...

    . RAMADR  0                     // Address 必须小于 0x100
    WORD          Stack1 [2]        // 2 个 WORD
    WORD          Stack2 [1]        // 1 个 WORD
    ...

FPPA0:
    MOV       A, 0x20               // 直接定位址，这是不好的习惯
    MOV       SP, A
    ...

FPPA1:
    MOV       A, LA @ Stack1[0]     // 指定 Stack1 给 FPPA1 用
    MOV       SP, A                 // 最多可 Call 两层 (因为 Stack1[2] )
    ...

FPPA2:
    SP    =   Stack2;               // 指定 Stack2 给 FPPA2 用
    …                               // 最多可 Call 一层 (因为 Stack2[1] )
```

在以上范例中，看到了下面的问题：

(1). 堆栈用了几层 ？

(2). 堆栈要放在哪里？

(3). 堆栈的其它限制？

但在 Mini-C 型式的项目中，这些问题被简化了。请执行 ICE，并看 Fig. 6-1 反组译视窗。

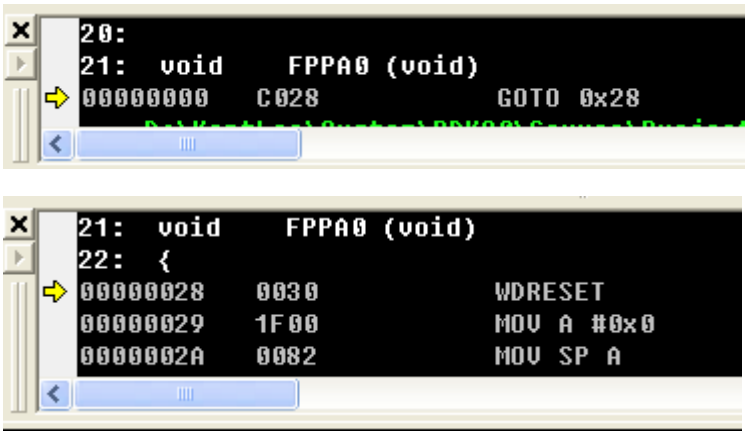


Fig.6-1. 反组译视窗

在进入你的程序之前，系统帮你作了下面的事：

(1). WDRESET：配合硬件架构之动作。

(2). 设定堆栈空间：系统帮我们计算堆栈大小，并配置空间给它。系统堆栈的计算，是依据下面的规则：

- 因为 FPPA0 负责 Interrupt，当 Interrupt 被使用时，

  Stack 总数 = (FPPA0 程序的堆栈) +
  (Interrupt 程序的堆栈) +
  (Interrupt 动作占一层堆栈)

- 使用指令 PUSHW RAM / PUSHW PCX / PUSHAF 时，亦当作呼叫一层堆栈。不过如果你用了多个 PUSH.. 指令，系统则无法帮忙分析用了几层堆栈，只能假设用一层堆栈，请自行检查与注意。

2.6.2. IDE 环境下之堆栈

在 Workspace 视窗的 Label View 的 Call View 中，显示所有程序的互属关系，并会对特殊指令的使用，显示讯息，供使用者参考。如 Fig. 6-2 所示，FPPA0( )呼叫了 LowVoltageDetect( )，共享了一层堆栈 (1 stack)。

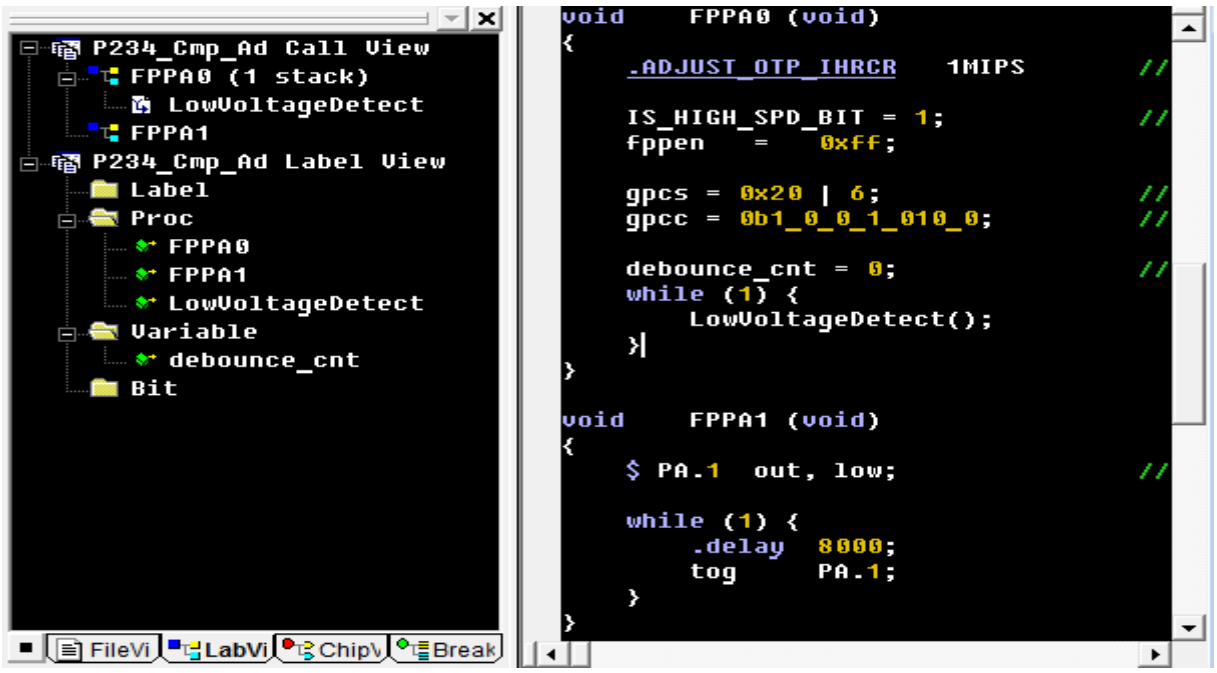


Fig.6-2. 堆栈讯息

2.6.3. 得到系统设定的堆栈位址

在有些场合，你可能须要知道 FPPA 的堆栈起始值，以方便对 FPPA 的堆栈再度初始化。利用语法_SYS(STACK:n)，n = 0~7 的 FPPA 编号，你就可以获得这个信息。

范例：

```
void    FPPA1 (void)
{                                          //    在进入程序前，系统已经帮你作 SP 的初始化。
    ....
FPPA1_A::
    SP    =    _SYS(STACK:1).              //    但是因为 FPPA0 的 POPW   PC1 指令，
    ....                                   //    使得 FPPA1_A 成为 FPPA1 的另一个起始点，
    ....                                   //    设计者必须重新对 SP 作初始化。
}

void    FPPA0 (void)
{
    ....
    WORD    point   =    FPPA1_A;
    PUSHW   point;
    POPW    PC1;                           //    FPPA0 更改 FPPA1 的行程。
    ....
}
```

2.6.4. 自定堆栈深度的方法

如何自订堆栈的深度? 用实际的范例来展示：

范例一：

```
void    FPPA0 (void) : Stack = 0x10      // 强制配置 0x10 个 WORD 的堆栈
{
}
```

范例二：

```
void    FPPA1 (void) : Stack = 0         // 不允许使用堆栈
{
}
```

范例三：

```
· RAMADR    0x8                          // 使用者手动定址
WORD        Stack0 [0x10];       // 为了确定堆栈的位址小于 0x100, 所以不用系统自动定址
· RAMADR    SYSTEM                       // 回复系统自动定址
void    FPPA2 (void) : Stack = ?         // 要求系统不要配置堆栈空间
{
    MOV         A, LA @Stack0[0]    // 使用者自行设定堆栈
    MOV         SP, A
        //      ...
}
```

要特别注意的是 `Stack = value` 的语法，只限使用在 FPPA0~FPPA7。使用者如果是采用自订堆栈深度的方式, 则系统不会去检查堆栈的深度是否足够程序使用? 在 Workspace 视窗的 Break View 的 IPx Stack 项目中，可以设定各个 FPP 的堆栈侦测开关，使 ICE 能侦测到 Stack Under，Stack Over 以及 Stack Trap 三种情形，下面是 IDE 在堆栈侦测的三种选项。

说明如下：

（1）Max：只检查堆栈是否在 RAM 的合理空间中

（2）User：由使用者自行决定堆栈的低值与高值，例如：10 ~ 1F，代表 RAM 10 ~ 1F 的 8 个 Word 为堆栈的范围，当使用者自订的堆栈高低值有错误时，系统会显示错误讯息通知使用者，并参考输入值，强制让堆栈高低值合乎规定，且低值为偶数高值为奇数。

（3）Auto：由系统自动设定堆栈检查开关，有下三种可能性：

- Stack --：只检查堆栈是否在 RAM 的合理空间中

  当有 "递回呼叫"，ICALL 或 IGOTO 的指令，或使用者自订堆栈（"Stack = ?")时，系统自动设定此项。

- Stack Null：不允许使用堆栈

  当系统分析出程序并未用到堆栈，或使用者自订堆栈（"Stack = 0"），系统自动设定此项。如果 ICE 侦测到例外情形，将送出 Stack Trap 讯息。

- Stack ll ~ hh：设定堆栈的低值与高值

  将系统分析出程序的堆栈深度或使用者自订堆栈（"Stack = n"），设定到低值与高值。如果 ICE 侦测到例外情形，将送出 Stack Under 或 Stack Over 讯息。

范例四: // 一般函式的自定堆栈

```
WORD        Stack0 [0x10];      // 由于大部份的 OTP，RAM Size < 0x100，所以也不须要用· RAMADR

void    User_Func (void)
{
        SP      =       Stack0;             // 使用者自行设定堆栈，如此例，可呼叫 10 层。
        //      ...
}

void    FPPA0 (void)
{
        //      …
        WORD        Point;
        Point       =       User_Func;
        pushw Point;
        popw        PC1;                    //      让 FPPA1 跳到 User_Func 去执行。
        //      …
}
```

2.7. 最佳化的讨论

在 Mini-C 型式的项目，本发展系统提供了最佳化的设计，以下是本系统最佳化之规则：

2.7.1. 自动删除多余的程序码

范例一：

If (0) { 多余的程序码 }

结果：当系统模拟程序的流程后，会将永远执行不到的程序码删除。

范例二 (例外)：

由于 FPPA™ 是个多核心架构，IDE 无法模拟出不同 processing unit 之程序跳跃，以下为例：

设计师可以利用 ICALL 命令跳到 Call_Ldtab，但是系统无法模拟出此情况。

```
void        Call_Ldtab (void)    { …. }
void        FPPA6 (void)
{
        WORD        ptr_call;
        ptr_call    =       Call_Ldtab
        ICALL ptr_call;
        goto        $;
```

}

IDE 发展系统订定了一个特例，凡是用 “**变量=Label**”， “**mov A, LA@label**” 或 “**mov A, HA@label**”，则将 Label 视为可能被执行的程序并将 Label 显示在 Table Vector 中，如 Fig. 6-3 示，代表着：假如 “ptr_call = Call_Ldtab” 会被执行，那么 Call_Ldtab 将会被保留；相反的，假如 “ptr_call = Call_Ldtab” 不会被执行，那么 Call_Ldtab 将会被移除。

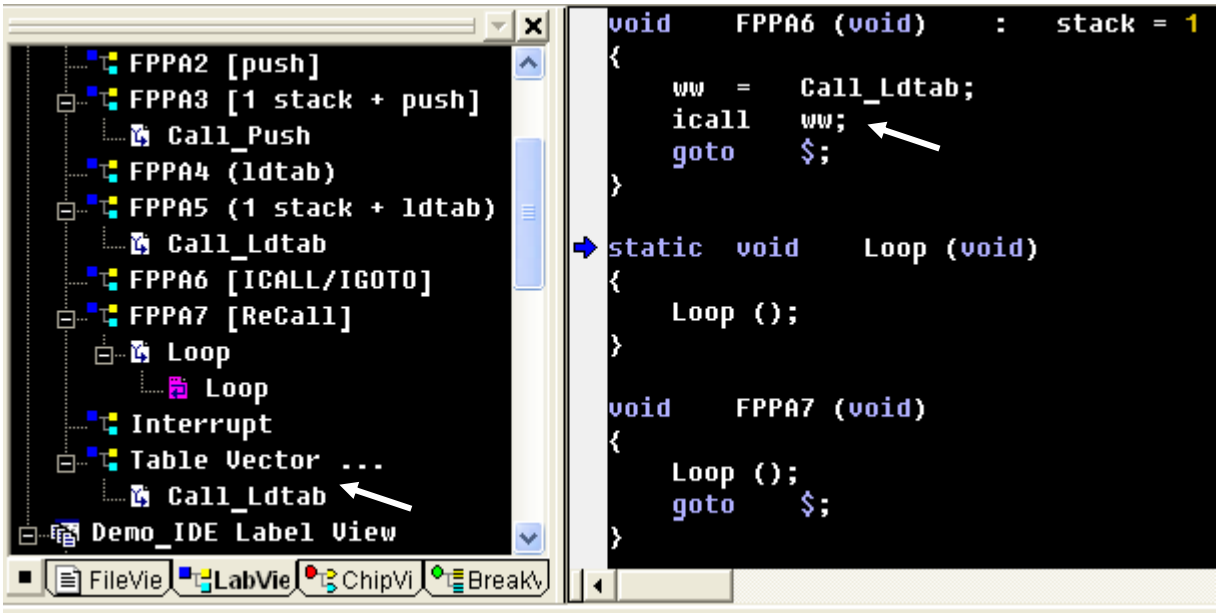


Fig.6-3. Label 当 Table Vector

范例三：

以下面程序为例，说明更清楚，虽然它符合 “变量=Label”，但该行又是多余的程序码，所以两行红色程序是会因最佳化而被删除。

```
void    Call_Ldtab (void)   { …. }
void            FPPA6 (void)
{
        WORD        ptr_call;
        If (0)      ptr_call    =    Call_Ldtab;        // 多余的
        ICALL ptr_call;
        goto        $;
 }
```

2.7.2. 不对以”汇编语言型式”写成的程序码作最佳化

范例一：

```
        @@:     t0sn        io_bit
                goto        @B
```

范例二：

```
                while  (io_bit)      NULL;
```

范例一与范例二都是等待 io_bit 为 1，但是范例一是以汇编语言型式写的，不作最佳化；范例二不是以汇编语言型式写的，故翻译成 “WAIT0 io_bit”。

2.7.3. 善用各种 FPPA 的硬件指令

一个好的 C Compile，必须善用硬件支援的各种指令，以减法为例：

```
BYTE bb1, bb2;                  //  翻译出的汇编语言
bb1  =  bb1 – bb2;              //  MOV   A, bb2+   SUB   bb1, A
bb2  =  bb1 – bb2;              //  MOV   A, bb1+   NADD  bb2, A
bb1  =  0 – bb1;                //  NEGC  bb1
bb2  =  0 – bb1;                //  NMOV  A, bb1+   MOV   bb2, A
```

充份利用本发展系统所提供的 Mini-C，来取代汇编语言，程序既简单又好写。

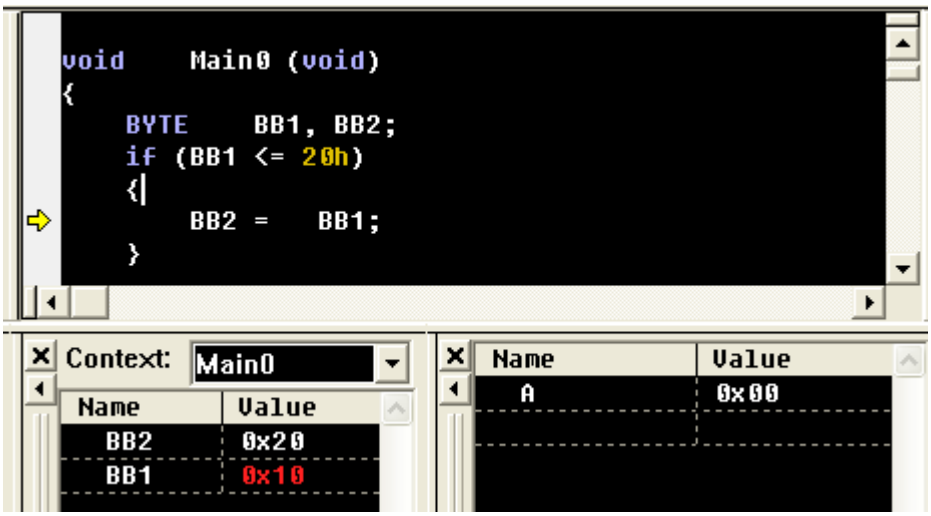
再以进位运算为例：

```
BYTE bb1;       WORD    ww2;
ww2  =  ww2 + bb1;              //  翻译出的汇编语言
                                //  MOV   A, bb1
                                //  ADD   ww2$0, A
                                //  ADDC  ww2$1

ww2  =  ww2 – bb1;              //  翻译出的汇编语言
                                //  MOV   A, bb1
                                //  SUB   ww2$0, A
                                //  SUBC  ww2$1
```

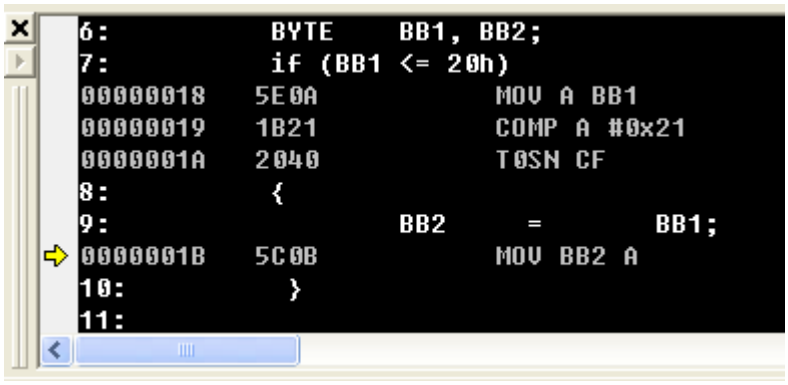
C compiler 要能精简的将 C 语言转化成汇编语言，就必需善用硬件的各种指令。

2.7.4. 最佳化带来的困扰

由于 Mini-C 做最佳化时，算法会去除掉不必要的步骤，因此有时在执行程序过程中，会有不是预期的现象；但是程序执行结果是一样的。以下为例来说明这现象：



原始程序是 “BB2 = BB1”，我们以为改掉 BB1 成为 0x10， 就可以让 BB2 变为 0x10 了吗?以 ICE 来做单步执行，得到的结果是 BB2 竟然为 0！ 为什么？看看反组译码就可知道：



本发展系统利用 ALU = BB1 的特性，做了最佳化，下表就是最佳化前与最佳化后之比较，虽然只是短短的几行指令，它却可以省掉两条指令。

最佳化前		最佳化后	
MOV	A, BB1	MOV	A, BB1
COMP	A, #21	COMP	A, #21
T1SN CF		T0SN CF	
GOTO$ + 3		//	省略
MOV	A, BB1	//	省略
MOV	BB2, A	MOV	BB2, A

3. IC 介绍

FPPA™ 是个划时代的产品，在一个芯片中，内含多个微处理单元，每个微处理单元可独立各自执行各自的程序，以确实达到平行处理的运作。虽然微处理器是崭新的架构，但是对于系统设计者而言，仍然可以沿袭传统微控制器在发展系统以及软件开发所累积的经验，系统设计者只要学习 FPPA™ 的指令集后，就可以享受到多核心架构带来的动力快感。

3.1. FPPA

1. FPPA 说明：

FPPA 为 IC 的一个执行核心。依不同系列 IC，可分为多核、双核、单双核、单核。

2. FPPA 指令执行速度：

PADAUK IC 分类	FPPA 执行速度	代表型号
多 核	依 PMODE 指令设定	PDK8xxx
双 核	System Clock / 2	P2xxx
单 核	System Clock	PMC1xx
单双核	由 Code Option 决定单/双核	PMC2xx

IHRC (ILRC/EOSC) 通过 CLKMD 缓存器设定，输出 System Clock； System Clock 经由上表的设定，输出 FPPA Clock。 大部份指令只须要一个 FPPA Clock 就可以完成，底下是不为 1T 的指令。

FPPA 数目	2 T 指令	2T/1T 指令(当 Skip Next 发生时为 2T)
双核以上	IDXM, PUSHW, POPW, LDTABx	---
单 核	GOTO, CALL, IDXM, PCADD, RETx	CEQSN, CNEQSN, T0SN, T1SN, IZSN, DZSN

FPPEN： 双核以上的 IC 皆有一个 FPPEN 缓存器。FPPEN 控制着每一个 FPPA 的执行与停止。 预设为 FPPA0 是执行，其他 FPPA 为停止。 此一缓存器的设定不影响每个 FPPA 执行的速度。

3.2. 缓存器介绍

3.2.1. 中断

3.2.1.1. 中断简介

1. 只有 FPPA0 能使用中断；也只有 FPPA0 能使用 ENGINT / DISGINT 这两条指令。
   一般步骤如下：

   (1) 设定 INTEN : INTEN = 1; 或 $ INTEN PA0; // 开启 PA0 中断
   或
   INTEN = 5; 或 $ INTEN PA0, T16; // 开启 PA0+T16M 中断

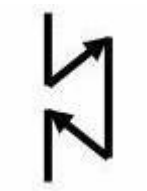
   (2) 清除 INTRQ : INTRQ = 0;

   (3) 允许中断 : ENGINT;

   … // 允许中断发生

   (4) 除能中断 : DISGINT;
   或
   INTEN = 0;

   所以，要产生中断，须要 (a) FPPA0 执行了 ENGINT，
   (b) INTEN & INTRQ 的值不为零；



```
void Interrupt (void)
{

}     //    RETI
```

中断发生后，FPPA0 跳入中断副程序，
自动进入 DISGINT 的状态，不允许新的中断产生。

直到执行了 RETI，才自动恢复到 ENGINT 的状态。

FPPA0 将再次检查 (INTEN & INTRQ) 的值，

决定是否进入新的中断程序。

PS： INTRQ 的 Bit 被设定，只代表讯号源被触发，不代表中断发生，需要跟 INTEN 的 Bit 同为 1 才算。

2.其它的 FPPA，完全不受中断影响，但能用设定 INTEN 来影响 FPPA0 的中断行为。

3.2.1.2. 初始值

1. 在旧系列 IC 的 INTEN 并没有初始值，所以要用中断 (ENGINT) 之前，先设定初始值是一个好习惯。

=> INTEN 在重置后，并没有初始值。

=> INTRQ 的设定，跟 INTEN 无关。

2. INTRQ 的值，不受 INTEN / ENGINT / DISGINT 的影响；
它随 PA0 / PB0 的正负缘，T16M 的溢时，ADC 的转换完成，...，而被触发为 1。

3.2.1.3. 中断检查

在中断程序中，你可以利用检查 INTRQ 的值，来确认何种中断被产生，范例如下。

```
void Interrupt (void)
{
    …
    if (INTRQ.T16)          // T16M 的溢位
    {
        INTRQ.T16 = 0;    // 清除已经处理过的中断 BIT
        …
    }
    if (INTRQ.PA0)          // PA0 准位改变
    {
        INTRQ.PA0 = 0;    // 清除已经处理过的中断 BIT
        …
    }
    …
}
```

如果忘了清除有使用的 INTRQ 的 BIT (对应于 INTEN 中也为 1 的 BIT)，会使中断重复进入，造成 FPPA0 无法运行。

另外，有些人在中断程序中，使用 INTRQ = 0，来取代 INTRQ.BIT = 0，这也是不好的习惯，时常造成一些 中断遗失，如下范例。

```
void Interrupt (void)
{
    …
    if (INTRQ.T16)          // 检查 T16M 的溢位
    {
        …
    }
    …                       // 万一程序执行到此时才发生 T16M 的溢位，
    INTRQ = 0;              //   尚未处理 T16 中断前，却又被清除，结果
    …                       //   造成中断遗失。所以，不建议用此种语法。
}
```

3.2.1.4. PUSHAF

(1) 中断发生时，请用 PUSHAF 指令，来保存 ALU 和 FLAG 缓存器；
在 RETI 之前，再用 POPAF 指令，来复原 ALU 和 FLAG 缓存器。

```
void   Interrupt (void)          // Mini-C 格式的中断向量写法
{
    PUSHAF;
    …
    POPAF;
}                                // 系统自动帮你填入 RETI
```

在汇编语言格式中，中断向量的表示法如下：

```
.ROMADR     0x10        // 定义 ROM 的地址到 0x10
            PUSHAF;
            …
            POPAF;
            RETI
```

(2) 中断本身占用 FPPA0 的一层堆栈 = 1 WORD，如果在中断中又使用 PUSHAF 指令，
你必须在 FPPA0 中多保留两层堆栈 = 2 WORD = 4 BYTE，给中断 + PUSHAF 使用。
不过，如果是以 Mini-C 的项目作开发，则堆栈配置的问题就不用你费心了。

3.2.1.5. WAIT 指令

在多核 FPPA 中，执行 WAIT0 / WAIT1 指令，如遇到中断，会先执行完中断，再回 WAIT 指令重新执行。
在单核 MCU 中，执行 WAIT0 / WAIT1 指令，会造成中断无法使用，所以 IDE 会禁用这类指令。
但你可以使用 .WAIT0 / .WAIT1 宏，它会展开成 BTSx + GOTO $-1 指令，一样可以达到相同功能。

3.2.1.6. DELAY 指令

在多核 FPPA 中执行 DELAY 指令，如遇到中断，会打断 DELAY 指令，先执行中断；等中断执行完，返回 DELAY 指令再重新执行 (不会记住先前的 DELAY 已用掉多少时间)。

曾经有客户，使用 DELAY 255，但是中断很频繁，造成 DELAY 永远执行不完，就是这原因造成系统无法执 行。

在单核 MCU 中，执行 DELAY 指令，会造成中断无法正常产生，所以 IDE 会禁用这类指令。但你可以使用 .DELAY 宏取代，它可能是由 DZSN / GOTO / .. 组合而成的优化循环；在单核 MCU 中使用 .DELAY 宏，也无须担心上述的中断会造成 .DELAY 执行不完的问题。

过去在范例中，会看到一个如下的宏：

```
Easy_Delay   macro    val, cmp
    #IF        val > cmp
               .DELAY   val - cmp;
    #ENDIF
endm
```

它是为了防止 .delay 输入一个负数而错误 (或者说是 极大的正数)，所做的保护机制。不过新的 IDE，都可 以支持如下语法，就不必再用此宏了。

```
.DELAY   add_value, sub_value
```

Ex : .delay 10, 9 等于 .delay 10 - 9 等于 .delay 1 等于 NOP
.delay 9, 10 等于 NULL (因为 9 < 10)

3.2.1.7. 2T 指令

在单核 MCU 中，执行 2T 指令，如 GOTO / CEQSN (执行 2T 时) /TxSN (执行 2T 时) / …，会造成中断暂时无法立即发出。

在一些旧系列的 MCU 中，如用使用这类 2T 指令组成的循环，也会导致中断一直无法产生，所以 IDE 会特别检查，防止客户误用到这种组合。

如下范例，你可以在 2T 循环中加入 1T 的 NOP 指令，就可以让中断正常发出。

	中断无法产生	反汇编	增加 NOP 解决
例一	goto $;	goto $;	while (1) NOP;

例二	while (PA.0)
	{
	if (PA.1) func();
	}
		t1sn PA.0;
		goto $+4;
		t0sn PA.1;
		call func;
		goto $-4;
			while (PA.0)
			{
			if (PA.1) func();
			NOP;
			}

3.2.1.8. APN 6

在部份 MCU 中，INTEN.7 有特殊用途，原本 WDRESET 须用.WDRESET 取代，CLKMD = xxx 须用.CLKMD = xxx 取代，CLKMD.1 = 0 须用 .CLKMD.1 = 0 取代， 详细信息请参考网站数据 ：APN006： 对抗电源急速波动的重要通知。同上，在单核模式时，执行 DISGINT 指令当下，恰巧又中断发生，在极少的情形， 会使 DISGINT 失效，建议使用宏 .DISGINT，就可以正常关闭中断了。

3.2.1.9. T16 的触发源

如果

1) 设定 T16M 的计数器在 Bit8 触发时产生中断。
2) 设定 INTEGS 为正缘时触发。
3) 清除 T16M 的计数，以重新计时。

```
$ T16M BIT8;          // 两者合起来，代表 T16M 的计数器
$ INTEGS BIT_R;       // 必须在 Bit 8 从 0->1 才会触发中断
WORD count = 0;
stt16 count;
```

则第一次中断是在 T16 计数到 0x100 (0x000 -> 0x100)，须要 0x100 单位时间。
第二次中断是在 T16 计数到 0x300 (0x100 -> 0x300)，须要 0x200 单位时间。
所以，两次中断所花的时间不同，很多第一次使用的设计者常在这里搞混了。

3.2.1.10. 巢状中断

如在中断程序中，想要让中断允许再重新进入，可使用指令 ENGINT，范例如下。

```
void Interrupt (void)
{
    PUSHAF;
     …
    ENGINT;
    …
    // 又允许新的中断发生 …
    …
    POPAF;
 }
```

经由 ENGINT，虽然可使中断重复进入，不过请注意堆栈的计算。

在这种允许重复中断的特殊用法中，Min-C 已经无法帮你计算堆栈了，但是，你可以利用下面方法，来设定你需要的堆栈。

```
void FPPA0 (void) : stack = 7        // 堆栈数由用户决定
{
    ...
}
```

3.2.2. T16M

3.2.3.1. 简介

以下是 T16M 在大部分 IC 里的 .INC 的定义表：

```
T16M            IO_RW           0x06
$ 7 ~ 5 :    STOP, SYSCLK, X, PA0_R, IHRC, EOSC, ILRC, PA0_F
$ 4 ~ 3 :    /1, /4, /16, /64
$ 2 ~ 0 :    BIT8, BIT9, BIT10, BIT11, BIT12, BIT13, BIT14, BIT15
```

在你使用 T16M 缓存器时，可以用如下语法来简化设计：

```
$   T16M    IHRC, /64, BIT15;
// 用 IHRC / 64 当 Clock Source，每 2^16 次使 INTRQ.T16=1。
```

```
// 如以 IHRC = 16 MHz 为例，IHRC/64 = 4 uS，约每 262 mS 产生 INTRQ.T16=1。

$ T16M SYSCLK, /64, BIT15;
// 用 SYSCLK / 64 当 Clock Source，每 2^16 次使 INTRQ.T16=1。
// 如以 .ADJUST_IC SYSCLK=IHRC/2 为例，System Clock = 8 MHz，
// SYSCLK / 64 = 8 MHz / 64 = 8 uS，约每 524 mS 产生 INTRQ.T16=1。

$ T16M EOSC, /1, BIT13;
// 用 EOSC / 1 当 Clock Source，每 2^14 次使 INTRQ.T16=1。
// 以外挂 32768 Hz 为例，32768 Hz / (2^14) = 2 Hz，每半秒产生 INTRQ.T16=1。

$ T16M PA0_F, /1, BIT8;
// 用 PA0 的下降源当 Clock Source，每 2^9 次使 INTRQ.T16=1。
// 所以每收到 512 次 PA0 的 Clock trigger，便产生 INTRQ.T16 =1。

$ T16M STOP;
// 停止 T16M 计数。
```

如果设定 T16M 的计数器为：

```
$ T16M ..., BIT8;        // BIT8 从 0 变 1 时, INTRQ.T16 会设为 1。
```

T16M 设为上升时触发：

```
$ INTEGS BIT_R;
```

并将 T16M 的计数值清为 0。

```
WORD count = 0;
stt16 count;
```

第一次 INTRQ.T16=1 是在 T16 计数从 0x000 到 0x100 时，共 256 次。
第二次 INTRQ.T16=1 是在 T16 计数从 0x100 到 0x300 时，共 512 次。

所以，设定 BIT8，是使 T16 计数 512 次才发出 INTRQ.T16=1 的意思。
很多人都会误会它为计数 256 次。

3.2.3.2. 常用计时单位

你可以使用一个 FPPA，依照 IHRC = 16MHz 的特性，产生大约接近 1S 的计时参考。

```
$   T16M        IHRC, /4, BIT15;
// 每次 T16 递增的时间          =     16MHz / 4 = 4 MHz = 0.25uS。
// INTRQ.T16 的触发时间         =     2^16 * 0.25 uS = 16,384 uS。

BYTE            count     =     0;
…
if   (INTRQ.T16)
{
     INTRQ.T16     =     0;
     count ++;
     if   (count >= 61)                // 16,384uS * 61 = 999,424 uS ≒ 1S
     {
          count     =     0;
          //   …                        // 每进入一次，约需 1S
     }
}
```

如需要 10S 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /16, BIT15;     // T16 递增时间 = 16MHz / 16 = 1 MHz。
                                      // 触发时间 = 2^16 * 1uS = 65,536 uS。
…
     if   (count >= 153)               // 65,536 uS * 153 = 10,027,008 uS ≒ 10 S
…
```

如需要 60S 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /64, BIT15;     // T16 递增时间 = 16MHz / 64 = 4 uS。
                                      // 触发时间 = 2^16 * 4uS = 262,144 uS。
…
     if   (count >= 229)              // 262,144 S * 229 = 60,030,976 uS ≒ 60 S
…
```

如需要 0.5S 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /4, BIT14;      // T16 递增时间 = 16MHz / 4 = 4 MHz。
                                      // 触发时间 = 2^15 * 0.25 uS = 8192 uS。
```

```
…
    if   (count >= 61)              // 8192 uS * 61 = 499,712 uS ≒ 0.5S
…
```

如需要 1/16 S 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /1, BIT13;    // T16 递增时间 = 16MHz。
                                    // 触发时间 = 2^14 * 1uS /16 = 1024 uS。
…
    if   (count >= 61)              // 1024 uS * 61 = 62,464 uS ≒ 1/16 S
…
```

如需要 100 mS 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /1, BIT12;    // T16 递增时间 = 16MHz。
                                    // 触发时间 = 2^13 * 1uS /16 = 512 uS。
…
    if   (count >= 195)             // 512 uS * 195 = 99,840 uS ≒ 100 mS
…
```

如需要 10 mS 的计时参考源，可改写 T16 设定如下：

```
$   T16M        IHRC, /1, BIT11;    // T16 递增时间 = 16MHz。
                                    // 触发时间 = 2^12 * 1uS /16 = 256 uS。
…
    if   (count >= 39)              // 256 uS * 39 = 9,984 uS ≒ 10 mS
…
```

另外程序架构也可稍微修改如下，占用的程序代码，会比上面的例子又少一点：

```
$   T16M        IHRC, /1, BIT11;    // T16 递增时间 = 16MHz。
                                    // 触发时间 = 2^12 * 1uS /16 = 256 uS。
BYTE            count;
…
if   (INTRQ.T16)
{
    INTRQ.T16    =    0;
    if (--count == 0)               // 被组译成 DZSN   count ，程序代码较少
```

```
    {
        count    =    39;          // 256uS * 39 = 9,984 uS ≒ 10 mS
        //  …                      // 每进入一次，约需 10 mS
    }
}
```

3.2.3.3. 两组计时单位

如果你须要同时有两个不同的计数时间，那上面范例的 INTRQ.T16 就无法同时共享，你可修改程序如下，就不须要使用 INTRQ.T16。

```
$   T16M       IHRC, /1, BIT11;     // T16 递增时间 = 16MHz。

WORD      T16_Cnt;
BYTE      count;
BIT       t16_over;
…
ldt16     T16_Cnt;
if   (T16_Cnt.15)                  // 每 2^15 次 T16 递增时间 = 2,048 uS
{                                  // 就会轮流进入区块一。
    t16_over     =    1;           //
}
else                               // 或区块二。
{
    if   (t16_over)
    {                              // 但进入区块三后，
        t16_over     =    0;       // 就不会重复进入。
        if   (--count == 0)        // 所以，每 4,096uS 才进入一次。
        {
            count    =    244;    // 4,096uS * 244 = 999,424 uS ≒ 1S
            //  …                  // 每进入一次，约需 1S
        }
    }
}
```

另外也有一种写法，看起来更精简，只是使用者得自行体会了。

```
$   T16M       IHRC, /1, BIT11;     // T16 递增时间 = 16MHz。
```

```
WORD            T16_Cnt;
BYTE            t16_flag, count;
…
ldt16           T16_Cnt;
A     =     (t16_flag ^ T16_Cnt$1) & 0x20;
if    (! ZF)                            // 每 2^13 次 T16 递增时间 = 512 uS
{                                       // 就会进入此区块。
      t16_flag     ^=    A;
      if    (--count == 0)
      {
            count     =     195;        // 512 uS * 195 = 99,840 uS ≒ 100 mS
            //    …                     // 每进入一次，约需 100 mS
      }
}
```

3.2.3.4. 范例研究

在范例项目 [Menu]->[File]->[Demo Project]->[T16_Key_LED1]，
将会利用了以上的计时组合，作按键触发，与点亮 LED 的功能；
当 LED 点亮 5 秒的计时后，LED 自动熄灭，在 5 秒的计时未到达以前，再按一次按键，也会熄灭 LED。

你可以用 ICE 版的现有电路来实验。

在以上的设计方法，虽然计时有点误差，但都在 1% 以内，唯独要注意的是，5 秒的计时，
如果以 1 秒为最小单位，误差就有可能 1 秒，所以范例中以 100mS 为计时单位。

3.2.3.5. Timer 的定时触发设定

如果想写一个定时触发 INTRQ 的程序，你可以选择 T16 / TM2 / TM3 三种硬件之一来完成。

但如果你懒得去想 定时定时器 如何设定，可以用底下宏帮你自动计算。

```
$ Timer     hw, time, src
    hw        =     定时器的硬件选择 :          T16    /    TM2    /    TM3
    time      =     定时器的触发周期，以 mS 为单位
    src       =     定时器的频率来源 :          省略时预设是 IHRC (并预设是 16 MHz)，
                    但如选用 SYSCLK，可以得到更长的触发周期，此时请输入相对应的除频 :
```

/2,/4,/8,.../64

范例一 ：

$ Timer T16, 256; // 以 T16 硬件 + IHRC，产生 256 mS 的 INTRQ.T16 触发

以 T16 当定时器，IHRC 当频率源，我们可以选择的周期有 1 / 2 / 4 / 8 / 16 / 32 / 64 / 128 / 256 mS。

假设我们设定成 1mS 触发 INTRQ，但实际 1mS 的周期是 16MHz / 16 / 1024 ==> 1.024 mS，会有 2% 的误差。

所以输入 256 ~ 262 mS 的触发周期，产生的程序代码是一样的。 (256 mS * 1.024 = 262.144 mS)

所以输入 128 ~ 131 mS 的触发周期，产生的程序代码是一样的。 (128 mS * 1.024 = 131.072 mS)

所以输入 64 ~ 65 mS 的触发周期，产生的程序代码是一样的。 (64 mS * 1.024 = 65.536 mS)

如果你不计较百分之 2 的误差，写成 2 的倍数是比较容易阅读的。 (1 / 2 / 4 / .. / 256)

至于其他周期，则不被允许。

范例二 ：

$ Timer TM2, 32; // 以 TM2 硬件 + IHRC，产生 32 mS 的 INTRQ.TM2 触发

以 TM2 当定时器，IHRC 当频率源，我们可以选择的周期有 1 / 2 / 3 / ... / 30 / 31 / 32 mS。

这些周期刚好可以被 TM2 的除频实现出来，所以误差比率只有跟 IHRC 有关。

范例三 ：

$ Timer TM3, 256, /8 // 以 TM3 硬件 + SYSCLK(IHRC/8)，产生 256 mS 的 INTRQ.TM3 触发

以 TM3 当定时器，SYSCLK (IHRC/8) 当频率源，我们可以选择的周期有 1 / 2 / 3 / ... / 255 / 256 mS。

在此范例中，虽然最长周期延长 8 倍，变为 256 mS，但不一定所有周期都没有误差，

不过经 PC 计算后，以 /8 为范例，最多只增加千分之一的误差，看来使用者可以放心使用。

如 $ Timer TM3, 200, /8; // 完全符合 200,000 uS 产生 INTRQ.TM3。

如 $ Timer TM3, 201, /8; // 计算后是 200,800 uS 产生 INTRQ.TM3，快千分之一。

如 $ Timer TM3, 202, /8; // 计算后是 202,176 uS 产生 INTRQ.TM3，慢千分之一。

输出暂时变量 ：

us_Timer : 实际的触发周期，单位为 uS，一般为输入变量 time 的 1000 倍。 (us_Timer => time * 1000)

如范例三，当你有需求细算误差时，就可以拿来使用。

范例四 ：

```
// 常数区
#define     hw_Timer    TM2
#define     ms_Timer    10
```

```
//  初始化
$ Timer       hw_Timer, ms_Timer;

//  主程序
while (1)
{
    if (INTRQ.hw_Timer)
    {
        INTRQ.hw_Timer  =  0;
        //  ...
    }
    //  ...
}
```

如果只想做单次的计时触发，这时就要做计时归零的动作了，语法如下，它会清除相对的 Count 与 INTRQ。

```
$ Timer       hw, Init;
```

hw 如为 TM2，会清除 TM2CT 与 INTRQ.TM2。
hw 如为 TM3，会清除 TM3CT 与 INTRQ.TM3。

hw 如为 T16，会使用 STT16 指令 与清除 INTRQ.T16，并确定程序中的 INTEGS 设定为 BIT_F。

(这是最可能疏忽的地方，忘了设定，时间就只剩一半。)

因为 计时归零 的动作并无法做到完全同步，存在着一个 计时 Clock 的误差是很合理的。

范例五 ：

```
$ Timer       TM2, 20;
$ Timer       TM2, Init;
while (1)
{
    if (INTRQ.TM2)  goto Over_Time;       //  超过 20 mS 就 Over Time
    //  ...
}
```

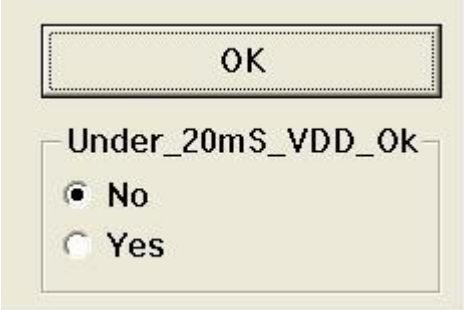
3.2.3. 其他讨论

3.2.3.1. LVR

LVR 必须要搭配 SYSCLK 的速度来设定，执行速度愈快，LVR 要愈高，在 IDE 中，为了防止使用者不小心将 LVR 选太低，会去检查彼此关系。

3.2.3.2. Under_20mS_VDD_Ok

在 IC 的 Code Option 中，常会遇见这样的选项：



如果 IC Power Up 可以在 20 mS 内，迅速达到正常的工作电压，例如 IC 电源为 电池，就是此例，它上电很快，而电池会因时间而缓慢降低电压，所以你可以致能该选项 (Under_20mS_VDD_Ok = YES)，使得 LVR 的选项增多。另一种相反的例子，就是 AC 阻容降压，它的 IC 上电时间可能很久，因此你必须设高你的 LVR 选项 (Under_20mS_VDD_Ok = NO)，以免 IC 在低压时就执行高速启动，造成误动作。

3.2.3.3. 缓上电

在缓上电的韧体设计中，除了将 LVR 提高，避免高速低压外，也有人会在开机时先执行在 ILRC，等电源稳定后，再执行 IHRC/n：

```
.ADJUST_IC    SYSCLK=ILRC(IHRC/2), ...; // WatchDog Disable…
…;                                      // 先执行在 ILRC，但以 SYSCLK=IHRC/2 作 IHRC 校正
……;                                    // 等待电源稳定，请依环境自行调整.
…;
CLKMD    =    0x34;                     // IHRC/2 = 8MIPS
```

等待电源达到稳定的侦测方法，有如下数种：

1) .delay ...; 最简单也最方便，但电源不见得已经稳定。

2) 有些系列 IC 有 RSTST，可以大约侦测电压。

```
@@: A          =     0;
    RSTST      =     A;
do
{
#if _SYS(AT_ICE)
    if (!FPPEN.0)     goto @B;      //  ICE 不支持，用虚拟运算顶替
#else
    if (RSTST.1)      goto @B;      //  如果电压低于 3V，继续等待
#endif
} while (--A);

//  至少须经过 3(循环内指令) * 256(循环次数) *2(PMODE=1/2)
//  = 1536 个 ILRC，才会离开循环。
```

3) 有些系列 IC 有 ADCC 与精准的 Bandgap，可以侦测电压。

```
$ ADCC   Enable, Bandgap;
$ ADCM   8BIT, /1;                    //  在 SYSCLK=ILRC 时，/1 就够了
while (1)
{                                     //  如果电压高于 3V，才会跳离循环
    AD_Start      =     1;
    wait1     AD_Done;
    #if _SYS(AT_ICE)                  //  ICE 的 Bandgap 并不精准
        if (ADCR < 102) break;        //  请依实际环境自行调整
    #else
        if (ADCR < 102) break;        //  ADC = 256 * (1.2V/3.0V)
    #endif                            //  电压越高，ADCR 值越小
}
$ ADCC   Disable;                     //  减少 ADC 耗电
```

4) 有些系列 IC 有 GPCC 与近似的 Bandgap，可以侦测电压。

3.2.3.4. PMx150xx

在型号为 ip5dXX / PMx150xx 的 IC，使用 STOPSYS / STOPEXE 指令之时，IDE 会依底下规则作检查：

1) 必须限定 .Code_Option 的 LVR 小于(等于) 2V。

```
.Code_Option       LVR        2V / 1.8V / Disable / ...
```

2) 必须要在 MISC 指令中，先关掉 LVR。

```
$ MISC    LVR_Disable;
STOPSYS / STOPEXE;
```

3) 必须限定 SYSCLK <= IHRC/8 = 2MIPS。

```
.ADJUST_IC    SYSCLK=IHRC/8;
```

4) 请自行确定，只有在 SYSCLK = ILRC/x，才允许使用 STOPxxx 指令。

请依照网站上的 APN 为标准。

3.2.3.5. PMx150/PMx156/PMC166/PMx153

在型号为 PMx150 / PMx156 / PMC166 / PMx153 的 IC，使用 STOPSYS / STOPEXE 指令之时，IDE 会依底下规则作检查：

1) 在 STOPxxx 之前，先将 SYSCLK 切到 ILRC/1。

```
$ CLKMD ILRC/1, En_IHRC, En_ILRC;     //    SYSCLK = ILRC/1

STOPSYS / STOPEXE

$ CLKMD IHRC/n, En_IHRC, En_ILRC;     //    SYSCLK = IHRC/2,4,8..
```

2) IDE 无法检查 STOPxxx 与 CLKMD 彼此的先后关系，请用户自行注意程序流程。

请依照网站上的 APN 为标准。

3.2.3.6. CLKMD 切换

在型号为 ip5dXX / PMx150xx / PMx154 的 IC，在 CLKMD 切换 SYSCLK 的来源后，必须增加 GOTO $+1 的指令，IDE 会特别检查以下规则限制：

```
CLKMD    =    0x14;    //  SYSCLK = IHRC/4
goto     $ + 1;        //  must exist
...
CLKMD    =    0xF4;    //  SYSCLK = ILRC/1
goto     $ + 1;        //  must exist
...
```

但不是切换 SYSCLK 的指令，则没有此限制。

```
CLKMD.En_IHRC          =    0;  //  关闭 IHRC 省电

CLKMD.En_WatchDog      =    0;  //  关掉 看门狗
```

3.2.3.7. P234

在型号为 P234xxx 的 IC，当 FPPA0 作查表指令 (LDTABL / LDTABH) 时，FPPA1 的程序区位置必须大于地址 0x40，IDE 会特别检查这项规则。

3.2.3.8. PMC131

在型号为 PMC131 的 IC，PB.3 最好当数字输出用。若使用 PB.3 来当数字输入，且 PB 有其他 I/O 口当作输出时，请勿使用 set1/set0 指令来改变 PB 的输出准位，此时 IDE 只接受 PB = xx 来对 PB 作输出改变，且相对的 Bit 3 也要为 0。

例如，使用 PB.3 当数字输入口，其他当输出口，并切换输出准位。

```
pbc      =    0b_1111_0111;    //  触发特殊检查机制
pb       =    0b_0000_0000;

set1     pb.7;                 //  Error : 使用 set1 指令
pb.6     =    0;               //  Error : 使用 set0 指令
```

```
pb      =   0b_0010_1000;   //  Error : bit 3 不能为 1
pb      =   0b_0010_0000;   //  Ok
```

3.2.3.9. P234

在型号为 P234xxx 的 IC，当 FPPA0 作查表指令 (LDTABL / LDTABH) 时，FPPA1 的程序区位置必须大于地址 0x40，IDE 会特别检查这项规则。

3.2.3.10. TKE

有些缓存器跟 IC 接脚关系比较复杂，需要两组以上缓存器才可定义完整，这时使用内定统一的宏，就比较方便。

```
Ex :    $ TKE   PA4;    //  不用在意 PA4 定义在 TKE1 或 TKE2
```

3.2.3.11. MISC_

有些缓存器跟 Code Option 设定关系比较复杂，通常被系统保留，在程序初始化时设定，如果想动态修正部分参数，而其他参数则由系统自行依照判断决定，则可依照下面写法简化程序撰写。

```
Ex :    $ MISC_ BG_Auto;    //  不用在意 BG_Auto 属于哪个缓存器，
                            //  该缓存器的其他参数由系统自动判断
```

或先检查相关设定在哪个 Register ?

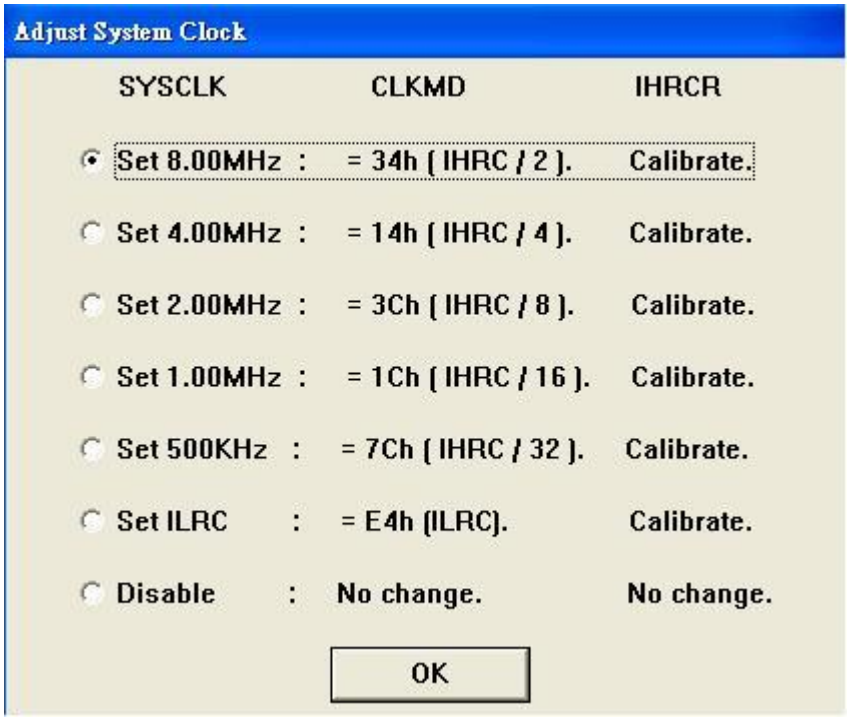
```
$ MISC_   ?   CS_Disable;
#ifnb    <MISC_REG>
    $ MISC_REG   CS_Disable
#endif

#ifnb    <MISC_REG>
    $ MISC_REG;     // 回复 Code Option 默认值
#endif
```

3.3. 指令介绍

3.3.1. .ADJUST_IC

3.3.1.1. 简介

当你第一次使用我们 IC 时，一定对这行指令 ".ADJUST_IC xxx"，或出现如下画面，感到很好奇。



对 FPPA 来说，在 Reset 后，IHRC 的值尚未设定，你可以利用指令 ".ADJUST_IC ..."，将 IHRC 校正到 16MHz。当你决定要校正 IHRC 时，顺便可以设定，.ADJUST_IC 后的指令要执行的系统频率，如上附图所示，System Clock 可以区分为 3 类：

(1) 执行在高速，如 IHRC/2 = 8MHz，IHRC/4 = 4MHz，…。

(2) 留在 ILRC，暂时不进入 IHRC，等待使用者在适当时机，自行切换 CLKMD。

(3) Disable：不校正 IHRC，通常用于外挂 Crystal。

选定 SYSCLK 后，有些旧系列 IC 需要 Bandgap 校正，会出现校正 Bandgap 选项。

3.3.1.2. 隐藏码

当你选择 SYSCLK=IHRC/2 时，将增加语法 ".ADJUST_IC SYSCLK=IHRC/2, ..." 到程序代码中，用来指引编译程序，隐藏性的增加如下的类似脚本：

```
…
CLKMD    =    0x34;                    // IHRC/2, Watch Dog disable。
A        =    ROM 纪录的 IHRC;         // 可能用查表 或 Call 得到校正值
if (A == 0xFF)
{
        …                               // 校正码运算，内容可以不用理它。
}
IHRCR    =    A;                       // 将校正值填写到 IHRCR 中。
…
```

当你选择 SYSCLK=IHRC/4 时，将增加语法 ".ADJUST_IC SYSCLK=IHRC/4, ..."，并隐含如下的类似脚本：

```
…
CLKMD    =    0x14;                    // IHRC/4, Watch Dog disable。
A        =    ROM 纪录的 IHRC;         // 可能用查表 或 Call 得到校正值
if (A == 0xFF)
{
        …                               // 校正码运算，内容可以不用理它。
}
IHRCR    =    A;                       // 将校正值填写到 IHRCR 中。
…
```

当你选择 ILRC 时，将增加语法 ".ADJUST_IC SYSCLK=ILRC (IHRC/16), ..."，并隐含如下的类似脚本：

```
…
A       =   ROM 纪录的 IHRC;        // 可能用查表 或 Call 得到校正值
if (A == 0xFF)
{
        …                           // 校正码运算，内容可以不用理它。
}
IHRCR   =   A;                      // 将校正值填写到 IHRCR 中。
…
```

因为选择 SYSCLK=ILRC 时，并无法指定校正 IHRC 的环境，所以增加语法：
(IHRC/16) 用来指定在 SYSCLK=IHRC/16 时校正 IHRC，你也可以选择(IHRC/2), (IHRC/4)或(IHRC/8)的环境。
只有未校正过 IHRC 的 IC，才会使用到 {..} 内的程序代码，正常的 IC，都会在 if (A == 0xFF)后跳离区块。

3.3.1.3. 特殊频率

对有些应用，并不须要标准的 16MHz，反而想要

(1) 适用于高速 UART 传输 115200 Hz： 15.6672MHz = 115.2KHz * 136，
(2) 适用于红外线传送的 455 KHz： 15.47 MHz = 455 KHz * 34。
(3) 适用于红外线传送的 38 KHz： 15.808 MHz = 38 KHz * 32 * 13。

在作 UART Baud rate = 115200 的应用时，你可以选择以下几种方法：

```
(a) .ADJUST_IC  ..., IHRC=15667200Hz, ...   // 15667200 = 115200 * 136
(b) .ADJUST_IC  ..., IHRC=16128000Hz, ...   // 16128000 = 115200 * 140
```

在作红外线应用时，你可以选择以下几种方法：

```
(a) .ADJUST_IC  ..., IHRC=15470KHz, ...     // 15470K = 455K * 34
(b) .ADJUST_IC  ..., IHRC=16380KHz, ...     // 16380K = 455K * 36
(c) .ADJUST_IC  ..., IHRC=15808KHz, ...     // 15808K = 38K * 416 (416 = 32 * 13)
(c) .ADJUST_IC  ..., IHRC=15960KHz, ...     // 15960K = 38K * 420 (420 = 4 * 105)
(d) .ADJUST_IC  ..., IHRC=16112KHz, ...     // 16112K = 38K * 424 (424 = 8 * 53)
```

当然，最后得再提醒，非标准的 16MHz，只能在一对一烧录器或一对四烧录器，才能使用；烧录工厂目前并不支持此项选择。

没有指定 IHRC 参数，预设将为 16MHz。

3.3.1.4. 校正电压

如果你想要修改校正 IHRC 时的工作电压，你可以修改如下的参数：

(a) .ADJUST_IC ..., Vdd=3.3V // 在 3.3 V 校正 IHRC

(b) (b) .ADJUST_IC ..., Vdd=4250mV // 在 4.25V 校正 IHRC

虽然校正的电压可以自由选择，但实际烧录器提供的电压，会约有 5 % 的误差；

另外 Bandgap 的校正电压，也是在这参考工作电压下，校正而得。

不过由于 IHRC 的电压飘移并不大，一般可以不用在意。

3.3.1.5. 看门狗 Watchdog

在指令 ".ADJUST_IC …" 后面的批注上，会有提到 Watch Dog 的开关情形。

```
.ADJUST_IC    Disable                  // No adjust IHRCR, WatchDog Enable
.ADJUST_IC    SYSCLK=ILRC, ...         // IHRCR adjust, WatchDog Disable, ...
```

请自行再依照须要，作 Watch Dog 的初始设定。

```
CLKMD.En_WatchDog    =    0/1;
```

3.3.1.6. 特殊选项

你可以在 .ADJUST_IC 后加上一些特殊选项，解决一些常见困扰。

1. .ADJUST_IC ...., Init_RAM;
   会把没有指定初始值的 RAM 全部清除，只有在 Mini-C 的项目才能另外指定初始值。
   如果想自行实作 RAM 的清除，请参考 清除 RAM。

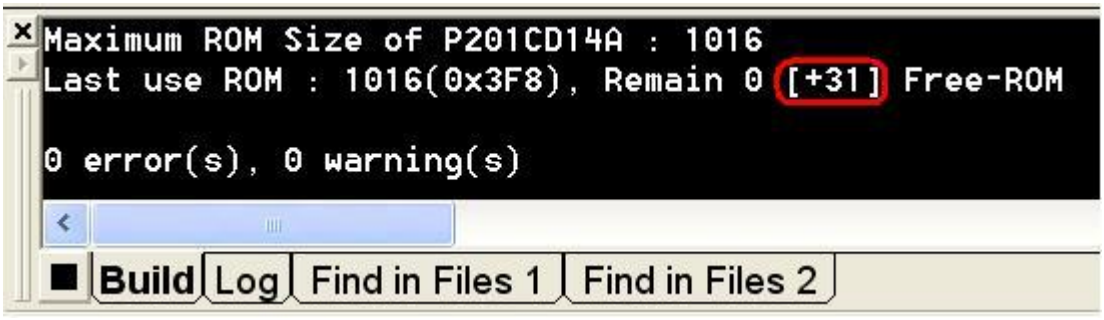
3.3.1.7. 杂项

1. 随电压 / 温度的影响，ICE 的频率变化会比较大，不过，在每次 ICE 下载程序时，IHRC 会重新再校正一次。

2. 由于制造流程的原因，调整 IHRC 所使用的缓存器 (IHRCR)，在每颗 IC 的偏移值都不一样，所以，完全不建议客户自行填写。

3. .ADJUST_IC 是目前的标准语法，当你想改变 ".ADJUST_IC …" 选项，只须直接移除该行，再重新执行一次编译，系统就会再出现一次 IHRC 的校正选单，供你选择。

4. 在部分系列 IC 中，如果使用 Mini-C 模式，而且程序未使用的空间够大，.ADJUST_IC 会以完整模式，多占用一些程序空间，来增加测试项目。当用户程序较大， Code Size 不足时，.ADJUST_IC 会自动改以精简模式，尽量减少占用的程序空间。



如上图，虽然剩余的 Code Size 为零，但如果使用者的程序代码再增加，.ADJUST_IC 将自动改成精简模式，并会再释放 31 个 WORD 的程序空间。

3.3.2. PCADD 指令

在 IC 中的 PCADD 指令，允许将 ALU 值加到 PC (Program Count)上，以作快速跳跃或查表的功能，如 下范例：

范例一： 用 ALU 来快速跳跃

```
A    =    data;              //   data = 1 ~ n
_pcadd
{
     goto     next_1;      //   if (data == 1)   goto next_1;
     goto     next_2;      //   if (data == 2)   goto next_2;
     …                     //   …
}
  next_1:   …
  next_2:   …
```

范例二： 用 ALU 来快速查表

```
A    =    data + 1;        //   data = 0 ~ n
_pcadd
{
     ret      12h;         //   A = 12h;      return;
     ret      34h;         //   A = 34h;      return;
```

```
    …                    //  …
}
```

范例三：　　(只限在遥控器系列的 IC)

```
pcadd    A                          //  在 PCADD 指令后面，强制需要一个大括号，括住可能的跳跃范围。
{
    …                    //  …
}
```

以汇编语言型态来表示，_PCADD 会被组译成 "PCADD　A"，
若 A == 1 时，会跳至下一行，A == 2 时，会跳至下二行，依此类推，
请注意，A == 0 时，会跳不出 PCADD 的循环。

自从有了 PCADD 这条指令后，作程序流程分析的人，总是非常头痛，因为 A 到底是多少呢?

因此建议写成如上的范例格式，　　_PCADD　{　…　}，
将 PCADD 的可能跳跃范围，全写在大括号内，这样程序流程分析就不会误判了。

下面的警告讯息，有时后，也是因为客户直接用汇编语言的 PCADD 指令，
以致程序流程分析误判，误以为 FPPA0 与 FPPA1 共享同一段程序了。

```
SM(119): The code is overlapped. [FPPA 0, 1] : 0xA2 to 0xA4
```

另外也请留意，经过 PCADD 指令后，A 的值会受影响。

3.3.3.　.DELAY 指令

虽然脚本中有 DELAY 指令，但它的最大时间只有 257 个指令时间。
在 Mini-C 项目中，支持宏语法 .DELAY　常数，自动产生最简化的脚本，而且常数允许超过 64K。

```
.DELAY      0;      //  不产生指令
.DELAY      1;      //  产生 NOP 指令
.DELAY      2;      //  产生 DELAY 1 指令
.DELAY      1000;   //  请自行参考反组译的结果
```

在 Mini-C 项目中，双核心以上的系统，最大的 .DELAY 可以到 0x5080302，但在 Asm 项目中，最大的 .DELAY 只能到 0x808，敬请见谅。

在单核心的系统，因为硬件的 DELAY 指令会造成中断无法正常执行，所以被禁用，在 PMC251 系列的 IC，也因 精简指令化 的关系而禁用 DELAY 指令，但软件的 .DELAY 宏不受影响，它会用 DZSN xxx 来自动进行取代。在多核心的系统，虽然使用硬件的 DELAY 指令可以减少程序代码，但也会因中断造成不可预期结果，建议使用 #pragma Disable HW_Delay_Ins 来禁止使用硬件的 DELAY 来当软件 .DELAY 的组成元素。

关于 DELAY 指令与中断的关系，你也可以在 DELAY 指令 里参考到更多信息。

为提高程序在不同系列 IC 的兼容性，建议直接以 .DELAY 宏取代 DELAY 指令，Mini-C 将会依照所选择 IC 支持的指令，做优化的替代与精简。

3.3.4. .WAIT0/1 指令

在单核系列的 IC，其指令集并不支持 WAIT0 及 WAIT1 两条指令。

Mini-C 支援 .WAIT0 / .WAIT1 两条宏指令，用来取代 WAIT0 及 WAIT1 两条 ASM 指令。

Mini-C 会自动判别 IC 的型号，同时兼顾兼容性与优化，产生最合适的程序代码。

关于 WAIT 指令与中断的关系，你也可以在 WAIT 指令 里参考到更多信息。

目前 Mini-C 支持的宏程序代码如下：

```
语法：              .WAIT1   IO_Pin

宏程序代码一：      T1SN   IO_Pin      // IC 不支持 wati1 指令，或是有特殊中断考虑
                    GOTO   $-1

宏程序代码二：      WAIT1 IO_Pin       // IC 支持 wait1 指令，且无特殊中断考虑

语法：              .WAIT0   IO_Pin

宏程序代码一：      T0SN   IO_Pin      // IC 不支持 wati0 指令，或是有特殊中断考虑
                    GOTO   $-1

宏程序代码二：      WAIT0 IO_Pin       // IC 支持 wait0 指令，且无特殊中断考虑
```

3.3.5. .TOG 指令

TOG 指令在某些系列的 IC 上并不被支持，你可以使用宏指令 .TOG，来达成程序的阅读性及兼容性。

```
语法：          .TOG    IO_Bit

宏程序代码一：   TOG     IO_Bit              // 如果 IC 支持 TOG 指令

宏程序代码二：   MOV     A, (1 << Bit);      // 注意 !! ALU 内容会受影响。
                XOR     IO, A;              // 如果 IC 不支持 TOG 指令，就改使用 XOR 指令

宏程序代码三：   MOV     A, IO;
                XOR     A, (1 << Bit);
                MOV     IO, A;              // 基本上，不太需要用到这种组合
```

3.3.6. .SWAPC 指令

SWAPC 指令在某些系列的 IC 上并不被支持，你可以使用宏指令 .SWAPC_I / .SWAPC_O，来达成程序的阅读性及兼容性。

.SWAPC_I 宏是将 CF 值设定与 IO_Pin 相同。
.SWAPC_O 宏则是将 IO_Pin 值设成与 CF 相同。

```
语法：          .SWAPC_I    IO_Pin

宏程序代码一：   SET0    CF          // 如果 IC 不支持 swapc 指令
                T0SN    IO_Pin
                SET1    CF

宏程序代码二：   SWAPC IO_Pin     // 如果 IC 支持 swapc 指令

语法：          .SWAPC_O    IO_Pin

宏程序代码一：   T1SN    CF          // 如果 IC 不支持 swapc 指令
                SET0    IO_Pin
                T0SN    CF
```

```
SET1    IO_Pin
```

宏程序代码二：

```
SWAPC IO_Pin     // 如果 IC 支持 swapc 指令
```

3.3.7. XOR IO 指令

在有些系列的 IC，新增加了指令 XOR IO, A，很适用于 IO 共享。

范例一：

如果 PA 是 Output Mode，而且 PA 只有被一个 FPPA 使用，
PA[3:0] 想要设定成 Output 0B_0101。

```
A    =    PA;
A    =    (A & 0xF0) | 0b_0000_0101;
PA   =    A;
```

范例二：

如果 PA 是 Output Mode，PA[3:0] 想要设定成 Output 0B_0101，
但是 PA[7:4] 是由其他的 FPPA 所控制，你可以用如下指令：

```
A    =    PA;
A    =    (A ^ 0b_0000_0101) & 0xF;
XOR       PA, A
```

你也可以在 IO 共享 里参考到更多信息。

范例三：

如果，您只有一只 IO 接脚须要 0/1 切换，可以使用如下指令：
(PS：单核系列的 IC 不支持此指令)

```
TOG       PA.0;          //  切换 PA.0 成 High / Low
```

范例四：

如果，您想将一组 IO 接脚切换，可以使用如下指令：

```
A    =    0b0000_1111;
XOR       PA, A;         //  PA[3:0] 电位切换
```

范例五：

如果想将 BYTE bb 的 bb[2:0] 不规则的对应到 PA [7,6,0]，在单核时：

```
A       =     PA & ~0xC1;
if (bb.0)   A   |=  0x01;
if (bb.1)   A   |=  0x40;
if (bb.2)   A   |=  0x80;
PA      =     A;
```

在多核且 IO 共享时：

```
A       =     PA & 0xC1;
if (bb.0)   A   ^=  0x01;
if (bb.1)   A   ^=  0x40;
if (bb.2)   A   ^=  0x80;
XOR     PA, A;
```

有些系列支持 SWAPC 指令，让你更易撰写程序：

```
A   =   bb;
A   >>= 1;      SWAPC   PA.0;
A   >>= 1;      SWAPC   PA.6;
A   >>= 1;      SWAPC   PA.7;
```

如果你不在意 IO 输出一点小 Gritch，底下方法也很简洁：

```
PA.0    =   0;
if (bb.0)   PA.0    =   1;
PA.6    =   0;
if (bb.1)   PA.6    =   1;
PA.7    =   0;
if (bb.2)   PA.7    =   1;
```

3.4. 多次烧录

3.4.1. 强制重烧

在已经被烧写好的 IC 中，如果某些常数须要修改，而且数值是从 1 变成 0，你可以使用强制烧录的功能。

原本常数	强制烧录
... ... A = 0x5F; ...	.Check_Sum Modify 0xXXXXXX; ... A = 0x5A; ...

如上例，A 从 0x5F (0101_1111) 变成 0x5A (0101_1010)，符合数值从 1 变 0，另外，因为程序修改，造成 Check_Sum 改变，这会使烧录器无法重烧；利用指令 .Check_Sum Modify 0xXXXXXX，强制指定 Check_Sum 成为旧的 CheckSum，以此方法通过烧录器的重烧防错检测，这样，强制烧录的功能就可以达成了。不过以上方法，对于 PDK82 系列并不能适用，为了更好的兼容性，请参考下节方法。

3.4.2. 多次烧录指令

在程序中，如果某些参数可能须要一再修改，你可以多保留一些未使用空间，给将来多次烧录时使用。

指令 .DC -1： 代表保留程序代码 (0xFFFF)，以供下次烧录。

指令 .DC 0 ： 代表清除程序代码 (0x0000)，不让下次使用。

在不会被优化消除的 GOTO / RET 指令以后，紧接着指令 .DC -1/0，编译程序会认为 .DC 程序代码也是不可以被优化消除，将保留此程序空间，以达成多次烧录。

第一次烧录	第二次烧录	第三次烧录
... ... A = 0x55; goto @F; .repeat 3 .dc -1;	.Check_Sum Modify ...; ... nop; nop; A = 0xAA; goto @F;	.Check_Sum Modify ...; ... nop; nop; nop; nop;

```
    .endm
@@:
```

```
    .dc   -1;
@@:
```

```
    A   =   0x5A;
```

如上例，第一次烧录时，保留 3 个 WORD 给未来使用， 第二次烧录时，将原本的程序代码设定成两个 NOP (0x0000)， 并将保留的其中两个 .DC -1 改成新的程序代码， 依此类推，保留的空间越大，允许重烧的次数越多。 如果你想更改一整区的子程序，你可以用如下方法：

第一次烧录

```
    ...
    ...
    goto    @F;
    .dc   -1;
@@: ...
    ...
    ...
```

第二次烧录

```
.Check_Sum Modify ...;
    ...
    nop;
    goto    @F;
    .repeat  xx
        .dc   0;
    .endm
@@:
```

在第二次烧录时，使用 .DC 0 取代第一次烧录时的旧 Code，虽然程序代码不会被执行，但它会避过优化的消除未使用 Code 的规则，顺利完成多次烧录的要求。但新的问题来了，要记算 .repeat xx 似乎有点麻烦，我们也可以采用旧 Code 不清除，但也不执行的另一个方法：

第一次烧录

```
    ...
    ...
    goto    lab1;
    .dc   -1;
    .dc   -1;
lab1: ...
      ...
//    程序结束
```

第二次烧录

```
.Check_Sum Modify ...;
    ...
    .virtual goto lab1;
    goto    lab2;
    .dc   -1;
lab1: ...
      ...
lab2: ...
      ...
//    程序结束
```

第三次烧录

```
.Check_Sum Modify ...;
    ...
    .virtual goto lab1;
    .virtual goto lab2;
    goto    lab3;
lab1: ...
      ...
lab2: ...
      ...
lab3: ...
//    程序结束
```

.VIRTUAL 会将后面的 goto / call 假设为是会被执行的程序代码，使得旧 Code 可以避过优化，成功被保留下来，但实际却以 NOP (0x0000) 取代，而使 IC 往下一条指令执行。

3.4.3. 重烧的限制

在烧录器上，对于要重烧程序的 IC，因为 OTP 已经加密的关系，没办法作防呆的检查，所以在使用上请自行小心，不要放错 IC。

在程序方面上，因为程序代码的改变，可能造成 Stack、Memory 位置的变动，以致能重复使用的程序代码变少，如果想要指定 RAM 位置的排列，请参考：

RAM 的寻址
STACK 的寻址
除能 LOCAL

如果使用了 Rolling Code 的 IC，又想要重烧，你只要在重烧时，修改烧录器的 Rolling Code 选单，如下法：

1) 把所有的 Rolling Value 设定成 FF / FFFF。
2) 把所有的 Rolling Step 设定成 00 / 0000。

于是对话盒会显示 Rolling Skip Check 字样，代表重烧时，Rolling Code 不改变。

3.4.4. 比对重烧来源

为了检查新产生的 .PDK 档，是否可以用于重新烧录，你可以使用指令

1) 在原始项目中加入

.Check_Sum From $

并产生参考用的 xxxx.PDK

2) 在新修改的项目中加入

.Check_Sum From xxxx.PDK

与旧的 xxxx.PDK 做比较

它除了自动使用旧 .PDK 的 Check_Sum (功能等同于 .Check_Sum Modify)，而且它也会检查是否有不可重烧录的数据发生，让你更安心的使用重烧功能。

底下是一个重烧的范例。

```
    #if 0          //   原始资料
        .Check_Sum   From      $          //
        .OutFile     Src.PDK             //   建立比对来源

        call         Old_Func;
        goto         @F;                  //
        .DC          -1;                  //   保留一次修改机会
    @@:

    #elseif 1    //   第一次重烧
        .Check_Sum   From      Src.PDK //   比对重烧来源

        .virtual     call      Old_Func;        //   清除前次用掉的程序代码
        nop;                                    //
        call         New_Func;
    #endif

        ....

    #if 1    //   第一次重烧 :  新增程序
New_Func:    ....
             ret
    #endif
```

如果发生比对错误，可能原因有:

1) : 在 Mini-C 的项目，程序随意跳出区块外，很容易造成 Memory 排列的错乱，
因为人工智能还是有其极限所在，所以，请对跳出区块外的程序，尽量作简单的处理。

2) : ICE 与 IC 编译的 Code 不同，可能也容易造成使用者的误解，你可以先关掉编译 ICE。
加入语法 : #pragma Set ICE Disable

3.4.5. 重烧 7 次功能

如果程序很小，又须要用 OTP 作实际烧录测试，利用以下的范例，可以让你重复使用 7 次烧录的功能。

请参考范例项目 [Menu]->[File]->[Demo Project]->[Mult_Program.PRJ]。

内容介绍如下： 在 xxx.PRE 檔中，使用 .Mult_Program_7 取代旧的格式。

```
//          .JMP      FPPA0   ...
//   .ROMADR        0x10
//          .PUT      Interrupt   ...

.Mult_Program_7       0
```

第一次组译后，编译程序会多产生一行注解，记录第二次组译须要的程序位移。

```
    .Mult_Program_7       0
//  .Mult_Program_7       1, 60
```

将产生的 xxx.PDK 拿来烧录 IC，完成第一次烧录。

第二次组译前，请将注解位置修改如下：

```
//  .Mult_Program_7       0
    .Mult_Program_7       1, 60
```

第二次组译后，编译程序会多产生一行注解，记录第三次组译须要的程序位移。

```
//  .Mult_Program_7       0
    .Mult_Program_7       1, 60
//  .Mult_Program_7       2, 97
```

将产生的 xxx.PDK 拿来烧录 IC，完成第二次烧录。

依此类推，你可以重复使用 7 次，烧录完全不同的程序，只须要遵守如下的限制：

(1)： 建议不要使用加密功能。

```
//{{PADAUK_CODE_OPTION
    ....
    .Code_Option      Security      Disable       //  Security Disable
//}}PADAUK_CODE_OPTION
```

如果使用加密功能，会导致第二次重烧时，数据无法作验证。

像 P234 系列 IC，就禁止同时使用加密功能与 .Mult_Program_7。

(2) : Code Option 的内容，重烧时必须一致 (或确定只有 1 -> 0)。
而 Code Option 的 单/双核选择 (FPPA = 2-FPPA / 1-FPPA)，一定不可以更改。

(3) : 在双核以上时，只有 FPPA0 / FPPA1 / INTERRUPT 可以被使用。
在单核时，只有 FPPA0 / INTERRUPT 可以被使用。

(4) : 只在 Mini-C 的项目才能使用这种功能。

(5) : Check Sum 默认值为 0。
你可以使用 .Check_Sum Modify 0xXXXXXX 来强制定义为其他数值。

3.4.6. 烧录次数限定

当你须要限定烧录器，最多只能烧录固定的芯片数量时，请在程序代码中，加入以下指令，并编译成 .PDK，下载到烧录器，就可以完成限定数量的工作了。

指令格式 (1)： .Writer Limit Start_Count, Limit_Count

Start_Count 为烧录器已经烧录 Ok 的 IC 数量，可以在烧录器画面得知，是一个 32 位的记数器，只在使用此功能且烧录成功时，才会递增。

Limit_Count 为想要限定的烧录数量，最大为 65535。

当系统开发者将开发好的 .PDK 寄到客户手中，就可以限定能使用的烧录数量。

指令格式 (2)： .Writer Limit (Serial_Number) Start_Count, Limit_Count

Serial_Number 为每一台烧录器的内建编号，可以在烧录器画面得知。

这样可以防止客户使用同一个 .PDK，下载到不同的烧录器中使用。

指令格式 (3)： .Writer Limit $ + Limit_Count

直接以烧录器目前的烧录 Ok 次数，再加上想要限定的烧录数量。

如果有多台烧录器就在开发者手边，下载后才需将烧录器交给客户，则用这个方法就比较方便。

注意：第一次下载限定数量的.PDK 时，会自动更新烧录器版本，并造成旧版 IDE 不再认得此新版烧录器，也就是说，从此必须以新版本 IDE (0.63 以后) 来使用新版烧录器。

在 IDE 0.92F 版以后，新支持了如下格式 ：
在客户端，选定要下载 PDK 的计算机跟烧录器，打开烧录器对话盒，选择 Convert PDK 里的 Writer Limit，再选择第一个选项 Generate INP，产生一个扩展名为 .INP 的输出档，交给开发商。

开发商也打开烧录器对话盒，选择 Convert PDK 里的 Writer Limit，选择第二个选项 Generate PDK，它会第一步要求输入客户的 .INP 档，第二步要求输入开发商的原始 .PDK 档，第三步要求烧录次数，然后打包产生新的 .PDK 档，就可以提供给客户下载。

注意原始 .PDK 里如果有 .Writer Limit 等参数，将不能再次转档，以保证不被误用。

只有使用有 .Write Limit 的 PDK 来烧录，才能使 Writer 的烧录流水号递增。

3.5. RAM 的分配

在 Mini-C 中，所有变量的地址由系统分配，原则上，stack 最先寻址，全局变量次之，再来为 static 变数，局部变量最后。

局部变量的地址，会被不同程序共享，以达到 RAM 的最佳利用。

如果在局部变量中，宣告一个大的 Array，造成 Allocate RAM fail，你可以将 局部变量 改成 全局变量，以优先排列位置。

变量分成 3 种类型：BIT / BYTE / WORD，并非全部的 RAM 都可以支持这三种型态，请参考各个 IC 介绍。

如果编译的错误讯息为 The value (MM.b), MM value over range !!!
应该就是 Bit 指令，无法安排适合的地址了。

在这里 WORD 寻址 代表的意思，是指有使用 ldt16 / stt16 / idxm / ldtabx 这些特殊指令的变量，跟我们平常使用的 BYTE / WORD / EWORD / DWORD 变量不一样，它只表示变量占用空间的大小，虽然同为 WORD，但意义容易混淆。

更多相关参考：

Mini.C 介绍 -> 数据型态 -> RAM 的寻址
Mini.C 介绍 -> WORD 的特殊应用 -> 清除 RAM

3.6. ROM 的分配

3.6.1. 未使用的 ROM

一般程序未被使用的区域，会被保留为不要烧录，以待不时之须。
请参考多次烧录。

不过有些使用者会希望把未用的区域，全部填为 RESET 指令，以防止程序跳跃到未定义区域的行为发生 (程序暴冲)。如下指令：

```
.Fill_Space        RESET;
```

至于选择那种方法，全依客户喜好了。

3.6.2. 固定数据区

在有些 IC，程序区后面还有固定数据区，储存一些可供查表用的数据，使用语法如下。

```
.CODE     name      AT    _SYS(INC.CODE_SIZE)

          table:    DC    0x....
```

如要更复杂的使用范例，请洽 fae，但一般这样的语法就足够使用了。

3.6.3. 第一个 WORD

在程序区的第一个 WORD，通常是直接跳到主程序 (FPPA0)，但有些系列例外，第一个位置系统保留。

原来为了支持 FT 测试，让不良的封装品可以提早筛选出来，有些系列的第一行指令是跳到测试程序区中；
当要烧录真正的客户代码时，才由第二个指令跳到主程序 (FPPA0)，并将第一个指令清除为 NOP。

不过使用者可以不用理会这种细节，因为对程序第一行的写法完全没有影响。

3.6.4. 最后 8 个 WORD

在程序区最后的 8 个 WORD，保留为系统使用，一般分配如下。

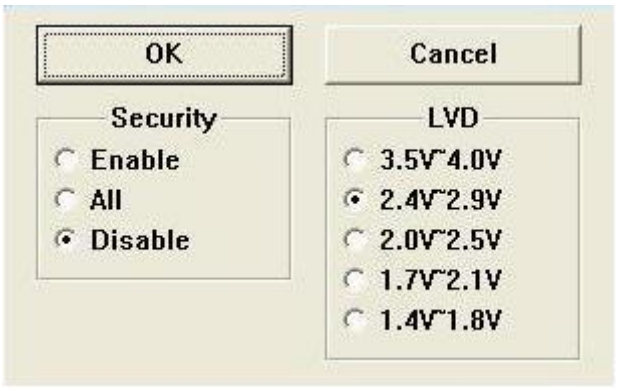
ROM 的地址： XXF8 XXF9 XXFA XXFB XXFC XXFD XXFE XXFF
使用功能： CheckSum Rolling Code ... IHRC CodeOption

xxF8 ~ xxFF 代表的是该颗 IC 的最后 8 个 WORD，在 1K WORD ROM 的系列 IC，此值通常为 3F8 ~ 3FF，在 4K WORD ROM 的系列 IC，此值通常为 FF8 ~ FFF，值得特别注意的事，有些 2K WORD ROM 的系列 IC，最后 8 个 WORD 不放在 7F8 ~ 7FF，却改放在 FF8 ~ FFF，所以，自行预测最后的位置是有风险的。

(1) 地址 XXFE 放的是 IHRCR 的校正码，只要使用如下语法，系统就帮你设定好 IHRC。

```
.ADJUST_OTP_IHRCR   8MIPS / 4MIPS / ILRC
.ADJUST_IC          SYSCLK=IHRC/n,,,,
```

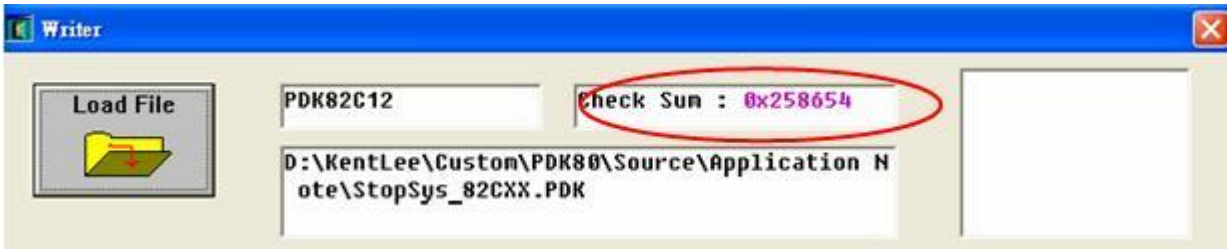
(2) 至于程序一开始就要选择的 Code Option 选项，则是放在最后一个 WORD 上。Code Option 选单如下，当你选定后，会把参数记录在程序中。




```
.CHIP   PDK82C13
//{{PADAUK_CODE_OPTION
        .Code_Option    LVD             2.4V~2.9V       // Maximum performance = 8 MIPS
        .Code_Option    Security        Disable         // Security Disable
//}}PADAUK_CODE_OPTION
```

(3) 在组译完程序后，你可从下图看到此次项目的 CheckSum，一般为 6 个 Nibble，而且此值会被烧录在 IC 的最后第 8 个 WORD，与第 7 个 WORD 的 Low Byte 中。

当你使用烧录器，执行 Load File 功能时，所显示的 CheckSum，也将是同一个数值。



另外请注意，CheckSum 是由 S/W 计算得来，它不是由 H/W 的 ROM 内容产生，所以由 IC 读回的 CheckSum 虽然一样，也不代表 ROM 的内容相同。

在 Mini-C 的项目中，Source File 可能包含好几个 .C 档，对调不同 .C 的顺序位置，也会影响 CheckSum 的计算，只好请使用时稍微留意一下。

3.6.5. Roll_Code

(1) 烧录器提供 Rolling Code 的功能，Low Word 在最前，High Word 在后。

目前 Rolling Code 的作法不同，分成两个 IC 族群：

一个是支持查表指令 (LDTABx) 的旧系列 IC (多 FPPA 的 IC，如 PMx271、PMC251、PMC232、PMC234、PDK82 系列)，每个 WORD 可以储存 2 个 BYTE 的数据；

另一个是支持 RET IMM 指令的新系列 IC（单一 FPPA 的 IC，如 PMx131、PMS130、PMx150、PMx153、PMx156 系列），每个 WORD 虽只能储存 1 个 BYTE 的数据，但呼叫方法更方便了。

对于支持查表指令 (LDTABx) 的 IC，你可由下列命令来决定 Rolling Code 的长度。

语法	ROM 的地址 ：	XXF9	XXFA	XXFB	XXFC	XXFD
.ROLLING 2WORD			Roll:0	Roll:1		
.ROLLING 3WORD			Roll:0	Roll:1	Roll:2	
.ROLLING 4WORD			Roll:0	Roll:1	Roll:2	Roll:3

建议用如下范例取得 Rolling Code 的数值，以方便未来的兼容性。
(再次提醒，自行预测 Rolling Code 的位置是有风险的。)

```
WORD    Point, Low_Roll, High_Roll;
Point       =   _SYS (ADR.ROLL);
Low_Roll    =   *Point$W;
Point++;
High_Roll   =   *Point$W;
```

如果你有使用语法 _SYS (ADR.ROLL)，但没有指定 .ROLLING ?WORD，则系统默认 Rolling Code 为 2WORD。PMC271、PMC232、PMC234 最多只能 3WORD，其余最多可以为 4WORD。

对于只支持 RET imm 指令的 IC，你可由下列命令来决定 Rolling Code 的长度。

语法	ROM 的地址 ：	XXFA	XXFB	XXFC	XXFD
call _SYS(ADR.ROLL);		Roll:0	Roll:1		
call _SYS(ADR.ROLL) + 1;		Roll:0	Roll:1		
call _SYS(ADR.ROLL) + 2;		Roll:0	Roll:1	Roll:2	
call _SYS(ADR.ROLL) + 3;		Roll:0	Roll:1	Roll:2	Roll:3

你加的 offset (1 ~ 3)越大，代表使用的 Rolling Code 占用的 ROM 空间越大。最长 4 Byte，最短 2 Byte。（不再使用 .ROLLING xWORD 来表示长度。）有 ADC 的 IC，只有 3 Byte 的 Rolling Code，如 PMS131。

你不可以自行使用常数来取代 _SYS(ADR.ROLL)，因为 IDE 须要这个 Key Word，来代表使用 Rolling Code。(依照如下语法，才能正确使用 Rolling Code。)

```
BYTE    Roll[4];
call    _SYS(ADR.ROLL);     //  读取 Roll:0
Roll[0] =   A;
call    _SYS(ADR.ROLL) +1;  //  读取 Roll:1
Roll[1] =   A;
call    _SYS(ADR.ROLL) +2;  //  读取 Roll:2
Roll[2] =   A;
```

```
call      _SYS(ADR.ROLL) +3;   //   读取  Roll:3
Roll[3] =     A;
```

(2) 如果由烧录厂代烧 Rolling Code，目前只先支持 2 WORD (使用 LDTABx)，或 4 BYTE (使用 RET imm)，编码方式由烧录厂制定，如果对编码方法有特别须求，请务必事先告知，并且要考虑到烧录厂对特殊流程的风险性。

在烧录器上，Rolling Code 无法重烧改变，但可以重烧时保留前一次的 Rolling 值，只须设定 Rolling Value = 0xFF, 0xFF,.../ 0xFFFF, 0xFFFF... (全为 1)，Rollin Step = 0,0,..(全为 0)，此时，烧录器会显示.Rolling Skip Check。

3.6.6. 外部 Roll_Code

在这里介绍 客户自定义的 Rolling Code 格式，长度/位置/滚码的方法 由客户自定义。

(1) 程序中加入如下命令

.User_Roll 长度 WORD|BYTE [, [产生 Rolling 的执行档] [, 储存 Rolling 的数据文件]]

长度最长为 63 个 Word，或 126 个 Byte。
假设产生的 .PDK 名称为 File_Name.PDK，
如果没有指定 产生 Rolling 的执行档，预设就是 File_Name.EXE。
如果没有指定 储存 Rolling 的数据文件，默认就是 File_Name.ROL。
默认路径就是 File_Name.PDK 的目录。

范例 (A)：支持查表指令：以 WORD 为单位

```
...
User_Table:
.User_Roll      18 WORD, "Gen_Roll.exe", "Rolling.txt"
                //      保留 18 个 Code_Word for Rolling
                //      并更改了默认的 执行文件/数据文件
...
{
                ...
                WORD    pnt     =     User_Table;
                WORD    data    =     *pnt$W;
                ...
}
```

范例 (B)：支持 RET imm 指令：以 BYTE 为单位

```
void    A_to_Table (void)
{
        A++;                    //  PCADD 必需从 1 起跳
        _pcadd
        {                       //  底下会替换成 RET imm 指令
                .User_Roll          0x40 BYTE, "Gen_Roll.exe" "Rolling.txt"
        }
}
...
{
        ...
        A   =   0;      //  A 可以是 0 ~ 0x3F
        A_to_Table();   //  读取数据
        BYTE    data    =   A;
        ...
}
```

(2) 烧录器下载 User_Name.PDK 后，
IDE 会呼叫 "Gen_Roll.exe 0x0"
以得到第一笔数据文件 Roll_Out.txt，内容类似如下 (以 WORD 为单位)：

```
Index=0x0
0x1234 0x5678 ....
0xABCD 0x0000 ....
```

(3) 如果烧录失败，就不会呼叫 Gen_Roll.exe。
当烧录完全正确后，
IDE 会呼叫 "Gen_Roll.exe 0x1"
以得到第二笔数据文件 Roll_Out.txt，内容类似如下：

```
Index=0x1
0x2345 0x6789 ....
```

(4) 再下一笔数据，IDE 会呼叫 "Gen_Roll.exe 0x2"，依此类推。

(5) 当下次重新使用烧录器时，IDE 会从 Roll_Out.txt，
读取上次的记录 Index=???，如无信息，Index 将归 0。

(6) 可以用烧录器上的按钮烧录 (也就是支持半自动机台)，
但 USB 线要连上 PC 的烧录软件，才能动态更新数据，
并且注意 PC 的更新数据速度会随环境而改变。

(7) Gen_Roll.exe 的范例 VC Code，可以参考附件：
[Menu]->[Help]->[Refer Source]->[GenRoll.C]

(8) 只有在 PDK3S-P-002 烧录器支持此功能。
(9) 而除了 PDK82 系列，其他 MCU 系列全部支持此功能。
如要烧录 PDK82 系列，会有在特定版本，无法下载程序的小麻烦。
(因为烧录器的程序空间，已经不够的关系)

3.6.7. 使用参数档

如果输出的 .PDK 档案，须要用外部参数档来调整细节内容，你可以使用如下的语法：

```
name1    PARAMETER    0x10          //  0x10    用于 ICE 模拟用
name2    PARAMETER    0x2345        //  0x2345  用于 ICE 模拟用
...

程序中，你可以使用这些参数来运算

A    =    name1;
...
WORD     mem2     =    0x55AA;
if (mem2 > name2) ....
```

当真正要下载 .PDK 档到烧录器时，你必须附带与 .PDK 同一个档名的 .PRM 档，烧录器才能下载修正后的内容。

.PRM 档案的内容如下：

```
name1    PARAMETER    0x30          //  0x30    用于实际 IC
name2    PARAMETER    0x4567        //  0x4567  用于实际 IC
...
```

这样，动态修正参数就完成了。
对于支持查表指令的 IC，你可能想用外部参数档来调整一整个区块的内容，

你可以使用如下的语法：

```
name3    PARAMETER
{
        0x1234, 0x5678,          //  以 WORD 为单位
        0x5555, 0xAAAA,          //  存放于 ROM 中
        0x0000, 0xFFFF           //  以上数值用于 ICE 模拟
}
```

程序中，你可以使用查表来得到这些参数：

```
WORD    point   =   name3;
WORD    data    =   *point$W;
point++;
....
```

当真正要下载 .PDK 档到烧录器时，你须有一个 .PRM 档案，内容如下：

```
name3    PARAMETER
{
        0x2345, 0x9876,          //  以 WORD 为单位
        0x5A5A, 0xA5A5,          //  将替代原本的 ROM
        0x0000, 0xFFFF           //  用于实际 IC
}
```

这样，动态修正区块就完成了。

你可以同时使用 "参数调整" 与 "区块调整" 作动态修正，另外，既然 .PRM 文件的语法与程序使用的语法完全相同，你可以使用

```
include     xxxx.PRM
```

来简化程序。

修改 .PRM 参数档会造成下载到烧录器的 Check Sum 改变；
没有 .PRM 参数档则无法下载 .PDK；使用 .CHECK_SUM FIX 语法，
可以固定 Check Sum，而在烧录器上就多显示一组识别用 Check ID；
如果已经在烧录器的数据内容跟 PC 不同，烧录软件也会特别提示。

关于 .PRM 档案的防呆处理，如果有多余未被使用的参数宣告，并不会引起错误，如果有需要的参数未被定义，则会引起错误。

如果在不同的模块中，重复的呼叫使用同一名称的 "区块调整"，IDE 会自动将 ROM 空间合并成同一块，最后放置的地址由系统决定；经过优化后，没有被程序代码引用的 "区块调整"，则会被删除。

3.6.8. 合并参数档

如果能将 .PRM 档跟 .PDK 档合成一个档案，在使用上会很方便，只要你的 .PRM 档遵守如下的格式：

```
//PARAMETER
...
xxx PARAMETER     ...
...
PARAMETER END
```

第一行以 //PARAMETER 作开头，前面不可以有空格符，最后一行以 PARAMETER END 作结尾，后面也不可以有空格符，再来将 .PRM 档贴在 .PDK 档后面，你可以利用 DOS 的 Command：

```
COPY      /B    Source.PDK + Source.PRM     Target.PDK
```

或者自己写程序合并。

也有使用者希望在 .PDK 中存放一些额外的参数，以方便 AP 使用，只要合并的 .PRM 内容，以 //USER 作开头，后面的内容就随使用者发挥了。

也可以在程序中加入 .APPEND_USER_INFO 语法，自定义附加内容。

```
.APPEND_USER_INFO        文件档案
                         或
.APPEND_USER_INFO
{
        自定义附加内容.....
}
```

如果同时有 //PARAMETER 与 //USER 区块要附加在 .PDK 后面，记得 //PARAMETER 区块一定要放在 //USER 区块之前，IDE 才能正确判读。