HYCON TECHNOLOGY

HYCON 8-bit MCU

C 编译器用户手册

目录

1. 导读

1.1 内容简介

本手册主要介绍基于HYCON 8-bit OTP MCU的C 编译器用法；以C语言为基础，介绍HYCON C IDE的用法及编程；本手册适合有C语言基础的开发人员，同样也适用无C语言基础的开发人员；本手册涉及HYCON的相关IC及开发工具资料，请到HYCON官网下载：www.hycontek.com 。

第二章介绍HYCON C Compiler的基本操作，包括工程的新建、调试；包括程序框架及工程设置；指导开发人员熟悉操作HYCON C Compiler。

第三章介绍C语言相关基础知识，主要提供给无C语言基础的开发人员学习，让用户更快入门HYCON C Complier。

第四章介绍HYCON C Compiler的扩展功能，介绍特殊使用方式及特殊变量定义、关键字和伪指令；介绍HYCON C Compiler自带的C 函数库，该函数库针对IC 硬体设置的，方便开发人员设置IC硬体配置。

第五章介绍HYCON C Compiler的汇编语言与C语言混合编程。

第六章介绍HYCON C Compiler优化功能。

第七章介绍基于HYCON C Compiler的C语言应用范例程序。

第八章指出一些在使用过程中遇到的错误提示及错误原因。

1.2 相关文档

本手册涉及到相关 IC 或软件资料，用户可以在我们公司官网上下载所有文档。

下载文档的网址：

http://www.hycontek.com/cn/products-cn/6083

(1)HYCON HY11P Series Data Sheet

(2)HYCON HY11P Series User’s Guide

(3)HYCON HY11P Series Hardware TOOL User Manual

(4)HYCON HY11P Series Software TOOL User Manual

2. C 编译器用法

2.1 新建工程项目

一个完整的工程项目，包含工程文件及程序文件，以下先介绍如何创建一个工程文件，注意：创建工程项目前，需要在对应地方新建一个文件夹来存放工程文件。

步骤01：选择菜单栏Files项目”New Project”，出现Project Option窗口，如图2-1。输入工程名称，选择IC型号、保存路径。目前只支持H08A CPU core的HY11P型号。

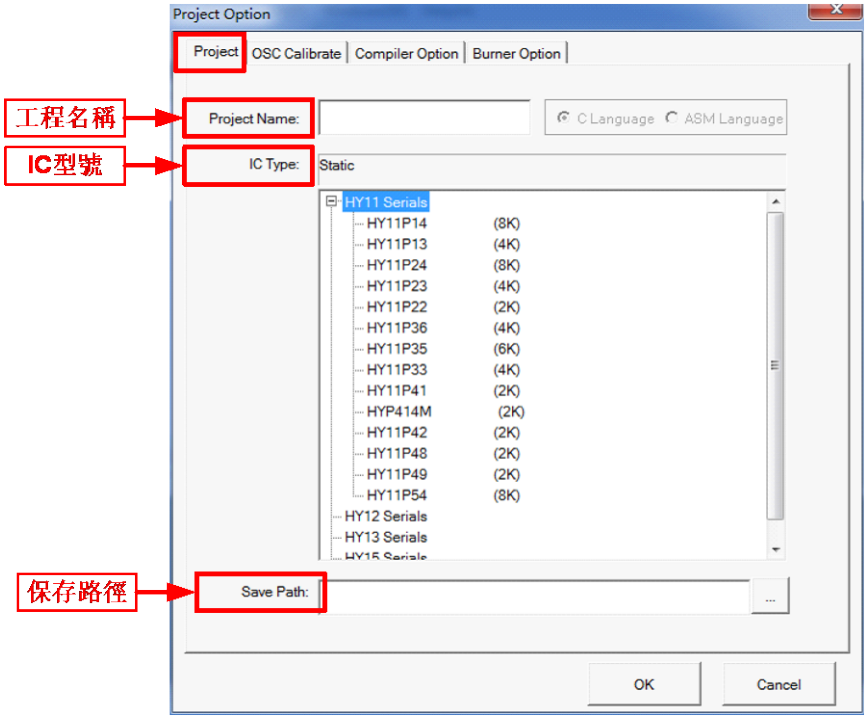


图 2-1

步骤02：烧录信息OSC频率校正设置对话框，如图2-2。

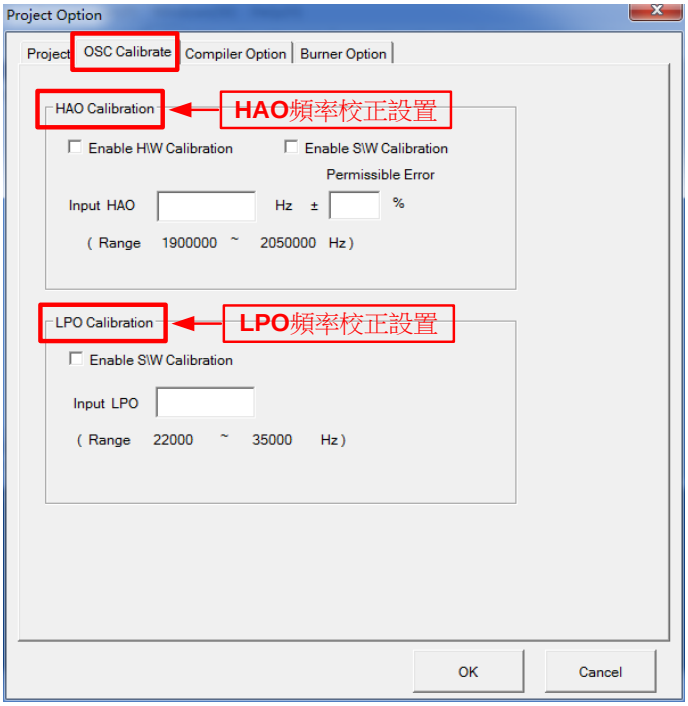


图 2-2

HAO Calibration:

名称	功能
Enable H/W Calibration	启动硬体HAO校正功能，依照烧录器所提供的VDD及待烧芯片的温度实际校正系统频率，此功能需要在选择芯片型号后，再次确认是否可硬体校正。
Enable S/W Calibration	启动软体HAO差值校正功能
Input HAO	为欲校正HAO频率数值
Permissible Error	校正后频率值与欲校正数值允许差异范围。 1.“Permissible Error”输入范围需≧±2%。 2.烧录器对HAO进行硬体校正后，待烧芯片的HAO频率为当时烧录器的提供的VDD电压及待烧芯片温度的条件下所测得，再对“Permissible Error”所输入范围筛选。

LPO Calibration:

名称	功能
Enable S/W Calibration	启动软体LPO差值校正功能
Input HAO	为欲校正LPO频率数值

步骤03：组译选项设置对话框，如图2-3。

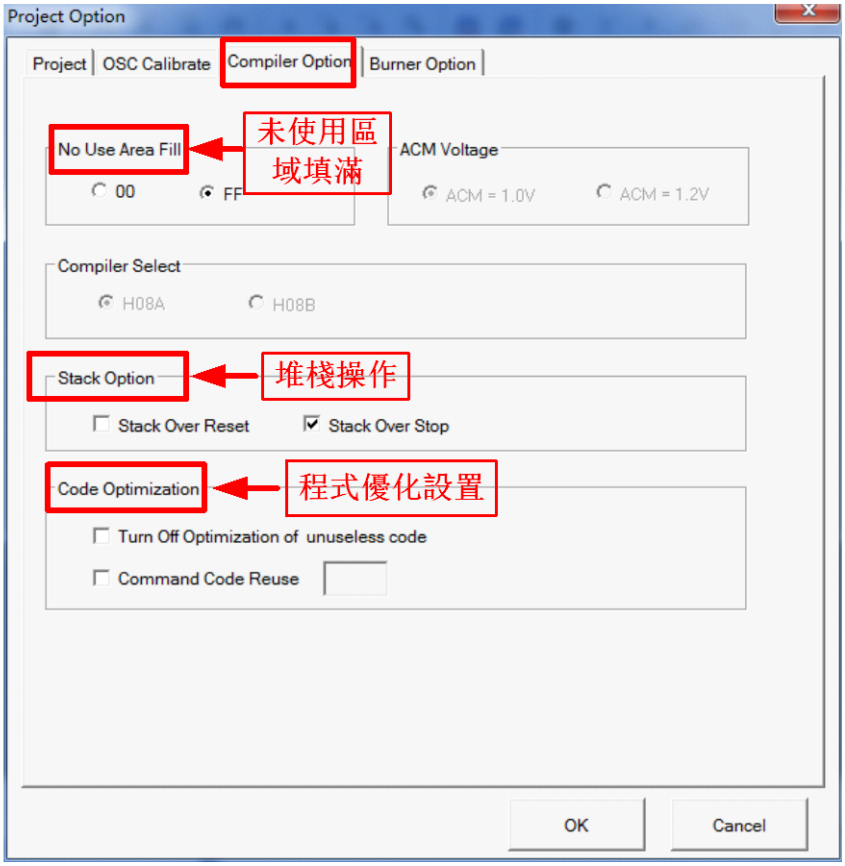


图 2-3

名称	功能
No Use Area Fill	未使用区域填满：组译程序后，在未使用的程序空间选择填满00或FF。
Stack Option	堆栈操作：选择当OTP程序运行发生堆栈满或溢位后是否要重置。
Code Optimization	程序最优化设置：默认开启最优化功能，未勾选下面2个选项. 1.Turn Off Optimization of unuseless code：不优化。 2.Command Code Reuse：开启最优化功能的同时，设置优化code中的相同程序，设置参数为0~5，若写入5，表示5行以上相同程序再进行优化。

步骤04：烧录信息设置对话框，如图2-4。选择完成后，点击”OK”，完成设定。

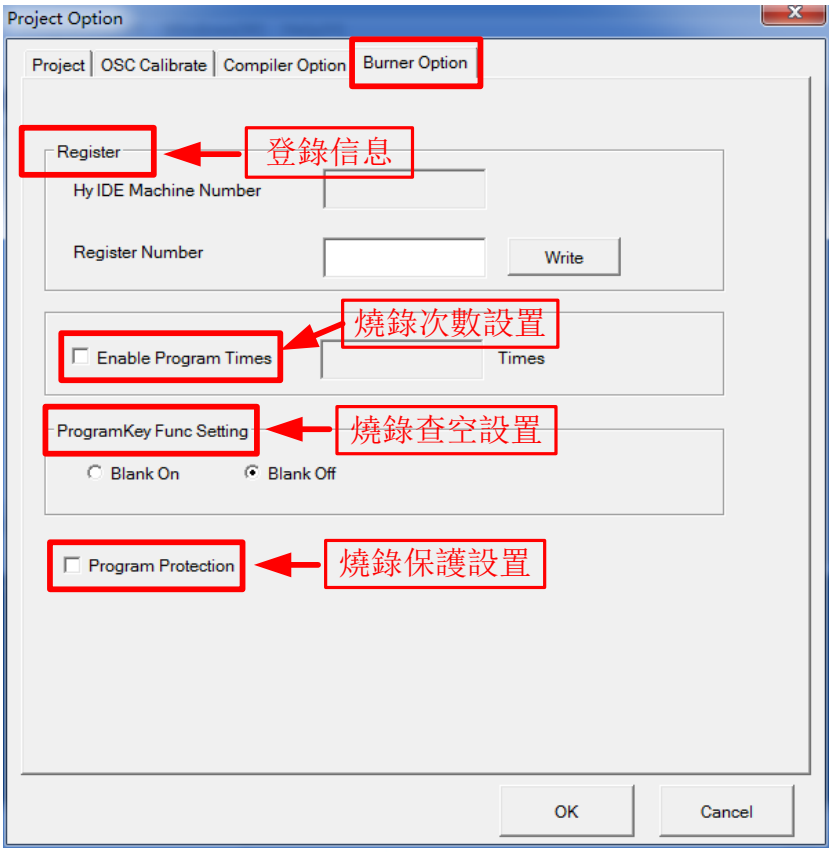


图 2-4

名称	功能
Register	1.Hy IDE Machine Number：读取工具序号。 2.Register Number：工具注册功能；工具在出厂前已完成注册，客户可以安心使用。
Enable Program Times	选择是否启动Download的程式能被烧录的次数，最大2147483646，最小1。
ProgramKeyFunc Setting	选择是否在对芯片烧录前，进行查空确认。
Program Protection	选择是否烧录保护。

步骤05：完成新建工程后，在工程栏出现工程项目，如图2-5。

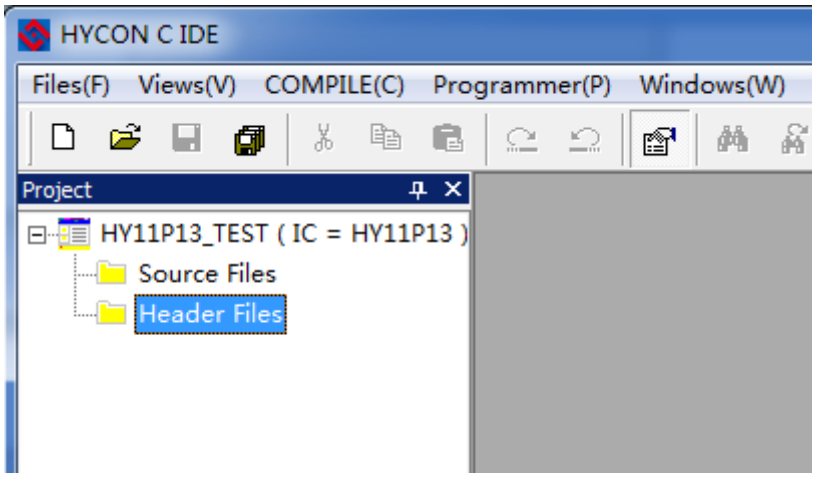


图 2-5

2.2 新建程序文件

新建程序文件包含头文件、C文件及ASM文件(此版本暂不支持ASM文件)，以下介绍如何创建对应的程序文件；程序文件都是与工程文件存放在同一文件夹下。操作步骤如下：

选择菜单栏Files项目"New"，如图2-6所示，按以下步骤操作。

I） 编辑代码，选择工具栏的"Save"按钮。

II） 在弹出的对话框中选择把文件保存到工程目录下。

III） 并且设定保存的文件名。

IV） 再将此文件添加到工程中，右击"Source Files",点击"Add File"如图2-7。

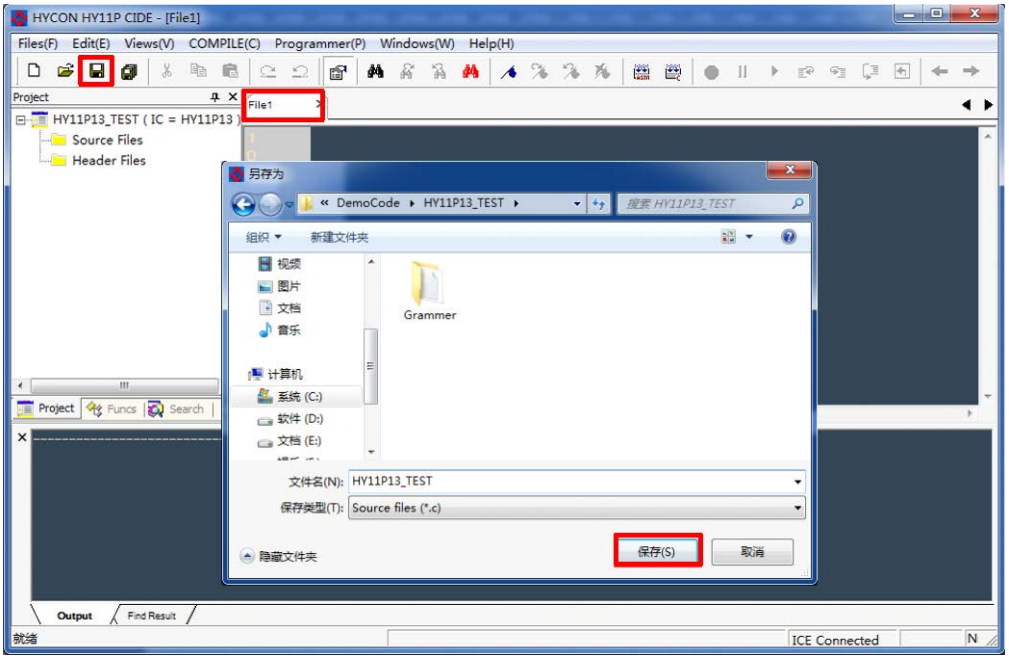


图2-6

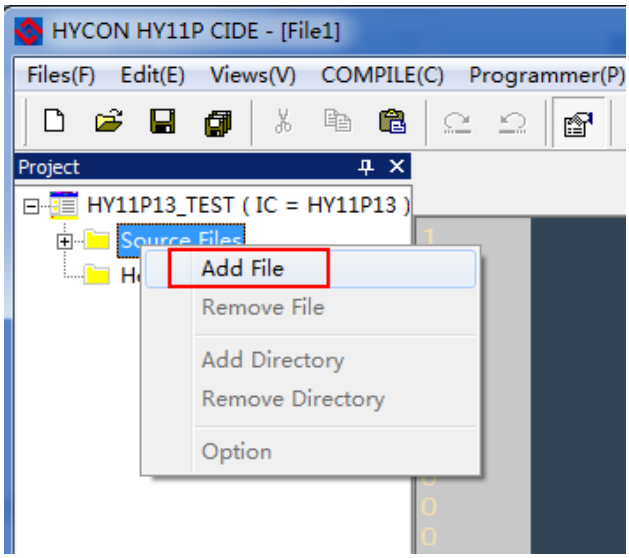


图2-7

2.3 C 程序架构

HYCON C 编译器基本支持标准C，配合HYCON MCU架构，该章节介绍撰写MCU的C语言应用程序。

C语言程序架构基本包含：IC母体声明、头文件声明、变量声明、主函数（main）定义、中断矢量函数、子函数、程序注解等，如下图2-8所示。

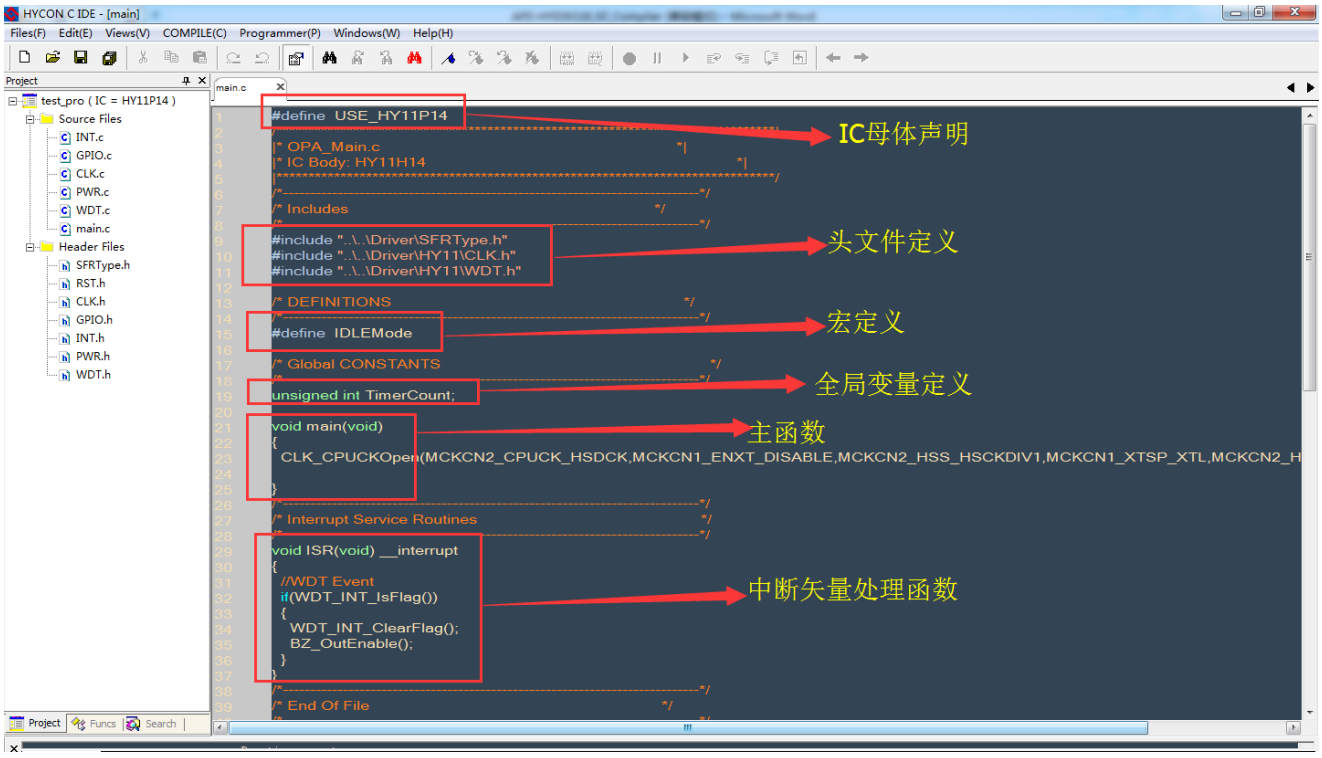


图2-8

IC 母体声明：

‘#define USE_HY11P14’在新建工程项目时选定母体后，自动添加到程序文件里，也可以在工程属性里修改；

头文件定义：

```
#include "..\..\Driver\SFRType.h"
#include "..\..\Driver\HY11\CLK.h"
#include "..\..\Driver\HY11\WDT.h"
```

使用#include关键字包含到程序里，且需要注意‘SFRType.h’头文件必须要包含进来，因为这个关于对应IC的寄存器定义；同时需要注意头文件的存放路径，这里一般都是使用相对路径，就是相对于工程项目路径；

宏定义：

```
#define  IDLEMode
```

使用关键字#define 定义一个标志符号；

全局变量：

作用域为全局的变量，用户自行定义；

主函数：

```
void main(void)
{
   CLK_CPUCKOpen(MCKCN2_CPUCK_HSDCK,MCKCN1_ENXT_DISABLE,MCKCN2_HSS_HSCKDIV1,
 MCKCN1_XTSP_XTL,MCKCN2_HSCK_HAO);
}
```

主程序函数其实也是函数，不过名称是main，所以有输入参数及返回参数；主程序main函数是程序执行起点，相当于汇编语言的start：

```
ORG 000H
JMP start

start：
....
```

中断矢量处理函数：

```
void ISR(void) __interrupt
 {
    //WDT Event
    if(WDT_INT_IsFlag())
    {
```

```
            WDT_INT_ClearFlag();
            BZ_OutEnable();
        }
    }
```

中断矢量处理函数是以‘void ISR(void)__interrupt’为名称的子函数；用于处理IC的中断功能及用户想要在中断里实现的功能。

2.4 调试程序

前面步骤完成，检查程序无误可以进行程序编译；没有连接开发工具时，只能点击‘Compile/Build’或者‘Compile/Rebuild All’。检查软件右下方提示工具已连接，用户可以点击‘Compile/Debug/C Language’或者‘Compile/Debug/ASM’直接进入debug界面；同时用户也可以点击以下命令按钮进入Debug界面，如图2-9。只有点击‘Compile/Build’或者‘Compile/Rebuild All’才能产生bin或hex档；

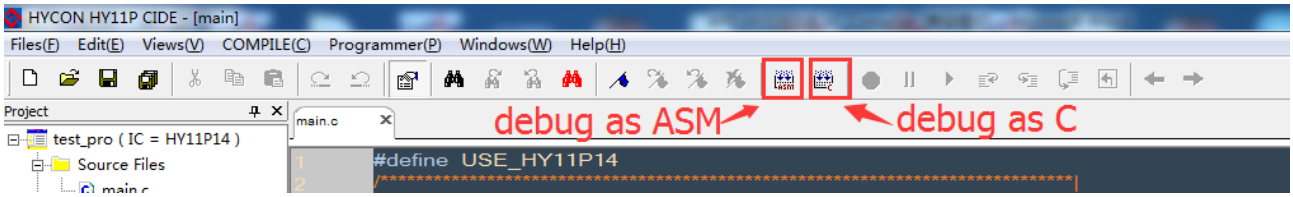


图 2-9

进入到debug界面后，用户可以进行单步调试，查看变量、打开寄存器视窗或者打开对应GUI界面进行操作等面向用户的可视化操作；图2-10是Debug命令按钮，提供调试程序用到的命令；图2-11是通过命令‘Compile/Windows’打开对应的可视化窗口；

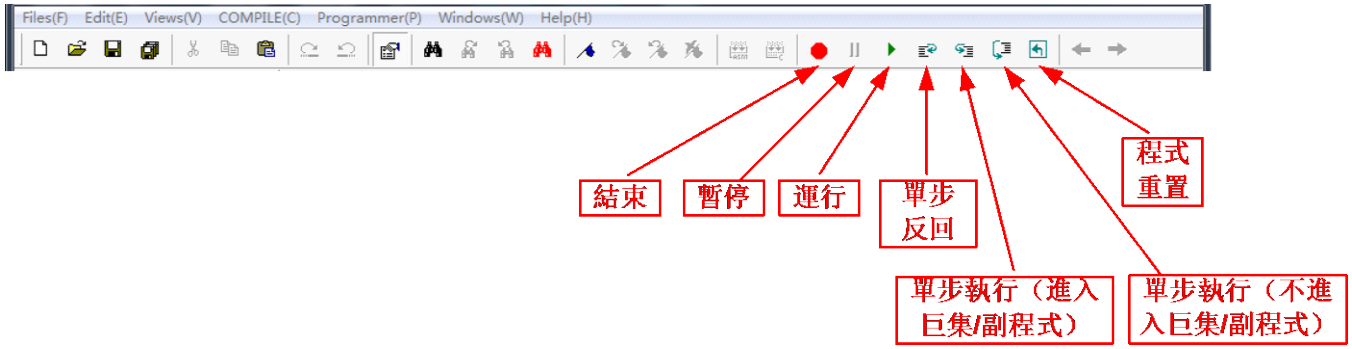


图 2-10 Debug 命令按钮

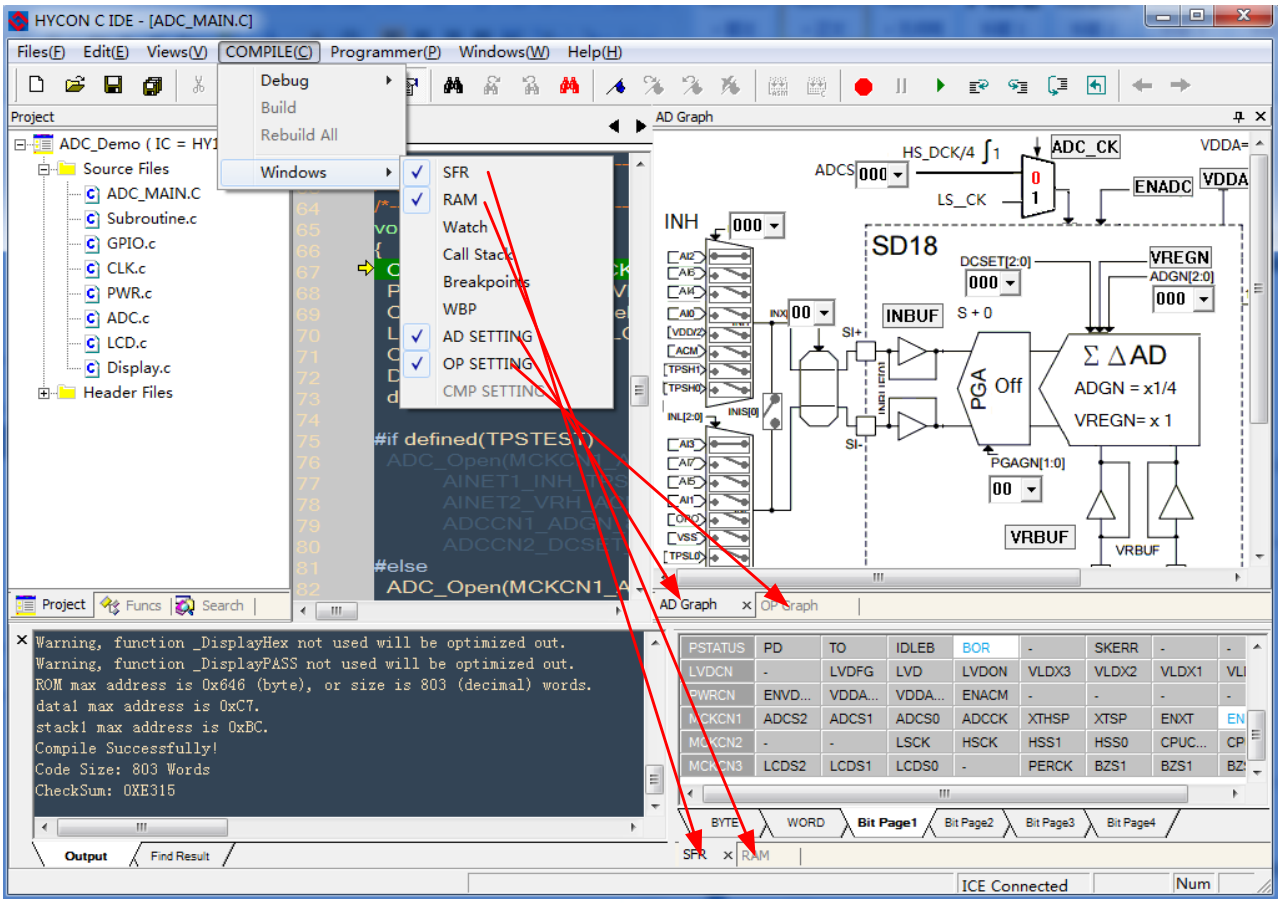

HYCON C IDE - [ADC_MAIN.C]
Files(F) Edit(E) Views(V) COMPILE(C) Programmer(P) Windows(W) Help(H)
Debug
Build
Rebuild All
Windows
SFR
RAM
Watch
Call Stack
Breakpoints
WBP
AD SETTING
OP SETTING
CMP SETTING
Project
ADC_Demo (IC = HY1
Source Files
ADC_MAIN.C
Subroutine.c
GPIO.c
CLK.c
PWR.c
ADC.c
LCD.c
Display.c
Header Files
AD Graph
SD18
INBUF
PGA
Off
ADGN = x1/4
VREGN= x 1
VRBUF
Warning, function _DisplayHex not used will be optimized out.
Warning, function _DisplayPASS not used will be optimized out.
ROM max address is 0x646 (byte), or size is 803 (decimal) words.
data1 max address is 0xC7.
stack1 max address is 0xBC.
Compile Successfully!
Code Size: 803 Words
CheckSum: 0XE315
Output
Find Result
Bit Page1
ICE Connected
Num

图 2-11 开启对应视窗

3. C 语言基础知识

本章节概括介绍 C 语言基础知识，方便无 C 基础知识开发人员快速学习 HYCON C IDE。由于是基于单片机硬件结构，因而只描述标准 C 语言的相关知识，主要内容包括如下：

- 标识符
- 数据类型
- 数组与指针
- 结构体、联合体与枚举类型
- 预处理
- 流程控制
- 作用域

3.1 标识符、常量、变量

3.1.1 标识符

在 C 语言中有很多东西需要命名，如变量名、函数名、数组名等，在命名的时候都需要遵守一定的命名规则，按照此规则命名的符号统称为标识符。在 C 语言中，合法的标识符是由字母、数字和下划线组成。

遵守以下命名规则：

- 必须以字母（大写或小写皆可）或者下划线开头；
- 随后可跟随若干个（包括 0 个）字母、数字、下划线；
- 标识符长度各个系统有不同，最好不要超过 8 个字符；

如：area　PI　_ini　a_array　s1234　P101p　　（合法）

456P　cade-y　w.w　a&b　　（不合法）

注意：C 语言是区分大小写字母的，如 Page 和 page 是两个不同的标识符；

在 C 语言中，标识符分三类：a）关键字　b）预定义标识符　c）用户标识符。

a） 关键字：C 语言规定一批标识符，它们在程序中代表着固定的含义，不能不能另作他用。例如用来定义数据类型的标识符 int、float 以及 if 语句中的 if、else 等都有专门的用途，它们不能不能再用作变量名或者函数名。请注意，所有的关键字必须小写。

b） 预定义标识符：这些预定的标识符在 C 语言中也都有特定的含义，如 C 语言提供的库函数名（如 include）和预编译处理命令（define）等。

c） 用户标识符：由用户根据需要而定义的标识符称为用户标识符；一般用来给变量、函数、数组或文件等命名；如上述定义的 area 、PI、a_array 三个变量。

3.1.2 常量

在程序运行过程中，其值不能被改变的量称为常量。常量有整型常量、实数常量、字符常量、字符串常量及符号常量。

整型常量：10、-2、0

实数常量（浮点型）：3.1、2.0、0.0、4.57

字符常量：‘c’、‘e’

字符串常量：“HYCON”、“Bei jing”

符号常量：在 C 语言中可以用一个标识符来代表一个常量，称为符号常量；但是这个标识符必须在程序中用（#define）特定指令进行特别的指定；例如 #define PI 3.14159 。

3.1.3 变量

变量是指在程序运行过程中，其值可以被改变的量。每个变量都有定义一个标识符，即变量名称，以便被区别及引用。C 语言规定，所有变量都必须先声明变量再使用；在程序中，声明变量就是为变量申请某个内存单元，使用变量实质就是把数据存入该变量所代表的内存单元中。比如:

```
int a;  //声明 a 为带符号整型变量
char b;  //声明 b 为字符型变量
```

在 C 语言中每个变量的存储方式有 2 种，静态存储和动态存储；具体又包含 4 种，自动的（auto）、静态的（static）、寄存器的（register）、外部的（external）、volatile。

I） auto：函数中的局部变量，如果不专门声明为 static 的存储方式，则默认为 auto，所以在函数内 auto char a 与 char a 是等价的。

II） static：分为局部静态存储及全局静态存储，全局变量加 static 声明后，变量只能再本文件范围内引用；局部静态存储则是局部变量的值在函数调用结束后不消失而保存原值，在下一次调用该函数时，该变量已经有值。

III） register：上述两种变量是存放在内存中，而 register 则是把变量存放在寄存器中，这里不展开叙述。

IV） extern：使用另一个文档中定义的变量，表示该变量是一个已经在外部文件定义过的变量，只要加上 external 声明就可以使用该变量。

V） volatile：一个类型修饰符；用来修饰被不同程序访问和修改的变量，使用 volatile 声明的变量，不会因编译程序的优化而被省去；建议用 volatile 声明的变量：特殊寄存器，中断函数用到的变量，为某些特殊用途的代码定义的变量；其他一般变量不建议声明为 volotile，这样会降低编译程序的优化效率。

3.1.4 作用域

无论是变量还是函数，都具有其作用域。全局变量/ 函数在整个工程中使用 extern 后就可以使用，如果在全局变量加 static 则只能在本档内使用，为了节省 ROM 空间，函数一般不使用 static，局部变量在函数内申请之后的语句都可以使用，如果是在循环，if，switch 内申请的变量，则执行完这些语句之后，下面的语句就不能再使用这些变量。

3.2 数据类型、运算符与表达式

3.2.1 数据类型

数据类型确定变量在内存中占用存储单元空间的大小及数据取值范围，所以在声明变量时首要确定变量的数据类型。数据类型分为基本数据类型、构造数据类型、指针类型（Pointer）和空类型（void）；基本数据类型

有整型、浮点型、字符型；构造数据类型有数组、结构体、共同体和枚举类型。列举 HYCON HY11 C Compiler 基本数据类型如表 3-2-1。

	数据类型	占用空间 (byte)	取值范围	备注
字符型	char	1	0 ~ 255	默认 unsigned
整型	int	2	-32768 ~ 32767	默认 signed
无符号整型	unsigned int	2	0 ~ 65535	
短整型	short 或 short int	2	-32768 ~ 32767	
无符号短整型	unsigned short	2	0 ~ 65535	
长整型	long 或 long int	4	-2147483648~2147483647	默认 signed
无符号长整型	unsigned long	4	0~4294967295	
长长整型	long long	8	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	默认 signed
无符号长长整型	unsigned long long	8	0 ~ 18,446,744,073,709,551,615	
浮点型	float	4	$3.4x10^{-38}$~$3.4x10^{38}$	IEEE-754
	General Pionter	3	MSB=1 means pointer to ROM	
	_data pointer	2	_data 可转换为_xdata	
	Const pointer (_code pointer)	2	单位是 Byte	
	Function pointer	2	单位是 Byte	
布尔型	‘bool’	（1/8）or 1	0、1；全局变量占 1bit，局部变量占 1Byte;不支持在 struct 里定义 bool	include<stdbool.h>

表 3-2-1 基本数据类型

3.2.2 运算符与表达式

运算符就是完成某种特定运算的符号。运算符按其表达式中与运算符的关系可分为单目运算符，双目运算符和三目运算符。单目就是指需要有一个运算对象，双目就要求有两个运算对象，三目则要三个运算对象。表达式则是由运算及运算对象所组成的具有特定含义的式子。C 是一种表达式语言，表达式后面加“;”号就构成了一个表达式语句。

各种运算符及其之间的优先级如表 3-2-2。

优先级	运算符	含义	运算类型	结合性
1	()	圆括号	单目	自左向右
	[]	下标运算符		
	->	指向结构成员运算符		

	.	结构体运算符		
2	!	逻辑非运算	单目	自右向左
	~	按位取反运算符		
	++、--	自增、自减运算符		
	（类型关键字）	强制类型转换		
	+、-	正负号运算符		
	*	指针运算符		
	&	取地址运算符		
	sizeof	长度运算符		
3	*、/、%	乘、除、求余运算符	双目	自左向右
4	+、-	加减运算符	双目	自左向右
5	<<	左移运算符	双目	自左向右
	>>	右移运算符		
6	<、<=、>、>=	小于、小于等于、大于、大于等于	关系	自左向右
7	==、!=	等于、不等于	关系	自左向右
8	&	按位与运算符	位运算	自左向右
9	^	按位异或运算符	位运算	自左向右
10	|	按位或运算符	位运算	自左向右
11	&&	逻辑与运算符	位运算	自左向右
12	||	逻辑或运算符	位运算	自左向右
13	? :	条件运算符	三目	自右向左
14	=、+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=	赋值运算符	双目	自右向左
15	,	逗号运算符	顺序	自左向右

表 3-2-2 运算符优先级

3.3 C 语言运算

C 语言运算主要包括算术运算、逻辑运算、位运算、赋值运算、条件运算、逗号运算等。各种运算符及其之间的优先级参考表 3-2-2。

3.3.1 数据类型转换

不同类型的数据（整型、实型、字符型）可以进行混合运算。运算时，如果参加运算的数据类型不同时，则首先要将它们转换为同一数据类型后再进行运算。类型转换基本原则是将优先级别的类型转换成优先级别高的类型，得到的结果为优先级别高的类型。

类型转换规则，具体如图 3-3-1：

- 混合类型的算术运算
  - ✧ 小于int类型的转换为int类型；

- ✧ 小类型想大类型转换；

- 不同类型之间的赋值
  - ✧ 以赋值语句左边的类型为转换后的类型；
- 函数参数/返回值的传递
  - ✧ 以参数/返回值的类型为转换后类型；

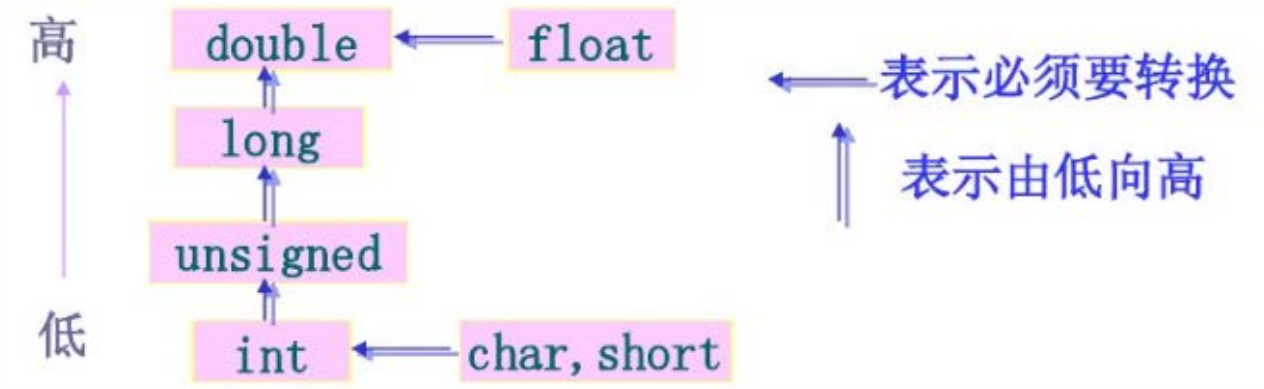


图3-3-1 数据类型转换优先级

类型转换例程：

```
float a = 2.0; int b = 6; int c =3;
void main (void)
{
   a=a*b/c-1.5;
}
```

结果：2.5；

说明：a为float，b为int，两者相乘，b先转换为float，得到a*b=12.0；

c为int，则12.0/c，将c转换为float，在进行除法，12.0/3.0=4.0；

最后4.0-1.5=2.5；

3.3.2 强制类型转换符

强制类型转换符可以将一个表达式强制转换为所需的数据类型；其一般形式为：

（数据类型）（表达式）

如：float PI=3.14159；int b；b= (int) PI ；运算后 b=3；

注意：在强制类型转换时，得到一个所需类型的中间变量，原来变量的类型没有发生变化。

类型转换例程：

```
int a = 20000; int b = 20000;
void main(void)
{
    long c = a + b;
}
```

结果： c = -25536；

```
char a = 100; char b =100;
void main(void)
{
    int c = a + b;
}
```

结果为c = 200；

说明：a 与b都是int 类型，使用int计算，结果为40000，超出int 的范围，所以结果为-25536； 解决方式： Long c = (long) a + b; 结果：c = 40000;	说明：a 与 b都小于int，转换为int 计算，结果为200；

3.3.3 位运算

C 语言中两个很具魅力的地方，一个是指针，另一个是位运算，本小节讲述的便是C 语言中的位运算，位运算符的作用是按位对变量进行运算，但是并不改变参与运算的变量的值。如果要求按位改变变量的值，则要利用相应的赋值运算。还有就是位运算符是不能用来对浮点型数据进行操作的。C 语言中位运算共有6 种，按位与(&)、按位或(|)、按位异或(^)、取反(~)、左移(<<) 和右移(>>)，巧妙利用位运算可以简化很多运算时间。

位运算一般的表达形式如下：

变量1 位运算符 变量2

位运算符也有优先级，从高到低依次是:“~”(按位取反)→“<<”(左移) →“>>”(右移) →“&”(按位与)→“^”(按位异或)→“|”(按位或)

表3-3-2是位逻辑运算符的真值表，X 表示变量1，Y 表示变量2

X	Y	~X	~Y	X&Y	X|Y	X^Y
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

表3-3-2 按位取反，与，或和异或的逻辑真值表

常见应用及操作：

- ✧ 把小写字母变为大写字母，清位：‘a’ & 0xDF，结果为‘A’
- ✧ 把大写字母变为小写字母，置位：‘A’ | 0x20，结果为‘a’
- ✧ 对某位取反， 某个位与1 异或即为取反(第1 位取反)：0xFF ^ 0x01,，运算的结果为0xFE
- ✧ 部分乘法的化简，与2 的n 次方相乘，相当于左移n 位，例如0x02 乘以4，0x02 << 2，这里的2，表示4 = 2 的‘2’次方，结果为8
- ✧ 部分除法的化简，与2 的n 次方相除，相当于右移n 位，例如0x08 除以4，0x08 >> 2，这里的2，表示4 = 2 的‘2’次方，结果为2
- ✧ 部分求余的化简，与2 的n 次方求余，跟2 的n 次方-1 与，如15 跟8 求余，相当于15 & 7，这里的7 是8-1 = 7，结果为8
- ✧ 其他乘法的化简，例如0x08 * 7 = 0x08 * (8 - 1) = (0x08 << 3) - 0x08
- ✧ 循环移位，对一个16 位的数循环左移n 位，0xXX >> (16 - n) | 0xXX << n
- ✧ 循环移位，对一个16 位的数循环右移n 位，0xXX << (16 - n) | 0xXX >> n

3.3.4 逻辑运算

关系运算符所能反映的是两个表达式之间的大小等于关系，那逻辑运算符则是用于求条件式的逻辑值，用逻辑运算符将关系表达式或逻辑量连接起来就是逻辑表达式了。要注意的是用关系运算符的运算结果只有0 和1 两种，也就是逻辑的真与假，换句话说也就是逻辑量，而逻辑运算符就用于对逻辑量运算的表达。逻辑表达式的一般形式为：

逻辑与：条件式1 && 条件式2

逻辑或：条件式1 || 条件式2

逻辑非：！条件式2逻辑运算具有短路性。

逻辑与，说白了就是当条件式1“与”条件式2 都为真时结果为真（非0 值），否则为假（0 值）。也就是说运算会先对条件式1 进行判断，如果为真（非0 值），则继续对条件式2 进行判断，当结果为真时，逻辑运算的结果为真（值为1），如果结果不为真时，逻辑运算的结果为假（0 值）。如果在判断条件式1 时就不为真的话，就不用再判断条件式2 了，而直接给出运算结果为假。

逻辑或，是指只要二个运算条件中有一个为真时，运算结果就为真，只有当条件式都不为真时，逻辑运算结果才为假。

逻辑非则是把逻辑运算结果值取反，也就是说如果两个条件式的运算值为真，进行逻辑非运算后则结果变为假，条件式运算值为假时最后逻辑结果为真。

同样逻辑运算符也有优先级别，！（逻辑非）→&&（逻辑与）→||（逻辑或）,逻辑非的优先值最高。

如有 !True || False && True

按逻辑运算的优先级别来分析则得到（True 代表真，False 代表假）

!True || False && True

False || False && True //!Ture先运算得False

False || False //False && True运算得False

False //最终False || False得False

例程：

```
char a, b, c, d, e, f, g, h;
void main()
{
a = 0x41;
b = 0x31;
e, f;
c = 0xaa, d = 0x55, g = 0x5a, h = 0xa5;
if((c = a > b) || (d = a)){ // 逻辑或的短路功能
e = 0x18;
}
else{
e = 0x81;
}
```

```
if((g = a < b) && (h = a)){ // 逻辑与的短路功能
f = 0x18;
}
else{
f = 0x81;
}
}
```

运算结果：a = 0x41, b = 0x31, c = 0x01, d = 0x55, e = 0x18，f = 0x81, g = 0x00, h = 0xa5，d 和h 没有被赋值。

非运算(!) 的特殊应用：!!(0xXX)，当0xXX 不为0 时，两次取非运算的结果则为1。

3.3.5 算术运算符和算术表达式

C语言基本算术运算符包含+、-、*、/、%，使用算术运算符将操作数按照C语言规格连接起来的式子叫做算术表达式；算术表达式经过运算得到一个数值，叫做算术表达式值；例如：2*x+y-1/a 。每个运算符都有相应的优先级和结合性。

运算优先级

算术运算符具有优先级高低之分，具体优先级可参考表3-3-1。表达式如果存在多个运算符时，则运算的先后顺序是按运算符的优先级从高到低进行运算；同一级中运算符则按照从左到右进行运算；

结合性

单目运算符的结合性是从右往左的，即表达式是从右往左进行计算的，称之为右结合性；如a-(b+c)。

其余运算符的结合性是从左往右的，即表达式是从左往右进行计算的，称之为左结合性；如a-b+c。

基本运算符都是左结合性的。

使用基本运算符注意事项：

- ✧ 两个整数相除，其值为整数；如：5/3=1。
- ✧ 两个实数相除，其值为实数；如：1.0/2.0=0.5。
- ✧ 除数、被除数只要有一个为实数，则运算结果为double类型；如1/2.0=0.5。
- ✧ 求余运算符（%）的运算操作数必须为整型数据；如：7%4=3。

3.3.6 自增自减运算符

自增、自减运算符分别为++、--；其作用是使变量值自动加1或减1。其有两种用法：一是在变量之前使用，称之为前置；二是在变量之后使用，称之为后置；

前置：++i,--i ，即是先执行i+1或i-1，再使用i的值。

后置：i++,i-- ，即是先使用i的值，再执行i+1或i-1。

使用自增、自减运算符需要注意：

- ✧ ++、--是单目运算符，运算对象只能是变量，不能是常量或表达式。
- ✧ ++、--、+、-是同一级运算符，运算的结合方式是自右向左。

3.3.7 赋值运算符和赋值表达式

赋值运算符为‘=’，由赋值运算符组成的表达式称为赋值表达式，它的的形式如下：

变量名 = 表达式

赋值号的左边必须是一个代表某一存储单元的变量名；赋值运算符的功能是先求出右边表达式的值，再赋值给左边的变量。

例如：a=10；b=a；

使用时需要注意：

- ✧ 赋值运算符的优先级别只高于逗号运算符，比其他任何运算符的优先级别都低，且具有自右向左的结合性；例如：a=3+5/2。
- ✧ 赋值号的左侧只能是变量，不能是常量或表达式；例如：a+b=c是不合法的赋值表达式。
- ✧ 赋值号的右侧也可以是一个赋值表达式；例如：a=b=7+1。
- ✧ C语言规定最左边变量中所得到的新值即是赋值表达式的值。

在赋值号前添加其他运算符，即可构成复合赋值运算符；例如：+=、-=、<<=、>>=、*=、/=、%=、^=、&=、|=。

3.3.8 关系运算符

对于关系运算符，同样我们也并不陌生。C 中有六种关系运算符，这些家伙同样是在小时候学算术时学习过的：

＞ 大于

＜ 小于

＞＝ 大于等于

＜＝ 小于等于

＝＝ 等于

！＝ 等于

或者你是个非C 程序员，那么对前四个一定是再熟悉不过的了。而“==”在VB 或PASCAL等中是用“=”,“!=”则是用“not ”。

小学时的数学课就教授过运算符是有优先级别的，计算机的语言也不过是人类语言的一种扩展，这里的运算符同样有着优先级别。前四个具有相同的优先级，后两个也具有相同的优先级，但是前四个的优先级要高于后2 个的。

当两个表达式用关系运算符连接起来时，这时就是关系表达式。关系表达式通常是用来判别某个条件是否满足。要注意的是用关系运算符的运算结果只有0 和1 两种，也就是逻辑的真与假，当指定的条件满足时结果为1，不满足时结果为0。

表达式1 关系运算符 表达式2

如：I＜J，I==J,(I=4)>(J=3),J+I>J

3.3.9 逗号运算符

如果你有编程的经验，那么对逗号的作用也不会陌生了。如在VB 中“Dim a,b,c”的逗号就是把多个变量定义为同一类型的变量，在C 也一样，如“int a,b,c”,这些例子说明逗号用于分隔表达式用。但在C 语言中逗号

还是一种特殊的运算符，也就是逗号运算符，可以用它将两个或多个表达式连接起来，形成逗号表达式。逗号表达式的一般形式为：

表达式1，表达式2，表达式3……表达式n

这样用逗号运算符组成的表达式在程序运行时，是从左到右计算出各个表达式的值，而整个用逗号运算符组成的表达式的值等于最右边表达式的值，就是“表达式n”的值。在实际的应用中，大部分情况下，使用逗号表达式的目的只是为了分别得到名个表达式的值，而并不一定要得到和使用整个逗号表达式的值。要注意的还有，并不是在程序的任何位置出现的逗号，都可以认为是逗号运算符。如函数中的参数，同类型变量的定义中的逗号只是用来间隔之用而不是逗号运算符。

3.3.10 条件运算符

上面我们说过C 语言中有一个三目运算符，它就是“?:”条件运算符，它要求有三个运算对象。它可以把三个表达式连接构成一个条件表达式。条件表达式的一般形式如下:

逻辑表达式? 表达式1 : 表达式2

条件运算符的作用简单来说就是根据逻辑表达式的值选择使用表达式的值。当逻辑表达式的值为真时（非0 值）时，整个表达式的值为表达式1 的值；当逻辑表达式的值为假（值为0）时，整个表达式的值为表达式2的值。要注意的是条件表达式中逻辑表达式的类型可以与表达式1 和表达式2 的类型不一样。下面是一个逻辑表达式的例子。

如有a=1,b=2 这时我们要求是取ab 两数中的较小的值放入min 变量中，也许你会这样写：

```
if (a<b)
    min = a;
else
    min = b; //这一段的意思是当a<b 时min 的值为a 的值，否则为b 的值。
```

用条件运算符去构成条件表达式就变得简单明了了：

```
min = (a<b)? a : b
```

很明显它的结果及含意都和上面的一段程序是一样的，但是代码却比上一段程序少很多，编译的效率也相对要高，但有着和复合赋值表达式一样的缺点就是可读性相对效差。在实际应用时根据自己要习惯使用，就我自己来说我喜欢使用较为好读的方式和加上适当的注解，这样可以有助于程序的调试和编写，也便于日后的修改读写。

3.3.11 指针和地址运算符

我们学习数据类型时，学习过指针类型，知道它是一种存放指向另一个数据的地址的变量类型。指针是C 语言中一个十分重要的概念，也是学习C 语言中的一个难点。对于指针将会在第九课中做详细的讲解。在这里我们先来了解一下C 语言中提供的两个专门用于指针和地址的运算符：

*** 取内容**

& 取地址

取内容和地址的一般形式分别为：

变量 = * 指针变量

指针变量 = & 目标变量

取内容运算是将指针变量所指向的目标变量的值赋给左边的变量；取地址运算是将目标变量的地址赋给左

边的变量。要注意的是：指针变量中只能存放地址（也就是指针型数据），一般情况下不要将非指针类型的数据赋值给一个指针变量。

3.4 流程控制语句

程序可以按照编写的顺序执行，但是为了适应自己的需要，我们必须要转移或者改变程序的执行顺序，达到这些目的的语句叫作流程控制语句；C 语言有三种执行流程：顺序执行、选择执行、循环执行。C 语言可以通过条件语句控制程序的流程，从而形成程序的分支和循环。C 语言提供以下流程控制关键字：

- ✧ 选择控制：if、else、switch、case
- ✧ 循环控制：while、for、do
- ✧ 跳转语句：break、continue
- ✧ 预编译控制：#if、#else、#endif

3.4.1 if 和 switch 语句使用

就如学习语文中的条件语句一样，C 语言也一样是“如果 XX 就 XX”或是“如果 XX 就 XX 否则 XX”。也就是当条件符合时就执行语句。条件语句又被称为分支语句，也有人会称为判断语句，其关键字是由 if 构成，这大众多的高级语言中都是基本相同的。C 语言提供了 3 种形式的条件语句:

A） if (条件表达式) 语句

当条件表达式的结果为真时，就执行语句，否则就跳过。

如 if (a==b) a++; 当 a 等于 b 时，a 就加 1

B） if (条件表达式) 语句 1

else 语句 2

当条件表达式成立时，就执行语句 1，否则就执行语句 2

如 if (a==b)

a++;

else

a--;

当 a 等于 b 时，a 加 1，否则 a-1。

C）if (条件表达式 1) 语句 1

else if (条件表达式 2) 语句 2

else if (条件表达式 3) 语句 3

else if (条件表达式 m) 语句 n

else 语句 m

这是由 if else 语句组成的嵌套，用来实现多方向条件分支，使用应注意 if 和 else 的配对使用，要是少了一个就会语法出错，记住 else 总是与最临近的 if 相配对。一般条件语句只会用作单一条件或少数量的分支，如果多数量的分支时则更多的会用到下一篇中的开关语句。如果使用条件语句来编写超过 3 个以上的分支程序的话，会使程序变得不是那么清晰易读。

if 和 switch 都具有判断的作用，当 if 的条件情况太多时一般使用 switch 替换， switch 的条件的类型只可以是除浮点型之外的基本数据类型和枚举型。case 只能带常量(不包括 const 申请的常量)，每个 case 执行后要使用 break，否则会继续执行下面的语句，多个 case 可以执行同一些语句，default 是默认的情况，default 可以不加 break。它的语法如下：

```
switch (表达式)
{
    case 常量表达式 1: 语句 1; break;
    case 常量表达式 2: 语句 2; break;
    case 常量表达式 3: 语句 3; break;
    case 常量表达式 n: 语句 n; break;
    default: 语句
}
```

运行中 switch 后面的表达式的值将会做为条件，与 case 后面的各个常量表达式的值相对比，如果相等时则执行 case 后面的语句，再执行 break（间断语句）语句，跳出 switch 语句。如果 case 后没有和条件相等的值时就执行 default 后的语句。当要求没有符合的条件时不做任何处理，则可以不写 default 语句。

例程如下：

```
unsigned char f;
……
switch(f){
        case 12:
        case 13:
                f += 1;
                break;
        case 14:
                f += 2;
                break;
        default:
                f += 3;
        }
```

3.4.2 循环语句

循环语句是几乎每个程序都会用到的，它的作用就是用来实现需要反复进行多次的操作。如一个 12M 的芯片应用电路中要求实现 1 毫秒的延时，那么就要执行 1000 次空语句才可以达到延时的目的（当然可以使用定时器来做，这里就不讨论），如果是写 1000 条空语句那是多么麻烦的事情，再者就是要占用很多的存储空间。我们可以知道这 1000 条空语句，无非就是一条空语句重复执行 1000 次，因此我们就可以用循环语句去写，这样不但使程序结构清晰明了，而且使其编译的效率大大的提高。在 C 语言中构成循环控制的语句有 while,do-while,for 和 goto 语句。同样都是起到循环作用，但具体的作用和用法又大不一样。我们具体来看看。

goto 语句

这个语句在很多高级语言中都会有，它是一个无条件的转向语句，只要执行到这个语句，程序指针就会跳转到 goto 后的标号所在的程序段。它的语法如下：

goto 语句标号;

其中的语句标号为一个带冒号的标识符。示例如下

```
void main(void)
{
    unsigned char a;
start: a++;
    if (a==10) goto end;
    goto start;
 end:;
}
```

上面一段程序只是说明一下 goto 的用法，实际编写很少使用这样的手法。这段程序的意思是在程序开始处用标识符“start:”标识，表示程序这是程序的开始，“end:”标识程序的结束，标识符的定义应遵循前面所讲的标识符定义原则，不能用 C 的关键字也不能和其它变量和函数名相同，不然就会出错了。程序执行 a++,a 的值加 1，当 a 等于 10 时程序会跳到 end 标识处结束程序，否则跳回到 start 标识处继续 a++,直到 a 等于 10。上面的示例说明 goto 不但可以无条件的转向,而且可以和 if 语句构成一个循环结构，这些在 C 程序员的程序中都不太常见，常见的 goto 语句用法是用它来跳出多重循环，不过它只可以从内层循环跳到外层循环，不能从外层循环跳到内层循环。在下面说到 for 循环语句时再略为提一提。为何大多数 C 程序员都不喜欢用 goto 语句？那是因为过多的使用它时会程序结构不清晰，过多的跳转就使程序又回到了汇编的编程风格，使程序失去了 C 的模块化的优点。

while 语句

while 语句的意思很容易理解，在英语中它的意思是“当…的时候…”，在这里我们可以理解为“当条件为真的时候就执行后面的语句”，它的语法如下：

while (条件表达式) 语句;

使用 while 语句时要注意当条件表达式为真时，它才执行后面的语句，执行完后再次回到 while 执行条件判断，为真时重复执行语句，为假时退出循环体。当条件一开始就为假时，那么 while 后面的循环体（语句或复合语句）将一次都不执行就退出循环。在调试程序时要注意 while 的判断条件不能为假而造成的死循环，调试时适当的在 while 处加入断点，也许会使你的调试工作更加顺利。当然有时会使用到死循环来等待中断或 IO 信号等。下面的例子是显示从 1 到 10 的累加和，读者可以修改一下 while 中的条件看看结果会如果，从而体会一下 while 的使用方法。

```
void main(void)
{
   unsigned int I = 1;
   unsigned int SUM = 0; //设初值
   while(I<=10)
```

```
    {
      SUM = I + SUM; //累加
      printf ("%d SUM=%d\n",I,SUM); //显示
      I++;
    }
    while(1); //这句是为了不让程序完后，程序指针继续向下造成程序“跑飞”
}//最后运行结果是 SUM=55;
```

do while 语句

do while 语句可以说是 while 语句的补充，while 是先判断条件是否成立再执行循环体，而 do while 则是先执行循环体，再根据条件判断是否要退出循环。这样就决定了循环体无论在任何条件下都会至少被执行一次。它的语法如下：

do 语句 while (条件表达式)

用 do while 怎么写上面那个例程呢？先想一想，再参考下面的程序。

```
void main(void)
{
    unsigned int I = 1;
    unsigned int SUM = 0; //设初值
    do
    {
      SUM = I + SUM; //累加
      printf ("%d SUM=%d\n",I,SUM); //显示
      I++;
    }
    while(I<=10);
    while(1);
}
```

在上面的程序看来 do while 语句和 while 语句似乎没有什么两样，但在实际的应用中要注意任何 do while 的循环体一定会被执行一次。如把上面两个程序中 I 的初值设为 11，那么前一个程序不会得到显示结果，而后一个程序则会得到 SUM=11。

for 语句

在明确循环次数的情况下，for 语句比以上说的循环语句都要方便简单。它的语法如下：

for ([初值设定表达式];[循环条件表达式];[条件更新表达式])

语句中括号中的表达式是可选的，这样 for 语句的变化就会很多样了。for 语句的执行：先代入初值，再判断条件是否为真，条件满足时执行循环体并更新条件，再判断条件是否为真……直到条件为假时，退出循环。下面的例子所要实现的是和上二个例子一样的，对照着看容易理解几个循环语句的差异。

```
void main(void)
```

```
{
    unsigned int I;
    unsigned int SUM = 0; //设初值
    for (I=1; I<=10; I++) //这里可以设初始值，所以变量定义时可以不设
    {
      SUM = I + SUM; //累加
      printf ("%d SUM=%d\n",I,SUM); //显示
    }
    while(1);
 }
```

如果我们把程序中的 for 改成 for(; I<=10; I++)这样条件的初值会变成当前 I 变量的值。如果改成 for(;;)会怎么样呢？试试看。

3.4.3 break 与 continue 语句

break 只能用于循环和 case 语句，continue 只能用于循环语句，两者的区别是： 在循环中如果遇到 continue 则执行下一次该循环体的语句，break 则是直接跳出了本层循环体，执行循环体外的语句。

continue 语句

continue 语句是用于中断的语句，通常使用在循环中，它的作用是结束本次循环，跳过循环体中没有执行的语句，跳转到下一次循环周期。语法为：

continue;

continue 同时也是一个无条件跳转语句，但功能和前面说到的 break 语句有所不同，continue 执行后不是跳出循环，而是跳到循环的开始并执行下一次的循环。

例子程序：

```
while(1)
{
    int j = 0;
    while(1)
    {
      j++;
      if(j == 5)
      {
        continue; // 执行本层循环的下一次循环
      }
      if(j == 10)
      {
        break; // 跳出本层循环
      }
```

```
        }
    }
```

3.5 函数

函数是相当于子程序，每一个函数都可用来实现一个特定的功能。在程序开发的过程中，常使用的功能模块编写成函数，以减少重复编写程序段的工作量，函数的本质是一个标号，即一个地址，调用该函数，相当于跳转到这个地址去执行指令

3.5.1 函数的声明与定义

通常 C 语言的编译器会自带标准的函数库，这些都是一些常用的函数，HYCON Compile 中也不例外。标准函数已由编译器软件商编写定义，使用者直接调用就可以了，而无需定义。但是标准的函数不足以满足使用者的特殊要求，因此 C 语言允许使用者根据需要编写特定功能的函数，要调用它必须要先对其进行定义。定义的模式如下：

```
函数类型 函数名称（形式参数表）
{
    函数体
}
```

函数类型是说明所定义函数返回值的类型。返回值其实就是一个变量，只要按变量类型来定义函数类型就行了。如函数不需要返回值函数类型可以写作“void”表示该函数没有返回值。注意的是函数体返回值的类型一定要和函数类型一致，否则会造成错误。函数名称的定义在遵循 C 语言变量命名规则的同时，不能在同一程序中定义同名的函数这将会造成编译错误（同一程序中是允许有同名变量的，因为变量有全局和局部变量之分）。形式参数是指调用函数时要传入到函数体内参与运算的变量，它可以有一个、几个或没有，当不需要形式参数也就是无参函数，括号内可以为空或写入“void”表示，但括号不能少。函数体中可以包含有局部变量的定义和程序语句，如函数要返回运算值则要使用 return 语句进行返回。在函数的｛｝号中也可以什么也不写，这就成了空函数，在一个程序项目中可以写一些空函数，在以后的修改和升级中可以方便的在这些空函数中进行功能扩充。

例子程序：求两个数的较大者。

```
int max(int, int); //  函数声明
int a;
void main()
{
    a = max(10, 20); //  函数调用
    a++;
}
int max(int a, int b) //  返回值类型  函数名(  参数列表)
{
```

```
        return a > b ? a : b;
    }
    运算结果 a = 21
```

3.5.2 函数的参数列表与返回值

参数列表，函数名之后，用括号括起来，在调用函数时要传递给函数的变量列表称为参数列表。

关于形参与实参的说明：

I） 实参可以是常量、变量或表达式，但要求具有确定的值，在函数调用时，把值赋给形参。

II） 函数的定义时，必须指定形参的数据类型。

III） 实参与形参必须要兼容。

IV） 形参的最大特点是单向值传递，即只是把值传过来，而并没有改变实际参数的值。

V） 在 HYCON C compiler 中，参数及函数局部变量的命名方式是_funname_2[n]，比如：函数 fun 的变量命名为_fun_2，n 则表示它共有多少个变量。

VI） 若返回值为 1byte，则存于 WREG，若为 2byte，则低字节存于 ra，高字节存于 rb，若为四 byte，由低字节到高字节分别存于 ra、rb、rc、rd。

VII） 由于 MCU 的限制(没有堆栈)，参数和内部变量会占用 RAM 空间，而互不影响(没有调用关系) 的函数变量可以共享同样的 RAM 空间。

例子程序：

```
    int a;
    void change(int b)
    {
        b = 7;
    }
    void main()
    {
        a = 15;
        change(a);
    }
    运算结果 a = 15
```

返回值是把运算的结果返回给该函数的调用者使用，void 类型的函数是否可以使用 return 呢？可以的，只是 return 后面不要加任何表达式。

3.5.3 函数的调用

调用就是指一个函数体中引用另一个已定义的函数来实现所需要的功能，这时函数体称为主调用函数，函数体中所引用的函数称为被调用函数。一个函数体中可以调用数个其它的函数，这些被调用的函数同样也可以调用其它函数，也可以嵌套调用。在C语言中有一个函数是不能被其它函数所调用的，它就是main 主函数。调用函数的一般形式如下:

函数名 (实际参数表)

“函数名”就是指被调用的函数。实际参数表可以为零或多个参数，多个参数时要用逗号隔开，每个参数的类型、位置应与函数定义时所的形式参数一一对应，它的作用就是把参数传到被调用函数中的形式参数，如果类型不对应就会产生一些错误。调用的函数是无参函数时不写参数，但不能省后面的括号。

函数的调用方式有3 种：

- 函数语句：只把函数调用作为一个语句，完成一定的功能，如change(a);
- 函数表达式：如int a = 3*max(10, 20);
- 函数参数：如int a = max(max(20, 30), 20);

调用函数前要对被调用的函数进行说明。标准库函数只要用#include 引入已写好说明的头文件，在程序就可以直接调用函数了。如调用的是自定义的函数则要用如下形式编写函数类型说明

类型标识符 函数的名称(形式参数表);

这样的说明方式是用在被调函数定义和主调函数是在同一文件中。你也可以把这些写到文件名.h 的文件中用#include "文件名.h"引入。如果被调函数的定义和主调函数不是在同一文件中的，则要用如下的方式进行说明，说明被调函数的定义在同一项目的不同文件之上，其实库函数的头文件也是如此说明库函数的，如果说明的函数也可以称为外部函数。

extern 类型标识符 函数的名称(形式参数表);

函数的定义和说明是完全不同的，在编译的角度上看函数的定义是把函数编译存放在ROM 的某一段地址上，而函数说明是告诉编译器要在程序中使用那些函数并确定函数的地址。如果在同一文件中被调函数的定义在主调函数之前，这时可以不用说明函数类型。也就是说在main 函数之前定义的函数，在程序中就可以不用写函数类型说明了。可以在一个函数体调用另一个函数（嵌套调用），但不允许在一个函数定义中定义另一个函数。还要注意的是函数定义和说明中的“类型、形参表、名称”等都要相一致。

HYCON C Compiler支持math.h、string.h、stdlid.h等ANSI C标准库函数；具体函数参考附录。

3.5.4 内联函数

HYCON C Compiler提供inline关键字，使用关键词inline 修饰函数，则该函数为内联函数，内联函数直接把函数体的语句替换函数的调用。以节省函数调用时因保护现场的时间代价和使用堆层的空间代价。多次调用时会大量增加代码。

例程程序：

```
unsigned char max;
inline void getmax(unsigned char var1, unsigned char var2)
{
    max = var2 > var1 ? var2 : var1;
}
void main()
{
    getmax(23,32);
}
```

这里直接把max = var2 > var1 ? var2 : var1; 这句代码代换了getmax(23,32); 函数。

3.6 数组与指针

数组是有序数据的集合，数组中的每一个函数都属于同一种数据类型，使用数组名和下标可以唯一确定数组中的一个元素。内存区中每个字节都有一个编号， 这个编号称为地址，指针变量的值就存放这些编号。

3.6.1 数组使用

下面是定义一维或多维数组的方式：

数据类型 数组名 [常量表达式];

数据类型 数组名 [常量表达式1]..... [常量表达式N];

“数据类型”是指数组中的各数据单元的类型，每个数组中的数据单元只能是同一数据类型。

“数组名”是整个数组的标识，命名方法和变量命名方法是一样的。在编译时系统会根据数组大小和类型为变量分配空间，数组名可以说就是所分配空间的首地址的标识。

“常量表达式”是表示数组的长度和维数，它必须用“[ ]”括起，括号里的数不能是变量只能是常量。

unsigned int xcount [10]; //定义无符号整形数组,有10 个数据单元

char inputstring [5]; //定义字符形数组，有5 个数据单元

float outnum [10],[10];//定义浮点型数组，有100 个数据单元

数组的初始化方式：

- 定义一个数组，如unsigned char led_table[5];[2] 然后逐个元素赋值。
- 定义的同时初始化，如unsigned char led_table[5] = {0, 1, 2, 3, 4};
- 不指定个数的初始化，如unsigned char led_table[] = {0, 1, 2, 3, 4}; 这时会确定元素个数为5 个。
- 部分初始化，如unsigned char led_table[5] = {2, 1};[3]

注：[1]、数组下标由0 开始，表示第一个元素，最后一个元素为个数-1。

[2]、逐个赋值时，数组索引不能超过元素个数，如本处元素个数为10，则下标最大为9， 超过会溢出。

[3]、led_table[0] = 2，led_table[1] = 1 其他没有被赋值的值为0。

使用时，直接用数组名加索引号，如led_table[2]，由于C 语言不检查数组界限， 所以在使用时要注意数组下标溢出的问题。

3.6.2 字符串使用

在C 语言中字符串是以字符数组来表达处理的，以 ‘\0’结束，例如，字符串char c[] = “chip”，所以 “chip” 实际占用了5 个字符空间。如果一个字符数组，最后一个元素为 ‘\0’也可当字符串使用。

数组名的本质是数组在内存空间中的首地址，例如led_table[5]，则led_table 为led_table[0] 的地址。

3.6.3 指针使用

指针的概念

在程序中定义的变量，会在内存中分配相应的内存单元，而该内存单元有个编号，这就是“地址”，指针存放的内容就是该内存单元的编号，也就是地址。

指针类型

指针可以指向不同类型变量的地址(RAM 地址)，比如int、char 等，所以在定义指针时必须指定指针要指向的数据类型，特殊的指针类型如指向函数的指针， 指向指针的指针，指向多维数组的指针。指针的大小与指针的类型无关，不管是何种类型的指针，其值都固定的且只与体系结构有关，如char *p; long *q 而使用sizeof(p)和sizeof(q) 时，其值都为2。

指针操作

操作指针分3 步：定义指针，初始化指针，使用指针。特别要注意的是在使用指针时必须初始化，如果没有初始化指针就向指针指向的地址写入数值会引起不可预见的错误。指针的操作符有两个：&(取地址操作) 和*(取内容操作)。

A）操作指向变量的指针：

```
char a;
void change(char *b)
{
    *b = ‘b’;
}
void main()
{
    a = ‘a’;
    change(&a);
}
```

运行结果：a = ‘b’，由于传进来的参数是a 的地址，改变该地址里面的值就相当于改变实参的值。

B）操作指针的指针：

```
char *a;
void change(char *b)
{
    b = (char *)0x81; // 修改指针本身的值
}
void main()
{
    a = (char *)0x80;
    change(a);
}
```

运行结果：a = 0x80， 由于参数的传递是单向的，所以如果要改变a 的值，则改为如下代码：

```
char *a;
void change(char **b)
{
    *b = (char *)0x81; // 修改指针内容
}
void main()
{
    a = (char *)0x80;
    change(&a);
}
```

运行结果：a = 0x81

C）操作数组指针和指针数组：

数组指针是指指向数组的指针，定义形式如：char (*b)[5];

指针数组则是存放数据类型为指针的数组，定义形式如：char* a[5]。

有什么区别呢？在这里a 是数组的首地址，在数组a 中存放的是5 个指标值， 而b 是指标，它指向的是长度为5 个char 型的数组的首地址，如果用sizeof 运算符便知a 的长度为10，而b 的长度为2。

D）溢出的下标：

```
unsigned char a8[5] = {0,1,2,3,4};
unsigned char *p = a8;
*(p + 5) = 8;
```

运行结果：如果a8 在存放在内存为0x0085 的位置，则0x008a 地址的值为8。如果此时0x008a 存放一个很重要的数据，则影响了整个程序的运行结果。

E）操作常量指针和指针常量：

常量指针表示指针指向的内容是常量，定义方式如：const char *c_p1;

指针常量表示指针本身的值不能变，但指向的内容可以变，定义方式如：

```
char *const p2_c;
void main()
{
    char a[5] = {0, 1, 2, 3, 4}, b[5];
    const char *p = "Hello Word!";
    char * const q = a;
    *(p + 1) = 'o'; // 企图修改字符串常量的内容，报错。
    p = "Hello! Word!"; // 可以修改指向常量的指针。
    p = b; // 也可以指向非常量地址。
    *(q + 1) = 2; // 可以修改指针常量的指向的内容。
    q = b; // 企图修改常量指针的值，报错。
}
```

3.6.4 数组与指针区别

数组名的本质是该数组在内存中的首地址，与指针的概念神似，但数组名和指针却有绝对的不同之处：

- ✧ 数组名是常量，不可修改，相当于指针常量；
- ✧ sizeof 运算结果不同，数组名：占用的空间，指针：固定为2；
- ✧ 数组名不可自增、自减等；

3.7 结构体、联合体和枚举

为了更有效的处理更复杂的数据，C 语言引入了构造类型的数据类型。构造类型就是将一批各种类型的数据放在一起形成一种特殊类型的数据。之前讨论过的数组也算是一种构造类型的数据，C语言中的构造类型还有结构、枚举和联合。

3.7.1 结构体

结构体定义

结构是一种数据的集合体，它可以按需要将不同类型的变量组合在一起，整个集合体用一个结构变量名表示，组成这个集合体的各个变量称为结构成员。使用结构变量时，要先定义结构类型。一般定义格式如下：

struct 结构名 {结构元素表};

例子：struct FileInfo

```
{
    unsigned char FileName[4];
    unsigned long Date;
    unsigned int Size;
}
```

结构类型的定义还可以有如下的两种格式。

```
struct
{
    结构元素表
} 结构变量名1，结构变量名2……结构变量名N;
```

例：struct

```
{
    unsigned char FileName[4];
    unsigned long Date;
    unsigned int Size;
} NewFileInfo, OleFileInfo;
```

这一种定义方式定义没有使用结构名，称为无名结构。通常会用于程序中只有几个确定的结构变量的场合，不能在其它结构中嵌套。

另一种定义方式如下：

struct 结构名

{

结构元素表

} 结构变量名1，结构变量名2……结构变量名N;

例：struct FileInfo

{

unsigned char FileName[4];

unsigned long Date;

unsigned int Size;

} NewFileInfo, OleFileInfo;

结构体用法

使用结构变量时是通过对它的结构元素的引用来实现的。引用的方法是使用存取结构元素成员运算符“.”来连接结构名和元素名，格式如下：

结构变量名.结构元素

要存取上例结构变量中的月份时，就要写成 NowDate..year。而嵌套的结构，在引用元素时就要使用多个成员运算符，一级一级连接到最低级的结构元素。要注意的是在C语言中只能对最低级的结构元素进行访问，而不可能对整个结构进行操作。操作例子：

NowDate.year = 2005;

NowDate.month = OleMonth+ 2; //月份数据在旧的基础上加2

NowDate.Time.min++; //分针加1，嵌套时只能引用最低一级元素

一个结构变量中元素的名字可以和程序中其他地方使用的变量同名，因为元素是属于它所在的结构中，使用时要用成员运算符指定。

3.7.2 联合体

联合同样是 C 语言中的构造类型的数据结构。它和结构类型一样可以包含不同类型的数据元素，所不同的是联合的数据元素都是从同一个数据地址开始存放。结构变量占用的内存大小是该结构中数据元素所占内存数的总和，而联合变量所占用内存大小只是该联合中最长的元素所占用的内存大小。如在结构中定义了一个int和一个char，那么结构变量就会占用3 个字节的内存，而在联合中同样定义一个int 和一个char，联合变量只会占用2 个字节。这种能充分利用内存空间的技术叫‘内存覆盖技术’，它可以使不同的变量分时的使用同一个内存空间。使用联合变量时要注意它的数据元素只能是分时使用，而不能同时使用。举个简单的例子，程序先为联合中的int 赋值1000，后来又为char 赋值10,那么这时就不能引用int 了，否则程序会出错，起作用的是最后一次赋值的元素，而上一次赋值的元素就失效了。使用中还要注意定义联合变量时不能对它的值初始化、可以使用指向联合变量的指针对其操作、联合变量不能作为函数的参数进行传递，数组和结构可以出现在联合中。

联合类型变量的定义方法和结构的定义方法差不多，只要把关键字 struct 换用union 就可以了。联合变量的引用方法除也是使用‘.’成员运算符。

结构体与联合体区别

结构体与联合体的最主要区别是空间的分配，结构体会为各成员分配内存空间， 而联合体内各成员共享一个内存空间，以占用内存空间最大的成员为联合体分配内存空间。

如例子8 中，如果使用sizeof(__pa_type) 求联合体占用的空间得到结果是1，如果把_pa5 置1，则__pa_type 内的byte 成员的第5 位也会被置1。

3.7.3 枚举

在程序中经常要用到一些变量去做程序中的判断标志。如经常要用一个字符或整型变量去储存1 和0 做判断条件真假的标志，但我们也许会疏忽这个变量只有当等于0 或1 才是有效的，而将它赋上别的值，而使程序出错或变的混乱。这时可以使用枚举数据类型去定义变量，限制错误赋值。枚举数据类型就是把某些整型常量的集合用一个名字表示，其中的整型常量就是这种枚举类型变量的可取的合法值。枚举类型的二种定义格式如下：

enum 枚举名 {枚举值列表} 变量列表;

例 enum TFFlag {False, True} TFF;

enum 枚举名 {枚举值列表};

emum 枚举名变量列表;

例 enum Week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

enum Week OldWeek, NewWeek;

在枚举列表中，每一项名称代表一个整数值，在默认的情况下，编译器会自动为每一项赋值，第一项赋值为0，第二项为1……如Week 中的Sun 为0，Fri 为5。C 语言也允许对各项值做初始化赋值，要注意的是在对某项值初始化后，它的后续的各项值也随之递增。如：

enum Week {Mon=1, Tue, Wed, Thu, Fri, Sat, Sun};

上例的枚举就使 Week 值从1 到7，这样会更符合我们的习惯。使用枚举就如变量一样，但在程序中不能为其赋值。

3.8 预处理

3.8.1 预处理

标准C 语言中预处理有3 种：宏定义(#define)、档包含(#include)、条件编译(#if等)，预处理是在编译之前执行的，如宏展开的时机是在预编译时进行。

I） 宏定义：注意使用宏定义时，只是简单的把宏名替换为宏定义的内容，并不做任何运算，如#define S(r) 3.1415926*r*r，如果使用S(6+6)，结果就成了3.1415926*6+6*6+6，而不是最初想要的结果，可以使用#define S(r)3.1415926*(r)*(r) 得到想要的结果。

II） 档包含：使用#include 可以把相关档包含进来，包含档时，使用双引号和使用“<>”不同，一般双引号用于用户自定义的文件，“<>”用于库文件，以减少在编译时因搜索路径所花费的时间，注意头档的重复包含问题。

III） 条件编译：选择源代码中的一支来编译，而不用把全部源代码都进行编译，常用在调试和在不同机

器上移植代码时使用，有3 种形式(else 分支可以没有)：

(1)、#ifdef 条件

程序段1

#else

程序段2

#endif

(2)、#ifndef 条件

程序段1

#else

程序段2

#endif

(3)、#if 条件

程序段1

#else

程序段2

#endif

3.8.2 宏定义与内联函数的区别

(1)、宏定义只做简单的替换，替换时不做任何运算。

(2)、宏定义在预编译时进行，内联函数则是在有函数调用时进行。

(3)、宏定义的参数不能指定数据类型，内联函数则必需要指定。

(4)、编译程序对宏定义本身的内容不做任何检查。如使用#define error **8，不会报错，而内联函数则不行。

4. HYCON C Compiler 扩展功能

4.1 HYCON C Compiler 使用限制

HYCON C Compiler 使用 SDCC 3.6.0 的前端 (FRONT-END), 结合自行开发的优化引擎后端 (BACK-END), 可以产生有效率的机器码. 此 Compiler 支持大部份的 C99/C89/C11标准, 但由于芯片硬件与其他效率的考虑,有以下使用限制, 使用者在使用之前必须留意, 否则无法产生正确的机器码。

- 硬件的PC STACK, 目前支持的量产芯片都只有 6 层PC STACK;
- 除了HY15P系列之外, 没有支持递归调用功能(function-re-entry), 也就是没有支持 recursive function;
- 除了HY15P系列之外,在 interrupt 里不能使用乘除法运算.[可以使用加减/逻辑/左右移常数运算]除非别的地方不使用该运算;
- 有限的RAM 空间, 默认的0页寻址空间只有 256 BYTE, 其中会包括常用的本地变量和全局变量. 较大的 array 建议使用 "__xdata" 区间;
- 除了HY15P系列之外, 没有支持 variable length function parameter, 也就是没有支持 <stdio.h> 里的 "printf(...)" 和 "sprintf(...)", 使用者可以改用 "_itoa() 或 _ltoa()" 取代;
- 不支持64/128位的浮点数(long double);
- 除了HY15P系列之外, Function pointer 所指的FUNCTION 只能有 1byte 的自变量(parameter)。
- 因为目前支持量产的芯片都只有很小的RAM, 所以没有支持 heap/stack memory management. 也就是没有支持 malloc(...), free(...) 等函式;

4.2 HYCON C Compiler 特色

HYCON C Compiler除了支持大部分的C89/C99/C11标准，也根据本身支持的芯片资源，糅合了支持 HYCON 8-Bit MCU 开发的特色功能。

4.2.1 布尔型（bool）

HYCON 8-BIT CPU 支持BIT 的SET/CLEAR/TOGGLE/SKIP 动作, 所以COMPILER 对于全局 (GLOBAL) 的 “bool” 数据类型支持以1 BIT 储存,以节省空间。比如:

	数据类型	占用空间(byte)	取值范围	备注
布尔型	‘bool’	（1/8）or 1	0、1；全局变量占 1bit，局部变量占 1Byte;不支持在 struct 里定义 bool	include<stdbool.h>

```
#include <stdbool.h>
bool has_led0=true;
bool has_led1;
```

同一个C 文档里的BOOL会打包成每8 BIT 一组。但不同C 文檔的BOOL会占用不同的BYTE。因为全局变量在初始化时被设置为0，所以”bool” 的初始值若未赋值 则皆为”false”。目前COMPILER 支持”bool” 初始值为 “true”。

目前的限制是 ”bool” 不可以在数组(ARRAY)/结构体(STRUCT) 里使用。

“bool” 与其它数据型态的运算是支持的，但”bool” 的运算因为不经过”WREG” 缓存器，所以做运算时程序代码会比”unsigned char” 型态来得大。

4.2.2 存储指针类型

HYCON C Compiler支持多种存储指针类型，如下表。在介绍存储指针前先看看MCU的寻址空间分布。

	数据类型	占用空间(byte)	取值范围
通用存储指针	General Pionter	3	MSB=1 means pointer to ROM
RAM 存储指针	_data pointer	2	_data 可转换为_xdata
ROM 存储指针	Const pointer (_code pointer)	2	单位是 Byte
函数寻址指针	Function pointer	2	单位是 Byte

表4-2-1

C Compiler 寻址空间(Addressing Spaces)

HYCON 8 位CPU 是 HARVARD MACHINE，最多有 4K RAM SPACE，32K WORD CODE SPACE。但其上的RAM 并不连续，MEMORY MAP 如下图。

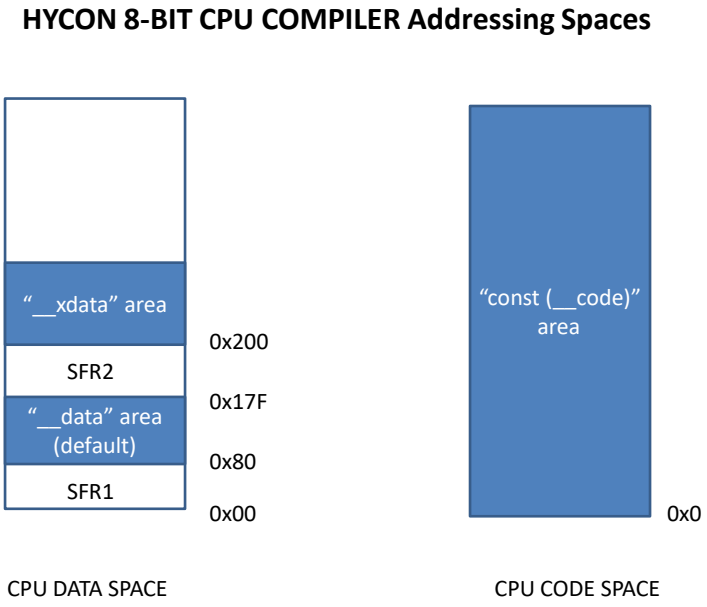


图4-2-1.C Compiler 寻址空间

COMPILER 产生的CODE，默认是 "__data" area，若不是要放在" __data" area，则要另外宣告，范例如下：

```
__xdata unsigned long   kk;
const unsigned char * __xdata ptr_array[5];
```

上面一行表示"kk" 是在 "__xdata" 的变量，其下一行表示在 __xdata 的 pointer array，那些pointer 都是指到ROM pointer。也就是说，如果量产芯片的RAM 大于256 BYTE，则比较大的ARRAY 建议可以放在"__xdata" area。

以硬件的角度来看，C COMPILER 在启动的时候会把BSR 设为1，也就可以用A BIT 寻址 00~1FF 的空间，其中256 BYTE 为RAM，256 BYTE 为SFR，其他的空间目前使用FSR 做间接寻址。这种做法对RAM SIZE <= 512 的产品有不错的效率，因为多是比较大的ARRAY 才会放到地址 0x200 之后。

寻址指针类型

通常若没有另外指定，C Compiler 会默认指针为通用指针（General Pionter），每个指针有 3 Bytes；注意这里所有指针都是以‘Byte’为单位，而 PC 以‘Word’为单位。通用指针的最高位(MSB) 为 1 时表示指向 ROM。其他类型的指针皆为 2 个 Byte。如果是通用指针，相关的动作不是比较慢就是 CODE-SIZE 比较大，所以有些 LIBRARY 使用要写入数据的 Pionter，就会预定义成”__xdata pointer”。因为”__data pointer”可以转换成” __xdata pointer”。

```
extern void __xdata *memcpy (void __xdata * dest, const void * src, size_t n);
```

值得一提的是，如果指针在”__data” 的区间，它的值可以由直接寻址的方式取得，程序就会比较短。如果在”__code” 上，取得要经过 TABLE LOOKUP 等间接寻址的方式，程序就会比较大。所以除非必要尽量不要使

用”通用指针”类型。

存储指针使用范例

一般如果指标只会指到 RAM，则建议定义”__xdata” 指标即可:

比如：

```
__xdata int* foo(__xdata int *k)
{
        (*k)++;
        return ++k;
}
```

上面是将‘foo’定址到__xdata，产生的程序会比下面的写法来得小。

```
int* foo(int *k)
{
        (*k)++;
        return ++k;
}
```

如果资料想要只放到 ROM 即可，可以参考下面的方式，将字符串以数组方式定义：

```
__code char modelname[]=”MODEL000”;
```

编译后会在 ROM 里占用一块内存，其产生的 ASSEMBLY 如下:

```
.area SEGC       (CODE) ;testb-0-code

_modelname:
        .ascii "MODEL000"
        .db 0x00
```

如果写成指针方式的字符串：

```
__code char *version = “VER1.0”;
```

编译后就有点复杂，它产生的 ASSEMBLY 如下：

```
        .area IDATAROM       (CODE) ;testb-1-data
_version_shadow:
        .byte ___str_0, (___str_0 >> 8)

        .area SEGC       (CODE) ;testb-2-code
```

```
___str_0:
        .db 0x56 ; 'V'
        .db 0x45 ; 'E'
        .db 0x52 ; 'R'
        .db 0x31 ; '1'
        .db 0x2e ; '.'
        .db 0x30 ; '0'
        .db 0x00 ; '.'
;--------------------------------------------------------
; initialized data - mirror
;--------------------------------------------------------

        .area IDATA       (DATA) ;testb-0-data
_version:
        .ds    2
```

上面的 ASSEMBLY 表示 “version” 是一个在 RAM 里的指标, 但它指到 ROM, ROM 的地址是 __str_0, 在 RESET 的时候软件要从 _version_shadow 的地址把 “__str_0” 的地址给 COPY 到 _version 在 RAM 的地址。

另一个写法如下：

```
__code char * __code version="VER1.0";
```

编译产生的 ASSEMBLY 如下：

```
        .area SEGC       (CODE) ;testb-0-code

_version:
        .byte ___str_0, (___str_0 >> 8)

        .area SEGC       (CODE) ;testb-1-code

___str_0:
        .db 0x56 ; 'V'
        .db 0x45 ; 'E'
        .db 0x52 ; 'R'
        .db 0x31 ; '1'
        .db 0x2e ; '.'
        .db 0x30 ; '0'
        .db 0x00 ; '.'
```

可以看出这是在 ROM 里直接放__STR_0 的地址，RESET 后不需要再 COPY，也不会占用 RAM 的空间。

另外一个范例是 “function pointer”，或它的 ARRAY。

```
void foo(char a)
{
    PT2+=a;
}
void bar(char b)
{
    PT2-=b;
}
void dummy(char a)
{
    return;
}
typedef void (*FUNC)(char); // this is the function pointer type!!

const FUNC keyf[4] ={dummy,foo,bar,foo};

main()
{
    while(1)
    {
        keyf[PT1&3](PT1);
    }
}
```

上面的范例表示 有 function-pointer (FUNC) 的 array “keyf” 在 ROM 里面，初始值为 dummy,foo,bar,foo。在 ”keyf” 的 ASSEMBLY CODE 为

```
    .area SEGC      (CODE) ;testb-0-code
_keyf:
    .byte _dummy, (_dummy >> 8)
    .byte _foo, (_foo >> 8)
    .byte _bar, (_bar >> 8)
    .byte _foo, (_foo >> 8)
```

4.2.3 const 关键字用法

在 ANSI C 里的”const” 后面的 “specifier”, 如果是 pointer，表示 point 到的内容不可以修改的意思。但因为 CPU 架构不同，目前”const” 是表示数据在 ROM 区域，而不是在 RAM 的区域。为了避免混淆，建议可以使用 “__code” 与 “__data” 来指定 ROM 或 RAM 的 POINTER。

4.2.4 硬体指针

HYCON C Compiler 使用 x86 相同的 "little-endian"。虽然硬件上有的 SFR 是以"big-endian" 排列，但所有的 compiler 运算都是使用 little-endian。

因为硬件的 ENDIAN 不同，直接使用硬件指针要经由特殊的变量 FSR0/FSR1/FSR2, 范例如下：

```
FSR1=(unsigned char*)&aa[4];
SPBUF=POINC1;
SPBUF=POINC1;
```

因为目前 COMPILER 没有使用 FSR1，所以使用者可以将 FSR1 用来做全局 FIFO 等功能。如果需要使用 PLUSWX 缓存器, 可能要配合 inline-assembly。

COMPILER 本身会用 FSR0L/FSR0H 去读取或写入数组或指针地址，所以 FSR0 原则上留给 COMPILER 使用。

4.2.5 本地变量与参数传递

对 HY15P 系列产品在有递归调用(RECURSIVE)，或 VARIABLE PARAMETER LENGTH，或 INTERRUPT/MAIN 同时调用同一函数时，自动使用 FSR2 做为 STACK POINTER，并将 LOCAL VAR 及 PARAMETER 放在 STACK 里，其他的函数则使用 LINKING-TIME LOCAL VAR ALLOCATION。

也就是说，除了 HY15P 系列以外的产品，HYCON C Compiler 目前只能使用 Linking-Time Local Variable Allocation/reuse By Call-Tree，也就是在 Linker 时候，根据各个函数的调用决定每个函数里的本地变量地址。而 parameter 也是 local-variable 的一部份，所以目前没有支持 re-entry & recursive functions。如果 interrupt 和 main thread 调用同一个函数自然也会有问题。

但目前做法的好处是自动 reuse local/parameter variables，还是可以省下大量的 RAM，这是使用 ASSEMBLY 工具很困难才做得到的功能。

另外就是除了 HY15P 系列产品以外，也不支持 "stdarg.h" 里的可变长度 parameter, 像是 printf(...)。还有是对于 function pointer 所指的 function 所用的 parameter 只能有 1 个 byte, 使用 WREG 去传递, 而其 local variables 会固定占去 RAM 的地址，因为不能在 linking time 建 call tree 去分配。

对于 HY15P 系列或更新的产品即没有以上的限制。

4.2.6 乘除法运算

乘法运算

C89/C99 规定 8-bit 的乘法结果是 8-bit，而 16-bit 的结果是 16-bit。但为了一般 GENERAL 的应用，此 COMPILER 对 unsigned 8-bit * unsigned 8-bit 可以直接把 16-bit 积写入 16-bit 的变数里。但其它的有符号数的乘积会符合 C89/C99 的规定。也就是有以下乘法的结果列表：

乘数型态	乘数值	被乘数型态	被乘数值	积型态	积的值	说明
unsigned char	10	unsigned char	10	signed/unsigned **char**	100	10*10=100
unsigned char	100	unsigned char	100	signed/unsigned **char**	0x10	100*100=0x2710, 但结果只有 8 bit.
signed char	-10	signed char	10	signed char	-100 (0x9c)	8 bit signed OK.
unsigned char	100	unsigned char	100	signed/unsigned short	**10000 (0x2710)**	this is **exception**, but correct
signed char	100	signed char	100	signed/unsigned short	**0x0010**	8bit * 8bit = 8 bit, **overflowed**
signed char	-10	signed char	10	signed short	-100 (0xff9c)	8bit * 8bit = 8 bit, signed
signed char	-100	signed char	10	signed short	**0x0018**	**overflowed** (-1000 = 0xfc18)

如果乘数或被乘数有 1 者为有符号(signed)，则此乘法会被视为有符号(signed)运算。如果运算有溢出的可能，则乘数或被乘数得先转换成更大的数再进行运算。

比如：

```
signed char a,b;
signed short c;
…
c = ((signed short)a)*b; //cast to prevent overflow
```

除法运算

目前 HYCON 8-bit C COMPILER 的除法与余数(MOD)在正数的结果不会有什么争议，但在有负数的时候是符合 C99/C89 的规范如下：

表达式	结果
-50/25	-2
-50/-12	4
-50/4	-12

100%17	15
100%(-7)	2
100%(-8)	4
(-49)%3	-1
(-49)%(-5)	-4
(-49)%4	-1

4.2.7 内联函数（inline function）

HYCON C COMPILER 对于 “static ” 声明的函数，如果只被呼叫一次，会直接以”inline” 的方式加入，以节省空间。

再次需要注意，诸多在 “ctype.h” 所定义的函数，多已宣告成”INLINE”，所以不在 library 里面。

也有少数指令没法用 C CODE 可以达成的，就必需要使用 inline-assembly. 方式如下:

```
#define __NOP__ __ASM NOP __ENDASM
    while(1)
    {
    __NOP__; // defined as MACRO is OK

    __asm
    CWDT
    SLP
    NOP
    __endasm;
    ...
```

4.2.8 中断(INTERRUPT)函数宣告

HYCON 8-BIT MCU 只有一个中断地址，所以只能有一个中断矢量函数，使用 "__interrupt" 关键字，声明方式如下：

```
void timer_routine(void) __interrupt
{...}
```

如果在中断函数里有用到指针或 "__xdata" 区域的变量，必须"手动" 保存 FSR0，像以下的程序:

```
void timer_routine(void) __interrupt
{
    unsigned char *fsr0bk=FSR0;
      ....
      INTF1&=(0xf7);
      FSR0=fsr0bk;
```

```
    }
```

以上的程序会在进入中断矢量函数时保存 FSR0L/FSR0H 至 "fsr0bk"，在离开时将之还原，使主程序的 FSR0L/FSR0H 保持不变。

4.3 HYCON C Compiler 项目开发

4.3.1 使用 C Compiler 开发项目流程

使用 HYCON C COMPILER 产生机器码与其他的 COMPILER 无异，皆是产生 ASSBMELY，再转成 OBJ，再与 HYCON 提供的 LIB 结合，LINK 成目标的机器码。

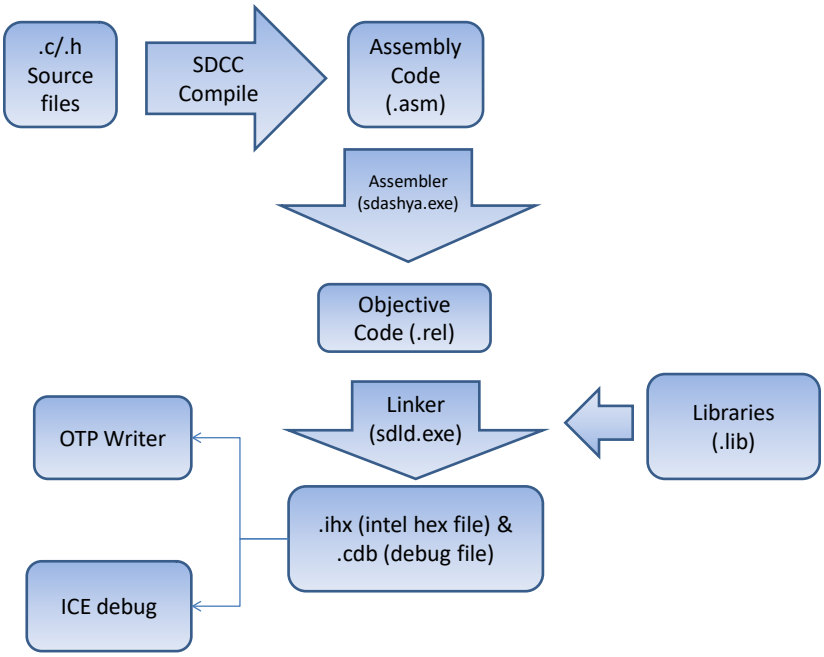


图 4-3-1 使用 C COMPILER 开发项目的流程

4.3.2 检查空间大小

COMPILER 工具在链接(LINK) 的时候会进行 ROM / RAM 大小的检查。检查的方式是使用各 LIBRARY 在 “hyXXXXsfr” 模块里有”_SWTGTID” 的标记。比如在程序里使用了#include <hy11p14sfr.h>，该头文件里有 “_SWTGTID”的定义，会传递到 LINKER 设定其相关的 ROM/RAM 空间大小。如果有多个 OBJ 都有该 “_SWTGTID”，则使用 main 的 OBJ 里的“_SWTGTID”。

如果该部分的 ROM 或 RAM 不够大，链接 LINK 的程序就会以失败退出。

如果所有的程序都没有定义“_SWTGTID”，则 LINK 就无法知道该母体的内存大小，也就会跳过空间大小的检查。

另一个限制是对于 HY15P41 等更新的母体(TARGET/PART)，有可能因为 variable length argument 或 recursive 使用 FSR2 做为堆栈的本地空间，则 FSR2 OVERFLOW 的问题不会在 LINK 的时候发现。如果程序使用 FSR2 的堆栈空间，则使用 ICE 测试是否 STACK OVERFLOW 是一个必要的步骤。

4.3.3 C 程序代码启动流程

产生的程序代码会调用‘__gsinit128/256/512’去做初始化再跳转到 "main"。初始化的动作包括：

1） 把 RAM 初始化为 0

2） 载入初始的 __data RAM(或称 bss)

3） 载入初始的 __xdata RAM(或称 xbss)

若 RAM SIZE 不同，则启动程序就会不同，不能混用。

相关的 GSINIT 及其 LIB 源码在 sdcc\lib\HY08A 可以找到，必要时使用者可以修改之。比如在 IC 发生复位动作的时候，有的产品需要保留部份 RAM 的值不要清除，则可以将 sdcc\lib\hy08a\gsinitxxx.asm 复制到软件的项目目录，修改后再链接（Link）。虽然在 LIBRARY 里有同名 EXPORT SYMBOL 的 MODULE，但如果 OBJ 里有提供 LINK 所需要的 EXPORT SYMBOL 的 MODULE，LINER 就不会再到 LIB 里寻找。

以下的命令会将目录下修改过的 gsinit512.asm 给编译成 gsinit512.rel 之后，再与 main.rel 进行链接（Link），产生 “main.ihx” 与 “main.cdb”，

```
sdashya –l –s –o –y gsinit512.asm
sdld –u –m –i –y main main gsinit512 –l hy11psdcc.lib
```

4.3.4 C Compiler 编译相关档案

一个 C 程序，编译成机器码会产生很多其他的文件档案，或者会参考其他的文件档案,在此做一个说明。

文件类型	延伸名	产生者	说明
程序源代码	.c	使用者	C 源代码
头文件	.h	使用者或其他	多是 extern 或 function 的 prototype 宣告.
汇编代码 (Assembly code)	.asm	compiler(sdcc) 或使用者	Compiler 主要是把 C CODE 转成 Assembly Code. 使用者也可以自己 CODEING ASSEMBLY CODE.
目标代码 (Objective code)	.rel	Assembler	使用者可以经 sdcc.exe 叫用 sdashya.exe 产生 OBJ code，或使用 MAKEFILE/BAT 直接叫用 sdashya.exe 将 .asm 转为 objective code.
列表文档 LIST FILE	.lst	Assembler	Assembler (sdashya.exe) 使用 "-l" 会产生 ".lst" 文档，是比较好读的 assembly file，内列大约的 code size, etc.
再定位列表文档 related list file	.rst	Linker	此 FILE 有每个 ASSEMBLY FILE 最终在 机器码内的信息，包括大小，内容，和地址.
符号文档	.sym	Assembler	Assembler 会列出该 ASSEMBLY 所使用的 symbol.

Symbol File			
地图文档 map file	.map	Linker	Linker 最后会产生 MAP file，列出每个 obj 里的 SYMBOL/位置，及每个"area" (segment) 的 信息，包括地址和大小
C 除错文档(CDB)	.cdb	Linker	CDB 是给 ICE 读取的文档，内有除错所需的相关信息.
INTEL 16 进位文档 (Intel Hex file)	.ihx	Linker	Linker 最后产生的机器码. 目前是用 IHX 格式，使用者可以用 hex2bin.exe 产生二进制的 ROM IMAGE.

图表 4-3-1 Compiler 相关的文件类型

4.3.5 C Compiler 编译命令及命令选项

如果用户不使用 C IDE 编译程序而使用命令行，要先设定环境变量：

1） 环境变量(Environment Variable) “SDCC_HOME” 设定到 sdcc 所解压的目录.

2） 环境变量 PATH 要新增解压的 sdcc\bin

可以参考解压后在 sdcc 目录的 "env.bat"

如果不用"BAT" 设定请参考

https://support.microsoft.com/zh-tw/kb/310519

http://www.computerhope.com/issues/ch000549.htm

常用的命令行选项如下表 4-3-2:

选项	说明
-S	只 compile 成 assembly code.
-c	compile + Assemble 成 OBJ code (.rel)
-D	preprocessor additionally defined macro.
--no-peep	不进行 peephole optimization；基本上这 option 会禁止大多数的优化动作；如果 compile 出来的 CODE 与期待的不同，可以用这个 option 看未优化的结果
-pHYXXXXX	指定 part/target number
-Wl-nx	在 linker 叫用的时候加 –nx，表示不把 ‘ BTSS STATUS,N’ 优化成 JXX 指令
-v	显示 COMPILER 的版本

表 4-3-2 常用的命令行选项

4.3.6 OBJ(.REL)路径

如果要将 OUTPUT 与 OBJ 放到不同的目录，并支持 ASSEMBLY FILE，则 MAKEFILE 要改成如下：

```
C_SRC = main.c foo.c
ASM_SRC =
SRCDIRS =.
PRJ = main
TARGET = HY11P14
```

```
OBJDIR = obj
OUTDIR = bin
AOPT1 =
COPT1 =
LOPT1 =
#following no change
LIBS1=hy11psdcc.lib
ASMBASE=$(basename $(notdir $(ASM_SRC)))
CBASE=$(basename $(notdir $(C_SRC)))
COBJS=$(addprefix $(OBJDIR)/, $(addsuffix .rel,$(CBASE)))
ASMOBJS=$(addprefix $(OBJDIR)/, $(addsuffix .rel,$(ASMBASE)))

OBJS= $(COBJS) $(ASMOBJS)
COPT = -p$(TARGET)

.PHONY: clean
ALL:$(OUTDIR)/$(PRJ).ihx $(OBJS)

#only C obj has .d

-include $(OBJS:.rel=.d)

VPATH=$(SRCDIRS)

$(OUTDIR)/%.ihx: $(OBJS)
    sdld -u -m -i -y $(LOPT1) $(OUTDIR)/$(PRJ) $^ -l $(LIBS1)
    hex2bin $@

$(OBJDIR)/%.rel: %.asm
    sdashya -l -s -y $(AOPT1) -dep $(basename $@).d -o $@ $<

$(OBJDIR)/%.rel: %.c
    sdcc $(COPT) $(COPT1) -c $< -o $@
    sdcc $(COPT) $(COPT1) -E -Wp-MT -Wp$(basename $@).rel -MM $< >
$(basename $@).d

clean:
del /Q $(OBJDIR)\*.d $(OBJDIR)\*.rel $(OBJDIR)\*.asm $(OBJDIR)\*.lst $(OBJDIR)\*.sym
$(OUTDIR)\$(PRJ).ihx $(OUTDIR)\$(PRJ).bin
```

以上的 MAKEFILE 也支持”main.c” 变成 “src\main.c”，在不同目录下。以上的 MAKEFILE 也支持独立的 ASSEMBLY FILE。

4.3.7 HYCON C 函数库

方便用户针对硬件的设置，我们定义对应的 C 函数，用户只需将所用到函数的头文件包含进程序，就可以在程序里调用函数。对应的 C 函数库的用法及函数功能定义说明，请参考 HYCON C 函数库用户手册。函数库头文件在安装目录下 driver 文件夹。

比如用户要设定使用的 MCU 工作频率及指令周期，调用 HYCON C 函数库操作如下：

```
#include "..\..\Driver\HY11\CLK.h"              //频率函数的头文件路径
CLK_CPUCKOpen(MCKCN2_CPUCK_HSDCK,MCKCN1_ENXT_DISABLE,MCKCN2_HSS_HSCKDIV1,MCKCN1_XTSP_XTL,MCKCN2_HSCK_HAO);
```

4.4 C Compiler 使用范例

4.4.1 单一 C 文件范例

如果源代码只有一个 C 文檔(包含”main()”), 则 MAKEFILE 只有 3 行即可

```
ALL: demo.ihx
demo.ihx: demo.c
sdcc –pHY11P14 demo.c
```

如果用 BAT 档, 自然就是一行

```
sdcc –pHY11P14 demo.c
```

一个最简单的 “demo.c” 如下:

```
include<hy11p14sfr.h>
main()
{
    TRISC2=0xff; // PT2 output
    while(1)
        PT2^=1;
}
```

在 MAKE/BAT 后会产生以下文档:

文档名称	说明	NOTE
demo.asm	绘编(Assembly) 文檔。 此文件为 COMPILE 出的 ASSEMBLY CODE.	由 SDCC.exe 产生.
demo.rel	物件文文件(OBJECT FILE)。此文件为经	Sdcc.exe 自动叫用 sdashya.exe 产

	过 ASSEMBLER 产生的 OBJ CODE.	生.
demo.sym	SYMBOL LIST FILE. 此文件为 ASSEMBLER 产生的 SYMBOL 文件。	同上。
demo.ihx	Intel Hex File. 此文件为 Linker 产生的机器码，使用 INTEL 的 16 进位格式。	SDCC.EXE 叫用 sdld.exe 产生. 工具另外附有 “hex2bin.exe” 可以转换 intel hex 格式至二进制机器码。
demo.map	地图文件，上面记载每个节区的地址及相关本地变量的数据。	SDCC.EXE 叫用 sdld.exe 产生.
demo.cdb	C Debug File. 此文件为 DEBUG 所用之文件，由 LINKER 产生。	同上。

表 4-4-1. COMPILER 产生的文档列表

4.4.2 多个 C 文件范例

使用高级语言的好处就是可以重复使用，要使用之前写过的程序代码，就可以把要用的程序给 COMPILE 成 OBJ 后再 LINK 起来。

比如一个 C 文档”FOO.C” 如下：

```
#include <hy11p14sfr.h>
int k=0x56;
int foo(int i, int j)
{
    if(j==2)
        return i*2;
    return i/4;
}
```

另一个文档”main.c”包含”main()”，如下：

```
#include <hy11p14sfr.h>
extern int k;
extern int foo(int i, int j);
main()
{
    TRISC2=0xff;
    while(1)
    {
        if(foo(k,PT2))
            k=PT2;
    }
}
```

则完整的 MAKEFILE 如下：

```
all: test1.bin
test1.bin: test1.ihx test1.asm
	hex2bin test1.ihx

test1.ihx: main.rel foo.rel
	sdld -u -m -i -y test1 main foo -l hy11psdcc.lib

%.rel: %.c
	sdcc -c -pHY11P14 $<

clean:
	del *.asm
del *.rel
```

4.5 预定义巨集

PRE-DEFINED MACROS 如下表，可以使用 sdcc –V 显示. 说明如下表.

宏(Macro)	值(Value)	说明
__SDCC_PROCESSOR	COMMAND LINE 的 -p	PROCESSOR 名称.
__SDCC_CHAR_UNSIGNED		默认 char 是 unsigned. 可以用 “—fsigned-char”更改.
__SDCC	3_6_0	SDCC 版本
SDCC	360	SDCC 版本
__SDCC_HY08A		表示是给 HYCON 8-BIT CPU 使用的 COMPILER.
__STDC_NO_THREADS__	1	不支持 threads.
__STDC_NO_ATOMICS__	1	不支持 atomic operations.
__STDC_NO_VLA__	1	不支持 variable length array.
__STDC_ISO_10646__	201409L	表示(有限)支持 wchar_t

4.6 汇编编译器

HYCON C COMPILER 附有 ASSEMBLER 和 LINKER。本文件对该 ASSEMBLER 和 LINKER 做进一步使用说明，若使用者原来即熟悉 H08 的指令，则 ASSEMBLER/LINKER 可用来编写更特殊的程序和应用。

4.6.1 Assembler Linker 使用流程

COMPILER TOOLSET 最终产生的是 INTEL HEX FORMAT (.IHX) 的 ROM IMAGE，配合产生的 CDB (C DEBUG FILE)，可以用在 ICE DEBUG。若 ICE 试起来没有问题，则可以把".IHX" 档案烧到 TARGET IC 上面开始试量产。".IHX" 档案产生的流程如下：

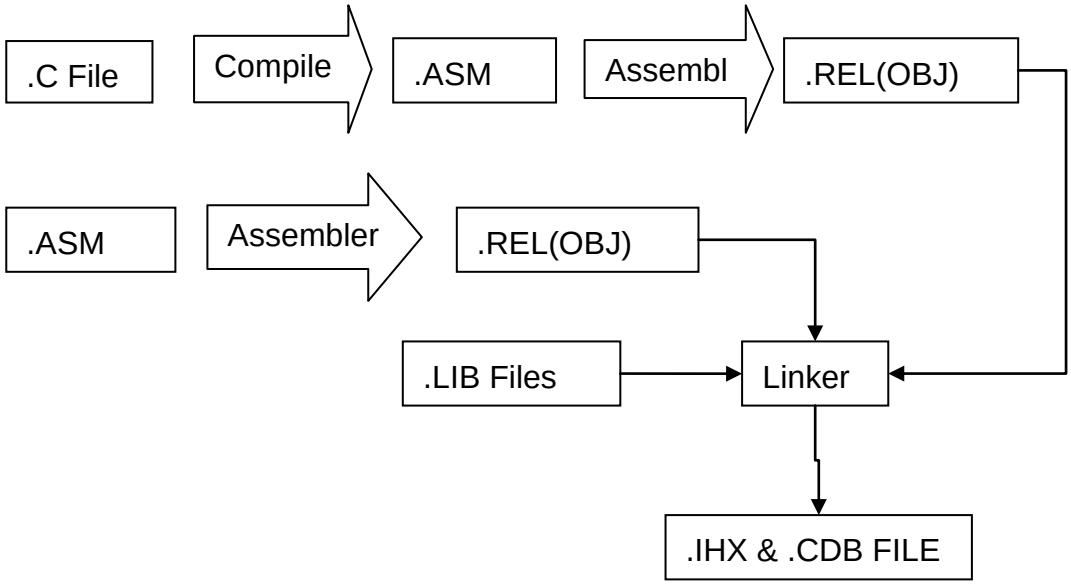


图 4-6-1 相关文文件产生流程

由以上上 4-6-1 图所示，COMPILER (SDCC)可以产生 ASSEMBLY，再由 ASSEMBLER 将 ASSEMBLY 转成 OBJ 档(.REL)，最后 LINKER 将所有的 OBJ 档与 LIB (LIBRARY) 一起 LINK 成 .IHX 和 .CDB 档案。

HYCON 所提供的 LINKER 会进行 BRANCH 的优化，包括：

1） JMP/CALL 指令自动转换成 RJ/RCALL 指令，如果跳跃的地址在 +1023~-1024 之内。

2） BSTSS/BSTSC _STATUS，.. + RJ 会自动转换成 JNC/JC/JZ/JNZ ..，如果跳跃的地址在 +127~-128 之内。

3） 对于 HY15P 系列产品，会对 RECURSIVE 及在 INTERRUPT & MAIN 都使用的 FUNCTION 的本地变数改置于 FSR2 的 STACK(堆栈) 区间。

4.6.2 Linker 使用

LINKER (sdld.exe) 一般由 IDE 或 MAKE.EXE 经 MAKEFILE 调用，使用者也可以在命令行里直接调用。其格式如下:

sdld [options] <target name> <obj name> [<obj name>...] [-l library] [-l library].

例如:

```
sdld -u -m -i -y target main foo -l hy11psdcc.lib
```

会读取 "main.rel"、"foo.rel"与 hy11psdcc.lib 给 LINK 起来，产生 "target.ihx"、"target.cdb"、"target.map"，并更新 list file 写入"main.rst" 和 "foo.rst"。其 sdld 相关的参数选项说明如下表 4-6-1：

Option	说明
-u	此选项会更新 LST 文件成位 RST 檔. RST 文档里有最后 LINK 出的 ADDRESS 和 CODE.
-m	产生 .map 档案. .map 档案里可以查到所有 SYMBOL 的地址。
-i	产生 intel hex format output. (.ihx)
-o	产生 motorolla hex format output. (.mot)
-y	产生 ".cdb" debug info file 给 ICE 用.
-l XXX.lib	以 XXX.lib 当成 LIBRARY 进行 LINK. XXX.lib 会在本地目录, $SDCC_HOME/lib/HY08A 或其它指定的 PATH (-k) 底下找寻。
-v	verbose, 会显示 Linker 的步骤。
-V	Version, 会显示 LINKER 的版本。
-k ppp	会增加 ppp 到 search path 里.
-no	不优化 JMP/CALL 到 RJ/RCALL
-nx	不优化 BTSZ/BTSS _STATUS,.. + RJ 转换成 JNC/JC/JZ/JNZ
-nr	不优化未叫用的 FUNCTION. 默认是移除未叫用的 FUNCTION.
-h08b	H08B 指令集, 等于 -no -nx.
-k	增加 LIB 的寻找目录。

表 4-6-1

4.6.3 Librarian 命令列使用

COMPILER 工具提供 LIBRARY TOOL，可以把 OBJ FILE (.REL) 打包成 LIB。在 LINK 时如果 OBJ 没有提供 EXPORT(.globl) 的 SYMBOL，则 LINKER 会去 LIBRARY 里找寻提供该 SYMBOL 的"MODULE" 并将它的内容加到 LINK 的结果里。而 LIBRARIAN (sdcclib.exe) 即是将相关的 code 给打包成 LIB 的工具。其命令行的格式如下：

sdcclib.exe [options] <lib name> <obj name> <obj name> …

比如:

```
sdcclib -a lib1.lib tool1.rel tool2.rel
```

会将 2 个 OBJ FILE "tool1.rel"与 "tool2.rel" 给打包成 lib1.lib 。未来在 LINK 的时候只要加加 –l lib1.lib 即可使用 tool1/tool2 所提供的 FUNCTION 等定义。

其 Librarian 命令相关的参数选项说明如下表 4-6-2：

Option	说明
-a 或 -r	加入或取代在 LIB 里的 OBJ FILE MODULE
-d	移除 LIB 里的 MODULE

-x 或 -e	将 OBJ 从 LIB 里取出
-s	列出 LIB 所 EXPORT 的 SYMBOL
-m	列出 LIB 里的所有 MODULE NAME
-v	显示版本

表4-6-2

4.6.4 Assembler 命令列使用

Assembler (sdashya.exe) 一般由 IDE 或 MAKE.EXE 经 MAKEFILE 调用，使用者也可以在命令行里直接掉用。其格式如下：

sdashya [options] xxx.asm

sdashya 命令相关参数选项说明如下表 4-6-3：

Option	说明
-l	产生 .LST 档案
-s	产生 .SYM 档案, 会列出 assembly 里定义的 Area 和 symbol
-o	产生 .rel (OBJ) 档案.
-y	在 .rel 里加入 debug infomation.
-d	dump MNE, debug only.
-z	将所有的 symbol 产生 .rel 时都变成大写.
-k path	把 path 加到 include 的 search path 里.
-m	compatible mode, 与旧版的 assembler 相容, 在 MVL/XORL/IORL/ANDL/ADDL/SUBL label 的指令上, 如果 label 是在 CODE 上, 会以 word 计算 label 地址, 而不是 byte.

表4-6-3

4.6.5 Assembly 大小写区分

Assembly 里的关键字如 module/area/define/db/dw/ascii/ifdef 是不区分大小写的，但 LABEL/SYMBOL 是默认有大小写区分，这是为了 C COMPILER 的需要。因为 ANSI C 本身就是有区分大小写的。如果为了一些兼容性的需求，需要 LABEL/SYMBOL 不区分大小写，需要在 sdashya.exe 后面加 "-z"，该选项会把所有的 SYMBOL/LABEL 变成大写给写入 .REL (OBJ) 文档，在 LINK 时，自然就是用全大写的 SYMBOL 去 LINK。加了"-z" 的主要目的是在 Assemble 的时候，会检查是不是都变成大写后有重复定义的 SYMBOL。比如说，

```
ABC = 10;
...
Abc = 20;
..
```

在 assemble 时如果没有 "-z" 的选项, 是正确的写法，因为 ABC/Abc 算是不同的 symbol。但如果加了"-z"选项，ABC/Abc 就算是同一个 SYMBOL 被重复定义了。于是 ASSEMBLE 就会有 ERROR 中止程序。

默认 C COMPILER 呼叫 ASSEMBLER 的语法如下:

```
sdashya –l –s –o –y XXX.asm
```

4.6.6 Assembly 区域定义

此 Assembler 最后产生的 .IHX 档案里会有数个区块, 由数个源码文件所定义. 区块的单位如下表所说明:

区块单位	说明
FUNCTION	C 语言的 FUNCTION, 会定义所 CALL 的 FUNCTION 以及本地变量 (Local Variable) 名称. Linker 会依此做 Local Variable Reuse. Assembly 里用 ".FUNC" 定义 function.
MODULE	一整个 Assembly 产生的 REL 文档算是同一个 Module. 这 Module 算是 Library 的最小单位. 也就是说, 如果从 LIBRARY 里 LINK 一个 FUNCTION 进来, 会把含有 FUNCTION 的整个 Module 包进来, 而不是只有包 function. Assembly 文档里面使用 .module 来定义 MODULE.
AREA	别的 TOOLCHAIN/COMPILER 可能称为 SEGMENT. 是放在一起的区块单位. 像是 所有的 module 可能都有 "CODE" AREA, 最后 Linker 会把所有 Module 里面同名字的"AREA" 放在一起. 当然, RAM/ROM 的数据不能放在同一个 AREA. 使用 AREA 的目的是为了 Linking Time Memory Allocation. Assembly 文档里面使用 .area 来定义 AREA.
.ORG	如果是 ABS 的 AREA, 可以用 ORG 来定义它的 ADDRESS.

.FUNC/.ENDFUNC 语法

.FUNC/.ENDFUNC 语法如下：

.FUNC <function name>:$C:callfunc1:$C:callfunc2:...:$L:localvar1:$L:localvar2 ...

.ENDFUNC <function name>

表示该 FUNCTION 所叫用的其他 FUNCION，还有它所用的本地变数。在 LINK 时，本地变量会建 CALL TREE 去共享所有 FUNCTION 的 LOCAL VARIABLE，所以可以省下很多的空间。比如：

```
.FUNC _foo:$C:__mulint:$C:__divsint:$C:__modsint\
:$L:_foo_STK00:$L:_foo_STK01:$L:_foo_STK02:$L:_foo_STK03:$L:r0x117A\
:$L:r0x117C:$L:r0x117B:$L:r0x117D:$L:r0x117E
```

表示 foo() 会调用_mulint()、_modsint()、_divsint()，而 _foo_STK00 ... 等为它的本地变数。

ASSEMBLY 里面有.FUNC 的宣告是为了给 LINKER 建 CALL TREE 建 CALL TREE 可以去分配/共享 LOCAL VARIABLE 的空间。比如 main()有 call foo() 和 bar() 而 foo()/bar()都没有 call 别的 function，则 foo(), bar()的 local variable 是可以共享同一块 RAM 的。如果 foo()有 call 了 bar(), 或 bar()有 call 了 foo()，则它们

的 local variable 的空间就要分开，不能放在一起。

C 语言里面建在 STACK 上的 LOCAL VARIABLE 做法，可以自动节省很多的 RAM，这是以往纯 ASSEMBLY 的 CODING 很难做到的地方。

.FUNC 宣告的另一个目的是给 ICE 识别，使用者可以直接跳到该 FUNCTION 上面设定断点，在比较大的 PROJECT 上面会是比较方便的。

.MODULE 用法

.MODULE 的用法如下:

.MODULE <MODULE NAME>

即指定 MODULE 的名称。一个 ASSEMBLY 文档只能指定 1 个 MODULE NAME。

.AREA 用法

".AREA" 的用法如下：

.AREA <area name> (region, REL/ABS, CON/OVL)

其中 "area name" 是指定的名称，region 可以是 CODE/DATA/XDATA，CODE 表示在 ROM 的区域, DATA 表示在 RAM ADDRESS 0~0x1FF，而 XDATA 表示在 RAM 0x200 之后。

REL 这个 AREA 是跟其他的 AREA 一起放在 RAM 或 ROM，地址会跟着改变，所以叫 REL (RELATIVE)。ABS 表示有这个 AREA 有自己的 ADDRESS，其内部可用 .ORG 的方式直接指定它的起始地址。

CON 表示每个 MODULE 里的这个 AREA 是接在一起放 (CONCATENATE)，OVL 表示每个 MODULE 里的这 AREA 是重迭的(OVERLAPPED)。

.ORG 的用法

一般形式如下：

.ORG <byte offset>

要注意即使在 ROM 区域, ORG 也是以 BYTE 为单位. 比如

```
.ORG 0x100
```

会从第 256 BYTE (WORD COUNT 128)开始。

4.7 Assembly 指令

MCU 支持的指令可参考以下文件，但有些特殊指令以下会做补充说明。

http://www.hycontek.com/wp-content/uploads/APD-CORE002_TC.pdf

4.7.1 指令含有表达式

所有 CPU 指令用到的变量/变数/地址多可以用表示式 (Expression) 的方式写在 ASSEMBLY 里面。使用 CONDITIONAL ASSEMBLY 亦同，只是在 conditional assemble 时，表示式在 assemble 时必需可以求得表示的常数，而用于 CPU 指令里的表示式只要在 LINK 时可以解得最后的值即可。比如

```
MVF var1+3,1,0
```

所用的 var1 要在 link 时才 ALLOCATE 其地址，所以指令的内容会在 LINK 时才决定。

4.7.2 表达式里字符含义

C 语的 EXPRESSION 里的变量是指变量的"内容"，而 ASSEMBLY 的 EXPRESSION 里的 SYMBOL/LABEL 则是参考它们的"地址"，这是基本的不同。比如

```
A=B+1
```

在 C 里面是指 A 内容会被 ASSIGN 成 B 的内容加 1;而在 ASSEMBLY 里面这表示 A 地址是 B 地址加 1。这是使用者要小心的地方。

另外这里要补充的是 LABEL 地址的计算。跟所有 ANSI C 所用的 COMPILER 一样, LABEL 计算要等同 POINTER 的运算，所以一定是以 BYTE 为单位。但因为某些旧版的 ASSEMBLER 对 ROM 的 LABEL 使用 WORD 为单位，所以 sdashya.exe 有 -c 的选项，使用之后对 ASSEMBLY 文档里 "literal Addressing mode" 指令所用的 在 ROM 区域的 label 会改以 WORD 为单位来计算 Expression。

4.7.3 表达式的运算符

表示式里可含有一些运算符，等同 ANSI C 的运算。表达式可用操作数运算符如下表 4-7-1：

操作数	说明	范例
+	加法	MVF VAR+OFFSET,1,0
-	减法	MVF VAR-OFFSET,1,0
HIGH	就是>>8, 取高位	MVL HIGH(LABEL1)
LOW	就是 &0xff, 取低位	MVL LOW(LABEL1)
D2	就是>>1, 主要是将 BYTE 地址变成 WORD 地址。	MVL D2(LABEL)
HIGHD2	就是>>9, 主要是将 BYTE 地址变成 WORD 地址再去高位。	MVL HIGHD2(LABEL)
>>	与 C 语言的>> 相同.	ADDL (VAR+5)>>1
()	与 C 语言的>> 相同	ADDL (VAR+5)>>1
==, >, <, <=, >=, !=	与 C 语言相同	.iftrue (MODEL==2) ... .endif

表4-7-1

4.7.4 资料存储

ASSEMBLY 文档除了 CPU 的指令之外，同时要存放很多資料。存放資料有以下的方式：

指令	说明	范例
.DB	BYTE BY BYTE 放资料.	.DB 1,2,3
.BYTE	跟.DB 一样	.BYTE 4,5,6
.DW	WORD BY WORD 放资料.注意，因 COMPILER 是 LITTLE ENDIAN，所以.DW 3 会等同于 .DB 3,0 也就是说，在 LST/RST 里会看到的都是 0300 而不是 0003.	.DW 3
.DS	DATA SPACE，保留空间. 一般在 RAM 里都使用 .DS 比如 KK: .DS 2 表示在 KK 的地址有 2 BYTE 的空。	KK: .DS 4
.ASCII	ASCII String. 可以放 ASCII 的字符串，最后有 NULL TERMINATOR.('\0')	str1: .ascii "This is a book."

4.7.5 预处理命令

Assembler提供一些预处理命令：.INCLUDE / .DEFINE/ .IFDEF/ .IFNDEF/ .IFTRUE/ .IFFALSE / .MACRO/ .REPT。且‘.IFDEF/ .IFNDEF/ .IFTRUE/ .IFFALSE/ .MACRO/ .REPT’是可以混合使用及多层嵌套使用；

.INCLUDE

‘.INCLUDE’可以将所用指定的文档包括到程序并链接。如‘.INCLUDE “common.inc” ’。

‘.INCLUDE’可以多层嵌套，比如a.asm里有‘.include “b.asm” ’，而b.asm里可以有‘.include “c.asm” ’。但是若b.asm里又include “a.asm”时就会造成错误。

若“.include”的文档不在当前目录下，则需要在命令列添加命令‘-k<path>’，把文档所在目录加上。

.DEFINE

‘.DEFINE’如同C语言的#define功能，其用法如下：

```
.DEFINE MODELN 1001

... MVL LOW MODELN
    MVF ...
..
.DEFINE FEATURE0_EN

.ifdef FEATURE0_EN
```

```
        MVL 0
        .else
        MVL 1
        .endif

        .DEFINE CLRW    MVL 0
    ...
        CLRW
```

.IFDEF/.IFNDEF

‘.IFDEF/.IFNDEF’如同C语言的#ifdef/#ifndef，其用法如下：

```
.ifdef USE_MODEL_A
  LCDDAT0 = 0x55
.else
  LCDDAT0=0x71
.endif
```

以上的写法改用‘.IFNDEF’实现，如下：

```
.ifndef USE_MODEL_A
  LCDDAT0 = 0x71
.else
  LCDDAT0=0x55
.endif
```

如果没有需要，也可省略‘.else’，直接用‘.endif’结束，写法如下：

```
.ifdef ICE_DEBUG
  call SHOW_ERR
.endif
```

需要注意的是，‘.IFDEF/.IFNDEF’后面的parameter只能由‘.define’进行宣告，不能由‘.EQU/LABEL’来宣告。

.IFTRUE(.IF)/.IFFALSE

‘.IFTRUE/.IFFALSE’的后面跟随的是一个‘表达式’，也可以用‘.EQU’配合一起使用，‘.IFTRUE/.IFFALSE’用法如下：

```
  MODEL_NUMBER   EQU   1003
...
  .IFTRUE (MODEL_NUMBER == 1002)
  ...
  .ENDIF
```

```
.IFTRUE (MODEL_NUMBER == 1003)
...
.ELSE
.ENDIF
```

.MACRO

定义 MACRO 以‘.MACRO’开始,‘.ENDM’结束。‘.MACRO’后面接续的是要输入的是 MACRO 名称, PARAMETER 和本地的LABEL，本地的LABEL 宣告时以'?'开头。 一般形式如下：

```
.MACRO <MACRONAME> [PARAMETERS & LOCAL LABELS]
```

PARAMETER 和 LOCAL LABEL 不是必需要写的。比如：

```
.MACRO PULSEN PORT,BIT,NUM,?LOOP
    MVL NUM
LOOP:
    BSF PORT,BIT
    BCF PORT,BIT
    DCSUZ _WREG,1,0
    JMP LOOP
.ENDM
...
    PULSEN _PT2,3,5 ; 在PT2.3 送 5 PULSE
    PULSEN _PT1,1,100; 在PT1.1 送 100 PULSE
```

‘.MACRO’可以与‘.IFDEF/.IFNDEF/.IFTRUE/.IFFALSE’混合使用且可以多层嵌套使用，最大嵌套层数为16层；混合使用如下：

```
.MACRO M1
    .IFDEF ICE_DEBUG
        .REPT 10
        BSF _PT1,1,0
        BCF _PT1,1,0
        .ENDM
    .ELSE
        .REPT 2
        BSF _PT1,1,0
        BCF _PT1,1,0
        .ENDM
    .ENDIF
.ENDM
```

.REPT

‘.REPT’是用来重复一段固定的程序指令，指令可以多行，最后以‘.ENDM’结束。用法如下：

```
.REPT 20
    .ASCII "HELLO"
    .DB 1,2,3
.ENDM
```

则 上面‘.ASCII/.DB’指令会重复20次。

5. 混合语言编程

由HYCON C COMPILER在C语言编程模式下，允许用户合理的插入汇编指令，或者使用内联函数的方式编写汇编程序；

5.1 宏定义方式使用汇编指令

使用#define关键字，以宏定义方式，将汇编指令声明为一个标志符，然后可以在程序中直接使用这个标识符；要插入的汇编指令行是以‘__ASM’开头，再以‘__ENDASM’结束；一般格式如下：

#define 标识符 __ASM ‘汇编指令’__ENDASM

如要定义一个空指令，操作如下：

```
#define __NOP__ __ASM NOP __ENDASM
```

有些指令没法用C CODE达到的，可以使用内联函数（inlne-assembly）的方式达到目的；且许多在‘ctype.h’定义的ANSI C 标准函数也被声明为inline-assembly形式。一般形式如下：

```
__ASM
汇编指令
...
__ENDASM
```

例子操作：

```
#define   __NOP__   __ASM NOP __ENDASM
While(1)
{
      __NOP__ ;
      __ASM
      CWDT
      SLP
      NOP
      __ENDASM ;
}
```

5.2 C 语言链接汇编语言

若用户需要在工程里添加汇编语言文件与 C 程序链接编译，建议新建一个空的 C 文件，编译产生汇编（.asm）文档后，再对 asm 文档进行修改，最后再用 asm 文档进行编译。比如使用“dummy.c”进行修改：

```
// dummy for assembly


int dummy(unsigned char k, unsigned char j)
{
      return k+j;
}
```

在使用以下 COMMAND 可以得到 ASSEMBLY “dummy.asm”

```
sdcc –pHY11P14 –S dummy.c
```

可以得到 “dummy.asm” 如下:

```
;--------------------------------------------------------
; Port for HYCON CPU
;--------------------------------------------------------
;     ;CCFROM:"D:\Work2016\test\test1"
;;    .file "dummy.c"
      .module dummy
      ;.list     p=HY11P14
      ;--cdb--S:G$dummy$0$0({2}DF,SI:S),C,0,0
      ;--cdb--F:dummy:G$dummy$0$0({2}DF,SI:S),C,0,0,0,0,0
;--------------------------------------------------------
; overlayable items in internal ram
;--------------------------------------------------------
;     udata_ovr
.area CODE (code,REL,CON) ; dummy-code
.globl _dummy

;--------------------------------------------------------
      .FUNC _dummy:$L:r0x11ED:$L:_dummy_STK00:$L:r0x11F1:$L:r0x11F2
;--------------------------------------------------------
;     ;.line     4; "dummy.c"     int dummy(unsigned char k, unsigned char j)
_dummy:    ;Function start
      MVF   r0x11ED,1,0
;     ;.line     6; "dummy.c"     return k+j;
      CLRF  r0x11F1,0
      MVF   _dummy_STK00,0,0
      ADDF  r0x11ED,0,0
      MVF   _dummy_STK00,1,0
      MVL   0x00
      ADDC  r0x11F1,0,0
      MVF   r0x11F2,1,0
      MVF   _dummy_STK00,0,0
      MVF   STK00,1,0
      MVF   r0x11F2,0,0
      RET
; exit point of _dummy
      .ENDFUNC   _dummy
;--------------------------------------------------------

      ;--cdb--S:G$dummy$0$0({2}DF,SI:S),C,0,0
      ;--cdb--S:Ldummy.dummy._dummy_j_1_1$j$1$1({1}SC:U),R,0,0,[_dummy_STK00]
      ;--cdb--S:Ldummy.dummy._dummy_k_1_1$k$1$1({1}SC:U),R,0,0,[r0x11ED]
;--------------------------------------------------------
; external declarations
;--------------------------------------------------------
```

```
      .globl WSAVE
      .globl STK06
      .globl STK05
      .globl STK04
      .globl STK03
      .globl STK02
      .globl STK01
      .globl STK00
;--------------------------------------------------------
; global -1 declarations
;--------------------------------------------------------
      .globl     _dummy

;--------------------------------------------------------
; global -2 definitions
;--------------------------------------------------------
;--------------------------------------------------------
; absolute symbol definitions
;--------------------------------------------------------
;--------------------------------------------------------
; compiler-defined variables
;--------------------------------------------------------
      .area IDATA (DATA,REL,CON); pre-def
      .area IDATAROM (CODE,REL,CON); pre-def
      .area UDATA (DATA,REL,CON); pre-def
      .area UDATA (DATA,REL,CON) ;UDL_dummy_0     udata
r0x11ED:   .ds   1
r0x11F1:   .ds   1
r0x11F2:   .ds   1
      .area LOCALSTK (STK); local stack var
_dummy_STK00:    .ds   1
      .globl _dummy_STK00
;--------------------------------------------------------
; initialized data
;--------------------------------------------------------
;--------------------------------------------------------
; initialized data - mirror
;--------------------------------------------------------
      ;Following is optimization info,
      ;xxcdbxxW:dst:src+offset:srclit:just-remove
      ;--cdb--W:r0x11ED:_dummy_STK00+0:-1:0
      ;--cdb--W:r0x11EF:r0x11ED+0:-1:0
      ;--cdb--W:r0x11F0:NULL+0:0:0
      ;--cdb--W:r0x11F0:NULL+0:-1:1
      end
```

修改”dummy.asm” 即可与其它的 C 联结，呼叫的时候只要如下叙述:

```
extern int dummy(unsigned char , unsigned char);

….

z = dummy(x,y); …
```

要注意是如果有其它的本地变量，要加到 “.FUNC” 的 LOCAL LIST 里面，如果有呼叫别的 FUNCTION, 也要加到 “.FUNC” 的 “CALL LIST” 里面. 否则 LINKER 无法正确分配本地变量都间。

6. HYCON C Compiler 优化功能

目前 C COMPILER 对于产生的机器码是默认优化到最小。因为是 RISC 指令集所以优化到最小也几乎是优化到最快。大部份的优化是由 COMPILER 本身进行(sdcc.exe), 有部份优化在 LINKER (sdld.exe) 进行。

6.1 COMPILER 优化项目

COMPILER 主要的优化项目如下表 6-1-1:

优化项目	说明	开关
GCSE	https://en.wikipedia.org/wiki/Common_subexpression_elimination	--nogcse
Peephole optimization	https://en.wikipedia.org/wiki/Peephole_optimization	--no-peep
No-use Assignment Optimization	如果 W 的值写入某本地变量地址而没有在后面 REFERENCE 到, 则该写入动作可以移除。	--no-peep
Constant propagation in Bytes	BYTE BASE, 如果 b=0.. a=b 可以直接优化成 a=0. Constant 包括位址等或其他不变的 VARIABLE。	--no-peep
FSR0 Optimization	((char*)&ADCData)[0]=ADCL 可以把原来经过 FSR0 再写入的 CODE 变成直接寻址。或把 POINTER+OFFSET 的存取改用 PLUSW0 进行。	--fsrno

表 6-1-1 COMPILER 的优化项目

6.2 LINKER 优化项目

由于很多 CODE 要在 LINK 时才知道最佳的优化方式，所以有部份的 OPTIMIZATION 移至 LINKER，LINKER 的优化项目如下表 6-2-1。

优化项目	说明	开关
CALL + RET -> JMP	若 CALL function 跟着 RET 指令, 则 CALL FUNCTION 会被优化成 JMP function. 此优化要在 LINKER 才进行，因为 HY15P41 以后的产品有可能在 RET 中间由 LINKER INSERT 一段 EPILOGUE, 以移动堆栈指标。	-nx
JMP+RET -> RET	如果 JMP 到的 LABEL 后面是 RET 指令, 则 JMP 会直接换成 RET 指令。此优化留到 LINKER 也是因为在 RET 之前可能由 LINKER 再 INSERT EPILOGUE CODE.	-nx
MVL+RET -> RETLW	如果中间没有 LABEL 的话, 会进行此优化。此优化留到 LINKER 也是因为在 RET 之前可能由 LINKER 再 INSERT EPILOGUE CODE.	-nx
LONG CALL/JMP -> SHORT	JMP/CALL 的 LABEL 若在寻址范围之内，会简化成	-no

RCALL/RJ	RJ/RCALL	
BTSZ/BTSC _STATUS,n + JMP … 优化成 JC/JNC/JZ/JNZ …	JC/JNC/JZ/JNZ 可以把原来 2 个 WORD 优化成 1 个 WORD, 但寻址范围较小，LINKER 会在记算 OFFSET 之后再进行优化。	-nx
移除没使用的 FUNCTION	如果 FUNCTION 都没有被 REFERENCE 到，在 LINK 的时候会被移除，以节省空间。	-nr
Common Code Reuse	一样的 CODE 如果重复则可以把它变成 SUBROUTINE 去呼叫以节省空间. 但这优化会使程序变慢，所以默认是没有 ENABLE 的. 要 ENABLE 可以加 -cm n. 表示如果一段 REPEAT 的代码如果超过 n words，就把它变成 FUNCTION 使用 CALL 的方式以节省空间. 因为 CALL 一个 FUNCTION 可能也要 2 个 WORDS，所以 LINKER 也只会优化 5 个 WORD 以上的片段. 另外要老虑的时 HARDWARE PC STACK 只有 6 层，如果在第六层再去 CALL COMMON CODE，就会飞了. 默认第 5 层以下可以用 COMMON CODE. 如果有 INTERRUPT 而且里面有 2 层 PC STACK，则要另外加 -cl 4. 表示 4 层以上不可以 叫 common code.	
FUNCTION 顺序的调整	调整 FUNCTION 顺序以减少 FAR CALL 的数目. FAR CALL 要 2 个 WORD 而 NEAR CALL 只要 1 个 WORD.	-nf

表 6-2-1. LINKER 的优化项目

6.3 C Compiler 优化软件设置

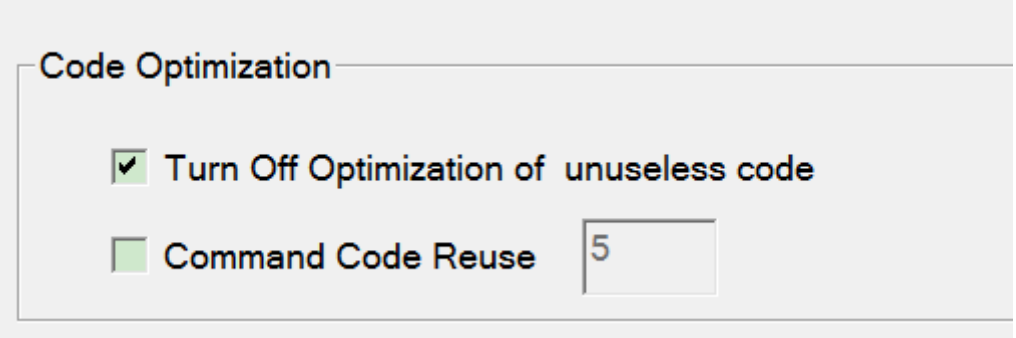
COMPLIER 的优化功能除了使用命令外，也可以通过软件设置达到程序优化目的，具体程序设置如下图 6-3-1。

图 6-3-1 对话框开启操作为：点击工程/右键/option/Complier option，即可打开对话框。

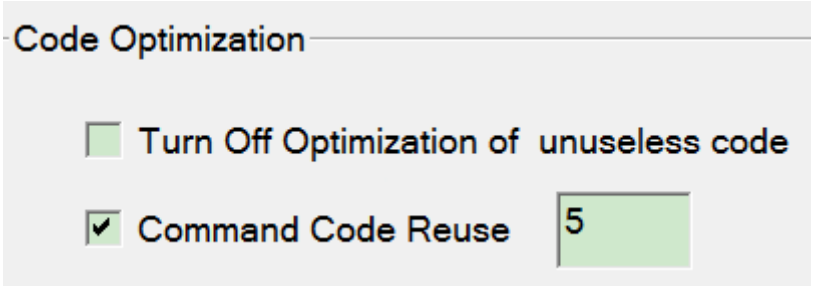
Code Optimization	功能定义
Turn off optimization unuseless code（图片选项 1）	对无用的代码（包括程序、数据）进行优化； 若勾选，则不进行优化； 若不勾选，则启动优化；
Command code Reuse（图片选项 2）	对重复使用的相同代码，数字表示多长的程序代码如果重复就把它变成 SUBROUTINE 用 CALL 的方式叫用. 因为 CALL 也可能要 2 个 WORD, RETURN 要 1 个 WORD, 即使设定 0~4, LINKER 也只会对 5 个 WORD 的重复片段进行优化. 数字大表示优化比较少，因为多个 WORD 的重复机率比较少，但也表示运行的速度比较不会受到影响. 该数字可以在 CODE-SIZE 与 PERFORMANCE 之

	间做一个平衡。

Turn off Optimization of unuseless code 勾选, and Common Code Reuse 不勾选，等于是都不启动优化的设定;



Turn off Optimization of unuseless code 不勾选, and Common Code Reuse(5)勾选，等于是全启动优化的设定;



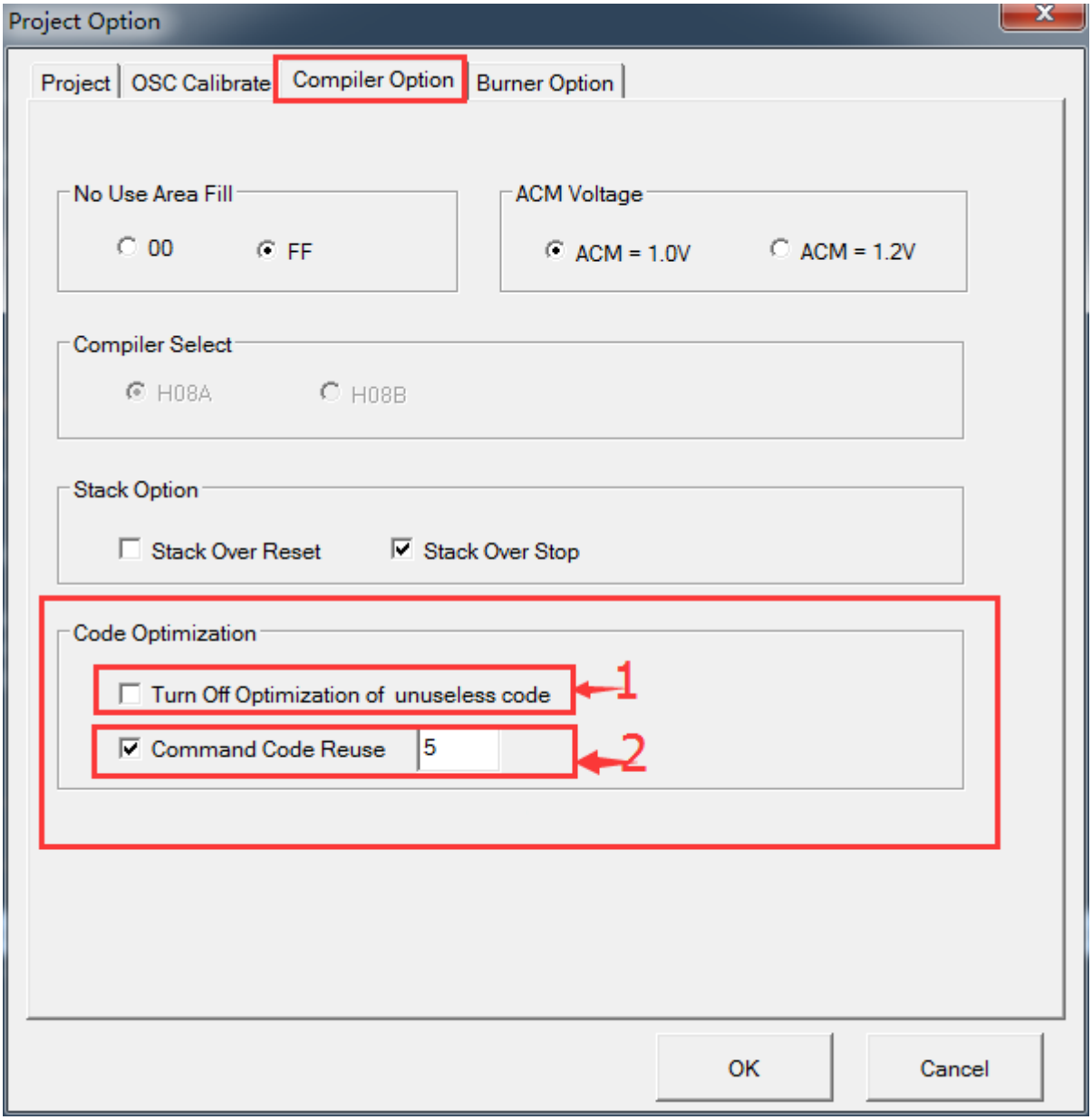

Project Option
Project
OSC Calibrate
Compiler Option
Burner Option
No Use Area Fill
00
FF
ACM Voltage
ACM = 1.0V
ACM = 1.2V
Compiler Select
H08A
H08B
Stack Option
Stack Over Reset
Stack Over Stop
Code Optimization
Turn Off Optimization of unuseless code
1
Command Code Reuse
5
2
OK
Cancel

图 6-3-1

7. HYCON C Compiler 程序范例

程序范例有调用到 HYCON C LIB 函数，阅读程序时请参照 HYCON C 函数库用户手册。

7.1 ADC 测量功能

程序功能：

I） 测量外部输入信号。ADC 输入端口为 AI0/AI1 ;参考电压输入端为 AI5/AI2，AI2=VDDA,AI5=VSS ; 放大倍数为 8*16 ;ADC 输出速率为 8HZ；

II） 测量内部温度传感器。ADC 输入端口为 TPSH0/TPSL0；参考电压输入端为 ACM/VSS ;放大倍数为 8；ADC 输出速率为 8HZ；

主文件及主函数 ADC_Main.c 如下：

```
#define   USE_HY11P14
/*******************************************************************************|
|* ADC_Main.c                                                                   *|
|* -----------------------------------------------------------------------------*|
|* Copyright 2017 HYCON Technology                                              *|
|* http://www.hycontek.com/                                                     *|
|* Program Description:                                                         *|
|* Function: ADC Demo For External Input                                        *|
|* IC Body: HY11H14                                                            *|
|* ADC INH | AI0 <- Input +                                                     *|
|* ADC INL | AI1 <- Input -                                                     *|
|* ADC VRH | AI2 <- VDDA                                                        *|
|* ADC VRL | AI5 <- VSS                                                         *|
|*         VSS |                                                               *|
|* ADC Demo For Internal Temperature Sensor                                     *|
|* IC Body: HY11H14                                                            *|
|* ADC INH <-TPS |                                                             *|
|* ADC INL <-TPS |                                                             *|
|* ADC VRH <-ACM |                                                             *|
|* ADC VRL <-VSS |                                                             *|
|* --------------                                                              *|
|*******************************************************************************/
/*-----------------------------------------------------------------------------*/
/* Includes                                                                    */
/*-----------------------------------------------------------------------------*/
```

```
#include "..\..\Driver\SFRType.h"
#include "..\..\Driver\HY11\RST.h"
#include "..\..\Driver\HY11\CLK.h"
#include "..\..\Driver\HY11\PWR.h"
#include "..\..\Driver\HY11\ADC.h"
#include "Display.h"
#include "main.h"
/*---------------------------------------------------------------------------*/
/* DEFINITIONS                                                               */
/*---------------------------------------------------------------------------*/
//#define    TPSTEST
/*---------------------------------------------------------------------------*/
/* Global CONSTANTS                                                          */
/*---------------------------------------------------------------------------*/
unsigned int TimerCount;
long ADC_Buffer;
MCUSTATUS    MCUSTATUSbits;
/*---------------------------------------------------------------------------*/
/* Function PROTOTYPES                                                       */
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* Main Function                                                             */
/*---------------------------------------------------------------------------*/
void main(void)
{

CLK_CPUCKOpen(MCKCN2_CPUCK_HSDCK,MCKCN1_ENXT_DISABLE,MCKCN2_HSS_HSCKDIV1,MC
KCN1_XTSP_XTL,MCKCN2_HSCK_HAO);
   PWR_Open(PWRCN_VDDAX_2V4,PWRCN_ENACM_ENABLE,50);
   CLK_PERCKHSDCKSel();

LCD_Open(LCDCN1_LCDPR_INTERNAL,MCKCN3_LCDS_PERACKDIV1,LCDCN1_VLCDX_3V3,LCDCN2
_LCDMX_duty4,LCDCN1_LCDBI_BIAS);
   ClearLCDframe();
   DisplayHYcon();
   delay(10000);

#if defined(TPSTEST)
   ADC_Open(MCKCN1_ADCCK_HSDCK,MCKCN1_ADCS_DIV1,
```

```
                AINET1_INH_TPSH0,AINET1_INL_TPSL0,
                AINET2_VRH_ACM,AINET2_VRL_VSS,
                ADCCN1_ADGN_8,ADCCN1_PGAGN_1,ADCCN2_VREGN_DIV2,
                ADCCN2_DCSET_0,ADCCN3_OSR_32768);
#else
   ADC_Open(MCKCN1_ADCCK_HSDCK,MCKCN1_ADCS_DIV1,
                AINET1_INH_AI0,AINET1_INL_AI1,
                AINET2_VRH_AI2,AINET2_VRL_AI3,
                ADCCN1_ADGN_16,ADCCN1_PGAGN_8,ADCCN2_VREGN_DIV2,
                ADCCN2_DCSET_0,ADCCN3_OSR_32768);
#endif
   ADC_Enable();
   ADC_INT_Enable();
   INT_GIE_Enable();
   MCUSTATUSbits._byte = 0;
   TimerCount=0;
   while(1)
   {
      Idle();
      if(MCUSTATUSbits.b_ADCdone==1)
      {
         DisplayNum(ADC_Buffer>>4);
         MCUSTATUSbits.b_ADCdone=0;
      }
      __asm__ ("NOP");
   }
}
/*---------------------------------------------------------------------------*/
/* Interrupt Service Routines                                                */
/*---------------------------------------------------------------------------*/
void ISR(void) __interrupt
{
   //ADC Event
   if(ADC_INT_IsFlag())          //
   {
      ADC_INT_ClearFlag();
      ADC_Buffer=ADC_GetData();
      MCUSTATUSbits.b_ADCdone=1;
   }
```

```
}
/*---------------------------------------------------------------------------*/
/* End Of File                                                               */
/*---------------------------------------------------------------------------*/
```

主函数头文件 main.h 如下：

```
/*---------------------------------------------------------------------------*/
/* STRUCTURES                                                                */
/*---------------------------------------------------------------------------*/
volatile typedef union _MCUSTATUS
{
   char    _byte;
   struct
   {
      unsigned b_ADCdone:1;
      unsigned b_TMAdone:1;
      unsigned b_TMBdone:1;
      unsigned b_TMCdone:1;
      unsigned b_Ext0done:1;
      unsigned b_Ext1done:1;
      unsigned b_UART_TxDone:1;
      unsigned b_UART_RxDone:1;
   };
} MCUSTATUS;
/*---------------------------------------------------------------------------*/
/* Global Variable/Function                                                  */
/*---------------------------------------------------------------------------*/
void delay(unsigned int i);
```

显示程序 Display.c 如下：

```
#define    USE_HY11P14
#include "..\..\Driver\SFRType.h"
#include "Display.h"
#include "LcdTable.h"
/*---------------------------------------------------------------------------*/
/* Clear LCD RAM Data                                                        */
/*---------------------------------------------------------------------------*/
```

```
void ClearLCDframe(void)
{
   unsigned char count;
   FSR0=&LCD0;
   for(count=10;count>0;count--)
      POINC0=0;
#if defined(USE_HY11P14) || defined(USE_HY11P35) || defined(USE_HY11P36) || defined(USE_HY11P54)
   FSR0=&LCD10;
   for(count=10;count>0;count--)
      POINC0=0;
#endif
}
/*---------------------------------------------------------------------------*/
/* Display HYcon Char                                                        */
/*---------------------------------------------------------------------------*/
void DisplayHYcon(void)
{
   LCD_WriteData(&LCD0,0x00);
   LCD_WriteData(&LCD1,Char_H);
   LCD_WriteData(&LCD2,Char_Y);
   LCD_WriteData(&LCD3,Char_C);
   LCD_WriteData(&LCD4,Char_O);
   LCD_WriteData(&LCD5,Char_N);
   LCD_WriteData(&LCD6,0x00);
   LCD_WriteData(&LCD7,0x00);
   LCD_WriteData(&LCD8,0x00);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY PASS                                                          */
/*---------------------------------------------------------------------------*/
void DisplayPASS(unsigned char Num)
{
   LCD_WriteData(&LCD0,Char_P);
   LCD_WriteData(&LCD1,Char_A);
   LCD_WriteData(&LCD2,Char_S);
   LCD_WriteData(&LCD3,Char_S);
   LCD_WriteData(&LCD4,seg[Num/10]);
   LCD_WriteData(&LCD5,seg[Num%10]);
   LCD_WriteData(&LCD6,0x00);
```

```
    LCD_WriteData(&LCD7,0x00);
    LCD_WriteData(&LCD8,0x00);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Number                                                        */
/*---------------------------------------------------------------------------*/
void DisplayNum(long Num)
{
    unsigned char count,MINUS;
    unsigned char *LCDAddr,LCDData;

    if((Num<0)||(Num>0x80000000))
    {
        Num=~Num;
        Num++;
        MINUS=1;
    }
    else
    {
        MINUS=0;
    }
    LCDAddr=&LCD5;
    for(count=0;count<6;count++)
    {
        LCDData=seg[Num%10];
        LCD_WriteData(LCDAddr,LCDData);
        Num=Num/10 ;
        LCDAddr--;
    }
    if(MINUS==1)
        LCD_WriteData(&LCD6,S_Minus);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Hexadecimal                                                   */
/*---------------------------------------------------------------------------*/
void DisplayHex(unsigned int Num)
{
    unsigned char count,*LCDAddr,LCDData;
```

```
    LCDAddr=&LCD5;
    for(count=0;count<6;count++)
    {
        LCDData=seg[Num%0x10];
        LCD_WriteData(LCDAddr,LCDData);
        Num=Num/0x10 ;
        LCDAddr--;
    }
}
```

显示程序头文件 Display.h 如下：

```
#include "..\..\Driver\HY11\LCD.h"

void ClearLCDframe(void);
void DisplayHYcon(void);
void DisplayPASS(unsigned char Num);
void DisplayNum(long Num);
void DisplayHex(unsigned int Num);
```

显示字符码 LcdTable.h 如下：

```
/*-------------------------------------------------------------------------|
| 7-segment display for LCD0 ~ LCD5                                        |
| -------------------------------------------------------------------------*/
                                    //                  a
#define   seg_a 0x10                //               ---------
#define   seg_b 0x20                //            |             |
#define   seg_c 0x40                //         f |             | b
#define   seg_d 0x08                //            |      g      |
#define   seg_e 0x04                //               ---------
#define   seg_f 0x01                //            |             |
#define   seg_g 0x02                //         e |             | c
#define   seg_h 0x80                //            |      d      |
                                    //               ---------     O <- h
const unsigned char seg[]={
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f,          // char "0"   0x00
            seg_b+seg_c,                                  // char "1"   0x01
            seg_a+seg_b+seg_d+seg_e+seg_g,                // char "2"   0x02
            seg_a+seg_b+seg_c+seg_d+seg_g,                // char "3"   0x03
```

```
            seg_b+seg_c+seg_f+seg_g,                          // char "4"   0x04
            seg_a+seg_c+seg_d+seg_f+seg_g,                    // char "5"   0x05
            seg_a+seg_c+seg_d+seg_e+seg_f+seg_g,              // char "6"   0x06
            seg_a+seg_b+seg_c+seg_f,                          // char "7"   0x07
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f+seg_g,        // char "8"   0x08
            seg_a+seg_b+seg_c+seg_d+seg_f+seg_g,              // char "9"   0x09
            seg_a+seg_b+seg_c+seg_e+seg_f+seg_g,              // char "A"   0x0a
            seg_c+seg_d+seg_e+seg_f+seg_g,                    // char "b"   0x0b
            seg_a+seg_d+seg_e+seg_f,                          // char "C"   0x0c
            seg_b+seg_c+seg_d+seg_e+seg_g,                    // char "d"   0x0d
            seg_a+seg_d+seg_e+seg_f+seg_g,                    // char "E"   0x0e
            seg_a+seg_e+seg_f+seg_g,                          // char "F"   0x0f
            seg_b+seg_c+seg_e+seg_f+seg_g,                    // char "H"   0x10
            seg_c,                                            // char "i"   0x11
            seg_b+seg_c+seg_d+seg_g,                          // char "J"   0x12
            seg_d+seg_e+seg_f,                                // char "L"   0x13
            seg_c+seg_e+seg_g,                                // char "n"   0x14
            seg_c+seg_d+seg_e+seg_g,                          // char "o"   0x15
            seg_a+seg_b+seg_e+seg_f+seg_g,                    // char "P"   0x16
            seg_a+seg_b+seg_c+seg_f+seg_g,                    // char "q"   0x17
            seg_e+seg_g,                                      // char "r"   0x18
            seg_d+seg_e+seg_f+seg_g,                          // char "t"   0x19
            seg_c+seg_e+seg_d,                                // char "u"   0x1a
            seg_b+seg_c+seg_d+seg_f+seg_g                     // char "y"   0x1b
};
#define   Char_A   seg_a+seg_b+seg_c+seg_e+seg_f+seg_g   // char "A"
#define   Char_B   seg_c+seg_d+seg_e+seg_f+seg_g         // char "b"
#define   Char_C   seg_a+seg_d+seg_e+seg_f               // char "C"
#define   Char_D   seg_b+seg_c+seg_d+seg_e+seg_g         // char "d"
#define   Char_E   seg_a+seg_d+seg_e+seg_f+seg_g         // char "E"
#define   Char_F   seg_a+seg_e+seg_f+seg_g               // char "F"
//Char_G                                                 // char "G"
#define   Char_H   seg_b+seg_c+seg_e+seg_f+seg_g         // char "H"
#define   Char_I   seg_c                                 // char "i"
#define   Char_J   seg_b+seg_c+seg_d+seg_g               // char "J"
//Char_K                                                 // char "K"
#define   Char_L   seg_d+seg_e+seg_f                     // char "L"
//Char_M                                                 // char "M"
#define   Char_N   seg_c+seg_e+seg_g                     // char "n"
```

```
#define   Char_O   seg_c+seg_d+seg_e+seg_g                  // char "o"
#define   Char_P   seg_a+seg_b+seg_e+seg_f+seg_g            // char "P"
#define   Char_Q   seg_a+seg_b+seg_c+seg_f+seg_g            // char "q"
#define   Char_R   seg_e+seg_g                              // char "r"
#define   Char_S   seg_a+seg_c+seg_d+seg_f+seg_g            // char "S"
#define   Char_T   seg_d+seg_e+seg_f+seg_g                  // char "t"
#define   Char_U   seg_c+seg_e+seg_d                        // char "u"
//Char_V                                                    // char "V"
//Char_W                                                     // char "W"
//Char_X                                                    // char "X"
#define   Char_Y   seg_b+seg_c+seg_d+seg_f+seg_g            // char "y"
#define   Char_Z   seg_a+seg_b+seg_d+seg_e+seg_g            // char "Z"
//---------------------------------------------------------------------------
// Define Symbols
//---------------------------------------------------------------------------

#define   S_g             0x80   //&LCD5   g
#define   S_m             0x80   //&LCD6   m
#define   S_V             0x80   //&LCD7   V
#define   S_A             0x20   //&LCD7   A
#define   S_K             0x10   //&LCD7   K
#define   S_M             0x01   //&LCD7   M
#define   S_Ohm           0x40   //&LCD7   Ohm
#define   S_Zero          0x40   //&LCD6   Zero
#define   S_Tare          0x20   //&LCD6   Tare
#define   S_Minus         0x10   //&LCD6   Minus
#define   S_Temp          0x02   //&LCD7   Temp
#define   S_Temp_C        0x08   //&LCD7   Temp_C
#define   S_Temp_F        0x04   //&LCD7   Temp_F
#define   S_Arrow1        0x01   //&LCD8   Arrow1
#define   S_Arrow2        0x02   //&LCD8   Arrow2
#define   S_Arrow3        0x04   //&LCD8   Arrow3
#define   S_Arrow4        0x08   //&LCD8   Arrow4
#define   S_Battery0      0x01   //&LCD6   Battery0
#define   S_Battery1      0x08   //&LCD6   Battery1
#define   S_Battery2      0x04   //&LCD6   Battery2
#define   S_Battery3      0x02   //&LCD6   Battery3
```

7.2 低电压检测功能（LVD）

程序功能：

PT1.2 作为低电压检测信号输入引脚，检测电压与 1.2v 做比较，判断输入信号是否比 1.2v 低。

主函数 LVD_Main.c 如下：

```
#define   USE_HY11P14
/*******************************************************************************|
|* LVD_Main.c                                                                  *|
|* ----------------------------------------------------------------------      *|
|* Copyright 2017 HYCON Technology                                             *|
|* http://www.hycontek.com/                                                    *|
|* Program Description:                                                        *|
|* IC Body: HY11P14                                                            *|
|*******************************************************************************/
/*-----------------------------------------------------------------------------*/
/* Includes                                                                    */
/*-----------------------------------------------------------------------------*/
#include "..\..\Driver\SFRType.h"
#include "..\..\Driver\HY11\RST.h"
#include "..\..\Driver\HY11\CLK.h"
#include "..\..\Driver\HY11\PWR.h"
#include "..\..\Driver\HY11\GPIO.h"
#include "..\..\Driver\HY11\LVD.h"
#include "Display.h"
#include "main.h"
/*-----------------------------------------------------------------------------*/
/* DEFINITIONS                                                                 */
/*-----------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------*/
/* Global CONSTANTS                                                            */
/*-----------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------*/
/* Function PROTOTYPES                                                         */
/*-----------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------*/
/* Main Function                                                               */
/*-----------------------------------------------------------------------------*/
```

```
void main(void)
{
   CLK_CPUCKOpen(MCKCN2_CPUCK_HSDCK,MCKCN1_ENXT_DISABLE,MCKCN2_HSS_HSCKDIV1,
                          MCKCN1_XTSP_XTL,MCKCN2_HSCK_HAO);
   PWR_Open(PWRCN_VDDAX_2V4,PWRCN_ENACM_ENABLE,50);
   CLK_PERCKHSDCKSel();
   LCD_Open(LCDCN1_LCDPR_INTERNAL,MCKCN3_LCDS_PERACKDIV1,
                    LCDCN1_VLCDX_3V3,LCDCN2_LCDMX_duty4,LCDCN1_LCDBI_BIAS);
   ClearLCDframe();
   DisplayHYcon();
   delay(20000);

   BZ_Open(MCKCN3_BZS_PERCKDIV16);
   GPIO_PT1AnalogMode(PT1DA_DA12_LVDIN);
   LVD_Open(LVDCN_VLDX_LVDIN);
   while(!LVD_GetLVDON());
   while(1)
   {
      if(LVD_GetStatus())
      {
         BZ_OutEnable();
         DisplayPASS(12);
      }
      else
      {
         BZ_OutDisable();
         DisplayHYcon();
      }
   }
}
/*---------------------------------------------------------------------------*/
/* Static Functions                                                          */
/*---------------------------------------------------------------------------*/
void delay(unsigned int i)
{
   while(i--);
}
/*---------------------------------------------------------------------------*/
/* End Of File                                                               */
```

```
/*---------------------------------------------------------------------------*/
```

主函数头文件 main.h 如下：

```
/*---------------------------------------------------------------------------*/
/* STRUCTURES                                                                */
/*---------------------------------------------------------------------------*/
volatile typedef union _MCUSTATUS
{
   char    _byte;
   struct
   {
      unsigned b_ADCdone:1;
      unsigned b_TMAdone:1;
      unsigned b_TMBdone:1;
      unsigned b_TMCdone:1;
      unsigned b_Ext0done:1;
      unsigned b_Ext1done:1;
      unsigned b_UART_TxDone:1;
      unsigned b_UART_RxDone:1;
   };
} MCUSTATUS;

/*---------------------------------------------------------------------------*/
/* Global Variable/Function                                                  */
/*---------------------------------------------------------------------------*/
void delay(unsigned int i);
```

显示函数 Display.c 如下：

```
#define   USE_HY11P13
#include "..\..\Driver\SFRType.h"
#include "Display.h"
#include "LcdTable.h"
/*---------------------------------------------------------------------------*/
/* Clear LCD RAM Data                                                        */
/*---------------------------------------------------------------------------*/
void ClearLCDframe(void)
{
   unsigned char count;
```

```
    FSR0=&LCD0;
    for(count=10;count>0;count--)
        POINC0=0;
#if defined(USE_HY11P14) || defined(USE_HY11P35) || defined(USE_HY11P36) || defined(USE_HY11P54)
    FSR0=&LCD10;
    for(count=10;count>0;count--)
        POINC0=0;
#endif
}
/*---------------------------------------------------------------------------*/
/* Display HYcon Char                                                        */
/*---------------------------------------------------------------------------*/
void DisplayHYcon(void)
{
    LCD_WriteData(&LCD0,0x00);
    LCD_WriteData(&LCD1,Char_H);
    LCD_WriteData(&LCD2,Char_Y);
    LCD_WriteData(&LCD3,Char_C);
    LCD_WriteData(&LCD4,Char_O);
    LCD_WriteData(&LCD5,Char_N);
    LCD_WriteData(&LCD6,0x00);
    LCD_WriteData(&LCD7,0x00);
    LCD_WriteData(&LCD8,0x00);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY PASS                                                          */
/*---------------------------------------------------------------------------*/
void DisplayPASS(unsigned char Num)
{
    LCD_WriteData(&LCD0,Char_P);
    LCD_WriteData(&LCD1,Char_A);
    LCD_WriteData(&LCD2,Char_S);
    LCD_WriteData(&LCD3,Char_S);
    LCD_WriteData(&LCD4,seg[Num/10]);
    LCD_WriteData(&LCD5,seg[Num%10]);
    LCD_WriteData(&LCD6,0x00);
    LCD_WriteData(&LCD7,0x00);
    LCD_WriteData(&LCD8,0x00);
}
```

```
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Number                                                        */
/*---------------------------------------------------------------------------*/
void DisplayNum(long Num)
{
   unsigned char count,MINUS;
   unsigned char *LCDAddr,LCDData;

   if((Num<0)||(Num>0x80000000))
   {
      Num=~Num;
      Num++;
      MINUS=1;
   }
   else
   {
      MINUS=0;
   }

   LCDAddr=&LCD5;
   for(count=0;count<6;count++)
   {
      LCDData=seg[Num%10];
      LCD_WriteData(LCDAddr,LCDData);
      Num=Num/10 ;
      LCDAddr--;
   }
   if(MINUS==1)
      LCD_WriteData(&LCD6,S_Minus);
}

/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Hexadecimal                                                   */
/*---------------------------------------------------------------------------*/
void DisplayHex(unsigned int Num)
{
   unsigned char count,*LCDAddr,LCDData;
```

```
    LCDAddr=&LCD5;
    for(count=0;count<6;count++)
    {
        LCDData=seg[Num%0x10];
        LCD_WriteData(LCDAddr,LCDData);
        Num=Num/0x10 ;
        LCDAddr--;
    }
}
```

显示函数头文件 Display.h 如下：

```
#include "..\..\Driver\HY11\LCD.h"
void ClearLCDframe(void);
void DisplayHYcon(void);
void DisplayPASS(unsigned char Num);
void DisplayNum(long Num);
void DisplayHex(unsigned int Num);
```

显示字符码 LcdTable.h 如下：

```
/*-------------------------------------------------------------------------|
| 7-segment display for LCD0 ~ LCD5                                        |
| -------------------------------------------------------------------------*/
                                    //                  a
#define   seg_a 0x10                //                ---------
#define   seg_b 0x20                //              |         |
#define   seg_c 0x40                //           f |         | b
#define   seg_d 0x08                //              |    g    |
#define   seg_e 0x04                //              ---------
#define   seg_f 0x01                //              |         |
#define   seg_g 0x02                //           e |         | c
#define   seg_h 0x80                //              |    d    |
                                    //              ---------    O <- h
const unsigned char seg[]={
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f,          // char "0"   0x00
            seg_b+seg_c,                                  // char "1"   0x01
            seg_a+seg_b+seg_d+seg_e+seg_g,                // char "2"   0x02
            seg_a+seg_b+seg_c+seg_d+seg_g,                // char "3"   0x03
            seg_b+seg_c+seg_f+seg_g,                      // char "4"   0x04
```

```
            seg_a+seg_c+seg_d+seg_f+seg_g,                  // char "5"   0x05
            seg_a+seg_c+seg_d+seg_e+seg_f+seg_g,            // char "6"   0x06
            seg_a+seg_b+seg_c+seg_f,                        // char "7"   0x07
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f+seg_g,      // char "8"   0x08
            seg_a+seg_b+seg_c+seg_d+seg_f+seg_g,            // char "9"   0x09
            seg_a+seg_b+seg_c+seg_e+seg_f+seg_g,            // char "A"   0x0a
            seg_c+seg_d+seg_e+seg_f+seg_g,                  // char "b"   0x0b
            seg_a+seg_d+seg_e+seg_f,                        // char "C"   0x0c
            seg_b+seg_c+seg_d+seg_e+seg_g,                  // char "d"   0x0d
            seg_a+seg_d+seg_e+seg_f+seg_g,                  // char "E"   0x0e
            seg_a+seg_e+seg_f+seg_g,                        // char "F"   0x0f
            seg_b+seg_c+seg_e+seg_f+seg_g,                  // char "H"   0x10
            seg_c,                                          // char "i"   0x11
            seg_b+seg_c+seg_d+seg_g,                        // char "J"   0x12
            seg_d+seg_e+seg_f,                              // char "L"   0x13
            seg_c+seg_e+seg_g,                              // char "n"   0x14
            seg_c+seg_d+seg_e+seg_g,                        // char "o"   0x15
            seg_a+seg_b+seg_e+seg_f+seg_g,                  // char "P"   0x16
            seg_a+seg_b+seg_c+seg_f+seg_g,                  // char "q"   0x17
            seg_e+seg_g,                                    // char "r"   0x18
            seg_d+seg_e+seg_f+seg_g,                        // char "t"   0x19
            seg_c+seg_e+seg_d,                              // char "u"   0x1a
            seg_b+seg_c+seg_d+seg_f+seg_g                   // char "y"   0x1b
};
#define   Char_A   seg_a+seg_b+seg_c+seg_e+seg_f+seg_g // char "A"
#define   Char_B   seg_c+seg_d+seg_e+seg_f+seg_g       // char "b"
#define   Char_C   seg_a+seg_d+seg_e+seg_f             // char "C"
#define   Char_D   seg_b+seg_c+seg_d+seg_e+seg_g       // char "d"
#define   Char_E   seg_a+seg_d+seg_e+seg_f+seg_g       // char "E"
#define   Char_F   seg_a+seg_e+seg_f+seg_g             // char "F"
//Char_G                                               // char "G"
#define   Char_H   seg_b+seg_c+seg_e+seg_f+seg_g       // char "H"
#define   Char_I   seg_c                               // char "i"
#define   Char_J   seg_b+seg_c+seg_d+seg_g             // char "J"
//Char_K                                               // char "K"
#define   Char_L   seg_d+seg_e+seg_f                   // char "L"
//Char_M                                               // char "M"
#define   Char_N   seg_c+seg_e+seg_g                   // char "n"
#define   Char_O   seg_c+seg_d+seg_e+seg_g             // char "o"
```

```
#define   Char_P   seg_a+seg_b+seg_e+seg_f+seg_g          // char "P"
#define   Char_Q   seg_a+seg_b+seg_c+seg_f+seg_g          // char "q"
#define   Char_R   seg_e+seg_g                            // char "r"
#define   Char_S   seg_a+seg_c+seg_d+seg_f+seg_g          // char "S"
#define   Char_T   seg_d+seg_e+seg_f+seg_g                // char "t"
#define   Char_U   seg_c+seg_e+seg_d                      // char "u"
//Char_V                                                 // char "V"
//Char_W                                                 // char "W"
//Char_X                                                 // char "X"
#define   Char_Y   seg_b+seg_c+seg_d+seg_f+seg_g          // char "y"
#define   Char_Z   seg_a+seg_b+seg_d+seg_e+seg_g          // char "Z"
//---------------------------------------------------------------------------
// Define Symbols
//---------------------------------------------------------------------------
#define   S_g             0x80   //&LCD5   g
#define   S_m             0x80   //&LCD6   m
#define   S_V             0x80   //&LCD7   V
#define   S_A             0x20   //&LCD7   A
#define   S_K             0x10   //&LCD7   K
#define   S_M             0x01   //&LCD7   M
#define   S_Ohm           0x40   //&LCD7   Ohm
#define   S_Zero          0x40   //&LCD6   Zero
#define   S_Tare          0x20   //&LCD6   Tare
#define   S_Minus         0x10   //&LCD6   Minus
#define   S_Temp          0x02   //&LCD7   Temp
#define   S_Temp_C        0x08   //&LCD7   Temp_C
#define   S_Temp_F        0x04   //&LCD7   Temp_F
#define   S_Arrow1        0x01   //&LCD8   Arrow1
#define   S_Arrow2        0x02   //&LCD8   Arrow2
#define   S_Arrow3        0x04   //&LCD8   Arrow3
#define   S_Arrow4        0x08   //&LCD8   Arrow4
#define   S_Battery0   0x01   //&LCD6   Battery0
#define   S_Battery1   0x08   //&LCD6   Battery1
#define   S_Battery2   0x04   //&LCD6   Battery2
#define   S_Battery3   0x02   //&LCD6   Battery3
```

7.3 中断功能使用

程序功能：

待机模式下，通过PT1.0/PT1.1外部中断功能唤醒MCU。

主函数INT_Main.c如下：

```
#define   USE_HY11P14
/*****************************************************************************|
|* INT_Main.c                                                                 *|
|* ------------------------------------------------------------------------- *|
|* Copyright 2017 HYCON Technology                                            *|
|* http://www.hycontek.com/                                                   *|
|* Program Description:                                                       *|
|* INT Demo For External Input                                                *|
|* IC Body: HY11H14                                                          *|
|* -------                                                                    *|
|*     INT0 |   <- S4                                                        *|
|*     INT1 |   <- S5                                                        *|
|* -------                                                                    *|
|*****************************************************************************/
/*---------------------------------------------------------------------------*/
/* Includes                                                                  */
/*---------------------------------------------------------------------------*/
#include "..\..\Driver\SFRType.h"
#include "..\..\Driver\HY11\RST.h"
#include "..\..\Driver\HY11\CLK.h"
#include "..\..\Driver\HY11\PWR.h"
#include "..\..\Driver\HY11\WDT.h"
#include "..\..\Driver\HY11\GPIO.h"
#include "Display.h"
#include "main.h"
/*---------------------------------------------------------------------------*/
/* DEFINITIONS                                                               */
/*---------------------------------------------------------------------------*/
#define   IDLEMode
/*---------------------------------------------------------------------------*/
/* Global CONSTANTS                                                          */
/*---------------------------------------------------------------------------*/
unsigned int TimerCount;
```

```
/*---------------------------------------------------------------------------*/
/* Function PROTOTYPES                                                       */
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* Main Function                                                             */
/*---------------------------------------------------------------------------*/
void main(void)
{
   CLK_CPUCKOpen(MCKCN2_CPUCK_HSDCK,MCKCN1_ENXT_DISABLE,MCKCN2_HSS_HSCKDIV1,
                          MCKCN1_XTSP_XTL,MCKCN2_HSCK_HAO);
   PWR_Open(PWRCN_VDDAX_2V4,PWRCN_ENACM_ENABLE,50);
   CLK_PERCKHSDCKSel();
   LCD_Open(LCDCN1_LCDPR_INTERNAL,MCKCN3_LCDS_PERACKDIV1,LCDCN1_VLCDX_3V3,
                LCDCN2_LCDMX_duty4,LCDCN1_LCDBI_BIAS);
   ClearLCDframe();
   DisplayHYcon();
   delay(10000);
   TimerCount=0;
   DisplayHex(TimerCount);

   BZ_Open(MCKCN3_BZS_PERCKDIV16);
   GPIO_PT1InputPullHight(PT1PU_PU10_ENABLE);
   GPIO_PT1InputPullHight(PT1PU_PU11_ENABLE);
   WDT_Open(TMACN_WDTS_512);
#if defined(IDLEMode)
   INT0_Enable();
   INT1_Enable();
   WDT_INT_Enable();
   INT_GIE_Enable();
   while(1)
   {
      Idle();
      __asm__ ("NOP");
      __asm__ ("NOP");
      DisplayHex(TimerCount);
   }

#else
   INT0_Disable();
```

```
    INT1_Disable();
    WDT_INT_Disable();
    INT_GIE_Disable();
    BZ_OutEnable();
    while(1)
    {
      if(!(PT1 & PT1_PT10_MSK))
      {
        TimerCount++;
        DisplayHex(TimerCount);
        WDT_Clear();
        BZ_OutDisable();
      }
      if(!(PT1 & PT1_PT11_MSK))
      {
        TimerCount--;
        DisplayHex(TimerCount);
        WDT_Clear();
        BZ_OutDisable();
      }
    }
#endif
}

/*---------------------------------------------------------------------------*/
/* Interrupt Service Routines                                                */
/*---------------------------------------------------------------------------*/
void ISR(void) __interrupt
{
    //INT0 Event
    if(INT0_IsFlag())
    {
      delay(1000);
      if(INT0_IsFlag())
      {
        INT0_ClearFlag();
        BZ_OutDisable();
        TimerCount++;
        WDT_Enable();
```

```
        WDT_Clear();
      }
    }

    //INT1 Event
    if(INT1_IsFlag())
    {
      delay(1000);
      if(INT1_IsFlag())
      {
        INT1_ClearFlag();
        BZ_OutDisable();
        TimerCount--;
        WDT_Enable();
        WDT_Clear();
      }
    }
    //WDT Event
    if(WDT_INT_IsFlag())
    {
      WDT_INT_ClearFlag();
      BZ_OutEnable();
    }
}
/*---------------------------------------------------------------------------*/
/* Static Functions                                                          */
/*---------------------------------------------------------------------------*/
void delay(unsigned int i)
{
    while(i--);
}
/*---------------------------------------------------------------------------*/
/* End Of File                                                               */
/*---------------------------------------------------------------------------*/
```

主函数头文件main.h如下：

```
/*---------------------------------------------------------------------------*/
/* STRUCTURES                                                                */
/*---------------------------------------------------------------------------*/
```

```
volatile typedef union _MCUSTATUS
{
   char    _byte;
   struct
   {
      unsigned b_ADCdone:1;
      unsigned b_TMAdone:1;
      unsigned b_TMBdone:1;
      unsigned b_TMCdone:1;
      unsigned b_Ext0done:1;
      unsigned b_Ext1done:1;
      unsigned b_UART_TxDone:1;
      unsigned b_UART_RxDone:1;
   };
} MCUSTATUS;
/*---------------------------------------------------------------------------*/
/* Global Variable/Function                                                  */
/*---------------------------------------------------------------------------*/
void delay(unsigned int i);
```

显示函数Display.c如下：

```
#define    USE_HY11P13
#include "..\..\Driver\SFRType.h"
#include "Display.h"
#include "LcdTable.h"
/*---------------------------------------------------------------------------*/
/* Clear LCD RAM Data                                                        */
/*---------------------------------------------------------------------------*/
void ClearLCDframe(void)
{
   unsigned char count;
   FSR0=&LCD0;
   for(count=10;count>0;count--)
      POINC0=0;
#if defined(USE_HY11P14) || defined(USE_HY11P35) || defined(USE_HY11P36) || defined(USE_HY11P54)
   FSR0=&LCD10;
   for(count=10;count>0;count--)
      POINC0=0;
#endif
```

```
}
/*---------------------------------------------------------------------------*/
/* Display HYcon Char                                                        */
/*---------------------------------------------------------------------------*/
void DisplayHYcon(void)
{
   LCD_WriteData(&LCD0,0x00);
   LCD_WriteData(&LCD1,Char_H);
   LCD_WriteData(&LCD2,Char_Y);
   LCD_WriteData(&LCD3,Char_C);
   LCD_WriteData(&LCD4,Char_O);
   LCD_WriteData(&LCD5,Char_N);
   LCD_WriteData(&LCD6,0x00);
   LCD_WriteData(&LCD7,0x00);
   LCD_WriteData(&LCD8,0x00);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY PASS                                                          */
/*---------------------------------------------------------------------------*/
void DisplayPASS(unsigned char Num)
{
   LCD_WriteData(&LCD0,Char_P);
   LCD_WriteData(&LCD1,Char_A);
   LCD_WriteData(&LCD2,Char_S);
   LCD_WriteData(&LCD3,Char_S);
   LCD_WriteData(&LCD4,seg[Num/10]);
   LCD_WriteData(&LCD5,seg[Num%10]);
   LCD_WriteData(&LCD6,0x00);
   LCD_WriteData(&LCD7,0x00);
   LCD_WriteData(&LCD8,0x00);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Number                                                        */
/*---------------------------------------------------------------------------*/
void DisplayNum(long Num)
{
   unsigned char count,MINUS;
   unsigned char *LCDAddr,LCDData;
   if((Num<0)||(Num>0x80000000))
```

```
  {
    Num=~Num;
    Num++;
    MINUS=1;
  }
  else
  {
    MINUS=0;
  }
  LCDAddr=&LCD5;
  for(count=0;count<6;count++)
  {
    LCDData=seg[Num%10];
    LCD_WriteData(LCDAddr,LCDData);
    Num=Num/10 ;
    LCDAddr--;
  }
  if(MINUS==1)
    LCD_WriteData(&LCD6,S_Minus);
}
/*---------------------------------------------------------------------------*/
/* LCD DISPLAY Hexadecimal                                                   */
/*---------------------------------------------------------------------------*/
void DisplayHex(unsigned int Num)
{
  unsigned char count,*LCDAddr,LCDData;

  LCDAddr=&LCD5;
  for(count=0;count<6;count++)
  {
    LCDData=seg[Num%0x10];
    LCD_WriteData(LCDAddr,LCDData);
    Num=Num/0x10 ;
    LCDAddr--;
  }
}
```

显示函数头文件Display.h如下：

```
#include "..\..\Driver\HY11\LCD.h"
```

```
void ClearLCDframe(void);
void DisplayHYcon(void);
void DisplayPASS(unsigned char Num);
void DisplayNum(long Num);
void DisplayHex(unsigned int Num);
```

显示字符码表LcdTable.h如下：

```
/*-------------------------------------------------------------------------|
| 7-segment display for LCD0 ~ LCD5                                        |
| -------------------------------------------------------------------------*/
                                    //                    a
#define   seg_a 0x10                //                 ---------
#define   seg_b 0x20                //               |         |
#define   seg_c 0x40                //            f |         | b
#define   seg_d 0x08                //               |    g    |
#define   seg_e 0x04                //                 ---------
#define   seg_f 0x01                //               |         |
#define   seg_g 0x02                //            e |         | c
#define   seg_h 0x80                //               |    d    |
                                    //                 ---------    O <- h
const unsigned char seg[]={
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f,          // char "0"   0x00
            seg_b+seg_c,                                  // char "1"   0x01
            seg_a+seg_b+seg_d+seg_e+seg_g,                // char "2"   0x02
            seg_a+seg_b+seg_c+seg_d+seg_g,                // char "3"   0x03
            seg_b+seg_c+seg_f+seg_g,                      // char "4"   0x04
            seg_a+seg_c+seg_d+seg_f+seg_g,                // char "5"   0x05
            seg_a+seg_c+seg_d+seg_e+seg_f+seg_g,          // char "6"   0x06
            seg_a+seg_b+seg_c+seg_f,                      // char "7"   0x07
            seg_a+seg_b+seg_c+seg_d+seg_e+seg_f+seg_g,    // char "8"   0x08
            seg_a+seg_b+seg_c+seg_d+seg_f+seg_g,          // char "9"   0x09
            seg_a+seg_b+seg_c+seg_e+seg_f+seg_g,          // char "A"   0x0a
            seg_c+seg_d+seg_e+seg_f+seg_g,                // char "b"   0x0b
            seg_a+seg_d+seg_e+seg_f,                      // char "C"   0x0c
            seg_b+seg_c+seg_d+seg_e+seg_g,                // char "d"   0x0d
            seg_a+seg_d+seg_e+seg_f+seg_g,                // char "E"   0x0e
            seg_a+seg_e+seg_f+seg_g,                      // char "F"   0x0f
            seg_b+seg_c+seg_e+seg_f+seg_g,                // char "H"   0x10
```

```
            seg_c,                                          // char "i"    0x11
            seg_b+seg_c+seg_d+seg_g,                        // char "J"    0x12
            seg_d+seg_e+seg_f,                              // char "L"    0x13
            seg_c+seg_e+seg_g,                              // char "n"    0x14
            seg_c+seg_d+seg_e+seg_g,                        // char "o"    0x15
            seg_a+seg_b+seg_e+seg_f+seg_g,                  // char "P"    0x16
            seg_a+seg_b+seg_c+seg_f+seg_g,                  // char "q"    0x17
            seg_e+seg_g,                                    // char "r"    0x18
            seg_d+seg_e+seg_f+seg_g,                        // char "t"    0x19
            seg_c+seg_e+seg_d,                              // char "u"    0x1a
            seg_b+seg_c+seg_d+seg_f+seg_g                   // char "y"    0x1b
};
#define   Char_A   seg_a+seg_b+seg_c+seg_e+seg_f+seg_g // char "A"
#define   Char_B   seg_c+seg_d+seg_e+seg_f+seg_g        // char "b"
#define   Char_C   seg_a+seg_d+seg_e+seg_f              // char "C"
#define   Char_D   seg_b+seg_c+seg_d+seg_e+seg_g        // char "d"
#define   Char_E   seg_a+seg_d+seg_e+seg_f+seg_g        // char "E"
#define   Char_F   seg_a+seg_e+seg_f+seg_g              // char "F"
//Char_G                                                // char "G"
#define   Char_H   seg_b+seg_c+seg_e+seg_f+seg_g        // char "H"
#define   Char_I   seg_c                                // char "i"
#define   Char_J   seg_b+seg_c+seg_d+seg_g              // char "J"
//Char_K                                                // char "K"
#define   Char_L   seg_d+seg_e+seg_f                    // char "L"
//Char_M                                                // char "M"
#define   Char_N   seg_c+seg_e+seg_g                    // char "n"
#define   Char_O   seg_c+seg_d+seg_e+seg_g              // char "o"
#define   Char_P   seg_a+seg_b+seg_e+seg_f+seg_g        // char "P"
#define   Char_Q   seg_a+seg_b+seg_c+seg_f+seg_g        // char "q"
#define   Char_R   seg_e+seg_g                          // char "r"
#define   Char_S   seg_a+seg_c+seg_d+seg_f+seg_g        // char "S"
#define   Char_T   seg_d+seg_e+seg_f+seg_g              // char "t"
#define   Char_U   seg_c+seg_e+seg_d                    // char "u"
//Char_V                                                // char "V"
//Char_W                                                // char "W"
//Char_X                                                // char "X"
#define   Char_Y   seg_b+seg_c+seg_d+seg_f+seg_g        // char "y"
#define   Char_Z   seg_a+seg_b+seg_d+seg_e+seg_g        // char "Z"
//------------------------------------------------------------------------
```

```
// Define Symbols
//---------------------------------------------------------------------------
#define   S_g             0x80   //&LCD5   g
#define   S_m             0x80   //&LCD6   m
#define   S_V             0x80   //&LCD7   V
#define   S_A             0x20   //&LCD7   A
#define   S_K             0x10   //&LCD7   K
#define   S_M             0x01   //&LCD7   M
#define   S_Ohm           0x40   //&LCD7   Ohm
#define   S_Zero          0x40   //&LCD6   Zero
#define   S_Tare          0x20   //&LCD6   Tare
#define   S_Minus         0x10   //&LCD6   Minus
#define   S_Temp          0x02   //&LCD7   Temp
#define   S_Temp_C        0x08   //&LCD7   Temp_C
#define   S_Temp_F        0x04   //&LCD7   Temp_F
#define   S_Arrow1        0x01   //&LCD8   Arrow1
#define   S_Arrow2        0x02   //&LCD8   Arrow2
#define   S_Arrow3        0x04   //&LCD8   Arrow3
#define   S_Arrow4        0x08   //&LCD8   Arrow4
#define   S_Battery0   0x01   //&LCD6   Battery0
#define   S_Battery1   0x08   //&LCD6   Battery1
#define   S_Battery2   0x04   //&LCD6   Battery2
#define   S_Battery3   0x02   //&LCD6   Battery3
```

8. HYCON C Compiler 常见问题

8.1 查看 ADC/TIMER 计数值

MCU 在 HALT 时，ADC/PWM/TIMER 等功能都是继续动作的状态；若要查看 ADC/TIMER 的计数值，需要先将计数值存储到 RAM 后再暂停下来查看。

8.2 不执行指令

断点设置在 BTSS/BTSZ/INSUZ/INSZ/DCSZ/DCSUZ/JC/JNC /JZ/JNZ 指令之后，就有可能在断点停下来而不执行指令。比如：

```
If (X>0)
   FOO();
```

如果断点设置在 FOO()函数，就有可能在 X<=0 时停下来而不执行；解决方式是在 FOO()函数之前添加 NOP 指令，解决如下：

```
        #define __NOP__   __asm NOP __endasm
        #define ICEDBG
        if(x>0)
        {
#ifdef ICEDBG
        __NOP__;
#endif
        foo();
        }
```

8.3 文件路径加载错误

如果在命令行有路径的问题，可以在 SDCC 加上 –V，表示 VERBOSE，可以显示 sdcc 叫用 pre-processor, assembler, linker 的 parameter。

用 COMPILER 编译时提示‘error: ADC.h: No such file or directory’，请检查对应文件的路径是否写正确。

8.4 watch 视窗添加变量个数限制

在watch视窗添加变量，当变量个数超过30个时，会提示输入个数超出范围。因为watch显示查看变量总数为30。

8.5 变量重复定义

如果一个变量定义在头文件，而这个头文件又被多个.c文件调用，则会出现变量重复定义，如下：

```
.\main.h:5: error 210: struct/union '_MCUSTATUS' redefined
.\main.h:4: error 177: previously defined here
.\main.h:18: error 0: Duplicate symbol 'MCUSTATUS', symbol IGNORED
.\main.h:18: error 177: previously defined here
.\main.h:18: error 51: typedef/enum 'MCUSTATUS' duplicate. Previous definition Ignored
.\ADC_MAIN.c:65: warning 93: type 'double' not supported assuming 'float'
```

解决方式：

不要在头文件定义变量，若多个.c文件都用到同一变量，则在其中一个.c文档定义变量，其他.c文档中用‘extern’声明即可。

9. 附录表

9.1 ASCII 码表

DEC	HEX	Symbol		DEC	HEX	Symbol		DEC	HEX	Symbol		DEC	HEX	Symbol
0	0	NUL		32	20	space		64	40	@		96	60	`
1	1	SOH		33	21	!		65	41	A		97	61	a
2	2	STX		34	22	"		66	42	B		98	62	b
3	3	ETX		35	23	#		67	43	C		99	63	c
4	4	EOT		36	24	$		68	44	D		100	64	d
5	5	ENQ		37	25	%		69	45	E		101	65	e
6	6	ACK		38	26	&		70	46	F		102	66	f
7	7	BEL		39	27	'		71	47	G		103	67	g
8	8	BS		40	28	(		72	48	H		104	68	h
9	9	HT		41	29	)		73	49	I		105	69	i
10	0A	LF		42	2A	*		74	4A	J		106	6A	j
11	0B	VT		43	2B	+		75	4B	K		107	6B	k
12	0C	FF		44	2C	,		76	4C	L		108	6C	l
13	0D	CR		45	2D	-		77	4D	M		109	6D	m
14	0E	SO		46	2E	.		78	4E	N		110	6E	n
15	0F	SI		47	2F	/		79	4F	O		111	6F	o
16	10	DLE		48	30	0		80	50	P		112	70	p
17	11	DC1		49	31	1		81	51	Q		113	71	q
18	12	DC2		50	32	2		82	52	R		114	72	r
19	13	DC3		51	33	3		83	53	S		115	73	s
20	14	DC4		52	34	4		84	54	T		116	74	t
21	15	NAK		53	35	5		85	55	U		117	75	u
22	16	SYN		54	36	6		86	56	V		118	76	v
23	17	ETB		55	37	7		87	57	W		119	77	w
24	18	CAN		56	38	8		88	58	X		120	78	x
25	19	EM		57	39	9		89	59	Y		121	79	y
26	1A	SUB		58	3A	:		90	5A	Z		122	7A	z
27	1B	ESC		59	3B	;		91	5B	[		123	7B	{
28	1C	FS		60	3C	<		92	5C	\		124	7C	|
29	1D	GS		61	3D	=		93	5D	]		125	7D	}
30	1E	RS		62	3E	>		94	5E	^		126	7E	~
31	1F	US		63	3F	?		95	5F	_		127	7F	

9.2 运算优先级表

优先级	运算符	含义	运算类型	结合性
1	()	圆括号	单目	自左向右
	[]	下标运算符		
	->	指向结构成员运算符		
	.	结构体运算符		
2	!	逻辑非运算	单目	自右向左
	~	按位取反运算符		
	++、--	自增、自减运算符		
	（类型关键字）	强制类型转换		
	+、-	正负号运算符		
	*	指针运算符		
	&	取地址运算符		
	sizeof	长度运算符		
3	*、/、%	乘、除、求余运算符	双目	自左向右
4	+、-	加减运算符	双目	自左向右
5	<<	左移运算符	双目	自左向右
	>>	右移运算符		
6	<、<=、>、>=	小于、小于等于、大于、大于等于	关系	自左向右
7	==、!=	等于、不等于	关系	自左向右
8	&	按位与运算符	位运算	自左向右
9	^	按位异或运算符	位运算	自左向右
10	|	按位或运算符	位运算	自左向右
11	&&	逻辑与运算符	位运算	自左向右
12	||	逻辑或运算符	位运算	自左向右
13	? :	条件运算符	三目	自右向左
14	=、+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=	赋值运算符	双目	自右向左
15	,	逗号运算符	顺序	自左向右

9.3 内建标准 ANSI C 函数表

Function	Description	Prototype Header File
int abs(int);	Get ABS value	<stdlib.h>
long int abs(long);	Get Long INT ABS	<stdlib.h>
float int atof(char *);	ASCII string to float.	<stdlib.h>
long int atol(char *);	ASCII string to long.	<stdlib.h>
long long int atoll(char*);	ASCII string to "long long".	<stdlib.h>
void _uitoa(unsigned int, __xdata char*, unsigned char radix);	Convert unsigned integer to ASCII.	<stdlib.h>
void _itoa(int, __xdata char*, unsigned char radix);	Convert integer to ASCII.	<stdlib.h>
void _ultoa(unsigned long, __xdata char*, unsigned char radix);	Convert unsigned long int to ASCII.	<stdlib.h>
void _ulltoa(unsigned long long , __xdata char*, unsigned char radix);	convert unsigned long long int to ASCII.	<stdlib.h>
void _lltoa(long long int, __xdata char *, unsigned char radix);	convert long long int to ASCII string.	<stdlib.h>
void _uitoa(long, __xdata char*, unsigned char radix);	Convert long int to ASCII string.	<stdlib.h>
int rand(void);	Generate (pseudo) random number;	<stdlib.h>
void srand(unsigned int seed);	Set random number seed.	<stdlib.h>
void __xdata *memcpy (void __xdata * dest, const void * src, size_t n);	Memory Copy.	<string.h>
void __xdata *memmove (void __xdata *dest, const __xdata void *src, size_t n);	Memory Move.	<string.h>
__xdata char *strcpy (__xdata char * dest, const char * src);	String Copy.	<string.h>
__xdata char *strncpy(__xdata char * dest, const char * src, size_t n);	String Copy with size limited.	<string.h>
__xdata char *strcat (__xdata char * dest, const char * src);	String concatenation.	<string.h>
__xdata char *strncat(__xdata char * dest, const char * src, size_t n);	String concatenation with size limit.	<string.h>
int memcmp (const void *s1, const void *s2, size_t n);	Memory Compare.	<string.h>
int strcmp (const char *s1, const char *s2);	String compare.	<string.h>

int strncmp(const char *s1, const char *s2, size_t n);	String compare with size limit.	<string.h>
char *strchr (const char *s, int c)	Find the location of char.	<string.h>
size_t strcspn(const char *s, const char *reject);	Get span until character in string	<string.h>
char *strstr (const char *haystack, const char *needle);	Find the occurrence of needle.	<string.h>
__xdata char *strtok (__xdata char * str, const char * delim);	Split string into tokens.	<string.h>
float sinf(float);	sine calculation.	<math.h>
float cosf(float);	cosine calculation.	<math.h>
float tanf(float);	the tangent of an angle of x radians.	<math.h>
float asinf(float);	arc-sine calculation.	<math.h>
float acosf(float);	arc-cosine calculation	<math.h>
float atanf(float);	arc-tangent calculation.	<math.h>
float atan2f(float x, float y)	calculation arc-tangent of x/y.	<math.h>
float sinhf(float)	hyperbolic sine calculation.	<math.h>
float coshf(float)	hyperbolic cosine calculation	<math.h>
float tanhf(float)	hyperbolic tangent calculation	<math.h>
float expf(float)	exponential calculation	<math.h>
float logf(float)	natural logarithm calculation.	<math.h>
float log10f(float)	logarithm calculation based of 10.0	<math.h>
float powf(float x, float y)	calculate y power of x.	<math.h>
float sqrtf(float)	square root calculation	<math.h>
float fabsf(float)	floating point absolute value.	<math.h>
float ceilf(float)	return nearest integer greater or equal.	<math.h>
float floorf(float)	return nearest integer less or equal.	<math.h>
float modff(float x, __xdata float *y);	Split floating point x to integer part and fraction part.	<math.h>

10. 参考资料

10.1 HY11P 系列规格书及用户手册

介绍 IC 的规格及用法，可到宏康官网下载：http://www.hycontek.com/cn/products-cn/6083

10.2 HYCON C IDE 用户手册

介绍 HYCON C IDE 使用，在安装目录下 Documents 文件夹取得。

10.3 函 HYCON C 函数库用户手册

介绍HYCON C函数库函数的定义功能及使用方式；在安装目录下Documents文件夹取得。

11. 版本记录

以下描述本档差异较大的地方，而标点符号与字形的改变不在此描述范围。

Version	Page	Revision Summary	The Date Of Revision
V01	ALL	First edition	2017/11/17