C for AT8 Series

C Compiler

Version 1.2

May 30, 2018

改 版 记 录

版本	日期	内 容 描 述	修正页
1.0	2017/08/14	新发布。	-
1.1	2017/10/27	增加常见问题。	17
1.2	2018/05/30	1. 增加 sbit 语法。 2. 增加选项说明。 3. 增加常见问题项目。	8 12 19

目 录

1 简介

C_AT8 为针对九齐科技的 AT8 系列 8 位 MCU IC 而提供的 C 语言编译程序（Compiler）。它被上层开发工具软件 *ATWIDE* 所调用以编译 C 程序，并结合 *ATASM* 组译器（Assembler）进一步组译及连结目的档来产生.bin 档，然后.bin 档可下载到板子或烧录到 OTP IC。

1.1 如何使用本手册

1. 简介

为何需要 *C_AT8* 与安装 *C_AT8* 的基本需求。

2. 使用 *C_AT8*

如何透过 *ATWIDE* 使用 *C_AT8*。

3. 语法与使用

介绍 *C_AT8* 的语法与使用方式

1.2 系统需求

以下列出使用 *C_AT8* 的系统需求。

- Pentium 1.3GHz 或更高级处理器，Win7、Win8、Win10 操作系统。
- 至少 2G SDRAM。
- 至少 2G 硬盘空间。

1.3 安装 *C_AT8*

请联系九齐科技来取得 *C_AT8* 的安装程序档，双击执行档后进入安装程序向导，然后依照画面提示将可轻松完成安装流程。

2 使用 C_AT8

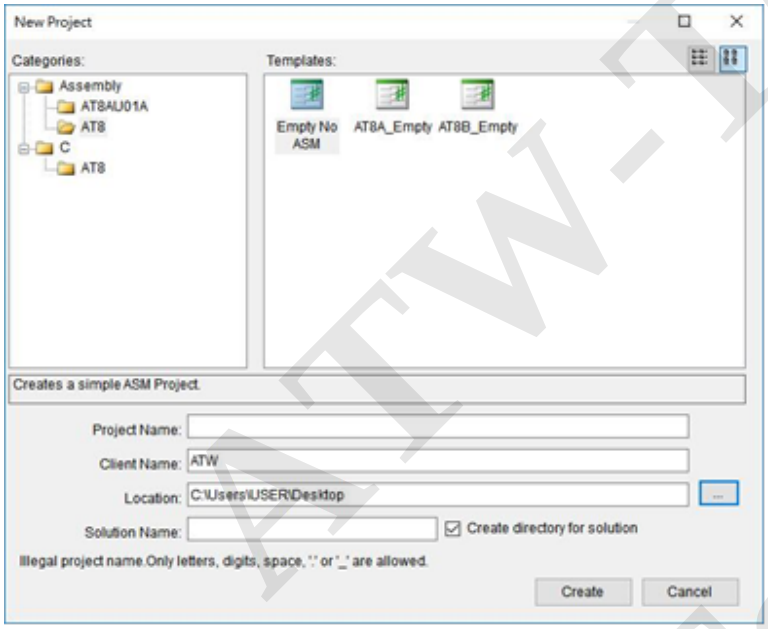
当用户使用 AT8 软件开发工具 *ATWIDE* 编写 C 程序后，在 *ATWIDE* 界面上按下 Build 时，软件开发工具会自动寻找并使用已安装于计算机上的 *C_AT8* 作编译。以下将说明透过 *ATWIDE* 来使用 *C_AT8* 的流程。

2.1 通过 *ATWIDE* 使用 *C_AT8*

ATWIDE 为九齐科技提供以开发 AT8 系列微控制器程序的整合性开发工具，主要目的为提供用户以汇编语言（Assembly）和 C 语言来编写程序，并拥有建置和强大的除错功能。当使用 *ATWIDE* 开发 AT8 项目，在建置和除错时，*ATWIDE* 会自动寻找并使用安装于计算机上的 *C_AT8* 工具链。以下简单的介绍使用 *ATWIDE* 开发 AT8 项目。更详细的操作方式，请参考 *ATWIDE* 使用手册。

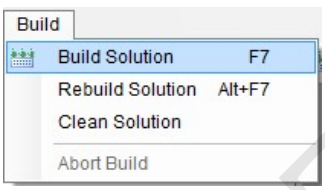
2.1.1 建立新项目

打开 *ATWIDE*，选择建立新项目。在 New Project 窗口内，左边的 Categories 选择 C，然后选择 AT8。指定项目名称及类型后，按下 Create。*ATWIDE* 将会自动产生必要文件，项目已于可建置状态。



2.1.2 建置

在 *ATWIDE* 主画面上的菜单选择 Build / Build Solution 进行建置（或按下快捷键 F7），即会调用 *C_AT8* 执行建置动作。若建置成功会在项目目录产生.bin 文件，以进一步提供下载或烧录。



3 语法与使用

C_AT8 支持标准的 ANSI C89 语法，并且针对 AT8 系列 IC 新增了一些特殊语法。

3.1 标准 C 语法

C_AT8 支持标准的 ANSI C89 语法，有关详细语言定义请参考：Standard ISO/IEC 9899 (http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899)

3.1.1 批注

批注支持两种格式，以双斜线起头的单行批注，以及/*起头，至*/结尾的多行批注。

范例：

```
// single line comment

/*
Multi line comment
*/
```

3.1.2 数据型态

以下表格列出 *C_AT8* 所使用的基本数据型态及允许的数据范围。其中 stdint 型态必须先行引用 stdint.h 方可使用。

型态	stdint	长度	范围
char	uint8_t	1 byte	0 ~ 255
signed char	int8_t	1 byte	-128 ~ 127
short	int16_t	2 bytes	-32768 ~ 32767
unsigned short	uint16_t	2 bytes	0 ~ 65535 (0xFFFF)
int	int16_t	2 bytes	-32768 ~ 32767
unsigned int	uint16_t	2 bytes	0 ~ 65535 (0xFFFF)
long	int32_t	4 bytes	-2147483648 ~ 2147483647
unsigned long	uint32_t	4 bytes	0 ~ 4294967295 (0xFFFFFFFF)

3.2 扩充语法

3.2.1 保留字

以下列出所有保留字，用户定义的符号不可和保留字相同。

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

__addressmod	__far	__pdata	__sram	_Static_assert
__asm__	__fixed16x16	__preserves_regs	__t0mdpage	register
__at	__flash	__reentrant	__trap	
__banked	__fpage	__sbit	__typeof	
__bit	__idata	__sfr	__using	
__builtin_offsetof	__interrupt	__sfr16	__wparam	
__code	__naked	__sfr32	__xdata	
__critical	__near	__shadowregs	__z88dk_callee	
__data	__nonbanked	__smallc	__z88dk_fastcall	
__eeprom	__overlay	__spage	_Alignas	

3.2.2 中断（interrupt）

中断服务副程序在 AT8 系列又分为硬件中断与软件中断，地址分别必须在于 0x08 与 0x01，在 C 语言中必须增加属性至函数定义，__interrupt(0)代表硬件中断服务程序、__interrupt(1)代表软件中断服务程序。编译程序会将此段程序安排在指定的地址，例如硬件中断必须在地址 0x08。编译程序会在进入函数前自动的保留当前系统状态，例如 ACC 寄存器、Status 寄存器、FSR 寄存器，并且在离开中断服务程序时自动还原状态。

范例：

```
void isr_hw(void) __interrupt(0)
{
    if(INTFbits.T0IF)
    {
        INTFbits.T0IF = 0;
        TMR0 = 0xc0;
        PORTB ^= 0x01;
```

```
    }
}

void isr_sw(void) __interrupt(1)
{
    // do something
}
```

3.2.3 寄存器位址定义

所有支持 AT8 IC 的特殊寄存器都已经被定义在程序安装文件夹的 include 目录下，档名为 IC 名称。建议用户直接使用该标头档，不须自行定义特殊寄存器。

3.2.4 寄存器位定义

__sbit 关键词可以将 8 位寄存器的其中一个位定义为新的变量。语法为：

```
__sbit <name> = <variable_8bit> : <bit>;
```

__sbit 仅可连结至已存在的 8 位变量其中某个位，而无法独立占用新的存储器空间。下面的程序范例示范如何使用 sbit 定义两个旗标，flag1 连结到 myvar 的第 0 个位，flag2 连结到 myvar 的第 3 个位（可选用的位为 0 到 7）。由 sbit 定义的变量只有单一位，可设定的值只有 0 和 1，读取的结果亦只有 0 和 1。

```
#include <stdint.h>

uint8_t myvar;
__sbit flag1 = myvar:0;
__sbit flag2 = myvar:3;

void main(void)
{
   flag2 = 1; // equals to myvar |= 0x08
   if (flag1)
       PORTB = 0;
   else
       PORTB = 0xff;
}
```

C_AT8 1.10 开始支持此语法，在此之前的版本并无支持。在先前的版本，必须建立独立 bit 定义的 struct，如下列范例所示：

```
typedef unsigned char uint8_t;

typedef union flag_t
{
    uint8_t all8bit;
    struct
    {
        unsigned FG0    : 1;
        unsigned FG1    : 1;
        unsigned FG2    : 1;
        unsigned FG3    : 1;
        unsigned FG4    : 1;
        unsigned FG5    : 1;
        unsigned FG6    : 1;
        unsigned FG7    : 1;
    };
} flag_t;

flag_t my_flag;

void main(void)
{
    // set value for 8bit register
    my_flag.all8bit = 0x12;

    // set value for 1bit flag
    my_flag.FG0 = 0;
}
```

3.2.5 内嵌汇编语言（Inline assembly）

在 C 语言之中可以内嵌汇编语言，使用__asm__关键词即可插入任意的汇编语言程序。下面的程序片段示范了内嵌汇编语言的写法，编译程序会将当前的地址存入 STK00 及 ACC 寄存器，并直接跳跃至另一个函数。

```
void switch_task_2(int current_pc);

void inline switch_task(void)
{
    __asm__("movia  $+4");
    __asm__("movar STK00");
    __asm__("movia  ($+2)>>8");
    __asm__("lgoto _switch_task_2");
}
```

3.2.6 指针属性

__code 和 __data 用于指定指标存放于 ROM 或 RAM。一般的指标占用 3 bytes，其中 2 bytes 存放地址，1 byte 存放指针型态以区分指针指向的位置是 ROM 或是 RAM。当编译程序掌握足够信息能判断指针型态时，可以省略指针型态的 1 byte。例如以下范例程序中的 data 为存放在 ROM 的数据，ptr1 及 ptr2 都是指向 data 的指标。然而 ptr1 有加上__code 属性，编译程序可以确定此指标只会指向 ROM 的数据，编译程序实际产生的机械码 ptr1 占用 2 bytes，而 ptr2 占用 3 bytes。使用指标时，若明确知道此指标只会指向 ROM 或 RAM，尽量事先指定 __code 或 __data 属性，以节省 RAM 使用量，亦可产生较为精简的指令。

```
const static char data[] = { 0, 1, 2, 3 };
__code const char *ptr1;
const char *ptr2;

void main(void)
{
    unsigned char i;
    ptr1 = data;
    ptr2 = data;

    for(i=0; i<(unsigned char)sizeof(data)/sizeof(data[0]); i++)
    {
        PORTB = *ptr1;
        PORTB = *ptr2;
        ptr1++;
        ptr2++;
    }
}
```

3.3 系统标头档

在 C_AT8 安装目录中的 include 文件夹有所有 C 语言使用的标头档（header file），本章节介绍这些标头档的内容及使用方法。

3.3.1 特殊指令宏

at8common.h 文件定义了常用的特殊汇编语言指令宏，以较为低阶的方式控制 IC 行为，用户可在适当的时机调用这些宏。

宏	说明
ENI()	打开中断功能
DISI()	关闭中断功能
INT()	引发软件中断
CLRWDT()	清除 watch dog 定时器
SLEEP()	进入 sleep
NOP()	空指令 nop

3.3.2 系统寄存器定义

AT8.h 会根据所选的 IC 自动引用该 IC 专用的标头档，所有 IC 支持的特殊寄存器都会定义在与 IC 同名的标头档。特殊寄存器又分为四种：general page 在宣告时加上属性 __sfr，F-page 在宣告时加上属性 __fpage，S-page 在宣告时加上属性 __spage，T0MD 在宣告时加上属性 __t0md。

在 C 语言层级，这些寄存器并没有任何分别。但用户仍然必须知道，这些寄存器实际存取的汇编语言代码并不相同。只有 general page 寄存器可以直接存取，像是直接设定某个位的值，或是直接对寄存器做互斥或(XOR)计算。除了 general page 以外的特殊寄存器无法直接存取，底层的汇编语言必须将特殊寄存器的值搬移到 ACC 寄存器，方能继续接下来的运算。

对于 general page 的特殊寄存器，我们鼓励用户单独的设定位为 1 或 0。而对于其他的特殊寄存器，则建议直接设定完整 8bit 的值。遵循这样的规则可得到较佳的机械码。

建议使用 AT8.h 而非直接使用 IC 专用标头档，可以减少更换 IC 造成标头档与函数库之间的不一致。此文件在 *C_AT8* 1.10 开始提供，用户使用古老版本的 *C_AT8* 时，必须注意切换 IC 之后必须同时更换标头档的引用。

3.3.3 ROM 数据读取

AT8_romaccess.h 定义了读取 ROM 数据的函数。

AT8 的 ROM 每个 word 为 14-bit，以一般 C 语言的指针仅能读取 14bit 中的低 8bit，使用 AT8_romaccess.h 之中定义的 read_14bit_rom 函数则可读取完整的 14 bits。

范例：

```
#include <AT8_romaccess.h>
……
__code char *rom_ptr;      //!< ROM pointer
int checksum_val;          //!< checksum value calculated by program.
checksum_val = 0;
for(rom_ptr=0; rom_ptr<(__code char*)&_checksum; ++rom_ptr)
    checksum_val += read_14bit_rom(rom_ptr);
```

详细范例可参考 *ATWIDE* 所内附的范例程序：Checksum。

3.4 选项

使用 *ATWIDE* 开发 C 语言项目，可以设定若干项目建置选项。这些选项可以控制编译程序、组译器、链接器的行为，点选菜单的项目（Project）/项目设定（Project Settings）可打开设定界面。

- 仅允许使用 RAM Bank0（Use RAM Bank0 only）：勾选此选项仅能使用 Bank0 的内存，产生的 Code size 较小，部分母体仅有单一 Bank，此选项将强制勾选。不勾选此选项将会在存取内存之前插入切换 bank 指令，允许使用所有的内存，但产生的 Code size 会较大。
- 开机清空内存（Clear RAM to zero on startup）：开机时先清空所有的内存，才执行 main 函数。而全局有初值变量并不受此选项影响，无论此选项勾选与否，有初值的全局变量会在进入 main 函数之前完成初值设定。取消此选项可以减少 Code size，但用户必须自行初始化全局无初值的变量，因为开机时的内存内容为不明。
- 产生汇编语言列表档（Generate ASM listing file）：组译完成产生列表档，档名为 *.lst。不勾选此选项可加快编译速度。
- 产生链接栏表档（Generate listing file）：链接后产生列表档，档名为 *.link.lst。此档为最终.bin 文件的反组译结果，不勾选此选项可以加快编译速度。
- 产生地址对应档（Generate map file）：链接后产生地址对应档，档名为 *.map。此文件包含地址分配信息，不勾选此选项可加快编译速度。
- 优化内存页面选择（Bank select optimization）：选择是否优化内存页面选择指令，当 Use RAM Bank0 only 选项未勾选时此选项才有效。勾选此选项，链接器将会试图移除重复的 banksel 以节省 Code size。必须注意此选项跟内嵌汇编语言（inline assembly）的搭配可能产生异常，若汇编语言有改变 RAM Bank 的行为，勾选此选项将会造成执行结果异常。
- 保留内存大小（Reserved RAM size）：保留给系统运作用的内存大小，用于中断服务程序进入前保留当前系统状态以及系统内部功能运作所需。其中用于存放函数参数的虚拟堆栈大小为保留内存大小减 3，例如保留内存大小设定为 16 byte，其中 13byte 为参数传递之虚拟堆栈。用户可视需求调整此值，最小值 6 最大值 16。
- 中断服务程序保留内存大小（Reserved RAM for interrupt）：保留给中断服务程序（ISR）所使用的内存大小，在进入中断前保存当前函数的变数状态。当在使用数组或调用函数而中断发生都有可能打断目前正在运算的寄存器，因此如果中断会造成行为不对，就需设定编译程序将运算被打断的变量先储存起来，依

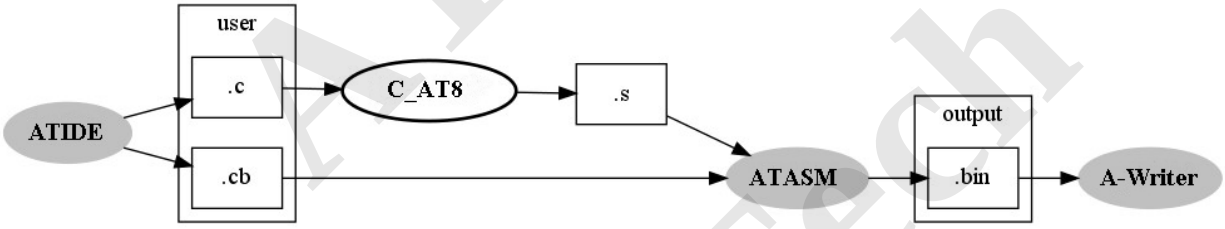
所用到的变量大小来设定需要保留给编译程序作备份的内存大小：最小值为 0，不保留任何函数调用的状态；最大值为虚拟堆栈大小（Reserved RAM size - 3）。设定值越大，会造成中断服务程序进入时间拉长，因为必须使用更多的指令来备份当前状态。实际指令数量因为备份用的内存位于不同的 Bank 而有少许不同，请参考下表。

保留大小	进入中断前额外指令	备注
0 byte	0	
1 byte	4 word	
2 byte	8 word	或 4 word
3 byte	10 word	或 6 word
4 byte	12 word	或 8 word
5 byte	14 word	或 10 word
……		
11 byte	26 word	或 22 word
12 byte	28 word	或 24 word
13 byte	30 word	或 26 word

- 引用文件路径（Include path）：设定 C 语言 include 关键词引用标头档的搜寻路径。默认的路径为项目根目录及 *C_AT8* 安装目录的 include 文件夹。用户可以增加自定义的路径。

3.5 开发流程

使用 *ATWIDE* 编写 C 语言程序，并设定项目所需的组态档.cb，*ATWIDE* 建置时自动调用 *C_AT8* 产生汇编语言文件.s，再调用 *ATASM* 组译汇编语言文件与组态档，产生最终.bin 档。最后可以藉由 *AWriter* 将.bin 档烧录至 IC。



3.6 进阶使用

本章节说明一些进阶使用的方式。

3.6.1 强制指定内存地址

一般 C 语言的变量并不会指定内存地址，而 MCU 程序开发偶尔会有指定地址的需求。C_AT8 有提供特殊语法供用户指定变量存放地址，在变量型态之前加上__at(addr)即可，addr 为指定的地址。

范例：

```
__at(0x23) unsigned char R0;
```

使用此功能时必须注意，不应该宣告变量在 SFR 区段，如果想要存取 SFR，请使用所选 IC 对应的标头档（Header file）预定义的变数。因为项目建置过程中，会连结 *C_AT8* 内建的静态函式库（static library），函式库已经预先宣告并占用所有的 SFR。若用户尝试重新定义 SFR，将会造成项目建置失败。如果想要重新命名 SFR，请使用#define 预处理指令。

范例：

```
#define BUTTON1 PORTBbits.PB0
...
if(BUTTON1 == 0)
{
  ...
}
```

而用户有多个.c 档时，也必须注意相似的状况。只能在其中一个.c 实际占用内存，其他的.c 必须使用 extern 关键词定义该变量为外部。

范例：

File: main.c

```
#include “my_var.h”
void main(void)
{
 R0 = 10; // use external variable
}
```

File: my_var.h

```
#ifndef MY_VAR_H
#define MY_VAR_H
extern __at(0x23) unsigned char R0;
#endif
```

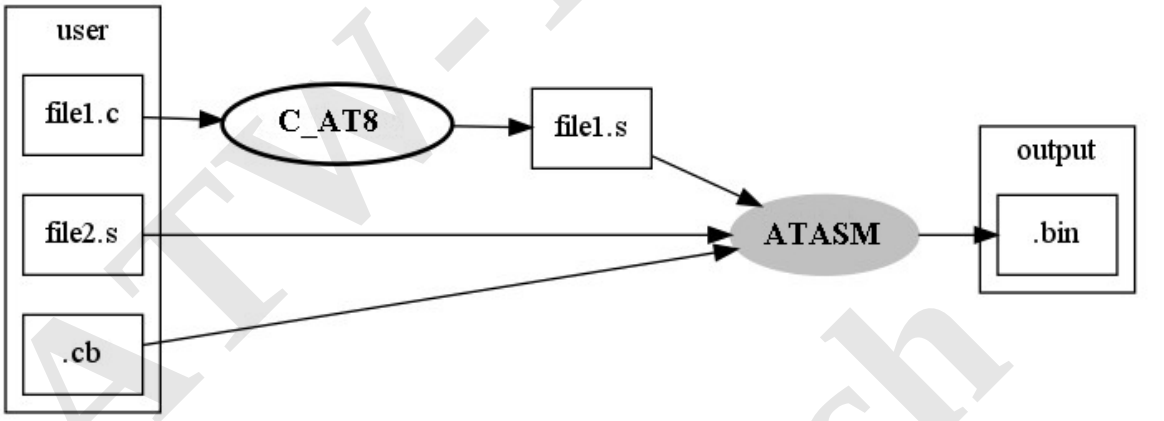
File: my_var.c

```
#include “my_var.h”
__at(0x23) unsigned char R0; // instance of variable
```

3.6.2 混合使用 C 与汇编语言

在开发流程我们可以看到，*C_AT8* 将.c 文件编译转换为.s 文件，之后由 *ATASM* 将.s 文件与.cb 文件整合

为.bin 文件。然而，.c 文件可以不只一个，对应产生的汇编语言.s 文件也可以不只一个。用户可以自行编写部分汇编语言.s 文件，与 *C_AT8* 所产生的.s 文件协同运作。在这将会介绍如何自行编写与 *C_AT8* 配合运作的.s 文件。



先从简单的范例开始——Rolling code 应用。Rolling code preset mode 应用必须将 ROM 的 0xE 与 0xF 留空白，待烧录时实际写入滚码，在编译的期间，0xE 与 0xF 不可有值。然而在 C 语言我们很难控制 IC 的某一地址必须填入什么值，除了__interrupt 关键词强制程序放在 0x1 或 0x8。解决方法是使用汇编语言与 C 语言协同运作，以下将示范如何使用汇编语言将 0xE 与 0xF 地址的值保留空白，在测试时则填入想要测试的值，并且用一段 C 语言程序读取 Rolling code 做验证。

范例文件共有三个：

- rom.s 将 0xE 及 0xF 地址填入 NOP，测试时填入 0x255、0x3AA 测试数据，并且导出符号___rolling_code_addr 供 C 语言使用。
- rom.h 定义外部符号__rolling_code_addr。
- main.c 主程序，读取 rolling code 并做验证。

File rom.s (assembly file):

```
        list c=on
    extern ___rolling_code_addr

    org 0x0e
___rolling_code_addr:
        nop             ; fill nop for rolling code
        nop

    end
```

在 rom.s 文件可以看到，导出的外部符号名称为___rolling_code_addr，有三个底线。当 C 语言被编译为汇编语言时，所有的符号都会被加一个底线，反过来说，我们希望汇编语言的符号被 C 语言使用时，人为多加上一个底线以作区别。

使用汇编语言，可以直接指定数据摆放的位置，透过 ORG 指令可以轻松做到这件事情。

File rom.h (C header file)

```
#ifndef ROM_H_D3SEKR8B
#define ROM_H_D3SEKR8B

extern __code char __rolling_code_addr;

#endif /* end of include guard: ROM_H_D3SEKR8B */
```

rom.h 主要只有一行，一个外部符号__rolling_code_addr 的定义，注意这边使用了__code 关键词明确的定义这个符号存在于 ROM。这符号名称的前缀底线只有两个，因为 C 语言编译为汇编语言时会自动在前面加上一个底线。

File main.c (C source code)

```
#include <AT8A53a.h>
#include <AT8_romaccess.h>
#include "rom.h"

char rolling_code[3];

// Assume the Rolling Code is 961109d = 0xEAA55
#define C_RC_B0 0x55 //Rolling Code bit7 ~ bit0
#define C_RC_B1 0xAA //Rolling Code bit15 ~ bit8
#define C_RC_B2 0x0E //Rolling Code bit19 ~ bit16

void main(void)
{
    int r_tmp;
    IOSTB = 0;                          // Set all PORTB are output mode
    IOSTA = 0;                          // Set all PORTA are output mode
    PORTB = 0;                          // PORTB data buffer = 0 (output low)
    PORTA = 0;                          // PORTA data buffer = 0 (output low)

    // Read content from Program Memory(ROM) address 0x0E & 0x0F

    // Read content of ROM address "0x0E"
    r_tmp = read_14bit_rom(&__rolling_code_addr);
    rolling_code[0] = r_tmp & 0xff;         // ROM data{0x00E} [7:0]
```

```
    rolling_code[1] = (r_tmp >> 8) & 0x03;  // ROM data{0x00E} [9:8]

    // Read content of ROM address "0x0F"
    r_tmp = read_14bit_rom(&__rolling_code_addr + 1);
    rolling_code[1] |= (r_tmp & 0x3f) << 2; // ROM data{0x00F} [15:10]
    rolling_code[2] = (r_tmp >> 6) & 0x0f;  // ROM data{0x00F} [19:16]

    if (rolling_code[0] == (char)C_RC_B0
        && rolling_code[1] == (char)C_RC_B1
        && rolling_code[2] == (char)C_RC_B2)
        PORTBbits.PB0 = 1;                  // Set PB0 output high (Rolling code is
match)

    while(1)
    {
        CLRWDT();
    }
}
```

main.c 使用 rom.h 所定义的符号__rolling_code_addr 抓取 ROM 数据，当然也可以选择不使用此符号，改为直接指定 0xE 地址，但这样 rolling code 换位置的时候就需要作较多的更动：必须更换 rom.s 里面 org 指令所指定的地址，以及 main.c 读取的地址。

接着我们再看范例 main.c，读取 ROM 数据所使用的函数 read_14bit_rom 是内建在 library 里面，在 AT8_romaccess.h 有函数原型宣号，然而它的实做并不是 C，而是汇编语言。这边一并将之列出，顺便用这个内建函数来说明如何从 C 语言调用汇编语言所定义的函数。

AT8_romaccess.h (system header file)

```
/** read 14bit data from ROM
 *
 * \param[in]   ptr     ROM address pointer
 * \return      14 bit data read from ROM
 */
int read_14bit_rom(const __code char *ptr);
```

read_14bit_rom.s (firmware implement)

```
    list c=on

#include "AT8_common.inc"
#include "macros.inc"
```

```
    ; export
    extern _read_14bit_rom

    ; import
    extern _TBHP
    extern _TBHD

    .segment "code"
_read_14bit_rom:
        sfun    _TBHP
        movr    STK00, W
        tablea
        movar   STK00       ; LSB in STK00
        sfunr   _TBHD       ; MSB in WREG
        ret

        END
```

上面这两个文件我们可以看到 C 语言的定义，以及组合语言的实做。第一个注意的是符号名称的差异，在 C 语言叫做 read_14bit_rom，而汇编语言则命名为_read_14bit_rom，多了一个底线。原因如同前面所讲的，C 语言在翻译成汇编语言之后，所有符号都会增加前缀底线。这个函数一个输入参数，为要读取的 ROM 地址指针，并且有一个回传值型态为 int（16-bit）。ROM 地址指针实际上占用 16 bits，两个 8-bit 寄存器。参数的传递优先使用 ACC，再来则是 STK00 到 STK12 公用寄存器。

以这个例子的 16-bit 指标为例，高位 8-bit 会被存放于 ACC，低 8-bit 会存放于 STK00。所以汇编语言实做此函数的第一步，就是将目前 ACC 存放的 ptr[15:8]移动到 TBHP 寄存器，将 STK00 存放的 ptr[7:0]移动到 ACC。

回传值的存放也是相同逻辑，高位放在 ACC，低位放在 STK00。当 TableA 完成 ROM 数据读取，ACC 存放的是 ROM[7:0]，我们立刻将 ACC 移动到 STK00，并将 TBHD 所存放的 ROM[13:8]移动到 ACC。最后加上 ret 返回这个函数。

对 main.c 而言，并不会在意 read_14bit_rom 究竟是用 C 写的，亦或汇编语言。只要输入参数与输出的回传值格式符合规范，就能完美的互助合作。

3.7 使用建议

以下提出一些开发 C 语言项目的建议。

- 尽量使用无符号（unsigned）变量，在部分的运算不用判断正负号会比较快。
- 表达式之中不要交互使用常数与变量，将常数集中才能有效的优化。

例如 1 + a + 2 就是不好的写法，1 跟 2 无法在编译时期计算。建议写成 a+1+2，如此一来 1+2 可在编译时期计算，执行期间只需要计算 a+3 即可。

- 使用 `if (INTFbits.T0IF)` 取代 `if(INTFbits.T0IF == 1)`，可以得到较为精简的程序。
- 不要连续单独设定 S-Page / F-Page 寄存器的某个 bit。
  因为 S-Page / F-Page 寄存器的读写皆需透过特别的指令，连续的单独位设定，会不断的读取、写入这些特殊寄存器，而不像 R-Page 的寄存器可使用 BCR / BSR 指令设定单独位。建议使用 S-Page / F-Page 寄存器时，先准备好所要设定的值，一次完成 8 个位的设定。
- 如果确定使用全局变量之前都有设定初值，可以指定 *C_AT8* 不要帮您做清 0 的动作以减少耗用 ROM。
- 如果 RAM 的使用量不大，可试着使用 small model，关闭 bank 切换。这样可以产生较为精简的程序。
- 不要将程序拆分为过多的.c 文件。这会影响优化，增加 RAM 的使用量。因为编译程序没办法假设两个函数不会同时执行，只能分配独立的内存给彼此。
- 尽量搭配使用同时期推出的 *ATASM*。因为 *C_AT8* 产生的文件会交给 *ATASM* 继续下一个步骤，若两者版本差距过大，或许会有不兼容的情形。例如 *C_AT8* 可能产生了旧版 *ATASM* 不支持的指令。
- 若已知指标只会指向 ROM 或 RAM，宣告时使用指针属性 __data 及 __code 告知编译程序。

3.8 常见问题

Q：打开多个中断源时为什么有时会漏掉中断？

A：

以同时打开 PortB change 中断与 Timer1 中断为例，使用 bit clear 指令来清除 T1IF 有可能会误清 PBIF，建议写立即值的方式针对 T1IF 写 0 来清除。以下为详细说明：

使用位 clear 指令（read modify write 指令）清除 T1IF（Timer1 interrupt flag）时，IC 会进行两个步骤：

1.1 先读取“INTF”所有位。

1.2 将 T1IF 位清为 0，其他位写 1，先前读取的值会回到“INTF”寄存器中。

但如果在“1.1”与“1.2”之间，PBIF 位因发生 PortB change 中断而被设为 1，那在“1.2”时就会被误清为 0，造成 PortB change 中断偶而会被忽略。

请参考下面程序代码来清除 T1IF（Timer1 interrupt flag）。

建议指令码	不建议指令码
`INTF = 0xF7;` 或 `INTFbits.T1IF = 0;`	`INTF &= 0xF7;`
产生汇编语言	产生汇编语言
`MOVIA 0xf7` `MOVAR _INTF`	`BCR _INTF, 3`

Q：在 main loop 及 interrupt service routine 皆有操作 Array 的程序，数据偶尔会读写到错误的地址？

A：

因为 Array 操作会使用到共享的系统寄存器，如果在操作到一半进入中断，且中断服务程序内也有操作 Array 的行为，则共享的系统寄存器状态将会被破坏，造成读写地址错误。

建议在这种情形使用 DISI()及 ENI()控制中断禁用启用，防止 Array 操作过程进入中断。

Q：我注意到 C:\ATW\ATWIDE\include\AT8A54.h 这类各种 IC body 的寄存器定义档，为何修改其中的寄存器名称后编译总是失败（Link fail）？

A：

寄存器的名称不仅定义于<icbody>.h，在静态链接函数库也必须存在相同的定义。静态链接函数库位于 C_AT8 安装目录的 lib 文件夹下，文件名称为<icbody>.a。静态链接函数库为二进制文件，无法由用户自行修改。修改标头档将造成链接时无法在函数库中找到相符的寄存器定义。

建议不要对系统内建的寄存器做重新命名。

Q：在中断服务程序之中设定变量值，并在一般程序流程中读取。读取结果异常？

A：

中断服务程序与正常流程共享的变量，建议在宣告时加上 volatile 关键词，防止变量被优化导致异常。用以下一个简单的例子说明共享变量被优化而导致程序异常：

```
uint8_t count;
void isr(void) __interrupt(0)
{
    if (INTFbits.T0IF)
    {
       INTFbits.T0IF=0;
       count++;
    }
}

void delay(uint8_t delay_count) {
    count=0;
    while (count < delay_count) {
        CLRWDT();
    }
}
```

在上面的例子中，当 delay 函数被调用，会先初始目前的 count 变数为 0，count 变数因打开 Timer 中断而在每次的中断递增，然后等 count 变数的值到达 delay_count，就会结束这个函数。但实际执行时 delay 中的 while 循环永远不会跳出，造成死循环。原因为编译程序优化机制认为 count 变量在设定为 0 之后并没有作其他运算，可以用常数 0 代换 count 变量，因此 while 循环的判断条件被优化成 `while (0 < delay_count)`，条件永远成立造成死循环。解决方法是改变 count 变量的宣告，以 volatile 关键词告知编译程序，count 变量不可被优化。

```
volatile uint8_t count;
```