九齐科技股份有限公司

Nyquest Technology Co., Ltd.

用户手册

NYC for NY8 Series

C Compiler

Version 2.2

May 26, 2025

目　录

1 简介

NYC_NY8 为针对九齐科技的 NY8 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 *NYIDE* 所调用以编译 C 程序，并结合 *NYASM* 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 *NYC_NY8* 与安装 *NYC_NY8* 的基本需求。

2. 使用 *NYC_NY8*

如何透过 *NYIDE* 使用 *NYC_NY8*。

3. 语法与使用

介绍 *NYC_NY8* 的语法与使用方式

1.2 系统需求

以下列出使用 *NYC_NY8* 的系统需求。

- Pentium 1.3GHz 或更高级处理器，Win7、Win8、Win10、Win11 操作系统。
- 至少 2G SDRAM。
- 至少 2G 硬盘空间。

1.3 安装 *NYC_NY8*

请联系九齐科技来取得 *NYC_NY8* 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 *NYC_NY8*

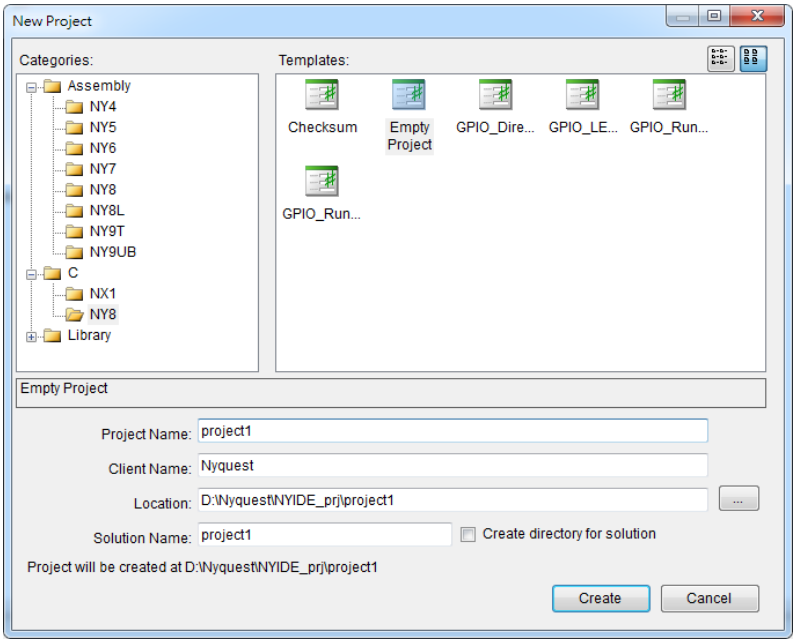
当用户使用 NY8 软件开发工具 *NYIDE* 编写 C 程序后，在 *NYIDE* 界面上按下 Build 时，软件开发工具会自动寻找并使用已安装于计算机上的 *NYC_NY8* 作编译。以下将说明透过 *NYIDE* 来使用 *NYC_NY8* 的流程。

2.1 通过 *NYIDE* 使用 *NYC_NY8*

NYIDE 为九齐科技提供以开发 NY4 / 5 / 6 / 7 / 8 / 9T / 9UB / NX1 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 *NYIDE* 开发 NY8 项目，在建置和除错时，*NYIDE* 会自动寻找并使用安装于计算机上的 *NYC_NY8* 工具链。以下简单的介绍使用 *NYIDE* 开发 NY8 项目。更详细的操作方式，请参考 *NYIDE* 使用手册。

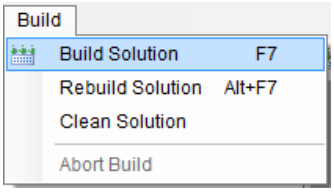
2.1.1 建立新项目

打开 *NYIDE*，选择建立新项目。在 New Project 窗口内，左边的 Categories 选择 C，然后选择 NY8。指定项目名称及类型后，按下 Create。*NYIDE* 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 *NYIDE* 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 *NYC_NY8* 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

NYC_NY8 支持标准的 ANSI C89 语法，并且针对 NY8 系列 IC 新增了一些特殊语法。

3.1 标准 C 语法

NYC_NY8 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 *NYC_NY8* 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 Calling Convention

函數調用的參數傳遞方式。需要手寫組合語言與 C 語言互相調用函數時，參數傳遞必須配合 C 語言的定義。在 14 bit NY8 與 16 bit NY8 由於採用不同的編譯器實作，Calling Convention 稍有不同。底下以一分相同的 C 語言函數宣告分別做說明

3.2.1 Calling Convention 14 bit NY8

參數從左到右，依序放在 Acc、STK00、STK01、STK02、……、STK13。unsigned int 有 2 byte，先放高位元再放低位元。

回傳值高位元在 Acc，低位元在 STK00。

以下面的函數調用為例

```
unsigned int add16(unsigned int a, unsigned int b);
g = add16(0x1234, 0x5678);
```

會得到調用的指令如下

```
MOVIA   0x78
MOVAR   STK02
MOVIA   0x56
MOVAR   STK01
MOVIA   0x34
MOVAR   STK00
MOVIA   0x12
MCALL   _add16
MOVAR   (_g + 1)
MOVR    STK00,W
MOVAR   _g
```

3.2.1 Calling Convention 16 bit NY8

參數從左到右，依序放在__rc0、__rc1、__rc2、……、__rc13。unsigned int 有 2 byte，先放低位元再放高位元。

回傳值也是依序放在 __rc0、__rc1、__rc2、……，先放低位元再放高位元。

這邊不使用 Acc 傳值。

以下面的函數調用為例

```
unsigned int add16(unsigned int a, unsigned int b);
g = add16(0x1234, 0x5678);
```

會得到調用的指令如下

```
movia   0x34
movar   __rc0
movia   0x12
movar   __rc1
movia   0x78
movar   __rc2
movia   0x56
movar   __rc3
lcall   add16
movr    __rc1, 0
movar   g+1
movr    __rc0, 0
movar   g
```

3.3 扩充语法

3.3.1 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	

3.3.2 中断（Interrupt）

中断服务副程序在 NY8 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
}
```

3.3.3 寄存器位址定义

所有支持 NY8 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。

3.3.4 寄存器位定义

用户可以自行定义单一位的数据结构，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;
// alias
#define flag1 my_flag.FG0
```

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

    // set value for 1bit flag, equals to my_flag.FG0 = 0
    flag1 = 0;
}
```

除了上述范例之外，在 14 bit NY8 有额外支持__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;
}
```

必须注意，只有 14 bit NY8 支持 sbit 语法，而 16 bit NY8 不支持，必须建立独立 bit 定义的 struct。

3.3.5 内嵌汇编语言（Inline Assembly）14 Bit NY8

在 14 Bit NY8 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程序。汇编程序会直接由 NYASM 处理。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");
}
```

3.3.6 内嵌汇编语言（Inline Assembly）16 Bit NY8

在 16 Bit NY8 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程序。汇编程序由 clang 处理，语法格式必须符合 GNU gas 规范。与 14 Bit NY8 不同之处除了语法，在汇编语言与外部 C 语言的互动也能更精细的控制。下面是一段清除 0x20~0x7F 内存数据的内嵌汇编语言范例。值得注意的是，汇编语言的字符串可以用 \n 换行，这样就能在一条 __asm__ 之中写出完整的程序逻辑。有需要做 label 的时候，可以使用数字编号产生暂时 label，比如说范例中使用了「1:」定义 label，并使用「lgoto 1b」跳跃到前面定义的 1 号 label (b 是往回头寻找的意思，与之对应的还有 f)。在汇编语言之中的 label 建议都使用数字形式，避免同一段程序多次展开，或是被 inline 之后造成 label 名称重复使用。

```
void f(void)
{
   __asm__("movia  0x20        \n"
           "movar  FSR         \n"
           "1:                \n"
           "clrr   INDF        \n"
           "incr   FSR,1       \n"
           "btrss  FSR,7       \n"
           "lgoto  1b          \n"
           );
}
```

前面的范例独立运作没有问题，但如果在 __asm__ 的前后加上其他 C 语言程序的话，可能会不如预期，譬如下面的程序，连续设定 PORTA 与 PORTB 为相同的数值 1，当设定 PORTB = 1 时，compiler 认为 Acc 当前的数值已知，所以省略 MOVIA 设定 Acc。

```
PORTA = 1; // codegen asm: MOVIA 1 + MOVAR PORTA
PORTB = 1; // codegen asm: MOVAR PORTB
```

当我们在这两行中间插入 __asm__ 改变了 Acc，必须告知 compiler 我们改变了那些系统寄存器。下面范例的 __asm__ 可以看到「::: "a", "fsr", "status"」标示改变的寄存器。

```
void f2(void)
{
    PORTA = 1;
    __asm__("movia  0x20         \n"
            "movar  FSR          \n"
            "1:                 \n"
            "clrr   INDF         \n"
            "incr   FSR,1        \n"
            "btrss  FSR,7        \n"
            "lgoto  1b           \n"
            ::: "a", "fsr", "status");
    PORTB = 1;
}
```

上面范例明确的告知 Acc、FSR、STATUS 寄存器会被改变，如此一来最后面的 C 程序:PORTB = 1;可以正确的动作。更详细的语法格式说明，可以参考 GNU gas 的 Extended Asm。

接着再来看一个与外部变量互动的例子，这个范例会将 a 与 b 相加，然后做高低 4 位交换。

```
#include <ny8.h>

NOINLINE char f(char a, char b)
{
    __asm__("movr %1,0            \n"
            "addar %0,1           \n"
            "swapr %0,1           \n"
            : "+r"(a)  // asm output
            : "r"(b)   // asm input
            : "a", "status");
    return a;
}

void main(void)
{
    while (1)
    {
        PORTA = f(PORTA, PORTB);
    }
}
```

在函数 f 之中，我们可以读取并修改 C 语言函数的变量 a、b。在这一段汇编程序，我们并不知道实际上变量会分配在哪个寄存器。我们可以利用%0 代表第 0 个参数、%1 代表第 1 个参数，并且在第 0 个参数链接到 C 语言的变量 a，也就是范例中的「"+r"(a)」。这个范例编译后的实际程序如下，可以在 lst 文件看到反组译的结果。

```
0000000c <f>:
       6: 1021          movr    0x21 <__rc1>, ToAcc
       7: 07a0          addar   0x20 <__rc0>, ToReg
       8: 16a0          swapr   0x20 <__rc0>, ToReg
       9: 0008          ret

00000014 <main>:
       a: 1005          movr    0x5, ToAcc
       b: 00a0          movar   0x20 <__rc0>
       c: 1006          movr    0x6, ToAcc
       d: 00a1          movar   0x21 <__rc1>
       e: 0806          lcall   0x6 <f>
       f: 1020          movr    0x20 <__rc0>, ToAcc
      10: 0085          movar   0x5
      11: 180a          lgoto   0xa <main>
```

3.3.7 内嵌組合語言區塊（Inline Assembly Block）

在 14 Bit NY8 有支持非标准的扩充语法(16 Bit NY8 不支持)，用 __asm……__endasm; 包住整个汇编程序区块。必须特别注意，结尾的__endasm; 有分号。在 NYIDE 使用单步执行功能的时候，这整块程序会一次全部执行。

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

在 16 Bit NY8 不支持这种语法。在 16 Bit NY8 建议使用前一章节的多行 asm，用\n 分隔，并且标示整段程序的输出入，以及影响的寄存器与旗标。

3.3.8 指标属性

__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++;
    }
}
```

3.4 系统标头档

在 NYC_NY8 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。

3.4.1 特殊指令宏

ny8common.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
UPDATE_REG(PORTx)	產生 movr 指令更新 port register
SetOSCCR(x)	设定 OSCCR 寄存器数值

3.4.2 系统寄存器定义

ny8.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。在 14 bit NY8，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 ny8.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 *NYC_NY8* 1.10 开始提供，用户使用古老版本的 *NYC_NY8* 时，必须注意切换 IC 之后必须同时更换标头档的引用。

3.4.3 ROM 数据读取

ny8_romaccess.h 定义了读取 ROM 数据的函数。

NY8 的 ROM 每个 word 为 14-bit 或 16-bit，以一般 C 语言的指针仅能读取每个 word 中的低 8bit，使用 ny8_romaccess.h 提供的 read_14bit_rom 函数可读取完整的 14 bits。16 bit NY8 提供的 read_16bit_rom 函数可以读取完整的 16 bits。

范例：

```
#include <ny8_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 *NYIDE* 所内附的范例程序：Checksum。

3.4.4 EEPROM 资料存取

部份 IC 具有内建的 EEPROM，必须使用特殊指令存取。NYC_NY8 提供了存取 EEPROM 的函数可以直接調用。

ny8_eeprom.h 内定義了存取 EEPROM 資料的函數，當選用具有內建 EEPROM 的 IC，則 ny8_eeprom.h 將會自動加入專案。提供函數如下：

函數	說明
eeprom_read	从指定地址读取一个 byte
eeprom_write	写入一个 byte 到指定地址(不再提供)
eeprom_write_timeout	写入一个 byte 到指定位址
eeprom_protect_lock	锁定/解除锁定 EEPROM 写保护
eeprom_protect_unlock	同上

- unsigned char eeprom_read (unsigned char address)
  参数 address 指定要读取的 EEPROM 地址。
  回传值为从指定地址读取的一个 byte 的资料。

- void eeprom_write (unsigned char address, unsigned char value)
  参数 address 指定要写入的 EEPROM 地址。
  参数 value 接受一个 byte 的数据，此数据将会写入到指定地址。
  这个 API 在 NYC_NY8 1.70 移除，请改用具备 Timeout 参数的 eeprom_write_timeout。

- void eeprom_write_timeout (unsigned char address, unsigned char value, unsigned char timeout)
  参数 address 指定要写入的 EEPROM 位址。
  参数 value 接受一个 byte 的资料，此资料将会写入到指定位址。
  参数 timeout 接受一个 byte 数值，代表本操作逾时的时间限制，每个 ic 可用的数值与对应的时间皆不同，请参考 ic datasheet。
  使用 eeprom_write 之前，必须先解除 EEPROM 的写保护。
  这个 API 新增于 NYC_NY8 1.43。

- void eeprom_protect_lock (void)
  锁定/解除锁定 EEPROM 写保护，依据 config block 选择 EEPROM Write Mode 而有所不同。一、单次写入模式（One Byte），每次調用此函数会解除 EEPROM 的写保护锁定，之后調用 eeprom_write 完成写入，硬件将会自动打開写保护锁定。在单次写入模式，用户必须在每一个写入动作之前执行解锁。二、在连续写入模式（Continuous Write），第一次調用 eeprom_protect_lock 会解除 EEPROM 的写保护锁定，后续可以执行任意数量的 eeprom_write。第二次調用 eeprom_protect_lock 重新打開写保护锁定。在连续写入模式，用户必须在所有写入完成后自行調用函数锁定。

- void eeprom_protect_unlock (void)

  与 void eeprom_protect_lock 完全相同的程序，使用不同的命名。同时調用 eeprom_protect_unlock 与 eeprom_protect_lock 会使用相同的程序空间，并不会额外消耗 ROM。

范例：

```
#include <ny8.h>
#include <ny8_eeprom.h>

void main(void) {
eeprom_protect_lock ();
eeprom_write (0, 2);
PORTB = eeprom_read (0);
}
```

詳細範例可參考 *NYIDE* 所內附的範例程式：eeprom-write-one-byte 与 eeprom-continuous-write。

3.4.5 内建函数 Multi_16b

普通 C 语言的乘法运算输入与输出的数据型态必须相同。两个 16 位整数相乘，只能得到 16 位的结果。如果希望得到 32 位的结果，必须将输入数据转型为 32 位(long)。内建函数 multi_16b 是特殊的乘法函数，输入为两个 16 位正整数，输出为 32 位正整数。对于 ROM、RAM 的资源消耗介于 16 位乘法到 32 位乘法之间。注意，multi_16b 无法计算负数。NYC_NY8 1.43 开始支持此函数。

范例：

```
#include <ny8.h>
unsigned int a = 0x1234;
unsigned int b = 0x5678;
unsigned long c;
void main(void) {

    c = multi_16b(a, b); // c == 0x6260060
}
```

3.4.6 内建函数 clear_ram

清除 ic 所有 RAM 为 0，不仅限于 C 语言宣告的变量，包含用户程序并未使用的 RAM、以及 Compiler 所产生的寄存变量都会设定为 0。特殊功能寄存器(SFR)则不会被改变。实际的程序逻辑与 NYIDE project setting 勾选 Clear RAM to zero 相同。两者不同之处在于执行时机，Clear RAM to zero 只会在进入 main 函数之前执行一次，而 clear_ram 可以在任何时间手动执行。这个函数在不同 ic 会自动链接正确的程序，确保设定的 RAM 范围符合 ic 规范。NYC_NY8 1.60 开始支持此函数。

函数原型宣告:

```
// ny8common.h
extern void clear_ram(void);
```

3.5 选项

使用 *NYIDE* 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）：只有 14 bit NY8 支持。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并且无初值的变量，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）：只有 14 bit NY8 支持。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（optimization）：只有 14 bit NY8 支持。可以选择 Level 1 到 Level 3 的优化等级，高等级的优化可以得到更精简的程序。但是必须注意，此选项跟内嵌汇编语言（inline assembly）的搭配可能产生异常。
- 中断处理程序备份寄存器（Backup register for interrupt）：选择特殊寄存器在进入中断服务程序时是否要备份状态。TBHP 寄存器使用于读取 ROM 数据，如果确定没有读 ROM 行为可以关闭备份。PCHBUF 寄存器使用于间接跳跃，如果确定没有间接跳跃，比如说 switch-case，可以关闭备份。建议在项目编译完成后，打开反组译产生的.lst 文件确认这些特殊寄存器是否真的没有被中断服务程序使用。
- 中断服务程序保留内存大小（Reserved RAM for interrupt）：只有 14 bit NY8 需要设定。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 13。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

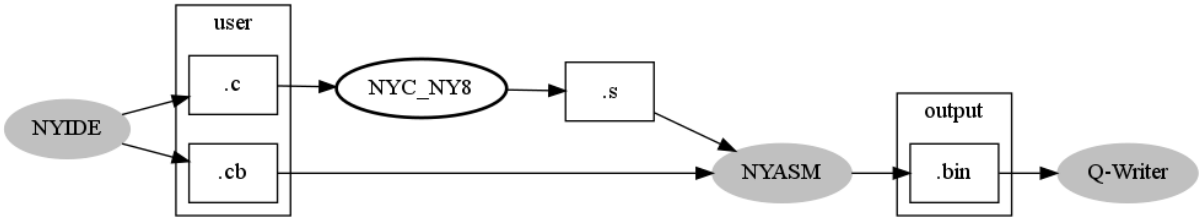
- 引用文件路径（Include path）：设定 C 语言 include 关键词引用标头档的搜寻路径。默认的路径为项目根目录及 *NYC_NY8* 安装目录的 include 文件夹。用户可以增加自定义的路径。

3.6 开发流程

NYC_NY8 一般都是由 NYIDE 自动调用，使用户只会看到 NYC_NY8 之中的标头档(.h)而不会直接执行内部的执行档(.exe)。这边说明 NYC_NY8 与其他软件的互动方式。

3.6.1 14 bit NY8

在 14 bit NY8，编译程序采用的是 sdcc。这是单纯的编译程序，需要搭配组译器(*NYASM*)一起使用。在 *NYIDE* 编写 C 语言程序，并设定项目所需的组态档.cb，*NYIDE* 建置时自动调用 *NYC_NY8* 产生汇编语言文件.s，再调用 *NYASM* 组译汇编语言文件与组态档，产生最终.bin 档。最后可以藉由 *Q-Writer* 将.bin 档烧录至 IC。



3.6.2 16 bit NY8

在 16 bit NY8，编译程序采用 clang/llvm。这是整合了编译程序、组译器、链接器的设计。不需要安装 NYASM 即可运作。在 *NYIDE* 编写 C 语言程序，并设定项目所需的组态档.cb，*NYIDE* 建置时自动调用 *NYC_NY8* 产生 bin 文件。最后可以藉由 *Q-Writer* 将.bin 档烧录至 IC。

3.7 进阶使用

本章节说明一些进阶使用的方式。

3.7.1 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。NYC_NY8 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;
```

请注意，变量不应该宣告在 SFR 所定义的位置，如果想要存取 SFR，请使用所选 IC 对应的头文件（Header file）预定义的变量。因为项目建置过程中，会连结 *NYC_NY8* 内建的静态函式库（static library），函式库使用的是在内建表头档中预先宣告的 SFR，若用户改变了 SFR 定义，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
}
```

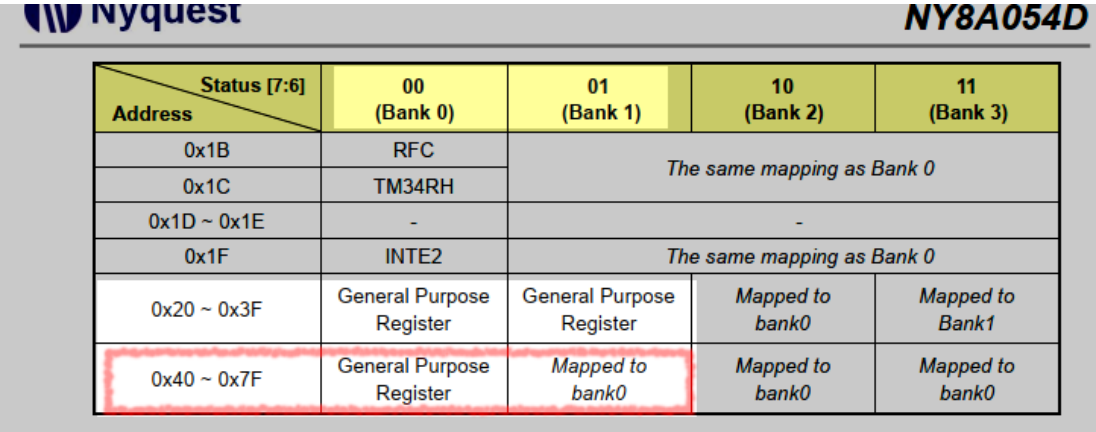
File: my_var.h

```
#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
```

强制指定地址必须注意项目设定选项的保留内存大小（Reserved RAM size），必须保留足够的 share bank 给系统使用。



Nyquest NY8A054D

Status [7:6] / Address	00 (Bank 0)	01 (Bank 1)	10 (Bank 2)	11 (Bank 3)
0x1B	RFC	*The same mapping as Bank 0*		
0x1C	TM34RH			
0x1D ~ 0x1E	-	-		
0x1F	INTE2	*The same mapping as Bank 0*		
0x20 ~ 0x3F	General Purpose Register	General Purpose Register	*Mapped to bank0*	*Mapped to Bank1*
0x40 ~ 0x7F	General Purpose Register	*Mapped to bank0*	*Mapped to bank0*	*Mapped to bank0*

上图是 NY8A054D datasheet 18 页，关于 R-Page address mapping 的部份。红色框框部份，无论 Bank0 或 Bank1 皆能存取相同的内存。系统保留内存必须位於紅框之內(0x40~0x7F)。

3.7.2 强制指定函数地址 16 bit NY8

与前一章节不同，在 16 bit NY8 并没有提供 __at(0x123) 这种语法。取而代之的是，标准的 C 语言指针。我们可以写 (volatile unsigned char*)0x123 将 0x123 常数转型为指标，前面再加一个星号做解指标(dereference)即可存取内存地址 0x123。更实际的范例如下

File: main.c

```
#define MY_RAM (*(volatile unsigned char*)0x123)
void f1(void)
{
    MY_RAM = 10;
}
```

必须注意前面范例的内存地址 0x123 不能与编译程序自动分配的地址重迭，否则可能会破坏系统状态。可以藉由项目编译后的.map 文件来得知已分配的内存地址。

3.7.3 强制指定函数地址 14 bit NY8

一般来说，C 语言的函数并不需指定起始地址，由链接器（linker）将函数自动安排在适当的位置。而 MCU 程式開發偶爾會有指定位址的需求。NYC_NY8 有提供特殊語法供使用者指定函数起始地址。在函数回传型态之前加上__at(addr)即可，addr 為指定的位址。

范例：

```
__at(0x0e) _Noreturn void get_rolling_code_0(void) __naked
{
    __asm__("retia 0x00");
}
```

请不要指定地址 0x00 给函数，因为地址 0x00 已经被 NYC_NY8 的启动程序占用。

3.7.4 强制指定函数地址 16 bit NY8

与前一章节不同，在 16 bit NY8 并没有提供 __at(0x123) 这种语法。对于 Rolling Code 的应用，需要强制指定地址，有提供宏 SET_ROLLING_CODE_ADDR 用来强制指定地址。

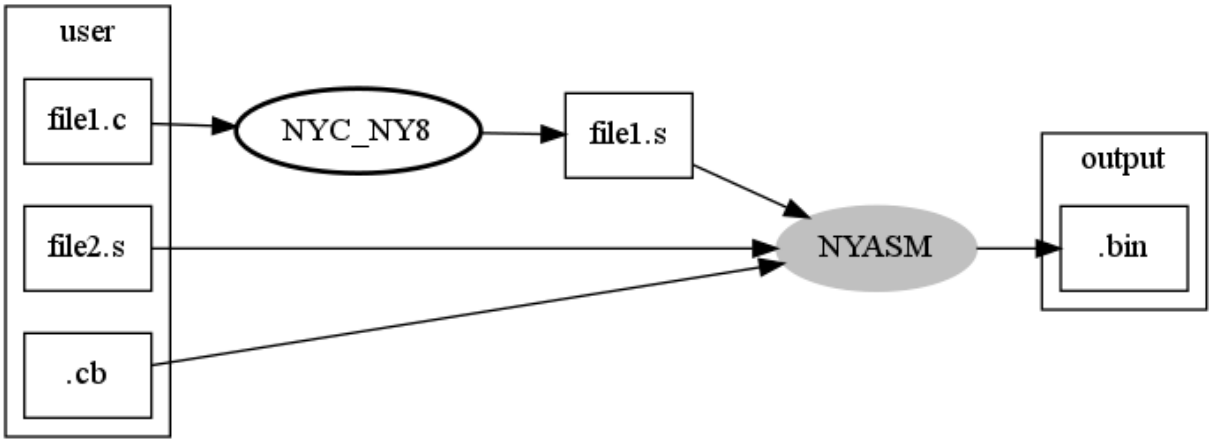
File: rolling-code-data.c

```
SET_ROLLING_CODE_ADDR(0x0D);
#define ATTR __attribute__((noinline, naked, ret_acc, section(".rolling")))
ATTR unsigned char get_rolling_code_0x0D(void)
{
    __asm__("retia 0");
}
ATTR unsigned char get_rolling_code_0x0E(void)
{
    __asm__("retia 0");
}
```

更详细的使用范例请参考 NYIDE 之中的 Example: Rolling Code。

3.7.5 混合使用 C 与汇编语言 14 bit NY8

在开发流程我们可以看到，*NYC_NY8* 将.c 文件编译转换为.s 文件，之后由 *NYASM* 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 *NYC_NY8* 所产生的.s 文件协同运作。在这将会介绍如何自行编写与 *NYC_NY8* 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 <ny8a053a.h>
#include <ny8_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 ny8_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。

ny8_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 "ny8_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。

3.7.6 混合使用 C 与汇编语言于 16 bit NY8

在 16 bit NY8 也可以混合使用 C 与汇编语言。内建的范例程序 Interrupt_with_assembly 示范了该如何在 C 语言项目加入汇编语言原始文件。由于 compiler 与 14 bit NY8 不同，有些特性必须特别注意。

- 扩展名大小写 .s 与 .S 不同。小写.s 是纯粹的汇编语言，而大写.S 会通过 C 语言的前处理器(preprocessor)。更具体的说，大写.S 可以支持「#define」、「#include」这类 C 语言的前编译指令。
- 汇编语言采用 NY8 16 bit 的语法，datasheet 上面的指令都有支持。虚指令采用的是 *GNU gas* 语

法，而不是 NYASM 语法。这部分请参考 *GNU gas* 的手册。

- C 语言与汇编语言的符号不需要加前缀底线。在 14 bit NY8 的 sdcc compiler，定义汇编语言函数必须加上前缀底线，才能让 C 语言調用汇编语言。而在 16 bit NY8 的 clang compiler，不需要额外的底线。

3.8 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。
- 不要使用浮点数(float)，浮点运算会消耗大量的内存，尽量使用整数运算取代浮点数。
- 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 *NYC_NY8* 不要帮您做清 0 的动作以减少耗用 ROM，从 NYIDE 的设定画面 Project Setting / Clear RAM to zero 可以控制此功能开关。
- 如果大量的全局变量初始值都是 0，使用 Clear RAM to zero 反而会节省程序空间。(大约 5 个 byte 以上)
- 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。
- 尽量给函数指定 static 属性，标示这个函数不会被外部.c 调用，能改善优化。
- 尽量搭配使用同时期推出的 *NYASM*。因为 *NYC_NY8* 产生的文件会交给 *NYASM* 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 *NYC_NY8* 可能产生了旧版 *NYASM* 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。

3.9 常见问题

Q1: 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 指令來清除 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

Q2: 操作 INTE2 寄存器的程序出现错误信息 Use BSR instruction to clear interrupt flag may cause other interrupt flags accidentally cleared if other interrupts are issued immediately after.

A:

8 個位元的 INTE2 寄存器分成兩組，高位元 INTE2[7:4]是中断旗标，低位元 INTE2[3:0]是中断功能打開设定。如果对 INTE2 寄存器使用 &= 或 |= 运算，C 编辑器将会产生 BCR 或 BSR 指令，这两个指令于芯片内部并非一个指令周期，在设置时当有中断进来而中断旗标举起时，值才被设置下去，使得中断旗标被清除而有机会发生漏掉中断的状况。建议对存取 INTE2 寄存器用以下两种方式：

1. 清除中断旗标的时候请直接设定完整 8 位的值，只清除要清除的中断旗标，并将其他的中断旗标设为 1。以下范例为 INTE2 仅有 bit 4 为 T3IF 和 bit 0 为 T3IE，欲清除 T3IF：

```
INTE2 = (unsigned char)((C_INF_TMR3^0xF0) | C_INE_TMR3);
```

此种方式会产生较为精简的汇编程序

```
MOVIA    0xE1
MOVAR    _INTE2
```

2. 如果在清除中断旗标的时候，不确定其他位的状况，可单独设置一个位。例如，欲清除 T3IF 请使用 INTE2bits.T3IF：

```
INTE2bits.T3IF = 0;
```

此种方式会让编译程序产生较为复杂的指令，确保除了 T3IF 之外的其余位保留原本状态。

```
MOVR    (_INTE2bits + 0),W
ANDIA    0xef
IORIA    0xe0
MOVAR    (_INTE2bits + 0)
```

Q3: 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。

Q4: 我注意到 C:\Nyquest\NYC_NY8\include\ny8a054a.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 NYC_NY8 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。

Q5: 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;
```

Q6: 连续的等于‘=’赋值，与多个独立赋值所产生的汇编语言不相同？

A:

没错，不相同。如果是设定初值，建议单独设定。

连续的设值，会从最尾端开始执行，并重新读取值之后设定给下一个目标寄存器。如此将会产生较为繁琐的汇编程序。例如下面这段程序，建议使用第一行，而不是第二行。

```
PA0 = 1; PB2 = 1;
PA0 = PB2 = 1;
```

Q7: INTE2 = ~(0x01) 出现警告信息 overflow in implicit constant conversion

A:

加上型态转换，INTE2 = (unsigned char) ~(0x01) 消除警告信息。

因为数学运算的反向 0x01，会得到 int 型态 0xFFFE (16 位)，将 16 位指定给 8 位的 INTE2，会自动舍去高位，并同时产生警告信息。明确的型态转换可以消除这个警告信息。

Q8: 警告信息 conditional flow changed by optimizer

A:

这通常是条件判断的条件有问题，比如说下面的程序就会产生这个警告，并且在产生警告之后，这整段 C 程序都不会产生 asm 程序。

```
if ((g1 & 0x00) == 0)
{
  /* nothing */
}
else
{
  g1++;
}
```

因为 compiler 觉得这段程序没有意义。(任何变量 AND 0 一定是 0，拿去判断是否等于 0 将永远成立)

Q9:多個 byte 合併進行運算的方法

A:

4 个 8 位变量组合为 32 位的 long 数据型态，不建议使用左移运算，因为会消耗较多的 ROM。下面列出两段程序，第一段是不建议的方式，第二段则是建议的方式。

```
unsigned char a,b,c,d;
```

```
unsigned long e;
unsigned long result;
void func(void)
{
  e = ((unsigned long)a << 24) | ((unsigned long)b << 16) |
      (c << 8) | d;
  result += e;
}
```

建议使用 union 定义 8 位与 32 位重迭的数据结构，省略左移运算与 OR 运算。范例如下面的程序:

```
typedef union long_byte_t {
  unsigned long l32;
  unsigned char l8[4];
} long_byte_t;

unsigned char a,b,c,d;
long_byte_t e;
unsigned long result;
void func (void)
{
  e.l8[0] = a; e.l8[1] = b; e.l8[2] = c; e.l8[3] = d;
  result += e.l32;
}
```

Q10: 兩種 compiler: sdcc、clang 使用上的差异

A:

14 bit NY8 使用 sdcc compiler，16 bit NY8 使用 clang compiler。在标准的 C 语言都是支持 C99 的语法规范。不同之处在于规范之外的扩充语法。底下列出几个不同之处:

功能	sdcc	clang
Interrupt	__interrupt(0)	Interrupt
混用 Asm 文件语法	NYASM 语法	GNU gas 语法
单一 bit 定义	struct 或 __sbit	struct 或指标转型
参数传递	A + STK00~STK12	__rc0 ~ __rc15
SFR 定义方式	外部符号	常数地址做解指针
Rolling Code 定位方式	__at	SET_ROLLING_CODE_ADDR

对参考(&)做解指标(*)	不行	可以
存取绝对地址内存	__at(0x60)	*(volatile char*)0x60

Q11: 找不到反组译的输出.lst 文件

A:

首先确认 NYIDE 的 project setting 在 linker 页面有勾选「Generate listing file」。再来是文件输出的目录在某些 IC、某些 NYIDE 版本是不一样的，有些在 bin 文件的旁边，有些在 OBJ 目录立面，有些在 build 目录。这些地方都找看看。

Q12: NY8 16 bit 在 OBJ 目录没有看到.s 文件

A:

可以在 NYIDE 对.c 文件右键单击，点选编译单档，可以产生.S 文件。额外提醒:副档名大小写是有区别的，大写.S 代表文件需要经过 C 语言前处理器(preprocessor)，而小写.s 则不会。更实际的差别在于大写.S 可以使用#define，而小写.s 则不行。

Q13: NY8 16 bit 的 header file 在哪里?

A:

NY8 14 bit 放在 C:\Nyquest\NYC_NY8\include\ny8.h

NY8 16 bit 放在 C:\Nyquest\NYC_NY8\llvm\include\ny8.h

另外还有一包 ny8_constant.h 放在 NYIDE 或是 NY8 Example Code 的安装目录。

4 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 20
1.3	2019/05/24	1. 增加 EEPROM 说明。 2. 增加强制指定函数地址说明。	11 15
1.4	2020/03/03	增加常见问题。	28
1.5	2020/08/18	1. 说明系统保留内存分配规则。 2. 增加使用建议。 3. 增加常见问题。	14 19 20
1.6	2022/02/14	1. 增加 multi_16b 函数说明。 2. Optimization 选项说明改名，原本为 Bank Select Optimize。	12 13
1.7	20022/09/13	1. 系统需求增加 Win11。 2. 增加常见问题项目。	3 23
1.8	2022/11/28	1. 删除 Reserved RAM Size，新版不需要此设定。 2. 增加常见问题项目。	- 23
1.9	2023/02/15	1. 增加内崁汇编语言区块。 2. 修正强制指定内存地址的说明。 3. 增加使用建议。	9 14 20
2.0	2023/08/23	增加 clear_ram 内建函数说明。	13
2.1	2024/08/23	1. EEPROM API。 2. Reserved RAM for Interrupt 最大值。	12 15

版本	日期	内 容 描 述	修正页
2.2	2025/05/26	增加 NY8 16 bit 系列说明。	7, 12, 20, 22, 23, 27