九齊科技股份有限公司
Nyquest Technology Co., Ltd.

使用手冊

NYC for NY8 Series

C Compiler

Version 2.1

Aug. 23, 2024

目　錄

1 簡介 4

1.1 如何使用本手冊 4

1.2 系統需求 4

1.3 安裝 *NYC_NY8* 4

2 使用 *NYC_NY8* 5

2.1 透過 *NYIDE* 使用 *NYC_NY8* 5

2.1.1 建立新專案 5

2.1.2 建置 5

3 語法與使用 6

3.1 標準 C 語法 6

3.1.1 註解 6

3.1.2 資料型態 6

3.2 擴充語法 7

3.2.1 保留字 7

3.2.2 中斷（interrupt） 7

3.2.3 暫存器位址定義 8

3.2.4 暫存器位元定義 8

3.2.5 內嵌組合語言（Inline assembly） 9

3.2.6 內嵌組合語言區塊（Inline assembly block） 10

3.2.7 指標屬性 10

3.3 系統標頭檔 11

3.3.1 特殊指令巨集 11

3.3.2 系統暫存器定義 11

3.3.3 ROM 資料讀取 12

3.3.4 EEPROM 資料存取 12

3.3.5 內建函數 multi_16b 13

3.3.6 內建函數 clear_ram 14

3.4 選項 14

3.5 開發流程 15

3.6 進階使用 16

3.6.1 強制指定記憶體位址 16

3.6.2 強制指定函數位址 17

3.6.3 混合使用 C 與組合語言 17
3.7 使用建議 21
3.8 常見問題 22
4 改版記錄 27

1 簡介

NYC_NY8 為針對九齊科技的 NY8 系列 8 位元 MCU IC 而提供的 C 語言編譯器（Compiler）。它被上層開發工具軟體 *NYIDE* 所呼叫以編譯 C 程式，並結合 *NYASM* 組譯器（Assembler）進一步組譯及連結目的檔來產生.bin 檔，然後.bin 檔可下載到板子或燒錄到 OTP IC。

1.1 如何使用本手冊

1. 簡介

為何需要 *NYC_NY8* 與安裝 *NYC_NY8* 的基本需求。

2. 使用 *NYC_NY8*

如何透過 *NYIDE* 使用 *NYC_NY8*。

3. 語法與使用

介紹 *NYC_NY8* 的語法與使用方式。

1.2 系統需求

底下列出使用 *NYC_NY8* 的系統需求。

- Pentium 1.3GHz 或更高級處理器，Win7、Win8、Win10、Win11 作業系統。
- 至少 2G SDRAM。
- 至少 2G 硬碟空間。

1.3 安裝 *NYC_NY8*

請聯繫九齊科技來取得 *NYC_NY8* 的安裝程式檔，雙擊執行後進入安裝程式嚮導，然後依照畫面指示將可輕易完成安裝流程。

2 使用 *NYC_NY8*

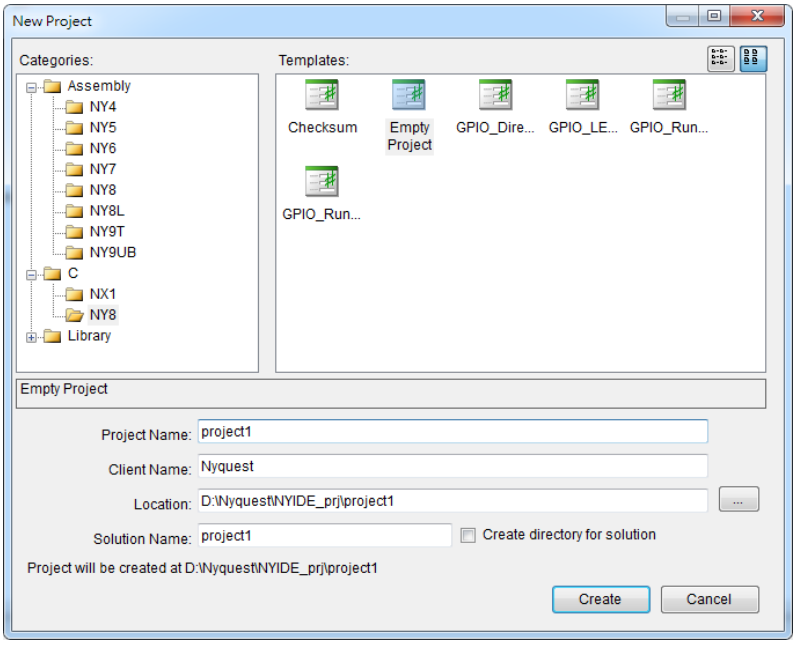
當使用者使用 NY8 軟體開發工具 *NYIDE* 編寫 C 程式後，在 *NYIDE* 介面上按下 Build 時，軟體開發工具會自動尋找並使用已安裝於電腦上的 *NYC_NY8* 作編譯。底下將說明透過 *NYIDE* 使用 *NYC_NY8* 的流程。

2.1 透過 *NYIDE* 使用 *NYC_NY8*

NYIDE 為九齊科技提供以開發 NY4 / 5 / 6 / 7 / 8 / 9T / 9UB / NX1 系列微控制器程式的整合性開發工具，主要目的為提供使用者以組合語言（Assembly）和 C 語言來編寫程式，並擁有建置和強大的除錯功能。當使用 *NYIDE* 開發 NY8 專案，在建置和除錯時，*NYIDE* 會自動尋找並使用安裝於電腦上的 *NYC_NY8* 工具鏈。底下簡單的介紹使用 *NYIDE* 開發 NY8 專案。更詳細的操作方式，請參考 *NYIDE* 使用手冊。

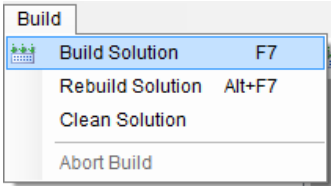
2.1.1 建立新專案

開啟 *NYIDE*，選擇建立新專案。在 New Project 視窗內，左邊的 Categories 選擇 C，然後選擇 NY8。指定專案名稱及類型後，按下 Create。*NYIDE* 將會自動產生必要檔案，專案已於可建置狀態。



2.1.2 建置

在 *NYIDE* 主畫面上的功能表選擇 Build / Build Solution 進行建置(或按下快捷鍵 F7)，即會呼叫 *NYC_NY8* 執行建置動作。若建置成功會在專案目錄產生.bin 檔案，以進一步提供下載或燒錄。



3 語法與使用

NYC_NY8 支援標準的 ANSI C89 語法，並且針對 NY8 系列 IC 新增了一些特殊語法。

3.1 標準 C 語法

NYC_NY8 支援標準的 ANSI C89 語法，有關詳細語言定義請參考：Standard ISO/IEC 9899 (http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899)

3.1.1 註解

註解支援兩種格式，以雙斜線起頭的單行註解，以及/*起頭，至*/結尾的多行註解。

範例：

```
// single line comment

/*
Multi line comment
*/
```

3.1.2 資料型態

底下表格列出*NYC_NY8*所使用的基本資料型態及允許的資料範圍。其中 stdint 型態必須先行引用 stdint.h 方可使用。

型態	stdint	長度	範圍
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 擴充語法

3.2.1 保留字

底下列出所有保留字，使用者定義之符號不可和保留字相同。

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 中斷（interrupt）

中斷服務副程式在 NY8 系列又分為硬體中斷與軟體中斷，位址分別必須在於 0x08 與 0x01，在 C 語言中必須增加屬性至函數定義，__interrupt(0)代表硬體中斷服務程式、__interrupt(1)代表軟體中斷服務程式。編譯器會將此段程式安排在指定的位址，例如硬體中斷必須在位址 0x08。編譯器會在進入函數前自動的保留當前系統狀態，例如 ACC 暫存器、Status 暫存器、FSR 暫存器，並且在離開中斷服務程式時自動還原狀態。

範例：

```
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 暫存器位址定義

所有支援 NY8 IC 的特殊暫存器都已經被定義在程式安裝資料夾的 include 目錄下，檔名為 IC 名稱。建議使用者直接使用該標頭檔，不須自行定義特殊暫存器。

3.2.4 暫存器位元定義

__sbit 關鍵字可以將 8 位元暫存器的其中一個位元定義為新的變數。語法為：

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

__sbit 僅可連結至已存在的 8 位元變數其中某個位元，而無法獨立佔用新的記憶體空間。下面的程式範例示範如何使用 sbit 定義兩個旗標，flag1 連結到 myvar 的第 0 個位元，flag2 連結到 myvar 的第 3 個位元（可選用的位元為 0 到 7）。由 sbit 定義的變數只有單一位元，可設定的值只有 0 和 1，讀取的結果亦只有 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;
}
```

NYC_NY8 1.10 開始支援此語法，在此之前的版本並無支援。在先前的版本，必須建立獨立 bit 定義的 struct，如下列範例所示：

```
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 內嵌組合語言（Inline assembly）

在 C 語言之中可以內嵌組合語言，使用__asm__關鍵字即可插入任意的組合語言程式。下面的程式片段示範了內嵌組合語言的寫法，編譯器會將當前的位址存入 STK00 及 ACC 暫存器，並直接跳躍至另一個函數。

```
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 內嵌組合語言區塊（Inline assembly block）

前一個範例的程式可以改寫為組合語言區塊，用 __asm……__endasm; 包住整個組合語言程式區塊。必須特別注意，結尾的__endasm; 有分號。

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

3.2.7 指標屬性

__code 和 __data 用於指定指標存放於 ROM 或 RAM。一般的指標佔用 3 bytes，其中 2 bytes 存放位址，1 byte 存放指標型態以區分指標指向的位置是 ROM 或是 RAM。當編譯器掌握足夠資訊能判斷指標型態時，可以省略指標型態的 1 byte。例如底下範例程式中的 data 為存放在 ROM 的資料，ptr1 及 ptr2 都是指向 data 的指標。然而 ptr1 有加上 __code 屬性，編譯器可以確定此指標只會指向 ROM 的資料，編譯器實際產生的機械碼 ptr1 佔用 2 bytes，而 ptr2 佔用 3 bytes。使用指標時，若明確知道此指標只會指向 ROM 或 RAM，盡量事先指定 __code 或 __data 屬性，以節省 RAM 使用量，亦可產生較為精簡的指令。

```
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 系統標頭檔

在 NYC_NY8 安裝目錄中的 include 資料夾有所有 C 語言使用的標頭檔（header file），本章節介紹這些標頭檔的內容及使用方法。

3.3.1 特殊指令巨集

ny8common.h 檔案定義了常用的特殊組合語言指令巨集，以較為低階的方式控制 IC 行為，使用者可在適當的時機呼叫這些巨集。

巨集	說明
ENI()	開啟中斷功能
DISI()	關閉中斷功能
INT()	引發軟體中斷
CLRWDT()	清除 watch dog 計時器
SLEEP()	進入 sleep
NOP()	空指令 nop

3.3.2 系統暫存器定義

ny8.h 會根據所選的 IC 自動引用該 IC 專用的標頭檔，所有 IC 支援的特殊暫存器都會定義在與 IC 同名的標頭檔。特殊暫存器又分為四種：general page 在宣告時加上屬性 __sfr，F-page 在宣告時加上屬性 __fpage，S-page 在宣告時加上屬性 __spage，T0MD 在宣告時加上屬性 __t0md。

在 C 語言層級，這些暫存器並沒有任何分別。但使用者仍然必須知道，這些暫存器實際存取的組合語言代碼並不相同。只有 general page 暫存器可以直接存取，像是直接設定某個位元的值，或是直接對暫存

器做互斥或(XOR)計算。除了 general page 以外的特殊暫存器無法直接存取，底層的組合語言必須將特殊暫存器的值搬移到 ACC 暫存器，方能繼續接下來的運算。

對於 general page 的特殊暫存器，我們鼓勵使用者單獨的設定位元為 1 或 0。而對於其他的特殊暫存器，則建議直接設定完整 8bit 的值。遵循這樣的規則可得到較佳的機械碼。

建議使用 ny8.h 而非直接使用 IC 專用標頭檔，可以減少更換 IC 造成標頭檔與函數庫之間的不一致。此檔案在 *NYC_NY8* 1.10 開始提供，使用者使用古老版本的 *NYC_NY8* 時，必須注意切換 IC 之後必須同時更換標頭檔的引用。

3.3.3 ROM 資料讀取

ny8_romaccess.h 定義了讀取 ROM 資料的函數。

NY8 的 ROM 每個 word 為 14-bit，以一般 C 語言的指標僅能讀取 14bit 中的低 8bit，使用 ny8_romaccess.h 之中定義的 read_14bit_rom 函數則可讀取完整的 14 bits。

範例：

```
#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);
```

詳細範例可參考 *NYIDE* 所內附的範例程式：Checksum。

3.3.4 EEPROM 資料存取

部份 IC 具有內建的 EEPROM，必須使用特殊指令存取。NYC_NY8 提供了存取 EEPROM 的函數可以直接呼叫。

ny8_eeprom.h 內定義了存取 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	從指定位址讀取一個 byte
eeprom_write	寫入一個 byte 到指定位址(不再提供)
eeprom_write_timeout	寫入一個 byte 到指定位址
eeprom_protect_lock	鎖定/解除鎖定 EEPROM 防寫保護
eeprom_protect_unlock	同上

- unsigned char eeprom_read (unsigned char address)

  參數 address 指定要讀取的 EEPROM 位址。

回傳值為從指定位址讀取的一個 byte 的資料。

- void eeprom_write (unsigned char address, unsigned char value)
  參數 address 指定要寫入的 EEPROM 位址。
  參數 value 接受一個 byte 的資料，此資料將會寫入到指定位址。

  這個 API 在 NYC_NY8 1.70 移除，請改用具備 Timeout 參數的 eeprom_write_timeout。

- void eeprom_write_timeout (unsigned char address, unsigned char value, unsigned char timeout)
  參數 address 指定要寫入的 EEPROM 位址。
  參數 value 接受一個 byte 的資料，此資料將會寫入到指定位址。
  參數 timeout 接受一個 byte 數值，代表本操作逾時的時間限制，每個 ic 可用的數值與對應的時間皆不同，請參考 ic datasheet。
  使用 eeprom_write 之前，必須先解除 EEPROM 的防寫保護。
  這個 API 新增於 NYC_NY8 1.43。

- void eeprom_protect_lock (void)
  鎖定/解除鎖定 EEPROM 防寫保護，依據 config block 選擇 EEPROM Write Mode 而有所不同。一、單次寫入模式（One Byte），每次呼叫此函數會解除 EEPROM 的防寫鎖定，之後呼叫 eeprom_write 完成寫入，硬體將會自動開啟防寫鎖定。在單次寫入模式，使用者必須在每一個寫入動作之前執行解鎖。二、在連續寫入模式（Continuous Write），第一次呼叫 eeprom_protect_lock 會解除 EEPROM 的防寫鎖定，後續可以執行任意數量的 eeprom_write。第二次呼叫 eeprom_protect_lock 重新開啟防寫鎖定。在連續寫入模式，使用者必須在所有寫入完成後自行呼叫函數鎖定。

- void eeprom_protect_unlock (void)
  與 void eeprom_protect_lock 完全相同的程式，使用不同的命名。同時呼叫 eeprom_protect_unlock 與 eeprom_protect_lock 會使用相同的程式空間，並不會額外消耗 ROM。

範例：

```
#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 與 eeprom-continuous-write。

3.3.5 內建函數 multi_16b

普通 C 語言的乘法運算輸入與輸出的資料型態必須相同。兩個 16 位元整數相乘，只能得到 16 位元的結

果。如果希望得到 32 位元的結果，必須將輸入資料轉型為 32 位元(long)。內建函數 multi_16b 是特殊的乘法函數，輸入為兩個 16 位元正整數，輸出為 32 位元正整數。對於 ROM、RAM 的資源消耗介於 16 位元乘法到 32 位元乘法之間。注意，multi_16b 無法計算負數。NYC_NY8 1.43 開始支援此函數。

範例：

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

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

3.3.6 內建函數 clear_ram

清除 ic 所有 RAM 為 0，不僅限於 C 語言宣告的變數，包含使用者程式並未使用的 RAM、以及 Compiler 所產生的暫存變數都會設定為 0。特殊功能暫存器(SFR)則不會被改變。實際的程式邏輯與 NYIDE project setting 勾選 Clear RAM to zero 相同。兩者不同之處在於執行時機，Clear RAM to zero 只會在進入 main 函數之前執行一次，而 clear_ram 可以在任何時間手動執行。這個函數在不同 ic 會自動連結正確的程式，確保設定的 RAM 範圍符合 ic 規範。NYC_NY8 1.60 開始支援此函數。

函數原型宣告:

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

3.4 選項

使用 *NYIDE* 開發 C 語言專案，可以設定若干專案建置選項。這些選項可以控制編譯器、組譯器、連結器的行為，點選功能表的專案（Project）/專案設定（Project Settings）可開啟設定界面。

- 僅允許使用 RAM Bank0（Use RAM Bank0 only）：勾選此項僅能使用 Bank0 的記憶體，產生的 Code size 較小，部份母體僅有單一 Bank，此選項將強制勾選。不勾選將會在存取記憶體之前插入切換 bank 指令，允許使用所有的記憶體，但產生的 Code size 會較大。
- 開機清空記憶體（Clear RAM to zero on startup）：開機時先清空所有的記憶體，才執行 main 函數。而全域有初值變數並不受此選項影響，無論此選項勾選與否，有初值的全域變數會在進入 main 函數之前完成初值設定。取消此選項可以減少 Code size，但使用者必須自行初始化全域無初值的變數，因為開機時的記憶體內容為不明。
- 產生組合語言列表檔（Generate ASM listing file）：組譯完成產生列表檔，檔名為 *.lst。不勾選此選項可加快編譯速度。

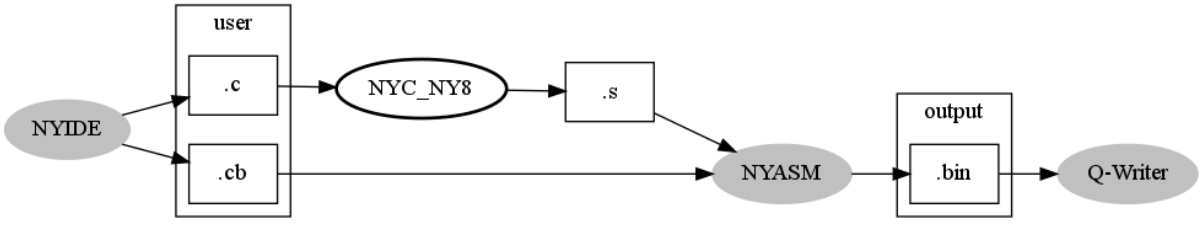
- 產生連結列表檔（Generate listing file）：連結後產生列表檔，檔名為 *.link.lst。此檔為最終.bin 檔案的反組譯結果，不勾選此選項可以加快編譯速度。
- 產生位址對應檔（Generate map file）：連結後產生位址對應檔，檔名為 *.map。此檔包含位址分配資訊，不勾選此選項可加快編譯速度。
- 最佳化（optimization）：可以選擇 Level 1 到 Level 3 的最佳化等級，高等級的最佳化可以得到更精簡的程式。但是必須注意，此選項跟內嵌組合語言（inline assembly）的搭配可能產生異常。
- 中斷服務程式保留記憶體大小（Reserved RAM for interrupt）：保留給中斷服務程式（ISR）所使用的記憶體大小，在進入中斷前保存當前函數的變數狀態。當在使用陣列或呼叫函數而中斷發生都有可能打斷目前正在運算的暫存器，因此如果中斷會造成行為不對，就需設定編譯器將運算被打斷的變數先儲存起來，依所用到的變數大小來設定需要保留給編譯器作備份的記憶體大小：最小值為 0，不保留任何函數呼叫的狀態；最大值為虛擬堆疊大小 13。設定值越大，會造成中斷服務程式進入時間拉長，因為必須使用更多的指令來備份當前狀態。實際指令數量因為備份用的記憶體位於不同的 Bank 而有少許不同，請參考下表。

保留大小	進入中斷前額外指令	備註
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

- 引用檔案路徑（Include path）：設定 C 語言 include 關鍵字引用標頭檔的搜尋路徑。預設的路徑為專案根目錄及 *NYC_NY8* 安裝目錄的 include 資料夾。使用者可以增加自定義的路徑。

3.5 開發流程

使用 *NYIDE* 編寫 C 語言程式，並設定專案所需的組態檔.cb，*NYIDE* 建置時自動呼叫 *NYC_NY8* 產生組合語言檔案.s，再呼叫 *NYASM* 組譯組合語言檔案與組態檔，產生最終.bin 檔。最後可以藉由 *Q-Writer* 將.bin 檔燒錄至 IC。



3.6 進階使用

本章節說明一些進階使用的方式。

3.6.1 強制指定記憶體位址

一般 C 語言的變數並不會指定記憶體位址，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變數存放位址，在變數型態之前加上__at(addr)即可，addr 為指定的位址。

範例：

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

請注意，變數不應該宣告在 SFR 所定義的位置，如果想要存取 SFR，請使用所選 IC 對應的標頭檔(Header file）預定義的變數。因為專案建置過程中，會連結 *NYC_NY8* 內建的靜態函式庫（static library），函式庫使用的是在內建表頭檔中預先宣告的 SFR，若使用者改變了 SFR 定義，將會造成專案建置失敗。如果想要重新命名 SFR，請使用#define 預處理指令。

範例：

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

而使用者有多個.c 檔時，也必須注意相似的狀況。只能在其中一個.c 實際佔用記憶體，其他的.c 必須使用 extern 關鍵字定義該變數為外部。

範例：

File: main.c

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

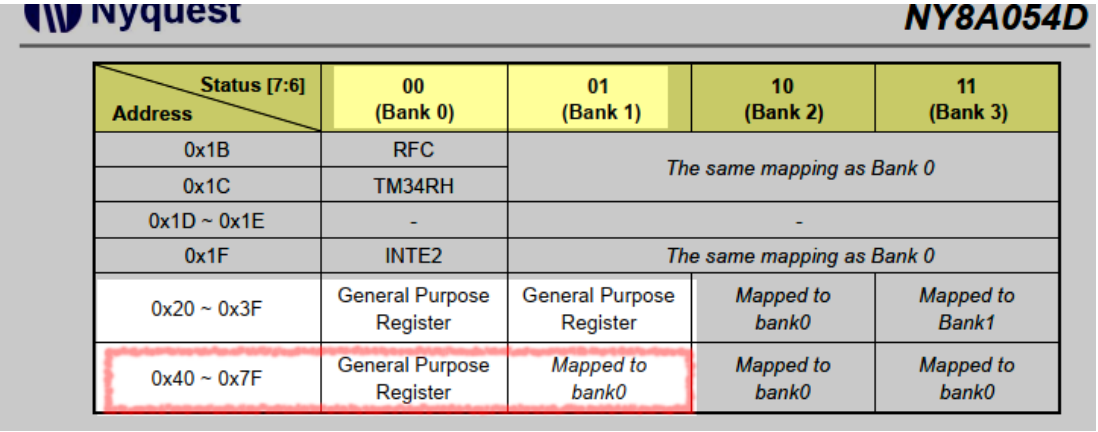
File: my_var.h

```
#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
```

強制指定位址必須注意專案設定選項的保留記憶體大小（Reserved RAM size），必須保留足夠的 share bank 給系統使用。



Nyquest NY8A054D

Address \ Status [7:6]	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*

上圖是 NY8A054D datasheet 18 頁，關於 R-Page address mapping 的部份。紅色框框部份，無論 Bank0 或 Bank1 皆能存取相同的記憶體。系統保留記憶體必須位於紅框之內(0x40~0x7F)。

3.6.2 強制指定函數位址

一般來說，C 語言的函數並不需指定起始位址，由連結器（linker）將函數自動安排在適當的位置。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數起始位址。在函數回傳型態之前加上__at(addr)即可，addr 為指定的位址。

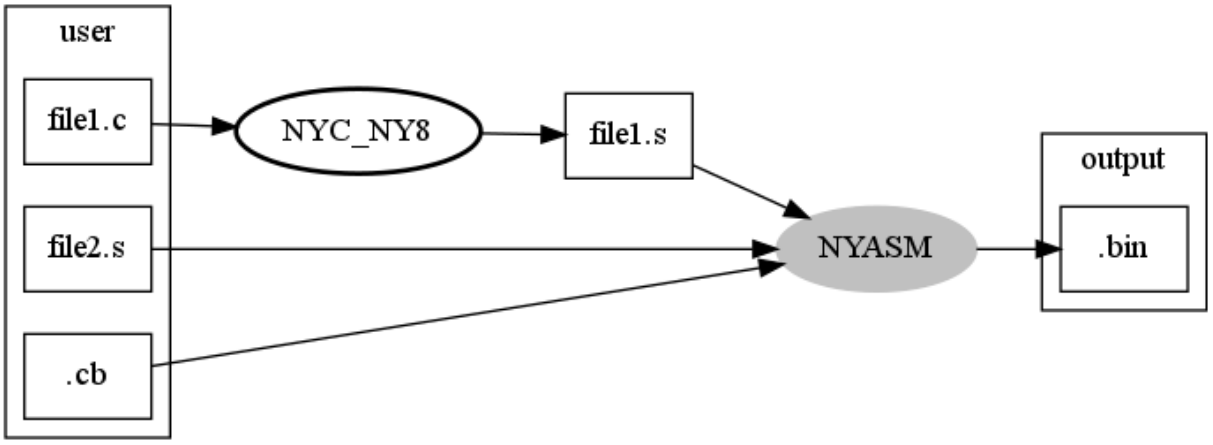
範例：

```
__at(0x110) void func(void)
{
    //......
}
```

強制指定函數位址通常用於二次燒錄來更正程式的錯誤。請不要指定位址 0x00 給函數，因為位址 0x00 已經被 NYC_NY8 的啟動程式佔用。

3.6.3 混合使用 C 與組合語言

在開發流程我們可以看到，*NYC_NY8* 將.c 檔案編譯轉換為.s 檔案，之後由 *NYASM* 將.s 檔案與.cb 檔案整合為.bin 檔案。然而，.c 檔案可以不只一個，對應產生的組合語言.s 檔案也可以不只一個。使用者可以自行編寫部份組合語言.s 檔案，與 *NYC_NY8* 所產生的.s 檔案協同運作。在這將會介紹如何自行編寫與 *NYC_NY8* 配合運作的.s 檔案。



先從簡單的範例開始——Rolling code 應用。Rolling code preset mode 應用必須將 ROM 的 0xE 與 0xF 留空白，待燒錄時實際寫入滾碼，在編譯的期間，0xE 與 0xF 不可有值。然而在 C 語言我們很難控制 IC 的某一位址必須填入什麼值，除了__interrupt 關鍵字強制程式放在 0x1 或 0x8。解決方法是使用組合語言與 C 語言協同運作，底下將示範如何使用組合語言將 0xE 與 0xF 位址的值保留空白，在測試時則填入想要測試的值，並且用一段 C 語言程式讀取 Rolling code 做驗證。

範例檔案共有三個：

- rom.s 將 0xE 及 0xF 位址填入 NOP，測試時填入 0x255、0x3AA 測試資料，並且匯出符號 ___rolling_code_addr 供 C 語言使用。
- rom.h 定義外部符號__rolling_code_addr。
- main.c 主程式，讀取 rolling code 並做驗證。

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 檔案可以看到，匯出的外部符號名稱為___rolling_code_addr，有三個底線。當 C 語言被編譯為組合語言時，所有的符號都會被加一個底線，反過來說，我們希望組合語言的符號被 C 語言使用時，人為多加上一個底線以作區別。

使用組合語言，可以直接指定資料擺放的位置，透過 ORG 指令可以輕易作到這件事情。

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，一個外部符號__rolling_code_addr 的定義，注意這邊使用了__code 關鍵字明確的定義這個符號存在於 ROM。這符號名稱的前綴底線只有兩個，因為 C 語言編譯為組合語言時會自動在前面加上一個底線。

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 所定義的符號__rolling_code_addr 抓取 ROM 資料，當然也可以選擇不使用此符號，改為直接指定 0xE 位址，但這樣 rolling code 換位置的時候就需要作較多的更動：必須更換 rom.s 裡面 org 指令所指定的位址，以及 main.c 讀取的位址。

接著我們再看範例 main.c，讀取 ROM 資料所使用的函數 read_14bit_rom 是內建在 library 裡面，在 ny8_romaccess.h 有著原型宣號，然而它的實做並不是 C，而是組合語言。這邊一併將之列出，順便用這個內建函數來說明如何從 C 語言呼叫組合語言所定義的函數。

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
```

上面這兩個檔案我們可以看到 C 語言的定義，以及組合語言的實做。第一個注意的是符號名稱的差異，在 C 語言叫做 read_14bit_rom，而組合語言則命名為_read_14bit_rom，多了一個底線。原因如同前面所講的，C 語言在翻譯成組合語言之後，所有符號都會增加前綴底線。這個函數一個輸入參數，為要讀取的 ROM 位址指標，並且有一個回傳值型態為 int（16-bit）。ROM 位址指標實際上佔用 16 bits，兩個 8-bit 暫存器。參數的傳遞優先使用 ACC，再來則是 STK00 到 STK12 公用暫存器。

以這個例子的 16-bit 指標為例，高位 8-bit 會被存放於 ACC，低 8-bit 會存放於 STK00。所以組合語言實做此函數的第一步，就是將目前 ACC 存放的 ptr[15:8]移動到 TBHP 暫存器，將 STK00 存放的 ptr[7:0]移動到 ACC。

回傳值的存放也是相同邏輯，高位放在 ACC，低位放在 STK00。當 TableA 完成 ROM 資料讀取，ACC 存放的是 ROM[7:0]，我們立刻將 ACC 移動到 STK00，並將 TBHD 所存放的 ROM[13:8]移動到 ACC。最後加上 ret 返回這個函數。

對 main.c 而言，並不會在意 read_14bit_rom 究竟是用 C 寫的，亦或組合語言。只要輸入參數與輸出的回傳值格式符合規範，就能完美的互助合作。

3.7 使用建議

底下提出一些開發 C 語言專案的建議。

- 盡量使用無符號（unsigned）變數，在部份的運算不用判斷正負號會比較快。
- 運算式之中不要交互使用常數與變數，將常數集中才能有效的最佳化。
  例如 1 + a + 2 就是不好的寫法，1 跟 2 無法在編譯時期計算。建議寫成 a+1+2，如此一來 1+2 可在編譯時期計算，執行期間只需要計算 a+3 即可。

- 不要使用浮點數(float)，浮點運算會消耗大量的記憶體，盡量使用整數運算取代浮點數。
- 使用 `if (INTFbits.T0IF)`取代 `if (INTFbits.T0IF == 1)`，可以得到較為精簡的程式。
- 不要連續單獨設定 S-Page / F-Page 暫存器的某個 bit。
  因為 S-Page / F-Page 暫存器的讀寫皆需透過特別的指令，連續的單獨位元設定，會不斷的讀取、寫入這些特殊暫存器，而不像 R-Page 的暫存器可使用 BCR / BSR 指令設定單獨位元。建議使用 S-Page / F-Page 暫存器時，先準備好所要設定的值，一次完成 8 個位元的設定。
- 如果確定使用全域變數之前都有設定初值，可以指定 *NYC_NY8* 不要幫您做清 0 的動作以減少耗用 ROM，從 NYIDE 的設定畫面 Project Setting / Clear RAM to zero 可以控制此功能開關。
- 如果大量的全域變數初始值都是 0，使用 Clear RAM to zero 反而會節省程式空間。(大約 5 個 byte 以上)
- 如果 RAM 的使用量不大，可試著使用 small model，關閉 bank 切換。這樣可以產生較為精簡的程式。
- 不要將程式拆分為過多的.c 檔案。這會影響最佳化，增加 RAM 的使用量。因為編譯器沒辦法假設兩個函數不會同時執行，只能分配獨立的記憶體給彼此。
- 盡量給函數指定 static 屬性，標示這個函數不會被外部.c 呼叫，能改善最佳化。
- 盡量搭配使用同時期推出的 *NYASM*。因為 *NYC_NY8* 產生的檔案會交給 *NYASM* 繼續下一個步驟，若兩者版本差距過大，或許會有不相容的情形。例如 *NYC_NY8* 可能產生了舊版 *NYASM* 不支援的指令。
- 若已知指標只會指向 ROM 或 RAM，宣告時使用指標屬性__data 及__code 告知編譯器。

3.8 常見問題

Q1: 開啟多個中斷源時為什麼有時會漏掉中斷？

A:

以同時開啟 PortB change 中斷與 Timer1 中斷為例，使用 bit clear 指令來清除 T1IF 有可能會誤清 PBIF，建議寫立即值的方式針對 T1IF 寫 0 來清除。以下為詳細說明：

使用位元 clear 指令（read modify write 指令）清除 T1IF（Timer1 interrupt flag）時，IC 會進行兩個步驟：

1.1 先讀取“INTF”所有位元。

1.2 將 T1IF 位元清為 0，其他位元寫 1，先前讀取的值會回到“NTF”暫存器中。

但如果在“1.1”與“1.2”之間，PBIF 位元因發生 PortB change 中斷而被設為 1，那在“1.2”時就會被誤清為 0，造成 PortB change 中斷偶而會被忽略。

請參考下面程式碼來清除 T1IF（Timer1 interrupt flag）。

建議指令碼	不建議指令碼
`INTF = 0xF7;` 或 `INTFbits.T1IF = 0;`	`INTF &= 0xF7;`
產生組合語言	產生組合語言
`MOVIA 0xf7` `MOVAR _INTF`	`BCR _INTF, 3`

Q2: 操作 INTE2 暫存器的程式出現錯誤訊息 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 暫存器分成兩組，高位元 INTE2[7:4]是中斷旗標，低位元 INTE2[3:0]是中斷功能開啟設定。如果對 INTE2 暫存器使用 &= 或 |= 運算，C 編輯器將會產生 BCR 或 BSR 指令，這兩個指令於芯片內部並非一個指令週期，在設置時當有中斷進來而中斷旗標舉起時，值才被設置下去，使得中斷旗標被清除而有機會發生漏掉中斷的狀況。建議對存取 INTE2 暫存器用以下兩種方式：

1. 清除中斷旗標的時候請直接設定完整 8 位元的值，只清除要清除的中斷旗標，並將其他的中斷旗標設為 1。以下範例為 INTE2 僅有 bit 4 為 T3IF 和 bit 0 為 T3IE，欲清除 T3IF：

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

此種方式會產生較為精簡的組合語言程式

```
MOVIA    0xE1
MOVAR    _INTE2
```

2. 如果在清除中斷旗標的時候，不確定其他位元的狀況，可單獨設置一個位元。例如，欲清除 T3IF 請使用 INTE2bits.T3IF：

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

此種方式會讓編譯器產生較為複雜的指令，確保除了 T3IF 之外的其餘位元保留原本狀態。

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

Q3: 在 main loop 及 interrupt service routine 皆有操作 Array 的程式，資料偶爾會讀寫到錯誤的位址？

A:

因為 Array 操作會使用到共用的系統暫存器，如果在操作到一半進入中斷，且中斷服務程式內也有操作 Array 的行為，則共用的系統暫存器狀態將會被破壞，造成讀寫位址錯誤。

建議在這種情形使用 DISI()及 ENI()控制中斷禁用啟用，防止 Array 操作過程進入中斷。

Q4: 我注意到 C:\Nyquest\NYC_NY8\include\ny8a054a.h 這類各種 IC body 的暫存器定義檔，為何修改其中的暫存器名稱後編譯總是失敗（Link fail）？

A:

暫存器的名稱不僅定義於<icbody>.h，在靜態連結函數庫也必須存在相同的定義。靜態連結函數庫位於 NYC_NY8 安裝目錄的 lib 資料夾下，檔案名稱為<icbody>.a。靜態連結函數庫為二進制檔案，無法由使用者自行修改。修改標頭檔將造成連結時無法在函數庫中找到相符的暫存器定義。

建議不要對系統內建的暫存器做重新命名。

Q5: 在中斷服務程式之中設定變數值，並在一般程式流程中讀取。讀取結果異常？

A:

中斷服務程式與正常流程共用的變數，建議在宣告時加上 volatile 關鍵字，防止變數被最佳化導致異常。用以下一個簡單的例子說明共用變數被最佳化而導致程式異常：

```
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，當 delay 函數被呼叫，會先初始目前的 count 變數為 0，count 變數因開啟 Timer 中斷而在每次的中斷遞增，然後等 count 變數的值到達 delay_count，就會結束這個函數。但實際執行時 delay 中的 while 迴圈永遠不會跳出，造成死迴圈。原因為編譯器最佳化機制認為 count 變數在設定為 0 之後並沒有作其他運算，可以用常數 0 代換 count 變數，因此 while 迴圈的判斷條件被最佳化成 `while (0 < delay_count)`，條件永遠成立造成死迴圈。解決方法是改變 count 變數的宣告，以 volatile 關鍵字告知編譯器，count 變數不可被最佳化。

```
volatile uint8_t count;
```

Q6: 連續的等於 '=' 賦值，與多個獨立賦值所產生的組合語言不相同？

A:

沒錯，不相同。如果是設定初值，建議單獨設定。

連續的設值，會從最尾端開始執行，並重新讀取值之後設定給下一個目標暫存器。如此將會產生較為繁瑣的組合語言程式。例如下面這段程式，建議使用第一行，而不是第二行。

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

Q7: INTE2 = ~(0x01) 出現警告訊息 overflow in implicit constant conversion

A:

使用者要消除警告訊息，要幫 INTE2 = ~(0x01)加上型態轉換，例如 INTE2 = (unsigned char) ~(0x01)

因為數學運算的反向 0x01，會得到 int 型態 0xFFFE (16 位元)，將 16 位元指定給 8 位元的 INTE2，會自動捨去高位元，並同時產生警告訊息。明確的型態轉換可以消除這個警告訊息。

Q8: 警告訊息 conditional flow changed by optimizer

A:

這通常是條件判斷的條件有問題，比如說下面的程式就會產生這個警告，並且在產生警告之後，這整段 C 程式都不會產生 asm 程式。

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

因為 compiler 覺得這段程式沒有意義。(任何變數 AND 0 一定是 0，拿去判斷是否等於 0 將永遠成立)

Q9: 多個 byte 合併進行運算的方法

A:

4 個 8 位元變數組合為 32 位元的 long 資料型態，不建議使用左移運算，因為會消耗較多的 ROM。下面列出兩段程式，第一段是不建議的方式，第二段則是建議的方式。

```
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;
}
```

建議使用 union 定義 8 位元與 32 位元重疊的資料結構，省略左移運算與 OR 運算。範例如下面的程式:

```
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;
}
```

4 改版記錄

版本	日期	內 容 描 述	修正頁
1.0	2017/08/14	新發佈。	-
1.1	2017/10/27	增加常見問題。	17
1.2	2018/05/30	1. 增加 sbit 語法。 2. 增加選項說明。 3. 增加常見問題項目。	8 12 20
1.3	2019/5/24	1. 增加 EEPROM 說明 2. 增加強制指定函數位址說明	11 15
1.4	2020/03/03	增加常見問題。	22
1.5	2020/08/18	1. 說明系統保留記憶體分配規則。 2. 增加使用建議。 3. 增加常見問題。	14 19 20
1.6	2022/02/14	1. 增加 multi_16b 函數說明。 2. Optimization 選項說明改名，原本為 Bank select optimize。	12 13
1.7	2022/09/13	1. 系統需求增加 Win11。 2. 增加常見問題項目。	3 23
1.8	2022/11/28	1. 刪除 Reserved RAM Size，新版不需要此設定。 2. 增加常見問題項目。	- 23
1.9	2023/02/15	1. 增加內嵌組合語言區塊。 2. 修正強制指定記憶體位址的說明。 3. 增加使用建議。	9 14 20
2.0	2023/08/23	增加 clear_ram 內建函數說明。	13
2.1	2024/08/23	1. EEPROM API。 2. Reserved RAM for interrupt 最大值。	12 15