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

User Manual

NYC for NY8 Series

C Compiler

Version 2.2

May 26, 2025

Table of Contents

1 Introduction

NYC_NY8 is the C Compiler for Nyquest 8-bit MCU “NY8 series”. *NYC_NY8* is called by the upper level development tools *NYIDE* to compile C program into assembly, *NYASM* Assembler will then assembly and link the object files to generate a .bin file, which is used to download to the board or program to OTP IC.

1.1 Outline of the manual

1. Introduction

   This chapter explains the role *NYC_NY8* plays and the basic requirements for the installation of *NYC_NY8*.

2. Use *NYC_NY8*

   How to use *NYC_NY8* through *NYIDE*.

3. Syntax and usage

   Introduce the syntax and usage of *NYC_NY8*.

1.2 System Requirements

- A PC equipped with Pentium 1.3GHz or higher CPU, Windows 7/ 8/ 10/ 11
- At least 2G SDRAM.
- At least 2G free space on the hard disk.

1.3 The Installation of *NYC_NY8*

Please contact Nyquest Technology to obtain the latest installation program. Double click the execution icon to activate installation wizard, and following the instructions to complete the installation process.

2 Use *NYC_NY8*

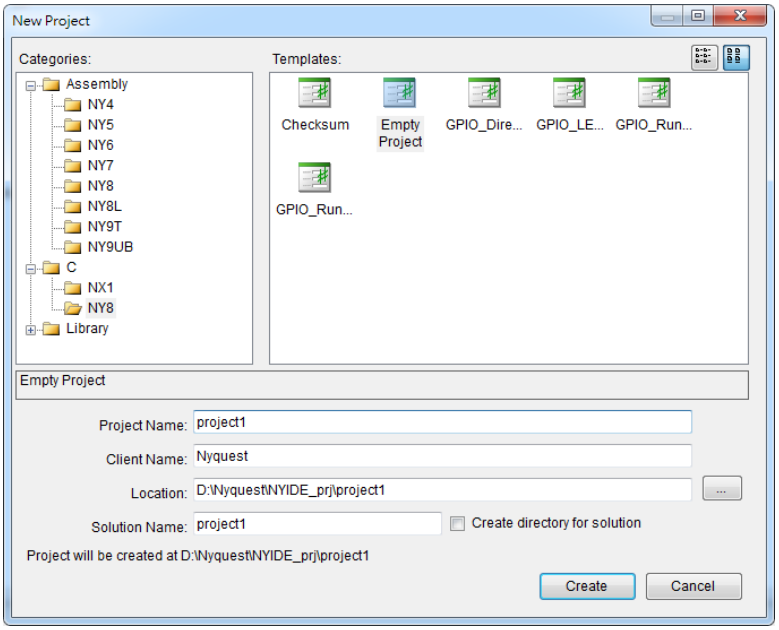
After finishing a program in NY8 software development tool - *NYIDE*, pressing Build in the *NYIDE* menu will automatically search for installed *NYC_NY8* to compile and link. The procedures for using *NYC_NY8* in *NYIDE* are described below.

2.1 Use *NYC_NY8* through *NYIDE*

NYIDE is an integrated tool provided by Nyquest for developing application of NY4 / 5 / 6 / 7 / 8 / 9T / 9UB / NX1 series microcontroller. The main purpose is to provide a platform for programming with Assembly language and C language, as well as build and strong debug functions. When using *NYIDE* to develop NY8 projects, *NYIDE* will automatically search for installed *NYC_NY8* toolchain on computer for building and debugging. The following is an introduction of using *NYIDE* to develop NY8 projects. More detailed operations, please refer to the *NYIDE* user manual.

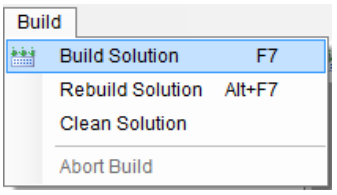
2.1.1 Create New Project

Open *NYIDE*, and select New Project. In the Project New window, choose C on the Categories and select NY8. Specify project name and type, then press “Create”, and *NYIDE* will automatically generate the necessary files.



2.1.2 Build

When user selects the Build / Build Solution menu (or press the shortcut key F7) on the *NYIDE* main screen, *NYC_NY8* will be called to perform the build action. If it is successfully built, the .bin file will be generated in the project directory for downloading or programming.



3 Syntax and Usage

NYC_NY8 supports standard ANSI C89 syntax, and adds some specific syntax for NY8 series IC.

3.1 Standard C Syntax

NYC_NY8 supports standard ANSI C89 syntax. For more detailed regarding language definitions, please refer to: Standard ISO/IEC 9899 (http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899).

3.1.1 Comment

There are 2 forms of Comment. The single line comment begins with double slash, and the multi-line comment begins with /* and ends with */.

Example:

```
// single line comment

/*
Multi line comment
*/
```

3.1.2 Data Type

The following table is the basic data types and the data range of *NYC_NY8*. Using stdint type must include the stdint.h file first.

Type	stdint	Length	Range
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

When manually writing assembly language and calling functions mutually between assembly and C, the method of parameter passing must comply with the definitions used in C. Due to the use of different compiler implementations for the 14-bit NY8 and 16-bit NY8 architectures, their calling conventions differ slightly. The following sections explain these differences using the same C function declaration as an example.

3.2.1 Calling Convention 14 bit NY8

Parameters are passed from left to right, and are placed in the following order: Acc, STK00, STK01, STK02, …, STK13.

For unsigned int types (which are 2 bytes), the high byte is placed first, followed by the low byte.

For return values, the high byte is stored in Acc, and the low byte is stored in STK00.

The following example demonstrates this using a function call:

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

The resulting call instructions are as follows:

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

3.2.2 Calling Convention 16 bit NY8

Parameters are passed from left to right, in order: __rc0, __rc1, __rc2, ..., __rc13.

For unsigned int (2 bytes), the low byte is passed first, followed by the high byte.

Return values are also placed sequentially in __rc0, __rc1, __rc2, ..., with the low byte first and the high byte second.

The Acc register is not used for value passing in this case.

The following example demonstrates this with a function call:

```
unsigned int add16(unsigned int a, unsigned int b);
```

```
g = add16(0x1234, 0x5678);
```

The resulting call instructions are as follows:

```
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 Extended Syntax

3.3.1 Reserved Word

All reserved words are listed below, and the user-defined symbols cannot be the same as the reserved words.

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 Interrupt

Interrupt service subprogram is divided into hardware interrupts and software interrupts in NY8 series, the addresses are 0x08 and 0x01 respectively. In C, an __interrupt attributes must be appended to the function definition for declaring function as interrupt service routine. __interrupt(0) represents the hardware interrupt service routine, and __interrupt(1) represents the software interrupt service routine. The compiler will then arrange this function at the specified address, for example, the hardware interrupt is at address 0x08. The compiler will automatically keep the current status before entering the interrupt service routine, such as register ACC, register Status, register FSR, and automatically restores the status when it leaves the interrupt service routine.

Ex.

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

//! software interrupt service routine
void isr_sw(void) __interrupt(1)
{
    // do something
}
```

3.3.3 Register Address Definition

All registers of NY8 IC have been defined in the header files located in “include” directory of the installation folder, header filename is IC part no. It is recommended to use the header file directly, which will save the efforts to define special registers.

3.3.4 Register Bits Definition

The following syntax allows users to define custom single-bit data structures.

```
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;
}
```

The __sbit keyword of 14-bit NY8can define one of the bits in the 8-bit register as a new variable. The syntax is as follows.

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

The __sbit can only be linked to one bit of the existing 8-bit variables, and it cannot occupy the new memory space independently. The following example demonstrates how to use the sbit to define two flags. The flag1 is linked to the 0th bit of myvar, the flag2 is linked to the 3rd bit of myvar (the optional bits are 0 to 7). The variable defined by sbit is a single bit, so the value can only be 0 or 1, and the result of read is also 0 or 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;
}
```

3.3.5 Inline Assembly for 14-bit NY8

To embed assembly within C code, use the ”__asm__“ keyword to insert inline assembly instructions. In the syntax below, the compiler stores the current program address in the STK00 and ACC registers before performing a direct jump to another function.

```
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 Inline Assenbly for for 16-bit NY8

In the 16-bit NY8 C environment, assembly code can be embedded directly using the __asm__ keyword. These embedded assembly instructions are processed by the Clang compiler and must follow the GNU gas syntax format.

Unlike the 14-bit NY8, which uses a different assembly syntax, the 16-bit version not only differs in syntax but also allows for finer control over interactions between assembly and external C code. Below is an example of embedded assembly that clears memory from address 0x20 to 0x7F. Assembly strings can span multiple lines using \n, allowing complex logic to be written cleanly within a single __asm__ statement. When defining labels within assembly, it's recommended to use numeric labels. For example, 1: defines a temporary label, and lgoto 1b jumps backward to that label (b for "backward"; f indicates "forward"). Using numeric labels helps avoid naming conflicts, especially in cases where inline assembly is expanded multiple times or reused in different contexts.

```
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"
            );
}
```

The previous example runs correctly on its own. However, if additional C code is placed before or after the __asm__ block, the behavior may not be as expected. For instance, in the following program, PORTA and PORTB are both set to the same value 1 consecutively. When setting PORTB = 1, the compiler assumes that the current value in Acc is already known, and thus omits the MOVIA instruction to load Acc.

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

When we insert an __asm__ block between these two lines and modify the Acc register, we must inform the compiler about which system registers have been altered. In the example below, the __asm__ statement includes ::: "a", "fsr", "status" to indicate the registers that are modified.

```
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;
}
```

The example above explicitly informs the compiler that the Acc, FSR, and STATUS registers are modified. This allows the final C statement PORTB = 1; to function correctly. For more detailed syntax and usage, please refer to GNU gas Extended Asm documentation. Next, let's look at an example that interacts with external variables. In this example, two variables a and b are added together, and then the result undergoes a nibble swap (swapping the upper and lower 4 bits).

```
#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);
    }
}
```

Inside function f, we can read and modify the C variables a and b. In this segment of assembly code, we don't know in advance which registers the variables will be assigned to. Instead, we can use %0 to refer to the first operand (i.e., operand 0) and %1 to refer to the second (i.e., operand 1). We then associate %0 with the C variable a using the syntax "+r"(a), as shown in the example. The actual compiled code for this example can be found in the .lst file, which displays the disassembled result.

```
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 Inline Assembly Block

The previous example program could be rewritten as assembly program block by using " __asm……__endasm" to completely wrap the assembly block. Please notice there is a semicolon at the end of " __endasm;".

In the 14-bit NY8, a non-standard extended syntax is supported (not available in the 16-bit NY8), which allows wrapping an entire assembly code block using "__asm …… __endasm;". It is important to note that the closing __endasm; must include a semicolon. When using the single-step execution feature in NYIDE, the entire assembly block will be executed all at once.

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

This syntax is not supported in the 16-bit NY8. For 16-bit NY8, it is recommended to use the multi-line asm format described in the previous section, using \n to separate lines. Be sure to specify the inputs, outputs, and the registers or flags affected by the entire assembly block.

3.3.8 Pointer Property

"__code" and "__data" are used to specify the pointer to be stored in ROM or RAM. The general pointer occupies 3 bytes, of which 2 bytes store address, 1 byte store pointer type to distinguish the pointer points to ROM or RAM. When the compiler have enough information to judge the pointer type, the 1 byte of pointer type can be omitted. For example, the array `data` in the following program is stored in ROM, and ptr1 and ptr2 are pointing to `data`. However, ptr1 has `__code` attribute, the compiler can determine that the pointer will only point to the ROM, then the compiler will actually generated machine code of ptr1 occupying 2 bytes, and ptr2 occupying 3 bytes. When using pointers, if user know that the pointer will only point to the ROM or RAM, please specify the `__code` or `__data` attribute in advance to save RAM usage, which also produces more streamlined instructions.

```
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 System Header File

The “include” folder in the *NYC_NY8* installation directory has C header files for all NY8 IC. This section describes the contents of these header files and how to use them.

3.4.1 Special Command Macro

The ny8common.h file defines commonly used assembly macros that control IC behavior in a lower-level, and the user can call these macros at the proper time.

Macro	Description
ENI()	Enable interrupt.
DISI()	Disable interrupt.
INT()	Trigger software interrupt.
CLRWDT()	Clear the watch dog timer.
SLEEP()	Sleep.
NOP()	Empty command.
UPDATE_REG(PORTx)	Update port register by movr instruction
SetOSCCR(x)	Set the parameter of OSCCR register

3.4.2 System Register Definition

The ny8.h will automatically include the dedicated header file according to the selected IC. For 14-bit NY8, all special registers supported by the IC are defined in the header file with the same name as IC. The special registers have four types: General page declared with attribute __sfr, F-page declared with

attribute __fpage, S-page declared with attribute __spage, and T0MD declared with attribute __t0md.

At the C language level, these registers do not have any differences. But the users still have to know that the actually assembly codes for accessing these registers are not the same. Only the general page register can be accessed directly, such as directly setting the value of a bit or directly exclusive or (XOR) a register. In addition to the general page register, other special registers cannot be directly accessed. The underlying assembly must move the value of these special register to the ACC register firstly, and then continue the next operation.

For the special register of general page, it's suggested to set individually bit to 1 or 0. But for other special registers, it's recommended to directly set the complete 8-bit value. Following such rules can get more compact machine codes.

It is recommended to use the ny8.h file instead of using the IC dedicated header file directly, which can reduce the inconsistency between the header file and the function library by replacing the IC. The ny8.h file is provided from *NYC_NY8* 1.10. If using the previous version of NYC_NY8, users must to replace the included header file after switching the IC.

3.4.3 ROM Data Access

Each ROM word in the NY8 architecture is either 14-bit or 16-bit. Using standard C pointers, only the lower 8 bits of each word can be accessed. To read the full 14 bits, use the read_14bit_rom function provided in ny8_romaccess.h. For the 16-bit NY8, the read_16bit_rom function is available to read the complete 16-bit word.

Ex.

```
#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);
```

For more examples, please refer to the sample program “Checksum” list in *NYIDE*.

3.4.4 EEPROM Data Access

Some ICs have built-in EEPROM that must use special commands to access. NYC_NY8 provides the C functions for accessing EEPROM.

The ny8_eeprom.h defines the functions to access EEPROM data. When using IC with the built-in EEPROM, ny8_eeprom.h will be automatically added to the project. The functions provided are as follows.

Function	Description
eeprom_read	Read a byte at the specified address.
eeprom_write	Write a byte at the specified address.(Deprecated)
eeprom_write_timeout	Write a byte at the specified address.
eeprom_protect_lock	Lock/unlock EEPROM write protection.
eeprom_protect_unlock	Lock/unlock EEPROM write protection.

- unsigned char eeprom_read (unsigned char address)

  The parameter specifies the address of EEPROM to read from.

  The return value is one byte data read from the specified address.

- void eeprom_write (unsigned char address, unsigned char value)

  The parameter address specifies the address of EEPROM to write to.

  The parameter value accepts one byte data, and it will be written to the specified address.

  This API was deprecated at NYC_NY8 1.70. Please use eeprom_write_timeout with timeout parameter instead.

- void eeprom_write_timeout (unsigned char address, unsigned char value, unsigned char timeout)

  The parameter address specifies the address of EEPROM to write to.

  The parameter value accepts one byte data, and it will be written to the specified address.

  The parameter timeout accepts one byte data which means the time limit for operation time. The available value for each IC and their corresponding time are different, please refer to the IC datasheet.

  User must unlock the EEPROM write protection before using eeprom_write.

  This API added at NYC_NY8 1.43.

- void eeprom_protect_lock (void)

  The ways of lock/unlock EEPROM write protection are different according to ‘EEPROM Write Mode’ option in config block. In ‘One Byte’ write mode, the EEPROM write protection will be unlocked while calling this funcion. After the eeprom_write finishing the write, the hardware will lock the write protection automatically. User must unlock the write protection everytime before writing in ‘One Byte’ write mode. In ‘Continuous Write’ mode, the EEPROM write protection will be unlock at the first call, user can then use eeprom_write function to write to EEPROM for multiple times. The eeprom_protect_lock will re-lock the write protection at the second call. User must call the function lock after all the writes are completed.

- void eeprom_protect_unlock (void)

  The void eeprom_protect_lock and the void eeprom_protect_unlock is the same program with different names. eeprom_protect_unlock and eeprom_protect_lock use the same program space without extra ROM consumption.

Ex.

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

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

For more examples, please refer to the sample programs “ eeprom-write-one-byte” and “ eeprom-continuous-write” listed in *NYIDE*.

3.4.5 Built-in Function Multi-16b

The input and output of C language multiplication operation must be the same data type. Multiplying two 16-bit integers produces only a 16-bit result. If a 32-bit result is required, the input data must be converted to 32-bit (long). The built-in function multi_16b is a special multiplication function. The input is two positive 16-bit integers and the output is a positive 32-bit integer. The resource consumption of ROM and RAM is between multiplication of 16-bit and 32-bit. Please note that the multi_16b function cannot calculate negative numbers. NYC_NY8 1.43 version supports this function.

Ex.

```
#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 Built-in Function clear_ram

Clear all RAM of IC to 0, not only the variables declared in C language, including unused RAM from user programs, and temporary variables generated by Compiler will be set to 0 as well. The Special Function Register (SFR) will not be changed. The actual program logic is the same as NYIDE project setting check Clear RAM to zero. The difference between the two is the execution timing. Clear RAM to zero will only be executed once before entering the main function, while clear_ram can be executed manually at any time. This function will automatically link the correct program in different IC to ensure that the set RAM range meets the IC specification. NYC_NY8 1.60 and newer versions support this function.

Function prototype claims:

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

3.5 Option

Using *NYIDE* to develop a C language project, there are several project build options can be set. These options can control the compiler, assembler and linker behavior. User can select the Project / Project Settings on Menu to open the setting interface.

- Use RAM Bank0 only: Support 14-bit NY8 only. Selecting this option can only uses Bank0 memory, and the generating Code size is smaller. Some IC body only has a single Bank and this option is forced to select. Deselecting this option will insert the switching bank command before accessing the memory and allow all memory to be used, but the resulting Code size will be larger.
- Clear RAM to zero on startup: Clear all the memory before starting the main function. The global initial variable is not affected by this option. No matter whether this option is selected, the global variable with initial values will complete the initial value setting before entering the main function. Disabling this option can reduce code size, but the user must manually initialize all global variables without initial values, as the memory content at startup is undefined.
- Generate ASM listing file: Support 14-bit NY8 only. The listing file named *.lst will be produced after assembling, deselecting this option can speed up the compiling speed.
- Generate listing file: The listing file named *.link.lst will be produced after linking. This file is the disassembled result of the final .bin file. Deselecting this option can speed up the compiling speed.
- Generate map file: The listing file named *.map will be produced after linking. This file contains address assignment information. Deselecting this option can speed up the compiling speed.
- Optimization: Support 14-bit NY8 only. Users can select Level 1~3 for optimization. The higher level, the better the optimized program. Please note that this option might cause abnormal while working with the inline assembly language.
- Backup register for interrupt: This option determines whether specific special registers should be backed up upon entering an interrupt service routine. The TBHP register is used when reading data from ROM; if the user is certain that no ROM access occurs during interrupts, backing up TBHP can be disabled. The PCHBUF register is used for indirect jumps; if indirect jumps (such as switch-case constructs) are not

used within the interrupt service routine, backing up PCHBUF can also be disabled. It is recommended to open the generated .lst disassembly file after building the project to verify whether these special registers are actually used within the interrupt service routine.

- Reserved RAM for interrupt: Support 14-bit NY8 only. Reserved memory for used by the ISR, to store the current state of variables in function before entering the interrupt. An interrupt may occur while an array is calculated or function is called, and it may break the calculation to registers currently in operation. Therefore, if the interrupt causes wrong behavior, user would need to instruct the compiler to store the variables that their operations are interrupted, and then set the memory size that needs to be reserved for the compiler to backup according to the variable size in use. The minimum is 0 and no function call state is reserved. The maximum is 13. The larger setting value will cause the entry time of ISR to be longer because more instructions must be used to backup the current state. The actual numbers of instruction is slightly different if the backup memory is located in a different bank. Please refer to the following table.

Reserved RAM Size	Additional instructions before entering the interrupt	Note
0 byte	0	
1 byte	4 word	
2 byte	8 word	or 4 word
3 byte	10 word	or 6 word
4 byte	12 word	or 8 word
5 byte	14 word	or 10 word
……		
11 byte	26 word	or 22 word
12 byte	28 word	or 24 word
13 byte	30 word	or 26 word

- Include path: Set the search path for the C language keyword “include” reference header file. The default path is the include folder of the project root directory and the *NYC_NY8* installation directory. User can add a custom path.

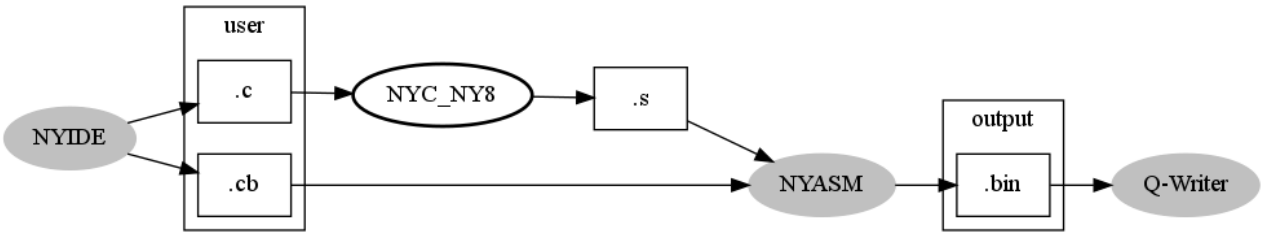
3.6 Development Process

NYC_NY8 is typically invoked automatically by NYIDE. Users will usually only interact with the header files (.h) within NYC_NY8 and will not execute the internal executable files (.exe) directly. This section explains

how NYC_NY8 interacts with other software components.

3.6.1 14-bit NY8

In the 14-bit NY8, the compiler used is SDCC. It is a standalone compiler and must be used in conjunction with an assembler (*NYASM*) to complete the build process .Use *NYIDE* to write the C language program and set the configuration file ".cb" required for the project. *NYIDE* will automatically call *NYC_NY8* to generate the assembly file ".s" when building, and then call the *NYASM* to assemble the assembly code and the configuration file to produce the final .bin file. Finally, user can use the *Q-Writer* to burn the .bin file to IC.



3.6.2 16-bit NY8

In the 16-bit NY8, the compiler used is Clang/LLVM. It is an integrated toolchain that combines the compiler, assembler, and linker into a unified design, eliminating the need to install NYASM separately. When writing C programs in *NYIDE*, users configure the project using a .cb configuration file. During the build process, *NYIDE* automatically invokes NYC_NY8 to generate the .bin file. Finally, the generated .bin file can be programmed into the IC using *Q-Writer*.

3.7 Advanced Usage

This section introduces some advanced usages for *NYC_NY8*.

3.7.1 Specify the Memory Address for 14-bit NY8

In general, the variables of C language do not need to specify memory addresses, they will be automatically relocated to a proper space via linker. However, there are requirements for specifying variable address in some occasions. *NYC_NY8* provides a special syntax for assigning address of the specified variable, add "__at(addr)" before the variable type, and addr is the specified address.

Ex.

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

It is important to note that variables should not be declared in the SFR (Special Function Register) section. If user wants to access the SFR, please use predefined variables defined in the Header file of the selected IC. Because *NYC_NY8* will link to the built-in static library during the project build process, and the library uses the SFR declared in the header. If the user redefine SFR in the header, the project build will fail. If you want to rename a SFR, please use the "#define" preprocessing instruction.

Ex.

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

When users have multiple .c files, they must also notice similar situations. Only one of the .c can actually occupy memory, and the other .c must use the keyword “extern” to define the variable as external.

Ex.

File: main.c

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

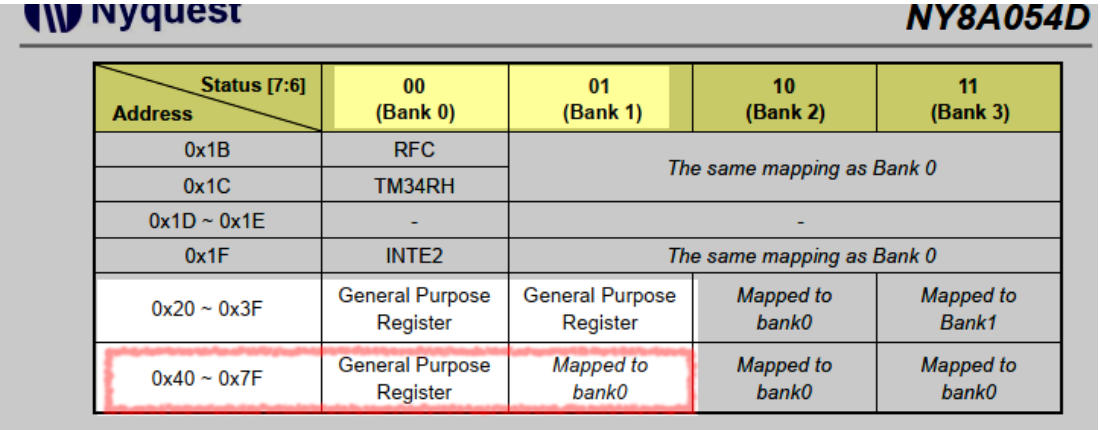
File: my_var.h

```
#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
```

User must notice and set the Reserve RAM Size form Options for the forced specified address, it must keep enough share bank for system.



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*

The figure is cut from page 18 of NY8A054D datasheet, it describes the R-Page address mapping . In the red frame, Bank0 or Ban1 can access the same memory. The reserved RAM must allocated in the red frame (0x40~0x7F).

3.7.2 Specify the Memory Address for 16-bit NY8

Unlike the previous section, the 16-bit NY8 does not support the __at(0x123) syntax. Instead, it uses standard C pointers. User can write (volatile unsigned char*)0x123 to cast the constant 0x123 into a pointer, and then use the dereference operator * to access the memory at address 0x123. A more practical example is shown below.

File: main.c

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

It is important to ensure that the memory address 0x123 used in the previous example does not overlap with addresses automatically allocated by the compiler, as this could potentially corrupt the system state. User can check the .map file generated after project compilation to determine which memory addresses have already been allocated.

3.7.3 Specify the Address of Function for 14-bit NY8

In general, functions of C language do not need to specify memory addresses, they will be automatically relocated to a proper space via linker. However, there are requirements for specifying function address in some occasions. *NYC_NY8* provides a special syntax for assigning address of the specified function, add "__at(addr)" before the function return type, and addr is the specified address.

Example:

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

Please do not assign the address of function to 0x00, because 0x00 is occupied by NYC_NY8 start program.

3.7.4 Specify the Address of Function for 16-bit NY8

Unlike the previous section, the 16-bit NY8 does not support the __at(0x123) syntax. For applications such as Rolling Code, where a fixed address must be assigned, the macro SET_ROLLING_CODE_ADDR is provided to explicitly specify the target address.

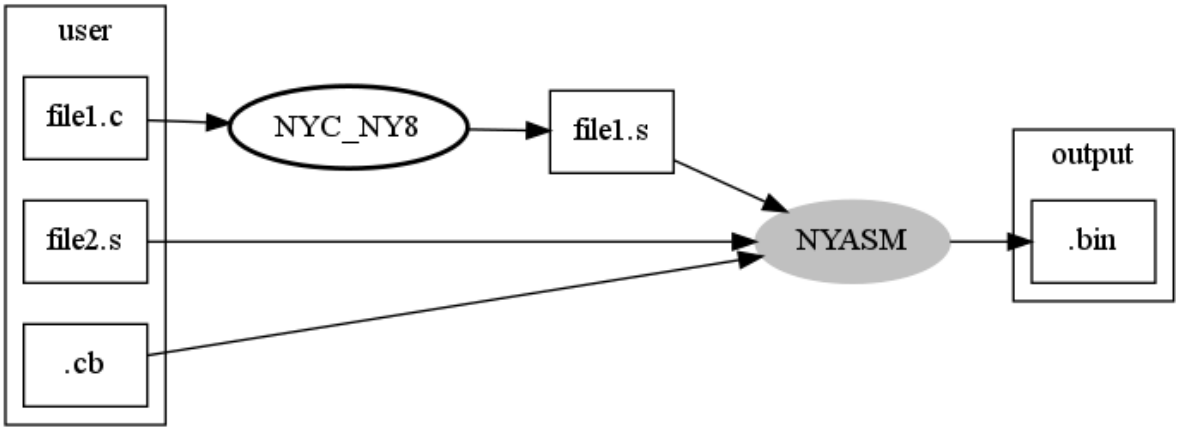
File: rolling-code-data.c

```
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");
}
```

For more details, please refer to the Example: Rolling Code from NYIDE.

3.7.5 Mixed Usage of C and Assembly for 14-bit NY8

In Development Process, we can see that *NYC_NY8* converts the ".c" file to ".s" file, and then the *NYASM* assembles the ".s" file with the ".cb" file into a ".bin" file. However, *NYIDE* allows more than one .c file, and also more than one compiled assembly .s file. User can even write his own .s files without using *NYC_NY8* to generate the files, and these files can operate with .s files generated by *NYC_NY8*. This chapter will introduce how to write .s file that cooperatively operates with .s file generated by *NYC_NY8*.



Start with a simple example – Rolling code application. Rolling code with preset mode application must keep ROM 0xE and 0xF blank, and rolling code will be written when programming afterward. In the compilation time, 0xE and 0xF must be reserved. However, in the C language, it's hard to fill in specified values to the given addresses of IC, except that the __interrupt keyword enforces program to put at 0x1 or 0x8. The solution is to use the assembly to work with the C language. The following will demonstrate how to use the assembly to keep blank at 0xE and 0xF addresses. When testing, the addresses 0xE and 0xF will be filled in the test values, to read Rolling codes to verify with C language program.

Here are three files:

- rom.s fills NOP in 0xE and 0xF, fills 0x255 and 0x3AA test data if testing, and exports the symbol "___rolling_code_addr" for C language use.
- rom.h defines the external symbol __rolling_code_addr.
- main.c contains main program reads rolling codes and verifies.

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
```

In the rom.s file, the exported external symbol name is ___rolling_code_addr, please note that there are three underlines. When the C code is compiled into assembly, an underline will be added to all symbols, on the other hand, in order to differentiate the symbols from C language, we will add an extra underline.

In assembly, it's easy to directly specify the location of the data using ORG command.

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 */
```

In rom.h, there has only one line, an external symbol __rolling_code_addr definition. Here the __code keyword explicitly defines this symbol is in ROM. There are only two prefix underlines for this symbol name, since the C compiler will automatically add an underline when compiling into assembly.

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();
    }
}
```

The main.c use the symbol __rolling_code_addr defined by rom.h to access ROM data, of course, user can choose not to use this symbol, and directly specify the address 0xE. If the address of rolling code is changed, it must also change the address of org instruction specified in rom.s, and the address in main.c.

Then we look at the example of the main.c, the function read_14bit_rom read the ROM data is defined in the library. ny8_romaccess.h contains its function prototype, the implementation is not really C but the assembly. It is listed below to describe how to use the built-in function to call the function defined by assembly from the C.

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
```

In the above two files, we can see the declaration of C and the implementation of assembly. The first thing to note is the difference in symbolic name. In the C language called read_14bit_rom, and the assembly is named _read_14bit_rom, which has one extra underline. The reason is as mentioned earlier, after the C language compiled into an assembly, all symbols will be added the prefix underline. This function has an input parameter that is the ROM address pointer, and a return value type “int (16-bit)”. ROM address pointer is actually 16 bits, two 8-bit registers. The passing parameter uses ACC first, then STK00 and STK12 public register.

In this example for the 16-bit pointer, the high 8-bit will be stored in ACC, the lower 8-bit will be stored in the STK00. So the first step of the assembly is to move ptr [15: 8] stored in ACC to the TBHP register, and move ptr [7: 0] stored in STK00 to ACC.

The storage of return value is also the same logic, high bits are in ACC, and lower bits are in STK00. When the TableA completes reading the ROM data, the ROM [7: 0] is stored in ACC, then ACC is moved to STK00, and move TBHD stored the ROM [13: 8] to ACC. Finally, ret command returns to this function.

For main.c, it does not care whether the read_14bit_rom is written by C or the assembly. As long as the input parameter and the format of output return value conform to specifications, they will perfectly mutual cooperate.

3.7.6 Mixed Usage of C and Assembly for 16-bit NY8

In the 16-bit NY8, it is also possible to mix C and assembly code. The built-in example program Interrupt_with_assembly demonstrates how to add assembly source files to a C project. Since the compiler differs from that of the 14-bit NY8, several key differences should be noted:

- File extension case matters: .s and .S are treated differently.A lowercase .s file is considered pure assembly, while an uppercase .S file is passed through the C preprocessor. Specifically, .S files support C-style preprocessor directives such as #define and #include.
- Assembly syntax: The assembly code follows the NY8 16-bit instruction set, as documented in the datasheet.
  However, pseudo-instructions use GNU gas syntax rather than NYASM syntax. For details, refer to the GNU gas manual.
- Symbol naming conventions: C and assembly symbols do not require an underscore prefix. In the 14-bit NY8 with the SDCC compiler, assembly functions needed a leading underscore to be callable from C. In the 16-bit NY8 with the Clang compiler, this is no longer necessary.

3.8 Suggestion

Some suggestions for developing C language projects are shown below.

- Try to use unsigned variables. In some operations which do not judge plus or minus, it will be faster.
- Do not use constants and variables interactively in the expression, intensively using the constants will have an optimized code.
  Ex. “1 + a + 2” is a bad coding style, as 1 and 2 cannot be calculated in the compilation time. It is recommended to write “a+1+2”, 1+2 can be calculated in the compilation time, and it only needs to calculate “a+3” in the execution time.
- Do not use float point. The float point operation consumes lot of memory, use integer operations instead of floating point .
- Using `if (INTFbits.T0IF)` to replace `if (INTFbits.T0IF == 1)` can get a more compact program.
- Do not set some bit of the S-Page / F-Page register continuously and individually.
  The S-Page / F-Page registers are read and written by special instructions, and continuously bit setting will have to read and write these special registers many times, unlike R-Page register can use a single BCR / BSR instruction to set the individual bits. When using the S-Page / F-Page register, it is recommended to set the bits at a time.
- If all global variables will be given initial values before using, it can specify the *NYC_NY8* not to clear the value as 0 to reduce ROM usage. User can control the setting through Project Setting / Clear RAM to zero from NYIDE setting window.
- If a lot of initial values of global variables are 0, using Clear RAM to zero will save program space. (about 5 bytes or more)
- If the RAM usage is not huge, try using the small model to turn off the bank switch. This can produce a

more compact code.

- Do not split the program into too many .c files. This will affect the optimization and increase the amount of RAM used. Because the compiler cannot assume that if the two functions will be executed at the same time, it must assign the separate memory to each other.
- Try to assign the static attribute to the function, and mark that this function will not be called by external .c, which can improve the optimization
- Use the NYASM version of the same released period. Because the files generated by *NYC_NY8* will be passed to *NYASM* for the next processing, if the version doesn't match, there may be incompatible situation. For example, *NYC_NY8* may produce the instructions that are not supported by old version of *NYASM*.
- If the pointers will only point to ROM or RAM, use the pointer attribute __data and __code to direct the compiler when declaring.

3.9 FAQ

Q1: Why is the interrupt missing when enabling multiple interrupt sources?

A:

Take enabling the PortB change interrupt and Timer1 interrupt simultaneously as an example, using the clearing bit command to clear the T1IF is likely to erroneously clear the PBIF. It is recommended to use the immediate value 0 to clear T1IF, the reason is described as follows.

When clearing T1IF (Timer1 interrupt flag), the IC will perform the following steps:

1.1 Read all bits of the "INTF" first.

1.2 Clear T1IF bit to 0 and other bits remain unchanged. The value will then be written to the "INTF" register.

If the PBIF bit is set due to a PortB change interrupt between step 1.1 and 1.2, which will then be overwritten by step 1.2 and erroneously cleared to 0, causing the PortB change interrupt to be occasionally ignored.

Please refer to the following code to clear T1IF (Timer1 interrupt flag).

Recommended Instruction Code	Not Recommended Instruction Code
"INTF = 0xF7;" or "INTFbits.T1IF = 0;"	INTF &= 0xF7;
Generating assemble language	Generating assemble language
MOVIA 0xf7 MOVAR _INTF	BCR _INTF, 3

Q2: The program of INTE2 register shows the error message: Use BSR instruction to clear interrupt flag may cause other interrupt flags accidentally cleared if other interrupts are issued immediately after.

A:

The 8-bit INTE2 register is consist of 2 parts, the high nibble INTE2[7:4] is the interrupt flag, and the low

nibble INTE2[3:0] is the setting for enabling the interrupt function. If user uses "&=" or "|=" operation on INTE2 register, the C compiler will generate BCR or BSR instruction. These instructions are not an instruction cycle within the IC. If the interrupt occurs and the interrupt flag is raised then the value is set, may cause the interrupt flag to be cleared, thus the interrupt is missing. It's recommended to access INTE2 register via the following 2 methods:

1. Write the complete 8-bit value directly while clearing the interrupt flag. Clear the target interrupt flag and set the others as 1. The following example shows the INTE2 register is consist of bit4:T3IF and bit0:T3IE only, and to clear the T3IF:

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

It will generate a simplified assembly program.

```
MOVIA    0xE1
MOVAR    _INTE2
```

2. If user is not sure of other bits while clearing an interrupt flag, user can set the bit individually. For example, use INTE2bits to clean T3IF.

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

It will generate a more complex instruction to make sure all the bits except T3IF remains the original status.

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

Q3: There are programs of accessing Array in both the main loop and the interrupt service routine, the data is occasionally read and written to the wrong address?

A:

Because accessing Array uses the common system register, if it enters the interrupt service routine and accessing Array is also in the interrupt service routine, the common system register status will be changed and cause reading and writing address error.

It is recommended to use DISI() and ENI() for interrupt service routine control in this case to prevent the accessing Array process from entering interrupt.

Q4: I noticed that the register definition files of various IC bodies in C:\Nyquest\NYC_NY8\include\ ny8a054a.h, but why it always links fail after changing the register name?

A:

The register name is not only defined in <icbody>.h but must also exist in the static library. The static library is in the lib folder of the NYC_NY8 installation directory and the file name is <icbody>.a. The static library is

a binary file and cannot be modified by the user. Modifying the header file will cause the defined registers not found in the library when linking.

It is recommended not to rename the built-in register in the system.

Q5: Set the variable value in the interrupt service routine and read in the general program process. The reading result is abnormal?

A:

The variable that the interrupt service routine shares with the normal process, it is recommended to declare with the keyword “volatile” to prevent the variable being optimized and cause program abnormally. The following example illustrates that the shared variable count is optimized and cause program abnormally.

```
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();
    }
}
```

In the above example, when the delay function is called, the count variable is initialized to 0 first. As timer interrupt is enabled, the count variable will increase in each interrupt. Then, when the value of the count variable reaches delay_count, the delay function will be suspended. However, in the actual execution, the while loop in the delay never jumps out, causing an infinite loop. Since the compiler optimization regards that the count variable does not perform any other operations after it is set to 0, count variable can be substituted with the constant 0. Therefore, the judgement condition of while loop is optimized as “`while (0 < delay_count)`”, and the condition is always true thus cause an infinite loop. The solution is to change the declaration of the count variable and declare it as volatile to make the count variable not being optimized.

```
volatile uint8_t count;
```

Q6: Why the continuous equal '=' assignment is different from the assemble language generated by multiple independent assignments?

A:

It's different for sure. To set the initial value, it's recommended to set separately.

The continuous setting will start the execution from the end, and read the value again and set it to the next target register. This will produce a more complicated assemble language program. For example, in the following program, it is recommended to use the first line instead of the second line.

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

Q7: When INTE2 = ~(0x01), the warning message shows overflow in implicit constant conversion

A:

To eliminate the warning message, user should add the type conversion to INTE2 = ~(0x01), for example INTE2 = (unsigned char) ~(0x01).

Because the reverse operation of 0x1, user will get the int type 0xFFFE (16 bits). The 16 bits will be specified to 8 bits of INTE2 and the high bits are automatically discarded and a warning message is generated at the same time. A clear type conversion can eliminate this warning message.

Q8: The warning message shows conditional flow changed by optimizer

A:

This is usually a problem with the condition of the judgment. For example, the following program will generate this warning, and after the warning is generated, the entire C program will not generate the asm program.

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

That is because the compiler sees the entire program as meaningless. (Any vairable AND 0 must be 0, judging whether it is equal to 0 will always be true)

Q9: How to operate the combined multiple bytes

A:

Four 8-bit variables are combined into a 32-bit long data type. It is not recommended to use the left shift operation because it will consume more ROM. There are two procedures listed below, the first one is not recommended and the second one is recommended.

```
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;
}
```

It is recommended to use the 8-bit and 32-bit overlapping data structuresthat are defined by union, it could omit left shift operation and OR operation. Please refer to the following program example.

```
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: What are the differences in usage among the two compilers: SDCC and Clang?

A:

The 14-bit NY8 uses the SDCC compiler, while the 16-bit NY8 uses the Clang compiler. Both follow the standard C99 syntax specification as supported by the C language.The differences lie in the extended syntax beyond the standard specification. Several key differences are listed below:

Feature / Functionality	SDCC	Clang
Interrupt declaration	__interrupt(0)	Interrupt
Mixed assembly file syntax	NYASM syntax	GNU gas syntax
Single-bit definition	struct or __sbit	struct or type casting
Parameter passing	A + STK00~STK12	__rc0 ~ __rc15
SFR definition method	External symbols	Dereferencing constant address
Rolling code positioning method	__at	SET_ROLLING_CODE_ADDR
Dereferencing of references (& and *)	Not supported	Supported
Accessing absolute memory address	__at(0x60)	*(volatile char*)0x60

Q11: Cannot find the disassembly output.lst filr.

A:

First, make sure that "Generate listing file" is checked in the Linker tab of the NYIDE project settings. Next, note that the output directory of the file may vary depending on the IC and the version of NYIDE being used. In some cases, the .lst file is located next to the .bin file; in others, it may be in the OBJ folder or the build directory. Please check all of these locations.

Q12: No .s file found in the OBJ directory for NY8 16-bit.

A:

Right-click on the .c file in NYIDE and select Compile Single File to generate the .S file. Additional note: File extension case matters—uppercase .S indicates that the file will be processed by the C preprocessor, while lowercase .s will not. A practical difference is that .S files can use #define macros, whereas .s files cannot.

Q13: Where is the header file of NY8 16 bit?

A:

For NY8 14 bit, the filepath is C:\Nyquest\NYC_NY8\include\ny8.h

For NY8 16 bit, the file path is C:\Nyquest\NYC_NY8\llvm\include\ny8.h

An additional ny8_constant.h file is located in either the NYIDE installation folder or the NY8 example code folder.

4 Revision History

Version	Date	Description	Modified Page
1.0	2017/08/14	Formal release.	-
1.1	2017/10/27	Add FAQ.	18
1.2	2018/05/30	1. Add sbit syntax. 2. Add new description to Option. 3. Add new FAQ.	8 13 21
1.3	2019/05/24	1. Add the description of EEPROM. 2. Add the description of Forced the Specified Function Address.	11 16
1.4	2020/03/03	Add FAQ.	22
1.5	2020/08/18	1. Add the descriptions of system specify memory. 2. Add item to Suggestion. 3. Add item to F&Q.	14 19 20
1.6	2022/02/14	1. Add the description of “multi_16b” Function. 2. Rename Bank Select Optimize as Optimization.	12 13
1.7	2022/09/13	1. Add Win11 to support system requirements. 2. Add item to F&Q.	3 25
1.8	2022/11/28	1. Remove the description of Reserved RAM Size. 2. Add item to F&Q.	- 25
1.9	2023/02/15	1. Add the Inline Assembly Block. 2. Modify the decriptions of Specify the Address of Variables 3. Add notes to Suggestion.	9 15 22
2.0	2023/08/23	Add the desctriptions of Built-in function clear_ram.	14
2.1	2024/08/23	1. EEPROM API 2. Update the maximum value of Reserved RAM for interrupt.	13 15
2.2	2025/05/28	Add NY8 16-bit description.	7, 12, 22, 24, 25, 30