九齐科技股份有限公司

Nyquest Technology Co., Ltd.

用户手册

NYASM

Nyquest MCU Assembler

Version 5.6

Nov. 25, 2025

目　　录

1 前言

第一章将介绍在使用*NYASM*所需的基本相关知识。

1.1 导览

1.1.1 大纲

这份文件主要介绍如何使用*NYASM*来开发九齐科技的微控制器应用程序。

用户指南大纲如下：

2 *NYASM简介*：定义及说明*NYASM*软件是如何与其他的开发工具联结使用。.

3 *NYASM*安装和开始：*NYASM*安装说明及操作概述。

4 在窗口操作系统中使用NYASM：介绍*NYASM*的操作界面及参数。

5 伪指令：介绍*NYASM*程序语言所包含陈述、操作数、变量和其他的组件。

6 宏指令：介绍如何使用*NYASM*内建的宏处理器。

7 表达式的语法与运算：为在*NYASM*源码文件中使用复杂的表达式提供一个指导方针。

附录A - 快速索引：*NYASM*快速参考、MCU列表。含*NYASM*所产生的错误信息及警告信息列表。

附录B - 词汇表：专业术语词汇表。

1.1.2 指南规范

使用手册时请依照下列的规范：

表格 1-1

型态	意义	范例
Arial 字型	用户程序或范例程序。	#define BITWIDTH
角括号: <>	用户定义变量。	<label>, <exp>
大括号与直立线段: { | }	互斥的参数选项。	an OR selection error level { 0 | 1 }
方括号: []	方括号内的内容是可省略的。	[<label>] db <expr>[,<expr>,...,<expr>]
省略号: ...	省略范例中已不需要特别说明的部分。	List “list_option”, ...,“list_option”
0xnn	16进位数值, n 是一个16进位数字。	0xFF, 0x3B

1.1.3 更新

NYASM 和其他九齐科技的开发工具将不断地依照用户需求做更新，因此用户可能会遇到在使用最新版软件时可能会有一些实际的交谈窗口或工具的描述与本份文件有些许的出入。最新版的文件请参照九齐科技的网站。

1.2 建议阅读

本文件只是一份介绍如何使用*NYASM*的入门指南。用户在做应用开发时需要再参考所使用微控制器的数据手册。

- Interface
  粗体字指定一个按键 OK、Cancel。
  三角括号内大写字符：< >定义为特殊键 <TAB>，<ESC>。
- Microsoft Windows Manuals
  本用户手册是假定用户熟悉微软的窗口操作系统。

1.3 九齐科技网站

九齐科技透过全球信息因特网提供在线支持。在这个网站上用户将可以很容易的获得九齐科技所提供的最新文件及信息。用户可以使用Microsoft® Internet Explorer或其他浏览器透过因特网就可以连上九齐科技网站。

- 联机到九齐科技因特网网站
  九齐科技网址可以使用你惯用的因特网浏览器连接到：**http://www.nyquest.com.tw**
  九齐科技的网站提供多样的服务，用户可以下载最新的开发工具、资料手册、应用手册、用户指南和文件。

其他特别需要注意的信息：

- 九齐科技最新的新闻稿。
- 产品信息。

1.4 开发系统的用户通知服务

九齐科技提供用户通知服务来帮助我们的用户花最少的精力去掌握目前九齐科技的产品。每当我们在产品系列或开发工具上有任何的改变、更新、修订或勘误，用户都能收到我们的email通知。

1.5 用户支持

九齐科技所有产品的用户将可以透过下列几种渠道获得所需要的帮助。

- 经销商或代理商。
- 应用工程师（FAE）。
- 热线。

用户若需要支持服务时可以与经销商、代理商或应用工程师联系。

2 *NYASM* 简介

*NYASM*为一套在PC执行的窗口版应用程序，它是针对九齐科技的微控制器家族所提供的一个汇编语言开发平台。

2.1 *NYASM* 系统需求

- Pentium 1.3GHz或更高级处理器，Windows 7、8、10、11操作系统。
- 至少1G以上的动态存取内存（SDRAM）。
- 至少2G 可用硬盘空间。
- 显示器和显示卡支持分辨率1366x768或更高。
- 需安装 .Net Framework 4.0。

2.2 *NYASM* 功能性

*NYASM*是九齐科技为8-bit和4-bit微控制器提供一个汇编语言开发的通用型解法方案。

*NYASM*主要的特色包含如下：

- 支持所有MCU的指令集。
- 窗口化的操作化界面。
- 完整的伪指令。

2.3 兼容性

*NYASM*兼容于目前九齐科技所有的产品开发系统，它包括了*Q-Code*、*NYIDE*等。*NYASM*支持一个明确且一致的方法（章节4）。*NYASM*也支持一些较旧的语法，所以用户可以安心的使用本文件所描述的方法去开发新的程序。

3 *NYASM* 安装和开始

本章将介绍如何安装*NYASM*到你的操作系统中，且将对*NYASM*这个编译程序做概略介绍。

内容：

3.1 安装

目前*NYASM*为支持于窗口XP/WIN7/WIN8的窗口版，*NYASM*.EXE有一个窗口图型化操作界面。*NYASM*.EXE可以安装在窗口XP/WIN7/WIN8上且被正常的执行。用户可以从九齐科技网站上下载*NYASM*。*NYASM*下载时的文件格式为ZIP压缩文件。

安装方法：

- 在你所要存放文件的地方建立一个目录。
- 使用WinZip对*NYASM*.zip做解压缩。

3.2 编译程序简介

*NYASM*所产生的完整程序代码可以直接被微控制器所执行。*NYASM*默认的条件即可以产生完整的程序代码。如图3.1所示的程序，当一个源码文件在这种方式下被编译时，程序中所使用的值都必须在源码文件中被定义，或者是在被引入的含括档中定义。假设整个编译的程序都没有出现任何的错误时，最后将会产生一个BIN文件。BIN档为源码文件经过组绎后可被执行的机械码，使用烧录器将这BIN档读入即可对演示版做烧录的动作，并验证功能。

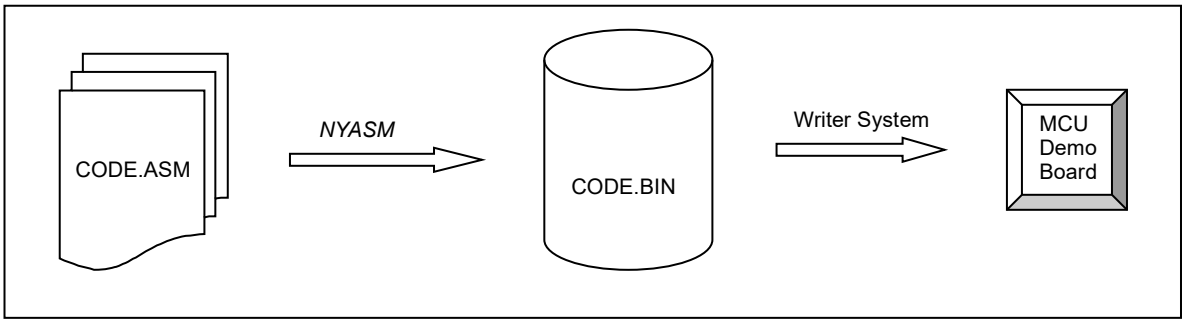

图 3-1 产生一个完整的程序代码并验证功能

3.3 编译程序输入/输出文件

*NYASM*有些默认的扩展名格式，其各代表着相关不同的功能。

表格 3-1默认扩展名

扩展名	用途
.ASM	输入到*NYASM*的源码文件。 <source_name>.ASM
.LST	编译完成后所产生的列表文件。 <source_name>.LST
.ERR	*NYASM*输出文件，内容为记录编译时所找到的警告与错误。 <source_name>.ERR
.BIN	编译完成后所产生的二进制格式应用程序机械码。 <source_name>.BIN
.HEX	编译完成后所产生用来取代二进制格式的十六进制格式应用程序代码。 <source_name>.HEX
.DBG	*NYASM* 输出的符号及除错文件，这个文件主要提供 *NYIDE* 除错模式使用。 <source_name>.DBG

3.3.1 源码格式 (.ASM)

源码文件可由一般的ASCII文本编辑器所产生。其内容格式必须符合下列基本的方针。源码文件的每一列可能包含了四种不同的信息：

- 标记（Labels）
- 指令助忆码（mnemonics）
- 操作数（operands）
- 批注（comments）

上述的顺序及位置是非常重要的。标记必须放在每一列的第一个非空白位置。指令助忆码若单独占用一列，必须放在第一个非空白位置。指令助忆栏若与在标记在同一列，必须在标记的后面并以冒号（：）隔开。操作数是跟随在指令助忆码的后面并以空格符隔开。批注可以是放在标记、指令助忆码、操作数后面，也可以是在任何列的任意位置。冒号或空格是用来分隔标记和指令助忆码及指令助忆码和操作数。复合的操作数必须使用逗号做分隔。范例如下：

范例：

***NYASM* 程序代码范例 (显示多个操作数)**

```
; sample NYQUEST assembler source code
    ;
    list    p=ny5c640b, c=off, r=hex
    ORG_OFF             equ     0x30
    ORG_SUBOFF          equ     0x00
    SUBPPTRADDR         equ     ORG_SUBOFF+ORG_OFF
```

```
#include   "2102.h"
org        0x10
mvma       0x20
jmp        start
org        0x30
start:
mvma       0x30
mvat       0x12
end
```

3.3.1.1 标记 (Labels)

一个标记必须放在每一列的第一个非空白位置。一个标记后面必须加冒号（：）。标记的开头必需为英文字母或是底线（_），标记内容则可由英文字母、底线（_）或是@符号所组成。标记栏的英文字母默认不区分大小写，但是也可以从*NYASM* 的命令列中选择打开。

3.3.1.2 指令助忆码 (Mnemonics)

汇编语言指令助忆码、汇编语言伪指令和宏调用可以放在每一列的任何行位置。如果一列同时存在有标记及指令助忆码，则在两者之间需用冒号隔开。

3.3.1.3 操作数 (Operands)

操作数必须使用一个以上的空格或tab来和指令助忆码隔开。多个操作数必需使用逗号来区隔。

3.3.1.4 批注 (Comments)

*NYASM*会将分号后面的任何文字、符号视为批注。从分号后面开始到一列的结束中间的所有字符都将被*NYASM*所忽略。若在字符串常数当中是允许包含了一个分号，并不会和批注搞混。

3.3.2 列表文件格式 (.LST)

范例：

***NYASM*列表文件（.LST）实例**

```
Nyquest Technology Co., Ltd.
NYASM 1.00 Copyright(c) Nyquest Technology Co., Ltd. [Build:Dec 20 2007]

 File=E:\MyProjects\Build\asm\NYASM\Sample\NYASMSample.asm
Date=2007/12/20, 06:22:21 pm

ADDR     OPCODE/VALUE   LINE     TAG SOURCE STATEMENT                PAGE:1
                        0-0001       ; sample NYQUEST assembler source cod
```

```
                        0-0002      ;
                        0-0003      list    p=ny5c640b , c=off ,r=hex
            000000030   0-0004      ORG_OFF         equ     0x30
            000000000   0-0005      ORG_SUBOFF      equ     0x00
            000000030   0-0006      SUBPPTRADDR     equ     ORG_SUBOFF+ORG_OFF
                        0-0007      #include    "2102.h"
                        1-0001      ;
            000000010   0-0008      org     0x10
000010      D020        0-0009      mvma    0x20
000011      6030        0-0010      jmp     start
            000000030   0-0011      org     0x30
            000000030   0-0012      start:
000030      D030        0-0013      mvma    0x30
000031      0112        0-0014      mvat    0x12
                        0-0015      end

 NYASM 1.00 Copyright(c) Nyquest Technology Co., Ltd. [Build:Dec 20 2007]

 File=E:\MyProjects\Build\asm\NYASM\Sample\NYASMSample.asm
Date=2007/12/20, 06:22:21 pm

SYMBOL TABLE            TYPE        VALUE           PAGE:2
__NY5C640B              Constant    00000001
ORG_OFF                 Constant    00000030
ORG_SUBOFF              Constant    00000000
Start                   Label       00000030
SUBPPTRADDR             Constant    00000030
SOURCE FILE TABLE
000    E:\MyProjects\Build\asm\NYASM\Sample\NYASMSample.asm
001    E:\MyProjects\Build\asm\NYASM\Sample\2102.h

PROCESSOR       = NY5C640B (4 bits)
PROGRAM ROM     = 0x00000000 - 0x000FFFFF
DATA ROM        = 0x00000000 - 0x000FFFFF
SRAM / SFR      = 0x00000000 - 0x000000FF
```

列表文件的格式是由 *NYASM* 直接产生。产品名称和版别，编译的日期和时间，和页码将会显示在每一页的最上面。第一个栏位的数字代表着程序代码将被放在内存内部的地址。第二个栏位表示任何符号经由EQU、VARIABLE、CONSTANT或CBLOCK等虚指令所产生的32-bit值。第三个栏位被保留给机械指令用，这些指令将被九齐科技的MCU所执行。第四个栏位列出程序源码文件中对应的列数。剩下的栏

位保留给产生机械码的源代码。错误、警告和信息将被嵌入于来源列之间且是位于相关联的来源列的下面。符号表将列出所有被程序所定义的符号。

3.3.3 错误信息的文件格式（.ERR)

*NYASM*默认值将会产生一个错误文件，这个文件在对程序代码做除错时非常的实用。

错误文件中的信息格式如下：

[<type>] <file> (<line>) <number> <description>

范例：

[Error] C:\PROG.ASM 7 (133) w001: Symbol not previously defined (start)

附录B为*NYASM* 所产生的错误信息说明。

3.3.4 十六进制档格式（.HEX)

*NYASM*能够产生不同的hex文件格式。

3.3.5 符号和除错档格式（.DBG)

当*NYASM*被*NYIDE*所调用时，将会产生一个DBG文件给ICE做程序除错使用。

4 在窗口操作系统中使用 *NYASM*

这一章将用来介绍窗口版的*NYASM*。窗口版的*NYASM*可以被单独执行的窗口程序或是被九齐科技其他的开发工具所连结。举例像是*Q-Code*和*NYIDE*。

4.1 窗口界面

窗口版的*NYASM*对于在编译时的所有选项提供了一个图型化的界面给用户去设定编译时相关的选项。在窗口中点选*NYASM*.EXE文件执行即可。

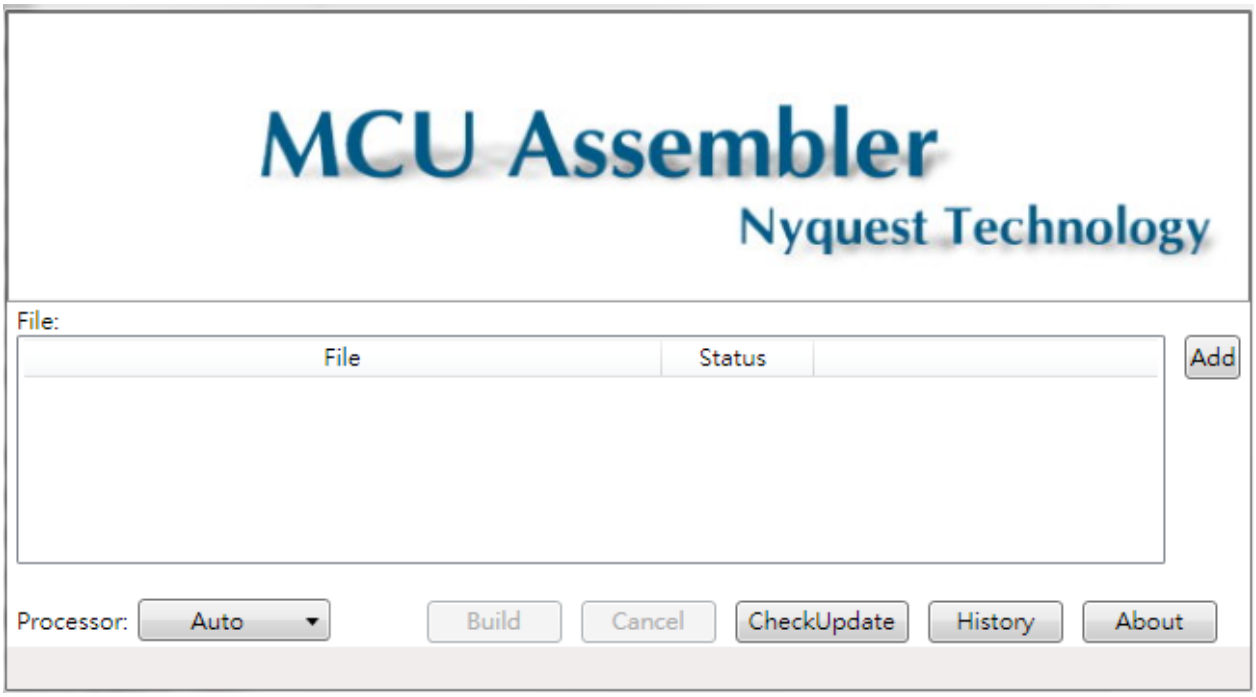

图 4-1 窗口界面

选择一个源码文件的方法有二：将文件拖拉至窗口内或者是按Add按钮。设定变量的选项说明如下。最后按下Build对源码文件做编译。

注意：当窗口版的 NYASM 被九齐科技其他的开发工具所引用时，则选项画面将不会出现。设定选项将会由其他的工具，以参数形式传递给 NYASM。

表格 4-1 汇编语言设定选项

选项	用法
Processor	取代源码文件的处理器设定。请参考A.2 MCU列表。

4.1 文字界面

*NYASM*除了图形用户界面之外，亦提供了文本模式界面。用户可以在指令稿（script）使用文字界面调用*NYASM*，进而将工作自动化。文字界面的执行文件为安装目录下的*NYASM*.exe。可使用的参数见下方表

表格 4-2可用选项

选项	用法
/o=<file>	指定输入的asm文件
/p=<icbody>	取代源码文件的处理器设定。请参考A.2 MCU列表。
/f=<file>	指定硬件组态配置文件。
/bypass	不显示图形界面，在编译完成后结束程序。
/unlockrsvmem	允许编辑保留内存区域。
/nocfgblk	在编译阶段忽略已存在的硬件组态配置文件。

5 伪指令

这一章将介绍*NYASM*的伪指令。伪指令是编译程序的命令，它是被加在源码文件的程序当中，但并不会直接转换成操作码。他是被用来控制编译程序的输入、输出和数据分配。

5.1 基本型态

*NYASM*提供了五种基本型态的伪指令：

- 控制型伪指令（Control Directives）：控制型伪指令允许有条件式的编译程序区块。
- 数据型伪指令（Data Directives）：数据型伪指令主要是控制内存配置且经由特殊的命名方式象征性的提供一种参考数据的方法。
- 打印型伪指令（Listing Directives）：打印型伪指令是控制*NYASM*产生列表文件格式的指令。他们允许的规格有标题、页码和其他打印的控制。
- 宏型伪指令（Macro Directives）：这些伪指令是控制在宏定义内部的执行与数据配置。

5.2 NY4、NY5、NY7、NY8A、NY9

5.2.1 伪指令摘要

表格 5-1含括了*NYASM*所提供的所有伪指令。章节后面将会专门的详细介绍*NYASM*所提供的伪指令。

表格 5-1伪指令集

伪指令	说明	语法
BREAK	从 **FOR**, **WHILE** 或 **REPEAT-UNTIL** 循环中跳离，或者从 **SWITCH** 区块中跳到 **SWITCH** 的最末端。	break [<Boolean expression>]
CASE	属于 **SWITCH** 区块的一部分，**CASE** 必须在 **SWITCH** 区块中使用。	switch <expression> case <expression 1>[,<expression 2>] <statements>
CBLOCK	定义一个常数区块。	cblock [<expr>]
CONSTANT	宣告符号常数。	constant
CONTINUE	跳至内部含有**CONTINUE**伪指令的**FOR**、**WHILE**或**REPEAT-UNTIL**循环的起始，在循环内**CONTINUE**后面的表达式都将会被忽略。	continue [<Boolean expression>]
DEFAULT	**SWITCH**区块的一部分，使用**SWITCH**区块时必须要有**DEFAULT**的条件式。**DEFAULT**为表示**SWITCH**判断式中的默认的编译区块。	default <statements>
#DEFINE	定义一个文字替代标记。	#define <name> [<value>] #define <name> [<arg>,...,<arg>]

伪指令	说明	语法
DW	宣告一个字符组(word)的数据。	[<label>] dw <expr>[,<expr>,...,<expr>]
DWS	编码文字为16 bit数据	[<label>:] dws "<string>"
ELSE	提供一个相对于**IF**的编译区块。*NYASM*在**IF**的编译区块与ELSE的编译区块二者之间只会选择一个做编译。	else <statements>
END	结束程序区段。	end
ENDC	结束一个常数区段。	endc
ENDFOR	结束一个**FOR**循环。	endfor
ENDIF	结束条件式编译区块。	endif
ENDM	结束一个宏定义。	endm
ENDS	提供一个方便管理的结束指令，可以使用在**ENDFOR, ENDW, ENDSW, ENDIF**。	ends
ENDSW	结束条件式**SWITCH**编译区块。	endsw
ENDW	结束一个**WHILE**循环。	endw
EQU	定义一个常数。	<label> equ <expr>
ERROR	发布一个错误信息。	error "<text_string>"
EXITM	从一个宏离开。	exitm
EXPAND	宏列表展开。	expand
EXTERN	外部符号。	extern <label>
FOR	执行**FOR**的循环计数。	for <iterator> = <expr1> to <expr2> [step <expr3>]
IF	条件式编译程序区块的起始。	if <expr>
IFDEF	假如符号已经有被定义就执行。	ifdef <label>
IFNDEF	假如符号没有被定义就执行。	ifndef <label>
#INCLUDATA	含括一个二进制数据文件。	#includata "<data_file>" [,<address>]
#INCLUDE	含括一个附加源码文件。	#include "<include_file>"
LINES	重新宣告每一页的列数。	lines <value>
LIST	列表的选项。	list [<list_option>,...,<list_option>]
LOCAL	宣告局部宏变量。	local <label>[,<label>]
MACRO	宣告宏定义。	<label> macro [<arg>,...,<arg>]
MAXMACRODEPTH	设定宏展开的最大层数。	Maxmacrodepth [=] <expr>
MESSG	产生用户定义的信息。	messg "<message_text>"
NEWPAGE	重新宣告每一页的列数。	Newpage <value>

伪指令	说明	语法
NOEXPAND	关闭宏的展开。	noexpand
ORG	设定程序的起始点。	[<label>:] org <expr>
ORGALIGN	对齐程序的起始点地址。	[<label>:] orgalign <expr>, <align>
RADIX	宣告设定数值格式。	radix <default_radix>
REPEAT	至少会执行一次循环。	Repeat <statements> until <Boolean expression>
SUBTITLE	指定程序副标题。	subtitle "<sub_text>"
SWITCH	条件式编译区块的起始。	switch <expr>
TITLE	指定程序标题。	title "<title_text>"
#UNDEFINE	删除一个替代的标记。	#undefine <label>
UNTIL	假如条件式成立就结束至少会执行一次的循环。	Repeat <statements> until <Boolean expression>
VARIABLE	宣告符号变量。	variable <label>[=<expr>,...,<label>[=<expr>]]
WHILE	**WHILE**所带的条件若是成立则执行循环。	while <expr>
.ALIGN2	对齐程序的起始点地址。	.align2 <expr>, <bit>

5.2.2 BREAK – 跳出目前所在的循环

◆ 语法

语法 1：

<for|while|repeat – loop begin>
    [<statements>]
    break [<Boolean expr>]
    [<statements>]
<for|while|repeat – loop end>

语法 2：

switch <expr>
    case <expr1>[,<expr2>]
    [<statements>]
    break [<Boolean expr>]
    [<statements>]
    [<case-statements>]
endsw

◆ 说明

从程序中WHILE、FOR或REPEAT-UNIT等循环中跳离目前正在执行的流程。Break也应用于switch区块当中，其作用是在条件式分支之间做切换。

◆ 范例

范例 1：

```
for i =0 to 4
nop
break i==2
halt
endfor
```

范例 2：

```
a=1
switch a
case 1, 2
nop
break
case 1
halt
endsw
```

◆ 请参阅

FOR, WHILE, REPEAT, SWITCH

5.2.3 CASE – 定义一个 SWITCH 选项

◆ 语法

```
switch <expr>
   case <expr1>[,<expr2>]
   [<statements>]
   :
   :
   default
   [<statements>]
endsw
```

◆ 说明

定义一个被选择的陈述句。一旦<expr>等于某一个在 case 后面的<exprN>，则目前执行的流程将会接续至 case 后面的项目。Case 是属于 switch 区块的一部分，且必须跟随着 switch 一起使用。

◆ 范例

```
a=1
switch a
    case 1, 2
    nop
    break
    case 1
    halt
endsw
```

◆ 请参阅

DEFAULT, SWITCH

5.2.4 CBLOCK – 定义一个常数区块

◆ 语法

```
cblock [<expr>]
[<label>[=<increment>][,<label>[=<increment>]]]
endc
```

◆ 说明

定义一个已被命名的常数列表。每一个标记<label>所被分配到的新数值将会递增于先前标记<label>的数值。这个伪指令的目的是在分配地址偏移量给很多个标记。在遇到 ENDC 这个伪指令时结束目前命名的列表。<expr>表示在这区块中第一个名称的起始值。假如没有<expr>，第一个名称的数值将会接续上一个 CBLOCK 最后一个名称的值往上加。假如在源码文件中的第一个 CBLOCK 没有<expr>，则将从 0 开始分配数值。假如<increment>有被指定，则下一个标记<label>将会分配到比先前标记<label>更高的<increment>的数值。多个命名可以写在同一列上，但需使用逗号作区隔。在程序及数据存储器中利用 cblock 定义常数是非常的实用。

◆ 范例

```
cblock 0x20                ; name_1 will be assigned 20
name_1, name_2             ; name_2 is 21
name_3 =0x30, name_4       ; name_4 is assigned 30,name_4 is assigned 31.
endc
```

◆ 请参阅

ENDC

5.2.5 CONSTANT – 宣告符号常数

◆ 语法

```
constant <label>=<expr> [...,<label>=<expr>]
```

◆ 说明

产生符号给 *NYASM* 表示使用。将符号用 CONSTANT 做宣告与使用。VARIABLE 做宣告的最主要差异在于使用 Constants 宣告作第一次初始后就不能再被复位并且在表达式<expr>在指定的同时就需要全部被处理。除此之外 constant 和 variables 在表达式中是可以交替使用。

◆ 范例

```
variable RecLength=64                    ; Set Default RecLength
constant BufLength=512                    ; Init BufLength
    :                                     ; RecLength may
:                                         ; be reset later
:                                         ; in RecLength=128
:                                         ;
constant MaxMem=RecLength+BufLength    ;CalcMaxMem
```

◆ 请参阅

VARIABLE

5.2.6 CONTINUE – 忽略后面的表达式并且从下一个循环开始

◆ 语法

```
<for|while|repeat – loop begin>
   [<statements>]
   continue [<Boolean expr>]
   [<statements>]
<for|while|repeat – loop end>
```

◆ 说明

在 WHILE、FOR 或 REPEAT-UNITL 循环区块当中使用 continue 设一个逻辑参考点，在这个 continue 逻辑参考点之后的述句将全部被忽略，然后跳至包含 continue 伪指令循环方块的起始。

◆ 范例

```
for i =0 to 4
        nop
        continue i==2
        halt
endfor
```

◆ 请参阅

FOR, WHILE, REPEAT

5.2.7 DEFAULT – 定义 SWITCH 中无条件式项目

◆ 符号

```
switch <expr>
```

```
case <expr1>[,<expr2>]
[<statements>]
:
:
default
[<statements>]
endsw
```

◆ **说明**

定义 SWITCH 中的一个无条件选项。一旦在 switch 后面的<expr>都没有与 case 后面的项目符合时，执行流程将会进入到 default 这个项目。Default 是 switch 区块的一部分，它必需与 switch 搭配使用。

◆ **范例**

```
a=1
switch a
case 1, 2
        nop
        break
        case 1
        halt
        default
nop
endsw
```

◆ **请参阅**

CASE, SWITCH

5.2.8 #DEFINE – 定义一个文字替代标记

◆ **语法**

```
#define <name> [<string>]
```

◆ **说明**

这个伪指令是定义一个文字替代字符串。在汇编程序中的所有<name>都将会被<string>所替代。使用#DEFINE 这个伪指令时若没有填入<string>，则所要定义的<name>将可能会被视为内部的批注并且作为 IFDEF 伪指令测试用。

◆ **范例**

```
#define length 20
#define control 0x19, 7
:
:
```

srbr control ; set bit 7 in 0x19

◆ 请参阅

#UNDEFINE, IFDEF, IFNDEF

5.2.9 DW – 宣告一个字符组的数据

◆ 语法

[<label>:] dw <expr>[,<expr>,...,<expr>]

◆ 说明

在程序内存中保留一段空间给字符组型态的数据使用。数值将被存进连续内存位置中且位置计数器是每次加一。表达式<expr>或许是文字字符串且被存放方式的说明请参照 DATA 伪指令的说明。

◆ 范例

dw 39, (d_list*2+d_offset)

dw diagbase-1

5.2.10 DWS – 编码文字为 16 bit 资料

◆ 语法

[<label>:] dws “<string>”

◆ 说明

在程序内存中保留一段空间给文字数据使用。两个字符为一组，以 little endian 保存于 16 bit 的 ROM。与 DW 不同的是，DW 将每一个字符单独存放，而 DWS 则是两两一组做编码。

这个指令只有在 16 bit 的 ic 有支持。

这个指令从 NYASM 2.80 开始提供。

◆ 范例

dws “abcdeAB”

;; ROM -> 6261 6463 4165 0042

5.2.11 ELSE – 相对于 IF 的编译区块

NYASM 会在 IF 的编译区块与 ELSE 的编译区块之间择一。

◆ 语法

else

◆ 说明

ELSE 需与 IF 伪指令搭配使用。它是在 IF 的判断式不成立时提供一个替代的程序区块。ELSE 在程序区块或宏里是经常被使用到。

◆ 范例

```
speed macro rate
If rate < 50
dw slow
else
dw fast
endif
endm
```

◆ **请参阅**

ENDIF, IF

5.2.12 END – 结束程序区段

◆ **语法**

```
end
```

◆ **说明**

表示程序结束。

◆ **范例**

```
list p= ny4b095a
:       ; executable code
:       ;
end ; end of instructions
```

5.2.13 ENDC – 结束一个常数区段

◆ **语法**

```
endc
```

◆ **说明**

将 ENDC 放在 CBLOCK 列表的末端做为结束。

◆ **请参阅**

CBLOCK

5.2.14 ENDFOR – 结束一个 For 的循环

◆ **语法**

```
endfor
```

◆ **说明**

ENDFOR 是做为 FOR 循环的结束。只要 FOR 伪指令的循环计数器被指定，在 FOR 与 ENDFOR 两个伪指令之间的程序代码将会不断的被展开并且连续的插入到源代码之间，它将会一直执行到条

件式所设定的边界时才会停止。ENDFOR 在程序区块或宏里是经常被使用到。

◆ **请参阅**

FOR

5.2.15 ENDIF – 结束条件式编译区块

◆ **语法**

endif

◆ **说明**

这个伪指令是做为条件式编译区块的结束。ENDIF 在程序区块或宏里是经常被使用到。

◆ **请参阅**

ELSE, IF

5.2.16 ENDM – 结束一个宏定义

◆ **语法**

endm

◆ **说明**

结束一个由 MACRO 开始的宏定义。

◆ **范例**

```
make_table macro arg1, arg2
    dw arg1, 0     ; null terminate table name
    res arg2  ; reserve storage
endm
```

◆ **请参阅**

MACRO, EXITM

5.2.17 ENDS – 通用结束指令

◆ **语法**

ends

◆ **说明**

使用在 ENDFOR, ENDW, ENDSW, ENDIF

◆ **请参阅**

ENDFOR, ENDW, ENDSW, ENDIF

5.2.18 ENDSW – 结束一个 Switch 的区块

◆ **语法**

endsw

◆ **说明**

结束一个由 SWITCH 开始的区块定义。

◆ **范例**

参考 SWITCH 的范例。

◆ **请参阅**

SWITCH

5.2.19 ENDW – 结束一个 While 的循环

◆ **语法**

endw

◆ **说明**

ENDW 作为 WHILE 循环的结束。只要在 WHILE 伪指令中所指令的条件式保持成立的状态，则在 WHILE 与 ENDW 两个伪指令之间的程序代码将会不断的被展开并且连续的插入到源代码之间。ENDW 在程序区块或宏里是经常被使用到。

◆ **范例**

参考 WHILE 的范例。

◆ **请参阅**

WHILE

5.2.20 EQU – 定义一个编译程序常数

◆ **语法**

<label> equ <expr>

◆ **说明**

标记<label>将以<expr>的值取代。

◆ **范例**

four equ 4 ; assigned the numeric value of 4 to label four

5.2.21 ERROR –发布一个错误信息

◆ **语法**

error "<text_string>"

◆ 说明

<text_string>是以相同的格式被打印到任何的 *NYASM* 错误信息。<text_string>可以包含字符数是 1 到 80 个。

◆ 范例

```
error_checking macro arg1
    if arg1 >= 55  ; if arg is out of range
    error "error_checking-01 arg out of range"
    endif
endm
```

◆ 请参阅

MESSG

5.2.22 EXITM – 从一个宏离开

◆ 语法

exitm

◆ 说明

在编译的时候立即强制从宏展开的函式中离开。这个伪指令与在编译时遇到 ENDM 伪指令有相同的效果。这个指令只有在 NY5+、NY6 系列可以使用。

◆ 范例

```
test macro filereg
    if filereg == 1 ; check for valid file
    exitm
    else
    error "bad file assignment"
    endif
endm
```

◆ 请参阅

ENDM MACRO

5.2.23 EXPAND – 展开宏打印

◆ 语法

expand

◆ 说明

在列表文件中展开所有的宏。这个伪指令的功用概略相等在编译程序上”Macro Expansion”的选项设

定。只是它能够在后面遇到 NOEXPAND 时，将展开的功能关闭。

◆ **请参阅**

MACRO, NOEXPAND

5.2.24 EXTERN – 外部符号

◆ **语法**

extern <label>

◆ **说明**

定义符号为公开(public symbol)，跨越不同模块使用符号时，必须定义符号为公开。比如说两个独立编译的 asm 文件呼叫对方定义的函数。只有在 NY8 C 语言项目可以使用。

5.2.25 FOR – 当设定值符合条件式时就执行 For 循环

◆ **语法**

```
for <iterator>=<expr1> to <expr2> [step <expr3>]
:
:
endfor
```

◆ **说明**

只要<iterator>等于<expr1>到<expr2>所设定的范围时，则在 FOR 与 ENDFOR 之间的所有程序将会被编译。一个 FOR 循环最大的循环数是 256 次。

◆ **范例**

```
for I=0 to 5
        nop
endfor
```

◆ **请参阅**

ENDFOR

5.2.26 IF – 一个有条件式可被编译的程序代码区块

◆ **语法**

if <expr>

◆ **说明**

开始执行一个条件式的编译区块。假如 <expr>判定为正确时，跟随在 IF 后面的程序代码将会立即被编译。否则在 IF 后面的程序代码将会被编译程序所跳过，一直到碰见 ELSE 或 ENDIF 伪指令。若<expr>最后的结果是等于零时则将被视为逻辑性的”正确”，反之若这个<expr>最后的结果是等于其他任意的值时，将被视为逻辑性的”错误”。在 IF 及 WHILE 等伪指令是将其表达式的结果视为一个逻辑性运算，”正确”表示将会回传一个非零的值，”错误”表示将会回传一个零值。

◆ **范例**

```
if version == 100; check current version
    :      ;executable cod
    :      ;executable cod
else
    :      ;executable cod
    :      ;executable cod
endif
```

◆ **请参阅**

ELSE, ENDIF

5.2.27 IFDEF – 假如符号已经被定义就执行

◆ **语法**

ifdef <label>

◆ **说明**

假如<label>在先前已经被定义过了，则条件式路径的内容就会被编译直到编译遇到 ELSE 或 ENDIF 时才会停止。先前的定义通常是用#DEFINE 伪指令来完成或者是在 *NYASM* 命令行上做设定。

◆ **范例**

```
#define testing 1 ; set testing "on"
:
:
ifdef testing
<execute test code> ; this path would be executed.
Endif
```

◆ **请参阅**

#DEFINE, ELSE, ENDIF, IFNDEF, #UNDEFINE

5.2.28 IFNDEF – 假如符号还没被定义就执行

◆ **语法**

ifndef <label>

◆ **说明**

假如<label>先前尚未被定义或者是已经被用伪指令#UNDEFINE 改成未定义的型态，则伪指令 IFNDEF 后面的程序码将被编译。编译将会被启动或关闭直到编译遇到 ELSE 或 ENDIF 时才会停止。

◆ **范例**

```
#define testing1 ; set testing on
:
:
#undefine testing1 ; set testing off
ifndef testing ; if not in testing mode
:   ; execute this path
:
endif
```

```
end ; end of source
```

◆ 请参阅

#DEFINE, ELSE, ENDIF, IFDEF, #UNDEFINE

5.2.29 #INCLUDATA – 含括一个二进制文件

◆ 语法

#includata "<binary_data_file>"[, address]

◆ 说明

将一个特殊文件读入当成二进制的数据。假如包含的数据文件需要插入到一个特殊的位置时，用户可以在 address 特别指定所要位置。#includata 必须是在伪指令 end 前面的最后一个陈述句。<binary_data_file> 必须置于引号内。假如有特别指定路径时，则编译程序只会到指定的路径去搜寻，否则将以源码文件所在的目录做搜寻。<binary_data_flie>在编译后将会变成一个标记。

◆ 范例

```
#includata "c:\music\s02.sog", 0x2000 ; insert data file at 0x2000
```

5.2.30 #INCLUDE – 含括额外的源码文件

◆ 语法

#include "<include_file>"

◆ 说明

将一个特殊文件读入当成源码文件的程序。在插入文件之后，编译程序将会重新开始编译原始源码文件。<include_file> 必须置于引号内。假如有特别指定路径时，则编译程序只会到指定的路径去搜寻。否则将以源码文件所在的目录做搜寻。

◆ 范例

```
#include "c:\sys\sysdefs.inc" ; system defs
#include “regs.h” ; register defs
```

5.2.31 LINES – 列表文件每一页的行数

◆ 语法

lines <value>

◆ 说明

设定列表文件中一页的最大行数。

◆ 请参阅

NEWPAGE

5.2.32 LIST – 打印选项

◆ 语法

list [<list_option>, ..., <list_option>]

◆ 说明

这个 LIST 伪指令和打开列表输出有相同的作用。此外，位于下表的 List 选项，都有个别控制编译的程序或输出列表文件的格式：

表格 5-2 List 伪指令选项

选项	默认状态	说明
c	Off	英文字母大小写区分打开/关闭 c=on 打开 c=off 关闭
p	None	设定处理器的类型： /p=<processor_type> 这里的<processor_type> 是指九齐科技的组件。例如：NY5A005A。
unlockrsvmem	Locked	/unlockrsvmem 只适用于4-bit MCU。允许编辑保留内存区域。
nocfgblk	Configuration Block required	/nocfgblk 只适用于4-bit MCU。在编译阶段忽略已存在的硬件组态设定。

◆ 范例

list p=ny5c640b, c=off

5.2.33 LOCAL – 宣告区域性的宏变量

◆ 语法

local <label>[,<label>...]

◆ 说明

在宏里宣告内部的标记，且仅在宏中具效力。在宣告时<label>可能与宏定义外的另一个标记是完全相同的；但二者之间并不会冲突。即使在巢状式的宏调用中，每个引用将有它自己局部的定义。

◆ 范例

```
<main code segment>
:
:
len equ 10 ; global version
size equ 20 ; note that a local variable may now be created and modified
test macro size
    local len, label ; local len and label
    len set size ; modify local len
```

```
        label res len ; reserve buffer
        len set len-20 ;
    endm ; end macro
```

◆ 请参阅

ENDM, MACRO

5.2.34 MACRO – 宣告一个宏定义

◆ 语法

<label> macro [<arg>, ..., <arg>]

◆ 说明

一个宏表示可以用单一个宏调用将一串序列的指令集插入到汇编程序档中。这个宏必须先被定义过，且可供后面的源程序码做索引。一个宏可以调用另一个宏或者是递归的调用自己本身。

◆ 范例

```
Read macro device, buffer, count
mvma device
mvma buffer
mvma count
endm
:
:
read 1,2,3
```

◆ 请参阅

ELSE, ENDIF, ENDM, EXITM, IF, LOCAL

5.2.35 MAXMACRODEPTH – 定义最大的宏层数

◆ 语法

maxmacrodepth[=]<expr>

◆ 说明

MAXMACRODEPTH 将宏的最大有效层数定义为<expr>。<expr>必须小于或等于最大的 256 层。

MAXMACRODEPTH 在源码文件中常被使用到。每次使用时就是重新定义宏最大的有效层数。

◆ 范例

```
list p=ny5c640b
maxmacrodepth 0x10
:
:
```

5.2.36 MESSG – 产生用户定义的信息

◆ 语法

messg "<message_text>"

◆ 说明

依据一个信息然后显示在列表文件中。发布一个 MESSG 伪指令不需要设定任何的错误回传码。

◆ 范例

```
mssg_macro macro
    messg "mssg_macro-001 invoked without argument"
endm
```

◆ 请参阅

ERROR

5.2.37 NEWPAGE – 在列表文件中换页

◆ 语法

newpage <value>

◆ 说明

在列表文件中换至新的一页继续打印。

◆ 请参阅

LINE

5.2.38 NOEXPAND – 关闭宏展开

◆ 语法

noexpand

◆ 说明

关闭在列表文件中宏展开的功能

◆ 请参阅

EXPAND

5.2.39 ORG – 设定程序的起始点

◆ 语法

[<label>:] org <expr>

◆ 说明

设定程序的起始点地址在<expr>。假如<label>有被指定，则将会给定这个<label>一个在<expr>所设定的值。假如没有使用 ORG，则程序代码将会在地址零的位置开始放。

◆ 范例

```
int_1: org 0x20
; Vector 20 code goes here
int_2: org int_1+0x10
; Vector 30 code goes here
```

5.2.40 ORGALIGN – 设定程序的起始点地址排列

◆ 语法

[<label>:] orgalign <expr>,<align>

◆ 说明

设定源码的起始位置<expr>|<align>。假如<label>是被指定的，则它将被给定一个<expr>|<align>的值。假使没有使用 ORGALIGN，源码将在地址零开始被产生。

◆ 范例

```
int_1: orgalign 0x20,0x7
```

◆ 请参阅

.align2

5.2.41 RADIX – 数值格式

◆ 语法

radix <default_radix>

◆ 说明

设定在数据表达式中数值格式的默认值。格式的默认值是 dec（十进制）。格式的有效设定值有：hex（十六进制）、dec（十进制）、oct（八进制）或是 bin（二进制）。

◆ 范例

```
radix dec
```

◆ 请参阅

LIST

5.2.42 REPEAT – 定义一个从 Repeat 至 Until 的循环方块

◆ 语法

```
repeat
:
:
until <expr>
```

◆ 说明

定义一个由 REPEAT 至 UNTIL 的循环区块。

◆ **范例**

```
test_mac macro count
   variable i
   i = 0
   repeat
      i += 1
   until i > count
endm
:
:
End
```

◆ **请参阅**

WHILE, UNTIL

5.2.43 SUBTITLE – 程序的副标题

◆ **语法**

subtitle "<sub_text>"

◆ **说明**

<sub_text> 是一个在双引号内部的 ASCII 字符串。字符串的最大长度是小于或等于 60 字。这个伪指令是建立第二个程序标题当作打印输出时的副标题。

◆ **范例**

```
subtitle "diagnostic section"
```

◆ **请参阅**

TITLE

5.2.44 SWITCH – 条件式的交换编译区块

◆ **语法**

```
switch <expr>
    case <expr1>[,<expr2>]
    [<statements>]
    case < exprM>[,<exprN>]
    :
    :
    default
    [<statements>]
endsw
```

◆ **说明**

开始执行条件式的交换编译区块。假如对<expr>判定的结果是与任何在 case 后的数值相符合，则

在 case 后的程序将会被编译。否则随后的程序代码将都会被编译程序省略跳过，直到碰到 default 或 ENDSW 伪指令为止。

◆ **范例**

```
a=1
switch a
    case 1, 2
    nop
    break
    case 1
    halt
    default
endsw
```

◆ **请参阅**

BREAK, DEFAULT

5.2.45 TITLE – 程序标题

◆ **语法**

title "<title_text>"

◆ **说明**

<title_text>是一个置于双引号内可被打印的 ASCII 字符串，且字符串数必须小于或等于 60 个字符，这个伪指令是将双引号内的文字置于列表文件中每一页的最上层。

◆ **范例**

```
title "operational code, rev 5.0"
```

◆ **请参阅**

SUBTITLE

5.2.46 #UNDEFINE – 删除一个替代的标记

◆ **语法**

#undefine <label>

◆ **说明**

<label>是一个先前使用#DEFINE 伪指令所定义的识别符号。它必须是一个对 *NYASM* 有效的标记。这个伪指令是将已经被命名的符号从符号表中被移除。

◆ **范例**

```
#define length 20
:
```

:

#undefine length

◆ 请参阅

#DEFINE, IFDEF, #INCLUDE, IFNDEF

5.2.47 UNTIL – 执行循环直到条件式成立

◆ 语法

repeat

:

:

until <expr>

◆ 说明

只要<expr>的判定结果是"否"，则在 REPEAT 及 UNTIL 之间的程序代码将至少会被编译过一次以上。一个 REPEAT 循环最大的重复次数为 256 次。

◆ 范例

```
test_mac macro count
    variable i
    i = 0
    repeat
    i += 1
    until i < count
endm
:
:
end
```

◆ 请参阅

WHILE, REPEAT

5.2.48 VARIABLE – 宣告一个符号变量

◆ 语法

variable <label>[=<expr>][,<label>[=<expr>]...]

◆ 说明

产生一个可以在 *NYASM* 表达式使用的符号。变量和常数在表达式中通常是可以交替的使用。要特别注意的是变量的值在运算时不会被更新，用户必须以等号(=)赋值，指定变量的改变。

◆ 范例

请参阅伪指令 CONSTANT 的范例说明。

◆ 请参阅

CONSTANT

5.2.49 WHILE – 当条件成立时就执行循环

◆ 语法

while <expr>

:

:

endw

◆ 说明

只要<expr>判定为真，在 WHILE 及 ENDW 之间的每一列将会被编译。若<expr>最后的结果是零时，被视为逻辑性的“错误”，反之若<expr>最后的结果是不等于零的任意数时，将被视为逻辑性的“正确”。“正确”的代表将会回传一个非零的值，“错误”表示将会回传一个零值。一个 WHILE 循环内部最多可以包含 100 列指令，循环重复次数最大到 256 次。

◆ 范例

```
test_mac macro count
variable i
i = 0
while i < count
movlw i
i += 1
endw
endm
start
test_mac 5
end
```

◆ 请参阅

ENDW IF

5.2.50 .ALIGN2 – 对齐程序地址

◆ 语法

.align2 <expr>, <bit>

◆ 说明

程序起始地址对齐<bit>个位，并且地址的低位为<expr>。

当我们需要地址为 0x41, 0x141, 0x241, 0x341, ……的时候，可以使用.align2 对齐 8 个位，并设定低位为 0x41。而 ORGALIGN 指令没办法指定位数，所以无法避免 0xC1 产生。

这个指令只有在 NY5+, NY6 有支持。

◆ **范例**

.align2 0x41, 8

◆ **请参阅**

ORGALIGN

5.3 NY8L

NY8L所使用的伪指令和其余系列皆不同，独立于本章节介绍。

5.3.1 伪指令摘要

表格 5-3含括了*NYASM* 所提供的所有伪指令。章节后面将会专门的详细介绍*NYASM*所提供的伪指令。

表格 5-3伪指令集

伪指令	说明	语法
.and	布尔 and 运算	<expr> .and <expr>
.bankbyte	取得 bank byte	.bankbyte(<expr>)
.bitand	位 and 运算	<expr> .bitand <expr>
.bitnot	位 not 运算	.bitnot <expr>
.bitor	位 or 运算	<expr> .bitor <expr>
.bitxor	位 xor 运算	<expr> .bitxor <expr>
.blank	参数是否为空	.blank(<symbol>)
.byte	低字节	.byte(<expr>)
.ceil	无条件进位	.ceil(<expr>)
.code	.segment “code”的简写	.code
.data	.segment “data”的简写	.data
.define	定义	.define <symbol> <expr>
.defined	查询是否有被定义过	.defined(<symbol>)
.else	条件式编译否则区块	.else
.elseif	条件式编译否则再判断区块	.elseif(<expr>)
.endif	结束条件式编译区块	.endif
.endmacro	结束宏定义区块	.endmacro

伪指令	说明	语法
.endrepeat	结束重复区块	.endrepeat
.endscope	结束变数范围区块	.endscope
.endstruct	结束结构区块	.endstruct
.equ	定义常数	<symbol> .equ <expr>
.error	产生一个编译错误	.error “<text>”
.export	导出符号	.export <symbol>
.exportzp	导出零页符号	.exportzp <symbol>
.extern	宣告全局符号	.extern <symbol>
.externzp	宣告全局零页符号	.externzp <symbol>
.floor	无条件舍去	.floor(<expr>)
.hibyte	高字节	.hibyte(<expr>)
.if	条件式编译	.if(<expr>)
.ifblank	条件编译若参数为空	.ifblank(<symbol>)
.ifdef	条件式编译若有定义	.ifdef(<symbol>)
.ifnblank	条件编译若参数不为空	.ifnblank(<symbol>)
.ifndef	条件式编译若无定义	.ifndef(<symbol>)
.import	导入符号	.import <symbol>
.importzp	导入零页符号	.importzp <symbol>
.incbin	插入二进制文件	.incbin “<file>”
.include	引用文件	.include “<file>”
.lobyte	低字节	.lobyte(<expr>)
.local	宏局部变量	.local <symbol>
.macro	定义宏	.macro <name> <arg1>, <arg2>, …
.mod	取余数运算	<expr> .mod <expr>
.not	布尔反向运算	.not <expr>

伪指令	说明	语法
.or	布尔或运算	<expr> .or <expr>
.org	设定程序起始点	.org <expr>
.repeat	重复展开	.repeat <expr>
.res	保留空间	.res <expr>, <expr>
.round	四舍五入	.round(<expr>)
.scope	变数范围区块	.scope <symbol>
.segment	程序片段	.segment “<symbol>”
.setcpu	设定 CPU	.setcpu <ic_body>
.shl	左移	<expr> .shl <expr>
.shr	右移	<expr> .shr <expr>
.string	取得字符串	.string(<symbol>)
.word	字組	.word <expr>
.xor	布尔互斥或	<expr> .xor <expr>

5.3.2 .AND – 布尔 AND 运算

◆ **语法**

<symbol> = <expr1> .and <expr2>

◆ **说明**

计算 expr1 && expr2

◆ **范例**

```
.if(0 .and 1)
    .error
.endif
```

5.3.3 .BANKBYTE – 取得 bank byte

◆ **语法**

.bankbyte(<expr>)

◆ **说明**

取得 <expr> 高位的 bank byte (bit 16~23)

◆ 范例

Label1_bank = .bankbyte(label1)

5.3.4 .BITAND – 位 AND 运算

◆ 语法

<symbol> = <expr1> .and <expr2>

◆ 说明

计算 expr1 & expr2

◆ 范例

Ans = 1 .bitand 3

; Ans = 1

5.3.5 .BITNOT – 位 NOT 运算

◆ 语法

<symbol> = .not <expr>

◆ 说明

将 expr 每个 bit 反向，这边位宽度是 32bit。若将一个 0 反向，将得到 32 bit 的 1。

◆ 范例

Ans = .bitnot 3

; Ans = 0xFFFFFFFC

5.3.6 .BITOR – 位 OR 运算

◆ 语法

<symbol> = <expr1> .bitor <expr>

◆ 说明

计算 expr1 | expr2

◆ 范例

Ans = 3 .bitor 6

; Ans = 7

5.3.7 .BITXOR – 位 XOR 运算

◆ 语法

<symbol> = <expr1> .xor <expr2>

◆ 说明

计算 expr1 互斥或 expr2

◆ 范例

Ans = 1 .xor 3

; Ans = 2

5.3.8 .BLANK – 参数是否为空

◆ 语法

.blank(<symbol>)

◆ 说明

会得到表示参数<symbol>是否为空的布尔值。用于宏之中可检查调用端是否有传递指定的参数。

◆ 范例

```
.macro M1 arg1
  .if ( .blank(arg1) )
    .error
  .endif
.endmacro
```

5.3.9 .BYTE – 低字节

◆ 语法

<symbol> = .byte(<expr>)

◆ 说明

取得 expr 的低位一个 byte.

◆ 范例

Ans = .byte(0x1234)

; Ans = 0x34

5.3.10 .CEIL – 无条件进位

◆ 语法

<symbol> = .ceil(<expr>)

◆ 说明

如果<expr> 是一个浮点数，则将之无条件进位到整数。

◆ 范例

Ans = .ceil(1.2)

; Ans = 2

5.3.11 .CODE – .segment “code” 的简写

◆ 语法

.code

◆ 说明

等同 .segment “code”

◆ 请参阅

.SEGMENT

5.3.12 .DATA – .segment “data” 的简写

◆ 语法

.data

◆ 说明

等同 .segment “data”

◆ 请参阅

.SEGMENT

5.3.13 .DEFINE – 定义

◆ 语法

.define <symbol> <expr>

◆ 说明

将表达式定义到符号，之后这个符号就代表着表达式。

◆ 范例

```
.define AAA 1 + 2
Ans = AAA
;Ans = 3
```

◆ 请参阅

.DEFINED

5.3.14 .DEFINED – 查询是否有被定义过

◆ 语法

.defined(<symbol>)

◆ 说明

若<symbol>曾经被定义过，则结果为1，否则为0。一般情形判断符号有无定义，使用 .ifdef <symbol> 即可。若要条件为多个符号皆有定义才算成立，势必使用巢状.ifdef。而.defined 正好可解决此种窘

境:

```
.ifdef (symbol1)
    .ifdef(symbol2)
        <statements>
    .endif
.endif
```

可改写成

```
.if (.defined(symbol1) && .defined(symbol2) )
    <statements>
.endif
```

◆ **范例**

```
.if ( .defined(def_name))
    .error
    ; 因为并没有定义 def_name 符号，此处程序不会被编译。
.endif
```

◆ **请参阅**

.DEFINE

5.3.15 .ELSE – 条件式编译否则区块

◆ **语法**

```
.if(<expr>)
    <statements>
.else
    <statements>
.endif
```

◆ **说明**

若 if 不成立则编译此区块

◆ **范例**

```
.if( 0 )
    .error
.else
    ; do something
.endif
```

◆ **请参阅**

.IF, .ELSEIF

5.3.16 .ELSEIF – 条件式编译否则再判断区块

◆ 语法

```
.if(<expr>)
    <statements>
.else if (<expr>)
    <statements>
.endif
```

◆ 说明

若先前的 if 或 elseif 皆不成立，则在判断 <expr>是否成立，成立则编译此区块。

◆ 范例

```
TempVar = 1
.if( TempVar < 1 )
    .error
.elseif(TempVar < 2)
    ; do something
.endif
```

◆ 请参阅

.IF, .ELSE

5.3.17 .ENDIF – 结束条件式编译区块

◆ 语法

```
.if(<expr>)
    <statements>
.else if (<expr>)
    <statements>
.endif
```

◆ 说明

结束由.if 开始的条件式编译区块

◆ 请参阅

.IF, .ELSE, ELSEIF

5.3.18 .ENDMACRO – 结束宏定义区块

◆ 语法

```
.macro <symbol> [<arg1> [,<arg2>…] ]
    <statements>
.endmacro
```

◆ 说明

结束由.macro 开始的宏定义区块。

◆ 请参阅

.macro

5.3.19 .ENDREPEAT – 结束重复区块

◆ 语法

```
.repeat <expr>
    <statements>
.endrepeat
```

◆ 说明

结束由.repeat 开始的重复区块。

◆ 请参阅

.repeat

5.3.20 .ENDSCOPE – 结束区块

◆ 语法

```
.scope <symbol>
    <statements>
.endscope
```

◆ 说明

结束由.repeat 开始的重复区块。

◆ 请参阅

.scope

5.3.21 .ENDSTRUCT – 结束结构区块

◆ 语法

.endstruct

◆ 说明

结束由.struct 开始的区块。

◆ 请参阅

.struct

5.3.22 .EQU – 定义常数

◆ 语法

<symbol> .equ <expr>

◆ 说明

定义一个常数<symbol>并赋值<expr>。<expr>必须是此时此刻能计算的常数值，常数一经定义则不可更动其值。

◆ 范例

MyInteger .equ 1

5.3.23 .ERROR – 产生一个编译错误

◆ 语法

.error [“<message>”]

◆ 说明

产生一个错误，若有指定错误信息则使用。错误信息必须为双引号引住的文字。

◆ 范例

.error “argument out of range”

5.3.24 .EXPORT – 导出符号

◆ 语法

.export <symbol>

◆ 说明

效果与.extern 相同。此项目仅是为了兼容性而建立的别名。

◆ 请参阅

.extern

5.3.25 .EXPORTZP – 导出零页符号

◆ 语法

.exportzp <symbol>

◆ 说明

效果与.externzp 相同。此项目仅是为了兼容性而建立的别名。

◆ 请参阅

.externzp

5.3.26 .EXTERN – 宣告外部符号

◆ 语法

.extern <symbol>

◆ 说明

宣告 symbol 为一个全局符号(Global symbol)。外部符号可以是本模块中定义的，亦可引用其他模块所定义的符号。多个模块做链接时，不可存在多个相同名称的全局符号。亦不可存在任何全局符号未曾赋值。

◆ **范例**

仅于多模块链接之情况，此指令方有意义。故以下范例将分为三个文件以便说明。

```
;----- header.h------
.ifndef HEADER_H
.define HEADER_H
.extern GLOBAL_LABEL
.endif

;------ module1.s ------
.include “header.h”
      jmp GLOBAL_LABEL ;jump to module2

;------ module2.s -----
.include “header.h”
GLOBAL_LABEL:
      nop
```

◆ **请参阅**

.externzp

5.3.27 .EXTERNZP – 宣告外部零页符号

◆ **语法**

.externzp <symbol>

◆ **说明**

宣告 symbol 为一个全局符号(Global symbol)，在指令使用到此符号时，优先使用零页寻址模式。为此符号赋值时，仅可给予零页范围内之值(0x00 ~ 0xFF)。赋值超出范围则编译程序报错。通常.externzp 用于全局的内存地址定义，而.extern 用于全局子程序定义。

◆ **请参阅**

.extern

5.3.28 .FLOOR –无条件舍去

◆ **语法**

<symbol> = .floor(<expr>)

◆ **说明**

如果<expr> 是一个浮点数，则将之无条件舍去到整数。

◆ 范例

```
Ans = .floor(1.2)
; Ans == 1
```

5.3.29 .HIBYTE – 高字节

◆ 语法

```
<symbol> = .hibyte( <expr> )
```

◆ 说明

取得 expr 的高位一个 byte. (bit 8~15)

◆ 范例

```
Ans = .hibyte(0x1234)
; Ans == 0x12
```

5.3.30 .IF – 条件式编译

◆ 语法

```
.if(<expr>)
    <statements>
.endif
```

◆ 说明

如果<expr>成立则编译此区块。

<expr>通常为布尔型态表达式，例如 a==b。

◆ 范例

```
Tmp = 1 + 2 * 3
.if(Tmp != 7)
    .error
.endif
```

5.3.31 .IFBLANK – 条件编译若参数为空

◆ 语法

```
.ifblank(<symbol>)
    <statements>
.endif
```

◆ 说明

此指令为 .if(.blank(<symbol>))的简写。

◆ 请参阅

.blank

5.3.32 .IFDEF – 条件式编译若有定义

◆ 语法

```
.ifdef(<symbol>)
    <statements>
.endif
```

◆ 说明

若<symbol>已定义则编译此区块。

◆ 范例

```
.ifdef(UNDEFINE_SYM)
    .error
.endif
```

◆ 请参阅

.if, .defined

5.3.33 .IFNBLANK – 条件式编译若参数不为空

◆ 语法

```
.ifnblank(<symbol>)
    <statements>
.endif
```

◆ 说明

此指令为 .if(！.blank(<symbol>))之简写。

◆ 请参阅

.blank

5.3.34 .IFNDEF – 条件式编译若无定义

◆ 语法

```
.ifndef(<symbol>)
    <statements>
.endif
```

◆ 说明

若<symbol>未曾定义则编译此区块。

◆ 范例

```
.ifdef(UNDEFINE_SYM)
    .error
.endif
```

◆ 请参阅

.if, .defined

5.3.35 .IMPORT – 导入符号

◆ 语法

.import <symbol>

◆ 说明

效果与.extern 相同。此项目仅是为了兼容性而建立的别名。

◆ 请参阅

.extern

5.3.36 .IMPORTZP – 导出零页符号

◆ 语法

.importzp <symbol>

◆ 说明

效果与.externzp 相同。此项目仅是为了兼容性而建立的别名。

◆ 请参阅

.externzp

5.3.37 .INCBIN – 插入二进制文件

◆ 语法

.incbin “<file>”

◆ 说明

以二进制的方式将插入文件<file>之内容。与.include 的差别是，.incbin 直接使用目标文件内容，而.include 会以文字的方式解译目标文件内的指令。.incbin 通常使用于音效、图形等二进制文件。

◆ 范例

```
L_RES_Voice1:
.incbin “d:\abc\voice1.v8lx”
```

5.3.38 .INCLUDE – 引用文件

◆ 语法

.include “<file>”

◆ 说明

<file>必须为另一个编译原始档，编译程序将会暂停目前文件的编译，开始编译<file>文件，并在<file>结束后回到目前位置。

◆ 范例

```
---- a1.h ----
.ifndef A1_H
.define A1_H
; content
.extern G_Func1

.endif

---- a1.s ----
.include “a1.h”
G_Func1:
ret
```

5.3.39 .LOBYTE – 低字节

◆ 语法

<symbol> = .lobyte(<expr>)

◆ 说明

取得 expr 的低字节一个 byte. (bit 0~7)

◆ 范例

```
Ans = .lobyte(0x1234)
; Ans == 0x34
```

5.3.40 .LOCAL – 宏区域变数

◆ 语法

.local <symbol>

◆ 说明

在宏里宣告内部的标记，且仅在宏中具效力。在宣告时<label>可能与宏定义外的另一个标记是完全相同的；但二者之间并不会冲突。即使在巢状式的宏呼叫中，每个引用将有它自己局部的定义。

◆ 范例

```
.macro M_x1
    .local LL_exit
    jmp LL_exit
```

```
LL_exit:
.endmacro
```

5.3.41 .MACRO – 定义宏

◆ 语法

```
.macro <symbol> [<arg1>, <arg2>, …]
    <statement>
.endmacro
```

◆ 说明

定义<symbol>为宏，尔后使用此符号则展开这段宏。宏可以有参数，调用宏所使用的参数数目必须符合宏定义之数目。

◆ 范例

```
.macro M_LDXY arg_value_x, arg_value_y
    LDX #arg_value_x
    LDY #arg_value_y
.endmacro
```

5.3.42 .MOD – 取余数运算

◆ 语法

```
<symbol> = <expr1> .mod <expr2>
```

◆ 说明

计算 expr1 / expr2 的余数

◆ 范例

```
ans = 5 .mod 3
; ans == 2
```

5.3.43 .NOT – 布尔反向运算

◆ 语法

```
<symbol> = .not <expr1>
```

◆ 说明

计算 expr1 的反向

◆ 范例

```
ans = .not 1
; ans == 0
```

5.3.44 .OR – 布尔或运算

◆ 语法

<symbol> = <expr1> .or <expr2>

◆ 说明

计算 expr1 || expr2

◆ 范例

ans = 0 .or 1

; ans == 1

5.3.45 .ORG – 设定程序起始点

◆ 语法

.org <expr>

◆ 说明

开始一个新的 segment，并且定位到<expr>计算出来的地址。

◆ 范例

```
.org 0x7e0
    .word L_TM2_INT
.code
L_TM2_INT:
    RTI
```

5.3.46 .REPEAT – 重复展开

◆ 语法

```
.repeat <expr>
    <statement>
.endrepeat
```

◆ 说明

重复编译<statement>多次，次数由<expr>指定。

◆ 范例

```
.org 0x7e0
    .word L_TM2_INT
.code
L_TM2_INT:
```

RTI

5.3.47 .RES – 保留空间

◆ 语法

.res <expr1>, <expr2>

◆ 说明

保留<expr1>指定大小的空间，并填入<expr2>所指定的值。

◆ 范例

```
; Reserve 12 bytes of memory with value $AA
.res 12, $AA
```

5.3.48 .ROUND – 四舍五入

◆ 语法

.round(<expr>)

◆ 说明

取得<expr>四舍五入到整数的值。

5.3.49 .SCOPE – 程序区域

◆ 语法

```
.scope <symbol>
    <statements>
.endscope
```

◆ 说明

开始一个指定名称的程序区域。在.scope 到.endscope 所包含的范围内，新定义的符号可以直接在内部存取。唯有在外部使用符号，须加上前缀字。

scope 的名称不可和其余符号冲突。

◆ 范例

```
.scope Error      ; Start new scope named Error
    None = 0
    File = 1
    Parse = 2
.endscope         ; close scope
LDA #Error::File   ; use symbol from scope Error
```

◆ 请参阅

.endscope

5.3.50 .SEGMENT – 程序片段

◆ 语法

.segment “<symbol>”

◆ 说明

切换到另一个程序片段。.segment 指令必须给一个字符串参数为 segment 名称。可用的名称与所选 IC 相关，请查阅指定 IC 文件。

◆ 范例

```
.segment “tm0_int”
      .word L_tm0_int
.code
L_tm0_int:
```

◆ 请参阅

.code

5.3.51 .SETCPU – 指定 CPU

◆ 语法

.setcpu <symbol>

◆ 说明

可以在文件的最前端标明 IC Body，不可以多次切换 IC Body。

◆ 范例

.setcpu NY8L030A

5.3.52 .SHL – 左移

◆ 语法

<expr1> .shl <expr2>

◆ 说明

计算<expr1> 左移 <expr2>的结果。

◆ 范例

```
Result = 2 .shl 1
; Result = 4
```

5.3.53 .SHR – 右移

◆ 语法

<expr1> .shr <expr2>

◆ 说明

计算<expr1> 右移 <expr2>的结果。

◆ 范例

```
Result = 2 .shr 1
; Result = 1
```

5.3.54 .STRING – 取得字符串

◆ 语法

.string(<symbol>)

◆ 说明

得到<symbol>符号所定义的字符串。

通常用于在 macro，可得到参数带入的字符串。

◆ 范例

```
.macro M_inc_v8lx name
      .incbin .string(name)
.endmacro
```

5.3.55 .WORD – 字組

◆ 语法

.word <expr1>

◆ 说明

在目前位置写入一个字组(2 字节)的数据，内容为 <expr1>。

◆ 范例

```
.word 0x12EF
```

5.3.56 .XOR – 布尔互斥或

◆ 语法

<expr1> .xor <expr2>

◆ 说明

计算<expr1> 互斥或 <expr2>的结果。

◆ 范例

```
Result = 0 .xor 1
; Result = 1
```

6 宏指令

宏内容为用户定义的程序代码，在编译程序时会用程序代码替换宏名称。

宏定义的本身还可以包含一个连续的汇编语言及伪指令，使用宏在汇编程序编码上具有较佳的可阅读性及弹性。使用宏的优点如下：

- 具高级语言的概念，可增加程序的可阅读性及可靠性。
- 对于被使用性较高的功能有统一固定的解决方案。
- 较易修正。
- 增加可测试性。

应用的范围可能包含了用来产生一个复杂的表、经常被使用的程序代码、和较复杂的运算。

6.1 宏指令 for NY4、NY5、NY7、NY8A、NY9

6.1.1 宏语法

*NYASM*的宏是依照下列的语法做定义：

<label> macro [<arg1>,<arg2> ..., <argn>]

:

:

endm

这里的<label>是一个在编译时有效的标记名称而且<arg>对于宏来说是一个可为任意个数的选项。在宏被展开的同时将会依相对应的参数位置指定这些参数的值。宏本身可以包含*NYASM*伪指令或是*NYASM*宏伪指令（例如LOCAL）。参考第5.2章，*NYASM*将会对宏本身做展开…等处理，直到*NYASM*碰到EXITM或ENDM这两个伪指令。

注意：在宏当中是不允许向前引用。

6.1.2 宏伪指令

这些伪指令只适于在宏本身内部使用。（请参照第5.2.1章，有关于这些指令的详细说明）：

- MACRO
- LOCAL
- EXITM
- ENDM

*NYASM*宏本身支持上述的伪指令及其他的伪指令。

6.1.3 文字替代

在宏本身的内部可以有字符串的取代及表达式的判定。宏所带的参数可以在宏本身内部的任何一个地方引用。

命令	说明
<arg>	参数的文字将在宏展开时做替代。

```
define_table macro num_of_entry
        local a = 0
        while a < num_of_entry
        dw 0
        a += 1
        endw
endm
```

when invoked, would generate:

```
dw 0    ; 1st
dw 0    ; 2nd
  :
  :
dw 0    ; (num_of_entry-1)-th
dw 0    ; (num_of_entry)-th
```

6.1.4 宏的用法

一个宏已经被事先定义过后，即可在程序来源模块当中的任何一个位置使用宏调用来引用，如下所述：

<macro_name> [<arg>, ..., <arg>]

这里的<macro_name>是一个先前已经被定义过的宏名称。而参数则是视实际需求再加入，参数之间彼此以逗号分隔。宏调用自己将不会占用到任何的内存位置。然而宏将会从目前的内存位置开始展开。EXITM与ENDM这两个伪指令是被用来终止目前宏展开，(参考第5章)。在宏展开的时候，伪指令EXITM将会暂停目前宏展开的动作，而在伪指令EXITM与ENDM之间的所有程序代码将会被忽略。假如是巢状式的宏，则EXITM将会终止目前这一层宏的展开，回到上一层宏展开。

6.2 NY8L

6.2.1 宏语法

NY8L的宏是依照下列的语法做定义：

```
.macro <label> [<arg1>,<arg2> ..., <argn>]
:
:
.endmacro
```

这里的<label>是一个在编译时有效的标记名称而且<arg>对于宏来说是一个可为任意个数的选项。在宏被展开的同时将会依相对应的参数位置指定这些参数的值。宏本身可以包含*NYASM* 伪指令或是

NYASM 宏伪指令(例如.LOCAL)。参考第5.3.章，*NYASM* 将会对宏本身做展开…等处理，直到*NYASM* 碰到.ENDMACRO伪指令。

注意：在宏当中是不允许向前引用。

6.2.2 宏伪指令

这些伪指令只适于在宏本身内部使用。(请参照第5.3章，有关于这些指令的详细说明)：

.MACRO – 定义宏

.ENDMACRO – 结束宏定义区块

.LOCAL – 宏区域变数

.IFBLANK – 条件编译若参数为空

.IFNBLANK – 条件式编译若参数不为空

.BLANK – 参数是否为空

NYASM 宏本身支持上述的伪指令及其他的伪指令。

6.2.3 文字替代

在宏本身的内部可以有字符串的取代及表达式的判定。宏所带的参数可以在宏本身内部的任何一个地方引用。

命令	说明
<arg>	参数的文字将在宏展开时做替代。

◆ **范例**

```
.macro CAJE value, label
        cmp     #value
        jz  label
.endmacro
```

6.2.4 宏的用法

一个宏已经被事先定义过后，即可在程序来源模块当中的任何一个位置使用宏调用来引用，如下所述：

<macro_name> [<arg>, ..., <arg>]

这里的<macro_name>是一个先前已经被定义过的宏名称，而参数则是视实际需求再加入，参数以逗号分隔。宏调用将不会占用到任何的内存位置。然而宏将会从目前的内存位置开始展开。宏调用时使用的参数可以少于定义时的参数，并可藉由.blank 检查调用端是否有传递指定的参数。

7 表达式的语法与运算

这一章将说明在 *NYASM* 中使用不同的表达式格式、语法及运算。

7.1 NY4、NY5、NY7、NY8A、NY9

内容：

7.1.1 数字常数和格式

NYASM 支持下列的数字格式：十六进制、十进制、八进制和二进制。数字系统在没有特别去变更时将会使用默认值做为目前数字系统，默认为十进制格式。数字系统将会决定机械码档上常数值。*NYASM* 只支持无号数。

以下表格提供不同种类数值型态表示：

表格 7-1数值格式

类型	语法	范例
十进制	D'<digits>'	D'100'
十六进制	H'<hex_digits>' 0x<hex_digits> <hex_digits>h	H'9f' 0x9f 9fh
八进制	O'<octal_digits>'	O'777'
二进制	B'<binary_digits>' <binary_digits>b	B'00111001' 00111001b

表格 7-2算术操作数及优先级

操作数		范例
$	取得程序计数器	goto $ + 3
(	左括号	1 + (d * 4)
)	右括号	(Length + 1) * 256
!	反逻辑 (逻辑补码)	if ! (a == b)
-	负 (二的补码)	-1 * Length
~	一的补码	flags = ~flags
high	取得24位值的最高位值	mvma high(0x121314) ;accumulator will contain 0x12
mid	取得24位值的中间位值	mvma mid(0x121314) ;accumulator will contain 0x13

操作数		范例
low	取得24位值的最低位值	mvma low(0x121314) ;accumulator will contain 0x14
high0	取得24位值的最高位的低位4位值	mvma high0(0x123456) ;accumulator will contain 0x2
high1	取得24位值的最高位的高位4位值	mvma high1(0x123456) ;accumulator will contain 0x1
mid0	取得24位值的中间位的低位4位值	mvma mid0(0x123456) ;accumulator will contain 0x4
mid1	取得24位值的中间位的高位4位值	mvma mid1(0x123456) ;accumulator will contain 0x3
low0	取得24位值的最低位的低位4位值	mvma low0(0x123456) ;accumulator will contain 0x6
low1	取得24位值的最低位的高位4位值	mvma low1(0x123456) ;accumulator will contain 0x5
*	算术乘	a = b * c
/	算术除	a = b / c
%	算术余	entry_len = tot_len % 16
+	算术加	tot_len = entry_len * 8 + 1
-	算术减	entry_len = (tot - 1) / 8
<<	位左移	flags = flags << 1
>>	位右移	flags = flags >> 1
>=	大或等于	if entry_idx >= num_entries
>	大于	if entry_idx > num_entries
<	小于	if entry_idx < num_entries
<=	小或等于	if entry_idx <= num_entries
==	等于	if entry_idx == num_entries
!=	不等于	if entry_idx != num_entries
&	位与	flags = flags & ERROR_BIT
^	位互斥或	flags = flags ^ ERROR_BIT
|	位或	flags = flags | ERROR_BIT
&&	逻辑与	if (len == 512) && (b == c)
||	逻辑或	if (len == 512) || (b == c)
=	设值	entry_index = 0
+=	加并设值	entry_index += 1
-=	减并设值	entry_index -= 1
*=	乘并设值	entry_index *= entry_length
/=	除并设值	entry_total /= entry_length

操作数		范例
%=	求余并设值	entry_index %= 8
<<=	位左移并设值	flags <<= 3
>>=	位右移并设值	flags >>= 3
&=	位与并设值	flags &= ERROR_FLAG
|=	位或并设值	flags |= ERROR_FLAG
^=	位互斥或并设值	flags ^= ERROR_FLAG
++	递增量为1	i ++
--	递减量为1	i --

7.1.2 高位/中位/低位

◆ **语法**

high <operand>

mid <operand>

low <operand>

◆ **说明**

这些操作数是被用来对一个多位的标记值取其中一个位做回传。这是被用来做动态指针的计算，也可能被用来当对一个目录的读取和写入指令。

7.1.3 增量/减量 (++/--)

◆ **语法**

<variable>++

<variable>--

◆ **说明**

为变量的值做增加或减少。这些操作数是自己独立一行且是以自己当作操作数。他们不可以再被加入到其他的表达式当中。

◆ **范例**

```
LoopCount = 4
while LoopCount > 0
   nop
   LoopCount --
Endw
```

7.2 NY8L

内容：

7.2.1 数字常数和格式

NYASM 支持下列的数字格式：十六进制、十进制、八进制和二进制。数字系统在没有特别去变更时将会使用默认值做为目前数字系统，默认为十进制格式。数字系统将会决定机械码档上常数值。*NYASM* 只支持无号数。

以下表格提供不同种类数值型态表示：

表格 7-3数值格式

类型	语法	范例
十进制	<digits>	100
十六进制	$<hex_digits>’ 0x<hex_digits>	$9f 0x9f
二进制	%<binary_digits>’	%00111001

表格 7-4算术操作数及优先级

操作数		范例
(	左括号	1 + (d * 4)
)	右括号	(Length + 1) * 256
!	反逻辑 (逻辑补码)	if ! (a == b)
-	负 (二的补码)	-1 * Length
~ .bitnot	一的补码	flags = ~flags
.bankbyte	取得24位值的最高位值	mvma high(0x121314) ;accumulator will contain 0x12
.hibyte	取得24位值的中间位值	mvma mid(0x121314) ;accumulator will contain 0x13
.lobyte	取得24位值的最低位值	mvma low(0x121314) ;accumulator will contain 0x14
*	算术乘	a = b * c
/	算术除	a = b / c
%	算术余	entry_len = tot_len % 16
+	算术加	tot_len = entry_len * 8 + 1
-	算术减	entry_len = (tot - 1) / 8

操作数		范例
<<	位左移	flags = flags << 1
>>	位右移	flags = flags >> 1
>=	大或等于	if entry_idx >= num_entries
>	大于	if entry_idx > num_entries
<	小于	if entry_idx < num_entries
<=	小或等于	if entry_idx <= num_entries
==	等于	if entry_idx == num_entries
!=	不等于	if entry_idx != num_entries
& .bitand	位与	flags = flags & ERROR_BIT
^	位互斥或	flags = flags ^ ERROR_BIT
| .bitor	位或	flags = flags | ERROR_BIT
&& .and	逻辑与	if (len == 512) && (b == c)
|| .or	逻辑或	if (len == 512) || (b == c)
.round	四舍五入	.round(2.345)
.ceil	无条件进位	.ceil(2.345)
.floor	无条件舍去	.floor(2.345)

7.2.2 高位/中位/低位

◆ **语法**

.bankbyte <operand>

.hibyte <operand>

.lobyte <operand>

◆ **说明**

这些操作数是被用来对一个多位的标记值取其中一个位做回传。这是被用来做动态指针的计算，也可能被用来当对一个目录的读取和写入指令。

8 改版记录

版本	日期	内 容 描 述	修正页
1.00	2007/12/20	新发布。	-
1.01	2009/10/12	改版。	-
1.1	2010/01/11	新增 NY4 系列 MCU。	58
1.2	2010/07/20	1. 新增 NY4/NY5 系列新母体。 2. 新增中文警告/错误信息。	59 63
1.3	2010/08/17	相容于 Windows 7。	12
1.4	2012/02/29	修改 NY5B/5C 系列 MCU。	59
1.5	2013/06/25	1. 使用 *NYASM* 需搭配 Windows XP 以上操作系统。 2. 新增 NY4(B)与 NY7 IC 系列 MCU 列表。	12 59
1.6	2013/08/16	修改 NY7 系列 MCU 列表。	61
1.7	2014/02/24	1. 修改 MACRO 程序范例。 2. 修改默认数值系统为十进制。 3. 修改警告与错误信息。 4. 词汇表新增 Forward Reference 向前参考解释。	40 42, 50, 70 62 69
1.8	2014/05/16	新增 NY8 系列 MCU。	60
1.9	2014/11/17	变更 MCU 列表 IC 母体：NY4B018C、NY4B038C、NY4B058C。 NY5C158C、NY5C185C、NY5C345C。	58
2.0	2015/01/29	1. 新增二进制数表示法。 2. 修改位运算范例。	50 50
2.1	2015/07/27	修改 UI 介绍。	18
2.2	2015/11/20	增加 NY9UB 系列 MCU。	61
2.3	2016/02/03	1. 移除 NY4xxxxA/NY5xxxxA 系列 MCU，仅保留 NY5AxxxA 系列。 2. 增加 NY8 系列 MCU。	- 63

版本	日期	内 容 描 述	修正页
2.4	2016/05/20	1. 增加 NY6 系列 MCU。 2. 增加 NY8A051C/51D MCU。	60 63
2.5	2016/08/22	增加 NY8A053D MCU。	63
2.6	2016/11/18	1. 移除 NY5AA 系列。 2. 增加 NY9UP01A。	- 79
2.7	2017/05/23	1. 支持 NY8L 系列。 2. 新增 NY8A054A MCU。 3. 新增 NY8L MCU。	42, 62, 67 78 78
2.8	2017/08/09	增加 NY8A054D MCU。	78
2.9	2017/11/17	1. 修正 List 指令说明。 2. 增加 NY8A051E MCU。	34, 72 79
3.0	2018/02/08	新增 NY8B062D MCU。	79
3.1	2018/08/27	1. 删除 DT 指令。 2. 移除 NY8A057A、NY8B073A、NY8B074A MCU。 3. 移除 NY6C450A ~ NY6C720A MCU。	- - -
3.2	2018/11/21	新增 NY8B062A MCU。	78
3.3	2019/02/19	新增 NY8A051F、NY9UP08A MCU。	77, 78
3.4	2019/05/28	新增 NY5P025J、NY5P055J、NY5P085J、NY5B035C、NY5B045C、NY8A050D、NY8AE51D、NY8B062B MCU。	76
3.5	2019/08/22	移除 NY8L005A、NY8L010A，并新增 NY8LP10A、NY8LP11A。	-
3.6	2019/11/14	新增 NY5P 系列、NY5A018C、NY5A025C、NY8BM72A MCU。	75
3.7	2020/03/16	新增 NY5AC 系列、NY5BC 系列 MCU。	75
3.8	2020/08/18	1. 增加.bitor 与.word 指令说明。 2. 增加 NY6P 系列、NY8A054E、NY8B061D。	42, 58 76

版本	日期	内 容 描 述	修正页
3.9	2020/11/12	1. 移除 NY4B018B / NY5AxxxB / NY5BxxxB / NY5C112B / 132B / 158B / 185B/ 225B / 265B / 305B / 345B。 2. 新增 NY8A053E / NY9UP02A。	- 79
4.0	2021/01/27	1. 移除 NY6A003A / NY6A005A / NY8L050A。 2. 新增 NY8B062E / NY8TM52D。	- 80
4.1	2021/05/18	新增 NY5QxxxA / NY8B060E / NY8BE62D 。	77
4.2	2021/09/10	1. 删除 DN、DB、DATA 指令。 2. 新增 NY5Q020A。	- 76
4.3	2021/11/11	新增 NY8TE64A。	79
4.4	2022/02/22	新增 NY5Q026A、NY5Q046A、NY5Q080A、NY5Q160A、NY8A051H、NY8AE51F。	76
4.5	2022/05/19	1. 新增 Win11 。 2. 新增 NY8B060D。	8 79
4.6	2022/08/24	新增 NY8B061E。	79
4.7	2022/11/28	新增 NY4P045C、NY8B062F。	74
4.8	2023/02/15	新增 NY4P018C、NY4P065C、NY4P085C、NY4P105C、NY8A050E。	74
4.9	2023/05/15	修正说明错误。	-
5.0	2023/08/21	增加 NY8A052E。	78
5.1	2024/02/22	1. 增加.align2 指令说明。 2. 增加 NY8BM61D、NY8BM62D。 3. 移除 NY8L020A、NY8L030A。	38 80 -
5.2	2024/08/22	删除 NY5P520J、NY5P720J、NY5P1K0J、NY5P1K2J、NY5C450B、NY5C520B 、NY5C640B 、NY5C720B 、NY7C450A 、NY7C520A 、NY8A051A 、NY8A051C 、NY8A051E 、NY8A053A 、NY8AE51D 、NY8B060E、NY8B061D、NY8B071A。	-

版本	日期	内 容 描 述	修正页
5.3	2025/02/27	新增 NY5Q019A、NY5Q039A、NY5Q079A、NY5Q159A、NY8A051J、NY8A051K、NY8A051L、NY8LP08A。	78, 80, 81
5.4	2025/05/27	1. 增加伪指令说明。 2. 增加 NY8A051H1、NY8B062F1、NY8BM84A。	27 80, 81
5.5	2025/08/31	增加 NY8A054E1、NY8F2481。	80, 81
5.6	2025/11/25	1. 增加 DWS 虚指令。 2. 增加 NY8A050E1、NY8F1141、NY8F1241。	23 80, 81

附录A - 快速索引

在使用 *NYASM* 开发系统时可以利用本附录所提供的简略数据做参考。

内容：

A.1 *NYASM* 快速参考

下列的快速参考指南包含了*NYASM*编译程序所有的指令、伪指令和命令行选项。

表格 A-1 伪指令集

伪指令	说明	语法
控制型伪指令		
CONSTANT	宣告符号常数。	constant <label>[=<expr>,...,<label>[=<expr>]]
#DEFINE	定义一个文字替代标记。	#define <name> [<value>] #define <name> [<arg>,...,<arg>]
END	结束程序区段。	end
EQU	定义一个编译常数。	<label> equ <expr>
ERROR	发布一个错误信息。	error "<text_string>"
#INCLUDATA	含括一个二进制资料文件。	#includata "<data_file>" [,<address>]
#INCLUDE	含括一个附加的源码文件。	#include "<include_file>"
LIST	打印的选项。	list [<list_option>,...,<list_option>]
MESSG	产生用户定义的信息。	messg "<message_text>"
ORG	设定程序的起始点。	[<label>:] org <expr>
LINES	重新宣告每一页的列数。	lines <value>
NEWPAGE	可在产生的List File中产生新的一页。	Newpage <value>
RADIX	宣告数字格式。	radix <default_radix>
SUBTITLE	程序副标题。	subtitle "<sub_text>"
TITLE	程序标题。	title "<title_text>"
#UNDEFINE	删除一个替代标记。	#undefine <label>
VARIABLE	宣告符号变量。	variable <label>[=<expr>,...,<label>[=<expr>]]
条件式编译指令		
BREAK	从 **FOR**、**WHILE** 或 **REPEAT-UNTIL** 循环中跳离，或者从 **SWITCH** 区块中跳到最 **SWITCH** 的最末端。	break [<Boolean expression>]

伪指令	说明	语法
CASE	属于 **SWITCH** 区块的一部分，**CASE** 必须在 **SWITCH** 区块中使用。	switch <expression> case <expression 1>[,<expression 2>] <statements>
CONTINUE	跳至内部含有**CONTINUE**伪指令的**FOR**、**WHILE**或**REPEAT-UNTIL**循环的起始位置。 在循环内**CONTINUE**后面的表达式都将会被忽略。	continue [<Boolean expression>]
DEFAULT	**SWITCH**区块的一部分，使用**SWITCH**区块时必须要有**DEFAULT**的条件式。**DEFAULT**为表示**SWITCH**判断式中的默认的编译区块。	default <statements>
ELSE	提供一个相对于**IF**的编译区块。*NYASM* 在**IF**的编译区块与**ELSE**的编译区块二者之间只会选择一个做编译。	else <statements>
ENDFOR	结束一个**FOR**循环。	endfor
ENDIF	结束条件式编译区块。	endif
ENDS	提供一个方便管理的结束指令，可以使用在**ENDFOR, ENDW, ENDSW, ENDIF**。	ends
ENDSW	结束条件式**SWITCH**编译区块。	endsw
ENDW	结束一个**WHILE**循环。	endw
FOR	执行FOR的循环计数。	for <iterator> = <expr1> to <expr2> [step <expr3>]
IF	条件式编译程序区块的起始。	if <expr>
IFDEF	假如符号已经有被定义过就执行。	ifdef <label>
IFNDEF	假如符号尚未被定义就执行。	ifndef <label>
REPEAT	至少会执行一次的循环。	Repeat <statements> until <Boolean expression>
SWITCH	条件式交换编译区块的起始。	switch <expr>
UNTIL	假如条件式成立就结束至少会执行一次的循环。	Repeat <statements> until <Boolean expression>
WHILE	**WHILE**所带的条件若是成立则执行循环。	while <expr>
资料		
CBLOCK	定义一个常数区块。	cblock [<expr>]
DW	宣告一个字符组（word）的数据。	[<label>] dw <expr>[,<expr>,...,<expr>]
ENDC	结束一个常数区段。	endc

伪指令	说明	语法
宏		
ENDM	结束一个宏定义。	endm
EXITM	从一个宏离开。	exitm
EXPAND	宏列表展开。	expand
LOCAL	宣告局部宏变量。	local <label>[,<label>]
MACRO	宣告宏定义。	<label> macro [<arg>,...,<arg>]
MAXMACRODEPTH	设定宏展开的最大深度。	Maxmacrodepth [=] <expr>
NOEXPAND	关闭宏的展开。	noexpand

表格 A-2 编译程序选项

选项	默认状态	说明
c	Off	英文字母大小写区分打开/关闭 c=on 打开 c=off 关闭
p	None	设定处理器的类型： /p=<processor_type> 这里的<processor_type> 是指九齐科技的组件。 例如, NY5A005A。
unlockrsvmem	Locked	/unlockrsvmem 只适用于4-bit MCU。允许编辑保留内存区域。
nocfgblk	Configuration block required	/nocfgblk 只适用于4-bit MCU。在编译阶段忽略已存在的硬件组态设定。

表格 A-3 所支持的数字系统基底型态

型态	语法	举例
十进制	D'<digits>'	D'100'
十六进制	H'<hex_digits>' 0x<hex_digits> <hex_digits>h	H'9f' 0x9f 9fh
八进制	O'<octal_digits>'	O'777'
二进制	B'<binary_digits>'	B'00111001'

表格 A-4 *NYASM* 算术操作数

操作数		举例
$	取得程序计数器	goto $ + 3
(	左括号	1 + (d * 4)
)	右括号	(Length + 1) * 256
!	反逻辑（逻辑补码）	if ! (a == b)
-	负（二的补码）	-1 * Length
~	一的补码	flags = ~flags
high	取得24位值的最高位值	mvma high 0x121314 ;accumulator will contain 0x12
mid	取得24位值的中间位值	mvma mid 0x121314 ;accumulator will contain 0x13
low	取得24位值的最低位值	mvma low 0x121314 ;accumulator will contain 0x14
high0	取得24位值的最高位的低位4位值	mvma high0 0x123456 ;accumulator will contain 0x2
high1	取得24位值的最高位的高位4位值	mvma high1 0x123456 ;accumulator will contain 0x1
mid0	取得24位值的中间位的低位4位值	mvma mid0 0x123456 ;accumulator will contain 0x4
mid1	取得24位值的中间位的高位4位值	mvma mid1 0x123456 ;accumulator will contain 0x3
low0	取得24位值的最低位的低位4位值	mvma low0 0x123456 ;accumulator will contain 0x6
low1	取得24位值的最低位的高位4位值	mvma low1 0x123456 ;accumulator will contain 0x5
*	算术乘	a = b * c
/	算术除	a = b / c
%	算术余	entry_len = tot_len % 16
+	算术加	tot_len = entry_len * 8 + 1
-	算术减	entry_len = (tot - 1) / 8
<<	位左移	flags = flags << 1
>>	位右移	flags = flags >> 1
>=	大或等于	if entry_idx >= num_entries
>	大于	if entry_idx > num_entries
<	小于	if entry_idx < num_entries
<=	小或等于	if entry_idx <= num_entries

操作数		举例
==	等于	if entry_idx == num_entries
!=	不等于	if entry_idx != num_entries
&	位与	flags = flags & ERROR_BIT
^	位互斥或	flags = flags ^ ERROR_BIT
|	位或	flags = flags | ERROR_BIT
&&	逻辑与	if (len == 512) && (b == c)
||	逻辑或	if (len == 512) || (b == c)
=	设值	entry_index = 0
+=	加并设值	entry_index += 1
-=	减并设值	entry_index -= 1
*=	乘并设值	entry_index *= entry_length
/=	除并设值	entry_total /= entry_length
%=	求余并设值	entry_index %= 8
<<=	位左移并设值	flags <<= 3
>>=	位右移并设值	flags >>= 3
&=	位与并设值	flags &= ERROR_FLAG
|=	位或并设值	flags |= ERROR_FLAG
^=	位互斥或并设值	flags ^= ERROR_FLAG
++	递增量为1	i ++
--	递减量为1	i --

A.2 MCU 列表

表格 A-5 MCU 列表

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
1	NY4P018C	16K x 10	48K x 10	0x001F--0x07FF	8 I/O
2	NY4P045C	16K x 10	112K x 10	0x001F--0x07FF	8 I/O
3	NY4P065C	16K x 10	160K x 10	0x001F--0x07FF	8 I/O
4	NY4P085C	16K x 10	208K x 10	0x001F--0x07FF	8 I/O
5	NY4P105C	16K x 10	256K x 10	0x001F--0x07FF	8 I/O
6	NY4A003B	12K x 10	12K x 10	0x001F--0x07FF	4 I/O
7	NY4A005B	16K x 10	16K x 10	0x001F--0x07FF	4 I/O
8	NY4A008B	16K x 10	24K x 10	0x001F--0x07FF	4 I/O
9	NY4A011B	16K x 10	32K x 10	0x001F--0x07FF	4 I/O
10	NY4B003B	12K x 10	12K x 10	0x001F--0x07FF	8 I/O
11	NY4B005B	16K x 10	16K x 10	0x001F--0x07FF	8 I/O
12	NY4B008B	16K x 10	24K x 10	0x001F--0x07FF	8 I/O
13	NY4B011B	16K x 10	32K x 10	0x001F--0x07FF	8 I/O
14	NY4B018C	16K x 10	48K x 10	0x001F--0x07FF	8 I/O
15	NY4B025B	16K x 10	64K x 10	0x001F--0x07FF	8 I/O
16	NY4B038C	16K x 10	96K x 10	0x001F--0x07FF	8 I/O
17	NY4B045B	16K x 10	112K x 10	0x001F--0x07FF	8 I/O
18	NY4B058C	16K x 10	144K x 10	0x001F--0x07FF	8 I/O
19	NY4B065B	16K x 10	160K x 10	0x001F--0x07FF	8 I/O
20	NY4B075B	16K x 10	184K x 10	0x001F--0x07FF	8 I/O
21	NY4B085B	16K x 10	208K x 10	0x001F--0x07FF	8 I/O
22	NY4B095B	16K x 10	232K x 10	0x001F--0x07FF	8 I/O
23	NY4B105B	16K x 10	256K x 10	0x001F--0x07FF	8 I/O
24	NY4B115B	16K x 10	280K x 10	0x001F--0x07FF	8 I/O
25	NY4B125B	16K x 10	304K x 10	0x001F--0x07FF	8 I/O
26	NY4B145B	16K x 10	352K x 10	0x001F--0x07FF	8 I/O
27	NY4B165B	16K x 10	400K x 10	0x001F--0x07FF	8 I/O
28	NY5P025B	16K x 10	64K x 10	0x001F--0x0BFF	16 I/O
29	NY5P055B	16K x 10	136K x 10	0x001F--0x0BFF	16 I/O
30	NY5P085B	16K x 10	208K x 10	0x001F--0x0BFF	16 I/O
31	NY5P185B	16K x 10	448K x 10	0x001F--0x0BFF	16 I/O
32	NY5P025J	16K x 10	64K x 10	0x001F--0x0BFF	16 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
33	NY5P055J	16K x 10	136K x 10	0x001F--0x0BFF	16 I/O
34	NY5P085J	16K x 10	208K x 10	0x001F--0x0BFF	16 I/O
35	NY5P185J	16K x 10	448K x 10	0x001F--0x0BFF	16 I/O
36	NY5P345J	16K x 10	832K x 10	0x001F--0x0BFF	16 I/O
37	NY5A003C	12K x 10	12K x 10	0x001F--0x0BFF	7+1 I/O
38	NY5A005C	16K x 10	16K x 10	0x001F--0x0BFF	7+1 I/O
39	NY5A008C	16K x 10	24K x 10	0x001F--0x0BFF	7+1 I/O
40	NY5A011C	16K x 10	32K x 10	0x001F--0x0BFF	7+1 I/O
41	NY5A018C	16K x 10	48K x 10	0x001F--0x0BFF	7+1 I/O
42	NY5A025C	16K x 10	64K x 10	0x001F--0x0BFF	7+1 I/O
43	NY5A035C	16K x 10	88K x 10	0x001F--0x0BFF	7+1 I/O
44	NY5A045C	16K x 10	112K x 10	0x001F--0x0BFF	7+1 I/O
45	NY5A055C	16K x 10	136K x 10	0x001F--0x0BFF	7+1 I/O
46	NY5A065C	16K x 10	160K x 10	0x001F--0x0BFF	7+1 I/O
47	NY5B005C	16K x 10	16K x 10	0x001F--0x0BFF	14+1 I/O
48	NY5B008C	16K x 10	24K x 10	0x001F--0x0BFF	14+1 I/O
49	NY5B011C	16K x 10	32K x 10	0x001F--0x0BFF	14+1 I/O
50	NY5B018C	16K x 10	48K x 10	0x001F--0x0BFF	14+1 I/O
51	NY5B025C	16K x 10	64K x 10	0x001F--0x0BFF	14+1 I/O
52	NY5B035C	16K x 10	88K x 10	0x001F--0x0BFF	14+1 I/O
53	NY5B046C	16K x 10	112K x 10	0x001F--0x0BFF	14+1 I/O
54	NY5B055C	16K x 10	136K x 10	0x001F--0x0BFF	14+1 I/O
55	NY5B065C	16K x 10	160K x 10	0x001F--0x0BFF	14+1 I/O
56	NY5B075C	16K x 10	184K x 10	0x001F--0x0BFF	14+1 I/O
57	NY5B085C	16K x 10	208K x 10	0x001F--0x0BFF	14+1 I/O
58	NY5B112C	16K x 10	272K x 10	0x001F--0x0BFF	14+1 I/O
59	NY5B132C	16K x 10	320K x 10	0x001F--0x0BFF	14+1 I/O
60	NY5B158C	16K x 10	384K x 10	0x001F--0x0BFF	14+1 I/O
61	NY5B185C	16K x 10	448K x 10	0x001F--0x0BFF	14+1 I/O
62	NY5C112C	16K x 10	272K x 10	0x001F--0x0BFF	19+1 I/O
63	NY5C132C	16K x 10	320K x 10	0x001F--0x0BFF	19+1 I/O
64	NY5C158C	16K x 10	384K x 10	0x001F--0x0BFF	19+1 I/O
65	NY5C185C	16K x 10	448K x 10	0x001F--0x0BFF	19+1 I/O
66	NY5C225C	16K x 10	544K x 10	0x001F--0x0BFF	19+1 I/O
67	NY5C265C	16K x 10	640K x 10	0x001F--0x0BFF	19+1 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
68	NY5C305C	16K x 10	736K x 10	0x001F--0x0BFF	19+1 I/O
69	NY5C345C	16K x 10	832K x 10	0x001F--0x0BFF	19+1 I/O
70	NY5Q019A	48K x 10	48K x 10	0x001F—0x07FF	8 I/O
71	NY5Q020A	48K x 10	48K x 10	0x001F—0x07FF	8 I/O
72	NY5Q026A	64K x 10	64K x 10	0x001F—0x07FF	4 I/O
73	NY5Q039A	64K x 10	96K x 10	0x001F—0x07FF	8 I/O
74	NY5Q040A	64K x 10	96K x 10	0x001F—0x07FF	8 I/O
75	NY5Q046A	64K x 10	112K x 10	0x001F—0x07FF	12 I/O
76	NY5Q060A	64K x 10	144K x 10	0x001F—0x07FF	16 I/O
77	NY5Q079A	64K x 10	192K x 10	0x001F—0x07FF	12 I/O
78	NY5Q080A	64K x 10	192K x 10	0x001F—0x07FF	12 I/O
79	NY5Q092A	64K x 10	224K x 10	0x001F—0x07FF	16 I/O
80	NY5Q159A	64K x 10	384K x 10	0x001F—0x07FF	12 I/O
81	NY5Q160A	64K x 10	384K x 10	0x001F—0x07FF	12 I/O
82	NY5Q172A	64K x 10	416K x 10	0x001F—0x07FF	16 I/O
83	NY5Q342A	64K x 10	832K x 10	0x001F—0x07FF	20 I/O
84	NY6P025A	64K x 10	64K x 10	0x001E—0x03FF	9 I/O
85	NY6P025J	64K x 10	64K x 10	0x001E—0x03FF	16 I/O
86	NY6P055J	136K x 10	136K x 10	0x001E—0x03FF	16 I/O
87	NY6P085J	208K x 10	208K x 10	0x001E—0x03FF	16 I/O
88	NY6P185J	448K x 10	448K x 10	0x001E—0x03FF	24 I/O
89	NY6P345J	832K x 10	832K x 10	0x001E—0x03FF	24 I/O
90	NY6A008A	24K x 10	24K x 10	0x001E—0x03FF	8 I/O
91	NY6A011A	32K x 10	32K x 10	0x001E—0x03FF	8 I/O
92	NY6A018A	48K x 10	48K x 10	0x001E—0x03FF	8 I/O
93	NY6A025A	64K x 10	64K x 10	0x001E—0x03FF	8 I/O
94	NY6A035A	88K x 10	88K x 10	0x001E—0x03FF	8 I/O
95	NY6A045A	112K x 10	112K x 10	0x001E—0x03FF	8 I/O
96	NY6A055A	136K x 10	136K x 10	0x001E—0x03FF	8 I/O
97	NY6A065A	160K x 10	160K x 10	0x001E—0x03FF	8 I/O
98	NY6B005A	16K x 10	16K x 10	0x001E—0x03FF	16 I/O
99	NY6B008A	24K x 10	24K x 10	0x001E—0x03FF	16 I/O
100	NY6B011A	32K x 10	32K x 10	0x001E—0x03FF	16 I/O
101	NY6B018A	48K x 10	48K x 10	0x001E—0x03FF	16 I/O
102	NY6B025A	64K x 10	64K x 10	0x001E—0x03FF	16 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
103	NY6B035A	88K x 10	88K x 10	0x001E—0x03FF	16 I/O
104	NY6B045A	112K x 10	112K x 10	0x001E—0x03FF	16 I/O
105	NY6B055A	136K x 10	136K x 10	0x001E—0x03FF	16 I/O
106	NY6B065A	160K x 10	160K x 10	0x001E—0x03FF	16 I/O
107	NY6B075A	184K x 10	184K x 10	0x001E—0x03FF	16 I/O
108	NY6B085A	208K x 10	208K x 10	0x001E—0x03FF	16 I/O
109	NY6C112A	272K x 10	272K x 10	0x001E—0x03FF	24 I/O
110	NY6C132A	320K x 10	320K x 10	0x001E—0x03FF	24 I/O
111	NY6C158A	384K x 10	384K x 10	0x001E—0x03FF	24 I/O
112	NY6C185A	448K x 10	448K x 10	0x001E—0x03FF	24 I/O
113	NY6C225A	544K x 10	544K x 10	0x001E—0x03FF	24 I/O
114	NY6C265A	640K x 10	640K x 10	0x001E—0x03FF	24 I/O
115	NY6C305A	736K x 10	736K x 10	0x001E—0x03FF	24 I/O
116	NY6C345A	832K x 10	832K x 10	0x001E—0x03FF	24 I/O
117	NY7A004A	16K x 12	16K x 12	0x0010 – 0x03FF	8 I/O
118	NY7A007A	24K x 12	24K x 12	0x0010 – 0x03FF	8 I/O
119	NY7A010A	32K x 12	32K x 12	0x0010 – 0x03FF	8 I/O
120	NY7A016A	48K x 12	48K x 12	0x0010 – 0x03FF	8 I/O
121	NY7A021A	64K x 12	64K x 12	0x0010 – 0x03FF	8 I/O
122	NY7A032A	96K x 12	96K x 12	0x0010 – 0x03FF	8 I/O
123	NY7A043A	128K x 12	128K x 12	0x0010 – 0x03FF	8 I/O
124	NY7A054A	160K x 12	160K x 12	0x0010 – 0x03FF	8 I/O
125	NY7A065A	192K x 12	192K x 12	0x0010 – 0x03FF	8 I/O
126	NY7B007A	24K x 12	24K x 12	0x0010 – 0x03FF	16 I/O
127	NY7B010A	32K x 12	32K x 12	0x0010 – 0x03FF	16 I/O
128	NY7B016A	48K x 12	48K x 12	0x0010 – 0x03FF	16 I/O
129	NY7B021A	64K x 12	64K x 12	0x0010 – 0x03FF	16 I/O
130	NY7B032A	96K x 12	96K x 12	0x0010 – 0x03FF	16 I/O
131	NY7B043A	128K x 12	128K x 12	0x0010 – 0x03FF	16 I/O
132	NY7B054A	160K x 12	160K x 12	0x0010 – 0x03FF	16 I/O
133	NY7B065A	192K x 12	192K x 12	0x0010 – 0x03FF	16 I/O
134	NY7B076A	224K x 12	224K x 12	0x0010 – 0x03FF	16 I/O
135	NY7B087A	256K x 12	256K x 12	0x0010 – 0x03FF	16 I/O
136	NY7C010A	32K x 12	32K x 12	0x0010 – 0x03FF	24 I/O
137	NY7C016A	48K x 12	48K x 12	0x0010 – 0x03FF	24 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
138	NY7C021A	64K x 12	64K x 12	0x0010 – 0x03FF	24 I/O
139	NY7C032A	96K x 12	96K x 12	0x0010 – 0x03FF	24 I/O
140	NY7C043A	128K x 12	128K x 12	0x0010 – 0x03FF	24 I/O
141	NY7C054A	160K x 12	160K x 12	0x0010 – 0x03FF	24 I/O
142	NY7C065A	192K x 12	192K x 12	0x0010 – 0x03FF	24 I/O
143	NY7C076A	224K x 12	224K x 12	0x0010 – 0x03FF	24 I/O
144	NY7C087A	256K x 12	256K x 12	0x0010 – 0x03FF	24 I/O
145	NY7C110A	328K x 12	328K x 12	0x0010 – 0x03FF	24 I/O
146	NY7C130A	384K x 12	384K x 12	0x0010 – 0x03FF	24 I/O
147	NY7C150A	448K x 12	448K x 12	0x0010 – 0x03FF	24 I/O
148	NY7C170A	512K x 12	512K x 12	0x0010 – 0x03FF	24 I/O
149	NY7C220A	656K x 12	656K x 12	0x0010 – 0x03FF	24 I/O
150	NY7C260A	768K x 12	768K x 12	0x0010 – 0x03FF	24 I/O
151	NY7C305A	896K x 12	896K x 12	0x0010 – 0x03FF	24 I/O
152	NY7C345A	1024K x 12	1024K x 12	0x0010 – 0x03FF	24 I/O
153	NY8A050D	512 x 14	512 x 14	-	6 I/O
154	NY8A050E	512 x 14	512 x 14	-	6 I/O
155	NY8A050E1	512 x 14	512 x 14	-	6 I/O
156	NY8A051B	1K x 14	1K x 14	-	6 I/O
157	NY8A051D	1K x 14	1K x 14	-	6 I/O
158	NY8A051F	1K x 14	1K x 14	-	6 I/O
159	NY8A051G	1K x 14	1K x 14	-	6 I/O
160	NY8A051H	1K x 14	1K x 14	-	6 I/O
161	NY8A051H1	1K x 14	1K x 14	-	6 I/O
162	NY8A051J	1K x 14	1K x 14	-	6 I/O
163	NY8A051K	1K x 14	1K x 14	-	6 I/O
164	NY8A051L	1K x 14	1K x 14	-	6 I/O
165	NY8A052E	1.5K x 14	1.5K x 14	-	14 I/O
166	NY8A053B	1K x 14	1K x 14	-	12 I/O
167	NY8A053D	1K x 14	1K x 14	-	12 I/O
168	NY8A053E	1K x 14	1K x 14	-	12 I/O
169	NY8A054A	2K x 14	2K x 14	-	14 I/O
170	NY8A054D	2K x 14	2K x 14	-	14 I/O
171	NY8A054E	2K x 14	2K x 14	-	14 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
172	NY8A054E1	2K x 14	2K x 14	-	14 I/O
173	NY8A056A	1K x 14	1K x 14	-	16 I/O
174	NY8AE51F	1K x 14	1K x 14	-	6 I/O
175	NY8B060D	1K x 14	1K x 14	-	6 I/O
176	NY8B061E	1.25K x 14	1.25K x 14	-	14 I/O
177	NY8B062A	2K x 14	2K x 14	-	14 I/O
178	NY8B062B	2K x 14	2K x 14	-	14 I/O
179	NY8B062D	2K x 14	2K x 14	-	14 I/O
180	NY8B062E	2K x 14	2K x 14	-	14 I/O
181	NY8B062F	2K x 14	2K x 14	-	14 I/O
182	NY8B062F1	2K x 14	2K x 14	-	14 I/O
183	NY8B072A	2K x 14	2K x 14	-	18 I/O
184	NY8BE62D	2K x 14	2K x 14	-	14 I/O
185	NY8BM61D	2K x 14	2K x 14	-	14 I/O
186	NY8BM62D	2K x 14	2K x 14	-	14 I/O
187	NY8BM72A	2K x 14	2K x 14	-	18 I/O
188	NY8BM84A	4K x 16	4K x 16	-	22 I/O
189	NY8F1141	2K x 16	2K x 16		14 I/O
190	NY8F1241	2K x 16	2K x 16		14 I/O
191	NY8F2481	4K x 16	4K x 16	-	22 I/O
192	NY8TE64A	4K x 14	4K x 14	-	18 I/O
193	NY8TM52D	2K x 14	2K x 14	-	6 I/O
194	NY8LP05A	5K x 8	5K x 8	0x0000~0x07FF	16 I/O
195	NY8LP08A	8K x 8	8K x 8	0x0000~0x07FF	24 I/O
196	NY8LP10A	17K x 8	17K x 8	0x0000~0x07FF	16 I/O
197	NY8LP11A	17K x 8	17K x 8	0x0000~0x07FF	16 I/O
198	NY9T001A	4K x 10	4K x 10	0x001F – 0x01FF	4 I/O
199	NY9T004A	8K x 10	8K x 10	0x001F – 0x01FF	8 I/O
200	NY9T008A	12K x 10	12K x 10	0x001F – 0x01FF	16 I/O
201	NY9T016A	16K x 10	16K x 10	0x001F – 0x01FF	24 I/O
202	NY9UP01A	768 x 10	768 x 10	0x0040 – 0x004F	13 I/O
203	NY9UP02A	1280 x 10	1280 x 10	0x0020 – 0x004F	13 I/O
204	NY9UP08A	8K x 10	8K x 10	0x0020 – 0x004F	13 I/O
205	NY9U032B	32K x 10	32K x 10	0x001F – 0x03FF	16 I/O

No.	IC 母体	PROG ROM 空间	DATA ROM 空间	ROM 保留地址	I/O 脚个数
206	NY9U064B	64K x 10	64K x 10	0x001F – 0x03FF	16 I/O

附录B - 词汇表

这个词汇表定义了关于本文所用的相关术语，可作为一个共同的对照表。这个词汇表包含了用于 *NYASM* 中的术语定义。

B.1 专门术语

Nyquest MCU (九齐科技微控制器)

九齐科技MCU参照NY4/NY5/NY7微控制器系列。

Application (应用)

一个由用户所开发的软件和硬件套件，通常被设计成九齐科技微控制器的相关应用产品。.

Assemble (编译)

NYASM 宏编译程序执行将来源码翻成机械码的动作。

Binary File

由*NYASM* 所输出且可以个别被执行。

Build

根据需求重新编译所有源码文件的功能。

Control Directives (控制型伪指令)

控制型伪指令允许有条件式的编译部分代码。

Data Directives (数据型伪指令)

数据型伪指令是用来控制内存的配置和将所要参照的数据做有意义的命名。

Data RAM (数据存储器)

MCU上的内存通用寄存器，它是属于RAM的一部分。

Directives (伪指令)

伪指令是被用来控制编译程序的操作，它告诉*NYASM* 如何处理助忆符、定义数据和列表文件的格式。伪指令可以使程序的编码变的更容易且依照实际的需求提供订制化的输出。

Expressions (表达式)

表达式是用于来源列的算术操作，运算栏可能包含了常数、符号或是任何个别常数与符号的组合。

Forward reference (向前参考)

在定义之前使用变量或函数。在*NYASM*里Forward reference是不允许的。例如：尚未定义Macro前不可使用Macro、尚未定义常数之前不可使用常数。例如格式，主要是当编译程序在估算表达式的时候使用。默认的格式是十六

Identifier (识别符)

一个功能或变量的名称。

Initialized Data (资料的初始化)

在定义数据的同时亦定义一个初始值。像C语言，int myVar = 5;定义一个变量并同时定义初始值。

Listing Directives (列表伪指令)

列表伪指令主要是用来控制*NYASM*列表文件的格式。他们所允许的规格有数字系统的基底，保留区内存的存取和其他列表的控制。

Listing File (列表文件)

列表文件是一个ASCII的文本文件，主要用来显示在源码文件中的每一个汇编语言所产生的机械码，*NYASM*的伪指令，或是所遇到的宏。

Local Label (区域标记)

使用LOCAL伪指令将一个标记定义成区域性标记。这些标记是在每一个宏实例中都具有个别性。换句话说，在宏当中将符号或标记被宣成local时，则在遇到ENDM时，会将这些符号标记从符号表中清除。

Macro (宏)

宏是一串汇编语言指令的汇集。当在来源当中下达一个宏的名称时，则宏将会被含括到汇编程序码当中。宏必须在使用前先被定义；并不允许被向前引用。在MACRO伪指令后的所有叙述皆属于宏定义的一部分。在宏内部使用标记必须将其宣告成local，如此这个宏将可以不断的重复被调用。

Macro Directives (宏伪指令)

这些伪指令是在控制在宏定义内部的执行和数据配置。

Mnemonics (助忆符号)

助忆符号将会直接被翻译成机械码。这些是被用来将微控制器的程序或数据存储器内的数据做算术和逻辑运算。它不但可以将寄存器和内存的数据的做移进移出且可以改变程序执行的流程。也被称为运算码。

***NYASM* (九齐科技编译程序)**

九齐科技股份有限公司的MCU编译程序。

Nesting Depth (巢状深度)

宏可以被相互套迭到16层的深度（默认值）。最大的深度是256。

Operators (操作数)

操作数是算术上的符号，像加就写'+'而减就写成'-'，主要被使用构成一个明确的表达式子。每一个操作数都有一个被分配的优先次序。

PC (个人计算机)

任何IBM PC或兼容的计算机。

PC Host (个人计算机主机)

正在执行Windows XP/7/8的计算机。

Precedence (优先次序)

优先次序是一个观念。在表达式上的一些元素可以比其他的能更先获得运算。优先次序相同的操作数在一起时会由左到右做运算。

Program Memory (程序内存)

仿真器或仿真器的内存也包含被下载到应用端的韧体。

Project (专案)

以一组源码文件及指令建立的应用代码。

Radix (数值格式)

Radix是设定基本的数值格式，主要是当编译程序在估算表达式的时候使用。默认的格式是十进制。你可以来改变默认的格式及取消原本设定的格式。

RAM (随机存取内存)

随机存取内存。

Raw Data (原始资料)

二进制格式的代码或数据的。

Recursion (递归)

这是一个宏的概念，已经被定义过，可以自己调用自己。当写递归的宏必须特别注意，若没有从递回离开的话，它很容易就死锁在无限循环内。

ROM (只读存储器)

只读存储器。

Source Code (源码)

源码是由九齐科技MCU的指令和*NYASM*的伪指令及宏所构成且将会被翻译成机械代码。这个代码将 适用于合九齐科技的开发系统，如*NYIDE*使用。

Source File (源码文件)

一个ASCII文本文件，内容可以是九齐科技MCU的指令和*NYASM*的伪指令及宏（原始代码），且最后将会被翻译成机械码。ASCII文本文件可以由任何的ASCII文本编辑器所产生和编辑。

Stack (堆栈)

在数据存储器中的一块区域被用来储存函数的参数、回传值、局部变量和返回地址。

Symbol (符号)

符号是一个通用的机制，用来描述包括一个程序里各式各样的片段。这些片段包含有功能名称、变量名称、文件名称、宏名称...等等。

Un-initialized Data (未初始化的资料)

一个被定义的数据，但没有被设定初始值。像是C语言中的Int myVar。