# 源码公开的 MCS－51 单片机的宏汇编器

Macro Assembler of MCS-51 micro controller

**Version：0.12**

广州周立功单片机发展有限公司

# 目 录

一、 汇编器概述 . . . . . . . . . . 5

二、 KEIL 公司的宏汇编器 A51 简介 . . . . . . . . . . 5

A51 宏汇编器的保留字 .......... 6
A51 宏汇编器的运算符 .......... 6
A51 宏汇编器中运算符的优先级 .......... 7
A51 的命令助记符 .......... 7
A51 的汇编伪指令 .......... 7
A51 的宏处理指令 .......... 7
A51 的汇编控制指令 .......... 7
A51 的条件汇编指令 .......... 7

三、 项目说明 . . . . . . . . . . 7

功能要求 .......... 8
设计方法 .......... 8
开发工具 .......... 8
开发环境 .......... 8

四、 项目分解 . . . . . . . . . . 8

词法分析 .......... 9
语法分析 .......... 9
语义分析 .......... 9
目标代码生成 .......... 9

五、 设计思路 . . . . . . . . . . 10

总体模块 .......... 10
指令系统模块 .......... 10
TOKEN 流模块 .......... 11
词法和语法分析模块 .......... 11
语义分析模块 .......... 11
代码生成模块 .......... 11
产生目标文件模块 .......... 12
产生列表文件模块 .......... 12
标号管理模块 .......... 12
出错处理模块 .......... 12
汇编段模块 .......... 12
OBJ 目标模块 .......... 12
目标记录模块 .......... 13
OBJ 文件的操作模块 .......... 13
列表文件的操作模块 .......... 13
主程序模块 .......... 13
其他模块 .......... 13

六、 实现方法 .......... 14
所有的类定义列表 .......... 14
主程序实现 .......... 15
宏汇编器对象 MASM 的创建 .......... 15
指令系统管理器 INSTM 的创建 .......... 16
标号管理器 LABMGER 的创建 .......... 17
宏汇编器的服务 COMPLYFILE() .......... 18
宏汇编器的词法分析服务 FILETOTOKEN() .......... 18
关于 TOKENFIELD 类 .......... 19
宏汇编器的服务 STRTOTOKEN() .......... 20
关于语法规则 .......... 21
宏汇编器的语法分析服务 ASMFILEPARSE() .......... 24
汇编指令语法分析服务 RESVINSTPARSE() .......... 25
汇编伪代码语法分析服务 PSDOWORDPARSE() .......... 25
TOKENOPER 类的服务 EXPTOPACK() .......... 25
TOKENOPER 类的服务 OPADTOPACK() .......... 27
宏汇编器语义分析服务 CREATEVARTABLE() .......... 28
宏汇编器的 FINDRESVINSTDO()内部服务 .......... 31
标号管理器的标号登记 ADDLABEL()服务 .......... 32
标号管理器的表达式求值 CALEXPRESSION()服务 .......... 32
算符优先关系表 .......... 33
宏汇编器代码生成服务 FILETOOBJCODE() .......... 34
关于 DAT4ARY 类（目标代码组类） .......... 35
关于 OBJ 文件的格式 .......... 36
关于宏处理 .......... 37
关于宏管理器 MACROMGER .......... 42
关于 MACRODEFBODY 宏定义体类 .......... 42
ZLG51 对宏定义指令的识别和处理 .......... 43
ZLG51 对宏调用的识别和处理 .......... 44
关于 REPT 宏指令的识别和处理 .......... 45
关于‘$’行汇编控制指令的识别 .......... 46
关于宏变量表 MVLIST .......... 47
关于条件汇编 .......... 47
七、 使用说明 .......... 47
使用方法 .......... 47
错误信息 .......... 48
警告信息 .......... 50
八、 程序说明 .......... 50
九、 运行测试 .......... 50
测试程序 .......... 50
输出信息 .......... 51
输出列表文件 .......... 51
输出目标文件 .......... 51
OBJ 查看器查看结果 .......... 52

十、 本宏汇编器的新特点 ........ 54

十一、 设计小结 ........ 54

十二、 心得体会 ........ 55

十三、 致谢 ........ 55

十四、 参考文献 ........ 55

这是一个应届生写的习作，主要的出发点是想通过这种方式培养员工的系统分析和系统设计的能力。尽管这个软件（A51）还未完全达到满意的效果，不过与 Keil 公司的 A51 配合起来使用还是不错的。公开源码和文档的目的是希望能够给一些爱好者作为学习编译原理和 C 程序设计的“靶子”，同时也希望有更多的热心人参与修改源码直至更加完善，因为我们的初衷并没有想到要做一个自己的 A51/L51 编译器。

周立功

2003 年 4 月 30 日

## 一、汇编器概述

计算机机器语言程序由一条条的机器指令组成，程序的执行也就是这一条条指令的执行。出现在指令中，或者说，出现在程序中只有两个符号：0 和 1。程序不论多么的简单，或是多么的复杂，也就是由 0 和 1 组成的符号串。只有它们才是计算机所能理解并直接执行的。

机器语言程序虽然只有两个符号，但有不少的缺点。首先是难记，很难记住指令系统中每条指令的操作码，造成了使用上的困难；其次是容易出错，并且一旦出错，不易查正；再次，因为各个机器指令系统不同，为某个机器写的解决某一特定问题的程序不能拿到另一个机器上正确运行，不得不重写一个程序来解决“已经解决” 了的问题。

为了使计算机的使用变的简单易用，五十年代初，人们开始使用记忆符号语言。用英语缩写来代替操作码，用字母、字符串（标识符）表示操作数地址，而在机器指令中，操作码和操作数地址都是用二进制数或十六进制数码表示的。这样汇编语言出现了，汇编语言易于记忆，易于使用和调试。 汇编语言使用的是指令的助忆符，符号地址和标号等。用汇编语言编写的源程序翻译成机器语言程序（目标程序）的过程叫做汇编。完成该功能的程序称为汇编器。汇编器自动完成一些功能：如按用户的要求分配存储区；把各种进制数转换成二进制数；计算表达式的值，对源程序进行语法检查，并指出错误信息（如非法格式，未定义符号）等。

后来，当程序的规模发展到一定程度上时，人们开始使用模块化的程序设计思想来组织编码，并且减少重复代码或相似代码的输入，这样就出现了宏汇编。宏汇编器允许用户以模块方式编程，以适应日益复杂的程序设计。模块是具有相对独立功能的程序，它能独立进行汇编或编译。模块化程序设计是将一个大而复杂的程序分成小的功能模块，每个模块程序单独编写、汇编和调试，最后再将这些模块用连接器连接起来，形成一个完整的用户程序。这样做比单块程序更易编写、调试和修改。

模块化的程序的开发只须根据模块的输入及输出定义，按其所需的输入并检查其输出以校核模块的正确性。由于程序具有良好的模块接口，可以把问题限定在模块内，一旦识别出有毛病的模块，解决该问题就变得简单了。当每个模块都测试完毕即可将各模块连接起来，最后再测试全模块。模块化程序的另一个好处是程序共享，即一个模块中的程序可以被其它模块引用。由于模块化程序是可重新定位的，因而也就允许在满足其输入及输出要求时被调用。

## 二、Keil 公司的宏汇编器 A51 简介

Keil 公司是德国一家著名的为 51 系列单片机编写开发软件的公司，著名的产品有 A51、L51 和 BL51、C51、uVision2 等等。其中 A51 是一个具有通用特性和用法的重定位宏汇编器，它与 Intel 公司的 MASM51 宏汇编器具有很好的兼容性，支持模块化编程，可以方便地与高级语言接口。A51 宏汇编器通常在 DOS 命令行上调用，调用格式如下：

A51 文件名 ［汇编控制指令］

“文件名”是以“.A51”或“.ASM”为扩展名的汇编语言源程序。“汇编控制指令”是可选项，用于控制 A51 宏汇编器产生列表文件、目标文件以及其他一些功能。如果省略该选项，A51 则按默认的汇编控制进行汇编，在源文件所在的路径上产生列表文件和目标文件。

8051 单片机汇编语言程序有若干条 8051 指令行组成，8051 指令行的一般形式为：

［标号:］8051 命令助记符［操作符 1］[,操作符 2］[,操作符 3］[;注解］

其中，“标号”是可选项，它可用来表示程序的转移地址，同时可方便程序的调试。

操作符 1～3”是可选项，它的作用依赖于不同的 8051 指令助记符。操作符可以是数字、符号或地址。数字可以使用 10 进制、16 进制、8 进制或 2 进制数。10 进制数以字符“D”为后缀，16 进制数以字符“H”为后缀，8 进制数以字符“O”或“Q”为后缀，2 进制数以字符“B”为后缀。省略后缀时默认为 10 进制

数。立即数的前面需冠以符号“＃”。

A51 宏汇编器允许使用符号来表示数值、地址和寄存器名等，以增加程序的可读性。符号名最长为 31 个字符，第一个字符为英文字母“A”～“Z”或“a”～“z”、下划线“_”或“?”，后续字符可为上述字符或数字“0”～“9”。标号也是一种符号。一些符号已经预定义为 A51 的保留字，用户不能对它们重新定义。

### A51 宏汇编器的保留字

| A51 的保留字 | 意　　义 |
|---|---|
| A | 累加器。 |
| R0 ～ R7 | 当前工作寄存器（共有 4 个寄存器组）。 |
| DPTR | 16 位数据指针，用于访问内部或外部的地址空间的数据。 |
| PC | 16 位程序计数器，其值为下一条将被执行的指令的地址。 |
| C | 进位标志。 |
| AB | 用于乘除操作的寄存器对。 |
| AR0～AR7 | 表示当前工作寄存器的绝对地址，其值取决于指令所选择的工作寄存器组。 |

符号“＄”是一个特殊的汇编符号，它表示当前段的当前地址计数器。CODE、DATA、IDATA、BIT 和 XDATA 这 5 个段都有不同的地址计数器。每执行一条指令，地址计数器的值也随之增加。如果当前段发生变化，地址计数器也将自动变到新段。

A51 中有三类运算符：算术运算符、逻辑运算符和关系运算符。

### A51 宏汇编器的运算符

| | 运算符 | 例　子 | 意　　义 |
|---|---|---|---|
| 算术运算符 | ＋、－ | ＋5、－4 | 数或表达式的正负号。 |
| | ＋、－ | 2＋10－5 | 加减运算。 |
| | ＊ | 1000H ＊ 2 | 乘法运算。 |
| | / | 17 / 4 | 除法运算。 |
| | MOD | 18 MOD 4 | 取模运算。 |
| | ( ) | 8＋（12－5） | 改变运算顺序。 |
| 逻辑运算符 | NOT | NOT 5 | 取反。 |
| | HIGH | HIGH 1234 | 选择操作符的高位字节。 |
| | LOW | LOW 1234H | 选择操作符的低位字节。 |
| | SHL、SHR | 2 SHL 3、8 SHR 4 | 左、右移位。 |
| | AND | 12H AND 0F0H | 逻辑与运算。 |
| | OR | 12H OR 177 | 逻辑或运算。 |
| | XOR | 12H XOR 14 | 逻辑异或运算。 |
| 关系运算符 | >= | 55 >= 17 | 大于等于。 |
| | <= | 23 <= 44 | 小于等于。 |
| | <> | 32 <> 54 | 不等于。 |
| | = | 12H = 18 | 等于。 |
| | < | 21H < 32H | 小于。 |
| | > | 55 > 17 | 大于。 |

A51 宏汇编器的运算符具有不同的优先级，一个表达式中存在多个不同优先级的运算时将按它们的优先级顺序进行运算。如果一个表达式中各个运算符都具有相同的优先级，则按从左到右的顺序进行运算。

### A51 宏汇编器中运算符的优先级

| 优 先 级 | 运 算 符 | 意 义 |
|---|---|---|
| 1 | （ ） | 括号。 |
| 2 | NOT、HIGH、LOW | 取反，取高、低地址。 |
| 3 | ＋、－ | 正、负号。 |
| 4 | *、/、MOD | 乘、除、取模运算。 |
| 5 | ＋、－ | 加、减运算。 |
| 6 | SHL、SHR | 左、右移位。 |
| 7 | AND、OR、XOR | 逻辑与、逻辑或、逻辑异或。 |
| 8 | ＞＝、＜＝、＝、＜、＞、＜＞ | 大于（等于）、小于（等于）、（不）等于。 |

### A51 的命令助记符

NOP、MOV、MOVC、MOVX、PUSH、POP、XCH、XCHD、SWAP、ADD、DJNZ、ADDC、SUBB、INC、DEC、MUL、DIV、DA、ANL、ORL、XRL、CLR、CPL、RL、RLC、RR、RRC、SETB、ACALL、LCALL、RET、RETI、AJMP、LJMP、SJMP、JZ、JNZ、JC、JNC、JB、JNB、JBC、CJNE、JMP。

### A51 的汇编伪指令

SEGMENT、CODE、XDATA、DATA、IDATA、BIT、PAGE、INPAGE、INBLOCK、BITADDRESSABLE、UNIT、OVERLAYABLE、EQU、SET、DS、DBIT、DB、DW、PUBLIC、EXTRN、NAME、END、ORG、USING、RSEG、CSEG、DSEG、XSEG、ISEG、BSEG。

### A51 的宏处理指令

MACRO、ENDM、LOCAL、REPT、IRP、IRPC、EXITM。

特殊的宏运算符有：＆、＜、＞、％、 ；；、 ！、 NUL。

### A51 的汇编控制指令

DATE(DA)、DEBUG(DB)、NODEBUG(NODB)、ERRORPRINT(EP)、NOERRORPRINT(NOEP)、OBJECT(OJ)、NOOBJECT(NOOJ)、PAGELENTH(PL)、PAGEWIDTH(PW)、PRINT(PR)、NOPRINT(NOPR)、SYMBOLS(SB)、NOSYMBOLS(NOSB)、XREF(XR)、NOXREF(NOXR)、TITLE(TT)、MOD51(MO)、NOMOD51(NOMO)、COND、NOCOND、MACRO、NOMACRO、REGISTERBANK(RB)、NOREGISTERBANK(NORB)、EJECT(EJ)、INCLUDE(IC)、LIST(LI)、NOLIST(NOLI)、GEN、NOGEN、SAVE(SA)、RESTORE(RS)。

### A51 的条件汇编指令

IF、ENDIF、ELSEIF、ELSE、SET、RESET。

## 三、项目说明

由于本公司还没有自主版权的 51 系列的汇编器，为填补该领域的空白，特编写具有自主版权的 51 系列单片机的宏汇编器 ZLG51。

### 功能要求

要求宏汇编器采用DOS命令行方式执行，带有各种编译参数开关，能控制生成的obj文件和lst文件的信息。

功能上要求与Keil公司的A51宏汇编器兼容，并且有所增强。

### 设计方法

面向对象的程序设计方法。

### 开发工具

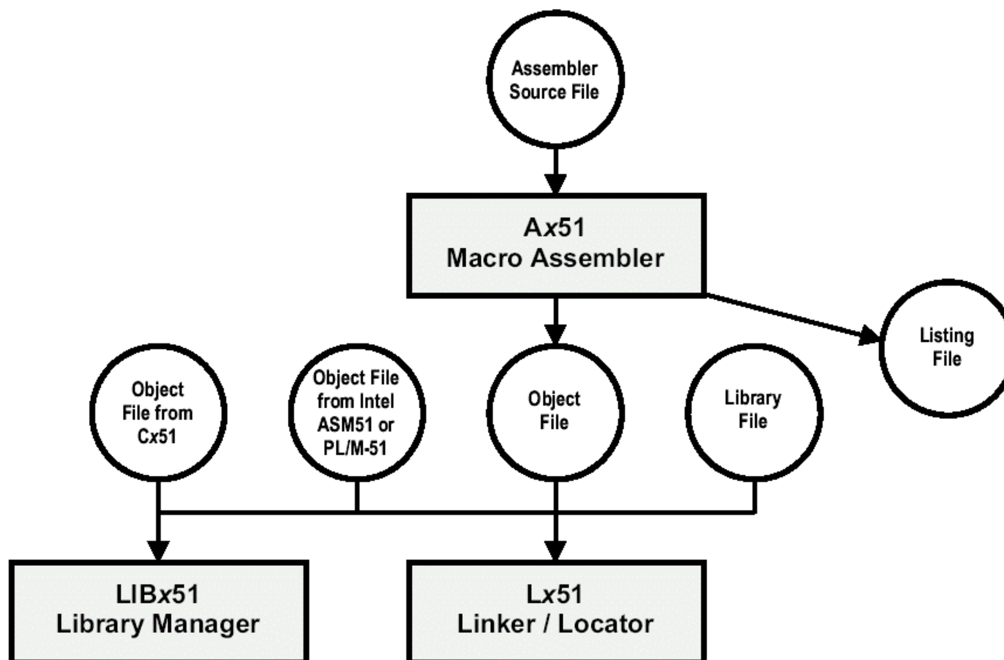
采用Borland C++语言。

### 开发环境

采用Borland C++ Builder 5.0集成环境来编码、调试和编译运行。

## 四、项目分解

宏汇编器的任务是：输入 *.asm源程序文件，输出 *.obj目标代码文件和 *.lst 列表文件——对输入的汇编源程序进行扫描，找出所有词法和语法的错误，然后生成有待重定位的目标代码（中间文件）和提供查看信息的列表文件。中间文件后缀名为obj，是连接器的输入文件。如下图所示：




一个宏汇编器的整个工作流程是划分成一个一个阶段进行的，每个阶段都将源程序的一种表示形式转换成另一种表示形式，各个阶段进行的操作在逻辑上是紧密联结在一起的。这几个阶段是：词法分析、语法分析、语义分析、中间代码生成。另外，两个重要的工作：符号表格的管理和出错处理贯穿以上的所有阶段。汇编过程中源程序的各种信息被保留在种种不同的表格里。汇编时自始至终涉及到表格的构造、查找和更新。如果汇编过程中发现源程序有错误，汇编程序应报告错误的性质和错误发生的地点，并且将错误所造成的影响限制在尽可能小的范围内，使得源程序的其余部分能继续被编译下去，有些地方汇编器还

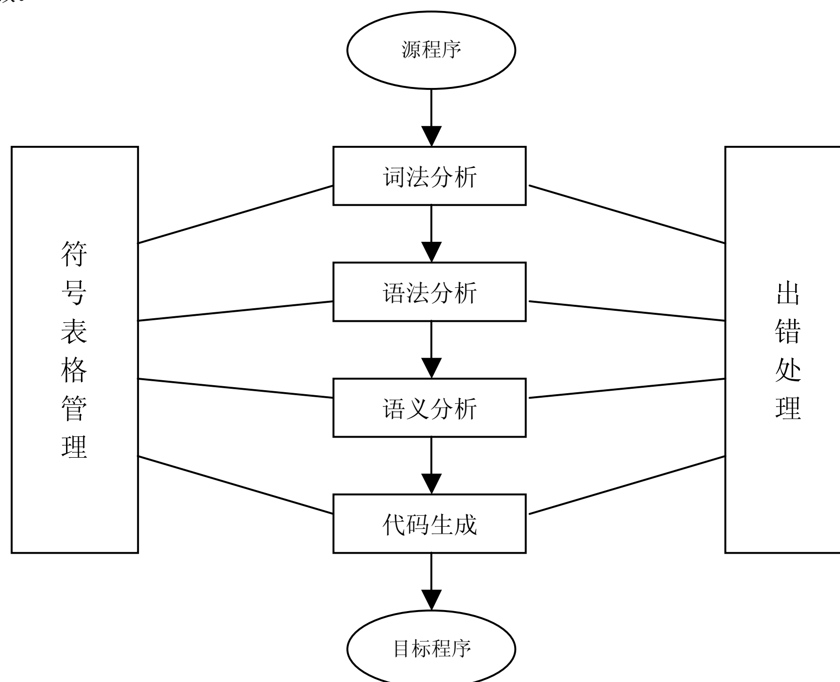
能自动校正错误，这些工作称之为出错处理。

### 词法分析

自左至右扫描源程序的字符串，并识别出一个个的单词（也称单词符号）。这里所谓的单词是指逻辑上紧密相连的按词的组成规则结合起来的一组字符，它们具有一定的意义。把每个单词的 ASCII 码序列替换成所谓的机内表示——Token 形式。这时还需要检查词法错误。词法分析阶段不依靠语法关系。

### 语法分析

扫描对象可能是源程序的 ASCII 码序列，也可能是词法分析后的 Token 序列。前者情形，词法分析程序作为语法分析程序的子程序。语法分析的主要任务是检查源程序的形式语法错误。每当发现错误时将输出有关信息。很多编译程序在进行语法分析时同时完成其他工作，但要注意，如果语法有错误，那么其他工作也就白做。




### 语义分析

扫描的对象通常是语法分析后的结果。这时候，源程序的 Token 序列已经变换成没有错误的符合语法的 Token 生成树。对于汇编器来说，它的任务是收集符号信息，登记在符号表格里，对汇编伪指令的语义进行解释，完成相应的动作。汇编伪指令的功能是改变汇编器的状态并将一些必要的信息（如段定义，变量声明）加入到目标文件中去。另外，表达式的求值，段的选择，地址计数器的计数，指令长度的计算等都在这个阶段中完成。语义分析阶段同样进行着对错误的处理。

### 目标代码生成

这时候的 Token 流在形式上已经比较一致，符号信息都已经登记在各种表格里。这个阶段的任务是根据汇编助记符的各种寻址方式决定它的目标代码。这一部分的工作与目标机的指令系统紧密相关。目标代码生成之后，还要根据 obj 文件的格式把目标代码写入文件。最后还要产生列表文件，为用户提供源码与目标码的对照。

## 五、设计思路

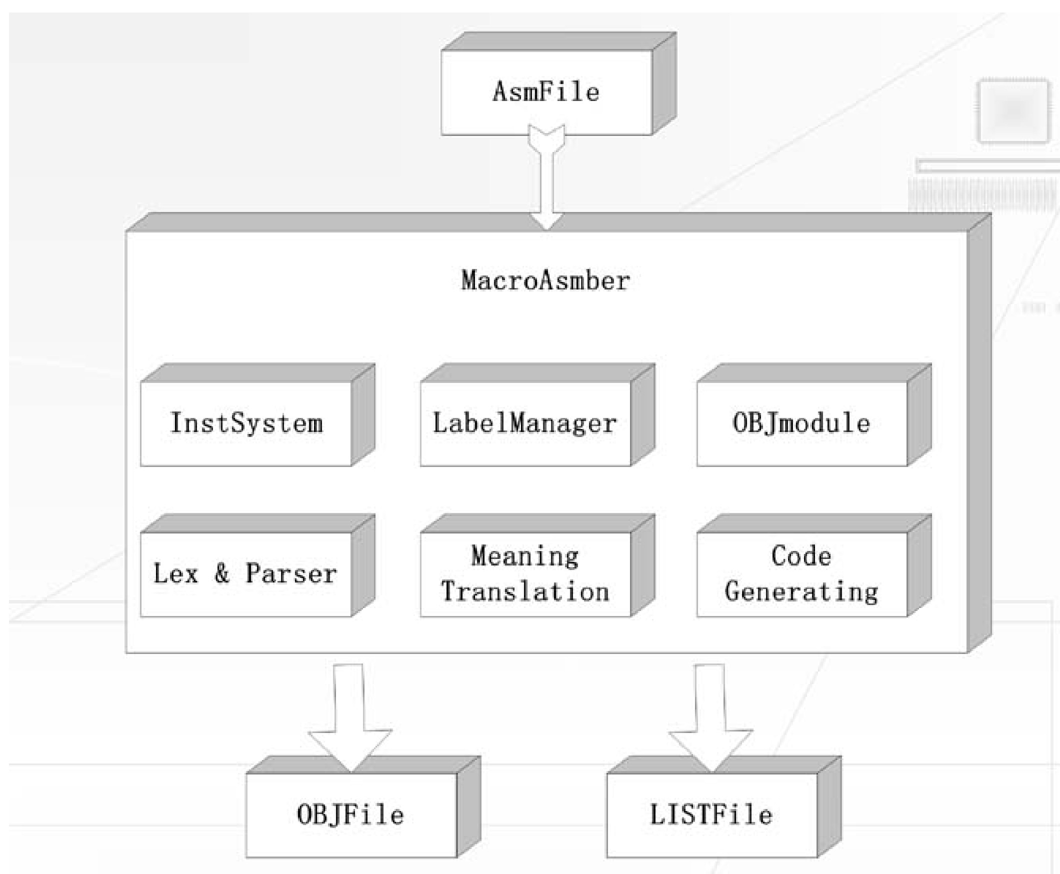
程序设计采用了面向对象的程序设计方法。面向对象的程序设计的好处在于使得计算机解决问题的形式能否更符合人的思维活动和概念，便于程序的模块化，数据抽象，信息隐蔽，增加软件的可扩充性和可重用性，并能控制维护软件的复杂性。

### 总体模块

作为一个宏汇编器程序，我把它看作是一个类：MacroAsmber 类。这个类包含了汇编流程中的各个模块，包括指令系统模块，源文件的 Token 流模块，词法和语法分析模块，语义分析模块，代码生成模块，产生目标文件模块，产生列表文件模块，标号管理模块，出错处理模块。在主程序中创建这个类的一个对象——名为 masm，然后调用它提供的一个服务 masm.ComplyFile()，就可以完成汇编的所有工作。

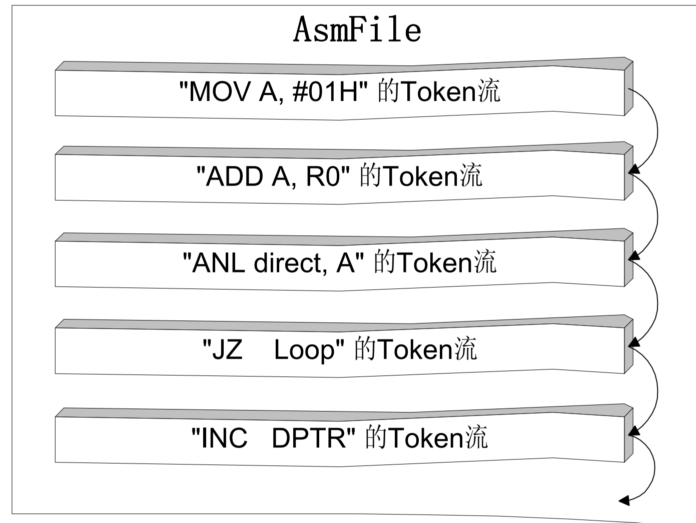
### 指令系统模块

对应指令系统模块，定义了 InstSystem 类。这个类的对象存储了用户使用的计算机对应的指令系统的信息，如指令的名称，Token 码，对应的寻址方式，所需操作数的个数，占用的长度，以及对生成的目标代码的组装方式。它还存储了宏汇编的伪指令信息，如伪指令的名称，Token 码，所属的类型等。为了提高程序运行的速度，提高查找保留字的效率，该类的对象把这些信息都以哈希（hash）表的形式存储。该哈希表在对象的创建时构造并初始化。基于好的哈希公式的哈希表的查找效率比一般的线性表的查找效率高许多倍，这由线性表的大小和哈希公式而定。作为我现今的程序实现来说，查找效率比不使用哈希表的存储方式提高将近 100 倍。

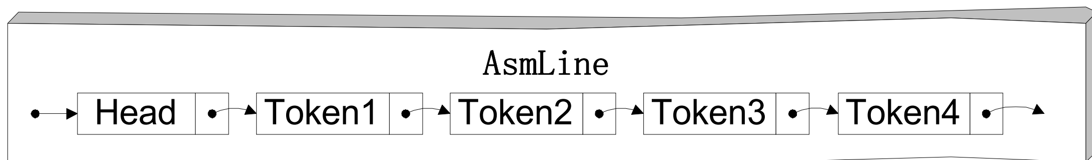


### Token 流模块

对应源文件的 Token 流模块，定义了 AsmFile 类。整个源文件（srcfile.asm），我把它看作是 AsmFile 类的一个对象。对象中包含了文件名，缓冲区指针，磁盘文件指针，和一张由 AsmLine 类的对象组成的线性链表。一个 AsmLine 类的对象表示文件中的一行 ASCII 码所对应的 Token 流。整个 AsmFile 类所提供的服务实质上是的对这张表进行的操作，其中最重要的是：AddAsmLine（AsmLine* line）服务。它的功能就是往 AsmFile 对象中添加一条 Token 流。




一个 AsmLine 类的对象是一组 Tokenfield 类的对象链接而成的单链表，对应着源文件中的一行。它还包含着该汇编行的行号,层次号(在宏嵌套中),偏移地址,长度,是否被启用等等的信息。一个 Tokenfield 类的对象就是一个“Token”，它可能来源于一个单词，或者来源于一个运算符。




### 词法和语法分析模块

对应词法和语法分析模块（Lex & Parser），MacroAsmber 类提供两个内部服务：FileToToken()和 AsmFileParse()来分别进行词法分析和语法分析。这两个服务只供 MacroAsmber 对象内部调用，如 ComplyFile()。

### 语义分析模块

对应语义分析模块（Meaning Translation），MacroAsmber 类提供一个内部服务 CreateVarTable()，该服务通过调用两次 FileFindVar()，把符号表建立起来，并完成一些汇编伪指令的操作，如建立段定义，进行段选择，寻找恰当的寻址方式，计算指令长度，决定汇编行的偏移地址和长度等等。

### 代码生成模块

对应代码生成模块（Code Generating），MacroAsmber 类提供一个内部服务 FileToObjCode()，该服务内部调用 LineToObjCode()，对每一能够产生目标代码的汇编行进行代码生成，生成的代码仍然存放在汇编行中。

### 产生目标文件模块

对应产生目标文件模块,MacroAsmber 类提供一个内部服务 GenOBJFile()。它实质上调用了 OBJModule 类的服务 WriteModuleToOBJFile()。这个服务的功能是把整个目标代码模块以 OBJ 文件的格式写到磁盘上。

### 产生列表文件模块

对应产生列表文件模块，MacroAsmber 类提供一个内部服务 GenListFile()。该服务则扫描整个 AsmFile 里的 Token 流，按照列表文件的格式往磁盘上输出 lst 文件。

### 标号管理模块

对应标号管理模块，定义了 LabelManiger 类。这是一个专门操控标号、变量、常量的管理器和计算器。它管理的对象是 JlabelNode 组成的表格。JLabelNode 类的对象是一个标号（常量或变量），存储有标号的信息：如名字，属性（绝对值、相对值或外部变量），类别（CODE、XDATA、DATA、IDATA、BIT、NUM、SEG、REG），数值，参照段，是否变量标志，是否设置为 PUBLIC 属性标志，数值有效标志，间接访问标志等。它提供的服务查找标号、添加标号、标号正确性检查、表达式求值等。

### 出错处理模块

对应出错处理模块，MacroAsmber 类提供两个内部服务：一个是输出错误信息 OutputErr()，一个是输出警告信息 OutWarning()。产生的错误必须由用户对源程序进行改正后再重新编译。警告则是可以忽略或汇编器已经帮助用户改正了其错误。错误信息和警告信息包含了出错的汇编源文件的文件名，出错行的行号，具体的出错信息等。

### 汇编段模块

段是一块程序代码，是一个模块。段是可以重新定位（地址浮动）的或绝对定位的。一个可重定位的段有一个名字，类型和属性。具有相同段名，但在不同模块的各个段，将由连接器把它们合并在一起。绝对段没有名字，因而也不能与其它段结合。

一个模块有一个或多个段，模块由用户指定的名字，模块的定义决定决定局部符号的作用域。一个目标文件有一个或多个模块。

为了存放段的信息，定义一个类：ASM_Segment 类。该类的对象存放下列信息：段的 ID 号，段名，段类型（CODE、XDATA、DATA、IDATA、BIT），再定位类型（PAGE、INPAGE、INBLICK、BITADDRESSABLE、UNIT、OVERLAYABLE），段基址（SegBase），段大小（SegSize），段地址计数器，段空标志等。

每个段里又分片段。片段是处在同一个段中但地址不连续的程序块。为了表示段的各个片段，再定义一个类：SegTriple 类，该类存放着汇编源程序中地址连续的指令块的开始行和结束行的指针。这些片段以链表的形式存放在 ASM_Segment 类的对象里。

### OBJ 目标模块

OBJ 文件是由目标模块组成的。在汇编阶段中，通常一个源文件产生一个目标模块。目标模块有特定的结构，需要有一个专门的管理器对其进行管理，由此定义 OBJModule 类。OBJModule 类的对象包含着目标模块的信息，如模块名称，创建该模块的汇编器或连接器的 ID，该模块中用到的寄存器组，各个段的定义，公共变量的定义，外部变量的定义，局部变量的定义，数路块的内容，需由连接器修正的条项和调

试信息，可执行代码的行号等等。OBJModule 类负责整个目标模块的拼装。

### 目标记录模块

目标模块由一条一条不同的记录联合而成。每条记录都有各自的记录类型号，记录长度，记录内容和记录校验和。而记录内容则根据记录类型的不同而相异甚远。记录本身构成一个实体，我用一个类来表示：OBJrecord。每个 OBJrecord 对象存放着记录的类型号，记录长度，校验和，数据块首址，数据块大小。数据块是在内存堆里面开辟的，由 OBJrecord 对象自行管理。它能对外提供如下几个服务：往数据块内添加一个字节 AddByte()，往数据块内添加一个字 AddWord()，往数据块内添加一个长字 AddLong()，往数据块内添加一个字串 AddJstr()，往数据块内添加一个数据块 AddJdat()。

### OBJ 文件的操作模块

为了更好地控制 OBJ 文件的操作，因此定义了一个 OBJfile 类，所有对 OBJ 文件的操作都封装在这个类提供的公共接口上。OBJfile 类的对象存放有目标文件的文件名，提供打开文件，关闭文件和写文件的服务。另外一个最重要的服务是 WriteARecord()，它的功能是写一个 OBJrecord 记录到文件上。

### 列表文件的操作模块

列表文件是纯文本文件，也有一定的格式。我把它也定义成一个类：LISTFile 类。这个类负责列表文件生成的所有动作。

### 主程序模块

主程序 main（）的功能是组织全盘的工作流程：输出汇编器版本号，接受汇编参数，根据源文件名创建 AsmFile 对象、OBJfile 对象、LISTFile 对象，然后创建宏汇编器 MacroAsmber 的对象 masm，调用 masm.ComplyFile()，最后输出程序结束信息。

为了更好地方便程序移植到 DOS 下，所有基本数据类型都用 typedef 语句定义成自用的类型名。如;

```
//---------------------------------------------------------------------------
typedef   signed char      int8;     //    -127 ~ 128
typedef unsigned char      int8u;    //       0 ~ 255
typedef   signed short     int16;    //  -32768 ~ 32767
typedef unsigned short     int16u;   //       0 ~ 65535
typedef   signed long      int32;    //   - 21M ~ + 21M
typedef unsigned long      int32u;   //       0 ~ + 42M
typedef   signed char      boolean;  // only have 0 & 1, but can be -1 too.
typedef unsigned char      ERR;      // 0 = No error; 1 ~ 255 = Have error.
typedef unsigned __int64   int64u;   //       0 ~ +oo
typedef   signed __int64   int64;    //     -oo ~ +oo
//---------------------------------------------------------------------------
const ERR OK_no_Err = 0;
const ERR Have_Errs = 1;
//---------------------------------------------------------------------------
```

### 其他模块

为了更好的管理字串，特定义字串类：Astring 类。该类的对象适合于长度不定的字符串的管理，尤其对于长度不断增加的字串来说会有比较高的效率。这个类提供多种构造器，以满足不同的字串的初始化构造的要求。它能自动管理内存的分配和收回。它能方便地进行取长度，字串比较、取子串、赋值、连接、复制、截断、取字符等的操作。所有的字串操作通过数据封装和信息隐蔽及接口规范，能保证程序正确运行。

在我的程序实现中，还定义了增强功能的 JString 类。该类继承了 Astring 类的所有功能，同时还增

加了以下功能：字符串转变为全大写 ToUp()；字符串转变为全小写 ToDown()；生成一个除去首字符的字符串（源字串本身不改变）DelHead()；删除字串的最后一个字符（源字串本身被改变）DelTail()；判断第一个字符是否为数字 FirstCharDigit()；判断字串是否数值串，并返回其进制值 IsValueStr()；把字串转换成 32 位整数 TransToint32u()；把字串中的分隔符‘$’去掉 dollarCat()等。这些服务都是为我的宏汇编器特别设计的。

为了便于 Token 码以及一些常量的管理，所有的 Token 码的定义，常量的定义，段类型、标号类型和再定位类型的定义，错误号、警告号的定义都填在同一个头文件 Tokens_H.h 里面。

AsmLine 类的对象里存放的是 Token 流。这每一个 Token 实质上构成一个实体。于是我定义了 Tokenfield 类，让它们作为 Tokenfield 类的一个对象。它存放的信息有：Token 码值，Token 的 Value 值，Token 的类别，Token 的名字，下一个 Token 的指针。

另外，还定义一个专门处理 Token 链表的类：TokenOper 类。它提供了形形式式的对 Token 链表的操作。这些操作全部是公共接口，供所有需要操控 Token 流的函数调用。

在进行表达式求值的时侯，考虑到在进行数值运算的同时，还要进行类型运算和段的匹配，以及判断结果是否需要被修正（Fix），所以需要定义一个类，名为 TriVal 类。它的对象是下面元素的组合：一个 32 位的数值，该值所参照的段的指针（如果是外部值则是外部变量的 ID），该数值的类型（NUM、CODE、DATA、BIT、XDATA、IDATA、SEG、REG 等），该数值的属性（绝对值、偏移值、外部值），还有修正类型（None，LOW，HIGH 等）。TriVal 类的对象贯穿在表达式求值过程中，作为操作数栈（TriValStack）压进、弹出的单位，扮演着重要的角色。

汇编过程中离不开对数据块的操作。为了更好地对数据块进行操作，我定义了 JdatBlock 类。它跟 JStrings 类一样，是个自动分配和回收内存的独立体。定义在该类上的服务有：取得数据块大小 GetLen()；清除数据块 clear()；转换并添加 JStrings 类的对象到数据块中 LoadData()；数据块合并 merge()；添加一个字节 AddByte()；添加一个字 AddWord()。

另外，在研究 OBJ 文件的过程中，由于 OBJ 文件都是二进制码文件，没有一个好的查看器是不可能看到它里面的内容的。于是我自己动手写了一个 OBJ 文件查看器，在 DOS 命令行中调用。这个查看器把晦涩难懂的二进制代码翻译成了一条一条可视性的文本记录，并写到一个文本文件里。

## 六、实现方法

### 所有的类定义列表

| | | | |
|---|---|---|---|
| ASM_Segment | 汇编段类 | JLabelNode | 标号结点类 |
| AsmFile | 汇编源文件类 | Jstring | 增强字串类 |
| AsmLine | 汇编行类 | JObject | 抽象对象类 |
| Astring | 字串类 | LabelManager | 标号管理器类 |
| Bstring | 基类字串类 | LISTFile | 列表文件类 |
| Dat4ary | 目标代码组类 | MacroAsmber | 宏汇编器类 |
| ErrPrintFile | 错误输出文件类 | MacroDefBody | 宏定义体类 |
| ErrWarnMsg | 错误和警告信息类 | MacroRegList | 宏定义登记表类 |
| ExtSymDef | 外部符号定义类 | MacrosManager | 宏管理器类 |
| IfeStack | 条件汇编状态栈类 | MacroVarList | 宏变量表类 |
| InstSystem | 指令系统类 | MacroVarNode | 宏变量节点类 |
| JdatBlock | 数据块类 | OBJ_Viewer | OBJ 文件查看器类 |

| | | | |
|---|---|---|---|
| OBJfile | 目标文件类 | SpecialRegs | 特殊寄存器描述结构体 |
| OBJmodule | 目标模块类 | Tokenfield | Token 实体类 |
| OBJrecord | 目标记录类 | TokenOper | Token 链表操控类 |
| OpndFixRec | 修正记录类 | TokenStack | Token 栈类 |
| OPset | 指令描述结构体 | TriVal | 计算元组类 |
| Resvs | 保留字描述结构体 | TriValStack | 计算元组栈类 |
| SegTriple | 汇编段的片段类 | ValueStack | 值域栈类 |

下面分步介绍各阶段中各部分的实现方法和各模块之间的联系。

## 主程序实现

第一步：输出版本信息。

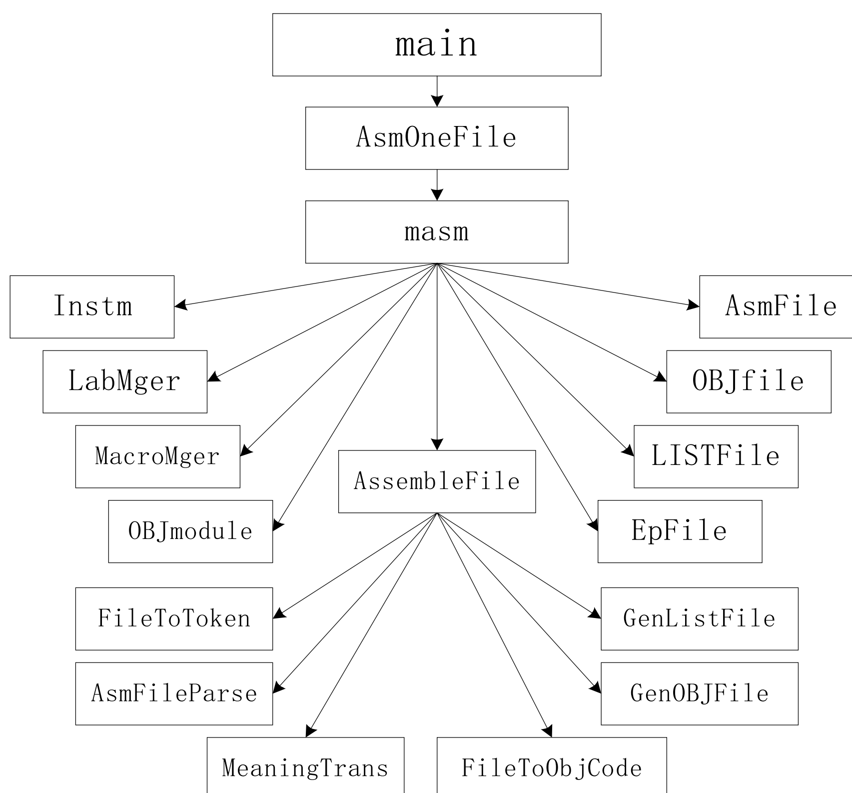
第二步：根据命令行调用参数创建汇编源文件类 AsmFile 的对象 SrcFile，创建 OBJ 目标文件的对象 ObjFile，创建 LST 列表文件的对象 LstFile。

第三步：以 SrcFile、ObjFile、LstFile 为输入，创建宏汇编器的对象 masm。

第四步：调用宏汇编器的服务：masm.ComplyFile()，对源文件进行汇编。

第五步：输出汇编结束信息并退出。

**主程序调用层次图：**




## 宏汇编器对象 masm 的创建

MacroAsmber 类的对象 masm 是由主程序创建的。它是整个宏汇编器的核心。在这个对象中包含了一些全程的变量，它们是：

Pass： 布尔值类型。它是一个扫描标记，在第一次扫描中 Pass 的值为 true，在第一次扫描中 Pass 的值为 false。

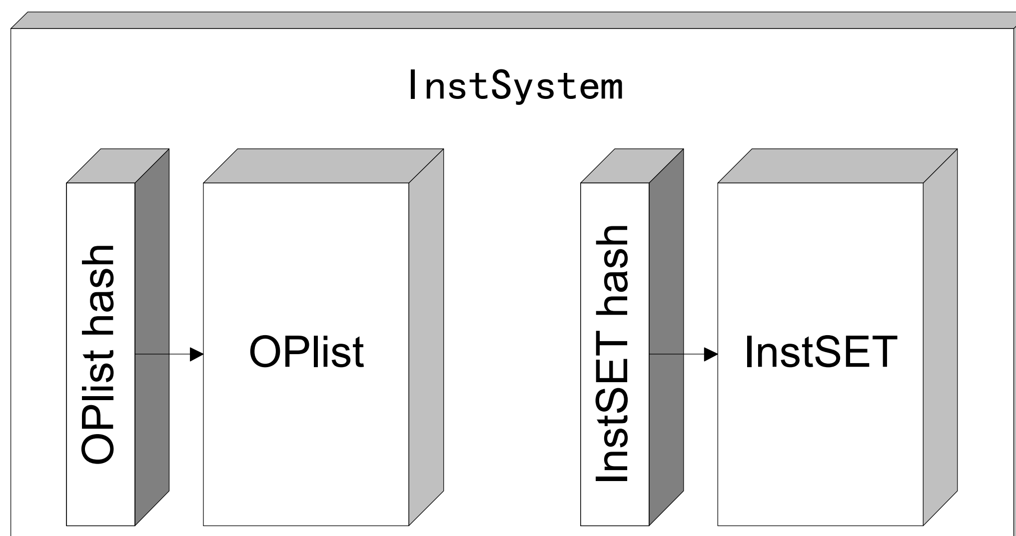
AsmFn ：一个指向当前汇编文件的指针。
objFilePt：一个指向当前目标文件的指针。
LstFilePt：一个指向当前列表文件的指针。
JModulePt：一个指向当前目标模块管理器的指针。
Ln ：一个指向当前汇编行的指针。
SegPt ：一个指向当前段的指针。
LabMger ：一个指向标号管理器的指针。
Instm ：一个指向指令系统管理器的指针。

在 MacroAsmber 的构造器中，给它们分别赋初始值：Pass 为 true；Fn 指向 SrcFile；Ln 指向 SrcFile 的第一行；objFilePt 指向 ObjFile；LstFilePt 指向 LstFile。新创建一个标号管理器，让 LabMger 指向它；新创建一个指令系统管理器，让 Instm 指向它。根据源文件的文件名，获取不带后缀的文件名，以此作为模块名的默认值，创建一个目标模块，然后让 JModulePt 指向它。给当前段指针赋初值，让它指向默认的 CODE 的绝对段。

这样，从 masm 被创建的一开始，masm 就具有了属于它自己的指令系统管理器、标号管理器、目标模块管理器。masm 可以对它们进行控制，直到 masm 被析构。

### 指令系统管理器 Instm 的创建

指令系统管理器从一开始就拥有几张静态表格，而且它们能被所有的指令系统管理器的对象共享。一张是指令描述表 OPlist，一张是保留字描述表 InstSET，一张是各指令需要的操作数的个数表，还有两张分别是指令描述索引哈希表和保留字索引哈希表，哈希表的表长均为 256，采用链地址法处理冲突。




指令描述表 **OPlist** 是由指令描述结构体组成的表。一个指令描述结构体 **OPset** 如下：

```
struct OPset
{ int8u OPnum;          // 指令 Token 码
  int8  ODmode[3];      // 操作数形式(最多可有三个操作数)
  int8u OPobj;          // 操作码目标码，十六进制 8 位数
  int8  act1;           // 动作码 1
  int8  act2;           // 动作码 2
  int8u len;            // 该指令长度
  OPset* next;          // 指向下一个指令描述结构体的指针
```

```
}; // end OPset
```

一些例子：

**指令描述结构体 OPset 表**（局部）：

| OPnum | ODmode0 | ODmode1 | Odmode2 | OPobj | act1 | act2 | len |
|---|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … | … |
| ADD | _A | _R0 | -1 | 0x28 | -1 | -1 | 1 |
| INC | direct | -1 | -1 | 0x05 | BRef_1 | -1 | 2 |
| JNB | _bit | _slabl | -1 | 0x30 | BRef_1 | Wrel_2 | 3 |
| CJNE | _aR0 | _xx_ | _slabl | 0xB6 | BRef_2 | Wrel_3 | 3 |
| MOVC | _A | _aA_PC | -1 | 0x83 | -1 | -1 | 1 |
| DIV | _AB | -1 | -1 | 0x84 | -1 | -1 | 1 |
| MOV | direct | direct | -1 | 0x85 | BRef_2 | BRef_1 | 3 |
| AJMP | _sdirt | -1 | -1 | 0x01 | H3L8_0 | -1 | 2 |
| LJMP | _num | -1 | -1 | 0x02 | H8L8_1 | 0 | 3 |
| … | … | … | … | … | … | … | … |

保留字描述表 InstSET 是由保留字描述结构体组成的表。一个保留字描述结构体 **Resvs** 如下：

```
struct Resvs
{ const char* name;     // 保留字名称（ASCII 码）
  int16u  tken;         // 保留字对应的 Token 码
  int16u  typ;          // 该保留字所属的类别（Pdw、Irv、Rgr 等）
  Resvs*  next;         //指向下一个保留字描述结构体的指针
}; // end Resvs
```

一些例子：

| Name | Tken | typ |
|---|---|---|
| “MOV” | MOV | Irv |
| “SEGMENT” | SEGMT | Pdw |
| “SHR” | SHR | SHR |
| “HIGH” | HIGH | HIGH |
| “PUBLIC” | PUBLICKn | Pdw |
| “A” | _A | Rgr |
| “DPTR” | _DPTR | Rgr |
| “CODE” | CODE | Pdw |
| “END” | END | END |

指令系统管理器被创建时，由它的构造器给两张哈希表赋初始值。构建好了两张哈希表之后，管理器内部就可以通过 GetHashNo()服务和 GetEntryNo()服务来分别得到保留字描述结构体和指令描述结构体的入口地址。指令系统管理器对外提供下列几个服务：

InsRecg()：指令识别，得到指令的 Token 码。
InstArgvSum()：查表得到指令所需操作数的个数。
TokenToName()：从 Token 码回到 ASCII 码的转换。
PackToKey()：操作码以及操作数的寻址方式的组合打包成键字。
OPsetRecg()：由指令操作码以及操作数的寻址方式到 OPset 的转换。

## 标号管理器 LabMger 的创建

标号管理器 LabMger 在创建时需获取指向宏汇编器 masm 的指针，这是通过 masm 传递 this 指针给 LabMger 的构造器来实现的。因为 LabMger 内部有时候需要访问 masm 的成员数据和调用 masm 的服务，其中最主要的是 Pass 以及出错处理 OutputErr()和 OutWarning()。

标号管理器 LabMger 的构造器完成以下工作：把需要预定义的常量、变量及特殊功能寄存器等填在标号表中。这会用到一个特殊功能寄存器描述结构体 SpecialRegs。

```
struct SpecialRegs
{ char* name;               // 特殊功能寄存器的名字。
  int8u vtyp;               // 类型（DATA、BIT、NUM 等）。
  int8u addr;               // 字节地址或位地址。
}; // end SpecialRegs
```

需要预定义的特殊功能寄存器存放在一个静态表 **SRegAddr** 中：

| name | Type | value |
|---|---|---|
| “P0” | LB_DATA | 0x80 |
| “SP” | LB_DATA | 0x81 |
| … | … | … |
| “PCON” | LB_DATA | 0x87 |
| “TCON” | LB_DATA | 0x88 |
| … | … | … |
| “IT0” | LB_BIT | 0x88 |
| “IE0” | LB_BIT | 0x89 |
| “CY” | LB_BIT | 0xd7 |
| … | … | … |

LabMger 的构造器会把这张表格里的数据构造成一个个标号结点 JLabelNode，作为表格管理的单位，供汇编器随时“查阅”。

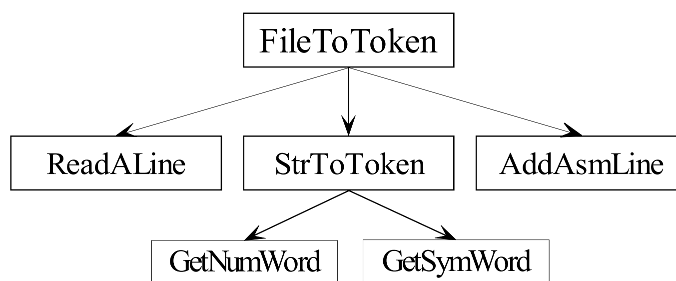
## 宏汇编器的服务 ComplyFile()

宏汇编器对源文件进行汇编的所有过程全部由服务 ComplyFile()来完成。他先后调用宏汇编器内部服务 FileToToken(),AsmFileParse(),CreateVarTable(),FileToObjCode(),GenOBJFile()和 GenListFile()来完成整个汇编流程。每做完一个流程，都检查是否产生了汇编错误。如果错误产生，则不再进入下一阶段而退出。

## 宏汇编器的词法分析服务 FileToToken()

在调用词法分析服务 FileToToken()之前，源文件应该是已经打开了的。调用 FileToToken()时，首先，它定义一个 ERR 类型的变量 erra，赋初始值为 OK_no_Err，表示程序在开始时是没有错误的。然后从源文件的第一行开始扫描，先读入一行 ASCII 码，放进读缓冲区中。接着创建一个新的 AsmLine 对象，让当前汇编行指针 Ln 指向它。给新行赋上一个行号。然后调用内部服务 StrToToken()，把该行 ASCII 码转换成 Token 流。调用后判断其返回值是否表示有发生错误，如果有错，设置 erra 为 Have_Errs 值。再判断该行是否空行，如果是空行，则不把 AsmLine 对象添加入 AsmFile 的对象 SrcFile 中。如果不是空行，则调用 AsmFile 类的服务 AddAsmLine()把该 AsmLine 对象加进 SrcFile，使之成为 SrcFile 中的一行 Token 链。最后判断文件是否遇到“END”，如果没遇上，则重复以上动作读入下一行 ASCII 码。如果源文件没有用“END”指令结尾，会产生一个 FileNeedEnd 警告，并自动为 SrcFile 填上 END。

**FileToToken()服务调用层次图：**



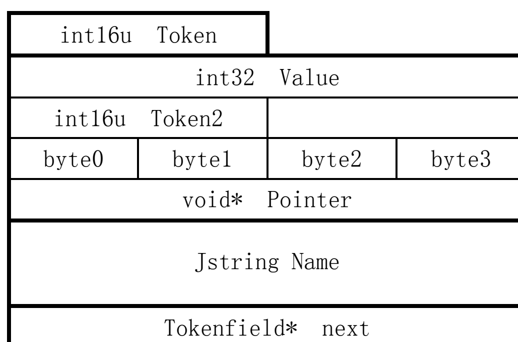

如果词法分析服务 FileToToken()正确返回，则整个源文件的 ASCII 码都已经正确地转换成 Token 流的形式。否则表示发生了词法错误，将不能进入语法分析阶段。

## 关于 Tokenfield 类

所谓 Token 流就是一个个“Token”单元组成的序列。这些“Token”单元被定义为 Tokenfield 类的对象。Tokenfield 的结构如下：

```
class Tokenfield
{ public:
  int16u Token;                    // Token
  union
  {  int32 Value;                  // Value
     int16u Token2;
    Tokenfield* ExpPt;
    struct OPset* Opset;
     class JLabelNode* LabPt;
     class JdatBlock* DatPt;
     class Dat4ary* D4ary;
     int8u byte[4];
  }; // end union
  JStrings Name;                   // Name
  Tokenfield* next;                // next
};   // end Tokenfield
```

**Tokenfield 结构图：**




Token 域里面存放的就是 Token 码，占用两个字节。然后下来的四个字节是一个共用体：它可以是一个 32 位的有符号整数（Value）；可以是 16 位的无符号整数（作为第二个 Token 码 Token2）；可以是一个有 4 个字节组成的数组；也可以是一个万能的指针，根据不同的 Token 类型指向不同的对象（例如指向一个表达式、一个指令描述结构体、一个标号结点结构体、一个数据块等等）。接着是一个 Jstring 对象。最后的 4 个字节是指向下一个 Tokenfield 的指针（next）。

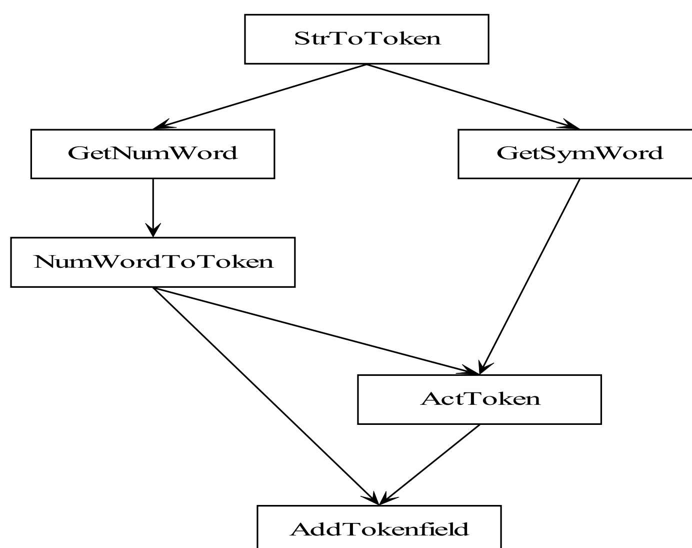
Tokenfield 类的对象贯穿整个汇编过程，是汇编器最为重要的一个操控对象。一个汇编过程，实质上就是这些“Token”组成的集合在不同的阶段上不断的变形、进化、改变组合的方式的过程。在不同的阶段上，“Token”集呈现出不同的形态。词法分析阶段是从源文本 ASCII 码生成 Token 码的阶段，就是这些会

变形的“小精灵”的诞生阶段。

## 宏汇编器的服务 StrToToken()

宏汇编器 MacroAsmber 类中有一个内部服务 StrToToken()，用以提供给 FileToToken()调用，它把一行 ASCII 文本字串转换成 Token 序列。它需要调用一个全局函数：GetNumWord()。这个函数的功能是从文本行上获取一个符合单词构词法的单词或一个符合数值构词法的数值。StrToToken()首先把光标移到文本行头，调用 GetNumWord()，得到一个单词（这个单词可能是一个数值的 ASCII 码形式）。如果成功得到一个这样的单词，则调用另外一个汇编器的内部服务 NumWordToToken()把单词转化成 Token。但是如果得不到这样的单词，则可能是遇到了一个符号或已经到了行末。于是调用另外一个汇编器的内部服务 GetSymWord()，把 ASCII 符号转换成符号 Token。最后还要调用一个内部服务 ActToken()，以决定得到的 Token 是否被添加进 Token 流中。

**StrToken()模块调用层次图：**




以上工作将循环进行，直到读入字符为行末符，或者遇到词法错误。如果遇到词法错误，则会把一个特殊的 Token——NormalERR 作为结点添加进 Token 流，而且把文本行中未被读入的字符忽略。

在这过程中，如果 SrcFile 中含有大块的注释（使用/*和*/），则注释里的所有文本都会被忽略。

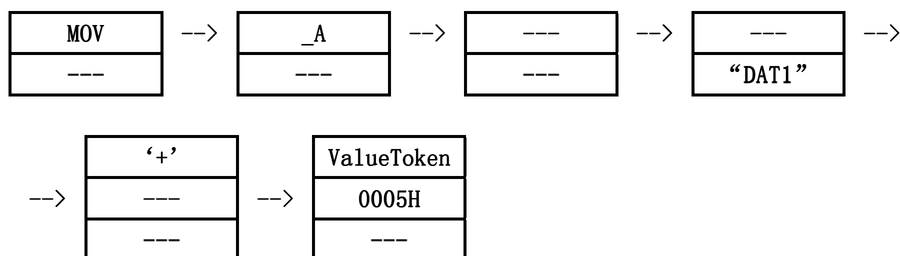
函数 GetNumWord()的功能是从 ASCII 码文本中获取一个单词或数。以可见符（非控制符）开头，中间由字母或‘?’、‘_’组成的不间断字符串叫做单词和数。例如 ABacd_34 是一个单词，345ABH 是一个数。

NumWordToToken()的功能就是把单词串和数串转化为 Token。字母会转换成大写。它首先判断 word 是不是数串，若是，则作为“Value”Token 添加进 AsmLine。如果不是，而是保留字，则作为“保留字”Token 添加进 AsmLine。如果是特殊字符串，则把它作为“符号”Token 添加。

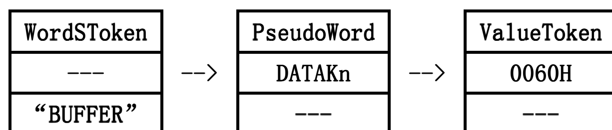
GetSymWord()的功能就是从 ASCII 码文本中获取一个标点符或运算符，并换算为 Token 码。例如‘@’、‘#’、‘$’、‘&’、‘+’、‘-’、‘*’、‘/’、‘%’、‘.’等。

ActToken()的功能是根据现在的 Token 结点和上一 Token 结点的关系，决定当前 Token 结点是添加到 Token 链表中，还是替换 Token 链表的最后一个 Token 结点，又或是根本不添加而直接把它忽略。

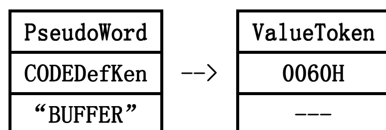
经过词法分析之后，像“MOV a,# Dat1+5”这样的语句会成为以下这样的 Token 流：



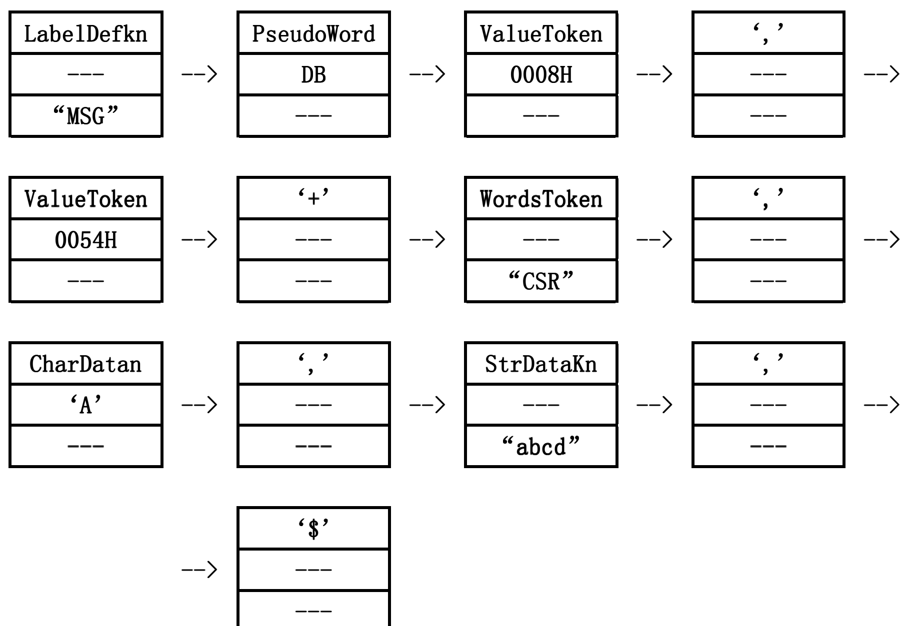

像“Buffer DATA 60H”这样的语句在不经过 ActToken()的调用下会成为以下这样的 Token 流：




但经过 ActToken()的调用下会成为下面这样的 Token 流：




可见经过调用 ActToken()后，Token 流变得简练些。Token 流形式越简单，处理起来就越方便。
再看下面一个例子：“Msg：DB 8，54h+CSR，‘A’，“abcd”，$”
经过词法分析（调用 ActToken()后）：




词法分析之后，整个源文件都被转化成 Token 流存储在内存中。它们还以 AsmLine 类的对象有组织地结合在一起，每个 AsmLine 对象对应源文件中的一个汇编行的相应的 Token 组。

如果没有发生致命的词法错误，就会进入语法分析阶段。

## 关于语法规则

一个正确的汇编行的语法如下：（方括号表示它的内容可省略）

（1） 空行

（2） 标号：
（3） ［标号：］ 汇编指令语句
（4） ［标号：］ 汇编伪代码语句
（5） 段定义
（6） 外部变量声明
（7） 公共变量声明
（8） END 语句

一个正确的汇编指令语句是：

汇编指令助记符 ［操作数 1］［，操作数 2］［，操作数 3］

一个正确的汇编伪代码语句是：

伪代码定义符 ［表达式列表］

其中伪代码定义符如：EQUDefken、SETDefken、CODEDefKen、DATADefKen、XDATADefKn 等等。
表达式列表是由数值表达式、字符、字串组成的，它们之间以‘，’分隔。

段定义语句是：

SEGMTDefkn 段类型 ［再定位类型］

其中段名已经隐含在 SEGMTDefkn Token 中。
段类型有：CODE、XDATA、DATA、IDATA、BIT。
再定位类型有：PAGE、INPAGE、INBLOCK、BITADDRESSABLE、UNIT、OVERLAYABLE。

外部变量声明：

EXTRNKn 段类型或 NUMBER 型 （符号表）

公共变量声明：

PUBLICKn 符号名［，符号名［，…］］

END 语句:

END 或 end

所谓“操作数”是如下的形式：
（1） 数值表达式
（2） # 数值表达式
（3） @ 数值表达式
（4） / 数值表达式
（5） A
（6） AB
（7） C
（8） DPTR
（9） Rn （ n = 0 ～ 7）
（10） @Ri （ i = 0，1 ）
（11） @ A + DPTR
（12） @ A + PC

所谓“数值表达式”是如下的形式：
（1） 单个数值
（2） 单个变量
（3） ‘$’

（4） （ 数值表达式 ）
（5） 数值表达式 . 数值表达式
（6） NOT 数值表达式
（7） HIGH 数值表达式
（8） LOW 数值表达式
（9） ＋ 数值表达式
（10） － 数值表达式
（11） 数值表达式 * 数值表达式
（12） 数值表达式 / 数值表达式
（13） 数值表达式 % 数值表达式
（14） 数值表达式 + 数值表达式
（15） 数值表达式 - 数值表达式
（16） 数值表达式 SHL 数值表达式
（17） 数值表达式 SHR 数值表达式
（18） 数值表达式 AND 数值表达式
（19） 数值表达式 OR 数值表达式
（20） 数值表达式 XOR 数值表达式
（21） 数值表达式 >= 数值表达式
（22） 数值表达式 <= 数值表达式
（23） 数值表达式 = 数值表达式
（24） 数值表达式 > 数值表达式
（25） 数值表达式 < 数值表达式

类型运算规则：
‘A’代表绝对量（Absolute），‘R’代表可重定位地址（Relocatable）或外部地址。
（1） A ＋ A ＝ A
（2） A ＋ R ＝ R
（3） R ＋ A ＝ R
（4） R ＋ R ＝ Error！（可重定位地址不可相加！）
（5） A － A ＝ A
（6） A － R ＝ Error！（绝对量不能减去可重定位量！）
（7） R － A ＝ R
（8） R － R ＝ A （只有这两个可重定位地址都在同一个段中才成立）
（9） A * A ＝ A
（10） A / A ＝ A
（11） A MOD A＝ A
（12） A . A ＝ A
（13） A AND A ＝ A， A OR A ＝ A， A XOR A ＝ A
（14） A == A ＝ A, A >= A ＝ A, A <= A ＝ A, A < A ＝ A, A > A ＝ A, A <> A ＝ A
（15） R == R ＝ A，R >= R ＝ A，R <= R ＝ A，R < R ＝ A，R > R ＝ A，R <> R ＝ A，(只有可重定位地址都在同一个段中才成立)
（16） NOT A ＝ A
（17） NOT R ＝ Error！（可重定位地址不可作取反运算！）
（18） HGIH A ＝ A
（19） LOW A ＝ A
（20） HIGH R ＝ R
（21） LOW R ＝ R
（22） A SHL A＝ A

（23） A SHR A＝ A

‘N’代表 NUMBER，‘C’代表 CODE，‘D’代表 DATA，‘X’代表 XDATA，‘I’代表 IDATA，‘B’代表 BIT，‘S’代表 SEG，‘R’代表 REGISTER。 ‘?’则代表以上各类型中除 NUMBER 外的任何一个类型。

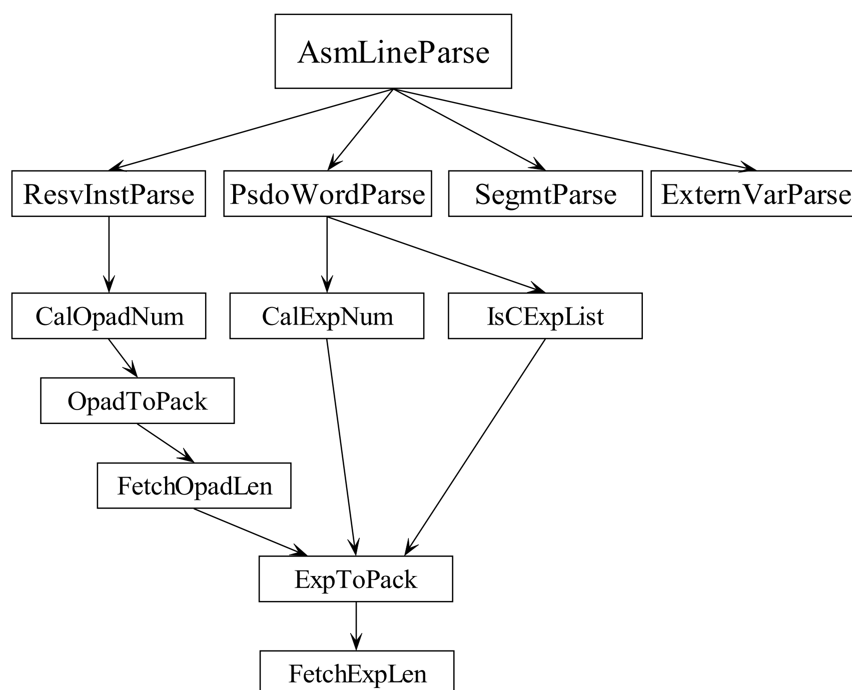
（1） N ＋ N ＝ N
（2） N ＋ C ＝ C，N ＋ D ＝ D，N ＋ X ＝ X，N ＋ I ＝ I，N ＋ B ＝ B，……（以上各式可简记为：N ＋ ？ ＝ ?，要注意这两个‘?’是同一个类型。下同。）
（3） ？ ＋ N ＝ ？ （例如，CODE ＋ 5 ＝ CODE。）
（4） ？ ＋ ？ ＝ Error！（例如，CODE ＋ CODE 是没有意义的。）
（5） N － N ＝ N
（6） N － ？ ＝ Error！（例如，16 － CODE 也是没有意义的。）
（7） ？ － N ＝ ？ （例如，DATA – 4 ＝ DATA。）
（8） ？ － ？ ＝ N （两个同类型的数相减，得到的是它们的相对偏移。）
（9） N ＊ N ＝ N
（10） N ／ N ＝ N
（11） N MOD N＝ N
（12） ？ . N ＝ B （例如，TCON 是 DATA，TCON.5 是 BIT，也就是 TF0。）
（13） N AND N ＝ N
（14） N AND ？ ＝ N
（15） ？ AND N ＝ N
（16） ？ AND ？ ＝ Error！（两个地址相与无意义！）
（17） N OR N ＝ N，N OR ？ ＝ N，？ OR N ＝ N，？ OR ？ ＝ Error！
（18） N XOR N ＝ N，N XOR ？ ＝ N，？ XOR N ＝ N，？ XOR ？ ＝ Error！
（19） N == N ＝ N， N == ？ ＝ N， ？ == N ＝ N， ？ == ？ ＝ N
（20） N >= N ＝ N， N >= ？ ＝ N， ？ >= N ＝ N， ？ >= ？ ＝ N
（21） N <= N ＝ N， N <= ？ ＝ N， ？ <= N ＝ N， ？ <= ？ ＝ N
（22） N ＞ N ＝ N， N ＞ ？ ＝ N， ？ ＞ N ＝ N， ？ ＞ ？ ＝ N
（23） N ＜ N ＝ N， N ＜ ？ ＝ N， ？ ＜ N ＝ N， ？ ＜ ？ ＝ N
（24） N <> N ＝ N， N <> ？ ＝ N， ？ <> N ＝ N， ？ <> ？ ＝ N
（25） NOT N ＝ N， NOT B ＝ B
（26） HIGH N ＝ N， HIGH ？ ＝ N
（27） LOW N ＝ N， LOW ？ ＝ N
（28） N SHL N ＝ N，N SHR N ＝ N
（29） ？ SHL N ＝ N，？ SHR N ＝ N
（30） N SHL ？ ＝ Error！
（31） ？ SHL ？ ＝ Error！

## 宏汇编器的语法分析服务 AsmFileParse()

语法分析的任务是判断每一个 AsmLine 的对象中 Token 链表是否符合以上的语法规则，以及汇编指令和汇编伪代码后面所带的参数的个数是否合法。如果有语法错误，报告错误并退出汇编。如果没有语法错误，则对 Token 进行变形，把 Token 流构造成语法树。

首先，AsmFileParse()定义一个 ERR 类型变量 erra，赋初始值为 OK_no_Err。然后循环调用 AsmLineParse()服务，对每一个 AsmLine 对象进行扫描，检查它们是否都符合汇编行的语法规则。若其中有一个不符合，把 erra 设置为 Have_Errs。最后返回 erra 值。在 AsmLineParse()服务中，需要根据不同的语句调用不同的语法分析模块：ResvInstParse()、PsdoWordParse()、SegmtParse()、ExternVarParse()等。

**AsmLineParse()服务调用层次图：**




### 汇编指令语法分析服务 ResvInstParse()

首先，它调用 InstSystem 类的服务 InstargvSum()取得该指令应该所带的操作数个数，并存放在argvsum 变量里。然后调用 TokenOper 类的服务 CalOPadNum()，获取汇编行中输入的操作数的个数。比较这个数是否与 argvsum 相等，如果不相等就说明有错误，要么多了，要么少了，根据具体情况报告错误信息。在 TokenOper 类的服务 CalOPadNum()中，它同时完成一个工作：把“操作数”Token 打包，即调用OpadToPack()服务。它能把符合“操作数”语法的 Token 组“打包”成一个“OperandKn”的 Token。

### 汇编伪代码语法分析服务 PsdoWordParse()

它根据不同的 Token 调用不同的模块进行语法分析。

像 ORG、EQU、SET、DS、DBIT 等后面只须带一个数值表达式的伪代码，调用 NeedOneExp()模块。

像 DB、DW 后面带一个表达式表（至少一个表达式）的伪代码，调用 NeedCExpList()模块。

像 CSEG、DSEG、XSEG、ISEG、BSEG 等可带一个绝对表达式或者不跟任何参数的伪代码，调用 Need01Exp()模块。

像 RSEG、NAME 后面必须跟一个字符串的伪代码，则调用 NeedOneWord()模块。

像 PUBLIC 后面带一个字串表（至少一个字符串）的伪代码，则调用 NeedWordList()模块。

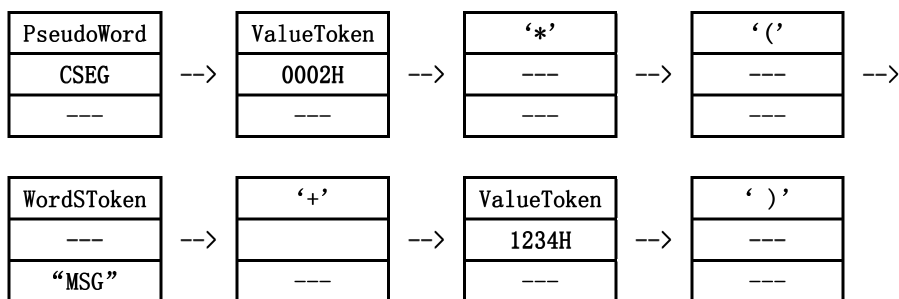
在这些模块中，无一例外地，需要调用一个 TokenOper 类的服务：CalExpNum()。该服务能计算伪代码后面所跟参数（表达式）的个数。

### TokenOper 类的服务 ExpToPack()

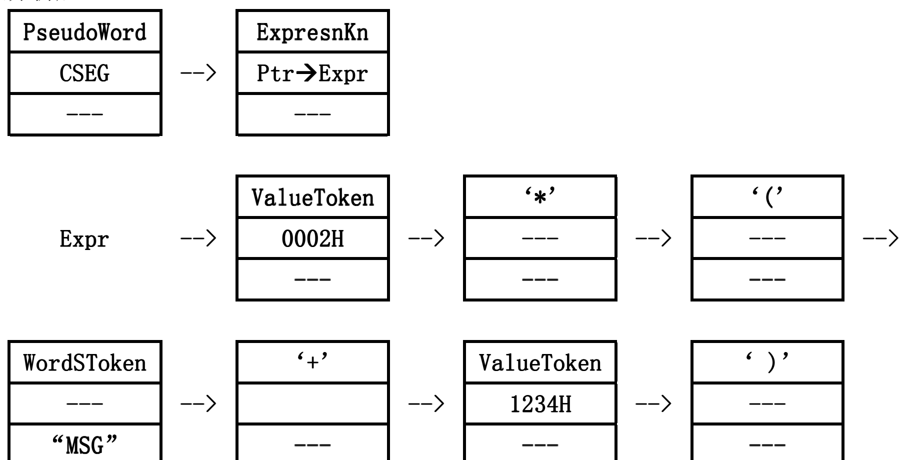
PsdoWordParse()服务需要调用 TokenOper 类的服务：CalExpNum()，而 CalExpNum()则需要调用TokenOper 类的服务 ExpToPack()。这个服务非常重要，它的功能是把构成表达式的那部分 Token 打包成一个“Package”，新创建成一个新的 Token 形式：Expresstion Token(ExpresnKn)。首先，它调用 FetchExpLen()模块，计算表达式的长度（即组成表达式的 Token 共有多少个），然后调用 ExpToToken()模块，把构成表达式的那部分 Token 连接到一个新创建的 Token——ExpresnKn 的指针域（Pointer）之下。

如：CSEG 2*(Msg＋1234H)

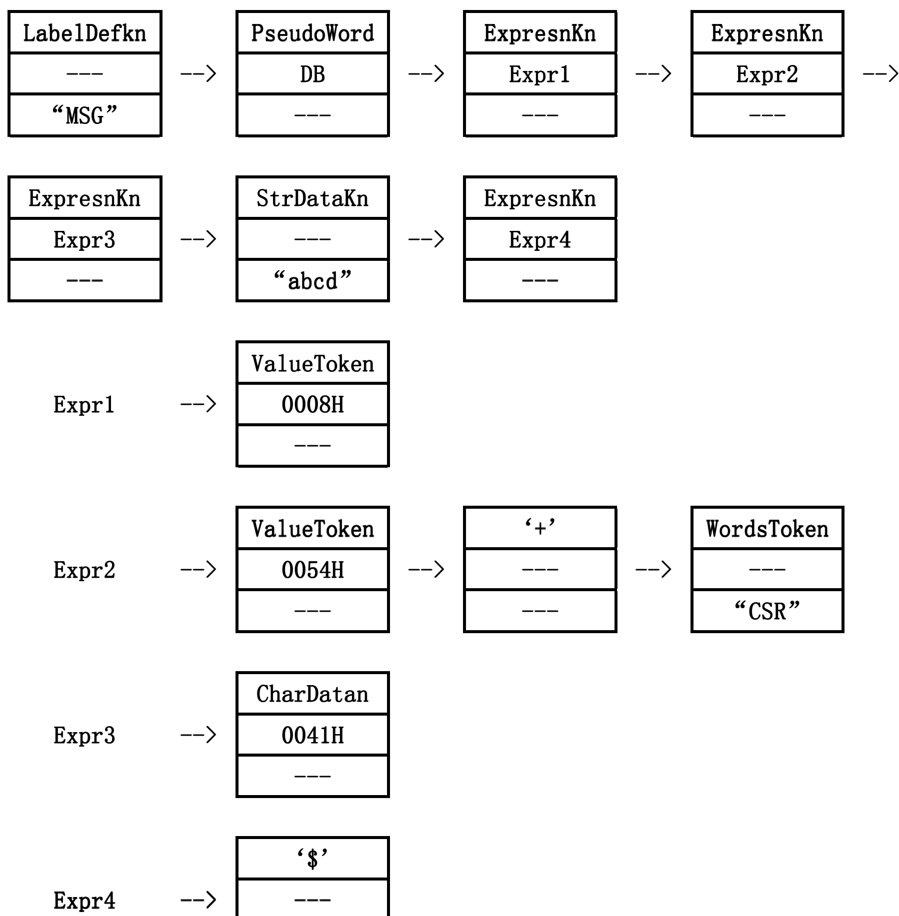
词法分析后：

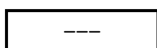



语法分析后：




又如上面的例子“Msg：DB 8，54h+CSR，‘A’，“abcd”，$”，语法分析后：






打包后的关于汇编伪命令 Token 序列的形式单一化为：汇编伪命令 Token＋表达式 Token 组。这为下一步进行语义分析提供了方便。

### TokenOper 类的服务 OpadToPack()

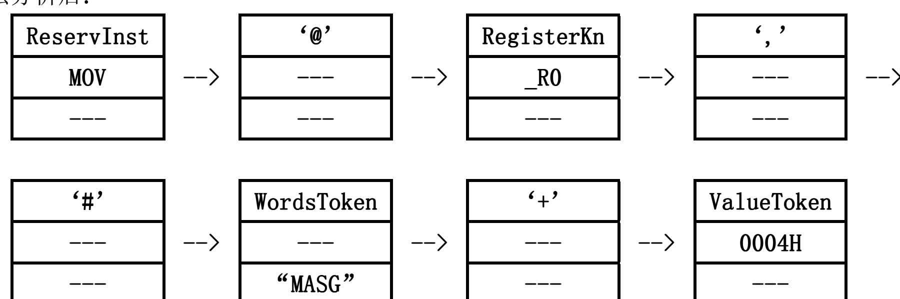
前面已经有所提及，这个服务的功能是把符合“操作数”语法的 Token 组“打包”成一个“OperandKn”的 Token。它实质上首先调用 FetchOpadLen()模块，取得操作数的长度，然后调用 OpadToToken()模块，把构成操作数的那部分 Token 连接到一个新创建的 Token——OperandKn 的指针域（Pointer）之下。

FetchOpadLen()也是 TokenOper 类中的一个服务，它首先判断当前光标所指向的 Token 是否为‘@’、‘#’、‘/’这几个 Token，如果是，则操作数长度加一，然后光标前移一个 Token。接着，调用 ExpToPack()，判断以当前光标所指的 Token 为开始的 Token 序列是否为数值表达式，如果不是，报错；否则把它们打包组成 ExpresnKn Token。FetchOpadLen()调用后的结果，是操作数里面的数值表达式成分的 Token 序列打包成了 ExpresnKn Token，并且连同它前面的‘@’、‘#’、‘/’ Token 一起作为操作数成分，返回操作数的长度。这个长度值，将为下一步调用 OpadToToken()模块打下基础。OpadToToken()模块就是根据这个长度值决定把以当前光标所指的 Token 为开始的多少个 Token 打包成操作数 Token（OperandKn Token）。

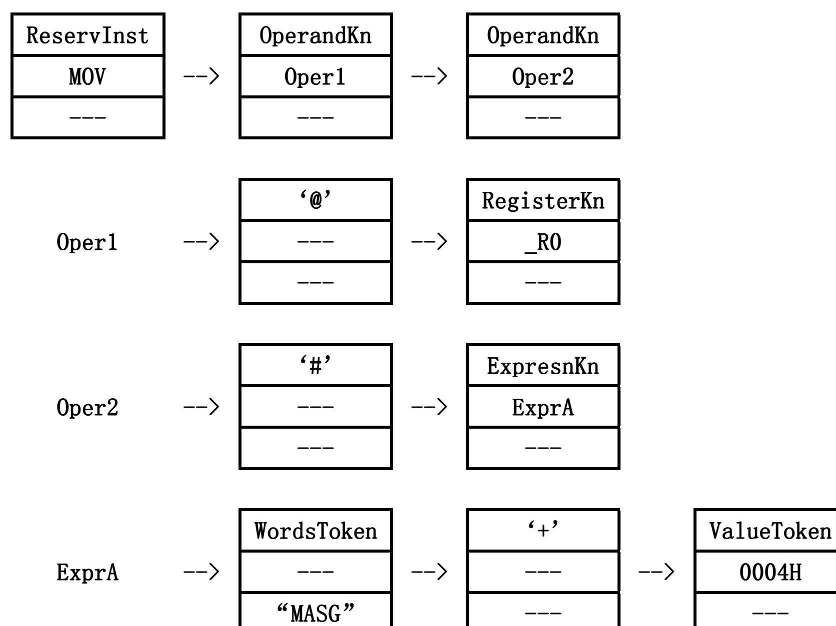
具体的情况如下例所示：

“MOV @R0，# MASG+4”

词法分析后：




语法分析后：



打包后的关于汇编指令 Token 序列的形式单一化为：汇编指令 Token＋操作数 Token 组。这也为下一步进行语义分析提供了方便。

## 宏汇编器语义分析服务 CreateVarTable()

源程序ASCII文本经过词法分析成为Token流，为语法分析打下基础;语法分析再把这些无组织的Token序列根据语法意义组织起来，成为形式比较单一的 Token 格式，这又为对它进行语义分析作了铺垫。

CreateVarTable()服务是通过对 Token 流的两次扫描，对汇编伪指令 Token 序列和汇编指令 Token 序列各自进行解释，并把所有的段定义、公共变量声明、外部变量声明、符号常量、局部变量和标号定义等登记在符号表格中。

首先，CreateVarTable()服务把宏汇编器的 Pass 成员变量设置为 true，接着调用一次 FileFindVar()服务，对整个 AsmFile 对象中的所有 AsmLine 对象的 Token 流进行一次扫描。然后，CreateVarTable()再把宏汇编器的 Pass 成员变量设置为 false，接着第二次调用 FileFindVar()服务，再对整个 AsmFile 对象进行一次扫描。在这两次扫描中，FileFindVar()的工作是不一样的。我把 Pass 的状态为 true 时进行的 Token 扫描称为“Pass1”扫描；而把 Pass 的状态为 false 时进行的 Token 扫描称为“Pass2”扫描。

扫描对象以汇编行（AsmLine）对象为单位。且先看 AsmLine 的定义：

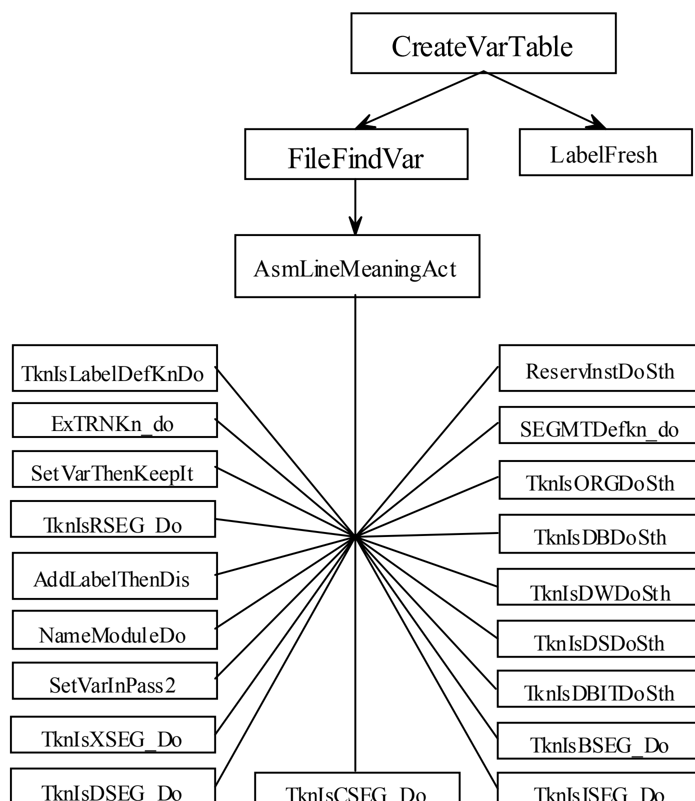
```
class AsmLine
{ private:
    Tokenfield* head;   // 指向头结点，一个有特殊用途的结点。
    Tokenfield* tail;   // 指向尾节点。
    union
    { int32u LineID;    // 备用
      struct
      { int16u NestNo;  // 初值为 0，当发生宏嵌套时，存储嵌套所在的层数。
        int16u LineNo;  // 存储行号，从 1 开始,数值唯一。未赋值时是 0。
      }; // end struct
    }; // end union
    int16u lc;                  // 当前行的偏移地址.
     int8u len;                 // 当前行的目标代码的长度。
      bool IsOK   :1;           // 说明该行的偏移地址（lc）是否已经得到。初值为 false。
      bool Enable :1;           // 是否启用。初值为设为 true。
     class ASM_Segment* sgpt;   // 当前行所属的段的指针。
     AsmLine* next;             // 指向下一个汇编行结点
}; // end class AsmLine
```

在这两次扫描之前，各 AsmLine 对象中的 Enable 值均为 true；IsOK 值均为 false。

在两次扫描的开始，FileFindVar()都要给宏汇编器的当前段指针 SegPt 赋初值，使之指向默认段——绝对段的 CODE 段。扫描从 AsmFile 对象的第一汇编行（AsmLine 对象）开始。

在“Pass1”扫描中，各个 AsmLine 对象中的所属段指针 sgpt 会被设置为当前段。当前段会随着伪指令的段选择指令而改变。但是 AsmLine 对象中的当前地址计数器 lc 的值还没有确定。在“Pass2”扫描中，才确定各 AsmLine 对象中的当前地址计数器 lc 的值。

**CreateVarTable()服务调用层次图：**




Pass1 扫描：
如果该汇编行被禁用（AsmLine 中的 Enable 值被设置为 false），则扫描忽略该行。
遇到空行，将其禁用。这样下一次扫描就会忽略该行。

对该汇编行的 Token 序列进行扫描，
（1） 如果是标号定义（LabelDef Token）：往标号表添加该标号。该标号在标号表中被登记后，就从汇编行 Token 流中删除它。把标号定义 Token 删除的目的，是相同的标号定义只能在标号表中被添加一次，在 Pass2 扫描中不能再为它登记一次。
（2） 如果是“END”指令 Token，将其所在的汇编行禁用。
（3） 如果遇到汇编指令（ReservInst Token），只须将其忽略，留给 Pass2 扫描才处理。
（4） 如果遇到段定义（SEGMTDefken Token），调用 SEGMTDefkn_do()模块，往目标模块中添加一个新段的段定义。最后将其所在的汇编行禁用。
（5） 如果遇到外部变量声明（EXTRNKn Token），调用 ExTRNKn_do()模块，往符号表添加一条外部变量记录。最后将其所在的汇编行禁用。
（6） 如果遇到 RSEG、CSEG、DSEG、XSEG、ISEG、BSEG 等选择段的汇编伪指令，则分别调用 RSEG_do()、CSEG_do()、DSEG_do()、XSEG_do()、ISEG_do()、BSEG_do()模块进行处理，并更新当前段指针 SegPt，使之指向所选择的段。最后将其所在的汇编行禁用。
（7） 如果遇到 NAME 伪指令（NAMEToKn Token），则调用 NameModuleDo()模块，把目标模块命名为 NAME 后面所带的字符串，然后将其所在的汇编行禁用。
（8） 如果遇到 EQU 伪指令（EQUsDefken Token），调用 AddLabelThenDisIt()模块，往符号表添加一个 EQU 语句决定的常量，该常量的类型由表达式的结果决定（默认为纯数值，即 LB_NUM）。最后将其所在的汇编行禁用。
（9） 如果遇到 SET 伪指令（SETsDefken Token），调用 SetVarThenKeepIt()模块，往符号表添加一个 SET 语句决定的变量，该变量的类型由表达式的结果决定（默认为纯数值，即 LB_NUM）。但最后并将不其所在的汇编行禁用。因为需要在扫描过程中跟踪这个变量的变化情况。
（10） 如果遇到 CODE 伪指令（CODEDefken Token），调用 AddLabelThenDisIt()模块，往符号表添

加一个 CODE 语句决定的常量，该常量的类型为 LB_CODE 型。最后将其所在的汇编行禁用。

（11） 如果遇到 DATA 伪指令（DATADefken Token），调用 AddLabelThenDisIt()模块，往符号表添加一个 DATA 语句决定的常量，该常量的类型为 LB_DATA 型。最后将其所在的汇编行禁用。

（12） 如果遇到 XDATA 伪指令（XDATADefken Token），调用 AddLabelThenDisIt()模块，往符号表添加一个 XDATA 语句决定的常量，该常量的类型为 LB_XDATA 型。最后将其所在的汇编行禁用。

（13） 如果遇到 IDATA 伪指令（IDATADefken Token），调用 AddLabelThenDisIt()模块，往符号表添加一个 IDATA 语句决定的常量，该常量的类型为 LB_IDATA 型。最后将其所在的汇编行禁用。

（14） 如果遇到 BIT 伪指令（BITDefken Token），调用 AddLabelThenDisIt()模块，往符号表添加一个 BIT 语句决定的常量，该常量的类型为 LB_BIT 型。最后将其所在的汇编行禁用。

（15） 如果遇到 ORG、DB、DW、DS、DBIT、PUBLIC、USING 伪指令，则忽略，留到 Pass2 才处理。

Pass1 扫描的结果，汇编行里的所有标号定义（LabelDef）Token 消失。但它们都已经登记在标号表里。标号表里的标号记录里存有指向创建它的汇编行的指针。一部分的常量定义汇编行被禁用，因为它们也已经成功的登记在符号表里。另外一部分的常量定义汇编行仍然存在，这是因为在这些常量定义的表达式中含有未定义的符号，其数值还不能确定。这些未定义的符号可能在后来才被定义。在 Pass1 扫描时未能成功登记的常量并不报错，而留到 Pass2 扫描阶段继续登记。

从 Pass2 扫描开始，宏汇编器要跟踪各段的地址计数器的值，把该值存放到每一汇编行的 lc 域中。并把地址连续的代码归成一片，地址不连续的代码则往该段添加一个新片段。

Pass2 扫描：

如果该汇编行被禁用（AsmLine 中的 Enable 值被设置为 false），则扫描忽略该行。

对该汇编行的 Token 序列进行扫描，

（1） 如果遇到汇编指令（ReservInst Token），则调用 ReservInstDo()模块对汇编指令进行识别。ReservInstDo()模块调用 FindResvInstDo()模块找到该汇编指令对应的寻址方式，得到指向 InstSystem 管理器中相应的指令描述结构体（OPset）的指针。这个指针被保存在 ReservInst Token 的指针域中。通过该指针，可以访问到该条指令的相关信息：如：指令目标码，从操作数到目标码的翻译的动作码和该指令的长度，即占用的字节数。把这个长度保存在汇编行的 len 域中。同时更新汇编行的 lc 值：上一行的偏移地址加上 len 等于本行的偏移地址。最后，把 ReservInst Token 变形为 KnownInst Token。

（2） 如果遇到 ORG 指令（ORG Token），则调用 TknIsORGDo()模块，把它所跟的绝对地址表达式计算出来，根据结果更新当前段的地址计数器（SegPt->SegLc）。在更新当前段的地址计数器之前，还要先保存上一片段的最大地址（用作更新段的大小）。然后在更新段地址计数器之后产生新片段（但不产生新段）。最后禁用该汇编行。ORG Token 的 Value 域保存着表达式的结果。

（3） 如果遇到 DB、DW 指令（DB Token、DW Token），则分别调用 TknIsDBDoSth()和 TknIsDWDoSth()模块。这两个模块计算本汇编行数据块的长度（表达式的计算留在代码生成阶段才做），保存在汇编行的 len 域中，并更新当前段的地址计数器。

（4） 如果遇到 DS、DBIT 指令（DS Token、DBIT Token），则分别调用 TknIsDSDoSth()和 TknIsDBITDoSth()模块，计算其后所跟的表达式，得到结果后把它存放到 Token 的 Value 域中，并更新当前段的地址计数器。最后将其所在的汇编行禁用。

（5） 如果遇到 SET 伪指令（SETsDefken Token），调用 SetVarThenKeepIt()模块，往符号表添加一个 SET 语句决定的变量，该变量的类型由表达式的结果决定（默认为纯数值，即 LB_NUM）。但最后并将不其所在的汇编行禁用。因为需要在扫描过程中跟踪这个变量的变化情况。

（6） 如果遇到上一遍留下来尚未成功登记到符号表格里的常量（EQU 语句、CODE 语句、DATA 语句、XDATA 语句、IDATA 语句和 BIT 语句），则继续登记。但如果这次登记还不成功，则表示发生了错误（通常由未定义的符号引起），通过 OutputErr 服务对外报告错误。

Pass2 扫描的结果，如果没有发生错误的话，各汇编行的 lc 值都已经确定，IsOK 被置为 true。所有的常量定义汇编行都被禁用，ORG 等改变当前段的地址计数器的指令的 Token 也已消失。没有被禁用的汇

编行只剩下：含有 KnownInst Token 的汇编行、含有 DB Token 的汇编行、含有 DW Token 的汇编行和含有 SETsDefkenToken 的汇编行。

经过 Pass1 和 Pass2 两次扫描之后，CreateVarTable()还要完成语义分析阶段最后的工作：根据 SegLc 更新段的大小，保存最后一段的最大地址，关闭最后一个使用的段的片段。对符号表中的所有符号进行一次刷新，填写一些有用的信息。

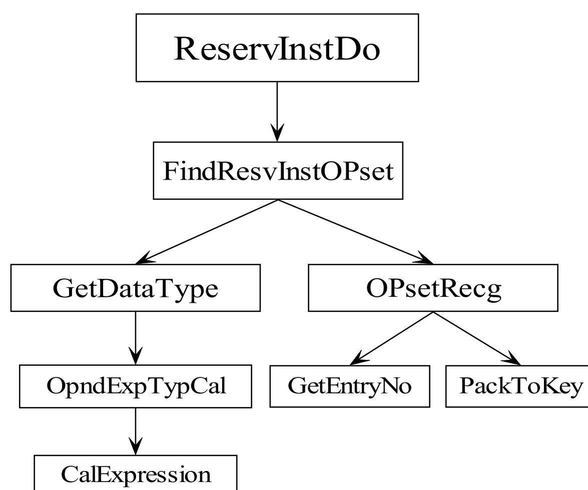
## 宏汇编器的 FindResvInstDo()内部服务

语义分析阶段中，当汇编行的 Token 序列是汇编指令语句时，将调用 ReservInstDo()模块对它进行识别。FindResvInstDo()服务是作为被 ReservInstDo()模块调用的服务。它的功能是分析该条汇编指令的各个操作数的寻址方式，返回指向一个与之相匹配汇编指令描述结构体(OPset)的指针。

首先，它定义一个由 4 个字节组成的数组 OPR。其中 OPR[0]用来存放汇编指令的 Token 码。OPR[1～3]用来存放该汇编指令所跟的最多三个操作数的寻址方式码。（如果没有那么多的操作数，用-1 代替其寻址方式码。）各操作数的寻址方式码是通过调用 GetDataType()内部服务得到的。当 OPR 数组设置好后，FindResvInstDo()调用指令系统管理器类（InstSystem）的 OPsetRecg()服务，进行 OPset 比较，找出相匹配汇编指令描述结构体(OPset)，返回指向它的指针。

GetDataType()内部服务需要调用 OpndExpTypCal()内部服务，来计算操作数中包含的数值表达式。

**ReservInstDo()调用层次图：**




这里有必要介绍一下指令系统管理器类（InstSystem）的 OPsetRecg()服务。它的工作是把汇编指令的 Token 码和三个操作数的寻址方式码（不足三个操作数以－1 代替）组成一个四元组(PackToKey)，以这个四元组为输入，通过特定的 hash 函数 GetEntryNo()映射成一个 8 位的地址，该地址是 OPset 的入口地址。然后通过该地址访问那个 OPset 结构体，比较寻址方式是否与输入的一致，是则返回该 OPset 的地址。否则比较链表的下一个 OPset，直到找到或找不到而报错。

语义分析结束时，所有定义的符号都应该登记在标号管理器对象中的标号表里。大部分的汇编伪指令的功能已完成，因此它们所在的汇编行（AsmLine）已被禁用。所有的汇编指令 Token 都应该从 ReservInst Token 形式进化成 KnownInst Token 形式，其 Pointer 域指向相应的汇编指令描述结构体(OPset)。各 AsmLine 对象的 lc 域都存有它所在的地址，len 域存有它占用的字节数，sgpt 存有指向它所属的段的指针。这些信息都为下一步进行代码生成和 OBJ 文件输出提供了支持。

### 标号管理器的标号登记 AddLabel()服务

标号管理是汇编器里的一个很重要的工作。在我的宏汇编器里，我把它交由专门的管理器——LabelManager 类的对象来完成。这个类对外提供以下几个服务：标号登记 AddLabel()服务、查找一个标号 SearchLabel()服务、计算数值表达式 CalExpression()服务、检查计算结果是否合法 CheckTriVal()服务、查找一个标号并计算它的值 SearchLabelCal()服务，等等。

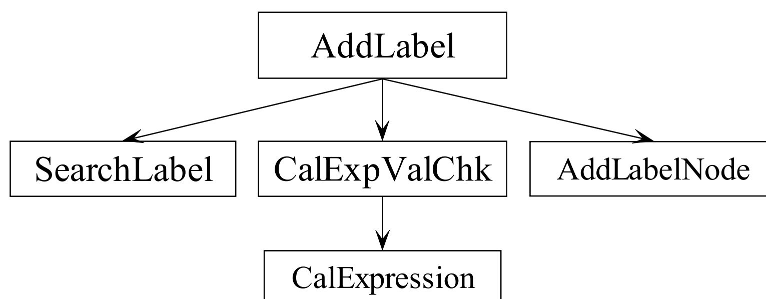
其中标号登记服务 AddLabel()是一个较重要的服务。它的原型如下：

```
JLabelNode* LabelManager::AddLabel( ERR &err, JStrings &name, int8u labelTp, bool var,
                AsmLine* Crln, Tokenfield* pt, ASM_Segment* rSeg, bool Ind, bool Ext )
```

err 是用作带回错误号的 ERR 型变量；name 是标号的名字；labelTp 是标号的类型；布尔量 var 说明标号是否变量；Crln 是创建该标号的汇编行的指针；pt 是标号定义 Token 的指针；rSeg 是标号所属的段的指针；Ind 是标明该标号是否需要间接访问的标志；Ext 是标明该标号是否外部变量的标志。这个函数的返回值是一个指向由标号管理器创建的该标号的描述结点（JLabelNode）的指针。

首先，它检查标号名的长度是否超过 40 字符，是则给出一个警告，并只截取字串的前 40 个字符（因为历史的原因，OBJ 文件记录只能处理 40 字符之内的标号）。然后在标号表格里搜索该名字的标号，以确定标号表中是否已经存在一个同名的标号。如果存在，并且该标号是变量，则重新更新该变量。如果存在而该标号不是变量，则报告标号重复定义错误。如果不存在，则创建一个新的标号结点，即一个 JLabelNode 的对象。然后调用内部服务 CalExpValChk()计算该标号的值并且对结果作正确性检查。计算成功，才往标号表中插入该结点。计算不成功，报告错误，该结点被丢弃。

**标号登记服务 AddLabel()调用层次图：**




### 标号管理器的表达式求值 CalExpression()服务

数值表达式求值是汇编器中的一个极重要的工作。汇编器中的许多模块都要调用这个服务。给出一个符合数值表达式语法的 Token 序列，怎样计算它的值？这项工作由标号管理器的 CalExpression()服务来完成。

要把一个表达式正确的求值，首先要正确解释表达式。例如根据运算符的优先关系，（1）先乘除，后加减；（2）从左到右；（3）先括号内，后括号外。任何一个数值表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的。一般地，操作数既可以是常数（Value），也可以是被说明为变量或常量的标识符（Variable）。运算符可以分为算术运算符、关系运算符和逻辑运算符三类。基本界限符有左右括号和表达式结束符等。我们把运算符和界限符统称为算符，它们构成一个集合，名为 OP。在运算的每一步中，任意两个相继出现的算符 op1 和 op2 之间的优先关系至多是以下三种关系之一：

- op1 ＜ op2 ， 即 op1 的优先权低于 op2。
- op1 ＝ op2 ， 即 op1 的优先权等于 op2。

- op1 ＞ op2 ， 即 op1 的优先权高于 op2。

下表定义了算符之间的这种优先关系：

## 算符优先关系表

| Op1 \ op2 | | N | ＝ | ｜ | & | S | ＋ | * | M | H | ～ | . | （ | ） |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| null | N | ＝ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | err |
| < > = <= >= <> | ＝ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| OR XOR | ｜ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| AND | & | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| SHL SHR | S | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| + - | ＋ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| * / MOD | * | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＜ | ＞ |
| 单目+ 单目- | M | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＜ | ＞ |
| HIGH LOW | H | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＜ | ＞ |
| NOT | ～ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＜ | ＞ |
| . | . | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＜ | ＞ |
| （ | （ | err | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＜ | ＝ |
| ） | ） | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | ＞ | err | ＞ |

首先，它设立了两个栈：运算符栈 OPTR 和操作数栈 OPND。运算符栈 OPTR 是 TokenStack 类的对象，操作数栈 OPND 是 TriValStack 类的对象。TokenStack 类栈的压入和弹出的单位是一个 Token 码，在 OPTR 栈中就是一个运算符的 Token 码；TriValStack 类栈的压入和弹出的单位是一个 TriVal 对象，在 OPND 栈中它就是参加运算的操作数。因为在宏汇编器中的表达式运算不仅包括操作数的数值运算(如 5＋7H＝0CH)，还包括这些操作数的类型运算(如绝对量＋可重定位量＝可重定位量，代码型地址＋纯数值＝代码型地址)，所以参加运算的操作数都是一个个 TriVal 对象：

```
class TriVal
{ public:
   int32 val;              // 数值(32 位)
   union
   { ASM_Segment* rseg;    // 参考的段
     int16u ExtID;
   }; // end union
   int8u typ;              // (class)NUM, CODE, DATA, BIT, ...
   int8u rel;              // 'A', 'R', 'E'
}; // end TriVal
```

表达式求值的算法基本如下：

（1） 置操作数栈和运算符栈为空栈。

（2） 依次读入表达式中的每个 Token，调用全局函数 IsOPTR()判断 token 是否为操作数，是操作数的话则把该 Token 转换成 TriVal 对象（调用 OPNDAct()服务）压进 OPND 栈。若是运算符，则和 OPTR 栈的栈顶运算符比较优先权(调用全局函数 OPsuperior())后作相应操作，直至整个表达式求值完毕。

这相应操作是：

（1） 如果当前运算符比栈顶元素优先权高，则把当前的运算符压进 OPTR 栈。取下一个 Token。

（2） 如果当前运算符与栈顶元素优先权相等，弹出 OPTR 栈顶运算符。取下一个 Token。

（3） 如果当前运算符比栈顶元素优先权低，则当前运算符留到下一轮用。弹出 OPTR 栈顶运算符，根据栈顶运算符弹出 OPND 栈的相应数量的的操作数，拿他们进行运算(调用 OPTRAct()服务)，并把运算结果压入 OPND 栈。

OPNDAct()是宏汇编器的一个内部服务，它的功能是根据指向的 Token 的不同，决定如何给一个 TriVal 对象赋值。

如果是纯数值（ValueToken），Trival 对象的 val 就填上该数值，typ 域填为 LB_NUM，rel 域就填为‘A’（Absolute），rseg 域填为 NULL，表示它不属于任何段。

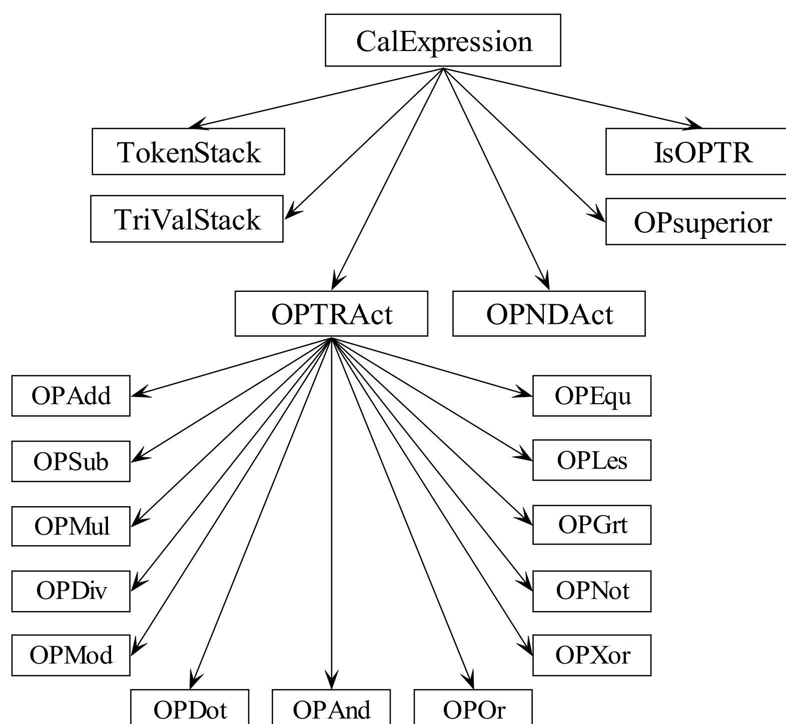
如果是寄存器 Token（RegisterKn），Trival 对象的 val 域填为寄存器的 Token 码，typ 域填为 LB_REG，rel 域填为‘A’（Absolute），rseg 域填为 NULL。

如果是标号或变量 Token（WordSToken），则要先调用 SearchLabelCal()服务，查找并计算该标号的值，然后填写 TriVal 的 val 域。typ 域与标号的 LBLTyp 属性一致，rel 域与标号的 Attrb 属性一致，rseg 域与标号的 RefSeg 属性一致。如果是外部变量，rseg 域存放的是外部变量的 ID。

如果是‘$’，则把当前汇编行的地址填进 TriVal 的 val 域。rseg 域当前汇编行的 sgpt 属性一致，typ 域和 rel 域与当前汇编行所属段的类型一致。

OPTRAct()也是宏汇编器的一个内部服务。它的功能是根据一个给定的运算符，和一个 TriValStack 栈，决定栈中的操作数进行何种运算。它通过函数指针分别调用不同的处理模块来进行操作数的运算。

**标号管理器的表达式求值服务调用层次图：**




## 宏汇编器代码生成服务 FileToObjCode()

在词法分析、语法分析和语义分析之后，如果没有遇到任何错误，就会进入代码生成阶段。FileToObjCode()服务从一开始就创建一个 ERR 类型的变量 erra，并赋初值为 OK_no_Err。然后调用内部服务 LineToObjCode()对全文件的 Token 流进行扫描。如果其中某一汇编行发生了错误，erra 就会被赋值为 Have_Errs。最后在 FileToObjCode()服务结束的时候，它返回 erra 的值。

宏汇编器的内部服务 LineToObjCode()的功能是对一个 AsmLine 对象做代码生成工作。它从汇编行 Token 链表的第一个 Token 开始，

判断该 Token：

（1） 是否为已识别的指令（KnownInst Token）。若是，调用内部服务 Translate()完成从 KnownInst Token 到 InstCode Token 转换。到时，转换后的目标代码将会以目标代码组类 Dat4ary 的对

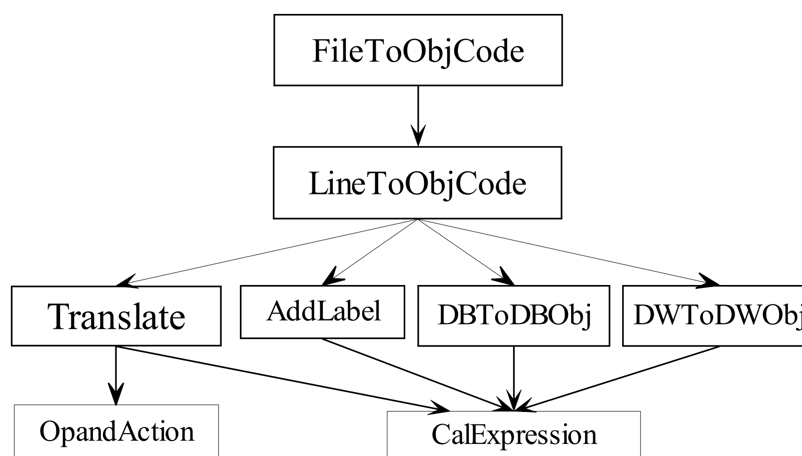
象的形式存放，其指针在该 Token 的 Pointer 域中。

（2） 是否为 SET 指令（SETsDefken Token）。若是，调用标号管理器的 AddLabel()服务，更新变量的值。在这里，SET 指令是最后一次被使用，然后该汇编行被禁用。

（3） 是否为 DB 指令（DB Token）。若是，调用内部服务 DBToDBObj()，把其后的表达式表转换成目标代码。目标代码将会以目标代码组类 Dat4ary 的对象的链表形式存放。DB Token 会进化成 DBObjCode Token，其 Pointer 域存放着 Dat4ary 对象的指针。

（4） 是否为 DW 指令（DW Token）。若是，调用内部服务 DWToDWObj()，把其后的表达式表转换成目标代码。目标代码将会以目标代码组类 Dat4ary 的对象的链表形式存放。DW Token 会进化成 DWObjCode Token，其 Pointer 域存放着 Dat4ary 对象的指针。

FileToObjCode()模块调用层次图：




## 关于 Dat4ary 类（目标代码组类）

目标代码生成后都以 Dat4ary 类的对象的形式存放在内存里。该类的定义如下：

```
class Dat4ary           // 该结构用于 OBJcode
{ public:
   int8u  dat[4];       //  dat[0~2]存数据，dat[3]存长度。
  OpndFixRec* Fix;      //  指向 OpndFixRec 的指针。初始值为 NULL。
}; // Dat4ary
```

有些目标代码是在汇编器中不能确定的，例如那些可重定位的地址。这留给连接器根据不同的情况对它们进行修正（Fixup）。一共有八种情况：

（1） LOW：截取某个由连接器连接好的 16 位目标代码的低字节，存入指定位置。

（2） BYTE：把某个由连接器连接好的 16 位目标代码的类型由字转换成字节，存入指定位置。

（3） RELATIVE：求出某个由连接器连接好的 16 位目标地址的相对位移，存入指定位置。

（4） HIGH：截取某个由连接器连接好的 16 位目标代码的高字节，存入指定位置。

（5） WORD：把某个由连接器连接好的 16 位目标地址的高字节存入指定位置，其低字节存入指定位置＋1 的位置。

（6） INBLOCK：把某个由连接器连接好的 16 位目标地址低三位存入指定位置的高三位，其低字节存入指定位置＋1 的位置。

（7） BIT：把某个由连接器连接好的 16 位目标地址的低字节（应该是个位地址）存入指定位置。

（8） CONV：位地址转换成字节地址，存入指定位置。

这就需要为目标代码设立一些标记，用以辨别这些代码的修正类型（如果这些代码需要修正的话）。于是创立 OpndFixRec 类。如果某个目标代码类 Dat4ary 的对象内的目标代码时需要在连接时进行修正的话，那么它的 Fix 指针就不是 NULL，而是指向一个新创建的 OpndFixRec 类的对象。

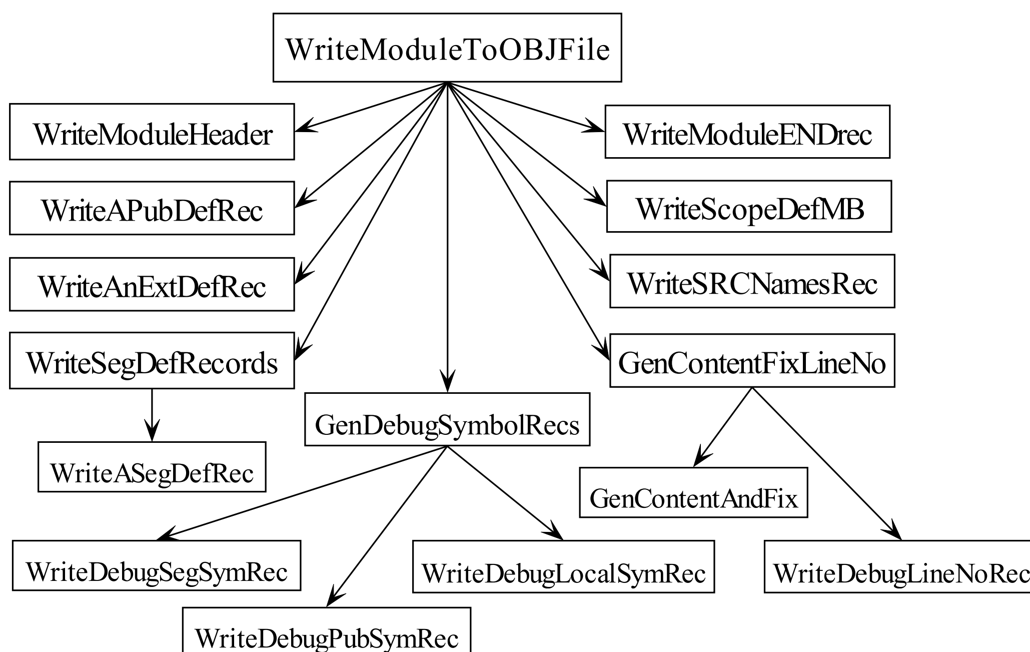
OpndFixRec 类的定义如下：

```
class OpndFixRec
{ public:
   int16u RefLoc;       // 指定的位置
    int8u RefTyp;       // 0 ~ 7 (RefTyp_LOW, RefTyp_BYTE, ... RefTyp_CONV)
    int8u rel;          // IDBLK_SEG_OPND, IDBLK_REL_OPND, IDBLK_EXT_OPND
   int16u Offset;       // Value
   int16u RefID;        // 修正的类型号
}; // end OpndFixRec
```

这个对象内的数据成员的内容均由 Dat4ary 类提供的服务填写。具体填写的内容与 OBJ 文件记录的协议有关。在这里不详述。

经过代码生成后的目标代码存放在内存里，还需要按照 OBJ 文件的格式把他们以记录的形式写到磁盘文件上。宏汇编器提供了这个服务：GenOBJFile()服务。而这个服务其实通过调用目标模块管理器的 WriteModuleToOBJFile()服务来实现的。另外，宏汇编器还提供 GenLISTFile()服务，它负责往磁盘写入 LST 列表文件。

**目标模块管理器 WriteModuleToOBJFile()服务调用层次图：**




整个汇编流程至此，如果都没有发生错误，则汇编成功。

## 关于 OBJ 文件的格式

一个 OBJ 文件由一系列记录组成。它们必须满足下列语法：
（注：<项目>* 表示<项目>可以有 0 个或 0 个以上。）

<OBJ 目标文件> ::= <目标模块>* | <库>

<目标模块> ::= 模块头记录
<定义记录>*
<数据/调试 记录>*

模块尾记录

〈定义记录〉 ::= 段定义记录
| 公共符号定义记录
| 外部符号定义记录

〈数据/调试 记录〉 ::= 〈数据项〉 | 〈调试记录〉
〈调试记录〉 ::= 域定义记录 | 调试项目记录
〈数据项〉 ::= 内容记录 | 修正记录*

〈库〉 ::= 库头记录
<目标模块>*
模块名记录
模块位置记录
库字典记录

记录格式：

每一条记录都由记录类型号，记录长度，记录内容，校验和这四项组成。

| 记录类型号 XXH | 记录长度 | 记录内容 | 校验和 |
|---|---|---|---|

各记录类型如下：

（1） 模块头记录（02H）[Module Header Record]
（2） 模块尾记录（04H）[Module End Record]
（3） 段定义记录（0EH）[Segment Definitions Record]
（4） 公共变量声明记录（16H）[Public Definitions Record]
（5） 外部变量声明记录（18H）[External Definitions Record]
（6） 域定义记录（10H）[Scope Definition Record]
（7） 调试项记录（12H）[Debug Items Record]
（8） 内容项记录（06H）[Content Record]
（9） 修正项记录（08H）[Fixup Record]
（10） 库头记录（2CH）[Library Header Record]
（11） 模块名记录（28H）[Library Module Names Record]
（12） 模块位置记录（26H）[Library Module Locations Record]
（13） 库字典记录（2AH）[Library Dictionary Rcord]

更详细的说明请参阅： omf51.doc。

## 关于宏处理

ZLG51 宏汇编器能够对宏进行识别并处理。宏处理分为宏定义和宏调用。宏定义是将一组经常使用的汇编指令定义成一个宏名，或者是具有完成一个功能能力的汇编指令的一个组合的名称。宏调用则是在程序中调用一个已定义过的宏名并将它扩展成对应的一组汇编指令（包括汇编控制指令）。宏与子程序是两个不同的概念。宏调用是由于要将宏名替换成一组指令，因此会使整个程序代码加长，但可以节省子程序调用

的时间开销从而加快程序的执行速度。子程序调用不会增加程序代码，但由于需要使用 CALL 指令，其中压栈和退栈增加了额外的执行时间，降低了程序的速度。一般而言，对于程序中需要经常调用的一些过程，在要求程序所占用的系统存储器空间尽量小的时候，采用子程序调用合适。当需要最大程度地发挥处理器的速度，同时系统存储器空间足够时，则可采用宏。

写汇编程序的时候，宏提供了大量的优点：可以用于使重复的短的汇编模块更方便进入宏；经常使用宏可以减少程序员引起的错误；宏非常适合创建简单的代码表，如果手工做这些表会非常乏味而且容易出错；在宏中使用的符号的有效范围只在宏内（LOCAL），你不需要关心有关初始化前面使用过的符号名的问题。

在程序中使用宏之前应先进行宏定义。与宏定义有关的指令是：MACRO、ENDM、LOCAL、REPT、IRP、IRPC 和 EXITM。

| 伪指令 | 描　　述 |
|---|---|
| MACRO | 开始定义宏而且指定了宏的名字和任何要传递到宏的参数。 |
| ENDM | 宏定义结束。 |
| LOCAL | 指定在宏中使用的局部符号。 |
| REPT | 为宏中接下来的行指定重复因子。 |
| IRP | 用指定的变量列表替代指定的参数，一次替代一个参数。 |
| IRPC | 用指定的变量代替<>表中指定的参数，一次只能一个字符。 |
| EXITM | 使宏展开立即中止。 |

宏定义用 MACRO 伪指令开始，它声明了宏的名字以及形式参数。宏定义必须用 ENDM 伪指令结束。在 MACRO 和 ENDM 伪指令之间的文字就叫宏体。

格式：　　宏名　MACRO　[形式参数表]

举例：

```
WAIT        MACRO      X                     ; macro definition
                REPT   X                     ; generate X NOP instructions
                NOP
                ENDM                         ; end REPT
            ENDM                             ; end MACRO
```

在这个例子中，WAIT 是宏的名字，而 X 是唯一的形式参数。

除了 ENDM 伪指令外，EXITM 伪指令也可以用于立即中止宏展开。当检测到 EXITM 伪指令后，宏处理器停止展开当前的宏，直到下一个 ENDM 伪指令后再继续处理。EXITM 伪指令在条件语句中很有用。

举例：

```
WAIT        MACRO      X                     ; macro definition
                IF NUL X                     ; make sure X has a value
                EXITM                        ; if not then exit
                ENDIF

                REPT   X                     ; generate X NOP instructions
                NOP
                ENDM                         ; end REPT
            ENDM                             ; end MACRO
```

在调用行，ZLG51 可以传递超过 16 个实际参数到宏体里面(Keil 的 A51 最多只能 16 个)，具体数目只受汇编行的长度限制，即定义汇编行长度不能超过 255 字符。形式参数的名字必须用 MACRO 伪指令定义。

举例

```
MNAME  MACRO  P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16
```

上例定义了一个有 16 个参数的宏。在宏定义和调用中，参数必须都用逗号分隔。这个宏的调用行如下所示：

MNAME A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P

其中的 A,B,C,…O,P 是对应于形式参数名 P1,P2,P3,…P15,P16 的参数。

空参数也能传递到宏。空参数的值是 NULL，可以用于测试后面描述的 NUL 运算符。如果要在调用宏的时候忽略参数列表中的一个参数，这个参数可被赋值为 NULL。

举例：

MNAME A,,C,,E,,G,,I,,K,,M,,O,

这样，P2、P4、P6、P8、P10、P12、P14 和 P16 在调用宏的时候会被赋值为 NULL。

在宏中的标号应当是局部标号。局部标号只在宏中可视，此时宏在一个源文件中多次使用也不会出现标号重复定义的错误。您可以在宏中用 LOCAL 伪指令定义一个局部标号（或任何局部变量）。ZLG51 宏汇编器可以定义超过 16 个局部标号(Keil 的 A51 最多只能定义 16 个局部符号)。

注意，LOCAL 必须紧接在 MACRO 定义的下一行。

举例：

```
CLRMEM          MACRO       ADDR, LEN
                LOCAL       LOOP
                MOV         R7, #LEN
                MOV         R0, #ADDR
                MOV         A, #0
LOOP:           MOV         @R0, A
                INC         R0
                DJNZ        R7, LOOP
                ENDM
```

在这个例子中，标号 LOOP 由于使用了 LOCAL 伪指令定义，所以是局部的。任何不是用 LOCAL 伪指令定义的符号都是一个全局符号。

ZLG51 为在宏中定义的局部符号产生了一个内部符号。内部符号的形式是??0000，每次调用宏的时候都会加 1。因此，宏中使用的局部符号是唯一的，而且不会产生错误。

ZLG51 提供了在宏中重复一个文本块的能力。REPT、IRP 和 IRPC 伪指令用于指定在宏中重复的文字。这些伪指令都要用 ENDM 伪指令结束。

REPT 伪指令以固定的次数重复文字块。下面的宏：

```
DELAY           MACRO                       ;macro definition
                  REPT     5                ;insert 5 NOP instructions
                  NOP
                  ENDM                      ;end REPT block
                ENDM                        ;end macro definition
```

当它被调用时插入 5 个 NOP 指令。

IRP 伪指令为指定列表的每个参数重复块一次。在文本块中指定的参数被各个变量代替。下面的宏：

```
CLRREGS         MACRO                                 ; macro definition
                  IRP   RNUM, <R0,R1,R2,R3,R4,R5,R6,R7>
                        MOV     RNUM, #0
                  ENDM  ; end IRP
                ENDM                                  ; end MACRO
```

用 R0、R1、R2、…R7 代替参数 RNUM。

当调用 CLRREGS 时，产生如下代码：

```
MOV       R0, #0
```

```
MOV        R1, #0
MOV        R2, #0
MOV        R3, #0
MOV        R4, #0
MOV        R5, #0
MOV        R6, #0
MOV        R7, #0
```

IRPC 伪指令为指定变量的每个字符重复块一次。文本块中指定的参数被每个字符代替。下面的宏

```
DEBUGOUT    MACRO                              ; macro definition
                 IRPC      CHR, <TEST>
                 JNB       TI, $               ; wait for xmitter
                 CLR       TI
                 MOV       A,#CHR
                 MOV       SBUF,A              ; xmit CHR
                 ENDM                          ; end IRPC
            ENDM                               ; end MACRO
```

用字符 T、E、S、T 代替参数 CHR。

宏定义的嵌套可深达 64 级(Keil 的 A51 只能达到 10 级)。

举例：

```
L1             MACRO
               LOCAL          L2
               L2             MACRO
                              INC R0
                              ENDM
               MOV            R0, #0
               L2
               ENDM
```

宏 L2 在宏定义 L1 中定义。由于 LOCAL 伪指令将 L2 定义为局部符号，因此它在 L1 外不能看到。如果想在 L1 外使用 L2，从 LOCAL 伪指令符号列表中排除 L2 就可以了。

调用 L1 宏产生如下代码：

```
MOV     R0, #0
INC     R0
```

你也可以用 REPT、IRP 和 IRPC 伪指令指定嵌套重复的模块。

举例：

```
PORTOUT        MACRO                           ; macro definition
                 IRPC      CHR, <Hello>
                   REPT    4                   ; wait for 4 cycles
                   NOP
                   ENDM                        ; end REPT
                 MOV       A,#'CHR'
                 MOV       P0,A                ; write CHR to P0
                 ENDM                          ; end IRPC
               ENDM                            ; end MACRO
```

这个宏在 IRPC 模块嵌套 REPT 模块。

宏可以直接或间接（通过其它宏）调用自己。总的递归级数量可达 60 级(Keil 的 A51 只达 9 级)。下面的例子是关于一个递归宏被一个非递归宏调用。

```
RECURSE              MACRO          X                    ; recursive macro
IF X<>0
                     RECURSE        %X - 1
```

```
                    ADD         A,#X            ; gen add a,#?
ENDIF
                    ENDM

SUMM                MACRO       X               ; macro to sum numbers
                    MOV         A,#0            ; start with zero
IF NUL X                                        ; exit if null argument
                    EXITM
ENDIF
IF X=0                                          ; exit if 0 argument
                    EXITM
ENDIF
                    RECURSE X                   ; sum to 0
                    ENDM
```

ZLG51 提供了可以在宏定义中使用的运算符。下表对每个运算符作了描述。

| 运算符 | 描　　述 |
|---|---|
| NUL | NUL 运算符可以用于确定一个宏变量是否是 NULL。如果变量是 NULL，NUL 将产生一个非零的值。如果变量不是 NULL，NUL 则产生一个 0。NUL 运算符可以和 IF 控制一起使用，使能条件宏汇编。 |
| & | 这个字符用于连接文字和参数。 |
| <> | 尖括号用于按字面解释定界符譬如逗号和空格。当要将这些字符传递到嵌套的宏时，就需要尖括号。每个嵌套级别需要一对尖括号。 |
| % | 百分号是应被解释为表达式的宏参数的前缀。当使用这个运算符时，后面的表达式数字值被算出。这个值被传递到宏，而不是表达式文字。 |
| ;; | 在 Keil 的 A51 里面双分号表示同一行剩下的文字应被忽略，剩下的文字不会被处理或执行，这有助于减少使用存储器。但是在 ZLG51 里面只需用一个分号就够了。 |
| ! | 如果字符前面有感叹号，表示字符将被按字面解释。这就允许将字符运算符作为参数传递到运算符。 |

当在调用宏的时候忽略了形式参数，参数的值将是 NULL 的。你可以在宏中使用 NUL 运算符和 IF 控制来检查 NULL 参数。NUL 运算符要求一个参数。如果没有找到参数，NUL 将返回 0 到 IF 控制。

例如，下面的宏定义：

```
EXAMPLE         MACRO     X
  IF  NUL  X
    EXITM
  ENDIF
ENDM
```

被如下调用时：

```
EXAMPLE
```

将通过 IF NUL X 的检验，处理 EXITM 语句，然后退出宏扩展。

注意空格符(‘’)的 ASCII 值是 20h，并不等于 NULL。

宏运算符（&）可用于连接文本和宏参数。下面的宏声明显示了这个运算符的正确用法。

```
MAK_NOP_LABEL           MACRO     X
LABEL&X:                NOP
                        ENDM
```

MAK_NOP_LABEL 宏将在每次调用插入一个新的标号和一个 NOP 指令。参数将附加到文本 LABEL 的后面组成该行的标号。

尖括号（<>）用于包含应被按照字面意义传递给宏的文字。一些字符（例如逗号）如果没有包含在尖括号中不能传递给宏。

感叹号(!)运算符用于表示要被作为文本传输到宏的特殊字符。这个运算符使你能够传递通常被解释为定界符的逗号和尖括号符号到宏。

当定义了一个宏后，它能在程序中被多次调用。宏调用由宏名字加上要传递到宏的参数组成。

在调用宏的时候，实际参数的位置与在宏定义中指定的参数名位置相对应。调用中传递的第一个参数每次都代替了宏定义中第一个形式参数，传递的第二个参数代替了宏定义中第二个形式参数，如此类推。

如果在宏调用中指定的参数比在宏定义的多，ZLG51 将忽略额外的参数。如果指定的参数比定义的少，ZLG51 就会用 NULL 字符赋值给缺少传递值的参数。

## 关于宏管理器 MacroMger

在 ZLG51 里面，大部分对宏的识别和处理的工作是由一个 MacrosManager 类的对象 MacroMger 来承担的。该类内含一个宏定义注册表 MRList。MacrosManager 类提供如下服务：

1. MacroRegister(); 宏定义注册。输入一个汇编行，该汇编行含有一个 MacroDef 和一些形参。
2. MacroDefParser(); 对 MacroDef 后的形参表进行分析。
3. IsMacroAndParse(); 对遇到的 Word 看看是否为宏，是则进行宏调用的语法分析。
4. MacroLineBacktoJstr(); 把宏定义体中的某一行拿出来，把其中的形参用实参替代。
5. MacroToToken(); 宏展开词法分析器。
6. MacroCall(); 宏调用处理。处理宏替代。
7. MacroREPTdo(); REPT 宏重复处理。
8. REPT_Parser(ERR &err, AsmLine* Ln); REPT 语法分析器。
9. TryLocalParse(); 寻找 LOCAL，并进行语法分析。
10. IRP_Parse(); IRP 的语法分析。
11. IRPC_Parse(); IRPC 的语法分析。
12. MacroIRP_do(); 遇到 IRP 指令的宏展开。
13. MacroIRPC_do(AsmLine* &Ln); 遇到 IRPC 指令的宏展开。
14. FetchAParaWord(); 从 str 中截取一个单词, 放入 para->Name 中。str 带值返回。
15. FetchAParaLetter(JStrings &str, Tokenfield ¶); 从 str 中截取一个字母, 放入 para->Name 中。str 带值返回。
16. AssaignLocalLabelNo(); 分派局参号码。
17. LocalArgvToJstr(); 局参名字的拼装：（"??xxxx"）。

## 关于 MacroDefBody 宏定义体类

一个宏定义体是指一个记录由用户使用 MACRO……ENDM 来定义的宏的描述信息的对象。内容包括宏名、宏体开头指针、宏体结尾指针、首个形参指针、首个局参指针等信息。

宏体是指宏定义中除去 MACRO、LOCAL 和 ENDM 汇编行后余下的那部分汇编行。

例如：

```
MEMCLR  MACRO  g1, g2, g3        ; 宏定义
LOCAL   loop                     ; 宏体内局部符号的定义
        MOV R0, #g1             ;          宏体开头
        MOV  A, #g2             ;    }
        MOV R7, #g3             ;    }
loop:   MOV @R0, A              ;    }  宏体
        INC  R0                 ;    }
        DJNZ R7, loop           ;    }
        ENDM                    ;          宏体结尾
```

**宏定义体对象的结构：**

| | | |
|---|---|---|
| Jstring | MacroName | 宏名 |
| AsmLine* | MacroBegin | 指向宏体开头 |
| AsmLine* | MacroEnd | 指向相应的 ENDM |
| Tokenfield* | arguPtr | 指向宏体的首个形参 |
| Tokenfield* | localPtr | 指向宏中的首个局参 |
| MacroDefBody* | next | 指向下一个结点 |

该类提供以下几个服务：

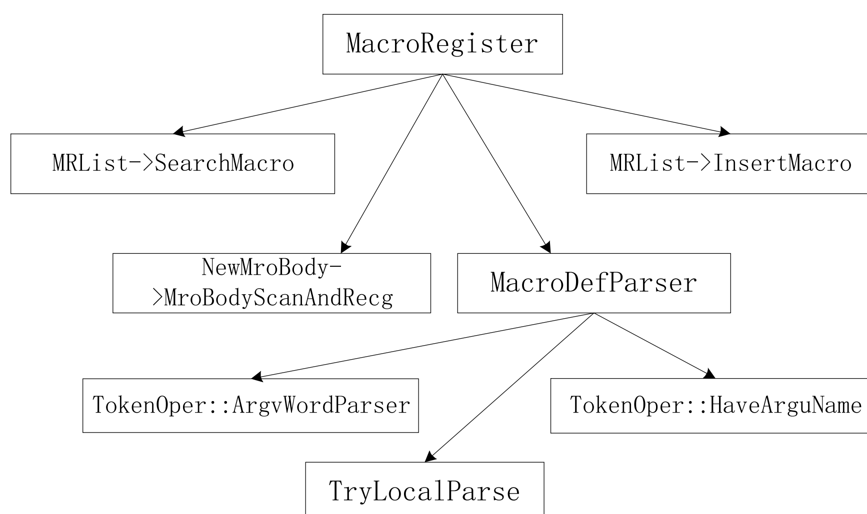
1. bool IsBodyEmpty()const；判断宏体是否为空。如果 MacroBegin 与 MacroEnd 相等，则认为宏体为空。
2. Tokenfield* SearchFormArgv(const Jstring& fname)const; 寻找形参。判别输入的字串是否为一个形参的名字。若是，返回指向该形参的 Token 的指针。否则返回 NULL。
3. Tokenfield* SearchLocalArg(const JStrings& fname)const; 寻找局参。判别输入的字串是否为一个局参的名字。若是，返回指向该局参的 Token 的指针。否则返回 NULL。
4. void SearchAndRecgArgvInMro(AsmLine* CLine); 对其中出现的 WordSToken 作判别，如果是形参，改成 FormalArgv。如果是局参，改成 LocalArgv。把 CharDatan, LongCharn, StrDataKn 这些 Token 都改成 WordSToken。
5. void MroBodyScanAndRecg(AsmLine* &CLine); 设置宏头和宏尾。扫描宏体，识别其中的形参和局参，改成 FormalArgv 和 LocalArgv。CLine 带值返回，返回时指向 ENDM。

如果一个宏定义体对象被创建并插入 MRList 表中，就表示该宏定义已被注册。只有被注册的宏才能供汇编程序以后的调用。MRList 是 MacroRegList 宏定义表类的对象，它实际上是一个单链表。MacroRegList 类提供宏定义节点的插入和查找等服务。

## ZLG51 对宏定义指令的识别和处理

宏定义的识别是在语法分析阶段进行的。当宏汇编器在语法分析阶段遇到一个 MacroDefkn 的 Token 时，它把该 Token 所在的汇编行作为输入，调用宏管理器 MacrosManager 的 MacroRegister()服务。该服务的功能是给新定义的宏注册。作为输入参数的汇编行应该含有一个 MacroDef Token 和一些形参。如果该汇编行的第一个 Token 不是 MacroDef,汇编器报错。是则把该汇编行的 Enable 域置为 false。然后根据 MacroDef Token 的 Name 域所指示的宏名查找宏定义注册表，看看该宏名是否已经在此之前已经被登记过。经查，若发现重名，则说明存在重复宏定义，宏汇编器报错。如果没有发现重名，则创建新的宏定义体 MacroDefBody 的对象。然后，对 MacroDef Token 后面的形参 Token 进行语法检查。这些形参必须全部是 WordSToken，而且不能有重名。检查完形参 Token 后，再看下一汇编行是否存在 LOCAL 伪指令。若有，则也对其后的局部标号进行词法检查。这些局部标号不能是数值、保留字、前面紧跟 MacroDef Token 之后已经定义的形参名或变量名，也不能是当前宏定义的宏名。这些都是通过调用 MacroDefParser()服务完成的。调用它之后，如果该宏定义没有语法错误，则继续扫描宏体，识别宏体中的形参和局参，相应有 Token 的进化。形参将改成 FormalArgv Token，局参将改成 LocalArgv Token。找出宏头宏尾所在的汇编行行号。如果一切顺利，新的宏定义节点将插入宏定义注册表 MRList 中。最后 MacroRegister()成功返回 OK_no_Err 值。

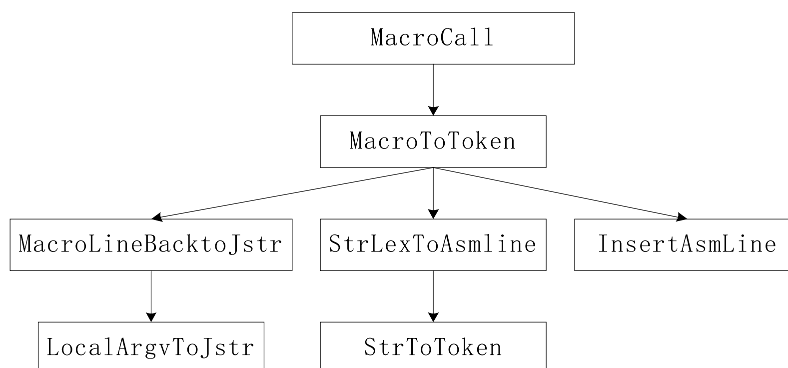
**宏管理器 MacroRegister()服务调用层次图：**




## ZLG51 对宏调用的识别和处理

ZLG51 对宏调用的识别和处理也是在语法分析阶段中进行。在 AsmLineParse()程序中，如果遇到了一个 WordSToken，说明该 Token 不是保留字，也不是伪指令，于是查看 MACRO 模式是否被启用（宏汇编器 masm 的 Dismacro 开关是否为 false 值），如果宏调用没有被禁用的话，则调用宏管理器的 IsMacroAndParse()服务搜寻当前已经登记了的宏定义，看宏定义表（MRList）是否有以该 WordSToken 的 Name 域中的字串作为宏名的宏定义。若有，则继续对其进行语法分析。通过语法分析之后没有发现错误的话，就调用宏管理器的 MacroCall()服务对其进行宏展开。

**宏管理器 MacroCall()服务调用层次图：**




宏管理器的 IsMacroAndParse()服务以一个指向 WordSToken Token 的指针作为输入。该服务首先搜索 MRList 表，找到该名字的宏，得到指向该宏的定义体 MacroDefBody 的指针。然后看宏调用有没有实参传递，有则把它后面所带的实参进行语法分析，并依顺序往宏体传递实参。如果实参个数少于宏的形参个数，则余下的形参赋 NULL。如果实参个数多于宏的形参个数，多出的实参被忽略，且发出一个 Warning 信息。IsMacroAndParse()最后返回一个布尔值，true 表示输入的 WordSToken Token 是一个宏，false 表示它不是一个宏，而是一个未定义的符号。

宏管理器的 MacroCall()服务给宏体初始化好形参和局参，使宏体内部每一个使用形参和局参的 Token 的 ExpPt 域指向实参 Token，然后送宏展开器 MacroToToken()。宏完全展开后，该宏调用汇编行被禁用。

宏管理器的 MacroToToken()服务既是宏展开器又是宏体的词法分析器。它首先判断宏嵌套是否超过最高嵌套层数，是则退出并报错。否则用 CLn 记录当前宏调用的汇编行，宏将在 CLn 和它的下一行之间展开。然后进入循环，每次从宏体中取出一行，调用 MacroLineBacktoJstr()服务把该行转换成字串对象 s，创建新汇编行 AsmLine 对象，给新汇编行赋行号值，给新汇编行的 NestNo 域赋嵌套层数，再给它的头(Ln->head)的 Name 域赋上先前得到字串对象 s。这样该汇编行的头节点上就有展开的宏字串。然后调用宏汇编器 masm 的词法分析服务 StrLexToAsmline()，把该行转换成 Token 流。如果转换成功，且转换后的新汇编行对象

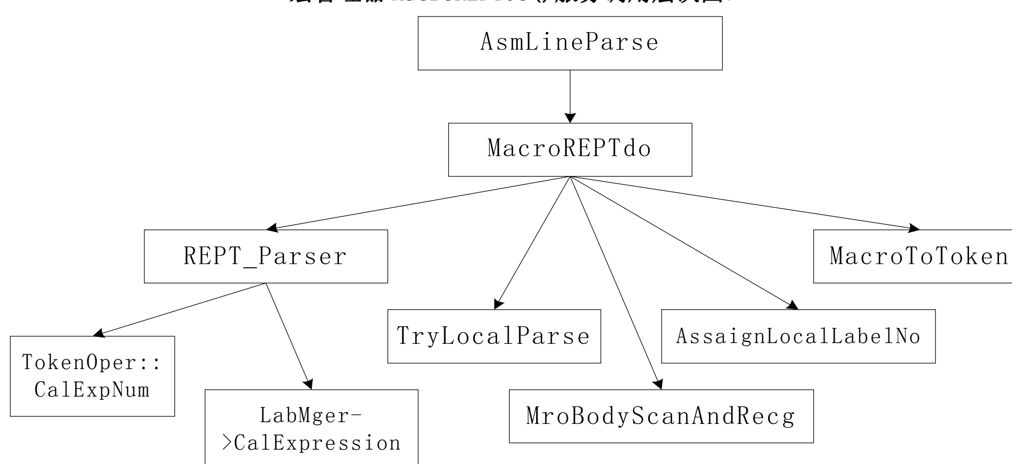
不是空行的话，就紧跟在 CLn 后插入（调用 InsertAsmLine()），然后更新 CLn 指向新汇编行。如果转换不成功或者转换后的新汇编行对象是空行的话，该汇编行对象销毁，不添加到汇编文件中。直到把整个宏体展开后，循环结束并返回。

## 关于 REPT 宏指令的识别和处理

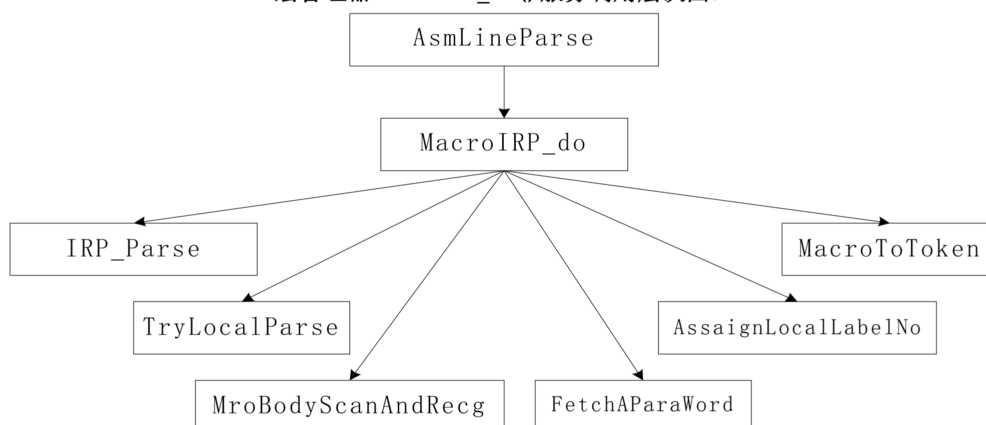
REPT 宏指令的识别和处理也都是在语法分析阶段中进行的。宏汇编器 masm 的服务 AsmLineParse()已检测到 REPTTkn Token，就调用宏管理器 MacroMge 的服务 MacroREPTdo()进行 REPT 处理。而 MacroREPTdo()一开始首先调用宏管理器 MacroMger 的服务 REPT_Parser()对其所在汇编行进行语法扫描。如果语法通过，REPT_Parser()返回一个 REPT 后面所带的数值表达式的值，该值就是要重复扩展宏体的次数。接着，MacroREPTdo()创建一个无宏名的宏定义体，并调用 TryLocalParse()扫描宏体，寻找 LOCAL 语句。然后调用MroBodyScanAndRecg()识别其中局参,把它们改成LocalArgv Token。最后,调用AssaignLocalLabelNo()给局参赋值，并调用 MacroToToken()，按重复值把他们拷贝到 REPT 的 ENDM 后， 并进行词法分析。

关于 IRP 宏指令、IRPC 宏指令的识别和处理也是和 REPT 宏指令的类似的。

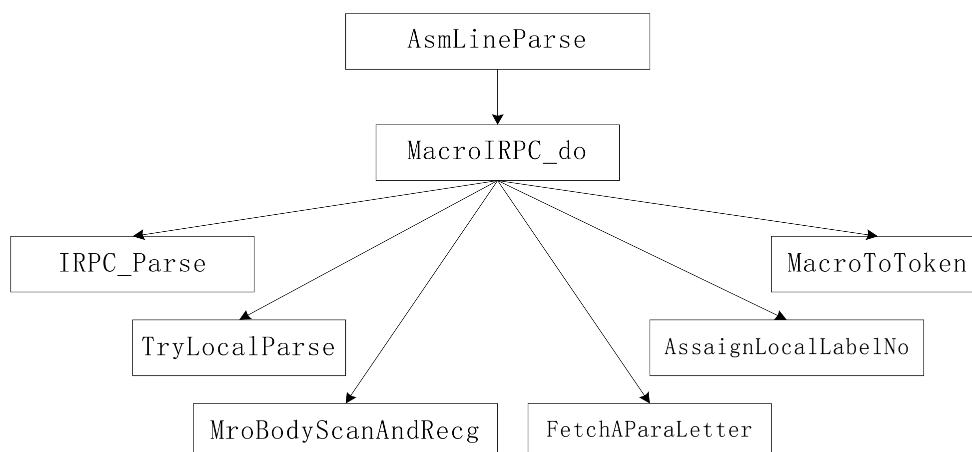
**宏管理器 MacroREPTdo()服务调用层次图：**




**宏管理器 MacroIRP_do()服务调用层次图：**




**宏管理器 MacroIRPC_do()服务调用层次图：**




## 关于‘$’行汇编控制指令的识别

ZLG51 宏汇编器能够识别两类汇编控制指令，首要控制指令和次要控制指令。首要控制指令只允许在程序中出现一次，并且它们不能被其它控制指令所改变。首要控制指令可以在 DOS 命令行上调用 ZLG51 宏汇编器时引用，也可以在汇编源程序的开始处引用。次要控制指令则能在源程序的任何地方引用，并且可以引用多次。对于在 DOS 命令行引用的汇编控制指令，引用格式如下以 sample.asm 为例）：

ZLG51 sample.asm DB NOMOD51 DATE(2002,12,31) EP

如果是在汇编源程序中引用汇编控制指令，运用行必需以符号‘$’开头。对于首要控制指令，最好将它们放在源程序的开始处。例如上面的汇编控制指令可按如下方式放在源程序的开头：

$ DB

$ NOMOD51

$ DATE(2002,12,31)

$ EP

也可以将它们放在一行：

$ DB NOMOD51 DATE(2002,12,31) EP

这时在 DOS 行上调用 ZLG51 宏汇编器的命令为：

ZLG51 sample.asm

关于宏汇编器的所有汇编控制指令的使用方法，在这里不再详述，用法与 Keil A51 的使用方法一样，请参考 A51 宏汇编器的使用说明。

汇编控制指令的识别是在每次对一个汇编行完成词法分析之后进行的。ZLG51 宏汇编器判断该汇编行是否以‘$’开头，是则调用 MajorCtrlParse()服务对其中的控制命令进行识别。该服务最后返回错误值，该值为零表示没有发生任何错误。

MajorCtrlParse()是 MacroAsmber 对象的服务，是首要汇编控制指令语法分析器。首要控制指令如果带有参数，其参数就在首要汇编控制指令的 Token 对象的 Name 域中。该服务根据输入的汇编行 Token 流依次对 Token 进行识别，然后根据各控制指令各自的语义转向不同的操作。但是在转向不同的操作之前，都先要判断一下该首要控制指令是否出现多于一次，是则报告错误。如果不发生错误的话，调用完 MajorCtrlParse()服务之后，该行中的首要控制指令的语义被执行，然后首要控制指令会被剔除。

而对于次要汇编控制指令的识别则是在汇编行的语法分析阶段调用 MacroAsmber 对象的 DogCtrlParse()服务来进行的。在汇编行语法分析程序中，首先判断汇编行是否以‘$’开头，是则调用 DogCtrlParse()这个次要汇编控制指令语法分析器。这时，该汇编行中不会有首要控制指令遗留（因为它们已经被剔除），余下的应该是可以多次引用的次要控制指令。该服务根据输入的汇编行 Token 流依次对 Token 进行识别，然后根据各控制指令各自的语义转向不同的操作。完成后，此服务返回一个错误值，该

值为零表示没有发生任何错误。次要控制指令不被剔除，仍将留在汇编行中。

### 关于宏变量表 MvList

在条件汇编指令里面，通常要访问一些由$SET 和$RESET 指令定义的符号。这些符号不同于用 EQU 等伪指令定义出来的符号，它们只能被$IF 和$ELSEIF 访问。这些变量还可以从外部调用的命令行参数引入。MvList 表就是存放这些符号的链表。它是 MacroVarList 类的对象。表中的元素是一个个 MacroVarNode 类的对象。MacroVarList 类提供：计算数值表达式的值服务 CalVarExpr()；对操作数进行相应的运算服务。OPTRAct()；从表头插入新变量服务 InsertVar()；更新或新建变量服务 UpdateVar()；从表中搜索变量的值服务 GetValue()。当需要访问这些符号时，宏管理器就对 MvList 进行操作。

### 关于条件汇编

条件汇编指令的作用是使源程序中的一部分程序行根据需要决定是否进行汇编。条件汇编指令可以作为$控制行的汇编控制，也可以作为不带$符的正常汇编控制。它们的差别在于前者只能访问由 SET 和 RESET 指令定义的符号，而后者可访问源程序中除 SET 和 RESET 符号之外的所有符号，因此可用于宏中。

IF 块可嵌套使用，最大嵌套深度为 63 层（Keil 的 A51 只达 10 层）。如果 IF、ELSEIF 和 ELSE 块不汇编，则其嵌套的条件块也不汇编。

根据两类条件汇编语句（IF 和$IF）的不同，要对其分别独立的实现。在宏汇编器 masm 对象中，包含了两个 IFeStack 栈。其中一个用于 IF 等语句，另外一个用于$IF 等语句。IFeStack 栈对象的每个元素都是一个二元组（s,e），其中 s 是当前 IF 块的状态，布尔类型，取值规则如下：若当前汇编行正处于一个 IF 语句的控制下，s 值为 true；若当前汇编行正处于一个 ELSE 语句的控制下，s 值为 false。e 是当前汇编行所在的 IF 语句嵌套层（包括 ELSE）控制下，该 IF 语句后面所带的数值表达式的值。表达式值为零，e 为 false；表达式值非零，e 为 true。

若当前汇编行正处于一个 ELSEIF 语句的控制下，s 所取得新值需要看 s 原值，若 s 原值已经是 true，则 s 新值必为 false，而且 e 值被设定恒为 true；若 s 原值已经是 true，则 s 新值必为 false，而 e 值要看当前汇编行所在的 IF 语句嵌套层（包括 ELSE）控制下，该 IF 语句后面所带的数值表达式的值。表达式值为零，e 为 false；表达式值非零，e 为 true。

条件汇编的运行机理如下：

语法分析阶段，在每次对一个汇编行进行汇编前，首先调用 CondLineJudge()过滤器服务，判断一下该汇编行是否应该进行汇编，只有过滤器认可之后，该汇编行才正式进入汇编；否则放弃该行。该过滤器实际上是对二元组（s,e）中的 s 和 e 作异或运算，也就是当 s 和 e 的值相同时返回结果为 true，当 s 和 e 的值相同时返回结果为 false。过滤器要对 IF 和$IF 两个 IFeStack 栈都进行检测，只有两关都通过了的汇编行才能得到汇编；否则该汇编行被忽略并禁用。

两个 IFeStack 栈（IfStack 对象和 MroIfStack 对象）初始化时，栈中各有一个二元组（0，0）。若遇到 IF 语句，则给 IfStack 压栈，即压进一个赋好值的二元组（s，e）。若遇到 ENDIF 语句，则给 IfStack 退栈，即从栈中弹出一个的二元组（s，e）。若遇到 ELSEIF 语句，则不压栈也不退栈，而是对栈顶元素直接修改。若遇到$IF 语句，则给 MroIfStack 压栈，即压进一个赋好值的二元组（s，e）。若遇到$ENDIF 语句，则给 MroIfStack 退栈，即从栈中弹出一个的二元组（s，e）。若遇到$ELSEIF 语句，则不压栈也不退栈，而是对栈顶元素直接修改。使用栈的形式是为了保存条件嵌套时各层的状态格局。由于对两个栈的操作是相互独立的，所以 IF 语句与$IF 语句互不影响。

## 七、使用说明

### 使用方法

使用方法很简单，在 DOS 命令提示符后输入：

ZLG51 源文件名.asm

程序就投入运行，屏幕上显示：

```
====== 51 系列汇编器 Version: 0.01 ======
=== 周立功单片机发展有限公司 2002.5~9.===
=================== 编程员: 郑智峰 =====
Compling file 源文件名.asm...
```

如果你的源文件没有错误，则会一步一步显示以下信息：

```
File Parse Ok!
CreateVarTable Ok!
ToObjCode Ok!
OBJ records have been written to mytest.obj successfully.
List file mytest.lst has been written successfully.
==== Assemblier ended. ====
```

如果你的源文件有错误，则会显示：

***Error(Line 行号, 源文件名): 错误信息

如果汇编器向你发出警告，则会显示：

***Warning(Line 行号, 源文件名): 警告信息

## 错误信息

错误信息有下列各条：

```
"Undefined label or variable.",                     // LabelUnDefErr     3
"Missing label defination before ':'.",             // LabelDefNullErr   4
"Missing variable name defination.",                // VarDefNullErr     5
"Name of parameter error.",                         // ParaNameDefErr    6
"Undefined word.",                                  // UndefWordErr      7
"Missing identifier definition of EQU.",            // EQUDefNullErr     8
"Missing identifier definition of SET.",            // SETDefNullErr     9
"Missing identifier definition of BIT.",            // BITDefNullErr    10
"Unidentified word here. Cannot find this macro.",  // InstNeededErr    11
"Insufficient arguments.",                          // ArguInsufErr     12
"Excessive arguments or illegal symbol.",           // ArguExcevErr     13
"Expression in NULL.",                              // ExpNULLErr       14
"Expression missing ')'.",                          // MissRtErr        15
"Expression missing '('.",                          // MissLtErr        16
"Expression illegal syntax.",                       // ExpSyrErr        17
"Expression element incompleted.",                  // MissEleErr       18
"Excessive symbol ','.",                            // ExcesComErr      19
"Missing symbol ',' between expressions.",          // ExpExcessErr     20
"Missing symbol ',' between operands.",             // OPadExcessErr    21
"Identifier name duplicated.",                      // LblNameDuplcate  22
"Identifier type not match.",                       // LbltypeNotmatch  23
"Data Exceeding length.",                           // DataExceedErr    24
"Divide by Zero.",                                  // DividerIsZeroErr 25
"Bit address out of range.",                      // BitAddrOutRangeErr 26
"Shifting data out of range.",                      // ShiftOutRangeErr 27
"Expression nest out of range.",                    // ExpNestOuterErr  28
"Illegal register expression.",                     // RegisterInExpErr 29
"Identifier missing declaration.",                  // UndefLabelErr    30
"Location address out of range.",                 // LocAddrOutRangeErr 31
"Label value is not ready yet.",                    // LabelNotReadyErr 32
"The first Operand is incorrect, type not match.",  // WrongOperandsErr 33
"The second Operand is incorrect, type not match.", //                  34
```

```
"The third Operand is incorrect, type not match.",     //                   35
"Address mode not match, instruction syntax error.",  // InstrctSyntaxErr 36
"ORG address error.",                                  // ORGdataCalErr    37
"Relative address range exceeded.",                    // RelativeAddrErr  38
"Operand illegal syntax.",                             // OPNDsyntaxErr    39
"Missing expression definition of CODE.",              // CODEDefNullErr   40
"Missing expression definition of DATA.",              // DATADefNullErr   41
"Missing expression definition of XDATA.",             // XDATADefNullErr  42
"Missing expression definition of IDATA.",             // IDATADefNullErr  43
"8-bits data address out of range.",                   // DATAOutRangeErr  44
"ORG address must not less than the segment base.",    // ORGdataLessErr   45
"Bad relocatable expression.",                         // BadRelExpErr     46
"Illegal type operation in expression.",               // IllTypExpErr     47
"Operand expression type not match.",                 // ExpTypNotMatchErr 48
"Refer segment not match.",                            // SegNotMatchErr   49
"Register expression out of range.",                   // REGOutRangeErr   50
"Missing identifier definition of segment.",           // SEGDefNullErr    51
"Misplace segment relocatable type.",                  // SegRelDefErr     52
"Number defination error.",                            // NumberDefErr     53
"The first character of identifier cannot be digit.", // BadNumberErr     54
"Segment Name has not been defined yet.",              // RSegNameErr      55
"Address must be absolute.",                           // NotAbsAddrErr    56
"Stack overflow, expression nest too deep.",           // StkOverFlowErr   57
"Missing identifier definition of MACRO.",             // MacroDefNullErr  58
"Can not find a match Macro for ENDM.",                // ENDMNotMatchErr  59
"Macro named duplicately.",                           // MacroDuplicateErr 60
"Formal parameters named duplicately.",                // MacroArgvDupErr  61
"Misplaced LOCAL. It must be placed in a Macro-body.",// MacroLocalErr    62
"Misplaced EXITM. It must be placed in a Macro-body.",// MacroExitmErr    63
"Missing identifier name after LOCAL.",                // MissLocalIDErr   64
"Illegal labels before Macro defination.",             // IllLBforMroErr   65
"String length beyond the limit in macro.",            // LenExcesMroErr   66
"Lex errors occured in macro expanding.",              // MacroLexErr      67
"Macro defination nested too deep.",                   // MroDefNestOutErr 68
"Macro calling nested too deep.",                     // MroCallNestOutErr 69
"Label is not allowed here.",                          // LabelNotAllowErr 70
"REPT cannot accept bad expression.",                  // BadExptReptErr   71
"Non-relocated data expected here.",                   // ExpeABSdataErr   72
"Negative data are not allowed here.",                // NegDatNotAllowErr 73
"LOCAL should not be here.",                          // LocaltNotAllowErr 74
"Registerbank out of range. USING 0 ~ 3.",             // UsingWrongErr    75
"Missing formal argument here.",                       // MissFomArguErr   76
"Missing parameters here.",                            // MissParasErr     77
"'<' expected here.",                                  // ParaMissLtErr    78
"'>' expected here.",                                  // ParaMissRtErr    79
"A <list> expected here, use '<' & '>' please.",       // ParaMissLRtErr   80
"Illegal assemble control instruction here.",          // DogCtrlIllErr    81
"Control instruction should not be defined twice.",    // CtrlConflictErr  82
"Cannot open the specified file.",                     // spFileOpenErr    83
"IF ELSE statements nest too deep.",                   // IfElseNestOutErr 84
"Misplaced ENDIF, without corresponding IF.",          // ENDIFMisplaceErr 85
"Missing ENDIF which corresponds IF.",                 // MissingENDIFErr  86
"IF expression error.",                                // IFexprErr        87
"Misplaced ELSE, without corresponding IF.",           // ELSEMisplaceErr  88
"Misplaced ELSEIF, without corresponding IF.",         // ESIFMisplaceErr  89
"Only one instruction should be in one line.",         // ExcvPswWordErr   90
"Operator '.' is not allowed here.",                   // ComNotAllowErr   91
"Bad operator in the expression here.",                // BadOPinExprErr   92
"NUL must immediately follow IF.",                     // NULnotFloIFErr   93
"Missing terminal symbol \'.",                         // SingleQuoteErr   94
"Missing terminal symbol \".",                         // DoubleQuoteErr   95
```

### 警告信息

警告信息有下列各条：

```
"Data have been truncated to 8-bit.",                  //  1  DataBeTrun8Warn
"Data have been truncated to 16-bit.",                 //  2  DataBeTrun16Warn
"File needs END to finish.",                           //  3  FileNeedEndWarn
"Data have been extended to 16-bit.",                  //  4  DataBeExtentWarn
"Data have been truncated to 11-bit.",                 //  5  DataBeTrun11Warn
"Label name has been truncated to 40 char.",           //  6  LBLTooLongWarn
"Label type not match.",                               //  7  LBLTypNotMatchWarn
"Label segment not match.",                            //  8  LBLSegNotMatchWarn
"Excessive parameters in Macro calling. Be ignored.",  //  9  ExcesParaMroWarn
"Insufficient parameter in Macro calling. Filled with NUL.", // 10 InsufParaMroWarn
"Data type are regarded as number here.",              // 11  DataTakeAsNUMWarn
"Negative data are regarded as their complement.",     // 12  NegTakeAsCplxWarn
"Missing title apointment.",                           // 13  MissTitleWarn
"Missing date apointment.",                            // 14  MissDateWarn
"Number out of range. Setting by default.",            // 15  NumOutRangeWarn
"Cannot restore without saveing first.",               // 16  RsStkRstorWarn
"Be cautious that the type of the result is not number.", // 17 IFexprWarn
"Negative number will be convented to positive number.",  // 18 NegContPosWarn
```

## 八、程序说明

SrcMasm.doc（205 页）

## 九、运行测试

### 测试程序

使用了以下一段测试程序：（mytest.asm）

```
        CR    equ  13
        LF    equ  10

extrn         code(serout)
public        start
samp          SEGMENT CODE

              rseg samp
start:        mov    scon,  #52h
              mov    tcon,  #0d2h
              mov    th1,   #13
              setb   TR1
load:         mov    dptr,#msg
loop:         clr    A
              movc   A,@A+DPTR
              LCALL  serout
              cjne   A,#LF,more
              sjmp   load
more:         mov    A,  DPL
              inc    A
              mov    DPL,  A
              jnc    loop
              inc    DPH
              sjmp   loop
msg:          db     "MESSAGE",CR,LF
              end
```

## 输出信息

输出以下信息:

```
====== 51 系列汇编器 Version: 0.12 ======
==== 周立功单片机发展有限公司 2002. =====
================ Programmer: 郑智峰 =====
Compling file mytest.asm...
File Parse Ok!
CreateVarTable Ok!
ToObjCode Ok!
OBJ records have been written to mytest.obj successfully.
List file mytest.lst has been written successfully.
==== Assemblier ended. ====
```

## 输出列表文件

输出的 LST 列表文件：

```
  MACRO ASSEMBLER A51 宏汇编器
  周立功单片机发展有限公司 2002.
-------------- List begin ---------------
LOC   OBJ         LINE  SOURCE
 (000D)  N NUMB      1  CR    equ   13
 (000A)  N NUMB      2  LF    equ   10
                     3
0000                 4  extrn code(serout)
                     5  public start
                     6  samp SEGMENT CODE
                     7
----  ----           8       rseg samp
0000  759852         9  start:  mov   scon, #52h
0003  7588D2        10  mov   tcon, #0d2h
0006  758D0D        11  mov   th1,#13
0009  D28E          12  setb  TR1
000B  900000 F      13  load:   mov   dptr,#msg
000E  E4            14  loop:   clr   A
000F  93            15          movc  A,@A+DPTR
0010  120000 F      16          LCALL serout
0013  B40A02        17          cjne  A,#LF,more
0016  80F3          18          sjmp  load
0018  E582          19  more:   mov   A, DPL
001A  04            20          inc   A
001B  F582          21          mov   DPL, A
001D  50EF          22          jnc   loop
001F  0583          23          inc   DPH
0021  80EB          24          sjmp  loop
0023  4D455353      25  msg:    db    "MESSAGE",CR,LF
0027  4147450D
002B  0A
002C                26          end

-------------- List end ----------------
Name                                      Type  Value  attribute
----------------------------------------------------------------
CR.................................... N NUMB  000DH  A  CrLine=[1]
LF.................................... N NUMB  000AH  A  CrLine=[2]
SEROUT................................ C ADDR  0000H  E  CrLine=[4]
SAMP..................................  SEG    0000H  R  CrLine=[6]
START................................. C ADDR  0000H  R  CrLine=[9]
LOAD.................................. C ADDR  000BH  R  CrLine=[13]
LOOP.................................. C ADDR  000EH  R  CrLine=[14]
MORE.................................. C ADDR  0018H  R  CrLine=[19]
MSG................................... C ADDR  0023H  R  CrLine=[25]
```

## 输出目标文件

输出的 OBJ 目标文件:

```
0000: 70 36 00 FF 11 41 35 31-20 22 6D 79 74 65 73 74   p6...A51 "mytest
0010: 2E 61 73 6D 22 20 00 00-00 00 00 00 0A 6D 79 74   .asm" .......myt
0020: 65 73 74 2E 6F 62 6A 01-00 00 00 00 00 0A 6D 79   est.obj.......my
0030: 74 65 73 74 2E 61 73 6D-D1 02 0A 00 06 6D 79 74   test.asm.....myt
0040: 65 73 74 FD 00 4B 0F 0F-00 01 00 00 01 00 00 00   est..K..........
0050: 2C 00 04 53 41 4D 50 7F-19 0D 00 02 00 00 00 00   ,..SAMP.........
0060: 06 53 45 52 4F 55 54 F0-17 0D 00 01 00 00 00 00   .SEROUT.........
0070: 00 05 53 54 41 52 54 48-10 09 00 00 06 6D 79 74   ..STARTH.....myt

0080: 65 73 74 3B 24 0F 00 00-00 00 0A 6D 79 74 65 73   est;$......mytes
0090: 74 2E 61 73 6D AE 23 29-00 02 00 00 82 00 00 00   t.asm.#)........
00A0: 00 00 00 81 00 00 00 00-00 00 83 00 00 00 00 00   ................
00B0: 00 84 00 00 00 00 01 00-00 00 00 00 04 53 41 4D   .............SAM
00C0: 50 72 23 0E 00 01 01 00-00 00 00 00 05 53 54 41   Pr#..........STA
00D0: 52 54 3A 23 5F 02 00 00-00 02 80 00 00 02 50 30   RT:#_.........P0
```

```
00E0: 00 00 02 81 00 00 02 53-50 00 00 02 82 00 00 03    .......SP.......
00F0: 44 50 4C 00 00 02 83 00-00 03 44 50 48 00 00 02    DPL.......DPH...

0100: 87 00 00 04 50 43 4F 4E-00 00 02 88 00 00 04 54    ....PCON.......T
0110: 43 4F 4E 00 00 02 89 00-00 04 54 4D 4F 44 00 00    CON.......TMOD..
0120: 02 8A 00 00 03 54 4C 30-00 00 02 8B 00 00 03 54    .....TL0.......T
0130: 4C 31 00 00 02 8C 00 00-03 54 48 30 00 00 02 8D    L1.......TH0....
0140: 00 00 03 54 48 31 00 00-02 90 00 00 02 50 31 00    ...TH1.......P1.
0150: 00 02 98 00 00 04 53 43-4F 4E 00 00 02 99 00 00    .......SCON.....
0160: 04 53 42 55 46 00 00 02-A0 00 00 02 50 32 00 00    .SBUF.......P2..
0170: 02 A8 00 00 02 49 45 00-00 02 B0 00 00 02 50 33    .....IE.......P3

0180: 00 00 02 B8 00 00 02 49-50 00 00 02 D0 00 00 03    .......IP.......
0190: 50 53 57 00 00 02 E0 00-00 03 41 43 43 00 00 02    PSW.......ACC...
01A0: F0 00 00 01 42 00 00 04-88 00 00 03 49 54 30 00    ....B.......IT0.
01B0: 00 04 89 00 00 03 49 45-30 00 00 04 8A 00 00 03    .......IE0......
01C0: 49 54 31 00 00 04 8B 00-00 03 49 45 31 00 00 04    IT1.......IE1...
01D0: 8C 00 00 03 54 52 30 00-00 04 8D 00 00 03 54 46    ....TR0.......TF
01E0: 30 00 00 04 8E 00 00 03-54 52 31 00 00 04 8F 00    0.......TR1.....
01F0: 00 03 54 46 31 00 00 04-98 00 00 02 52 49 00 00    ..TF1.......RI..

0200: 04 99 00 00 02 54 49 00-00 04 9A 00 00 03 52 42    .....TI.......RB
0210: 38 00 00 04 9B 00 00 03-54 42 38 00 00 04 9C 00    8.......TB8.....
0220: 00 03 44 45 4D 00 00 04-9D 00 00 03 53 4D 32 00    ..DEM.......SM2.
0230: 00 04 9E 00 00 03 53 4D-31 00 00 04 9F 00 00 03    .......SM1......
0240: 53 4D 30 00 00 04 A8 00-00 03 45 58 30 00 00 04    SM0.......EX0...
0250: A9 00 00 03 45 54 30 00-00 04 AA 00 00 03 45 58    ....ET0.......EX
0260: 31 00 00 04 AB 00 00 03-45 54 31 00 00 04 AC 00    1.......ET1.....
0270: 00 02 45 53 00 00 04 AF-00 00 02 45 41 00 00 04    ..ES.......EA...

0280: B8 00 00 03 54 58 30 00-00 04 B9 00 00 03 50 54    ....TX0.......PT
0290: 30 00 00 04 BA 00 00 03-50 4B 31 00 00 04 BB 00    0.......PK1.....
02A0: 00 03 50 54 31 00 00 04-BC 00 00 02 50 53 00 00    ..PT1.......PS..
02B0: 04 D0 00 00 01 50 00 00-04 D1 00 00 02 46 31 00    .....P.......F1.
02C0: 00 04 D2 00 00 02 4F 56-00 00 04 D3 00 00 03 52    .......OV.......R
02D0: 53 30 00 00 04 D4 00 00-03 52 53 31 00 00 04 D5    S0.......RS1....
02E0: 00 00 02 46 30 00 00 04-D6 00 00 02 41 43 00 00    ...F0.......AC..
02F0: 04 D7 00 00 02 43 59 00-00 05 0D 00 00 02 43 52    .....CY.......CR

0300: 00 00 05 0A 00 00 02 4C-46 01 00 00 0B 00 00 04    ........LF......
0310: 4C 4F 41 44 01 00 00 0E-00 00 04 4C 4F 4F 50 01    LOAD.......LOOP.
0320: 00 00 18 00 00 04 4D 4F-52 45 01 00 00 23 00 00    ......MORE...#..
0330: 03 4D 53 47 8D 07 31 00-01 00 00 00 75 98 52 75    .MSG..1.....u.Ru
0340: 88 D2 75 8D 0D D2 8E 90-00 00 E4 93 12 00 00 B4    ..u.............
0350: 0A 02 80 F3 E5 82 04 F5-82 50 EF 05 83 80 EB 4D    .........P.....M
0360: 45 53 53 41 47 45 0D 0A-AE 09 11 00 0C 00 04 01    ESSAGE..........
0370: 01 00 23 00 11 00 04 02-00 00 00 00 9A 23 62 00    ..#..........#b.

0380: 03 01 00 00 00 09 00 01-00 03 00 0A 00 01 00 06    ................
0390: 00 0B 00 01 00 09 00 0C-00 01 00 0B 00 0D 00 01    ................
03A0: 00 0E 00 0E 00 01 00 0F-00 0F 00 01 00 10 00 10    ................
03B0: 00 01 00 13 00 11 00 01-00 16 00 12 00 01 00 18    ................
03C0: 00 13 00 01 00 1A 00 14-00 01 00 1B 00 15 00 01    ................
03D0: 00 1D 00 16 00 01 00 1F-00 17 00 01 00 21 00 18    .............!..
03E0: 00 43 10 09 00 03 06 6D-79 74 65 73 74 38 04 0C    .C.....mytest8..
03F0: 00 06 6D 79 74 65 73 74-00 00 01 00 43             ..mytest....C
```

## OBJ 查看器查看结果

用我写的 OBJ 查看器查看的结果如下:

```
Hello! OBJ ANALIZER.
        Opening file test.obj...
        --------------------------------------------------------------------------
        [Module 70H?] FF   [?] A51 "mytest.asm"
           [?] 0000   [Modify time] Thu Jan 01 08:00:00 1970  mytest.obj
           [?] 0001   [Modify time] Thu Jan 01 08:00:00 1970  mytest.asm
        --------------------------------------------------------------------------
        [Module Header Record 02H]
           [module name]=mytest
           [TRN ID]=FDH    (This module is generated by ASM51.)  [unused byte]=00H
        --------------------------------------------------------------------------
        [Segment Definitions Record 0FH]
           [SEG ID]=0001 (Relocatable segment #0000)
           [SEG INFO]=00
           [E=0] Not empty segment.              [bit6=0]
           [OVL=0] The segment is NOT overlayable.
           [SEG REG]= register bank #0
           [SEG TYPE]=CODE(0)
           [REL TYP]=01 (UNIT)                   [unused byte]=00
           [Segment Base]=0000
           [Segment Size]=002C
           [Segment Name]=SAMP
        --------------------------------------------------------------------------
        [External Definitions Record 19H]
          [ID BLK]=02 [EXT ID]=0000 [SYM INFO]=00 [unused]=00 [Ext Name]SEROUT
        --------------------------------------------------------------------------
        [Public Definitions Record 17H]
          [SEG ID]=0001 [SYM INFO]=00 [Offset]=0000 [unused]=00 [Public Name]START
        --------------------------------------------------------------------------
        [Scope Definition Record 10H]
           [BLK TYP]=00 (module block)      [Block Name]mytest
        --------------------------------------------------------------------------
        [SOURCE NAME Record 24H]
           [unused byte]=00 00 00      [File Name]mytest.asm
        --------------------------------------------------------------------------
        [DEBUG Record 23H]
```

```
 [Defined Type]=02 (Segment symbols)
  [SEG ID]=0000 [TYPE]= ???  [Offset]=0000 [unused]=00 [Symbol Name]
  [SEG ID]=0000 [TYPE]= ???  [Offset]=0000 [unused]=00 [Symbol Name]
  [SEG ID]=0000 [TYPE]= ???  [Offset]=0000 [unused]=00 [Symbol Name]
  [SEG ID]=0000 [TYPE]= ???  [Offset]=0000 [unused]=00 [Symbol Name]
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=0000 [unused]=00 [Symbol Name]SAMP
--------------------------------------------------------------------------
[DEBUG Record 23H]
 [Defined Type]=01 (Public symbols)
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=0000 [unused]=00 [Symbol Name]START
--------------------------------------------------------------------------
[DEBUG Record 23H]
 [Defined Type]=00 (Local symbols)
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0080 [unused]=00 [Symbol Name]P0
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0081 [unused]=00 [Symbol Name]SP
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0082 [unused]=00 [Symbol Name]DPL
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0083 [unused]=00 [Symbol Name]DPH
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0087 [unused]=00 [Symbol Name]PCON
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0088 [unused]=00 [Symbol Name]TCON
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0089 [unused]=00 [Symbol Name]TMOD
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=008A [unused]=00 [Symbol Name]TL0
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=008B [unused]=00 [Symbol Name]TL1
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=008C [unused]=00 [Symbol Name]TH0
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=008D [unused]=00 [Symbol Name]TH1
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0090 [unused]=00 [Symbol Name]P1
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0098 [unused]=00 [Symbol Name]SCON
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=0099 [unused]=00 [Symbol Name]SBUF
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00A0 [unused]=00 [Symbol Name]P2
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00A8 [unused]=00 [Symbol Name]IE
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00B0 [unused]=00 [Symbol Name]P3
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00B8 [unused]=00 [Symbol Name]IP
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00D0 [unused]=00 [Symbol Name]PSW
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00E0 [unused]=00 [Symbol Name]ACC
  [SEG ID]=0000 [TYPE]=DATA  [Offset]=00F0 [unused]=00 [Symbol Name]B
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=0088 [unused]=00 [Symbol Name]IT0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=0089 [unused]=00 [Symbol Name]IE0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008A [unused]=00 [Symbol Name]IT1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008B [unused]=00 [Symbol Name]IE1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008C [unused]=00 [Symbol Name]TR0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008D [unused]=00 [Symbol Name]TF0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008E [unused]=00 [Symbol Name]TR1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=008F [unused]=00 [Symbol Name]TF1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=0098 [unused]=00 [Symbol Name]RI
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=0099 [unused]=00 [Symbol Name]TI
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009A [unused]=00 [Symbol Name]RB8
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009B [unused]=00 [Symbol Name]TB8
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009C [unused]=00 [Symbol Name]DEM
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009D [unused]=00 [Symbol Name]SM2
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009E [unused]=00 [Symbol Name]SM1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=009F [unused]=00 [Symbol Name]SM0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00A8 [unused]=00 [Symbol Name]EX0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00A9 [unused]=00 [Symbol Name]ET0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00AA [unused]=00 [Symbol Name]EX1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00AB [unused]=00 [Symbol Name]ET1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00AC [unused]=00 [Symbol Name]ES
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00AF [unused]=00 [Symbol Name]EA
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00B8 [unused]=00 [Symbol Name]TX0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00B9 [unused]=00 [Symbol Name]PT0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00BA [unused]=00 [Symbol Name]PK1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00BB [unused]=00 [Symbol Name]PT1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00BC [unused]=00 [Symbol Name]PS
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D0 [unused]=00 [Symbol Name]P
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D1 [unused]=00 [Symbol Name]F1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D2 [unused]=00 [Symbol Name]OV
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D3 [unused]=00 [Symbol Name]RS0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D4 [unused]=00 [Symbol Name]RS1
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D5 [unused]=00 [Symbol Name]F0
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D6 [unused]=00 [Symbol Name]AC
  [SEG ID]=0000 [TYPE]=BIT   [Offset]=00D7 [unused]=00 [Symbol Name]CY
  [SEG ID]=0000 [TYPE]=NUM   [Offset]=000D [unused]=00 [Symbol Name]CR
  [SEG ID]=0000 [TYPE]=NUM   [Offset]=000A [unused]=00 [Symbol Name]LF
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=000B [unused]=00 [Symbol Name]LOAD
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=000E [unused]=00 [Symbol Name]LOOP
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=0018 [unused]=00 [Symbol Name]MORE
  [SEG ID]=0001 [TYPE]=CODE  [Offset]=0023 [unused]=00 [Symbol Name]MSG
--------------------------------------------------------------------------
[Content Record 07H]
   [SEG ID]=0001            [Offset]=0000
   [DAT] 75 98 52 75 88 D2 75 8D ...... 0A
--------------------------------------------------------------------------
[Fixup Record 09H]
   [Ref Loc]=000C  [REF TYP]=04(WORD)
  [OPERAND]: [ID BLK]=01(rel)  [ID]=0001 [Offset]=0023
   [Ref Loc]=0011  [REF TYP]=04(WORD)
  [OPERAND]: [ID BLK]=02(ext)  [ID]=0000 [Offset]=0000
--------------------------------------------------------------------------
[DEBUG Record 23H]
```

```
 [Defined Type]=03 (Line numbers)
   [SEG ID]=0001    [Offset]=0000    [Line Number]=0009
   [SEG ID]=0001    [Offset]=0003    [Line Number]=000A
   [SEG ID]=0001    [Offset]=0006    [Line Number]=000B
   [SEG ID]=0001    [Offset]=0009    [Line Number]=000C
   [SEG ID]=0001    [Offset]=000B    [Line Number]=000D
   [SEG ID]=0001    [Offset]=000E    [Line Number]=000E
   [SEG ID]=0001    [Offset]=000F    [Line Number]=000F
   [SEG ID]=0001    [Offset]=0010    [Line Number]=0010
   [SEG ID]=0001    [Offset]=0013    [Line Number]=0011
   [SEG ID]=0001    [Offset]=0016    [Line Number]=0012
   [SEG ID]=0001    [Offset]=0018    [Line Number]=0013
   [SEG ID]=0001    [Offset]=001A    [Line Number]=0014
   [SEG ID]=0001    [Offset]=001B    [Line Number]=0015
   [SEG ID]=0001    [Offset]=001D    [Line Number]=0016
   [SEG ID]=0001    [Offset]=001F    [Line Number]=0017
   [SEG ID]=0001    [Offset]=0021    [Line Number]=0018
-----------------------------------------------------------------------------
[Scope Definition Record 10H]
   [BLK TYP]=03 (module end)        [Block Name]mytest
-----------------------------------------------------------------------------
[Module End Record 04H]
   [module name]=mytest       [unused byte]=00H 00H
   [REG MSK]=01H   Used bank #0.         [unused byte]=00H
-----------------------------------------------------------------------------
Press any key...
```

## 十、本宏汇编器的新特点

下面这些新特点是目前其他宏汇编器不具有的。它们是本宏汇编器的一些新功能：

（1）能使用大块的注释，只要您把要注释的文本用“/*”和“*/”标注起来就行。这种用法跟 C 语言的用法是一样的，不过请注意不要在同一汇编行中把“*/”放在“;”（分号）后面。因为分号会把它后面直到行末的所有字符“吃掉”。

（2）能使用 EQU 和 SET 指令把寄存器名赋给一个指定的符号名，并且，可以是：A、R0～R7、DPTR、PC、C、AB，A+DPTR，A+PC。更甚，寄存器名可以参加“加减运算”！例如：R0＋1＝R1，R7－2＝R5。您可以这样使用：

```
RegX    set    R0
…
RegX    set    RegX + 1
…
```

寄存器从小到大排序是：R0、R1、R2、R3、R4、R5、R6、R7、PC、DPTR、A、AB、C、A+DPTR、A+PC，如果运算结果越界，宏汇编器会给你报告一个错误。

（3）在 CSEG、DSEG、XSEG、ISEG、BSEG 指令的后面，如果需要使用 AT 指定一个基地址，那么 AT 这个单词可省略。如写成：CSEG 1234H。

（4）可以传递超过 16 个实际参数到宏体里面，您可以定义超过 64 个形式参数，前提是所有的形参都必须在一个汇编行中定义完毕，这意味着您的形参定义汇编行不能超过 255 个字符。

（5）在一个宏里面定义的 LOCAL 标号可突破 16 个的限制，您可以定义超过 64 个 LOCAL 标号，前提是所有的 LOCAL 标号都必须在一个汇编行中定义完毕，这意味着您的 LOCAL 标号定义汇编行不能超过 255 个字符。

（6）宏嵌套可达 64 层（Keil 的 A51 只达 10 层）。

## 十一、 设计小结

通过半年多的设计和编码，终于得到了这个支持宏指令的汇编器，它凝结了不少的心血。它能支持 51 系列单片机的所有汇编指令及其寻址方式，支持所有的汇编伪指令，能正确产生 OBJ 文件和 LST 文件，能报告汇编过程中发生的各种错误（包括词法、语法、语义等的错误），并显示出错行的行号和源文件名，以便用户改正。

有待实现的功能：产生交叉参考表。

## 十二、 心得体会

我从接受这个项目开始，就觉得这是一个挑战，也是一个很好的锻炼机会。做好任何一件事都是不简单的。尽管有些事情表面上看不很难，但随着研究的深入，就会发觉事物相互间的复杂的联系并不是从一开始就能看清楚的。牵一丝而引万钧。联系是普遍存在的，有表面上的，也有隐藏在深处的；有现象的，也有本质的；有必然的，也有偶然的。所谓规律就是这些本质的必然的联系。研究一个事物，就是要从外至内、由表及里、去粗取精、去伪存真地发现它们。我们编程的任务，则不仅要尽量地把事物的面目和状态详尽地用数据表示出来，还要通过条件的约束如实地反映它们内部及之间的各种联系。上世纪五十年代出现的世界上第一个编译程序，花费了十八人年，即十八个人工作一年。可见编译程序的复杂性。现在由于程序设计环境的改善和软件工具的发展，开发编译程序的周期可大大缩短。编译原理也逐渐发展完善，成为一个理论体系。从我接触编译原理，到如今部分地实现这个汇编器，在这个过程中我学到了不少知识，得到了不少经验。它令我对汇编的流程及其机理有了比较深入的理解。

一个好的设计风格，能保证软件规模渐渐庞大起来时保持软件的正确性、可读性和健壮性。同时，采用好的算法，能极大地提高其运行效率。通过使用面向对象的程序设计，我在软件编程的设计思想、风格上有了长足的改进，培养了发现和创造的能力，造就了自己作为一个软件编程者应该具有的科学的严谨的精益求精的态度，也增强了自己开发更大规模的软件项目的信心。

纸上得来终觉浅，绝知此事要躬行。学习离不开实践。求学阶段中是在学习中实践，工作阶段中则是在实践中学习。学以致用，受益匪浅。

## 十三、 致谢

感谢周立功先生对我的栽培，感谢公司全体员工对我的支持，尤其是尹寒冬工程师、戚军工程师、黄晓清工程师、刘亚林、黄邵跃、王迪明等同事给我的技术支持和鼓励，才使我的得到现今的成果。我衷心的感谢您们!

## 十四、 参考文献

《编译原理与实现》，金成植编著，高等教育出版社。
《编译原理》，吕映芝、张素琴、蒋维杜编著，清华大学出版社。
《编译设计原理》，张益新编著，华南理工大学出版社。
《80C51 宏汇编程序设计语言》，微计算机信息特刊，《微计算机信息》编辑部。
《单片机原理及接口技术》，朱定华编著，电子工业出版社。
《单片机高级语言 C51 应用程序设计》，徐爱钧，彭秀华编著，电子工业出版社。
《数据结构（C 语言版）》，严蔚敏，吴伟民编著，清华大学出版社。
《TURBO C 实用大全》，徐金梧、杨德斌、徐科编著，机械工业出版社。
《C++语言和面向对象程序设计（第二版）》，宛延闿编著，清华大学出版社。
《Borland C++ 3.1 百科大全》，施小龙，葛玉宝，邓明辉编著，学苑出版社。
《Borland C++ 4.5 库函数详解》，钱文广，潘志勇等编著，北京航空航天大学出版社。