NOWSMART ASSEMBLER

Version 1.40

2001年03月04日
NOWSMARTSOFT

基本仕様

分野	開発言語
機能	MASM 準拠のアセンブラ
対応 CPU 命令	8086,80186,80286,80386,80486,Pentium,PentiumPro までの全命令 MMX 命令（マルチメディア・エクステンション命令） AMD 社の 3DNow! 命令(Athlon,Duron 等)
擬似命令の仕様	MASM 準拠
ファイル名	NWSA.EXE
動作環境	Windows 95/98/Me/NT/2000
他に必要なもの	Visual C++,Borland C++,Microsoft C++ などのリンカ
実行形態	32 BIT コンソールアプリケーション
取り扱い	完全フリーソフト

注意事項

同梱されているのはアセンブラのみで、リンカはなく、ソースファイルからオブジェクトファイルを作成する機能しかありません。

このアセンブラは、Windows の 32 BIT コンソールプログラムですので、DOS 上では実行できません。

通常は、Visual C++ と共に使用してください。

Install 方法

インストール方法は特にありません。

nwsa.exe がアセンブラ本体であり、コマンドラインから使用したいときは、これにパスが通るようにするだけです。

Visual C++ だけから使用し、コマンドラインから使用する予定が無い場合は、nwsa.exe を任意のフォルダに置いておいて構いません。

Visual C++ の設定

このアセンブラを Visual C++ で使用したい場合、次のように設定します。

1. ワークスペースダイアログの FileView タブを表示させ、対象のプロジェクトの Source Files を右クリックします。

2. 出てきたポップアップメニューの”ファイルをフォルダへ追加”をクリックし、”プロジェクトへファイルを追加”のダイアログを出します。

3. このダイアログで既存のアセンブラのソースファイルを選択するか、または新規に作成したいソースファイル名を入力します。新規に作成する場合は、確認のダイアログボックスが表示されます。

4. FileView に上記で入力したソースファイル名が表示されていることを確認します。

5. そのソースファイル名を右クリックし、ポップアップメニューの”設定”をクリックします。

6. 出てきた”プロジェクトの設定”ダイアログの”カスタムビルド”タブが選択されていることを確認します。

ア．”コマンド”の欄に例えば次のように入力します。
c:\????????\nwsa -Zd -coff -Fo$(OutDir)\$(InputName).obj $(InputPath)
~~~~~~~~~~
下線の部分には nwsa.exe を置いてあるディレクトリ名を指定します。
~~~~~~~~~~

８．同様に出力欄に、
　$(OutDir)\$(InputName).obj
　と入力します。

９．アセンブラソースファイルが複数ある場合は同様の作業を繰り返します。

　上記の設定で、通常の様にビルド作業を行うと自動的に指定したソースファイルがアセンブルされ、CやC++のプログラムとリンクされてEXE形式が出来上がります。

コマンドラインオプション

　コマンドラインからパラメータなしで nwsa.exe を実行すると次のようなヘルプメッセージが出力されます。

```
NOWSMART ASSEMBLER Version 1.10 COPYRIGHT (C) NOWSMARTSOFT 2001

syntax : nwsa <paras>
func   : assemble specified source files.
<paras>::={ -<opt> | /<opt> | <sourcefile> | @<configfile> }
<opt>::=
l        [generate list file] |
fo<file>  [change object filename] |
fl<file>  [change list filename] |
jo       [optimize jump] |
pl       [use local label within procedure] |
coff     [output a coff object] |
zd       [bury linenumber information for debug]
```

コマンドラインオプションは次の通りです。

-l	リストファイルを出力します。デフォルトのファイル名は *.lst です。
-fo<file>	オブジェクトファイル名を変更します。デフォルトのファイル名は *.obj です。

-fl<file>	リストファイルのファイル名を変更します。デフォルトのファイル名は *.lst です。
-jo	jmp 命令を最適化し、１バイトで飛べる範囲の near ラベルへは short jmp を使用します。
-pl	プロシージャ内部（proc～endp）で label 名: で定義されたラベルをプロシージャ内部のローカルラベルにします。
-coff	obj ファイルを COFF 形式にします。デフォルトでは OMF 形式です。
-zd	デバッグ用に関数情報と行番号情報を埋め込みます。現在は COFF 形式のみに対応します。

オプションは、- と / のいずれでも構いません。

複数のソースファイルを指定することが出来ます。その場合それらのファイルを連結したファイルをアセンブルしたのと同じ結果が出力されます。

コマンドラインが長くなる場合は、その内容をコンフィギュレーションファイルに記述して、@ConfigFilename の様に指定します。コンフィギュレーションファイル中では、スペースの変わりに改行でオプションやファイル名を区切ります。; の後ろにコメントを記述することも出来ます。

例:

abc.cfg

```
-l          ;リストファイルを出力する。
-jo         ;short jmp を自動的に生成する。
-pl         ;プロシージャ内の ：ラベルをローカルラベルにする。
-coff       ;COFF 形式で出力する。
-zd         ;デバッグ情報を出力する。
Test.asm        ;アセンブラソースファイル
```

コマンドライン

```
nwsa @abc.cfg
```

なお、オプションに関しては、コンフィギュレーションファイル中のものよりもコマンドラインで指定したものの方が優先されます。相反するものでない限り、合成した結果となります。

明示的にコンフィギュレーションファイルを指定しないと、nwsa.cfg というファイルが自動的にコンフィギュレーションファイルとして使用されます。nwsa.cfg の中には通常はオプションだけを記述し、ファイル名はコマンドラインから指定するようにします。

サンプルコード１

Visual C++ から、nwsa を使用する際にポイントとなる点について例を使って説明します。

まず、C++ 言語とアセンブラとの間で関数名や変数名の受け渡しをするために、C++言語のソース側では、ソースファイルの先頭などで次のように、extern “C” { } を記述します。

```
extern “C” {
    int   _cdecl AsmAdd( int Para1, int Para2 );
}
```

純粋なC言語ではこの extern “C” { と } の部分は必要なく、単に、

```
int   _cdecl AsmAdd( int Para1, int Para2 );
```

で構いません。

この例では、AsmAdd() という関数をアセンブラで記述するものとします。AsmAdd()関数は、int 型の二つの引数を持ち、それらの和を int型で返すような関数です。

C++言語やC言語のソースでは、この関数を使用したい箇所で通常の関数呼び出しを行います。例えば、

```
    int      Sum;
    Sum    = AsmAdd( 100, 200 );
```

という具合です。この例では、100 と 200 という整数値をアセンブラの関数 AsmAdd()

に渡し、その関数が返した結果を Sum に入れようとしています。当然のことながら、100+200=300 という数値が Sum に入ることを期待しています。

次にアセンブラのソースファイルの記述について説明します。アセンブラソースファイルは通常、拡張子を asm にして作成します。nwsa では、拡張子を省略したファイルネームをパラメータとして渡すと、自動的に asm という拡張子を付加したファイルをアセンブルしようとします。もし、拡張子が無いファイルをアセンブルしたい場合は、ファイル名の最後に”.”(ピリオド、ドット)を付加してパラメータに渡して下さい。

この例では、アセンブラソースファイルの中身は次のようになります。

```
                .386
                .model  small, c

_text           segment para use32

AsmAdd          proc    near
                arg     Para1:dword, Para2:dword

                mov     eax,Para1
                add     eax,Para2

                ret

AsmAdd          endp

_text           ends

                end
```

最初の行の .386 という擬似命令は、80386 以降の CPU の命令を使用許可するものです。これを指定することで、32 BIT レジスタが使用できるようになるほか、80386 で新しく付け加わった命令が使用できるようになります。今回のケースでも、32 BIT の eax レジスタを使用しますので必要です。ここで、386 以上の命令も使用許可する場合には、.486 や.Pentium,.PentiumPro などを使用することも出来ます。一般に、80486 以降と、80386 以前ではそれほど大幅な変更点は無いと考えられていますが、80486 ではキャッシュ制御

命令のほか、整数命令などでも重要な命令が追加されています。Pentium や PentiumPro でも、使いこなせば重要な命令の追加があります。また、MMX 命令も許可する場合は、.MMX を指定します。AMD の Athlon 等では 3DNow! 命令が追加されていますが、それらの命令を使用する際には、.3DNow を指定します。

次の行の、.model 擬似命令は、メモリモデルと、アセンブラと共に使用する高級言語の種類を指定します。nwsa では、現在C言語呼び出しにしか正式には対応していません。また、メモリモデルというのは、16 BIT プログラムを作る場合以外には関係ありません。またメモリモデルを指定しても、簡易セグメント擬似命令を使用しなければ何の効果もありません。今回の場合、small の部分がメモリモデルを指定していますが、他に、large,compact,medium,huge 等が指定できますが、Windows 95/98/Me/NT/2000 用のプログラムを作る際にはほとんど関係ありません。必ず small を指定すると考えてください。この指定は、PROC 擬似命令で near や far 指定を省略した場合などに影響を与えますが、省略しない場合は意味を持ちません。最近の MASM では、FLAT という指定も出来るようですが、nwsa ではまだ対応していません。カンマで区切られた第二パラメータの c の部分が高級言語の種類を表しています。これで、C言語のインターフェースを使用することを宣言しています。C言語のデフォルトの呼び出しに対応しています。C言語では、一般に、_cdecl 呼び出しと、_stdcall 呼び出しの二つが使用されます。何も指定しなければ、_cdecl 呼び出しになりますが、先の例では、念のため関数のプロトタイプ宣言で、_cdecl キーワードで関数を修飾し、そのことを明示していました。_cdecl 呼び出しは、スタックの管理を全て呼び出し側が行う方式です。例えば、2つの DWORD パラメータをスタックに push した場合には、責任を持って 2*4=8 バイト分のスタックの除去を呼び出し側が行います。この方式は可変長の引数を呼び出し側が push した場合に呼び出された側がスタックの管理をミスするのを防ぐ働きがあり安全性が高い方法です。一方、_stdcall 呼び出しでは、呼び出された側がスタックの除去を行います。この方式の利点はスタックを除去するコードの量を減らせることにあります。ただし、呼び出し元の push したスタックの量と呼び出された側がスタックポインタを add する量とが完全に一致していなければならず、注意が必要です。Windows API は、基本的に _stdcall 呼び出しを使用しています。

さらに次の行の、_text segment para use32 は、セグメント擬似命令です。この命令は end 文の直前の _text ends 命令と対応し、この間に囲まれた部分のコードを指定したセグメントに収めることを指示する命令です。para は、セグメントの先頭アドレスのアラインメントを指定していますが、通常は気にしないで下さい。use32 は、このセグメントが 32 BIT セグメントであることを示します。他に use16 というものがありますがこちらは 16 BIT セグメントであることを示します。この USE 指定は非常に重要で、現在の Windows では特殊な場合を除いて必ず USE32 を指定する必要があります。32 BIT セグ

メントと 16 BIT セグメントは、386 系のセグメントディスクリプタによって完全に区別されます。32 BIT セグメント内部の命令では、デフォルトで 32 BIT 幅のデータ転送を行い、アドレス計算も 32 BIT で行われます。レジスタも EAX,EBX,ECX,EDX,ESI,EDI,EBP,ESP,EIP
が使用されます。逆に 16 BIT セグメント内部ではデフォルトでデータ幅が 16 BIT、アドレス計算も 16 BIT で行われます。ただし、いずれの場合でも命令の先頭にプリフィクスバイトというものを付加することでデフォルトの指定を反転させることが可能です。バス幅に関連したプリフィクスバイトには2種類有り、データ転送の幅を反転するものと、アドレス計算の幅を反転するものとがあります。これらのプリフィクスはアセンブラによって自動的に挿入されますので、nwsa を使用するユーザーは普段は何も気にする必要はありません。ただし、USE32 の指定がなぜ重要かは知っておいてください。例えば、USE32 指定のセグメント内で eax レジスタを使用する命令を書いた時には、データ幅反転プリフィクスは付きません。ところが、USE32 指定のセグメント内で ax レジスタを使用する命令を書いた場合は自動的にデータ幅反転プリフィクスが付加されます。現在の 32BIT- Windows ではこのセグメントは正しく実行されます。ところが、仮にこのコードが実際には、16 BIT のセグメントで走行した場合には、eax を使うはずの命令が ax を使い、ax を使うはずの命令が eax を使用してしまいます。つまり、USE32 を指定した論理セグメントは、実行段階でも 32 BIT セグメントで実行されなければならないのです。アドレス幅変更プリフィクスの場合も全く同様なことが起こります。重要なことは細部を理解することではなく、今言ったような USE32 指定の概略的な意味を知っておくことです。

次に、AsmAdd 関数本体の宣言に入ります。最初の行では、AsmAdd proc near と書かれていますが、これがこの関数の入り口を示すものです、これは、後の方の AsmAdd endp と対応しており、これらの間に囲まれた部分が AsmAdd 関数であることを示しています。near というキーワードは、この関数を call 命令で呼び出すとき、near コールを使用し、かつ、関数内部で書かれた ret 命令が near return(retn) にコード化されることを指定します。現在の 32 BIT - Windows では、ユーザープログラムで far コールを使用する利点はまず有りませんので、必ず near を指定するようにしてください。far を指定した場合、現在の Visual C++ 環境で何が生じるかは保証できません。恐らくリンク段階で何かのエラーになるか、リンクがうまくいったとしても実行段階で一般保護例外などが生じる可能性が高いです。また、nwsa 自体も far コールや far ジャンプに関しては保証いたしません。そもそも COFF 形式も far コールや far ジャンプに対応していません。また、この proc 擬似命令は、AsmAdd というラベルをグローバル名前空間に登録し、アセンブラソースファイルの任意の位置から参照できるようにし、かつ、public 指定します。public 指定とは、シンボルをリンク段階で他のオブジェクトから参照できるようにすることです。つまり、AsmAdd というラベルは、Visual C++ の中の他の C++ のソースファイルから参照でき

る様になります。次の行の arg 擬似命令は、_cdecl 呼び出しまたは、_stdcall 呼び出しの引数をアセンブラから簡単に使用できるようにするための命令です。この例では、Para1 と Para2 という二つの DWORD(32BIT)型の引数を取ることを示しています。arg 擬似命令で指定した引数名は、指定した型を持ったラベルとして登録されます。今の例では、Para1 や Para2 自体が、DWORD 型を持っています。例の様に、mov 命令や、add 命令のオペランドに自由に使用することが出来ます。内部的には、スタックフレームを作成して、ss:[ebp+?]の形式でスタック上のメモリ内容を参照するようなコードが自動的に作成されます。このようにして、次の行の mov eax,Para1 は、第一引数の内容を eax に代入します。その次の行の add eax,Para2 は、第二引数の内容を eax に加算します。そして最後に、ret 命令が書かれていますが、この命令は、自動的に複合的な命令を埋め込みます。今の場合、スタックフレームを除去するための pop ebp 命令と、near リターン命令である、retn です。これが、near リターン命令になるのは、この関数が、proc near として near 宣言されているためです。proc far としている場合は、retf 命令が吐き出されます。なお、ソースレベルで、retn や retf を明示的に指定することも可能です。その場合はその指示に従った命令を埋め込みます。ただし、retn や retf もやはり自動的にスタックフレームの除去のために pop ebp 命令が埋め込まれることになります。

なお、int 型を戻り値に持つ関数では、戻り値は eax レジスタに入れて戻ってきます。今の例でも、eax レジスタに入った計算結果が、C（C++）言語に受け渡されることになります。

なお、最後の行の end 命令は特に必要ありませんが、書いておくと、それ以後の行がアセンブルされなくなります。この後には任意のコメントを入れることも可能です。

サンプル2

アセンブラから Windows API を直接呼び出す方法について説明します。

通常の Windows API は、_stdcall 呼び出しです。_stdcall 呼び出しは、呼び出し側が push したスタック引数を、呼び出された関数内部で除去して戻ってくるような呼び出し規約です。_stdcall 呼び出しはスタック除去コードを呼び出し側に持つ必要が無いためにコードサイズの節約にはなる反面、呼び出し側と呼び出された側のスタックサイズの仮定のミスマッチがあると、大変危険な方式です。そこで、Windows では、内部的には高級言語から見える関数名の後に @ マークに続けて除去するスタックサイズを 10 進数表記で記したものが実際のラベル名として宣言されています。

例えば、簡単な Windows API として、Sleep() 関数を取り上げます。この関数の働きは(ms)単位で指定した時間だけ制御を Windows に返し、関数を呼び出した側はその時間だけ実行を停止するものです。この関数のプロトタイプ宣言は、次の通りです。

```
void WINAPI     Sleep(DWORD dwMilliseconds);
```

また、他の場所で WINAPI は次のようにマクロ定義されています。

```
#define WINAPI     __stdcall
```

よって説明したとおり、Sleep() API は、_stdcall 呼び出し規約に従った関数です。実際には、DLL になっていますが、とりあえず、DLL であることは気にする必要はありません。

プロトタイプ宣言から分かるように、この関数は DWORD 型の引数を一つだけとりますから、自動除去するスタックのサイズは4です。よって実際のラベル名は、Sleep@4 となります。よって、Sleep() API を　nwsa のアセンブルソースコードから呼び出すには、次のようにします。

```
                .386
                .model  small, c

                extrn   Sleep@4:near

_textsegment    para use32

TestAPI         proc    near

                push    dword ptr 1000
                call    Sleep@4

                ret

TestAPI         endp

_text       ends

                end
```

まず、Sleep@4 ラベルを関数として登録するために、:near を指定して extrn 宣言します。

次に、TestAPI 関数の内部で、1000 という DWORD 値をスタックに push した後、call 文で Sleep@4 関数を呼び出します。スタックの除去は、_stdcall 規約に従い、Sleep@4 関数が行いますから、こちらでは何もする必要はなく、call 関数から戻ってくると既に、push する前のスタックポインタに戻されています。したがって、ここで、ret 文を書けば、TestAPI 関数は正しく呼び出し元へ戻れます。

なお、スタックに push される際、引数の型が、BYTE 型（つまり char）や WORD 型（つまり short）の場合は、規約上、全て、4バイトの DWORD 型 に“切り上げ”が行われます。したがって、スタックから除去される量も1や2ではなく4になることに注意してください。

複数の引数がある場合はそれらのスタックサイズの合計したものがスタック除去量になります。

サンプルコード3

このサンプルでは、MMX 命令や 3DNow! 命令のサンプルコードを書いておきます。ただし、動作内容については全くの無意味です。このようなニモニックが使用できることだけを参考にしてください。

```
                .586
            .PentiumPro
                .MMX
                .3DNow
                .model  large, c

_text           segment para use32

MMX3DNow    proc    near
                arg     Para1:dword, Para2:dword

                femms
                prefetch        byte        ptr        ds:
[edi*4+1000h]



                pfmul   mm1,mm2
                pfmul   mm1,[ebx]
                pfmul   mm1,[ebx+10]
                pfmul   mm1,es:[ebx]
                pfmul   mm1,[ebx+eax*4+10]

                fcomp   st(1)
                fcomi   st(0),st(1)

                cmova  eax,ebx
```

```
                cmove  ebx,ds:[edi]
                mov    eax,Para1
                add    eax,Para2

                paddw  mm0,mm1
                psubsw mm0,mm1

                psllw  mm0,mm1
                psllw  mm1,5

                shl    eax,1

                movd   mm0,eax
                movd   mm0,ds:[eax]
                movd   mm0,dword ptr ds:[eax]
        ;     movd      mm0,qword ptr ds:[eax]

                movq   mm0,qword ptr ds:[eax]
        ;     movq     mm0,dword ptr ds:[eax]
                movq   mm0,mm1
                movq   mm0,ds:[eax]

                movq   ds:[eax],mm5
                movd   dword ptr ds:[eax],mm7
                movd   ecx,mm3

                emms

                ret

MMX3DNow    endp

_text       ends

                end
```

nwsa を使う利点

Visual C++ では、C や C++ のソースファイル中に _asm 構文によってインラインアセンブラが使用できますが、単体のアセンブラが使用したくなるケースもあります。例えば、構造体をインラインアセンブラで使用することはできませんが、単体のアセンブラでは 、struc, ends 構文にて使用できます。

また、masm に比べると tasm と同様に構造体のメンバ指定が単純になっています。

例えば、

```
Sperson         struc

SPName          db      16 dup (?)
SPAge           db      ?
SPSex           db      ?
SPAddress       db      16 dup (?)
SPTel           db      8 dup(?)

Sperson                 ends


PersonData      Sperson         <>
```

に対し、

```
1.      mov     cl,PersonData.SPAge

2.      mov     esi,offset PersonData
        mov     cl,ds:[esi].SPAge
```

のいずれでも記述できます。**MASM** では後の方の記述は不可能で、

3.　　mov　　esi,offset PersonData
　　mov　　cl,ds:[esi].SPerson.SPAge

など、構造体のメンバを指定するのにその都度、構造体の型名を指定する必要があります。この仕様はメンバの名前空間を構造体内部にとどめることが出来ますが、アセンブラではアドレス指定にさまざまなケースが考えられ、この仕様では経験的に記述が面倒になります。

現在の重大な制限事項

1. struc,ends による構造体型を定義時に、メンバのデフォルト初期化値が無視される。
2. 構造体型の変数の定義時、オペランドが完全に無視され、何を書いてもひとつ分の構造体が埋め込まれる。
3. extrn 宣言において構造体が使用できない。
4. db,dw,dd,df,dq 等において繰り返し回数の大きい dup 演算子を使うとアセンブル速度が極端に遅くなる。
5. end 文において start ラベルが指定できない。

著作権について

MASM は単独では現在でも3万5000円程度で販売されています。

MASM(ML.EXE) が Visual Studio 製品を持っているかどうかにかかわらず無料でダウンロードできるようになっているサイトもあるようですが、違法です。このようなサイトから偶然、ML.EXE というファイルをダウンロードしてしまった場合、速やかにハードディスク等から削除して使用しないようにして下さい。そのまま使用しつづけることも違法行為です。また、もしこのようなサイトを見つけた場合は、"**コンピュータソフトウェア著作権協会**"等に連絡してみてください。

ただし、MASM は、Visual Studio 製品を持っている場合は、Visual Studio Processor Pack という名称で無料で Microsoft サイトからダウンロードできます。

一方、NWSA はアセンブラの文法が MASM に準拠しているのみで、プログラム自体は0から作成されたものです。したがって著作権は完全に NOWSMARTSOFT にあります。よっ

て、NWSA は、NOWSMARTSOFT が自由に配布や販売する権利を有します。この点を誤解無きようお願いします。

サポート事項

・整数命令のサポート：
次に列挙するものを除く 8086,80186,80286,80386,80486,Pentium,PentiumPro の全 MPU 命令をサポートしている。
- LODS mem , STOS mem , MOVS mem,mem , CMPS mem,mem , SCAS mem
  INS mem,dx , OUTS dx,mem
  - ただし、lodsb,lodsw,lodsd 等は使用できる。
- 現在のところ jmp far ptr , call far ptr でセグメントが埋めこまれない

・486 で追加された全命令をサポートしている。
・Pentium で追加された全命令をサポートしている。
・PentiumPro で追加された全命令をサポートしている。
・MMX 命令を全命令サポートしている。cf. .MMX 擬似命令, .NOMMX 擬似命令
・3DNow! 命令を全命令サポートしている。cf. .3DNOW 擬似命令, .NO3DNOW 擬似命令

・浮動小数点命令のサポート：8087,80287,80387 の命令および、PentiumPro で
追加された全浮動小数点命令をサポートしている。

```
;一行コメントを利用できる
";" コメント
;複数行にわたるコメントを記述できる
COMMENT <Mark>･･･<Mark>

;条件に応じてアセンブル動作を変更可能な構造化 IF 擬似命令が使用できる
IF          <const>
IFE         <const>
IFDEF       <name>
IFNDEF      <name>
IFB         <<x>>
IFNB        <<x>>
```

```
IFIDN          <<x>>,<<y>>
IFDIF          <<x>>,<<y>>
IFIDNI         <<x>>,<<y>>
IFDIFI         <<x>>,<<y>>
IF1
IF2
ELSEIF         <const>
ELSEIFE        <const>
ELSEIFDEF      <name>
ELSEIFNDEF     <name>
ELSEIFB        <<x>>
ELSEIFNB       <<x>>
ELSEIFIDN      <<x>>,<<y>>
ELSEIFDIF      <<x>>,<<y>>
ELSEIFIDNI     <<x>>,<<y>>
ELSEIFDIFI     <<x>>,<<y>>
ELSEIF1
ELSEIF2
ENDIF
ELSE
```

- 四則演算、論理演算、ビット演算の他、式中で多彩な演算子を使用できる。
- () によって優先順位を変更できる
- 分岐命令において short と明示したものはエラーになるか、short ラベルしか生成しない。
- 分岐命令において near と明示したものはエラーになるか、near ラベルしか生成しない。
- 分岐命令において far と明示したものはエラーになるか、far ラベルしか生成しない。
- 分岐命令において距離を明示せず、かつ near ラベルを使用していれば、near または short が自動生成される。
- 分岐命令において距離を明示せず、かつ far ラベルを使用していれば、必ず far 命令が生成される。

ソースファイルの任意の位置でアセンブルを中断できる

```
END
```

任意のオフセットからコードを展開できる
ORG <オフセット>

アラインメントを調整できる
EVEN
ALIGN <アラインメント>

セグメントレジスタに入っている値をアセンブラに知らせる事が出来る
ASSUME <セグメントレジスタ>":"(<セグメント名>|<グループ名>)
　　{","<セグメントレジスタ>":"(<セグメント名>|<グループ名>)}

デフォルトの基数を変更できる
.RADIX <基数>

任意の複数行を引数付きでマクロ定義出来る
・引数付マクロを利用できる
・現在のところネスト不可
・LOCAL を指定することでローカルラベルが使用可能
・EXITM を実行するとマクロ展開を中断できる
<NAME>MACRO~~ENDM

任意の複数行を繰り返す事が出来る
・ネストが可能
・＝ラベルによって変化する値を扱える
・EXITM を実行すると繰り返しを中断できる
REPT<回数>~~ENDM　　;ネスト可能

ネストして何度でもファイルを INCLUDE 出来る
INCLUDE

外部ラベルを参照できる
EXTRN
ラベルを PUBLIC 宣言できる
PUBLIC

DB
DW
DD
DF
DQ
DT

NEAR ラベルを定義できる。PROC 中ではローカルラベルになる
<NAME>:
NEAR ラベルを定義できる。PROC 中でもグローバルラベルになる
<NAME>LABEL

評価値記憶型非固定一行マクロ定義
・定義時に評価値として有効なもののみ定義できる
・何度でも定義できる
・使用される前に定義済みでなければならない
<NAME>=
トークン記憶型固定一行マクロ定義
・定義時にはトークン自体を記憶するので、任意の項目が記憶できる。
・１トランスレーション単位では一意的な値しか記憶できない。
・フェーズエラーが発生する可能性は有るが、前方定義も可能。
<NAME>EQU

マルチセグメントモデルのプログラムを開発可能
<NAME>SEGMENT
<NAME>ENDS

論理的に分かりやすいプロシージャ機能が使用可能
- プロシージャ内部で定義したラベルは -pl オプションを指定しているとローカルな名前空間に記憶され、プロシージャ外部からは見えなく出来る。これによって良く似たサブルーチンを作成するときはラベル名の変更の必要が無く、単にコピーするだけでよい。
- C言語の仕様と同様に PROC-ENDP をネストすると外側の名前空間で定義した名前をを内側から参照できる
- 上記のように PROC 内部で定義したラベルは、-pl オプションを指定しているとローカルラベルになるが、LABEL 命令で定義したラベルは、常にグローバルな名前空間に登録

され、全てのソース位置から参照できるようになる。
- -pl オプションを指定しないと、プロシージャ内部のラベルもグローバルな名前空間に登録される。
- ARG 擬似命令によって、cdecl 呼び出しの引数をサポートしている(TASM 互換)。
- ARG 擬似命令を用いずに、PROC に続けて引数を指定することも出来る(MASM 互換)。
- LOCAL 擬似命令によって、スタックローカル変数(AUTO 変数)をサポートしている。
- ARG 擬似命令や LOCAL 擬似命令は、同一行にコンマで区切って複数の変数を指定できるだけでなく、複数行に繰り返し指定することが出来る。
- ARG や LOCAL で指定された変数はプロシージャ内部の名前空間に登録される。

```
<NAME>PROC     [<NEAR|NEAR16|NEAR32|FAR|FAR16|FAR32|PROC>]
            [PUBLIC|PRIVATE]   [Arg01:type01, Arg02:type02, ...]
        ARG       Arg11:type11, Arg12:type12, ...
        ARG       Arg21:type21, Arg22:type22, ...
        LOCAL     Work31:type31, Work32:type32, ...
        LOCAL     Work41:type41, Work42:type42, ...

<NAME>ENDP

<NAME>STRUC
<NAME>ENDS

.ERR
.ERRNZ         <const>
.ERRE          <const>
.ERRDEF        <name>
.ERRNDEF       <name>
.ERRB          <<x>>
.ERRNB         <<x>>
.ERRIDN        <<x>>,<<y>>
.ERRDIF        <<x>>,<<y>>
.ERRIDNI       <<x>>,<<y>>
.ERRDIFI       <<x>>,<<y>>
.ERR1
.ERR2
```

```
.8086
.186
.286
.386                //386 の命令を使用できるようになる
.486                //486 の命令を使用できるようになる
.586
.286P
.386P
.486P
.586P
.Pentium              //.586 と同様の意味だが、PRIVILEGE かどうかは変化しない
                    //Pentium で追加された命令を使用できるようになる。
.PentiumPro           //PentiumPro で追加された命令を使用できるようになる。
.PentiumII
.PentiumIII
.PentiumIV

.MMX                //MMX 命令を使用できるようにする
.NOMMX

.3DNOW              //3DNow! 命令を使用できるようにする
.NO3DNOW

.PRIVILEGE          //486P の  P と同じ意味
.NOPRIVILEGE         //PRIVILEGE の作用を打ち消す

.MODEL  <メモリモデル>[,<言語名>]

<メモリモデル>=(SMALL|COMPACT|MEDIUM|LARGE|HUGE)
<言語名>=<C,STDCALL,SYSCALL,BASIC,FORTRAN,PASCAL>

<メモリモデル>  @CODESIZE       @DATASIZE
SMALL          0              0
COMPACT        0              1
MEDIUM         1              0
LARGE          1              1
```

```
HUGE            1               2

<言語名>        宣言形式
UNKNOWN     %s
C               _%s
STDCALL       _%s
SYSCALL
BASIC
FORTRAN
PASCAL

@CODESIZE=0 :   PROC PTR        =NEAR PTR
                PROC PROC       =PROC NEAR
@CODESIZE!=0 :  PROC PTR        =FAR  PTR
                PROC PROC       =PROC FAR

;セグメントの強制配置リンカオプションを指定出来る
DOSSEG

;特殊定数
@DATASIZE       =0,1,2
@CODESIZE       =0,1

//言語指定にかかわらず外部シンボルの先頭に下線"_"を付加する。
.UNDERSCORE

//言語指定にかかわらず外部シンボルの先頭に下線"_"を付加しない。
.NOUNDERSCORE



;・TITLE 文字列は  EQU での置換が行われない
;・ただし ；以降の文字は除去される。
TITLE   <タイトル>

;・%OUT 文字列はトークンとして解釈されてから表示される。
;・任意の空白はトークンの区切りとしてのみ意味を持ち、空白は保持されない。
```

;・少なくとも片方のトークンが記号以外であるトークンが接している場合は自動
;　的に１つのスペース文字がその間に挿入される。
%OUT　<表示トークン列>

オーバーライド

<SREG>:　　--->SREG を用いる事を強制する
<SEGMNAME>:　　--->セグメントでオーバーライドする
<GROUPNAME>:　　--->グループでオーバーライドする

ラベルを含んだ評価値には
FrameReg：オーバーライド用セグメントレジスタ識別番号
　　メモリオペランドがある場合にオーバーライドするべきフレームレジスタ
FrameNo ：オーバーライド用 NAME 識別番号
　　NAME 識別番号はセグメント名、グループ名、ラベル名等、異なった名前
　　をもつ任意の種類の項目を一括して同じ領域に記憶し、区別するために
　　共通につけられた番号で、番号が分かると項目の種類とその修飾情報が
　　取得できる。例えば、実際に定義したセグメント名の識別番号とグルー
　　プ名の識別番号は必ず異なった値になる。（項目の種類が違う場合に同
　　じ番号が付けられ混同すること）はない。

を記憶する場所がそれぞれ一つづつある。

<SREG>: でオーバーライドすると、FrameReg は設定されるが、FrameNo は以前の設定されない。
<SEGNAME>:や<GROUPNAME>:でオーバーライドすると、FrameNo は設定されるが 、FrameReg
は設定されない。

FrameReg はメモリオペランドを使うときのみ指定し、混乱を招くため即値オペランドには指定すべきではない。本アセンブラでは即値オペランドに<SREG>:でオーバーライドすると警告を表示する。

```
mov　eax,ds:Label1　　;Ok
mov　eax,_DATA:Label1　　;Ok
mov　eax,offset ds:Label1　;Warning
mov　eax,offset _DATA:Label1 ;Ok
```

FreameNo が明示されないで FrameReg だけが指定された場合は、ASSUME から直接に FrameNo を決定する。

FrameReg が明示されないで FrameNo だけが指定されたメモリオペランドでは、ASSUME 指定の情報を逆にたどって、使用するフレームセグメントレジスタを決定する。FrameNo で示されたフレームがセグメントレジスタに ASSUME 割り付けされていない場合はエラーを報告する。

FReg	FNo	I/M	SEG-PREFIX	OBJ-FRAME	エラー
R_N	S_B	I	無付加	SEARCH_BAD	無
R_N	指定	I	無付加	FNo	無
指定	S_B	I	無付加	Ass[FReg]	警告
指定	指定	I	無付加	FNo	警告
R_N	S_B	M	REG_NOUSE	SEARCH_BAD	無
R_N	指定	M	IAss[FNo]	FNo	無またはエラー
指定	S_B	M	FReg	Ass[FReg]	無
指定	指定	M	FReg	FNo	無

※Ass[r] はセグメントレジスタ r に仮定されているセグメント又はグループの識別番号を返す。
※IAss[n] はセグメント又はグループの識別番号 n を仮定しているセグメントレジスタの番号を返す。条件を満たすセグメントレジスタが無い場合はエラーになる。
※SEG-PREFIX=REG_NOUSE は前置バイトを付加しないことを示し、I(即値)の無付加と同様の意味を持つ。
※OBJ-FRAME=SEARCH_BAD はラベルが存在する場所によってオーバーライドすることを示す。
※R_N は REG_NOUSE の略で FrameReg が明示的には指定されていないことを意味する。
※S_B は SEARCH_BAD の略で FrameNo が明示的には指定されていないことを意味する。
※初期値は FReg=REG_NOUSE,FNo=SEARCH_BAD である。
※ASSUME x:nothing と指定すると、Ass[x]=SEARCH_BAD になる。

例：

```
DGROUP  group   _DATA
        assume  ds:DGROUP , es:_ANOTHER
```

```
mov     eax,es:(DGROUP:Data1)
;・実際のコードは ES: バイトが付く
;・OBJ ファイルには DGROUP でオーバーライドすることを知らせる
;・標準的なリンカではグループ全体の先頭からの相対アドレスが
;  offset オペランドに埋めこまれる。
mov     eax,Data1
;・実際のコードはセグメント前置バイトは着かない。
;  ・暗黙に DS がベースアドレスになる
;・OBJ ファイルには Data1 自体によってオーバーライドすることを知らせる
;・実際に Data1 が DGROUP に属しているならば、標準的なリンカでは、
;  DGORUP: を付加したのと同じく、グループ全体の先頭アドレスからの相対
;  アドレスが offset オペランドに埋めこまれる。
mov     eax,es:Data1
;・実際のコードは ES: バイトが付く
;・OBJ ファイルには _ANOTHER によってオーバーライドすることを知らせる
;・標準的なリンカではグループ全体の先頭からの相対アドレスが
;  offset オペランドに埋めこまれる。

mov     eax,offset _DATA:Data1
;・OBJ ファイルには _DATA セグメントでオーバーライドすることを知らせる
;・標準的なリンカではグループの途中のセグメントの先頭からの相対アド
;  レスが即値オペランドに埋めこまれる。
mov     eax,offset DGROUP:Data1
;・OBJ ファイルには DGROUP でオーバーライドすることを知らせる
;・標準的なリンカではグループの先頭からの相対アドレスが即値オペラ
;  ンドに埋めこまれる。
mov     eax,offset Data1
;・OBJ ファイルには Data1 自体によってオーバーライドすることを知らせる
;・標準的なリンカでは DGROUP の先頭からの相対アドレスが即値オペラ
;  ンドに埋めこまれる。
```

非サポートまたは制限事項

・構造体の初期化が出来ない
・構造体のデフォルト値の意味が無い

・extrn の型として構造体型を指定できない
・END 文に先頭番地を指定しても現在、正しく OBJ に書かれない
・REPT はネストできるが、MACRO はネスト出来ない
　・両者の混在は現在のところサポートしていない（動作不定）
・IRP,IRPC をサポートしていない
・マクロ展開時、EQU で置換後に EXITM が出現した場合、EXITM は働かない。
・PURGE をサポートしていない
・DB,DW,DD,DF,DQ,DT のサポートが甘い
　・現在、dd や df で far ptr を埋めこめない。
・浮動小数点定数が使えない
・簡易化セグメント擬似命令が使えない
・COMM をサポートしていない
・PUBLIC や EXTRN で AS による名前の付け替えが出来ない
・絶対値を PUBLIC 宣言できない
・絶対値を EXTRN で扱えない
・RECORD 文をサポートしていない
・LOCK 命令で LOCK出来る命令を厳密にはチェックしていない。
　・本来は LOCK化される命令がメモリオペランドを取っていることを調べる必要が有る。
・PAGE,OPTION,NAME,SUBTITLE をサポートしていない
・.XLIST,.LIST,.LFCOND,.SFCOND,.TFCOND,.LALL,.SALL,.XALL,.CREF,.XCREF をサポートしていない
・デバッグ情報は、COFF 形式の OBJ ファイルにのみ挿入できる。OMF 形式の OBJ ファイルには挿入できない。
・　シンボルは無制限に登録可能だが、proc, endp で定義する関数の最大値は 65535 個に制限されている。
・　imm8 オペランドを自動生成しない。
・速度的な「詰めの作業」が全く行われていない。
・今のところ最低でも２パス必要なので速度的に多少問題が有る
---> 以下のような技術を用いて１パスで終えるようにすべきである
　　・jmp 命令オペランドのバックパッチによる埋め込み
　　・jmp 命令のサイズのバックパッチ的な修正

将来

このプログラムのソースを希望があれば、有料（１０万円程度）で公開しようと思って

います。

連絡先

本アセンブラに関してのご質問は、次の E-MAIL アドレスにお願いします。前向きなご意見やご要望も承っておりますので是非お気軽にメイルください。
また、専用のホームページにて会議室(BBS)を開いております。お気軽にご意見をお書き込み下さい。

E-Mail:
LightCone@nifty.ne.jp (KHF12570@nifty.ne.jp と同一)

ホームページ URL:
http://homepage2.nifty.com/nowsmart/

Update 情報

2001/02/23 NWSA140.LZH

修正項目	原因	対処
PROC の引数を、ARG 擬似命令を用いず直接指定できるようにもし、MASM の書式に対応した。	関数の引数は、PROC の次の行から、ARG 擬似命令を用いて指定する方法(TASM 仕様)のみに対応していた。	PROC 擬似命令の NEAR や PUBLIC 指定に引き続き、引数を指定できるようにした。

2001/02/23 NWSA130.LZH

修正項目	原因	対処
PROC～ENDP 内部で LOCAL 変数(AUTO 変数)が使用できるようになった。	LOCAL 擬似命令をサポートしていなかった。	LOCAL 擬似命令をサポートした。

PROC～ENDP 内部で ARG や LOCAL を指定しなかった場合はスタックフレームを生成しないようにした。スタックフレームとは、
push ebp
mov ebp,esp
sub esp,8
～
mov esp,ebp
pop ebp
のようなコード。
	PROC～ENDP では無条件にスタックフレームを作成していた。	ARG や LOCAL が使用されているかどうかによってスタックフレームを作成するかどうかを自動判別するようにした。

PUBLIC や EXTRN で外部に公開されるシンボルの先頭に下線を付加するかどうかを、個別に指定できるようにした。	.MODEL 命令で言語を C や STDCALL に指定したときは必ず下線を付加していた。	.UNDERSCORE 擬似命令と .NOUNDERSCORE 擬似命令を追加し、強制指示できるようにした。

2001/01/21 NWSA120.LZH

修正項目	原因	対処
シンボルの最大定義数を固定ではなく、無制限にした。	シンボルの格納方法が固定型だった。	シンボルの格納方法を可変型にした。

2001/02/04 NWSA111.LZH

修正項目	原因	対処
mov eax,dword ptr -1 等としたときに、-1 が文法エラーになってしまうバグを修正。	???? ptr の後につく文法要素を文法定義段階でミスしていた。	文法要素を正しく修正。
フェーズエラーが何パスにも渡り延々と生じるバグを修正。	ラベルフェーズエラーが生じたとき、それ以後のラベルフェーズエラーを事前に予測し、ラベルの値を自動修正するプログラムにおいて、負の値の加算値に対して動作がおかしくなっていた。	変数を符号付きから符号無しへ修正。
セグメントオーバーライドを明示しなかった場合に、ASSUME 指定の内容から自動的にセグメントレジスタを選び出すアルゴリズムにおいて、選び出す優先順位が適切でなかたった部分を修正。	セグメントレジスタの選択アルゴリズムにおいて、レジスタの命令イメージの順に探していた。	スタック以外では、 DS,CS,SS,ES,FS,GS スタックでは、 SS,DS,CS,ES,FS,GS の順に優先順位を付けて探すようにした。

2001/01/30 NWSA110_2.LZH

ドキュメントを TXT 形式からワード形式に変更して整理。追加。

2001/01/28 NWSA110.LZH

NOWSMART ASSEMBLER VER 1.10 としてあらためて公開。

ファイルネームを SASM.EXE から NWSA.EXE に変更。

COFF 形式に対応。

COFF 形式においては、行番号デバッグ情報を埋め込めるようにした。

同じセグメントの offset アドレスを書き込むときのリロケーション情報の誤りを是正。

SAMPLE.ASM を書いておいた。

STARTUP.TXT を書いておいた。

2001/01/23 SASM100.LZH

SMART ASSEMBLER VER 1.00 としてまだ非公開のホームページに載せてみた。

ファイルネームは SASM.EXE

COFF 形式には全く未対応。