The Telemark Assembler (TASM) User Manual

Version 3.2

Thomas N. Anderson
Squak Valley Software
837 Front Street South
Issaquah, WA 98027
email: tnetherly@comcast.net
http://www.tasmhome.com

TABLE OF CONTENTS

INTRODUCTION

The Telemark Assembler (TASM) is a table driven cross assembler for the MS-DOS and LINUX environments. Assembly source code, written in the appropriate dialect (generally very close to the manufacturers assembly language), can be assembled with TASM, and the resulting object code transferred to the target microprocessor system via PROM or other mechanisms.

The microprocessor families supported by TASM are:

- 6502
- 6800/6801/68HC11
- 6805
- 8048
- 8051
- 8080/8085, Z80

- TMS32010, TMS320C25
- TMS7000
- 8096/80196

The user so inclined may build tables for other microprocessors. The descriptions of the various existing tables and instructions on building new tables are not in this document but can be found in the TASMTABS.HTM file on the TASM distribution disk.

TASM characteristics include:

1. Powerful expression parsing (17 operators).
2. Supports a subset of the 'C' preprocessor commands.
3. Macro capability (through use of DEFINE directive).
4. Multiple statements per line.
5. Four object file formats: Intel hex, MOS Technology hex, Motorola hex, binary.
6. Absolute code generation only.
7. Source code available (in C).
8. Uniform syntax across versions for different target machines.
9. Features in support of PROM programming (preset memory, contiguous block).
10. Supports extended instructions for many of the supported microprocessor families.
11. Tables read at run time - single TASM executable for all table versions.
12. Symbol table export for inclusion in subsequent assemblies.
13. Symbol table export file for import with some simulator products.

SHAREWARE

TASM is distributed as shareware. TASM is not in the public domain. The TASM distribution files may be freely copied (excluding the source code files) and freely used for the purpose of evaluating the suitability of TASM for a given purpose. Use of TASM beyond a reasonable evaluation period requires registration. Prolonged use without registration is unethical.

INVOCATION

TASM can be invoked as follows (optional fields shown in brackets, symbolic fields in italics):

```
tasm -pn [-options ...] src_file [obj_file [lst_file [exp_file [sym_file]]]]
```

Where *options* can be one or more of the following:

-*table*	Specify version (*table* = table designation)
-t*table*	Table (alternate form of above)
-a*amask*	Assembly control (optional error checking)
-b	Produce object in binary (.COM) format
-c	Object file written as a contiguous block
-d*macro*	Define a macro (or just a macro label)
-e	Show source lines after macro expansion

-f*fillbyte*	Fill entire memory space with fillbyte (hex)
-g*objtype*	Object file (0=Intel Hex, 1=MOS Tech, 2=Motorola, 3=binary,4=Intel Hex (Word))
-h	Produce hex table of the assembled code (in list file)
-i	Ignore case for labels
-l[al]	Produce a label table in the listing
-m	Produce object in MOS Technology format
-o*obytes*	Bytes per object record (for hex obj formats)
-p[*lines*]	Page the listing file (lines per page. default=60)
-q	Quiet, disable the listing file
-s	Write a symbol table file
-x[*xmask*]	Enable extended instruction set (if any)
-y	Time the assembly

The filename parameters are defined as follows:

src_file	Source file name
obj_file	Object code file name
lst_file	Listing file name
exp_file	Symbol export file (only if the EXPORT directive is used).
sym_file	Symbol table file (only if the **-s** option or the SYM/AVSYM directives are used).

The source file must be specified. If not, some usage information is displayed. Default file names for all the other files are generated if they are not explicitly provided. The filename is formed by taking the source filename and changing the extension to one of the following:

Extension	**File type**
.OBJ	Object file
.LST	Listing file
.EXP	Symbol export file
.SYM	Symbol table file

TASM has no built-in instruction set tables. Instruction set definition files are read at run time. TASM determines which table to use based on the '*-table'* field shown above. For example, to assemble the code in a file called *source.asm*, one would enter

`tasm -48 source.asm`	for an 8048 assembly
`tasm -65 source.asm`	for a 6502 assembly
`tasm -51 source.asm`	for an 8051 assembly.
`tasm -85 source.asm`	for an 8085 assembly.
`tasm -80 source.asm`	for a Z80 assembly.

`tasm -05 source.asm`	for a 6805 assembly.
`tasm -68 source.asm`	for a 6800/6801/68HC11 assembly.
`tasm -70 source.asm`	for a TMS7000 assembly.
`tasm -3210 source.asm`	for a TMS32010 assembly.
`tasm -3225 source.asm`	for a TMS320C25 assembly.
`tasm -96 source.asm`	for a 8096/80196 assembly

Tables are read from a file named by taking the digits specified after the '-' and appending it to 'TASM' then appending the '.TAB' extension. Thus, the **-48** flag would cause the tables to be read from the file 'TASM48.TAB'.

It is possible to designate tables by non numeric part numbers if the **-t** flag is used. For example, if a user built a table called TASMF8.TAB then TASM could be invoked as follows:

```
tasm -tf8 source.asm
```

Each option flag must be preceded by a dash. Options need not precede the file names. The various options are described in the sections that follow.

a - Assembly Control

TASM can provide additional error checking by specifying the *-a* option at the time of execution. If the *-a* is provided without a digit following, then all the available error checking is done. If a digit follows, then it is used as a mask to determine the error checks to be made. The bits of the mask are defined as follows:

Bit	Option	Default	Description
0	-a1	OFF	Check for apparent illegal use of indirection
1	-a2	ON	Check for unused data in the arguments
2	-a4	ON	Check for duplicate labels
3	-a8	OFF	Check for non-unary operators at start of expression.

Combinations of the above bits can also be used. For example, *-a5* would enable the checking for illegal indirection and duplicate labels.

Illegal indirection applies to micros that use parenthesis around an argument to indicate indirection. Since it is always legal to put an extra pair of parenthesis around any expression (as far as the expression parser is concerned), the user may think that he/she is indicating indirection for an instruction that has no indirection and TASM would not complain. Enabling this checking will result in an error message (warning) whenever an outer pair of parenthesis is used and the instruction set definition table does not explicitly indicate that to be a valid form of addressing.

Unused data in arguments applies to cases where a single byte of data is needed from an argument, but the argument contains more than one byte of data. If a full sixteen bit address is used in a 'Load Immediate' type instruction that needs only a single byte, for example, an error message would be generated. Here is an example (6502 code):

```
0001   1234                    .org $1234
test.asm line 0002: Unused data in MS byte of argument.
0002   1234 A9 34       start  lda  #start
```

To make the above checks occur whenever you do an assembly, add a line like this to your AUTOEXEC.BAT file:

```
SET TASMOPTS=-a
```

b - Binary Object Format

This option causes the object file to be written in binary - one byte for each byte of code/data. Note that no address information is included in the object file in this format. The contiguous block (-c) output mode is forced when this option is invoked. This flag is equivalent to *-g3*.

c - Contiguous Block Output

If this option is specified, then all bytes in the range from the lowest used byte to the highest will be defined in the object file. Normally, with the default Intel Hex object format enabled, if the Program Counter (PC) jumps forward because of an .ORG directive, the bytes skipped over will not have any value assigned them in the object file. With this option enabled, no output to the object file occurs until the end of the assembly at which time the whole block is written. This is useful when using TASM to generate code that will be put into a PROM so that all locations will have a known value. This option is often used in conjunction with the -f option to ensure all unused bytes will have a known value.

d - Define a Macro

Macros are defined on the command line generally to control the assembly of various IFDEF's that are in the source file. This is a convenient way to generate various versions of object code from a single source file.

e - Expand Source.

Normally TASM shows lines in the listing file just as they are in the source file. If macros are in use (via the DEFINE directive) it is sometimes desirable to see the source lines after expansion. Use the '-e' flag to accomplish this.

f - Fill Memory.

This option causes the memory image that TASM maintains to be initialized to the value specified by the two hex characters immediately following the 'f'. TASM maintains a memory image that is a full 64K bytes in size (even if the target processor cannot utilize that memory space). Invocation of this option introduces a delay at start up time.

g - Object File Format.

TASM can generate object code in the formats indicated below:

Option	Description
-g0	Intel hex (default)
-g1	MOS Technology hex (same as **-m**)
-g2	Motorola hex
-g3	binary (same as **-b**)
-g4	Intel hex with word addresses

The **-m** and **-b** flags may also be used, as indicated above. If both are used the right-most option on the command line will be obeyed.

See the section on **OBJECT FILE FORMATS** for descriptions of each of the above.

h - Hex Object Code Table.

This option causes a hex table of the produced object code to appear in the listing file. Each line of the table shows sixteen bytes of code.

i - Ignore Case in Labels.

TASM is normally case sensitive when dealing with labels. For those that prefer case insensitivity, the '-i' command line option can be employed.

l - Label Table.

This option causes a label table to appear in the listing file. Each label is shown with its corresponding value. Macro labels (as established via the DEFINE directives) do not appear.

Two optional suffixes may follow the **-l** option:

Suffix	Description
l	Use long form listing
a	Show all labels (including local labels)

The suffix should immediately follow the '-l'. Here are some examples:

`-l`	to show non-local labels in the short form
`-la`	to show all labels in the short form
`-ll`	to show non-local labels in the long form
`-lal`	to show all labels in the long form

m - MOS Technology Object Format.

This option causes the object file to be written in MOS Technology hex format rather than the default Intel hex format. See section on OBJECT FILE FORMATS for a description of the format.

o - Set Number of Bytes per Object Record.

When generating object code in either the MOS Technology format or the Intel hex format, a default of 24 (decimal) bytes of object are defined on each record. This can be altered by invoking the '-o' option immediately followed by two hex digits defining the number of bytes per record desired. For example, if 32 bytes per record are desired, one might invoke TASM as:

```
tasm -48 -o20 source.asm
```

p - Page Listing File.

This option causes the listing file to have top of page headers and form feeds inserted at appropriate intervals (every sixty lines of output). To override the default of sixty lines per page, indicate the desired number of lines per page as a decimal number immediately following the '-p'. Here is an example:

```
tasm -48 -p56 source.asm
```

q - Disable Listing File.

This option causes all output to the listing file to be suppressed, unless a .LIST directive is encountered in the source file (see LIST/NOLIST directives).

s - Enable Symbol File Generation.

If this flag is set, a symbol file is generated at the end of the assembly. The format of the file is one line per label, each label starts in the first column and is followed by white space and then four hexadecimal digits representing the value of the label. The following illustrates the format:

```
label1          FFFE
label2          FFFF
label3          1000
```

The symbol file name can be provided as the fifth file name on the command line, or the name will be generated from the source file name with a '.SYM' extension. The symbol table file can also be generated by invoking the SYM directive. The AVSYM directive also generates the symbol file but in a different format (see section on ASSEMBLER DIRECTIVES).

t - Table Name.

As an alternative to specifying the instruction set table as two decimal digits, the table indication may be proceeded by the '-t' option. This is useful if the desired table name starts with a non-numeric. Thus, a table for an F8 might be selected as:

```
tasm -tf8 source.asm
```

TASM would expect to read the instruction set definition tables from a file named TASMF8.TAB.

x - Enable Extended Instruction Set.

If a processor family has instructions that are valid for only certain members, this option can be used to enable those beyond the basic standard instruction set. A hex digit may follow the 'x' to indicate a mask value used in selecting the appropriate instruction set. Bit 0 of the mask selects the basic instruction set, thus a '-x1' would have no effect. A '-x3' would enable the basic set plus whatever instructions have bit 1 set in their class mask. A '-x' without a digit following is equivalent to a '-xf' which sets all four of the mask bits. The following table indicates the current extended instruction sets available in the TASM tables:

Base Table	Base Family	Ext 1 (-x3)	Ext 2 (-x7)	Ext 3 (-x5)	Ext 4 (-x9)
48	8048	8041A		8022	8021
65	6502	R65C02		R65C00/21	
05	6805	M146805 CMOS		HC05C4	
80	Z80	HD64180			
68	6800	6801/6803	68HC11		
51	8051				
85	8080				
3210	TMS32010				
3225	TMS320C25			TMS320C26	
70	TMS7000				

The above table does not attempt to show the many microprocessor family members that may apply under a given column.

See the TASMTABS.HTM on-line document for details on each specific table.

y - Enable Assembly Timing

If this option is enabled TASM will generate a statement of elapsed time and assembled lines per second at the end of the assembly.

ENVIRONMENT VARIABLES

The TASM environment can be customized by using the environment variables listed below:

TASMTABS

The TASMTABS variable specifies the path to be searched for TASM instruction set definition tables. If it is not defined then the table(s) must exist in the current working directory. The following examples illustrate possible usage:

For MSDOS `set TASMTABS=C:\TASM`
For LINUX `TASMTABS=/tasm`

TASMOPTS

This variable specifies TASM command line options that are to be invoked every time TASM is executed. For example, if TASM is being used for 8048 assemblies with binary object file output desired, the following statement would be appropriate in the AUTOEXEC.BAT file:

```
set TASMOPTS=-48 -b
```

TASMERRFORMAT

The TASMERRFORMAT variable provides user control of the format of error messages. The format string must be a valid printf format string for ANSI C. The default value is:

```
"%s line %04d: %s %s"
```

Which provides error messages like this:

```
Main.asm line 1234: No such label: Start
```

The four fields associated with an error message are:

Field Description	Associated Data Type (ANSI C)
File name	char *
Line number within file	int

Error description	char *
Error data (optional)	char *

No user control of the order of the error message fields is provided.
Here are sample usages:

For MSDOS:

```
set TASMERRFORMAT="%s(%d) %s %s"
```

For LINUX:

```
TASMERRFORMAT="%s(%d) %s %s"
```

EXIT CODES

When TASM terminates, it will return to the OS the following exit codes:

Exit Code	Definition
0	Normal completion, no assembly errors
1	Normal completion, with assembly errors
2	Abnormal completion, insufficient memory
3	Abnormal completion, file access error
4	Abnormal completion, general error

Exit codes 2 and above will also be accompanied by messages to the console concerning the error.

SOURCE FILE FORMAT

Statements in the source file must conform to a format as follows (except for assembler directive statements which are described in a subsequent section):

label operation operand comment

All of the fields are optional, under appropriate circumstances. An arbitrary amount of white space (space and tabs) can separate each field (as long as the maximum line length of 255 characters is not exceeded). Each of the fields are described in the following sections.

Label Field.

If the first character of the line is alphabetic, it is assumed to be the start of a label. Subsequent characters are accepted as part of that label until a space, tab, or ':' is encountered. The assembler assigns a value to the label corresponding to the current location counter. Labels can be a maximum of 32 characters long. Labels can contain upper and lower case letters, digits, underscores, and periods (the first character must be alphabetic). Labels are case sensitive - the label 'START' is a different label from 'start' - unless the '-i' (ignore case) option is enabled.

Operation Field.

The operation field contains an instruction mnemonic which specifies the action to be carried out by the target processor when this instruction is executed. The interpretation of each mnemonic is dependent on the target microprocessor (as indicated by the selected TASM table). The operation field may begin in any column except the first. The operation field is case insensitive.

Operand Field.

The operand field specifies the data to be operated on by the instruction. It may include expressions and/or special symbols describing the addressing mode to be used. The actual format and interpretation is dependent on the target processor. For a description of the format for currently supported processors, see the TASMTABS.DOC file on the TASM distribution disk.

Comment Field.

The comment field always begins with a semicolon. The rest of the line from the semicolon to the end of the line is ignored by TASM, but passed on to the listing file for annotation purposes. The comment field must be the last field on a line, but it may be the only field, starting in column one, if desired.

Multiple Statement Lines.

If the backslash character is encountered on a source line, it is treated as a newline. The remainder of the line following the backslash will be processed as an independent line of source code. This allows one to put multiple statements on a line. This facility is not so useful of itself, but when coupled with the capability of the DEFINE directive, powerful multiple statement macros can be constructed (see section on **ASSEMBLER DIRECTIVES**). Note that when using the statement separator, the character immediately following it should be considered the first character of a new line, and thus must either be a start of a label or white space (not an instruction). As the examples show, a space is put between the backslash and the start of the next instruction.

Sample Source Listing.

Some examples of valid source statements follow (6502 mnemonics shown):

```
  lab1       lda    byte1   ;get the first byte
             dec    byte1
             jne    label1
  ;
  lab2       sta    byte2,X
  ;  a multiple statement line follows
             lda    byte1\ sta byte1+4\ lda byte2\ sta byte2+4
```

EXPRESSIONS

Expressions are made up of various syntactic elements combined according to a set of syntactical rules. Expressions can be comprised of the following elements:

- Labels
- Constants
- Location Counter Symbol

- Operators
- Parenthesis

Labels

Labels are strings of characters that have a numeric value associated with them, generally representing an address. Labels can contain upper and lower case letters, digits, underscores, and periods. The first character must be a letter or the local label prefix (default '_'). The value of a label is limited to 32 bit precision. Labels can contain up to 32 characters, all of which are significant (none are ignored when looking at a label's value, as in some assemblers). Case is significant unless the '-i' command line option is invoked.

Local labels must only be unique within the scope of the current module. Modules are defined with the MODULE directive. Here is an example:

```
        .MODULE xxx
        lda regx
        jne _skip
        dec
_skip   rts
        .MODULE yyy
        lda regy
        jne _skip
        dec
_skip   rts
```

In the above example, the *_skip* label is reused without harm. As a default, local labels are not shown in the label table listing (resulting from the '-l' command line option). See also sections on MODULE and LOCALLABELCHAR directives.

Numeric Constants

Numeric constants must always begin with a decimal digit (thus hexadecimal constants that start with a letter must be prefixed by a '0' unless the '$' prefix is used). The radix is determined by a letter immediately following the digit string according to the following table:

Radix	Suffix	Prefix
2	B or b	%
8	O or o	@
10	D or d (or nothing)	
16	H or h	$

Decimal is the default radix, so decimal constants need no suffix or prefix.

The following representations are equivalent:

```
1234H          or     $1234
100d           or     100
177400o        or     @177400
01011000b      or     %01011000
```

The prefixes are provided for compatibility with some other source code formats but introduce a problem of ambiguity. Both '%' and '$' have alternate uses ('%' for modulo, '$' for location counter symbol). The ambiguity is resolved by examining the context. The '%' character is interpreted as the modulo operator only if it is in a position

suitable for a binary operator. Similarly, if the first character following a '$' is a valid hexadecimal digit, it is assumed to be a radix specifier and not the location counter.

Character Constants

Character constants are single characters surrounded by single quotes. The ASCII value of the character in the quotes is returned. No escape provision exists to represent non-printable characters within the quotes, but this is not necessary since these can be just as easily represented as numeric constants (or using the TEXT directive which does allow escapes).

String Constants.

String constants are one or more characters surrounded by double quotes. Note that string constants are not allowed in expressions. They are only allowable following the TITLE, BYTE, DB, and TEXT assembler directives. The quoted strings may also contain escape sequences to put in unprintable values. The following escape sequences are supported:

Escape Sequence	Description
\n	Line Feed
\r	Carriage return
\b	Backspace
\t	Tab
\f	Formfeed
\\	Backslash
\"	Quote
\000	Octal value of character

Location Counter Symbol

The current value of the location counter (PC) can be used in expressions by placing a '$' in the desired place. The Location Counter Symbol is allowable anywhere a numeric constant is. (Note that if the '$' is followed by a decimal digit then it is taken to be the hexadecimal radix indicator instead of the Location Counter symbol, as mentioned above). The '*' may also be used to represent the location counter, but is less preferred because of its ambiguity with the multiplicative operator.

Operators

Expressions can optionally contain operators to perform some alterations or calculations on particular values. The operators are summarized as follows:

Operator	Type	Description
+	Additive	addition
-		subtraction
*	Multiplicative	multiplication
/		division
%		modulo

<<		logical shift left
>>		logical shift right
~	Unary	bit inversion (one's complement)
-		unary negation
=	Relational	equal
==		equal
!=		not equal
<		less than
>		greater than
<=		less than or equal
>=		greater than or equal
&	Binary	binary 'and'
|		binary 'or'
^		binary 'exclusive or'

The syntax is much the same as in ANSI C with the following notes:

1. No operator precedence is in effect. Evaluation is from left to right unless grouped by parenthesis (see example below).
2. All evaluations are done with 32 bit signed precision.
3. Both '=' and '==' are allowable equality checkers. This is allowed since the syntax does not provide assignment capability (as '=' would normally imply).

The relational operators return a value of 1 if the relation is true and 0 if it is false. Thirty-two bit signed arithmetic is used.

It is always a good idea to explicitly indicate the desired order of evaluation with parenthesis, especially to maintain portability since TASM does not evaluate expressions in the same manner as many other assemblers. To understand how it does arrive at the values for expressions, consider the following example:

```
1 + 2*3 + 4
```

TASM would evaluate this as:

```
(((1 + 2) * 3) + 4) = 13
```

Typical rules of precedence would cause the (2*3) to be evaluated first, such as:

```
1 + (2*3) + 4       = 11
```

To make sure you get the desired order of evaluation, use parenthesis liberally. Here are some examples of valid expressions:

```
(0f800H + tab)
(label_2 >> 8)
(label_3 << 8) & $f000
$ + 4
010010000100100b + 'a'
(base + ((label_4 >> 5) & (mask << 2))
```

ASSEMBLER DIRECTIVES

Most of the assembler directives have a format similar to the machine instruction format. However, instead of specifying operations for the processor to carry out, the directives cause the assembler to perform some function related to the assembly process. TASM has two types of assembler directives - those that mimic the 'C' preprocessor functions, and those that resemble the more traditional assembler directive functions. Each of these will be discussed.

The ANSI C preprocessor style directives are invoked with a '#' as the first character of the line followed by the appropriate directive (just as in 'C'). Thus, these directives cannot have a label preceding them (on the same line). Note that in the examples directives are shown in upper case, however, either upper or lower case is acceptable.

ADDINSTR

The ADDINSTR directive can be used to define additional instructions for TASM to use in this assembly. The format is:

```
    [label] .ADDINSTR inst args opcode nbytes rule class shift binor
```

The fields are separated by white space just as they would appear in an instruction definition file. See the TASMTABS.HTM file on the TASM distribution disk for more detail.

AVSYM

See SYM/AVSYM.

BLOCK

The BLOCK directive causes the Instruction Pointer to advance the specified number of bytes without assigning values to the skipped over locations. The format is:

```
    [label] .BLOCK        expr
```

Some valid examples are:

```
    word1   .BLOCK     2
    byte1   .block     1
    buffer  .block     80
```

BSEG/CSEG/DSEG/NSEG/XSEG

These directives can be invoked to indicate the appropriate address space for symbols and labels defined in the subsequent code. The invocation of these directives in no way affects the code generated, only provides more information in the symbol table file if the AVSYM directive is employed. Segment control directives such as these are generally supported by assemblers that generate relocatable object code. TASM does not generate relocatable object code and does not support a link phase, so these directives have no direct effect on the resulting object code. The segments are defined as follows:

Directive	Segment Description
BSEG	Bit address
CSEG	Code address
DSEG	Data address (internal RAM)
NSEG	Number or constant (EQU)
XSEG	External data address (external RAM)

BYTE

The BYTE directive allows a value assignment to the byte pointed to by the current Instruction Pointer. The format is:

```
[label] .BYTE   expr [, expr ...]
```

Only the lower eight bits of *expr* are used. Multiple bytes may be assigned by separating them with commas or (for printable strings) enclosed in double quotes. Here are some examples:

```
label1   .BYTE     10010110B
         .byte     'a'
         .byte     0
         .byte     100010110b,'a',0
         .byte     "Hello", 10, 13, "World"
```

CHK

The CHK directive causes a checksum to be computed and deposited at the current location. The starting point of the checksum calculation is indicated as an argument. Here is the format:

```
[label]    .CHK    starting_addr
```

Here is an example:

```
start: NOP
       LDA #1
       .CHK start
```

The checksum is calculated as the simple arithmetic sum of all bytes starting at the *starting_addr* up to but not including the address of the CHK directive. The least significant byte is all that is used.

CODES/NOCODES

The CODES/NOCODES directives can be used to alternately turn on or off the generation of formatted listing output with line numbers, opcodes, data, etc. With NOCODES in effect, the source lines are sent to the listing file untouched. This is useful around blocks of comments that need a full 80 columns of width for clarity.

DB

This is alternate form of the BYTE directive.

DW

This is alternate form of the WORD directive.

DEFINE

The DEFINE directive is one of the most powerful of the directives and allows string substitution with optional arguments (macros). The format is as follows:

```
#DEFINE  macro_label[(arg_list)]  [macro_definition]
```

Where:

macro_label
: character string to be expanded when found in the source file

arg_list
: optional argument list for variable substitution in macro expansion

macro_def
: string to replace the occurrences of *macro_label* in the source file.

The simplest form of the DEFINE directive might look like this:

```
#DEFINE          MLABEL
```

Notice that no substitutionary string is specified. The purpose of a statement like this would typically be to define a label for the purpose of controlling some subsequent conditional assembly (IFDEF or IFNDEF).

A more complicated example, performing simple substitution, might look like this:

```
#DEFINE         VAR1_LO         (VAR1 & 255)
```

This statement would cause all occurrences of the string 'VAR1_LO' in the source to be substituted with '(VAR1 & 255)'.

As a more complicated example, using the argument expansion capability, consider this:

```
#DEFINE  ADD(xx,yy)    clc\ lda xx\ adc yy\ sta xx
```

If the source file then contained a line like this:

```
ADD(VARX,VARY)
```

It would be expanded to:

```
clc\ lda VARX\ adc VARY\ sta VARX
```

The above example shows the use of the backslash ('\') character as a multiple instruction statement delimiter. This approach allows the definition of fairly powerful, multiple statement macros. The example shown generates 6502 instructions to add one memory location to another.

Some rules associated with the argument list:

1. Use a maximum of 10 arguments.
2. Each argument should be a maximum of 15 characters.

Note that macros can be defined on the TASM command line, also, with the **-d** option flag.

DEFCONT

The DEFCONT directive can be used to add to the last macro started with a DEFINE directive. This provides a convenient way to define long macros without running off the edge of the page. The ADD macro shown above could be defined as follows:

```
#DEFINE         ADD(xx,yy)      clc
#DEFCONT                        \ lda xx
#DEFCONT                        \ adc yy
#DEFCONT                        \ sta xx
```

ECHO

The ECHO directive can be used to send output to the console (stderr). It can accept either a quoted text string (with the standard escape sequences allowed) or a valid expression. It can accept only one or the other, however. Multiple instances of the directive may be used to create output that contains both. Consider the following example:

```
.ECHO "The size of the table is "
.ECHO (table_end - table_start)
.ECHO " bytes long.\n"
```

This would result in a single line of output something like this:

```
The size of the table is 196 bytes long.
```

EJECT

The EJECT directive can be used to force a top-of-form and the generation of a page header on the list file. It has no effect if the paging mode is off (see PAGE/NOPAGE). The format is:

```
[label]     .EJECT
```

ELSE

The ELSE directive can optionally be used with IFDEF, IFNDEF and IF to delineate an alternate block of code to be assembled if the block immediately following the IFDEF, IFNDEF or IF is not assembled.

Here are some examples of the use of IFDEF, IFNDEF, IF, ELSE, and ENDIF:

```
#IFDEF   label1
    lda      byte1
    sta      byte2
#ENDIF

#ifdef   label1
    lda      byte1
#else
    lda      byte2
#endif

#ifndef  label1
    lda      byte2
#else
    lda      byte1
#endif

#if ($ >= 1000h)
; generate an invalid statement to cause an error
;  when we go over the 4K boundary.
!!! PROM bounds exceeded.
#endif
```

END

The END directive should follow all code/data generating statements in the source file. It forces the last record to be written to the object file. The format is:

```
[label]       .END [addr]
```

The optional *addr* will appear in the last object record (Motorola S9 record type) if the object format is Motorola hex. The *addr* field is ignored for all other object formats.

ENDIF

The ENDIF directive must always follow an IFDEF, IFNDEF, or IF directive and signifies the end of the conditional block.

EQU

The EQU directive can be used to assign values to labels. The labels can then be used in expressions in place of the literal constant. The format is:

```
label   .EQU   expr
```

Here is an example:

```
MASK   .EQU  0F0H
;
       lda   IN_BYTE
       and   MASK
       sta   OUT_BYTE
```

An alternate form of the EQU directive is '='. The previous example is equivalent to any of the following:

```
MASK    =   0F0H
MASK    =0F0H
MASK    =$F0
```

White space must exist after the *label*, but none is required after the '='.

EXPORT

The EXPORT directive can be used to define labels (symbols) that are to be written to the export symbol file. The symbols are written as equates (using the .EQU directive) so that the resulting file can be included in a subsequent assembly. This feature can help overcome some of the deficiencies of TASM due to its lack of a relocating linker. The format is:

```
[label]  .EXPORT      label [,label...]
```

The following example illustrates the use of the EXPORT directive and the format of the resulting export file:

Source file:

```
EXPORT        read_byte
EXPORT        write_byte, open_file
```

Resulting export file:

```
read_byte      .EQU   $1243
write_byte     .EQU   $12AF
open_file      .EQU   $1301
```

FILL

The FILL directive can be used to fill a selected number of object bytes with a fixed value. Object memory is filled from the current program counter forward. The format is as follows:

```
[label]  .FILL      number_of_bytes [,fill_value]
```

The *number_of_bytes* value can be provided as any valid expression. The optional fill_value can also be any valid expression. If *fill_value* is not provided, a default value of 255 ($FF) is used.

IFDEF

The IFDEF directive can be used to optionally assemble a block of code. It has the following form:

```
#IFDEF  macro_label
```

When invoked, the list of macro labels (established via DEFINE directives) is searched. If the label is found, the following lines of code are assembled. If not found, the input file is skipped until an ENDIF or ELSE directive is found.

Lines that are skipped over still appear in the listing file, but a '~' will appear immediately after the current PC and no object code will be generated (this is applicable to IFDEF, IFNDEF, and IF).

IFNDEF

The IFNDEF directive is the opposite of the IFDEF directive. The block of code following is assembled only if the specified *macro_label* is undefined. It has the following form:

```
#IFNDEF  macro_label
```

When invoked, the list of macro labels (established via DEFINE directives) is searched. If the label is not found, the following lines of code are assembled. If it is found, the input file is skipped until an ENDIF or ELSE directive is found.

IF

The IF directive can be used to optionally assemble a block of code dependent on the value of a given expression. The format is as follows:

```
#IF     expr
```

If the expression *expr* evaluates to non-zero, the following block of code is assembled (until an ENDIF or ELSE is encountered).

INCLUDE

The INCLUDE directive reads in and assembles the indicated source file. INCLUDEs can be nested up to four levels. This allows a convenient means to keep common definitions, declarations, or subroutines in files to be included as needed. The format is as follows:

```
#INCLUDE        filename
```

The *filename* must be enclosed in double quotes. Here are some examples:

```
#INCLUDE        "macros.h"
#include        "equates"
#include        "subs.asm"
```

LIST/NOLIST

The LIST and NOLIST directives can be used to alternately turn the output to the list file on (LIST) or off (NOLIST). The formats are:

```
.LIST
.NOLIST
```

LOCALLABELCHAR

The LOCALLABELCHAR directive can be used to override the default "_" as the label prefix indicating a local label. For example, to change the prefix to "?" do this:

```
[label]    .LOCALLABELCHAR "?"
```

Be careful to use only characters that are not operators for expression evaluation. To do so causes ambiguity for the expression evaluator. Some safe characters are "?", "{", and "}". See Labels for an example of local label usage.

LSFIRST/MSFIRST

The LSFIRST and MSFIRST directives determine the byte order rule to be employed for the WORD directive. The default (whether correct or not) for all TASM versions is the least significant byte first (LSFIRST). The following illustrates its effect:

```
0000  34 12     .word $1234
0002            .msfirst
0002  12 34     .word $1234
0004            .lsfirst
0004  34 12     .word $1234
```

MODULE

The MODULE directive defines the scope of local labels. The format is:

```
[label]    .MODULE label
```

Here is an example:

```
          .MODULE module_x
          lda regx
          jne _skip
          dec
   _skip  rts
```

```
        .MODULE module_y
        lda regy
        jne _skip
        dec
_skip   rts
```

In the above example, the local label *_skip* is reused without harm since the two usages are in separate modules. See also section LOCALLABELCHAR directive.

ORG

The ORG directive provides the means to set the Instruction Pointer (a.k.a. Program Counter) to the desired value. The format is:

```
[label] .ORG    expr
```

The *label* is optional. The Instruction pointer is assigned the value of the *expr*. For example, to generate code starting at address 1000H, the following could be done:

```
start   .ORG    1000H
```

The expression (*expr*) may contain references to the current Instruction Pointer, thus allowing various manipulations to be done. For example, to align the Instruction Pointer on the next 256 byte boundary, the following could be done:

```
.ORG  (($ + 0FFH) & 0FF00H)
```

ORG can also be used to reserve space without assigning values:

```
.ORG    $+8
```

An alternate form of ORG is '*=' or '$='. Thus the following two examples are exactly equivalent to the previous example:

```
*=*+8
$=$+8
```

PAGE/NOPAGE

The PAGE/NOPAGE directives can be used to alternately turn the paging mode on (PAGE) or off (NOPAGE). If paging is in effect, then every sixty lines of output will be followed by a Top of Form character and a two line header containing page number, filename, and the title. The format is:

```
.PAGE
.NOPAGE
```

The number of lines per page can be set with the '-p' command line option.

SET

The SET directive allows the value of an existing label to be changed. The format is:

```
label   .SET    expr
```

The use of the SET directive should be avoided since changing the value of a label can sometimes cause phase errors between pass 1 and pass 2 of the assembly.

SYM/AVSYM

These directives can be used to cause a symbol table file to be generated. The format is:

```
.SYM    ["symbol_filename"]
.AVSYM  ["symbol_filename"]
```

For example:

```
.SYM        "symbol.map"
.SYM
.AVSYM      "prog.sym"
.AVSYM
```

The two directives are similar, but result in a different format of the symbol table file. The format of the SYM file is one line per symbol, each symbol starts in the first column and is followed by white space and then four hexadecimal digits representing the value of the symbol. The following illustrates the format:

```
label1          FFFE
label2          FFFF
label3          1000
```

The AVSYM directive is provided to generate symbol tables compatible with the Avocet 8051 simulator. The format is similar, but each line is prefixed by an 'AS' and each symbol value is prefixed by a segment indicator:

```
AS      start           C:1000
AS      read_byte       C:1245
AS      write_byte      C:1280
AS      low_nib_mask    N:000F
AS      buffer          X:0080
```

The segment prefixes are determined by the most recent segment directive invoked (see BSEG/CSEG/DSEG/NSEG/XSEG directives).

TEXT

This directive allows an ASCII string to be used to assign values to a sequence of locations starting at the current Instruction Pointer. The format is:

```
[label] .TEXT   "string"
```

The ASCII value of each character in string is taken and assigned to the next sequential location. Some escape sequences are supported as follows:

Escape Sequence	Description
\n	Line Feed
\r	Carriage return
\b	Backspace
\t	Tab
\f	Formfeed
\\	Backslash
\"	Quote
\000	Octal value of character

Here are some examples:

```
message1    .TEXT   "Disk I/O error"
message2    .text   "Enter file name "
            .text   "abcdefg\n\r"
            .text   "I said \"NO\""
```

TITLE

The TITLE directive allows the user to define a title string that appears at the top of each page of the list file (assuming the PAGE mode is on). The format is:

```
[label]     .TITLE  "string"
```

The *string* should not exceed 80 characters. Here are some examples:

```
.TITLE  "Controller version 1.1"
.title  "This is the title of the assembly"
.title  ""
```

WORD

The WORD directive allows a value assignment to the next two bytes pointed to by the current Instruction Pointer. The format is:

```
[label] .WORD expr [,expr...]
```

The least significant byte of expr is put at the current Instruction Pointer with the most significant byte at the next sequential location (unless the MSFIRST directive has been invoked). Here are some examples:

```
data_table      .WORD   (data_table + 1)
                .word   $1234
                .Word   (('x' - 'a')  << 2)
                .Word  12, 55, 32
```

OBJECT FILE FORMATS

TASM can generate object code in the formats indicated below:

Option	Description
-g0	Intel hex (default)
-g1	MOS Technology hex (same as **-m**)
-g2	Motorola hex
-g3	binary (same as **-b**)
-g4	Intel hex with word addresses

The **-m** and **-b** flags may also be used, as indicated above. If both are used the right-most option on the command line will be obeyed.

Intel Hex Object Format

This is the default object file format. This format is line oriented and uses only printable ASCII characters except for the carriage return/line feed at the end of each line. The format is symbolically represented as:

:*NN AAAA RR HH CC CRLF*

Where:

:	Record Start Character (colon)
NN	Byte Count (2 hex digits)
AAAA	Address of first byte (4 hex digits)
RR	Record Type (**00** except for last record which is **01**)
HH	Data Bytes (a pair of hex digits for each byte of data in the record)
CC	Check Sum (2 hex digits)
CRLF	Line Terminator (CR/LF for DOS, LF for LINUX)

The last line of the file will be a record conforming to the above format with a byte count of zero.

The checksum is defined as:

sum = byte_count+address_hi+address_lo+record_type+(sum of all data bytes)
checksum = ((-sum) & ffh)

Here is a sample listing file followed by the resulting object file:

```
0001   0000
0002   1000                    .org   $1000
0003   1000 010203040506        .byte  1, 2, 3, 4, 5, 6, 7, 8
0003   1006 0708
0004   1008 090A0B0C0D0E        .byte  9,10,11,12,13,14,15,16
0004   100E 0F10
0005   1010 111213141516        .byte 17,18,19,20,21,22,23,24,25,26
0005   1016 1718191A
0006   101A                    .end
    :1810000001020304050607080 90A0B0C0D0E0F10111213141516 1718AC
    :02101800191AA3
    :00000001FF
```

Intel Hex Word Address Object Format

This format is identical to the **Intel Hex Object Format** except that the address for each line of object code is divided by two thus converting it to a word address (16 bit word). All other fields are identical.

Here is an example:

```
:1808000001020304050607080 90A0B0C0D0E0F10111213141516 1718AC
:02080C00191AA3
:00000001FF
```

MOS Technology Hex Object Format

This format is line oriented and uses only printable ASCII characters except for the carriage return/line feed at the end of each line. Each line in the file is of the following format:

:*NN AAAA HH CC CRLF*

Where:

;	Record Start Character (semicolon)
NN	Byte Count (2 hex digits)
AAAA	Address of first byte (4 hex digits)
HH	Data Bytes (a pair of hex digits for each byte of data in the record)
CCCC	Check Sum (4 hex digits)
CRLF	Line Terminator (CR/LF for DOS, LF for LINUX)

The last line of the file will be a record conforming to the above format with a byte count of zero.

The checksum is defined as:

sum =byte_count+address_hi+address_lo+record_type+(sum of all data bytes)
checksum = (sum & ffffh)

Here is a sample object file:

```
;1810000102030405060708090A0B0C0D0E0F10111213141516171801 54
;021018191A005D
;00
```

Motorola Hex Object Format

This format is line oriented and uses only printable ASCII characters except for the carriage return/line feed at the end of each line. The format is symbolically represented as:

S1 NN AAAA HH CCCC CRLF

Where:

S1	Record Start tag
NN	Byte Count (2 hex digits) (data byte count + 3)
AAAA	Address of first byte (4 hex digits)
HH	Data Bytes (a pair of hex digits for each byte of data in the record)
CC	Check Sum (2 hex digits)
CRLF	Line Terminator (CR/LF for DOS, LF for LINUX)

The checksum is defined as:

sum = byte_count+address_hi+address_lo+(sum of all data bytes)
checksum = ((~sum) & ffh)

Here is a sample file:

```
S11B1000010203040506070 8090A0B0C0D0E0F10111213141516171 8A8
S1051018191A9F
S9030000FC
```

The last line of the file will be a record with a byte count of zero and a tag of S9. The address field will be 0000 unless and address was provided with the END directive in which case it will appear in the address field.

Binary Object Format.

This file format is essentially a memory image of the object code without address, checksum or format description information.

Note that when this object format is selected (-b option), the -c option is forced. This is done so that no ambiguity results from the lack of address information in the file. Without the -c option, discontinuous blocks of object code would appear contiguous.

LISTING FILE FORMAT

Each line of source code generates one (or more) lines of output in the listing file. The fields of the output line are as follows:

1. Current source file line number (4 decimal digits).
2. An optional '+' appears if this is an 'INCLUDE' file. (One '+' for each level of INCLUDE invoked).
3. Current Instruction Pointer (4 hex digits). An optional '~' follows the Instruction Pointer if the line of source code is not being assembled because of an IFDEF, IFNDEF, or IF directive.
4. Resulting code/data generated from this source line (two hex digits per byte, each byte separated by a space, up to six bytes per line).
5. The source line exactly as it appears in the source file.

If paging is enabled (by either the '-p' option flag or the .PAGE directive) some additional fields will be inserted into the listing file every 60 lines. These fields are:

1. Top of Form (form feed).
2. Assembler identifier (e.g. "TASM 6502 Assembler").
3. Initial source file name.
4. Page number.
5. Title.

If errors are encountered, then error messages will be interspersed in the listing. TASM outputs error messages proceeding the offending line. The following example illustrates the format:

```
0001   0000             label1  .equ  40h
0002   0000             label2  .equ  44h
0003   0000
0004   1000             start:  .org  1000h
0005   1000 E6 40               inc   label1
0006   1002 E6 44               inc   label2
tt.asm line 0007: Label not found: (label3)
0007   1004 EE 00 00            inc   label3
0008   1007 4C 00 10            jmp   start
0009   100A                     .end
0010   100A
tasm: Number of errors = 1
```

PROM PROGRAMMING

A wide variety of PROM programming equipment is available that can use object code in one or more of the formats supported by TASM. Here are some notes concerning the generation of code to be programmed into PROMs:

PRESET MEMORY

It is often desirable to have all bytes in the PROM programmed even if not explicitly assigned a value in the source code (e.g. the bytes are skipped over with a .ORG statement). This can be accomplished by using the *-c* (contiguous block) and the *-f* (fill) command line option flags. The *-c* will ensure that every byte from the lowest byte assigned a value to the highest byte assigned a value will be in the object file with no gaps. The *-f* flag will assign the specified value to all bytes before the assembly begins so that when the object file is written, all bytes not assigned a value in the source code will have a known value. As an example, the following command line will generate object code in the default Intel Hex format with all bytes not assigned a value in the source set to EA (hex, 6502 NOP instruction):

```
tasm -65 -c -fEA test.asm
```

CONTIGUOUS BLOCKS

To ensure that TASM generates object code to cover the full address range of the target PROM, put a .ORG statement at the end of the source file set to the last address desired. For example, to generate code to be put in a 2716 EPROM (2 Kbytes) from hex address $1000 to $17ff, do something like this in the source file:

```
;start of the file
    .ORG    $1000
;rest of the source code follows
source code
;end of the source code
    .ORG    $17ff
    .BYTE   0
    .END
```

Now, to invoke TASM to generate the code in the binary format with all unassigned bytes set to 00 (6502 BRK instruction), do the following:

```
tasm -65 -b -f00 test.asm
```

Note: -b forces the -c option.

ERROR MESSAGES

Error Message Format

TASM error messages take the following general form:

filename line *line_number*: *error_message*

For example:

```
main.asm line 0032: Duplicate label (start)
```

Many editors and IDE's can run assemblies from within and parse the error messages to make it convenient to jump to each error and correct. If the above format is not satisfactory for your editor or IDE, use the TASMERRFORMAT environment variable to alter the error message format.

When using automatic invocation of TASM from within an editor or IDE, it may be useful to use the TASMOPTS environment variable to set other command line options. For example, to cause TASM to perform 6502 assemblies, set TASMOPTS like this:

```
SET TASMOPTS=-65
```

ERROR MESSAGE DESCRIPTIONS

Binary operator where value expected
: Two binary operators in a row indicate a missing value.

Branch off of current 2K page
: An instruction is attempting to branch to a location not within the current 2K byte page.

Branch off of current page
: An instruction is attempting to branch to a location not within the current 256 byte page.

Cannot malloc for label storage
: Insufficient memory to store more labels. See **LIMITATIONS**.

Duplicate label

The label for the current line has already been assigned a value. Duplicate label checks are optionally enabled by the *-a* option.

File name too short

A file name on the command line is fewer than 3 characters. A two character file name may be valid, of course, but it is detected as an error to prevent a garbled option flag from being taken as a source file, which in turn can result in the true source file being taken as the object file. Since the object file is truncated at startup time, the source file could be clobbered.

Forward reference in equate

An EQU directive is using a label on the right hand side that has not yet been defined.

Heap overflow on label definition

TASM was unable to allocate memory to store the label.

Imbalanced conditional.

An end-of-file was encountered at which time the level of descent in conditional directives was different from when the file was entered. Conditional directives include IF, IFDEF, and IFNDEF.

Invalid Object file type.

An object file type was requested by the -g command line option that is not valid. See section on Option g - Object File Format.

Invalid operand.

No indirection for this instruction. The first character of an operand was a left parenthesis for an instruction that does not explicitly specify that as the format. Some micros use the parenthesis as an indicator of indirection, but putting a layer of parenthesis around an expression is always a valid thing to do (as far as the expression evaluator is concerned). The test for this case is only done if the *-a4* option is selected. See section on **ASSEMBLY CONTROL**.

Invalid token where value expected.

Two binary operators in a row are not allowed.

Label too long.

Labels are limited to 31 characters.

Label value misaligned

The value of a label appears to have a different value on the second pass then it was computed to have on the first pass. This is generally due to Zero Page Addressing mode problems with the 6502 version of TASM. Labels that are used in operands for statements that could utilize Zero Page addressing mode should always be defined before used as an operand.

Label not found

A label used in an expression was not found in the current label table.

Label must pre-exist for SET.

The SET directive can only be applied to an existing label.

Label table overflow

To many labels have been encountered.

List file open error

TASM was not able to open the specified list file.

Macro expansion too long.

The expansion of a macro resulted in a line that exceeded the maximum length.

Max number of nested conditionals exceeded

Too many levels of IF, IFDEF, or IFNDEF.

Maximum number of args exceeded

Too many macro arguments.

Maximum number of macros exceeded

Too many macros (DEFINEs) have been encountered.

No END directive before EOF

The source file did not have an END directive in it. This is not fatal, but may cause the last object file record to be lost.

No files specified
TASM was invoked with no source file specified.

No such label
A SET directive was encountered for a label not yet defined. The value of labels that are modified by the SET directive must already exist.

No terminating quote
A double quote was used at the start of a a text string but was not used at the end of the string.

No indirection for this instruction.
A parenthesis was found around the operand expression. This may indicate an attempt to use indirection where it is inappropriate.

Non-unary operator at start of expression
A binary operator (such as '*') was found at the beginning of an expression. Some micros use '*' as an indirection operator. Since it is also a legitimate operator in an expression, some ambiguity can arise. If a particular instruction/addressing mode does not allow indirection, and a '*' is placed in front of the associated expression, the assembler will assume this error. See the *-a8* option of ASSEMBLY CONTROL.

Object file open error
TASM was not able to open the specified object file.

Range of argument exceeded
The value of an argument exceeds the valid range for the current instruction and addressing mode.

Range of relative branch exceeded
A branch instruction exceeds the maximum range.

Source file open error
TASM was not able to open the specified source file.

Unrecognized directive
A statement starting with a '.' or '#' has a mnemonic that is not defined as a directive.

Unrecognized instruction
A statement has an opcode mnemonic that is not defined.

Unrecognized argument
A statement has an operand format that is not defined.

Unknown token
Unexpected characters were encountered while parsing an expression.

Unused data in MS byte of argument
An instruction or directive used the least significant byte of an argument and left the most significant byte unused, but it was non-zero.

Unknown option Flag.
Invalid option flag has been specified on the command line. invoke TASM with nothing on the command line to see a list of valid options.

LIMITATIONS

Maximum number of labels	15000
Maximum length of labels	32 characters
Maximum address space	64 Kbytes (65536 bytes)
Maximum number of nested INCLUDES	4

Maximum length of TITLE string	79 characters
Maximum source line length	511 characters
Maximum length after macro expansion	511 characters
Maximum length of expressions	511 characters
Maximum length of pathnames	79 characters
Maximum length of command line	127 characters
Maximum number of instructions (per table)	1200
Maximum number of macros	1000
Maximum number of macro arguments	10
Maximum length of macro argument	16 characters
Memory requirements	512K

Other Limitations

1. The 8048 version of TASM does not check for use of memory beyond any reasonable bounds (e.g. an 8048 has a maximum address space of 4 Kbytes but TASM will let you pretend that you have 64 Kbytes).
2. Expression evaluation has no operator precedence in effect which can make for unexpected results if not explicitly grouped with parenthesis.
3. First page of listing file will not show a user defined title (defined via TITLE directive).

Top