ASSEMBLER PSEUDO OPCODE DESCRIPTIONS

EQU or (=) (EQUate)

```
Label EQU expression
Label  =  expression  (alternate syntax)
   START EQU $1000          [ equate START to $1000 ]
   CHAR  EQU "A"            [ equate CHAR to ASCII value of A ]
   PTR   =   *              [ PTR equals present address in the assembled source listing ]
   LABEL =   55             [ LABEL equals the decimal value of 55 ]
```

```
LABEL  EQU  $25
       LDA  LABEL
```

This will load the accumulator with the value stored in location $25.

```
LABEL  EQU  #$25
       LDA  LABEL
```

This will load the accumulator with the value of $25.

IMPORTANT: Forgetting to include the # symbol to load an immediate value is probably the number-one cause of program bugs. If you're having a problem, double check immediate value syntax first!

EQU is used to define the value of a label, usually an exterior address or an often used constant for which a meaningful name is desired. All EQUates should be located at the beginning of the program.

NOTE: The assembler will not permit an EQUate to a zero page number after the label equated has been used, since bad code could result from such a situation . Also see the section on Variables.

(1) For Example:

```
1  LABEL  LDA    #LEN
2  LABEL  DFB    $00
3         DFB    $01
4  LEN    EQU    * - LABEL
```

When assembled, this will give an "ILLEGAL FORWARD REFERENCE IN LINE 4" ERROR message. The solution is as follows:

```
1         LDA    #END - LABEL
2  LABEL  DFB    $00
3         DFB    $01
4  END
```

Note that labels are CASE SENSITIVE. Therefore, the assembler will consider the following labels as different labels:

```
START       [ upper case label ]
Start       [ mixed case label ]
start       [ lower case label ]
```

EXT (EXTernal label)

```
label EXT       [ label is external labels name ]
   PRINT EXT    [ define PRINT as external ]
```

This pseudo op defines a label as an external label for use by the Linker. The value of the label, at assembly time, is set to $8000, but the final value is resolved by the Linker. The symbol table will list the label as having the value of $8000 plus its external reference number (0-$FE). See the Linker section of the manual for more information on this opcode.

ENT (ENTry label)

```
label ENT
   PRINT ENT    [ define PRINT as entry label ]
```

This pseudo-op will define the label column as an ENTRY label. An entry label is a label that may be referred to as an EXTernal label by another REL code module, which may refer to the ENT label just as if it were an ordinary label. It can be EQUated, jumped to, branched to, etc. The true address of an entry label will be resolved by the Linker.

See *The Linker* section of the manual for more information on this opcode.

ORG (set ORiGin)

ORG expression
ORG

```
ORG  $1000          [ start code at $1000 ]
ORG  START+END      [ start at value of expression ]
ORG                 [ re-ORG ]
```

Establishes the address at which the program is designed to run, and where it will be automatically BLOADed in memory if it is a BINary type object file. This is not necessarily where Merlin 8/16 will actually assemble the code with the ASM command. ORG defaults to $8000. Ordinarily there will be only one ORG and it will be at the start of the program. If more than one ORG is used, the first one establishes the BLOAD address, while the second actually establishes a new origin for any code that follows it. This can be used to create an object file that would load to one address though it may be designed to run at another address.

NOTE: If you need to back up the object pointers you must use DS-1 . This *cannot* be done by ORG*-1.

ORG without an operand is accepted and is treated as a "REORG" type command. It is intended to be used to re-establish the correct address pointer after a segment of code which has a different ORG. When used in a REL file, all labels in a section between an "ORG address" and an "ORG noaddress" are regarded as absolute addresses. This should only be used in a section that is to be moved to an explicit address.

Example of ORG without an operand:

```
                    1             ORG  $1000
1000: A0 00         2             LDY  #0
1002: 20 21 10      3             JSR  MOVE       ;"MOVE" IS
1005: 4C 12 10      4             JMP  CONTINUE   ;NOT LISTED.
                    5             ORG  $300       ;ROUTINE TO
0300: 8D 08 C0      6  PAGE3      STA  MAINZP     ;BE MOVED
0303: 20 ED FD      7             JSR  COUT
0306: 8D 09 C0      8             STA  AUXZP
0309: 60            9             RTS
                    10            ORG            ;REORG
1012: A9 C1         11 CONTINUE   LDA  #"A"
1014: 20 00 03      12            JSR  PAGE3
```

Sometimes, you will want to generate two blocks of code with separate ORGs in one assembly. There are four ways of doing this involving four different directives. These are DSK, SAV, DS and REL. All four are described later in this manual, and are presented here in the interest of continuity.

METHOD #1: USING THE DSK OPCODE

In this first example, two separate disk files are created with independent ORG values by using the DSK command. This command directs the assembler to assemble all code to disk *following* the DSK command. The file is closed when either the assembly ends or another DSK command is encountered.

```
 1  ******************
 2  * MULTIPLE ORG'S *
 3  *  SOLUTION # 1  *
 4  *  DSK COMMAND   *
 5  ******************
 6
 7       DSK FILEONE          ; CREATE 1ST FILE
 8       ORG $8000            ; DEFINE ITS LOAD ADDRESS
 9       LDA #0               ; SAMPLE PROGRAM LINE
10
11       DSK FILETWO          ; CLOSE 1ST FILE, START 2ND
12       ORG $8100            ; DEFINE ITS LOAD ADDRESS
13       LDY #1               ; ANOTHER PROGRAM, FILE CLOSED AT END
```

METHOD #2: USING THE SAV OPCODE

In this second example, two separate disk files are again created with independent ORG values, but this time by using the SAV command. This command directs the assembler to save all code assembled *previous* to the SAV code disk.

```
 1  ******************
 2  * MULTIPLE ORG'S *
 3  *  SOLUTION # 2  *
 4  *  SAV COMMAND   *
 5  ******************
 6
 7       ORG $8000            ; LOAD ADDRESS FOR 1ST FILE
 8       LDA #0               ; SAMPLE PROGRAM LINE
 9       SAV FILEONE          ; SAVE 1ST FILE
10
11       ORG $8100            ; LOAD ADDRESS FOR 2ND FILE
12       LDY #1               ; SAMPLE PROGRAM LINE
13       SAV FILETWO          ; SAVE 2ND FILE
```

METHOD #3: USING THE DS OPCODE

In this third example, just one file is created on disk, but the two blocks of code are separated by approximately a $100 byte gap, less the size of the first code block.

This might be useful, for example, if you wanted your code to skip over the Hi-Res page 1 area of memory. Please read the section on SAV for more information about multiple ORGs in a program.

```
1  ******************
2  * MULTIPLE ORG'S *
3  *  SOLUTION # 3  *
4  *   DS COMMAND   *
5  ******************
6
7       ORG $8000          ; LOAD ADDRESS OF FILE
8       LDA #0             ; SAMPLE PROGRAM LINE
9       DS \               ; FILL WITH $0 TO NEXT PG. BOUNDARY
10                         ; or could have been DS $8100-*
11      LDY #1             ; SAMPLE LINE OF 2ND SEGMENT
12                         ; THIS WILL START AT $8100
```

METHOD #4: USING THE REL OPCODE

The REL directive is used to create relocatable files. The Linker use these REL files to create the final object code to run at a given location. The Merlin 16 Linker supports multiple output files, and so can be used to create two or more files with independent ORG values.

```
1  ******************
2  * MULTIPLE ORG'S *
3  *  SOLUTION #4A  *
4  *  REL COMMAND   *
5  ******************
6
7       REL                ; RELOCATABLE FILE TYPE (LNK)
8       DSK FILEONE.L      ; CREATE 1ST LNK FILE
9       LDA #0             ; SAMPLE PROGRAM LINE
```

```
1  ******************
2  * MULTIPLE ORG'S *
3  *  SOLUTION #4B  *
4  *  REL COMMAND   *
5  ******************
6
7       REL                ; RELOCATABLE FILE TYPE (LNK)
8       DSK FILETWO.L      ; CREATE 2ND LNK FILE
9       LDA #0             ; SAMPLE PROGRAM LINE
```

This example is for the Merlin 16 Linker only. These two files would be linked for the desired ORG addresses with a Link command file like this:

```
 1  ******************
 2  * MULTIPLE ORG'S *
 3  *      LINKER    *
 4  *  COMMAND FILE  *
 5  ******************
 6
 7       ORG $8000            ; SPECIFY 1ST ADDRESS
 8       LNK FILEONE.L        ; LINK 1ST FILE
 9       SAV FILE1            ; SAVE 1ST OBJECT FILE
10
11       ORG $8100            ; SPECIFY 2ND ADDRESS
12       LNK FILETWO.L        ; LINK 2ND FILE
13       SAV FILE2            ; SAVE 2ND OBJECT FILE
```

Although the Linker is normally used to *combine* several source files, or to communicate label values between programs, it can be used to assemble even unrelated files.

REL (RELocatable code module)

REL
    REL [only option for this opcode]

This opcode instructs the assembler to generate code files compatible with the relocating linker. This opcode must occur *prior* to the use or definition of any labels. See the Linker section of this manual for more information on this opcode.

OBJ (set OBJect)

OBJ expression
    OBJ $4000 [use of hex address]
    OBJ START [use with a label]

The OBJ opcode is accepted only *prior* to the start of the code and it only sets the division line between the symbol table and object code areas in memory, which defaults to $8000. *The OBJ address is accepted only if it lies between $4000 and $BFE0.* This may cause a problem if you try to assemble a listing OBJ'ed to $300, for example.

Nothing disastrous will happen if OBJ is out of range; when you return to the Main Menu to save your object file, no object file address and length values will be displayed on the screen, and Merlin 8/16 will simply beep at you if you try to save an object file.

The main reason for using OBJ is to be able to quit the assembler directly, test a routine in memory, and then be able to immediately return to the assembler to make any corrections. If you want to do this, simply use the GET command (Example: GET $300) in the DOS 3.3 version of Merlin 8 before quitting to BASIC.

In the ProDOS version of Merlin 8/16, this isn't an option because you can't temporarily quit Merlin 8/16 to BASIC. *For ProDOS, it is recommended that you disregard the use of OBJ entirely.* To test a program from the Main Menu, you should save the source code, save the object code, then quit to BASIC.SYSTEM. Then BLOAD the object file. The file will automatically load at the proper location.

Most people should never have to use OBJ. If the REL opcode is used then OBJ is disregarded. If DSK is used then you can, but may not have to, set OBJ to $BFE0 to maximize the space for the symbol table.

In Merlin 16, the address range of the symbol table is printed in hex, at the end of an assembly. This allows you to see when a new OBJ value may be needed. You can also use the DSK command should the object file become too big.

PUT (PUT a text file in assembly)

PUT filename

DOS 3.3 Examples:

```
PUT  SOURCEFILE      [ PUTs file T.SOURCEFILE ]
PUT  !SOURCE         [ PUTs file SOURCE ]
PUT  !SOURCE,D2      [ PUTs file SOURCE from drive 2 ]
```

ProDOS Examples:

```
PUT  SOURCEFILE      [ PUTs file SOURCEFILE.S ]
PUT  /PRE/SOURCE     [ PUTs file SOURCE.S from subdirectory PRE ]
```

"PUT filename" reads the named file and inserts it at the location of the opcode.

Occasionally your source file will become too large to assemble in memory. This could be due to a very long program, extensive comments, dummy segments, etc. In any case, this is where the PUT opcode can make life easy. All you have to do is divide your program into sections, then save each section as a separate text file. The PUT opcode will load these text files and insert them in the "Master" source file at the location of the PUT opcode. This "Master" source file usually only contains equates, macro definitions (if used), and *all* of your PUT opcodes.

A Master source file might look something like this:

```
********************
*  Master Source   *
********************

* LABEL DEFINITIONS

  LABEL1   EQU  $00
  LABEL2   EQU  $02
  COUT     EQU  $FDED

* MACRO DEFINITIONS

  SWAP     MAC
           LDA ]1
           STA ]2
           <<<

* SAMPLE SOURCE CODE

           LDA  #LABEL1
           STA  LABEL2
           LDA  #/LABEL1
           STA  LABEL2+1
           LDA  LABEL1
           JSR  COUT
           RTS

* BEGIN PUTFILES

           PUT  FILE1       ; FIRST SOURCE FILE SEGMENT
           PUT  FILE2       ; SECOND SOURCE FILE SEGMENT
           PUT  FILE3       ; THIRD SOURCE FILE SEGMENT
```

NOTE: You cannot define macros from within a PUT file. Also, you cannot call the next PUT file from within a PUT file. All macro definitions and PUT opcodes must be in the Master source file. There are other uses for PUT files such as PUTting portions of code as subroutines, PUTting a file of ProDOS global page equates, etc. The possibilities are almost endless.

Here's an example of a Master program that uses 3 PUT files to create a final object file called FINAL.OBJ, which is called from an Applesoft BASIC program. The DSK command is not required when using PUT files, but may be needed for object files that are too large to fit in memory, or where a special filetype, other than BIN, is desired for the object file.

```
1  * MASTER CALLING PROGRAM
2
3  COUT   EQU  $FDED
4  HOME   EQU  $FC58
5
```

```
 6          ORG  $8000
 7
 8          DSK  FINAL.OBJ  ; OUTPUT FILE
 9          JSR  HOME
10          PUT  FILE1      ; Named "T.FILE1" on disk (Merlin 8, DOS 3.3)
11          PUT  FILE2      ; Named "T.FILE2" on disk
12          PUT  FILE3      ; Named "T.FILE3" on disk
13                          ; Named "FILE1.S, etc. on ProDOS disk)
```

And here are the text files that the Master program calls in by using the PUT commands:

```
 1  * FILE1
 2
 3          LDX  #0
 4  LOOP1   LDA  STRING1,X
 5          BEQ  FILE2
 6          JSR  COUT
 7          INX
 8          BNE  LOOP1
 9  STRING1 ASC "THIS IS FILE 1"
10          HEX  8D00
```

```
 1  * FILE2
 2
 3  FILE2   LDX  #0
 4  LOOP2   LDA  STRING2,X
 5          BEQ  FILE3
 6          JSR  COUT
 7          INX
 8          BNE  LOOP2
 9  STRING2 ASC "NOW ITS FILE 2"
10          HEX  8D00
```

```
 1  * FILE3
 2
 3  FILE3   LDX  #0
 4  LOOP3   LDA  STRING3,X
 5          BEQ  DONE
 6          JSR  COUT
 7          INX
 8          BNE  LOOP3
 9  DONE    RTS
10  STRING3 ASC "FINALLY FILE 3"
11          HEX 8D00
```

Each PUT file (FILE1, FILE2, FILE3) prints a message identifying which file is in operation.

The final assembly is tested by this Applesoft program:

```
10  TEXT : HOME
20  PRINT  CHR$ (4);"BLOAD FINAL.OBJ"
25  CALL 32768
30  VTAB 10: HTAB 10: PRINT "IT REALLY WORKS!"
40  VTAB 15: LIST : END
```

When this program is run, the following lines of text should appear on the screen:

```
THIS IS FILE 1
NOW ITS FILE 2
FINALLY FILE 3
IT REALLY WORKS!
```

DOS 3.3 NOTE: Drive and slot parameters are accepted in the standard DOS syntax. The "filename" specified must be a text file with the "T." prefix. If it doesn't have the "T." prefix in the disk catalog, the "filename" specified must start with a character less than "@" in ASCII value. This tells Merlin 8/16 to look for a file without the "T." prefix. The "!" character can be used for this purpose. For example:

Disk file name = T.SOURCE CODE [name in catalog]
PUT file name = SOURCE CODE [name in PUT opcode]

Disk file name = SOURCE CODE [name in catalog]
PUT file name = !SOURCE CODE [name in PUT opcode]

ProDOS NOTE: Drive and slot parameters are not accepted; pathnames must be used. Note that the above name conventions do not apply to ProDOS, since all source files under ProDOS are text files.

NOTE: "Insert" refers to the effect on assembly and not to the *location* of the source. The file itself is actually placed just following the main source. These files are only in memory one at a time, so a very large program can be assembled using the PUT facility.

There are two restrictions on a PUT file. First, there cannot be macro definitions inside a file which is PUT; they must be in the Master source file or in a USE file. Second, a PUT file may not call another PUT file with the PUT opcode. Of course, linking can be simulated by having the Master program just contain the macro definitions and call, in turn, all the others with the PUT opcode.

Any variables, such as]LABEL, may be used as "local" variables. The usual local variables]1 through]8 may be set up for this purpose using the VAR opcode.

The PUT facility provides a simple way to incorporate often used subroutines, such as SENDMSG or PRDEC, in a program.

USE (USE a text file as a macro library)

USE filename

```
USE MACRO LIBRARY    [DOS 3.3 example]
USE !MACROS          [DOS 3.3, no "T." prefix]
USE MACROS,S5,D1     [DOS 3.3 with slot/drive]
USE /LIB/MACROS      [ProDOS pathname]
```

This works similarly to PUT but the file is kept in memory. It is intended for loading a macro library that is USEd by the source file.

It can also be used for including a common library of equates in source files to avoid having to type them into every new program you write. For example, this equate file:

```
**********************
* COMMON EQUATE FILE *
**********************

HOME    EQU  $FC58      ; MONITOR CLEAR SCREEN ROUTINE
VTAB    EQU  $FC22      ; MONITOR VERTICAL TAB ROUTINE
CH      EQU  $24        ; HORIZ. CURSOR POSITION
 Etc...
```

Could be included in every program you write using the USE command:

```
**********************
*   SAMPLE PROGRAM   *
**********************

PTR    EQU  $06         ; POINTER FOR MY PROGRAM
       USE  EQUATES     ; USE PRE-DEFINED EQUATES
BEGIN JSR   HOME        ; CLEAR SCREEN (USE HOME LABEL)
Etc.
```

Normally, the assembled listing will print out all the labels defined in the EQUATES file, but you could use LST ON and LST RTN at the beginning and end of the EQUATES file to suppress the listing of just the defined labels.

VAR (setup VARiables)

```
VAR expr;expr;expr...
   VAR 1;$3;LABEL          [ set up VAR's 1,2 and 3 ]
```

This is a convenient way to equate the variables]1 -]8. For example, VAR 3;$42;LABEL will set]1 = 3,]2 = $42, and]3 = LABEL. This is designed for use just *prior* to a PUT. If a PUT file uses]1 -]8, except in lines for calling macros, there *must* be a previous declaration of these.

SAV (SAVe object code)

```
SAV filename
   SAV  FILE               [ ProDOS or DOS 3.3 syntax ]
   SAV  /OBJ/PROG          [ ProDOS pathname syntax ]
```

SAV filename will save the current object code under the specified name. This acts the same as the Main Menu object saving command, but it can be done several times during assembly.

This pseudo-op provides a means of saving portions of a program having more than one ORG. It also enables the assembly of extremely large files. After a save, the object address is reset to the last specification of OBJ or to $8000 by default.

Files saved with the SAV command will be saved to BLOAD at the correct address.

SAV allows you to save sections of assembled object code during an assembly. It saves all assembled code in the current assembly at the point at which the SAV opcode occurs. This applies *only* to the first SAV in a source. With each additional SAV, Merlin 8/16 only saves the object code generated since the last SAV. This feature allows you to use one source file to assemble code and then SAV sections in separate files. Together with PUT and DSK, SAV makes it possible to assemble extremely large files.

For example, suppose you have a program that uses Hi-Res graphics and is located in memory at two different places. The first part is located at $800 and the second part is at $6000. Your program is divided this way because it is 16K bytes long and thus the Hi-Res pages fall in the middle of your program.

When you first assembled your program you didn't realize the Hi-Res pages were a problem. Your program worked for about two seconds, but when it cleared the Hi-Res screens, it bombed to the Monitor. Clearing the Hi-Res screens also cleared your program! What do you do now?

Just determine in your program where address $2000 is, since this is the start of the Hi-Res Page 1. Once you find this point, it is a simple matter to put in a JMP opcode, follow it immediately with an ORG to $6000, then reassemble the program. You look at the assembly listing and sure enough,

all of the code that used to reside at $2000 is shown at $6000. Then you run your program and it crashes again!

You go into the Monitor and find that none of your code is at $6000. It's just a bunch of hex garbage! The answer is that when more than one ORG statement is used, Merlin 8/16 *does not* physically move the generated code to the new address, it adds it to the end of the previous code. Therefore, the code that should have started at $6000 was *assembled* with all of its *addresses* correct for $6000, but its actual *location* was still down at $2000.

Merlin 8/16 SAV's the day! You need to assemble your source as one file since the two sections refer to each other, but each section needs to be put in different memory locations. The answer is to assemble the entire file with SAV's. Each section will be saved as a binary file with the proper load address. Thus in the following example, when the entire file is assembled, two binary files will be generated and saved. The first will be called FILE1 and will have a load address of $800. The second will be called FILE2 and will have a load address of $6000.

Therefore, *SAV is used to save sections of code to separate individual binary files during an assembly*. With SAV, you can assemble code that may not be continuous in memory but which must be assembled all at once because the sections refer to each other, and may share labels, data, and/or subroutines.

See the example of the multiple-ORG files using SAV at the beginning of this section for an illustration of the SAV command.

NOTE: The Linker provides an alternate way of acheiving this same result. A linker is used more often for large programs because the each segment can be individually created, assembled, and then linked into the final program without re-assembling the other segments, thus saving time during program development.

TYP (set ProDOS file type for DSK and SAV)

TYP expression

```
TYP  $00          [ no file type ]
TYP  $06          [ binary file type ]
```

This sets the file type to be used by the DSK or SAV opcodes. The default is the BIN type. Valid file types for Merlin 8 are 0,6,$F0-$F7, and $FF (no type, BIN, CMD, user defined, SYS). In Merlin 16, there are no restrictions on the filetypes available.

DSK (assemble directly to DiSK)

DSK filename (or pathname for ProDOS)

```
DSK PROG              [ DOS 3.3 or ProDOS ]
DSK /OBJ/PROG         [ ProDOS pathname example ]
```

"DSK filename" will cause Merlin 8/16 to open a file specified in the opcode and place all assembled code in that file. It is used at the *start* of a source file before any code is generated. Merlin 8/16 then writes all the following code directly to disk. If DSK is already in effect, the old file will be closed and the new one opened. This is useful primarily for extremely large files.

NOTE: Files intended for use with the Linker must be saved with the DSK pseudo op. See the REL opcode for details.

The DSK opcode has three basic purposes:

1) It allows you to assemble programs that result in object code larger than Merlin 8/16 can normally keep in memory.

2) It allows you to automatically put your object code on disk without having to remember to use the Main Menu Object save command.

3) It is used in conjunction with the TYP command to create object files with a filetype other than BIN.

The first purpose is the most often used reason for utilizing the DSK opcode.

NOTE: Using DSK will slow assembly significantly. This is because Merlin 8/16 will write a sector to disk every time 256 bytes of object code have been generated. If you don't need a copy of the object code on disk, you should not use the DSK opcode, or use a conditional to defeat it. This is illustrated in the APPLESOFT.S source also.

The assembly speed of source programs that use DSK, PUT, USE or SAV can be improved significantly by putting the referenced files on a RAM disk.

Here is an example listing of a program that creates two separate object files using the DSK command:

```
1  * DSK SAMPLE *
2            DSK   FILEONE     ;ASSEMBLE 'FILEONE' TO DISK
3            ORG   $300        ;'FILEONE' AT $300 (CALL 768)
4   COUT     EQU   $FDED
5   HOME     EQU   $FC58
6            JSR   HOME
7            LDX   #0
8   LOOP1    LDA   STRING1,X
```

```
 9              BEQ   DONE1
10              JSR   COUT
11              INX
12              BNE   LOOP1
13   DONE1      RTS
14   STRING1    ASC   "THIS IS ONE"
15              HEX   8D00
16
17              DSK   FILETWO     ;ASSEMBLE 'FILETWO' TO DISK
18              ORG   $8000       ;'FILETWO' AT $8000 (CALL 32768)
19               .
20              LDX   #0
21   LOOP2      LDA   STRING2,X
22              BEQ   DONE2
23              JSR   COUT
24              INX
25              BNE   LOOP2
26   DONE2      RTS
27   STRING2    ASC "NOW IT'S TWO"
28              HEX 8D00
```

This can be tested with the following Applesoft program.

```
10  PRINT  CHR$ (4);"BLOAD FILEONE"
15  CALL 768
20  PRINT  CHR$ (4);"BLOAD FILETWO"
25  CALL 32768
30  END
```

When run, the following text should appear on the screen:

```
THIS IS ONE
NOW IT'S TWO
```

END (END of source file)

END
    END [only option for this opcode]

This rarely used or needed pseudo opcode instructs the assembler to ignore the rest of the source. Labels occurring after END will not be recognized.

DUM (DUMmy section)

DUM expression
    DUM $1000 [start DUMmy code at $1000]
    DUM LABEL [start code at value of LABEL]
    DUM END-START [start at val of END-START]

This starts a section of code that will be examined for the values of labels but will produce no object code. The expression must give the desired ORG of this section. It is possible to re-ORG such a section using another DUMMY opcode or using ORG. Note that although no object code is produced from a dummy section, the text output of the assembler will appear as if code is being produced, so you can see the addresses as they are referenced.

DEND (Dummy END)

DEND
DEND [only option for this opcode]

This ends a dummy section and re-establishes the ORG address to the value it had upon entry to the dummy section.

DUM and DEND are used most often to create a set of labels that will exist outside of your program, but that your program needs to reference. Thus, the labels and their values need to be available, but you don't want any code actually assembled for that particular part of the listing.

Sample usage of DUM and DEND:

```
 1          ORG  $1000
 2
 3 IOBADRS =     $B7EB
 4
 5          DUM  IOBADRS
 6 IOBTYPE DFB   1
 7 IOBSLOT DFB   $60
 8 IOBDRV  DFB   1
 9 IOBVOL  DFB   0
10 IOBTRCK DFB   0
11 IOBSECT DFB   0
12          DS   2          ;pointer to DCT
13 IOBBUF  DA    0
14          DA   0
15 IOBCMD  DFB   1
16 IOBERR  DFB   0
17 ACTVOL  DFB   0
18 PREVSL  DFB   0
19 PREVDR  DFB   0
20          DEND
21
22 START   LDA   #SLOT
23          STA  IOBSLOT
24 *  And so on
```

Note that no code is generated for lines 5 through 20, but the labels are available to the program itself, for example, on line 23.

FORMATTING PSEUDO OPS

AST (send a line of ASTerisks)

AST expression

AST 30	[send 30 asterisks to listing]
AST NUM	[send NUM asterisks]

This sends a number of asterisks (*) to the listing equal to the value of the operand. The number format is base 10, so that AST10 will send decimal 10 asterisks, for example. The number is treated modulo 256 with 0 being 256 asterisks.

CYC (calculate and print CYCle times for code)

CYC
CYC OFF
CYC AVE
CYC FLAGS

CYC	[print opcode cycles & total]
CYC OFF	[stop cycle time printing]
CYC AVE	[print cycles & average]
CYC FLAGS	[print cycles & current mx flag status - Merlin 16 only]

This opcode will cause a program cycle count to be printed during assembly. A second CYC opcode will cause the accumulated total to go to zero. CYC OFF causes it to stop printing cycles. CYC AVE will average in the cycles that are underterminable due to branches, indexed and indirect addressing.

The cycle times will be printed or displayed to the right of the comment field and will appear similar to any one of the following:

```
5  ,0326      or       5' ,0326     or     5'',0326
```

The first number displayed, the 5 in the example above, is the cycle count for the current instruction. The second number displayed is the accumulated total of cycles in decimal.

An apostrophe or single quote after the cycle count indicates a possible added cycle, depending on certain conditions the assembler cannot forsee. If this appears on a branch instruction then it indicates that one cycle should be added if the branch occurs. For non-branch instructions, the single quote indicates that one cycle should be added if a page boundary is crossed.

A double quote after the cycle count indicates that the assembler has determined that a branch would be taken and that the branch would cross a page boundary. In this case the extra cycle is displayed and added to the total.

The CYC opcode will also work for the extra 65C02 opcodes in Merlin 8/16. It will not work for the additional 65C02 opcodes present in the Rockwell 65C02, i.e. RMB#, SMB#, BBR# and BSS#. These opcodes are not supported by Merlin 8/16, except when USEing the ROCKWELL macro library. All of these unsupported opcodes are 5-cycle instructions with the usual possible one or two extra cycles for the branch instructions BBS and BBR.

In Merlin 8, the CYC opcode will also work for the 65802 opcodes, but it will *not* add the extra cycles required when M=0 or when X=0. In Merlin 16, there is an additional option, CYC FLAGS, that will print out the current assembler status of the registers sizes, M and X. This can be useful for verifying that register states are as you want them throughout a listing. The CYC function in Merlin 16 *does* correctly take the M and X bits into account when calculating cycle times.

DAT (DATe stamp assembly listing - ProDOS only)

```
DAT
   DAT                    [ only option for this opcode ]
```

This prints the current date and time on the second pass of the assembler. Available only in the ProDOS versions of Merlin 8/16.

EXP ON/OFF/ONLY (macro EXPand control)

```
EXP ON or OFF or ONLY
   EXP ON                 [ macro exapand on ]
   EXP OFF                [ print only macro call ]
   EXP ONLY               [ print only generated code ]
```

EXP ON will print an entire macro during the assembly. The OFF condition will print only the PMC pseudo-op. EXP defaults to ON. This has no effect on the object code generated. EXP ONLY will cause expansion of the macro to the listing omitting the call line and end of macro line. However, if the macro call line is labeled, it is printed. This mode will print out just as if the macro lines were written out in the source.

LST ON/OFF/RTN (LiSTing control)

LST ON or OFF or RTN

LST ON	[turn listing on]
LST OFF	[turn listing off]
LST	[turn listing on, optional]
LST RTN	[return LST state to that in effect before previous LST command. Merlin 16 only]

This controls whether the assembly listing is to be sent to the Apple screen, or other output device, or not. For example, you may use this to send only a portion of the assembly listing to your printer. Any number of LST instructions may be in the source. If the LST condition is OFF at the end of assembly, the symbol table will not be printed.

The assembler actually only checks the third character of the operand to see whether or not it is a space. Therefore, LST will have the same effect as LST ON. The LST directive will have no effect on the actual generation of object code. If the LST condition is OFF, the object code will be generated much faster, but this is recommended only for debugged programs.

LST RTN is available in Merlin 16 only and will return the LST status to what it was previous to the last instance of LST. For example, if a macro library had LST OFF at the beginning and LST RTN at the end, the library would not be listed in an assembly, but in addition, the list status of the main source file, either on or off, would not be disturbed by the included LST commands of the macro library.

NOTE: Control-D from the keyboard toggles this flag during the second pass, and thus can be used to manually turn on or off the screen or printer listing during assembly.

LSTDO or LSTDO OFF (LiST DO OFF areas of code)

LSTDO
LSTDO OFF

LSTDO	[list the DO OFF areas]
LSTDO OFF	[don't list DO OFF areas]

This opcode causes the listing of DO OFF areas of code to be printed in listings or not to be printed.

PAG (new PAGe)

PAG
```
PAG                 [ only option for this opcode ]
```

This sends a formfeed, i.e. $8C, to the printer. It has no effect on the screen listing even when using an 80 column card.

TTL (define Title heading - Merlin 16 only)

TTL string
```
TTL "Segment Title"     [ only option for this opcode ]
```

This has the same syntax as the ASC pseudo op, and sets the page title in use by the PRTR command. This is used for changing the title at the top of the page during a source listing printout, and is usually followed by a PAG pseudo op.

SKP (SKiP lines)

SKP expression
```
SKP  5              [ skip 5 lines in listing ]
SKP  LINES          [ skip "LINES" lines in listing]
```

This sends the number of carriage returns in "expression" to the listing. The number format is the same as in AST.

TR ON/OFF (TRuncate control)

TR ON or OFF or ADR
```
TR  ON              [ limit object code printing ]
TR  OFF             [ don't limit object code print]
TR  ADR             [ suppress bank byte of addresses - Merlin 16 only ]
```

TR ON or just TR limits object code printout to three bytes per source line, even if the line generates more than three. TR OFF resets it to print all object bytes.

TR ADR can be used in Merlin 16 to suppress the bank byte part of the address listing at the left of an assembly listing.