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.

STRING DATA PSEUDO OPS

GENERAL NOTES ON STRING DATA AND STRING DELIMITERS

Different delimiters have different effects. Any delimiter with an ASCII value less than the apostrophe (') will produce a string with the high-bits on, otherwise the high-bits will be off. For example, the delimiters !"#$%& will produce a string in *negative* ASCII, and the delimiters '()+? will produce one in *positive* ASCII. The quote (") and apostrophe (') are the usual delimiters of choice, but other delimiters provide the means of inserting a string containing the quote or apostrophe as part of the string. Example delimiter effects:

```
"HELLO"                 [ negative ASCII, hi bit set ]
!HELLO!                 [ negative ASCII, hi bit set ]
#HELLO#                 [ negative ASCII, hi bit set ]
&HELLO&                 [ negative ASCII, hi bit set ]
!ENTER "HELLO"!         [ string with embedded quotes - negative ASCII]
'HELLO                  [ positive ASCII, hi bit clear ]
(HELLO(                 [ positive ASCII, hi bit clear ]
'ENTER "HELLO"'         [ string with embedded quotes - positive ASCII ]
```

All of the opcodes in this section, except REV, also accept hex data after the string. Any of the following syntaxes are acceptable:

```
ASC "string"878D00
FLS "string",878D00
DCI "string",87,8D,00
STR "STRING",878D00
INV "string",878D00
```

ASC (define ASCii text)

```
ASC d-string
   ASC  "STRING"          [ negative ASCII string ]
   ASC  'STRING'          [ positive ASCII string ]
   ASC  "Bye,Bye",8D      [ negative with added hex bytes]
```

This puts a delimited ASCII string into the object code. The only restriction on the delimiter is that it does not occur in the string itself.

DCI (Dextral Character Inverted)

DCI d-string

```
DCI "STRING"          [ negative ASCII, except for the "G" ]
DCI 'STRING'          [ positive ASCII, except for the "G" ]
DCI 'Hello',878D      [ positive with two added hex bytes ]
```

This is the same as ASC except that the string is put into memory with the last character having the opposite high bit to the others. When a hex suffix is added, the bit inversion will still be on the last character of the *delimited* string. Thus, only the 'o' in Hello will have the high bit inverted.

INV (define INVerse text)

INV d-string

```
INV "STOP!"           [ negative ASCII, inverse on printing]
INV 'END',878D        [ positive, added bytes ]
```

This puts a delimited string in memory in inverse format.

FLS (define FLaShing text)

FLS d-string

```
FLS "The End"         [ negative ASCII, flash on printing]
FLS 'The End',8D00    [ positive, flash with added bytes ]
```

This puts a delimited string in memory in flashing format.

REV (REVerse)

REV d-string

```
REV "Insert"          [ negative ASCII, reversed in memory ]
REV 'Insert"          [ same as above but positive ]
```

This puts the d-string backwards in memory. Example:

```
REV "DISK VOLUME"
```

gives EMULOV KSID (delimiter choice as in ASC). HEX data may *not* be added after the string terminator.

STR (define a STRing with a leading length byte)

```
STR d-string
   STR  "/PATH/"          [ positive ASCII, (ProDOS pathname?)]
   STR  "HI"              [ result= 02 C8 C9 ]
   STR  'HI',8D           [ result= 02 48 49 8D ]
```

This puts a delimited string into memory with a leading length byte. Otherwise it works the same as the ASC opcode. This facility is mainly intended for use with ProDOS which uses this type of data extensively.

Note that following HEX bytes, if any, are not counted in the length. Thus, although the third example above will not generate an error, it should not be used since any hex bytes appended to the end of a defined string would not be printed or otherwise recognized by a routine using the length byte as part of the descriptor for the string data.

DATA AND STORAGE ALLOCATION PSEUDO OPS

DA or DW (Define Address or Define Word)

DA expression or DW expression

```
DA $FDF0          [ results: F0 FD in memory ]
DA 10,$300        [ results: 0A 00 00 03 ]
DW LAB1,LAB2      [ example of use with labels ]
```

This stores the two-byte value of the operand, usually an address, in the object code, low-byte first.

These two pseudo ops also accept multiple data separated by commas (such as DA 1,10,100).

DDB (Define Double-Byte)

DDB expression

```
DDB $FDED+1       [ results: FD EE in memory ]
DDB 10,$300       [ results: 00 0A 03 00 ]
```

As above with DA, but places high-byte first. DDB also accepts multiple data (such as DDB 1,10,100).

DFB or DB (DeFine Byte or Define Byte)

DFB expression or DB expression

```
DFB 10            [ results: 0A in memory ]
DFB $10           [ results: 10 in memory ]
DB >$FDED+2       [ results: FD in memory ]
DB LAB            [ example of use with label ]
```

This puts the byte specified by the operand into the object code. It accepts several bytes of data, which must be separated by commas and contain no spaces. The standard number format is used and arithmetic is done as usual.

The pound sign (#) is acceptable but ignored, as is the less-than sign (<). The greater-than sign (>) may be used to specify the hi-byte of an expression, otherwise the low-byte is always taken. The > should appear as the first character only of an expression or immediately after #. That is, the instruction DFB >LAB1-LAB2 will produce the hi-byte of the value of LAB1-LAB2.

For example:

```
DFB $34,100,LAB-LAB2,%011,>LAB1-LAB2
```

is a properly formatted DFB statement which will generate the hex object code:

34 64 DE 0B 09

assuming that LAB1=$81A2 and LAB2=$77C4.

ADR (Define Long Address - 3 bytes - Merlin 16 only)

ADR expression

ADR $01FDF0	[results: F0 FD 01 in memory]
ADR 10,$020300	[results: 0A 00 00 00 03 02]
ADR LAB1,LAB2	[example of use with labels]

This stores the three-byte value of the operand, usually an address, in the object code, low-byte, hi-byte, then bank byte. This pseudo op also accepts multiple data separated by commas such as ADR 1,10,100).

ADRL (Define Long Address - 4 bytes - Merlin 16 only)

ADRL expression

ADRL $01FDF0	[results: F0 FD 01 00 in memory]
ADRL 10,$020300	[results: 0A 00 00 00 00 03 02 00]
ADRL LAB1,LAB2	[example of use with labels]

This stores the four-byte value of the operand, usually an address, in the object code, low-byte, hi-byte, bank byte, then hi-byte of the high word. This pseudo op also accepts multiple data separated by commas (such as ADR 1,10,100).

The decision as to whether to use ADR or ADRL will depend largely on whether the addresses defined are to be used as an indirect pointer (for example, JSR [PTR]), or as a address to be accessed with two LDA type instructions (LDA LABEL, LDA LABEL+2).

HEX (define HEX data)

HEX hex-data

HEX 0102030F	[results: 01 02 03 0F in memory]
HEX FD,ED,C0	[results: FD ED C0 in memory]

This is an alternative to DFB which allows convenient insertion of hex data. Unlike all other cases, the $ is not required or accepted here. The operand should consist of hex numbers having two hex digits, thus you would use 0F, not F. They may be separated by commas or may be adjacent. An error message will be generated if the operand contains an odd number of digits or ends in a comma, or in any case, contains more than 64 characters.

DS (Define Storage)

```
DS experesssion
DS expression1, expression2
DS \
   DS 10              [ zero out 10 bytes of memory ]
   DS 10,$80          [ put $80 in 10 bytes of memory ]
   DS \               [ zero memory to next memory page ]
   DS \,$80           [ put $80 in memory to next page ]
   DS \,expression2   [ put expresson2 in memory to next page ]
```

This reserves space for string storage data. It zeros out this space if the expression is positive. DS 10, for example, will set aside 10 bytes for storage.

Because DS adjusts the object code pointer, an instruction like DS-1 can be used to back up the object and address pointers one byte.

The first alternate form of DS, with two expressions, will fill expression1 bytes with the value of the low-byte of expression2, provided expression2 is positive. If expression2 is missing, 0 is used for the fill.

The second alternate form, DS \, will fill memory with zeroes until the next memory page. The "DS \,expression2" form does the same but fills using the low-byte of expression2.

Notes for REL files and the Linker

The back slash (\) options are intended for use mainly with REL files and work slightly differently with these files. Any DS \ opcode occurring in a REL file will cause the linker to load the next file at the first available page boundary, and to fill with zeroes or the indicated byte. Note that for REL files, the location of this code has *no effect* on its action. *To avoid confusion, you should only use this code at the end of a file.*

USING DATA TABLES IN PROGRAMS

Merlin's various data commands are used by the programmer to store pure data bytes, as opposed to executable program instruction bytes, in memory for use by the program. As an example, here is a program that prints the square of three numbers.

```
1    * DATA TABLE DEMO *
2
3             ORG   $8000
4
5    HOME     EQU   $FC58
6    COUT     EQU   $FDED
7
8    START    JSR   HOME        ;CLEAR SCREEN
9             LDY   #0          ;SET Y TO ZERO
10
11   PRINT1   LDA   DATA1,Y     ;PRINT NUMBER TO BE SQUARED
12            JSR   COUT
13            LDX   #0          ;SET X TO ZERO
14   LOOP1    LDA   DATA2,X     ;LOOP TO PRINT TEXT
15            BEQ   PRINT2
16            JSR   COUT
17            INX
18            BNE   LOOP1
19   PRINT2   LDA   DATA3,Y     ;PRINT SQUARED VALUE
20            JSR   COUT
21            LDA   #$8D
22            JSR   COUT
23            INY
24            CPY   #$03        ;ARE 3 LOOPS COMPLETED?
25            BCS   DONE        ;IF SO WE'RE DONE
26            JMP   PRINT1      ;IF NOT BEGIN AGAIN
27   DONE     RTS
28   DATA1    DFB   #177,178,179
29   DATA2    ASC   " SQUARED IS "
30            HEX   00
31   DATA3    DFB   #177,180,185
```

Notice how the data portion of the program (DATA1, DATA2, DATA3) is referenced in the main body of the program. Also notice that for the purpose of illustration several data definition styles have been used. The actual numbers printed by the program (example, 3 SQUARED IS 9) are stored in the program as defined bytes (DFB) on lines 28 and 31. This could just as easily been done with the ASC pseudo-op. The pseudo-op HEX is also used on line 30 to create the zero byte that terminates the string " SQUARED IS ".

CONDITIONAL PSEUDO OPS

DO (DO if true)

DO expression

```
DO 0          [ turn assembly off ]
DO 1          [ turn it on ]
DO LABEL      [ if LABEL<>0 then on ]
DO LAB1/LAB2  [ if LAB1<LAB2 then off ]
DO LAB1-LAB2  [ if LAB1=LAB2 then off ]
DO LABEL-1    [ if LABEL = 0, only if LABEL = 0 or 1 ]
```

This together with ELSE and FIN are the conditional assembly pseudo ops. If the operand evaluates to zero, then the assembler will stop generating object code (until it sees another conditional). See the section on "Building Expressions" for more examples of testing for certain values. Except for macro names, it will not recognize any labels in such an area of code. If the operand evaluates to a non-zero number, then assembly will proceed as usual. This is very useful for macros.

It is also useful for sources designed to generate slightly different code for different situations. For example, if you are designing a program to go on a ROM chip, you would want one version for the ROM and another with small differences as a RAM version for debugging purposes. Conditionals can be used to create these different object codes without requiring two sources.

Similarly, in a program with text, you may wish to have one version for Apples with mousetext characters and one for those without. By using conditional assembly, modification of such programs becomes much simpler, since you do not have to make the modification in two separate versions of the source code.

Every DO should be terminated somewhere later by a FIN and each FIN should be preceded by a DO. An ELSE should occur only inside such a DO/FIN structure. DO/FIN structures may be nested up to eight deep, possibly with some ELSE's between. If the DO condition is off, i.e. value 0, then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO/FIN structures are valuable for putting conditionals in macros.

ELSE (ELSE do this)

ELSE

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

This inverts the assembly condition for the last DO. Thus, ON becomes OFF and OFF becomes ON.

IF (IF so then do)

IF char,]var (IF char is the first character of]var)
IF MX plus expression

```
IF (,]1          [ if first char of ]1 is "(" then assemble following code]
IF ",]TEMP       [ if first char is ", assem ]
IF "=]1          [ alternate use with "=" ]
IF MX            [ if MX = 1, 2 or 3; Merlin 16 only ]
```

This checks to see if *char* is the leading character of the replacement string for]var. IF cannot be used for testing whether a label is equal to a value, etc. Use the DO pseudo-op for value tests.

NOTE: Position is important since the assembler checks the first and third characters of the operand for a match. If a match is found then the following code will be assembled. As with DO, this must be terminated with a FIN, with optional ELSEs between. The comma is not examined, so any character, such as the equal sign, may be used there. For example:

```
IF "=]1
```

could be used to test if the first character of the variable]1 is a double quote (") or not, perhaps needed in a macro which could be given either an ASCII or a hex parameter.

In Merlin 16, IF can be used to check the status of the assembler M & X bits. MX is interpreted as though it has a value in the range 0-3, depending on the current MX flag. The MX can then be included in an expression to control a conditional assembly. This is intended for use in macros to determine register length. For example:

```
IF MX/2          ; DO if M is short
IF MX/2-1        ; DO if M is long
IF MX&1          ; DO if X is short
IF MX&1-1        ; DO if X is long
IF MX/3          ; DO if both M and X are short
IF MX!3/3        ; DO if both M and X are long
IF MX-2/-1       ; DO if M is long and X is short
IF MX-3/-1       ; DO if M is short and X is long
IF MX+1&3        ; DO if either M or X or both are long
IF MX            ; DO if either M or X or both are short
```

FIN (FINish conditional)

FIN

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

This cancels the last DO or IF and continues assembly with the next highest level of conditional assembly, or it cancels ON if the FIN concluded the last or outer DO or IF.

USING CONDITIONAL ASSEMBLY

Here's a short example that shows how different program segments can be controlled with conditional assembly:

```
*********************************
* CONDITIONAL ASSEMBLY EXAMPLE *
*********************************

HOME    EQU  $FC58        ; MONITOR CLEAR SCREEN ROUTINE
COUT    EQU  $FDED        ; MONITOR PRINT ROUTINE
BELL    EQU  $FBDD        ; MONITOR "BELL" ROUTINE

FLAG    EQU  1            ; FLAG = 1 = DO THIS VERSION
                          ; IN YOUR PROGRAMS, FLAG CAN HAVE ANY NAME AND
                          ; HAVE WHATEVER RANGE OF VALUES YOU NEED FOR THE
                          ; NUMBER OF POSSIBLE ASSEMBLIES YOU WISH.

BEGIN   JSR  HOME         ; CLEAR SCREEN - ALL PROGRAMS DO THIS

        DO   FLAG         ; ASSEMBLE THIS PART IF FLAG = 1

PART1   LDA  #"A"
        JSR  COUT         ; PRINT LETTER "A"

        ELSE              ; DO PART2 IF FLAG = 0

PART2   LDA  #"B"
        JSR  COUT         ; PRINT LETTER "B"

        FIN               ; END OF CONDITIONAL SEGMENT

BELL    JSR  BELL         ; RING BELL IN ALL VERSIONS

        DO   FLAG-1       ; DO NEXT PART IF FLAG = 0
                          ; THIS SHOWS HOW TO DO INVERSE LOGIC OF 'FLAG'
                          ; (ASSUMES FLAG = 0 OR FLAG = 1)

PART2A  LDA  #"b"
        JSR  COUT         ; PRINT LETTER "b"

        ELSE              ; DO THIS PART IF FLAG = 1

PART1A  LDA #"a"
        JSR  COUT         ; PRINT LETTER "a"

        FIN               ; END OF CONDITIONAL SEGMENT

DONE    RTS               ; ALL VERSIONS END HERE
```

Using IF to test the first character of a parameter passed to a macro lets you add a variety of possible addressing modes to a macro that will depend on the input parameters. Assume we start with a simple macro to move data from one location to another:

The MOV macro moves data from]1 to]2:

```
MOV     MAC
        LDA  ]1
        STA  ]2
        <<<
```

We can then construct a more sophisticated macro that uses MOV, but which supports a wide variety of addressing modes:

The MOVD macro moves data from]1 to]2 with many available syntaxes

```
MOVD    MAC
        MOV  ]1;]2
        IF   (,]1           ; Syntax MOVD (ADR1),Y;????
        INY
        IF   (,]2           ;  MOVD (ADR1),Y;(ADR2),Y
        MOV  ]1;]2
        ELSE                ;  MOVD (ADR1),Y;ADR2
        MOV  ]1;]2+1
        FIN
        ELSE
        IF   (,]2           ;Syntax MOVD ????;(ADR2),Y
        INY
        IF   #,]1           ;  MOVD #ADR1;(ADR2),Y
        MOV  ]1/$100;]2
        ELSE                ;  MOVD ADR1;(ADR2),Y
        MOV  ]1+1;]2
        FIN
        ELSE                ;Syntax MOVD ????;ADR2
        IF   #,]1           ;  MOVD #ADR1;ADR2
        MOV  ]1/$100;]2+1
        ELSE                ;  MOVD ADR1;ADR2
        MOV  ]1+1;]2+1
        FIN                 ;MUST close ALL
        FIN                 ;conditionals, Count DOs
        FIN                 ;& IFs, deduct FINs.  Must
        <<<                 ;yield zero at end.
```

*The call syntaxes supported by MOVD are:

```
MOVD ADR1;ADR2
MOVD (ADR1),Y;ADR2
MOVD ADR1;(ADR2),Y
MOVD (ADR1),Y;(ADR2),Y
MOVD #ADR1;ADR2
MOVD #ADR1;(ADR2),Y
```

```
MOVD #ADR1;ADR2
MOVD #ADR1;(ADR2),Y
```

Here's a macro that can be created for use with the 65816 to push an immediate value on the stack using PEA, or to first load the contents of another memory location, and then push that value on the stack with a PHA. This type of operation is very common when programming on the Apple IIgs.

```
PushWord    MAC             ; DEFINE MACRO
            IF #=]1         ; IF FIRST CHARACTER OF ]1 IS A '#'
            PEA  ]1         ; PUSH VALUE OF ]1 ON STACK
            ELSE            ; OTHERWISE
            LDA  ]1         ; GET contents OF ]1
            PHA             ; PUSH THAT ON STACK
            FIN             ; END OF CONDITIONAL PART OF MACRO
            EOM             ; END OF MACRO DEFINITION
```

Thus, the PushWord macro could be used in any of these forms:

```
            PushWord #$80     ; PUSH VALUE $80 ON STACK
            PushWord #LABEL   ; PUSH VALUE LABEL ON STACK
```

or,

```
            PushWord $80      ; PUSH CONTENTS OF LOCATION $80 ON STACK
            PushWord LABEL    ; PUSH CONTENTS OF LOCATION LABEL ON STACK
```

MISCELLANEOUS PSEUDO OPS

CHK (place CHecKsum in object code)

CHK
CHK [only option for this opcode]

This places a checksum byte into object code at the location of the CHK opcode. This is usually placed at the end of the program and can be used by your program at runtime to verify the existence of an accurate image of the program in memory.

The checksum is calculated with Exclusive-ORing each successive byte with the running result. That is, byte 1 is EORed with byte 2 and the result put in the accumulator. Then that value is EORed with byte 3 and the process continued until the last byte in memory has been involved in the calculation. It is not a foolproof error checking scheme, but is adequate for most uses. If you will be publishing your source listing in a magazine, or loading object code in any situation in which you want to assure that a functional copy of the object code has been loaded, then the use of the checksum pseudo-op is recommended.

The following program segment will confirm the checksum at run time:

```
1     STARTCHK LDA   #<STARTCHK
2              STA   PTR
3              LDA   #>STARTCHK
4              STA   PTR+1
5              LDY   #$00
6              LDA   #$00
7              PHA                ; PUSH ZERO ON STACK
8
9     LOOP     PLA                ; RETRIEVE CURRENT CHKSUM
10             EOR   (PTR),Y
11             PHA                ; PUT TEMP BACK
12             INC   PTR
13             BNE   CHK          ; WRAP AROUND YET?
14             INC   PTR+1        ; YEP
15    CHK      LDA   PTR+1
16             CMP   #>PROGEND    ; SEE IF WE'RE DONE YET...
17             BCC   LOOP         ; NOT YET...
18             LDA   PTR
19             CMP   #<PROGEND
20             BCC   LOOP         ; NOPE
21             BEQ   LOOP
22    CHKCS    PLA                ; RETRIEVE CALCULATED VALUE
23             CMP   CHKSUM       ; COMPARE TO MERLIN'S VALUE
24             BNE   ERROR        ; ERROR HANDLER....
25                                ; FALL THROUGH IF O.K.
26  REALSTART  ???                ; REAL PROGRAM STARTS HERE
27             ???
```

```
...

998  PROGEND  RTS              ; END OF FUNCTIONAL PROGRAM
999  CHKSUM   CHK              ; Merlin 8/16 CHECKSUM DIRECTIVE
```

ERR (force ERRor)

ERR expression
ERR \expression

```
   ERR  ($300)-$80          [ error if $80 not in $300 ]
   ERR  *-1/$4100           [ error if PC > $4100 ]
   ERR  \$5000              [ error if REL code address exceeds $5000 ]
```

"ERR expression" will force an error if the expression has a non-zero value and the message "BREAK IN LINE ???" will be printed. This may be used to ensure your program does not exceed, for example, $95FF by adding the final line:

```
        ERR  *-1/$9600
```

NOTE: The above example would only alert you that the program is too long, and will not prevent writing above $9600 during assembly, but there can be no harm in this, since the assembler will cease generating object code in such an instance. The error occurs only on the second pass of the assembly and does not abort the assembly.

Another available syntax is:

```
        ERR ($300)-$4C
```

which will produce an error on the first pass and abort assembly if location $300 in main memory does not contain the value $4C. The primary purpose for this function is to allow your source file to check to see if a USR defined opcode routine has been loaded prior to the assembly. This *does not* check a memory location in the object code.

NOTES ON REL FILES AND THE ERR PSEUDO OP

The "ERR \expression" syntax gives an error on the second pass if the address pointer reaches expression or beyond. This is equivalent to ERR *-1/expr, but when used with REL files, it instructs the Linker to check that the last byte of the current module does not extend to expression or beyond. The expression must be absolute. If the Linker finds that the current module *does* extend beyond expression, linking will abort with a message "Constraint error:" followed by the value of expression in the ERR opcode. You can see how this works by linking the PI files which are a series of sample file on the Merlin 8/16 disks. They should be linked to an address over $81C. Note that the position of this opcode in a REL file has no bearing on its action, so that it is best to put it at the end.

KBD (define label from KeyBoarD)

label KBD
label KBD d-string
OUTPUT KBD [get value of OUTPUT from keyboard]
OUTPUT KBD "send to printer" [prompt with the d-string for the value of OUTPUT]

This allows a label to be equated from the keyboard during assembly. Any expression may be input, including expressions referencing previously defined labels, however a BAD INPUT error will occur if the input cannot be evaluated.

The optional delimited string will be printed on the screen instead of the standard "Give value for LABEL:" message. A colon is appended to the string.

KBD generated labels are used most often to control conditional assemblies. For example, this code segment asks the user to press 0 or 1 to signify which version of a program should be assembled:

```
FLAG    KBD  "Assemble Part 1 or Part 2? (0/1)"

        DO   FLAG-1          ; DO IF FLAG = 0
PART1   LDA  #"A"
        JSR  COUT            ; PRINT "A"
        FIN
        DO   FLAG            ; DO IF FLAG = 1
PART2   LDA  #"B"
        JSR  COUT            ; PRINT "B"
        FIN
DONE    RTS
```

Instead of pressing 0 or 1, you can use the fact that KBD will accept a label as input to accept a Y or N input:

```
N       EQU  0               ; NO = 0
Y       EQU  1               ; YES = 1

FLAG    KBD  "Assemble Part 1 or Part 2? (Y/N)"
```

LUP (begin a loop)

```
LUP expression  (Loop)
--^             (end of LUP)
```

The LUP pseudo-opcode is used to repeat portions of source between the LUP and the --^ "expression" number of times. An example of this is:

```
LUP 4
ASL
--^
```

which will assemble as:

```
ASL
ASL
ASL
ASL
```

and will show that way in the assembly listing, with repeated line numbers.

Perhaps the major use of this is for table building. As an example:

```
]A      =    0
        LUP  $FF
]A      =    ]A+1
        DFB  ]A
        --^
```

will assemble the table 1, 2, 3, ...,$FF.

The maximum LUP value is $8000 and the LUP opcode will simply be ignored if you try to use more than this.

NOTE: The above use of incrementing variables in order to build a table *will not* work if used within a macro. Program structures such as this must be included as part of the main program source.

In a LUP, if the @ character appears in the label column, it will be increased by the loop count, thus A,B,C...etc. Since the loop count is a countdown, these labels will go backwards, i.e. the last label has the A. This makes it possible to label items inside a LUP. This will work in a LUP with a maximum length of 26 counts, otherwise you will get a BAD LABEL error and possibly some DUPLICATE LABEL errors.

MX (long status Mode of 65802)

MX expression

MX %00	[M & X = 16 bit modes]
MX %01	[M = 16 bits, X = 8 bits]
MX %10	[X = 8 bits, M = 16 bits]
MX %11	[X = 8 bits, M = 8 bits]
MX 3	[same as MX %11]

This pseudo-op is used to inform Merlin 8/16 of the intended status of the long status of the 65802 or 65816 processor. In Merlin 8, it functions only when the assembler is in the 65802 mode, i.e. when two consetutive XC opcodes have been given. The assembler cannot determine if the processor is in 16 bit memory mode (M status bit=0) or 16 bit index register mode (X status bit=0). The purpose of the MX opcode is to inform the assembler of the current status of these bits.

Three of the above examples use binary expressions as the operand of the MX opcode. Note that any valid expression may be used as long as it is within the range of 0-3.

NOTE: This opcode *must* be used when using 65802 or 65816 instructions on either Merlin 8 or Merlin 16 to inform the assembler of the proper mode to use in order to insure proper assembly of immediate mode commands such as LDA #expression, etc.

At startup, Merlin 16 assumes the MX setting to be MX %11, that is, Emulation Mode with both Accumulator and Memory register sizes set to 8 bits, although this default can be changed in the Merlin 16 PARMS file.

PAU (PAUse)

PAU

PAU	[only option for this opcode]

On the second pass, PAU causes assembly to pause until a key is pressed. This can also be done from the keyboard by pressing the space bar. This is handy for debugging.

SW (SWeet 16 opcodes - Merlin 8 only)

SW

SW	[only option for this opcode]

This enables Sweet 16 opcodes available in Merlin 8 only. If SW, and similarly for XC, is not selected then those opcode names can be used for macros. Thus, if you are not using Sweet 16, you can use macros named ADD, SUB, etc.

USR (USeR definable op-code)

USR optional expressions
USR expression [examples depend on definition]

This is a user-definable pseudo-opcode. It does a JSR $B6DA. This location will contain an RTS after a boot, a BRUN MERLIN or BRUN BOOT ASM. To set up your routine you should BRUN it from the Main Menu as a disk command. This should just set up a JMP at $B6DA to the your main routine and then RTS.

The following flags and entry points may be used by your routine:

```
USRADS    = $B6DA  ;must have a JMP to your routine
PUTBYTE   = $E5F6  ;see below
EVAL      = $E5F9  ;see below
PASSNUM   = $2     ;contains assembly pass number
ERRCNT    = $1D    ;error count
VALUE     = $55    ;value returned by EVAL
OPNDLEN   = $BB    ;contains combined length of
                   ;operand and comment
NOTFOUND  = $FD    ;see discussion of EVAL
WORKSP    = $280   ;contains the operand and
                   ;comment in positive ASCII
```

Your routine will be called by the USR opcode with A=0, Y=0 and carry set. To direct the assembler to put a byte in the object code, you should JSR PUTBYTE with the byte in A.

PUTBYTE will preserve Y but will scramble A and X. It returns with the zero flag clear so that BNE always branches. On the first pass PUTBYTE *only* adjusts the object and address pointers, so that the contents of the registers are not important. You *must* call PUTBYTE the *same number of times* on each pass or the pointers will not be kept correctly and the assembly of other parts of the program will be incorrect!

If your routine needs to evaluate the operand, or part of it, you can do this by a JSR EVAL. The X register must point to the first character of the portion of the operand you wish to evaluate, thus set X=0 to evaluate the expression at the start of the operand. On return from EVAL, X will point to the character following the evaluated expression. The Y register will be 0, 1, or 2 depending on whether this character is a right parenthesis, a space, a comma, or the end of an operand.

Any character not allowed in an expression will cause assembly to abort with a BAD OPERAND or other error. If some label in the expression is not recognized then location NOTFOUND will be non-zero. On the second pass, however, you will get an UNKNOWN LABEL error and the rest of your routine will be ignored. On return from EVAL, the computed value of the expression will be in location VALUE and VALUE+1, low-byte first. On the first pass this value will be insignificant if NOTFOUND is non-zero.

Appropriate locations for your routine are $300-$3CF and $8A0-$8FF. You must not write to $900.

You may use zero page locations $60-$6F, but should not alter other locations. Also, you must not change any thing from $226 to $27F, or anything from $2C4 to $2FF. Upon return from your routine with an RTS, the USR line will be printed on the second pass.

When you use the USR opcode in a source file, it is wise to include some sort of check in source that the required routine is in memory.

If, for example, your routine contains an RTS at location $310 then:

```
ERR ($310)-$60
```

will test that byte and abort assembly if the RTS is not there. Similarly, if you know that the required routine should assemble exactly two bytes of data, then you can roughly check for it with the following code:

```
LABEL     USR  OPERAND
          ERR  *-LABEL-2
```

This will force an error on the second pass if USR does not produce exactly two object bytes.

It is possible to use USR for several different routines in the same source. For example, your routine could check the first operand expression for an index to the desired routine and act accordingly. Thus "USR 1, whatever" would branch to the first routine, "USR 2,stuff" to the second, etc.

In Merlin 16, the USR opcode has been extended to allow up to 10 USR opcodes, USR0 through USR9. The Merlin 8 USR is equivalent to USR0 and is upward compatible. The number 0-9 is doubled and placed in the X Register and then a JSR $B6DA is done, the standard USR vector. At $B6DA you can place a JMP (VECTORTBL,X) instruction, where VECTORTBL is a list of addresses of your routines placed at any free spot such as page 3. To use routines that would not fit on page 3, you could set the source address at $A,B higher though you may have to copy this to $E00A,$E00B, or you could set HIMEM lower. To do the latter, set both $C,D and $73,74 to the lower address.

USR routines are entered in native 8 bit mode and can be exited in any mode. The documented routines that can be called must be entered in native 8 bit mode.

An example source file with 3 USR routines is provided in the Merlin 16 file SOURCE/USR.EXAMPLE.S.

XC (eXtended 65C02, 65802 and 65816 opCodes)

```
XC
   XC                      [ enable the 65C02 option ]
   XC (twice in a row)     [ enable the 65802/65816 option ]
```

NOTE: On Merlin 8, if XC is used at the beginning of the listing, the 65C02 opcodes are enabled. If XC is used twice, that is, if it is used on the first two lines of the listing, the 65802/65816 codes can also be assembled.

On Merlin 8, some of the 65802 long addressing codes are not enabled since they have no application on the 65802. In Merlin 16, all 65816 opcodes are enabled. *With Merlin 16, you will not have to use the XC pseudo-ops unless you have altered the PARMS file to require their use.*

The XC pseudo-op will not enable the extended BIT opcodes used on the Rockwell 65C02 chip. There is, however, a macro library file included on the Merlin disk that can be USEd to implement these additional codes.

To use Sourceror to disassemble 65C02 code with the older (unenhanced) IIe ROMs, you must first BRUN MON.65C02. See the section on Sourceror for details. This utility is not needed with the newer IIe (enhanced) or IIc (Unidisk 3.5 compatible) ROMs.

Whether you are using the ProDOS or DOS 3.3 version of Merlin 8, you *must* use the XC opcode as the very first line in your code. This serves as a flag to tell Merlin 8 that you are using the 65C02 or 65802 opcodes.

You may wonder why the XC opcode is needed. After all, if simply using it on a line within a source listing enables the extended opcodes within Merlin 8/16, surely the ability to assemble the opcodes are there all along. Why burden the user with an extra requirement? The reason is in the interest of efficient de-bugging and ultimately, your sanity. Merlin 8 does its best to alert you to possible errors in a source listing, but what happens if you use 65C02 opcodes on the older 6502 microprocessor? The 6502 will perform quite unpredictably, and yet Merlin 8 can't tell what system your program ultimately is destined for, so an error is not necessarily in order.

The solution is to make the programmer deliberately set a flag signifying that he knows he's using the extended codes. That way you're less likely to get in the habit of using codes like INC (Increment accumulator directly, available on the 65C02), and then accidentally use the same opcode on a 6502.

MACROS

WHY MACROS?

Macros represent a shorthand method of programming that allows multiple lines of code to be generated from a single statement, or macro call. They can be used as a simple means to eliminate repetitive entry of frequently used program segments, or they can be used to generate complex portions of code that the programmer may not even understand.

Examples of the first type of macro call are presented throughout this manual and in the files called MACROS.S and MACROS.816.S on the Merlin 16 disk, MACROS.S on the Merlin 8 ProDOS disk, and T.MACRO LIBRARY on the Merlin 8 DOS 3.3 disk. Examples of the second, more complex type, can be found in the FPMACROS.S on the Merlin 8 ProDOS disk and in the T.FP MACROS and T.RWTS MACROS libraries found on the DOS 3.3 disk.

Macros can also be used to simulate opcodes from other microprocessors such as the Rockwell 65C02 extended bit-related opcodes, as shown in the ROCKWELL.S file on the Merlin 8 ProDOS disk and the T.ROCKWELL MACROS file on the DOS 3.3 disk.

Macros literally allow you to write your own language and then turn that language into machine code with just a few lines of source code. Some people even take great pride in how many bytes of source code they can generate with a single macro call.

MACRO PSEUDO OPS

MAC (begin MACro definition)

Label MAC

This signals the start of a macro definition. It must be labeled with the macro name. The name is then reserved and cannot be referenced by anything other than that macro pseudo-op. For example, DA NAME will not be accepted if NAME is the label assigned to MAC.

EOM (End of Macro
<<< (Alternate form)

EOM
<<< (alternate syntax)

This signals the end of the definition of a macro. It may be labeled and used for branches to the end of a macro.

PMC (Put Macro Call)
>>> (alternate form)

PMC macro-name
>>> macro-name (alternate syntax #1)
macro-name (alternate syntax #2)

This instructs the assembler to assemble a copy of the named macro at the present location. It may be labeled.

HOW A MACRO WORKS

A macro is simply a user-named sequence of assembly language statements. To create the macro, you indicate the beginning of a definition with the macro name in the label field, followed by the definition of the macro itself.

The macro definition ends with a terminator command in the opcode field of either EOM or the alternate form (<<<).

For example, suppose in your program that locations $06 and $07 need to be incremented by one, as in this listing:

```
1   INCR  INC  $06    ; INCREMENT LO BYTE
2         BNE  DONE
3         INC  $07    ; INCREMENT HIGH BYTE
4   DONE  ???         ; PROGRAM CONTINUES HERE...
```

If this is to be done a number of different times throughout the program, you could make the operation a subroutine, and JSR to it, or you could write the three lines of code every time you need it.

However, a macro could be defined to do the same thing like this:

```
1   INK   MAC         ; DEFINE A MACRO NAMED INK
2         INC  $06
3         BNE  DONE
4         INC  $07
5   DONE              ; NO OPCODE NEEDED
6         <<<         ; THIS SIGNALS THE END OF THE MACRO
```

Now whenever you want to increment bytes $06,07 in your program, you could just use the macro call:

```
100       INK         ; use the macro "INK"
```

You could also use either of these alternate forms:

```
100         PMC  INK   ; alternate form of macro call
```

or:

```
100         >>>  INK   ; alternate form of macro call
```

Now, suppose you notice that there are a number of different byte-pair locations that get incremented throughout your program. Do you have to write a macro for each one? Wouldn't it be nice if there was a way to include a variable within a macro definition? You could then define the macro in a general way, and when you use it, via a macro call, "fill in the blanks" left when you defined it. Here's a new example:

```
1   INK   MAC        ; define a macro named INK
2         INC  ]1    ; increment 1st location
3         BNE  DONE
4         INC  ]1+1  ; increment location + 1
5   DONE             ; NO OPCODE NEEDED
6         <<<        ; this signals the end of the macro
```

This can now be called in a program with the statement:

```
100         INK  $06
```

In the assembled object code, this would be assembled as:

```
100         INC  $06
100         BNE  DONE
100         INC  $07
100   DONE          ; NO OPCODE NEEDED
```

Notice that during the assembly, all the object code generated within the macro is listed with the same line number. Don't worry though, the bytes are being placed properly in memory, as will be evidenced by the addresses printed to the left in the actual assembly.

Later, if you need to increment locations $0A,0B, this would do the trick:

```
150         INK  $0A
```

In the assembled object code, this would be assembled as:

```
150         INC  $0A
150         BNE  DONE
150         INC  $0B
150   DONE            ; NO OPCODE NEEDED
```

As you can see, once a macro has been defined, you can use it just like any other assembler opcode.

Let's suppose you want to use several variables within a macro definition. No problem! Merlin 8/16 lets you use 8 variables within a macro,]1 through]8. Here's another example:

```
MOVE  MAC      ; define a macro named MOVE
      LDA ]1   ; load accum with variable ]1
      STA ]2   ; store accum in location ]2
      <<<      ; this signals the end of the macro
```

This is a macro that moves a byte or value from one location to another. In this example, the variables are]1 and]2. When you call the MOVE macro you provide a parameter list that "fills in" variables]1 and]2. What actually happens is that the assembler substitutes the parameters you provide at assembly time for the variables. The order of substitution is determined by the parameter's place in the parameter list and the location of the corresponding variable in the macro definition. Here's how MOVE would be called and then filled in:

```
MOVE  $00;$01
```

MOVE: macro being called
$00: takes place of]1 (1st variable)
$01: takes place of]2 (2nd variable)

Then, the macro will be expanded into assembly code:

```
MOVE  $00;$01
LDA   $00          {$00 in place of ]1}
STA   $01          {$01 in place of ]2}
```

It is very important to realize that *anything* used in the parameter list will be substituted for the variables. For example,

```
MOVE  #"A";DATA
```

would result in the following:

```
MOVE  #"A";DATA
LDA   #"A"
STA   DATA
```

You can get even fancier if you like:

```
MOVE  #"A";(STRING),Y
LDA   #"A"
STA   (STRING),Y
```

As illustrated, the substitution of the user supplied parameters for the variables is quite literal. It is also possible to get into trouble this way, but Merlin 8/16 will inform you with an error message if you get too carried away.

One common problem encountered is forgetting the difference between immediate mode *numbers* and *addresses*. The following two macro calls will do quite different things:

```
MOVE  10;20
MOVE  #10;#20
```

The first stores the contents of memory location 10 (decimal) into memory location 20 (decimal). The second macro call will attempt to store the *number* 10 (decimal) in the *number* 20! What has happened here is that an illegal addressing mode was attempted. If it were possible, the illegal macro call would have been expanded into something like this:

```
         MOVE  #10;#20        ; call the MOVE macro
         LDA   #10            ; nothing wrong here
         STA   #20            ; oops! can't do this!
*** BAD ADDRESS MODE ***       ; Merlin will let you know!
```

In order to use the macros provided with Merlin, or to write your own, study the macro in question and try to visualize how the required parameters would be substituted.

The number of values must match the number of variables used in the macro definition. A BAD VARIABLE error will be generated if the number of values is less than the number of variables used. No error message will be generated, however, if there are more values than variables.

Note that in giving the parameter list, the Macro is followed by a space, and then each parameter separated with a semicolon. When used in the opcode column, the macro name cannot be the same as any regular opcode or pseudo opcode, such as LDA, STA, ORG, EXP, etc. Also, it cannot begin with the letters DEND or POPD.

The PMC and >>> forms of a macro call are not subject to the above restrictions. In that case, the macro name will be in the operand column, and a comma is usually used to separate the macro from the parameter list. For example,

```
>>> MOVE,#10;#20
```

The assembler will accept some other characters in place of the comma between the macro name and the expressions in a macro call. You may use any of these characters:

. / , - (and the space character

The semicolons are required, however, between the expressions, and no extra spaces are allowed.

NOTE: When the assembler sees a macro name in the opcode field like FIND, it first looks to see if there is a macro defined by that name. If, for example, the needed macro library was not included with the USE function, or wasn't defined at the beginning of the source listing, and thus the macro was not found, then the first three characters (FIN) are taken as the opcode. If this is a legal opcode or pseudo opcode, and in this example FIN is, then it is treated as such, and no error is generated. This can be a

source of confusion, but fortunately there are few such potential conflicts in the macro definitions. This can also be avoided by using, for example, an underscore as the first character of a macro, as in _FindControl.

Macros will accept literal data. Thus the assembler will accept the following type of macro call:

```
MUV  MAC              ;    MACRO DEFINITION
     LDA  ]1
     STA  ]2
     <<<

     MUV  (PNTR),Y;DEST
     MUV  #3;FLAG,X
```

with the resultant code from the above two macro calls being:

```
MUV  (PNTR),Y;DEST      ; macro call
LDA  (PNTR),Y           ; substitute first parm
STA  DEST               ; substitute second parm
```

and,

```
MUV  #3;FLAG,X          ; macro call
LDA  #3                 ; substitute first parm
STA  FLAG,X             ; substitute second parm
```

Variables passed can be used as the operand to pseudo-ops like ASC:

```
MACRO DEFINITION          RESULTANT CODE EXAMPLE

PRINT  MAC                PRINT "Example"
       JSR  SENDMSG       JSR   SENDMSG
       ASC  ]1            ASC   "Example"
       BRK                BRK
       <<<
```

Some additional examples of the PRINT macro call:

```
PRINT  !"quote"!
PRINT  'This is an example'
PRINT  "So's this, understand?"
```

NOTE: If such strings contain spaces or semicolons, they *must* be delimited by single or double quotes. Also, literals in macros such as PRINT "A" must have the final delimiter. This is only true in macro calls or VAR statements, but it is good practice in all cases.

MORE ABOUT DEFINING A MACRO

A macro definition begins with the line:

```
Name    MAC     (no operand)
```

with Name in the label field. Its definition is terminated by the pseudo-op EOM or <<<. The label you use as Name cannot be referenced by anything other than a valid macro call: NAME, PMC NAME or >>> NAME.

Forward reference to a macro definition is not possible, and would result in a NOT MACRO error message. That is, the macro must be defined before it is called by NAME, PMC or >>>.

The conditionals DO, IF, ELSE and FIN may be used within a macro.

Although it is possible to define and invoke a macro at the same time, it is not recommended. The more common approach is to turn off assembly, define the macros to be used later, then turn assembly back on. The DO 0 ... FIN conditional assembly opcodes are used to enclose the macro definitions as follows:

```
****************** * SAMPLE PROGRAM * ******************
*
          DO 0              ;TURN OFF ASSEMBLY
*
ERROR     MAC               ;GIVE THE MACRO A NAME
          LDA #$88          ;ASCII CODE FOR CTRL-G (BELL)
          JSR COUT          ;PRINT TO SCREEN (CAUSES A BEEP)
          JSR COUT          ;DO IT AGAIN
          JSR COUT          ;ONE MORE TIME
          <<<               ;END OF MACRO *
          FIN               ;TURN ASSEMBLY BACK ON * * PROGRAM CONTINUES HERE ...
```

You can also give the EOM or <<< opcode a label so you could branch to it:

```
****************** * SAMPLE PROGRAM * ******************
*
          DO 0              ;TURN OFF ASSEMBLY
*
ERROR     MAC               ;GIVE THE MACRO A NAME
          LDA #$88          ;ASCII CODE FOR CTRL-G (BELL)
          LDY #$04
ERROR1    DEY               ;BEGIN COUNTDOWN
          BEQ FINISH        ;IF Y = 0 THEN EXIT
          JSR COUT          ;BEEP THE SPEAKER
          JMP ERROR1        ;GO BACK FOR ANOTHER
FINISH    <<<               ;END OF MACRO
*
          FIN               ;TURN ASSEMBLY BACK ON
*
* PROGRAM CONTINUES HERE ...
```

Labels inside macros are updated each time the macro NAME, PMC or >>> NAME is encountered. Error messages generated by errors in macros usually abort assembly because of possibly harmful effects.

NOTE: Such messages will usually indicate the line number of the macro call rather than the line inside the macro where the error occurred. Thus, if you get an error on a line in which a macro has been used, you should check the macro definition itself for the offending statement.

NESTED MACROS

Macros may be nested to a depth of 15. Here is an example of a nested macro in which the definition itself is nested. This can only be done when both definitions end at the same place.

```
TRDB MAC
     TR  ]1+1;]2+1
TR   MAC
     LDA ]1
     STA ]2
     <<<
```

In this example TR LOC;DEST will assemble as:

```
     LDA  LOC
     STA  DEST
```

and TRDB LOC;DEST will assemble as:

```
     LDA  LOC+1
     STA  DEST+1
     LDA  LOC
     STA  DEST
```

A more common form of nesting is illustrated by these two macro definitions:

```
CH   EQU  $24
POKE MAC
     LDA  #]2
     STA  ]1
     <<<
HTAB MAC
     POKE  CH;]1
     <<<
```

The HTAB macro could then be used like this:

```
     HTAB 20            ; htab to column 20 decimal
```

and would generate the following code:

```
LDA  #20          ; ]2 in POKE macro
STA  CH           ; ]1 in POKE macro, 1st parm
                  ; in HTAB macro
```

Flexible Variable Lists (Merlin 16 only)

Merlin 16 supports an additional macro variable,]0, which returns the number of variables in the parameter list of the macro call. This lets you create macros with a flexible input. For example, here's a macro that uses the number of input variables to decide whether to store a value just pulled off the stack:

```
PullByte  MAC          ; MACRO DEFINITION
          PLA          ; PULL BYTE (OR WORD) OFF STACK
          DO   ]0      ; IF A LABEL IS GIVEN
          STA  ]1      ; STORE VALUE FROM STACK IN ]1
          FIN          ; END OF CONDITIONAL PART
          EOM          ; END OF MACRO DEFINITION
```

Thus, the macro call:

```
          PullByte
```

could be used to pull a value off the stack and leave it in the accumulator, whereas

```
          PullByte  LABEL
```

would pull the value off the stack and then store it in location LABEL. You could even get more fancy by adding the IF MX tests to see whether one or two PLAs and STAs were needed to get a two-byte word off the stack in the 8 bit mode, as opposed to a single PLA/STA pair in the 16 bit mode.

MACRO LIBRARIES AND THE USE PSEUDO OP

There are a number of macro libraries on the Merlin 8/16 disks. These libraries are examples of how one could set up a library of often used macros.

The requirements for a file to be considered a macro library are:

1) Only Macro definitions and label definitions exist in the file.
2) The file is a text file.
3) If it is a DOS 3.3 library, the file name must be prefixed with "T."
4) The file must be accessable at assembly time, i.e. it must be in an active disk drive.

The macro libraries included with Merlin 8/16 include:

DOS 3.3	ProDOS	Macro Libary functions
T.MACRO LIBRARY	MACROS.S	Often used macros for general use.
	MACROS.816.S	General use macros for 65816 programs.
	TOOL.MACROS	Directory of IIgs Toolbox macros. Must be used with MACROS.816.S.
T.FPMACROS	FPMACROS.S	Allow easy access to Applesoft floating point math routines.
T.OUTPUT	<none>	To be used with SENDMSG.
T.PRDEC	PRDEC.S	Prints A,X in decimal.
T.ROCKWELL MACROS	ROCKWELL.S	Implements extended bit related opcodes on the Rockwell 65C02.
T.RWTS	<none>	Allow easy access to DOS 3.3's RWTS disk routines.
<none>	TOOL.EQUATES	Directory of IIgs label equates for use with toolset data structures.

Any of these macro libraries may be included in an assembly by simply including a USE pseudo op with the appropriate library name. There is no limit to the number of libraries that may be in memory at any one time, except for available memory space. See the documentation on the USE pseudo op for a discussion on its use in a program.

THE MERLIN 8/16 LINKERS

WHY A LINKER?

The linking facilities built into Merlin offer a number of advantages over assemblers without this capability:

1) Extremely large programs may be assembled in one operation.

2) Large programs may be assembled much more quickly with a corresponding decrease in development time.

3) Libraries of subroutines, i.e. for disk access, graphics, screen/modem/printer drivers, etc., may be developed and linked to any Merlin program.

4) Programs may be quickly re-assembled to run at any address.

With a linker, you can write portions of code that perform specific tasks, such as general disk I/O handler, and perform whatever testing and debugging is required. When the code is correct, it is assembled as a REL file and placed on a disk. Whenever you need to write a program that uses disk I/O you won't have to re-write or re-assemble the disk I/O portion of your new program. Just link your general disk I/O handler to your new program and away you go. This technique can be used for a variety of often-used subroutines.

Wouldn't a PUT file or macro USES library serve the same purpose? A PUT file comes the closest to duplicating the utility of REL files and the linker, but there are a few rather large drawbacks for certain programs. First, using a PUT file to add a general purpose subroutine would result in slower assembly because the entire program has to be assembled even when changes are made only to the subroutine. Second, any label definitions contained in the PUT file would be global within the entire program. This means the person writing the subroutine would have to be careful not to use a label like LOOP that might occur in one of the other modules, or in the main program itself. With a REL file, only labels defined as ENTry in the REL file, and EXTernal in the current file, would be shared by both programs. There is no chance for duplicate label errors when using the Linker. Consider the following simple example:

A REL file has been assembled that drives a plotter. There are six entry points into the driver: PENUP, PENDOWN, NORTH, SOUTH, EAST, WEST. To further illustrate the value of a linker, assume the driver was written by a friend who has moved 2000 miles from you. Your job is to write a simple program to draw a box.

The code would look something like this:

```
 1              REL                    ;RELOCATABLE CODE
 2 PENUP        EXT                    ;EXTERNAL LABEL
 3 PENDOWN      EXT                    ;ANOTHER ONE
 4 NORTH        EXT
 5 SOUTH        EXT
 6 EAST         EXT
 7 WEST         EXT
 8
 9 BOX          LDY  #00               ; INITIALIZE Y
10              JSR  PENDOWN           ; GET READY TO DRAW
11 LOOP         JSR  NORTH             ; MOVE UP
12              INY                    ; INC COUNTER
13              CPY  #100              ; 100 MOVES YET?
14              BNE  LOOP              ; NOTICE LOCAL LABEL
15              LDY  #00               ; INIT Y AGAIN
16 LOOP2        JSR  EAST              ; NOW MOVE TO RIGHT
17              INY
18              CPY  #100
19              BNE  LOOP2             ; FINISH MOVING RIGHT
20 * YOU GET THE IDEA, DO SOUTH, THEN WEST, AND DONE!
```

This simple sample program illustrates some of the power of RELocatable, linked files. Your program doesn't have to concern itself with conflicts between its labels and the REL files labels, you don't concern yourself with the location of the EXTernal labels, your program listing is only 30 to 40 lines and it is capable of drawing a box on a plotter. Also, notice that you are free to use the label LOOP because it is *local* to your module, and will not be known to any of the other modules.

In addition, changes to your module will not require re-assembly of the plotter driver. In short, with REL files and a linker, changes to large programs can be made quickly and efficiently, greatly speeding the program development process.

ABOUT THE LINKER DOCUMENTATION

There are three pseudo opcodes that deal directly with relocatable modules and the linking process. These are:

REL - instructs the assembler to generate relocatable files.
EXT - defines a label as external to the current file.
ENT - defines a label in the current file as accessable to other REL files.

There are two other pseudo opcodes that behave differently when used in a REL file, relative to a normal file. These are:

DS - define Storage opcode.
ERR - force an ERRor opcode.

Each of these five pseudo opcodes will be defined or redefined in this section as they relate to REL files. Also, an Editor command unique to REL files will be defined: LINK.

In order to use the Linker, the files to be linked must be specifed. The Linker uses a file containing the names of the files to be linked for this purpose. The format of this *linker name file* differs from DOS 3.3 and ProDOS. These differences will be illustrated here.

PSEUDO OPCODES FOR USE WITH RELOCATABLE CODE FILES

REL (generate a RELocatable code file)

REL [only options for this opcode]

This opcode instructs the assembler to generate a relocatable code file for subsequent use with the relocating Linker.

This *must* occur prior to definition of any labels. You will get a BAD REL error if not. REL files are not compatible with the SAV pseudo-op and with the Main Menu's Save Object Code command. *To get an object file to the disk you must use the* DSK *opcode for direct assembly to disk.*

There are additional illegal opcodes and procedures that are normally allowed with standard files, but not with REL files. For example, an ORG at the start of the code is not allowed. The ORG address is specified at link time. A further restriction on REL files is that multiplication, division or logical operations can be appled to absolute expressions, but not to relative ones.

Examples of absolute expressions are:

- An EQUate to an explicit address
- The difference between two relative labels
- Labels defined in DUMMY code sections

Examples of relative expressions that are not allowed are:

- Ordinary labels
- Expressions that utilize the current Program Counter (PC), like: LABEL = *

The initial reference address of a REL file is $8000. Note that this is only a fictional address, since it will later be changed by the Linker. It is for this reason that no ORG opcode is allowed.

There are some restrictions with the Merlin 8 Linker involving use of EXTernal labels in operand expressions. No operand can contain more than one external. For operands of the following form:

```
   #>expression
or
   >expression
```

where the expression contains an external, the value of the expression must be within 7 bytes of the external labels' value.

For example:

```
LDA #>EXTERNAL+8        [ illegal expression ]
DFB >EXTERNAL-1         [ legal expression ]
```

Object files generated with the REL opcode are given the file type LNK under ProDOS. This is the type that will show if the disk is cataloged by Merlin 8/16. This type is file type $F8. These restrictions do not apply to the Merlin 16 Linkers.

EXT (define a label EXTernal to the current REL module)

```
label EXT
      EXT label1, label2, etc.
PRINT EXT                    [ define label PRINT as EXT ]
      EXT LABEL1, LABEL2 [ define LABEL1 and LABEL2 and entries - Merlin 16 only ]
```

This defines the label in the label column as an external label. This means that references to this label within the source will assume the value for LABEL is an as-yet undefined address, presumably found in another module that will be ultimately linked with this source file. Any external label must be defined as an ENTry label in its own REL module, otherwise it will not be reconciled by the Linker since the label would not have been found in any of the other linked modules. The EXTernal and ENTry label concepts are what allows REL modules to communicate and use each other as subroutines, etc.

The value of the label is set to $8000 and will be resolved by the Linker. In the symbol table listing, the value of an external will be $8000 plus the external reference number ($0-$FE) and the symbol will be flagged with an X.

In Merlin 16, the EXT and ENT opcodes accept the following syntax:

```
ENT LABEL1, LABEL2, LABEL3
```

This makes it possible to declare absolute symbols as entries. Thus, if LABEL1 was an equate instead of a location in the code, then it can still be used as an external by other modules. Thus, it does not have to be equated in all the files using it. For example, if all three modules in a linked system used the labels HOME, COUT and BELL, the first module could define these labels with the usual HOME EQU $FC58, etc. equates, and then use ENT HOME, COUT, BELL to make these equates available to the other modules being linked. This would avoid having to define the labels in each of the other modules.

With this particular syntax, you *must not* use a label in the label column. That will cause the assembler to assume you are using the more usual syntax.

NOTE: Using this function instead of putting all the common equates in all the source files, most easily accomplished by a common USE file, does take more space in the Linker symbol library, and hastens the time of a memory overflow.

ENT (define a label as an ENTry label in a REL code module)

```
label ENT
      ENT  label1, label2, etc.
PRINT ENT                        [ define label PRINT as ENTry ]
      ENT LABEL1, LABEL2  [ define LABEL1 and LABEL2 and entries - Merlin 16 only ]
```

This defines the label in the label column as an ENTry label. This means that the label can be referred to by an EXTernal label in another source file somewhere. This facility allows other REL modules to use the label as if it were part of their source file. If a label is meant to be made available to other REL modules it must be defined with the ENT opcode, otherwise other modules wouldn't know it existed and the Linker would not be able to reconcile it.

The following example of a REL module segment illustrates the use of this opcode:

```
21            STA  POINTER        ;some meaningless code
22            INC  POINTER        ;for our example
23            BNE  SWAP           ;CAN BE USED AS NORMAL
24            JMP  CONTINUE
25  SWAP      ENT                 ;MUST BE DEFINED IN THE
26            LDA  POINTER        ;CODE PORTION OF THE
27            STA  PTR            ;MODULE AND NOT USED
28            LDA  POINTER+1      ;AS AN EQUated label
29            STA  PTR+1
30 * etc.
```

Note that the label SWAP is associated with the code in line 26 and that the label may be used just like any other label in a program. It can be branched to, jumped to, used as a subroutine, etc.

ENT labels will be flagged in the symbol table listing with an E.

DS (Define Storage)

```
DS \
DS \expression
   DS \                    [ skip to next REL file, fill mem with zeros to next
                             page break ]
   DS \1                   [ skip to next REL file, fill mem with the value 1 to
                             next page ]
```

When this opcode is found in an REL file, it causes the Linker to load the next file in the *linker name file* at the first available page boundary, and to fill memory either with zeros or with the value specified by the expression. If used, this opcode should only be placed at the *end* of your source file. Notice that DS *expression* , for example DS 5, still has the usual function even in REL files.

ERR (force an ERRor)

```
ERR \expression
   ERR \$4200              [ error if current code passes address $4200 ]
```

This opcode will instruct the Linker to check that the last byte of the current file does not extend to "expression" or beyond. The expression must be absoulute and not a relative expression.

If the Linker finds this is not the case, linking will abort with the message: CONSTRAINT ERROR: followed by the value of the expression in the ERR opcode.

The position of this opcode in a REL file has no bearing on its action. It is recommended that it be put at the end of a file.

You can see how this works by trying to link the sample PI files on the Merlin 8/16 disks to an address greater than $81C.

THE MERLIN 8 LINKER

In Merlin 8, linking is done by using the LINK command in the Command Mode to specify the address at which the final file will be loaded, i.e. the non-linker equivalent to the ORG function, and the name of the file which contains a list of the REL files created by the individual assemblies of the files that make up the complete application. The Linker is part of the Editor/Assembler system, and is available at any time with the LINK command.

The general use of the LINK command looks like this:

```
LINK $1000 "NAMES          [ link files in NAMES - DOS 3.3 ]
LINK $2000 "/VOL/NAMES"    [ link files in NAMES - ProDOS ]
```

The LINK command invokes the linking loader. For example, suppose you want to link the object files whose names are held in a "linker name file" called NAMES. Suppose the start address desired for the linked program is $1000. Then you would type: LINK $1000 "NAMES" and press Return. If you are using the ProDOS version, this assumes the prefix has been set. The final quote mark in the name is optional. You can use other delimiters such as the apostrophe (') or colon (;). The specified start address has no effect on the space available to the Linker.

To provide space for the Linker, any source file must be removed from memory with the NEW command. This command is only accepted if there is no source file in memory.

LINKER NAME FILES (DOS 3.3)

The linker name file is just a text file containing the file names of the REL object modules to be linked. It should be created with the Merlin editor and written to the disk with the Write Text File from the Main Menu. Remember to type a space to start the filename for the W command if you don't want the T. prefix appended to the start of the filename. Thus, if you want to link the object files named MYPROG.START, MYPROG.MID, and LIB.ROUTINE,D2, you would create a text file with these lines:

```
MYPROG.START
MYPROG.MID
LIB.ROUTINE,D2
```

In this example, you would write this to disk using the W command under the filename MYPROG.NAMES. You can use any filename you wish here; it is not required to call it NAMES. Then you would link these files with a start address of $1000 by typing NEW and then issuing the editor command as follows:

```
LINK $1000 "MYPROG.NAMES"
```

The Linker will not save the object file it creates. Instead, it sets up the object file pointers for the Main Menu Save Object Code command and returns directly to the Main Menu upon the completion of the linking process.

LINKER NAME FILES (ProDOS)

The linker name file is just a specially formatted file containing the pathnames of the LNK files to be linked. This file is most easily created by assembling a source file with the proper format, as follows:

Each pathname in the source file should be given the form STR "pathname",00

NOTE: The 00 must be include at the end. The entire source file must end with a BRK, i.e. another 00. This tells the Linker that there are no more pathnames in the file. Thus, if you want to link the LNK files names /MYDISK/START, /MYDISK/MID, AND /OTHERDISK/END, you would make a source file containing these lines:

```
STR  "/MYDISK/START,00
STR  "/MYDISK/MID",00
STR  "/OTHERDISK/END",00
BRK
```

It is best to use full pathnames as shown, but this is not required. You should then assemble this file and save the object code as, for example, /MY DISK/MYPROG/NAMES. You can use any pathname you want here; it is not necessary to have NAMES in a subdirectory nor to call it NAMES. Then you can link these files to address $803 by typing NEW and then:

```
LINK $803 "/MYDISK/MYPROG/NAMES"
```

The file type used by the Save Object Code command is always the file type used in the last assembly. Thus it is BIN unless the last assembly had a TYP opcode and then it will be that type. This will be used by the Save Object Code command after you link a group of files. That is, the Linker does not change this type. If you make a mistake and the file gets saved under a type you did not want, just assemble an empty file, which would reset the object type to BIN ($06). You will, however, have to link the files again.

THE LINKING PROCESS

Various error messages may be sent during the linking process See the ERRORS section of this manual for more information. If a DOS error occurs involving the file loading, then that error message will be seen and linking will abort. If the DOS error FILE TYPE MISMATCH occurs after the message "Externals:" has been printed then it is being sent by the Linker and means that the file structure of one of the files is incorrect and the linking cannot be done.

The messsage MEMORY IN USE may occur for two reasons. Either the object program is too large to accept, i.e. the total object size of the linked file cannot exceed about $A100, or the linking dictionary has exceeded its allotted space, i.e. it is greater than $B000 in length. Each of these possibilities is exceedingly remote.

After all files have been loaded, the externals will be resolved. Each external label referenced will be printed to the screen and will be indicated to have been resolved or not resolved. An indication is also given if an external reference corresponds to duplicate entry symbols. With both of these errors, the address of the one or two byte field effected is printed. This is the address the field will have when the final code is BLOADed.

If you use the TRON command prior to the LINK command, only the errors will be printed in the external list, i.e. NOT RESOLVED and DUPLICATE errors.

This listing may be stopped at any point using the space bar. The space bar may also be used to single step through the list. If you press the space bar while the files are loading then the Linker will pause right after resolving the first external reference.

The list can be sent to a printer by using the PRTR command prior to the LINK command. At the end, the total number of errors, i.e. external references not resolved and references to duplicate entry symbols, will be printed. After pressing a key, you will be sent to the Main Menu and can save the linked object file with the Save Object Code command, using any desired filename or pathname. You can also return to the Editor and use the GET command to move the linked code to main memory.

USING MERLIN 8 LINKED FILES

The following example shows how two source files are used to generate the LNK files that will be combined by the Linker into a final application:

```
*********************************
*   RELOCATING LINKER SAMPLE    *
*         PART ONE              *
*********************************

         REL
         DSK   FILE1.L

HOME     EQU   $FC58
COUT     EQU   $FDED

BEGIN2   EXT               ; EXTERNAL LABEL

BEGIN    JSR   HOME        ; CLEAR SCREEN

         LDX   #$00        ; INITIALIZE COUNTER
LOOP     LDA   STRING,X    ; GET CHARACTER
         BEQ   DONE        ; END OF STRING
         JSR   COUT
         INX
         BNE   LOOP        ; ALWAYS

DONE     JMP   BEGIN2      ; JUMP TO 2ND SEGMENT...

STRING   ASC   "THIS IS FILE #1"
         HEX   8D,00       ; END OF STRING
         LST   OFF
```

And this is the second part:

```
*********************************
*   RELOCATING LINKER SAMPLE    *
*         PART TWO              *
*********************************

         REL
         DSK   FILE2.L

HOME     EQU   $FC58
COUT     EQU   $FDED

BEGIN2   ENT               ; ENTRY LABEL
         LDX   #$00        ; INITIALIZE COUNTER
LOOP     LDA   STRING,X    ; GET CHARACTER
         BEQ   DONE        ; END OF STRING
         JSR   COUT
```

```
         INX
         BNE   LOOP        ; ALWAYS

DONE     RTS               ; END OF PROGRAM

STRING   ASC   "THIS IS FILE #2"
         HEX   8D,00       ; END OF STRING
         LST   OFF
```

Having entered and assembled each of the source files, FILE1.L and FILE2.L will be created on the disk. These are the intermediate REL files that will be linked to create the final application program. Now you need to create the file containing the list of files to be linked. In this example, the file will be called NAMES and it will link the FILE1.L and FILE2.L files.

For the DOS 3.3 version of Merlin 8, you would just use the editor to type in the lines:

```
FILE1.L
FILE2.L
```

There are no leading spaces; the names FILE1.L and FILE2.L go in the *label* column of the Editor. This file is then saved as a text file using the Write Text File command from the Main Menu. Remember to add a space at the beginning of the filename to save under if you wish to avoid the T. prefix in the name.

To link the file, type NEW to clear the source workspace, then type LINK $8000 "NAMES" and press Return. The file NAMES will be read to determine which files to link. If there are no errors, you will be returned to the Main Menu when the link is complete. At that point, use the Save Object Code command to save the object file to disk under the name FINAL.OBJ.

To test the program, use this Applesoft BASIC program:

```
10 TEXT: HOME
20 PRINT CHR$(4);"BLOAD FINAL.OBJ"
30 CALL 32768: REM $8000
40 VTAB 12: HTAB 15: PRINT "IT REALLY WORKS!"
50 LIST: END
```

To create the names file for the ProDOS version of Merlin 8, you will need to create a separate source file for the names list. This is because the ProDOS Linker requires that the names list be in a special format. Remember that each name in the ProDOS Merlin 8 names list must be defined with the STR pseudo-op, terminated with a zero, and that the list itself is terminated with a zero also. To create the names list, type in this text:

```
*********************************
* MERLIN 8 LINKER NAMES FILE    *
*           ProDOS              *
*********************************

         DSK  "NAMES"          ; CREATE NAMES FILE
         STR, "FILE1.L",00     ; 1ST LINK FILE
         STR  "FILE2.L",00     ; 2ND LINK FILE
         BRK                   ; ZERO TO TERMINATE LIST
```

Assemble this file, which will create the actual NAMES file for the Linker, and save the source file under the file name NAMES. This will actually be saved on the disk as NAMES.S, so you needn't worry about any confusion when you use NAMES in the Linker.

To link the files under ProDOS, type NEW to clear the workspace, and then type LINK $8000 "NAMES" and press Return. When the link is complete, use the Save Object Code command at the Main Menu and save it under the name FINAL.OBJ. You can use the same Applesoft program shown for the DOS 3.3 example to test the program.

In looking at the example, notice how the ENT and EXT pseudo-ops are used to communicate the label BEGIN2 between the two programs. Also notice how there is no conflict over the use of the labels LOOP, DONE and STRING.

Compare this example to the sample program shown for the PUT directive. Notice how the same result of combining separate source files is achieved, but without the disadvantages of PUT files discussed at the beginning of this chapter.

THE MERLIN 16 LINKERS

The Merlin 16 assembler supports three different linkers. These are separate and distinct from the built-in Linker in Merlin 8, and are as follows:

LINKER: This Linker combines multiple relocatable LNK files into a file that runs at a specified address, such as a ProDOS 8 BINary or SYStem type file. This is also referred to as the Absolute Linker, since the final output file must be run at a specified location.

LINKER.GS: This is a Linker specifically for creating Object Module Format (OMF) files to be run on the Apple IIgs under ProDOS 16 and the System Loader. On the Apple IIgs, applications have no way of knowing where in memory they will ultimately be loaded and run, and so must contain a relocation dictionary as part of the final output file. If you wish to write Apple IIgs ProDOS 16 programs, you should use Linker GS or Linker.XL, discussed next.

LINKER.XL: This is special version of Linker GS which makes two passes and links to disk to produce a multi-segment file. This feature is a trade-off with linking speed, that is, Linker XL is slower than Linker GS, and so you will normally want to use Linker GS unless you specifically require multi-segment files. A multi-segment file is where one program is broken up into multiple segments on the disk, but which ultimately all come together to form the final program in memory. This is different from a program which just happens to have other segments that it loads under its own control, such as printer drivers or program overlays.

Linker.GS is automatically loaded into memory when Merlin 16 is run, and so is available for immediate use. You can manually set up either of the other two Linkers by just typing -LINKER or -LINKER.XL as a Disk Command from the Main Menu. Alternatively, you can also change the PARMS.S file to load any version of the Linker that you wish. See the discussion of the PARMS file in the Technical Information section.

The Linkers use the same space as USER programs. LNK files that you may have created on Merlin Pro, an earlier version of Merlin 8, should be upward compatible as long as they do not have externals or entry declarations. If your programs have external or entry declarations, or you are in doubt or encounter difficulties, just re-assemble the source files with Merlin 16 to create updated LNK files.

As was discussed for the Merlin 8 Linker, creating an output file consists of several steps. First, source files are created using the Merlin editor. These are assembled and the output file (type $F8 = LNK) is created using the REL and DSK or SAV directives. This intermediate file is not usable in and of itself. Rather, it is to be used as the input for the next step, which is the actual linking of several LNK files to create the final object file. This file may be a stand-alone BIN or SYS file for ProDOS 8, or it may be an OMF file for ProDOS 16, such as a SYS16, CDA (Classic Desk Accessory), or any other ProDOS 16 loadable file.

The linking process in Merlin 16 is controlled by a *linker command file*. The linker command files are a more advanced form of the Merlin 8 NAME files, and have much more flexibility. They support comments, are able to do batch assemblies before linking, and are able to create multiple output files. Let's look at each Merlin 16 Linker:

THE ABSOLUTE LINKER (LINKER)

To start the link, you must first delete any source file in memory to provide the memory for reading the Link command file. Remember to save the file first if necessary. To start the link, type Open-Apple-O to open the Command Box, and then type LINK "FILENAME" and press Return, where FILENAME is the name of the linker command file. The LINK command from the Merlin 16 editor has slightly different syntax from the Merlin 8 Linker in that you do not specify an address. Instead, the address is provided within the command file with an ORG directive.

NOTE: The command file is just a text file looking very much like a standard source file, *but you do not assemble it.*

The command file can have comments in the usual comment format for source files. Commands to the Linker are put in the opcode field. Commands supported are:

ASM, PUT, OVR, LNK (or LINK), ORG, ADR, SAV, TYP, EXT, ENT, DAT, LKV, END, and LIB.

These have the following syntax and meanings:

ASM pathname
ASM FILENAME.S

Assemble the source file specified in pathname. The source should do a DSK or SAV to create the LNK file to ultimately be used by the Linker. All ASMs must be done *before* any other linker commands. The ASM command is a conditional operation, and only assembles those source files that have been changed since the last time the files were linked. This is a convenience feature that lets you create a command file to build a final application. Re-assemblies will only be done on just those parts of the application that have changed since your last linking.

To determine whether to do an assembly, the ASM command checks bit 0 of the "auxtype" of the source file. It does not do the assembly if this set. Otherwise it sets that bit and does the assembly. You can defeat this by zeroing the appropriate bit in the PARMS file. See the Technical Information section for details. This bit is cleared whenever you save a source file, so this will force assembly of that file. Also see the PUT command below.

PUT pathname
PUT FILENAME.S

Check and set auxtype bit 0 of this file, presumably a PUT file in the next source. If the file has been modified then force the assembly of the next ASM instruction. This should precede the ASM command for any files that use a particular PUT file, and is used to cause a re-assembly in the event you change a PUT file used by a particular master file. This command is not needed if your source files do not use PUT files, or you don't wish to use the feature.

OVR [ALL]
OVR or OVR ALL

Override the auxtype 0-bit check and force assembly of the next ASM instruction. This flag is reset by any ASM. With the OVR ALL syntax, it will force assembly of all files in the linker list, so you don't have to use OVR before each ASM instruction if you want all assemblies to be done.

LNK pathname
LNK FILENAME.L

Link the LNK file specified. Generally you will have several of these in a row.

ORG hex address
ORG $2000

Sets the run time address of the following LNKs. Must be used between each SAV and the next group of LNKs. The address will also be put in the auxtype if no ADR command follows.

NOTE: The Linkers have only hexadecimal address calculation ability. All addresses must be in hex and preceded with the $ sign.

ADR address
ADR $2000

Sets the load address of the next linked file. Must be used only after an ORG setting the run time address. The ORG automatically sets this address, so you don't have to have an ADR command if it is the same as the ORG. The *load address* is the address put in the auxtype of the file. For non-BIN files it could have some other meaning.

SAV pathname
SAV FINAL.SYSTEM

Saves the linked file. This *must* be in the command file or there will be no resulting linked file. It should come after all the LNKs for a given output file. The Absolute Linker supports up to 64 separate output files.

SAV must *only* be used after one or more LNKs. It is not to be used to save the object code after an ASM. The assembly source should do this with the SAV or DSK command, or DSK if linking is to be done and the Linker is not just being used to do batch assemblies. All ASMs must precede any LNKs and SAVs.

TYP byte
TYP $06

Sets the file type for the next linked file. If all the SAVs are to use the same type output file, this need only be used once in the command file.

EXT
EXT

Tells the Linker to print addresses of all resolved externals and not only the ones with errors. This is turned off after each SAV.

ENT
ENT

Tells the Linker to print the entry list. This should come after all the linking of all output files.

DAT
DAT

Causes the Linker to print the current date and time.

END
END

Marks end of linker command file. Optional.

LIB directory name
LIB TOOL.LIBR

If used, this should come after all LNKs and before the last SAV. It tells the Linker to look for any unresolved (at that point) external labels and search the given directory for corresponding files. The files must be LNK files of the *same name* as the entry label to which they correspond. Any such files found will be linked to the present module. Not finding a file will not cause an error because some other file linked this way may contain the entry in question. If not, an error will result when the final external resolution is done. This feature could have been made automatic, but was not because that would substantially impair performance when it is not needed. Making it an option in the command file provides more versatility.

For example, suppose one of the linked files has the label PRINT declared as an external. The Linker comes to the line LIB LIBRARY, and suppose that at that point, none of the linked files has an entry called PRINT. The Linker will look in the directory LIBRARY for a file called PRINT, and if that file is found, it will be linked to the present module. Then the Linker will search for further unresolved externals, including those from the file PRINT just linked, and act on them in a similar way.

LKV byte
LKV $02

Verifies that the cortrect version of the Linker in us. LINKER is version 0, LINKER.GS is version 1, and LINKER.XL is version 2. For example, if you want to guarantee that LINKER.XL is the Linker in use, you would put the command LKV $02 in the linker command file.

THE LINKER COMMAND FILE

The linker command file is a standard Merlin 16 source file, i.e. BUILD.S, *but it is not assembled* itself. Instead, the linker command file is executed with the LINK command.

A simple linker command file would look like this:

```
ORG  $2000     ; DEFINE LOAD ADDRESS
LNK  FILE.L    ; SPECIFY LNK FILE
SAV  FILE      ; SAVE THE OBJECT FILE
```

This would take the LNK file FILE.L and adjust all internal address references, i.e. JMPs, JSRs, etc. for a load address of $2000. The final object file would be saved on the disk under the name FILE.

The default filetype for SAV is BIN, i.e. type $06. Thus, if you were writing a SYStem file, you would have to add a TYP command to tell the Linker to save the final object file with the appropriate file type:

```
ORG  $2000     ; DEFINE LOAD ADDRESS
LNK  FILE.L    ; SPECIFY LNK FILE
TYP  $FF       ; SYSTEM FILETYPE
SAV  FILE      ; SAVE THE OBJECT FILE
```

If TYP is used, all succeeding SAVs, within a given Link operation, use the current TYP value.

ASM is added to a command file to automatically re-assemble a file that might have been changed without a re-assembly of a new copy of the corresponding LNK file:

```
ORG  $2000     ; DEFINE LOAD ADDRESS
ASM  FILE.S    ; RE-ASSEMBLE IF CHANGED
LNK  FILE.L    ; SPECIFY LNK FILE
TYP  $FF       ; SYSTEM FILETYPE
SAV  FILE      ; SAVE THE OBJECT FILE
```

USING MERLIN 16 LINKED FILES

Although linked files may be of any filetype and are not restricted to use by BASIC, here's an example that combines the output from two assemblies to create an object file that is loaded and called from BASIC. Compare this to a similar PUT file example. This is the first part of the program:

```
*********************************
*   RELOCATING LINKER SAMPLE    *
*         PART ONE              *
*********************************

          REL
          DSK   FILE1.L

HOME      EQU   $FC58
COUT      EQU   $FDED

BEGIN2    EXT               ; EXTERNAL LABEL

BEGIN     JSR   HOME        ; CLEAR SCREEN

          LDX   #$00        ; INITIALIZE COUNTER
LOOP      LDA   STRING,X    ; GET CHARACTER
          BEQ   DONE        ; END OF STRING
          JSR   COUT
          INX
          BNE   LOOP        ; ALWAYS
```

```
DONE     JMP   BEGIN2     ; JUMP TO 2ND SEGMENT...

STRING   ASC   "THIS IS FILE #1"
         HEX   8D,00       ; END OF STRING
         LST   OFF
```

And this is the second part:

```
*********************************
*   RELOCATING LINKER SAMPLE    *
*         PART TWO              *
*********************************

         REL
         DSK   FILE2.L

HOME     EQU   $FC58
COUT     EQU   $FDED

BEGIN2   ENT               ; ENTRY LABEL
         LDX   #$00        ; INITIALIZE COUNTER
LOOP     LDA   STRING,X    ; GET CHARACTER
         BEQ   DONE        ; END OF STRING
         JSR   COUT
         INX
         BNE   LOOP        ; ALWAYS

DONE     RTS               ; END OF PROGRAM

STRING   ASC   "THIS IS FILE #2"
         HEX   8D,00       ; END OF STRING
         LST   OFF
```

Having entered and saved these source listings, you would then enter and save this Linker command file:

```
*********************************
* MERLIN 16 LINKER NAMES FILE   *
*********************************

         LKV   $00         ; OPTIONAL CHECK FOR "LINKER"

         ASM   PART1.S     ; ASSEMBLE IF NEEDED
         ASM   PART2.S     ; ASSEMBLE IF NEEDED

         ORG   $8000       ; SPECIFY LOAD ADDRESS

         LNK   FILE1.L     ; LINK FILE
         LNK   FILE2.L     ; LINK FILE

         TYP   $06         ; BINARY
         SAV   FINAL.OBJ
```

These are assembled and linked together by typing LINK "FILENAME" where FILENAME is whatever name the command file has been saved under on the disk. Before linking the command file, remember to clear the workspace with NEW from the Command Box.

The generated object file, FINAL.OBJ could be tested with this Applesoft program:

```
10 TEXT: HOME
20 PRINT CHR$(4);"BLOAD FINAL.OBJ"
30 CALL 32768: REM $8000
40 VTAB 12: HTAB 15: PRINT "IT REALLY WORKS!"
50 LIST: END
```

Things to notice in the linker command file are the way that all the assemblies are done first, before any use of ORG or LNK instructions. In general, all ASMs should be done at once, followed by ORG and all LNKs.

The ORG $8000 specifies the load address of the final object file. This is used when linking the two LNK files called FILE1.L and FILE2.L into the final output file called FINAL.OBJ.

Both ASM and the check for the proper version of Linker with LKV are optional, and are not specifically required for this example. The main reason for including LKV in a command file is to make sure that the proper Linker is used for linking a particular command file. If you are developing both ProDOS 8 and ProDOS 16 applications at the same time, it would be easy to accidentally have the *wrong* Linker in the machine when you were trying to link a file for the *other* operating system.

In looking at the example, notice how the ENT and EXT pseudo-ops are used to communicate the label BEGIN2 between the two programs. Also notice there is no conflict over the use of the lables LOOP, DONE and STRING.

Compare this example to the sample program show for the PUT directive. The same result of combining separate source files is achieved, but without the disadvantages of PUT files discussed at the beginning of this Chapter.

MULTIPLE OUTPUT FILES

For applications running under ProDOS 8, you can use the Absolute Linker to generate separate output files during the linking process. The advantage is that each output file has access to the values of any entry point definitions in the other modules. Thus, if you wanted to write a program with other modules that needed to know the address of entry points within the main program, you could use the Absolute Linker to generate all the files at the same time.

For multiple output files, start each group of LNKs with an ORG and end it with a SAV. External references will be resolved between such groups by a second linker pass.

For example:

```
*********************************
* MERLIN 16 LINKER NAMES FILE   *
*   MULTIPLE OUTPUT SAMPLE      *
*********************************

        LKV   $00         ; OPTIONAL CHECK FOR "LINKER"

        ASM   PROG1A.S    ; ASSEMBLE IF NEEDED
        ASM   PROG1B.S    ; ASSEMBLE IF NEEDED

        ORG   $2000       ; SPECIFY LOAD ADDRESS

        LNK   FILE1A.L    ; LINK FILE
        LNK   FILE1B.L    ; LINK FILE

        TYP   $FF         ; FILETYPE = SYSTEM
        SAV   PROGRAM1    ; SAVE 1ST OUTPUT FILE

        ASM   PROG2A.S    ; ASSEMBLE IF NEEDED
        ASM   PROG2B.S    ; ASSEMBLE IF NEEDED

        ORG   $8000       ; SPECIFY LOAD ADDRESS

        LNK   FILE2A.L    ; LINK FILE
        LNK   FILE2B.L    ; LINK FILE

        TYP   $06         ; FILETYPE = BINARY
        SAV   OVERLAY     ; SAVE 2ND OUTPUT FILE
```

THE GS LINKER (LINKER.GS)

The GS Linker, on the Merlin 16 disk as LINKER.GS, works in the same way as the Absolute Linker except that it creates OMF (Object Module Format) files for the Apple IIgs only. The ORG is accepted, but should not be used, since memory is assumed to be assigned on an "as-available" basisin the Apple IIgs. However, you can use the ORG if you want to specify a specific load address for an object file.

NOTE: For Linker GS, the default SAV type is S16. Only one output file is supported, and there is a maximum length of the final file of about 32K for the code portion and another 32K for the relocation dictionary. LINKER.XL does not have these restrictions.

The main reason for using the GS Linker will be for creating application programs to run under ProDOS 16. Such files cannot be called from an Applesoft program, since ProDOS 16 is not available to Applesoft. In addition, the file format is such that a relocating dictionary is included as part of the final output object file. Thus, BLOADing it and listing it in the Monitor would reveal a certain amount of additional data saved with the file.

In addition to the linker commands of the standard Linker, the GS Linker has the following commands:

VER (version)
VER $01

This is used to specify the version of the OMF, i.e. the System Loader, that is to be used with the output object file. Versions 1 and 2 are supported. Thus, the operand must be $1 or $2. The VER instruction should come *before* any other linking instructions except ASMs, which are not dependent on the version of the Loader in use. ProDOS 16 version 1.2 uses version 2 of the loader format, and this is the default version used by the GS Linker if VER is not specified. You will have to decide what OMF version to use. If you specify OMF version 1, your file can be loaded by any version of ProDOS 16. If you specify OMF version 2, ProDOS 16 vers. 1.2 or later will be required.

KND address
KND $80
KND $8000

This specifies the value that you want put in the KIND location of the OMF header. You would use 1 byte for VER $1 and 2 bytes for VER $2. This must come *after* the VER so that the format is known to the Linker. If in doubt, ignore this command and accept the default.

ALI address
ALI $10000

This specifies the ALIGN field in the file header. It defaults to 0. Use only $10000 to align to a bank boundary or $100 to align to a page boundary. In most cases, you should leave this at the default 0.

NOTE: Our tests indicate that the bank align does not work on OMF version 1.

DS address
DS $2000

This tells the Linker to reserve this number of bytes to be zeroed by the loader at the END of the program. This number is put in the RESSPC field of the header. Using this instead of reserving space with a DS in the source file will result in smaller object files.

QUICK LINK (LINKER.GS ONLY)

In LINKER.GS, but not in the other Linkers, the command LINK =, or simply LINK without a file name, from the Command Box will assemble the file given by the default file name, i.e. the name appearing when you give the Load or Save commands from the Main Menu, then will link the resulting file. This assumes that a LNK file was produced by the DSK and REL opcodes. The linked file will then be saved using the name of the LNK file with the last two characters cut off.

It is suggested that you add a .L suffix to the DSK filename, i.e. MYFILE.L, so the saved file will be MYFILE. For example, a source file with the lines:

```
REL
DSK          MYFILE.L
```

could be saved to disk, and then assembled and linked with the Editor LINK command, and the final object file will be saved on the disk under the name MYFILE.

NOTE: This command does not require a linker command file, and that it uses the defaults in the Link which produces an S16 filetype in object module format version 2, and of "kind" $1000, i.e. code segment that cannot be loaded to special memory. This syntax should not be used if you wish load preferences different than the defaults, or if you require the advanced features of a command file. The command LINK1 can be used to produce a file in object module format 1 of "kind" 0 and type S16.

If there is a source file in memory when this command is issued, you will be asked if it is OK to save the source file to disk using the current name. If you agree, that file will automatically be saved under the default file name *Caution: this can be dangerous.* If the workspace is empty when the LINK command is used, then the file given by the default file name will be loaded, assembled, linked and the object file saved.

MULTIPLE LNK INPUT FILES

If you have multiple LNK files being linked to create the final object file, you should use a command file. However, Linker.GS does provide a short-cut way of linking up to 10 LNK files without requiring a command file. If the LNK file produced by the assembly has a name ending in ".x" where x is a digit from 0 to 9, i.e. MYFILE.3, then the Linker will not immediately link that file into the output file. Instead, it will look for the file with the same name but ending in .0, i.e. MYFILE.0, and will link that file with subsequently numbered LNK object files, i.e. MYFILE.1, MYFILE.2, MYFILE.3, etc., until it finds no more.

Up to 10 files can be linked without using a linker command file with this method. Note that only the default name source file is re-assembled. The source files for the other LNK files in the sequence are not assembled; only the LNK files are used as is, in the final object file. If a file is missing from the sequence, i.e. MYFILE.2 not present on the disk, the linking is terminated at that point, although no error is generated. Thus some care must be taken to make sure that all the needed files are on the disk.

USING THE GS LINKER

Because the GS Linker is used most often for linking a single input file, and creating a ProDOS 16 application of some sort, this example will demonstrate a simple ProDOS 16 program that waits for a keypress, and then returns to whatever program launched it.

Notice that the characteristic ENT and EXT, etc. items do not show up, since the primary goal is to just produce a single OMF file that can be loaded and run by a ProDOS 16 program launcher.

```
 1 ********************************
 2 *    SIMPLE P16 SYSTEM FILE    *
 3 *      MERLIN 16 ASSEMBLER     *
 4 ********************************
 5
 6           MX    %00          ; FULL 16 BIT MODE
 7           REL                ; RELOCATABLE OUTPUT
 8           DSK   P16.SYSTEM.L
 9
10 PRODOS    EQU   $E100A8      ; PRODOS 16 ENTRY POINT
11 KYBD      EQU   $00C000
12 STROBE    EQU   $00C010
13 SCREEN    EQU   $000400      ; LINE 1 ON SCREEN
14
15 ENTRY     PHK                ; GET PROGRAM BANK
16           PLB                ; SET DATA BANK
17
18 PRINT     LDX   #$00         ; INIT X-REG
19 LOOP      LDA   MSSG,X       ; GET CHAR TO PRINT
20           BEQ   GETKEY       ; END OF MSSG.
21           STAL  SCREEN,X     ; "PRINT" IT
22           INX                ; NEXT TWO CHARS
23           INX                ; X = X + 2
24           BNE   LOOP         ; WRAP-AROUND PROTECT
25
26 GETKEY    LDAL  KYBD         ; CHECK KEYBOARD
27           AND   #$00FF       ; CLEAR HI BYTE
28           CMP   #$0080       ; KEYPRESS?
29           BCC   GETKEY       ; NOPE
30           STAL  STROBE       ; CLEAR KEYPRESS
31
32 QUIT      JSL   PRODOS       ; DO QUIT CALL
33           DA    $29          ; QUIT CODE
34           ADRL  PARMBL       ; ADDRESS OF PARM TABLE
35           BCS   ERROR        ; NEVER TAKEN
36           BRK   $00          ; SHOULD NEVER GET HERE...
37
38 PARMBL    ADRL  $0000        ; PTR TO PATHNAME
39 FLAG      DA    $00          ; ABSOLUTE QUIT
40
41 ERROR     BRK   $00          ; WE'LL NEVER GET HERE?
42
43 MSSG      ASC   "PLEASE PRESS A KEY -> " ; EVEN NUMBER OF CHARACTERS
```

```
44          DA    $0000       ; TWO ZEROS
45
46          LST   OFF
```

First, enter and save this source file. Do not worry about assembling it yet. Once it is saved, use Open-Apple-O to open the Command Box and type NEW to erase the source file.

To link the file, just press Open-Apple-6. The source will automatically be loaded, assembled, the intermediate REL file written to disk, and the final output file, P16.SYSTEM, written to disk.

To test your program, just type -P16.SYSTEM as a disk command from the Main Menu. When the program quits, it will return to whatever program launcher started Merlin.16. You can alternatively quit Merlin to either the Apple DeskTop, Finder, or the Program Launcher, and select the file P16.SYSTEM. When you press a key, control should return to the program selector. Note that either approach *requires* that you have first started up using a ProDOS 16 disk, so that the ProDOS 16 operating system is available.

That's all there is to writing and assembling an Apple IIgs ProDOS 16 program using Merlin 16. For more complex files that require several input REL files, a command file can be used with the GS Linker. Also, try loading the P16.SYSTEM file, and type Open-Apple-6 with the source file in memory. The GS Linker will automatically save the source file before starting the link process. This is so that you can make changes to a program, and then automatically do the whole save-assemble-link process with a single keystroke.

NOTE: If you have the Roger Wagner Publishing IIgs program switcher called SoftSwitch, you may also want to put Merlin 16 in one Workspace and the Apple DeskTop program selector in another. With this combination, it is possible to assemble and link a program, save Merlin 16 intact in a Workspace, and switch instantly to the DeskTop to launch your ProDOS 16 program. When the program quits and returns to the DeskTop, you can switch back to Merlin 16 with the proper ProDOS prefixes set, ready to load the source file for further changes. With SoftSwitch and Merlin 16, it is possible to assemble, link and test a ProDOS 16 program, and to be back in Merlin 16 making changes in less than 60 seconds for the total cycle time!

LARGE FILE GS LINKER (LINKER.XL)

LINKER.XL is a version of LINKER.GS which makes two passes and writes the output file to disk to produce a multi-segment file. Use of the SAV command will cause the Linker to process the code to that point as one segment of a load file. This can produce much larger object files than the other Linker. However it is also much slower, so the other version is probably what you'll want to use most often. LINKER.XL does not have the command box LINK = capability of Linker.GS discussed earlier, so a command file is always required to use it. The maximum number of segments in the output file is 25, each with up to about 32K of code, excluding the relocation dictionary.

For LINKER.XL, the filename in the *first* SAV command is taken as the output file name. The rest, including the first, are placed in the *segment name* field of their respective segment header. If the name is more than the allowed 10 character space in the header, it is truncated to 10 characters.

LINKER.GS and LINKER.XL do not support the EXT or ENT linker commands that print the resolved address of labels, since these are generally meaningless on the Apple IIgs. The linker EXT and ENT commands should not be confused with the assembler EXT and ENT commands which are, of course, supported in all versions.