MERLIN-128 Version 1.10 — A Contrived User's Guide

Please note that this "guide" was compiled from several online sources, mostly pertaining to the Apple][versions of Merlin. The information here has not been completely tested with Merlin-128 and may not be 100% accurate for the Commodore version.

Editor Mode Keyboard Commands

Cursor Movement

C= B	move cursor to top of document
C= N	move cursor to bottom of document
Ctrl B	move cursor to beginning of line
Ctrl N	move cursor to end of line
Ctrl W	tab to next field

Program Control

C= H	toggle split screen mode
C= F,W	find string
C= Q	quit to main menu
C= +	quit to line editor
C= L	goto label
C= E	find and replace
Ctrl E	display free memory

Text Manipulation

C= X	cut text selection
C= C	copy text selection
C= P	paste text selection
C= A	select all text
C= D	delete cursor line
C= Del	delete line above cursor
C= R	juggle two lines of text
C= I	insert line
Ctrl D	rubout character after cursor
Ctrl K	convert upper/lowercase
Ctrl A	clear from cursor to end
Ctrl I	toggle insert/overwrite mode
Tab	toggle insert/overwrite mode
Ctrl T	same as backspace
Ctrl K	change character to lower case
Ctrl L	toggle upper/lowercase entry

Directives

`EQU` - equate label

label `EQU` *expression*

Create a label definition.

`ORG` - set origin

`ORG` *expression*

Establishes the address at which the program is designed to run. It defaults to $8000. If more than one `org` is used, the first one establishes the `bload` address and the second establishes the origin. You cannot use `org*-1` to back up the object pointers as is done in some assemblers. This must be done instead by using `DS-1`.

`OBJ` - set object

`OBJ` *address*

This is accepted only prior to the start of code. It only sets the division line between the symbol table and object code read in memory (which defaults to $8000). If the `REL` opcode is used then `OBJ` is disregarded.

`PUT` - put a text file in assembly

`PUT` *filename*

`PUT` reads the named file and inserts it at the location of the opcode. There are two restrictions on a `PUT` file: one, there cannot be macro definitions inside of a file which is `PUT`; second, a `PUT` file may not call another `PUT` file with this pseudo-op. Of course, linking can be simulated by having the main program just contain the macro definitions and call in turn all the others with the `PUT` opcode. Any variable may be used as a local variable.

`USE` - use a text file as a macro library

`USE` *filename*

This works like `PUT` but the file is kept in memory. It is intended for loading a macro library that is `USE`d by the source file.

`VAR` - setup variables

`VAR` *exprssion;expression;expression*

`VAR 1;$3;LABEL` *[setup variables 1, 2, and 3]*

This is just a convenient way to equate the variables `]1` through `]8`. `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` through `]8` there must be a previous declaration of these.

`SAV` - save object code

`SAV` *filename*

This will save the current object code. This can be done several times during assembly. Together with `put` these pseudo-ops make it posslble to assemble very large files.

`TYP` - file type for `DSK` or `SAV`

`TYP` *expression*

This sets the file type to be used by `DSK` or `SAV`. The default is `BIN`. Valid filetypes are 0, 6, $F0-$F7 and $FF (no type, `BIN`, `CMD`, user-defined, and `SYS`.) This probably doesn't have any effect in the Commodore version.

`DSK` - assembly directly to disk

`DSK` *filename*

This pseudo-op will force assembly output directly to disk.

`END` - end of source file

`END`

This rarely used opcode instructs the assembler to ignore the rest of the source.

`DUM` - dummy section

`DUM` *expression*

This starts a section of code that will be examined for value of labels but will produce no object code.

`DEND` - dummy end

`DEND`

This ends a dummy section and reestablishes the `org` address to the value it had upon entry to the dummy section. Shown below is a sample usage of `DUM` and `DEND`.

```
             ORG   $1000

IOBADRS      EQU   $B7EB

             DUM   IOBADRS
IOBTYPE      DFB   1
IOBSLOT      DFB   $60
IOBDRV       DFB   1
IOBVOL       DFB   0
IOBTRACK     DFB   0
IOBSECT      DFB   0
             DS    2
IOBBUF       DA    0
             DA    0
IOBCMD       DFB   1
IOBERR       DFB   0
ACTVOL       DFB   0
PREVSL       DFB   0
PREVDR       DFB   0
             DEND
```

```
START         LDA    #SLOT
              STA    IOBSLOT
*   And so on
```

Conditional Assembly

DO - do if true

DO *expression*

`DO 0`	*[turn assembly off]*
`DO 1`	*[turn asssembly on]*
`DO label`	*[if label<>0 then on]*
`DO label/l2`	*[if label<l2 then off]*
`DO label-l2`	*[if label=l2 then off]*

This along with `ELSE` and `FIN` are conditional assembly ops. If the operand evaluates to zero then assembler will stop generating object code (until it sees another conditional). Except for macro names, it will not recognize any labels in such an area of code. If the operand evaluates to a non zero then assembly will proceed as usual.

This is useful for sources to designed to generate slightly different code for different situations. For example in a program with text, you may wish to have one version for Apples with lower case adapters 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 (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 - do if not true

ELSE

This inverts the assembly condition (on becomes off and off becomes on) for the last `DO`.

IF - if true then do

IF *character,variable*

`IF (,]1`	[if first char of `]1` is **‘(‘** then assemble following code]
`IF ",]TEMP`	[if first char of `]TEMP` is **‘"‘** then assemble]
`IF "=]1`	[alternate use with equals sign instead of comma]

This checks to see if char is the leading character of the replacement string for]var. Postion is important the assembler checks the first and third characters of the operand for a match. If a match if found then the following code will be assembled. As with `DO` this must be terminated with a `FIN`, with optional `ELSE`'s between. The comma is not examined, so any character 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.

FIN - finish conditional

FIN

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

String Data

General notes on string data and string delimiters. Different delimiters have effects. Any delimiter less than (in ASCII value) the single quote (‘) will produce a string with the high bits on, otherwise the high bits will be off. All of the opcodes in this section except `REV` also accept hexadecimal data after the string. Any of the following syntaxes are acceptable:

```
ASC “STRING”878D00             DCI “STRING”,87,8D,00
FLS “STRING”,878D00            INV “STRING”,878D00
INV “STRING”,878D00
```

`ASC` - define ASCII text

`ASC “STRING”`	*[negative ASCII string]*
`ASC ‘STRING’`	*[positive ASCII string]*
`ASC “Bye, Bye”,8D`	*[negative with added hex byte]*

Insert ASCII string (and any hex data included) into memory.

`DCI` - dextral character inverted

`DCI “STRING”`	*[neg ascii except for the G]*

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.

`INV` - define inverse text

`INV “STOP”`	*[negative ASCII, inverse on printing]*

This puts a delimited string in memory in inverse format.

`FSL` - define flashing text

`FSL “FLASHING”`	*[negative ASCII, flashing on display]*

This is the same as `INV` except that the text attribute is flashing.

`REV` - reverse

`REV “sdrawkcab”`	*[negative ASCII, reverse order]*

This puts the delimited string in memory backwards. For example, `REV “sdrawkcab”` gives `“backwards”`. Hex data may not be added after the string terminator.

`STR` - define string with a leading length byte

`STR “HI”`	*[result = $02 C8 C9]*
`STR ‘HI’,8D`	*[result = $02 48 49 8D]*

This puts a delimited string into memory, adding one byte to record the string length at its beginning, otherwise working the same as `ASC`. The following hex bytes are not included in the count of the length.

Data and Storage Allocation

`DA` / `DW` - define address / define word

`DA $FDFD`	*[result = $FD FD in memory]*
`DA 10,$300`	*[result = $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 oblect code, least-significant-byte first. These two pseudo-ops also accept multiple data separated by commas (such as `DA 1,10,100`).

`DDB` - define double byte

`DDB $FDED+1`	*[result = $FD EE in memory]*
`DDB 10,$300`	*[result = $00 0A 03 00]*

This works just like `DA` except that the most-significant-byte is placed first.

`DFB` / `DB` - define byte

`DFB 10`	*[result = $0A in memory]*
`DFB $10`	*[result = $10 in memory]*
`DB >$FDED+2`	*[result = $FD in memory]*
`DB LAB`	*[example of use with labels]*

This puts the specified byte into the object code. It can accept several bytes of data, which must be separated by commas and contain spaces. Byte selection for 16-bit expressions is possible with the following operators:

`#<expression`	*[least-sig.-byte of the expression]*
`#>expression`	*[most-sig.-byte of the expression]*
`#expression`	*[alternate form for LSB]*
`#/expression`	*[alternate form for MSB]*

`HEX` - define hexadecimal data

`HEX 0102030F`	*[result = $01 02 03 0F in memory]*
`HEX FD,ED,C0`	*[result = $FD ED C0 in memory]*

This is a more convenient alternative to `DFB` for the insertion of hexadecimal data. Unlike all other pseudo-ops, use of the "$" is not required or accepted by `HEX`.

`DS` - define storage

`DS 10`	*[zero 10 bytes of memory]*
`DS 10,$80`	*[fill 10 bytes of memory with $80]*
`DS £`	*[zero memory to next page boundary]*
`DS £,$80`	*[fill to next page boundary with $80]*

This reserves space for string storage data, or whatever data structures are needed. The £ options are intended mainly for use with `REL` files and work slightly differently there. Any `DS £` opcode occurring in a `REL` file will cause the linker to load the next file at the first available page boundary.

Formatting

`LST` - control listing

`LST ON`	*[turn listing on]*
`LST OFF`	*[turn listing off]*
`LST`	*[turn listing on]*

This controls the assembly listing to be sent to the screen (or other output device). If the `LST` is off the object code will be generated much faster, but this is recommended only for debugged programs. (Note: Control-D from the keyboard toggles this flag during the second pass.)

`EXP` - macro expansion control

`EXP ON`	*[macro expand on]*
`EXP OFF`	*[print only macro call]*
`EXP ONLY`	*[print only generated code]*

With `EXP` on the entire macro will be printed during assembly. The off condition will print only the `>>>` 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. (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.

`LSTDO` - list the `do off` areas of the code

`LSTDO`	*[list the do off areas]*
`LSTDO OFF`	*[supress listing of the do off areas]*

This opcode determines whether `do` off areas of code are printed in the listing.

`PAU` - pause

```
PAU
```

On the second pass this causes the assembler to pause until a key is pressed. This can also be done from the keyboard by pressing the space bar.

`PAG` - insert page

```
PAG
```

This sends a formfeed signal to the listing.

`AST` - insert asterisks

`AST 30`	*[print 30 asterisks in listing]*

This sends a line of asterisks to the listing, handy for marking off sections of code.

`SKP` - skip lines

`SKP 5`	*[skip 5 lines in listing]*

This sends a given number of carriage returns to the listing.

`TR` - truncate control

`TR ON`	*[limit object code printing]*
`TR OFF`	*[don't limit object code printing]*
`TR`	*[limit object code printing]*

These commands can limit the object code printout to three bytes per line of source, even if the line generates more than three.

`DAT` - date stamp assembly listing

```
DAT
```

This command only works on the ProDos version of Merlin. It prints the current date and time on the seccond pass of assembly.

`CYC` - calculate and print cycle times

`CYC`	*[print opcode cycles and total]*
`CYC OFF`	*[stop cycle time printing]*
`CYC AVE`	*[print cycles and average]*

This opcode will cause a program cycle count to be printed during assembly. A second `CYC` opcode will cause the accumulated total to be zeroed. `CYC` off causes it to stop printing cycles. `CYC` ave will average in the cycles that are undeterminable due to branches, and indexed or indirect addressing. The cycle times will be printed to the right of the comment field and will appear similar to any of the following:

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

The first number displayed is the cycle count for the current instruction. The second number displayed is the accumulated total of cycles in decimal. A single quote after the cycle count indicates a possible added cycle, depending on certain conditions that the assembler cannot foresee. 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 once 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. It will not work for the additional R65C02 instructions present in the Rockwell chip.

Miscellaneous Pseudo-Opcodes

`KBD` - define label from keyboard

```
KBD
KBD string
```

This allows a label to be equated from the keyboard during assembly. Any expression may be input, including expressions referencing previously defined labels, a bad input error will occur if the input cannot be evaluated. If a string is given, it will be used for the user prompt.

`LUP` - loop expression

```
LUP value
--^
```
[used to indicate 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:

```
Source code:        Assembles as:
  LUP 4               ASL
  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:

```
1      ]A    =    0
2      LUP   $FF
3      ]     =    ]A+1
4      DFB   ]A
5      --^
```

Which 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 variable 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.

`CHK` - place checksum in object code

```
CHK
```

This places a checksum byte into object code as 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.

`ERR` - force error

```
ERR exprssion
ERR $80-($300)      [ error if $80 not in $300 ]
ERR *-1/$4000       [ error if PC>$4100 ]
ERR  \$5000         [ error if REL code address exceeds $5000 ]
```

The `ERR` expression will force an error if the expression has a non zero value and the message break in line `???` will be printed. Note for `REL` files: 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/`*expression*, but if when used with `REL` files it instructs the linker to check that the last byte of the current module does not extend to expressions or beyond (expression must be absolute). If the linker finds that the current module does extend beyond expression, linking will abort with a message *Constraint Error*.

Macros

MAC - start macro definition

`MAC` *macroname*

This signals the start of a macro definition.

<<< - end macro definition

`<<<`

This signals the end of a macro definition.

>>> - insert macro

`>>>` *macroname*

This instructs the assembler to assemble a copy of the named macro at the present location. Alternatively, the `>>>` can be omitted and the macro can be called simply by using the *macroname* as an opcode.

Macros represent a shorthand method of programming that allows multiple lines of code to be generated from a single statement. Macros can be used to simulate unimplemented opcodes or to simulate the Rockwell 65C02. A macro is a user named sequence of assembly language statements, with general purpose operands. You define the macro in a general way and when you use it via a macro call, you fill in the blanks left when you defined it. EXAMPLE:

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

Would assemble as follows if]1=$300 and]2=$400

```
LDA $300
STA $400
```

It is very important to realize that anything used in the parameter list will be substituted for the variables. Forward reference to a macro definition is not possible and will result in a not macro error message. A macro must be defined before it is called by *name* or `>>>`. The conditionals `DO`, `IF`, `ELSE` and `FIN` may be used within a macro. Labels inside macros are updated each time the macro *name* or `>>>` *name* is encountered. Error messages will usually indicate the line number of the macro call rather than the line inside the macro where the error occurred.

Macros may be nested up to 15 deep. Macros names may be put in the opcode column, without using `>>>`, but the macro name cannot be the same as any regular opcode or pseudo opcode such as `LDA`, `STA`, etc. It cannot begin with the letters `DEND` or `POPD`. The `>>>` opcode is not subject to this.

Eight variables, named `]1` through `]8` are predefined and are designed for convenience in macros. These are used in a `>>>` statement. The instruction `>>>` *name expr1*;*expr2*,*expr3*.... will assign the value of *expr1* to the variable `]1` and *expr2* to `]2` and so on. Example:

```
     MACRO DEFINITION                RESULT CODE EXAMPLE
TEMP       EQU   $10            SWAP.$6;$7;TEMP   ; MACRO CALL
           MAC
           LDA   ]1                  LDA   $06
           STA   ]3                  STA   TEMP
           LDA   ]2                  LDA   $07
           STA   ]1                  STA   $06
           LDA   ]3                  LDA   TEMP
           STA   ]2                  STA   $07
           <<<

           >>>   SWAP.$6;$7;TEMP
           <<<   SWAP.$1000;$6;TEMP
```

This segment swaps the contents of location $6 with that of $7 using `TEMP` as a scratch depository, then swaps the contents of $6 with that of $1000.

If as above some of the special variable are used in the macro definition, then values for them must be specified in the `>>>` statement. In the assembly listing, the special variables will be replaced by their corresponding expressions.

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

```
MUV  MAC
     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
```

The designed purpose of variables is for use in macros, but they are not confinded to that use. Forward reference to a variable is impossible with correct results, but the assembler will assign some value to it. It is possible to use variables for backward branching, using the same label at numerous places in the source. This simplifies label naming for large programs and uses much less space than the equivalent once used labels. For example:

```
1              LDY  #0
2   ]JLOOP     LDA  TABLE,Y
3              BEQ  NOGOOD
4              JSR  DOIT
5              INY
6              BNE  ]JLOOP     ;BRANCH TO LINE 2
7   NOGOOD     LDX  #-1
8   ]JLOOP     INX
9              STA  DATA,X
10             LDA  TBL2,X
11             BNE  ]JLOOP     ;BRANCH TO LINE 8
```

The Linker

The linking facilities offer these advantages:

1) Extremely large programs may be assembled in one operation over 41000 bytes long
2) Large programs may be assembled much more quickly with a corresponding decrease in development time.
3) Libraries of subroutines 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, say a 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.

PUT files or macro USES library don't serve the same purpose. Using a PUT file to add a general purpose subroutine would result in slower assembly. Any label definitions contained in the PUT file would be global within the entire program. 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.

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

REL - Informs 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 accessible to other REL files.

There are two other pseudo opcodes that behave differently when used in a REL file, than to a normal file, they are DS and ERR.

In order to use the Linker, the files to be linked must be specified. The linker uses a file containing the names of the files to be linked for this purpose.

REL - relocatable code module

```
REL
```

This instructs the assembler to generate code files compatible with the relocating linker. This must occur prior to definition of any labels. To get an object file to the disk you Must use the DSK opcode for direct assembly to disk.

An ORG at the start of the code is not allowed. Multiplication, division, or logical operations can be applied 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 not allowed are ordinary labels, expressions that use the PC such as “LABEL = *”.

The starting address of an REL file, supplied by the assembler, is $8000. It will be changed by the linker, which is why no ORG opcode is allowed. There are some restrictions 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 ]
```

EXT - external label

label EXT   *[label is external labels name]*

PRINT EXT   *[define PRINT as external label]*

This pseudo-op defines a label as an EXTernal label for use by the linker. Any external label must be defined as an ENTry label in its own REL module, otherwise it will not be reconciled by the linker. The EXTernal and ENTry label concepts are what allows REL modules to communicate and use each other as subroutines

ENT - entry label

label ENT

PRINT ENT   *[define PRINT as an entry label]*

This pesudo-op will define the label in 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. If a label is meant to be made available to other REL modules it must be defined with the ENT opcode. The true address of an ENTry label will be resolved by the linker. The example of a segment of a REL module will show the use of this opcode:

```
        STA  POINTER
        INC  POINTER
        BNE  SWAP
        JMP CONTINUE
```

```
25 SWAP   EXT                   ;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 EQU 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.

DS - skip to next REL file

`DS \`   *[skip to next REL file, fill mem with 0's to next page break]*

`DS \1`   *[skip to next REL file, fill mem with 1's 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 boundry and to fill memory either with zeros or with the value specified by the expression. This opcode should be placed at the end of your source file.

LINK - use linker

`LINK` *address, filename*

`LINK $1000 “NAMES”`   *[link files in NAMES]*

This editor command invokes the linking loader.To link the objext files whose names are held in the linker name file called NAMES use above command, this will give it a starting address of $1000. This is only accepted if there is no current source file in memory, since the linker would destroy it.

The linker name file is a text file containing the file names of the REL object modules you want linked. Write it with the Merlin editor and save to disk with the W command. The linker will not save the object file it creates, you must do this. Linker name files are a specially formatted file that contains the names of the link files to be linked. Example:

```
STR  "START”,00
STR  "MID",00
STR  "END",00
BRK
```

The break tells the linker there are no more pathnames. The file type used by the object save command is always the file type used in the last assembly.