Merlin 32

> What is Merlin 32 ?

Merlin 32 is a multi-pass **Cross Assembler** running under **Windows**, **Linux** and **Mac OS X** targeting 8 bit processors in the **6502** series (such as 6502 and 65c02) and the 16 bit **65c816** processor.

It is compatible with **Glen Bredon**'s **Merlin 16+** syntax, including support for Macros, Pre-processor, Logical Expressions, Conditional Operations, Variables, Loops, Local Labels...

It can build fixed position object code or **relocatable** executables (**OMF v2.1**) as we can find on 16 bits **Apple IIgs** operating systems like **Prodos 16** or **GS/OS (S16, Exe, CDA, NDA, FST, PIF, Library, Tool...**).

Merlin 32 is part of the Brutal Deluxe's Cross Development Tools Project, a full set of utilities available on Windows (and other) platforms to enable the creation of new Apple IIgs software : 65c816 Assembler, 65c816 Disassembler, 65c816 Simulator, Graphic File Converter, Resource Catcher...

> About Merlin 32

The idea behind the creation of **Merlin 32** was not to re-build a **Merlin 16+** clone on a modern computer like a PC running Windows. **Merlin 16+** is a great software including a full screen **Text Editor**, an 6502 / 65c02 / 65c816 **Assembler**, a **Linker** (including **OMF** support for Apple IIgs executable), a set of **Disk Utilities** (copy files, delete files, rename files...), a **Disassembler** (Source Error) and much more. But **Merlin 16+** is running on a single-process machine (the Apple IIgs) and this is now outdated. You can only perform one task after the other and there is no way to read / edit several source files at the same time (you have to save / close the first file before opening the other one). The editor, tailor made for assembly language editing, is limited to **24** lines and **80** columns, in **2** colors. You have to quit the editing of a source code to run it. And if it crash, you have to restart the operating system and restart everything (starting Merlin 16+, loading the source files...). Because of the Apple IIgs limitations, **Merlin 16+** is limited (a source file can't be larger than 64 KB). There is no way to extend it while it is running inside an Apple IIgs and there is no guarantee you are not going to crash the system while you are trying to execute your code (no memory protection due to 65c816 architecture).

It was time to provide a way to continue the Apple IIgs programming with modern tools, on a modern computer. Everyone has its own habits, so there was no need to clone the Full Text Editor. There are many very good IDE that can be used to write 65c816 source code (Eclipse, Visual Studio, ...). You can also use your favorite Text editor (**Emacs**, **PSPad**, **UltraEdit**...) where several files can be edited together (you can copy / paste from one to the other, split the screen to see several files on the screen at the same time) and the syntax highlighting helps you to read the code (one color per category of items). You can take advantage of the screen resolution (a 23" screen provides a text editor of 55 lines and 230 columns !) and you can keep the source file opened in the editor while you are assembling / linking the program in another window and there is no risk anymore to crash the system while trying to execute the program in the emulator (or in a real Apple IIgs). The speed, even if it is not the core argument for a cross assembler, lets you assemble large projects in few seconds, instead of minutes (if not hours) on a real Apple IIgs. All the data exchanges are simplified. You can copy / paste source code from a Web Page or a text file and use it directly on your text editor. No need any more to convert the file into a valid **Merlin 16+** format (high bit set to 1) before moving it to a disk image and use it under **Merlin 16+**. Because the source files are now stored on your modern computer as standard text files, you can use Source Control utilities like **SVN** to share your sources, backup them and check for modifications using revision tool.

With **Merlin 32**, we provide the **assembler** and the **linker** to turn the source code (**6502** / **65c02** / **65c816**) as a binary object (fixed position or relocatable with **OMF** support). All the edit job has to be done outside (with the text editor). You can **assemble** and **link** from a command window or you can use you IDE to associate the assembling syntax to a button. You probably have to work with **CADIUS**, another cross-development utility, to perform some basic tasks like indenting the source code (in the assembler style) or transferring the output of the assembly process (object code) or the source code into an Apple II disk image (**.2mg**, **.po**...).

There are already many cross-assemblers running on Windows capable to assemble 65c816 source code (**xa**, **wla dx**, **xasm**, **mads**...). Most of them were used to assemble source code targeting the **Super Nintendo** system (using a 65c816 like the Apple IIgs) or used as extension of 6502 cross assemblers dedicated to **Commodore 64** or **Atari XL** computers. They could be used to assemble 65c816 code for the Apple IIgs but at least two major features are always missing :

- The capability to assemble source code using **Merlin 16+** syntax (directives, macro, expressions, variables...)
- The capability to build relocated object code using **OMF format** (Apple IIgs 16 bit executables)

Merlin 16+ was one of the two most popular assemblers at the time for the **Apple IIgs** (the other one was **Orca M**) and many source codes are written using **Merlin 16+** syntax (like our tools & games). We do not have to be compatible with **Merlin 16+** syntax just to be able to re-assemble old files. We could have done a source converter to solve that issue. We have to be compatible with **Merlin 16+** syntax because we have to make sure that the source code used into **Merlin 32** on our PC running Windows could be sent back to the Apple IIgs to be also assembled with **Merlin 16+**. Even if we do 90% of the job with a cross-assembler, there are always few things that requires an Apple IIgs and its development toolset to build some parts like the Resource ones (menu, icon, about...). We don't say that **Merlin 32** is going to replace **Merlin 16+** and all the terrific development tools that already exist on the Apple IIgs platform. We say that we can speed up the process of writing code by using 90% of the time the cross assembler and 10% of the time the native Apple IIgs tools like **Merlin 16+**, **Genesys**, **Iconed, GS Bug**... With **OMFAnalyzer** tool, you can compare the output of **Merlin 16+** and the output of **Merlin 32** to ensure they both have generated the same object code (fixed address or relocatable) from the same source code.

The capability to build valid **OMF** relocatable executable files is something that the **Super Nintendo** cross-assemblers can't provide. The **Apple IIgs** is the most advanced software environment using the 65c816 processor and because of its operating system, it required a shared memory system capable to run several programs together in the same memory space. This implies memory management tools, dynamic loading of files, relocatable code, etc. At the opposite, the **Super Nintendo** games code run in ROM and don't have to deal with dynamic allocation or relocatable code.

Due to memory constraints, **Merlin 16+** has some internal limitations :

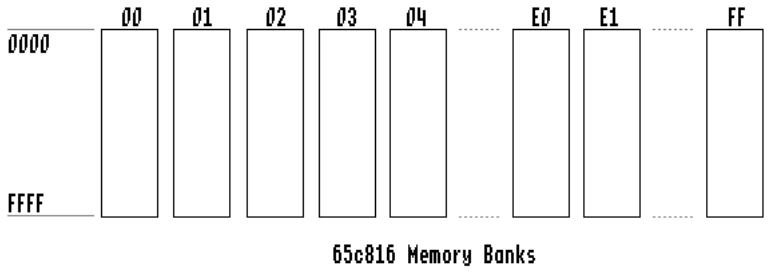
- a Source File can't be larger to 64 KB
- a Source Line can't be larger than 255 characters
- a Label can't be larger than 26 characters
- the Operand part can't be larger than 80 characters
- the number of Externals is limited to 255
- Macros can be nested to a depth > 15

- Conditions can't be nested to a depth > 8
- Symbol table is limited to 4096 symbols of length less than 12 and 2048 symbols of length 12 or over

Merlin 32 doesn't have any of these arbitrary limits. You can write your source code as you want but if you wish one day to send back the source code to the Apple IIgs and re-assemble it with **Merlin 16+**, check first your source code with the list above.

> Merlin 32 output

The **65c816** addressing space is 16 MB, divided into 256 memory **Banks** of 64 KB each (from 00 to FF). Bank 00 contains the **Stack** and the **Direct Page**.

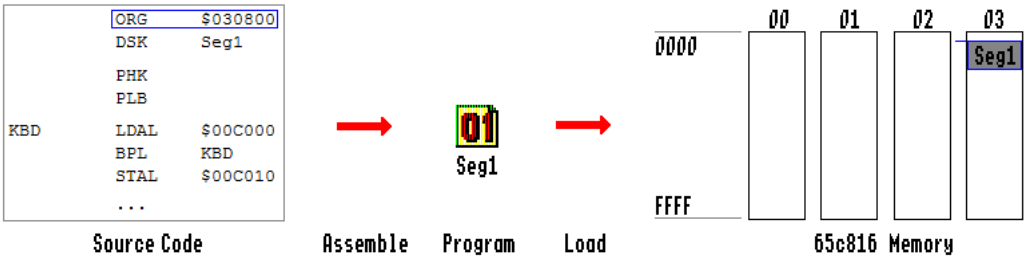


The **PC** is 16 bit, so the code execution is limited within the current bank boundary ($FFFF + 1 = $0000). If a code is bigger than 64 KB, it has to be split into small chunks of code (each of them < 64 KB) and spread over the memory banks. The connection between the chunks of code from different memory banks use **LONG** addressing mode instructions (LDAL, STAL, JMPL, JSL...). In the **Merlin 32** documentation, we will use the word **Segment** to define a chunk of 65c816 object code (with a size < 64 KB) located in one memory bank (not boundary cross). A **Program**, depending on its size, can use one or several **Segments**.

Merlin 32 lets you build **5** types of **Programs** :

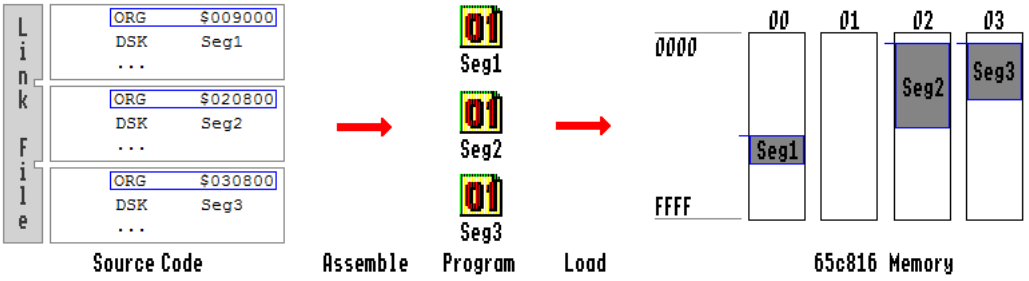
> SINGLE SEGMENT / FIXED ADDRESS

The source files are assembled as **One Binary File** and it has to be loaded at a **fixed** address in memory (defined by the **ORG** directive of the source file) :



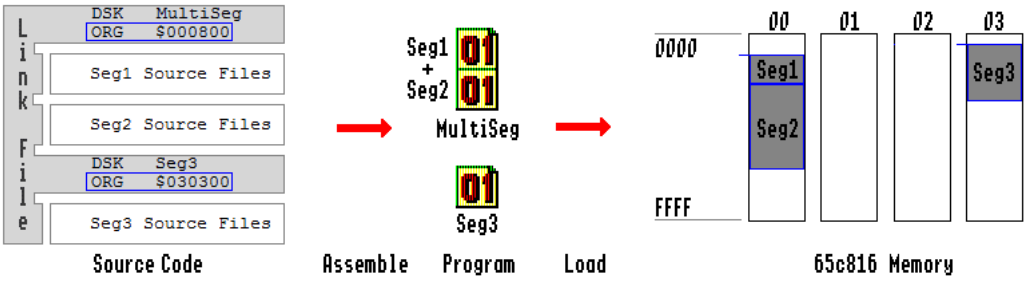
> MULTI SEGMENTS / FIXED ADDRESS

The source files (one set of files per segment) are assembled as **Several Binary Files** (one per segment) and they have to be loaded at a **fixed** address in memory (defined by the **ORG** directives of the source files) :



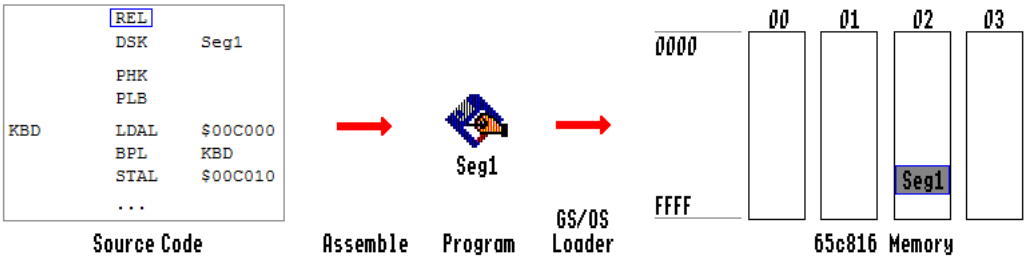
> MULTI SEGMENTS / FIXED ADDRESS / MERGED

The source files (one set of files per segment) are assembled as **One** or **Several Binary Files** (**several** segments may be merged into **one** binary file). They have to be loaded at a **fixed** address in memory. If several segments are merged into one binary file, the beginning address of a segment is set as the end address + 1 of the previous segment. The Fixed Address of the **First** segment of the binary files are defined by the **ORG** directives of the **Link** file. The names of the binary files are defined by the **DSK** directives of the **Link** file :



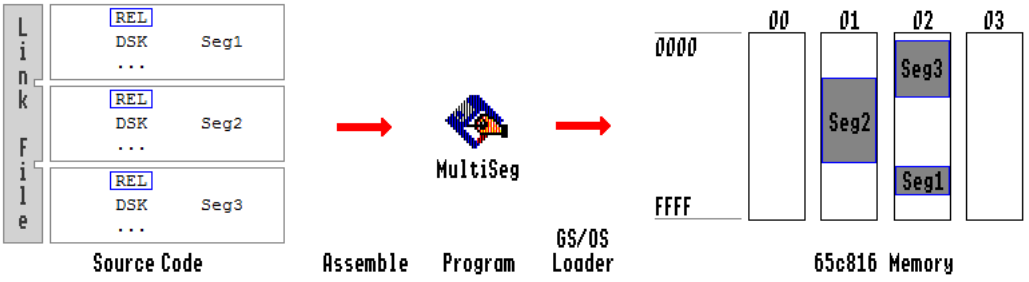
> SINGLE SEGMENT / RELOCATABLE

The source files are assembled as a **Single OMF Segment** file and will to be loaded by **GS/OS** at **ANY** address in memory (use of **REL** directive in the source file) :



> MULTI SEGMENTS / RELOCATABLE

The source files (one set of files per segment) are assembled as a **Multi-OMF Segments** file and will to be loaded by **GS/OS** at **ANY** address in memory (use of **REL** directive in the source files) :



Building a multi-segments programs (fixed address or relocatable) requires a definition file named **Link File**. The syntax of the Link file is described below, in the sections named **Building Multi-Segments Fixed-Address Files** and **Building Multi-Segments OMF Files**.

The **Fixed Address** binary files can be used in any system using a 65c816 processor like the **Apple IIgs**, the **SNES**, the Commodore **PET 65816** CPU card, the **CS/A 65816** CPU board, the CMD **SuperCPU**...

The **Relocatable Programs** can only be used on an **Apple IIgs** running **GS/OS**. The details about **OMF Files** data structure (Header + Object Code + Relocation Dictionary) can be found in the **Apple IIgs GS/OS Reference** book, **Appendix F : Object Module Format version 2.1**. You can DUMP / COMPARE **OMF Files** using our **OMFAnalyzer** Tool.

> Command List

If you do not provide any parameter on the command line, **Merlin32** displays a quick reminder of the required parameters :

```
C:\AppleIIgs>Merlin32.exe
Merlin32.exe v 1.0 (c) Brutal Deluxe 2011-2015
   Usage : Merlin32.exe  [-V]  <macro_folder_path>  <source_file_path>.
```

Syntax

```
Merlin32.exe  [-V]  <macro_folder_path>  <source_file_path>
```

Example

```
Merlin32.exe  -V  c:\AppleIIgs\Merlin\Library  c:\AppleIIgs\Source\Cogito\Cogito.s
```

Here are the parameters description :

- The first parameter **-V** (Verbose) is optional. If set, it builds a text file containing the output of the assembly process
- The second parameter (`<macro_folder_path>`)is the path of the Folder containing all Macro definition files (`*.Macs.s`)
- The third parameter (`<source_file_path>`) is the path of the *Master* source file (or the *Link* file) to be assembled

Few remarks about the parameters required on the Command Line and the software behavior :

- If the **Windows** File or folder paths contains **Space** characters, **quote** the path to avoid conflicts (`Merlin32.exe -V "c:\Users and Settings\Merlin\Library" c:\Source\Cogito\Cogito.s`).

- Any error occurring during the execution of the assembly process is immediately displayed on the screen.

- If you are transferring Source files or Macro files from a disk image or FTP server, make sure you transfer the file as a text file.

- If you are transferring Merlin source file (*.s) from a disk image, you may have to **clear the high bit**. You can use **Cadius** for that job (`Cadius.exe CLEARHIGHBIT <source_file_path>`).

- If you are opening Merlin source file (*.s) from a text file or by getting the code by a copy / paste from a web site, you may have to **indent the source** to make it easier to read with a Text editor on Windows. You can use **Cadius** for that job (`Cadius.exe INDENT <source_file_path>`).

During the execution of the process, a progression status is displayed on the screen :

```
C:\AppleIIgs\Merlin\>Merlin32.exe -V C:\AppleIIgs\Merlin\Library
C:\AppleIIgs\Source\Cogito\Cogito.s
Merlin32.exe v 1.0, (c) Brutal Deluxe 2011-2015
  + Assemble project files...
    o Loading Sources files...
        - Cogito.s
        - Cogito.Main.s
        - Cogito.Bout.s
    o Loading Macro files...
        - Int.Macs.s
        - Locator.Macs.s
        - Mem.Macs.s
        - Misc.Macs.s
        - Sound.Macs.s
        - Tool220.Macs.s
        - Util.Macs.s
    o Check for duplicated Macros...
    o Decoding lines types...
    o Process local/variable Labels...
    o Process Asterisk lines...
    o Build External table...
    o Build Equivalence table...
    o Build Variable table...
    o Process Equivalence values...
    o Replace Lup with code...
    o Replace Macros with Code...
    o Process MX directives...
    o Process Conditional directives...
    o Build Label table...
    o Check for duplicated Labels...
    o Check for unknown Source lines...
    o Check for Dum lines...
    o Compute Operand Code size...
    o Compute Operand Data size...
    o Compute Line address...
    o Build Code Line...
    o Check for Err lines...
    o Build Data Line...
    o Build Object Code...
  + Link project files...
    o Build OMF output file...
        => Creating OMF file 'C:\AppleIIgs\Source\Cogito\Cogito'
  + Create Output Text file...
     => Creating Output file 'C:\AppleIIgs\Source\Cogito\Cogito_Output.txt'
```

As a result, if everything went ok, you get one binary file (**fixed position object code** or **OMF file**) and, if the **-V** option was enabled, a text file containing the output of the assembly process (`Cogito_Output.txt`) :

```
------+-------------------------+-------------+----+-------+------+--------------------+----
-------------------------------------------------------------------
 Line | # File              Line | Line Type    | MX | Reloc | Size | Address Object Code|
Source Code
------+-------------------------+-------------+----+-------+------+--------------------+----
-------------------------------------------------------------------
    1 |  1      Cogito.s       1 | Comment      | 11 |       |    0 | 0000
| *-------------------------*
    2 |  1      Cogito.s       2 | Comment      | 11 |       |    0 | 0000
| *                         *
    3 |  1      Cogito.s       3 | Comment      | 11 |       |    0 | 0000
| *           COGITO          *
    4 |  1      Cogito.s       4 | Comment      | 11 |       |    0 | 0000
| *                         *
    5 |  1      Cogito.s       5 | Comment      | 11 |       |    0 | 0000
| *      Brutal  Deluxe      *
    6 |  1      Cogito.s       6 | Comment      | 11 |       |    0 | 0000
| *                         *
    7 |  1      Cogito.s       7 | Comment      | 11 |       |    0 | 0000
| * Version: 2.0 du 26/08/94 *
    8 |  1      Cogito.s       8 | Comment      | 11 |       |    0 | 0000
| *-------------------------*
```

```
    9 |  1       Cogito.s      9 | Empty        | 11 |        |     0 | 0000
|
   10 |  1       Cogito.s     10 | Directive    | 00 |        |     0 | 0000
|                   mx         %00
   11 |  1       Cogito.s     11 | Empty        | 00 |        |     0 | 0000
|
   12 |  1       Cogito.s     12 | Directive    | 00 |        |     0 | 0000
|                   lst        off
   13 |  1       Cogito.s     13 | Directive    | 00 |        |     0 | 0000
|                   rel
   14 |  1       Cogito.s     14 | Directive    | 00 |        |     0 | 0000
|                   dsk        Cogito.l
   15 |  1       Cogito.s     15 | Empty        | 00 |        |     0 | 0000
|
   16 |  1       Cogito.s     16 | Directive    | 00 |        |     0 | 0000
|                   use        4/Int.Macs
   17 |  1       Cogito.s     17 | Directive    | 00 |        |     0 | 0000
|                   use        4/Locator.Macs
   18 |  1       Cogito.s     18 | Directive    | 00 |        |     0 | 0000
|                   use        4/Mem.Macs
   19 |  1       Cogito.s     19 | Directive    | 00 |        |     0 | 0000
|                   use        4/Misc.Macs
   20 |  1       Cogito.s     20 | Directive    | 00 |        |     0 | 0000
|                   use        4/Sound.Macs
   21 |  1       Cogito.s     21 | Directive    | 00 |        |     0 | 0000
|                   use        4/Tool220.Macs
   22 |  1       Cogito.s     22 | Directive    | 00 |        |     0 | 0000
|                   use        4/Util.Macs
   23 |  1       Cogito.s     23 | Empty        | 00 |        |     0 | 0000
|
   24 |  1       Cogito.s     24 | Comment      | 00 |        |     0 | 0000
| *--- Parametres Page Zero
   25 |  1       Cogito.s     25 | Empty        | 00 |        |     0 | 0000
|
   26 |  1       Cogito.s     26 | Equivalence  | 00 |        |     0 | 0000
| Debut             =          $00
   27 |  1       Cogito.s     27 | Equivalence  | 00 |        |     0 | 0000
| Arrivee           =          $04
   28 |  1       Cogito.s     28 | Empty        | 00 |        |     0 | 0000
|
   29 |  1       Cogito.s     29 | Equivalence  | 00 |        |     0 | 0000
| proDOS            =          $e100a8
   30 |  1       Cogito.s     30 | Empty        | 00 |        |     0 | 0000
|
   31 |  1       Cogito.s     31 | Comment      | 00 |        |     0 | 0000
| *--------------------------
   32 |  1       Cogito.s     32 | Comment      | 00 |        |     0 | 0000
| * Initialisations d'entree
   33 |  1       Cogito.s     33 | Comment      | 00 |        |     0 | 0000
| *--------------------------
   34 |  1       Cogito.s     34 | Empty        | 00 |        |     0 | 0000
|
   35 |  1       Cogito.s     35 | Code         | 00 |        |     1 | 0000 : 4B
|                   phk
   36 |  1       Cogito.s     36 | Code         | 00 |        |     1 | 0001 : AB
|                   plb
   37 |  1       Cogito.s     37 | Empty        | 00 |        |     0 | 0002
|
   38 |  1       Cogito.s     38 | Macro        | 00 |        |     0 | 0002
|                   _TLStartUp
   40 |  1       Cogito.s     38 | Code         | 00 |        |     3 | 0002 : A2 01 02
|                   LDX        #$201                ; load tool call #
   41 |  1       Cogito.s     38 | Code         | 00 |        |     4 | 0005 : 22 00 00 E1
|                   JSL        $E10000              ; go to dispatcher
   42 |  1       Cogito.s     39 | Code         | 00 |        |     1 | 0009 : 48
|                   pha
   43 |  1       Cogito.s     40 | Macro        | 00 |        |     0 | 000A
|                   _MMStartUp
   45 |  1       Cogito.s     40 | Code         | 00 |        |     3 | 000A : A2 02 02
|                   LDX        #$202                ; load tool call #
   46 |  1       Cogito.s     40 | Code         | 00 |        |     4 | 000D : 22 00 00 E1
|                   JSL        $E10000              ; go to dispatcher
   47 |  1       Cogito.s     41 | Code         | 00 |        |     1 | 0011 : 68
|                   pla
   48 |  1       Cogito.s     42 | Code         | 00 | 2      |     3 | 0012 : 8D D8 AD
|                   sta        myID
   49 |  1       Cogito.s     43 | Macro        | 00 |        |     0 | 0015
|                   _MTStartUp
   51 |  1       Cogito.s     43 | Code         | 00 |        |     3 | 0015 : A2 03 02
|                   LDX        #$203                ; load tool call #
   52 |  1       Cogito.s     43 | Code         | 00 |        |     4 | 0018 : 22 00 00 E1
|                   JSL        $E10000              ; go to dispatcher
```

```
   53 |  1       Cogito.s      44 | Macro        | 00 |        |     0 | 001C
|                  _IMStartUp
   55 |  1       Cogito.s      44 | Code         | 00 |        |     3 | 001C : A2 0B 02
|                  LDX         #$20B                ; load tool call #
   56 |  1       Cogito.s      44 | Code         | 00 |        |     4 | 001F : 22 00 00 E1
|                  JSL         $E10000              ; go to dispatcher
   57 |  1       Cogito.s      45 | Empty        | 00 |        |     0 | 0023
|
   58 |  1       Cogito.s      46 | Code         | 10 |        |     2 | 0023 : E2 20
|                  sep         #$20
   59 |  1       Cogito.s      47 | Code         | 10 |        |     4 | 0025 : AF 22 C0 E0
|                  ldal        $e0c022
...
```

The output file lets you check the **pre-processor** job (replace Macros with code, expand Lups, resolve local labels, compute expressions...), the **assembler** job (addressing mode, A X Y registers size, object code, ...) and the **linker** job (multi-org directives, addresses to be patched for relocated code, ...).

Here is a quick explanation for the columns available in the output file :

- **Line** : Global line number (1 to N).
- **# File Line** : Source file number (>1 if several source files are involved using **PUT** directive) and Local source file line number.
- **Line Type** : Type of source code line : Empty, Comment, Directive, Equivalence, Macro, Code or Data
- **MX** : Size for **M** (**Accumulator**) and **X** (**X** and **Y Registers**).This is helpful to understand if **Merlin 32** is assembling **8 bit** or **16 bit** code. MX values are usually modified by **MX** directive or **SEP** / **REP** opcode.
- **Reloc** : For relocatable code, you will find here the number of bytes to be relocated and the shift operation performed on the address (>> 8, >>16...). If the label is **EXT**ernal to the segment, the letter **E** is added in the column.
- **Size** : Number of bytes used to encode this line.
- **Address Object Code** : Address (16 bit) of the line. If the **ORG** directive is used, the first address starts there. If the code is relocatable (**REL** directive), the first address is $0000. The bytes used to encode this line follow the address. We don't put more than 4 bytes / line.
- **Source Code** : The source code of the line has been processed (since we got it from source file) : Macros have been expanded, Loops has been exploded, local Labels have been replaced by unique names, Expressions have been resolved...

If you want to be sure that the source assembled with **Merlin 32** on **Windows** create the same binary file than **Merlin 16+** on **GS/OS**, you can compare the two result files with **OMF Analyzer**. If you are assembling a **fixed position object code**, use the **COMPAREBIN** command, if you are assembling an **OMF file**, use the **COMPARE** command.

> Merlin 32 Syntax

Because **Merlin 32** uses the same syntax than **Merlin 16+**, the easiest way to learn about **Merlin 32** syntax is probably to read documentation about **Merlin 16+**. You can pick up the **Merlin 16+ documentation** or any assembly book using **Merlin 16+** syntax like **Apple IIgs Machine Language for Beginners** written by **Roger Wagner**.

The section provides information on writing assembly language programs with **Merlin 32**. You can skip this reminder if you are already familiar with **Merlin 16+**.

INDENTATION

An assembly source code is organized in 4 columns :

- **LABEL** : Contains the identifier name for this line. It can be the label where to branch, the name of a new Macro, the name of a Variable...
- **OPCODE** : Contains the action to be performed by the line. It can be a valid 65816 opcode, a Merlin 32 Directive, the name of a Macro to call...
- **OPERAND** : Contains the parameter of the OPCODE. It can be the operand of the opcode, the Macro parameters, the value of the variable...
- **COMMENT** : Starts with a **;** character and contains a text explaining the Line purpose.

Merlin 32 is case sensitive for Labels, Macros, Operand values, Variables, Equates... You can write either `LDA` or `Lda` for opcode but `PushLong` and `pushlong` are not the same Macro !

We can use blank characters (SPACEs or TABs) to define the beginning / end of a column.

```
LABEL           OPCODE          OPERAND          COMMENT

                mx              %00
                use             4/Int.Macs

proDOS          =               $e100a8

                phk
                plb

                clc
                xce
                rep             #$30

memERR          bcs             memERR1          ; Memory Error
                rts
memERR1         PushWord        #0
                PushLong        #memSTR1
                PushLong        #memSTR2
                PushLong        #proSTR3
                PushLong        #proSTR4
```

```
                    _TLTextMount
                    pla
memERR2             jmp             initOFF

proKill             dw              1
                    adrl            pTEMP               ; Pathname
```

Do not bother with indentation when you write your code in a Windows Text editor. Just add few Spaces or Tabs to separate columns. Once the lines have been written (or copy / pasted from another location), use **CADIUS** to indent automatically your source code :

```
CADIUS.exe  INDENT  <source_file_path>
```

After processing, the code is easier to read :

```
SOURIS LDA BOUT ; ANCIEN BOUT=NOUVEAU BOUT
 STA BOUT1

SOURIS0 JSR SLECT ; LECTURE SOURIS
 CPY #$FFFF
 BEQ SECR ; DONNEES NON DISPONIBLES

SOURIS1 LDA A1 ; A1 POSITION ACTUELLE
 STA AP ; AP ANCIENNE POSITION
 LDA POSX
 LSR
 STA SOURIS2+1
 LDA POSY
 ASL
 TAX
 LDA TABLE,X
 CLC
SOURIS2 ADC #$0000 ; CALCUL DE A1 (160*POSY+POSX)
```

```
SOURIS      LDA    BOUT
             STA    BOUT1

SOURIS0     JSR    SLECT
             CPY    #$FFFF
             BEQ    SECR

SOURIS1     LDA    A1
             STA    AP
             LDA    POSX
             LSR
             STA    SOURIS2+1
             LDA    POSY
             ASL
             TAX
             LDA    TABLE,X
             CLC
SOURIS2     ADC    #$0000
```

Repeat the indent process as many times as you need.

COMMENT

A valid comment line starts with a ***** or a **;** character. A comment line is never indented and does not have to enter into the LABEL / OPCODE / OPERAND / COMMENT scheme. If the line contains only blank characters like SPACEs or TABs, the line is considered as empty. If the first valid (non blank) character of the line is a **;** with some blank characters before, the line is indented and the content is transferred in the COMMENT column.

```
*-------------------------------------------
*--  Check we have at least 512 KB available
*-------------------------------------------
; _FreeMem
                                                ; Memory Allocation
okIT2               PushLong   #0              ; Ask for Shadowing
                    PushLong   #$8000
                    PushWord   myID
```

OPCODE

You can use all the 65c816 opcodes, with the following standard mnemonics :

```
ADC AND ASL
BCC BLT BCS BGE BEQ BIT BMI BNE BPL BRA BRK BRL BVC BVS
CLC CLD CLI CLV CMP COP CPX CPY
DEC DEX DEY
EOR
INC INX INY
JMP JML JSR JSL
LDA LDX LDY LSR
MVN MVP
NOP
ORA
PEA PEI PER PHA PHB PHD PHK PHP PHX PHY PLA PLB PLD PLP PLX PLY
REP ROL ROR RTI RTL RTS
SBC SEC SED SEI SEP STA STP STX STY STZ
TAX TAY TCD TCS TDC TRB TSB TSC TSX TXA TXS TXY TYA TYX
WAI WDM
XBA XCE
```

Opcodes modifying the **Accumulator** such as ASL, LSR, DEC and INC have no operand value. Write them `ASL`, not `ASL A`.

For **Long** addressing modes (24 bits address), you can add a `L` character at the end of the mnemonic :

```
ADCL SBCL
ANDL EORL ORAL
CMPL
LDAL STAL
JMPL
```

If you want to use alternate opcodes such as BGE (=BCS) or BLT (=BCC), you can easily define them as Macros.

ADDRESSING MODE

Merlin 32 handles all the 65c816 addressing modes, with the following syntax :

```
ASL                  ; A           Implicit
LDA   #$2000         ; #const      Immediate
LDA   $C000          ; addr2       Absolute
LDA   ($2000,X)      ; (addr2,X)   Absolute Indexed,X Indirect
LDA   $2000,X        ; addr2,X     Absolute Indexed,X
LDA   $2000,Y        ; addr2,Y     Absolute Indexed,Y
LDA   ($2000)        ; (addr2)     Absolute Indirect
LDA   [$2000]        ; [addr2]     Absolute Indirect Long
LDAL  $E12000        ; addr3       Absolute Long
LDAL  $E12000,X      ; addr3,X     Absolute Long Indexed,X
LDA   $10            ; dp           Direct Page
LDA   $10,X          ; dp,X         Direct Page Indexed,X
LDA   $10,Y          ; dp,Y         Direct Page Indexed,Y
LDA   ($10)          ; (dp)         Direct Page Indirect
LDA   [$10]          ; [dp]         Direct Page Indirect Long
LDA   ($10,X)        ; (dp,X)       Direct Page Indexed Indirect,X
LDA   ($10),Y        ; (dp),Y       Direct Page Indirect Indexed,Y
LDA   [$10],Y        ; [dp],Y       Direct Page Indirect Long Indexed,Y
BEQ   LABEL          ; relative1    Program Counter Relative
BRL   LABEL          ; relative2    Program Counter Relative Long
LDA   ($10,S),Y      ; (sr,S),Y     Stack Relative Indirect Indexed,Y
LDA   $10,S          ; sr,S         Stack Relative
PEA   $1010          ; #const       Stack Immediate
PEI   ($10)          ; (dp)         Stack Direct Page Indirect
PER   $2000          ; #const       Stack Program Counter Relative Long
```

By convention, some Opcodes like PEA or PER receive addresses (starting with $) as Operand even if it should be constants (`PEA $A0A0` stores at the top of the stack the constant value `#$A0A0`, not the value found at address `$A0A0`).

The purpose of the **Merlin 32** syntax is to remove any ambiguity regarding what the assembly process is supposed to build as output code.

For example, such code is not very clear :

```
LDA   0              ; ???
```

Do we want to load the *constant Zero* in the accumulator (8 or 16 bit ?) or do we want to load the *value located at address 0* (but is it Page Direct `$00`, Current Bank address `$0000` or Long address `$00/0000` ?).

The first thing is to tell the difference between **Data** and **Address**. **Data** Operand starts with a # while **Address** is everything else (numeric value, Label...) :

```
LDA   #0             ; Data (Decimal)
LDA   #$2000         ; Data (Hexadecimal)
LDA   #%11110000     ; Data (Binary)

LDA   0              ; Address (Decimal)
LDA   $2000          ; Address (Hexadecimal)
LDA   %00100000      ; Address (Binary)
LDA   LABEL          ; Address (Label)
LDA   LABEL+2        ; Address (Expression with Label)
```

The only times where Operands could be Data without using the # as leading character is when we build expressions with an **even** number of Labels. For example, we compute here the number of bytes between two Labels :

```
LDA   END-BEGIN      ; Data (Number of bytes between the two labels)
```

For immediate addressing modes (Operand is a Data), we have to figure out if the Operand is 8 bit or 16 bit. The following code :

```
LDA   #1             ; Store 1 into the accumulator
```

could be assembled as :

```
+----+-------+------+-------------------+----------------------------------------
| MX | Reloc | Size | Address Object Code|  Source Code

+----+-------+------+-------------------+----------------------------------------
| 11 |       |    2 | 8000 : A9 01      |            LDA   #1         ; A is 8
bit (M=1)
```

or

```
+----+-------+------+-------------------+----------------------------------------
| MX | Reloc | Size | Address Object Code|  Source Code
```

```
+----+-------+------+--------------------+----------------------------------------
| 00 |       |    3 | 8000 : A9 01 00    |            LDA   #1         ; A is 16
bit (M=0)
```

Merlin 32 keeps the status of the **M** (Accumulator) and **X** (X and Y registers) bits of the State Register for each line of the source code. In the Output text file, you can see them in the MX column (0=16 bit, 1= 8 bit). The choice between 8 or 16 bit for Data Operand is based on the MX values. You can set the value of the MX bits using the **MX** directive in the source code. The **MX** directive use as Operand a value between 0 and 3, usually display using Binary format (%00, %01, %10 or %11) :

```
+----+-------+------+--------------------+----------------------------------------
| MX | Reloc | Size | Address Object Code|  Source Code

+----+-------+------+--------------------+----------------------------------------
| -- |       |      |                    |            MX    %00        ;
Assemble next lines with M and X in 16 bit
| 00 |       |    3 | 8000 : A9 01 00    |            LDA   #1         ; A is 16
bit (M=0)
...
| -- |       |      |                    |            MX    %11
 ; Assemble next lines with M and X in 8 bit
| 11 |       |    2 | 8003 : A9 01       |            LDA   #1         ; A is 8
bit (M=1)
```

Merlin 32 , furthermore, analyzes the Source Code for **SEP** or **REP** Opcodes and change the MX values based on the Operand value :

```
+----+-------+------+--------------------+----------------------------------------
| MX | Reloc | Size | Address Object Code|  Source Code

+----+-------+------+--------------------+----------------------------------------
| -- |       |    2 | 8000 : C2 30       |            REP   #$30       ; Force M
and X bits from Status Register in 16 bit
| 00 |       |    3 | 8002 : A9 01 00    |            LDA   #1         ; A is 16
bit (M=0)
...
| -- |       |    2 | 8005   E2 30       |            SEP   #$30       ; Force M
and X bits from Status Register in 8 bit
| 11 |       |    2 | 8007 : A9 01       |            LDA   #1         ; A is 8
bit (M=1)
```

Unlike the **REP** and **SEP** Opcodes, the **MX** directive doesn't change anything for code execution, it only impacts the assembly process.Up to you to control that 16 bit assembled code is called with 16 bit accumulator & registers.

Some Operand expressions may represent values larger than the Accumulator (or Register) size. By using some operators (< > ^) right after the #, **Merlin 32** lets you select the bytes(s) you want to keep :

IMMEDIATE 8 BIT

We take only 1 byte from the Operand :

```
A9 00         LDA  #LABEL      ; with LABEL = $00E12000
A9 00         LDA  #<LABEL     ; with LABEL = $00E12000
A9 20         LDA  #>LABEL     ; with LABEL = $00E12000
A9 E1         LDA  #^LABEL     ; with LABEL = $00E12000
```

IMMEDIATE 16 BIT

We take 2 bytes from the Operand :

```
A9 00 20      LDA  #LABEL      ; with LABEL = $00E12000
A9 00 20      LDA  #<LABEL     ; with LABEL = $00E12000
A9 20 E1      LDA  #>LABEL     ; with LABEL = $00E12000
A9 E1 00      LDA  #^LABEL     ; with LABEL = $00E12000
```

The **PEA** Opcode acts like an *Immediate 16 bit* Opcode, even if the Operand is seen as an address (no #) :

```
F4 00 20      PEA  LABEL       ; with LABEL = $00E12000
F4 00 20      PEA  <LABEL      ; with LABEL = $00E12000
F4 20 E1      PEA  >LABEL      ; with LABEL = $00E12000
F4 E1 00      PEA  ^LABEL      ; with LABEL = $00E12000
```

When the Operand is an Address, **Merlin 32** has to figure out how many bytes (between 1 and 3) is used for the address encoding :

```
+----+-------+------+--------------------+----------------------------------------
| MX | Reloc | Size | Address Object Code|  Source Code

+----+-------+------+--------------------+----------------------------------------
| 00 |       |    2 | 8000 : A5 10       |            LDA   $10       ; Direct
Page   (1 byte)
| 00 |       |    3 | 8002 : AD 00 C0    |            LDA   $C000     ; Absolute
     (2 bytes)
| 00 |       |    4 | 8005 : AF 00 20 E1 |            LDA   $E12000   ; Long
```

```
(3 bytes)
```

Here is how **Merlin 32** chooses among the 3 different addressing modes :

DIRECT PAGE

By default, **Merlin 32** uses the **Direct Page** addressing mode for any Operand having a value in the range `$00-$FF` :

```
A5 10          LDA   $10       ; Direct Page  (1 byte)
A5 E1          LDA   LABEL     ; with LABEL = $E1
```

ABSOLUTE

The Absolute address mode is the default on for any Address other than the range `$00-$FF`. If the Operand is in the range `$00-$FF`, you can force an Absolute addressing mode by adding any character (except L) at the end of the Opcode :

```
AD 00 20       LDA   $E12000   ; Use only the 2 low bytes of the address
AD 00 20       LDA   $2000     ;
AD 11 00       LDA:  $11       ; Force Absolute with :
AD 00 20       LDA   LABEL     ; with LABEL = $E12000
AD 00 20       LDA   LABEL     ; with LABEL = $2000
AD 11 00       LDA:  LABEL     ; with LABEL = $11
```

LONG

The Long addressing mode is forced by adding a L character at the end of the Opcode or a > character at the beginning of the Operand :

```
AF 00 20 E1   LDAL   $E12000  ;
AF 00 20 E1   LDAL   LABEL    ; with LABEL = $E12000
AF 00 20 AA   LDAL   LABEL    ; with LABEL = $2000  ($AA is the LABEL Bank)
AF 00 00 AA   LDAL   LABEL    ; with LABEL = $00    ($AA is the LABEL Bank)
AF 00 20 E1   LDA    >$E12000 ;
AF 00 20 E1   LDA    >LABEL   ; with LABEL = $E12000
AF 00 20 AA   LDA    >LABEL   ; with LABEL = $2000  ($AA is the LABEL Bank)
AF 00 00 AA   LDA    >LABEL   ; with LABEL = $00    ($AA is the LABEL Bank)
```

NUMBER

You can use **decimal**, **hexadecimal** or **binary** numerical data :

- Hexadecimal numbers start with a $: `$E12000`, `$00A0`, `$BD`
- Binary numbers start with a % and can use _ as visual separator : `%01100101`, `%0000_1111_0000_1111`
- Decimal numbers don't use any specific prefix : `15`, `635`, `32768`

For opcodes accepting both **data** and **addresses**, you have to use the # as first character in the operand, in order to specify a **data** value :

```
A9 A0 00    LDA  #$00A0       ; Load a 16 bit constant numeric data 160 ($A0) in the accumulator.
AD 00 20    LDA   $2000       ; Load value stored at address $2000 in the accumulator.
```

For opcodes accepting only one type of operand (data or address) such as REP, PEA, JSR, MVN, STA... you don't need to add the **#** but is it always a good idea to insert it when **data** is involved (REP, SEP, PEA...).

STRING

The Apple IIgs recognizes only the following characters (the first one is the *Space* character) :

```
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ▉
```

A string is a set of ASCII characters enclosed by **quotes** (') or **double quotes** (") :

```
48 65 6C 6C 6F    ASC  'Hello'       ; Using simple quote, the high bit is set to 0 (standard ASCII)
C8 E5 EC EC EF    ASC  "Hello"       ; Using double quotes, the high bit is set to 1 (for Text Screen encoding)
```

You can encode any ASCII character in a string by inserting before / in the middle / after the Hexadecimal value of the character(s) :

```
ErrorMsgLoad      ASC  'Can',27,'t load file !'     ; Can't load file, $27 is the hexadecimal value for '
```

DATA STORAGE

There are many pseudo opcodes used to define Data Storage (tables...).

HEX define HEXadecimal data

```
00 01 02 03     HEX  00010203
00 01 02 03     HEX  00,01,02,03
00 01 02 03     HEX  0001,0203
```

The operand consists of hexadecimal numbers (0-F) having even number of Hex digits (so 0F, not F). They may be separated by commas or may be adjacent. The $ is not required here.

DFB or **DB** DeFine Byte

```
0A 0B 0E 0F     DFB  $0A,$0B,14,%0000_1111
EE              DFB  LAB+2              ; LAB Address is $FDEC, so LAB+2=$FD EE
FD              DFB  >LAB               ; LAB Address is $ FD EC
```

The operand consists of several bytes of data, separated by commas. It accepts all kinds of numeric formats (decimal, $hexadecimal and %binary) and arithmetic expressions. The low byte of the expression is always taken, except if you use the > sign (get high byte).

DDB Define Double Byte

```
00 0A 00 0E     DDB  $000A,14
FD EC FD EE     DDB  LAB,LAB+2          ; LAB Address is $FDEC, so LAB+2=$FDEE
```

The operand consists of several two-byte of data, separated by commas. It accepts all kind of numeric formats (decimal, $hexadecimal and %binary) and arithmetic expressions. The bytes are placed high-byte first.

DA or **DW** Define Address or Define Word

```
0A 00 0E 00     DA   $000A,14
EC FD EE FD     DA   LAB,LAB+2          ; LAB Address is $FDEC, so LAB+2=$FDEE
```

The operand consists of several two-byte of data, separated by commas. It accepts all kind of numeric formats (decimal, $hexadecimal and %binary) and arithmetic expressions. The bytes are placed low-byte first.

ADR Define ADdRess - 3 bytes

```
0A 00 00     ADR   $0A
00 20 E1     ADR   SCREEN               ; SCREEN Address is $E1/2000
```

The operand consists of several three-byte of data, separated by commas. It accepts all kind of numeric formats (decimal, $hexadecimal and %binary) and arithmetic expressions. The bytes are placed low-byte first.

ADRL Define Long ADdRess - 4 bytes

```
0A 00 00 00     ADRL   $0A
00 20 E1 00     ADRL   SCREEN           ; SCREEN Address is $E1/2000
```

The operand consists of several four-byte of data, separated by commas. It accepts all kind of numeric formats (decimal, $hexadecimal and %binary) and arithmetic expressions. The bytes are placed low-byte first.

DS Define Storage

```
00 00 00 00 00 00 00 00     DS   8        ; Reserve 8 byte of data, filled with 0x00
EE EE EE EE EE EE EE EE     DS   8,$EE    ; Reserve 8 byte of data, filled with 0xFF
A0 A0 A0 ...                DS   \,$A0    ; Fill memory with 0xA0 values until the next memory page
```

Reserve space for *Operand* bytes of data (set to 0x00). You can choose to fill the reserved space with values other than 0x00 by providing a value (or an expression) as second operand. If you use the keyword \ as first operand, the memory is filled until the next page boundary. On relocatable code, the DS \ should only be used at the end of the file.

ASC define ASCii text

```
48 65 6C 6C 6F     ASC   'Hello'         ; Using simple quote, the high bit  is
set to 0
C8 E5 EC EC EF     ASC   "Hello"         ; Using double quotes, the high bit is
set to 1
```

This puts a delimited ASCII string in the object code. The simple quote is standard Ascii, used in Text files, GS/OS calls, file paths....

The double quotes (high bit set to 1) is used to display Text on Apple IIgs Text Mode screen (Page 1 or 2). The valid characters for Screen display are :

```
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ 
```

The encoding goes from $A0 (Space) to $FF ( ).

DCI Dextral Character Inverter

```
48 65 6C 6C EF     DCI   'Hello'         ; The high bit is set to 0, except for
the last character
C8 E5 EC EC 6F     DCI   "Hello"         ; The high bit is set to 1, except for
```

```
the last character
```

This puts a delimited ASCII string in the object code, with the last character having the opposite high bit to the others.

INV define INVerse text

```
08 05 0C 0C 0F     INV   'HELLO'         ; Inverse works only with Uppercase
characters + Special characters
08 05 0C 0C 0F     INV   "HELLO"
```

This puts a delimited ASCII string in the object code, in Inverse video format. The valid characters for **Inverse Video** are :

```
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
```

The encoding goes from $00 (@) to $3F (?).

FLS define FLaShing text

```
48 65 6C 6C 6F     FLS   'HELLO'         ; Flashing works only with Uppercase
characters + Special characters
48 65 6C 6C 6F     FLS   "HELLO"
```

This puts a delimited ASCII string in the object code, in Flashing video format. The valid characters for **Flashing Video** are :

```
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
  ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
```

The encoding goes from $40 (@) to $7F (?).

REV define REVerse text

```
6F 6C 6C 65 48     REV   'Hello'         ; The high bit is set to 0
EF EC EC E5 C8     REV   "Hello"         ; The high bit is set to 1
```

This puts a delimited ASCII string in the object code, in backward order.

STR define STRing with leading length byte

```
05 6F 6C 6C 65 48   STR   'Hello'        ; The high bit is set to 0
05 EF EC EC E5 C8   STR   "Hello"        ; The high bit is set to 1
```

This puts a delimited ASCII string in the object code with leading length byte. Following hex values, if any, are not counted in the length.

STRL define Long STRing with leading length word

```
05 00 6F 6C 6C 65 48   STRL   'Hello'    ; The high bit is set to 0
05 00 EF EC EC E5 C8   STRL   "Hello"    ; The high bit is set to 1
```

This puts a delimited ASCII string in the object code with leading length word. Following hex values, if any, are not counted in the length. This is intended for use with GS/OS for Class 1 strings

LABEL

A Label is **case sensitive** and it has to be **unique**. Backward and forward references are allowed :

```
            JSR    GET_KEY
            ...
GET_KEY     LDA    $C000          ; Wait for a key
            BPL    GET_KEY
            BIT    $C010
            RTS
```

A label can't contain any characters less (in ASCII value) than the Zero (Space, !, ", #, $, %, &, ', (,), *, +, ', -, ., /). It must begin with a character other than 0 to 9. If you want to keep your source code compatible with **Merlin 16+**, the label length can't exceed **26** characters.

A label can be used without any Opcode on the line. In this case it has the same address value than the next line :

```
GET_KEY                            ; Wait for a key
            LDA    $C000
            ...
```

Labels starting with] or : characters are defined as **Local Labels**. Unlike Global Labels, they can be found at numerous places in the source code. Local Labels can't be used inside **Macros** or with **ENT** / **EXT** directives. The first Label in a program can't be a Local Label.

Local Labels starting with] can only be used for **backward** branching. They always refers the closest backward local label with the same name :

```
            LDX    #$00
```

```
]LOOP       LDA    TABLE1,X       ; Line 1
            BEQ    NEXT
            INX
            BRA    ]LOOP          ; Branch to Line 1
NEXT        LDY    #$00
]LOOP       LDA    TABLE2,Y       ; Line 2
            BEQ    END
            INY
            BRA    ]LOOP          ; Branch to Line 2
END         RTS
```

Local Labels starting with **:** can be used for **backward** and **forward** branching but their scope is limited by the two embracing Global Labels :

```
BEGIN       CPX    #$A0           ; :LOOP is defined between BEGIN and END
            BEQ    :LOOP
            LDX    #$00
:LOOP       LDA    TABLE1,X
            BEQ    END
            INX
            BRA    :LOOP
END         RTS
```

In the output text file created during assembling process, the **Local Labels** are replaced by **Global Labels** (using unique ids `ozunid_*`) to show how the assembler has resolved the references :

```
            LDX    #$00                                  |            LDX    #$00
]LOOP       LDA    TABLE1,X       ; Line 1               |ozunid_1    LDA    TABLE1,X       ; Line
            BEQ    NEXT                                  |            BEQ    NEXT
            INX                                          |            INX
            BRA    ]LOOP          ; Branch to Line 1     |            BRA    ozunid_1       ; Bra
NEXT        LDY    #$00                                  |NEXT        LDY    #$00
]LOOP       LDA    TABLE2,Y       ; Line 2               |ozunid_2    LDA    TABLE2,Y       ; Line
            BEQ    END                                   |            BEQ    END
            INY                                          |            INY
            BRA    ]LOOP          ; Branch to Line 2     |            BRA    ozunid_2       ; Bra
END         RTS                                          |END         RTS
```

EXPRESSION

Expressions are build using **Data** (number, label, ASCII character or current address *) combined with following Comparison / Arithmetic / Logical **Operators** (lowest priority comes first) :

```
<   =   >   #      Less_Than   Equal   More_Than   Not_Equal
+   -              Addition    Subtraction
*   /              Multiplication   Integer_Division
&   .   !          AND   OR   Exclusive_OR
-                  Unary_Negation
```

Beware about the usage of character * because it is both **Data** (current line address) and **Operator** (Multiplication).

By default, Expressions are evaluated from left to right, without caring about the operators priority :

```
1+2*3     is evaluated as 9, not 7 (1+2*3 = 3*3 = 9)
```

If you want to evaluate the expression using operators priority (=algebraically), you have to enclose the expression with **braces { }** (parenthesis are reserved for indirect addressing modes) :

```
{1+2*3}   is evaluated as 7 (1+2*3 = 1+6 = 7)
```

Comparison operators (< = > #) return **1** for **True** and **0** for **False**.

Here are few examples of common Expressions in **Merlin 32** :

```
1024+$FF                  ; 1024 plus 255 = 1279
"K"-"A"+1                 ; Ascii K minus Ascii A plus 1 = $CB - $C1 + 1 = 11
LABEL+2                   ; LABEL plus 2
LABEL2-LABEL1             ; LABEL2 minus LABEL1 = number of bytes between two
labels
*-2                       ; Current address minus 2
#$9F&"A"                  ; $9F AND $C1 = $81 (Control-A)
LABEL1/LABEL2             ; 0 if LABEL1 < LABEL2, 1 if LABEL1 >= LABEL2
```

EQUIVALENCE

The **EQU** (EQUivalence) directive is used to define constant values for which a meaningful name is desired. A constant name is **case sensitive** and can't start with a] character (reserved for **Variables**, see below). Forward references are not allowed so define your constants before using them (most of the time at the beginning of the program). You can either use **EQU** or **=** to define them :

```
HOME        EQU   $FC58     ; Clear Screen routine address
KDB         EQU   $C000     ; Keyboard Softswitch
```

```
PTR         =     *          ; Current address in the assembled source
PIXEL_SIZE  =     160*200    ; (160 bytes / line) * 200 lines
SCB_SIZE    =     256        ; 256 bytes (even if we only use the first 200)
PAL_SIZE    =     16*16*2    ; 16 palettes of 16 colors with 2 bytes / color
SHR_SIZE    =     PIXEL_SIZE+PAL_SIZE+PAL_SIZE   ; Total SHR Page size
```

The evaluation of a constant value is done at the definition time. So SHR_SIZE is properly evaluated as `32000+256+512` (=`32768`) and **not** as `160*200+256+16*16*2` (=`1032704` because of left-to-right evaluation).

Constants can be used anywhere in the Operand field :

```
             JSR   HOME      ; Clear Screen
WaitKey      LDA   KDB       ; Wait for a key
             BPL   WaitKey
```

VARIABLE

A Variable name is **case sensitive** and always beginning with a `]`. Variables are mostly used in Macros and Loops. The first declaration of a Variable is used for its initialization :

```
]LINE       =     $2000          ; First line address is $E1/2000
```

It can be redefined (=modified) as often as you need :

```
]LINE       =     ]LINE+160      ; Next line
             DA    ]LINE
```

Forward reference to a Variable is not allowed, so define your variables before using them.

LOOP

The **LUP** directive is used to repeat portions of the source code between the pseudo Opcode **LUP** and the `--^`. The number of iterations is defined by the Operand value :

```
]LINE       =      $2000          ; Build the Table of the 200 SHR lines
             LUP    200
             DA     ]LINE          ; Assembled as DA  $2000,$20A0,$2140,$21E0...
]LINE       =      ]LINE+$A0
             --^
```

The maximum number of iterations is $8000. The above use of incrementing variables in order to build a table **will not work** if used within a Macro.

If you want to use Labels in a loop, you have to use a **@** character in the Label name in order build dynamic label names :

```
             LUP    3
KBD_@        LDA    $C000
             BPL    KBD_@
             BIT    $C010
             --^
```

is assembled as :

```
KBD_Z        LDA    $C000          ; Each Label has a unique name
             BPL    KBD_Z
             BIT    $C010
KBD_Y        LDA    $C000
             BPL    KBD_Y
             BIT    $C010
KBD_X        LDA    $C000
             BPL    KBD_X
             BIT    $C010
```

The @ is replaced by uppercase letters (Z, Y, X, ..., B ,A). The maximum iteration number is **26**.

CONDITION

Conditions are used to build different code based on different situations (6502 / 65c02 processors, 8 bit / 16 bit environments, ROM / RAM context, Macro inner code...). There are two ways to use conditional pseudo opcodes in **Merlin 32** :

- **DO ELSE FIN**
- **IF ELSE FIN**

ELSE is optional but the **FIN** is mandatory. You can nest **DO** or **IF** :

```
DO     16_BIT         ; 8 bit or 16 bit ?
...                   ;   65c816 opcodes
ELSE
DO     6502           ; Apple IIe or IIc ?
...                   ;   6502 opcodes
ELSE
...                   ;   65c02 opcodes
```

```
FIN
FIN
```

If you want to keep your source code compatible with **Merlin 16+**, the nest depth is limited to **8** levels.

If the expression following the **DO** / **IF** is evaluated as **True** (everything but **0**), the code between the **DO** / **IF** and the **ELSE** (or between the **DO** / **IF** and the **FIN** if the **ELSE** is not there) is assembled :

```
DO     0                ; Turn assembly OFF
DO     1                ; Turn assembly ON
DO     16_BIT           ; Turn assembly ON if 16_BIT != 0
DO     LABEL1/LABEL2    ; Turn assembly OFF if LABEL1<LABEL2
DO     LABEL1-LABEL2    ; Turn assembly OFF if LABEL1=LABEL2
```

The **IF ELSE FIN** is used to check the status of the **M** and **X** bit (size of **Accumulator** and **X** / **Y registers**). M and X bits may be 0 (=16 bit) or 1 (=8 bit) so **MX** can be 0 (%00), 1 (%01) , 2 (%10) or 3 (%11) :

```
IF     MX/2             ; Turn assembly ON if M is  8 bit (%00/2=0, %01/2=0, %10/2=1,
%11/2=1)
IF     MX/2-1           ; Turn assembly ON if M is 16 bit (%00/2-1=-1 %01/2-1=-1
%10/2-1=0 %11/2-1=0)
IF     MX&1             ; Turn assembly ON if X is  8 bit (%00&1=0 %01&1=1 %10&1=0
%11&1=1)
IF     MX&1-1           ; Turn assembly ON if X is 16 bit (%00&1-1=-1 %01&1-1=0
%10&1-1=-1 %11&1-1=0)
IF     MX/3             ; Turn assembly ON if M and X are  8 bit (%00/3=0, %01/3=0,
%10/3=0, %11/3=1)
IF     MX!3/3           ; Turn assembly ON if M and X are 16 bit (%00!3/3=1,
%01!3/3=0, %10!3/3=0, %11!3/3=0)
```

The **IF ELSE FIN** can also be used to check the value of the leading character of a variable (mostly used in Macros) :

```
IF     "=]TEMP          ; Turn assembly ON if the first character of variable ]TEMP
is "
IF     #,]VAR1          ; Turn assembly ON is the first character of variable ]VAR1
is #
```

In the Operand of pseudo Opcode **IF**, you can use either **=** or **,** as separator between the value of the first character (comes first in the Operand) and the name of the Variable.

MACRO

A Macro is a user-named sequence of assembly language statements. You start the definition of the Macro with a **MAC** pseudo Opcode and you end it with **EOM** (**E**nd **O**f **M**acro) or **<<<** (alternate form). The name of the Macro takes place in the Label column :

```
WaitForKey  MAC             ; Define the WaitForKey Macro
WFK1         LDA     $C000  ; Wait until a key is pressed
             BPL     WFK1
             BIT     $C010
             <<<             ; End of Macro
```

In the source code, simply put the name of the Macro as Opcode to call it :

```
             SEP     #$30
             WaitForKey          ; Call WaitForKey Macro
             REP     #$30
             JSR     PlaySound
```

You can use alternate forms (PMC and >>>) to call a Macro from the source code :

```
             PMC     WaitForKey  ; Call WaitForKey Macro using PMC (Put Macro Call)
             >>>     WaitForKey  ; Call WaitForKey Macro using >>>
```

During assembly process, the Macro code will be inserted at the Macro call location :

```
             SEP     #$30
ozunid_1     LDA     $C000       ; Wait until a key is pressed
             BPL     ozunid_1
             BIT     $C010
             REP     #$30
             JSR     PlaySound
```

Because the same Macro can be used several times in the source code, the Macro inner Labels will be replaced by unique names (ozunid_*).

In the Output text file, we let the Macro call visible in the Source Code column and we identify it as **Macro** in the Line Type column :

```
+-------------+----+-------+------+--------------------+-----------------------------
-------------------------------------
| Line Type    | MX | Reloc | Size | Address Object Code|  Source
Code
```

```
+-------------+----+-------+------+--------------------+-----------------------------
-------------------------------------
...
| Code        | 00 |       |    2 | 8000 : E2 30       |                    SEP
#$30
| Macro       | 11 |       |    0 | 8000               |
WaitForKey
| Code        | 11 |       |    3 | 8002 : AD 00 C0    | ozunid_1          LDA
$C000          ; Wait until a key is pressed
| Code        | 11 |       |    2 | 8005 : 10 FB       |                    BPL
ozunid_1
| Code        | 11 |       |    3 | 8007 : 2C 10 C0    |                    BIT
$C010
| Code        | 11 |       |    2 | 800A : C2 30       |                    REP
#$30
| Code        | 00 |       |    3 | 800C : 20 A2 80    |                    JSR
PlaySound
...
```

Forward reference to a Macro is not possible, so a Macro must be defined before it is called. Usually, you declare the Macros at the start of the source code. You may also write the Macros in dedicated files and include such files using a **USE** directive :

```
            USE     Locator.Macs   ; Use Macros defined in the Locator.Macs.s file
located in the Macro folder
            USE     4/Mem.Macs     ; Use Macros defined in the Mem.Macs.s file
located in the Macro folder
```

The Operand indicates the names of the Macro definition file (without the **.s** extension). By convention, the file name ends with `.Macs` but it is not mandatory. In **Merlin 16+**, the Macro definition files are stored in dedicated sub-folders, so you have to enter the relative file path (4/Mem.Macs). In **Merlin 32**, all the Macro definition files are stored in the `Macro Folder` (second parameter of the command line), so we don't need anymore the subfolder part, we just look at the file name (for **Merlin 16+** compatibility, you can let the subfolder path without any issue, it will be ignored).

Macros can receive parameters (up to **8**) referenced as Macro variables `]1` to `]8` :

```
WaitForKey  MAC             ; Define the WaitForKey Macro
WFK1        LDA     $C000   ; Wait until a key is pressed
            BPL     WFK1
            BIT     $C010
            CMP     ]1      ; Check if the Key is the expected one
            BNE     WFK1
            <<<             ; End of Macro
```

In the source code, add the parameter value after the Macro name, as Operand value :

```
            SEP     #$30
            WaitForKey #$95  ; Wait for -> key (right arrow)
            REP     #$30
```

If you are using **PMC** or **>>>** form, you have to group the Macro name and the parameters together in the Operand column :

```
            SEP     #$30
            PMC     WaitForKey,#$95  ; Wait for -> key (right arrow)
            REP     #$30
```

You can use the following characters to separate the Macro name from the parameters : `.` `/` `,` `-` `(` `Space`

If your Macro receives several parameters, you have to use the `;` as separator in the call :

```
Move        MAC             ; Define the Move Macro
            LDA     ]1
            STA     ]2
            <<<             ; End of Macro

...
            Move    $00;$02          ; Call the Move Macro with two addresses
            Move    #$00;$02         ; Call the Move Macro with one constant and
one address
            Move    #"A";(STRING),Y  ; Call the Move Macro with one constant and an
indexed address
```

There is no control of the parameters value. You can put there what you want (constant, address, label, expression...). The check will be done, by the assembly process, after the substitution.

The Macro variable `]0` returns the number of variables in the parameter list of the Macro call. This lets you create Macros with flexible input using conditional pseudo Opcodes **DO**, **ELSE** and **FIN** :

```
Pull        MAC             ; Define the Pull Macro
            PLA
            DO      ]0      ;  If a parameter is given (]0 != 0)
```

```
                STA     ]1          ;     Use it as target address to store the data
                FIN                 ;  End of Condition
                <<<                 ; End of Macro

...
                Pull                ; Pull a value off the stack

                Pull    LABEL       ; Pull a value off the stack and store it in location
LABEL
```

The conditional pseudo Opcodes **IF**, **ELSE** and **FIN** can be used to distinguish address (or Label) from constant :

```
PushWord    MAC                 ; Define the PushWord Macro
            IF      #=]1        ;  If a the first character of parameter ]1 is #
(=constant)
            PEA     ]1          ;    Push the constant value on the stack with a PEA
            ELSE                ;  Else
            LDA     ]1          ;    Load the value in the accumulator
            PHA                 ;    Push the accumulator on the stack with a PHA
            FIN                 ;  End of Condition
            <<<                 ; End of Macro

...
            PushWord #$000 ; Push a constant value on the stack

            PushWord LABEL ; Push a value stored at LABEL address on the stack
```

A Macro code can call another Macro. If you want to keep compatibility with **Merlin 16+**, the nest depth is limited to **15** levels :

```
MoveWord    MAC                 ; Define the MoveWord Macro (Accumulator is 16 bit)
            LDA     ]1
            STA     ]2
            <<<                 ; End of Macro

MoveLong    MAC                 ; Define the MoveLong Macro (Accumulator is 16 bit)
            MoveWord   ]1+2;]2+2
            MoveWord   ]1;]2
            <<<                 ; End of Macro
```

You can also **nest** the definition of the inner Macro (**MoveWord**) within the code of the calling Macro (**MoveLong**). The final <<< closes the two Macros together.

```
MoveLong    MAC                 ; Define the MoveLong Macro
            MoveWord   ]1+2;]2+2

MoveWord    MAC                 ;    Define the MoveWord Macro inside the MoveLong Macro
definition
            LDA     ]1
            STA     ]2

            <<<                 ; End of both Macros
```

The **MoveLong** Macro is assembled as follows :

```
            LDA     ]1+2    ; From MoveWord call
            STA     ]2+2
            LDA     ]1      ; From MoveWord definition
            STA     ]2
```

ORIGIN

If your program is supposed to run from a **fixed memory address**, you have to use the **ORG** directive at the start of the source code to define the start address. The operand may be 16 bit (for bank $00) or 24 bit :

```
            ORG   $2000          ; The program will run from bank $00, at address $2000

            ORG   $038000        ; The program will run from bank $03, at address $8000
```

If **ORG** directive is missing, the default start address will be `$8000` in bank `$00`.

If your **ORG** operand is inferior to `$0100` or inferior to `$000100`, the code start address will match with **Direct Page** (former Page Zero) and all references from `$0000` to `$00FF` will use Direct Page addressing mode :

```
                                ORG   $0000          ; The program will run from
bank $00, at address $0000

$00/0000 : A5 03               LDA   SCORE          ; Beware, the Direct addressing
mode has been used here
$00/0002 : 60                  RTS
$00/0003 : 00 20        SCORE  HEX   0020
```

If your code is suppose to run from $0000 in a bank which is not bank $00, think about giving a 24 bit address as Operand (ex: ORG $030000).

You can use **ORG** directive without Operand when several **ORG** are used in the source code, as a **RE-ORG** to re-establish the correct address pointer after a segment of code which has a different **ORG** :

```
                                ORG  $8000          ; This code is assembled to run
from $00/8000

$00/8000 : A9 00 20            LDA  #$2000
$00/8003 : 20 AC 80            JSR  SCREEN
$00/8006 : 4C 0D 80            JMP  NEXT

                                ORG  $0400          ; This code is assembled to run
from $00/0400
$00/0400 : AD 00 C0            LDA  $C000
$00/0403 : 60                  RTS

                                ORG                 ; RE-ORG in $00/800D
$00/800D : 8D AE 80    NEXT    STA  VBL
```

If you want to write16 bit **relocatable code**, you have to use the directive **REL** at the start of your program :

```
                                REL                ; Relocatable code for Apple
IIgs S16 executable (OMF 2.1 format)
                                DSK   Cogito.L

$00/0000 : A9 00 20            LDA  #$2000
$00/0003 : 20 AC 80            JSR  SCREEN
```

Merlin 32 will assemble the source code from a virtual $00/0000 address (without Direct Page addressing mode usage) and the object code will be embedded into an **OMF file** (release 2.1). The output is a **S16** program running under **Prodos 16** or **GS/OS**.

DISK

The following directives are used to include external files into your project or to define the properties of the output files created by the assembly process.

The **USE** and **PUT** directives are used to insert the content of a **Text** file (Source or Macro) at the location of the Directive :

```
               USE     4/Int.Macs       ; Use Int.Macs.s Macro file definitions
               USE     4/Locator.Macs   ; Use Locator.Macs.s Macro file definitions
...
               PUT     Cogito.Main      ; Insert Cogito.Main.s Source file
               PUT     Cogito.Bout      ; Insert Cogito.Bout.s Source file
```

By convention, the **USE** directive is used to include Macros (*.Macs.s) and Equivalence files and the **PUT** directive is used to include Source Code files. Because **Merlin 16+** source files were limited to 64 KB, there was a need to cut a large source file into smaller ones. Such restriction doesn't exist anymore in **Merlin 32** but it is always a good idea to split your project into small independent files (Music, Graphic, Data Compression, I/O, Mouse, Joystick; Keyboard...) so you can re-use some of the files among several projects. If you want to use your source files in **Merlin 16+**, keep them < 64 KB.

The **PUTBIN** directive is used to insert the content of a **Binary** file at the location of the Directive :

```
Logo           PUTBIN Cogito.Logo      ; Insert Cogito.Logo Binary file
Sound          PUTBIN Cogito.Sound     ; Insert Cogito.Sound Binary file
```

The content of the Binary file is transfered in the source code as **Hexadecimal** data :

```
Logo           HEX    00,12,59,AE,00,11,FE,8C,A9,D4,14,87,CD,DE,9A,6E
               ...

Sound          HEX    87,E6,4A,26,41,6E,FF,AE,31,58,2A,F9,6C,D7,28,9B
               ...
End
```

The size of the Binary file can be computed inside the source code by using the labels :

```
LogoSize     EQU     #Sound-#Logo
SoundSize    EQU     #End-#Sound
```

Beware, the **PUTBIN** directive does not exist in **Merlin 16+**. Merlin 16+ lets you include Binary files during the Link process (you have to use the LNK directive in the Linker file).

The usage of **USE** and **PUT** / **PUTBIN** directives are limited to **ONE** source file (named *Master* source file) : you can't use PUT / PUTBIN directives within a PUT file (same for USE directive). The *Master* source file contains all the USE and PUT / PUTBIN directives and this is the one we put as source file parameter of the **Merlin 32** command line.

The **DSK**, **SAV** and **LNK** directives are used to define the name of the output file created by the assembly process. The **DSK** directive is used to define the name of the output binary file for the code *following* the **DSK** directive :

```
DSK     Cogito      ; Assemble the following code as 'Cogito' file
ORG     $8000
LDA     #$0000
...
```

while the **SAV** directive is used to define the name of the output binary file for the code located *before* the **SAV** directive :

```
ORG     $8000
LDA     #$0000
...
SAV     Cogito      ; Assemble the previous code as 'Cogito' file
```

You may encounter several **DSK** or **SAV** directives in the same source code. In this case, the assembly process will generate several output files :

```
DSK     CogitoMain ; Assemble the following code as 'CogitoMain' file
ORG     $030000
LDA     #$0002
...

DSK     CogitoAux  ; Assemble the following code as 'CogitoAux' file
ORG     $038000
LDX     #$00A0
..
```

The **LNK** directive is often used for relocatable code, in association with the **REL** directive. In **Merlin 32**, it has the same behavior than the **DSK** directive. It is located at the beginning of the source file :

```
REL                ; Relocatable Code
LNK     Cogito.l   ; Assemble and Link the file as a S16 program named
'Cogito'
LDA     #$0002
...
```

The **TYP** directive is used to set the output file type (one byte : $00-$FF). It is usually associated with **DSK** or **SAV** directives :

```
TYP     $06              ; Binary File Type
DSK     File1
...
```

Because **Merlin 32** creates the output binary file on a Windows file system, there is no way to set the Prodos file type. The **TYP** directive will be ignored by **Merlin 32** (you can let it in the source code for **Merlin 16+** compatibility purpose). If you want to set the Prodos file type, you have to set it during the transfer of file into a Prodos disk image. If you are using **CADIUS** for this job, you can define the file type and the file attributes in the **_FileInformation.txt** file (see **CADIUS** documentation for more details).

MISC

The following miscellaneous directives are not often used in Source code so we only provide here basic explanations for them. Please refer to the **Merlin 16+** manual for more details.

DUM **DUM**my section
DEND **D**ummy **END**

This defines a section of code that will be examined for the values of the labels but will produce no object code. The **DUM** directive uses as Operand the **ORG** value of this section :

```
            DUM     $E12000       ; SHR Page is located in $E12000 :
PIXEL       DFB     160*200      ;   200 lines
SCB         DFB     256           ;   200 SCB used
PAL0        DFB     32            ;   Palette 0
PAL1        DFB     32            ;   Palette 1
PAL2        DFB     32            ;   Palette 2
            ...
            DEND

            LDX     #$0000
            LDAL    PIXEL,X
            ...
```

DUM and **DEND** are often used to create a set of labels that will exist outside 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.

The **DUM** and **DEND** can be efficiently used to describe the organization of the **Direct Page** (list of variables) :

```
            DUM     $000000       ; Direct Page is located at $000000
UP          HEX     0000          ;   $00
DOWN        HEX     0000          ;   $02
LEFT        HEX     0000          ;   $04
RIGHT       HEX     0000          ;   $06
BUTTON      HEX     0000          ;   $08
```

```
...
DEND

LDA     LEFT          ; = LDA  $04
CMP     #$0001
...
```

END **END** of source file

Tells the assembler to ignore the rest of the source code (including Labels).

CHK place a **CH**ec**K**sum in object code

This places a **checksum byte** into object code at the location of the **CHK** directive. This is usually placed at the end of the program and can be used by the 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. Of course, such directive can't be used with relocatable program, because the loader is patching the program's addresses in memory at runtime.

DAT place the current **DAT**e in object code

This places the **current Date/Time** (date of the build) in the object code, as a Text string (High bit Clear or Set). The Operand value (**1** to **8**) is used to control the Date/Time format and the encoding :

```
              DAT     1               ; Date Only, High Bit Set Ascii   : "31-
DEC-14"
              DAT     2               ; Date Only, High Bit Set Ascii   :
"12/31/14"
              DAT     3               ; Date/Time, High Bit Set Ascii   : "31-
DEC-14   5:46:12 PM"
              DAT     4               ; Date/Time, High Bit Set Ascii   :
"12/31/14   5:46:12 PM"
              DAT     5               ; Date Only, High Bit Clear Ascii : '31-
DEC-14'
              DAT     6               ; Date Only, High Bit Clear Ascii :
'12/31/14'
              DAT     7               ; Date/Time, High Bit Clear Ascii : '31-
DEC-14   5:46:12 PM'
              DAT     8               ; Date/Time, High Bit Clear Ascii :
'12/31/14   5:46:12 PM'
```

ERR force **ERR**or

ERR will force an error during the assembly process if the expression has a non-zero value :

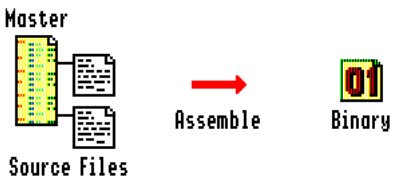
```
              ERR     *-1/$9600       ; Error if PC > $9600
```

This may be used to ensure your program does not exceed a specific length.

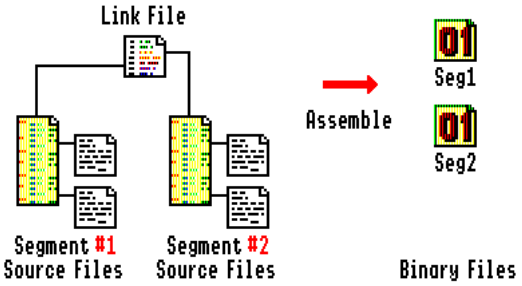
> Building Multi-Segments Fixed-Address Files

Fixed-Address Binary files are used in 65c816 based systems that do not have an Operating System providing dynamic relocation : **Apple IIgs** running **Prodos 8** or **Custom OS** (No Tools...), the **SNES**, various **65c816 CPU boards**... These binary files are loaded in memory and executed at a fixed address (defined during the assembly process). The binary files contain the object code, nothing else (no header, no checksum, no padding...). If you are using an Apple IIgs running **GS/OS**, it is more convenient to create **relocatable OMF programs** (see next section).

For small projects, the programs can be smaller than **64 KB** and fit in **One** Binary file. In this case, you don't need dedicated **Merlin 32** syntax. Simply use the directives **ORG** and **DSK** (or **LNK**) in your *Master* source file to build such programs. **Merlin 32** takes the *Master* source file as parameter, loads the other source files (inserted in the project using **PUT** directives) and assembles all these files as a **One** Fixed-Address Binary program :



If your target program is larger than **64 KB**, it has to be split into **several** Binary Segments (each Segment is < **64 KB**) and assembled & linked together to build the Binary files. This time, **Merlin 32** takes a **Link file** as parameter. This **Link** file contains information about the several Segments (*Master* Source files path, Target Binary files names...). **Merlin 32** loads all the files involved in the project (Link file, *Master* files, extra Sources files...) and assembles all of them as **Several** Fixed-Address Binary files :



The **Link file** format is very close to the Source files. It uses the same Label / Opcode / Operand / Comment line structure and accept full Line Comments like in the Source files (`*` and `;`). The prefix is usually `.S` or `.txt` and the file is divided into several sections (one for the header + one per Binary Segment) :

```
*--------------------------*
*           COGITO          *
*                            *
*       Brutal  Deluxe       *
*--------------------------*

             TYP     $06             ; Binary File / Fixed Address

*-------------------------
*  Segment #1
*-------------------------

             ASM     Cogito.Main.s   ; Master Source File for Segment #1
             SNA     Main             ;    Segment Name ('Main')

*-------------------------
*  Segment #2
*-------------------------

             ASM     Cogito.Aux.s    ; Master Source File for Segment #2
             SNA     Aux              ;    Segment Name ('Aux')

*-------------------------
```

The directives found in the Link file are used to define the Binary files names and the Master Source files paths.

The following directives should be found at the top of the Link file. They should not appear more than **one time** in the Link file (the **TYP** directive with value **$06** is mandatory) :

TYP : GS/OS File Type

The value must be **$06** (Binary file). This one byte value specifies the **Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image.

AUX : GS/OS File Auxiliary Type

This two bytes value specifies the **Auxiliary Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image. The default value is **$0000**.

The following directives are used to define the **Segments** properties. They should not appear more than one time for **each Segment** of the Link file. The **ASM** directive is mandatory, it defines the beginning of a new Segment and the end of the previous one :

ASM : *Master* source file path to be assembled

Defines the file name (or Path) of the *Master* source file for this Segment.

SNA : Segment Name = Binary File name

Specifies the name of the segment. If this directive is missing, the name of the segment is taken from the Operand of the **DSK** (or **LNK**) directive found in the *Master* source file.

The type of the program (Binary file or other) is defined by the GS/OS file Type. Because the output of the assembly process goes to a Windows file system, there is no way to set the file Type and AuxType. You have to set them manually while you are transferring the file back to a Prodos disk image (you can also take advantage of **CADIUS** facilities with its `_FileInformation.txt` file).

The Source files of a Segment in a Multi-Segment program look like the same than in Single-Segment program. They both have a **ORG** directive in the *Master* source file to define the code as Fixed-Address. In Multi-Segments source files, you can use **2** new directives, all of them used to refer to addresses located in another Segment of the program :

ENT : defines a label as an ENTry label in a REL Segment. It is 'visible' from the other Segments of the program.
EXT : defines a label EXTernal to the current REL Segment. It is located in another Segment of the program.

The following example shows how the source code from Segment #1 can call a sub-routine or read data located in Segment #2 :

```
*--------------------------
*  Segment #1 Master File
*--------------------------

            ORG     $030800       ; Fixed-Address code
            DSK     Main.l        ; Binary File Name 'Main'

            MX      %00           ; 16 bit

WaitForKey  EXT                   ; Define EXTernal Labels
SHRLineTab  EXT                   ; located in another
Segment

            PHK
            PLB

            JSL     WaitForKey    ; Wait for Key press

            LDX     #$0000
LOOP        LDAL    SHRLineTab,X  ; Get Line Address
            JSR     ClearLine
            INX
            INX
            CPX     #400
            BNE     LOOP

            ...
```

```
*--------------------------
*  Segment #2 Master File
*--------------------------

            ORG     $052000
            DSK     Aux.l

            MX      %00

WaitForKey  ENT
Subroutine
            LDAL    $00BFFF
            BPL     WaitForKey
            STAL    $00C010
            RTL

SHRLineTab  ENT
Table
]LINE       =       $2000
            LUP     200
            DA      ]LINE
             $2000,$20A0,$2140,$21E0...
]LINE       =       ]LINE+$A0
            --^
```

We define in the Segment #2 two **global** labels, `WaitForKey` and `SHRLineTab`, so they can be called from another segment of the same program. We simply add the **ENT** (**ent**ry point for other segments) directive as Opcode of the Labels.

In Segment #1, where we need to refer to these Labels, we declare them as **EXT** (**ext**ernal to the current segment), at the beginning of the source code. So we can use them anywhere in the source code of Segment #1, but always using **Long** addressing mode (the two segments may be located in different memory banks).

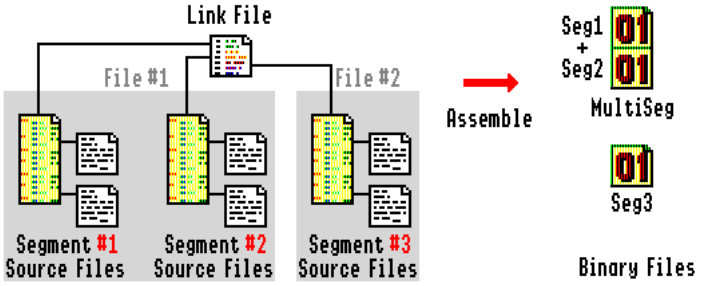
You can use EXTernal labels in expressions, but always using forward reference (EXT Label + Constant), never backward (EXT Label - Constant). You are not authorized to build expression involving several labels, where at least one is External (EXT Label - local Label + 2). You can use the Addressing Mode operators (`< > ^`) on them :

```
            LDAL    SHRLineTab+2,X

            PEA     <WaitForKey
            PEA     ^WaitForKey
```

Merlin 32 will assemble both segments separately and will search for **EXT**ernal labels during the linkage (creation of several Binary files). If an EXTernal label can't be found in the other segments of the program, an error message will be displayed and the whole assembly process will fail. You won't get the Binary files created but you will get the output Text files (one per segment) created during the assembly step.

> Building Multi-Segments Fixed-Address Merged Files

This is about the same logic than the previous **Multi-Segments Fixed-Address** files except the fact than **Several** Segments may now be merged into **One** Binary file :



The **Link** file now defines the **ORG** **Address** of each Binary File and its **Name** (using the **DSK** directive) :

```
*--------------------------*
*           COGITO         *
*                          *
*      Brutal  Deluxe      *
*--------------------------*

            TYP     $06             ; Binary File / Fixed Address

*--------------------------
*  File #1
*--------------------------
```

```
                DSK     MultiSeg        ; File Name for File #1
                ORG     $000800         ; ORG Address for File #1

        *--------  Segment #1

                ASM     Cogito.Main.s   ; Master Source File for Segment #1
                SNA     Segment1        ;    Segment Name ('Segment1')

        *--------  Segment #2

                ASM     Cogito.Aux.s    ; Master Source File for Segment #2
                SNA     Segment2        ;    Segment Name ('Segment2')

        *--------------------------
        *  File #2
        *--------------------------

                DSK     Seg3            ; File Name for File #2
                ORG     $030300         ; ORG Address for File #2

        *--------  Segment #3

                ASM     Cogito.Util.s   ; Master Source File for Segment #3
                SNA     Segment3        ;    Segment Name ('Segment3')

        *--------------------------
```

The directives found in the Source files (**REL**, **ORG**, **DSK**...) are ignored and the directives defined in the **Link** file take precedence. If several Segments are merged into one binary files, the **ORG Address** of the first segment is defined by the **ORG** Directive of the **Link** file and the other Segments (of the same file) starts at the end of the previous Segment (**ORG Address** of Segment **#N** = 1 + **ORG Address** of Segment **#N-1**). With one **Link** file, you can create as many **Binary files** as you want, using as many **Segments** as you want (within the limit of 64 KB per Binary file).

The following directives should be found at the top of the Link file. They should not appear more than **one time** in the Link file (the **TYP** directive with value **$06** is mandatory) :

TYP : GS/OS File Type

The value must be **$06** (Binary file). This one byte value specifies the **Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image.

AUX : GS/OS File Auxiliary Type

This two bytes value specifies the **Auxiliary Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image. The default value is **$0000**.

The following directives are used to define the **Files** properties. They should not appear more than one time for **each File** of the Link file. The **DSK** and the **ORG** directives are mandatory (**DSK** before **ORG**), they define the beginning of a new **File** and the end of the previous one :

DSK : Binary File name

Defines the File Name of the Binary file to be created.

ORG : ORG Address of the Binary file

Set the **ORG Address** of the first Segment of the file.

The following directives are used to define the **Segments** properties. They should not appear more than one time for **each Segment** of the Link file. The **ASM** directive is mandatory, it defines the beginning of a new Segment and the end of the previous one :

ASM : *Master* source file path to be assembled

Defines the file name (or Path) of the *Master* source file for this Segment.

SNA : Segment Name

Specifies the name of the segment. If this directive is missing, the name of the segment is taken from the Operand of the **DSK** (or **LNK**) directive found in the *Master* source file.

In Multi-Segments source files, you can use **2** new directives, all of them used to refer to addresses located in another Segment of the program :

ENT : defines a label as an ENTry label in a REL Segment. It is 'visible' from the other Segments of the program.
EXT : defines a label EXTernal to the current REL Segment. It is located in another Segment of the program.

The following example shows how the source code from Segment #1 can call a sub-routine or read data located in Segment #2 (there is no **ORG** directives in the source files because the **ORG Address** is set from the **Link** file):

```
*--------------------------
*  Segment #1 Master File
*--------------------------

            MX       %00            ; 16 bit

WaitForKey  EXT                     ; Define EXTernal Labels
SHRLineTab  EXT                     ; located in another
Segment

            PHK
            PLB

            JSL      WaitForKey     ; Wait for Key press

            LDX      #$0000
LOOP        LDAL     SHRLineTab,X   ; Get Line Address
            JSR      ClearLine
            INX
            INX
            CPX      #400
            BNE      LOOP

            ...
```

```
*--------------------------
*  Segment #2 Master File
*--------------------------

            MX      %00

WaitForKey  ENT
Subroutine
            LDAL    $00BFFF
            BPL     WaitForKey
            STAL    $00C010
            RTL

SHRLineTab  ENT
Table
]LINE       =       $2000
            LUP     200
            DA      ]LINE
 $2000,$20A0,$2140,$21E0...
]LINE       =       ]LINE+$A0
             --^
```

We define in the Segment #2 two **global** labels, `WaitForKey` and `SHRLineTab`, so they can be called from another segment of the same program. We simply add the **ENT** (**ent**ry point for other segments) directive as Opcode of the Labels.

In Segment #1, where we need to refer to these Labels, we declare them as **EXT** (**ext**ernal to the current segment), at the beginning of the source code. So we can use them anywhere in the source code of Segment #1, but always using **Long** addressing mode if the two segments are **not** merged into the same Binary file (the two segments may be located in different memory banks).

You can use EXTernal labels in expressions, but always using forward reference (EXT Label + Constant), never backward (EXT Label - Constant). You are not authorized to build expression involving several labels, where at least one is External (EXT Label - local Label + 2). You can use the Addressing Mode operators (`< > ^`) on them :

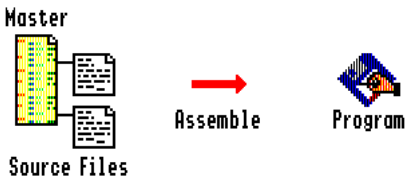
```
LDAL    SHRLineTab+2,X

PEA     <WaitForKey
PEA     ^WaitForKey
```

Merlin 32 will assemble both segments separately and will search for **EXT**ernal labels during the linkage (creation of several Binary files). If an EXTernal label can't be found in the other segments of the program, an error message will be displayed and the whole assembly process will fail. You won't get the Binary files created but you will get the output Text files (one per segment) created during the assembly step.

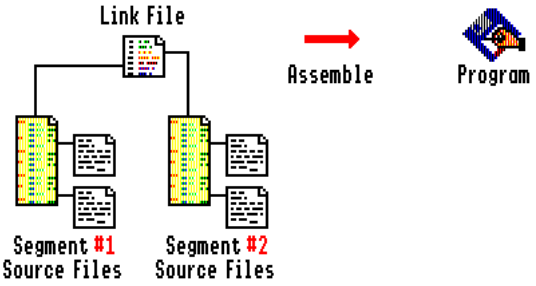
> Building Multi-Segments OMF Files

OMF files are the core of any executable code on the **Apple IIgs** system (S16, Exe, CDA, NDA, FST, PIF, Library, Tool...). Each OMF file contains one or more segments. Each segment in an OMF file contains a set of records that provide relocation information and contain code or data. The System Loader loads the code parts in memory and process the information found in the relocation dictionary to patch the addresses of the code. The code located in Segment #1 is executed. Other segments may contains Code or Data.

For small projects, the executables can be smaller than **64 KB** and fit in One-Segment OMF files. In this case, you don't need dedicated **Merlin 32** syntax. Simply use the directives **REL** and **LNK** in your *Master* source file to build such executables. **Merlin 32** takes the *Master* source file as parameter, loads the other source files (inserted in the project using **PUT** directives) and assembles all these files as a **Single-Segment** OMF program :



If your target program is larger than **64 KB**, it has to be splited into **several** OMF Segments (each OMF Segment is < **64 KB**) and assembled & linked together to build the executable. This time, **Merlin 32** takes a **Link file** as parameter. This **Link** file contains information about the several Segments (*Master* Source file path, Segment properties, Target Program name...). **Merlin 32** loads all the files involved in the project (Link file, *Master* files, extra Sources files...) and assembles all of them as a **Multi-Segments** OMF program :



The **Link file** format is very close to the Source files. It uses the same Label / Opcode / Operand / Comment line structure and accept full Line Comments like in the Source files (* and ;). The prefix is usually .S or .txt and the file is divided into several sections (one for the Program + one per OMF Segment) :

```
*--------------------------*
*           COGITO          *
*                            *
*       Brutal  Deluxe       *
*--------------------------*

             DSK     Cogito          ; Program File Name is 'Cogito'
             TYP     $B3             ; S16, GS/OS Application
             XPL                     ; Add the ~ExpressLoad Segment

*-------------------------
*  Segment #1
*-------------------------

             ASM     Cogito.Main.s   ; Master Source File for Segment #1
             DS      0               ;   Number of bytes of 0's to add at the end of the
Segment
             KND     #$1100          ;   Type and Attributes ($11=Static+Bank
Relative,$00=Code)
             ALI     None            ;   Boundary Alignment (None)
             LNA     Cogito.S16      ;   Load Name ('Cogito.S16')
             SNA     Main            ;   Segment Name ('Main')

*-------------------------
*  Segment #2
*-------------------------

             ASM     Cogito.Aux.s    ; Master Source File for Segment #2
             DS      0               ;   Number of bytes of 0's to add at the end of the
Segment
             KND     #$1100          ;   Type and Attributes ($11=Static+Bank
Relative,$00=Code)
             ALI     None            ;   Boundary Alignment (None)
             LNA     Cogito.S16      ;   Load Name ('Cogito.S16')
             SNA     Aux             ;   Segment Name ('Aux')

*-------------------------
```

The directives found in the Link file are used to define the Program file name and the OMF Segments properties. Some OMF Segment general information like NUMLEN (length, in bytes, of a number field), VERSION (version number of the OMF), REVISION (revision number of the OMF) or NUMSEX (order of the bytes in a number field) receive **constant fixed** values. There are no directive in **Merlin 32** Link file to change their values. Refer to the **Apple IIgs GS/OS Reference** book (**Appendix F : Object Module Format**) for full details about data structure definitions and naming convention used in OMF Segments.

The following directives should be found at the top of the Link file. They should not appear more than **one time** in the Link file (the **DSK** directive is mandatory) :

DSK : Name of the Program file

Defines the name (or Path) of the output program file. A valid **Prodos** File **Name** is **15** characters long (max), starts with a letter (**A-Z** or **a-z**), may contains Numerics (**0-9**) or a period (.).

TYP : GS/OS File Type

This one byte value specifies the **Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image. The default value is **$B3** (GS/OS application).

Some common GS/OS file types related to program files are listed below :

```
$B2     LIB     Library
$B3     S16     GS/OS or ProDOS 16 application
$B4     RTL     Run-time library
$B5     EXE     Shell application
$B6     PIF     Permanent initialization
$B7     TIF     Temporary Initialization
$B8     NDA     New desk accessory
$B9     CDA     Classic desk accessory
$BA     TOL     Tool set file
```

```
$BB     DVR     Apple IIgs Device Driver File
$BC     LDF     Generic load file
$BD     FST     GS/OS file system translator
```

AUX : GS/OS File Auxiliary Type

This two bytes value specifies the **Auxiliary Type** of the file under a Prodos file system. Such value is stored in the `_FileInformation.txt` file and can be used by **Cadius** during the transfer of the program to the disk image. The default value is **$0000**.

XPL : Add ExpressLoad Segment

If set, it asks **Merlin 32** to add a Segment named ~ExpressLoad at first position in the OMF file. This Segment is a summary of all the following Segments available in the OMF file. It is used by GS/OS to speed up the load of the program.

The following directives are used to define the Segment properties. They should not appear more than one time for **each Segment** of the Link file. The **ASM** directive is mandatory, it defines the beginning of a new Segment and the end of the previous one :

ASM : *Master* source file path to be assembled

Defines the file name (or Path) of the *Master* source file for this Segment.

DS : Number of zero bytes to reserve at the end of the file

Specifies the number of bytes of 0's to add to the end of the Segment. This can be used in an object Segment instead of a large block of zeros at the end of a Segment.
The default value is **0**.

KND : Type and Attributes

This two bytes value specifies the type and the attributes of the Segment. A Segment can have only one type byte but any combination of attributes.

The **low** byte defines the type :

```
$00     Code
$01     Data
$02     Jump-Table segment
$04     Pathname segment
$08     Library dictionary segment
$10     Initialization segment
$12     Direct-page/stack segment
```

The **high** byte defines the attributes list :

```
%0000_0001     Bit 0 : If 1 = Bank-relative segment
%0000_0010     Bit 1 : If 1 = Skip segment
%0000_0100     Bit 2 : If 1 = Reload segment
%0000_1000     Bit 3 : If 1 = Absolute-bank segment
%0001_0000     Bit 4 : If 0 = Can be loaded in special memory
%0010_0000     Bit 5 : If 1 = Position independent
%0100_0000     Bit 6 : If 1 = Private
%1000_0000     Bit 7 : If 0 = Static, If 1 = Dynamic
```

The default value is #$1100 (Static+Bank Relative,Code). You can't have more than one **Jump-Table** or **Direct-page/stack** segment per program file.

ALI : Boundary Alignment

Indicates the boundary on which the segment must be aligned.

The possible values are :

```
BANK : The segment is to be aligned on a Bank boundary  ($10000)
PAGE : The segment is to be aligned on a Page boundary  (  $100)
NONE : No alignment is needed                             (    $0)
```

The default value is **NONE**.

BSZ : Bank Size

Number indicating the maximum memory-bank size for then segment.

For **Code** segments, the value is $10000 (64 KB). For **Data** segments, the value is between $00 and $10000 (64 KB). A value of 0 indicates that the segment can cross bank boundaries.

The default value is **$10000** (64 KB) .

ORG : Origin

Indicates the absolute address at which the segment is to be loaded in memory.

A value of 0 indicates that the segment is relocatable and can be loaded anywhere in memory.

The default value is **0**.

LNA : Load Name

Specifies the name of the load segment that will contain the code generated by the linker for this segment. This is usually left empty. The maximum length is 10 bytes.

SNA : Segment Name

Specifies the name of the segment. If this directive is missing, the name of the segment is taken from the Operand of the **DSK** (or **LNK**) directive found in the *Master* source file.

The type of the program (GS/OS application, Shell application, Permanent Init file, New desk accessory, Classic desk accessory, Tool set file...) is defined by the GS/OS file Type. Because the output of the assembly process goes to a Windows file system, there is no way to set the file Type and AuxType. You have to set them manually while you are transfering the file back to a Prodos disk image (you can also take advantage of **CADIUS** facilities with its `_FileInformation.txt` file).

The Source files of a Segment in a Multi-Segment program look like the same than in Single-Segment program. They both have a **REL** directive in the *Master* source file to define the code as relocatable. In Multi-Segments source files, you can use **2** new directives, all of them used to refer to addresses located in another Segment of the program :

ENT : defines a label as an ENTry label in a REL Segment. It is 'visible' from the other Segments of the program.
EXT : defines a label EXTernal to the current REL Segment. It is located in another Segment of the program.

The following example shows how the source code from Segment #1 can call a sub-routine or read data located in Segment #2 :

```
*--------------------------
*  Segment #1 Master File
*--------------------------

            REL                     ; The code is relocatable
            DSK     Main.l          ; Segment Name 'Main'
            MX      %00             ; 16 bit

WaitForKey  EXT                     ; Define EXTernal Labels
SHRLineTab  EXT                     ; located in another
Segment

            PHK
            PLB

            JSL     WaitForKey      ; Wait for Key press

            LDX     #$0000
LOOP        LDAL    SHRLineTab,X    ; Get Line Address
            JSR     ClearLine
            INX
            INX
            CPX     #400
            BNE     LOOP

            ...
```

```
*--------------------------
*  Segment #2 Master File
*--------------------------

            REL
            DSK     Aux.l
            MX      %00

WaitForKey  ENT
Subroutine
            LDAL    $00BFFF
            BPL     WaitForKey
            STAL    $00C010
            RTL

SHRLineTab  ENT
Table
]LINE       =       $2000
            LUP     200
            DA      ]LINE
             $2000,$20A0,$2140,$21E0...
]LINE       =       ]LINE+$A0
            --^
```

We define in the Segment #2 two **global** labels, `WaitForKey` and `SHRLineTab`, so they can be called from another segment of the same program. We simply add the **ENT** (**ent**ry point for oher segments) directive as Opcode of the Labels.

In Segment #1, where we need to refer to these Labels, we declare them as **EXT** (**ext**ernal to the current segment), at the beginning of the source code. So we can use them anywhere in the source code of Segment #1, but always using **Long** addressing mode (the two segments may be located in different memory banks).

You can use EXTernal labels in expressions, but always using forward reference (EXT Label + Constant), never backward (EXT Label - Constant). You are not authorized to build expression involving several labels, where at least one is External (EXT Label - local Label + 2). You can use the Adressing Mode operators (`< > ^`) on them :

```
            LDAL    SHRLineTab+2,X

            PEA     <WaitForKey
            PEA     ^WaitForKey
```

Merlin 32 will assemble both segments separately and will search for **EXT**ernal labels during the linkage (creation of the multi-segments OMF file). If an EXTernal label can't be found in the other segments of the program, an error message will be displayed and the whole assembly process will fail. You won't get the program file created but you will get the output Text files (one per segment) created during the assembly step.

> Unsupported Merlin 16+ Commands

Even if we have tried to be as accurate as possible with **Merlin 16+** syntax, there are a few commands or directives not supported (=ignored) in **Merlin 32**. The first set of

commands which are not supported are the ones linked to the Merlin 16+ **editor**, the **interaction** during assembly or the **formatting** of the listing :

AST : send a line of ASTerisks
CYC : calcule and print CYCle times for the code
DAT : DATe stamp assembly listing
EXP : macro EXPand control
KBD : define label from KeyBoarD
LST : LiSTing control
LSTDO : LiSTDO OFF areas of code
PAG : new PAGe
PAU : PAUse
SW : SWeet 16 opcodes
TTL : define TiTLe heading
SKP : SKiP lines
TR : TRuncate control
EXD : define a label as Direct Page EXternal to the current REL Segment. You can use **EXT** instead of EXD.

The other thing we have decided not to support are the way the string may be delimited in **Merlin 16+**. In **Merlin 32**, the two different delimiters for a string are ' (simple quote = high bit clear) and " (double quotes = high bit set). In **Merlin 16+**, you can use virtually any character as delimiter. Here are few examples of valid *Hello World* strings in **Merlin 16+** :

- "Hello World"
- 'Hello World'
- #Hello World#
- @Hello World@
- !Hello World!
- (Hello World(
- ZHello WorldZ

...

Depending on the delimiters, the result string had the high bit clear (' ( ) + and ?) or set (" # @ ! ...). In order to simplify the reading of the source code, we have decided to support only **simple quote** and **double quotes** as valid strings delimiters.

The last part where **Merlin 32** is different from **Merlin 16+** is in the format of the intermediate object files. **Merlin 16+** assembles source code files (*.S) into object files (*.L) and link them to build the final program file. **Merlin 32** does everything in one operation (assemble + link), so there is no intermediate file available. **Merlin 32** can't use existing object files coming from **Merlin 16+**.You need to provide all the Source files to build a program file. For Multi-Segments OMF file, you will have to write a dedicated **Link** file. The one previously used with LINKER.XL in **Merlin 16+** can't be used with **Merlin 32**. Most of the Linker directives of **Merlin 16+** (LKV, VER, SAV, TYP, LIB, END, OVR...) are not supported by **Merlin 32** which uses its own syntax.

> F.A.Q

The same source files are assembled without any error with Merlin 16+ but raise errors with Merlin 32. Is Merlin 32 not supposed to be fully compatible with Merlin 16+ syntax ?

Merlin 32 syntax is strict and you can face situations where **Merlin 16+** lets you assemble invalid source files without displaying errors. For example, **Merlin 16+** truncates the Opcodes to 3 characters. So LDA**L**, LDA**d**, LDA**p** end up as LDA and **Merlin 16+** accept them. If you try to use invalid Opcodes such as LDAd with **Merlin 32**, you get immediately an error. You can easily fix such issues by using only valid Opcodes. Other problems can occur with local Labels starting with]. Forward references to Local Labels are not authorized. A local Label starting with] has to be defined before beeing used. But **Merlin 16+** won't complain if you make a forward branching to a local label starting with] **if the label is the only one** of the source file. **Merlin 32** is more strict and enforce the 'no forward reference' rule, so you get an error. You can fix this issue by replacing your local label by a global Label. The same source code may be assembled in adifferent ways by **Merlin 16+** and **Merlin 32** if **EQU** values are involved :

```
BIRD    EQU   #7

        LDA   BIRD
```

- **Merlin 16+** assembles the previous source code as : `LDA $7 ; Page Direct Address $07`
- **Merlin 32** assembles the previous source code as : `LDA #7 ; Constant`

Merlin 16+ evaluates the EQU very early in the assembling process and replace the value (BIRD) in the Operand with its value (7). The # is lost, so the LDA 7 is interpreted as a LDA $7 = Direct Page. **Merlin 32** evaluates the expressions at the end so the # is kept and the LDA becomes a LDA #7 = Constant value.

As we have seen with previous examples, there are some differences that may raise errors with **Merlin 32**, but with light modifications (LDA #BIRD instead of LDA BIRD), you can have a source code valid for both environments. Check also the unsupported commands list and think about the String delimiters which are more restrictive on **Merlin 32**. Always use the Output file to check the object code generated by **Merlin 32** from your source file.

Is the Source code of Merlin 32 available somewhere ?

The Source code is freely available in the Zip file (see download section).

It is currently packaged as a **Visual Studio 2010** Project set of files. The tool is only using **C Language**, so you can recompile it with any other C ANSI compiler (**gcc**...).

What about a Macintosh or Linux release ?

Everything has be done to make **Merlin 32** as independent as possible from the Operating System (command line utility, no UI). The source code is written in **C Ansi** and the only Operating Systems calls have been isolated in a specific file. The first release is available on **Windows** environment because it is the one used to create the software.

The **Macintosh** and **Linux** ports are available as Binary files (make sure to apply the **chmod 755** command to tun the file as executable). If the binary file is not working on your configuration, simply download **gcc** for Linux or Mac OS X and re-compile the project (`make -f linux_makefile`). The source files are available in the Zip file and it is the same for the 3 operating systems supported (Windows / Linux / Mac OS X). The current surce files are **Intel** only. The **PowerPC** support will be added soon.

> References

Merlin 16+ documentation by **Glen Bredon, Roger Wagner Publishing**
Apple IIgs GS/OS Reference, Appendix F : Object Module Format version 2.1
Programming the 65816 - Including the 6502, 65C02 and 65802 by **Western Design Center**
Le IIgs Epluché written by **D.BAR, D. DELAY, Y. DURANT, J.L SCHMITT** and **E. WEYLAND**
ORCA/M 2.0 documentation by **Mike Westerfield** and **Phil Montoya, Byte Works Inc**

> Download

Merlin 32 v1.0 for Windows 32 & 64 bits / Linux 64 bits / MacOS X 10.5 + Source Code