# PROFIMAT

## (USER MANUAL)

### PROFI-MON 64 v2.0

### PROFI-ASS 64 v2.0

***the monitor and the macro-assembler***
***for the COMMODORE 64***



## DATA BECKER

**(1981 – 2014)**

**(Rest in Peace †)**

**PROFIMAT** is a floppy disk programming system released in **1984** for creating and testing machine language programs for the **Commodore 64** conveniently from its own **BASIC** editor.

**PROFIMAT** consists of the '**PROFI-MON 64**' machine code monitor and the '**PROFI-ASS 64**' macro-assembler that can be loaded into memory separately or at the same time.

If you want to program in machine code **directly** from a real Commodore 64 (or from an emulator) without leaving the original system and without distractions, this is your opportunity.

On the following pages you will find a comprehensive description of both programs.

Downloadable **PROFIMAT** links:

PROFI-MON 64 V2.0 from csdb.dk (Recommended)

PROFI-ASS 64 V2.0 from csdb.dk (Recommended)

PROFIMAT V2.0 (1984) from planetemu.net (Original)

PROFIMAT V2.0 (1984) from archive.org (Original)

The_Machine_Language_Book_for_the_C64

The_Advanced_Machine_Language_Book_for_the_C64

# Table of Contents

## Part I. PROFI-MON 64 v2.0, THE MONITOR.


1. SUMMARY OF THE PROFI-MON 64 COMMANDS. 1
2. LOADING PROFI-MON 64. 2

MONITOR COMMAND DESCRIPTIONS :

- 3. R, Display the registers contents. 2
- 4. M, Display memory contents. 3
- 5. G, Execute program. 4
- 6. L, Load a machine language program. 4
- 7. S, Save a machine language program. 4
- 8. D, Disassemble a machine language program. 5
- 9. C, Compare memory areas. 5
- 10. T, Transfer memory area. 6
- 11. Searching memory areas. 6
  - 11.1. H, Search for byte combination. 6
  - 11.2. H, Search for text. 6
- 12. F, Fill memory range. 7
- 13. B, Switch memory configuration. 7
- 14. W, Single-step mode. 8
- 15. Q, Program execution with breakpoints. 8
- 16. U, Set a breakpoint. 9
- 17. X, Return to Commodore BASIC. 9
- 18. Error messages. 10


## Part II. PROFI-ASS v2.0, THE MACRO-ASSEMBLER.


1. USE AND TECHNICAL DETAILS OF PROFI-ASS 64. 11
2. EXPRESSIONS. 15
3. PSEUDO OPCODES. 18
   - 3.1. Symbol Value Assignment. 18
   - 3.2. Redefining Symbol Values. 18
   - 3.3. Program Counter Assignment. 19
4. LIST OF PSEUDO OPCODES.
   - 4.1. .BYTE 21
   - 4.2. .WORD 21
   - 4.3. .FILE 22
   - 4.4. .IF 22
   - 4.5. .GOTO 23

4.6. .GTB 24
4.7. .ASC 24
4.8. .SYS 24
4.9. .STM 25
4.10. .SST 25
4.11. .LST 26
4.12. .FLP 26
4.13. .END 27
4.14. .SYM 27
4.15. .PAG or .PAGE 28
4.16. .TIT or .TITLE 29
4.17. .OPT 29
4.17.1. P 29
4.17.2. P# 30
4.17.3. P=expression 30
4.17.4. O 30
4.17.5. OO 30
4.17.6. O# 31
4.17.7. O=expression 31
4.17.8. M 32
4.17.9. N 32
5. A SAMPLE PROGRAM. 33
6. INTRODUCTION TO MACROS. 35
6.1. A sample program with macros. 38
7. ERROR MESSAGES. 44
7.1. Error statistics. 45
7.2. Description of error messages. 45

8. APPENDIX A. WRITE OBJECT CODE TO THE DATASETTE. 48

9. APPENDIX B. ADVANCED SAMPLE, SPRITE MULTIPLEXING. 52

10. APPENDIX C. 6502/6510 INSTRUCTION SET. 56

11. END OF THE PROFIMAT 64 USER MANUAL. 57

12. SOME ANNOTATIONS ABOUT THE PROFIMAT 64 PACKAGE. 58

# PROFI-MON 64 v2.0
# Comfortable 6510 Monitor
# For the Commodore 64.



**PROFI-MON 64** is an extended machine language monitor that has features not found in more conventional software. It can be loaded concurrently with **PROFI-ASS 64** and thus forms a complete machine language development package.

## 1. Summary of the PROFI-MON 64 Commands :

| | |
|---|---|
| **R** - Register display. | Display register contents. |
| **M** - Memory display. | Display memory contents. |
| **G** - Go. | Execute machine language program. |
| **L** - Load. | Load machine language program. |
| **S** - Save. | Save machine language program. |
| **D** - Disassemble. | Disassemble machine language program. |
| **C** - Compare. | Compare memory areas. |
| **T** - Transfer. | Move memory area. |
| **H** - Hunt. | Search through memory range. |
| **F** - Fill. | Fill memory range with value. |
| **B** - Bank. | Select memory configuration. |
| **W** - Walk. | Single-step mode. |
| **Q** - Quicktrace. | Trace execution with break points. |
| **U** - Breakpoint. | Set breakpoint. |
| **X** - Exit to BASIC. | Return to BASIC. |

## 2. Loading PROFI-MON 64.

The monitor occupies **3K bytes** of memory, from **$C000** to **$CBFF** outside the **BASIC** area and is loaded from diskette. To do this, type :

```
LOAD "PROFI-MON V2.0",8,1
```

and press <RETURN>. The messages:

```
SEARCHING FOR PROFI-MON 64
LOADING

MONITOR 64 V2.0 IS LOADING…
```

appear on the screen. Once loaded, the monitor responds with :

```
*** PROFI-MON 64 V2.0 ***
(C) 1984 DATA BECKER GMBH

C*
```

and displays the **6510** registers contents.
All monitor input and output is done using 2 or 4 digit hexadecimal numbers. Below is a description of all the **PROFI-MON 64** commands.

## 3. Display the register contents: >R

The contents of the processor registers are displayed.

```
Register PC  -   Program Counter.
Register IRQ -   Interrupt vector.
Register SR  -   Status Register.
Register AC  -   Accumulator.
Register XR  -   X Register.
Register YR  -   Y Register.
Register SP  -   Stack Pointer.
```

In addition, the flags of the status register are displayed individually:

| | | |
|---|---|---|
| **N** | - | Negative flag. |
| **V** | - | Overflow flag. |
| **-** | - | Not used. |
| **B** | - | Break flag. |
| **D** | - | Decimal flag. |
| **I** | - | Interrupt flag. |
| **Z** | - | Zero flag. |
| **C** | - | Carry flag. |

Example : **>R** <RETURN>

```
    PC   IRQ   SR  AC   XR   YR   SP   NV-BDIZC
>; 0003  EA31  32  34   02   A2   FB   00110010
```

If you want to change the register contents, you simply move the cursor to the appropriate place, overwrite the old contents with the new value and press <RETURN>. The new register contents are placed into the register. If the contents of the **status register** are changed, the flags are also changed and displayed.

## 4. Display memory contents: >M XXXX YYYY

The contents of memory starting at **XXXX** and ending at **YYYY** is displayed. Both **XXXX** and **YYYY** are four digit hexadecimal numbers. If the ending address **YYYY** is omitted, only one line is displayed. The **ASCII** representation of the memory contents is displayed in 'Reverse On ' characters mode following the hexadecimal representation. Unprintable control characters are displayed as a period. As example, the command **>M A0A0 A0AF** displays :

```
                                    Lower case PETSCII :
>: A0A0 C4 46 4F D2 4E 45 58 D4  DfoRnexT
>: A0A8 44 41 54 C1 49 4E 50 55  datAinpu
```

Memory contents can be changed in the same way as register contents command **R**, by overwriting the byte value and pressing <RETURN>.

## 5. Execute program: >G XXXX

The **Go** command executes a jump to address **XXXX** and executes the machine language program found there. If **XXXX** is not entered, the value of the **program counter** (**PC**) is used as the starting value.

If the machine language program encounters the command **BRK ($00)**, control returns to the **monitor** which displays ***B (Break)** and displays the register contents. The **program counter** points to the address after the **BRK** command. When testing programs, we recommend that you terminate them with **BRK ($00)**.

## 6. Load a machine language program: >L "NAME", XX, YYYY

The program "**NAME**" is loaded beginning at address **YYYY** from device **XX**. Normally **YYYY** is omitted; the program then loads at the address from which it was saved. If the device address is also omitted, device **08** is assumed.

Example:
```
>L “PROG”,08 <RETURN>
SEARCHING FOR PROG
LOADING
>
```

If you want to load from cassette, enter the value **01** for **XX**.

## 7. Save machine language program: >S "NAME", XX, YYYY, ZZZZ

**XX** is again the device number, **YYYY** is the starting address, and **ZZZZ** is the ending address plus one of the program to be saved.

Example:
```
>S “PROG”,01,C900,C9DE <RETURN>
```

The program “**PROG**”, is saved onto cassette from address **$C900** to **$C9DF**.

## 8. Disassemble a machine language program: >D XXXX YYYY

The machine language program beginning at address **XXXX** through **YYYY** will be displayed in **mnemonic form** (as '**operation code**' or '**opcode**'). If the ending address **YYYY** is omitted, only one line is displayed. Three question marks '**???**' will be displayed for invalid instructions.

```
                Program in memory          Mnemonics/Opcodes
Example:        >D B016 B021 <RETURN>
                >, B016  20 90 AD          JSR  $AD90
                >, B019  B0 13             BCS  $B02E
                >, B01B  A5 6E             LDA  $6E
                >, B01D  09 7F             ORA  #$7F
                >, B01F  25 6A             AND  $6A
                >, B021  85 6A             STA  $6A
```

If the displayed addresses are in **RAM**, then you can change the bytes following the address. Type in your change and press <RETURN>, to make the change. The instruction is re-disassembled. On the next line, the following address is automatically displayed and the cursor is placed over the first byte of the instruction, so that the next instruction can be changed. This mode can be exited by erasing the character after the address before pressing <RETURN>.

## 9. Compare memory areas: >C XXXX YYYY ZZZZ

The memory area from addresses **XXXX** through **YYYY** is compared with the area starting at **ZZZZ** byte by byte. Any address whose contents differ are displayed.

```
Example:        >C 8000 8100 9000 <RETURN>
                 8056
                >
```

The contents of address **$8056** differ from the contents of address **$9056**.

## 10. Transfer memory area: >T XXXX YYYY ZZZZ

The memory area from addresses **XXXX** through **YYYY** are moved to the memory area beginning at **ZZZZ**.

Example: **>T 6000 6FFF 3000**

The memory range from **$6000** through **$6FFF** is transferred to **$3000** to **$3FFF**. The contents of the original range remains unchanged.

**11. Searching memory areas.** There are two options when searching: search for a byte combination or search for **ASCII** text.

## 11.1 Search for byte combination: >H XXXX YYYY BB BB BB.

The memory range from addresses **XXXX** through **YYYY** is searched for the byte combination {**BB**}. The combination can be up to **29** bytes long.

Example: **>H E000 EFFF 20 D2 FF** <RETURN>
**E10C**

The memory area from addresses **XXXX** through **YYYY** is searched for the combination **$20 $D2 $FF** (which corresponds to the **subroutine call**). Addresses at which this combination is located are displayed, **$E10C** in this example.

## 11.2 Search for text: >H XXXX YYYY "TEXT"

The memory area from address **XXXX** to **YYYY** will be searched for the **ASCII** text "**TEXT**". The text can be up to **29** characters long. The starting memory address from which the searched text is located is displayed.

Example: **>H A000 AFFF "READY"** <RETURN>
**A378**

## 12. Fill memory range: >F XXXX YYYY ZZ

The area from addresses **XXXX** through **YYYY** are filled with the byte **ZZ**.

Example: **>F 8000 8FFF 00** <RETURN>

## 13. Switch memory configuration: >BX

With this command you can have access to the entire memory of the Commodore 64. After starting the monitor, all commands operate on the normal memory configuration. With '**BA** <RETURN>' you can switch the memory configuration to all **RAM**, while '**BC** <RETURN>' also adds the character generator. You can switch back to the normal **ROM** configuration with '**BR** <RETURN>'. This configuration effects only the commands :

**M**, **D**, **C**, **T**, **H** and **F**.

The following table illustrates the three configurations :

| Address range | >BR | >BA | >BC |
|---|---|---|---|
| $E000 - $FFFF | ROM | RAM | RAM |
| $D000 - $DFFF | I/O | RAM | ROM CHARACTERS |
| $C000 - $CFFF | RAM | RAM | RAM |
| $A000 - $BFFF | ROM | RAM | RAM |
| $0000 - $9FFF | RAM | RAM | RAM |

## 14. Single-step mode: >W XXXX

One special feature of **PROFI-MON 64** is the single-step (walk) mode. With this you can execute machine language programs instruction by instruction. The command has the same syntax as the **G** command, either starting at address **XXXX** or at the address contained in the **program counter** if only a **W** is given. When you enter **W**, the command at that address is executed and the contents of the registers and flags are displayed in the same format as with the **R** command. Displayed on the next line is the following instruction in disassembled form. If you press a key, the next command is executed and the resulting register contents are again displayed. You can exit the single-step mode with the **<RUN/STOP>** key.

Example:
```
>W BC16 <RETURN>
>; BC11 EA31 20 3A 00 38 F8  00100000
>, BC11   B5 60      LDA $60,X
```

The single-step mode works with all "normal" programs. It should not be used with programs that use the **I/O KERNAL** functions.

## 15. Program execution with breakpoints: >Q XXXX

The single-step mode often takes too long when working with machine language programs. Therefore **PROFI-MON 64** offers you the option of controlling machine language programs by setting breakpoints.

You can specify that a machine language program is to be interrupted when it reaches a certain place. Should the program never reach the breakpoint, it can be stopped by pressing the **<RUN/STOP>** key. The breakpoints are set with the **U** command, described shortly. The syntax of the **Q** command is the same as for the **G** and **W** commands.

## 16. Set a breakpoint: >U XXXX YYYY

If you want to use the **Q** command, you must first set a breakpoint. The **U** command performs this function. **XXXX** is the address at which the program is to stopped. If you start your program with the **Q** command, it will stop executing at the address given by **XXXX**. You are then placed in the single-step mode (**W**). With **<RUN/STOP>** you can halt or single-step a program. The **U** command offers the additional option of stopping the program after it reaches the given breakpoint a certain number of times. The **YYYY** parameter specifies the number of times the breakpoint is ignored before execution is halted.

**>U 1000 0050** <RETURN>

Here the program is interrupted when it passes address **$1000** for the **80**th time (hexadecimal 50). Values up to **$FFFF** = **65535** are allowed.

## 17. Return to Commodore BASIC: >X

The command :

>**X** <RETURN>

returns you to Commodore **BASIC** in the same state as before the call. After exiting the **monitor** with the **X** command you can enter '**SYS 2** <RETURN>' or a **SYS** to any location containing a **zero** (as long as the **<RUN/STOP><RESTORE>** key has not been pressed) in order to return to **PROFI-MON 64.**

Otherwise use :

**SYS 12*4096** <RETURN>

to return to the monitor.

## 18. Error messages.

If you have made an syntax error in your input, **PROFI-MON 64** will echo the input along with a question mark '**?**'. You can then correct the input.

In addition to these syntactical errors, the error routines of the C64 **KERNAL** are activated through **PROFI-MON 64.** If an error occurs when saving or loading, for example, an error message of the following form appears :

**I/O ERROR #X**

in which **X** can be a number from **1** to **9** and has the following significance :

**1 . . . too many files**

**2 . . . file open**

**3 . . . file not open**

**4 . . . file not found**

**5 . . . device not present**

**6 . . . not input file**

**7 . . . not output file**

**8 . . . missing filename**

**9 . . . illegal device number**

PROFI-ASS 64 2.0
Comfortable 6510 Macro-Assembler
For the Commodore 64.



**PROFI-ASS 64** is a **two-pass 6510** or **6502** macro-assembler for the the **Commodore 64.** It is written entirely in machine language and occupies **8K bytes** of **RAM. PROFI-ASS 64** allows free-form input using the built-in Commodore **BASIC** editor, produces complete **assembly listings**, loadable **symbol tables**, various options for storing created object codes, **re-definable symbols**, and a comprehensive set of **pseudo-opcodes** (assembler directives) for such things as creating **macros** or **conditional assembly**. The syntax for the most part adheres to the **MOS** standard.

## 1. USE AND TECHNICAL DETAILS OF PROFI-ASS 64.

**PROFI-ASS 64** is loaded from floppy disk and requires the higher **8K bytes** of the **BASIC RAM** (addresses **$8000-$9FFF**). The area most frequently used for machine language programs from **$C000** to **$CFFF** is left free and can be used to load the monitor **PROFI-MON 64** (addresses **$C000-$CBFF**) or for your own machine language programs.

### Load of PROFI-ASS 64.

Insert the **PROFI-ASS 64** diskette and type:

**LOAD "PROFI ASS PLUS",8,1** <RETURN>

The following will appear on the screen:

```
LOADING
PROFI-ASS 64 V2.0 IS LOADING...

*** PROFI-ASS 64 V2.0 ***
(C) 1984 DATA BECKER GMBH
2
 0000-0000
NO ERRORS
READY.
```

When loading, **PROFI-ASS 64** protects itself from being overwritten by **BASIC**. A **BASIC** or **assembler source program** that is eventually in memory is preserved. You are left **30717** bytes for your assembly language source programs.

The **2** in the **PROFI-ASS 64** message indicates the start of **pass 2.** Following is the address range of the generated object code, **0000-0000** in this case, and the number of errors detected.

The assembler programs are entered using line numbers just like **BASIC** programs. Lines can be changed, deleted, or inserted exactly as in **BASIC**. No other editor is necessary and more storage space is available for your source programs, a total of **30K bytes**. You can separate several assembler commands on the same line using colons "**:**" as in **BASIC**.

You can make your assembly language programs easier to read by placing an up arrow (↑) as the first character of a line. After this, all spaces are accepted and the arrow is ignored by **PROFI-ASS 64**. This allows you to indent your programs as desired.

**PROFI-ASS 64** uses almost the same source format as the **MOS** standard. If even you are familiar with this standard, you should read this user manual because it also explains the departures from the **MOS** standard. The examples illustrate the instructions.

This manual is not intended to teach **6510** assembly language programming. You should already be familiar with the syntax of the **6510** assembly code. For this we recommend the books: "The_Machine_Language_Book_for_the_C64" and "The_Advanced_Machine_Language_Book_for_the_C64" by Lothar Englisch and from this same publisher. The first book is an introduction to the topic and the second book deepens in the use of macros, floating-point arithmetic and the, always surprising, programming with interrupts.

Lines of PROFI-ASS 64 source code consist of the following fields : **line numbers**, **symbols**, **opcode or pseudo-opcode commands**, **operands**, **labels** and **comments.**

The **pseudo-opcodes** are not machine language instructions but rather tell the assembler to do special things; these **assembler directives** are described later in the manual.

Each program line contains a mnemonic (a **opcode**) or **pseudo-opcode** and may begin with a **LABEL** (a symbol). If a line is supposed to contain a label, simply place it in front of the instruction, followed by one or more spaces. A label must begin with a letter followed by other letters, numbers or periods. The first 8 characters of a label must be unique, that is, no labels may have the same first 8 characters. Non-alphanumeric characters are not allowed.

Instruction mnemonics may follow a label or may begin at the start of a line if no label is present. All mnemonics consist of 3 letters. Mnemonics are reserved words and may not be used as labels.

If an instruction begins with a period "**.**", it is treated as a **pseudo-opcode.** There are three pseudo-opcodes which do not begin with a period, but start with special characters. All pseudo-opcodes must be separated from their operands by spaces, with the exception of '**=**' and '***=**'. Pseudo-opcodes which begin with a period are distinguished by the first three characters only, although they will be printed in full in the assembly listing.

A line can be terminated by a semicolon "**;**". Everything following the semicolon is ignored by the assembler and can contain comments. The comments are included and printed on the assembler listing, but the comments are ignored by the assembler. A colon "**:**" within a comment ends it and begins a new instruction, as long as the colon is not placed within quotation marks. If a line begins with a semicolon, the assembler treats the entire line as a comment. Such lines are printed without a line number.

The **operand** field contains the addressing mode and an expression for the command or pseudo-opcode. A semicolon may follow.

The addressing modes with expressions have the following syntax :

| | |
|---|---|
| **#expression** | Immediate addressing. |
| **expression** | Absolute or relative addressing. |
| **expression,X** | Absolute,X  indexed by X. |
| **expression,Y** | Absolute,Y  indexed by Y. |
| **(expression,X)** | Indexed indirect addressing. |
| **(expression),Y** | Indirect indexed addressing. |
| **(expression)** | Indirect addressing. |

**PROFI-ASS 64** automatically converts **absolute addressing** to **zero-page addressing** if the expression has a value less than **256**. If you want to force **absolute addressing**, you can place an exclamation point in front of the expression. **LDA !5,X** generates the code **BD 05 00**, the absolute form of **LDA**, while **LDA 5,X** yields the **zero-page** addressing **B5 05**. This is useful if you want to avoid the **wrap-around effect** of indexed addressing with addresses under **256**.

## 2. EXPRESSIONS.

**PROFI-ASS 64** is unique among assembler in its ability to calculate complex **expressions**. The assembler has a recursive routine for calculating nested expressions, which gives you more capabilities than other assemblers. A **PROFI-ASS 64** valid expression can be located anywhere described in this manual where the word "**expression**" appears. Such an **expression** is also allowed for the pseudo-opcodes which expect a numerical argument. The expression evaluation of **PROFI-ASS 64** is so efficient that your programs can be written entirely using symbols. This makes changing and transporting **PROFI-ASS 64** programs especially simple and easy to understand.

The syntax of expressions is very simple and is a superset of the **MOS** standard. Expressions are entered exactly as they would be on a pocket calculator which does not use an algebraic evaluation system but does allow parentheses. All operators are evaluated strictly from left to right, although square brackets are allowed as well as parentheses in order to alter the order of evaluation.

An **expression** can be terminated by a variety of characters. The end of a line always ends an **expression**. Colons '**:**', semicolons '**;**', and commas '**,**' also ends an **expression**, provided that these are not enclosed in quotation marks. A closing parenthesis '**)**' ends an **expression** provided no unpaired open parenthesis '**(**' remain. This makes nested expressions possible with indexed addressing.

You can use the following **operators** in expressions :

| | |
|---|---|
| **+** | Add values. |
| **-** | Subtract right value from left value. |
| ***** | Multiply values. |
| **/** | Divide left value by right value. |
| **!** | Logical **OR** of two values. |
| **&** | Logical **AND** of two values. |
| **↑** | Logical **XOR** (exclusive **OR**) of two values. |

**>** Shift left argument as many bits to the right as the right argument specifies.
**<** Shift left argument as many bits to the left as the right argument specifies.

All operations are performed using **16 bit** arithmetic, although various operations will lead to overflows, such as multiplication by a value greater than **32767**, or shifting left more than **15** bits. These cause an '**ILLEGAL QUANTITY ERROR**'. This error message also appears for a **division by zero.** For addition and subtraction, a result greater than **65535** is interpreted as a negative number in **two's complement** form.

The operands themselves can appear in a variety of forms. In the following, the syntax is given together with an example.

**Types of operands:**

| Type | Example | Syntax |
|---|---|---|
| Hexadecimal | **$1C3** | **$**{hexadecimal digit} |
| Decimal | **127** | {digit} |
| Binary | **%110011** | **%**{**0** or **1**} |
| PC | ***** | |
| ASCII character | **"A"** | "character" |
| Label | **SYMB** | alphabetic{alphanumeric} |
| Expression | **("Z"+6)** | {expression} |

Under 'Syntax' items placed within **{}** may be repeated as often as necessary.

Each of the above terms can be combined with the previously-described **operators**. These can be enclosed in parentheses as desired in order to alter the order of evaluation. A minus sign can be placed in front of every **operand**, including parenthesized **expressions**, to yield a two's complement value.

An entire **expression** can be changed by a single modifying character. One example is the use of "**!**" to select an absolute addressing mode.

In addition, the "greater than" and "less than" signs are allowed. "**>**" in front of **expressions** tells the assembler to take only the most significant byte of the **expression**'s result (first 8 bits of the 16 bit expression), while "**<**" denotes the least significant byte. This is necessary for direct addressing or with the **.BYTE** pseudo-opcode.

The most significant byte operator (**>**) performs the same operation as :

**expression > 8**

The least significant operator can also be represented as :

**expression & $FF**

Sample expressions :

**>LABEL-1+(TABLE*2)**

**VALUE-***

**"O"-"A" < 3 + ("D" - "A" > 2&%111)**

Parentheses may be nested as deep as necessary. **Modifiers** cannot be used on parenthesized parts of expressions.

## 3. PSEUDO-OPCODES.

Most **PROFI-ASS 64** pseudo-opcodes begin with a period ("**.**"). All of these "**period**" opcodes must be separated from following character by at least one space. In addition, there are three special pseudo-opcodes which are defined by special characters. Pseudo-opcodes are recognized only by their first three letters; everything else up to the next space will be ignored, although it will be printed in the listing. The three special pseudo-opcodes serve to define **symbols** and the **program counter**.

### 3.1. Symbol Value Assignment.

The simplest of these is the operator for **symbol definition**, the equal sign "**=**". In order to assign a value (expression) to a symbol, you simply write :

```
SYMBOL = EXPRESSION
```

The assignment is made only during pass 1 of the assembly. Any subsequent definition of this same symbol in the source program results in a '**REDEFINITION ERROR**'. The "**=**" sign is used to define constants and addresses in symbolic form, so that only one line need be changed to alter all occurrences of the value. Here's a few examples (note that spaces between "**=**" sign are required) :

```
BEGIN = $C000   ;Define start of program.
TAPEBUFFER = 828   ;Define tape buffer at $33C
```

### 3.2. Redefining Symbol Values.

Similar to the operator for symbol definition is the **assignment operator**, which is written as a left arrow "**←**", and is used with the same syntax :

```
SYMBOL ← EXPRESSION
```

By contrast to the previous operator, it is possible to redefine a symbol. In this case, the assignment is made during pass 2 as well as pass 1. This can be used for various purposes, most often during conditional assembly (see **.GOTO**). Here are some examples:

```
SYMBOL  ← VALUE
NUMBER  ← NUMBER -1 ; Decrement value
PROGRAM ← *
```

## 3.3. Program Counter Assignment.

The third special pseudo-opcode controls the **program counter**. It is written as asterisk equal "***=**" which means "**assigns a value to the program counter**". The primary use of this symbol is to specify the starting address of the program. If not specified, it defaults to **$C000**.

Storage for data may also be reserved. The statement ***=*+32**, for example, defines a 32-byte block beginning at the current **program counter** location. The value of the program counter is then incremented by 32. If a **symbol** is found in the LABEL field, the value of the **program counter** is assigned to it before the **program counter** is incremented. Here's an example that defines variables in **Page 2** of the RAM memory :

```
*= $200 ; Sets the program counter to the start of Page 2.
ADDRESS *= *+1  ; A one-byte address, set to zero.
TABLE *= *+32   ; Table begins at $201.
LABEL *= *+1    ; LABEL has the value $233.
TWO *= *+2      ; Two-byte pointer.
TEST *= $800    ; TEST has the value $236;
                ;    following code begins at $800.
```

To define a table within a program, the following can be used :

```
          . . .
          LDA #5
          RTS
TABLE     *=   *+256 ; 256-byte table
TEST      LDA  #>ADDRESS*3
          . . .
```

In general, you can use "***=**" to define symbols by altering the program counter. You should not, however, move it backwards. This is allowed only : **1)** if you assemble object code directly into memory and execute it there; or **2)** when you do not create object code at all. When you assemble code at **$1000**, for example, you cannot normally set the program counter back to **$0F00** to assemble code there. This is allowed for LABEL definition, but you must then return to an address which was higher than the address into which the last byte of object code was assembled.

That is why it is usual to use the pseudo-opcodes of **symbol** and **assignment** definition, and jump with the **program counter**. In the assignment with "***=**", a LABEL is always assigned the value prior to the incremented of the **program counter**. If a LABEL is found in an assembler line, the listing always shows the value of the LABEL instead of the **program counter**, so that if in doubt, you can take it directly from the listing.

## 4. LIST OF PSEUDO OPCODES.

### .BYTE expression

The .BYTE pseudo-opcode is used to place one-byte values into the **object code** at the location contained in the **program counter**. Any legal **PROFI-ASS 64** expressions, separated by commas, may be used as operands. The number of operands is only limited by line length and the length of the **PROFI-ASS 64**'s buffer. Any expression with the desired operators may be used; but the expression must evaluate to a one-byte value, or an '**ILLEGAL QUANTITY ERROR**' occurs. Two-byte values can be modified with "**>**" and "**<**" in order to take the high or low byte, respectively. A one-byte value lies in the range **0** to **255** or **$FF80** to **$FFFF**. The higher values are allowed because they normally signify negative numbers from **-1** to **-128**. Therefore the line "**.BYTE -1**" is allowed. **.BYTE** can be used to define tables such as **jump tables** or **pointers**. With **.BYTE** is also possible to invoke some "ILLEGAL" **6510** hidden commands, for example the **BIT** command; a well-known programming trick :

```
            .BYTE $2C   ; Absolute BIT instruction
LABEL1      LDA #-1     ; Hidden LDA 6510 instruction.
```

### .WORD expression

The **.WORD** pseudo-opcode is used in order to place two-byte addresses into the **object code** at the location contained in the **program counter**. For example the following statements :

```
START =     $C000
      .WORD START
```

would assemble the bytes **00 C0** as the value of the symbol **START** into the **object code**. The address is stored with the least significant byte **00** first followed by the most significant byte **C0**.

The statement : .WORD address

is equivalent to the statement :

.BYTE <address,>address

The .WORD and the .BYTE pseudo-opcodes permit multiple values on a line, separated by semicolons. The .WORD pseudo-opcode is most often used for creating address tables.

## .FILE

The .FILE pseudo-opcode is used to chain several source programs. The syntax is as follows :

.FILE device_number, "filename"

where 'device number' is '8' for the disk drive or '1' for the datasette, and 'filename' is the name of the assembly language source program which is to be loaded next. If you are writing a very long assembly language program, yo can break it up into several parts an chain these together in the assembly with .FILE. The last file in this chain must contain an .END pseudo-opcode that specifies the first file of the chain :

.END device_number,"Name_of_the_last_program".

See for this the .END section and the examples.

## .IF expression

The .IF pseudo-opcode is used for conditional assembly. The syntax is as follows :

.IF expression : .GOTO line_number

The argument **expression** is evaluated in both pass 1 and pass 2. If the **expression** is not zero, the code following the **.IF** in the same line is performed. Usually, this will be a **.GOTO** to direct the assembly to a different line. The additional code in the line must be separate by colons "**:**".

With **.IF**, **.GOTO**, and **symbol re-definitions** (with the help of the **assignment operator** "**←**"), it is possible to create assembler loops. Although **.IF** only tests for zero, other comparisons are possible by using simple techniques. For example, shifting **15** bits to the right yields a result of **1** if the expression was negative, and **0** if positive. Two numbers may be compared by subtracting one from the other and testing the result for positive or negative.

## .GOTO line-number

The **.GOTO** pseudo-opcode instructs the assembler to continue assembly at the **line-number** given as the argument.

This line number may also be an expression. The line number must be contained in the currently loaded program (if you are using **.FILE** to chain multiple source programs). You cannot jump between different files. This line number may be located either before or after the line number containing the **.GOTO** pseudo-opcode. When used with **.IF** and **re-defining symbols**, it's possible to build a loop for conditional assembly.

The following **conditional assembly** example creates a loop by **re-defining symbols**. You can see the assignment operator '**←**' in action here :

```
10 SYS 32768 : REM CALL THE ASSEMBLER
20 .OPT P ; LISTING TO SCREEN
30 OFFSET ← 5 ; NUMBER OF LOOPS
40 LDA $C000 + OFFSET
50 OFFSET ← OFFSET -1 ; DECREMENT
60 .IF OFFSET : .GOTO 40
70 .END
```

## .GTB

This pseudo-opcode stands for '**Go To BASIC**'. It has no argument and simply returns control to Commodore **BASIC**. The **BASIC** commands in following program lines will be executed. You may return to the assembler by using **SYS 40954** <RETURN>. The address **40954** is **5 bytes** before of the last direction **$9FFF** of the macro-assembler, that is **$9FFA** (decimal **40954**).

You should note that the **BASIC** commands that can be executed before return to assembler are limited. Some **BASIC** statements may overwrite the work areas used by **PROFI-ASS 64** and should not be executed. In particular, the **INPUT** command, or any other **BASIC** commands which writes to byte **9** (that is, exceeding byte **8**) of the **BASIC** input buffer (address **$0209**) must be avoided. The **GET** statement is allowed. You should never return control to the user during assembly.

## .ASC "text"

This pseudo-opcode places the **ASCII** value(s) for the "**text**" into the **object code** at the location contained in the **program counter**. The text is enclosed in quotation marks. It is thereby possible to insert cursor or color control characters into the text. The text can be up to **55** characters long. Longer texts must be divided up into several **.ASC** statements. The **MOS** standard uses the **.BYTE** pseudo-opcode for this purpose, in which strings are enclosed in apostrophes ('). You should take this into account when converting programs. Note the use of double quotes here instead of single quotes.

## .SYS expression

This pseudo-opcode allows machine language programs to be called during assembly. The value of **expression** determines the jump address. This pseudo-opcode is identical to the **SYS** command in **BASIC**.

The routine located at the address specified by **expression** is called during both pass 1 and pass 2. The **SYS** command can be used by those familiar with the internal workings of **PROFI-ASS 64** to create custom pseudo-opcodes.

## .STM expression

This pseudo-opcode is used to raise the lower boundary of the **symbol table**. The **symbol table** grows downward from the end of the storage (**$8000**), exactly as strings are saved in **BASIC**. At the start of assembly, this lower boundary is set to the end of the **BASIC** program and variables. You can set it higher if you are working with **.FILE** or buffered object code (**.OPT O**). If the space for the **symbol table** is too small, the message "**SYM TABLE OWERFLOW**" is given and the assembly stopped.

## .SST device number, secondary address, "filename"

**Symbol tables** may be saved to storage **IEC BUS** devices such as the floppy disk, and from there loaded in again. **.SST** is executed in pass 1 only, and saves the **symbol table** that has been generated up to that point.

The first argument is the **IEC** device number, normally **8** for the disk drive. The secondary address can lie between **2** and **14**. The "**filename**" is given as in an **OPEN** command, and therefore requires an "**,S,W**" following the '**filename**' (for **S**equential and **W**rite).

This pseudo-opcode is required if you want to later print a sorted list of **symbols** and **LABELS**. The program **SYMPRINT** (see page 51 for more details about the SYMPRINT program) then uses this file to list the symbols to your printer.

The **.SST** command is also useful when assembling source programs separately, but which must access subroutines from the other programs.

Simply save the **symbol table** at the end of first assembler program and read this same **symbol table** into the second program using **.LST**.

## .LST device number, secondary address, "filename"

This pseudo-opcode loads the **symbol table** that was saved by the **.SST** pseudo-opcode. You can use **.LST** to load a **symbol table** created by other programs, such as a table of the C64 **KERNAL** routines. Duplicate symbols are not checked.

The last definition of a duplicate symbol is used and previous definitions are simply ignored. Overflow of the **symbol table** is not recognized when loading, although an error will occur as soon as you try to define another symbol.

## .FLP expression

If you often use the **floating-point arithmetic** of the **BASIC** interpreter, you can use **.FLP** to place floating-point constants into the object code. This simplifies the use of floating-point routines. One or more floating-point constants separated by commas can follow the **.FLP** command, for example :

```
.FLP 10, 1E8
```

Each floating-point number occupies **5 bytes**; therefore our example generates **10 bytes**. Note that only the first **3 bytes** of the converted number are printed in the object code listing.

## .END [device, "filename"]

This pseudo-opcode ends a source program and is optional. **.END** executes a **.GTB** at the end of pass 2. If there are additional **BASIC** statements following the **.END** pseudo-opcode, they will be executed. You can, for example, call the machine language program just assembled with a **SYS** statement, preferably without errors.

When chaining source programs, **.END** must have the additional arguments. The arguments are in the same format as the **.FILE** pseudo-opcode and direct the assembler to re-load the first source program at the end of pass 1 and continue with pass 2 at the line containing the **SYS 32768**. "**filename**" must therefore be the name of the first program in the chain (which contains the **SYS 32768**). "**filename**" has no further effect in pass 2.

## .SYM

This pseudo-opcode can be used to list a table of all the defined symbols and their values after the assembly of the program. This list is sent to the screen, the printer or other device according to the **output option** (**.OPT, see section**). Four symbols, together with their values in hexadecimal form, are printed per line. If you want a different number of symbols per line, you can use this number as an argument for the **.SYM** command. **.SYM** is useful when working on the screen, for example. You can use **.SYM 2** if you work on the screen. The **symbols** are listed in the reverse order from that in which they were defined. If you want an alphabetically sorted list, you must save the **symbol table** with **.SST** and use the program "**SYMPRINT**" found on your **PROFIMAT 64** distribution floppy disk.

If you have worked with macros, the names of the different macros will also be output. In addition, it appears next to each macro, the number of times was called in the program as a two-position hexadecimal number.

## .PAG page-lenght, left-margin offset

This pseudo-opcode has three different functions and serves to control the assembly language listing. Without additional parameters, it forces a form feed in the listing. This allows you to place a certain section of an assembler listing on a new page. **PROFI-ASS 64** automatically inserts a form feed after every **60**$^{th}$ line, and begins the next page with a title and the current page number. If you want to change the page length, you can set the number of lines per page with the **.PAG** (or **.PAGE**) command, for example :

```
.PAG 64
```

This instructs **PROFI-ASS 64** to write **64** lines on each page. Values up to **255** are accepted. An additional function is the determination of the left margin. This is very useful for printed listings which you want to put in a notebook. The second parameter of **.PAG** gives the number of spaces to be printer in front of each assembler line in the listing. The standard value is **zero**. With :

```
.PAG ,10
```

the listing can be indented **10** characters to the right. The comma is necessary in order to denote the **10** as the second parameter. The two parameters can also be combined :

```
.PAG 64,10
```

## .TIT “text”

The pseudo-opcode .TIT (or .TITLE) allows you to add text to the standard title :

```
PROFI-ASS 64 V2.0     PAGE 1
```

which appears on every page of the listing. This text is given after the .TIT command within quotation marks, such as :

```
.TIT   “HARDCOPY ROUTINE”
```

This text will then be placed before the standard title, and we get :

```
HARDCOPY ROUTINE    PROFI-ASS 64 V2.0    PAGE1
```

## .OPT options{,options}

The .OPT pseudo-opcode stands for OPTion and gives you control over the assembly listing and the object code. The syntax is the following :

```
.OPT option, option, option. . .
```

The following options are available :

### P :

Print. You select this option when you want the assembly listing to appear on the screen. All other P options (see below) also output to the screen because the screen is the fastest output medium. The listing will be formatted automatically. Lines which contain errors or a .FILE command will be printed in passes 1 and 2 regardless of the P option.

**P# :**

**Print to file.** With this option, you can send a listing to the printer, for example. In order to do so, you must first open a logical file before the **SYS 32768** command with an **OPEN** command, such as "**OPEN 1,4**". The logical file number (**1** in our example) then replaces the number sign (**#**), such as **.OPT P1**. Using this technique, you can also write the assembly listing to disk or cassette with the appropriate **OPEN** command. You can specify that a line feed **(CHR$(10))** be sent after each carriage return **(CHR$(13))** when selecting the logical file number in **BASIC**. This accomplished by using a logical file number greater than **127**, such as **OPEN 130,4** and then **.OPT P130**.

**P=expression :**

With this option you can direct the output to a routine of your own. The **start address** of your routine must be given as the **expression**. The character to be outputted is passed in the **accumulator**. A zero indicates the last character (close file). This allows custom output devices to be used (such as an interface on the C64 **user port**).

**O :**

**Object** means "**object code output**". Without additional characters, the **object code** goes to a special **buffer** located immediately after the assembler program, where the **array variables** normally lie; the same pointers are also used.

**OO :**

**Object at Origin**. This option writes the **object code** directly to the memory locations for which it was written.

This is very useful for quickly testing programs, and allows maximum freedom when moving the **program pointer**. Saving code to tape (cassette) is also made possible using the **monitor**. If an assembly language program is intended to run in the memory range where the source program or assembler lies, this method may naturally not be used.

### O# :

As with **P#**, this allows output of the **object code** to a file through the **IEC** bus. The file must be previously opened as a program file for writing (secondary address **1**), such as **OPEN 1,8,1,"program"**. With **.OPT O1**, the **object code** goes to this file. First **PROFI-ASS 64** writes the start address to the file, and then the generated code. If the assembler operation ends normally, the program file will be closed again. The machine language program created in this manner can be loaded directly with **LOAD** or with a **monitor**. Note that **.OPT O#** to a cassette is not possible. See the next option and the 'Appendix A'.

### O=expression :

This allows the **object code** to be sent to a user-defined routine with the same syntax as the **.OPT P=** command. The **object code** output routine must be somewhat more complicated because it is called only once per assembler line. Some **symbols** which are required are found in the 'Appendix A'. The most important is **LENGTH**, which gives the number of bytes to be output minus 1. If **LENGTH** is **zero**, for example, one byte must be output. Your routine must be test for two special values. A value of **$80** means that the "**output file must be open**", in this moment you can transfer the start address. A value of **$C0** means "**close the file**". Otherwise, **LENGTH** contains a small number from **zero** on up. The data to be output are stored in two places. The first **three bytes** are stored in the **zero-page** at address **OP**.

If more than three bytes of object code are created (for **.BYTE**, **.WORD**, **.ASC**, for example), the additional bytes are stored at address **OBJBUF**. Your output routine may change any **registers** or **flags** (with the exception of the **decimal flag**). Caution is advised in using the **zero-page** however.

A program is listed in the 'Appendix A' which makes it possible to output the object code to a file in hex format. It is therefore possible in principle to save data directly to the **datasette**.

**M :**

If you work with macros, you can decide whether you want the entire macro containing the actual parameters to be listed for each macro call, or just the line containing the macro call. If you do not enter this command, the complete macro will be listed. You can suppress this with **.OPT M** and cause only the line with the macro call to be listed.

**N :**

You can cancel the output options at any time with "**.OPT N**". **N** cancels all of the options except the **M** option. If an option is supposed to remain in effect or switched on again later, just add that option again when you need it. If, for example, you want to turn off the screen listing, but still want the object code to go to **file 2**, you would write :

```
.OPT N,O2
```

and

```
.OPT P
```

when the listing is to go to the screen again.

## 5. A SAMPLE PROGRAM.

The following example program writes the contents of the **zero-page** at line **LINE** on the screen. It illustrates the general use of the assembler.

To avoid syntax errors **respect all blank spaces that appear in the listing**, especially between assignments with the '**=**' symbol. Make sure that the same line does **not contain, by mistake, two consecutive lines,** it could lead to hidden variables not declared or other types of errors; use "**LIST line_number** <RETURN>" to check this type of errors.

To save your source code, use: **SAVE "program_name",8** and to load it again: **LOAD "program_name",8**, as usual with **BASIC**.

```
10 SYS 32768 ; CALL TO PROFI-ASS 64
20 .OPT P,OO ; OBJECT CODE TO MEMORY
30 *= $C000  ; START ADDRESS PROGRAM
40 LINE = 10 ; LINE ON SCREEN
50 VIDEO = $400  ; SCREEN MEMORY ADDRESS
60 COLOR = $D800 ; COLOR MEMORY ADDRESS
70 INK   = 1     ; WHITE COLOR CHARACTERS
80 LDX #0        ; ZERO INDEX REGISTER
90 BUCLE LDA 0,X ; GET BYTE FROM ZERO-PAGE
100 STA VIDEO+(40*LINE),X ; OUTPUT CHARACTER
110 LDA #INK
120 STA COLOR+(40*LINE),X ; SET COLOR
130 INX                   ; NEXT BYTE
140 BNE BUCLE
150 RTS                   ; DONE
160 .END
```

To assemble this source program, type "**RUN** <RETURN>", and the following listing will be generated on screen :

```
PROFI-ASS 64 V2.0           PAGE 1

20:       C000                .OPT P,OO   ; OBJECT CODE TO MEMORY
30:       C000                *=  $C000   ; START ADDRESS PROGRAM
40:       000A         LINE   =   10      ; LINE ON SCREEN
```

```
50:       0400          VIDEO   =   $400   ; SCREEN MEMORY ADDRESS
60:       D800          COLOR   =   $D800  ; COLOR MEMORY ADDRESS
70:       0001          INK     =   1      ; WHITE COLOR CHARACTERS
80:       C000 A2 00            LDX #0     ; ZERO INDEX REGISTER
90:       C002 B5 00    BUCLE   LDA 0,X    ; GET BYTE FROM ZERO-PAGE
100:      C004 9D 90 05         STA VIDEO+(40*LINE),X  ; OUTPUT CHARACTER
110:      C007 A9 01            LDA #INK
120:      C009 9D 90 D9         STA COLOR+(40*LINE),X  ; SET COLOR
130:      C00C E8               INX                    ; NEXT BYTE
140:      C00D D0 F3            BNE BUCLE
150:      C00F 60               RTS                    ; DONE
>C000-C010
NO ERRORS
```

In the following example, the object code is sent directly to disk and the listing is sent to printer. The source program consists of several individual programs.

PROGRAM 1 contains :

```
10 OPEN 1,8,1,  "0:OBJECT CODE"
20 OPEN 2,4 : REM PRINTER
30 SYS 32768
40 .OPT O1,P2
50 ; ASSEMBLER COMMANDS
. . .
1000 .FILE 8, "PROGRAM 2"
```

PROGRAM 2 contains :

```
10 ; ADDITIONAL COMMANDS
. . .
1000 .FILE 8, "PROGRAM 3"
```

PROGRAM 3 contains :

```
10 ; ADDITIONAL COMMANDS
. . .
1000 .END 8, "PROGRAM 1"
```

whereby "**PROGRAM 1**" is the program which contains the **SYS 32768** command.

## 6. INTRODUCTION TO MACROS.

After making contact with most assembler commands, we now come to a powerful feature of **PROFI-ASS 64**, the **macros.** What are the macros and what are they used for? With macros we have the ability to combine a series of instructions and assembler directives **and give them a name.** If you have defined a macro in this manner, you can later insert this set of instructions into the source code as often as desired by simply using the name of the macro. An example will make this clear.

In machine language programs, one repetitive task often comes up in programming - namely incrementing the contents of a **16-bit variable** located in two consecutive **zero-page** locations. The instruction to do this might look like this :

```
            INC POINTER
            BNE LABEL
            INC POINTER+1
LABEL       ...
```

At another place you might have to increment a different variable called, for example, **TEMP** :

```
            INC TEMP
            BNE LABEL1
            INC TEMP+1
LABEL1      ...
```

With macros we can define a set of instructions once and use this definition later. To define a macro, two new pseudo-opcodes are used. The first declares the macro definition, and the second ends it. In order to able to refer to a macro later, it must have a name. The same conventions apply as for the other symbols (first character must be a letter, then letters, digits or periods and eight significant places).

Our particular macro definitions will look like this:

```
INC.PNT  .MAC ADDRESS
         INC  ADDRESS
         BNE  .LABEL
         INC  ADDRESS+1
LABEL    .MEND
```

The name of this macro is **INC.PNT**. A macro definition is introduced with the pseudo-opcode **.MAC**. Parameters may follow. Here we have a parameter called **ADDRESS**. Next the executable instructions follow in their standard form. One especial feature is found in the line "**BNE .LABEL**". Every time we refer to a **LABEL** that is defined inside the macro, we must put a period '**.**' to the name. The last line contains the **LABEL definition** and the end of the **macro definition** with **.MEND** (or **.MEN**). Now we can make a call to the newly-defined macro with :

```
'INC.PNT POINTER
```

This line replaces the above set of instructions. We write an apostrophe "**'**" followed by the macro name and any parameters. In our case there was one parameter, although a macro can have no parameters, or several parameters separated by commas. When assembled, the macro is replaced by the instructions :

```
         INC POINTER
         BNE LABEL:00
         INC POINTER+1
LABEL:00
```

The next example illustrates a macro without parameters :

```
RAM .MAC
    SEI
    LDA $01
    AND #%11111110
    STA $01
    .MEND
```

This macro requires no parameters and no so-called local **LABELS** (labels within the macro definition). Macros without parameters generate the same code each time and can in principle be replaced by subroutines.

Briefly, something fundamental regarding the difference between macros and subroutines. Macros are auxiliary tools during the assembly and each call generate object code. The subprograms or subroutines, on the other hand, can be considered how aid during run-time execution and are found only once in the object program.

Macros are specially useful in combination with **conditional assembly**. If you have macros ready for a variety of fundamental tasks, the main program can consist in the basic structure and a set of macro calls.

A few notes about the correct use of macros :

Macros must be defined at the start of the assembly language source, before they are called. If you are chaining source programs using **.FILE**, all macros must be contained in the first program. If you define labels within a macro, a period '**.**' must be placed before references to the label, as illustrated earlier. This also applies within **expressions** (check the sample expressions on page 17). Such labels are only significant to six characters. If you call such macros several times and output the **symbol table**, the labels are listed as many times, together with different values. In order to distinguish these from each other, the name is followed by a colon and the number of the label, for example :

```
LABEL:00    0006    LABEL:01    C020    LABEL:02    C035
```

The number zero indicates the **LABEL** value within the definition, relative to the start of the macro.

If labels are defined with a macro, different names must be used within different macros, or a 'REDEFINITION ERROR' will occur. Parameters may have the same names because these are replaced by the actual values during a macro call anyway.

Arbitrary PROFI-ASS 64 expressions can be used in a macro call; these are calculated by the assembler and transmitted as parameters, for example :

```
'INC.PNT   POINTER-8*2
```

Here, for example, the value of POINTER is taken and the result of 8 times 2 is subtracted from it. The order of evaluation can be determined through the use of parentheses as usual.

## 6.1. A sample program with macros.

As an example, we have a program which consists almost entirely of macro calls. Two macros are defined. The first serves to set the cursor position. The operating system of the Commodore 64 places this routine at our disposal. The macro with the name CURSOR expects two parameters. The first is the line in which the cursor is to be placed, and the second is the column. If we want to set the cursor at a specific place in our program, we need only call the macro, for example :

```
'CURSOR 10,20
```

The second macro serves to output text. The parameter is the address location of the text. The string must be terminated by a zero byte.

In the program you find first the definition of the two macros and then the actual program which consists only of four macro calls and a RTS command. The strings are listed at the end of the program.

Here the source program, followed by the assembly listing :

```
100 SYS 32768
110 .OPT P,OO    ; The parameter is two letters, not zeros.
120 ; PROGRAM DEMOSTRATION FOR MACROS
130 ;
140 ; CURSOR POSITION
150 CURSOR .MAC LINE,COLUMN
160 LDX #LINE
170 LDY #COLUMN
180 STX $D6
190 STY $D3
200 JSR SETCRSR ; SET CURSOR POSITION
210 .MEN
220 ;
230 ; CHARACTER OUTPUT
240 PRTSTR .MAC TEXT
250 LDA #<TEXT
260 LDY #>TEXT
270 JSR STROUT ; TEXT OUTPUT
280 .MEN
290 ;
300 SETCRSR = $E56C
310 STROUT = $AB1E
320 ;
330 *= $C000
340 ;
350 'CURSOR 10,10
360 'PRTSTR TEXT1
370 'CURSOR 0,20
380 'PRTSTR TEXT2
390 RTS
400 ;
410 TEXT1 .ASC "TEXT N. 1" : .BYT 0
420 TEXT2 .ASC "TEXT N. 2" : .BYT 0
430 ;
440 .END
```

Assembly listing after the "**RUN** <RETURN>" command:

```
PROFI-ASS 64 V2.0     PAGE 1
110:   C000           .OPT P,OO ; The parameter is two letters, not zeros.
120:                  ; PROGRAM DEMOSTRATION FOR MACROS
130:                  ;
140:                  ; CURSOR POSITION
150:   '              CURSOR .MAC LINE,COLUMN
160:   '                     LDX #LINE
170:   '                     LDY #COLUMN
180:   '                     STX $D6
190:   '                     STY $D3
200:   '                     JSR SETCRSR ; SET CURSOR POSITION
210:   '                     .MEN
220:                  ;
230:   '              ; SALIDA CADENA
240:   '              PRTSTR .MAC TEXT
250:   '                     LDA #<TEXT
260:   '                     LDY #>TEXT
270:   '                     JSR STROUT ; TEXT OUTPUT
280:   '                     .MEN
290:   '              ;
300:   E565           SETCRSR = $E56C
310:   AB1E           STROUT  = $AB1E
320:                  ;
330:   C000                  *= $C000
340:                  ;
350:   C000                  'CURSOR 10,10
+      C000 A2 0A             LDX #LINE
+      C002 A0 0A             LDY #COLUMN
+      C004 86 D6             STX $D6
+      C006 84 D3             STY $D3
+      C008 20 6C E5          JSR SETCRSR ; SET CURSOR POSITION
+      C00B                   .MEN
360:   C00B                  'PRTSTR TEXT1
+      C00B A9 25             LDA #<TEXT
+      C00D A0 C0             LDY #>TEXT
+      C00F 20 1E AB          JSR STROUT ; TEXT OUTPUT
+      C012                   .MEN
370:   C012                  .CURSOR 0,20
```

```
+       C012 A2 00              LDX #LINE
+       C014 A0 14              LDY #COLUMN
+       C016 86 D6              STX $D6
+       C018 84 D3              STY $D3
+       C01A 20 6C E5           JSR SETCRSR ; SET CURSOR POSITION
+       C01D                    .MEN
380:    C01D                    'PRTSTR TEXT2
+       C01D A9 30              LDA #<TEXT
+       C01F A0 C0              LDY #>TEXT
+       C021 20 1E AB           JSR STROUT ; TEXT OUTPUT
+       C024                    .MEN
390:    C024 60                 RTS
400:                    ;
410:    C025 54 45 58   TEXT1    .ASC "TEXT N. 1"
410:    C02F 00                 .BYT 0
420:    C030 54 45 58   TEXT2    .ASC "TEXT N. 2"
420:    C03A 00                 .BYT 0
430:                    ;
>C000-C03B
NO ERRORS
```

Let's take a closer look at the listing. You recognize that within the macro definition, an apostrophe (') appears instead of the program counter. The object code field is empty because no code is created by the macro definition (lines **150-210**, **240-280**).

The first macro call is in line **350**. The actual **program counter** as well as the code created appear in the listing. A plus sigh (**+**) appears in place of the line number, which shows that the created code comes from a macro call. You recognize that the symbols **LINE** and **COLUMN** have the values which they were assigned by the macro call. The subsequent macro calls proceed in the same manner.

If you have many macros in your source program or you call certain macros often, you have the option of suppressing the macro-created code in the assembly listing. Only the line containing the actual macro call will appear. The option **.OPT M** performs this function. See the next example :

```
PROFI-ASS 64 V2.0     PAGE 1

110:   C000                    .OPT P,M,OO
120:                  ; DEMOSTRATION PROGRAM FOR MACROS
130:                  ;
140:                  ; CURSOR POSITION
150:   ‘              CURSOR .MAC LINE,COLUMN
160:   ‘                     LDX #LINE
170:   ‘                     LDY #COLUMN
180:   ‘                     STX $D6
190:   ‘                     STY $D3
200:   ‘                     JSR SETCRSR ; SET CURSOR POSITION
210:   ‘                     .MEN
220:   ‘              ;
230:   ‘              ; STRING OUTPUT
240:   ‘              PRTSTR .MAC TEXT
250:   ‘                     LDA #<TEXT
260:   ‘                     LDY #>TEXT
270:   ‘                     JSR STROUT ; TEXT OUTPUT
280:   ‘                     .MEN
290:   ‘              ;
300:   E565           SETCRSR = $E56C
310:   AB1E           STROUT  = $AB1E
320:                  ;
330:   C000                  *= $C000
340:                  ;
350:   C000                  ‘CURSOR 10,10
360:   C00B                  ‘PRTSTR TEXT1
370:   C012                  ‘CURSOR 0,20
380:   C01D                  ‘PRTSTR TEXT2
390:   C024 60               RTS
400:                  ;
410:   C025 54 45 58 TEXT1    .ASC “TEXT N. 1”
410:   C02F 00               .BYT 0
420:   C030 54 45 58 TEXT2    .ASC “TEXT N. 2”
420:   C03A 00               .BYT 0
430:                  ;
>C000-C03B
NO ERRORS
```

Suppressing the macros makes the listing shorter and often easier to read. In the next example we have added a print with the **.SYM** pseudo-opcode, a list of the symbols and their values together with the defined macros as well as the number of macro calls that have been made as a two-digit hexadecimal number. The first part of the listing is the same as the previous page. In this example listing we only pay attention to the **symbol table** and **macro table** :

```
PROFI-ASS 64 V2.0     PAGE 1

110:   C000                    .OPT P,M,OO
115:   C000                    .SYM
120:                   ; DEMOSTRATION PROGRAM FOR MACROS
. . . ; (the listing is the same as the previous page)
430:                   ;
>C000-C03B
NO ERRORS
-------------------------------------------------------------
PROFI-ASS 64 V2.0     PAGE 2

SYMBOLTABLE:
TEXT2 C030    TEXT1  C025   TEXT    C030   COLUMN 0014
LINE  0000    STROUT AB1E   SETCRSR E56C

7 SYMBOLS DEFINED

MACROTABLE:
PRTSTR    02   CURSOR    02

2 MACROS DEFINED.
```

To execute the program (once assembled without errors with the "**RUN** <RETURN>" command) simply execute the command "**SYS 49152** <RETURN>"; where the decimal address **49152** corresponds to the start address of the **$C000** declared in the source code. The two **text lines** will appear on screen in the screen locations declared in the source code. :)

## 7. ERROR MESSAGES.

**PROFI-ASS 64** has a set of error messages. Errors are printed in both pass 1 and pass 2. If the assembler recognizes an error, four asterisks "********" are displayed followed by the error message. The line containing the error will then be displayed on the screen, regardless of the **.OPT P** settings. For a **syntax error**, a digit will also be displayed in front of the four asterisks which describes the error in greater detail. There are **10** different types of **syntax errors** which can occur. They are listed below. Still other errors can occur when using macros; these are indicated by a prefixed letter.

Some errors are "**FATAL**", meaning that they cause the assembly to stop. An exclamation point (**!**) is displayed in front of lines containing fatal errors. The assembly is stopped after the message is displayed. The first byte of the object code created for such a line is a zero, which is the **6502 BRK** command. If you try to execute such a program, a **BRK** command is executed when it comes to the erroneous line, which either performs a warm start, or returns you the **monitor**, if it is loaded. In general, you should first correct the errors before you execute an assembly language program.

One type of error which **PROFI-ASS 64** cannot detect is the so-called **phase error**. This error does not usually occur, but can be encountered with certain combinations of conditional assembly containing **.BYTE** or **.WORD** pseudo-opcodes. A **phase error** occurs when the **program counter** is different in pass 2 than it was in pass 1. From there a **useless code** arises. However, you can recognize a **phase error** with an **.IF** instruction :

```
PHASE .IF PHASE-* : PHASE ERROR
```

Normally, **PHASE** has the same value as the **program counter** and the code behind the colon '**:**' is never executed.

If a phase shift occurs, the result is not zero and the additional statement results in a syntax error which you can recognize as a confirmed **phase error.**

## 7.1. Error statistics.

Before the start of pass 2, **PROFI-ASS 64** outputs the number of errors in pass 1, if any were found. For example :

**2 ERRORS IN PASS 1**

After pass 2, when the assembly is complete, the number of errors in pass 2 is displayed. If the assembly was error-free, the message :

**NO ERRORS**

is displayed. If errors were encountered, that number of errors is displayed. For example :

**4 ERRORS**

## 7.2 Description of error messages.

**SYNTAX : This error message is preceded by a digit which describes the error in greater detail. These digits have the following meaning :**

**0 - LABEL :** for empty assignment not allowed (the line contains only one string).
**1 - Invalid opcode.**
**2 - Invalid addressing mode :** (this command may not be used with this addressing mode).
**3 - Unknown operator in expression :** (unallowed character in an expression).
**4 - Unpaired parentheses :** (a parenthesis was opened but it was not closed).

5 - **Invalid expression :** (invalid character in an expression, or an empty string " ").
6 - **Comma expected :** (a pseudo-opcode is expecting a comma).
7 - **Invalid pseudo-opcode :** (The **.XXX** string was not recognized as pseudo-opcode).
8 - **Symbol does not start with a letter :** (A symbol was expected, but an alphabetic character was not found).
9 - **Opcode with unallowed addressing mode.**

The following syntax errors can occur for macros :

B - **.MEN command without previous .MAC.**
C - **Unclosed macro definition.**
D - **Nested macro definition :** Macros within macros are not allowed.
F - **Erroneous number of parameters :** The number of parameters in the macro call does not match the number of parameters in the macro definition.

**ILLEGAL QUANTITY :** The expression evaluated to a value which lies outside the borders for this command or pseudo-opcode. The expression yields a value greater than **65535**.

**OVERFLOW :** The input buffer which **PROFI-ASS 64** uses in order to decode source lines is to small. Divide the line into several instructions or use a temporary variable in order to simplify the expression.

**BRANCH OUT OF RANGE :** A relative jump (**branch** command) over a distance greater than **128** bytes was attempted.

**REDEFINITION :** An attempt was made to define a symbol twice without using the redefinition operator.

**UNDEF'D STATEMENT :** A LABEL or expression is not defined.

**REVERSAL** : An attempt was made to assemble code at an address which is lower than the last address. This error does not occur when you assemble directly to memory. This is a "**Fatal Error**", as are all of the following errors.

**SYM TABLE OVERFLOW** : You have tried to define more symbols than space in the **symbol table** permits. Either set the minimum lower with **.STM** or divide your program into several parts. This error message can also appear when loading a source program with **.FILE** if the program is too large and part of the symbol table has been overwritten. Divide the program into smaller parts.

**OUT OF MEMORY** : The buffer for the object code (**.OPT O** mode) is too small. You should choose some other type of output, such as disk.

**UNDEF'D STATEMENT** : A **.GOTO** to a non-existent line in the program (just like in **BASIC**). In contrast to the error named before, this one is fatal and leads to a system fall.

**DEVICE NOT PRESENT** : The addressed **IEC** device is not present on the bus, or does not answer.

**IEEE** : Another error occurs on the IEEE bus.

**DISK** : Disk error. The disk drive error message was given just prior to this.

## 8. APPENDIX A. WRITE OBJECT CODE TO THE DATASETTE.

The following source program is another example of the use of **PROFI-ASS 64**. It demonstrates outputting of object code by a user-defined routine. It sends each byte in hex format to a previously opened file with the logical file number **1**. It is therefore possible to write the object code directly to the **datasette**, for example. It is possible to read code in this format with the **BASIC** program following it.

```
100 SYS 32768      ; CALL TO THE ASSEMBLER
110 .OPT P,OO
120 LENGHT  = $4E   ; OUTPUT BYTE LESS 1
130 OP      = $4B   ; BUFFER FOR THE 3 FIRST BYTES
140 ADR     = $56   ; START OF THE ADDRESS PROGRAM
150 OBJBUF  = $15B  ; BUFFER FOR THE REMAINING BYTES
160 CHKOUT  = $FFC9 ; OUTPUT TO LOGIC FILE
170 CLRCH   = $FFCC ; DEFAULT OUTPUT
180 PRINT   = $FFD2 ; ONE CHARACTER OUTPUT
190 CLOSE   = $FFC3
200 LF      = 1             ; LOGIC FILE NUMBER
210 *= $C000                ; START ADDRESS
220 LDA LENGHT
230 CMP #$C0                ; TO CLOSE
240 BEQ CLOSEF
250 LDX #LF : JSR CHKOUT    ; OUTPUT TO LOGIC FILE 1
260 LDX #0 : LDA LENGHT
270 CMP #$80                ; TO OPEN
280 BEQ STARTADR
290 OUT LDA OP,X
300 OUT1 JSR WROB           ; OUTPUT BYTE AS HEXADECIMAL
310 CPX LENGHT
320 BEQ EX1
330 INX
340 CPX #3
350 BCC OUT
360 LDA OBJBUF-3,X
370 JMP OUT1
380 EX1 JMP CLRCH
390 CLOSEF LDA #LF
```

```
400 JMP CLOSE
410 STARTADR LDA ADR : JSR WROB ; INITIAL ADDRESS LOW
420 LDA ADR+1        : JSR WROB ; INITIAL ADDRESS HIGH
430 JMP CLRCH
440 WROB PHA         ; OUTPUT BYTE AS HEXADECIMAL NUMBER
450 LSR : LSR : LSR : LSR       ; UPPER NIBBLE
460 JSR ASCII
470 PLA
480 AND #%1111                  ; LOWER NIBBLE
490 ASCII CLC
500 ADC #-10
510 BCC ASC1
520 ADC #6
530 ASC1 ADC #"9"+1
540 JMP PRINT
550 .END
```

If you assemble this program with "**RUN** <RETURN>", you get the following assembler listing :

```
PROFI-ASS 64 V2.0      PAGE 1

110:   C000                        .OPT P,OO
120:   004E            LENGHT   =      $4E    ; OUTPUT BYTE LESS 1
130:   004B            OP       =      $4B    ; BUFFER FOR THE 3 FIRST BYTES
140:   0056            ADR      =      $56    ; PROGRAM START ADDRESS
150:   015B            OBJBUF   =      $15B   ; BUFFER FOR THE REMAINDER BYTES
160:   FFC9            CHKOUT   =     $FFC9   ; OUTPUT TO LOGIC FILE
170:   FFCC            CLRCH    =     $FFCC   ; DEFAULT OUTPUT
180:   FFD2            PRINT    =     $FFD2   ; ONE CHARACTER OUTPUT
190:   FFC3            CLOSE    =     $FFC3
200:   0001            LF       =     1       ; LOGIC FILE NUMBER
210:   C000                     *=    $C000   ; START ADDRESS
220:   C000 A5 4E               LDA LENGHT
230:   C002 C9 C0               CMP #$C0       ; TO CLOSE
240:   C004 F0 24               BEQ CLOSEF
250:   C006 A2  01              LDX #LF
250:   C008 20 C9 FF            JSR  CHKOUT    ; OUTPUT TO LOGIC FILE 1
260:   C00B A2 00               LDX #0
260:   C00D A5 4E               LDA LENGHT
270:   C00F C9 80               CMP #$80       ; TO OPEN
280:   C011 F0 1C               BEQ STARTADR
290:   C013 B5 4B      OUT      LDA OP,X
300:   C015 20 3C C0   OUT1     JSR WROB       ; OUTPUT BYTE AS HEXADECIMAL
310:   C018 E4 4E               CPX LENGHT
320:   C01A F0 0B               BEQ EX1
330:   C01C E8                  INX
```

```
340:   C01D E0 03              CPX #3
350:   C01F 90 F2              BCC OUT
360:   C021 BD 58 01           LDA OBJBUF-3,X
370:   C024 4C 15 C0           JMP OUT1
380:   C027 4C CC FF    EX1    JMP CLRCH
390:   C02A A9 01       CLOSEF LDA #LF
400:   C02C 4C C3 FF           JMP CLOSE
410:   C02F A5 56       STARTADR LDA ADR
410:   C031 20 3C C0           JSR WROB       ; INITIAL ADDRESS LOW
420:   C034 A5 57              LDA ADR+1
420:   C036 20 3C C0           JSR WROB       ; INITIAL ADDRESS HIGH
430:   C039 4C CC FF           JMP CLRCH
440:   C03C 48          WROB   PHA            ; OUTPUT BYTE AS HEXADECIMAL
450:   C03D 4A                 LSR
450:   C03E 4A                 LSR
450:   C03F 4A                 LSR
450:   C040 4A                 LSR            ; UPPER NIBBLE
460:   C041 20 47 C0           JSR ASCII
470:   C044 68                 PLA
480:   C045 29 0F              AND #%1111     ; LOWER NIBBLE
490:   C047 18          ASCII  CLC
500:   C048 69 F6              ADC #-10
510:   C04A 90 02              BCC ASC1
520:   C04C 69 06              ADC #6
530:   C04E 69 3A       ASC1   ADC #"9"+1
540:   C050 4C D2 FF           JPM PRINT
>C000-C053
NO ERRORS
```

If you assemble this program, you can write the **object code** in hexadecimal format to the **datasette** with this format :

```
100 OPEN 1,1,1,"OBJECT CODE" : REM WRITE TO TAPE
110 SYS 32768
120 .OPT P,O=$C000 ; OBJECT CODE TO CUSTOM ROUTINE
. . .
```

The program can be loaded from tape with a small loader program in **BASIC** :

```
100 OPEN 1,1,0,"OBJECT CODE" : REM READ FROM TAPE
110 GOSUB 1000 : AD = A : REM LOW BYTE OF START ADDRESS
120 GOSUB 1000 : REM HIGH BYTE OF START ADDRESS
130 AD = A*256 + AD : REM START ADDRESS
140 IF ST=64 THEN CLOSE 1 : END : REM PROGRAM END
150 GOSUB 1000 : REM READ BYTE
```

```
160 POKE AD,A : AD = AD + 1
170 GOTO 140
1000 REM READ HEX NUMBER
1010 GET #1, A$,B$
1020 H = ASC(A$)-48+(A$>="A")*7 : REM HIGH NIBBLE
1030 L = ASC(B$)-48+(B$>="A")*7 : REM LOW  NIBBLE
1040 A = L+16*H : RETURN
```

Your **PROFIMAT** distribution diskette contains a **BASIC** program called "**SYMPRINT**". This program serves to output a **symbol table** in alphabetical order, which you have written to disk previously with **.SST**.

The program asks for the name of the **symbol table** on disk as well as the number of output device (**3** = screen, **4** = printer, **8** = floppy disk). For disk output, you must give the name of the file to which the **symbol table** will be written.

Finally, you can determine how many symbols will be printed per line. **Two** fit per line on the screen, **4** on a printer. The output format corresponds to that of the **.SYM** command when assembling.

## 9. APPENDIX B. ADVANCED EXAMPLE, SPRITE MULTIPLEXING.

The following assembler source code is taken from pages 106, 107, 108 and 109 of the recommended book: "The_Advanced_Machine_Language_Book_for_the_C64" that you can download with a simple click on the title.

If you copy the assembly listing directly from the book and compile it with the "**RUN**" command, **PROFI-ASS 64 v2.0** will respond with an almost infinite list of syntax errors.

This source code was probably intended for an earlier version of **PROFI-ASS**. Below is the listing of the assembler source code with small modifications that solve all the problems.

You will notice that the listing is written in lowercase letters and without spaces between the line numbers and the code. This way can you mark the entire listing with the mouse to copy and paste it directly into the **VICE C64 Emulator**. Be careful not to also copy the number of each page in this manual when you are marking the assembler code listing. You can copy and paste it in various parts in the VICE emulator. Press the <RETURN> key between each paste to enter the last line. In 'VICE v3.4 GTK3' you can paste text from the '**EDIT**' menu. If you want to type the code by hand, we recommend that you write it all in capital letters on the C64. The code contains simple comments, for more details on how this interrupt-based program works we recommend the previous book.

**These are the steps you need to follow :**

1. Insert the **PROFI-ASS 64 V2.0** floppy disk and Type:

   **LOAD "PROFI ASS PLUS",8,1** <RETURN>

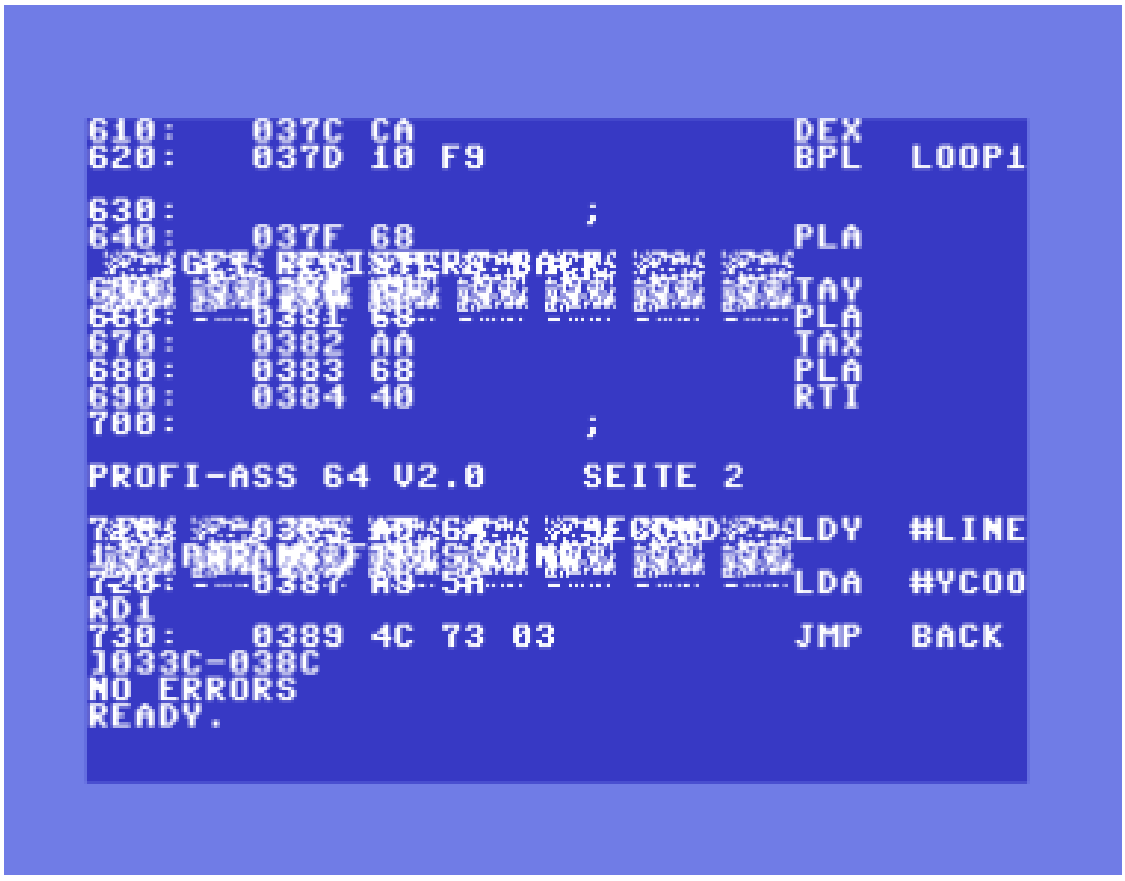
   Once loaded, type : **RUN** <RETURN>

2. Insert your working floppy disk and LOAD or SAVE your source assembly code into memory by typing :

   **LOAD "YOUR-ASSEMBLER-SOURCE-CODE-PROGRAM-NAME",8**
   **SAVE "YOUR-ASSEMBLER-SOURCE-CODE-PROGRAM-NAME",8**

3. Once you have loaded **PROFI-ASS 64** assembler, and loaded (or typed) the source code, write and execute the "**RUN** <RETURN>" command.

4. As a result you will get **16 multiplexed software sprites** on your C64 screen. The **BASIC** program lines **810-900** at the end of the listing initializes the 8 hardware sprites of the C64 and invoke the previous assembler routine that activates the sprite multiplexing on screen. **:)**




On the next two pages you will find the complete assembly language source code along with the final small portion of the code written in **Commodore BASIC**.

Here is the entire language assembler listing :

```
10sys 32768 ;rem call to profi-ass 64 macro-assembler
100.opt p,oo ;rem object code output to the c64 memory
110;
120;raster interrupt
130;
140vic = $d000 ;video controller
160spritey = vic+1 ;sprite y-coordinate
165raster = vic+18 ;raster line
170irr = vic+25 ;interrupt request register
180imr = vic+26 ;interrupt mask register
190line1 = 100 ;first line
200line2 = 200 ;second line
202ycoord1 = 90 ;firts y-coordinate
203ycoord2 = 170 ;second y-coordinate
210;
220irqvec = $314
230irqold = $ea31
240;
300*= 828
310init sei
320lda #line1 ;first interrupt
330sta raster ;at line 100
340lda raster-1
350and #%01111111 ;erase high bit
360sta raster-1
370lda #%10000001 ;interrupt by
380sta imr ;raster line
390lda #<testirq
400ldy #>testirq
410sta irqvec ;vector to new
420sty irqvec+1 ;routine
430cli
440rts
460testirq lda irr ;read register
470sta irr ;and erase
480and #%1 ;irq by raster line
490bne ok ;yes
500jmp irqold ;normal irq
```

```
510;
520ok lda raster ;current line
530cmp #line2 ;>= second lineprint
540bcs second ;yes
545;
550ldy #line2 ;next irq at 2nd line
555lda #ycoord2 ;new sprite coordinate
560back sty raster ;set raster line
570ldx #14
590loop1 sta spritey,x ;sprite coordinates
600dex ;change
610dex
620bpl loop1
630;
640pla ;get registers back
650tay
660pla
670tax
680pla
690rti
700;
710second ldy #line1 ;params firts line
720lda #ycoord1
730jmp back
800.end :rem go to basic to execute basic code
810for i=0 to 7:poke 2040+i,12:next
820v=53248
830poke v+21,255
840for i=0 to 7:pokev+2*i,(i+1)*30
850poke v+2*i+1,70:next
860for i=0 to 7:pokev+39+i,1:next
900sys 828 :rem call to the sprite multiplexer routine
```

## 10. APPENDIX C. 6502/6510 INSTRUCTION SET.

| | |
|---|---|
| ADC | Add Memory to Accumulator whit Carry |
| AND | "AND" Memory with Accumulator |
| ASL | Shift left One Bit (Memory or Accumulator) |
| BCC | Branch on Carry Clear |
| BCS | Branch on Carry Set |
| BEQ | Branch on Result Zero |
| BIT | Test Bits in Memory with Accumulator |
| BMI | Branch on Result Minus |
| BNE | Branch on Result not Zero |
| BPL | Branch on Result Plus |
| BRK | Force Break |
| BVC | Branch on Overflow Clear |
| BVS | Branch on Overflow Set |
| CLC | Clear Carry Flag |
| CLD | Clear Decimal Mode |
| CLI | Clear Interrupt Disable Bit |
| CLV | Clear Overflow Flag |
| CMP | Compare Memory and Accumulator |
| CPX | Compare Memory and Index X |
| CPY | Compare Memory and Index Y |
| DEC | Decrement Memory by One |
| DEX | Decrement Index X by One |
| DEY | Decrement Index Y by One |
| EOR | "Exclusive-OR" Memory with Accumulator |
| INC | Increment Memory by One |
| INX | Increment Index X by One |
| INY | Increment Index Y by One |
| JMP | Jump to New Location |
| JSR | Jump to New Location Saving Return Address |
| LDA | Load Accumulator with Memory |
| LDX | Load Index X with Memory |

| | |
|---|---|
| LDY | Load Index Y with Memory |
| LSR | Shift One Bit Right (Memory or Accumulator) |
| NOP | No Operation |
| ORA | "OR" Memory with Accumulator |
| PHA | Push Accumulator on Stack |
| PHP | Push Processor Status on Stack |
| PLA | Pull Accumulator from Stack |
| PLP | Pull Processor Status from Stack |
| ROL | Rotate One Bit Left (Memory or Accumulator) |
| RTI | Return from Interrupt |
| RTS | Return from Subroutine |
| SBC | Subtract Memory from Accumulator with Borrow |
| SEC | Set Carry Flag |
| SED | Set Decimal Mode |
| SEI | Set Interrupt Disable Status |
| STA | Store Accumulator in Memory |
| STX | Store Index X in Memory |
| STY | Store Index Y in Memory |
| TAX | Transfer Accumulator to Index X |
| TAY | Transfer Accumulator to Index Y |
| TSX | Transfer Stack Pointer to Index X |
| TXA | Transfer Index X to Accumulator |
| TXS | Transfer Index X to Stack Pointer |
| TYA | Transfer Index Y to Accumulator |

## 11. End of the PROFIMAT 64 user manual :

And here is the end of the user manual of the PROFIMAT 64 Monitor & Macro-Assembler software package. We hope that with it you can develop your own software tools, games and applications in the exciting world of assembly language and machine code along with the unique and personal hardware features that your Commodore 64 offers.

## 12. SOME ANNOTATIONS ABOUT THE PROFIMAT 64 PACKAGE.

This user manual has been completely reviewed in 2021. Several times. Certain syntax errors and misprints have been corrected and the manual has been expanded with two short but useful appendices.

The original floppy disk of **PROFIMAT** includes an anti-copy **DRM** protection system that hinders both execution and its own existence. In fact in the available images of the original **PROFIMAT** floppy disk in **.d64** format nor the **PROFI-ASS** assembler nor the **PROFI-MON** monitor run correctly after its load in memory.

On the first page of this manual it has **two links** to both programs, **assembler** and **monitor**, conveniently unprotected. **Both work properly.** However I also recommend downloading the original **.d64** image of **PROFIMAT** since it contains some **BASIC** programs that do work correctly and complete this software package for the **C64.**

The german company **Data Becker** as editorial and company closed its doors in 2014. Today it is impossible to claim original and/or updated copies of the **PROFIMAT 64** software.

I have a original **PROFIMAT** package that includes **a single floppy disk** and his user manual. A floppy disk that according to the publisher **could be itself destroyed in a copy attempt** even with the only intention of having backup floppies of a legitimately acquired **PROFIMAT** software package.

This type of “business practices” covered in the so-called 'copyright' **in the long run they only lead to the disappearance by wear and aging of the magnetic supports** of commercial products that were at the time available to the general public.

This deserve to be denounced and deserves a deep rethinking about the suitability of laws that tend to condemn to disappear all kinds of developments, both technical, as artistic and cultural creations.

Even despite the educational and practical qualities of the **PROFIMAT** software, it may seem irrelevant by today's standards. However, this same complaint is applicable regarding recent or current commercial software developments that under no circumstances deserve to be condemned to their disappearance in the future with **uncomfortable or destructive DRM anti-copy protection systems** or with another modality of "business abuse" based on the so-called "Intellectual Property" using two antagonistic concepts in a same phrase.

The transcription and disinterested translation of this text is an effort in an opposite and necessary sense. I hope that in the not so distant future ceases to be necessary that anonymous people, with extensive technical knowledge, spend time to "hack" software applications to return them to life and the public domain. I hope that ceases to be necessary spend time to transcribe “User Manuals” impossible to find because they will not be published again.

The Commodore 64 and its user community will surely appreciate it deeply.

**FRANKIE SAY... BIT MORE. :)**

With the **PROFIMAT 64** floppy disk software package you can enter the world of machine code programming for the Commodore 64 directly from the BASIC editor.

With the brief but clear examples included in this manual, after loading the **PROFI-ASS** macro-assembler, in a few minutes you will get your first program compiled into machine code. The **PROFI-MON** monitor opens up a whole new world of possibilities for managing and examine every last byte of memory on your Commodore 64.

This manual includes links to the full **PROFIMAT 64** package and to recommended books so that you can dig deeper and experiment with direct programming of the 6502 and 6510 processors along with the added power of your C64's original chipset.