TURBO MACRO PRO

TMP Version 1.2 (September 2006)

CONTENT

Introduction

Turbo Macro Pro is a heavily modified and improved descendant of the original Turbo Assembler Macro, which is itself a follow up version to the original Turbo Assembler. The software can be thought of as having two distinct yet highly integrated parts - editor and assembler. The editor provides interactive commands for moving quickly through source code; buffering and copying lines or blocks of source code; saving, loading, and printing source code; search and replace; etc. In addition to editing and I/O, the user is able to trigger an assembly from within this editor itself. At that point the assembler takes over, producing 6502 machine code and providing error output and, upon successful assembly, an option to execute the code or return directly to the editor.

The integrated nature of Turbo Assembler is just one reason for using it. When you learn the key commands and develop a familiarity with our REU and X2 mods (here forward called Turbo Macro Pro+REU or TMP+REU, and Turbo Macro Pro(x2) or TMP(x2) for short) you will have at your control a very powerful native rapid development environment that speeds up the code-test-code cycle.

Furthermore, the Turbo Macro Pro family continues to grow as we add new support for exotic hardware and memory configurations, including in the present the DTV[1] and DTV v2 (or PTV) joystick systems, and in the future additional hardware such as IDE64[2] and memory expansions such as GEO-RAM[3].

Editor Commands

Special editor commands are accessed by pressing the 'command key', followed by another key to invoke a specific command. Certain commands will themselves invoke a submenu of additional commands. Several other commands may prompt for user input such as filenames, memory locations, and so on. The command key in all versions of TA, TAM, and TMP is the {back arrow} which is located to the immediate left of the number 1 key.

The table below shows all the commands that are invoked by first pressing the command key (C64: back arrow, PC keyboard: ^), following by the key in the 'KEY' column below:

KEY	ACTION
1	exit-basic: Leaves TMP to a BASIC prompt. TMP can be re-entered immediately after doing this with a 'sys 8*4096'.
!	view-seq: Prompts for a SEQ filename, then displays the contents of the file on screen. CBM pauses output, run/stop aborts.

[1] The C64 Direct-to-TV, called C64DTV for short, is a single-chip implementation of the Commodore 64 computer, contained in a joystick.
[2] The IDE64 cartridge is a device for connecting a CompactFlash (CF), IDE hard disk drive and/or DVD-ROM drive to a Commodore 64. It is plugged into the Expansion Port.
[3] GEO-RAM from Berkeley Softworks is a memory expansion unit for use on the Commodore 64 / 128 computer with GEOS operating system.

2	paste-separator: Outputs a 'separator' comment line to the current cursor line, overwriting anything else there.
3	assemble: Assembles to the assembly bank, which is bank 0 in all mods supporting extra RAM, or simply directly to c64 main memory on the unexpanded c64 mod.
#	assemble-to-object: Assembles to the current object bank. Only enabled on mods supporting extra RAM.
4	print-source: Prompts for a filename to which a source code print-out will be saved as a SEQ file. Instead of a filename, enter '?' by itself to print directly to a printer (device 4) or '*' by itself to output to the screen (CBM pauses output, run/stop aborts).
5	assemble-to-disk: Prompts for a filename to which assembled code is saved on disk.
%	assemble-to-slave: Assembles across a cable to a slave machine - only enabled in the X2/R2 mods.
6	make-data: Prompts for a region of memory (start and end address). Data in that region are read and translated into. byte statements which are inserted into the source code at the current cursor line. In mods supporting extra RAM, the data is read from the current object bank. Otherwise the data is read from the c64 main memory.
7	set-tab-return: Takes the current cursor location and sets that as the column at which the cursor is placed after pressing return.
8	set-tab-source: Takes the current cursor location and sets that as the column at which the source code (opcode/pseudo) is aligned at.
+	add-hex: Prompts for two 16bit values, adding them and outputting the result in hex and decimal.
-	sub-hex: Prompts for two 16bit values, subtracting the second from the first and outputting the result in hex and decimal.
DEL	delete-line: Removes the entire line under cursor, moving subsequent lines up by one.
INST	toggle-line-insert: Switches the editing mode for line insertion. When off, hitting return does not insert a new blank line.
^	copy-line-buffer: Copies the line under cursor to the line buffer.
#	paste-line-buffer: Pastes the line buffer to the current cursor line.
q	cursor-left-edge: Move cursor to column 0 (the leftmost column)
w	write-seq: Prompts for a filename to which the current source is saved to disk as a SEQ file.
e	enter-seq: Prompts for a (.SEQ) filename from which lines of source code are read off disk and inserted at the current cursor line.
r	replace-string: Prompts for search and replacement strings and then finds the first occurrence of the search string, leaving the cursor at that occurrence.
R	ram-submenu: Activate the expansion RAM submenu (where applicable), with these additional key commands:
l	load-to-ram: Prompts for a filename. A second prompt is given for a load address, using the file's actual load address as a default. Then the file is loaded into the current object bank at the given address.
b	backup-to-bank: Prompts for an available source bank number, and then copies the active source code into that bank.
o	set-object-bank: Prompts for an available RAM bank and uses that as the new object bank.
s	swap-to-bank: Prompts for an available source bank number. If the selected bank contains a TMP / source instance, then the active source code is backed up to its

	assigned bank and the selected bank is swapped into c64 main memory, becoming the new active source code. TMP will not allow swapping in a bank that doesn't already have a TMP instance in it.
j	set-jumpback: Prompts for an address, which will be the location to which the jumpback routine will be copied when executing an assembled code. If the address is set to $0000 then TMP will not copy the jumpback routine at all. The default address is $0140.
t	replace-one: Executes one replacement of a just-found search string and then searches for the next occurrence, moving the cursor to it.
y	replace-all: Executes replacements for every occurrence of the search string located from the current cursor location to the bottom of the source code.
u	list-labels: Prompts for a filename to which a listing of each label and the memory address that it resolves to will be saved to disk as a SEQ file. Instead of a filename, enter '?' by itself to print directly to a printer (device 4) or '*' by itself to output to the screen (CBM pauses output, run/stop aborts). Note that the output is based on the label resolutions from the last time the source code was assembled, and if list-labels is invoked prior to assembling any source code, the output will be empty.
U	list-labels-vice: Identical to list-labels, except the format of the output is compatible for loading into the VICE emulator monitor.
i	find-label: Prompts for a string to search for as a label, then places the cursor at the first occurrence of that string where it is used as a label definition.
p	preferences-submenu: Activates the preferences editor submenu which can be used to modify the separator style and color scheme: 0 Edits the color used for $d020 (border). 1 Edits the color used for $d021 (background). 2 Edits the color used for the message line. 3 Edits the color used for the status line. 4 Edits the color used for regular source. 5 Edits the color used for source with a syntax error. 6 Edits the color used for source in the current marked block. s Edits the separator template ^ Prompts for a filename, to which a complete copy of the newly customized TMP program is written to disk.
@	disk-command: Prompts for a string that is sent to the current serial device as a disk command, the results of which are displayed on the message line. Entering a '@' by itself will send a status command to the device.
*	view-directory: Outputs a directory listing from the current serial device to the screen. CBM pauses output, run/stop aborts.
a	petscii-mode: Sets the editor into a literal petscii mode. The cursor flashes slightly faster to indicate the setting. During this mode, all keypresses are taken literally as petscii sequences which are stored on the line. This mode is useful for entering color or cursor movement codes into .text strings that can be 'printed' from within your program. Hitting the {back-arrow} key will quit this mode and return to normal editing.
s	save-source: Prompts for a filename, to which the current source code is saved to disk as a binary PRG file.
d	increment-device: Selects the next available device number on the bus between 8 and 15.

f	find-string: Prompts for a string and then searches for the first occurrence in the source code.
g	goto-mark: Prompts for a mark (1-9, s, e) and moves the cursor to the line number corresponding to the selected mark.
h	find-next: Finds the next occurrence of the search string.
k	define-fkeys: Prompts for a redefinable function key (f3-f6) then allows the user to enter a new definition for the selected key.
K	reset-fkeys: The function keys f3-f6 are all reset to TMP's internal defaults. This is used when loading a source code that has a different set of key definitions than TMP.
l	load-source: Prompts for a filename, from which source code is read. Note, the loaded source code will *replace* any current source!
:	list-marks: Displays the current line numbers associated with each mark.
;	kill-mark: Prompts for a mark (1-9, s, e) and blanks out the line number corresponding to the selected mark.
z	undo-edit: Undos whatever current line editing has occurred. This can only be used as long as the cursor has not left the current line where editing has taken place!
c	cold-start: This performs a hard reset of TMP, initializing and blanking out the current source code.
b	block-submenu: Activates the block commands submenu, with these additional key commands:
w	write-block: Prompts for a filename to which the source code within the current marked block is saved to disk as a SEQ file.
c	copy-block: Copies the current marked block to the current cursor position.
m	move-block: Moves the current marked block to current cursor position.
k	kill-block: Erases the current marked block
n	goto-line: Prompts for a line number and then moves the cursor to the entered line.
m	set-mark: Prompts for a mark (1-9, s, e) and then sets the selected mark to the current cursor line. 's' and 'e' stand for start and end. Setting these marks defines a contiguous set of lines upon which subsequent block operations (invoked by the block-submenu) will operate.
=	join-line: The remainder of the current line (everything to the right and under the current cursor location) is moved into the same location on the line immediately above, deleting the old line in the process.
/	blank-to-end: Blanks out everything on the current line to the right and under the current cursor location.
RTRN	split-line: The remainder of the current line (everything to the right and under the current cursor location) is moved into a new line which is inserted immediately below the current line.
SPACE	blank-line: Blanks out the current line
CRSR-R	cursor-right-edge: Moves the cursor to column 39
CRSR-L	cursor-left-edge: Moved cursor to column 0
CRSR-D	Moves the cursor down by 200 lines
CRSR-U	Moves the cursor up by 200 lines
BARROW	The back arrow key, when pressed twice in a row, outputs an actual backarrow character.

The next table shows all the behavior of additional keys that are not combined with first pressing the command key:

KEY	ACTION
INST	toggle-char-insert: Switches the editing mode for character insertion. When off, new key presses overwrite instead of insert.
F1	Moves the cursor up by 20 lines
F2	Moves the cursor to line 0 (top of the source)
F7	Moves the cursor down by 20 lines
F8	Moves the cursor to the bottom of the source

Function keys f3-f6 are redefinable, and have these default operations:

KEY	ACTION
F3	Identical to {back-arrow}+CRSR-U (cursor up by 200 lines).
F4	Assembles (as with {back-arrow}+3) and then auto executes the code.
F5	Identical to {back-arrow}+CRSR-D (cursor down by 200 lines).
F6	Identical to {back-arrow}+R which invokes the RAM submenu.

TURBO MACRO PRO/TMPx SYNTAX

Assembler Overview

TMPx source code syntax is a superset of its c64-based sibling, Turbo Macro Pro. Basic elements of the syntax are:

labels	Something like variable names, a label is used to refer to a specific memory address or retain a value that can be used in an expression. Labels can be assigned a value like a memory address or just a numeric constant explicitly or are set by the assembler as it resolves unassigned labels.
opcodes	These are the standard 6502 mnemonics, in the form of three letter abbreviations. TMPx observes 6502 mnemonics as well as so-called illegal mnemonics and opcodes specific to the DTV v2 (aka PTV) joystick and wheel hardware.
pseudo ops	Assembler directives that instruct TMPx to perform some operations instead of merely translating an opcode into machine code.
comments	A simple way to annotate source code; comments are completely ignored by the assembler.

Assembler Details

Constant Values

Constant values can be expressed in either decimal, hexadecimal, binary, or as characters depending on how the value is written. The largest constant value TMPx can recognize is $ffff (65,535).

$	denotes a hexadecimal value
%	denotes a binary value (limited to single bytes)
'a'	denotes a character value, which translates to a byte

Any value that is not preceded by $ or % or wrapped in quotes is treated as a decimal value.

Examples

$20 = hex $20, dec 32
$2000 = hex $2000, dec 8192
15 = hex $0f, dec 15
%10001000 = hex $88, dec 135
'1' = hex $31, dec 49

ASCII/PETSCII

<ONLY TMPx> TMPx assembles from ASCII source code files. This presents some challenge when developing software for the C64 which uses PETSCII. TMPx resolves this by allowing the use of tokenized representations of PETSCII as necessary. The token notation chosen is referred to as 'bastext', which is the name of a program developed in the 90's to help bridge PETSCII to ASCII when translating BASIC programs and that used either short text identifiers or three digit (zero/left padded) decimal values inside curly brackets. TMPx recognizes the following bastext tokens:

Examples

{pound} = british pound sign, PETSCII code 92
{white} = color white, PETSCII code 5
{127} = graphical character, PETSCII code 127

The bastext notation is further expanded to include any three-digit decimal or hexidecimal value enclosed in curly braces. Such tokens are translated directly into the corresponding PETSCII code:

Examples

{$20} = space, PETSCII code 32
{063} = question mark, PETSCII code 63

A bastext token can be used as a single character constant value enclosed in single quotes or as or or more characters in a double quoted string. Bastext tokens are referred to as either 'named' (they use a short text identifier) or un-named (they use decimal or hexidecimal). ~~A full list of the named bastext tokens recognized by TMPx is given at the end of this documentation.~~

Labels

Valid labels in TMPx can be composed of any combination of letters, numbers, and the underscore character (obtained by pressing CBM+@). Labels must begin with a letter or underscore. Label names can be used in expressions (see below) in the same manner as constant values.

To define a label explicitly, set the label to either a constant value, to another label, or to an expression which will be resolved into an value. Once set, a label's value can not change. Attempting to define the same label twice will result in an error.

Examples

```
bah = $1000     ;sets the label 'bah' to $1000.
lab = bah       ;sets the label 'lab' to equal whatever address bah
                ; resolves to (in this case, it would also be $1000).

tmp = bah+$8    ; sets the label 'tmp' to equal the address that the
                ; expression 'bah+$8" resolves to ($1008).
```

There is a special label, which is called the 'program counter' and is represented in source code as the asterisk (*). It is important to set the value of * at the top of your code. When TMPx assembles your source, * is used to tell the assembler where the machine code should be assembled to. Set the value of * just like a label:

```
*   = $1000   ; sets the program counter to $1000.
```

Note that unlike normal labels, you can reset the value of * anywhere you want in your source code. Each time you redefine *, the assembler simply uses the new value and continues assembling machine code at the new address.

Expressions

Expressions in TMPx can be formed with the following operators, all of which are binary operators (i.e. they require a value on the right AND left side of the operator):

+	for addition
-	for subtraction
*****	for multiplication
/	for division
&	for bitwise and
.	for bitwise or
:	for bitwise eor

Values used with expressions can be constant values described above, or labels. Note that you can also use parenthesis and as you would expect they affect the order in which the expression is evaluated:

Examples

```
$20 + 4        = hex $24, dec  36
15 - %00000011  = hex $0c, dec  12
$ff & $f0      = hex $f0, dec 240
(2*$10)+1      = hex $21, dec  33
2*($10+1)      = hex $22, dec  34
```

Unlike standard expressions in C/C++, there is no operator precedence. For example, multiplication has no higher precedence than addition:

```
2*$10+1        = hex $21, dec  33
1+2*$10        = hex $20, dec  32
```

In the first example above, the multiplication is resolved first, followed by the addition. In the next example, the addition is processed first! As this is a simplified way of resolving expressions than is more commonly seen in programming languages, it is important to keep this in mind. Careful use of parenthesis can help make the expected results explicit.

The special '*' label can also be used in expressions, and it will always resolve to whatever the current program counter is where the assembler is outputting machine code to:

```
*= $1000         ; sets the program counter to $1000
```

```
jmp *+$03       ; assembles as jmp $1003
jmp *           ; also assembles as jmp $1003 because the program counter
                ; at this line will actually be $1003.
```

The value of an expressions can also be modified by the following characters, which must appear at the very front of the expression:

<	denotes that the low byte of the following constant or expression should be taken
>	denotes that the hi byte of the following constant or expression should be taken
!	denotes an expression whose value may currently or later be determined as needing only 1 byte (i.e. 0-255) but which should be expanded into 2 bytes for the purposes of outputting machine code. See below for an example.

Examples

```
>$0314              = hex $03
<$0314              = hex $14
<$0400+(6*$28)      = hex $f0
```

Examples of the use of '!' are shown below.

```
lda $02         assembles as lda $02
lda !$02        assembles as lda $0002
```

Another use of the '!' is to prevent phase errors. A phase error occurs when the length of code as determined by pass one and pass two of the assembler do not match. One way this can occur is by labels referenced before they are defined:

```
    lda bah  ; here, bah is undefined and TMPx assumes it is a word
    rts
bah = $02    ; bah is now defined as a byte value
```

Assembling this code block results in a phase error, because TMPx assumes undefined labels represent words during the first pass. If the label is then found defined as a byte value, or if it actually resolves to a byte value, then the second pass will be shorter than the first and a phase error occurs. To prevent the phase error change the code to look like:

```
    lda !bah
    rts
bah = $02
```

The '!' tells TMPx that the expression following should be treated as a word no matter how the label was defined. Obviously, the best choice would be to define labels before referencing them but in our own experience this is sometimes undesired so using '!' notation will help avoid the woes of phase errors.

Comments

You can add comments to your source code by using the ';' (semicolon). Any ';' found by turbo will cause the assembler to ignore whatever follows the ';' til the end of that line. So comments can follow an opcode, or a pseudo-op, or they can be on a line by themselves.

Examples

```
 ; this is a comment
lda #$01        ; this is a comment
sta $d020       ; so is this!
```

Pseudo-ops: Data

The following pseudo-ops are the standard way to make data tables. Note that values are not necessarily constrained to constants - they can be complete expressions as shown above.

.byte	takes a list of byte values which can be decimal, hex, binary or character constants and produces a table of bytes.
.word	takes a list of word values which can be decimal or hex and produces a table of bytes where the words are arranged in low-byte hi-byte order.
.rta	takes a list of word values which can be decimal or hex and produces a table of bytes where the words are decremented by 1 and set in low-byte hi-byte order this is useful for doing stack manipulation with e.g. return addresses.

Examples

```
.byte 25,"a",$cc   =  $19 $41 $cc
.word $fce2        =  $e2 $fc
.rta $fce2         =  $e1 $fc
```

A similar set of pseudo-ops can be used to make string tables. Note that bastext tokens can be used inside strings.

.text	takes a quoted string and converts it into a span of bytes. Characters are converted from ASCII to PETSCII, including any bastest token conversion.
.null	as .text, but the span of bytes is terminated with a null (0) byte.
.shift	as .text, but the last byte has its high-bit set to 1.
.screen	as .text, but the characters are converted into screen codes.

Examples

```
.text "hi"       =  $48 $49
.null "hi"       =  $48 $49 $00
.shift "hi"      =  $48 $c9
.screen "hi"     =  $08 $09
```

<ONLY TMPx> Another pseudo-op is provided for convenience when you want to specify a large span of repetitive values. .repeat with a count followed by a sequence of one or more values or expressions will create as many copies of the value/expression as you've specified.

.repeat	produce 1 or more copies of a sequence of 1 or more values.

Examples

label = $9876

```
.repeat 8,$ff        =  $ff $$ff $ff $ff $ff $ff $ff $ff
.repeat 3,label      =  $76 $98 $76 $98 $76 $98
.repeat 2,"a","b","c"  =  $41 $42 $43 $41 $42 $43
```

Pseudo-ops: Conditional Assembly

These pseudo-ops allow TMPx to assemble chunks of code based on the evaluation of an expression - if it is equal or not equal to zero or if it is positive or negative. You can use the functionality to selectively assemble code based on the value of a label, for example, or even the current value of the program counter.

.if	tests value or expression for inequality with zero (same as .ifne).
.ifne	tests value or expression for inequality with zero (same as .if).
.ifeq	tests value or expression for equality with zero.
.ifpl	tests value or expression for positive (the condition passes if the evaluated expression is from $0 - $7fff and fails if the evaluated expression is from $8000-$ffff).
.ifmi	tests value or expression for negative (the condition passes if the evaluated expression is from $8000 - $ffff and fails if the evaluated expression is from $0-$7fff).
.endif	marks the end of the code block that is assembled when the conditional test is met.

Examples

```
cycle = 65
   .ifne cycle-65 ;if cycle != 65
   nop          ;produce one nop
   .endif
```

<ONLY TMPx> In addition to conditions based on the value of an expression, two other pseudo-ops test based simply on whether a label had been defined or not, regardless of its value. Both of the following pseudo-ops are also closed by .endif.

.ifdef	tests whether a label has been defined, condition passes if it has.
.ifndef	tests whether a label has been defined, condition passes if it has not.

Examples

```
ntsc = 1
   .ifdef label
   nop          ;produce one nop
   .endif
```

Note that .ifdef and .ifndef can be most useful for testing whether a label definition has been specified in the command line arguments to TMPx. A scenario where this may be applied would be activating debugging code in output based on passing TMPx a "-D DEBUG" command line argument and testing for that in the source with ".ifdef DEBUG"

Pseudo-ops: Blocks (Scope Control)

The block pseudo-op allows a very useful operation: marking a section of code so that it can contain its own local labels, which can be redefined outside of the block or within a different block. Using blocks when coding reusable subroutines can ease their integration into new code... when the blocked subroutine is inserted into another source, a coder needn't worry that the labels used inside the subroutine block will conflict with any labels already defined in the other code. Note that blocks can be nested inside other blocks, each one creating a local scope for labels defined inside.

.block	starts a code block
.bend	ends a code block

Examples

```
tmp = $02       ; tmp is defined as $02
lda tmp         ; assembles as lda $02
sub             ; our subroutine label name, 'sub'
   .block
tmp = $ff       ; tmp is defined as $ff but only applies in the block
   clc
   adc tmp      ; assembles as  adc $ff, not adc $02!
   rts
   .bend
   lda tmp      ; assembles as  lda $02 again...
```

Pseudo-ops: Variables

Variables are a special kind of label that can be redefined (even without worrying about localized blocks). This is the only way to change the value of a label in the same context without getting a 'double defined' error. These are most useful when combined with other pseudos' like conditionals (see above) and goto (see below).

.var	defines a variable identified by the label preceding '.var', which is set to the value of the expression following the '.var'.

Examples

```
va   .var $01    ; defines a var named va with value of 1
    lda #va      ; results in lda #$01
```

```
va    .var va+1    ; redefines va to its current value+1
      lda #va      ; results in lda #$02
```

Pseudo-ops: Unconditional Goto

These pseudo-ops force the assembler to jump to a given label and continue assembling from there. This can be extremely useful for repeating a short code block 10, 100, or even more times. Only labels flagged with a .lbl pseudo-op can be referenced by a .goto!

.lbl	identifies a label that can be referenced by a .goto
.goto	causes assembler to goto the label referenced and continue assembling from that point on.

Examples

```
cnt    .var $0100 ;cnt = $100
loop   .lbl       ; label 'loop' is prepared to be a .goto target
       nop
cnt    .var cnt-1 ; dec cnt
       .ifne cnt  ; if cnt != 0
       .goto loop ; goto loop
       .endif     ; until all $100 nop's are assembled
```

Pseudo-ops: Macros

Macros are a means of inserting a larger span of code by using a short macro label followed by 0 to 8 arguments. This "macro call" is expanded into the code at assembly time, with substitutions made for the arguments given. A macro call is identified by a '#' followed by the macro label.

.macro	starts a macro definition that will implicitly treat the expanded code block as though it were enclosed by .block/.bend. around the macro.
.segment	starts a macro definition but without any implied block.
.endm	ends a macro definition.

Macro argument substitutions are denoted in the macro code with either a '\' (ASCII backslash) to mark a numeric substitution or an '@' symbol to mark a string substitution.

Examples

```
poke   .macro    ; start macro def
       lda #\2   ; the \2 means an argument substitution, in this case the
                 ; second macro argument will replace the \2
       sta \1    ; another substitution
       .endm     ; end macro def
```

In the source code, a macro call for the above macro definition would look like:

```
    #poke $d020,0
```

Upon assembly the macro call is expanded, and substitutions are made (in this case \1 is replaced by $d020 and \2 is replaced by 0, resulting in the following assembly being generated to replace the macro call:

```
    lda #$00
    sta $d020
```

Here is a more complex example showing recursion with macros:

```
table .macro     ; start macro def
cnt   .var \1+1  ; set var cnt to arg 1 plus 1
    #tab         ; a macro call within a macro!
    .endm        ; end macro def
tab   .segment   ; start macro def
cnt   .var cnt-1 ; decrement cnt
    .if cnt      ; if cnt != 0
    #tab         ; then recursively macro call itself
    .endif       ; end of the if
    .byte cnt    ; assemble a byte with current value of cnt
cnt   .var cnt+1 ; increment cnt
    .endm        ; end macro def
    #table 64    ; call the macro to create a byte table from 0 to 64
```

Here is an example of using a text parameter in a macro definition and call:

```
error  .macro    ; start macro def
    lda #<tx     ; lo-byte of tx
    ldy #>tx     ; hi-byte of tx
    jsr $ab1e    ; print it
    jmp end      ; skip text data
tx    .null "@1" ; @1 is replaced by a textual argument
end   .endm      ; end macro def
    #error "doc too boring"
```

So you see, use \x for numerical arguments, and @x for text arguments, where x denotes the argument number.

Text arguments can be even used to modify an opcode! Example:

```
do    .segment
    .block
loop  .endm
while .segment   ; start macro def
    b@1 loop     ; the @1 will be replaced by a text argument
    .bend
    .endm        ; end macro def
```

```
ldx #5          ; start of code
#do             ; call macro do
ldy #2
lda #"*"
#do             ; call macro do
jsr $ffd2
dey
#while "pl"  ; call macro 'while' using parameter "pl"
lda #" "        ; print space
jsr $ffd2
dex
#while "ne" ; call macro 'while' with parameter "ne"
```

Pseudo-ops: Includes

The .include pseudo-op allows you to assemble code from a different source file. For example, you could use .include to assemble in a file with label definitions for all of the kernal jumptable, or you could write small reusable subroutines separately and include them in as you need them.

.include	load and assemble the specified file

Example

```
.include "kernel.s"
```

The .binary pseudo-op will include the binary data in the given file directly into the assembled output at the location of the .binary call. You may optionally specify a number of data bytes to skip from the start of the included file by specifying that after the filename. This would be used to, for example, skip a two byte load address at the start of the file.

.binary	load and output bytes from the specified file, optionally skipping bytes at the start of the file

Example

```
.include "3d.dat"
.include "music.prg",2
```

Pseudo-ops: Code Offsets

The .offs pseudo op is used to alter the memory address to where code is being assembled. This sounds identical to the function of the program counter (*) but there is a distinction: * sets the "effective address" for assembled code, meaning, object code is generated assuming that it is located at the address in the program counter. But .offs changes the "actual address" to where object code is assembled. If you do not use calls to .offs in your code, then the actual address and effective address will remain identical. Used together, you can assemble code so that it will execute properly in a different part of memory than where the code itself is located in the assembled output.

.offs	change the location that code is assembled to. Code will be assembled forward in memory the value of the expression following .offs

Example

```
   (this code)   ->  (assembles as)
    * = $2000

start  bit base0      bit $2009
     bit base       bit $8000

     jmp *          jmp $2006

base0  * = $8000

base   .offs base0-*

     lda #>start    lda #$20

     jmp *          jmp $8002
```

This code is assembled from $2000 to $200d; by using an expression for the .offs we can generate code in a contiguous memory span, even though part of the code has been assembled to execute at a different memory address (in this case, $8000). In this example the .offs is effectively offsetting the assembled code backwards in memory. Naturally this can be extremely useful for coding routines that need to execute in zero page, or for coding routines to run in drive memory.

Pseudo-ops: Printer Control

The printer pseudo-ops allow some control of output during assembly. .proff and .pron allow you to skip printing sections of code, while .hidemac and .showmac let you control whether the expanded macro gets printed or just the macro call.

.pron	turn on printing
.proff	turn off printing
.hidemac	show unexpanded calls
.showmac	show expanded calls

Pseudo-ops: Miscellaneous

.eor	followed by some value, with which all code after the .eor is eor'd with.
.end	terminate assembling
.bounce	<ONLY TMPx>try ".bounce 2,256,0,255,1,0,0,100,100,0,0" for a good time.

HISTORY & GENEALOGY

The first version of Turbo Assembler (referred to in shorter form as TA when speaking generally, or as oTA when speaking specifically about the 'original' version) was developed in 1985 by a German company called Omikron Software, and in particular a man named Wolfram Römhild. As with most software, cracked versions eventually made their way, and spurred adoption of TA as the de facto standard assembler for most of the c64 scene.

Over the following years multiple versions of TA were subsequently released by various persons and groups claiming to have improved the tool. In truth, most of these variants are fairly simple hacks where little more than the color scheme and the opening credit line are "improved"... Nonetheless, some versions did achieve useful additions or even significant steps in advancing the tool.

Perhaps the most significant alterations have been those adding the capability of using the REU (RAM Expansion Unit or also referred as 'xmem' by some sceners). REU capable versions of TA include Fairlight's "Xass v3.3" and Micron/Success's "Tasm v5.6x".

But what about Turbo Assembler Macro (referred to in shorter form TAM or oTAM)? A significant upgrade of TA, it kept the editor system essentially the same while making major upgrades to the capabilities of the assembler including macros, local labels, assembler variables and loops, etc. For whatever reason, the scene adopted the first version, the 'original' Turbo Assembler, to a huge majority over Turbo Assembler Macro. Still, a few people have also claimed to make upgrades to the Macro version, with a similar ratio of dubious to genuine improvements. REU versions of TAM were also produced; perhaps the first was Antitrack's mod, and also Paradroid/Sharks who made an REU mod.

Finally, we come to our version. Its root is the Antitrack mod of oTAM (called "Turbo Assembler Macro+ 1764"). This was subsequently updated by Massive Onslaught who added an invokable 'REU Menu' and the 'jumpback routine' concept that allowed an easy and built in way to return back to the editor after testing code; this version was called "Turbo Assembler Macro++ REU". Then, in late 1993 Massive Onslaught and Count Zero worked together to actually reassemble that code base. The project stalled for a while, Massive Onslaught joined Style and eventually collaborated with Elwix, brain storming ideas and upgrades going far beyond a standard REU modification. They finally launched into the real coding in fall 1995, removing significant sections of redundant code or size optimizing the existing code. After over a year of lazy off and on coding, optimizing, and bug fixing, they arrived at a much improved version of TAM with several new editor functions including fully integrated REU commands, as well as introducing in this release the unique capability to bank and swap between up to 6 separate source codes (with a 512k REU) at any time, giving the user full control over source and object banks; you could assemble one source directly to your non-volatile object bank; you could then assemble a different source and start the code. The potentials were endless for quickly backing up a source to the REU while making changes, or for using source code libraries in separate files, or for working on very large projects where the source is better handled split into 2 or more parts.

The first release of this advanced mod was made in January, 1997 and dubbed "Turbo Macro Pro v1.0". In March and October 1997 two additional releases were made with more improvements as well as a version for non-REU expanded systems and dual-c64 systems. Then after 7 years of absence from the

niche scene of Turbo modding, Style returned in 2004 with a brand new modification supporting the DTV joystick system, and a year later expanding that for the DTV v2 joystick/wheel systems.

CREDITS & THANKS

Coding

Elwix/Style: General improvements including features, bug fixes, optimizations; DTV/PTV mods.

Massive Onslaught/Style: General improvements including features, bug fixes, optimizations.

The Wiz/Style: X2/R2 mods for dual-c64 systems.

Reassembling: Massive Onslaught/Style

Count Zero/TRSI

Thanks

Bacchus/Fairlight: for *numerous* suggestions and bug reports and generally taking a real interest in our project and sending so many friendly and helpful emails!

Antitrack: for the fascinating chat about Turbo's dongle protection and for several suggestions. Also some of the examples in the general reference document come from the Vizawrite (argh!) files ATT made by translating parts of the original German manual.

Paradroid/Sharks: for our discussions about Turbo Assembler modding and sharing experiences about same.

XmikeX: for making good suggestions.

Macbeth/PSW: for a highly frustrating but necessary collection of bug reports, and also for generally being enthusiastic about this project!

Fungus/Carcass: for more frustrating bug reports.

Six/Style: DTV expertise (assistance, testing, prototype borrowing, joystick hacking, and general ass kicking).