

# TurboForth TMS9900 Assembler Version 1.0

For TurboForth V1.2

Ported by Mark Wills from the original  
TI-Forth Assembler source code

## Table of Contents

1 -Introduction to the TurboForth 9900 Assembler.....	3
2 -TMS900 Assembly Mnemonics.....	3
3 -TurboForth's Workspace Registers.....	4
4 -Using the Assembler .....	5
5 -TMS9900 Addressing Modes with the Forth Assembler .....	5
5.1 -Workspace Register Addressing.....	5
5.1.1 -Enhanced Functionality.....	5
5.2 -Symbolic Memory Addressing.....	6
5.2.1 -Enhanced Functionality.....	6
5.3 -Workspace Register Indirect Addressing.....	7
5.3.1 -Enhanced Functionality.....	7
5.4 -Workspace Register Indirect Auto-Increment Addressing.....	8
5.4.1 -Enhanced Functionality.....	8
5.5 -Indexed Memory Addressing.....	9
5.5.1 -Enhanced Functionality.....	9
5.6 -Addressing Mode Words for Special Registers.....	9
6 -Structured Assembler Constructs.....	10
7 -Assembler Jump Tokens.....	11
7.1 -Assembly Example for Structured Constructs.....	12
8 -Test Code.....	13
8.1 -Add two numbers.....	13
8.2 -Fill screen with a character.....	13
8.3 -Nested Delay.....	15
9 -General Usage Notes.....	16
9.1 -Using Your Own Workspace.....	16
9.1.1 -Restoring TurboForth's Workspace.....	16
9.2 -Changing TurboForth's Workspace.....	17
9.3 -Calling Other Code Words.....	18
9.3.1 -Using High-level Definitions to Glue Assembly Words Together.....	19
9.4 -Mixing Forth and Assembly Language in an ASM: Definition.....	20
9.5 -Accessing Forth Strings from Assembly Language.....	21
9.5.1 -Code Breakdown.....	24
10 -TurboForth Assembler Code.....	27
10.1 -Errors Corrected from the Original Source Code.....	30
11 -Conclusion.....	31

## 1 - Introduction to the TurboForth 9900 Assembler

The assembler presented here is ported from the original FIG Forth implementation for TI Forth, modified for use with F83 TurboForth by Mark Wills.

The assembler is typical of assemblers supplied with Forth systems; it provides the capability of using all the op-codes of the TMS9900 as well as the ability to use structured assembly instructions.

Labels are not supported; they are not actually required, since the assembler adds high-level control structures such as `IF, ...ENDIF`, and `BEGIN, ...WHILE, ...REPEAT`, making labels completely redundant.

The complete Forth language is available to the user to assist in macro type assembly if desired. The assembler uses the standard Forth convention of Reverse Polish Notation (RPN) for each instruction. For example the instruction to add register 1 to register 2 is:

**R1 R2 A,**

As can be seen in the above example, the 'add' instruction mnemonic is followed by a comma. Every op-code in the assembler is followed by a comma. The significance is that when the op-code is reached during the assembly process, the instruction is compiled into the dictionary at that point. The comma convention serves as a reminder of this compile operation. It also serves to assist in differentiating assembler words from the rest of the words in the Forth language. Note that the *order* in which registers and other operands are specified is *exactly the same as standard 9900 assembly language*.

## 2 - TMS900 Assembly Mnemonics

A,	JEQ,	RSET,
AB,	JGT,	RTWP,
ABS,	JH,	S,
AI,	JHE,	SB,
ANDI,	JL,	SBO,
B,	JLE,	SBZ,
BL,	JLT,	SETO,
BLWP,	JMP,	SLA,
<b><u>CMP,</u></b>	JNC,	SOC,
CB,	JNE,	SOCB,
CI,	JNO,	SRA,
CKOF,	JOC,	SRC,
CKON,	JOP,	SRL,
CLR,	LDCR,	STCR,
COC,	LI,	STST,
CZC,	LIMI,	STWP,
DEC,	LREX,	SWPB,
DECT,	LWPI,	SZC,
DIV,	MOV,	SZBC,
IDLE,	MOVB,	TB,
INC,	MPY,	X,
INCT,	NEG,	XOP,
INV,	ORI,	XOR,

The above words are available when the assembler is loaded. Note that the 9900 instruction `C` (compare) has been renamed to `CMP`, - this is to avoid collision with the standard Forth word `C`, which is used to compile a byte to memory.

### 3 - TurboForth's Workspace Registers

Most assembly code in Forth will probably use the Forth workspace registers. The following table describes the register allocation.

Register Name	Usage
<b>R0</b>	Available.
<b>R1</b>	Available.
<b>R2</b>	Available.
<b>IP (R3)</b>	Interpretive pointer.
<b>SP (R4)</b>	Stack pointer.
<b>RP (R5)</b>	Return stack pointer.
<b>W (R6)</b>	Inner interpreter current word pointer.
<b>R7</b>	Available.
<b>R8</b>	Available.
<b>R9</b>	Available.
<b>R10</b>	Available.
<b>R11</b>	Available.
<b>NEXT (R12)</b>	Points to the next instruction fetch routine at >8328. May be used as long its value is restored before returning to the TurboForth environment.
<b>R13</b>	Available.
<b>R14</b>	Available.
<b>R15</b>	Available.

As can be seen from the above table, the only registers that absolutely must not be changed are R3, R4, R5 and R12. Whilst R6 is used it is transitory and can generally be used in an assembly language word. All other registers may be used, however it should be noted that TurboForth code words (the words inside the TurboForth dictionary implemented in machine code) also use these registers (obviously). Therefore, if you wish to persist data between assembly language routines you should either use the stack, or use a separate workspace. Please see section 9.1, Using Your Own Workspace, page 16 and 9.3, Calling Other Code Words, page 18 for more information.

## 4 - Using the Assembler

To use the assembler, simply load it from the appropriate block. Assembly definitions begin with the word **ASM:** and end with the word **;ASM**. The following example copies the current frame count (from the VDP interrupt) to the stack:

```
ASM: FRAME
    SP INCT,           \ make space on stack
    *SP CLR,           \ zero the top of stack
    $8379 @() *SP MOVB, \ move to stack
    *SP SWPB,          \ move to low byte
;ASM
```

To run the example, just type **FRAME**, and a value will be pushed to the stack.

## 5 - TMS9900 Addressing Modes with the Forth Assembler

All of the addressing modes of the 9900 are supported by the assembler. Each of the following examples will show both the Forth assembler code for various instructions, and the more conventional TMS9900 assembly method of coding the same instructions. The TurboForth version of the assembler has been enhanced with some additional functionality with respect to addressing modes, which considerably improve the readability of the resulting assembly code. This is indicated where appropriate in the following sections.

### 5.1 - Workspace Register Addressing

TMS9900 registers in Forth assembler are referenced directly by number:

Forth Assembler	Conventional Assembler
<b>ASM: EX1</b>	DEF EX1
1 2 A,	EX1 A R1,R2
3 INC,	INC R3
3 FFFC ANDI,	ANDI R3,>FFFC
<b>;ASM</b>	B *NEXT

Note how the operands are specified in the same order as standard TMS9900 assembly language; the only difference at this point is that the instruction mnemonic has moved to the end.

#### 5.1.1 - Enhanced Functionality

Note that, for enhanced clarity, the ability to use register numbers starting with **R** has been added. This functionality is not included in the original TI-Forth assembler. For example, the above code can be changed as follows, which makes it more readable:

Forth Assembler	Conventional Assembler
<b>ASM: EX1</b>	DEF EX1
<b>R1 R2 A,</b>	EX1 A R1,R2
<b>R3 INC,</b>	INC R3
<b>R3 FFFC ANDI,</b>	ANDI R3,>FFFC
<b>;ASM</b>	B *NEXT

## 5.2 - Symbolic Memory Addressing

Symbolic addressing is done with the @ ( ) word. It is used *after* the address. Note how, in the Forth assembler, it is possible to refer to earlier defined *Forth* variables (and indeed other words, constants etc.)

Forth Assembler	Conventional Assembler
<b>VARIABLE VAR1</b>	VAR1 BSS 2
<b>VARIABLE VAR2 5 VAR2 !</b>	VAR2 DATA 5
<b>ASM: EX2</b>	DEF EX2
<b>VAR2 @ ( ) R1 MOV,</b>	EX2 MOV @VAR2,R1
<b>R1 2 SRC,</b>	SRC R1,2
<b>R1 VAR1 @ ( ) S,</b>	S R1,@VAR1
<b>VAR2 @ ( ) VAR1 @ ( ) SOC,</b>	SOC @VAR2,@VAR1
<b>;ASM</b>	B *NEXT

### 5.2.1 - Enhanced Functionality

In addition to the @ ( ) notation shown above, Symbolic Memory Addressing can also be specified by using the @@ notation, as found in the Wycove Forth assembler. This author finds the Wycove notation much more readable than the TI-Forth notation. The modified code is shown below:

Forth Assembler	Conventional Assembler
<b>VARIABLE VAR1</b>	VAR1 BSS 2
<b>VARIABLE VAR2 5 VAR2 !</b>	VAR2 DATA 5
<b>ASM: EX2</b>	DEF EX2
<b>VAR2 @@ R1 MOV,</b>	EX2 MOV @VAR2,R1
<b>R1 2 SRC,</b>	SRC R1,2
<b>R1 VAR1 @@ S,</b>	S R1,@VAR1
<b>VAR2 @@ VAR1 @@ SOC,</b>	SOC @VAR2,@VAR1
<b>;ASM</b>	B *NEXT

The key to understanding this is simply to realise that when using an addressing mode, it simply follows the operand to which you want it to apply.

Note how, in the example above @@ *follows* the **VAR2** operand, giving us:

```
VAR2 @@
```

This is directly equivalent to @**VAR2**.

## 5.3 - Workspace Register Indirect Addressing

Workspace Register Indirect Addressing is done with the **\*?** word. It is used *after* the register number to which it pertains.

Forth Assembler	Conventional Assembler
2000 CONSTANT XRAM	XRAM EQU >2000
ASM: EX3	DEF EX3
R1 XRAM LI,	EX3 LI R1,XRAM
R1 *? R2 MOV,	MOV *R1,R2
;ASM	B *NEXT

### 5.3.1 - Enhanced Functionality

In addition to the **\*?** notation shown above, Workspace Register Indirect Addressing can also be specified by using the **\*\*** notation, as found in the Wycove Forth assembler. The modified code is shown below:

Forth Assembler	Conventional Assembler
2000 CONSTANT XRAM	XRAM EQU >2000
ASM: EX3	DEF EX3
R1 XRAM LI,	EX3 LI R1,XRAM
R1 ** R2 MOV,	MOV *R1,R2
;ASM	B *NEXT

Once again, notice how the addressing mode, in this case **\*?** or **\*\*** *follows* the register:

```
R1 **
```

...is directly equivalent to **\*R1**.

## 5.4 - Workspace Register Indirect Auto-Increment Addressing

Workspace Register Indirect Auto-Increment Addressing is performed with the **\*?+** word. It is used *after* the register to which it pertains:

Forth Assembler	Conventional Assembler
<b>2000 CONSTANT XRAM</b>	XRAM EQU >2000
<b>ASM: EX4</b>	DEF EX4
<b>R1 XRAM LI,</b>	EX4 LI R1,XRAM
<b>R1 *?+ R2 MOV,</b>	MOV *R1+,R2
<b>;ASM</b>	B *NEXT

### 5.4.1 - Enhanced Functionality

In addition to the **\*?+** notation shown above, Workspace Register Indirect Auto-Increment Addressing can also be specified by using the **\*+** notation, as found in the Wycove Forth assembler. The modified code is shown below:

Forth Assembler	Conventional Assembler
<b>2000 CONSTANT XRAM</b>	XRAM EQU >2000
<b>ASM: EX4</b>	DEF EX4
<b>R1 XRAM LI,</b>	EX4 LI R1,XRAM
<b>R1 *+ R2 MOV,</b>	MOV *R1+,R2
<b>;ASM</b>	B *NEXT

Once again, the important distinction is that the addressing mode follows the register to which it pertains.

**R1 \*+** is equivalent to **\*R1+**

The instruction **MOV \*R1+, \*R2+** would be coded as **R1 \*+ R2 \*+ MOV,**

The ovals show the relationship between the register, and its addressing mode modifier.



## 5.5 - Indexed Memory Addressing

The final addressing type is Indexed Memory Addressing. This is performed with the **@ (?)** word used *after* the index *and* register, as shown below:

Forth Assembler	Conventional Assembler
<b>2000 CONSTANT XRAM</b>	XRAM EQU >2000
<b>ASM: EX5</b>	DEF EX5
<b>XRAM R1 @ (?) R2 MOV,</b>	EX5 MOV @XRASM(R1), R2
<b>XRAM 22 + R2 @ (?)</b> <b>XRAM 26 + R3 @ (?) MOV,</b>	MOV XRAM+22@ (R2), XRAM+26@ (R3)
<b>;ASM</b>	B *NEXT

### 5.5.1 - Enhanced Functionality

In addition to the **@ (?)** notation shown above, Indexed Memory Addressing can also be specified by using the **()** notation, as found in the Wycove Forth assembler. The modified code is shown below:

Forth Assembler	Conventional Assembler
<b>2000 CONSTANT XRAM</b>	XRAM EQU >2000
<b>ASM: EX5</b>	DEF EX5
<b>XRAM R1 () R2 MOV,</b>	EX5 MOV @XRASM(R1), R2
<b>XRAM 22 + R2 ()</b> <b>XRAM 26 + R3 () MOV,</b>	MOV XRAM+22@ (R2), XRAM+26@ (R3)
<b>;ASM</b>	B *NEXT

## 5.6 - Addressing Mode Words for Special Registers

In order to make addressing modes easier for the **W**, **RP**, **IP**, **SP** and **NEXT** registers, the following words are available and eliminate the need to enter the register name separately:

Register Address	Indirect Mode	Indexed Mode	Indexed Auto-Increment Mode
W	*W	@ (W)	*W+
RP	*RP	@ (RP)	*RP+
IP	*IP	@ (IP)	*IP+
SP	*SP	@ (SP)	*SP+
NEXT	*NEXT	@ (NEXT)	*NEXT+

## 6 - Structured Assembler Constructs

This assembler also permits the user to write structured (label-less) code. This is done in a manner very similar to the way that Forth implements conditional constructs. The major difference is that rather than taking a value from the stack and using it as a true/false flag, the *processors' status register* is used to determine whether or not to jump.

Using this technique, it is possible to use constructs such as `IF, ...ENDIF,` and `BEGIN, ...WHILE, ...REPEAT,` etc. in assembly code.

The following structured constructs are implemented:

- `IF, ... ENDIF,`
- `IF, ... ELSE, ... ENDIF,`
- `BEGIN, ... UNTIL,`
- `BEGIN, ... AGAIN,`
- `BEGIN, ... WHILE, ... REPEAT,`

As noted above, these constructs work in the same way as their Forth counterparts, except that they *check the processors' status register*, rather than take a value from the data stack.

### Note:

The three *conditional* words (`IF`, `UNTIL`, and `WHILE`, ) *must* each be preceded by one of the jump tokens shown in the next section.

## 7 - Assembler Jump Tokens


The following assembler jump tokens are defined in the assembler. These jump tokens are used in conjunction with *conditional constructs* in order to assemble jump instructions, as discussed in the following sections.

Token	Comment	Instruction Used
<b>EQ</b>	True if =	JNE
<b>GT</b>	True if signed >	JGT \$+1 JMP
<b>GTE</b>	True if signed > or =	JLT
<b>HI</b>	True if unsigned >	JLE
<b>HE</b>	True if unsigned > or =	JL
<b>LO</b>	True if unsigned <	JHE
<b>LE</b>	True if unsigned < or =	JH
<b>LT</b>	True if signed <	JLT \$+1 JMP
<b>LTE</b>	True if signed < or =	JGT
<b>NC</b>	True if no carry	JOC
<b>NE</b>	True if equal bit not set	JEQ
<b>NO</b>	True if no overflow	JNO \$+1 JMP
<b>NP</b>	True if not odd parity	JOP
<b>OC</b>	True if carry bit is set	JNC
<b>OO</b>	True if overflow	JNO
<b>OP</b>	True if odd parity	JOP \$+1 JMP

Conditional constructs **IF**, **UNTIL**, and **WHILE**, *must* be *preceded* by a jump token from the above table. This is because **IF**, **UNTIL**, and **WHILE**, actually assemble jump instructions into your code, and the *jump token* tells the assembler which type of jump instruction to assemble.

The actual jump instruction assembled into your assembly language routine may appear to be counter-intuitive. For example, if the **EQ** jump token is used, the actual instruction assembled is a **JNE** (jump if not equal). The reason for this is simply that the assembler writes the code such that the jump is taken if the condition is *false*:

Forth Assembly Style	Actual Assembled Code	Comment
<b>R0 R0 MOV,</b>	MOV R0,R0	\ check R0
<b>EQ IF,</b>	JNE xxx	\ jump if R0 < 0
<b>R2 INCT,</b>	INCT R2	\ else add 2 to R2
<b>ENDIF,</b>		
<b>R1 INC,</b>	INC R1	\ then increment R1



As can be seen, if the condition is false the code will jump *over* the **IF** block. The **ENDIF**, construct causes the assembler to calculate the jump offset and complete the assembly of the jump instruction. Constructs can be nested with no problems. The assembler will track and assemble the appropriate jump instructions with the correct offsets automatically.

## 7.1 - Assembly Example for Structured Constructs

The following example is designed to show how these jump tokens and structured constructs are used:

Forth Assembly Code	9900 Assembly Code	Comment
<b>( GENERALISED SHIFTER)</b>	* GENERALISED SHIFTER	
<b>ASM: SHIFT ( s v - v)</b>	DEF SHIFT	\ begin assembly word
<b>*SP R0 MOV,</b>	SHIFT MOV *SP,R0	\ get value to shift from stack into R0
<b>SP DECT,</b>	DECT SP	\ reduce stack pointer
<b>R0 R0 MOV,</b>	MOV R0,R0	\ check if value is 0
<b>NE IF,</b>	JEQ L3	\ just exit if yes
<b>*SP R1 MOV,</b>	MOV *SP,R1	\ get shift count from stack into R1
<b>GTE IF,</b>	JLT L1	\ if shift count is positive...
<b>R1 0 SLA,</b>	SLA R1,0	\ ... then shift to the left
<b>ELSE,</b>	JMP L2	\ otherwise...
<b>R1 0 SRL,</b>	L1 SRL R1,0	\ ... shift to the right
<b>ENDIF,</b>		
<b>R1 *SP MOV,</b>	L2 MOV R1,*SP	\ move value back to the stack
<b>ENDIF,</b>		
<b>;ASM</b>	L3 B *NEXT	\ exit back to TurboForth

Note: The structured words shown above *do not* check to ensure that the jump target is within range (+127,-128 words).

## 8 - Test Code

The following routines demonstrate use of the assembler. Each example increases in complexity in terms of exercising the abilities and facilities of the assembler.

### 8.1 - Add two numbers

The following code takes two numbers from the stack, adds them, and places the result on the stack. Functionally, it is equivalent to the Forth *code word* `+` though in practice, the code is slightly different.

Forth Assembly Code	Comment
<b>ASM: ADD ( n1 n2 - n1+n2)</b>	\ begin definition of assembly language word "ADD"
<b>*SP 0 MOV,</b>	\ get n2 in R0
<b>SP DECT,</b>	\ reduce stack pointer (now pointing at n1)
<b>*SP R1 MOV,</b>	\ get n1 in R1
<b>R0 R1 A,</b>	\ add n1 & n2 (result in R1)
<b>R1 *SP MOV,</b>	\ push result to stack
<b>;ASM</b>	\ end definition of assembly language word

To test, simply type:

```
100 99 ADD .
```

### 8.2 - Fill screen with a character

The following program takes an ASCII code from the stack, and fills the screen with it. The program directly sets the VDP Write Address and writes to the VDP write register. Note the use of the **BEGIN**, ... **UNTIL**, looping construct, which removes the requirement for a label. Also note the use of the **EQ** jump token. The code will loop *until* the EQ bit in the 9900 status register is set. Jump tokens are discussed in section 7, page 11, Assembler Jump Tokens.

See the next page for the code.

Code	Comment
<b>\$8C02 CONSTANT VDP A</b>	\ address of VDP address register
<b>\$8C00 CONSTANT VDP W</b>	\ address of VDP write register
<b>ASM: WIPE ( ascii --)</b>	\ create assembly language word “WIPE”
<b>R0 CLR,</b>	\ screen address 0
<b>R2 960 LI,</b>	\ counter
<b>*SP SWPB,</b>	\ get ASCII value on stack in high byte
<b>BEGIN,</b>	\ begin a loop
<b>R0 SWPB,</b>	\ get address low byte
<b>R0 VDP A @@ MOV B,</b>	\ write low byte to address register
<b>R0 SWPB,</b>	\ get address high byte
<b>R0 VDP A @@ MOV B,</b>	\ write hi byte to address register
<b>*SP VDP W @@ MOV B,</b>	\ write to VDP write register
<b>R0 INC,</b>	\ move to next address
<b>R2 DEC,</b>	\ decrement counter
<b>EQ UNTIL,</b>	\ repeat loop if R2 is not 0
<b>SP DECT,</b>	\ remove ASCII value from the stack
<b>;ASM</b>	\ end definition of assembly language word

To test, type:

**65 WIPE**

to fill the screen with A's

**42 WIPE**

to fill the screen with \* symbols, etc.

## 8.3 - Nested Delay

The following code takes a value from the stack, and uses it as part of the outer loop value in a nested delay. Nested delay loops are used to produce long delays.

The `BEGIN, ... WHILE, ... REPEAT,` loop construct in the assembler works in the same way as in high-level Forth: The loop starts at `BEGIN,`. At this point, a condition should be evaluated, and the result fed to `WHILE,`. *While* the condition is *true*, execution will continue with the code immediately following `WHILE,` and will loop back to the code immediately after `BEGIN,` upon encountering `REPEAT..`

If the condition fed to `WHILE,` evaluates to *false*, the code jumps *out* of the loop, running the code immediately after `REPEAT,`.

Code	Comment
<b>ASM: DELAY ( delay --)</b>	\ begin assembly language word "DELAY"
<b>*SP R0 MOV,</b>	\ get loop factor from the stack
<b>SP DECT,</b>	\ reduce stack pointer
<b>BEGIN,</b>	\ begin a loop
<b>R0 R0 MOV,</b>	\ test R0 for 0
<b>NE WHILE,</b>	\ if not equal to zero then proceed, otherwise exit
<b>R1 \$FFFF LI,</b>	\ load R1 with >FFFF
<b>BEGIN,</b>	\ begin another loop
<b>R1 DEC,</b>	\ decrement R1...
<b>EQ UNTIL,</b>	\ ...and continue to do so until R1=0
<b>R0 DEC,</b>	\ decrement R0
<b>REPEAT,</b>	\ jump back to the code immediately after BEGIN,
<b>;ASM</b>	\ exit back to TurboForth

Note the `BEGIN, ... UNTIL,` loop nested inside the `BEGIN, ... REPEAT,` loop. The assembler compiles the appropriate jump instructions and calculates the jump offsets with no intervention (or thought) required by the programmer.

## 9 - General Usage Notes

The following notes are offered to assist the Forth Assembly language developer. As will be seen, integrating assembly language sub-routines into Forth applications is no more complex than referencing the assembly language sub-routine by name. Data can also be passed between Forth and assembly via the Forth data stack with ease, as shown.

### 9.1 - Using Your Own Workspace

It may be desirable to use your own workspace registers for some routines. TurboForth is quite economical in its use of registers; it uses R3, R4, R5 and R12 for its own use – all other registers are available. However, where this is inconvenient you can use your own workspace by `ALLOC`ing data space within the dictionary, and using this space in your assembly language program:

Forth/Assembly Code	Comment
<b>CREATE WORKSPACE</b>	\ create an entry in the dictionary
<b>32 CHARS ALLOT</b>	\ reserve 32 bytes of dictionary space
<b>ASM: TEST</b>	\ create an assembly word called TEST
<b>WORKSPACE LWPI ,</b>	\ set CPU register workspace
...	
...	
...	
<b>\$A012 @@ BLWP ,</b>	\ restore TurboForth workspace and exit
<b>;ASM</b>	

#### 9.1.1 - Restoring TurboForth's Workspace

When changing the workspace, it is essential to change the workspace back again before your machine code routine exits.

In TurboForth versions 1.2 and greater, it is possible to change TurboForth's workspace. Therefore it is recommended, in the interests of writing portable, well-behaved code, to save the workspace pointer address (because you can't necessarily guarantee that TurboForth's workspace is where you think it is) and restore it prior to returning to the TurboForth environment. In practice this is more complex than it might first appear, since the only machine code instruction that directly changes the workspace pointer is `LWPI` which requires an immediate operand. This can lead to writing self-modifying code which can get rather messy and difficult to maintain.

To solve this problem, TurboForth V1.2 has introduced a vector at address `$A012` (this address is fixed and will not change in subsequent releases) which allows one to restore TurboForth's workspace and exit back to TurboForth with a single `BLWP` instruction, as shown in the example code in section 9.1 above.

A side-effect of the `BLWP` instruction is that registers R13, R14 & R15 of TurboForth's workspace are changed, but TurboForth is not affected by this, and in some instances it can be useful since the



previous execution context is saved in these registers as follows:

- R13 – Previous workspace address
- R14 – Previous program counter address
- R15 – Previous value of status register

## 9.2 - Changing TurboForth's Workspace

By default, TurboForth's workspace is at \$8300 to \$831F inclusive, and does not change. This address is in fast 16-bit wide PAD ram. It may be desirable to change TurboForth's workspace so that assembly subroutines can use this space instead.

Introduced in version 1.2, it is now possible to change Turbo's workspace in order to maximise the available space in PAD ram for other uses.

Note that if TurboForth's workspace is moved into 8-bit CPU RAM then TurboForth performance will degrade slightly as two address and data bus cycles are required to read/write CPU RAM.

To change the workspace, R3, R4, R5 and R12 of the current workspace must be copied to their respective positions in the new workspace, a copy of the address of the workspace must be saved to >A012 and finally, the new workspace must be loaded using a LWPI instruction.

The following assembly code (in Forth assembler) provides a word, called TFWS which changes the workspace address from a value passed on the stack:

Forth/Assembly Code	Comment
<b>ASM: TFWS ( ADDR --)</b>	\ begin definition
<b>*SP+ R1 MOV,</b>	\ get new ws address
<b>R1 \$A012 @@ MOV,</b>	\ load into ws restore vector
<b>R1 R2 MOV,</b>	\ copy addr
<b>R2 6 AI,</b>	\ point to r3 in target ws
<b>R3 R2 *+ MOV,</b>	\ copy program counter
<b>R4 R2 *+ MOV,</b>	\ data stack pointer
<b>R5 R2 *+ MOV,</b>	\ copy return stack pointer
<b>R2 12 AI,</b>	\ point to r12 in target ws
<b>R12 R2 *+ MOV,</b>	\ copy NEXT pointer
<b>R0 \$02E0 LI,</b>	\ LWPI op-code
<b>R2 \$045C LI,</b>	\ NEXT op-code
<b>R0 B,</b>	\ execute code in this workspace
<b>;ASM</b>	\ end definition

## 9.3 - Calling Other Code Words

Due to the ITC (Indirect Threaded Code) system employed in TurboForth (and indeed, most 16-bit Forth systems) it is not possible to call other *code words* directly from *inside* another code word. The reason for this is that the *start* of an ASM definition does not contain machine code, therefore you cannot branch to it. It actually contains a pointer to the machine code, which begins exactly 2 bytes after.

For example:

```

ASM: MULTIPLY
  R1 R8 MPY,          \ multiply r1 by r8
  R11 ** B,           \ return to the routine that called us
;ASM

ASM: DO-WORK
  R1 200 LI,          \ load R1
  R8 4 LI,            \ load R8
  MULTIPLY @@ BL,     \ call the MULTIPLY sub-routine
;ASM                  \ exit to Forth

```

The above example will not work for two reasons:

- MULTIPLY does not point to the first executable machine code instruction in the word MULTIPLY. It points to the *pointer* that points to the machine code of MULTIPLY.
- The reference to MULTIPLY in DO-WORK causes MULTIPLY to be executed, it is not supplied as an argument to the BL instruction.

In order for the above code to work, the following changes need to be made in DO-WORK:

- MULTIPLY must be ticked in order to get its address on the stack, to be supplied to BL,
- An offset of 2 must be added to the address of MULTIPLY to account for the pointer.

Therefore, the correct code would be as follows:

```

ASM: DO-WORK
  R1 200 LI,          \ load R1
  R8 4 LI,            \ load R8
  ' MULTIPLY 2+ @@ BL, \ call the MULTIPLY sub-routine
;ASM                  \ exit to Forth

```

However, the above is rather messy. Could there be a better solution? The answer is yes.

The solution is not to use BL (branch and link) to 'chain' sub-routines together. Rather, use a high-level Forth word to 'glue' your assembly language sub-routines together. Using this approach, the "pointer problem" is completely avoided, and actually adds a significant benefit, as it allows assembly language words to be nested easily without any additional code, since the Forth system takes care of nesting and de-nesting via the return stack (assembly language programmers will be used to the 'problem' of having to save the contents of R11 if wishing to nest calls via the BL instruction).

The following section looks at how high-level definitions can be leveraged to allow assembly language words to use each other effectively and cleanly.

### 9.3.1 - Using High-level Definitions to Glue Assembly Words Together

The above solution, whilst it would work, is rather messy and fraught with pitfalls. A much better solution is to write your assembly sub-routines individually, just like we did with MULTIPLY and DO-WORK, but to use Forth colon definitions to 'glue' them together. Using this technique, assembly words can communicate with each other either via the stack, or by declaring their own workspace (see section 9.1).

Let us look at a more useful example. In this example, we wish to create an assembly routine to fetch a random number from the VDP interrupt timer at >8379 and make it available to us in the TurboForth environment by pushing the value to the data stack. However, we will separate the fetching of the random number, and the pushing to stack into separate routines, since the action of pushing a value to the data stack could be used in multiple places, therefore it makes sense to 'factor' it into its own sub-routine.

First, the routine to get a number from the VDP interrupt timer:

```

ASM: RND
    2 LIMI,           \ enable interrupts
    0 LIMI,           \ disable interrupts
    R0 CLR,           \ clear R0
    $8379 @@ R0 MOVB, \ move random number (0 to 255) to R0
    R1 SWPB,          \ move value to the low byte in R0
;ASM

```

Next, the routine to push a value in R0 to the stack. Since this is a subroutine, we will also document the code:

```

ASM: PUSH-R0 ( - n)
    \ takes a value in R0 and pushes it to the data stack
    SP INCT,1      \ make space on the return stack
    R0 *SP MOV,      \ move the value in R0 to the stack
;ASM

```

We can now use high-level Forth to get a random number, using these two routines:

```

: GET-RND ( - n )  RND PUSH-R0 ;

```

This can be tested with:

```

GET-RND .

```

#### Caution:

This particular example exposes the programmer to a certain amount of risk. The issue is that data is left in R0 by RND for PUSH-R0 to use later on. In this *particular* example, there is no risk, since PUSH-R0 is called directly after RND in the Forth word GET-RND. As it happens, the TurboForth inner interpreter does not use R0, so the data stored in R0 remains untouched. However, there is *no* such guarantee if another resident TurboForth word is called "in-between" calls to RND and PUSH-R0. For example:

```

: GET-RND ( - n)  RND  PAGE  PUSH-R0 ;

```

---

<sup>1</sup> Note: In TurboForth version 1.2 the stack directions have been reversed. Therefore DECT, should be used rather than INCT,.

This example calls `PAGE` (which clears the screen) before calling `PUSH-R0`. It so happens that `PAGE` uses `R0`, so the data left by `RND` is overwritten.

Therefore, if your assembly language sub-routines will be “bridged” using high-level Forth colon definitions, *and* you wish to pass data between them you must either provide a separate workspace for them, or use the data stack.

## 9.4 - Mixing Forth and Assembly Language in an ASM: Definition

It might not be immediately obvious, but when you are writing assembly language instructions in TurboForth, you are actually writing Forth code. Words such as `MOV`, `INC`, `DECT`, `ABS`, etc. are all high level Forth colon definitions that simply compile op-codes to the current position in memory.

It is easy to fall into the “trap” of thinking that when you type `ASM:` to begin an assembly language word that you change into some special 'mode' that leaves the Forth environment behind, and enters an assembly language environment. This actually isn't true at all. You are still “in Forth” with all of the facilities of the Forth system and environment at your disposal.

The stack is heavily used by the assembler, just as it is in Forth. Let us look at an example:

```
R1 $1234 LI,
```

Here, `R1` is loaded with the value 1234 hex. However, it is not immediately obvious but what happened is this:

- `R1` is a high-level Forth word, which pushes 1 to the stack;
- `$1234` simply goes on the stack, as does any number in Forth;
- `LI,` is a high-level Forth word that takes two numbers from the stack, and encodes them into an `LI` op-code.

What this means is that you can use Forth to enhance your assembly language code, with practically unlimited power at your disposal.

For example, the above code might theoretically be replaced with something like:

```
R1 GET-VALUE LI,
```

Where `GET-VALUE` is a standard Forth colon definition that pushes some value to the stack. In fact, we could use our earlier example in our assembly language code:

```
R1 GET-RND LI,
```

Here, the word we previously built to return a random number to the stack is directly used in an *assembly language* definition. What would actually happen here is that at *assembly time* (when the `LI` instruction is physically compiled/assembled to memory) a random number would be chosen (using previously assembled assembly language routines to generate the number) and pushed to the data stack. `LI`, would then simply use that value and assemble it into the `LI` instruction.

We could take this a step further. Let us assume we have built a Forth application that relies on a number of assembly routines to do work for us. In those routines, we have used `R13` as a holding register (a routine places data in `R13` for another routine to access later).

However, at some point, it becomes inconvenient to use R13, and we instead wish to use R14. We can do this in one simple action, by using a `CONSTANT`:

```
14 CONSTANT HREG \ holding register is R14
ASM: SOME-THING
    HREG GET-RND LI,
    ...
    ...
;ASM
```

Here, instead of referencing R14 throughout our assembly code, we reference HREG instead. Thus, if we want to change the holding register designation, we only have to change the *constant, in one place*. The assembly language source code remains unchanged.

Of course, you could use extremely complex Forth words in your assembly code that carry out complex calculations, or fetch some data from a disk block, or a disk file – you have the complete functionality of Forth at your disposal.

## 9.5 - Accessing Forth Strings from Assembly Language

At some point you may wish to access a string from inside your assembly language routine, to read the string and perhaps modify it in some way. Here, we will discuss an assembly language routine that can change all lower case characters in a string to upper case. The string will be declared “Forth side” using Forth nomenclature and we will call an assembly language routine to do the case conversion for us.

For example, when the code is complete, it will be possible to execute:

```
S" this is an upper case string!" >UCASE TYPE
```

and see the output

```
THIS IS AN UPPER CASE STRING!
```

First, we need to discuss strings in TurboForth, and indeed Forth in general. Strings are a little strange in Forth, because you can 'execute' them. Indeed, this is how one gains access to strings in Forth. When one “executes” a string, the address, and the length of the string are pushed to the stack. Once we have this information, we can access/manipulate the string in any way.

Type this into TurboForth as an example:

```
S" Hello, world!"
```

You will see two values pushed to the stack, which you can examine with `.S`. We can feed this directly into `TYPE` and have the string echoed back to us:

```
S" I AM FORTH" TYPE
```

Indeed, we can place a string in a colon definition, and it will behave the same way:

```
: STRING S" Hello Mother!" ;
STRING TYPE
```

The fact that we get the address of the string, and its length is very useful to us, and we will make use of this in the following assembly language sub-routine.

The following sub-routine expects the address and the length of the string to be on the stack before the routine is called (obviously). It will leave the address and the length on the stack, unchanged, so that other words such as TYPE may immediately access the string after processing. Only characters between ASCII codes 97 (a) and 122 (z) will be modified. Any other characters in the string are left unchanged.

Subroutine follows on the next page:

Code	Comment
ASM: >UCASE	\ declare assembly language word >UCASE
*SP R0 MOV,	\ get the string length from stack into R0
-2 SP ( ) R1 MOV,	\ get the string address in R1 [MOV @-2(SP),R1]
R2 CLR,	\ we'll use r2 for byte-wise operations, so clear it
BEGIN, ←	\ begin a loop
R1 ** R2 MOVB,	\ get a character from the string into r2
R2 CHAR a 256 * CI,	\ compare the upper byte to 'a'
HE IF,	\ if higher or equal to 'a' then...
R2 CHAR z 256 * CI,	\ ...compare the upper byte to 'z'
LE IF,	\ if lower or equal to 'z' then we are in range, so...
R2 -32 256 * AI,	\ ...subtract 32 from the value in the upper byte
ENDIF,	
R2 R1 *+ MOVB,	\ move the byte back to memory
R0 DEC,	\ decrement the length counter
EQ UNTIL, —	\ loop back to BEGIN if R0 is not 0
;ASM	\ exit back to Forth

Assemble the above code and test with:

**S" hello mother! How are you today?" >UCASE TYPE**

The above would perhaps look scarcely familiar, even to the experienced assembly language programmer; however, the above code is an excellent example of leveraging the full power of the combination of Forth and assembly code.

As can be seen, we have actually applied high-level programming constructs such as looping, IF and ENDIF to low-level assembly code. Note that there are no labels used in the above code, and no jump instructions (they are compiled into the code for us by the assembler, “behind the scenes”). In addition, we have mixed Forth code into the assembly source code to do some grunt-work for us at assembly time.

Whilst the RPN nature of the code is initially confusing, it actually becomes quite natural quite quickly (this author has been using the assembler less than 24 hours at the time of writing, and is having no difficulty making the mental adjustment), especially if one uses R notation for register names, and uses the Wycove addressing notation, which aligns much better to 9900 assembly than the original TI-Forth addressing mode notation.

### 9.5.1 - Code Breakdown

We will now examine the above code line-by-line and discuss how it works, and the techniques employed. Standard 9900 assembler is also shown so that one can compare the difference in syntax.

#### ASM: >UCASE

The above begins a new assembly language definition called >UCASE. In Forth parlance, where a > precedes a word, it generally means “to”. Thus this word could be called “to upper case”.

```
*SP R0 MOV,      [ MOV *SP,R0      ]
```

Here we get the contents of the top word of the stack (the length of the string) into register R0. SP is a 'special' register name, and is equivalent to R4. See section 3, TurboForth's Workspace Registers, on page 4 for a description of special register names.

```
-2 SP () R1 MOV,   [ MOV @-2(SP),R1   ]
```

Here we get the *second* value on the stack (the value “underneath” the length) which is the address of the string.

```
R2 CLR,           [ CLR R2           ]
```

Since we will be working with single bytes, we need a register to hold them. Bytes are always operated on in the upper (high) byte of a register, so we zero this register before we start.

#### BEGIN,

We now begin a loop. This is simply a 'return' point to return to if the exit condition of the loop is not satisfied. In practice, the code loops to the instruction immediately *after* BEGIN,. BEGIN, itself is a place holder.

```
R1 ** R2 MOVB,     [ MOVB *R1,R2     ]
```

Here, we read a byte from the string, and place it into the upper byte of R2.

```
R2 CHAR a 256 * CI, [ CI R2,'a'*256   ]
```

At this point, we have mixed Forth and assembly together. We want the ASCII code of 'a' (97) in the *upper* byte, because the character we are examining is in the upper byte. Therefore we actually want 97\*256. However, rather than calculate 97\*256 (24832 decimal) we ask Forth to work it out for us by using the CHAR command. CHAR pushes the ASCII character of the character immediately following it to the stack. We then multiply it by 256. Note that this calculation takes place at *assembly* time, not run-time, so what actually gets assembled is CI R2,24832 (or, in hex, CI R2,>6100)



```
HE IF, [ JL xxxx ]
```

If the character is higher or equal to >6100 (which is ASCII 97 in the upper byte) then we enter this IF block. Other wise we skip it completely.

```
R2 CHAR z 256 * CI, [ CI R2, 'z'*256 ]
```

If we do enter the IF block, then we check to see if the character is less than or equal to a 'z' character. Again, we get Forth to calculate the value of ASCII code 112\*256, using CHAR.

We could have instead written it as R2 112 256 \* CI, however the “112” doesn't really show any context. By using CHAR z it is much clearer that we are performing a comparison to the 'z' character.

```
LE IF, [ JH xxxx ]
```

Here, if the character code *is* less than or equal to 'z' then we know that our character is in the target a-z range, so we enter the IF block. Otherwise, the character lies outside the range, and we skip the IF block.

```
R2 -32 256 * AI, [ AI R2, -32*256 ]
```

If the character is in range, then we subtract 32 from its value to convert it to upper case. Since the value is in the upper byte, we need 32\*256. Finally, since there is no “subtract immediate” instruction in 9900 assembly language, we use the AI (add immediate) instruction, so we need to add -32\*256 to the value in R2.

```
ENDIF, [ none ]
```

This marks the end of the 'inner' IF block (the part that executes if the character is in range). It generates no code, it simply resolves the offset of the JH instruction, allowing the code following the JH instruction to be skipped if the condition is not met.

```
ENDIF, [ none ]
```

Again, this generates no code, it simply resolves the offset of the JL instruction (the section that checks for 'a' or greater) allowing the code to jump over if the condition is not met.

```
R2 R1 *+ MOVB, [ MOVB R2, *R1+ ]
```

Here we write the character value (which may, or may not have been modified) back to the same address in the buffer. We also increment the buffer pointer address.

```
R0 DEC, [ DEC R0 ]
```

We then decrement R0 (length)

**EQ UNTIL,**                    **[ JNE xxxx        ]**

When R0 reaches 0 the EQ flag will be set in the status register. The EQ UNTIL, causes the code to loop back (to the instruction following BEGIN,) *until* the EQ flag is on (i.e. until R0=0)

**;ASM**                            **[ B \*NEXT        ]**

This ends the assembly language definition and assembles the appropriate exit code (a B\* R12) instruction into the definition, which returns back to the Forth system.

## 10 - TurboForth Assembler Code

The following represents the code for the TurboForth Assembler. The code here is shown starting on block 9, but in practice it can be located anywhere. The code occupies 5 blocks in source-code form, and compiles into 2.7K of object code when loaded.

### Block 9

```
00: .( Loading assembler V1.0...)
01:  0 CONSTANT  R0    1 CONSTANT  R1
02:  2 CONSTANT  R2    3 CONSTANT  R3
03:  4 CONSTANT  R4    5 CONSTANT  R5
04:  6 CONSTANT  R6    7 CONSTANT  R7
05:  8 CONSTANT  R8    9 CONSTANT  R9
06: 10 CONSTANT R10   11 CONSTANT R11
07: 12 CONSTANT R12   13 CONSTANT R13
08: 14 CONSTANT R14   15 CONSTANT R15
09: CHAR . EMIT -->
10:
11:
12:
13:
14:
15:
```

### **Block 10**

```
00: : ASM:  BL WORD HEADER HERE 2+ , LATEST @ HIDDEN ;
01: : ;ASM  $045C , LATEST @ HIDDEN ;
02: : ?PAIRS XOR ABORT" Conditionals not paired" ;
03: : ERROR CR ." Block " BLK @ . ." line " >IN @ 64 / .
04: 1 2 ?PAIRS ;
05: BASE @ >R  HEX
06: : GOP' OVER DUP 1F > SWAP 30 < AND IF + , , ELSE + , THEN ;
07: : GOP CREATE , DOES> @ GOP' ;
08: 0440 GOP B,      0680 GOP BL,      0400 GOP BLWP,
09: 04C0 GOP CLR,    0700 GOP SETO, 0540 GOP INV,
10: 0500 GOP NEG,    0740 GOP ABS,    06C0 GOP SWPB,
11: 0580 GOP INC,    05C0 GOP INCT, 0600 GOP DEC,
12: 0640 GOP DECT, 0480 GOP X,
13: CHAR . EMIT -->
14:
15:
```

### **Block 11**

```
00: : GROP CREATE , DOES> @ SWAP 40 * + GOP' ;
01: 2000 GROP COC,   2400 GROP CZC,   2800 GROP XOR,
02: 3800 GROP MPY,   3C00 GROP DIV,   2C00 GROP XOP,
03:
04: : GGOP CREATE , DOES> @ SWAP DUP DUP 1F > SWAP 30 < AND
05: IF 40 * + SWAP >R GOP' R> , ELSE 40 * + GOP' THEN ;
06: A000 GGOP A,     B000 GGOP AB,     8000 GGOP CMP, 9000 GGOP CB,
07: 6000 GGOP S,     7000 GGOP SB,     E000 GGOP SOC, F000 GGOP SOCB,
08: 4000 GGOP SZC,   5000 GGOP SZCB,   C000 GGOP MOV, D000 GGOP MOV,
09:
10: : OOP CREATE , DOES> @ , ;
11: 0340 OOP IDLE,    0360 OOP RSET,    03C0 OOP CKOF,
12: 03A0 OOP CKON,    03E0 OOP LREX,    0380 OOP RTWP,
13:
14: CHAR . EMIT -->
15:
```

**Block 12**

```

00: : ROP CREATE , DOES> @ + , ; 02C0 ROP STST, 02A0 ROP STWP,
01: : IOP CREATE , DOES> @ , , ; 02E0 IOP LWPI, 0300 IOP LIMT,
02: : RIOP CREATE , DOES> @ ROT + , , ;
03: 0220 RIOP AI, 0240 RIOP ANDI, 0280 RIOP CI, 0200 RIOP LI,
04: 0260 RIOP ORI,
05: : RCOP CREATE , DOES> @ SWAP 10 * + + , ;
06: 0A00 RCOP SLA, 0800 RCOP SRA, 0B00 RCOP SRC, 0900 RCOP SRL,
07:
08: : DOP CREATE , DOES> @ SWAP 00FF AND OR , ;
09: 1300 DOP JEQ, 1500 DOP JGT, 1B00 DOP JH, 1400 DOP JHE,
10: 1A00 DOP JL, 1200 DOP JLE, 1100 DOP JLT, 1000 DOP JMP,
11: 1700 DOP JNC, 1600 DOP JNE, 1900 DOP JNO, 1800 DOP JOC,
12: 1C00 DOP JOP, 1D00 DOP SBO, 1E00 DOP SBZ, 1F00 DOP TB,
13:
14: : GCOP CREATE , DOES> @ SWAP 000F AND 40 * + GOP' ;
15: 3000 GCOP LDCR, 3400 GCOP STCR, CHAR . EMIT -->

```

**Block 13**

```

00: : @() 020 ; : *? 010 + ; : *?+ 030 + ;
01: : @(?) 020 + ; : W 06 ; : @ (W) W @ (?) ;
02: : *W W *? ; : *W+ W *?+ ; : RP 05 ;
03: : @ (RP) RP @ (?) ; : *RP RP *? ; : *RP+ RP *?+ ;
04: : IP 03 ; : @ (IP) IP @ (?) ; : *IP IP *? ;
05: : *IP+ IP *?+ ; : SP 04 ; : @ (SP) SP @ (?) ;
06: : *SP SP *? ; : *SP+ SP *?+ ; : NXT 0C ;
07: : *NXT+ NXT *?+ ; : *NXT NXT *? ; : @ (NXT) NXT @ (?) ;
08: : GTE 1 ; : HI 2 ; : NE 3 ; : LO 4 ; : LTE 5 ; : EQ 6 ;
09: : OC 7 ; : NC 8 ; : OO 9 ; : HE 0A ; : LE 0B ; : NP 0C ;
10: : LT 0D ; : GT 0E ; : NO 0F ; : OP 10 ;
11: : CJMP CASE LT OF 1101 , 0 ENDOF GT OF 1501 , 0 ENDOF
12: NO OF 1901 , 0 ENDOF OP OF 1C01 , 0 ENDOF
13: DUP 0< OVER 10 > OR IF ERROR THEN DUP
14: ENDCASE 100 * 1000 + , ;
15: CHAR . EMIT -->

```

#### **Block 14**

```
00: : IF, CJMP HERE 2- 42 ; IMMEDIATE
01: : ENDIF, 42 ?PAIRS HERE OVER - 2- 2 / SWAP 1+ C! ; IMMEDIATE
02: : ELSE, 42 ?PAIRS 0 CJMP HERE 2- SWAP 42 [COMPILE] ENDIF,
03: 42 ; IMMEDIATE
04: : BEGIN, HERE 41 ; IMMEDIATE
05: : UNTIL, SWAP 41 ?PAIRS CJMP HERE - 2 / 00FF AND HERE 1- C!
06: ; IMMEDIATE
07: : AGAIN, 0 [COMPILE] UNTIL, ; IMMEDIATE
08: : REPEAT, >R >R [COMPILE] AGAIN, R> R> 2- [COMPILE] ENDIF,
09: ; IMMEDIATE
10: : WHILE, [COMPILE] IF, 2+ ; IMMEDIATE
11: \ Wycove assembler register syntax:
12: : @@ @() ; : ** *? ; : *+ *?+ ; : () @(?) ;
13: : RT R11 ** B, ; R> BASE !
14: XY? NIP 0 SWAP GOTOXY .( Assembler loaded. Usage: )
15: .( ASM: TEST 1 2 MOV, 3 4 A, ;ASM)
```

## **10.1 - Errors Corrected from the Original Source Code**

References to ?EXEC in the original TI-Forth code have been removed as **?EXEC** is not used in F83.

There are erroneous [COMPILE] CJMP sequences in IF, ELSE, and UNTIL, which are not required as CJMP is not immediate. These have also been removed.

## 11 - Conclusion

The ability to write assembly language code directly from within TurboForth, or load and assemble assembly language source code directly from blocks represents a major leap forward in the power and versatility of TurboForth. High level code constructs such as looping and IF...THEN blocks allow one to apply high-level programming paradigms to low level assembly code, removing completely the requirement for error-prone labels.

Further, the ability to mix, completely freely, assembly language mnemonics and high-level Forth code in an assembly definition provides unprecedented power and flexibility. High-level Forth words can be used to add complex macro like functionality to the assembly language programmer.

This author has been truly delighted and empowered by the extra flexibility afforded by the assembler, and hopes that other users of TurboForth will give it a try and 'unleash' the power of the Forth and assembly language.