Appendix A

Glossary of TILE Words

This glossary includes a complete description of the entire TILE word set together with information regarding stack arguments, source code (where applicable) and any other relevant information.

The following abbreviations are used to characterise the words

I	An immediate word.
C	May only be used in compilation mode.
E	May only be used in execute mode.
P	Code primitive.
PRE	Defined in the 'prelude' file.

In addition to the abbreviations specified above, there are also a number of stack related abbreviations used in describing the type of stack arguments consumed or produced by a word. The conventional format of stack abbreviations is (**stack before** -- **stack after**) thus a word specified by the abbreviation (**n addr** -- **c**) consumes a single length signed number and an address from the parameter stack respectively, and produces a 7-bit character value.

n	Single-length signed number.
d	Double-length signed number.
u	single-length unsigned number.
ud	Double-length unsigned number.
c	7-bit character value.
b	8-bit byte.
f	Boolean flag; or; True (-1) False (0)
addr	Address
nfa	Name field address.
lfa	Link field address.
cfa	Code field address.
pfa	Parameter field address.

The glossary is ordered according to **ASCII** value of the words with the smallest value first and the largest value last. To aid in the location of a word the **ASCII** alphabet is printed at the bottom of each page.

! "store" P
(n addr --)
Stores n at addr.

"sharp" PRE
(d1 -- d2)
The rightmost digit of d1 is converted to an ASCII character according to the value of **BASE** and appended to a pictured (formatted) output string (for subsequent output by **type**). d2 is a number with the remaining digits and is maintained for further processing.Used between <# and #>. A word to display a signed double-length number of pence as pounds and pence could be defined thus :-

```
: .pounds ( d -- )
    swap over dabs <# # # 46 hold #s 36 hold rot sign #> type ;
```

See also <# , # , #s , **hold** , **sign** , #> .

#> "sharp-greater" PRE
(d -- addr n)
Ends conversion of a number to a pictured (formatted) output string. addr is the address of the resulting output string and n is the number of characters in it. addr and n together are suitable for use by **type**. See # for an example.See also <# , #, #s , **hold** , **sign**.

#s "sharp-s" PRE
(d -- 0 0)
Converts d to a pictured (formatted) output string, digit by digit, according to the current value in **BASE**. See # for an example. A single zero is placed in the output string if d was initially zero. Used between <# and #>.

' "tick" PRE , I
(-- cfa)
Used in the form :-

```
' <name>
```

returns the code filed address of <name>, or false if <name> is not in the vocabulary list.

(+loop) "brackets-plus_loop" P
(n --)
When used in the form :-

```
: < name > ... compile (+loop) ... ;
```

a loop termination operation is compiled into a colon-definition. When executed, the top single-length value of the return stack is incremented by n, and compared with the second single-length value on the return stack. If n is positive and the second is greater than the first, or n is negative and second is smaller than the first then a relative backward jump is made, otherwise execution continues with the word compiled immediately after the branch address that follows **(+loop)**. Each occurrence of **(+loop)** should be accompanied with a **(do)** to prevent the return stack from becoming corrupted.

(.") "brackets-dot-quote" P
(--)
Displays the counted string that follows immediately after **(.")** and continues execution with the word that follows immediately after the last character.

(:) "bracket-colon" P , E , I
(--)
A defining word that creates a colon-word. Used only in execution mode, in the form :-

(:) < name > ... (;)

creates a dictionary entry for <name> in the current vocabulary and sets **STATE** to compile mode, causing the following words and numbers from the input stream to be compiled. < name > cannot be found in the dictionary until a corresponding **(;)** or **;code** is successfully processed. **(:)** sets the search order so that the first vocabulary to be searched is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. Used in the definition of **:** but contains no error checking.

(;) "brackets-semi-colon" P , I , C
(--)
Stops compilation of a colon-definition. Used in the form :-

(:) < name > ... (;)

enables < name > to be found in the dictionary and sets **STATE** to execute mode. Used in the definition of **;** but contains no error checking.

(do) "brackets-do" P
(n1 n2 --)
When used in the form :-

: < name > ... compile (do) ... ;

a loop initialisation operation is compiled into a colon-definition. When executed, n1 and n2 are transferred to the return stack and used as a loop counter and loop delimiter respectively. Each occurrence of a **(do)** must be accompanied with either a **(loop)** or **(+loop)** to prevent the return stack from becoming unbalanced.

(emit) "brackets-emit" P
(n --)
Sends the character represented by the least significant byte of n to the console but does not increments **OUT.**

(endif) "brackets-end-if" PRE , C
(addr 2 --)
Calculate the relative difference between addr and the next available dictionary location (the value returned by **here**) and store the result at addr. The 2 is used to trap errors within constructs such as **if** , **else** , **then** which use **(endif)**.

(find) "brackets-find" P
(addr1 addr2 -- addr2 False) or (addr1 addr2 -- addr2 nfa True)
Searches the vocabulary starting at addr2 for the counted string at addr2. If no match is found then the address of the counted string together with False (0) are returned. If a match is found then the address of the counted string

together with the name-field address of the matching dictionary entry and True (-1) are retuned.

(loop) "brackets-loop" P
(--)
When used in the form :-

: < name > ... compile (loop) ... ;

a loop termination operation is compiled into a colon-definition. When executed, the top single-length value of the return stack is incremented and compared with the second single-length value on the return stack. If the second is greater than the first then a relative backward jump is made, otherwise execution continues with the word compiled immediately after the branch address that follows **(loop)**. Each occurrence of **(loop)** should be accompanied with a **(do)** to prevent the return stack from becoming corrupted.

(toggle) "brackets-toggle" PRE
(n1 n2 -- n3)
Performs the bit-by-bit toggle of n1 with the bits specified by the single-length mask n2 to produce n3. See also **toggle**.

***** "times" PRE
(n1 n2 -- n3)
Multiplies n1 by n2, giving the product n3. n3 will contain the least significant 16 bits of the product even if arithmetic overflow occurs.

***/** "times-divide" PRE
(n1 n2 n3 -- n4)
Multiplies n1 by n2, divides the result by n3, and leaves the quotient n4. The product of n1 times n2 is maintained as an intermediate double-length number, giving greater precision than the otherwise equivalent sequence.

***/mod** "times-divide-mod" PRE
(n1 n2 n3 -- n4 n5)
Multiplies n1 by n2, divides the result by n3, and leaves the remainder n4 and quotient n5. A double-length intermediate product is used as for */.

+ "plus" P
(n1 n2 -- n3)
Adds n2 to n1 to give the sum n3. n3 contains the least significant 16 bits of the sum even if arithmetic overflow occurs.

++ "plus-plus" PRE
(addr --)
Increment the variable at addr by one.

+ - "plus-minus" PRE
(n1 n2 -- n3)
Returns the absolute value of the signed single-length number n1 if n2 is negative.

+! "plus-store" PRE
(n1 addr --)
Adds n1 to the single-length number at addr, replacing the original value at that address with the sum.

+C! "plus-c-store" PRE
(n1 addr --)
Adds the least significant byte of n1 to the 8 bit byte at addr, replacing the original value at that address with the sum.

+loop "plus-loop" PRE
(n --)
Terminates a do-loop, allowing the loop index to be incremented (or decremented) with values other than 1. Used in the form :-

: <name> ... do ... +loop ;

compiles a do-loop structure. When <name> is executed, **+loop** expects a number(n) on the stack that is used to determine by now much to increment (or decrement) the loop index. The loop is terminated when the index becomes greater than or equal to the limit (in a signed sense). Execution will jump to just after the corresponding **do** if the loop is not terminated an to just after **+loop** if it is. Contrast with **loop**. See also **do**.

, "comma" PRE
(n --)
Stores n into the next available dictionary location (the address returned by **here**) and increments the dictionary pointer by 2 to allocate space for the storage of n. It is said that n is "compiled" at **here**. See also **C,**

- "minus" PRE
(n1 n2 -- n3)
Subtracts n2 from n1, leaving the remainder n3.

-- "minus-minus" PRE
(addr --)
Decrements the variable at addr by one.

-dup "minus-dupe" PRE
(n1 -- n1 n1) or (n1 -- n1)
Duplicates the single-length number n1 only if n1 is not zero, otherwise the stack remains unchanged.

-find "minus-find" PRE
(-- addr False) or (-- addr nfa True)
Used in the form :-

-find < name >

Searches the dictionary chain for <name>. If <name> is not found then the address of <name> in the text-input buffer together with False (0) is returned. If <name> is found the the address of <name> in the text-input buffer, the name-field address of <name> together with True (-1) is returned.

. "dot" PRE
(n --)
Displays the value of n (including a leading minus sign if n is negative), according to the value of **base**, at the current display position, followed by a space.

." "dot-quote" PRE , I , C , E
(--)

Compiles and/or displays text. Used in the form

: <name> ... ." ccc" ... ;

compiles the text ccc following ." up to the delimiter" (double-quote) into the definition of <name>. A blank must follow ." (and is not included in ccc) but a blank need not precede "double-quote",which is a delimiter, not a TILE word. The compiled text will then be displayed when <name> is executed. ." may also be used in execution mode, in which case the text is immediately displayed.

.r "dot-r" PRE
(n1 n2 --)
Displays the value of n1 right justified in a field n2 characters wide according to the value of **BASE**. A leading minus sign is displayed if n1 is negative. It is not an error if the number characters required to display n1 is greater than n2. The results in the number being displayed without a leading space. Also if n2 is less than 1 no leading blanks will be supplied.

/ "divide" PRE
(n1 n2 -- n3)
Divides n1 by n2 and leaves the quotient n3. n3 is rounded toward zero. An error is generated if divisor is zero.

/mod "divide-mod" PRE
(n1 n2 -- n3 n4)
Divides n1 by n2 and leaves the remainder n3 and quotient n4. n4 is rounded toward zero. An error occurs if the divisor is zero.

0< "zero-less" PRE
(n -- f)
Compares n with 0 and returns a true flag if n is less than zero.

0= "zero-equals" PRE
(n -- f)
Compares n with 0 and returns a True (-1) flag if n is zero. See also **not**.

0> "zero-greater" PRE
(n -- f)
Compares n with 0 and returns a True (-1) flag if n is greater than zero.

0branch "zero-branch" P
(f --)
When used in the form :-

: < name > ... compile 0branch ... ;

a conditional branch operation is compiled into a colon-definition. When executed, if a flag on the stack is False (0), the branch is performed as with **branch**. If the flag is True (-1), execution continues with the word compiled immediately after the branch address that follows **0branch**. Not normally used for routine programming, used in the definition of **if** , **while**, and **until**.

1+ "one-plus" PRE
(n1 -- n2)
Adds 1 to n1 to give n2.

1- "one-minus" PRE
(n1 -- n2)
Subtracts 1 from n1 to give n2.

2+ "two-plus" PRE
(n1 -- n2)
Adds 2 to n1 to give n2.

2- "two-minus" PRE
(n1 -- n2)
Subtracts 2 from n1 to give n2.

2dup "two-dupe" PRE
(d -- d d) or (n1 n2 -- n1 n2 n1 n2)
Duplicates the double-length number on the stack or the first and second single-length numbers.

2drop "two-drop" PRE
(d --) or (n1 n2 --)
Removes a double-length number (or two single-length numbers) from the stack.

2over "two-over" PRE
(d1 d2 -- d1 d2 d1) or (n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2)
Copies the second double-length number on the stack to the top of the stack or copies the third and fourth single-length numbers to the top of the stack.

2swap "two-swap" PRE
(d1 d2 -- d2 d1) or (n1 n2 n3 n4 -- n3 n4 n1 n2)
Swaps the top two double-length numbers on the stack or swaps the first and the second second single-length numbers with the third and fourth.

: "colon" PRE , I , E
(--)
A defining word that creates a colon-word. Used only in execution mode, in the form :-

: < name > ... ;

creates a dictionary entry for <name> in the current vocabulary and sets **STATE** to compile mode, causing the following words and numbers from the input stream to be compiled. <name> cannot be found in the dictionary until a corresponding **;** or **;code** is successfully processed. **:** sets the search order so that the first vocabulary to be searched is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. Colon-words are the main means for TILE programming. See also **;** , **;code**.

; "semi-colon" PRE , I , C
(--)
Stops compilation of a colon-definition. Used in the form :-

: < name > ... ;

enables < name > to be found in the dictionary and sets **STATE** to execute mode.

;code "semi-colon-code" PRE , I , C
(--)
A defining-word terminator that may only be used in compilation mode. Used in the form :-

: < namex > ... create ... ;code ... **machine code** ...

stops compilation, terminates the defining word <namex> and compiles **scode**. When <namex> is executed in the form :-

<namex> < name >

to create <nam >, the code-field contents of <name> will be the address of the machine code following the **;code** in <namex>. Thus **;code** permits the create of parent words that create child words whose action is defined by the machine code after **;code** in the definition of <namex>. Analogous in action to **does>**. See also **scode , create , does>**.

< "less-than" P
(n1 n2 -- f)
Compares n1 with n2 and returns a True (-1) flag if n1 is less than n2.

<# "less-sharp" PRE
(--)
Starts conversion of a number to a pictured (formatted) output string. See # for an example. See also **#> , #s , hold , sign .**

<> "not-equal" PRE
(n1 n2 -- ?)
Compares n1 with n2 and returns a True (-1) flag if n1 and n2 are not equal.

<builds "builds" PRE , C
(--)
A defining word to create a dictionary entry. Used in the form :-

<builds < name >

creates a dictionary entry for <name> and allocates space in <names>'s parameter field. When <name> is executed, the value 0 is left on the stack. <builds is used within a colon definition in the form :-

: < name > ... <builds ... does> ... ;

to specify the compile-time action of a defining word. See also **does>**.

= "equals" P
(n1 n2 -- f)
Compares n1 with n2 and returns a True (-1) flag if n1 is equal to n2.

> "greater-than" P
(n1 n2 -- f)
Compares n1 with n2 and returns a True (-1) flag if n1 is greater than n2.