

# StrongForth 3.1 Glossary: forth

**! ( complex address -> 1st -- )**

Store complex at address -> 1st.

**! ( complex dfaddress -> 1st -- )**

Store complex as a complex double-precision floating-point number at dfaddress -> 1st.

**! ( complex sfaddress -> 1st -- )**

Store complex as a complex single-precision floating-point number at sfaddress -> 1st.

**! ( double address -> 1st -- )**

Store double at address -> 1st.

**! ( float address -> 1st -- )**

Store float at address -> 1st.

**! ( float dfaddress -> 1st -- )**

Store float as a double-precision floating-point number at dfaddress -> 1st.

**! ( float sfaddress -> 1st -- )**

Store float as a single-precision floating-point number at sfaddress -> 1st.

**! ( single address -> 1st -- )**

Store single at address -> 1st.

**! ( single caddress -> 1st -- )**

Store single at caddress -> 1st. Only the low-order bits corresponding to character size are transferred.

**" ( "ccc<delimiter>" -- ) compile-only**

strongforth.sf

Compilation: Parse *ccc* delimited by a quote ("). Allot as many characters in the data-space memory space as are required for storing *ccc*. Copy the character string *ccc* to the allotted memory area. Append the runtime semantics given below to the current definition. If the data-space memory space is character aligned when " begins execution, it will remain character aligned when " finishes execution. An ambiguous condition exists if the first unused address of the data-space memory space is not character aligned prior to execution of ". An exception is thrown if the data-space memory space overflows.

Runtime: Place the copied character string as `caddress -> character unsigned` on the stack.

**" ( "ccc<delimiter>" -- caddress -> character unsigned )**

strongforth.sf

Parse `ccc` delimited by a quote (`"`).

`caddress -> character` is the address of the first parsed character within the input buffer and `unsigned` is the length of the parsed string. If the parse area was empty, `unsigned` is zero.

**", ( caddress -> character unsigned -- )**

Reserve space for a string with `unsigned` characters in the default memory space and copy the string `caddress -> character unsigned` in the default memory space. If the default memory space is character aligned when `",` begins execution, it will remain character aligned when `",` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not character aligned prior to execution of `",`. An exception is thrown if the default memory space overflows.

**", ( caddress -> character unsigned memory-space -- )**

Reserve space for a string with `unsigned` characters in `memory-space` and copy the string `caddress -> character unsigned` in `memory-space`. If `memory-space` is character aligned when `",` begins execution, it will remain character aligned when `",` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not character aligned prior to execution of `",`. An exception is thrown if `memory-space` overflows.

**# ( number-double -- 1st )**

strongforth.sf

Divide `number-double` by the current number-conversion radix base giving the quotient `1st` and the remainder `n` (`n` is the least-significant digit of `number-double`). Convert `n` to external form and add the resulting character to the beginning of the pictured numeric output string. An exception is thrown if the transient area used for storing the pictured numeric output overflows.

**#> ( number-double -- caddress -> character unsigned )**

strongforth.sf

Drop `number-double`. Make the pictured numeric output string available as `caddress -> character unsigned`. A program may replace characters within the string.

**#blocks ( -- unsigned )**

block.sf

`unsigned` is the total number of blocks in the block file.

**#friends ( class-attributes -- caddress -> unsigned )**

`caddress -> unsigned` is the address of an unsigned character-size value indicating the length in cells of the friend class table of the class associated with `class-attributes`.

`#friends` is a member of the `class-attributes` class.

**#hold ( -- caddress -> unsigned )**

`caddress -> unsigned` is the address of the offset with respect to `line` where the pictured numeric output string or the next character of a string to be composed starts.

**#locals ( -- caddress -> unsigned )**

`caddress -> unsigned` is the address of an unsigned character-size value indicating the number of cells reserved for locals in the stack frame of the current definition.

**#s ( number-double -- 1st )**

strongforth.sf

Convert one digit of `number-double` according to the rule for `#`. Continue conversion until the quotient is zero. `1st` is zero.

**#vtable ( data-type-attributes -- caddress -> unsigned )**

`caddress -> unsigned` is the address of an unsigned character-size value indicating the length in cells of the virtual method table of the data type associated with `data-type-attributes`. If the data type has no virtual method table, this value is always zero.

`#vtable` is a member of the `data-type-attributes` class.

**' ( "<spaces>name" -- definition )**

Skip leading space delimiters. Parse *name* delimited by a space. Search the context vocabularies for *name* and return its latest occurrence as *definition*. An exception is thrown if *name* is not found.

**'friends ( class-attributes -- address -> address -> class-attributes )**

`address -> address -> class-attributes` is the address of a pointer to the friend class table of the class associated with `class-attributes`.

`'friends` is a member of the `class-attributes` class.

**'last ( class-attributes -- address -> definition )**

`address -> definition` is the address of the most recent protected definition of the class associated with `class-attributes`.

`'last` is a member of the `class-attributes` class.

**'length ( structure-attributes -- address -> object-size )**

struct.sf

`address -> object-size` is the address of a cell containing the size in bytes of the structure associated with `structure-attributes`.

`'length` is a member of the `structure-attributes` class.

**'object-size ( vtable -- address -> object-size )**

strongforth.sf

address -> object-size is the address of the first entry within the virtual method table vtable. This entry contains the size in bits of the associated object.

**'parent ( data-type-attributes -- address -> data-type-attributes )**

address -> data-type-attributes is the address of a cell containing the data-type-attributes of the parent of the data type associated with data-type-attributes.

'parent is a member of the data-type-attributes class.

**'register ( data-type-attributes -- caddress -> logical )**

caddress -> logical is the address of a character-size logical value indicating the default register attributes of the data type associated with data-type-attributes. The default register has to be considered only when programming in assembler.

'register is a member of the data-type-attributes class.

**'size ( data-type-attributes -- caddress -> unsigned )**

caddress -> unsigned is the address of a character-size unsigned value indicating the size in address units of the data type associated with data-type-attributes.

'size is a member of the data-type-attributes class.

**'virtual ( unsigned vtable -- address -> token )**

strongforth.sf

address -> token is the address of the entry with index unsigned within the virtual method table vtable.

**'vocabulary ( class-attributes -- address -> vocabulary )**

address -> vocabulary is the address of a pointer to the private vocabulary of the class associated with class-attributes.

'vocabulary is a member of the class-attributes class.

**'vtable ( data-type-attributes -- address -> vtable )**

address -> vtable is the address of a pointer to the virtual method table of the data type associated with data-type-attributes.

'vtable is a member of the data-type-attributes class.

**( ( -- stack-diagram ) immediate**

Save the value of state. Create an empty stack diagram stack-diagram. Enter interpretation state.

( starts a stack diagram. Note that the semantics of ( is not the same as in Forth-2012.

**(+loop) ( integer address -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer address -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of two floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> complex` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer address -> double -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of two cells in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> double` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer address -> float -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of a floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> float` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer address -> single -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of a cell in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> single` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer caddress -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of a character in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `caddress` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer dfaddress -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of a double-precision floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `dfaddress` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer dfaddress -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of two double-precision floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `dfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer integer -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add the first `integer` to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. The second `integer` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer sfaddress -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of a single-precision floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `sfaddress` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(+loop) ( integer sfaddress -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` times the size of two single-precision floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `sfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(+loop)` is an internal definition compiled by `+loop`.

**`(--)( stack-diagram -- 1st )`**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type `(--)`. This data type is the qualified token of a definition with the stack diagram `( -- )`.

**`(--string)( stack-diagram -- 1st )`**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type (`--string`). This data type is the qualified token of a definition with the stack diagram (`--caddress -> character unsigned`).

**(-loop) ( integer address -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address` is a dummy parameter indicating the data type of the loop index.

(-loop) is an internal definition compiled by -loop.

**(-loop) ( integer address -> complex -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of two floating-point numbers in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> complex` is a dummy parameter indicating the data type of the loop index.

(-loop) is an internal definition compiled by -loop.

**(-loop) ( integer address -> double -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of two cells in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> double` is a dummy parameter indicating the data type of the loop index.

(-loop) is an internal definition compiled by -loop.

**(-loop) ( integer address -> float -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of a floating-point number in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> float` is a dummy parameter indicating the data type of the loop index.

(-loop) is an internal definition compiled by -loop.

**(-loop) ( integer address -> single -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of a cell in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `address -> single` is a dummy parameter indicating the data type of the loop index.

(-loop) is an internal definition compiled by -loop.

**(-loop) ( integer caddress -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of a character in address units from the loop index. `flag` is `true` if and only if the

loop index crosses the boundary between the loop limit minus one and the loop limit. `caddress` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(-loop) ( integer dfaddress -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of a double-precision floating-point number in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `dfaddress` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(-loop) ( integer dfaddress -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of two double-precision floating-point numbers in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `dfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(-loop) ( integer integer -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract the first `integer` from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. The second `integer` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(-loop) ( integer sfaddress -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of a single-precision floating-point number in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `sfaddress` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(-loop) ( integer sfaddress -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Subtract `integer` times the size of two single-precision floating-point number in address units from the loop index. `flag` is `true` if and only if the loop index crosses the boundary between the loop limit minus one and the loop limit. `sfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(-loop)` is an internal definition compiled by `-loop`.

**`(0branch) ( single -- )`**



If `single` is zero, branch forward or backward within the current definition. Otherwise continue execution.

`0branch` is an internal definition compiled by `if`, `until` and `of`.

**(0lbranch) ( single -- )**

If `single` is zero, perform a long branch forward or backward within the current definition. Otherwise continue execution.

`0lbranch` is an internal definition.

**(>r) ( complex -- )**

Store `complex` as a local in the stack frame of the current definition.

(>r) is an internal definition compiled by `>r` and `(local)`.

**(>r) ( double -- )**

Store `double` as a local in the stack frame of the current definition.

(>r) is an internal definition compiled by `>r` and `(local)`.

**(>r) ( float -- )**

Store `float` as a local in the stack frame of the current definition.

(>r) is an internal definition compiled by `>r` and `(local)`.

**(>r) ( single -- )**

Store `single` as a local in the stack frame of the current definition.

(>r) is an internal definition compiled by `>r` and `(local)`.

**(>token) ( definition stack-diagram flag -- token )**

`token` is the execution token of `definition`. An exception is thrown if the stack diagram of `definition` does not match `stack-diagram` according to the rules of the StrongForth data type system. If `definition` does not have an execution token or if matching the stack diagrams requires some register shuffling, a chunk of code is compiled to generate a valid execution token. This works in either compilation or interpretation state. Delete `stack-diagram`.

If `flag` is `true` or the stack diagrams do not match, the chunk of code is removed before (>token) returns, and `token` is zero.

**(abort") ( single caddress -> character unsigned -- )**

strongforth.sf

If `single` is not equal to zero, copy the string `caddress -> character unsigned` to `line`, fill the remainder of `line` with spaces and throw an exception with code -2.

(abort") is an internal definition compiled by `abort"`.

**(bind) ( class-attributes "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find a virtual definition *name* that matches the compiler data type heap according to the rules of the StrongForth data type system. If no such virtual definition is found, compile *this* and try finding *name* again. Append the runtime semantics of the virtual definition *name* that is bound to the class associated with *class-attributes* to the current definition. An exception is thrown if no suitable virtual definition *name* is found or if *name* is not a virtual definition within the scope of the class associated with *class-attributes*. An ambiguous condition exists if (bind) is executed in interpretation state.

**(branch) ( -- )**

Unconditionally branch forward or backward within the current definition.

branch is an internal definition compiled by ahead, again and endof.

**(compile) ( compiler-workspace definition -- )**

Compile machine code instructions that perform the semantics of definition into the code-space memory space, using compiler-workspace.

(compile) is a low-level compilation word.

(compile) is a virtual method of the definition class.

**(create) ( caddress -> character unsigned -- )**

strongforth.sf

Create a definition with the name specified by the string *caddress -> character unsigned* and the execution semantics defined below, and make it the current definition. *create* does not allocate memory in the definition's data field.

Execution: Execute the definition. The default execution semantics of the new definition is placing the address of its data field onto the stack.

Note that the stack diagram of the new definition has to be explicitly specified. The execution semantics may be extended by *does>* or *;code*.

**(do) ( address 1st -- )**

Store the loop limit *address* and the loop index *1st* as locals in the stack frame of the current definition.

(do) is an internal definition compiled by do and ?do.

**(do) ( integer 1st -- )**

Store the loop limit *integer* and the loop index *1st* as locals in the stack frame of the current definition.

(do) is an internal definition compiled by do and ?do.

**(does) ( code-definition -- )**

strongforth.sf

Finish the current definition by specifying `code-definition` as the definition performing its runtime code. An exception is thrown if the current definition was not created by `create`. If the current definition has no stack diagram, use the stack diagram of `code-definition`, except for the last input parameter, as its stack diagram. An exception is thrown if `code-definition` has no input parameters or if `code-definition` has one or more output parameters that reference the last input parameter.

(`does`) is an internal definition compiled by `does>`.

**(execute) ( token -- )**

Execute the machine code instructions starting at `token`.

Note that (`execute`) does not verify or update the data type heap. (`execute`) is a low-level definition that should be used carefully, because it may corrupt the data type system. Especially, it should not be used in place of `execute`.

**(forget) ( definition definition -- )**

strongforth.sf

Delete the second `definition` and all previous definitions in the same vocabulary up to and excluding the first `definition`.

**(lbranch) ( -- )**

Perform an unconditional long branch forward or backward within the current definition.

`lbranch` is an internal definition.

**(literal) ( complex data-type -- )**

Compile machine code instructions for the complex floating-point literal `complex` into the code-space memory space, using the register specification given by `data-type`.

(`literal`) is a low-level compilation word. An exception is thrown if the register specification is inappropriate for a complex floating-point literal.

**(literal) ( double data-type -- )**

Compile machine code instructions for the double-cell literal `double` into the code-space memory space, using the register specification given by `data-type`. (`literal`) is a low-level compilation word. An exception is thrown if the register specification is inappropriate for a double-cell literal.

**(literal) ( float data-type -- )**

Compile machine code instructions for the floating-point literal `float` into the code-space memory space, using the register specification given by `data-type`. (`literal`) is a low-level compilation word. An exception is thrown if the register specification is inappropriate for a floating-point literal.

**(literal) ( single data-type -- )**

Compile machine code instructions for the single-cell literal `single` into the code-space memory space, using the register specification given by `data-type`. `(literal)` is a low-level compilation word. An exception is thrown if the register specification is inappropriate for a single-cell literal.

**`(local) ( caddress -> character unsigned --)`**

strongforth.sf

Execution: If `unsigned` is non-zero, create a new local whose definition name is given by the character string `caddress -> character unsigned`. If `unsigned` is zero, `caddress -> character` has no significance.

An ambiguous condition exists if `(local)` is executed in interpretation state.

The result of executing `(local)` during compilation is creating a set of named local identifiers, each of which is a definition that only has execution semantics within the scope of that definition's source.

Runtime: Push the local's value onto the stack. The value can be either a single-cell item, a double-cell item or a floating-point number.

**`(loop) ( address -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add one address unit to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `address` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**`(loop) ( address -> complex -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of two floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `address -> complex` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**`(loop) ( address -> double -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of two cells in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `address -> double` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**`(loop) ( address -> float -- flag )`**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of a floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `address -> float` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( address -> single -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of a cell in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `address -> single` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( caddress -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of a character in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `caddress` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( dfaddress -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of a double-precision floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `dfaddress` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( dfaddress -> complex -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of two double-precision floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `dfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( integer -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add one to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `integer` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( sfaddress -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of a single-precision floating-point number in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `sfaddress` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**(loop) ( sfaddress -> complex -- flag )**

An ambiguous condition exists if the loop control parameters are unavailable. Add the size of two single-precision floating-point numbers in address units to the loop index. `flag` is `true` if and only if the loop index is then equal to the loop limit. `sfaddress -> complex` is a dummy parameter indicating the data type of the loop index.

`(loop)` is an internal definition compiled by `loop`.

**`(member) ( unsigned object-size data-type "<spaces>name" - 2nd )`**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `2nd` is equal to `object-size plus unsigned`.

*name* is referred to as a class member. `(members)` reserves unsigned bits for class members of the data type pointed to by `dt-here`.

Execution: `( x -- addr -> y )`

`addr -> y` is the address of an array of class members of the object *x*, that were reserved at the time *name* was created. `addr` is data-type. *y* is the data type pointed to by `dt-here`.

`(member)` is an internal definition used by all versions of `members`.

**`(new) ( address -> object vtable -> 2nd -- 2nd )`**

Initialize the virtual method table pointer of an object starting at `address -> object` with `vtable -> 2nd`, and return the object as `2nd`. The initial contents of the object's members are undefined. An ambiguous condition exists if `vtable -> 2nd` is not the address of the virtual method table of the class of `object`.

**`(new) ( memory-space vtable -> object -- 3rd )`**

Allot a chunk of contiguous memory space from `memory-space` for `object`. Initialize the virtual method table pointer of the object with `vtable -> object`, and return the object as `3rd`. The initial contents of the allotted object's members are undefined. An exception is thrown if `memory-space` does not have enough unused memory cells. An ambiguous condition exists if `vtable -> object` is not the address of the virtual method table of the class of `object`.

**`(new) ( vtable -> object -- 2nd )`**

Allocate a chunk of contiguous dynamic memory space for `object`. Initialize the virtual method table pointer of the object with `vtable -> object`, and return the object as `2nd`. The initial contents of the allocated object's members are undefined. An exception is thrown if memory allocation fails. An ambiguous condition exists if `vtable -> object` is not the address of the virtual method table of the class of `object`.

**`(params>dt) ( data-type address -> data-type -- )`**

If `data-type` does not reference another data type, append `data-type` to the data type heap selected by `state` and finish execution. Otherwise, append the referenced compound data type from the list of data types starting at `address -> data-type` to the data type heap selected by `state`. If the referenced compound data type contains itself a reference to another compound

data type, the tail of the referenced compound data type is recursively substituted by the referenced data type. An exception is thrown if the data type heap overflows.

**(quit) ( -- )**

(quit) is executed by quit immediately before entering the interpreter loop. The semantics is initialized with ignore-friends.

(quit) is a deferred definition.

**(replaces) ( (--string) caddress -> character unsigned -- )**

strex.sf

Create a definition with the name specified by the character string `caddress -> character unsigned` with the execution semantics defined below. The definition specifies a replacement string for substitute.

Execution: ( -- caddress -> character unsigned )

`caddress -> character unsigned` is the text returned by executing `(--string)` at execution time.

**(replaces) ( caddress -> character unsigned caddress -> character unsigned -- )**

strex.sf

Create a definition with the name specified by the second character string `caddress -> character unsigned` with the execution semantics defined below. The definition specifies a replacement string for substitute.

Execution: ( -- caddress -> character unsigned )

`caddress -> character unsigned` is a copy of the second character string `caddress -> character unsigned` provided to (replaces).

**(represent) ( caddress -> character unsigned float -- )**

At `caddress -> character`, place the character-string external representation of the rounded integer part of the absolute value of the floating-point number `float`. The character string consists of unsigned digits, extended by leading zeros as required. Rounding follows the round to nearest rule. An ambiguous condition exists if the absolute value of `float` is bigger than what can be represented with unsigned digits, or if `float` is not a valid floating-point number.

**(se.) ( float signed -- )**

float.sf

Send `float` with a trailing space using exponential notation to the default output stream. The significand is greater than or equal to 1.0 and less than 10.0 to the power `signed`, and the decimal exponent is a multiple of `signed`:

```
Exponential notation := <significand><exponent>
<significand>       := [-]<digits>.<digits0>
<exponent>          := e[-|+]<digit><digit><digit>
<digits>             := <digit><digits0>
<digits0>            := <digit>*
<digit>              := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An ambiguous condition exists if `signed` is not greater than zero. An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

`(se.)` is an internal definition used by `e.` and `s..`

**`(substitute) ( unsigned address -> character unsigned -- 1st 2nd 4 th )`** strext.sf

Scan the character string `address -> character unsigned` for the first delimiter character. If the name before the delimiter character is a valid replacement string, send the replacement to the default output stream and increment the first unsigned, giving 1st. Otherwise, send a delimiter character and the text up to and including the delimiter character to the default output stream.

2nd 4th is `address -> character unsigned`, adjusted with `/string` by the number of characters up to and including the first delimiter character.

`(substitute)` is an internal definition used by `substitute`.

**`(unsigned--)( stack-diagram -- 1st )`** strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type `(unsigned--)`. This data type is the qualified token of a definition with the stack diagram `( unsigned -- )`.

**`(value) ( "<spaces>name" -- value-definition )`** strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create value-definition for *name* with the data-space memory space pointer as the address of the value. The data type of value-definition is the most recently dropped compound data type on the data type heap.

`(value)` is an internal definition used by all versions of `value`.

**`(variable) ( data-type "<spaces>name" -- single-definition )`** strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create single-definition for *name* with the current data-space memory space pointer as the value. The data type of single-definition is composed of data-type as the head and the most recently dropped compound data type on the data type heap as the tail.

`(variable)` is an internal definition used by all versions of `variables`.

**`(vtable) ( -- vtable )`** strongforth.sf

Interpretation: Creates the input parameter with data type `vtable -> object` for `(new)`. `object` is the data type that has most recently been dropped from the interpreter data type heap. `vtable` is the virtual method table of `object`.

Compilation: An ambiguous condition exists if `(vtable)` is executed in compilation state.

`(vtable)` is an internal definition used by `new`.

**`) ( colon-definition stack-diagram -- 1st )`**



Throws an exception if `stack-diagram` is incomplete. Make `stack-diagram` the stack diagram of `colon-definition`. Delete `stack-diagram`. Append the input parameters of `colon-definition` to the compiler data type heap, starting with the first input parameter. Data type references within the input parameters are being resolved by recursively appending the referenced data types onto the compiler data type heap. An exception is thrown if the data type heap overflows. `1st` is `colon-definition`.

) marks the end of a colon definition's stack diagram.

**) ( stack-diagram -- )**

Throws an exception if `stack-diagram` is incomplete. Make `stack-diagram` the stack diagram of the latest definition. Delete `stack-diagram`.

) marks the end of a definition's stack diagram.

**)' ( stack-diagram "<spaces>name" -- definition )**

strongforth.sf

Throws an exception if `stack-diagram` is incomplete. Skip leading space delimiters. Parse *name* delimited by a space. Search the context vocabularies for *name* with exactly the given stack diagram and return its latest occurrence as *definition*. Delete `stack-diagram`. An exception is thrown if *name* with exactly the given stack diagram is not found.

**)procreates ( stack-diagram "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. The definition is a new data type that is a direct subtype of data type `token`. It is called a qualified token.

Create a definition `execute`, whose stack diagram is a copy of `stack-diagram`, supplemented with the previously created qualified token as the last input parameter.

)procreates marks the end of a stack diagram.

*name* Execution: ( stack-diagram -- 1st )

When used in a stack diagram, specifies an input or output parameter with the data type of the qualified token.

`execute` Execution: ( *x0* ... *xm* *name* -- *y0* ... *yn* )

Execute the qualified token with data type *name*. ( *x0* ... *xm* -- *y0* ... *yn* ) is the stack diagram that was supplied to )procreates. *x0* ... *xm* are the input parameters of the qualified token. *y0* ... *yn* are the output parameters of the qualified token.

**\* ( complex complex -- 1st )**

complex.sf

Multiply two complex floating-point numbers `complex` giving the product `1st`. The result has the same data type as the multiplicand.

**\* ( complex float -- 1st )**

complex.sf

Multiply the complex floating-point number `complex` by the real floating-point number `float`, giving the product `1st`. The result has the same data type as the multiplicand.

**\* ( float float -- 1st )**

Multiply two floating-point numbers `float` giving the product `1st`. The result has the same data type as the multiplicand.

**\* ( integer unsigned -- 1st )**

Multiply `integer` by `unsigned` giving the product `1st`. Note that the multiplicand can be any integer (signed or unsigned), while the multiplier is unsigned. The result has the same data type as the multiplicand. Since the product of an unsigned number (multiplicand) and a signed number (multiplier) should be a signed number, the two operands have to be swapped in this case.

**\* ( integer-double unsigned -- 1st )**

Multiply `integer-double` by `unsigned` giving the double-precision product `1st`. Note that the multiplicand can be any integer (signed or unsigned), while the multiplier is unsigned. The result has the same data type as the multiplicand.

**\* ( signed signed -- 1st )**

Multiply two `signed` numbers giving the product `1st`. The result has the same data type as the multiplicand.

**\* ( signed-double signed -- 1st )**

Multiply `signed-double` by `signed` giving the double-precision product `1st`. The result has the same data type as the multiplicand.

**\*\* ( complex complex -- 1st )**

`complex.sf`

Raise the first `complex` to the power given by the second `complex`, giving `1st`. An ambiguous condition exists if the quotient lies outside of the range of floating-point numbers. This operation is based on complex floating-point numbers.

**\*\* ( float float -- 1st )**

Raise the first `float` to the power given by the second `float`, giving `1st`. An ambiguous condition exists if the second `float` is negative, or if the quotient lies outside of the range of a floating-point number.

**\*/ ( signed signed signed -- 1st )**

Multiply the first `signed` by the second `signed` producing an intermediate signed double-precision result. Divide the intermediate result by the third `signed` giving the signed single-precision quotient `1st`. An exception is thrown if the third `signed` is zero. An ambiguous condition exists if the quotient `1st` lies outside the range of a signed single-precision number.

**\*/ ( signed-double signed signed -- 1st )**

`strongforth.sf`

Multiply `signed-double` by the first `signed` producing an intermediate signed triple-precision result. Divide the intermediate result by the second `signed` giving the signed double-

precision quotient 1st. An exception is thrown if the second signed is zero. An ambiguous condition exists if the quotient 1st lies outside of the range of a signed double-precision number.

**`*/ ( unsigned unsigned unsigned -- 1st )`**

Multiply the first unsigned by the second unsigned producing an intermediate unsigned double-precision result. Divide the intermediate result by the third unsigned giving the unsigned single-precision quotient 1st. An exception is thrown if the third unsigned is zero. An ambiguous condition exists if the quotient 1st lies outside the range of an unsigned single-precision number.

**`*/ ( unsigned-double unsigned unsigned -- 1st )`**

Multiply unsigned-double by the first unsigned producing an intermediate unsigned triple-precision result. Divide the intermediate result by the second unsigned giving the unsigned double-precision quotient 1st. An exception is thrown if the second unsigned is zero. An ambiguous condition exists if the quotient 1st lies outside of the range of an unsigned double-precision number.

**`*/mod ( signed signed signed -- 3rd 1st )`**

Multiply the first signed by the second signed producing an intermediate signed double-precision result. Divide the intermediate result by the third signed giving the signed single-precision remainder 3rd and the signed single-precision quotient 1st. An exception is thrown if the third signed is zero. An ambiguous condition exists if the quotient 1st lies outside the range of a signed single-precision number.

**`*/mod ( signed-double signed signed -- 3rd 1st )`**

strongforth.sf

Multiply signed-double by the first signed producing an intermediate signed triple-precision result. Divide the intermediate result by the second signed giving the signed single-precision remainder 3rd and the signed double-precision quotient 1st. An exception is thrown if the second signed is zero. An ambiguous condition exists if the quotient 1st lies outside of the range of a signed double-precision number.

**`*/mod ( unsigned unsigned unsigned -- 3rd 1st )`**

Multiply the first unsigned by the second unsigned producing an intermediate unsigned double-precision result. Divide the intermediate result by the third unsigned giving the unsigned single-precision remainder 3rd and the unsigned single-precision quotient 1st. An exception is thrown if the third unsigned is zero. An ambiguous condition exists if the quotient 1st lies outside the range of an unsigned single-precision number.

**`*/mod ( unsigned-double unsigned unsigned -- 3rd 1st )`**

Multiply unsigned-double by the first unsigned producing an intermediate unsigned triple-precision result. Divide the intermediate result by the second unsigned giving the unsigned single-precision remainder 3rd and the unsigned double-precision quotient 1st. An exception is thrown if the second unsigned is zero. An ambiguous condition exists if the quotient 1st lies outside of the range of an unsigned double-precision number.

**\*10^n ( float signed -- 1st )**

float.sf

1st is equal to float multiplied by 10 raised to the power of signed. signed may be positive, negative, or zero.

**+ ( address -> complex integer -- 1st )**

Add integer to address -> complex, giving the sum 1st. Since address points to a complex floating-point number, integer is multiplied with the number of address units per complex floating-point number before the actual addition takes place.

**+ ( address -> double integer -- 1st )**

Add integer to address -> double, giving the sum 1st. Since address points to a double cell, integer is multiplied with the number of address units per double cell before the actual addition takes place.

**+ ( address -> float integer -- 1st )**

Add integer to address -> float, giving the sum 1st. Since address points to a floating-point number, integer is multiplied with the number of address units per floating-point number before the actual addition takes place.

**+ ( address -> single integer -- 1st )**

Add integer to address -> single, giving the sum 1st. Since address points to a cell, integer is multiplied with the number of address units per cell before the actual addition takes place.

**+ ( address integer -- 1st )**

Add integer to address, giving the sum 1st.

**+ ( caddress integer -- 1st )**

Add integer to caddress, giving the sum 1st. Since caddress points to an item of character size, integer is multiplied with the number of address units per character before the actual addition takes place.

**+ ( complex complex -- 1st )**

complex.sf

Add two complex floating-point numbers complex giving the sum 1st. The result has the same data type as the first operand.

**+ ( complex float -- 1st )**

complex.sf

Add the real floating-point number float to the complex floating-point number complex, giving the sum 1st. The result has the same data type as complex.

**+ ( dfaddress -> complex integer -- 1st )**

Add integer to dfaddress -> complex, giving the sum 1st. Since dfaddress -> complex points to a complex double-precision floating-point number, integer is multiplied with the number of address units per complex double-precision floating-point number before the actual addition takes place.

**+ ( dfaddress integer -- 1st )**

Add integer to dfaddress, giving the sum 1st. Since dfaddress points to a double-precision floating-point number, integer is multiplied with the number of address units per double-precision floating-point number before the actual addition takes place.

**+ ( float float -- 1st )**

Add the second float to the first float, giving the sum 1st.

**+ ( integer integer -- 1st )**

Add the second integer to the first integer, giving the sum 1st.

**+ ( integer-double integer -- 1st )**

Add integer with zero extension to integer-double, giving the double-precision sum 1st.

**+ ( integer-double integer-double -- 1st )**

Add the second integer-double to the first integer-double, giving the sum 1st.

**+ ( integer-double signed -- 1st )**

Add signed with sign extension to integer-double, giving the double-precision sum 1st.

**+ ( sfaddress -> complex integer -- 1st )**

Add integer to sfaddress -> complex, giving the sum 1st. Since sfaddress -> complex points to a complex single-precision floating-point number, integer is multiplied with the number of address units per complex single-precision floating-point number before the actual addition takes place.

**+ ( sfaddress integer -- 1st )**

Add integer to sfaddress, giving the sum 1st. Since sfaddress points to a single-precision floating-point number, integer is multiplied with the number of address units per single-precision floating-point number before the actual addition takes place.

**+! ( complex address -> complex -- )**

Add complex to the complex floating-point number stored at address -> complex.

**+! ( complex dfaddress -> complex -- )**

Add `complex` to the complex double-precision floating-point number stored at `dfaddress` -> `complex`.

**+! ( complex sfaddress -> complex -- )**

Add `complex` to the complex single-precision floating-point number stored at `sfaddress` -> `complex`.

**+! ( float address -> float -- )**

Add `float` to the floating-point number stored at `address` -> `float`.

**+! ( float dfaddress -> float -- )**

Add `float` to the double-precision floating-point number stored at `dfaddress` -> `float`.

**+! ( float sfaddress -> float -- )**

Add `float` to the single-precision floating-point number stored at `sfaddress` -> `float`.

**+! ( integer address -> address -- )**

Add `integer` to the address stored at `address` -> `address`.

**+! ( integer address -> address -> complex -- )**

Add `integer` to the address stored at `address` -> `address` -> `complex`. Since the address points to a complex floating-point number, `integer` is multiplied with the number of address units per complex floating-point number before the actual addition takes place.

**+! ( integer address -> address -> double -- )**

Add `integer` to the address stored at `address` -> `address` -> `double`. Since the address points to a double cell, `integer` is multiplied with the number of address units per double cell before the actual addition takes place.

**+! ( integer address -> address -> float -- )**

Add `integer` to the address stored at `address` -> `address` -> `float`. Since the address points to a floating-point number, `integer` is multiplied with the number of address units per floating-point number before the actual addition takes place.

**+! ( integer address -> address -> single -- )**

Add `integer` to the address stored at `address` -> `address` -> `single`. Since the address points to a cell, `integer` is multiplied with the number of address units per cell before the actual addition takes place.

**+! ( integer address -> caddress -- )**

Add *integer* to the address stored at *address* -> *caddress*. Since the address points to an item of character size, *integer* is multiplied with the number of address units per character before the actual addition takes place.

**+! ( integer address -> dfaddress -- )**

Add *integer* to the address stored at *address* -> *dfaddress*. Since the address points to a double-precision floating-point number, *integer* is multiplied with the number of address units per double-precision floating-point number before the actual addition takes place.

**+! ( integer address -> dfaddress -> complex -- )**

Add *integer* to the address stored at *address* -> *dfaddress* -> *complex*. Since the address points to a complex double-precision floating-point number, *integer* is multiplied with the number of address units per complex double-precision floating-point number before the actual addition takes place.

**+! ( integer address -> integer -- )**

Add *integer* to the integer number stored at *address* -> *integer*.

**+! ( integer address -> integer-double -- )**

Add *integer* with zero extension to the double-precision integer number stored at *address* -> *integer-double*.

**+! ( integer address -> sfaddress -- )**

Add *integer* to the address stored at *address* -> *sfaddress*. Since the address points to a single-precision floating-point number, *integer* is multiplied with the number of address units per single-precision floating-point number before the actual addition takes place.

**+! ( integer address -> sfaddress -> complex -- )**

Add *integer* to the address stored at *address* -> *sfaddress* -> *complex*. Since the address points to a complex single-precision floating-point number, *integer* is multiplied with the number of address units per complex single-precision floating-point number before the actual addition takes place.

**+! ( integer caddress -> integer -- )**

Add *integer* to the character size integer number stored at *caddress* -> *integer*.

**+! ( integer-double address -> integer-double -- )**

Add *integer-double* to the double-precision integer number stored at *address* -> *integer-double*.

**+! ( signed address -> integer-double -- )**

Add `signed` with sign extension to the double-precision integer number stored at `address` -> `integer-double`.

**+loop ( do-destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. Resolve both the forward references and the backward reference of `do-destination`. Delete the loop index `i`. Rename the loop index `j`, if it exists, to `i`. An exception is thrown if the contents of the compiler data type heap do not exactly match the copy that was saved when `do-destination` was created.

Runtime: ( `integer` -- )

An ambiguous condition exists if the loop control parameters are unavailable. Add `integer` to the loop index. If the loop index crosses the boundary between the loop limit minus one and the loop limit, discard the current loop control parameters and continue execution. Otherwise, branch to the beginning of the loop.

Note: `+loop` takes regard of the data type of the loop index.

If the loop index is an address of a single cell, `integer` is multiplied with the size of a single cell in address units before it is added to the loop index.

If the loop index is an address of a double cell, `integer` is multiplied with the size of a double cell in address units before it is added to the loop index.

If the loop index is a character address, `integer` is multiplied with the size of a character in address units before it is added to the loop index.

If the loop index is an address of a floating-point number, `integer` is multiplied with the size of a floating-point number in address units before it is added to the loop index.

If the loop index is an address of a single-precision floating-point number, `integer` is multiplied with the size of a single-precision floating-point number in address units before it is added to the loop index.

If the loop index is an address of a double-precision floating-point number, `integer` is multiplied with the size of a double-precision floating-point number in address units before it is added to the loop index.

If the loop index is an address of a complex floating-point number, `integer` is multiplied with the size of a complex floating-point number in address units before it is added to the loop index.

If the loop index is an address of a complex single-precision floating-point number, `integer` is multiplied with the size of a complex single-precision floating-point number in address units before it is added to the loop index.

If the loop index is an address of a complex double-precision floating-point number, `integer` is multiplied with the size of a complex double-precision floating-point number in address units before it is added to the loop index.

**, ( complex -- )**

Reserve space for two floating-point numbers in the default memory space and store `complex` in it. An ambiguous condition exists if the first unused address of the default memory space is not aligned prior to execution of `,`. An exception is thrown if the default memory space overflows.



**, ( complex memory-space -- )**

Reserve space for two floating-point numbers in `memory-space` and store `complex` in it. An ambiguous condition exists if the first unused address of `memory-space` is not aligned prior to execution of `,`. An exception is thrown if `memory-space` overflows.

**, ( double -- )**

Reserve two cells in the default memory space and store `double` in the two cells. If the first unused address of the default memory space is aligned prior to execution of `,`, it will remain aligned when `,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not aligned prior to execution of `,`. An exception is thrown if the default memory space overflows.

**, ( double memory-space -- )**

Reserve two cells in `memory-space` and store `double` in the two cells. If the first unused address of `memory-space` is aligned prior to execution of `,`, it will remain aligned when `,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not aligned prior to execution of `,`. An exception is thrown if `memory-space` overflows.

**, ( float -- )**

Reserve space for one floating-point number in the default memory space and store `float` in it. An ambiguous condition exists if the first unused address of the default memory space is not aligned prior to execution of `,`. An exception is thrown if the default memory space overflows.

**, ( float memory-space -- )**

Reserve space for one floating-point number in `memory-space` and store `float` in it. An ambiguous condition exists if the first unused address of `memory-space` is not aligned prior to execution of `,`. An exception is thrown if `memory-space` overflows.

**, ( single -- )**

Reserve one cell in the default memory space and store `single` in the cell. If the first unused address of the default memory space is aligned prior to execution of `,`, it will remain aligned when `,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not aligned prior to execution of `,`. An exception is thrown if the default memory space overflows.

**, ( single memory-space -- )**

Reserve one cell in `memory-space` and store `single` in the cell. If the first unused address of `memory-space` is aligned prior to execution of `,`, it will remain aligned when `,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not aligned prior to execution of `,`. An exception is thrown if `memory-space` overflows.

**- ( address -> complex 1st -- signed )**

Subtract 1st from address -> complex, giving an intermediate difference. Since address -> complex points to a complex floating-point number, the result signed is equal to the difference divided by the number of address units per complex floating-point number.

**- ( address -> complex integer -- 1st )**

Subtract integer from address -> complex, giving the difference 1st. Since address points to a complex floating-point number, integer is multiplied with the number of address units per complex floating-point number before the actual subtraction takes place.

**- ( address -> double 1st -- signed )**

Subtract 1st from address -> double, giving an intermediate difference. Since address -> double points to a double cell, the result signed is equal to the difference divided by the number of address units per double cell.

**- ( address -> double integer -- 1st )**

Subtract integer from address -> double, giving the difference 1st. Since address points to a double cell, integer is multiplied with the number of address units per double cell before the actual subtraction takes place.

**- ( address -> float 1st -- signed )**

Subtract 1st from address -> float, giving an intermediate difference. Since address -> float points to a floating-point number, the result signed is equal to the difference divided by the number of address units per floating-point number.

**- ( address -> float integer -- 1st )**

Subtract integer from address -> float, giving the difference 1st. Since address points to a floating-point number, integer is multiplied with the number of address units per floating-point number before the actual subtraction takes place.

**- ( address -> single 1st -- signed )**

Subtract 1st from address -> single, giving an intermediate difference. Since address -> single points to a cell, the result signed is equal to the difference divided by the number of address units per cell.

**- ( address -> single integer -- 1st )**

Subtract integer from address -> single, giving the difference 1st. Since address points to a cell, integer is multiplied with the number of address units per cell before the actual subtraction takes place.

**- ( address 1st -- signed )**

Subtract 1st from address, giving signed.

**- ( address integer -- 1st )**

Subtract `integer` from `address`, giving the difference `1st`.

**- ( caddress 1st -- signed )**

Subtract `1st` from `caddress`, giving an intermediate difference. Since `caddress` points to a character-size item, the result `signed` is equal to the difference divided by the number of address units per character.

**- ( caddress integer -- 1st )**

Subtract `integer` from `caddress`, giving the difference `1st`. Since `caddress` points to a character, `integer` is multiplied with the number of address units per character before the actual subtraction takes place.

**- ( complex complex -- 1st )**

`complex.sf`

Subtract the second `complex` from the first `complex`, giving the difference `1st`.

**- ( complex float -- 1st )**

`complex.sf`

Subtract the real floating-point number `float` from the complex floating-point number `complex`, giving the difference `1st`. The result has the same data type as `complex`.

**- ( dfaddress -> complex 1st -- signed )**

Subtract `1st` from `dfaddress -> complex`, giving an intermediate difference. Since `dfaddress -> complex` points to a complex double-precision floating-point number, the result `signed` is equal to the difference divided by the number of address units per complex double-precision floating-point number.

**- ( dfaddress -> complex integer -- 1st )**

Subtract `integer` from `dfaddress -> complex`, giving the difference `1st`. Since `dfaddress -> complex` points to a complex double-precision floating-point number, `integer` is multiplied with the number of address units per complex double-precision floating-point number before the actual subtraction takes place.

**- ( dfaddress 1st -- signed )**

Subtract `1st` from `dfaddress`, giving an intermediate difference. Since `dfaddress` points to a double-precision floating-point number, the result `signed` is equal to the difference divided by the number of address units per double-precision floating-point number.

**- ( dfaddress integer -- 1st )**

Subtract `integer` from `dfaddress`, giving the difference `1st`. Since `dfaddress` points to a double-precision floating-point number, `integer` is multiplied with the number of address units per double-precision floating-point number before the actual subtraction takes place.

**- ( float float -- 1st )**

Subtract the second float from the first float, giving the difference 1st.

**- ( integer integer -- 1st )**

Subtract the second integer from the first integer, giving the difference 1st.

**- ( integer-double integer -- 1st )**

Subtract integer with zero extension from integer-double, giving the double-precision difference 1st.

**- ( integer-double integer-double -- 1st )**

Subtract the second integer-double from the first integer-double, giving the difference 1st.

**- ( integer-double signed -- 1st )**

Subtract signed with sign extension from integer-double, giving the double-precision difference 1st.

**- ( sfaddress -> complex 1st -- signed )**

Subtract 1st from sfaddress -> complex, giving an intermediate difference. Since sfaddress -> complex points to a complex single-precision floating-point number, the result signed is equal to the difference divided by the number of address units per complex single-precision floating-point number.

**- ( sfaddress -> complex integer -- 1st )**

Subtract integer from sfaddress -> complex, giving the difference 1st. Since sfaddress -> complex points to a complex single-precision floating-point number, integer is multiplied with the number of address units per complex single-precision floating-point number before the actual subtraction takes place.

**- ( sfaddress 1st -- signed )**

Subtract 1st from sfaddress, giving an intermediate difference. Since sfaddress points to a single-precision floating-point number, the result signed is equal to the difference divided by the number of address units per single-precision floating-point number.

**- ( sfaddress integer -- 1st )**

Subtract integer from sfaddress, giving the difference 1st. Since sfaddress points to a single-precision floating-point number, integer is multiplied with the number of address units per single-precision floating-point number before the actual subtraction takes place.

**-! ( complex address -> complex -- )**

Subtract `complex` from the complex floating-point number stored at `address` -> `complex`.

**-! ( complex dfaddress -> complex -- )**

Subtract `complex` from the complex double-precision floating-point number stored at `dfaddress` -> `complex`.

**-! ( complex sfaddress -> complex -- )**

Subtract `complex` from the complex single-precision floating-point number stored at `sfaddress` -> `complex`.

**-! ( float address -> float -- )**

Subtract `float` from the floating-point number stored at `address` -> `float`.

**-! ( float dfaddress -> float -- )**

Subtract `float` from the double-precision floating-point number stored at `dfaddress` -> `float`.

**-! ( float sfaddress -> float -- )**

Subtract `float` from the single-precision floating-point number stored at `sfaddress` -> `float`.

**-! ( integer address -> address -- )**

Subtract `integer` from the address stored at `address` -> `address`.

**-! ( integer address -> address -> complex -- )**

Subtract `integer` from the address stored at `address` -> `address` -> `complex`. Since the address points to a complex floating-point number, `integer` is multiplied with the number of address units per complex floating-point number before the actual subtraction takes place.

**-! ( integer address -> address -> double -- )**

Subtract `integer` from the address stored at `address` -> `address` -> `double`. Since the address points to a double cell, `integer` is multiplied with the number of address units per double cell before the actual subtraction takes place.

**-! ( integer address -> address -> float -- )**

Subtract `integer` from the address stored at `address` -> `address` -> `float`. Since the address points to a floating-point number, `integer` is multiplied with the number of address units per floating-point number before the actual subtraction takes place.

**-! ( integer address -> address -> single -- )**

Subtract `integer` from the address stored at `address -> address -> single`. Since the address points to a cell, `integer` is multiplied with the number of address units per cell before the actual subtraction takes place.

**-! ( integer address -> caddress -- )**

Subtract `integer` from the address stored at `address -> caddress`. Since the address points to a character, `integer` is multiplied with the number of address units per character before the actual subtraction takes place.

**-! ( integer address -> dfaddress -- )**

Subtract `integer` from the address stored at `address -> dfaddress`. Since the address points to a double-precision floating-point number, `integer` is multiplied with the number of address units per double-precision floating-point number before the actual subtraction takes place.

**-! ( integer address -> dfaddress -> complex -- )**

Subtract `integer` from the address stored at `address -> dfaddress -> complex`. Since the address points to a complex double-precision floating-point number, `integer` is multiplied with the number of address units per complex double-precision floating-point number before the actual subtraction takes place.

**-! ( integer address -> integer -- )**

Subtract `integer` from the integer number stored at `address -> integer`.

**-! ( integer address -> integer-double -- )**

Subtract `integer` with zero extension from the double-cell integer number stored at `address -> integer-double`.

**-! ( integer address -> sfaddress -- )**

Subtract `integer` from the address stored at `address -> sfaddress`. Since the address points to a single-precision floating-point number, `integer` is multiplied with the number of address units per single-precision floating-point number before the actual subtraction takes place.

**-! ( integer address -> sfaddress -> complex -- )**

Subtract `integer` from the address stored at `address -> sfaddress -> complex`. Since the address points to a complex single-precision floating-point number, `integer` is multiplied with the number of address units per complex single-precision floating-point number before the actual subtraction takes place.

**-! ( integer caddress -> integer -- )**

Subtract `integer` from the character size integer number stored at `caddress -> integer`.

**-! ( integer-double address -> integer-double -- )**

Subtract integer-double from the double-precision integer number stored at address -> integer-double.

**-! ( signed address -> integer-double -- )**

Subtract signed with sign extension from the double-precision integer number stored at address -> integer-double.

**-- ( stack-diagram -- 1st )**

Set a private flag in stack-diagram. From now on, all appended data types are output parameters. 1st is stack-diagram. An exception is thrown if -- is preceded by -> or if it is used more than once within the same stack diagram.

-- is used in a stack diagram to separate input and output parameters.

**-> ( stack-diagram -- 1st )**

Add the prefix attribute to the data-type most recently appended to stack-diagram. 1st is stack-diagram. An exception is thrown if stack-diagram is still empty, if -> is preceded by --, if the prefix attribute is already set or if the most recently appended data type is a reference.

-> is used in a stack diagram to create compound data types as input or output parameters.

**-> ( x "<spaces>name" -- y ) immediate**

Skip leading space delimiters. Parse *name* delimited by a space. Convert *x* to *y*, where *x* is any data type and *y* is a compound data type created by appending the basic data type identified by *name* to *x*. An exception is thrown if *name* is not the name of a data type.

**-carry? ( -- flag )**

flag is true if and only if the directly preceding operation caused the processor's carry flag to be cleared.

**-i ( -- complex )**

complex.sf

complex is the complex floating-point literal with 0e0 as the real part and -1e0 as the imaginary part.

**-i\* ( complex -- 1st )**

complex.sf

Multiply complex by the negative value of the imaginary unit i, giving 1st.

**-leading ( caddress -> character unsigned -- 1st 3rd )**

strongforth.sf

If unsigned is greater than zero, 3rd is equal to unsigned less the number of spaces at the beginning of the character string specified by caddress -> character unsigned, and

1st is equal to caddress -> character plus the number of spaces at the beginning of the character string. If unsigned is zero, 3rd is zero and 1st is equal to caddress -> character.

**-leading ( caddress -> character unsigned 2nd -- 1st 3rd )**

strongforth.sf

If unsigned is greater than zero, 3rd is equal to unsigned less the number of characters equal to 2nd at the beginning of the character string specified by caddress -> character unsigned, and 1st is equal to caddress -> character plus the number of characters equal to 2nd at the beginning of the character string. If unsigned is zero, 3rd is zero and 1st is equal to caddress -> character.

**-loop ( do-destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. Resolve both the forward references and the backward reference of do-destination. Delete the loop index i. Rename the loop index j, if it exists, to i. An exception is thrown if the contents of the compiler data type heap do not exactly match the copy that was saved when do-destination was created.

Runtime: ( integer -- )

An ambiguous condition exists if the loop control parameters are unavailable. Subtract integer from the loop index. If the loop index crosses the boundary between the loop limit minus one and the loop limit, discard the current loop control parameters and continue execution. Otherwise, branch to the beginning of the loop.

Note: +loop takes regard of the data type of the loop index.

If the loop index is an address of a single cell, integer is multiplied with the size of a single cell in address units before it is subtracted from the loop index.

If the loop index is an address of a double cell, integer is multiplied with the size of a double cell in address units before it is subtracted from the loop index.

If the loop index is a character address, integer is multiplied with the size of a character in address units before it is subtracted from the loop index.

If the loop index is an address of a floating-point number, integer is multiplied with the size of a floating-point number in address units before it is subtracted from the loop index.

If the loop index is an address of a single-precision floating-point number, integer is multiplied with the size of a single-precision floating-point number in address units before it is subtracted from the loop index.

If the loop index is an address of a double-precision floating-point number, integer is multiplied with the size of a double-precision floating-point number in address units before it is subtracted from the loop index.

If the loop index is an address of a complex floating-point number, integer is multiplied with the size of a complex floating-point number in address units before it is subtracted from the loop index.

If the loop index is an address of a complex single-precision floating-point number, integer is multiplied with the size of a complex single-precision floating-point number in address units before it is subtracted from the loop index.



If the loop index is an address of a complex double-precision floating-point number, `integer` is multiplied with the size of a complex double-precision floating-point number in address units before it is subtracted from the loop index.

**-overflow? ( -- flag )**

`flag` is true if and only if the directly preceding operation caused the processor's overflow flag to be cleared.

**-trailing ( caddress -> character unsigned -- 1st 3rd )**

strongforth.sf

`1st` is equal to `caddress -> character`. If `unsigned` is greater than zero, `3rd` is equal to `unsigned` less the number of spaces at the end of the character string specified by `caddress -> character unsigned`. If `unsigned` is zero or the entire string consists of spaces, `3rd` is zero.

**-trailing ( caddress -> character unsigned 2nd -- 1st 3rd )**

strongforth.sf

`1st` is equal to `caddress -> character`. If `unsigned` is greater than zero, `3rd` is equal to `unsigned` less the number of characters equal to `2nd` at the end of the character string specified by `caddress -> character unsigned`. If `unsigned` is zero or the entire string consists of characters equal to `2nd`, `3rd` is zero.

**. ( character -- )**

strongforth.sf

If `character` is a graphic character in the ASCII character set, send `character` to the default output stream. The effect of `.` for all other values of `character` is undefined.

**. ( complex -- )**

complex.sf

Send the real part and the imaginary part of `complex` with a trailing space using fixed-point notation to the default output stream:

```
fixed-point notation := <re> + <im> i
<re>                 := <significand>
<im>                 := <significand>
<significand>        := [-]<digits>.<digits0>
<digits>              := <digit><digits0>
<digits0>             := <digit>*
<digit>               := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**. ( data-type -- )**

strongforth.sf

Send the name of `data-type` as a character string plus a trailing space to the default output stream. If `data-type` is a reference, send `1st`, `2nd`, `3rd` or `nth`, depending on the value `n` of the offset. An exception is thrown if `data-type` is not a valid data type.

**. ( definition -- )**

strongforth.sf

Send the name, the stack diagram and the attributes of `definition` to the default output stream.

**. ( double -- )** strongforth.sf

Send `double` as an unsigned double-precision number in free field format to the default output stream.

**. ( flag -- )** strongforth.sf

If `flag` is true, send `true` and a trailing space to the default output stream. If `flag` is false, Send `false` and a trailing space to the default output stream.

**. ( float -- )** float.sf

Send `float` with a trailing space using fixed-point notation to the default output stream:

```
fixed-point notation := <significand>
<significand>        := [-]<digits>.<digits0>
<digits>              := <digit><digits0>
<digits0>             := <digit>*
<digit>               := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix `base` is not (decimal) 10.

**. ( signed -- )** strongforth.sf

Send `signed` as a signed number in free field format to the default output stream.

**. ( signed-double -- )** strongforth.sf

Send `signed-double` as a signed double-precision number in free field format to the default output stream.

**. ( single -- )** strongforth.sf

Send `single` as an unsigned number in free field format to the default output stream.

**. ( vocabulary -- )** strongforth.sf

If `vocabulary` is the protected vocabulary of a class, send the class name plus a trailing space to the default output stream. Otherwise, send the name of the vocabulary to the default output stream.

**. " ( "ccc<quote>" -- ) compile-only** strongforth.sf

Parse `ccc` delimited by " (quote). Append the runtime semantics given below to the current definition.

Runtime: Send `ccc` to the default output stream.

**. " ( "ccc<quote>" -- ) execute-only** strongforth.sf

Parse `ccc` delimited by " (quote). Send `ccc` to the default output stream.

**.( ( "ccc<right-paren>" -- ) compile-only** strongforth.sf

Parse *ccc* delimited by *)* (right parenthesis). Append the runtime semantics given below to the current definition.

Runtime: Send *ccc* to the default output stream.

**.( ( "ccc<right-paren>" -- ) execute-only** strongforth.sf

Parse *ccc* delimited by *)* (right parenthesis). Send *ccc* to the default output stream.

**.addr ( address -- )** strongforth.sf

Send *address* in an eight-digit hexadecimal format with a trailing colon to the default output stream.

**.attributes ( definition -- )** strongforth.sf

If *definition* has been marked as immediate, send *immediate* plus a trailing space to the default output stream.

If *definition* has been marked as execute-only, send *execute-only* plus a trailing space to the default output stream.

If *definition* has been marked as compile-only, send *compile-only* plus a trailing space to the default output stream.

**.byte ( single -- )** strongforth.sf

Send the least significant byte of *single* in a two-digit hexadecimal format with no trailing space to the default output stream.

**.cell ( single -- )** strongforth.sf

Send *single* in an eight-digit hexadecimal format with no trailing space to the default output stream.

**.error ( signed -- )** strongforth.sf

Send the error message `Error nnn` to the default output stream, where *nnn* is the decimal value of *signed* as a signed number in free field format.

**.exponent ( signed -- )** float.sf

Send *signed* as a floating point exponent in the format *e<sub>s</sub>nnn* to the default output stream, where *s* is the sign (+ or -) and *nnn* is the absolute value of *signed* represented as a three-digit decimal value.

**.message ( signed -- )** strongforth.sf

Send an error message to the default output stream. If *signed* is between -399 and 0, obtain the message from the text file `StrongForth.msg`. If *signed* is between -511 and -400, obtain the

*MSVCRT* system error message with the index `|signed|-400`. An exception is thrown if `signed` is below `-511` or greater than zero.

**.name ( definition -- )**

strongforth.sf

If `definition` has a name, send the name plus a trailing space to the default output stream.

**.params ( address -> data-type unsigned -- )**

strongforth.sf

Send a list of `unsigned` data types starting at `address -> data-type`, including prefix and reference attributes, to the default output stream.

**.prefix ( data-type -- )**

strongforth.sf

If `data-type` has the prefix attribute, send `->` plus a trailing space to the default output stream.

**.r ( double integer -- )**

strongforth.sf

Send `double` as an unsigned double-precision number right aligned in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading spaces in a field as wide as necessary.

**.r ( signed integer -- )**

strongforth.sf

Send `signed` as a signed number right aligned in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading spaces in a field as wide as necessary.

**.r ( signed-double integer -- )**

strongforth.sf

Send `signed-double` as a signed double-precision number right aligned in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading spaces in a field as wide as necessary.

**.r ( single integer -- )**

strongforth.sf

Send `single` as an unsigned number right aligned in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading spaces in a field as wide as necessary.

**.s ( -- ) immediate**

strongforth.sf

Interpretation: Send the names of the data types on the interpreter data type heap, including prefix attributes, to the default output stream.

Compilation: Send the names of the data types on the compiler data type heap, including prefix attributes, to the default output stream.

Note: `.s` does not send the values of the items on the data stack to the default output stream.

**`.sign ( flag -- )`**

float.sf

If `flag` is `true`, send a minus sign (`-`) to the default output stream.

**`.sign+ ( flag -- )`**

float.sf

If `flag` is `true`, send a minus sign (`-`) to the default output stream. Otherwise, send a plus sign (`+`) to the default output stream.

**`.source ( -- )`**

strongforth.sf

Send the already parsed area of the input buffer of the default input stream to the default output stream.

**`/ ( complex complex -- 1st )`**

complex.sf

Divide the first `complex` by the second `complex`, giving the quotient `1st`. The result has the same data type as the dividend. An exception is thrown if the second `complex` is zero, or if the quotient lies outside of the range of a floating-point number.

**`/ ( complex float -- 1st )`**

complex.sf

Divide the complex floating-point number `complex` by the real floating-point number `float`, giving the quotient `1st`. The result has the same data type as the dividend.

**`/ ( float float -- 1st )`**

Divide the first `float` by the second `float`, giving the quotient `1st`. The result has the same data type as the dividend. An exception is thrown if the second `float` is zero, or if the quotient lies outside of the range of a floating-point number.

**`/ ( signed signed -- 1st )`**

Divide the first `signed` by the second `signed`, giving the signed quotient `1st`. An exception is thrown if the second `signed` is zero. If both operands differ in sign, the result returned will be the same as that returned by the phrase `swap s>d swap sm/rem nip`.

**`/ ( signed-double signed -- 1st )`**

strongforth.sf

Divide `signed-double` by `signed`, giving the signed double-precision quotient `1st`. An exception is thrown if `signed` is zero.

**`/ ( unsigned unsigned -- 1st )`**

Divide the first `unsigned` by the second `unsigned`, giving the unsigned quotient `1st`. An exception is thrown if the second `unsigned` is zero.

**/ ( unsigned-double unsigned -- 1st )**

Divide unsigned-double by unsigned, giving the unsigned double-precision quotient 1st. An exception is thrown if unsigned is zero.

**/10^n ( float signed -- 1st )**

float.sf

1st is equal to float divided by 10 raised to the power of signed. An ambiguous condition exists if signed is negative.

**/counted-string ( -- unsigned )**

strongforth.sf

unsigned is the maximum size of a counted string, in characters.

**/hold ( -- unsigned )**

strongforth.sf

unsigned is the size in characters of the pictured numeric output string buffer.

**/mod ( signed signed -- 2nd 1st )**

Divide the first signed by the second signed, giving the signed remainder 2nd and the signed quotient 1st. An exception is thrown if the second signed is zero. If both operands differ in sign, the result returned will be the same as that returned by the phrase swap s>d swap sm/rem.

**/mod ( signed-double signed -- 2nd 1st )**

strongforth.sf

Divide signed-double by signed, giving the signed single-precision remainder 2nd and the signed double-precision quotient 1st. An exception is thrown if signed is zero.

**/mod ( unsigned unsigned -- 2nd 1st )**

Divide the first unsigned by the second unsigned, giving the unsigned remainder 2nd and the unsigned quotient 1st. An exception is thrown if the second unsigned is zero.

**/mod ( unsigned-double unsigned -- 2nd 1st )**

Divide unsigned-double by unsigned, giving the unsigned single-precision remainder 2nd and the unsigned double-precision quotient 1st. An exception is thrown if unsigned is zero.

**/pad ( -- unsigned )**

strongforth.sf

unsigned is the size in characters of the scratch area pointed to by pad.

**/params ( -- unsigned )**

strongforth.sf

unsigned is the maximum number of basic data types in a stack diagram.

**/string ( caddress -> character unsigned -- 1st 3rd )**

Adjust the character string at `caddress -> character` with length unsigned by one character. The resulting character string, specified by `1st 3rd`, begins at `caddress -> character` plus one character and is unsigned minus one characters long.

**/string ( caddress -> character unsigned integer -- 1st 3rd )**

Adjust the character string at `caddress -> character` with length unsigned by integer characters. The resulting character string, specified by `1st 3rd`, begins at `caddress -> character` plus integer characters and is unsigned minus integer characters long.

Note: integer may be a negative value.

**0.r ( double integer -- )**

strongforth.sf

Send `double` as an unsigned double-precision number right aligned with leading zeros (if required) in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading zeros in a field as wide as necessary.

**0.r ( signed integer -- )**

strongforth.sf

Send `signed` as a signed number right aligned with leading zeros (if required) in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading zeros in a field as wide as necessary.

**0.r ( signed-double integer -- )**

strongforth.sf

Send `signed-double` as a signed double-precision number right aligned with leading zeros (if required) in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading zeros in a field as wide as necessary.

**0.r ( single integer -- )**

strongforth.sf

Send `single` as an unsigned number right aligned with leading zeros (if required) in a field `integer` characters wide to the default output stream. `integer` is assumed to be a signed number. If `integer` is not positive or the number of characters required is greater than `integer`, all digits are sent with no leading zeros in a field as wide as necessary.

**0< ( float -- flag )**

`flag` is true if and only if `float` is less than zero.

**0< ( signed -- flag )**

`flag` is true if and only if `signed` is less than zero.

**0< ( signed-double -- flag )**

flag is true if and only if signed-double is less than zero.

**0<= ( float -- flag )**

flag is true if and only if float is less than or equal to zero.

**0<= ( signed -- flag )**

flag is true if and only if signed is less than or equal to zero.

**0<= ( signed-double -- flag )**

flag is true if and only if signed-double is less than or equal to zero.

**0<> ( complex -- flag )**

complex.sf

flag is true if and only if either the real or the imaginary part or both of complex are not equal to zero.

**0<> ( double -- flag )**

flag is true if and only if double is not equal to zero.

**0<> ( float -- flag )**

flag is true if and only if float is not equal to zero.

**0<> ( single -- flag )**

flag is true if and only if single is not equal to zero.

**0= ( complex -- flag )**

complex.sf

flag is true if and only if both the real and the imaginary part of complex are equal to zero.

**0= ( double -- flag )**

flag is true if and only if double is equal to zero.

**0= ( float -- flag )**

flag is true if and only if float is equal to zero.

**0= ( single -- flag )**

flag is true if and only if single is equal to zero.

**0> ( float -- flag )**



flag is true if and only if float is greater than zero.

**0> ( signed -- flag )**

flag is true if and only if signed is greater than zero.

**0> ( signed-double -- flag )**

flag is true if and only if signed-double is greater than zero.

**0>= ( float -- flag )**

flag is true if and only if float is greater than or equal to zero.

**0>= ( signed -- flag )**

flag is true if and only if signed is greater than or equal to zero.

**0>= ( signed-double -- flag )**

flag is true if and only if signed-double is greater than or equal to zero.

**0i+ ( float -- complex )**

complex.sf

complex is the complex floating-point literal with float as the real part and 0e0 as the imaginary part.

**0i0 ( -- complex )**

complex.sf

complex is the complex floating-point literal with 0e0 as the real part and 0e0 as the imaginary part.

**1+ ( address -- 1st )**

Add one to address giving 1st.

**1+ ( address -> complex -- 1st )**

Add the number of address units per complex floating-point number to address -> complex, giving 1st.

**1+ ( address -> double -- 1st )**

Add the number of address units per double cell to address -> double, giving 1st.

**1+ ( address -> float -- 1st )**

Add the number of address units per floating-point number to address -> float, giving 1st.

**1+ ( address -> single -- 1st )**

Add the number of address units per cell to address -> single, giving 1st.

**1+ ( caddress -- 1st )**

Add the number of address units per character to caddress, giving 1st.

**1+ ( dfaddress -- 1st )**

Add the number of address units per double-precision floating-point number to dfaddress, giving 1st.

**1+ ( dfaddress -> complex -- 1st )**

Add the number of address units per complex double-precision floating-point number to dfaddress -> complex, giving 1st.

**1+ ( integer -- 1st )**

Add one to integer, giving 1st.

**1+ ( integer-double -- 1st )**

Add one to integer-double, giving 1st.

**1+ ( sfaddress -- 1st )**

Add the number of address units per single-precision floating-point number to sfaddress, giving 1st.

**1+ ( sfaddress -> complex -- 1st )**

Add the number of address units per complex single-precision floating-point number to sfaddress -> complex, giving 1st.

**1- ( address -- 1st )**

Subtract one from address, giving 1st.

**1- ( address -> complex -- 1st )**

Subtract the number of address units per complex floating-point number from address -> complex, giving 1st.

**1- ( address -> double -- 1st )**

Subtract the number of address units per double cell from address -> double, giving 1st.

**1- ( address -> float -- 1st )**

Subtract the number of address units per floating-point number from `address -> float`, giving `1st`.

**1- ( address -> single -- 1st )**

Subtract the number of address units per cell from `address -> single`, giving `1st`.

**1- ( caddress -- 1st )**

Subtract the number of address units per character from `caddress`, giving `1st`.

**1- ( dfaddress -- 1st )**

Subtract the number of address units per double-precision floating-point number from `dfaddress`, giving `1st`.

**1- ( dfaddress -> complex -- 1st )**

Subtract the number of address units per complex double-precision floating-point number from `dfaddress -> complex`, giving `1st`.

**1- ( integer -- 1st )**

Subtract one from `integer`, giving `1st`.

**1- ( integer-double -- 1st )**

Subtract one from `integer-double`, giving `1st`.

**1- ( sfaddress -- 1st )**

Subtract the number of address units per single-precision floating-point number from `sfaddress`, giving `1st`.

**1- ( sfaddress -> complex -- 1st )**

Subtract the number of address units per complex single-precision floating-point number from `sfaddress -> complex`, giving `1st`.

**1st ( stack-diagram -- 1st )**

`strongforth.sf`

Append a reference to the basic data type at the first position of the input parameter list as an input or output parameter to `stack-diagram`.

`1st` is used in a stack diagram to specify input or output parameters which should have exactly the same data type as the first data type in the input parameter list of the same definition.

An exception is thrown if the input parameter list is empty, or if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**2\* ( integer -- 1st )**

Multiply `integer` by 2 giving the product `1st`.

Note that `2*` may only be used on integer values. Use `lshift` for shifting bits to the left.

**2\* ( integer-double -- 1st )**

Multiply `integer-double` by 2 giving the product `1st`.

**2/ ( integer -- 1st )**

Divide `integer` by 2 giving the quotient `1st`.

Note that `2/` may only be used on unsigned numbers. Use `rshift` for shifting bits to the right.

**2/ ( integer-double -- 1st )**

Divide `integer-double` by 2 giving the quotient `1st`.

`integer-double` is assumed to be an unsigned numeric value.

**2/ ( signed -- 1st )**

Divide `signed` by 2 giving the quotient `1st`.

Note that `2/` may only be used on signed numbers. Use `rshift` for shifting bits to the right.

**2/ ( signed-double -- 1st )**

Divide `signed-double` by 2 giving the quotient `1st`.

**2literal ( single single -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: ( `single single --` )

Place the first `single` and then the second `single` on the stack. Both items have the same values and data types as were supplied at compilation time.

**2nd ( stack-diagram -- 1st )**

strongforth.sf

Append a reference to the basic data type at the second position of the input parameter list as an input or output parameter to `stack-diagram`.

`2nd` is used in a stack diagram to specify input or output parameters which should have exactly the same data type as the second data type in the input parameter list of the same definition. Since the index refers to the basic data types in the input parameter list, it is possible to build a reference to the tail of a compound data type representing an input parameter.

An exception is thrown if the input parameter list contains less than two basic data types, if the referenced data type is itself a reference, or if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**3rd ( stack-diagram -- 1st )**

strongforth.sf

Append a reference to the basic data type at the third position of the input parameter list, as an input or output parameter to *stack-diagram*.

*3rd* is used in a stack diagram to specify input or output parameters which should have exactly the same data type as the third data type in the input parameter list of the same definition. Since the index refers to the basic data types in the input parameter list, it is possible to build a reference to the tail of a compound data type representing an input parameter.

An exception is thrown if the input parameter list contains less than three basic data types, if the referenced data type is itself a reference, or if the internal storage for input and output parameters of *stack-diagram* is exceeded.

**: ( "<spaces>name" -- colon-definition )**

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name*, called a colon definition. Enter compilation state. Empty and unlock the compiler data type heap. Initialize the number of locals to zero. Start the current definition, producing *colon-definition*. Append the initiation semantics given below to the current definition.

Initiation: Continue execution.

The execution semantics of *name* will be determined by the words compiled into the body of the definition. The current definition cannot be found in the dictionary until it is finished or until the execution of *does>* or *;* code.

*name* Execution: ( -- )

Execute the definition *name*.

Note that the new definition does have no stack effects by default. Stack effects have to be specified separately if they are intended. By using a stack diagram phrase ( ... -- ... ) immediately following *:* and *name*, the new definition is modified to incorporate stack effects.

**:noname ( -- colon-definition colon-definition )**

strongforth.sf

Create a definition with no name, called a colon definition. Enter compilation state. Empty and unlock the compiler data type heap. Initialize the number of locals to zero. Start the current definition, producing *colon-definition* and a copy of it. Append the initiation semantics given below to the current definition.

Initiation: Continue execution.

The execution semantics of the definition will be determined by the words compiled into its body. The definition is incomplete until it is finished or until the execution of *does>* or *;* code, leaving one copy of *colon-definition* on the stack.

Execution: ( -- )

Execute the definition.

Note that the new definition does have no stack effects by default. Stack effects have to be specified separately if they are intended. By using a stack diagram phrase ( ... -- ... ) immediately following *:noname*, the new definition is modified to incorporate stack effects.

**; ( colon-definition -- ) compile-only**

Compilation: Append the runtime semantics given below to the current definition. End the current definition and enter interpretation state, consuming `colon-definition`. Empty the locals vocabulary. Lock the compiler data type heap. An exception is thrown if the contents of the compiler data type heap do not exactly match the output parameters of the current definition.

Runtime: If the compiler data type heap is not locked, return to the calling definition.

**`;code ( colon-definition - code-definition ) compile-only`**

asm.sf

Compilation: Append the runtime semantics given below to the current definition. End the current definition and enter interpretation state, consuming `colon-definition` and creating `code-definition`. Empty the locals vocabulary. Lock the compiler data type heap. An exception is thrown if the contents of the compiler data type heap does not exactly match the output parameters of the current definition. Subsequent characters in the parse area typically represent source code in assembly language, generating machine code.

Runtime: Specify the execution semantics of the most recent definition, referred to as *name*, as given below. An ambiguous condition exists if the most recent definition was not defined with `create` or a user-defined definition that calls `create`.

Initiation: Place the address of *name*'s data field in register `ebx`.

*name* Execution: ( -- )

Perform the machine code sequence that was generated following `;code`.

Note that new definitions do have no stack effects by default. Stack effects have to be specified separately if they are intended. By using a stack diagram phrase ( ... -- ... ) immediately following `;code`, the new definition is modified to incorporate stack effects. Specifying a stack diagram is mandatory, because at least the data type of *name*'s data field address has to be present. The data field address is always the last input parameter. The stack effect of *name* is defined by the stack diagram following `;code`, omitting the data field address.

**`< ( address 1st -- flag )`**

flag is true if and only if address is less than 1st.

**`< ( float 1st -- flag )`**

flag is true if and only if float is less than 1st.

**`< ( integer 1st -- flag )`**

flag is true if and only if integer is less than 1st. integer is assumed to be an unsigned numeric value.

**`< ( integer-double 1st -- flag )`**

flag is true if and only if integer-double is less than 1st. integer-double is assumed to be an unsigned numeric value.

**`< ( signed 1st -- flag )`**

flag is true if and only if signed is less than 1st.

**< ( signed-double 1st -- flag )**

flag is true if and only if signed-double is less than 1st.

**<# ( double -- number-double )**

strongforth.sf

Initialize pictured numeric output conversion. number-double is equal to double.

**<= ( address 1st -- flag )**

flag is true if and only if address is less than or equal to 1st.

**<= ( float 1st -- flag )**

flag is true if and only if float is less than or equal to 1st.

**<= ( integer 1st -- flag )**

flag is true if and only if integer is less than or equal to 1st. integer is assumed to be an unsigned numeric value.

**<= ( integer-double 1st -- flag )**

flag is true if and only if integer-double is less than or equal to 1st. integer-double is assumed to be an unsigned numeric value.

**<= ( signed 1st -- flag )**

flag is true if and only if signed is less than or equal to 1st.

**<= ( signed-double 1st -- flag )**

flag is true if and only if signed-double is less than or equal to 1st.

**<> ( complex 1st -- flag )**

complex.sf

flag is true if and only if complex is not equal to 1st.

**<> ( double 1st -- flag )**

flag is true if and only if double is not bit-by-bit identical with 1st.

**<> ( float 1st -- flag )**

flag is true if and only if float is not equal to 1st.

**<> ( single 1st -- flag )**

flag is true if and only if single is not bit-by-bit identical with 1st.

**<ack> ( -- character )**

ascii.sf

character is the ASCII “acknowledge” control character (code 6).

**<bel> ( -- character )**

ascii.sf

character is the ASCII “bell” control character (code 7).

**<bs> ( -- character )**

ascii.sf

character is the ASCII “backspace” control character (code 8).

**<can> ( -- character )**

ascii.sf

character is the ASCII “cancel” control character (code 24).

**<cr> ( -- character )**

ascii.sf

character is the ASCII “carriage return” control character (code 13).

**<dc1> ( -- character )**

ascii.sf

character is the ASCII “device control 1” control character (code 17).

**<dc2> ( -- character )**

ascii.sf

character is the ASCII “device control 2” control character (code 18).

**<dc3> ( -- character )**

ascii.sf

character is the ASCII “device control 3” control character (code 19).

**<dc4> ( -- character )**

ascii.sf

character is the ASCII “device control 4” control character (code 20).

**<del> ( -- character )**

ascii.sf

character is the ASCII “delete” control character (code 127).

**<dle> ( -- character )**

ascii.sf

character is the ASCII “data link escape” control character (code 16).

**<em> ( -- character )**

ascii.sf

character is the ASCII “end of medium” control character (code 25).



<b>&lt;enq&gt; ( -- character )</b>	ascii.sf
character is the ASCII “enquiry” control character (code 5).	
<b>&lt;eot&gt; ( -- character )</b>	ascii.sf
character is the ASCII “end of transmission” control character (code 4).	
<b>&lt;esc&gt; ( -- character )</b>	ascii.sf
character is the ASCII “escape” control character (code 27).	
<b>&lt;etb&gt; ( -- character )</b>	ascii.sf
character is the ASCII “end of transmission block” control character (code 23).	
<b>&lt;etx&gt; ( -- character )</b>	ascii.sf
character is the ASCII “end of text” control character (code 3).	
<b>&lt;ff&gt; ( -- character )</b>	ascii.sf
character is the ASCII “form feed” control character (code 12).	
<b>&lt;fs&gt; ( -- character )</b>	ascii.sf
character is the ASCII “file separator” control character (code 28).	
<b>&lt;gs&gt; ( -- character )</b>	ascii.sf
character is the ASCII “group separator” control character (code 29).	
<b>&lt;ht&gt; ( -- character )</b>	ascii.sf
character is the ASCII “horizontal tabulator” control character (code 9).	
<b>&lt;lf&gt; ( -- character )</b>	ascii.sf
character is the ASCII “line feed” control character (code 10).	
<b>&lt;nak&gt; ( -- character )</b>	ascii.sf
character is the ASCII “negative acknowledge” control character (code 21).	
<b>&lt;nul&gt; ( -- character )</b>	ascii.sf
character is the ASCII “null” control character (code 0).	

<b>&lt;rs&gt; ( -- character )</b>	ascii.sf
character is the ASCII “record separator” control character (code 30).	
<b>&lt;si&gt; ( -- character )</b>	ascii.sf
character is the ASCII “shift in” control character (code 15).	
<b>&lt;so&gt; ( -- character )</b>	ascii.sf
character is the ASCII “shift out” control character (code 14).	
<b>&lt;soh&gt; ( -- character )</b>	ascii.sf
character is the ASCII “start of header” control character (code 1).	
<b>&lt;stx&gt; ( -- character )</b>	ascii.sf
character is the ASCII “start of text” control character (code 2).	
<b>&lt;sub&gt; ( -- character )</b>	ascii.sf
character is the ASCII “substitute” control character (code 26).	
<b>&lt;syn&gt; ( -- character )</b>	ascii.sf
character is the ASCII “synchronous idle” control character (code 22).	
<b>&lt;us&gt; ( -- character )</b>	ascii.sf
character is the ASCII “unit separator” control character (code 31).	
<b>&lt;vt&gt; ( -- character )</b>	ascii.sf
character is the ASCII “vertical tabulator” control character (code 11).	
<b>= ( complex 1st -- flag )</b>	complex.sf
flag is true if and only if complex is equal to 1st.	
<b>= ( double 1st -- flag )</b>	
flag is true if and only if double is bit-by-bit identical with 1st.	
<b>= ( float 1st -- flag )</b>	
flag is true if and only if float is equal to 1st.	
<b>= ( single 1st -- flag )</b>	

flag is true if and only if single is bit-by-bit identical with 1st.

**> ( address 1st -- flag )**

flag is true if and only if address is greater than 1st.

**> ( float 1st -- flag )**

flag is true if and only if float is greater than 1st.

**> ( integer 1st -- flag )**

flag is true if and only if integer is greater than 1st. integer is assumed to be an unsigned numeric value.

**> ( integer-double 1st -- flag )**

flag is true if and only if integer-double is greater than 1st. integer-double is assumed to be an unsigned numeric value.

**> ( signed 1st -- flag )**

flag is true if and only if signed is greater than 1st.

**> ( signed-double 1st -- flag )**

flag is true if and only if signed-double is greater than 1st.

**>= ( address 1st -- flag )**

flag is true if and only if address is greater than or equal to 1st.

**>= ( float 1st -- flag )**

flag is true if and only if float is greater than or equal to 1st.

**>= ( integer 1st -- flag )**

flag is true if and only if integer is greater than or equal to 1st. integer is assumed to be an unsigned numeric value.

**>= ( integer-double 1st -- flag )**

flag is true if and only if integer-double is greater than or equal to 1st. integer-double is assumed to be an unsigned numeric value.

**>= ( signed 1st -- flag )**

flag is true if and only if signed is greater than or equal to 1st.

**>= ( signed-double 1st -- flag )**

flag is true if and only if signed-double is greater than or equal to 1st.

**>attributes ( data-type -- data-type-attributes )**

data-type-attributes is the data type attributes of data-type.

**>body ( definition -- address )**

If definition is a created definition, address is the address of its data field. Otherwise, address is null.

>body is a virtual method of the definition class.

**>class-attributes ( data-type -- class-attributes )**

If data-type is directly or indirectly derived from object, class-attributes is the class attributes of data-type. Otherwise, an exception is thrown and class-attributes is null.

**>context ( vocabulary -- )**

Remove vocabulary from both the context vocabulary list and the hidden vocabulary list. Make vocabulary the head of the context vocabulary list. An ambiguous condition exists if vocabulary was not included in one of the two vocabulary lists before >context is executed.

**>data-type ( data-type-attributes -- data-type )**

data-type is the data type associated with data-type-attributes. No additional attributes are set in data-type.

**>definition ( data-type -- created-definition )**

strongforth.sf

created-definition is the definition associated with data-type, that is used within stack diagrams. An exception is thrown if no such definition is found.

**>definition ( vocabulary -- created-definition )**

strongforth.sf

created-definition is the definition associated with vocabulary, that is used to move a vocabulary the top of the context vocabulary list. An exception is thrown if no such definition is found.

**>dt ( address -> data-type -- )**

In interpretation state (state is false), append the compound data type stored at address -> data-type to the interpreter data type heap. In compilation state (state is true), append the compound data type stored at address -> data-type to the compiler data type heap. An exception is thrown if the respective data type heap overflows.

**>dt ( data-type -- )**

In interpretation state (`state` is `false`), append the basic data type `data-type` to the interpreter data type heap. In compilation state (`state` is `true`), append the basic data type `data-type` to the compiler data type heap. An exception is thrown if the respective data type heap overflows.

**>flag ( single -- 1st )**

`1st` is equal to `false` if `single` is zero. `1st` is equal to `true` if `single` is not zero. This proprietary definition is required by the definition of `until` to convert a single cell occupying a register or a stack location into a flag that resides in the processor status register.

**>float ( caddress -> character unsigned -- float flag )**

float.sf

An attempt is made to convert the string specified by `caddress -> character unsigned` to internal floating-point representation. If the string represents a valid floating-point number according to the syntax below, `float` is its value and `flag` is `true`. If the string does not represent a valid floating-point number, `float` is undefined and `flag` is `false`.

A string of blanks is being treated as a special case representing zero.

```
convertible string := <significand>[<exponent>]
<significand>      := [<sign>]{<digits>[.<digits0>] | .<digits>}
<exponent>        := <marker><digits0>
<marker>          := {<e-form> | <sign>}
<e-form>          := <e-char>[<sign>]
<sign>            := { + | - }
<e-char>          := { D | d | E | e }
<digits>          := <digit><digits0>
<digits0>         := <digit>*
<digit>           := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

**>in ( -- address -> unsigned )**

`address -> unsigned` is the address of a cell containing the offset in characters from the start of the input buffer to the start of the parse area of the default input stream.

**>in ( input-stream -- address -> unsigned )**

`address -> unsigned` is the address of a cell containing the offset in characters from the start of the input buffer to the start of the parse area of `input-stream`.

**>number ( integer-double caddress -> character unsigned -- 1st 2nd 4 th )**

`1st` is the unsigned result of converting the characters within the string specified by `caddress -> character unsigned` into digits, and adding each into `integer-double` after multiplying `integer-double` by the number-conversion radix in base. Conversion continues left-to-right until a character that is not convertible, including any `+` or `-`, is encountered, or the string is entirely converted. `2nd` is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. `4 th` is the number of unconverted characters in the string.

**>r ( -- r-index ) compile-only**

strongforth.sf

Compilation: Create a local with the name `r@`. Append the runtime semantics given below to the current definition. `r-index` is the number of cells reserved for locals in the stack frame of the current definition after creating `r@`.

Runtime: ( `single --` ) or ( `double --` ) or ( `float --` ) or ( `complex --` )

Store `single` or `double` or `float` or `complex` into the local `r@`.

**>sign ( character -- signed )**

`signed` is +1 if `character` is equal to +, and -1 if `character` is equal to -. For all other values of `character`, `signed` is zero.

**>structure-attributes ( data-type -- structure-attributes )**

struct.sf

If `data-type` is directly or indirectly derived from `structure`, `structure-attributes` is the structure attributes of `data-type`. Otherwise, an exception is thrown and `structure-attributes` is null.

**>token ( definition data-type -- token )**

strongforth.sf

An exception is thrown if `data-type` is not a qualified token. `token` is the execution token of `definition`. An exception is thrown if the stack diagram of `definition` does not match the stack diagram represented by the qualified token `data-type` according to the rules of the StrongForth data type system. If `definition` does not have an execution token, or if matching the stack diagrams requires some register shuffling, a chunk of code is compiled to generate a valid execution token. This works in either compilation or interpretation state.

**>token ( definition deferred-definition -- token )**

strongforth.sf

`token` is the execution token of `definition`. An exception is thrown if the stack diagram of `definition` does not match the stack diagram of `deferred-definition` according to the rules of the StrongForth data type system. If `definition` does not have an execution token, or if matching the stack diagrams requires some register shuffling, a chunk of code is compiled to generate a valid execution token. This works in either compilation or interpretation state.

**>token ( definition virtual-definition -- token )**

strongforth.sf

`token` is the execution token of `definition`. An exception is thrown if the stack diagram of `definition` does not match the stack diagram of `virtual-definition`, with the last input parameter replaced with the data type of the class that is currently being defined, according to the rules of the StrongForth data type system. If `definition` does not have an execution token, or if matching the stack diagrams requires some register shuffling, a chunk of code is compiled to generate a valid execution token. This works in either compilation or interpretation state.

**? ( -- ) immediate**

strongforth.sf

Interpretation: ( `address -> x --` )

Send *x* to the default output stream by using a suitable version of `.. x` can be a single-cell or a double-cell item or a floating-point number or a complex floating-point number.

Compilation: Append the runtime semantics given below to the current definition.

Runtime: ( `address -> x --` )

Send *x* to the default output stream by using a suitable version of `.. x` can be a single-cell or a double-cell item or a floating-point number or a complex floating-point number.

**?alias ( code-definition -- )**

If the machine code instructions of `code-definition` consist of nothing else but a `call`, followed by a `ret`, instruction, remove this code sequence from the `code-space` memory space and make `code-definition` an alias for the definition that is actually being performed. `?alias` is a low-level internal optimization.

**?block ( unsigned -- 1st )**

block.sf

`1st` is unsigned. An exception is thrown if `unsigned` is not a valid block number between 1 and `#blocks`.

**?byte ( integer -- 1st )**

strongforth.sf

`1st` is integer. An exception is thrown if `integer` cannot be represented as an unsigned byte-size integer, i. e., its value is not between 0 and 255.

**?byte ( signed -- 1st )**

strongforth.sf

`1st` is signed. An exception is thrown if `signed` cannot be represented as a signed byte-size integer, i. e., its value is not between -128 and +127.

**?congruent ( definition -- )**

In interpretation state, compare the interpreter data type heap with the output parameters of `definition`. In compilation state, compare the compiler data type heap with the output parameters of `definition`. An exception is thrown if the data types do not exactly match. `?congruent` resolves data type references to the input parameters of `definition`.

**?create-vtable ( object-size -- )**

strongforth.sf

If the class that is currently being defined does not yet have a virtual method table, create a virtual method table in the `data-space` memory space and initialize it with `object-size` and the tokens of the parent class plus unassigned tokens for newly added virtual methods. Otherwise, just update the existing virtual method table with `object-size`.

**?created-definition ( definition -- created-definition )**

strongforth.sf

`created-definition` is `definition`. An exception is thrown if `definition` was not defined by `create`.

**?data-type ( caddress -> character unsigned -- data-type )**

Search all vocabularies for a definition with the name given by the string `caddress` -> `character unsigned` that was created by `procreates`. If such a definition is found, return the data type the definition is associated with as `data-type`. If no data type with this name is found, an exception is thrown and `data-type` is null.

**?data-type ( definition -- data-type )**

strongforth.sf

If `definition` was created by `procreates`, `data-type` is the data type it is associated with. Otherwise, an exception is thrown and `data-type` is null.

**?decimal ( -- )**

float.sf

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**?do ( -- do-destination ) compile-only**

strongforth.sf

Compilation: Create and initialize `do-destination`. Save a copy of the compiler data type heap. Rename the loop index `i` into `j`, if it already exists, and define a new local `i` as loop index. Append the runtime semantics given below to the current definition. The semantics are incomplete until resolved by a consumer of `do-destination` such as `loop`, `+loop` and `-loop`.

Runtime: ( `integer 1st --` ) or ( `address 1st --` )

If the limit `integer` or `address` is equal to the index `1st`, branch to the location given by the consumer of `do-destination`. Otherwise initialize the loop control parameters with limit `integer` or `address` and index `1st`, and continue execution.

**?loop ( -- local-definition )**

strongforth.sf

`local-definition` is the loop index of the innermost `do` loop. An exception is thrown if `?loop` is executed in interpretation state, if no local with name `i` exists or if this local has no enclosing `do` loop.

**?negate ( float signed -- 1st )**

float.sf

If `signed` is negative, change the sign of `float`, giving `1st`.

**?negate ( integer signed -- 1st )**

If `signed` is negative, `1st` is the arithmetic inverse of `integer`. Otherwise, `1st` is equal to `integer`.

**?negate ( integer-double signed -- 1st )**

If `signed` is negative, `1st` is the arithmetic inverse of `integer-double`. Otherwise, `1st` is equal to `integer-double`.

**?overflow ( address -- 1st )**



`1st` is equal to `address`. An exception is thrown if the preceding arithmetic operation on `address` caused an unsigned numeric overflow, i. e., the carry bit is set.

**?overflow ( integer -- 1st )**

`1st` is equal to `integer`. An exception is thrown if the preceding arithmetic operation on `integer` caused an unsigned numeric overflow, i. e., the carry bit is set.

**?overflow ( integer-double -- 1st )**

`1st` is equal to `integer-double`. An exception is thrown if the preceding arithmetic operation on `integer-double` caused an unsigned numeric overflow, i. e., the carry bit is set.

**?overflow ( signed -- 1st )**

`1st` is equal to `signed`. An exception is thrown if the preceding arithmetic operation on `signed` caused a two's complement numeric overflow, i. e., the overflow bit is set.

**?overflow ( signed-double -- 1st )**

`1st` is equal to `signed-double`. An exception is thrown if the preceding arithmetic operation on `signed-double` caused a two's complement numeric overflow, i. e., the overflow bit is set.

**?parse-byte ( "xx" -- character )**

escape.sf

Parse two hexadecimal digits `xx` and return the resulting two-digit ASCII code as `character`. If the parse area does not contain at least two characters or one or both of these characters is no hexadecimal digit, an exception is thrown and `character` is undefined.

**?parse-char ( "c" -- character )**

escape.sf

Parse `c` and return it as `character`. If the parse area is empty, an exception is thrown and `character` is null.

**?qualified-token ( data-type -- definition )**

strongforth.sf

`definition` is a definition with the name `execute` that executes a qualified token with data type `data-type`. An exception is thrown if no such definition exists.

**?range ( flag -- )**

strongforth.sf

An exception is thrown if `flag` is false.

**?single ( integer-double -- 1st )**

strongforth.sf

`1st` is `integer-double`. An exception is thrown if `integer-double` cannot be represented as an unsigned single-cell integer, i. e., its value is not between 0 and `max-unsigned`.

**?single ( signed-double -- 1st )**

strongforth.sf

1st is signed-double. An exception is thrown if signed-double cannot be represented as a signed single-cell integer, i. e., its value is not between max-signed negate 1- and max-signed.

**?stack-diagram ( stack-diagram -- )**

Throws an exception if stack-diagram is incomplete. A stack diagram is incomplete, if it does not contain -- or if it ends with ->.

**?to ( definition -- )**

strongforth.sf

In interpretation state, check if the compound data type on top of the interpreter data type heap matches the output parameter of definition. In compilation state, check if the compound data type on top of the compiler data type heap matches the output parameter of definition. An exception is thrown if the check fails. An ambiguous condition exists if definition is not either a value definition or a locals definition, which both have only one output parameter.

**?value-definition ( caddress -> character unsigned --  
value-definition )**

strongforth.sf

Search all vocabularies for an instance of class value-definition with the name given by the string caddress -> character unsigned. If such a definition is found, return it as value-definition. An exception is thrown if no instance of class value-definition with this name is found, and value-definition is null.

**@ ( address -> complex -- 2nd )**

2nd is the complex floating-point number stored at address -> complex.

**@ ( address -> double -- 2nd )**

2nd is the double-cell item stored at address -> double.

**@ ( address -> float -- 2nd )**

2nd is the floating-point number stored at address -> float.

**@ ( address -> single -- 2nd )**

2nd is the single-cell item stored at address -> single.

**@ ( caddress -> flag -- 2nd )**

2nd is the flag stored at caddress -> flag. Since the flag is assumed to have character size, while 2nd has cell size, it is extended to either false or true.

**@ ( caddress -> signed -- 2nd )**

2nd is the signed number stored at caddress -> signed. Since the number is assumed to have the character size, while 2nd has cell size, its value is sign extended.

**@ ( caddress -> single -- 2nd )**

2nd is the item stored at `caddress -> single`. Since the item is assumed to have character size, while 2nd has cell size, its value is extended with leading zero bits.

**@ ( dfaddress -> complex -- 2nd )**

2nd is the complex double-precision floating-point number stored at `dfaddress -> complex`.

**@ ( dfaddress -> float -- 2nd )**

2nd is the double-precision floating-point number stored at `dfaddress -> float`.

**@ ( sfaddress -> complex -- 2nd )**

2nd is the complex single-precision floating-point number stored at `sfaddress -> complex`.

**@ ( sfaddress -> float -- 2nd )**

2nd is the single-precision floating-point number stored at `sfaddress -> float`.

**abort ( -- )**

strongforth.sf

Throw an exception with code -1.

**abort" ( "ccc<delimiter>" -- ) compile-only**

strongforth.sf

Parse `ccc` delimited by a quote (`"`). Append the runtime semantics given below to the current definition.

Runtime: ( `single --` )

If `single` is not equal to zero, copy the string `ccc` to line, fill the remainder of line with spaces and throw an exception with code -2.

**abs ( complex -- float )**

complex.sf

`float` is the absolute value of the complex floating-point number `complex`.

**abs ( float -- 1st )**

`1st` is the absolute value of `float`.

**abs ( integer -- 1st )**

`1st` is the absolute value of `integer`. `integer` is assumed to be a signed numeric value.

**abs ( integer-double -- 1st )**

1st is the absolute value of integer-double. integer-double is assumed to be a signed numeric value.

**abs\*\*2 ( complex -- float )**

complex.sf

float is the square of the absolute value of the complex floating-point number complex.

**accept ( caddress -> character integer -- 3rd )**

Receive a character string of at most integer characters at caddress -> character from the user input device. Send graphic characters to the user output device as they are received. The usual editing functions that the system performs in order to construct the character string (backspace etc.), might be used.

Input terminates when a carriage return character is received. When input terminates, nothing is appended to the character string.

3rd is the length of the character string stored at caddress -> character.

accept is a deferred word.

**access ( object-size "<spaces>name" -- 1st )**

strongforth.sf

Skip leading space delimiters. Parse name delimited by a space. Find class name. Add the word list that combines the private and protected vocabularies of class name to the list of context vocabularies. An exception is thrown if name is not the name of a class the class currently being defined is a friend to.

object-size is a dummy parameter that ensures that access is always used within the body of a class definition. 1st is equal to object-size.

**acos ( complex -- 1st )**

complex.sf

1st is the principal radian angle whose cosine is complex. This operation is based on complex floating-point numbers.

**acos ( float -- 1st )**

float.sf

1st is the principal radian angle whose cosine is float. An ambiguous condition exists if the absolute value of float is greater than one.

**acosh ( complex -- 1st )**

complex.sf

1st is the complex floating-point value whose hyperbolic cosine is complex. This operation is based on complex floating-point numbers.

**acosh ( float -- 1st )**

float.sf

1st is the floating-point value whose hyperbolic cosine is float. An ambiguous condition exists if float is less than one.

**action-of ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Find a deferred definition with the name *name*. Append the run-time semantics given below to the current definition. An exception is thrown if a deferred definition with the name *name* does not exist.

Run-time: ( -- token )

token is the execution token that the deferred definition *name* is set to execute.

**action-of ( "<spaces>name" -- token ) execute-only**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find a deferred definition with the name *name*. token is the execution token that the deferred definition *name* is set to execute. An exception is thrown if a deferred definition with the name *name* does not exist.

**add-origin ( control-flow origin -- )**

Add *origin* to the linked list kept by *control-flow*. *origin* will be resolved and deleted recursively together with *control-flow*.

**address ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *address*.

**address-unit-bits ( -- unsigned )**

strongforth.sf

*unsigned* is 8, the number of bits in each address unit.

**again ( destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition, resolving the backward reference *destination*. Delete *destination*. Lock the compiler data type heap. An exception is thrown if the contents of the compiler data type heap do not exactly match the copy that was saved when *destination* was created.

Runtime: Continue execution at the location specified by *destination*. If no other control flow words are used, any program code after *again* will never be executed.

**ahead ( -- origin ) compile-only**

Compilation: Put a new unresolved forward reference *origin* onto the stack and save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition. Lock the compiler data type heap. The semantics are incomplete until *origin* is resolved.

Runtime: Continue execution at the location specified by the resolution of *origin*. If no other control flow words are used, the program code immediately following *ahead* will never be executed.

**align ( -- )**

If the first unused address of the default memory space is not cell aligned, reserve the required minimum number of address units to make it cell aligned.

**align ( memory-space -- )**

If the first unused address of `memory-space` is not cell aligned, reserve the required minimum number of address units to make it cell aligned.

**aligned ( address -- 1st )**

`1st` is the lowest cell aligned address greater than or equal to `address`.

**aligned ( unsigned unsigned -- 1st )**

`1st` is the lowest unsigned number greater than or equal to the first unsigned, that is a multiple of the second unsigned.

**allocate ( unsigned -- address )**

Allocate unsigned address units of contiguous dynamic memory space. The initial content of the allocated memory space is undefined. If the allocation succeeds, `address` is the aligned starting address of the allocated memory space. An exception is thrown if the operation fails.

**allocate-counted-string ( caddress -> character unsigned -- 1st )**

strex.sf

Allocate unsigned plus 1 characters of dynamic memory. Copy the character string `caddress -> character unsigned` as a counted string in the allocated memory space. `1st` is the address of the counted string.

**allot ( integer -- )**

If `integer` is greater than zero, reserve `integer` address units of the default memory space. If `integer` is less than zero, release `|integer|` address units of the default memory space. If `integer` is zero, leave the default memory space unchanged.

If the first unused address of the default memory space is cell aligned and `integer` is a multiple of cell size in address units prior to execution of `allot`, it will remain cell aligned when `allot` finishes execution.

If the first unused address of the default memory space is character aligned and `integer` is a multiple of character size in address units prior to execution of `allot`, it will remain character aligned when `allot` finishes execution.

**allot ( integer memory-space -- )**

If `integer` is greater than zero, reserve `integer` address units of `memory-space`. If `integer` is less than zero, release `|integer|` address units of `memory-space`. If `integer` is zero, leave `memory-space` unchanged.

If the first unused address of `memory-space` is cell aligned and `integer` is a multiple of cell size in address units prior to execution of `allot`, it will remain cell aligned when `allot` finishes execution.

If the first unused address of `memory-space` is character aligned and `integer` is a multiple of character size in address units prior to execution of `allot`, it will remain character aligned when `allot` finishes execution.

**alog ( complex -- 1st )**

complex.sf

Raise ten to the power `complex`, giving `1st`. This operation is based on complex floating-point numbers.

**alog ( float -- 1st )**

Raise ten to the power `float`, giving `1st`.

**alpha-numeric ( -- )**

strongforth.sf

Set the number-conversion radix to 36 (alpha-numeric).

**also ( -- )**

order.sf

In StrongForth, this definition has no semantics.

**ancestor? ( data-type data-type -- flag )**

strongforth.sf

`flag` is true if and only if the second `data-type` is equal to the first `data-type`, or if the second `data-type` is directly or indirectly derived from the first `data-type`.

**and ( data-type data-type -- 1st )**

`1st` is the first `data-type` with attributes that are the bit-by-bit logical and of the attributes of both parameters `data-type`.

**and ( object-size object-size object-size -- 1st 2nd 3rd )**

strongforth.sf

Terminate a block of a union of members within a class definition, and start a new one. `1st` and `3rd` are equal to the first `object-size`. `2nd` is the maximum of the second and the third `object-size`.

**and ( single logical -- 1st )**

`1st` is the bit-by-bit logical and of `single` and `logical`.

**any: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to any register or register pair. An exception is thrown if the input or output parameter does not fit into a register or a register pair.

**arg ( complex -- float )**

complex.sf

`float` is the angle of the complex floating-point number `complex`.

**asin ( complex -- 1st )**

complex.sf

1st is the principal radian angle whose sine is complex. This operation is based on complex floating-point numbers.

**asin ( float -- 1st )**

float.sf

1st is the principal radian angle whose sine is float. An ambiguous condition exists if the absolute value of float is greater than one.

**asinh ( complex -- 1st )**

complex.sf

1st is the floating-point value whose hyperbolic sine is complex. This operation is based on complex floating-point numbers.

**asinh ( float -- 1st )**

float.sf

1st is the floating-point value whose hyperbolic sine is float. An ambiguous condition exists if float is less than zero.

**assembler ( -- ) immediate**

Remove the assembler vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the assembler vocabulary the head of the context vocabulary list. An ambiguous condition exists if the assembler vocabulary was not included in one of the two vocabulary lists before assembler is executed.

**assign ( complex complex-definition -- )**

Assigns the value of complex to complex-definition. complex-definition will from now on compile the value of complex as a literal.

**assign ( double double-definition -- )**

Assigns the value of double to double-definition. double-definition will from now on compile the value of double as a literal.

**assign ( float float-definition -- )**

Assigns the value of float to float-definition. float-definition will from now on compile the value of float as a literal.

**assign ( single single-definition -- )**

Assigns the value of single to single-definition. single-definition will from now on compile the value of single as a literal.

**at-xy ( unsigned unsigned -- )**

Within the console window, position the cursor at the column number specified by the first unsigned and the row number specified by the second unsigned. The next character displayed



will appear at this position. The upper left corner of the console window has column zero and row zero. An exception is thrown if invalid column or row numbers are specified.

**atan ( complex -- 1st )**

complex.sf

1st is the principal radian angle whose tangent is complex. This operation is based on complex floating-point numbers.

**atan ( float -- 1st )**

float.sf

1st is the principal radian angle whose tangent is float.

**atan2 ( float float -- 1st )**

1st is the radian angle whose tangent is the first float divided by the second float. An ambiguous condition exists if both parameters float are zero.

**atanh ( complex -- 1st )**

complex.sf

1st is the floating-point value whose hyperbolic tangent is complex. This operation is based on complex floating-point numbers.

**atanh ( float -- 1st )**

float.sf

1st is the floating-point value whose hyperbolic tangent is float. An ambiguous condition exists if the absolute value of float is greater than one.

**attributes! ( logical definition -- )**

Set the attributes given by logical within definition. Note that there's no direct way to clear attributes of definition once they have been set.

**attributes? ( data-type data-type -- flag )**

strongforth.sf

flag is true if and only if at least one of the attributes of the first data-type is set in the second data-type as well.

**attributes? ( logical definition -- flag )**

flag is true if and only if at least one of the attributes given by logical is set within definition.

**base ( -- caddress -> unsigned )**

caddress -> unsigned is the address of the current number-conversion radix (2...36).

**begin ( -- destination ) compile-only**

strongforth.sf

Compilation: Create and initialize destination and save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition.

Runtime: Continue execution.

**begin-compilation ( -- )**

Enter compilation state. Empty and unlock the compiler data type heap. Set `#locals` to zero.

**begin-loop ( local-definition -- do-destination )**

strongforth.sf

Create and initialize `do-destination` and save a copy of the compiler data type heap. Make `local-definition` the loop index by assigning `do-destination` as the destination of the associated `do` loop.

**begin-structure ( "<spaces>name" -- structure-attributes object-size )**

struct.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. The definition identifies a new data type that is a direct subtype of `structure`. Assign the attributes of the new data type to `this-attributes`. `structure-attributes` is the data type attributes associated with the new data type. `object-size` is zero.

*name* Execution: ( `stack-diagram` -- `1st` )

When used in a stack diagram, specifies an input or output parameter with the new data type.

**bin ( fam -- 1st )**

strongforth.sf

Modify `fam` to additionally select a “binary” file access method, and return it as `1st`.

**binary ( -- )**

strongforth.sf

Set the number-conversion radix to 2 (binary).

**bit ( unsigned -- logical )**

strongforth.sf

The bit at position `unsigned` of `logical` is 1. All other bits of `logical` are 0.

**b1 ( -- character )**

strongforth.sf

`character` is the space character.

**blank ( caddress -> character unsigned -- )**

strongforth.sf

If `unsigned` is greater than zero, store the character value for space in `unsigned` consecutive character positions beginning at `ccaddress -> character`.

**blk ( block-input-stream -- address -> unsigned )**

block.sf

`address -> unsigned` is the address of a cell containing the number of the block being interpreted.

`blk` is a member of the `block-input-stream` class.

**blk# ( -- address -> unsigned )**

block.sf

`address -> unsigned` is the address of a cell containing the number of the block that is stored in the block buffer, or zero if the block buffer is currently unused.

**block ( unsigned -- caddress -> character )**

block.sf

If the block buffer is unassigned, transfer block `unsigned` from the block file to the block buffer.

If block `unsigned` is not already in the block buffer, and the block buffer is assigned but has not been modified, transfer block `unsigned` from the block file to the block buffer.

If block `unsigned` is not already in the block buffer, and the block buffer is assigned and has been modified, transfer the block to the block file and then transfer block `unsigned` from the block file to the block buffer.

Assign block `unsigned` to the block buffer. `caddress -> character` is the address of the block buffer. An exception is thrown if `unsigned` is not a valid block number.

**block-buffer ( -- caddress -> character )**

block.sf

`caddress -> character` is the address of a single buffer for c/b characters, called block buffer. This buffer is always used when transferring a block from or to the block file.

**block-file ( -- file )**

block.sf

`file` is the file containing all blocks.

**block-input-stream ( stack-diagram -- 1st )**

block.sf

When used in a stack diagram, specifies an input or output parameter with data type `block-input-stream`.

**block-input-stream ( unsigned block-input-stream -- 2nd )**

block.sf

Initialize `block-input-stream` by erasing all members. Make `unsigned` the number of the block to be interpreted. Transfer block `unsigned` from the block file to the block buffer. Make the block buffer the input buffer. `2nd` is `block-input-stream`. An exception is thrown if `unsigned` is not a valid block number.

`block-input-stream` is a constructor of the `block-input-stream` class.

**bmember-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `bmember-definition`.

**bmember-definition ( unsigned unsigned caddress -> character  
unsigned member-definition -- 6 th )**

Initialize `bmember-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `bmember-`

definition is deleted. The first unsigned is the length of the new member in bits. The second unsigned is the position of the new member in bits with respect to the start address of the object. Assign `bmember-definition` a name given by the character string `address -> character unsigned` and return it as 6 th.

`bmember-definition` is the constructor of the `bmember-definition` class.

**body! ( address created-definition -- )**

Specifies `address` as the data field of `created-definition`.

`body!` is a method of class `created-definition`.

**body-criterion ( -- search-criterion )**

strongforth.sf

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

`flag` is true if and only if the value of `single` is the data field of `definition`.

Note: Provide `search-criterion` to search in order to find a created definition with a specific data field.

**break ( -- )**

Breaks execution. Used for debugging purposes only.

**buffer ( unsigned -- caddress -> character )**

block.sf

If block `unsigned` is not already in the block buffer, and the block buffer is assigned and has been modified, transfer the block to the block file.

Assign block `unsigned` to the block buffer. `caddress -> character` is the address of the block buffer. An exception is thrown if `unsigned` is not a valid block number.

**buffer: ( unsigned "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a new definition for `name` with the execution semantics defined below, and make it the latest definition. Allocate `unsigned` address units in the definition's data field.

*name* Execution: Execute the definition `name`. The default execution semantics of the new definition is placing the address of its data field onto the stack.

Note that the stack diagram of the new definition has to be explicitly specified. The execution semantics may be extended by `does>` or `; code`.

**bye ( -- )**

Terminate StrongForth with result code zero and return control to the operating system.

**bye ( integer -- )**

Terminate StrongForth with result code `integer` and return control to the operating system.

**byte? ( integer -- flag )**

`flag` is true if and only if `integer` can be represented as an unsigned byte-size integer, i. e., its value is between 0 and 255.

**byte? ( signed -- flag )**

`flag` is true if and only if `signed` can be represented as a signed byte-size integer, i. e., its value is between -128 and +127.

**c, ( single -- )**

Reserve space for one character in the default memory space and store `single` truncated to character size in it. If the default memory space is character aligned when `c,` begins execution, it will remain character aligned when `c,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not character aligned prior to execution of `c,`. An exception is thrown if the default memory space overflows.

**c, ( single memory-space -- )**

Reserve space for one character in `memory-space` and store `single` truncated to character size in it. If `memory-space` is character aligned when `c,` begins execution, it will remain character aligned when `c,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not character aligned prior to execution of `c,`. An exception is thrown if `memory-space` overflows.

**c/b ( -- unsigned )**

block.sf

`unsigned` is 1024, the number of characters per block.

**c/l ( -- unsigned )**

strongforth.sf

`unsigned` is 64, the number of characters in a line of a block.

**caddress ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `caddress`.

**callocate ( unsigned -- caddress )**

Allocate `unsigned` address units of contiguous dynamic memory space. The initial content of the allocated memory space is undefined. If the allocation succeeds, `caddress` is the aligned starting address of the allocated memory space. An exception is thrown if the operation fails.

**carry? ( -- flag )**

`flag` is true if and only if the directly preceding operation caused the processor's carry flag to be set.

**case ( -- endof-origin of-origin ) compile-only**

strongforth.sf

Compilation: Mark the start of a `case ... of ... endof ... endcase` structure by putting `endof-origin` and `of-origin` onto the stack. Save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition.

Runtime: Continue execution.

**cast ( x "<spaces>name" -- y ) immediate**

Skip leading space delimiters. Parse *name* delimited by a space. An exception is thrown if *name* is not the name of a data type.

Convert *x* to *y*, where *x* is any data type and *y* is the data type identified by *name*. If *x* and *y* have the same size, the actual bit image is not changed. If *x* and *y* have different sizes, `cast` uses one of the following conversion words to adjust the size and bit image of *y*: `s>d`, `s>f`, `d>s`, `d>f`, `f>s` or `f>d`. Complex numbers are converted to scalar numbers by calculating their absolute value. Scalar numbers are converted to complex numbers as their real part, while the imaginary part becomes zero. These are conversions applied to different combinations of source and destination data types:

x↓ y→	single	double	float	complex
single	noop	s>d	s>f	s>f 0i+
double	d>s	noop	d>f	d>f 0i+
float	f>s	f>d	noop	0i+
complex	abs f>s	abs f>d	abs	noop

**catch ( "<spaces>name" -- ) compile-only**

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Search the context vocabularies for a definition with the name *name*, whose input parameters match the compiler data type heap according to the rules of the StrongForth data type system. An exception is thrown if no matching definition is found. Append the runtime semantics given below to the current definition.

Runtime: Create and initialize a new exception frame. Execute *name*. Obtain the error code from the current exception frame. Delete the current exception frame.

**cells ( integer -- 1st )**

*1st* is the size in address units of `integer` cells.

**changed ( stack-diagram -- 1st )**

During specification of the stack diagram `stack-diagram` of a definition, mark a list of registers as being destroyed by the definition. *1st* is equal to `stack-diagram`. Registers have to be considered only when programming in assembler.

**char ( "<spaces>name" -- character )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. *character* is the value of *name*'s first character. If the length of the parsed area is zero, *character* is the space character.

**character ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type character.

**chars ( integer -- 1st )**

1st is the size in address units of integer characters.

**chere ( -- caddress )**

caddress is the first unused address of the default memory space.

**chere ( memory-space -- caddress )**

caddress is the first unused address of memory-space.

**class ( "<spaces>name" -- vocabulary object-size )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find the data type associated with *name* and assign its data type attributes to *this-attributes*. An exception is thrown if *name* is not a class, or if the parent of *name* has not yet been defined. Initialize the virtual method table length, the protected vocabulary and the private vocabulary. *vocabulary* is the current compilation vocabulary. *object-size* is the size in bits of objects of the parent class, or zero if an exception is thrown.

class starts a class definition.

**class-attributes ( data-type unsigned class-attributes -- 3rd )**

Initialize *class-attributes* by erasing all members. Store the attributes of *data-type* as the parent of the class associated with *class-attributes*. Store *unsigned* as the size of the class associated with *class-attributes*.

*class-attributes* is the constructor of the *class-attributes* class.

**class-attributes ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *class-attributes*.

**clear? ( single logical -- flag )**

flag is true if and only if the bit-by-bit logical and of *single* and *logical* is equal to zero.

**close ( file -- )**

Close the file identified by *file*.

**cmember ( object-size single "<spaces>name" -- 1st )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to object-size aligned to character size, plus the number of bits in one character.

*name* is referred to as a class member. *cmember* reserves a character-size class member of the same data type as *single* in the class that is currently being defined.

Execution: ( *x* -- *caddress* -> *y* )

*caddress* -> *y* is the address of a character-size class member of the object *x*, that were reserved at the time *name* was created. *y* is the actual data type that was provided to *cmember* as *single*.

***cmembers* ( object-size single unsigned "<spaces>name" -- 1st )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to object-size aligned to character size, plus unsigned times the number of bits in one character.

*name* is referred to as a class member. *cmembers* reserves unsigned characters for an array of unsigned character-size class members of the same data type as *single* in the class that is currently being defined.

Execution: ( *x* -- *caddress* -> *y* )

*caddress* -> *y* is the address of an array of unsigned character-size class members of the object *x*, that were reserved at the time *name* was created. *y* is the actual data type that was provided to *cmembers* as *single*.

***cmove* ( *caddress* 1st unsigned -- )**

If unsigned is greater than zero, copy unsigned consecutive characters starting at *caddress* to that starting at *1st*, proceeding character-by-character from lower addresses to higher addresses.

***cmove>* ( *caddress* 1st unsigned -- )**

If unsigned is greater than zero, copy unsigned consecutive characters starting at *caddress* to that starting at *1st*, proceeding character-by-character from higher addresses to lower addresses.

***code* ( "<spaces>name" - code-definition )**

asm.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a code definition for *name* with the execution semantics defined below, and make it the latest definition. The runtime code of the code definition begins at the first unused address of the code-space memory space. An ambiguous condition exists if *code* is executed in compilation state.

Subsequent words in the parse area typically represent source code in assembly language, generating machine code. The new code definition is not automatically added to the current compilation word list, This can be achieved by executing *endcode* after the last machine code instruction.

*name* Execution: Execute the definition *name*.



Note that the new code definition does have no stack effects by default. Stack effects have to be specified separately if they are intended. By specifying a stack diagram ( ... -- ... ) immediately following `code` and the definition name, the new definition is modified to incorporate stack effects.

**`code-definition ( caddress -> character unsigned code-definition -  
- 4 th )`**

Initialize `code-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `code-definition` is deleted. Store the first unused address of the code-space memory space as the address of the first machine code instruction of `code-definition`. Assign `code-definition` a name given by the character string `caddress -> character unsigned` and return it as `4 th`.

`code-definition` is a constructor of the `code-definition` class.

**`code-definition ( code-definition -- 1st )`**

Initialize `code-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `code-definition` is deleted. Store the first unused address of the code-space memory space as the address of the first machine code instruction of `code-definition`. Return `code-definition` as `1st`.

`code-definition` is a constructor of the `code-definition` class.

**`code-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `code-definition`.

**`code-definition ( virtual-definition token code-definition -- 3rd`**      `strongforth.sf`  
**`)`**

Initialize `code-definition` by erasing all members. Copy the attributes of `virtual-definition`. Store `token` as the address of the first machine code instruction of `code-definition`. Return `code-definition` as `3rd`.

`code-definition` is a constructor of the `code-definition` class.

**`code-space ( -- memory-space )`**

`memory-space` is the system's code space. Executable code is stored in the code space.

**`colon-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `colon-definition`.

**`compare ( caddress -> character unsigned 1st 3rd -- signed )`**

Compare the string specified by `caddress -> character unsigned` to the string specified by `1st 3rd`. The strings are compared, beginning at the given addresses, character by character, up to the length of the shorter string or until a difference is found.

If the two strings are identical, `signed` is zero.

If the two strings are identical up to the length of the shorter string, `signed` is -1 if `unsigned` is less than `3rd`, and +1 otherwise.

If the two strings are not identical up to the length of the shorter string, `signed` is -1 if the first non-matching character in the string specified by `caddress -> character unsigned` has a lesser numeric value than the corresponding character in the string specified by `1st 3rd`, and +1 otherwise.

#### **`compile, ( definition -- )`**

In interpretation state, apply the stack effect of `definition` to the interpreter data type heap and execute the semantics of `definition`.

In compilations state, apply the stack effect of `definition` to the compiler data type heap and append the semantics of `definition` to the runtime semantics of the current definition.

#### **`compile-only ( -- )`**

strongforth.sf

Make the latest definition a compile-only word. The interpreter finds this definition only if in compilation state, and executes it like an immediate word.

#### **`compiler-workspace ( compiler-workspace -- 1st )`**

Initialize `compiler-workspace` by emptying its data type reference table. In interpretation state (`state` is `false`), save the interpreter data type heap pointer. In compilation state (`state` is `false`), save the compiler data type heap pointer.

`compiler-workspace` is the constructor of the `compiler-workspace` class.

#### **`compiler-workspace ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `compiler-workspace`.

#### **`complex ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `complex`.

#### **`complex-definition ( caddress -> character unsigned complex-definition -- 4 th )`**

Initialize `complex-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `complex-definition` is deleted. Assign `complex-definition` a name given by the character string `caddress -> character unsigned` and return it as 4 th.

`complex-definition` is a constructor of the `complex-definition` class.

**complex-definition ( complex-definition -- 1st )**

Initialize `complex-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `complex-definition` is deleted.

`complex-definition` is a constructor of the `complex-definition` class.

**complex-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `complex-definition`.

**conj ( complex -- 1st )**

`complex.sf`

`1st` is the conjugate complex number of the complex floating-point number `complex`.

**constant ( complex "<spaces>name" -- )**

`complex.sf`

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. *name* is referred to as a constant.

Execution: ( -- x )

Place *x* on the stack. *x* has the same data type as was supplied to `constant` as `complex`.

**constant ( double "<spaces>name" -- )**

`strongforth.sf`

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. *name* is referred to as a constant.

Execution: ( -- x )

Place *x* on the stack. *x* has the same data type as was supplied to `constant` as `double`.

**constant ( float "<spaces>name" -- )**

`float.sf`

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. *name* is referred to as a constant.

Execution: ( -- x )

Place *x* on the stack. *x* has the same data type as was supplied to `constant` as `float`.

**constant ( single "<spaces>name" -- )**

`strongforth.sf`

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. *name* is referred to as a constant.

Execution: ( -- x )

Place *x* on the stack. *x* has the same data type as was supplied to `constant` as `single`.

**context ( -- address -> vocabulary )**

`address -> vocabulary` is the address of an object indicating the vocabulary that is searched first by `search-context` and all words using it. This vocabulary is actually the head of a linked list of vocabularies that are searched in the given order. The list of vocabularies is referred to as the context vocabulary list.

#### **control-flow ( control-flow -- 1st )**

Initialize `control-flow` by erasing all members. If the compiler data type heap is not locked, save a copy of the present compiler data type heap. Save the value of the `code-space` memory space pointer as the code origin or code destination.

`control-flow` is the constructor of the `control-flow` class.

#### **control-flow ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `control-flow`.

#### **copy ( object 1st -- )**

Copy all members of `object` to `1st`. If `object` and `1st` have different memory sizes, the minimum number of members are copied and the remaining members are not copied or remain unchanged.

#### **cos ( complex -- 1st )**

`complex.sf`

`1st` is the complex cosine of the radian angle `complex`.

#### **cos ( float -- 1st )**

`1st` is the cosine of the radian angle `float`.

#### **cosh ( complex -- 1st )**

`complex.sf`

`1st` is the complex hyperbolic cosine of `complex`.

#### **cosh ( float -- 1st )**

`float.sf`

`1st` is the hyperbolic cosine of `float`.

#### **count ( caddress -> character -- 1st unsigned )**

`strect.sf`

Return the character string specification for the counted string stored at `caddress -> character`. `1st` is the address of the first character after `caddress -> character`. `unsigned` is the numeric value of the character at `caddress -> character`, which is the length in characters of the string at `1st`.

#### **cr ( -- )**

Send a carriage return character followed by a line feed character to the default output stream.

**create ( "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `create` does not allocate memory in the definition's data field.

*name* Execution: Execute the definition *name*. The default execution semantics of the new definition is placing the address of its data field onto the stack.

Note that the stack diagram of the new definition has to be explicitly specified. The execution semantics may be extended by `does>` or `;code`.

**create ( caddress -> character unsigned fam -- file )**

strongforth.sf

Create a file with the name given by the character string `ccaddress -> character unsigned`, and open it as `file` with file access method `fam`. If a file with the same name already exists, recreate it as an empty file. An exception is thrown if `r/o` is specified as the file access method or if the operation fails.

**create-index ( -- local-definition )**

strongforth.sf

Search the locals vocabulary for a local with the name *i*. If it exists, rename it to *j*. Create a new local with the name *i* in the locals vocabulary.

**created-definition ( caddress -> character unsigned created-definition -- 4 th )**

Initialize `created-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `created-definition` is deleted. Assign `created-definition` a name given by the character string `ccaddress -> character unsigned` and return it as `4 th`.

`created-definition` is a constructor of the `created-definition` class.

**created-definition ( created-definition -- 1st )**

Initialize `created-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `created-definition` is deleted. Return `created-definition` as `1st`.

`created-definition` is a constructor of the `created-definition` class.

**created-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `created-definition`.

**created-definition? ( address -- created-definition flag )**

strongforth.sf

Search for a definition that was created with `create` and whose data field is equal to *address*. If such a definition exist, `created-definition` is the definition and *flag* is `true`. Otherwise, `created-definition` is `null` and *flag* is `false`.

**created-definition? ( definition -- created-definition flag )**

`created-definition` is `definition`. `flag` is true if and only if `definition` was defined by `create`.

**ctrl ( "<spaces>name" -- character )**

ascii.sf

Skip leading space delimiters. Parse *name* delimited by a space. `character` is the ASCII control character the keyboard generates when typing *name*'s first character while holding the CTRL key. An exception is thrown if *name*'s first character is not a lowercase or uppercase letter. If the length of the parsed area is zero, `character` is the null character.

**current ( -- address -> vocabulary )**

`address -> vocabulary` is the address of an object indicating the vocabulary to which new definitions are added.

**current-exception-frame ( -- address -> exception-frame )**

`address -> exception-frame` is the address of an object indicating the current lowest-level exception frame.

**cvariable ( single "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve one character of the data-space memory space and store `single` at the address.

*name* is referred to as a variable.

Execution: ( -- caddress -> x )

`caddress -> x` is the address of the reserved character. *x* has the same data type as was supplied to `cvariable` as `single`.

**cvariables ( single unsigned "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve unsigned characters of the data-space memory space and store `single` in each of them.

*name* is referred to as a variable.

Execution: ( -- caddress -> x )

`caddress -> x` is the address of the first reserved character. *x* has the same data type as was supplied to `cvariable` as `single`.

**d>f ( double -- float )**

`float` is the floating-point equivalent of the unsigned double number `double`. An ambiguous condition exists if `double` cannot be precisely represented as a floating-point value.

**d>f ( signed-double -- float )**

float is the floating-point equivalent of the signed double number signed-double. An ambiguous condition exists if signed-double cannot be precisely represented as a floating-point value.

**d>s ( double -- single )**

single is the numeric equivalent of double. An ambiguous condition exists if double lies outside the range of a single number.

**d>s ( integer-double -- integer )**

integer is the numeric equivalent of integer-double. An ambiguous condition exists if integer lies outside the range of a single number.

**d>s ( signed-double -- signed )**

signed is the numeric equivalent of signed-double. An ambiguous condition exists if signed lies outside the range of a single signed number.

**d>s ( unsigned-double -- unsigned )**

unsigned is the numeric equivalent of unsigned-double. An ambiguous condition exists if unsigned lies outside the range of a single unsigned number.

**data-space ( -- memory-space )**

memory-space is the system's data space. Variables, values, floating-point literals, character strings and arrays are stored in the data space.

**data-type ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type data-type.

**data-type-attributes ( data-type unsigned data-type-attributes -- 3rd )**

Initialize data-type-attributes by erasing all members. Store the attributes of data-type as the parent of the data type associated with data-type-attributes. Store unsigned as the size of the data type associated with data-type-attributes.

data-space-attributes is the constructor of the data-space-attributes class.

**data-type-attributes ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type data-type-attributes.

**data-type? ( definition -- data-type flag )** strongforth.sf

If *definition* was created by *procreates*, *data-type* is the data type it is associated with, and *flag* is true. Otherwise, *data-type* is null and *flag* is false.

**decimal ( -- )** strongforth.sf

Set the number-conversion radix to 10 (decimal).

**default ( input-stream -- )** strongforth.sf

Make *input-stream* the present default input stream.

**default ( memory-space -- )** strongforth.sf

Make *memory-space* the present default memory space.

**default ( output-stream -- )** strongforth.sf

Make *output-stream* the present default output stream.

**default-input-stream ( -- address -> input-stream )**

*address -> input-stream* is the address of an object indicating the present default input stream.

**default-memory-space ( -- address -> memory-space )**

*address -> memory-space* is the address of an object indicating the present default memory space.

**default-output-stream ( -- address -> output-stream )**

*address -> output-stream* is the address of an object indicating the present default output stream.

**defer ( "<spaces>name" -- )** strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition.

*name* Execution: Execute the definition that has been assigned to *name* by a succeeding execution of *defer!* or *is*. *name* is called a deferred definition. An exception is thrown if *name* is executed before it has been assigned an execution semantics by *defer!* or *is*.

Note that the new definition does have no stack effects by default. Stack effects have to be specified separately if they are intended. By using a stack diagram phrase ( ... -- ... ) immediately following *defer* and the definition name, the new definition is modified to incorporate stack effects.

**defer! ( token deferred-definition -- )**



Set deferred-definition to execute token. An ambiguous condition exists if token is not for a definition defined by defer.

**defer@ ( deferred-definition -- token )**

token is the execution token deferred-definition is set to execute.

**deferred-definition ( address -> token address -> character  
unsigned deferred-definition -- 6 th )**

Initialize deferred-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when deferred-definition is deleted. address -> token is the address of the token that is to be executed or compiled by the deferred definition. Assign deferred-definition a name given by the character string address -> character unsigned and return it as 6 th.

Execution: Execute the execution token that deferred-definition is set to execute. An exception is thrown if deferred-definition has not been set to execute an execution token.

deferred-definition is the constructor of the deferred-definition class.

**deferred-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type deferred-definition.

**definition ( caddress -> character unsigned definition -- 4 th )**

Initialize definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when definition is deleted. Assign definition a name given by the character string caddress -> character unsigned and return it as 4 th.

definition is a constructor of the definition class.

**definition ( caddress -> character unsigned definition definition  
-- 5 th )**

Initialize the second definition by copying all members of the first definition. For the second definition, establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when definition is deleted. Assign the second definition a name given by the character string caddress -> character unsigned. Clear the stack diagram of the second definition and return it as 5 th.

definition is a constructor of the definition class.

**definition ( definition -- 1st )**

Initialize definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when definition is deleted. Return definition as 1st.

definition is a constructor of the definition class.

**definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `definition`.

**definitions ( -- )**

strongforth.sf

Make the current compilation vocabulary the same as the head of the context vocabulary list.

`definitions` specifies that the names of subsequent definitions will be placed in the current compilation vocabulary. Subsequent changes in the current compilation vocabulary will not affect the vocabularies of already compiled definitions.

**delete ( caddress -> character unsigned -- )**

strongforth.sf

Delete the file with the path given by the string `caddress -> character unsigned`. An exception is thrown if the operation fails.

**delete ( object -- )**

Return all dynamic memory occupied by `object` to the system. An ambiguous condition exists if `object` was not allocated from dynamic memory.

`delete` is a virtual method of the `object` class.

**delimiter ( -- character )**

strex.sf

`character` is the delimiter used by `unescape` and `substitute`. It is initialized with `%` (percent character). Since `delimiter` is a value, it can be reassigned by `to`.

**destination ( destination -- 1st )**

Initialize `destination` by erasing all members. If the compiler data type heap is not locked, save a copy of the present compiler data type heap. Save the value of the code-space memory space pointer as the code destination.

`destination` is the constructor of the `destination` class.

**destination ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `destination`.

**df, ( complex -- )**

Reserve space for two double-precision floating-point numbers in the default memory space and store `complex` as a complex double-precision floating-point number in it. If the first unused address of the default memory space is cell aligned prior to execution of `df,`, it will remain cell aligned when `df,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not cell aligned prior to execution of `df,`. An exception is thrown if the default memory space overflows.

**df, ( complex memory-space -- )**

Reserve space for two double-precision floating-point numbers in `memory-space` and store `complex` as a complex double-precision floating-point number in it. If the first unused address of `memory-space` is cell aligned prior to execution of `df,`, it will remain cell aligned when `df,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not cell aligned prior to execution of `df,`. An exception is thrown if `memory-space` overflows.

**df, ( float -- )**

Reserve space for a double-precision floating-point number in the default memory space and store `float` as a double-precision floating-point number in it. If the first unused address of the default memory space is cell aligned prior to execution of `df,`, it will remain cell aligned when `df,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not cell aligned prior to execution of `df,`. An exception is thrown if the default memory space overflows.

**df, ( float memory-space -- )**

Reserve space for a double-precision floating-point number in `memory-space` and store `float` as a double-precision floating-point number in it. If the first unused address of `memory-space` is cell aligned prior to execution of `df,`, it will remain cell aligned when `df,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not cell aligned prior to execution of `df,`. An exception is thrown if `memory-space` overflows.

**dfaddress ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `dfaddress`.

**dfalign ( -- )**

`float.sf`

If the first unused address of the default memory space is not double-precision float aligned, reserve the required number of address units to make it double-precision float aligned.

**dfalign ( memory-space -- )**

`float.sf`

If the first unused address of `memory-space` is not double-precision float aligned, reserve the required number of address units to make it double-precision float aligned.

**dfaligned ( address -- 1st )**

`float.sf`

`1st` is the lowest double-precision float aligned address greater than or equal to `address`.

**dfallocate ( unsigned -- dfaddress )**

Allocate `unsigned` address units of contiguous dynamic memory space. The initial content of the allocated memory space is undefined. If the allocation succeeds, `dfaddress` is the aligned starting address of the allocated memory space. An exception is thrown if the operation fails.

**dfhere ( -- dfaddress )**

*dfaddress* is the first unused address of the default memory space.

**dfhere ( memory-space -- dfaddress )**

*dfaddress* is the first unused address of *memory-space*.

**dfloats ( integer -- 1st )**

float.sf

*1st* is the size in address units of *integer* double-precision floating-point numbers.

**dfmember ( object-size complex "<spaces>name" -- 1st )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to *object-size* aligned to double-precision floating-point numbers, plus the size in bits of two double-precision floating-point numbers.

*name* is referred to as a class member. *dfmember* reserves space for two double-precision floating-point numbers for a class member of the same data type as *complex* in the class that is currently being defined.

Execution: ( *x* -- *dfaddress* -> *y* )

*dfaddress* -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to *dfmember* as *complex*.

**dfmember ( object-size float "<spaces>name" -- 1st )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to *object-size* aligned to double-precision floating-point numbers, plus the size in bits of a double-precision floating-point number.

*name* is referred to as a class member. *dfmember* reserves space for one double-precision floating-point number for a class member of the same data type as *float* in the class that is currently being defined.

Execution: ( *x* -- *dfaddress* -> *y* )

*dfaddress* -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to *dfmember* as *float*.

**dfmembers ( object-size complex unsigned "<spaces>name" -- 1st )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to *object-size* aligned to double-precision floating-point numbers, plus unsigned times the size in bits of a complex double-precision floating-point number.

*name* is referred to as a class member. *dfmembers* reserves unsigned complex double-precision floating-point numbers for an array of unsigned class members of the same data type as *complex* in the class that is currently being defined.

Execution: ( *x* -- dfaddress -> *y* )

dfaddress -> *y* is the address of an array of unsigned class members of the object *x*, that were reserved at the time *name* was created. *y* is the actual data type that was provided to dfmember as complex.

**dfmembers ( object-size float unsigned "<spaces>name" -- 1st )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. 1st is equal to object-size aligned to double-precision floating-point numbers, plus unsigned times the size in bits of a double-precision floating-point number.

*name* is referred to as a class member. dfmembers reserves unsigned double-precision floating-point numbers for an array of unsigned class members of the same data type as float in the class that is currently being defined.

Execution: ( *x* -- dfaddress -> *y* )

dfaddress -> *y* is the address of an array of unsigned class members of the object *x*, that were reserved at the time *name* was created. *y* is the actual data type that was provided to dfmember as float.

**dfvariable ( complex "<spaces>name" -- )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for two double-precision floating-point numbers at a double-precision floating-point aligned address in the data-space memory space and store complex at the address.

*name* is referred to as a variable.

Execution: ( -- dfaddress -> *x* )

dfaddress -> *x* is the address of the complex double-precision floating-point number. *x* has the same data type as was supplied to variable.

**dfvariable ( float "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for a double-precision floating-point number at a double-precision floating-point aligned address in the data-space memory space and store float at the address.

*name* is referred to as a variable.

Execution: ( -- dfaddress -> *x* )

dfaddress -> *x* is the address of the double-precision floating-point number. *x* has the same data type as was supplied to variable.

**dfvariables ( complex unsigned "<spaces>name" -- )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for unsigned complex double-precision

floating-point numbers at a double-precision floating-point aligned address in the data-space memory space and store `complex` in each of them.

*name* is referred to as a variable.

Execution: ( -- dfaddress -> x )

dfaddress -> x is the address of the first complex double-precision floating-point number. x has the same data type as was supplied to variable.

**dfvariables ( float unsigned "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for unsigned double-precision floating-point numbers at a double-precision floating-point aligned address in the data-space memory space and store `float` in each of them.

*name* is referred to as a variable.

Execution: ( -- dfaddress -> x )

dfaddress -> x is the address of the first double-precision floating-point number. x has the same data type as was supplied to variable.

**digit? ( character -- unsigned flag )**

Converts *character* into a digit unsigned. Characters 0 to 9 are converted into digits 0 to 9, and characters a to z or A to Z are converted into digits 10 to 35, respectively. *flag* is true if *character* is alphanumeric and the conversion result is less than the number-conversion radix base. Otherwise *flag* is false and the value of unsigned is undefined.

**do ( -- do-destination ) compile-only**

strongforth.sf

Compilation: Create and initialize *do-destination* and save a copy of the compiler data type heap. Rename the loop index *i* into *j*, if it already exists, and define a new local *i* as loop index. Append the runtime semantics given below to the current definition. The semantics are incomplete until resolved by a consumer of *do-destination* such as `loop`, `+loop` and `-loop`.

Runtime: ( integer 1st -- ) or ( address 1st -- )

Initialize the loop control parameters with limit *integer* or *address* and index *1st*, and continue execution.

**do-destination ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *do-destination*.

**does-data-type ( -- )**

strongforth.sf

Finish the latest definition by specifying the execution semantics given below. An exception is thrown if the latest definition was not created by `create`. An ambiguous condition exists if the data field of the latest definition does not contain a `data-type-attributes` object.

Execution: ( stack-diagram -- 1st )

When used in a stack diagram, specifies an input or output parameter with the data type associated with the `data-type-attributes` object in the data field.

### **does-vocabulary ( -- )**

strongforth.sf

Finish the latest definition by specifying the execution semantics given below. An exception is thrown if the latest definition was not created by `create`. An ambiguous condition exists if the data field of the latest definition does not contain a `vocabulary` object.

Execution: Remove the `vocabulary` object in the data field from both the context vocabulary list and the hidden vocabulary list. Make the `vocabulary` object the head of the context vocabulary list. An ambiguous condition exists if the `vocabulary` object is not included in one of the two vocabulary lists.

### **does> ( colon-definition -- 1st ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. An exception is thrown if the contents of the compiler data type heap do not exactly match the output parameters of the current definition. Append the initiation semantics given below to the current definition.

Runtime: Specify the execution semantics of the most recent definition, referred to as *name*, as given below. Return control to the calling definition.

Initiation: Place the address of *name*'s data field on the stack.

*name* Execution: Execute the portion of the definition that begins with the initiation semantics appended by the `does>` which modified *name*.

Note that *name* does have no stack effects by default. Stack effects have to be specified explicitly. By using a stack diagram phrase immediately following `does>`, *name* is modified to incorporate stack effects. Specifying a stack diagram is mandatory, because at least the data type of *name*'s data field address has to be present. The data field address is always the last input parameter. The stack effect of *name* is defined by the stack diagram following `does>`, omitting the data field address.

### **double ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `double`.

### **double-definition ( caddress -> character unsigned double-definition -- 4 th )**

Initialize `double-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `double-definition` is deleted. Assign `double-definition` a name given by the character string `caddress -> character unsigned` and return it as 4 th.

`double-definition` is a constructor of the `double-definition` class.

### **double-definition ( double-definition -- 1st )**

Initialize `double-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `double-definition` is deleted.

`double-definition` is a constructor of the `double-definition` class.

**`double-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `double-definition`.

**`drop ( complex -- )`**

Remove `complex` from the stack.

**`drop ( double -- )`**

Remove `double` from the stack.

**`drop ( float -- )`**

Remove `float` from the stack.

**`drop ( single -- )`**

Remove `single` from the stack.

**`dt ( "<spaces>name" -- data-type )`**

Skip leading space delimiters. Parse *name* delimited by a space. Search all vocabularies for a definition with this name that was created by `procreates`. If such a definition is found, return the data type the definition is associated with as `data-type`. An exception is thrown if *name* is not the name of a data type, and `data-type` is null.

**`dt-allot ( -- )`**

In interpretation state (`state` is `false`), reserve space for one basic data type on the interpreter data type heap. In compilation state (`state` is `true`), reserve space for one basic data type on the compiler data type heap. An exception is thrown if the respective data type heap overflows.

**`dt-allot ( integer -- )`**

In interpretation state (`state` is `false`), reserve space for `integer` basic data types on the interpreter data type heap. In compilation state (`state` is `true`), reserve space for `integer` basic data types on the compiler data type heap. An exception is thrown if the respective data type heap overflows.

**`dt-bottom ( -- address -> data-type )`**

In interpretation state (`state` is `false`), `address -> data-type` is the bottom of the interpreter data type heap. In compilation state (`state` is `true`), `address -> data-type` is the bottom of the compiler data type heap.



**dt-compare ( address -> data-type unsigned 1st 3rd -- flag )**

strongforth.sf

Compare the unsigned basic data types stored at address -> data-type with the 3rd basic data types stored at address 1st. flag is true if and only if 3rd is equal to unsigned and all basic data types are one by one identical. Data type attributes other than the prefix and the reference attributes are not considered.

**dt-depth ( -- unsigned )**

In interpretation state (state is false), unsigned is the number of basic data types on the interpreter data type heap. In compilation state (state is true), unsigned is the number of basic data types on the compiler data type heap. An ambiguous condition exists if the compiler data type heap is locked and state is true.

**dt-drop ( -- )**

In interpretation state (state is false), remove the topmost compound data type from the interpreter data type heap. In compilation state (state is true), remove the topmost compound data type from compiler data type heap. An exception is thrown if the respective data type heap is empty prior to executing dt-drop.

**dt-here ( -- address -> data-type )**

In interpretation state (state is false), address -> data-type is the first unused address of the interpreter data type heap. In compilation state (state is true), address -> data-type is the first unused address of the compiler data type heap.

**dt-init ( -- )**

In interpretation state (state is false), empty the interpreter data type heap. In compilation state (state is true), unlock and empty the compiler data type heap.

**dt-length ( address -> data-type -- unsigned )**

unsigned is the number of basic data types the compound data type stored at address -> data-type consist of.

**dt-lock ( -- )**

In compilation state (state is true), lock the compiler data type heap to prevent further usage until its previous state is restored. While being locked, dt-here returns a null pointer in compilation state.

**dt-next ( address -> data-type -- 1st )**

1st is address -> data-type plus the size in address units of the compound data type stored at address -> data-type.

**dt-prefix ( -- data-type )**

`data-type` is a null data type with the prefix attribute. The prefix attribute is set in all but the last basic data types of a compound data type.

**`dt-reference ( -- data-type )`**

`data-type` is a null data type with the reference attribute. The reference attribute is set in data types diagram that reference previous data types within the same stack diagram.

**`dt-restore ( -- )`**

strongforth.sf

In interpretation state, restore a compound data type that has been dropped from the interpreter data type heap. In compilation state, restore a compound data type that has been dropped from the compiler data type heap.

**`dt-stripped ( address -> data-type unsigned -- 1st 3rd )`**

`address -> data-type` is the address of a list of unsigned basic data types which may contain compound data types linked by prefix attributes. `1st` is equal to `address`. `3rd` is `unsigned` minus the number of basic data types in the last compound data type in the list. If `unsigned` is zero, `3rd` is zero as well.

**`dump ( address -> double unsigned -- )`**

strongforth.sf

Send two times `unsigned` consecutive cells starting at `address -> double` to the default output stream, formatted as a sequence of eight-digit hexadecimal numbers. If necessary, multiple lines are sent. Each line contains the hexadecimal starting address and the contents of up to 4 memory cells.

**`dump ( address unsigned -- )`**

strongforth.sf

Send `unsigned` consecutive cells starting at `address` to the default output stream, formatted as a sequence of eight-digit hexadecimal numbers. If necessary, multiple lines are sent. Each line contains the hexadecimal starting address and the contents of up to 4 memory cells.

**`dump ( caddress unsigned -- )`**

strongforth.sf

Send `unsigned` consecutive characters starting at `caddress` to the default output stream, formatted as a sequence of two-digit hexadecimal numbers. If necessary, multiple lines are sent. Each line contains the hexadecimal starting address and the contents of up to 16 character-size memory cells.

**`dup ( complex -- 1st 1st )`**

Duplicate `complex`.

**`dup ( double -- 1st 1st )`**

Duplicate `double`.

**`dup ( float -- 1st 1st )`**

Duplicate float.

**dup ( single -- 1st 1st )**

Duplicate single.

**e. ( complex -- )**

complex.sf

Send the real part and the imaginary part of `complex` with a trailing space using engineering notation to the default output stream. The significands are greater than or equal to 1.0 and less than 1000.0, and the decimal exponent is a multiple of three:

```
exponential notation := <re> + <im> i
<re>                  := <significand><exponent>
<im>                  := <significand><exponent>
<significand>         := [-]<digits>.<digits0>
<exponent>            := e[-]<digit><digit><digit>
<digits>              := <digit><digits0>
<digits0>             := <digit>*
<digit>               := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**e. ( float -- )**

float.sf

Send `float` with a trailing space using engineering notation to the default output stream. The significand is greater than or equal to 1.0 and less than 1000.0, and the decimal exponent is a multiple of three:

```
exponential notation := <significand><exponent>
<significand>        := [-]<digits>.<digits0>
<exponent>           := e[-]<digit><digit><digit>
<digits>             := <digit><digits0>
<digits0>            := <digit>*
<digit>              := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**eax/edx: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to double register `eax/edx` (if it occupies a double cell) or to either register `eax` or register `edx` (if it occupies a single cell). An exception is thrown if the input or output parameter does not fit into a single register or a register pair.

**eax: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `eax`. An exception is thrown if the input or output parameter does not fit into a single register.

**ebx/ecx: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to double register `ebx/ecx` (if it occupies a double cell) or to either register `ebx` or register `ecx` (if it occupies a single cell). An exception is thrown if the input or output parameter does not fit into a single register or a register pair.

**`ebx: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `ebx`. An exception is thrown if the input or output parameter does not fit into a single register.

**`ecx: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `ecx`. An exception is thrown if the input or output parameter does not fit into a single register.

**`edi: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `edi`. An exception is thrown if the input or output parameter does not fit into a single register.

**`edx: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `edx`. An exception is thrown if the input or output parameter does not fit into a single register.

**`eflags: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to a flag in the processor status register. An exception is thrown if the input or output parameter cannot be represented by a flag.

**`ekey ( -- keyboard-event )`**

strongforth.sf

Receive one keyboard event `keyboard-event` from the user input device.

**`ekey? ( -- flag )`**

strongforth.sf

`flag` is `true` if and only if a keyboard event is available at the user input device. The event will be returned by the next execution of `ekey`.

After `ekey?` returns with a value of `true`, subsequent executions of `ekey?` prior to the execution of `key`, `key?` or `ekey` also return `true`, referring to the same event.

**`ekey>char ( keyboard-event -- character flag )`**

strongforth.sf

If the keyboard event `keyboard-event` corresponds to a character, return its ASCII value as `character` and `true` as `flag`. Otherwise, the value of `character` is equal to `keyboard-event` and `flag` is `false`.

**ekey>fkey ( keyboard-event -- unsigned flag )**

strongforth.sf

If the keyboard event `keyboard-event` corresponds to a special key, return its identification code as `unsigned` and `true` as `flag`. Otherwise, the value of `unsigned` is equal to `keyboard-event` and `flag` is `false`.

**else ( origin -- 1st ) compile-only**

Compilation: Put a new unresolved forward reference `1st` onto the stack and save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition. The semantics are incomplete until `1st` is resolved, e. g., by `then`. Resolve the forward reference `origin` using the location following the appended runtime semantics. Restore the compiler data type heap to the state that was saved when `origin` was created.

Runtime: Continue execution at the location given by the resolution of `1st`.

**emit ( integer -- )**

Send the ASCII code of `integer` to the default output stream.

**emit ( integer output-stream -- )**

Send the ASCII code of `integer` to `output-stream`.

`emit` is a virtual method of the `output-stream` class.

**emit? ( -- flag )**

strongforth.sf

`flag` is `true` if the user output device is ready to accept data and the execution of `emit` in place of `emit?` would not have suffered an indefinite delay.

**empty-buffers ( -- )**

block.sf

Unassign the block buffer. Do not transfer the contents of the block buffer to the block file.

**enclosing ( local-definition -- do-destination )**

If `local-definition` is a loop index, `do-destination` identifies the `do` loop. Otherwise, `do-destination` is `null`.

**enclosing! ( do-destination local-definition -- )**

Make `local-definition` a loop index by assigning `do-destination` as the destination of the associated `do` loop.

**end-compilation ( -- )**

Enter interpretation state. Delete all definitions in the locals vocabulary.

**end-loop ( do-destination local-definition -- )**

strongforth.sf

An ambiguous condition exists if `end-loop` is executed in interpretation state.

Delete the loop index `local-definition`. Search the locals vocabulary for a local index with the name `j`. If it exists, rename it to `i`. Append the runtime semantics given below to the current definition, resolving the backward reference `do-destination`. An exception is thrown if the contents of the compiler data type heap, after consuming `single`, do not exactly match the copy that was saved when `do-destination` was created.

Runtime: ( `single` -- )

If `single` is zero, continue execution at the location specified by `do-destination`. Otherwise, continue execution.

**end-structure ( structure-attributes object-size -- )**

struct.sf

Terminate the definition of a structure started by `begin-structure`.

**endcase ( endof-origin of-origin -- ) compile-only**

strongforth.sf

Compilation: Mark the end of a `case ... of ... endof ... endcase` structure. Delete `of-origin`. Use `endof-origin` to resolve the entire structure. Compare the current contents of the compiler data type heap with the one that was saved by the first `endof`. An exception is thrown if a difference is detected. Append the runtime semantics given below to the current definition.

Runtime: Continue execution.

Note that `endcase` does not discard the case selector.

**endclass ( vocabulary object-size -- )**

strongforth.sf

Store `object-size` in the virtual method table of the class that is currently being defined. If the virtual method table does not yet exist, create it in the `data-space` memory space. Save the protected vocabulary. If the class has friends, create a new vocabulary for the definitions of the concatenated private and protected vocabularies, that may be accessed by friends. Restore `vocabulary` as the current compilation vocabulary. Remove the private and protected vocabularies as well as the vocabularies of all friend classes from the context vocabulary list.

`endclass` ends a class definition.

**enddef ( code-definition -- )**

Make `code-definition` the definition most recently added to the current vocabulary. This is an extended version of `enddef ( definition )` that takes care of register usage by `code-definition`.

**enddef ( definition -- )**

Makes `definition` the definition most recently added to the current vocabulary.

**enddef ( local-definition -- )**

Makes `local-definition` the definition most recently added to the locals vocabulary. This is an extended version of `enddef ( definition )` that takes care of register usage by `local-definition`.

**enddef ( value-definition -- )**

Makes `value-definition` the definition most recently added to the current vocabulary. This is an extended version of `enddef ( definition )` that takes care of register usage by `value-definition`.

**endof ( of-origin endof-origin -- 2nd 1st ) compile-only**

strongforth.sf

Compilation: Resolve the reference given by `of-origin`. Append the runtime semantics given below to the current definition. Lock the compiler data type heap. If this is the first occurrence of `endof` within a `case` structure, save the contents of the compiler data type heap. Otherwise, compare the contents of the compiler data type heap with the one that was saved by the first `endof`. An exception is thrown if a difference is detected.

Runtime: Continue execution at the location specified by the consumer of `2nd`.

**endof-origin ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `endof-origin`.

**endunion ( object-size object-size object-size -- 1st )**

strongforth.sf

Terminate a union of members within a class definition. `1st` is the maximum of the second and the third `object-size`.

**erase ( address -> complex unsigned -- )**

complex.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive complex floating-point numbers beginning at `address -> complex`.

**erase ( address -> double unsigned -- )**

strongforth.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive double cells of memory beginning at `address -> double`.

**erase ( address -> float unsigned -- )**

float.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive floating-point numbers beginning at `address -> float`.

**erase ( address -> single unsigned -- )**

strongforth.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive cells of memory beginning at `address -> single`.

**erase ( caddress -> single unsigned -- )**

strongforth.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive character-size memory locations beginning at `caddress -> single`.

**erase ( dfaddress -> complex unsigned -- )**

complex.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive complex double-precision floating-point numbers beginning at `dfaddress -> complex`.

**erase ( dfaddress -> float unsigned -- )**

float.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive double-precision floating-point numbers beginning at `dfaddress -> float`.

**erase ( object -- )**

Set all members of `object` to zero.

**erase ( sfaddress -> complex unsigned -- )**

complex.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive complex single-precision floating-point numbers beginning at `sfaddress -> complex`.

**erase ( sfaddress -> float unsigned -- )**

float.sf

If `unsigned` is not equal to zero, clear all bits in each of `unsigned` consecutive single-precision floating-point numbers beginning at `sfaddress -> float`.

**error ( signed -- )**

If `signed` is not equal to zero, send an error message depending on the value of `signed` to the default output stream. Then perform the function of `quit`. If `signed` is -1, the error message is empty. If `signed` is -2, the error message is obtained from `line`.

`error` is a deferred definition.

**escaped-char ( character - 1st )**

escape.sf

Convert `character`, the second character of a string escape sequence starting with a `\` (backslash), into the associated substitution `1st` according to the following translation rules. All other characters remain unchanged.

Escape sequence	Substitution
<code>\a</code>	<code>&lt;bel&gt;</code>
<code>\b</code>	<code>&lt;bs&gt;</code>
<code>\e</code>	<code>&lt;esc&gt;</code>
<code>\f</code>	<code>&lt;ff&gt;</code>
<code>\l</code>	<code>&lt;lf&gt;</code>
<code>\m</code>	<code>&lt;lf&gt;</code>
<code>\n</code>	<code>&lt;lf&gt;</code>



<code>\q</code>	" (quote)
<code>\r</code>	<cr>
<code>\t</code>	<ht>
<code>\v</code>	<vt>
<code>\xyy</code>	(see below)
<code>\z</code>	<nul>

`\m` and `\n` perform the following additional semantics: Add <cr> to the end of the string conversion area.

`\xyy` performs the following semantics: Parse two hexadecimal digits `yy` and return the resulting two-digit ASCII code. An exception is thrown if `\x` is not followed by two hexadecimal characters.

**esi/edi: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to double register `esi/edi` (if it occupies a double cell) or to either register `esi` or register `edi` (if it occupies a single cell). An exception is thrown if the input or output parameter does not fit into a single register or a register pair.

**esi: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to register `esi`. An exception is thrown if the input or output parameter does not fit into a single register.

**evaluate ( caddress -> character unsigned -- )**

Save the default input stream. Create a new string input stream, initialize it with `caddress -> character unsigned` and make it the new default input stream. Set `>in` to zero, and interpret. When the parse area is empty, delete the associated input source and restore the default input source to its saved value. Other stack effects are due to the words evaluated.

**even-parity? ( character -- flag )**

`flag` is true if and only if `character` has even parity.

**exception-frame ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `exception-frame`.

**exception-frame ( unsigned address address exception-frame -- 4 th )**

Initialize `exception-frame` by erasing all members. Establish a link to the current exception frame and make `exception-frame` the current exception frame. Save the default input stream and the input source specification. Save `unsigned` as the hardware floating-point stack pointer, the first `address` as the instruction pointer and the second `address` as the stack pointer. Return `exception-frame` as 4 `th`.

Once `exception-frame` is deleted, the previous exception frame is restored.

exception-frame is the constructor of the exception-frame class.

**execute ( (--) -- )**

strongforth.sf

Performs the semantics of the definition identified by the qualified execution token (--) . The definition has no input and output parameters.

**execute ( (--string) - caddress -> character unsigned )**

strongforth.sf

Performs the semantics of the definition identified by the qualified execution token (--string) . caddress -> character and unsigned are the output parameters of the definition.

**execute ( definition single search-criterion -- flag )**

Performs the semantics of the definition identified by the qualified execution token search-definition. definition, single and flag are the input and output parameters of the definition.

**execute ( unsigned (unsigned--) --)**

strongforth.sf

Performs the semantics of the definition identified by the qualified execution token (unsigned-) . unsigned is the input parameter of the definition.

**execute-criterion ( -- search-criterion )**

strongforth.sf

search-criterion is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

flag is true if and only if the virtual method table of definition is equal to the virtual method table of (execute) and the last input parameter of definition is equal to the qualified token with the data type attributes single.

Note: Provide search-criterion to search in order to find a definition created by )procreates that executes a qualified token of a given data type.

**execute-only ( -- )**

strongforth.sf

Make the latest definition an execute-only word. The interpreter finds this definition only if in interpretation state.

**exit ( -- ) compile-only**

Compilation: Append the runtime semantics given below to the current definition. An exception is thrown if the contents of the compiler data type heap do not exactly match the output parameters of the current definition. Lock the compiler data type heap.

Runtime: Remove the stack frame and return to the calling definition.

**exp ( complex -- 1st )**

complex.sf

Raise `e` to the power `complex`, giving `1st`. This operation is based on complex floating-point numbers.

**exp ( float -- 1st )**

Raise `e` to the power `float`, giving `1st`.

**expm1 ( float -- 1st )**

Raise `e` to the power `float` and subtract one, giving `1st`.

**f>d ( float -- signed-double )**

`signed-double` is the numerical equivalent of the integer portion of `float`. The fractional portion of `float` is discarded. An ambiguous condition exists if the integer portion of `float` cannot be represented as a double-cell signed integer.

Rounding the floating-point value prior to calling `f>s` is advised, because `f>s` always rounds towards zero.

**f>s ( float -- signed )**

`signed` is the numerical equivalent of the integer portion of `float`. The fractional portion of `float` is discarded. An ambiguous condition exists if the integer portion of `float` cannot be represented as a single-cell signed integer.

Rounding the floating-point value prior to calling `f>s` is advised, because `f>s` always rounds towards zero.

**falign ( memory-space -- )**

float.sf

If the first unused address of `memory-space` is not float aligned, reserve the required number of address units to make it float aligned.

Floating-point numbers are stored in a 10-byte format. Each address that is a multiple of 2 is assumed to be float aligned.

**falign ( -- )**

float.sf

If the first unused address of the default memory space is not float aligned, reserve the required number of address units to make it float aligned.

Floating-point numbers are stored in a 10-byte format. Each address that is a multiple of 2 is assumed to be float aligned.

**faligned ( address -- 1st )**

float.sf

`1st` is the lowest float aligned address greater than or equal to `address`.

Floating-point numbers are stored in a 10-byte format. Each address that is a multiple of 2 is assumed to be float aligned.

**false ( -- flag )**

flag is a false flag, a single-cell item with all bits set to 0.

**fam ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type fam.

**fdepth ( -- unsigned )**

float.sf

unsigned is the number of floating-point numbers on the hardware floating-point stack (0 to 8).

**file ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type file.

**file-input-stream ( file unsigned file-input-stream -- 3rd )**

Initialize file-input-stream by erasing all members. Make file the input file. Allocate unsigned characters from dynamic memory as input buffer. 3rd is file-input-stream.

file-input-stream is a constructor of the file-input-stream class.

**file-input-stream ( file-input-stream 1st - 1st )**

Copy all members of file-input-stream to 1st. 1st is 1st. 1st shares the same input buffer as file-input-stream. The input buffer will not be deallocated when 1st is deleted. An ambiguous condition exists if 1st is used after file-input-stream has been deleted.

file-input-stream is a constructor of the file-input-stream class.

**file-input-stream ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type file-input-stream.

**file-output-stream ( file file-output-stream -- 2nd )**

strongforth.sf

Initialize file-output-stream by erasing all members. Make file the output file. 2nd is file-output-stream.

file-output-stream is a constructor of the file-output-stream class.

**file-output-stream ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type file-output-stream.

**fill ( address -> complex unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive complex floating-point numbers, beginning at address -> complex.

**fill ( address -> double unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive double cells of memory, beginning at address -> double.

**fill ( address -> float unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive floating-point numbers, beginning at address -> float.

**fill ( address -> single unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive cells of memory, beginning at address -> single.

**fill ( caddress -> character unsigned 1st 3rd --)**

float.sf

Copy the character string 1st 3rd to the buffer caddress -> character unsigned. If 3rd is greater than unsigned, the character string is truncated to unsigned characters. If 3rd is less than unsigned, the buffer is padded with trailing spaces.

**fill ( caddress -> single unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive character-size memory locations, beginning at caddress -> single.

**fill ( dfaddress -> complex unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive complex double-precision floating-point numbers, beginning at dfaddress -> complex.

**fill ( dfaddress -> float unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive double-precision floating-point numbers, beginning at dfaddress -> float.

**fill ( sfaddress -> complex unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive complex single-precision floating-point numbers, beginning at sfaddress -> complex.

**fill ( sfaddress -> float unsigned 2nd -- )**

If unsigned is not equal to zero, store 2nd in each of unsigned consecutive single-precision floating-point numbers, beginning at sfaddress -> float.

**first ( vocabulary -- definition )**

strongforth.sf

definition is the first definition that has been added to vocabulary.

**flag ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type flag.

**float ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type float.

**float-definition ( caddress -> character unsigned float-definition -- 4 th )**

Initialize float-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when float-definition is deleted. Assign float-definition a name given by the character string caddress -> character unsigned and return it as 4 th.

float-definition is a constructor of the float-definition class.

**float-definition ( float-definition -- 1st )**

Initialize float-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when float-definition is deleted.

float-definition is a constructor of the float-definition class.

**float-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type float-definition.

**float-lit ( -- ) immediate**

float.sf

Remove the floating-point literal vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the floating-point literal vocabulary the head of the context vocabulary list. An ambiguous condition exists if the floating-point literal vocabulary was not included in one of the two vocabulary lists before float-lit is executed.

The search virtual method of the floating-point literal vocabulary recognizes and converts floating-point numbers in the following format, if the number-conversion radix base is (decimal) 10.

```
convertible string := <significand><exponent>
<significand>      := [<sign>]<digits>[.<digits0>]
<exponent>         := e[<sign>]<digits0>
<sign>              := { + | - }
<digits>            := <digit><digits0>
<digits0>           := <digit>*
<digit>             := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

**float-vocabulary ( float-vocabulary -- 1st )**

float.sf

Make float-vocabulary an empty vocabulary and add it as the first item in the hidden vocabulary list.

`float-vocabulary` is the constructor of the `float-vocabulary` class.

**`float-vocabulary ( stack-diagram -- 1st )`**

float.sf

When used in a stack diagram, specifies an input or output parameter with data type `float-vocabulary`.

**`float: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to the hardware floating-point stack. An exception is thrown if the input or output parameter is not a floating-point number.

**`floating-stack ( -- unsigned )`**

float.sf

`unsigned` is the size of the hardware floating-point stack.

**`floats ( integer -- 1st )`**

float.sf

`1st` is the size in address units of `integer` floating-point numbers.

**`floor ( float -- 1st )`**

Round `float` to an integral value using the round toward negative infinity rule, giving `1st`.

**`floored ( -- flag )`**

strongforth.sf

`flag` is true if and only if floored division is the default.

**`flush ( -- )`**

block.sf

If the block buffer is marked as modified, transfer its contents to the block file. Unassign the block buffer. An exception is thrown if the block buffer is assigned to an invalid block.

**`flush ( file -- )`**

strongforth.sf

Attempt to force any buffered information written to `file` to be written to mass storage, and the size information to be recorded in the storage directory if changed. If `file` was opened in write mode, the contents of the stream buffer are written to the file or device and the buffer is discarded. If `file` was opened in read mode, or if the stream has no buffer, `flush` has no effect, and any buffer is retained. An exception is thrown if the operation fails.

**`fm/mod ( signed-double signed -- 2nd signed )`**

Divide `signed-double` by `signed`, giving the floored quotient `signed` and the remainder `2nd`. An exception is thrown if `signed` is zero. An ambiguous condition exists if the quotient lies outside the range of a signed single-precision number.

**`forget ( "<spaces>name" -- )`**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Delete all definitions in the current compilation vocabulary starting at the last definition up to and including *name*.

**forget-locals ( -- )**

Delete all definitions in the locals vocabulary.

**forth ( -- ) immediate**

Remove the *forth* vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the *forth* vocabulary the head of the context vocabulary list. An ambiguous condition exists if the *forth* vocabulary was not included in one of the two vocabulary lists before *forth* is executed.

**forth-vocabulary ( -- vocabulary )**

order.sf

*vocabulary* is the forth vocabulary.

**fp! ( unsigned -- )**

Make *unsigned* the current index of the hardware floating-point stack.

**fp@ ( -- unsigned )**

*unsigned* is the index of the hardware floating-point stack.

**fpe ( -- signed )**

Checks the exception flags in the hardware floating-point status word. *signed* is an appropriate error code for *throw*:

Exception	Floating-point status word								signed
	IE	SF	C1	DE	ZE	OE	UE	PE	
Invalid operation	1	0	x	x	x	x	x	x	-55
Stack overflow	1	1	1	x	x	x	x	x	-44
Stack underflow	1	1	0	x	x	x	x	x	-45
Denormalized operand	0	x	x	1	x	x	x	x	-46
Division by zero	0	x	x	0	1	x	x	x	-42
Numeric overflow	0	x	x	0	0	1	x	x	-43
Numeric underflow	0	x	x	0	0	0	1	x	-54
Inexact result	0	x	x	0	0	0	0	1	-41
none	0	x	x	0	0	0	0	0	0

**free ( address -- )**

Return the contiguous memory space starting at *address* to the system for later allocation. An ambiguous condition exists if *address* does not indicate a memory space that was previously obtained by *allocate*, *callocate*, *dfallocate*, *sfallocate* or *resize*.



**friend-criterion ( -- search-criterion )**

strongforth.sf

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

`flag` is true if and only if `definition` is associated with the data type of a class whose protected vocabulary is equal to the value of `single`.

Note: Provide `search-criterion` to `search` in order to find the definition identifying the class with a given protected vocabulary.

**friend? ( vocabulary -- data-type flag )**

strongforth.sf

If `vocabulary` is a protected vocabulary, `flag` is true and `data-type` is the data type identifying the class it belongs to. Otherwise, `flag` is false and `data-type` is null.

**friends( ( object-size "<spaces>name<sub>1</sub><spaces>name<sub>2</sub> ...  
<spaces>name<sub>n</sub> )" -- 1st )**

strongforth.sf

Create a list of  $n$  friend classes of the class currently being defined by repeatedly skipping leading spaces, parsing `name`, and adding the class identified by `name` to the list of friends of the currently defined class. The list of friend classes is terminated by `)`.

`object-size` is a dummy parameter that ensures that `friends (` is always used within the body of a class definition. `1st` is `object-size`.

`friends (` may be used zero or one time within the body of a class definition. An exception is thrown if `friends (` is executed more than once within the body of a class definition, or if a name is parsed that does not identify a class.

**fwait ( -- )**

Handle pending floating-point exceptions.

**fxam ( -- logical )**

`logical` is the content of the floating-point processor's status word, after examining the number on top of the hardware floating-point stack. Bits 14, 10, 9 and 8 of `logical` are set depending on the number on top of the hardware floating-point stack:

Status	Bit 14	Bit 10	Bit 9	Bit 8
Unsupported	0	0	s	0
NaN (Not a Number)	0	0	s	1
Valid	0	1	s	0
Infinity	0	1	s	1
Zero	1	0	s	0
Free	1	0	x	1
Denormalized	1	1	s	0

`x` is undefined. `s` is 0 if the number on top of the hardware floating-point stack is positive, and 1 if it is negative.

**get-current ( -- vocabulary )**

order.sf

`vocabulary` is the current compilation vocabulary.

**get-order ( -- search-order )**

order.sf

Allocate memory and store the context vocabulary list and the hidden vocabulary list in it. `search-order` is an identifier that enables `set-order` to restore the context vocabulary list and the hidden vocabulary list.

**here ( -- address )**

`address` is the first unused address of the default memory space.

**here ( memory-space -- address )**

`address` is the first unused address of `memory-space`.

**hex ( -- )**

strongforth.sf

Set the number-conversion radix to 16 (hexadecimal).

**hidden ( -- address -> vocabulary )**

`address -> vocabulary` is the address of a vocabulary that is not searched by `search-context` and all words using it. This vocabulary is actually the head of a linked list of vocabularies not to be searched.

**high ( double -- single )**

`single` is the most significant cell of `double`.

**hold ( caddress -> character unsigned -- )**

strongforth.sf

Prepend unsigned characters starting at `caddress -> character` to the beginning of the pictured numeric output string. An ambiguous condition exists if `hold` executes outside of a `<# ... #>` delimited pictured numeric output conversion. An exception is thrown if the transient area used for storing the pictured numeric output overflows.

**hold ( character -- )**

strongforth.sf

Prepend `character` to the beginning of the pictured numeric output string. An ambiguous condition exists if `hold` executes outside of a `<# ... #>` delimited pictured numeric output conversion. An exception is thrown if the transient area used for storing the pictured numeric output overflows.

**hold> ( caddress -> character unsigned -- )**

strongforth.sf

Append unsigned characters starting at `caddress -> character` to the end of the string conversion area. An ambiguous condition exists if the string conversion area is used for other

purposes before string conversion is done. An exception is thrown if the string conversion area overflows.

**hold> ( character -- )**

strongforth.sf

Append `character` to the end of the string conversion area. An ambiguous condition exists if the string conversion area is used for other purposes before string conversion is done. An exception is thrown if the string conversion area overflows.

**home ( -- )**

strongforth.sf

Position the cursor of the console window at the upper left corner.

**i ( -- complex )**

complex.sf

`complex` is the complex floating-point literal with `0e0` as the real part and `1e0` as the imaginary part.

**i\* ( complex -- 1st )**

complex.sf

Multiply `complex` by the imaginary unit `i`, giving `1st`.

**i\*+ ( float float -- complex )**

Merges two floating-point numbers `float` into a complex floating-point number `complex`. The first one becomes the real part, the second one becomes the imaginary part.

`i*+` is the synonym to `merge`. Its name suggests that the floating-point number on top of the stack is being multiplied by the imaginary unit `i` and then added to the number next on the stack.

**identity-criterion ( -- search-criterion )**

strongforth.sf

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( `definition single -- flag` )

`flag` is true if and only if the input and output parameters of the stack diagram `single` are identical to the input and output parameters of `definition`.

Note: Provide `search-criterion` to search in order to find a definition with a given name and stack diagram.

**if ( -- origin ) compile-only**

Compilation: Create a new unresolved forward reference `origin` and save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition. The semantics are incomplete until `origin` is resolved, e. g., by `then` or `else`.

Runtime: ( `single --` )

If `single` is zero, continue execution at the location specified by the resolution of `origin`.

### **ignore ( -- ) immediate**

If the context vocabulary list is not empty, remove the vocabulary at the head of the context vocabulary list and prepend it to the hidden vocabulary list.

### **ignore-all ( -- )**

order.sf

Remove all vocabularies from the context vocabulary list and prepend them to the hidden vocabulary list.

### **ignore-friends ( -- )**

strongforth.sf

Remove all protected vocabularies from the context vocabulary list and prepend them to the hidden vocabulary list.

### **im ( complex -- float )**

float is the imaginary part of complex.

### **immediate ( -- )**

strongforth.sf

Make the latest definition an immediate word.

### **immediate? ( definition -- flag )**

strongforth.sf

flag is true if and only if definition is an immediate definition or a compile-only definition.

### **include ( "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Open the file with the name *name* in read-only mode. Position the file pointer to the start of the file. An exception is thrown if the file does not exist or the operation fails.

Save the default input stream. Create a new file input stream, initialize it with the opened file and make it the new default input stream. Other stack effects are due to the words included.

Repeat until end of file: Read a line from the file, fill the input buffer from the contents of that line, set >in to zero, and interpret. Text interpretation begins at the file position where the next file read would occur.

When the end of the file is reached, close the file, delete the associated input source and restore the default input source to its saved value. An exception is thrown if there is an I/O exception reading the file, or if an I/O exception occurs while closing the file. When an exception is thrown, the status (open or closed) of any files that were being interpreted is undefined. Create a definition *name* with the execution semantics defined below, and make it the latest definition.

Execution: Throw an exception.

### **include ( caddress -> character unsigned -- )**

strongforth.sf

Open the file with the name given by the character string *caddress -> character unsigned* in read-only mode. Position the file pointer to the start of the file. An exception is thrown if the file does not exist or the operation fails.

Save the default input stream. Create a new file input stream, initialize it with the opened file and make it the new default input stream. Other stack effects are due to the words included.

Repeat until end of file: Read a line from the file, fill the input buffer from the contents of that line, set `>in` to zero, and interpret. Text interpretation begins at the file position where the next file read would occur.

When the end of the file is reached, close the file, delete the associated input source and restore the default input source to its saved value. An exception is thrown if there is an I/O exception reading the file, or if an I/O exception occurs while closing the file. When an exception is thrown, the status (open or closed) of any files that were being interpreted is undefined. Create a definition with the name specified by the string `caddress -> character unsigned` and the execution semantics defined below, and make it the latest definition.

Execution: Throw an exception.

**`include ( file -- )`**

strongforth.sf

Save the default input stream. Create a new file input stream, initialize it with `file` and make it the new default input stream. Other stack effects are due to the words included.

Repeat until end of file: Read a line from the file, fill the input buffer from the contents of that line, set `>in` to zero, and interpret. Text interpretation begins at the file position where the next file read would occur.

When the end of the file is reached, close the file, delete the associated input source and restore the default input source to its saved value. An exception is thrown if `file` is invalid, if there is an I/O exception reading `file`, or if an I/O exception occurs while closing `file`. When an exception is thrown, the status (open or closed) of any files that were being interpreted is undefined.

**`index ( virtual-definition -- unsigned )`**

`unsigned` is the index of the virtual method `virtual-definition` within the virtual method table.

**`inline ( -- )`**

strongforth.sf

If the latest definition is a code definition, mark it as an inline code definition, otherwise throw an exception. An inline code definition will be compiled as a series of inline machine code instructions instead of a single `call`, instruction.

**`inline! ( code-definition -- )`**

Marks `code-definition` as an inline code definition. An inline code definition will be compiled as a series of inline machine code instructions instead of a single `call`, instruction.

**`inline? ( code-definition -- flag )`**

`flag` is true if and only if `code-definition` has been marked as an inline code definition.

**`input-param, ( data-type stack-diagram -- )`**

Append `data-type` as an additional input parameter to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**input-params ( definition -- address -> data-type unsigned )**

address -> data-type is the address of the first input parameter of the stack diagram of definition. unsigned is the number of basic data types in the input parameter list of the stack diagram of definition.

**input-params ( stack-diagram -- address -> data-type unsigned )**

address -> data-type is the address of the first input parameter of stack-diagram. unsigned is the number of basic data types in the input parameter list of stack-diagram.

**input-stream ( input-stream 1st - 1st )**

Copy all members of input-stream to 1st. 1st is 1st. 1st shares the same input buffer as input-stream. The input buffer will not be deallocated when 1st is deleted. An ambiguous condition exists if 1st is used after input-stream has been deleted.

input-stream is a constructor of the input-stream class.

**input-stream ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type input-stream.

**input-stream ( unsigned input-stream -- 2nd )**

Initialize input-stream by erasing all members. Allocate unsigned characters from dynamic memory as input buffer. 2nd is input-stream.

input-stream is a constructor of the input-stream class.

**integer ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type integer.

**integer-double ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type integer-double.

**integer-lit ( -- ) immediate**

Remove the integer literal vocabulary from both the context vocabulary list and the hidden vocabulary list. Prepend the integer literal vocabulary to the context vocabulary list. An ambiguous condition exists if the integer literal vocabulary was not included in one of the two vocabulary lists before integer-lit is executed.

The search virtual method of the integer literal vocabulary recognizes and converts integer numbers in the following format, if the digits are within the allowed range of the number-conversion radix base.

convertible string := [<prefix>][<sign>]<digits>[.] | '<char>'

```

<prefix>          := { # | $ | % }
<sign>            := { + | - }
<digits>          := <digit><digits0>
<digits0>         := <digit>*
<digit>           := { { 0-9 } | { A-Z } | { a-z } }

```

If a prefix is provided, the contents of `base` is temporarily changed to 10 (#), 16 (\$) or 2 (%). A digit has a value ranging from zero to one less than the contents of `base`. The digit with the value zero corresponds to the character 0. This representation of digits proceeds through the character set to the decimal value nine corresponding to the character 9. For digits beginning with the decimal value ten the graphic characters beginning with the characters A or a are used. This correspondence continues up to and including the digit with the decimal value thirty-five which is represented by the characters Z or z.

Convertible strings with no leading sign and no trailing period are converted into `unsigned` numbers. Convertible strings with a leading sign and no trailing period are converted into `signed` numbers. Convertible strings with no leading sign and a trailing period are converted into `unsigned-double` numbers. Convertible strings with a leading sign and a trailing period are converted into `signed-double` numbers.

If the convertible string consists of a graphic character `<char>` enclosed by single quotes, it is converted into a `character` with the ASCII value of the graphic character.

### **integer-vocabulary ( integer-vocabulary -- 1st )**

Make `integer-vocabulary` an empty vocabulary and prepend to the hidden vocabulary list.

`integer-vocabulary` is the constructor of the `integer-vocabulary` class.

### **integer-vocabulary ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `integer-vocabulary`.

### **interpret ( -- )**

Interpret the contents of the parse area.

Search and compile or execute each word in the parse area. The search process is done in the following order:

1. In compilation state, search in the locals vocabulary using `search-local`. If a local with the given name is found, compile this local.
2. Search in the context vocabularies using `search-context` with `match-criterion`. If a matching definition is found, compile or execute it depending on `state` and the attributes of the definition:

state	attributes	semantics
false		execute
false	immediate	execute
false	execute-only	execute
false	compile-only	none
true		compile
true	immediate	execute
true	execute-only	none

true	compile-only	execute
------	--------------	---------

3. If this is in the locals vocabulary, compile it and search again in the context vocabularies using `search-context` with `match-criterion`., then compile the matching definition.

An exception is thrown if one of the words in the parse area does not match any part of the search process.

**invert ( data-type -- 1st )**

`1st` is `data-type` with attributes that are the bit-by-bit logical inverse of the attributes of `data-type`.

**invert ( logical -- 1st )**

Invert all bits of `logical`, giving its logical inverse `1st`.

**is ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Find a deferred definition with the name *name*. An exception is thrown if no deferred definition with the name *name* exists. Append the run-time semantics given below to the current definition.

Run-time: ( definition -- )

Assign the execution semantics of *definition* to *name*. An exception is thrown if the stack diagram of *definition* does not match the stack diagram of *name* according to the rules of the StrongForth data type system.

Deferred definitions are words defined by `defer`.

**is ( definition "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find a deferred definition with the name *name*. An exception is thrown if no deferred definition with name *name* exists or if the stack diagram of *definition* does not match the stack diagram of *name* according to the rules of the StrongForth data type system. Assign the execution semantics of *definition* to *name*.

Deferred definitions are words defined by `defer`.

**is ( object-size definition "<spaces>name" -- 1st )**

strongforth.sf

If the class that is currently being defined does not yet have a virtual method table, create a virtual method table in the `data-space` memory space and initialize it with `object-size` and the tokens of the parent class plus unassigned tokens for newly added virtual methods. Otherwise, just update the existing virtual method table with `object-size`.

Skip leading space delimiters. Parse *name* delimited by a space. Find a virtual definition with the name *name*. An exception is thrown if no virtual definition with name *name* exists or if the stack diagram of *definition* does not match the stack diagram of *name* according to the rules of the StrongForth data type system. Assign the execution semantics of *definition* to *name*.

Virtual definitions are words defined by `virtual`.



<b>k-alt-mask ( -- logical )</b>	strongforth.sf
logical is a mask for the ALT key, that can be ored with the key value to produce a value that the sequence <code>ekey ekey&gt;fkey</code> may produce when the user presses the corresponding key combination.	
<b>k-ctrl-mask ( -- logical )</b>	strongforth.sf
logical is a mask for the CTRL key, that can be ored with the key value to produce a value that the sequence <code>ekey ekey&gt;fkey</code> may produce when the user presses the corresponding key combination.	
<b>k-delete ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “delete” key.	
<b>k-down ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “cursor down” key.	
<b>k-end ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “end” key.	
<b>k-F1 ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “F1” key.	
<b>k-F10 ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “F10” key.	
<b>k-F11 ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “F11” key.	
<b>k-F12 ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “F12” key.	
<b>k-F2 ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “F2” key.	

**k-F3 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F3” key.

**k-F4 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F4” key.

**k-F5 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F5” key.

**k-F6 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F6” key.

**k-F7 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F7” key.

**k-F8 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F8” key.

**k-F9 ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “F9” key.

**k-home ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “home” or “pos1” key.

**k-insert ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “insert” key.

**k-left ( -- unsigned )**

strongforth.sf

unsigned is the value that the sequence `ekey ekey>fkey` would produce when the user presses the “cursor left” key.

<b>k-next ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “PgDn” key.	
<b>k-prior ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “PgUp” key.	
<b>k-reverse-tab ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the shift key and the “tabulator” key.	
<b>k-right ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “cursor right” key.	
<b>k-shift-mask ( -- logical )</b>	strongforth.sf
logical is a mask for the SHIFT key, that can be <code>ored</code> with the key value to produce a value that the sequence <code>ekey ekey&gt;fkey</code> may produce when the user presses the corresponding key combination.	
<b>k-up ( -- unsigned )</b>	strongforth.sf
unsigned is the value that the sequence <code>ekey ekey&gt;fkey</code> would produce when the user presses the “cursor up” key.	
<b>key ( -- character )</b>	strongforth.sf
Receive <code>character</code> from the user input device. All standard characters can be received. Characters received by <code>key</code> are not echoed.	
<b>key? ( -- flag )</b>	strongforth.sf
<code>flag</code> is <code>true</code> if and only if a character is available at the user input device. If non-character keyboard events are available before the first valid character, they are discarded and are subsequently unavailable. The character will be returned by the next execution of <code>key</code> .	
After <code>key?</code> returns with a value of <code>true</code> , subsequent executions of <code>key?</code> prior to the execution of <code>key</code> or <code>ekey</code> also return <code>true</code> , without discarding keyboard events.	
<b>keyboard-event ( stack-diagram -- 1st )</b>	strongforth.sf
When used in a stack diagram, specifies an input or output parameter with data type <code>keyboard-event</code> .	

**label ( "<spaces>name" -- )**

asm.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. *name* is referred to as a label.

Execution: ( -- address )

address is the value of the code-space memory space pointer at the time *name* is being defined.

**last ( vocabulary -- definition )**

definition is the definition most recently added to vocabulary.

**last! ( definition vocabulary -- )**

Makes definition the definition most recently added to vocabulary.

**latest ( -- definition )**

definition is the latest compiled definition. Since latest is a value, it can be reassigned with to.

**leave ( -- ) compile-only**

strongforth.sf

Compilation: An exception is thrown if no loop control parameters are available or if the contents of the compiler data type heap do not exactly match the copy that was saved when the current loop control parameters were created. Lock the compiler data type heap. Append the runtime semantics given below to the current definition.

Runtime: Discard the current loop control parameters. An ambiguous condition exists if they are unavailable. Continue execution immediately following the innermost syntactically enclosing do loop.

**line ( -- caddress -> character )**

caddress -> character is the address of a transient area used to hold data for intermediate character string processing. The transient area is /hold characters long.

Note: This transient area is used by the system for storing error messages, for escape string processing and for pictured numeric output.

**link ( definition definition -- )**

Establish a link from the second definition to the first definition without updating latest.

**link-criterion ( -- search-criterion )**

search-criterion is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

`flag` is true if and only if `definition` is the successor of the `definition single` within its vocabulary, i. e., `definition` is linked to `single`.

Note: Provide `search-criterion` to search in order to find the successor of a given definition.

**`list ( unsigned -- )`**

block.sf

Store `unsigned` in `scr`. Send `block unsigned` as 16 lines of text to the default output stream.

**`lit: ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to a literal value. An exception is thrown if the input or output parameter does not fit into a single-cell or double-cell literal.

**`literal ( complex -- ) compile-only`**

complex.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: Place `complex` on the stack. `complex` has the same data type as was supplied at compilation time.

**`literal ( double -- ) compile-only`**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: Place `double` on the stack. `double` has the same data type as was supplied at compilation time.

**`literal ( float -- ) compile-only`**

float.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: Place `float` on the stack. `float` has the same data type as was supplied at compilation time.

**`literal ( single -- ) compile-only`**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: Place `single` on the stack. `single` has the same data type as was supplied at compilation time.

**`literal, ( complex address -> data-type -- )`**

Append the runtime semantics given below to the current definition.

Runtime: ( -- `x` )

Place complex floating-point literal `x` on the stack. `x` has the compound data type stored at `address -> data-type` and the value given by `complex`.

**`literal, ( double address -> data-type -- )`**

Append the runtime semantics given below to the current definition.

Runtime: ( -- *x* )

Place double-cell literal *x* on the stack. *x* has the compound data type stored at address -> data-type and the value given by double.

**literal, ( float address -> data-type -- )**

Append the runtime semantics given below to the current definition.

Runtime: ( -- *x* )

Place floating-point literal *x* on the stack. *x* has the compound data type stored at address -> data-type and the value given by float.

**literal, ( single address -> data-type -- )**

Append the runtime semantics given below to the current definition.

Runtime: ( -- *x* )

Place single-cell literal *x* on the stack. *x* has the compound data type stored at address -> data-type and the value given by single.

**ln ( complex -- 1st )**

complex.sf

1st is the complex natural logarithm of complex. An ambiguous condition exists if complex is equal to zero.

**ln ( float -- 1st )**

1st is the natural logarithm of float. An ambiguous condition exists if float is less than or equal to zero.

**lnp1 ( float -- 1st )**

1st is the natural logarithm of the quantity float plus one. An ambiguous condition exists if float is less than or equal to -1.

**load ( unsigned -- )**

block.sf

Save the default input stream. Create a new block input stream, and make it the new default input stream. Store unsigned in blk. Set >in to zero, and interpret. Once the parse area cannot be refilled, delete the associated input source and restore the default input source to its saved value. Other stack effects are due to the words loaded. An exception is thrown if unsigned is not a valid block number.

**local ( colon-definition "<spaces>name" -- 1st ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Create a new local whose definition name is given by *name*. Append the runtime semantics given below to the current definition.

`local` may be used multiple times within a definition in order to define more than one local. `colon-definition` is a dummy parameter that prevents `local` to be used within loops, in conditional clauses, or between `>r` and `r>`. `1st` is `colon-definition`.

Runtime: ( `x` -- )

Initialize the local with the value of `x`. When invoked, the local will return its value. The value of the local may be changed using `to`. `x` can be either a single-cell item, a double-cell item or a real or complex floating-point number.

**`local-definition ( caddress -> character unsigned local-definition -- 4 th )`**

Initialize `local-definition` by erasing all members. Establish a link to the previous definition in the locals vocabulary without updating `latest`. Links will be removed when `local-definition` is deleted. Assign the value of `#locals` as the locals index to `local-definition`. Assign `local-definition` a name given by the character string `caddress -> character unsigned` and return it as `4 th`.

`local-definition` is the constructor of the `local-definition` class.

**`local-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `local-definition`.

**`locals ( -- ) immediate`**

Remove the `locals` vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the `locals` vocabulary the head of the context vocabulary list. An ambiguous condition exists if the `locals` vocabulary was not included in one of the two vocabulary lists before `locals` is executed.

**`locals( ( colon-definition "<spaces>name1<spaces>name2 ... <spaces>namen<spaces>" -- 1st ) compile-only`**

strongforth.sf

Compilation: Create local identifiers by repeatedly skipping leading spaces, parsing `name`, and executing `(local)`. The list of locals to be defined is terminated by `)`. An ambiguous condition exists if the list of locals is not terminated by `)`. Append the runtime semantics given below to the current definition.

`locals` ( may be used multiple times within a definition in order to define more than one set of locals. `colon-definition` is a dummy parameter that prevents `locals` ( to be used within loops, in conditional clauses, or between `>r` and `r>`. `1st` is `colon-definition`.

Runtime: ( `x1 x2 ... xn` -- )

Initialize `n` local identifiers, each of which takes as its initial value one of the values on the stack, in the given order. The first identifier `name1` is initialized with `x1`, identifier `name2` with `x2`, etc. When invoked, each local will return its value. The value of a local may be changed using `to`. `x` can be either a single-cell item, a double-cell item or a real or complex floating-point number.

**`locals-vocabulary ( -- vocabulary )`**

strongforth.sf

`vocabulary` is the locals vocabulary. Note that the locals vocabulary is emptied after a definition has been compiled.

**locals| ( colon-definition "<spaces>name<sub>1</sub><spaces>name<sub>2</sub> ...  
<spaces>name<sub>n</sub><spaces>" -- 1st ) compile-only**

strongforth.sf

Compilation: Create local identifiers by repeatedly skipping leading spaces, parsing *name*, and executing (`local`). The list of locals to be defined is terminated by `|`. An ambiguous condition exists if the list of locals is not terminated by `|`. Append the runtime semantics given below to the current definition.

`locals|` may be used multiple times within a definition in order to define more than one set of locals. `colon-definition` is a dummy parameter that prevents `locals|` to be used within loops, in conditional clauses, or between `>r` and `r>`. `1st` is `colon-definition`.

Runtime: ( *x<sub>n</sub>* ... *x<sub>2</sub>* *x<sub>1</sub>* -- )

Initialize *n* local identifiers, each of which takes as its initial value the top stack item, removing it from the stack. The first identifier *name<sub>1</sub>* is initialized with *x<sub>1</sub>*, identifier *name<sub>2</sub>* with *x<sub>2</sub>*, etc. When invoked, each local will return its value. The value of a local may be changed using `to`. *x* can be either a single-cell item, a double-cell item or a real or complex floating-point number.

**locase ( caddress -> character unsigned -- )**

ascii.sf

Replace each uppercase letter within the character string `caddress -> character unsigned` by the equivalent lowercase letter. All other characters remain unchanged. `locase` works for German umlauts.

**locase ( character -- 1st )**

ascii.sf

If `character` is an uppercase letter, `1st` is the equivalent lowercase letter. Otherwise, `1st` is equal to `character`. `locase` works for German umlauts.

**log ( complex -- 1st )**

complex.sf

`1st` is the complex base-ten logarithm of `complex`. An ambiguous condition exists if `float` is equal to zero.

**log ( float -- 1st )**

`1st` is the base-ten logarithm of `float`. An ambiguous condition exists if `float` is less than or equal to zero.

**logical ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `logical`.

**loop ( do-destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. Resolve both the forward references and the backward reference of `do-destination`. Delete the loop index *i*. Rename the loop index *j*, if it exists, to *i*. An exception is thrown if the contents of the compiler



data type heap do not exactly match the copy that was saved when do-destination was created.

Runtime: An ambiguous condition exists if the loop control parameters are unavailable. Add one to the loop index. If the loop index is then equal to the loop limit, discard the loop control parameters and continue execution. Otherwise, branch to the beginning of the loop.

Note: loop takes regard of the data type of the loop index.

If the loop index is an address of a single cell, the size of a single cell in address units is added to the loop index.

If the loop index is an address of a double cell, the size of a double cell in address units is added to the loop index.

If the loop index is a character address, the size of a character in address units is added to the loop index.

If the loop index is an address of a floating-point number, the size of a floating-point number in address units is added to the loop index.

If the loop index is an address of a single-precision floating-point number, the size of a single-precision floating-point number in address units is added to the loop index.

If the loop index is an address of a double-precision floating-point number, the size of a double-precision floating-point number in address units is added to the loop index.

If the loop index is an address of a complex floating-point number, the size of a complex floating-point number in address units is added to the loop index.

If the loop index is an address of a complex single-precision floating-point number, the size of a complex single-precision floating-point number in address units is added to the loop index.

If the loop index is an address of a complex double-precision floating-point number, the size of a complex double-precision floating-point number in address units is added to the loop index.

**low ( double -- single )**

single is the least significant cell of double.

**lrotate ( logical -- 1st )**

Perform a logical left rotation of one bit-place on logical, giving 1st.

**lrotate ( logical unsigned -- 1st )**

Perform a logical left rotation of unsigned bit-places on logical, giving 1st.

**lshift ( logical -- 1st )**

Perform a logical left shift of one bit-place on logical, giving 1st. Put zero into the least significant bit vacated by the shift.

**lshift ( logical unsigned -- 1st )**

Perform a logical left shift of unsigned bit-places on logical, giving 1st. Put zeros into the least significant bits vacated by the shift.

**m\* ( signed signed -- signed-double )**

`signed-double` is the double-cell product of the first `signed` and the second `signed`. All numbers and arithmetic are signed.

**m\* ( unsigned unsigned -- unsigned-double )**

`unsigned-double` is the double-cell product of the first `unsigned` and the second `unsigned`. All numbers and arithmetic are unsigned.

**m/ ( signed-double signed -- signed )**

Divide `signed-double` by `signed`, giving the symmetric quotient `signed`. An exception is thrown if `signed` is zero. An ambiguous condition exists if the quotient lies outside the range of a signed single-precision number. If both operands differ in sign, the result returned will be the same as that returned by the phrase `sm/rem nip`.

**m/ ( unsigned-double unsigned -- unsigned )**

Divide `unsigned-double` by `unsigned`, giving the quotient `unsigned`. An exception is thrown if `unsigned` is zero. An ambiguous condition exists if the quotient lies outside the range of an unsigned single-precision number.

**m/mod ( signed-double signed -- 2nd signed )**

Divide `signed-double` by `signed`, giving the symmetric quotient `signed` and the remainder `2nd`. An exception is thrown if `signed` is zero. An ambiguous condition exists if the quotient lies outside the range of a signed single-precision number. If both operands differ in sign, the result returned will be the same as that returned by `sm/rem`.

**m/mod ( unsigned-double unsigned -- 2nd unsigned )**

Divide `unsigned-double` by `unsigned`, giving the quotient `unsigned` and the remainder `2nd`. An exception is thrown if `unsigned` is zero. An ambiguous condition exists if the quotient lies outside the range of an unsigned single-precision number.

**marker ( "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

*name* Execution: Restore pointers of the data, code and stack memory spaces and the vocabulary structure as well as `default-memory-space` and `latest` as they were just prior to the definition of *name*. Remove the definition of *name* and all subsequent definitions. Restoration of any structures still existing that could refer to deleted definitions or refer to allocated memory spaces is not necessarily provided. No other contextual information such as the numeric base is affected.

**marker-class ( marker-class -- 1st )**

strongforth.sf

Initialize marker-class by storing the pointers of the data, code and stack memory spaces and the vocabulary structure as well as default-memory-space and latest as they were prior to the creation of marker-class.

marker-class is the constructor of the marker-class class.

#### **marker-class ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type marker-class.

#### **match ( control-flow -- )**

If the compiler data type heap was locked when control-flow was initialized, do nothing. Otherwise perform the semantics given below.

If the compiler data type heap is locked, restore the compiler data type heap to the state that was saved when control-flow was initialized. If the compiler data-type heap is not locked, compare the compiler data type heap with the one that was saved when control-flow was initialized. An exception is thrown if they do not exactly match.

#### **match-criterion ( -- search-criterion )**

search-criterion is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

flag is true if and only if the selected data type heap matches the input parameter list of definition. The matching algorithm follows the rules of the StrongForth data type system.

The selected data type heap depends on state, the attributes of definition and the value of single:

single	state	attributes	data type heap
false	false		interpreter
false	false	immediate	interpreter
false	false	execute-only	interpreter
false	false	compile-only	(no match)
false	true		compiler
false	true	immediate	interpreter
false	true	execute-only	(no match)
false	true	compile-only	interpreter
true	false		interpreter
true	false	immediate	interpreter
true	false	execute-only	interpreter
true	false	compile-only	(no match)
true	true		compiler
true	true	immediate	compiler
true	true	execute-only	(no match)
true	true	compile-only	compiler

**Note:** Provide search-criterion to search in order to find a definition with matching input parameters according to the rules of the StrongForth data type system.

**match? ( address -> data-type unsigned compiler-workspace -- flag )**

flag is true if and only if the data type heap saved in compiler-workspace matches the list of unsigned basic data types starting at address -> data-type. The matching algorithm follows the rules of the StrongForth data type system. The list of basic data types may contain compound data types and data type references.

**max ( address 1st -- 1st )**

1st is the unsigned maximum of address and 1st.

**max ( float 1st -- 1st )**

1st is the maximum of float and 1st.

**max ( integer 1st -- 1st )**

1st is the unsigned maximum of integer and 1st.

**max ( integer-double 1st -- 1st )**

1st is the unsigned maximum of integer-double and 1st.

**max ( signed 1st -- 1st )**

1st is the signed maximum of signed and 1st.

**max ( signed-double 1st -- 1st )**

1st is the signed maximum of signed-double and 1st.

**max-character ( -- unsigned )**

strongforth.sf

unsigned is the maximum value a character can assume.

**max-float ( -- float )**

float.sf

float is the largest usable floating-point number.

**max-precision ( -- unsigned )**

float.sf

unsigned is the maximum value precision can assume.

**max-signed ( -- signed )**

strongforth.sf

signed is the maximum value a signed single-precision number can assume.

**max-signed-double ( -- signed-double )**

strongforth.sf

`signed-double` is the maximum value a signed double-precision number can assume.

**`max-unsigned ( -- unsigned )`**

strongforth.sf

`unsigned` is the maximum value an unsigned single-precision number can assume.

**`max-unsigned-double ( -- unsigned-double )`**

strongforth.sf

`unsigned-double` is the maximum value an unsigned double-precision number can assume.

**`member ( object-size complex "<spaces>name" -- 1st )`**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to cell size, plus the size in bits of a complex floating-point number.

*name* is referred to as a class member. `member` reserves space for two floating-point numbers for a class member of the same data type as `complex` in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to `member` as `complex`.

**`member ( object-size double "<spaces>name" -- 1st )`**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to cell size, plus the number of bits in two cells.

*name* is referred to as a class member. `member` reserves two cells for a class member of the same data type as `double` in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to `member` as `double`.

**`member ( object-size float "<spaces>name" -- 1st )`**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to floating-point numbers, plus the size in bits of a floating-point number.

*name* is referred to as a class member. `member` reserves space for one floating-point number for a class member of the same data type as `float` in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to `member` as `float`.

**`member ( object-size single "<spaces>name" -- 1st )`**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to object-size aligned to cell size, plus the number of bits in one cell.

*name* is referred to as a class member. *member* reserves one cell for a class member of the same data type as *single* in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of the class member of the object *x*, that was reserved at the time *name* was created. *y* is the actual data type that was provided to *member* as *single*.

#### **member-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *member-definition*.

#### **member-definition ( unsigned address -> character unsigned member-definition -- 5 th )**

Initialize *member-definition* by erasing all members. Establish a link to the previous definition in the current vocabulary and update *latest*. Links will be removed when *member-definition* is deleted. The first unsigned is the position of the new member in bits with respect to the start address of the object. Assign *member-definition* a name given by the character string *address -> character unsigned* and return it as *5 th*.

*member-definition* is the constructor of the *member-definition* class.

#### **members ( object-size complex unsigned "<spaces>name" -- 1st )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to object-size aligned to cell size, plus unsigned times the size in bits of a complex floating-point number.

*name* is referred to as a class member. *members* reserves unsigned complex floating-point numbers for an array of unsigned class members of the same data type as *complex* in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of an array of unsigned class members of the object *x*, that were reserved at the time *name* was created. *y* is the actual data type that was provided to *member* as *complex*.

#### **members ( object-size double unsigned "<spaces>name" -- 1st )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. *1st* is equal to object-size aligned to cell size, plus unsigned times the number of bits in two cells.

*name* is referred to as a class member. *members* reserves unsigned double cells for an array of unsigned class members of the same data type as *double* in the class that is currently being defined.

Execution: ( *x* -- address -> *y* )

`address -> y` is the address of an array of unsigned class members of the object `x`, that were reserved at the time `name` was created. `y` is the actual data type that was provided to `member` as `double`.

**members ( object-size float unsigned "<spaces>name" -- 1st )** float.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a new definition for `name` with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to floating-point numbers, plus `unsigned` times the size in bits of a floating-point number.

`name` is referred to as a class member. `members` reserves unsigned floating-point numbers for an array of unsigned class members of the same data type as `float` in the class that is currently being defined.

Execution: ( `x` -- `address -> y` )

`address -> y` is the address of an array of unsigned class members of the object `x`, that were reserved at the time `name` was created. `y` is the actual data type that was provided to `member` as `float`.

**members ( object-size single unsigned "<spaces>name" -- 1st )** strongforth.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a new definition for `name` with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to cell size, plus `unsigned` times the number of bits in one cell.

`name` is referred to as a class member. `members` reserves unsigned cells for an array of unsigned class members of the same data type as `single` in the class that is currently being defined.

Execution: ( `x` -- `address -> y` )

`address -> y` is the address of an array of unsigned class members of the object `x`, that were reserved at the time `name` was created. `y` is the actual data type that was provided to `member` as `single`.

**memory-space ( address unsigned memory-space -- 3rd )**

Initialize `memory-space` by assigning `address` as its bottom and `unsigned` as its size in address units. The memory space consists of unsigned unused address units.

`memory-space` is a constructor of the `memory-space` class.

**memory-space ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `memory-space`.

**memory-space ( unsigned memory-space -- 2nd )**

Allocate unsigned address units of dynamic memory. Initialize `memory-space` by assigning it the address of the allocated memory as its bottom and `unsigned` as its size. The memory space consists of unsigned unused address units.

`memory-space` is a constructor of the `memory-space` class.

**`merge ( float float -- complex )`**

Merges two floating-point numbers `float` into a complex floating-point number `complex`. The first one becomes the real part, the second one becomes the imaginary part.

**`merge ( single single -- double )`**

Merges two single-cell items `single` into a double-cell item `double`. The most significant part is expected on top of the stack.

**`min ( address 1st -- 1st )`**

`1st` is the unsigned minimum of `address` and `1st`.

**`min ( float 1st -- 1st )`**

`1st` is the minimum of `float` and `1st`.

**`min ( integer 1st -- 1st )`**

`1st` is the unsigned minimum of `integer` and `1st`.

**`min ( integer-double 1st -- 1st )`**

`1st` is the unsigned minimum of `integer-double` and `1st`.

**`min ( signed 1st -- 1st )`**

`1st` is the signed minimum of `signed` and `1st`.

**`min ( signed-double 1st -- 1st )`**

`1st` is the signed minimum of `signed-double` and `1st`.

**`mod ( signed signed -- 2nd )`**

Divide the first `signed` by the second `signed`, giving the remainder `2nd`. An exception is thrown if the second `signed` is zero. If the first `signed` and the second `signed` differ in sign, the result returned will be the same as that returned by the phrase `swap s>d swap sm/rem drop`.

**`mod ( signed-double signed -- 2nd )`**

`strongforth.sf`

Divide `signed-double` by `signed`, giving the remainder `2nd`. An exception is thrown if `signed` is zero. If `signed-double` and `signed` differ in sign, the result returned will be the same as that returned by the phrase `sm/rem drop`.



**mod ( unsigned unsigned -- 2nd )**

Divide the first unsigned by the second unsigned, giving the remainder 2nd. An exception is thrown if the second unsigned is zero.

**mod ( unsigned-double unsigned -- 2nd )**

Divide unsigned-double by unsigned, giving the single-precision remainder 2nd. An exception is thrown if unsigned is zero.

**move ( address -> complex 1st unsigned -- )**

If unsigned is not equal to zero, copy the contents of unsigned consecutive complex floating-point numbers starting at address -> complex to address 1st. After move completes, the unsigned consecutive complex floating-point numbers starting at address 1st contain exactly what the unsigned consecutive complex floating-point numbers starting at address -> complex contained before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( address -> double 1st unsigned -- )**

If unsigned is not equal to zero, copy the contents of unsigned consecutive double cells starting at address -> double to the unsigned consecutive double cells starting at address 1st. After move completes, the unsigned consecutive double cells starting at address 1st contain exactly what the unsigned consecutive double cells starting at address -> double contained before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( address -> float 1st unsigned -- )**

If unsigned is not equal to zero, copy unsigned consecutive floating-point numbers starting at address -> float to address 1st. After move completes, the unsigned consecutive floating-point numbers starting at address 1st are identical to the unsigned consecutive floating-point numbers starting at address -> float before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( address -> single 1st unsigned -- )**

If unsigned is not equal to zero, copy the contents of unsigned consecutive cells starting at address -> single to the unsigned consecutive cells starting at address 1st. After move completes, the unsigned consecutive cells starting at address 1st contain exactly what the unsigned consecutive cells starting at address -> single contained before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( caddress 1st unsigned -- )**

If unsigned is not equal to zero, copy the contents of unsigned consecutive character-size items starting at caddress to the unsigned consecutive character-size items starting at address 1st. After move completes, the unsigned consecutive character-size items starting at address

1st contain exactly what the unsigned consecutive character-size items starting at address contained before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( dfaddress -> complex 1st unsigned -- )**

If unsigned is not equal to zero, copy unsigned consecutive complex double-precision floating-point numbers starting at dfaddress -> complex to address 1st. After move completes, the unsigned consecutive complex double-precision floating-point numbers starting at address 1st are identical to the unsigned consecutive complex double-precision floating-point numbers starting at dfaddress -> complex before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( dfaddress 1st unsigned -- )**

If unsigned is not equal to zero, copy unsigned consecutive double-precision floating-point numbers starting at dfaddress to address 1st. After move completes, the unsigned consecutive double-precision floating-point numbers starting at address 1st are identical to the unsigned consecutive double-precision floating-point numbers starting at dfaddress before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( sfaddress -> complex 1st unsigned -- )**

If unsigned is not equal to zero, copy unsigned consecutive complex single-precision floating-point numbers starting at sfaddress -> complex to address 1st. After move completes, the unsigned consecutive complex single-precision floating-point numbers starting at address 1st are identical to the unsigned consecutive complex single-precision floating-point numbers starting at sfaddress -> complex before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**move ( sfaddress 1st unsigned -- )**

If unsigned is not equal to zero, copy unsigned consecutive single-precision floating-point numbers starting at sfaddress to address 1st. After move completes, the unsigned consecutive single-precision floating-point numbers starting at address 1st are identical to the unsigned consecutive single-precision floating-point numbers starting at sfaddress before.

Note: The source memory will be partly overwritten if the memory areas overlap.

**ms ( unsigned -- )**

strongforth.sf

Wait at least unsigned milliseconds.

**msvcrt ( -- ) immediate**

Remove the msvcrt vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the msvcrt vocabulary the head of the context vocabulary list. An ambiguous condition exists if the msvcrt vocabulary was not included in one of the two vocabulary lists before msvcrt is executed.

**name ( definition -- caddress -> character unsigned )**

*caddress -> character unsigned* is a character string representing the name of definition. *caddress -> character* is a null address and *unsigned* is zero if definition has no name.

**negate ( complex -- 1st )**

complex.sf

Negate complex, giving its arithmetic inverse 1st.

**negate ( float -- 1st )**

Negate float, giving its arithmetic inverse 1st.

**negate ( integer -- 1st )**

Negate integer, giving its arithmetic inverse 1st. *integer* is assumed to be a signed numeric value.

**negate ( integer-double -- 1st )**

Negate integer-double, giving its arithmetic inverse 1st. *integer-double* is assumed to be a signed numeric value.

**new ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Append the runtime semantics given below to the current definition.

Runtime: Execute (new) in order to create an object with data type *name*. Initialize the new object by compiling *name*.

**new ( "<spaces>name" -- ) execute-only**

strongforth.sf

Interpretation: Skip leading space delimiters. Parse *name* delimited by a space. Evaluate (new) in order to create an object with data type *name*. Initialize the new object by interpreting *name*.

**new-included-file ( caddress -> character unsigned -- )**

strongforth.sf

Create a definition with the name specified by the string *caddress -> character unsigned* and the execution semantics defined below, and make it the latest definition. The definition is a marker for an included source file. It is not supposed to be executed.

Execution: Throw an exception.

**next ( vocabulary -- vocabulary )**

*vocabulary* (output parameter) is the vocabulary succeeding *vocabulary* (input parameter) in the vocabulary list it belongs to, or null if *vocabulary* (input parameter) is the last vocabulary of the list.

**next! ( vocabulary vocabulary -- )**

Make the first `vocabulary` succeed the second `vocabulary` in the `vocabulary` list the second one belongs to.

**nip ( complex complex -- 2nd )**

Remove the first item below the top of the stack.

**nip ( complex double -- 2nd )**

Remove the first item below the top of the stack.

**nip ( complex float -- 2nd )**

Remove the first item below the top of the stack.

**nip ( complex single -- 2nd )**

Remove the first item below the top of the stack.

**nip ( double complex -- 2nd )**

Remove the first item below the top of the stack.

**nip ( double double -- 2nd )**

Remove the first item below the top of the stack.

**nip ( double float -- 2nd )**

Remove the first item below the top of the stack.

**nip ( double single -- 2nd )**

Remove the first item below the top of the stack.

**nip ( float complex -- 2nd )**

Remove the first item below the top of the stack.

**nip ( float double -- 2nd )**

Remove the first item below the top of the stack.

**nip ( float float -- 2nd )**

Remove the first item below the top of the stack.

**nip ( float single -- 2nd )**

Remove the first item below the top of the stack.

**nip ( single complex -- 2nd )**

Remove the first item below the top of the stack.

**nip ( single double -- 2nd )**

Remove the first item below the top of the stack.

**nip ( single float -- 2nd )**

Remove the first item below the top of the stack.

**nip ( single single -- 2nd )**

Remove the first item below the top of the stack.

**no-criterion ( -- search-criterion )**

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

Drop definition and single. flag is true.

Note: Provide `search-criterion` to search if no additional search criterion shall be applied.

**nodelete ( object -- )**

Throws an exception, indicating that `object` can or may not be deleted.

**noop ( -- )**

Interpretation: No operation.

Compilation: No operation.

**not-decimal? ( -- flag )**

float.sf

flag is true if and only if the value of the number-conversion radix base is not (decimal) 10.

**null ( "<spaces>name" -- ) immediate**

strongforth.sf

Interpretation: ( -- y )

Skip leading space delimiters. Parse *name* delimited by a space. Return *y*, which has the numerical value 0 (all bits are zero) and the data type identified by *name*. *y* can be a single-cell or double-cell item or a real or complex floating-point number. An exception is thrown if *name* is not the name of a data type.

Compilation: ( -- )

Skip leading space delimiters. Parse *name* delimited by a space. Append the runtime semantics given below to the current definition. An exception is thrown if *name* is not the name of a data type.

Runtime: ( -- *y* )

Return *y*, which has the numerical value 0 (all bits are zero) and the data type identified by *name*. *y* can be a single-cell or double-cell item or a real or complex floating-point number.

**number-double ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type `number-double`.

**object ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `object`.

**object-size ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type `object-size`.

**object? ( data-type -- flag )**

strongforth.sf

*flag* is true if and only if *data-type* is equal to `object`, or if *data-type* is directly or indirectly derived from `object`.

**octal ( -- )**

Set the number-conversion radix to 8 (octal).

**odd-parity? ( character -- flag )**

*flag* is true if and only if *character* has odd parity.

**of ( endof-origin of-origin -- 2nd 1st ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. Check if the contents of the compiler data type heap exactly matches the one that was saved when `of-origin` was created. An exception is thrown if a difference is detected. *2nd* is `endof-origin`. The semantics are incomplete until resolved by a consumer of *2nd*, such as `endcase`, and *1st*, such as `endof`.

Runtime: ( single *1st* -- ) | ( single *1st* -- *1st* )

If *single* and *1st* are equal, discard both items and continue execution. Otherwise, discard *1st* and continue execution at the location specified by the consumer of *2nd* and *1st*.

**of-origin ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `of-origin`.

**offset ( data-type -- unsigned )**

If `data-type` has the reference attribute, `unsigned` is the index of the basic data type it refers to, starting with 1. Otherwise, `unsigned` is 0.

**only ( -- )**

`order.sf`

Remove all vocabularies from the context vocabulary list and add them to the hidden vocabulary list. Add the `forth` vocabulary to the context vocabulary list and remove it from the hidden vocabulary list.

**open ( caddress -> character unsigned fam -- file )**

`strongforth.sf`

Open the file with the name given by the character string `ccaddress -> character unsigned` as `file` with file access method `fam`. Position the file pointer to the start of the file. An exception is thrown if the file does not exist or the operation fails.

Note: If the file is opened with “write only” file access method, its contents will be destroyed.

**or ( data-type data-type -- 1st )**

`1st` is the first `data-type` with attributes that are the bit-by-bit logical or of the attributes of both parameters `data-type`.

**or ( single logical -- 1st )**

`1st` is the bit-by-bit inclusive-or of `single` with `logical`.

**order ( -- )**

`strongforth.sf`

Send the names of the current compilation vocabulary and the names of all vocabularies in the context vocabulary list to the default output stream. If a vocabulary is the protected vocabulary of a class, send the class name instead of the vocabulary name to the default output stream.

**origin ( origin -- 1st )**

Initialize `origin` by erasing all members. If the compiler data type heap is not locked, save a copy of the present compiler data type heap. Save the value of the `code-space` memory space pointer as the code origin.

`origin` is the constructor of the `origin` class.

**origin ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `origin`.

**output-params ( definition -- address -> data-type unsigned )**

`address -> data-type` is the address of the first output parameter of the stack diagram of definition. `unsigned` is the number of basic data types in the output parameter list of the stack diagram of definition.

**`output-params ( stack-diagram -- address -> data-type unsigned )`**

`address -> data-type` is the address of the first output parameter of `stack-diagram`. `unsigned` is the number of basic data types in the output parameter list of `stack-diagram`.

**`output-stream ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `output-stream`.

**`over ( complex complex -- 1st 2nd 1st )`**

Place a copy of the first `complex` on top of the stack.

**`over ( complex double -- 1st 2nd 1st )`**

Place a copy of `complex` on top of the stack.

**`over ( complex float -- 1st 2nd 1st )`**

Place a copy of `complex` on top of the stack.

**`over ( complex single -- 1st 2nd 1st )`**

Place a copy of `complex` on top of the stack.

**`over ( double complex -- 1st 2nd 1st )`**

Place a copy of `double` on top of the stack.

**`over ( double double -- 1st 2nd 1st )`**

Place a copy of the first `double` on top of the stack.

**`over ( double float -- 1st 2nd 1st )`**

Place a copy of `double` on top of the stack.

**`over ( double single -- 1st 2nd 1st )`**

Place a copy of `double` on top of the stack.

**`over ( float complex -- 1st 2nd 1st )`**

Place a copy of `float` on top of the stack.



**over ( float double -- 1st 2nd 1st )**

Place a copy of float on top of the stack.

**over ( float float -- 1st 2nd 1st )**

Place a copy of the first float on top of the stack.

**over ( float single -- 1st 2nd 1st )**

Place a copy of float on top of the stack.

**over ( single complex -- 1st 2nd 1st )**

Place a copy of single on top of the stack.

**over ( single double -- 1st 2nd 1st )**

Place a copy of single on top of the stack.

**over ( single float -- 1st 2nd 1st )**

Place a copy of single on top of the stack.

**over ( single single -- 1st 2nd 1st )**

Place a copy of the first single on top of the stack.

**overflow? ( -- flag )**

flag is true if and only if the directly preceding operation caused the processor's overflow flag to be set.

**pack ( single single single single -- single )**

Packs the least significant bytes of four items with data type single into one cell single.

The least significant byte of the first single becomes the most significant byte of single. The least significant byte of the second single becomes the second most significant byte of single. The least significant byte of the third single becomes the second least significant byte of single. The least significant byte of the fourth single becomes the least significant byte of single.

**pad ( -- caddress -> character )**

strongforth.sf

caddress -> character is the address of a scratch area that can be used to hold data for intermediate character string processing. The scratch area is /pad characters long.

Note: This scratch area is reserved for applications. It will not be used by the system.

**page ( -- )**

strongforth.sf

Clear the console window by writing 25 lines of 80 space characters each and positioning the cursor at the upper left corner of the screen.

**param, ( data-type stack-diagram -- )**

Append `data-type` as an input or output parameter to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params! ( address -> data-type definition -- )**

Assign a stack diagram to `definition`. The stack diagram has no input parameters and one output parameter, which is stored as a compound data type at `address -> data-type`.

**params! ( code-definition created-definition -- )**

strongforth.sf

Assign a stack diagram to `created-definition`, using the input and output parameters of `code-definition`. The last input parameter of `code-definition` is excluded.

**params! ( data-type address -> data-type definition -- )**

strongforth.sf

Assign a stack diagram to `definition`. The stack diagram has no input parameters and one output parameter. The output parameter is a compound data type which is composed of `data-type` as the head and the compound data type at `address -> data-type` as the tail.

**params! ( data-type address -> data-type member-definition -- )**

strongforth.sf

Assign a stack diagram to `member-definition`. The stack diagram has one input parameter and one output parameter. The input parameter has the data type of the class that is currently being defined. The output parameter is a compound data type which is composed of `data-type` as the head and the compound data type at `address -> data-type` as the tail.

**params! ( data-type member-definition -- )**

struct.sf

Assign a stack diagram to `member-definition`. The stack diagram has one input parameter and one output parameter. The input parameter has the data type of the class that is currently being defined. The output parameter is `data-type`.

**params! ( stack-diagram definition -- )**

Throws an exception if `stack-diagram` is incomplete. Assign `stack-diagram` to `definition`. Delete `stack-diagram`.

**params, ( address -> data-type stack-diagram -- )**

strongforth.sf

Append a compound data type stored at `address -> data-type` as an input or output parameter to `stack-diagram`. Data type attributes are removed, except for the reference and the prefix attributes. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params, ( address -> data-type unsigned stack-diagram -- )**

strongforth.sf

Append unsigned basic data types stored at `address -> data-type` as input or output parameters to `stack-diagram`. Data type attributes are removed, except for the reference and the prefix attributes. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params, ( definition stack-diagram -- )**

strongforth.sf

Append the input parameters of `definition` as input parameters to `stack-diagram`. Append the output parameters of `definition` as output parameters to `stack-diagram`. Data type attributes are removed, except for the reference and the prefix attributes. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params-alias, ( address -> data-type unsigned stack-diagram -- )**

strongforth.sf

Append unsigned basic data types stored at `address -> data-type` as input or output parameters to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params-alias, ( definition stack-diagram -- )**

strongforth.sf

Append the input parameters of `definition` as input parameters to `stack-diagram`. Append the output parameters of `definition` as output parameters to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params-stripped, ( definition stack-diagram -- )**

strongforth.sf

Append the input parameters of `definition`, except for the last input parameter, as input parameters to `stack-diagram`. Append the output parameters of `definition` as output parameters to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params-virtual, ( definition stack-diagram -- )**

strongforth.sf

Append the input parameters of `definition` as input parameters to `stack-diagram`, replacing the last input parameter with the data type of the class that is currently being defined. Append the output parameters of `definition` as output parameters to `stack-diagram`. An exception is thrown if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**params>dt ( definition -- )**

Append the input parameters of `definition` to the data type heap selected by `state`, starting with the first input parameter. Data type references within the input parameter list are being resolved by recursively appending the referenced data types onto the data type heap. An exception is thrown if the data type heap overflows.

**parent ( data-type -- 1st )**

1st is the parent data type of data-type without any attributes set. If data-type does not have a parent data type, 1st is null.

**parent-attributes ( -- class-attributes )**

strongforth.sf

class-attributes is the attributes of the parent of the data type of the class that is currently being defined.

**parent-vtable ( -- vtable )**

strongforth.sf

vtable is the virtual method table of the parent of the class that is currently being defined.

**parse ( character "ccc<delimiter>" -- caddress -> character  
unsigned )**

Parse *ccc* delimited by character.

caddress -> character is the address within the input buffer and unsigned is the length of the parsed string. If the parse area was empty, unsigned is zero.

**parse-deferred-definition ( "<spaces>name" -- deferred-definition  
)**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Find a deferred definition with the name *name* and return it as deferred-definition. If no such deferred definition exists, throw an exception and return null as deferred-definition.

Deferred definitions are words defined by defer.

**parse-name ( "<spaces>name" -- caddress -> character unsigned )**

Skip leading space delimiters. Parse *name* delimited by a space.

caddress -> character is the address within the input buffer and unsigned is the length of *name*. If the parse area was empty, unsigned is zero.

Delimiters are the space character and any character with an ASCII value less than that of a space character.

**period ( -- )**

float.sf

Send a period (.) to the default output stream.

**pi ( -- float )**

float is  $\pi$  (3.14159265358979323846).

**picture ( double -- caddress -> character unsigned )**

strongforth.sf

caddress -> character unsigned is the picture of double as an unsigned double-precision number in free field format.

<b>picture ( signed-double -- caddress -&gt; character unsigned )</b>	strongforth.sf
caddress -> character unsigned is the picture of signed-double as a signed double-precision number in free field format.	
<b>position ( file -- unsigned-double )</b>	strongforth.sf
unsigned-double is the current file position for the file identified by <i>file</i> . An exception is thrown if the operation fails.	
<b>position-block ( unsigned -- )</b>	block.sf
Position the file pointer of the block file to the first character of the block with the number <i>unsigned</i> .	
<b>postpone ( "&lt;spaces&gt;name" -- ) compile-only</b>	strongforth.sf
Compilation: Skip leading space delimiters. Parse <i>name</i> delimited by a space. Find <i>name</i> . Append the compilation semantics of <i>name</i> to the current definition. An exception is thrown if <i>name</i> is not found.	
<b>precision ( -- unsigned )</b>	float.sf
unsigned is the number of significant digits currently used by ., e., or s.. Since <i>precision</i> is a value, it can be reassigned with <i>to</i> .	
<b>prefix? ( data-type -- flag )</b>	strongforth.sf
<i>flag</i> is true if and only if <i>data-type</i> has the <i>dt-prefix</i> attribute.	
<b>prev ( definition -- definition )</b>	
The second <i>definition</i> is the predecessor of the first <i>definition</i> in the same vocabulary, or null if the first <i>definition</i> has no predecessor.	
<b>prev ( exception-frame -- exception-frame )</b>	
The second <i>exception-frame</i> is the next higher level exception frame of the first <i>exception-frame</i> , or null if the first <i>exception-frame</i> is at the highest level.	
<b>private ( -- ) immediate</b>	
Remove the <i>private</i> vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the <i>private</i> vocabulary the head of the context vocabulary list. An ambiguous condition exists if the <i>private</i> vocabulary was not included in one of the two vocabulary lists before <i>private</i> is executed.	
<b>private-vocabulary ( -- vocabulary )</b>	strongforth.sf

`vocabulary` is the private vocabulary. Note that the private vocabulary is not available outside the scope of a class definition.

**procreates ( data-type "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. The definition identifies a new data type that is a direct subtype of `data-type` and has the same size. An ambiguous condition exists if `procreates` is executed in compilation state or if `data-type` is not a valid data type.

Execution: ( stack-diagram -- 1st )

When used in a stack diagram, specifies an input or output parameter with the new data type.

**procreates ( data-type unsigned "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. The definition identifies a new data type that is a direct subtype of `data-type` and has a size of unsigned address units. If `data-type` is null, the new data type has no parent data type. An ambiguous condition exists if `procreates` is executed in compilation state.

Execution: ( stack-diagram -- 1st )

When used in a stack diagram, specifies an input or output parameter with the new data type.

**prompt ( -- )**

If in interpretation state, type a command line prompt.

A typical command line prompt is " ok" (including a leading space) followed by carriage return and line feed.

`prompt` is a deferred definition.

**protected ( -- ) immediate**

Remove the `protected` vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the `protected` vocabulary the head of the context vocabulary list. An ambiguous condition exists if the `protected` vocabulary was not included in one of the two vocabulary lists before `protected` is executed.

**protected-vocabulary ( -- vocabulary )**

strongforth.sf

`vocabulary` is the protected vocabulary. Note that the protected vocabulary is not available outside the scope of a class definition.

**quit ( -- )**

If in compilation state, end compilation. Empty the stack. Empty the interpreter data type heap. Remove the `private`, `protected` and `assembler` vocabularies from the context vocabulary list. Delete all exception frames. Make the user input device the default input source. Make the user output device the default output source. Execute the semantics of the deferred definition (`quit`).

Do not send a message to the default output stream. Repeat the following until the end of the input source:

Accept a line from the input source, set `>in` to zero, and interpret. When all processing has been completed and no exception is thrown, execute `prompt`.

After the end of the input source has exceeded, terminate StrongForth and return control to the operating system.

**r-index ( stack-diagram -- 1st )**

strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type `r-index`.

**r/o ( -- fam )**

strongforth.sf

`fam` is a constant value for selecting the “read only” file access method when a file is created or opened.

**r/w ( -- fam )**

strongforth.sf

`fam` is a constant value for selecting the “read and write” file access method when a file is created or opened.

**r> ( r-index -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition. Consume `r-index`, the number of cells reserved for locals in the stack frame of the current definition, and remove the local `r@` from the locals vocabulary. An exception is thrown if `r@` does not exist.

Runtime: ( -- `x` )

`x` is the value of the local `r@`.

**re ( complex -- float )**

`float` is the real part of `complex`.

**read ( caddress -> character unsigned file -- 3rd )**

Read unsigned consecutive characters to `caddress -> character` from the current position of the file identified by `file`. If unsigned characters are read without an exception, `3rd` is equal to `unsigned`. If the end of the file is reached before unsigned characters are read, `3rd` is the number of characters actually read. If the operation is initiated when the value returned by `position` is equal to the value returned by `size` for the file identified by `file`, `3rd` is zero.

An ambiguous condition exists if the operation is initiated when the value returned by `position` is greater than the value returned by `size` for the file identified by `file`, or if the requested operation attempts to read portions of the file not written. At the conclusion of the operation, `position` returns the next file position after the last character read.

**read-line ( caddress -> character unsigned file -- 3rd flag )**

Read the next line from the file specified by `file` into memory at `caddress -> character`. At most `unsigned` characters are read. Up to two line-terminating characters (carriage return and line feed) may be read into memory at the end of the line, but are not included in the count `3rd`. The line buffer provided by `caddress -> character` should be at least `unsigned + 2` characters long. If the operation succeeded, `flag` is `true`. If a line terminator was received before `unsigned` characters were read, then `3rd` is the number of characters actually read, not including the line terminator ( $0 \leq 3rd \leq \text{unsigned}$ ). When `unsigned = 3rd`, the line terminator has yet to be reached. If the operation is initiated when the value returned by `position` is equal to the value returned by `size` for the file identified by `file`, `flag` is `false`, and `3rd` is zero.

An ambiguous condition exists if the operation is initiated when the value returned by `position` is greater than the value returned by `size` for the file identified by `file`, or if the requested operation attempts to read portions of the file not written. At the conclusion of the operation, `position` returns the next file position after the last character read.

`read-line` works correctly with files containing `<cr>/<lf>` or `<lf>`-only end-of-line sequences.

#### **`recurse ( -- ) compile-only`**

strongforth.sf

Compilation: Change the compiler data type heap according to the stack effect of the current definition. Append the execution semantics of the current definition to the execution semantics of current definition. An ambiguous condition exists if `recurse` appears in a definition after `does>`.

#### **`reempty ( string-output-stream -- )`**

strongforth.sf

Empty the output buffer of `string-output-stream`.

#### **`reference? ( data-type -- flag )`**

strongforth.sf

`flag` is `true` if and only if `data-type` has the `dt-reference` attribute.

#### **`refill ( -- flag )`**

Attempt to fill the input buffer of the default input stream. If successful, set `>in` to zero, and return `true` as `flag`. Receipt of a line containing no characters is considered successful. If there is no input available from the default input stream, return `false` as `flag`.

#### **`refill ( input-stream -- flag )`**

Attempt to fill the input buffer of `input-stream`. If successful set `>in` to zero and return `true` as `flag`. Receipt of a line containing no characters is considered successful. If there is no input available from `input-stream`, return `false` as `flag`.

`refill` is a virtual method of the `input-stream` class.

#### **`relocate ( address control-flow -- )`**

Specifies `address` as the code origin or code destination of `control-flow`. The code origin is typically the location of the first instruction of a conditional branch after a jump instruction. The code destination is typically the location of the first instruction of a loop.



**rename ( address -> character unsigned address -> character unsigned -- )**

strongforth.sf

Rename the file with the path given by the first character string `address -> character unsigned` to the name given by the second character string `address -> character unsigned`. An exception is thrown if the operation fails.

**repeat ( origin destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition, resolving the backward reference `destination`. An exception is thrown if the contents of the compiler data type heap do not exactly match the copy that was saved when `destination` was created. Resolve the forward reference `origin` using the location following the appended runtime semantics. Restore the compiler data type heap to the state that was saved when `origin` was created.

Runtime: Continue execution at the location given by `destination`.

**replaces ( (--string) address -> character unsigned -- )**

strext.sf

Set `(--string)` as the execution token that provides the text to substitute for the substitution named `address -> character unsigned`. An ambiguous condition exists if the name of the substitution contains the delimiter character.

If the substitution does not exist, create a definition with the name specified by the character string `address -> character unsigned` with the execution semantics defined below.

Execution: ( -- address -> character unsigned )

`address -> character unsigned` is the text returned by executing `(--string)`.

**replaces ( address -> character unsigned 1st 3rd -- )**

strext.sf

Set the string `address -> character unsigned` as the text to substitute for the substitution named `1st 3rd`. An ambiguous condition exists if the name of the substitution contains the delimiter character.

If the substitution does not exist, create a definition with the name specified by the character string `1st 3rd` with the execution semantics defined below.

Execution: ( -- address -> character unsigned )

`address -> character unsigned` is a copy of the text provided to `replaces`.

**reposition ( unsigned-double file -- )**

strongforth.sf

Reposition the file identified by `file` to the file position `unsigned-double`. An exception is thrown if the operation fails.

**represent ( float address -> character unsigned -- signed flag flag )**

float.sf

At `address -> character`, place the character-string external representation of the significand of the floating-point number `float`. Return the decimal-base exponent as signed,

the sign of the significand as the first `flag` and valid result as the second `flag`. The character string consists of the unsigned most significant digits of the significand represented as a decimal fraction with the implied decimal point to the left of the first digit, and the first digit zero only if all digits are zero. The significand is rounded to unsigned digits following the round to nearest rule; signed is adjusted, if necessary, to correspond to the rounded magnitude of the significand. The second `flag` is `true` if and only if `float` was a valid floating-point number. The first `flag` is `true` if and only if `float` is negative.

An exception is thrown if the value of the number-conversion radix `base` is not (decimal) 10.

When the second `flag` is `false`, `signed` is always zero, and the string at `caddress -> character` contains one of the following, cut off to unsigned characters or extended by trailing spaces:

```
unsupported
nan
infinity
denormalized
free
```

**`require ( "<spaces>name" -- )`**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. If the file specified by *name* has not already been included or required, but not between the definition and execution of a marker, perform the function of `include`.

An ambiguous condition exists if a file is required while it is being required or included. An ambiguous condition exists, if a marker is defined outside and executed inside a file or vice versa, and the file is required again. An ambiguous condition exists if the same file is required twice using different names, or different files with the same name are required.

**`require ( caddress -> character unsigned -- )`**

strongforth.sf

If the file specified by the string `caddress -> character unsigned` has already been included or required, but not between the definition and execution of a marker, drop `caddress -> character unsigned`. Otherwise, perform the function of `include`.

An ambiguous condition exists if a file is required while it is being required or included. An ambiguous condition exists, if a marker is defined outside and executed inside a file or vice versa, and the file is required again. An ambiguous condition exists if the same file is required twice using different names, or different files with the same name are required.

**`resize ( address unsigned -- 1st )`**

strongforth.sf

Change the size of a memory block that has been previously allocated at `address`. `unsigned` is the new size of the memory block. If `address` is null, `resize` performs the semantics of `allocate`. If `unsigned` is zero, `resize` performs the semantics of `free`. `1st` is the address of the first byte of the reallocated memory block. Note that `1st` may differ from `address`. An exception is thrown if the operation fails. An ambiguous condition exists if `address` does not indicate a memory space that was previously obtained by `allocate`, `callocate`, `dfallocate`, `sfallocate` or `resize`.

**`resize ( unsigned-double file -- )`**

strongforth.sf

Set the size of the file identified by `file` to `unsigned-double`. If the resultant file is larger than the file before the operation, the portion of the file added as a result of the operation is undefined. At the conclusion of the operation, both `size` and `position` return the value `unsigned-double`. An exception is thrown if the operation fails.

#### **resolve ( control-flow -- )**

Resolve the conditional branch to or from the location that was saved when `control-flow` was initialized. Recursively resolve the conditional branches from the locations of all linked origins of `control-flow`.

`resolve` is a virtual method of the `control-flow` class.

#### **restore ( control-flow -- )**

Restore the compiler data type heap to the state that was saved when `control-flow` was initialized.

#### **restore-input ( input-stream -- flag )**

Attempt to restore the default input stream from `input-stream`, which was created by `save-input`. `flag` is `true` if and only if the default input stream cannot be restored.

An ambiguous condition exists if the input source of the default input stream is not the same as that of `input-stream`.

#### **restore-input ( input-stream 1st -- flag )**

Attempt to restore `1st` from `input-stream`, which was created by `save-input`. `flag` is `true` if and only if `1st` cannot be restored.

An ambiguous condition exists if the input source of `1st` is not the same as that of `input-stream`.

`restore-input` is a virtual method of the `input-stream` class.

#### **retreat ( unsigned -- )**

strongforth.sf

Change the order in which definitions are linked within the current vocabulary by inserting the latest definition immediately before the previous `unsigned`<sup>th</sup> overloaded version with the same name. An exception is thrown if the latest definition has less than `unsigned` previously defined overloaded versions.

#### **return-stack-cells ( -- unsigned )**

strongforth.sf

`unsigned` is the size of the return stack in cells. Note that StrongForth uses the return stack to store data as well.

#### **rot ( complex complex complex -- 2nd 3rd 1st )**

Rotate the top three stack entries.

```
rot ( complex complex double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex complex float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex complex single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex double complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex double double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex double float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex double single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex float complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex float double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex float float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex float single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex single complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex single double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex single float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( complex single single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double complex complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double complex double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double complex float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double complex single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double double complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double double double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double double float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double double single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double float complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double float double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double float float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double float single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double single complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double single double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double single float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( double single single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float complex complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float complex double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float complex float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float complex single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float double complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float double double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float double float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float double single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float float complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float float double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float float float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float float single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float single complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float single double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float single float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( float single single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single complex complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single complex double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single complex float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single complex single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single double complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single double double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single double float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single double single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single float complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single float double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single float float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single float single -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single single complex -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single single double -- 2nd 3rd 1st )
```

Rotate the top three stack entries.

```
rot ( single single float -- 2nd 3rd 1st )
```

Rotate the top three stack entries.



**rot ( single single single -- 2nd 3rd 1st )**

Rotate the top three stack entries.

**round ( float -- 1st )**

Round `float` to an integral value using the round to nearest rule, giving 1st.

**rrotate ( logical -- 1st )**

Perform a logical right rotation of one bit-place on `logical`, giving 1st.

**rrotate ( logical unsigned -- 1st )**

Perform a logical right rotation of unsigned bit-places on `logical`, giving 1st.

**rshift ( logical -- 1st )**

Perform a logical right shift of one bit-place on `logical`, giving 1st. Put zero into the most significant bit vacated by the shift.

**rshift ( logical unsigned -- 1st )**

Perform a logical right shift of unsigned bit-places on `logical`, giving 1st. Put zeros into the most significant bits vacated by the shift.

**runtime ( created-definition -- code-definition )**

`code-definition` is the definition containing the runtime code of `created-definition`.

**runtime! ( code-definition created-definition -- )**

Specifies `code-definition` as the definition containing the runtime code of `created-definition`.

**runtime-criterion ( -- search-criterion )**

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( `definition single -- flag` )

`flag` is true if and only if `definition` was created by `create` and its runtime code-definition is equal to `single`.

Note: Provide `search-criterion` to search if only definitions created by `create` with a given runtime code-definition shall be found.

**s. ( complex -- )**

complex.sf

Send the real part and the imaginary part of `complex` with a trailing space using scientific notation to the default output stream. The significands are greater than or equal to 1.0 and less than 10.0:

```
exponential notation := <re> + <im> i
<re>                 := <significand><exponent>
<im>                 := <significand><exponent>
<significand>        := [-]<digits>.<digits0>
<exponent>           := e[-]<digit><digit><digit>
<digits>             := <digit><digits0>
<digits0>            := <digit>*
<digit>              := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**s. ( float -- )**

float.sf

Send `float` with a trailing space using scientific notation to the default output stream. The significand is greater than or equal to 1.0 and less than 10.0:

```
exponential notation := <significand><exponent>
<significand>        := [-]<digits>.<digits0>
<exponent>           := e[-]<digit><digit><digit>
<digits>             := <digit><digits0>
<digits0>            := <digit>*
<digit>              := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

An exception is thrown if the value of the number-conversion radix base is not (decimal) 10.

**s>d ( integer -- integer-double )**

Convert the unsigned single number `integer` to the unsigned double number `integer-double` with the same numerical value.

**s>d ( signed -- signed-double )**

Convert the signed single number `signed` to the signed double number `signed-double` with the same numerical value.

**s>d ( single -- double )**

Convert the unsigned single number `single` to the unsigned double number `double` with the same numerical value.

**s>d ( unsigned -- unsigned-double )**

Convert the unsigned single number `unsigned` to the unsigned double number `unsigned-double` with the same numerical value.

**s>f ( signed -- float )**

`float` is the floating-point equivalent of the signed single number `signed`.

**s>f ( single -- float )**

float is the floating-point equivalent of the unsigned single number single.

**save-buffers ( -- )**

block.sf

If the block buffer is marked as modified, transfer its contents to the block file. Mark the block buffer as not modified. An exception is thrown if the block buffer is assigned to an invalid block.

**save-input ( -- input-stream )**

input-stream is a copy of the default input stream made for later use by restore-input. Note that the copy shares the input buffer with the original.

**save-input ( input-stream -- 1st )**

1st is a copy of the input-stream made for later use by restore-input. Note that the copy shares the input buffer with the original.

save-input is a virtual method of the input-stream class.

**scr ( -- address -> unsigned )**

block.sf

address -> unsigned is the address of a cell containing the number of the block most recently listed.

**search ( caddress -> character unsigned 1st 3rd -- 1st 3rd flag )**

Search the string specified by caddress -> character unsigned for the sub-string specified by 1st 3rd. If flag is true, a match was found at 1st with 3rd characters remaining. If flag is false there was no match and 1st is caddress -> character and 3rd is unsigned.

**search ( caddress -> character unsigned single search-criterion vocabulary -- definition flag )**

Search vocabulary for the definition whose name is given by the character string caddress -> character unsigned. If the definition is found, return it as definition and true as flag. If the definition is not found, return null as definition and false as flag.

search-criterion is the token of an additional match criterion. If the additional match criterion requires a parameter, the parameter is passed by single.

If unsigned is zero, the definition's name is not considered. Only the match criterion specified by search-criterion and single matters.

search is a virtual method of the vocabulary class.

**search-all ( caddress -> character unsigned single search-criterion -- definition flag )**

Search the first context vocabulary for the definition whose name is given by the character string caddress -> character unsigned. search-criterion is the token of an additional

match criterion. If the additional match criterion requires a parameter, the parameter is passed by `single`.

If `unsigned` is zero, the definition's name is not considered. Only the match criterion specified by `search-criterion` and `single` matters.

If the definition is found, return it as `definition` and `true` as `flag`. If the definition is not found, try to search all other context vocabularies and then all hidden vocabularies, until either a definition is found or until the last vocabulary in the hidden vocabulary list has been searched. If the definition is not found in any vocabulary, return zero as `definition` and `false` as `flag`.

**`search-context ( caddress -> character unsigned single search-criterion -- definition flag )`**

Search the first context vocabulary for the definition whose name is given by the character string `caddress -> character unsigned`. `search-criterion` is the token of an additional match criterion. If the additional match criterion requires a parameter, the parameter is passed by `single`.

If `unsigned` is zero, the definition's name is not considered. Only the match criterion specified by `search-criterion` and `single` matters.

If the definition is found, return it as `definition` and `true` as `flag`. If the definition is not found, try to search the other context vocabularies, until either a definition is found or until the last vocabulary in the context vocabulary list has been searched. If the definition is not found in any of the context vocabularies, return zero as `definition` and `false` as `flag`.

**`search-criterion ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `search-criterion`.

**`search-list ( caddress -> character unsigned single search-criterion vocabulary -- definition flag )`**

Search vocabulary for the definition whose name is given by the character string `caddress -> character unsigned`. `search-criterion` is the token of an additional match criterion. If the additional match criterion requires a parameter, the parameter is passed by `single`.

If `unsigned` is zero, the definition's name is not considered. Only the match criterion specified by `search-criterion` and `single` matters.

If the definition is found, return it as `definition` and `true` as `flag`. If the definition is not found, try to search the other vocabularies linked to `vocabulary`, until either a definition is found or until the last vocabulary in the vocabulary list has been searched. If the definition is not found in any of the vocabularies, return zero as `definition` and `false` as `flag`.

**`search-local ( caddress -> character unsigned -- local-definition flag )`**

Search the locals vocabulary for the definition whose name is given by the character string `caddress -> character unsigned`. If `state` is `true` and the definition is found, return it as `local-definition` and `true` as `flag`. If `state` is `false` or the definition is not found, return zero as `definition` and `false` as `flag`.

**search-order ( stack-diagram -- 1st )**

order.sf

When used in a stack diagram, specifies an input or output parameter with data type `search-order`.

**see ( "<spaces>name" -- )**

Skip leading space delimiters. Parse *name* delimited by a space. Search the context vocabularies for *name*. Send a human-readable representation of the definition of *name*, including its stack diagram, to the default output stream.

**see ( definition -- )**

Send a human-readable representation *definition*, including its stack diagram, to the default output stream.

`see` is a virtual method of the `definition` class.

**set-current ( vocabulary -- )**

order.sf

Make *vocabulary* the current compilation vocabulary.

**set-order ( search-order -- )**

order.sf

Restore the context vocabulary list and the hidden vocabulary list from `search-order`. `search-order` is an identifier created by `get-order`.

**set-precision ( unsigned -- )**

float.sf

Set the number of significant digits currently used by `.`, `e.`, and `s.` to the minimum of `max-precision` and `unsigned`.

**set? ( single logical -- flag )**

`flag` is true if and only if the bit-by-bit logical and of `single` and `logical` is not equal to zero.

**sf, ( complex -- )**

Reserve space for two single-precision floating-point numbers in the default memory space and store `complex` as a complex single-precision floating-point number in it. If the first unused address of the default memory space is cell aligned prior to execution of `sf,`, it will remain cell aligned when `sf,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not cell aligned prior to execution of `sf,`. An exception is thrown if the default memory space overflows.

**sf, ( complex memory-space -- )**

Reserve space for two single-precision floating-point numbers in `memory-space` and store `complex` as a complex single-precision floating-point number in it. If the first unused address of

`memory-space` is cell aligned prior to execution of `sf,`, it will remain cell aligned when `sf,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not cell aligned prior to execution of `sf,`. An exception is thrown if `memory-space` overflows.

**`sf, ( float -- )`**

Reserve space for a single-precision floating-point number in the default memory space and store `float` as a single-precision floating-point number in it. If the first unused address of the default memory space is cell aligned prior to execution of `sf,`, it will remain cell aligned when `sf,` finishes execution. An ambiguous condition exists if the first unused address of the default memory space is not cell aligned prior to execution of `sf,`. An exception is thrown if the default memory space overflows.

**`sf, ( float memory-space -- )`**

Reserve space for a single-precision floating-point number in `memory-space` and store `float` as a single-precision floating-point number in it. If the first unused address of `memory-space` is cell aligned prior to execution of `sf,`, it will remain cell aligned when `sf,` finishes execution. An ambiguous condition exists if the first unused address of `memory-space` is not cell aligned prior to execution of `sf,`. An exception is thrown if `memory-space` overflows.

**`sfaddress ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `sfaddress`.

**`sfalign ( -- )`**

float.sf

If the first unused address of the default memory space is not single-precision floating-point aligned, reserve the required number of address units to make it single-precision floating-point aligned.

**`sfalign ( memory-space -- )`**

float.sf

If the first unused address of `memory-space` is not single-precision floating-point aligned, reserve the required number of address units to make it single-precision floating-point aligned.

**`sfaligned ( address -- 1st )`**

float.sf

`1st` is the lowest single-precision floating-point aligned address greater than or equal to `address`.

**`sfallocate ( unsigned -- sfaddress )`**

Allocate unsigned address units of contiguous dynamic memory space. The initial content of the allocated memory space is undefined. If the allocation succeeds, `sfaddress` is the aligned starting address of the allocated memory space. An exception is thrown if the operation fails.

**`sfhere ( -- sfaddress )`**

`sfaddress` is the first unused address of the default memory space.

**`sfhere ( memory-space -- sfaddress )`**

`sfaddress` is the first unused address of `memory-space`.

**`sfloats ( integer -- 1st )`**

float.sf

`1st` is the size in address units of `integer` single-precision floating-point numbers.

**`sfmember ( object-size complex "<spaces>name" -- 1st )`**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to single-precision floating-point numbers, plus the size in bits of two single-precision floating-point numbers.

*name* is referred to as a class member. `sfmember` reserves space for two single-precision floating-point numbers for a class member of the same data type as `complex` in the class that is currently being defined.

Execution: ( `x -- sfaddress -> y` )

`sfaddress -> y` is the address of the class member of the object `x`, that was reserved at the time *name* was created. `y` is the actual data type that was provided to `sfmember` as `complex`.

**`sfmember ( object-size float "<spaces>name" -- 1st )`**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to single-precision floating-point numbers, plus the size in bits of a single-precision floating-point number.

*name* is referred to as a class member. `sfmember` reserves space for one single-precision floating-point number for a class member of the same data type as `float` in the class that is currently being defined.

Execution: ( `x -- sfaddress -> y` )

`sfaddress -> y` is the address of the class member of the object `x`, that was reserved at the time *name* was created. `y` is the actual data type that was provided to `sfmember` as `float`.

**`sfmembers ( object-size complex unsigned "<spaces>name" -- 1st )`**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to single-precision floating-point numbers, plus `unsigned` times the size in bits of a complex single-precision floating-point number.

*name* is referred to as a class member. `sfmembers` reserves unsigned complex single-precision floating-point numbers for an array of unsigned class members of the same data type as `complex` in the class that is currently being defined.

Execution: ( `x -- sfaddress -> y` )

`sfaddress -> y` is the address of an array of unsigned class members of the object `x`, that were reserved at the time `name` was created. `y` is the actual data type that was provided to `sfmembers` as `complex`.

**`sfmembers ( object-size float unsigned "<spaces>name" -- 1st )`** float.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a new definition for `name` with the execution semantics defined below, and make it the latest definition. `1st` is equal to `object-size` aligned to single-precision floating-point numbers, plus `unsigned` times the size in bits of a single-precision floating-point number.

`name` is referred to as a class member. `sfmembers` reserves unsigned single-precision floating-point numbers for an array of unsigned class members of the same data type as `float` in the class that is currently being defined.

Execution: ( `x -- sfaddress -> y` )

`sfaddress -> y` is the address of an array of unsigned class members of the object `x`, that were reserved at the time `name` was created. `y` is the actual data type that was provided to `sfmembers` as `float`.

**`sfvariable ( complex "<spaces>name" -- )`** complex.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a definition for `name` with the execution semantics defined below. Reserve space for two single-precision floating-point numbers at a single-precision floating-point aligned address in the data-space memory space and store `complex` at the address.

`name` is referred to as a variable.

Execution: ( `-- sfaddress -> x` )

`sfaddress -> x` is the address of the complex single-precision floating-point number. `x` has the same data type as was supplied to `variable`.

**`sfvariable ( float "<spaces>name" -- )`** float.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a definition for `name` with the execution semantics defined below. Reserve space for a single-precision floating-point number at a single-precision floating-point aligned address in the data-space memory space and store `float` at the address.

`name` is referred to as a variable.

Execution: ( `-- sfaddress -> x` )

`sfaddress -> x` is the address of the single-precision floating-point number. `x` has the same data type as was supplied to `variable`.

**`sfvariables ( complex unsigned "<spaces>name" -- )`** complex.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a definition for `name` with the execution semantics defined below. Reserve space for unsigned complex single-precision floating-point numbers at a single-precision floating-point aligned address in the data-space memory space and store `complex` in each of them.



*name* is referred to as a variable.

Execution: ( -- sfaddress -> x )

*sfaddress* -> *x* is the address of the first complex single-precision floating-point number. *x* has the same data type as was supplied to *variable*.

**sfvariables ( float unsigned "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for unsigned single-precision floating-point numbers at a single-precision floating-point aligned address in the data-space memory space and store *float* in each of them.

*name* is referred to as a variable.

Execution: ( -- sfaddress -> x )

*sfaddress* -> *x* is the address of the first single-precision floating-point number. *x* has the same data type as was supplied to *variable*.

**shrink ( integer -- )**

Reduce the size of the default memory space by *integer* address units. An exception is thrown if *integer* is negative or greater than the number of unused address units in the default memory space.

The released memory is not automatically returned to the system.

**shrink ( integer memory-space -- )**

Reduce the size of *memory-space* by *integer* address units. An exception is thrown if *integer* is negative or greater than the number of unused address units in *memory-space*.

The released memory is not automatically returned to the system.

**sign ( signed-double -- )**

strongforth.sf

If *signed-double* is true, add a minus sign to the beginning of the pictured numeric output string.

**signed ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *signed*.

**signed-double ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type *signed-double*.

**sin ( complex -- 1st )**

complex.sf

*1st* is the complex sine of the radian angle *complex*.

**sin ( float -- 1st )**

1st is the sine of the radian angle float.

**sincos ( float -- 1st 1st )**

The first 1st is the sine of the radian angle float. The second 1st is the cosine of the radian angle float.

**single ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type single. single has no parent data type.

**single-definition ( caddress -> character unsigned single-definition -- 4 th )**

Initialize single-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when single-definition is deleted. Assign single-definition a name given by the character string caddress -> character unsigned and return it as 4 th.

single-definition is a constructor of the single-definition class.

**single-definition ( single-definition -- 1st )**

Initialize single-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when single-definition is deleted. 1st is single-definition.

single-definition is a constructor of the single-definition class.

**single-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type single-definition.

**single? ( integer-double -- flag )**

flag is true if and only if integer-double can be represented as an unsigned single-cell integer, i. e., its value is less than max-unsigned.

**single? ( signed-double -- flag )**

flag is true if and only if signed-double can be represented as a signed single-cell integer, i. e., its value is between max-signed negate 1- and max-signed.

**sinh ( complex -- 1st )**

complex.sf

1st is the complex hyperbolic sine of complex.

**sinh ( float -- 1st )**

float.sf

1st is the hyperbolic sine of float.

**size ( address -- unsigned )**

strongforth.sf

unsigned is the size in address units of the allocated memory block at address. An ambiguous condition exists if address does not indicate a memory space that was previously obtained by allocate, callocate, dfallocate, sfallocate or resize.

**size ( data-type -- unsigned )**

unsigned is the size in address units of items of data type data-type, or zero if the size cannot be determined.

**size ( file -- unsigned-double )**

strongforth.sf

unsigned-double is the size in characters of the file identified by file. size does not affect the value returned by position. An exception is thrown if the operation fails.

**size ( object -- unsigned )**

unsigned is the memory size in address units of object.

**size ( vtable -- unsigned )**

unsigned is the memory size in address units of objects whose virtual method table is vtable.

**sliteral ( caddress -> character unsigned -- ) compile-only**

strongforth.sf

Compilation: Allot unsigned characters in the data-space memory space. Align the data-space memory space. Copy the character string specified by caddress -> character unsigned to the allotted memory area. Append the runtime semantics given below to the current definition.

Runtime: ( -- caddress -> character unsigned )

Place the address of the copied character string caddress -> character unsigned onto the stack. Both items have the same data types as were supplied at compilation time.

**sm/rem ( signed-double signed -- 2nd signed )**

Divide signed-double by signed, giving the symmetric quotient signed and the remainder 2nd. An exception is thrown if signed is zero. An ambiguous condition exists if the quotient lies outside the range of a signed single-precision number.

**smember ( object-size data-type "<spaces>name" -- 1st )**

struct.sf

Skip leading space delimiters. Parse name delimited by a space. Create a new definition for name with the execution semantics defined below, and make it the latest definition. 1st is equal to object-size, plus the size in bits of the structure associated with data-type. An exception is thrown if data-type is not directly or indirectly derived from structure.

*name* is referred to as a class member. `smember` reserves space for an embedded structure with data type `data-type` in the structure that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of the member of the structure *x*, that was reserved at the time *name* was created. *y* is an item with data type `data-type`.

**smembers ( object-size data-type unsigned "<spaces>name" -- 1st )** struct.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below, and make it the latest definition. 1st is equal to object-size, plus unsigned times the size in bits of the structure associated with `data-type`. An exception is thrown if `data-type` is not directly or indirectly derived from structure.

*name* is referred to as a class member. `smember` reserves space for an array of unsigned embedded structures with data type `data-type` in the structure that is currently being defined.

Execution: ( *x* -- address -> *y* )

address -> *y* is the address of an array of unsigned members of the structure *x*, that were reserved at the time *name* was created. *y* is an item with data type `data-type`.

**source ( -- caddress -> character unsigned )**

caddress -> character is the address of the input buffer of the default input stream. unsigned is the number of characters in the input buffer of the default input stream.

**source ( input-stream -- caddress -> character unsigned )**

caddress -> character is the address of the input buffer of input-stream. unsigned is the number of characters in the input buffer of input-stream.

**sp! ( address -- )**

Make address the current value of the stack pointer.

**sp@ ( -- address )**

address is the value of the stack pointer before address was pushed onto the stack.

**space ( -- )**

Send a space character to the default output stream.

**spaces ( integer -- )**

strongforth.sf

If integer is greater than zero, send integer spaces to the default output stream.

Note: integer is assumed to be a signed number.

**split ( complex -- float float )**

Splits a complex floating-point number `complex` into two floating-point numbers `float`. The first one is the real part, the second one is the imaginary part.

**split ( double -- single single )**

Splits a double-cell item `double` into two single-cell items `single`. The most significant part is on top of the stack.

**split ( float -- signed 1st )**

`signed` is the exponent of `float`. `1st` is the mantissa of `float` with the exponent being zero.

**sqrt ( complex -- 1st )**

`complex.sf`

`1st` is the complex square root of `complex`.

**sqrt ( float -- 1st )**

`1st` is the square root of `float`. An ambiguous condition exists if `float` is less than zero.

**stack-cells ( -- unsigned )**

`strongforth.sf`

`unsigned` is the size of the data stack in cells. Since StrongForth has no data stack, `stack-cells` is the number of cells than can be stored in general-purpose processor registers.

**stack-diagram ( flag stack-diagram -- 2nd )**

Initialize `stack-diagram` by erasing all members. Save `flag` as the compilation state. The compilation state is restored once `stack-diagram` is deleted.

`stack-diagram` is the constructor of the `stack-diagram` class.

**stack-diagram ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `stack-diagram`.

**stack-space ( -- memory-space )**

`memory-space` is the system's stack space. The combined data and return stack as well as user variables are stored in the stack space.

**stack: ( stack-diagram -- 1st )**

When used in a stack diagram, specifies the succeeding input or output parameter shall be assigned to the stack. An exception is thrown if the input or output parameter is not a single-cell or double-cell parameter.

**state ( -- caddress -> flag )**

`caddress -> flag` is the address of the compilation state. `state` is `true` when in compilation state, and `false` when in interpretation state.

**`static-compiler-workspace ( -- compiler-workspace )`**

`compiler-workspace` is a static instance of the `compiler-workspace` class that is used by `compile`, `literal`, `?congruent` and `match-criterion`.

**`status ( caddress -> character unsigned -- address )`**

strongforth.sf

Allocate 9 consecutive memory cells and return the address of the first cell as `address`. Store information about the file with the path given by the string `caddress -> character unsigned` in the allocated memory cells. An exception is thrown if the operation fails.

**`stderr ( -- file )`**

`file` is the device handle of the system's standard error output. Since `stderr` is a value, it can be redirected with `to`.

**`stdin ( -- file )`**

`file` is the device handle of the system's standard input. Since `stdin` is a value, it can be redirected with `to`.

**`stdout ( -- file )`**

`file` is the device handle of the system's standard output. Since `stdout` is a value, it can be redirected with `to`.

**`store ( complex value-definition -- )`**

Store `complex` in the value whose memory location is represented by `value-definition`. Note that `store` is not type-save. An ambiguous condition exists if the data type of the value is not a complex floating-point number.

**`store ( double value-definition -- )`**

Store `double` in the value whose memory location is represented by `value-definition`. Note that `store` is not type-save. An ambiguous condition exists if the data type of the value is not a double cell.

**`store ( float value-definition -- )`**

Store `float` in the value whose memory location is represented by `value-definition`. Note that `store` is not type-save. An ambiguous condition exists if the data type of the value is not a floating-point number.

**`store ( single value-definition -- )`**

Store single in the value whose memory location is represented by value-definition.  
Note that store is not type-save. An ambiguous condition exists if the data type of the value is not a single cell.

**string ( string-output-stream -- caddress -> character unsigned )** strongforth.sf

caddress -> character is the address of the output buffer of string-output-stream.  
unsigned is the number of characters that have been sent to string-output-stream.

**string-input-stream ( caddress -> character unsigned string-input-stream -- 4 th )**

Initialize string-input-stream by erasing all members. Make caddress -> character unsigned the input buffer. 4 th is string-input-stream. The input buffer will not be deallocated when string-input-stream is deleted.

string-input-stream is a constructor of the string-input-stream class.

**string-input-stream ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type string-input-stream.

**string-input-stream ( string-input-stream 1st - 1st )**

Copy all members of string-input-stream to 1st. 1st is 1st. 1st shares the same input buffer as string-input-stream. The input buffer will not be deallocated when 1st is deleted.

string-input-stream is a constructor of the string-input-stream class.

**string-output-stream ( caddress -> character unsigned string-output-stream -- 4 th )** strongforth.sf

Initialize string-output-stream by erasing all members. Make caddress -> character unsigned the output buffer. 4 th is string-output-stream. The output buffer will not be deallocated when string-output-stream is deleted.

string-output-stream is the constructor of the string-output-stream class.

**string-output-stream ( stack-diagram -- 1st )** strongforth.sf

When used in a stack diagram, specifies an input or output parameter with data type string-output-stream.

**structure ( stack-diagram -- 1st )** struct.sf

When used in a stack diagram, specifies an input or output parameter with data type structure.

**structure-attributes ( data-type unsigned structure-attributes -- 3rd )** struct.sf

Initialize `structure-attributes` by erasing all members. Store the attributes of `data-type` as the parent of the data type associated with `structure-attributes`. Store `unsigned` as the size of the data type associated with `structure-attributes`.  
`structure-attributes` is the constructor of the `structure-attributes` class.

**`structure-attributes ( stack-diagram -- 1st )`** struct.sf

When used in a stack diagram, specifies an input or output parameter with data type `structure-attributes`.

**`structure-size ( data-type -- unsigned )`** struct.sf

`unsigned` is the memory size in address units of a structure with data type `data-type`. An exception is thrown if `data-type` is not directly or indirectly derived from `structure`.

**`structure? ( data-type -- flag )`** struct.sf

`flag` is true if and only if `data-type` is equal to `structure`, or if `data-type` is directly or indirectly derived from `structure`.

**`substitute ( caddress -> character unsigned -- unsigned )`** strect.sf

Perform substitution on the string `caddress -> character unsigned` sending the result to the default output stream. `unsigned` is the number of substitutions made, or zero if an error occurred. Substitution occurs left to right starting at `caddress -> character` in one pass and is non-recursive.

When text of a potential substitution name, surrounded by `delimiters`, is encountered by `substitute`, the following occurs:

If the text is null, a single `delimiter` character is sent to the default output stream, i.e., two `delimiter` characters are replaced by one. The current number of substitutions is not changed.

If the text is a valid substitution name acceptable to `replaces`, the leading and trailing `delimiter` characters and the enclosed substitution name are replaced by the substitution text. The current number of substitutions is incremented.

If the text is not a valid substitution name, the name with leading and trailing `delimiters` is sent unchanged to the default output stream. The current number of substitutions is not changed. Parsing of the input string resumes after the trailing `delimiter`.

If after processing any pairs of `delimiters`, the residue of the input string contains a single `delimiter`, the residue is sent unchanged to the default output stream.

**`substitute ( caddress -> character unsigned caddress -> character unsigned -- 4 th 6 th unsigned )`** strect.sf

Perform substitution on the first string `caddress -> character unsigned` placing the result at the second string `caddress -> character unsigned`. `4 th` is the second `caddress -> character`, `6th` is the length of the resulting string. An exception is thrown if the resulting string is longer than the second `unsigned`. The return value `unsigned` is the number of substitutions made, or zero if an error occurred, leaving `4 th` and `6 th` undefined.



Substitution occurs left to right starting at `caddress` -> `character` in one pass and is non-recursive.

When text of a potential substitution name, surrounded by `delimiters`, is encountered by `substitute`, the following occurs:

If the text is null, a single `delimiter` character is passed to the output, i.e., two `delimiter` characters are replaced by one. The current number of substitutions is not changed.

If the text is a valid substitution name acceptable to `replaces`, the leading and trailing `delimiter` characters and the enclosed substitution name are replaced by the substitution text. The current number of substitutions is incremented.

If the text is not a valid substitution name, the name with leading and trailing `delimiters` is passed unchanged to the output. The current number of substitutions is not changed. Parsing of the input string resumes after the trailing `delimiter`.

If after processing any pairs of `delimiters`, the residue of the input string contains a single `delimiter`, the residue is passed unchanged to the output.

**`substitute ( file file -- unsigned )`**

strex.sf

Perform substitution on the contents of the first `file` writing the result to the second `file`. The return value `unsigned` is the number of substitutions made, or zero if an error occurred. Substitution occurs from the start to the end of the first `file` in one pass and is non-recursive.

When text of a potential substitution name, surrounded by `delimiters`, is encountered by `substitute`, the following occurs:

If the text is null, a single `delimiter` character is written to the output file, i.e., two `delimiter` characters are replaced by one. The current number of substitutions is not changed.

If the text is a valid substitution name acceptable to `replaces`, the leading and trailing `delimiter` characters and the enclosed substitution name are replaced by the substitution text. The current number of substitutions is incremented.

If the text is not a valid substitution name, the name with leading and trailing `delimiters` is written unchanged to the output file. The current number of substitutions is not changed. Parsing of the input string resumes after the trailing `delimiter`.

If after processing any pairs of `delimiters`, the residue of the current line contains a single `delimiter`, the residue is written unchanged to the output file.

**`swap ( complex complex -- 2nd 1st )`**

Exchange the two items on top of the stack.

**`swap ( complex double -- 2nd 1st )`**

Exchange the two items on top of the stack.

**`swap ( complex float -- 2nd 1st )`**

Exchange the two items on top of the stack.

**`swap ( complex single -- 2nd 1st )`**

Exchange the two items on top of the stack.

**swap ( double complex -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( double double -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( double float -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( double single -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( float complex -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( float double -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( float float -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( float single -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( single complex -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( single double -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( single float -- 2nd 1st )**

Exchange the two items on top of the stack.

**swap ( single single -- 2nd 1st )**

Exchange the two items on top of the stack.

**tan ( complex -- 1st )**

complex.sf

1st is the complex tangent of the radian angle complex. An ambiguous condition exists if the complex cosine of complex is zero.

**tan ( float -- 1st )**

float.sf

1st is the tangent of the radian angle float. An ambiguous condition exists if the cosine of float is zero.

**tanh ( complex -- 1st )**

complex.sf

1st is the complex hyperbolic tangent of complex.

**tanh ( float -- 1st )**

float.sf

1st is the hyperbolic tangent of float.

**terminal-input-stream ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type terminal-input-stream.

**terminal-input-stream ( terminal-input-stream 1st - 1st )**

Copy all members of terminal-input-stream to 1st. 1st is 1st. 1st shares the same input buffer as terminal-input-stream. The input buffer will not be deallocated when 1st is deleted. An ambiguous condition exists if 1st is used after terminal-input-stream has been deleted.

terminal-input-stream is a constructor of the terminal-input-stream class.

**terminal-input-stream ( unsigned terminal-input-stream -- 2nd )**

Initialize terminal-input-stream by erasing all members. Allocate unsigned characters from dynamic memory as input buffer. 2nd is terminal-input-stream.

terminal-input-stream is a constructor of the terminal-input-stream class.

**terminal-output-stream ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type terminal-output-stream.

**terminal-output-stream ( terminal-output-stream -- 1st )**

No operation, because class terminal-output-stream has no members. 1st is terminal-output-stream.

terminal-output-stream is the constructor of the terminal-output-stream class.

**th ( stack-diagram unsigned -- 1st )**

Append a reference to the basic data type with the position `unsigned` of the input parameter list, starting with 1, as an input or output parameter to `stack-diagram`.

`th` is used in a stack diagram to specify input or output parameters which should have exactly the same data type as the actual data type at the position `unsigned` in the input parameter list of the same definition. Since the index refers to the basic data types in the input parameter list, it is possible to build a reference to the tail of a compound data type representing an input parameter.

An exception is thrown if `unsigned` is zero, if `unsigned` is greater than the length of the input parameter list, if the referenced data type is itself a reference, or if the internal storage for input and output parameters of `stack-diagram` is exceeded.

**then ( origin -- ) compile-only**

Compilation: Append the runtime semantics given below to the current definition. Resolve the forward reference `origin` using the current location. An exception is thrown if the contents of the compiler data type heap do not match the copy that was saved when `origin` was created.

Runtime: Continue execution.

**this-attributes ( -- class-attributes )**

strongforth.sf

`class-attributes` is the attributes of the data type of the class that is currently being defined.

`this-attributes` is a value.

**this-vtable ( -- vtable )**

strongforth.sf

`vtable` is the virtual method table of the class that is currently being defined.

**throw ( signed -- )**

If `signed` is not equal to zero, perform the following semantics:

If a current exception frame exists, throw an exception with `signed` using this exception frame. If no exception frame exists, execute the semantics of the deferred definition `error`.

**throw ( signed exception-frame -- )**

Save `signed` as error code in `exception-frame`. Restore the default input stream and the input source specification to the state they had when `exception-frame` was initialized.

Continue execution at the location at which `exception-frame` was created, immediately before obtaining the error code.

**thru ( unsigned 1st -- )**

block.sf

load the blocks number `unsigned` through `1st` in sequence. Other stack effects are due to the words loaded. An exception is thrown if `unsigned` is greater than `1st`, or if either `unsigned` or `1st` are no valid block numbers.

**ticks ( -- unsigned )**

strongforth.sf

*unsigned* is the number of milliseconds elapsed since initialization of the *MSVCRT* library. An exception is thrown if the operation fails.

**time&date ( -- unsigned unsigned unsigned unsigned unsigned  
unsigned )**

Return the current time and date represented by six *unsigned* numbers in the given order: second (0 to 59), minute (0 to 59), hour (0 to 23), day (1 to 31), month (1 to 12), and year (e.g., 2007). The year is on top of the stack.

**to ( "<spaces>name" -- ) compile-only**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Append the runtime semantics given below to the current definition. An exception is thrown if *name* is not either a value definition or a locals definition, or if the compound data type on top of the compiler data type heap does not match the tail of the compound data type of the output parameter of *name*.

Runtime: ( *x* -- )

Store *x* in the value or local identified by *name*.

**to ( complex "<spaces>name" -- ) execute-only**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Store *complex* in the value identified by *name*. An exception is thrown if *name* is not a value definition or if *complex* does not match the tail of the compound data type of the output parameter of *name*.

**to ( double "<spaces>name" -- ) execute-only**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Store *double* in the value identified by *name*. An exception is thrown if *name* is not a value definition or if *double* does not match the tail of the compound data type of the output parameter of *name*.

**to ( float "<spaces>name" -- ) execute-only**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Store *float* in the value identified by *name*. An exception is thrown if *name* is not a value definition or if *float* does not match the tail of the compound data type of the output parameter of *name*.

**to ( single "<spaces>name" -- ) execute-only**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Store *single* in the value identified by *name*. An exception is thrown if *name* is not a value definition or if *single* does not match the tail of the compound data type of the output parameter of *name*.

**to-local-definition ( local-definition to-local-definition -- 2nd  
)**

Initialize `to-local-definition` by copying all members of `local-definition`. Assign the null string as the name of `to-local-definition`. The output parameter of `local-definition` becomes the input parameter of `to-local-definition`.

Note: `to-local-definition` is used as a temporary definition that allows changing the value of `local-definition` in a type-safe way.

`to-local-definition` is the constructor of the `to-local-definition` class.

**`to-local-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `to-local-definition`.

**`to-value-definition ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `to-value-definition`.

**`to-value-definition ( value-definition to-value-definition -- 2nd )`**

Initialize `to-value-definition` by copying all members of `value-definition`. Assign the null string as the name of `to-value-definition`. The output parameter of `value-definition` becomes the input parameter of `to-value-definition`.

Note: `to-value-definition` is used as a temporary definition that allows changing the value of `value-definition` in a type-safe way.

`to-value-definition` is the constructor of the `to-value-definition` class.

**`token ( definition -- token )`**

If `definition` is a code definition, `token` is the execution token of `definition`. The execution token is actually the address of the first machine code instruction of the code definition. Otherwise, `token` is null.

`token` is a virtual method of the `definition` class.

**`token ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `token`.

**`token-criterion ( -- search-criterion )`**

`strongforth.sf`

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: `( definition single -- flag )`

`flag` is true if and only if the value of `single` is the execution token of `definition`.

Note: Provide `search-criterion` to `search` in order to find a code definition or a colon definition with a specific execution token.

**true ( -- flag )**

flag is a true flag, a single-cell item with all bits set to 1.

**trunc ( float -- 1st )**

Round float to an integral value using the round toward zero rule, giving 1st.

**tuck ( complex complex -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( complex double -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( complex float -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( complex single -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( double complex -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( double double -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( double float -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( double single -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( float complex -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( float double -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( float float -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( float single -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( single complex -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( single double -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( single float -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**tuck ( single single -- 2nd 1st 2nd )**

Copy the first (top) stack item below the second stack item.

**type ( caddress -> character unsigned -- )**

Send unsigned characters starting at `ccaddress -> character` to the default output stream.

**type ( caddress -> character unsigned output-stream -- )**

Send unsigned characters starting at `ccaddress -> character` to `output-stream`.

`type` is a virtual method of the `output-stream` class.

**unassigned ( object -- )**

Throws an exception, indicating that a virtual method of `object` has not yet been assigned.

**unchanged ( stack-diagram -- 1st )**

During specification of the stack diagram `stack-diagram` of a code definition or a colon definition, mark a list of registers as being preserved by the definition. The preservation is guaranteed by compiling appropriate `push`, and `pop`, assembler instructions at the beginning and at the end of the definition, respectively. `1st` is equal to `stack-diagram`. Registers have to be considered only when programming in assembler.

**unescape ( caddress -> character unsigned -- )**

`strex.sf`

Perform substitution on the string `ccaddress -> character unsigned` sending the result to the default output stream. Substitution occurs left to right starting at `ccaddress -> character` in one pass and is non-recursive.



Replace each `delimiter` character in the input string with two `delimiter` characters.

**`unescape ( caddress -> character unsigned caddress -> character unsigned - 4 th 6 th )`**

`strex.sf`

Perform substitution on the first string `caddress -> character unsigned` placing the result at the second string `caddress -> character unsigned`. 4 `th` is the second `caddress -> character`, 6`th` is the length of the resulting string. An exception is thrown if the resulting string is longer than the second `unsigned`. Substitution occurs left to right starting at `caddress -> character` in one pass and is non-recursive.

Replace each `delimiter` character in the input string with two `delimiter` characters.

**`unescape ( file file -- )`**

`strex.sf`

Perform substitution on the contents of the first `file` writing the result to the second `file`. Substitution occurs from the start to the end of the first `file` in one pass and is non-recursive.

Replace each `delimiter` character in the input file with two `delimiter` characters.

**`union ( object-size -- 1st 1st 1st )`**

`strongforth.sf`

Starts a union of members within a class definition. All parameters `1st` are equal to `object-size`. The first `1st` is the starting bit position of the union, the second `1st` is the end bit position of the largest block so far, and the third `1st` is the current bit position of the current block.

**`unpack ( single -- single single single single )`**

Unpacks the four bytes of `single` with zero extension into four items with data type `single`.

The first output `single` is the most significant byte of input `single`. The second output `single` is the second most significant byte of input `single`. The third output `single` is the second least significant byte of input `single`. The fourth output `single` is the least significant byte of input `single`.

**`unpack ( signed -- signed signed signed signed )`**

Unpacks the four bytes of `signed` with sign extension into four items with data type `signed`.

The first output `signed` is the most significant byte of input `signed`. The second output `signed` is the second most significant byte of input `signed`. The third output `signed` is the second least significant byte of input `signed`. The fourth output `signed` is the least significant byte of input `signed`.

**`unsigned ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `unsigned`.

**`unsigned-double ( stack-diagram -- 1st )`**

When used in a stack diagram, specifies an input or output parameter with data type `unsigned-double`.

**until ( destination -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition, resolving the backward reference `destination`. An exception is thrown if the contents of the compiler data type heap, after consuming `single`, do not exactly match the copy that was saved when `destination` was created.

Runtime: ( `single` -- )

If `single` is zero, continue execution at the location specified by `destination`. Otherwise, continue execution. `single` is not taken into consideration when comparing the contents of the compiler data type heap with the copy that was saved when `destination` was created.

**unused ( -- unsigned )**

`unsigned` is the number of address units remaining in the default memory space, starting at here.

**unused ( memory-space -- unsigned )**

`unsigned` is the number of address units remaining in `memory-space`, starting at here.

**upcase ( caddress -> character unsigned -- )**

ascii.sf

Replace each lowercase letter within the character string `caddress -> character` into the equivalent uppercase letter. All other characters remain unchanged. `upcase` works for German umlauts.

**upcase ( character -- 1st )**

ascii.sf

If `character` is a lowercase letter, `1st` is the equivalent uppercase letter. Otherwise, `1st` is equal to `character`. `upcase` works for German umlauts.

**update ( -- )**

block.sf

Mark the block buffer as modified by storing `true` in `updated`.

Note: `update` does not immediately cause a transfer to the block file.

**updated ( -- address -> flag )**

block.sf

`address -> flag` is the address of a cell containing a flag. The flag is `true` if and only if the block stored in the block buffer has been modified.

**user-input-device ( -- terminal-input-stream )**

`terminal-input-stream` is the predefined terminal input device.

**user-output-device ( -- terminal-output-stream )**

`terminal-output-stream` is the predefined terminal output device.

**value ( complex "<spaces>name" -- )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for two floating-point numbers at a cell aligned address in the data-space memory space and store `complex` at the address.

*name* is referred to as a value.

Execution: ( -- *x* )

*x* is the content of the reserved space. The value of *x* is that given when *name* was created, until the phrase `to name` is executed, causing a new value of *x* to be associated with *name*. *x* has the same data type as was supplied to `value`.

**value ( double "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve two cells at a cell aligned address in the data-space memory space and store `double` at the address.

*name* is referred to as a value.

Execution: ( -- *x* )

*x* is the content of the reserved pair of cells. The value of *x* is that given when *name* was created, until the phrase `to name` is executed, causing a new value of *x* to be associated with *name*. *x* has the same data type as was supplied to `value`.

**value ( float "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for a floating-point number at a float aligned address in the data-space memory space and store `float` at the address.

*name* is referred to as a value.

Execution: ( -- *x* )

*x* is the content of the reserved space. The value of *x* is that given when *name* was created, until the phrase `to name` is executed, causing a new value of *x* to be associated with *name*. *x* has the same data type as was supplied to `value`.

**value ( single "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve one cell at a cell aligned address in the data-space memory space and store `single` at the address.

*name* is referred to as a value.

Execution: ( -- *x* )

*x* is the content of the reserved cell. The value of *x* is that given when *name* was created, until the phrase `to name` is executed, causing a new value of *x* to be associated with *name*. *x* has the same data type as was supplied to `value`.

**value-definition ( caddress -> character unsigned value-definition -- 4 th )**

Initialize value-definition by erasing all members. Establish a link to the previous definition in the current vocabulary and update latest. Links will be removed when value-definition is deleted. Assign value-definition a name given by the character string caddress -> character unsigned and return it as 4 th.

value-definition is the constructor of the value-definition class.

**value-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type value-definition.

**variable ( complex "<spaces>name" -- )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for two floating-point numbers at a cell aligned address in the data-space memory space and store complex at the address.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the complex floating-point number. x has the same data type as was supplied to variable.

**variable ( double "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve two cells at a cell aligned address in the data-space memory space and store double at the address.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the reserved pair of cells. x has the same data type as was supplied to variable.

**variable ( float "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for a floating-point number at a float aligned address in the data-space memory space and store float at the address.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the floating-point number. x has the same data type as was supplied to variable.

**variable ( single "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve one cell at a cell aligned address in the data-space memory space and store *single* at the address.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the reserved cell. x has the same data type as was supplied to variable.

**variables ( complex unsigned "<spaces>name" -- )**

complex.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for unsigned complex floating-point numbers at a cell aligned address in the data-space memory space and store *complex* in each of them.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the first floating-point number. x has the same data type as was supplied to variables.

**variables ( double unsigned "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve two times unsigned cells at an aligned address in the data-space memory space and store *double* in each pair of them.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the first pair of reserved cells. x has the same data type as was supplied to variables.

**variables ( float unsigned "<spaces>name" -- )**

float.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve space for unsigned floating-point numbers at a float aligned address in the data-space memory space and store *float* in each of them.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the first floating-point number. x has the same data type as was supplied to variables.

**variables ( single unsigned "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve unsigned cells at a cell aligned address in the data-space memory space and store *single* in each of them.

*name* is referred to as a variable.

Execution: ( -- address -> x )

address -> x is the address of the first reserved cell. x has the same data type as was supplied to variables.

**virtual ( object-size "<spaces>name" -- 1st )**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. Create a new definition for *name* with the execution semantics defined below. 1st is object-size. An ambiguous condition exists if `virtual` is executed in compilation state.

Note that the new definition does have no stack effects by default. Stack effects have to be specified separately. By providing a stack diagram phrase ( ... -- ... ) immediately following `virtual` and the definition name, the new definition is modified to incorporate stack effects. An ambiguous condition exists if the stack diagram does not contain at least one input parameter, or if the last input parameter is not a class.

*name* Execution: ( ... class -- ... )

Execute the definition whose token is stored in the virtual method table of *class*, the last input parameter of *name*. *class* is object or a direct or indirect subtype of object. The token is stored in the virtual method table by a succeeding execution of `is`. *name* is called a virtual method. An exception is thrown if *name* is executed before it is being assigned an execution semantics by `is`.

**virtual-definition ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type `virtual-definition`.

**virtual-definition ( unsigned address -> character unsigned  
virtual-definition -- 5 th )**

Initialize `virtual-definition` by erasing all members. Establish a link to the previous definition in the current vocabulary and update `latest`. Links will be removed when `virtual-definition` is deleted. The first unsigned is the index of the new virtual method within the virtual method table. Assign `virtual-definition` a name given by the character string `address -> character unsigned` and return it as 5 th.

`virtual-definition` is the constructor of the `virtual-definition` class.

**virtual-match-criterion ( -- search-criterion )**

strongforth.sf

`search-criterion` is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

`flag` is true if `definition` is a virtual definition and the selected data type heap matches the input parameter list of `definition`. The matching algorithm follows the rules of the StrongForth data type system.

The selected data type heap depends on state, the attributes of `definition` and the value of `single`:

single	state	attributes	data type heap
false	false		interpreter
false	false	immediate	interpreter
false	false	execute-only	interpreter
false	false	compile-only	(no match)
false	true		compiler
false	true	immediate	interpreter
false	true	execute-only	(no match)
false	true	compile-only	interpreter
true	false		interpreter
true	false	immediate	interpreter
true	false	execute-only	interpreter
true	false	compile-only	(no match)
true	true		compiler
true	true	immediate	compiler
true	true	execute-only	(no match)
true	true	compile-only	compiler

Note: Provide search-criterion to search in order to find a virtual definition with matching input parameters according to the rules of the StrongForth data type system.

#### **vocabulary ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type vocabulary.

#### **vocabulary ( vocabulary -- 1st )**

Make vocabulary an empty vocabulary and add it as the first item in the hidden vocabulary list.  
vocabulary is the constructor of the vocabulary class.

#### **vtable ( data-type -- vtable )**

strongforth.sf

vtable is the address of the virtual method table of the object with the data type data-type, or null if data-type is no object.

#### **vtable ( object -- vtable )**

vtable is the address of the virtual method table of object.

#### **vtable ( stack-diagram -- 1st )**

When used in a stack diagram, specifies an input or output parameter with data type vtable.

#### **vtable-criterion ( -- search-criterion )**

strongforth.sf

search-criterion is the qualified token of a definition with the execution semantics as specified below.

Execution: ( definition single -- flag )

`flag` is true if and only if the virtual method table of `definition` is equal to the value of `single`.

Note: Provide `search-criterion` to search in order to find a definition with a given name that is an object of a specific class.

**w/o ( -- fam )**

strongforth.sf

`fam` is a constant value for selecting the “write only” file access method when a file is created or opened.

**while ( destination -- origin 1st ) compile-only**

strongforth.sf

Compilation: Create and initialize `origin` and save a copy of the compiler data type heap. Append the runtime semantics given below to the current definition. `1st` is equal to `destination`. The semantics are incomplete until `origin` and `1st` are resolved.

Runtime: ( `single` -- )

If `single` is zero, continue execution at the location specified by the resolution of `origin`. Otherwise, continue execution.

**within ( address 1st 1st -- flag )**

Perform a comparison of a test value `address` with a lower limit `1st` (second parameter) and an upper limit `1st` (third parameter). `flag` is true if and only if either (lower limit < upper limit and (lower limit <= test value and test value < upper limit)) or (lower limit > upper limit and (lower limit <= test value or test value < upper limit)).

**within ( integer 1st 1st -- flag )**

Perform a comparison of a test value `integer` with a lower limit `1st` (second parameter) and an upper limit `1st` (third parameter). `flag` is true if and only if either (lower limit < upper limit and (lower limit <= test value and test value < upper limit)) or (lower limit > upper limit and (lower limit <= test value or test value < upper limit)).

**wordlist ( "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse `name` delimited by a space. Create a new empty vocabulary and add it as the first item in the hidden vocabulary list. Create a definition for `name` with the execution semantics defined below. Store the new vocabulary in the data field of the new definition.

*name* Execution: Remove the vocabulary from both the context vocabulary list and the hidden vocabulary list. Make the vocabulary the head of the context vocabulary list. An ambiguous condition exists if the vocabulary is not included in one of the two vocabulary lists.

**words ( "<spaces>name" -- )**

strongforth.sf

Skip leading space delimiters. Parse `name` delimited by a space. Send a list of all definitions in the vocabulary at the head of the context vocabulary list, whose name are identical to `name`, to the default output stream. If `name` is not provided, send a list of all definitions in the vocabulary at the



head of the context vocabulary list to the default output stream. Each definition occupies a separate line of text.

**write ( caddress -> character unsigned file -- )**

Write unsigned characters from address `caddress -> character` to the file identified by `file` starting at its current position. At the conclusion of the operation, `position` returns the next file position after the last character written to the file, and `size` returns a value greater than or equal to the value returned by `position`.

**write-eol ( file -- )**

Write a line terminator (carriage return and line feed) to the file identified by `file` at its current position. At the conclusion of the operation, `position` returns the next file position after the second character written to the file, and `size` returns a value greater than or equal to the value returned by `position`.

**write-line ( caddress -> character unsigned file -- )**

strongforth.sf

Write unsigned characters from address `caddress -> character` plus a line terminator (carriage return and line feed) to the file identified by `file` starting at its current position. At the conclusion of the operation, `position` returns the next file position after the last character written to the file, and `size` returns a value greater than or equal to the value returned by `position`.

**xalign ( -- )**

complex.sf

If the first unused address of the default memory space is not cell aligned, reserve the required minimum number of address units to make it aligned.

**xalign ( memory-space -- )**

complex.sf

If the first unused address of `memory-space` is not cell aligned, reserve the required minimum number of address units to make it aligned.

**xaligned ( address -- 1st )**

complex.sf

`1st` is the lowest cell aligned address greater than or equal to `address`.

**xor ( data-type data-type -- 1st )**

`1st` is the first `data-type` with attributes that are the bit-by-bit exclusive-or of the attributes of both parameters `data-type`.

**xor ( single logical -- 1st )**

`1st` is the bit-by-bit exclusive-or of `single` with `logical`.

**zero ( -- )**

strongforth.sf

Send a zero character (0) to the default output stream.

**zeros ( integer -- )**

strongforth.sf

If *integer* is greater than zero, send *integer* zero characters (0) to the default output stream.

Note: *integer* is assumed to be a signed number.

**[ ( -- ) immediate**

strongforth.sf

Interpretation: Stay in interpretation state.

Compilation: Perform the execution semantics given below.

Execution: Enter interpretation state.

**[ ' ] ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Append the runtime semantics given below to the current definition. An exception is thrown if *name* is not found.

Runtime: ( -- definition )

*definition* is the definition identified by *name*.

**[bind] ( "<spaces>name<sub>1</sub><spaces>name<sub>2</sub>" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name<sub>1</sub>* delimited by a space. Skip spaces. Parse *name<sub>2</sub>* delimited by a space. Find class *name<sub>1</sub>*. Find a virtual method *name<sub>2</sub>* that matches the compiler data type heap according to the rules of the StrongForth data type system. If no such virtual definition is found, compile *this* and try finding *name<sub>2</sub>* again. Append the runtime semantics of the virtual definition *name<sub>2</sub>* that is bound to the class identified by *name<sub>1</sub>* to the current definition. An exception is thrown if *name<sub>1</sub>* does not identify a class, if no suitable virtual definition *name<sub>2</sub>* is found or if *name<sub>2</sub>* is not a virtual definition within the scope of the class identified by *name<sub>1</sub>*.

**[char] ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Append the runtime semantics given below to the current definition.

Runtime: ( -- character )

*character* is the value of the first character of *name*. If the length of *name* is zero, *character* is the space character.

**[compile] ( "<spaces>name" -- ) compile-only**

Skip leading space delimiters. Parse *name* delimited by a space. Search the context vocabularies for a definition with the name *name*, whose input parameters match the compiler data type heap according to the rules of the StrongForth data type system. Change the compiler data type heap according to the stack effect of this definition. Append the semantics of the definition to the current definition. An exception is thrown if no matching definition is found.

**[ctrl] ( -- ) compile-only**

ascii.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. An exception is thrown if *name*'s first character is not a lowercase or uppercase letter. Append the runtime semantics given below to the current definition.

Runtime: ( -- character )

*character* is the ASCII control character the keyboard generates when typing *name*'s first character while holding the CTRL key. If the length of *name* is zero, *character* is the null character.

**[defined] ( "<spaces>name" -- flag ) immediate**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. *flag* is true if and only if *name* is the name of a definition that can be found by *search-context*.

**[dt] ( "<spaces>name" -- ) compile-only**

strongforth.sf

Compilation: Skip leading space delimiters. Parse *name* delimited by a space. Append the runtime semantics given below to the current definition. An exception is thrown if *name* is not the name of a data type.

Runtime: ( -- data-type )

Place *data-type*, the data type identified by *name*, on the stack.

**[else] ( -- ) immediate**

strongforth.sf

Compilation: Perform the execution semantics given below.

Execution: Skip leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of `[if] ... [then]` and `[if] ... [else] ... [then]`, until `[then]` has been parsed and discarded. If the parse area becomes exhausted, it is refilled with *refill*.

**[if] ( single -- ) immediate**

strongforth.sf

Compilation: Perform the execution semantics given below.

Execution: If any bit of *single* is not zero, continue execution. Otherwise, skip leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of `[if] ... [then]` and `[if] ... [else] ... [then]`, until either `[else]` or `[then]` has been parsed and discarded. If the parse area becomes exhausted, it is refilled with *refill*.

An ambiguous condition exists if `[if]` is postponed, or if the end of the input buffer is reached and cannot be refilled before the terminating `[else]` or `[then]` is parsed.

**[literal] ( -- ) compile-only**

strongforth.sf

Compilation: Append the runtime semantics given below to the current definition.

Runtime: ( x -- )

Allocate memory in the `data-space` memory space and save the compound data type of `x` in it. Append the runtime semantics given below to the current definition.

Runtime: ( -- `x` )

Place literal `x` on the stack. `x` has the same data type and the same value as those at compile time.

**[parent] ( "<spaces>name" -- ) compile-only**

strongforth.sf

Skip leading space delimiters. Pares *name* delimited by a space. Find a virtual definition *name* that matches the compiler data type heap according to the rules of the StrongForth data type system. If no such virtual definition is found, compile `this` and try finding *name* again. Append the runtime semantics of the virtual definition *name* that is bound to the parent of the class currently being defined to the current definition. An exception is thrown if no suitable virtual definition *name* is found or if *name* is not a virtual definition within the scope of the parent of the class currently being defined.

**[then] ( -- ) immediate**

strongforth.sf

Compilation: Perform the execution semantics given below.

Execution: Continue execution.

**[undefined] ( "<spaces>name" -- flag ) immediate**

strongforth.sf

Skip leading space delimiters. Parse *name* delimited by a space. `flag` is `false` if and only if *name* is the name of a definition that can be found by `search-context`.

**\ ( "ccc\" -- ) immediate**

strongforth.sf

Compilation: Perform the execution semantics given below.

Execution: Parse and discard `ccc` delimited by a `\` (backslash), but at most until the end the parse area. The number of characters in `ccc` may be zero to the number of characters in the parse area.

**\" ( "ccc<quote>" -- ) compile-only**

escape.sf

Parse `ccc` delimited by `"` (quote), using the translation rules below. Append the run-time semantics given below to the current definition.

Translation rules: Characters are processed one at a time and appended to the compiled string. If the character is a `\` character it is processed by parsing and substituting one or more characters as follows, where the character after the backslash is case sensitive:

Escape sequence	Substitution
<code>\a</code>	<code>&lt;bel&gt;</code>
<code>\b</code>	<code>&lt;bs&gt;</code>
<code>\e</code>	<code>&lt;esc&gt;</code>
<code>\f</code>	<code>&lt;ff&gt;</code>
<code>\l</code>	<code>&lt;lf&gt;</code>
<code>\m</code>	<code>&lt;cr&gt;&lt;lf&gt;</code>
<code>\n</code>	<code>&lt;cr&gt;&lt;lf&gt;</code>
<code>\q</code>	<code>" (quote)</code>
<code>\r</code>	<code>&lt;cr&gt;</code>

<code>\t</code>	<code>&lt;ht&gt;</code>
<code>\v</code>	<code>&lt;vt&gt;</code>
<code>\xyy</code>	(see below)
<code>\z</code>	<code>&lt;nul&gt;</code>

`\xyy` performs the following semantics: Parse two hexadecimal digits `yy` and return the resulting two-digit ASCII code. An exception is thrown if `\x` is not followed by two hexadecimal characters.

All other characters remain unchanged.

Runtime: ( `"ccc )`

`caddress -> character unsigned` is the translated string.

`\ " ( "ccc`

escape.sf

Parse `ccc` delimited by `"` (quote), using the translation rules below.

Translation rules: Characters are processed one at a time and appended to the compiled string. If the character is a `\` character it is processed by parsing and substituting one or more characters as follows, where the character after the backslash is case sensitive:

Escape sequence	Substitution
<code>\a</code>	<code>&lt;bel&gt;</code>
<code>\b</code>	<code>&lt;bs&gt;</code>
<code>\e</code>	<code>&lt;esc&gt;</code>
<code>\f</code>	<code>&lt;ff&gt;</code>
<code>\l</code>	<code>&lt;lf&gt;</code>
<code>\m</code>	<code>&lt;cr&gt;&lt;lf&gt;</code>
<code>\n</code>	<code>&lt;cr&gt;&lt;lf&gt;</code>
<code>\q</code>	<code>" (quote)</code>
<code>\r</code>	<code>&lt;cr&gt;</code>
<code>\t</code>	<code>&lt;ht&gt;</code>
<code>\v</code>	<code>&lt;vt&gt;</code>
<code>\xyy</code>	(see below)
<code>\z</code>	<code>&lt;nul&gt;</code>

All other characters remain unchanged.

`\xyy` performs the following semantics: Parse two hexadecimal digits `yy` and return the resulting two-digit ASCII code. An exception is thrown if `\x` is not followed by two hexadecimal characters.

`caddress -> character unsigned` is the translated string.

`] ( -- )`

strongforth.sf

Enter compilation state.

`~ ( complex complex float -- flag )`

complex.sf

If `float` is positive, `flag` is `true` if and only if the absolute value of the first `complex` minus the second `complex` is less than `float`.

If `float` is zero, `flag` is `true` if and only if the first `complex` and the second `complex` are exactly identical.

If `float` is negative, `flag` is true if the absolute value of the first `complex` minus the second `complex` is less than the absolute value of `float` times the sum of the absolute values of the first `complex` and the second `complex`.

**`~ ( float float float -- flag )`**

`float.sf`

If the third `float` is positive, `flag` is true if and only if the absolute value of the first `float` minus the second `float` is less than the third `float`.

If the third `float` is zero, `flag` is true if and only if the first `float` and the second `float` are exactly identical.

If the third `float` is negative, `flag` is true if the absolute value of the first `float` minus the second `float` is less than the absolute value of the third `float` times the sum of the absolute values of the first `float` and the second `float`.