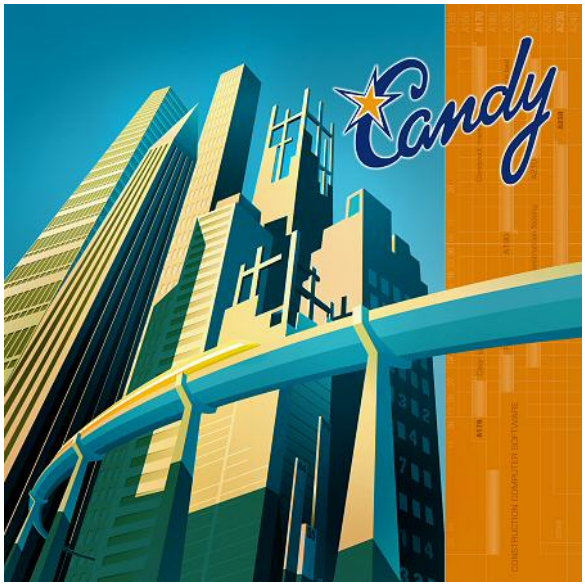
ClassVFX Intelligent Objects

v4

Microprocessor Engineering Limited



Published by Microprocessor Engineering

ClassVFX Intelligent Objects
User manual
Manual revision 4
19 October 2011

Software
Software version 4

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 CCS Object Model

1.1 Introduction

The CCS object model has been in use for 25 years in an application that now (2011) contains over one million lines of Forth source code.

The programmers working on this system range from deep systems programmers to quantity surveyors who have become application programmers. Above all, this is a pragmatic and usable system.

From the outset, this was not designed as the ultimate OOP system. It was an evolution from intelligent structures, driven by the requirements of increasing complexity in a real application.

1.1.1 Why Objects?

OOP has gained successs as a means of handling software complexity through hierarchical structures of carefully designed object types. Each class encapsulates a set o reusuable functions and reuses functions provided by its base class or classes. Reuse of OO code eliminates much of the bug prone code repeated throughout complex software projects.

1.1.2 Definition of an Object

Objects combine data and behavior and separation of interface from implementation. Objects are complete entities from a problem domain.

1.1.3 Extensibility

In a truly extensible system, components use other components. The network of these components is continuously expanded at run time by the addition of new or existing subclassed components. A system wide garbage collector is required to deallocate components.

1.1.4 Object Methods

Objects can only be addressed through methods associated with the object, inherited by the object from a previously defined class or type cast element of complex class using the dot notation.

Methods are classified to operate on the class or the instance of the class. Methods are defined using the Object Definition Language word `OPERATOR:` followed by the name of the method. In the CCS method naming convention the name must start with the '`<`' character and end with '`>`' character.

The word `OPERATOR:` is the same word used to define `TO` and other operators accepted by children of `VALUE`.

1.1.5 Class methods

Class methods uses the `TYPE:` definition structure and can not address any data space. These methods will be excuted during compilation as oposed to laying down code that will be executed during runtime.

1.1.6 Instance methods

Instance methods use the instance of a `TYPE:` structure and can access the dataspace of the instance if required. If access to the instance type: structure is required during runtime then the method can be made immediate with the object definition word `STRUCTURE-METHOD`. The method will then be executed at compile time and not compiled inline.

1.1.7 Super Object Methods

```
 Class Methods
 CLASS
   mruns <default> make-inst
   mruns <sizeof>  sizeprovider
   mruns <addr>    ?complit
   mruns <self>    ?complit
Instance Methods
INST
   mruns <offsetof>  offsetprovider   structure-method
   mruns +offsetof +provider        structure-method
   mruns <sizeof>  sizeprovider     structure-method
   mruns <default> inst-addr        structure-method
   mruns <ptr>     inst-addr        structure-method
   mruns <&ptr>    inst-addr        structure-method
   mruns <addr>    inst-addr        structure-method
   mruns <copy>    inst-copy        structure-method
   mruns <blank>   inst-blank       structure-method
   mruns <erase>   inst-erase       structure-method
   mruns <init>    inst-erase       structure-method
   mruns <zero>    inst-erase       structure-method
   mruns <count>   inst-count       structure-method
   mruns <self>    inst-self        structure-method
   mruns <#bpe>    inst-bpeprovider structure-method
```

Methods are considered to be Modifiers, Inspectors, Constructors or Destructors.

`Inspectors`
: return the properties of an object instance.

`Modifiers`
: changes the properties or data of and object instance.

`Constructors`
: builds an object instance so that Instance Inspector methods can be used on the object.

`Destructors`
: destroy the object instance.

1.1.8 Object terminology

`{type:}` A Forth word defined with `TYPE:`.

`{instance}`
: A Forth word defined with an instance of `TYPE:`.

`ODL` Object Definition Language. Words that are only valid during the definition of a Object. (compile time)

1.2 Creating Object Types

The CCS Object model has an Object Definition Language that is only valid when defining object types.

```
Object definition Language
TYPE: [name:]
  SUPERCLASS
  INHERITS  ( alias for SUPERCLASS )
EXTEND-TYPE [name:]
  MRUNS <method>
  STRUCTURE-METHOD
  CLASS
  INST
  n SKIPPED
END-TYPE
```

CCS types are defined with `TYPE:` followed by a name ending with a colon.

```
TYPE: <name:> ... END-TYPE
```

The colon implies that the new word can itself create new Forth words that behave according to the type definition. This is an instance of a type. An instance of a type can not create new Forth words. The type definition describes any data space requirements and valid methods that can be used to address the object.

1.2.1 Inheritance

All objects inherit basic methods from the super object. The CCS object model allows a single inheritance oppurtunity during definition of the type: using the object definition word `INHERITS` followed by the type: name.

1.2.2 Unary Objects

An object with a single data field or member is called a unary object. Some Objects have multiple fields but if the Instance Modifier Methods address the data space as a monolithic unit then the object can be classified as unary.

```
Example of an Unary object
TYPE: C-VAR:
  1 SKIPPED          \ reserve 1 byte of data space
  MRUNS <to> C!      \ method to modify data space
END-TYPE
```

1.2.3 Complex Objects

`TYPE:` definition with multiple fields or members are called complex objects.

1.2.4 Array Object

An array is a rectangular multi dimensional data structure that holds data elements of the same type in subdivisions called cells. The bounds of an array are the lowest and highest permissible subscript integers. The value of a dimension's lower bound is called its base.

1.3 Instance of a Type

1.3.1 Static Instance

A static instance of an object is compiled in the Forth dictionary and its instance structure and structure data space is inside the Forth dictionary space. The structure data space is erased for numeric and zero terminated string objects. Counted string object structure data space are blanked.

1.3.2 Local Instances

A local instance of an object is compiled on the Forth locals stack and only the structure data space requirement is allocated. Unary object data space is not initialized. It is assumed that the developer will be assigning values to these instances just now. Complex objects with a sub classed <default> class method can lay down initialization code. The default class method for complex objects is to erase the structure data space.

1.4 Object Service Definitions

The client/server model is very good model to use when implementing an object model. The 'server code' operates on an object instance or creates an object instance from an existing class.

Object Service Definitions provide a service to an object. The idea of a specific object service definition is to avoid creating a new language using the object methods. Use Forth and apply some discipline to parameter stack inputs and outputs. Document the service requirement and object effect. Announce and guarantee the service. Using standard naming conventions developers can use standard tools to get access to the service definitions.

The stack input of the specific object service definition can only be one of the following three formats to qualify as an object service.

1. object — flag or object
2. object> — flag or void
3. vobject> — flag or void

1.4.1 Service type 1

The definition copies the object to its own data space, performs the service by operating on that data space and returns a result flag or copies modified data space to the object.

1.4.2 Service type 2

The definition performs the service by operating on the client data space. A result flag or void is returned. The client can use the modified object data space.

1.4.3 Service type 3

The definition performs the service by operating on the client virtual data space and/or object data space. A result flag or nothing is returned. The client can use the modified object data space. The service defintion does not destroy the client virtual data space, the client must make and destroy the virtual data space.

1.5 Coding Standards

1.5.1 Parameter stack comments

These are the basic CCS house rules.

- The stack comment must indicate the `TYPE:` instance.
- Object Pointers must have a trailing `">"`.
- Virtual object instances must have a leading `"v"` and a trailing `">"`.

1.6 Elements of Simple Design

1. Passes its tests
2. Minimizes duplication
3. Maximizes clarity
4. Has fewer elements
5. Keep it simple
6. If it ain't broke, don't fix it

2 ClassVfx OOP

2.1 Introduction

The source code is in the directory *Lib\OOP\ClassVFX*. The file *MakeClassVfx.bld* is compiled to produce the production version of the code. *TestClassVfx.fth* contains test code.

ClassVFX was developed over a number of years in collaboration with Construction Computer Software of Cape Town, South Africa. We gratefully acknowledge their collaboration and permission to release it. ClassVFX is heavily used in their construction industry planning software, which is one of the largest Forth applications ever written. Modifications to ClassVFX will only be released after the agreement of CCS.

ClassVFX is a halfway house between a full object oriented system and an intelligent structures system. Types, or classes, can be defined with single inheritance. Method names have to be predefined using

```
OPERATOR: <method-name>
```

Field, or data member, names are private, but are accessible using a dot notation. There are no equivalents of `SUPER` and `SELF`. There is no late binding.

In this documentation **type**s and **class**es are synonymous. **Object**s are **instance**s of a **type**. Objects have a default action if no method is specified. Usually the default action is to fetch the contents of the object, but in a few cases the default action is to return an address.

Types/Classes can have both class and instance methods. The default method for a type is to create an instance. If a type is used inside a colon definition a local variable version is created and destroyed at run time.

Operators, or methods, must be declared as above before use.

2.2 How to use TYPE: words

`TYPE:` definitions may be used in four ways:

- as an abstract template which is used with a base address on the stack.

  ```
  Point.x or x
  ```

- to define an instance of a structure in the dictionary, e.g.

  ```
  Point: MyPoint
  ```

- to define a local variable inside a colon definition, but outside any other local variable defining mechanism. If another locals defining mechanism such as the ANS `LOCALS| ... |` mechanism or the MPE `{ ... }` mechanism has been used the use of `Point: foo` inside a colon definition will simply add `FOO` to the existing local frame.
- to define a field inside another `TYPE:` definition.

```
operator: <method1>
operator: <method2>
operator: <method3>

type: line:
  point: start
  point: end

  :m <method1>     ... ;m
  :m <method2>     ... ;m
  mruns <method3>  <some-word>
  :m <xxx>         a b c d  ;m  structure-method

end-type

Line: MyLine
  1 2 to Myline.start
  5 to Myline.end.y
```

At runtime, the method operates on the address of the data. Because of this, a method which requires the address of the instance structure has to be marked by the word `STRUCTURE-METHOD` which causes the compiler to generate the address of the instance structure, **not** the type structure.

ClassVFX allows both `CLASS` and `INST`ance methods to be defined for a type. `INST`ance methods, the default, operate on the address of the data item. `CLASS` methods operate on the address of the type data structure. As described above, `STRUCTURE-METHOD`s operate on the instance data structure.

Single inheritance can be defined using `SUPERCLASS <type>` or `INHERITS <type>` before any field or method is defined.

```
TYPE: <type>  SUPERCLASS <supertype>
  ...
END-TYPE
```

At run-time, methods are provided with the address of the required data. CLASS/TYPE methods receive the address of the TYPE/CLASS data structure, INSTance methods receive the address of the data item. INSTance methods that require the address of the instance data structure must be marked by `STRUCTURE-METHOD`. Methods may be defined as nameless words:

```
:M <method-name> ...  ;M
```

or as the action of a method:

```
MRUNS <method-name> <action-name>
```

The code below is taken from the definition of the default type.

```
class     \ define methods for the type
   :m default    make-inst  ;m
   :m sizeof     type-size @ ?complit  ;m
   :m addr       ?complit  ;m
inst      \ define methods for the instance
   mruns default noop
   :m sizeof     type-size @ ?complit  ;m  structure-method
   mruns addr    noop
   :m offsetof   off-start @ ?complit  ;m  structure-method
   :m +offsetof  off-start @ ?complit+  ;m  structure-method
```

To use the nested field system, the Forth system has been modified to accept compound names in which the elements of the structure are separated by the '.' character. This feature is enabled and disabled by the words `+STRUCTURES` and `-STRUCTURES`.

2.3 Predefined types

```
char:         byte - 8 bit variable
word:         word - 16 bit variable
int:          long - 32 bit variable and synonyms
  dword:
  long:
  ptr:
xlong:        longlong - 64 bit variable
bytes:        byte array: size specified by n BYTES BYTES:
cstring:      counted string: size specified by BYTES before CSTRING:
zstring:      zero term. string: size specified by BYTES before ZSTRING:
field:        byte array, only ADDR operator, size specified by BYTES
```

2.4 Predefined methods/operators

Note that not all predefined types support all methods.

```
 0 operator default            usually a fetch operation
 1 operator ->                 store operator
 1 operator to                   "
 2 operator addr               address operator
 3 operator inc                increment by one
 4 operator dec                decrement by one
 5 operator add                n add to
 6 operator zero               set to 0
 7 operator sub                subtract from
 8 operator sizeof             size
 9 operator set                set to -1
10 operator offsetof           offset in object
11 operator +offsetof          add offset in object
12 OPERATOR FETCH              get contents
13 OPERATOR ADDR\CNT           address under count
14 OPERATOR TWIST              change endian of the data type
15 OPERATOR CONSTRUCT          build an instance of this type
15 OPERATOR MAKE               build an instance of this type
op# ADDR OPERATOR ADDROF
OPERATOR: <=>    type_addr_y <=> <type_x> --- set typedef_x = typedef_y
  op# <=> OPERATOR <copy>
OPERATOR: <blank>              blank object for object size
OPERATOR: <erase>              fill obj with null for object size
OPERATOR: <COUNT>
OPERATOR: <make>
OPERATOR: <destroy>
OPERATOR: <INIT>
OPERATOR: <fetch>
```

2.5 Example structure

```
TYPE: POINT:  \ --
\ Defines a type called POINT: with the following fields )
  PROVIDER: NOOP        \ defines the address provider, defaults to NOOP
  0 OFFSET:             \ defines the initial offset, defaults to 0
  INT: Y
  INT: X
  10 BYTES FIELD: FOO
                        \ fetch operation
  :m default
     2@
  ;m
  mruns to      2!
  ...

END-TYPE
```

2.6 Data structures created by TYPE:

`TYPE:` definitions, fields, objects and so on all use a common data structure that is generated by the defining words.

These structures are associated with a word (the address provider) that can provide the starting address of the structure implementation. By supplying the cfa of NOOP, no address is provided, and so the structure is purely a template. For templates, address provider = 0 or NOOP, an offset may also be defined. NOOP and 0 are the default address provider and offset of templates.)

A similar structure is used for instances of a TYPE:. These are created by the word MAKE-INST.

2.6.1 TYPE: definitions

The following structure is created by TYPE:

```
   header                          standard PFW layout
0  jmp do_type                     5 bytes
1  cfa of address provider         4 bytes
2  initial offset                  4 bytes  0 for class
3  link to last field defined      4 bytes
4  type size - final offset        4 bytes
5  Magic number                    4 bytes
6  anchor of instance method chain 4 bytes
7  anchor of type method chain     4 bytes
8  link to previous type defined   4 bytes
9  private wordlist                ? bytes
```

2.6.2 MAKE-INST definitions

The following structure is created by MAKE-INST

```
   header                          standard PFW layout
0  jmp do_inst                     5 bytes
1  cfa of address provider         4 bytes
2  offset from start of type       4 bytes
3  link to last instance of type   4 bytes
4  size of instance data           4 bytes
5  0                               4 bytes
6  pointer to TYPE/CLASS           4 bytes
7  data if static
```

2.7 Local variable instances

When an instance is defined inside a colon definition, an uninitialised local variable/array is built. Several instances can be built. Normally the size of all local variables is rounded up to a cell boundary by the compiler

2.8 Defining methods

```
: :M            \ struct -- struct  ; :M <operator> <actions ...> ;M
```

defines the start of a method. The method/operator name must follow.

```
: ;M            \ struct -- struct ; SFP012
```

marks the end of a method definition.

```
: MRUNS          \ struct -- struct  ; MRUNS <operator> <actions> ;M
```
Defines a method which runs a previously defined word.

2.9 Create Instance of an object

```
: CREATE-INST    \ "<name>" -- ; -- addr
```
From VFX Forth v4.4, this is a synonym for `CREATE`. When compiling on previous VFX versions instances needed to be immediate.

```
: make-inst      \ class -- ; i*x -- j*y ; build instance of type
```
Builds an instance of a `TYPE:`. This word has serious carnal knowledge of the internal workings of VFX Forth. Don't call us for help!

2.10 Defining TYPE: and friends

```
create type-template     \ -- addr
```
The type chain from which others are derived.

```
CREATE ptr-template      \ -- addr
```
The ptr chain from which others are derived.

2.10.1 TYPE definition

```
: type:-runtime \ type-struct --
```
The run-time action of children of `TYPE:`.

```
: TypeChildComp,         \ xt --
```
Compile a child of `TYPE:`.

```
: type:          \ -- struct ; --
```
Start a new `TYPE:` definition.

```
: PTR:           \ -- struct ; --
```
Make a new structure defining word.

```
: end-type       \ struct --
```
Finish off a `TYPE:` definition

```
: EXTEND-TYPE    \ "<type>" -- struct ; EXTEND-TYPE <type> ... END-TYPE
```
Extend the given `TYPE:` definition.

```
: SUPERCLASS     \ struct "<type>" -- struct
```
Use this inside a `TYPE:` definition before defining any data or methods. The current type will inherit the data and methods of the superclass.

```
: INHERITS       \ struct "type" -- struct
```
A synonym for `SUPERCLASS`.

```
: provider:      \ struct "name" -- struct ; <name> is address provider
```
Sets a different address provider.

```
: with:          \ -- ; WITH: <some-provider> LINE: <myline>
```
Used before declaring an instance to override the default address provider.

```
: SKIPPED        \ struct size -- struct
```
Increase overall size of struct by size. `SKIPPED` can be used to jump over items from a previous instance.

```
: OFFSET:        \ struct offset -- struct
```

Define the offset of a `TYPE:` as starting at a value other than zero. Must be used before any data is defined.

`: TypeCast: \ -- ; TYPECAST: <inst> <type>`
Forces a previously defined instance to be a pointer to a type/class.

`SYNONYM PointsTo: TypeCast: \ -- ; synonym for TYPECAST:`
Forces a previously defined instance to be a pointer to a type/class.

`: type-self \ -- type`
Used in TYPE: <name> ... END-TYPE to refer to the type/class being defined.

`: EXECUTE-MEMBER-METHOD \ struct-inst member-inst methodid ---`
Attempt to execute method for inst. Return true if successful.

`: EXECUTE-PTR-MEMBER-METHOD \ member-inst methodid ---`
Attempt to execute method for inst. Return true if successful.

`: EXECUTE-MEMBERS \ inst method --`
Apply the given method to all members of the instance of a type/class.

`: TWIST-STRUCTURE \ inst --`
Twist structure method

`: INIT-STRUCTURE \ inst --`
Init structure method

2.11 Dot notation parser

In order to deal with structures and fields without having to backtrack the input stream or the execution order, an additional stage is added to the Forth parser to allow phrases of the forms:

```
inst.field
inst.field.field
type.field
type.field.field
```

to be parsed, where each item is separated by a dot character. The first item must be an instance of a type or a type. If it is an instance, the address is provided, otherwise the base address is assumed to be on the stack. Any items between the first and last item add their offsets to the address, and the last item performs the usual operation of the field as defined by an operator. For example:

```
type: point:
  int: x0
  int: y0

  :m <op1> ... ;m
  :m <op2> ... ;m
end-type

point: Mypoint
  5 to MyPoint.x0
```

We might define a line as joining two points:

```
type: line:
  point: p1
  point: p2
  ...
end-type

line: MyLine
  5 to MyLine.p2.y0
```

`: +structures  runword \ --`
Switch on the structure compiler.

`: -structures  runword \ --`
Switch off the structure compiler.

3 Typeops.fth

Operators used with ClassVFX

3.1 Array Object

```
OPERATOR: <elm$!>                                    \ addr cnt elm# ---
OPERATOR: <elm$+>                                         \ addr cnt ---
OPERATOR: <elm$@>                                   \  elm# --- addr cnt
OPERATOR: <elmaddr>                                \ n0...n1 --- elmaddr
OPERATOR: <#bpe>                                             \ --- #bpe
```

3.2 Class Property Operators

3.2.1 Class Inspector Methods

```
op# sizeof OPERATOR <sizeof>                                  ( --- un )
```

Return the overall data space requirement of the class. Class Inspector.

```
<sizeof> {type:}
Stack effect : ( --- un )
```

3.2.2 Instance Modifier Methods

```
OPERATOR: <copy>                                               \ addr ---
```

Copy instance of same type: to an instance of type: Instance Modifier

```
<ptr> {instance} <copy> {instance}
Stack effect : ( addr --- )
```

```
OPERATOR: <erase>
```

Fill the instance data space with zeroes. Instance Modifier

```
<erase> {instance}
Stack effect : ( --- )
```

```
OPERATOR: <blank>
```

Fill the instance data space with blanks. Instance Modifier.

```
<blank> {instance}
Stack effect : ( --- )
```

3.2.3 Instance Property Methods

```
OPERATOR: <count>       ( --- addr u-cnt )
```

Return the data address and size of the type instance. Instance inspector.

```
<count> {instance}
<count> {ptr>.instance}
<count> {complex_instance.instance}
Stack effect : ( --- addr u-cnt )
```

OPERATOR: <mem_count> (--- addr u-cnt)
Return the data address and available size of the memory object Instance inspector.

op# ADDR OPERATOR ADDROF (--- addr)
Return the data address of the type. Instance inspector.

op# ADDR OPERATOR <addrof> (--- addr)
Return the data address of the type. Instance inspector.

op# ADDR OPERATOR <addr> (--- addr)
Return the data address of the type. Instance inspector. Synonym <addrof>

op# offsetof OPERATOR <offsetof> (--- un)
Return the offset of the instance member of a complex type. Instance or class inspector.

```
<offsetof> {type:.instance}
<offsetof> {ptr>.instance}
<offsetof> {complex_instance.instance}
Stack effect : ( --- un )
```

3.2.4 Virtual Objects

Objects that allocate their data space in virtual memory and their type instance structure in the FORTH dictionary or local return stack are called Virtual Objects.

When using Virtual Objects one should be careful when passing the instance address and the data address to use the correct method.

The <&ptr> method returns the instance address and the <ptr@> method returns the data address.

Virtual Objects has a strict naming convection that must be adhered to. This will prevent confusion and incorrect methods applied to instances. Start the name of a Virtual Object with a lowercase "v".

Example: A local vzFILENAME: %vzfile

3.2.5 Pointer Methods

OPERATOR: <ptr> (--- addr)
Return the data address of the first element of the type structure. Instance inspector.

```
<ptr> {instance}
Stack effect : ( --- addr )
```

OPERATOR: <&ptr> (--- addr)
Return the data address of an instance of pointer type. This is used to pass an instance of a pointer. Instance inspector.

```
<&ptr> {pointer_instance}
Stack effect : ( --- addr )
```

OPERATOR: <ptr!> (instance_addr ---)
Set the data address of an instance of a type in an instance of POINTER: Type at address must match the type of POINTER: Instance Modifier.

```
<ptr!> {pointer_instance}
Stack effect : ( addr --- )
```

OPERATOR: <ptr@> (--- instance_addr)
Return the data address of the first element of the type structure of a virtual object. Instance inspector.

```
<ptr@> {virtual_instance}
Stack effect : ( --- addr )
```

OPERATOR: <ptr0>
Zero the data of an instance of pointer. Instance Modifier.

```
<ptr0> {pointer_instance}
Stack effect : ( --- )
```

3.2.6 INSTANCE MODIFIER OPERATORS

op# to OPERATOR <to>
Primary store method.

op# ZERO OPERATOR <zero> (---)
Fill the data space of an instance with zeroes. Instance Modifier Restrictions: Does not apply to counted string instances.

```
<zero> {instance} ---
Stack effect : ( --- )
```

op# SET OPERATOR <set>
Assign the value held by the instance to highest possible value that it could hold. (all value bits to 1) for numeric value objects. Instance Modifier. Restrictions: Numeric simple instance only. Avoid complex types.

```
<set> {instance}
Stack effect : ( --- )
```

OPERATOR: <@off>
Fetch the current value of an unary numeric type and the fill the data space of an instance with zeroes. Instance Modifier. Restrictions: Does not apply to counted string instances.

```
<@off> {instance}
Stack effect : ( --- n )
```

OPERATOR: <toggle> (---)

Check th e current value of the object, if zero then assign the value held by the instance to highest possible value that it could hold. (all value bits to 1) for numeric value objects. If any value other than zero, then zero all the bits in the object. Instance Modifier. Restrictions: Numeric simple instance only. Avoid complex types.

```
<toggle> {instance}
Stack effect : ( --- )
```

3.3 Arithmetic Methods

OPERATOR: <mul>

Multiply operator for unary numeric types. The input type must be on the correct numeric stack for the type. Instance Modifier.

```
 [type] <mul> {instance}
 Stack effect : ( [type] --- )
```

OPERATOR:

Divide operator for unary numeric types. The input type must be on the correct numeric stack for the type. Instance Modifier.

```
Usage : [type] <div> {instance}
Stack effect : ( [type] --- )
```

op# SUB OPERATOR <sub>

Subtract operator for unary numeric types. The input type must be on the correct numeric stack for the type. Instance Modifier.

```
 [type] <sub> {instance}
 Stack effect : ( [type] --- )
```

op# ADD OPERATOR <add>

Addition operator for unary numeric types. The input type must be on the correct numeric stack for the type. Instance Modifier.

```
 [type] <add> {instance}
 Stack effect : ( [type] --- )
```

op# INC OPERATOR <inc> (---)

Increment numeric value with one. Operator for unary numeric types. Instance Modifier.

```
 <inc> {instance}
 Stack effect : ( --- )
```

op# DEC OPERATOR <dec> (---)

Decrement numeric value with one. Operator for unary numeric types. Instance Modifier.

```
<dec> {instance}
Stack effect : ( --- )
```

OPERATOR: <sign?>
Make the numeric value positive. Return a flag indicating the sign. Flag = true negative & flag = false positive. Operator for unary numeric types. Instance Modifier.

```
<sign?> {instance}
Stack effect : ( --- flag )
```

OPERATOR: <negate>
change sign of value.

OPERATOR: <neg>
change sign of value to negative

OPERATOR: <abs>
change sign of value to positive

3.4 Float Methods

OPERATOR: <f@>
Float operator for unary numeric types. Fetch object to float stack. Fetch from object output as f-var type

```
<f@> {instance}
Stack effect : ( --- f )
```

OPERATOR: <f!>
Float operator for unary numeric types. Store float from stack in object. Float will be converted. Float input store in object.

```
<f!> {instance}
Stack effect : ( f --- )
```

OPERATOR: <f>d>
Float operator for unary numeric types. Fetch float from object and convert integer portion to 64bit signed value

```
<f>d> {instance}
Stack effect : ( --- d1 )
```

OPERATOR: <f>b>
Float operator for unary numeric types. Fetch float from object and convert to 64bit signed value and dpl. Fetch from float object output as b-var type

```
<f>b> {instance}
Stack effect : ( --- b1 )
```

3.5 Instance Constructor And Destructor Methods

OPERATOR: <make>

Set the object structure members to the init state. Allocate any virtual data space required to function and allow inspector methods to operate.

`OPERATOR: <destroy>`
Clear the object structure members. Deallocate any virtual data space and invalidate all inspector methods.

`OPERATOR: <init>`
Set the object structure members to the init state.

`OPERATOR: <valid?>`
Check the current state of the object. Return True if object passes the object validation test else false. Instance Inspector.

```
<valid?> {instance}
Stack effect : ( --- flag )
```

`OPERATOR: <data>`
Return the data address of an instance

`OPERATOR: <busy?>`
Return the current busy value of the object. Instance Inspector.
Usage : <busy?> {instance}

Stack effect : (— busy#)

`OPERATOR: <busy!>`
Set the current busy value of the object. Instance modifier
Usage : busy# <busy!> {instance}

Stack effect : (busy# —)

`OPERATOR: <prime>`
Used after init for first time structure setup. eg dim matrix within a structure

3.6 String Object

3.6.1 Instance Property Operators

`OPERATOR: <len>  ( --- len )`
Return -trailing length of counted string objects or trailing zero character of zero terminated string objects. Instance Inspector.

```
<len> {instance}
Stack effect : ( --- u )
```

`op# <len> OPERATOR <chars>  ( --- u )`
Return -trailing length of counted string objects or trailing zero character of zero terminated string objects. Instance Inspector.

```
<len> {instance}
Stack effect : ( --- u )
```

OPERATOR: <null?> (--- flag)
Return true if no chars in counted string objects or zero terminated string objects else false if any chars. Instance Inspector.

```
<null?> {instance}
Stack effect : ( --- flag )
```

OPERATOR: <trim_count> (--- addr u-cnt)
Return remaining addr and cnt after performing -trailing and -leading on the data space of the object. Instance Inspector.

```
<trim_count> {instance}
Stack effect : ( --- c-addr u-cnt )
```

3.6.2 Instance Modifier methods

OPERATOR: <concat-> (c-addr u-cnt ---)
Add counted string to -trailing end of string object. Instance Modifier.

```
<concat-> {instance}
 Stack effect : ( c-addr u-cnt --- )
```

OPERATOR: <concat> (c-addr u-cnt ---)
Add counted string to start of string object. Instance Modifier.

```
<concat> {instance}
Stack effect :  ( c-addr u-cnt --- )
```

OPERATOR: <trim> (---)
Remove leading and trailing blank characters from string object. Instance Modifier.

```
<trim> {instance}
Stack effect : ( --- )
```

OPERATOR: <trim-trailing> (---)
Remove trailing blank characters from string object. Instance Modifier.

```
<trim-trailing> {instance}
Stack effect : ( --- )
```

OPERATOR: <trim-leading> (---)
Remove leading blank characters from string object. Instance Modifier.

```
<trim-leading> {instance}
Stack effect : ( --- )
```

OPERATOR: <upper>
ASCII Upper case string object. Only char values greater or equal to ASCII a and lower or equal to ASCII z will be effected.

`OPERATOR: <lower>`
ASCII lower case string object. Only char values greater or equal to ASCII A and lower or equal to ASCII Z will be effected.

`OPERATOR: <ansi_upper>`
Upper case string object to current ANSI locale setting

`OPERATOR: <ansi_lower>`
Lower case string object to current ANSI locale setting

`OPERATOR: <$xlat>  ( addr cnt --- )`
Replace next string token found in string buffer instance.
`Usage : <$xlat> [instance] ( addr cnt --- )`

`OPERATOR: <xlat>  ( --- )`
Replace all constant string tokens found in string buffer instance.

`OPERATOR: <expandmacro> ( --- )`
Expand any macros in string buffer instance.

`OPERATOR: <char+>  ( chr obj --- )`
Add char on stack at end of string object

`OPERATOR: <char->  ( obj --- char )`
Remove and return char at end of object

3.7 8-bit counted buffer operators

`OPERATOR: <b$place>  ( addr cnt buf> --- )`
Place string at buf> with 8-bit count

`OPERATOR: <b$append>  ( addr cnt buf> --- )`
Append string at buf> with 8-bit count

`OPERATOR: <b$count>  ( buf> --- addr cnt )`
8-bit COUNT of string at buf>

`OPERATOR: <b$addr>  ( --- addr )`
String byte COUNT address

`OPERATOR: <b$len>  ( --- ulen )`
String byte byte length

`OPERATOR: <b$char+>  ( char --- )`
Add char to end of byte counted string

3.8 16-bit counted buffer operators

`OPERATOR: <w$place>  ( addr cnt buf> --- )`
Place string at buf> with 16-bit count

`OPERATOR: <w$append>  ( addr cnt buf> --- )`
Append string at buf> with 16-bit count

`OPERATOR: <w$count>  ( buf> --- addr cnt )`
16-bit COUNT of string at buf>

`OPERATOR: <w$addr>  ( --- addr )`
String word COUNT address

`OPERATOR: <w$len>  ( --- ulen )`
String 16-bit byte length

3.9 32-bit buffer operators

```
OPERATOR: <l$place>                                    ( addr cnt buf> --- )
```

Place string at buf> with 32-bit count

```
OPERATOR: <l$append>                                   ( addr cnt buf> --- )
```

Append string at buf> with 32-bit count

```
OPERATOR: <l$count>                                    ( buf> --- addr cnt )
```

32-bit COUNT of string at buf>

```
OPERATOR: <l$addr>                                              ( --- addr )
```

String long COUNT address

```
OPERATOR: <l$len>                                               ( --- ulen )
```

String 32-bit byte length

3.10 Filename Operators

3.10.1 Instance Inspector Operators

```
OPERATOR: <drive?>      ( --- flag )
```

Does the file name object include a valid drive specifier. True or false Valid drive specifier <drive_letter>: <drive_letter> must be in character position 1 and ColonChar must be in character position 2. Instance Inspector. FORTH - drive_query

```
<drive?> {instance}
Stack effect : ( --- flag )
```

```
OPERATOR: <path?>       ( --- flag )
```

Does the file name object include any path or folder specifier. True or false. Valid path specifier BackSlashchar. Instance Inspector. FORTH - path_query

```
<path?> {instance}
Stack effect : ( --- flag )
```

```
OPERATOR: <unc?>        ( --- flag )
```

Does the file name object include a valid UNC share name. True or false. Instance Inspector. FORTH - U_N_C_query

```
Valid UNC share name   \\<computer_name>\<share_name>
<\\> must be in character position 1.
first leave is the computername.
second leave the share name.
<unc?> {instance}
Stack effect : ( --- flag )
```

```
OPERATOR: <wild?>       ( --- flag )
```

Does the file name object include any valid wild filename characters. True or false. Valid valid wild filename characters is * and ?. Instance Inspector. FORTH - wild_query

```
<wild?> {instance}
Stack effect : ( --- flag )
```

3.11 Filename Instance Modifier Methods

3.11.1 Path Methods

`OPERATOR: <path>  ( --- )`

Remove remove one leave from filename object. Instance Modifier.

```
<path> {instance}
Stack effect : ( --- )
```

`OPERATOR: <+path>  ( --- )`

Add path to object using the filename according to CCS virtual directory rules. Instance Modifier. FORTH - plus_path

```
<+path> {instance}
Stack effect : ( --- )
```

`OPERATOR: <-path>  ( --- )`

Remove all leaves from filename object except last. Instance Modifier. FORTH - minus_path

```
<-path> {instance}
Stack effect : ( --- )
```

`OPERATOR: <share>  ( --- )`

Remove all leaves from filename object except drive mapping or share name. Instance Modifier.

```
<share> {instance}
Stack effect : ( --- )
```

`OPERATOR: <-share>  ( --- )`

Remove share or drive mapping from zfilename. Instance Modifier.

```
<-share> {instance}
Stack effect : ( --- )
```

`OPERATOR: <+filename>  ( addr cnt --- )`

Add address count to zFILENAME object as the last leaf of the folder path. Instance Modifier.

```
<-share> {instance}
Stack effect : ( --- )
```

`OPERATOR: <-ext>  ( --- )`

Remove existing file extesion from filename object Instance Modifier. Instance Modifier.

```
<-ext> {instance}
```

`OPERATOR: <+ext>  ( addr u-cnt --- )`

Remove existing file extesion from filename object and add extension at address count. Instance Modifier.

```
<+ext> {instance}  (c-addr u-cnt  --- )
```

```
OPERATOR: <ext$>        ( --- c-addr u-cnt|false )
```
Does the file name object include any file extension. Instance Inspector.

```
<ext$> {instance}
Stack effect : ( --- c-addr u-cnt|false )
```

```
OPERATOR: <+cjj>        ( --- )
```
Add current Coy$ and Job$ as file extension to filename object name. Instance Modifier.

```
<+cjj> {instance}  ( --- )
```

```
OPERATOR: <squeeze>     ( --- )
```
Remove all blanks within string object. Instance Modifier.

```
<squeeze> {instance}
Stack effect : ( --- )
```

3.12 Matrix Cells

3.12.1 <mat!> Operators

[class] = the defined class of the data represented in one cell. If the class is a simple numeric type then the data will be on the appropriate stack. If the class is a complex type then the class <copy> method will be used to assign the value of the cell. ie the data address of the instance.

```
OPERATOR: <mat!>
```
Store class type for row# in matching class matrix instance.

```
Usage :   <mat!> [instance] \ [class] row# ---
```

```
OPERATOR: <matx2!>
```
Store class type for row# and col# in matching class matrix table instance.

```
Usage :   <matx2!> [instance] \ [class] row# col# ---
```

```
OPERATOR: <matx3!>
```
Store class type for row# and col# and table# in matching class matrix table instance.

```
Usage :   <matx3!> [instance] \ [class] row# col# table# ---
```

```
OPERATOR: <matx4!>
```
Store class type for row# and col# and table# and group# in matching class matrix table instance.

```
Usage :   <matx4!> [instance] \ [class] row# col# table# group# ---
```

3.12.2 <&cell..> Operators

```
OPERATOR: <&cellx1>     ( row# --- addr )
```
Fetch the class instance address of the required matrix element at row#.

```
OPERATOR: <&cellx2>     ( row# col# --- addr )
```
Fetch the class instance address of the required matrix element at row# and col#.

```
OPERATOR: <&cellx3>     ( row# col# table# --- addr )
```

Fetch the class instance address of the required matrix element at row# and col# and table#.

```
OPERATOR: <&cellx4>      ( row# col# table# group# --- addr )
```
Fetch the class instance address of the required matrix element at row# and col# and table# and group#.

3.12.3 <mat@> Operators

[class] = the defined class of the data represented in one cell. If the class is a simple numeric type then the data will be fetched on the appropriate stack. If the class is a complex type then the class <default> method will be used to point to the data of the cell.

```
OPERATOR: <mat@>         ( row# --- [class] )
```
Fetch class type for row# in matching class matrix instance.

```
Usage :   <mat@> [instance]                                  \ row# --- [class]
```

Can not be used with dot notation

```
OPERATOR: <matx2@>       ( row# col# --- [class] )
```
Fetch class type for row# and col# in matching class matrix table instance.

```
Usage :   <matx2@> [instance] \ row# col# --- [class]
```

Can not be used with dot notation

```
OPERATOR: <matx3@>       ( row# col# table# --- [class] )
```
Fetch class type for row# and col# and table# in matching class matrix table instance.

```
Usage :   <matx3@> [instance] \ row# col# table# --- [class]
```

Can not be used with dot notation

```
OPERATOR: <matx4@>       ( row# col# table# group# --- [class] )
```
Fetch class type for row# and col# and table# and group# in matching class matrix table instance.

```
Usage :   <matx4@> [instance] \ row# col# table# group# --- [class]
```

Can not be used with dot notation

4 Standard Types

The file *TypeDef.fth* contains a class library.

4.1 VALUE: types

`TYPE: VALUE:`

32bit signded constant value
Usage define - n VALUE: {instance name}

Usage runtime - {instance name}

`TYPE: WORD-VALUE:`

16bit unsignded constant value
Usage define - uw WORD-VALUE: {instance name}

Usage runtime - {instance name}

`TYPE: BYTE-VALUE:`

8bit unsignded constant value
Usage define - uc BYTE-VALUE: {instance name}

Usage runtime - {instance name}

4.2 Common class code

`: LOCAL-MAKE-INST runword ( type --- )`

ODL Compile erase of local object Object Class specific method - <default>

4.3 Unary Types

4.3.1 Byte types

`TYPE: C-VAR:`

8bit unsigned value.

`TYPE: <C-VAR: inherits C-VAR:`

8bit signed value

4.3.2 16 bit types

Primitives

`: TWISTW runword ( w --- w' )`

Toggle endian of 16bit value on the pstack.

`: TW-COUNT runword ( addr --- addr' u-cnt )`

Twisted WCOUNT

`: TWIST-WVAR runword ( addr --- )`

Toggle endian of 16bit value on the pstack.

```
TYPE: W-VAR:
```

4.3.3 Type Definitions

16bit unsigned value variable `W-VAR: members`

```
        C-VAR: x0   \ LSB
        C-VAR: x1   \ MSB
```

```
TYPE: W<VAR: inherits W-VAR:
```

16bit signed value

```
TYPE: WORD:
```

Windows API type WORD

4.4 L-VAR: 32Bit Signed Value Unary Object

```
: TWIST-L                runword                ( n --- n' )
```

4.4.1 L-VAR: Primitives

Toggle endian of 32bit value on the pstack.

```
: TWIST-LVAR             runword                ( addr --- )
```

Toggle endian of 32bit value at address.

```
TYPE: L-VAR:
```

4.4.2 L-VAR: Structure Definition

Structure for 32Bit Signed Value `L-VAR: members`

```
        W-VAR: x0  \ LSW
        W-VAR: x1  \ MSW
```

```
L-VAR: Methods
     Instance methods
     1.0 Modifiers
     1.1 Arithmatic
         <inc>                                                ( --- )
         <dec>                                                ( --- )
         <add>                                              ( n --- )
         <sub>                                              ( n --- )
     1.2 Assignment
         <set>                                                ( --- )
         <zero>                                               ( --- )
         <to>                                               ( n --- )
         <toggle>                                             ( --- )
         <invert>                                             ( --- )
         <twist>                                              ( --- )
         <erase>                                              ( --- )
         <init>                                               ( --- )
         <blank>                                             ( ---- )
     1.3 Sign
         <sign?>                                         ( --- flag )
         <abs>                                                ( --- )
         <negate>                                             ( --- )
         <neg>                                                ( --- )
     2.0 Inspectors
         <fetch>                                            ( --- n )
         <default>                                          ( --- n )
         <f@>                                               ( --- r )
         <m!>                                                 ( --- )
         <dbkey>                                              ( --- )
         <count>                                     ( --- addr cnt )
         <sizeof>                                           ( --- n )
         <addr>                                          ( --- addr )
         <ptr>                                           ( --- addr )
         <&ptr>                                          ( --- addr )
```

TYPE: INT:
Windows API type INT

TYPE: R-VAR: INHERITS L-VAR:
Structure for 32Bit Signed resetting Value

```
R-VAR: Methods
     See L-VAR: Methods
     <default>                                              ( --- n )
        sub classed to zero data value
```

4.5 32bit signed value and 8bit signed dpl

4.5.1 Primitives

: S@ RUNWORD (addr --- n dpl)
Fetch 32bit signed value and 8bit sign dpl at address.

: S! RUNWORD (n dpl addr ---)

Store 32bit signed value and 8bit sign dpl at address.

```
TYPE: S-VAR:                                                    ( --- )
```

32bit Value and signed dpl structure `S-VAR: members`

```
L-VAR:  val
<C-VAR: dpl
```

4.6 64bit signed value

4.6.1 Primitives

```
: D@            runword ( addr --- n0 n1 )
```

Fetch 64bit value at address

```
: D!            runword ( n0 n1 addr --- )
```

Store 64bit value at address

```
: TWISTD        runword ( n0\n1 --- n1'\n0' )
```

Toggle endian of 64bit value on the pstack.

```
: TWIST-DVAR    runword ( addr --- )
```

Toggle endian of 64bit value at address.

4.6.2 D-VAR: Structure Definition

```
TYPE: D-VAR:
```

Structure for 64bit signed value `D-VAR: members`

```
L-VAR: x0  \ LSL
L-VAR: x1  \ MSL
```

4.7 64bit signed value and 8bit signed dpl

4.7.1 Primitives

```
: B!            runword ( d dpl addr --- )
```

Store 64bit signed value and 8bit sign dpl at address.

```
: B@            runword ( addr --- d dpl )
```

Fetch 64bit signed value and 8bit sign dpl at address.

```
: TWISTB        runword ( d dpl ---  d' dpl )
```

Twist B-VAR on pstack.

4.7.2 Structure Definitions

```
TYPE: B-VAR:
```

Structure for 64bit signed value and 8bit signed dpl `B-VAR: members`

```
D-VAR:  val
<C-VAR: dpl
```

```
B-VAR: Methods
     Instance methods
     1.0 Modifiers
     1.1 Arithmatic
         <dec>                                                ( --- )
         <inc>                                                ( --- )
         <add>                                              ( b1 --- )
         <sub>                                              ( b1 --- )
     1.2 Assignment
         <twist>                                              ( --- )
         <to>                                               ( b1 --- )
     1.3 Sign
         <abs>                                                ( --- )
         <neg>                                                ( --- )
         <negate>                                             ( --- )
         <sign?>                                         ( --- flag )
     2.0 Inspectors
         <default>                                          ( --- b1 )
         <fetch>                                            ( --- b1 )
         <f@>                                               ( --- f1 )
         <count>                                     ( --- addr cnt )
         <sizeof>                                            ( --- n )
         <addr>                                         ( --- addr )
         <ptr>                                          ( --- addr )
         <&ptr>                                         ( --- addr )
```

```
: FIELD:           runword                                    ( n --- )
```

ODL Untyped field in structure definition.

4.8 Three way Database Intersection structure with Instances

`TYPE: J-VAR:`

Master and secondary accesion number intersection with many instances. One to many with instances. `J-VAR: members`

```
        L-VAR: macc#                                          \ master acc#
        L-VAR: sacc#                                       \ secondary acc#
        L-VAR: sacc2#                                      \ secondary acc#
        L-VAR: order#
```

4.9 Database Intersection structure with Instances

`TYPE: A-VAR:`

Master and secondary accesion number intersection with many instances. One to many with instances. `A-VAR: members`

```
        L-VAR: macc#                                          \ master acc#
        L-VAR: sacc#                                       \ secondary acc#
        L-VAR: order#
```

4.10 Database Intersection structure

`TYPE: I-VAR:`

Master and secondary accesion number intersection. Intersection dbase key variable. One to many. I-VAR: members

```
L-VAR: macc#                                          \ master acc#
L-VAR: sacc#                                        \ secondary acc#
```

4.11 Pointer Types

PTR: (POINTER):

Structure for 32Bit Unsigned long pointer to an instance of an object

```
(POINTER): Methods
     Instance methods
     1.0 Modifiers
     1.2 Assignment
         <ptr!>                                          ( addr --- )
         <ptr0>                                               ( --- )
     2.0 Inspectors
         <default>                                       ( --- addr )
         <ptr@>                                          ( --- addr )
         <&ptr>                                        ( --- instance )
```

```
: POINTER:              runword              ( [instance_name] {type:} --- )
```

Makes a pointer object of the type

```
POINTER: {name} [type:]
```

4.12 OBJID: 32Bit Unsigned unique identification Value

TYPE: OBJID:

Structure for 32Bit UnSigned unique identification value

```
OBJID: Methods
     Instance methods
     1.0 Modifiers
         <set>                                                ( --- )
            Increment the global unique identification value and assign
            this value to the object
         <to>                                                 ( n --- )
     2.0 Inspectors
         <default>                                            ( --- n )
         <fetch>                                              ( --- n )
         <count>                                        ( --- addr cnt )
         <sizeof>                                             ( --- n )
         <addr>                                          ( --- addr )
         <ptr>                                           ( --- addr )
         <&ptr>                                          ( --- addr )
```

4.13 Buffer type definition

TYPE: sADDRLEN:

Address and count structure `sADDRLEN: members`

```
L-VAR: address
L-VAR: length
```

4.14 Windows Standard Data Types

```
SYNONYM uint:       int:
SYNONYM lptstr:     int:
SYNONYM hmenu:      int:
SYNONYM hbitmap:    int:
SYNONYM handle:     int:
SYNONYM HWND:       int:
SYNONYM WPARAM:     int:
SYNONYM LPARAM:     int:
SYNONYM dword:      int:
SYNONYM long:       int:
SYNONYM char:       c-var:
SYNONYM byte:       c-var:
SYNONYM r-int:      r-var:
SYNONYM sword:      W<VAR:
```

`: ?PATH-ZFILE!  runword  ( addr cnt --- )`
Remove trailing backslash and blank chars from a zfilename path

4.15 String Primitives

`: $-<LEN>  runword  ( c-addr u-cnt --- u-cnt' )`
Trim string at addr cnt and return revised cnt

`: $-LEN  runword  ( c-addr u-cnt --- u-cnt' )`
Trim string at addr cnt and return revised cnt

`: $-NULL?  runword  ( c-addr u-cnt --- flag )`
Scan string at addr cnt and return true if oly blank char are found or false if any non blank chars is found.

`: $TO$  runword  ( source_addr source_cnt dest_addr dest_cnt --- )`
Assign dest_addr for dest_cnt to source_addr for source_cnt. The trailing postive difference between dest_cnt and source_cnt will be filled with blanks.

`: (SQUEEZE)  runword  ( c-addr u-cnt char --- c-addr u-cnt' )`
Move all characters of value char to end of string.

`: PARAGRAPH  runword  ( addr cnt --- addr cnt' )`
remove multple blank chars in situ

`: $-<PARAGRAPH>  runword  ( addr cnt --- )`
Remove extra blanks from text.

`: SQUEEZE  runword  ( c-addr u-cnt --- c-addr u-cnt' )`
Move blanks to end of string.

`: Z$CHAR+ RUNWORD  ( char zaddr --- )`
add char to end of zstring.

`: Z$CHAR- RUNWORD  ( zaddr --- char )`

remove and return char at end of zstring

```
: Z$-TO                    runword              ( saddr scnt c-addr u-cnt --- )
```
Given a source address and count and a target zero terminated string object. The target object u-cnt includes space for the terminator char. The buffer will always be zero terminated! Zero terminated string object Service definition.

```
: Z$-LEN                   runword                ( c-addr u-cnt --- u-cnt' )
```
Return address count of string or failing return count

```
: Z$-COUNT                runword            ( c-addr u-cnt --- c-addr u-cnt' )
```
Return address count of string or failing return count.

```
: Z$-LFT                   runword                ( u-cnt1 c-addr u-cnt2 --- )
```
Make the zero terminated string c-addr u-cnt2 u-cnt1 long preserving the current values to the left of u-cnt1 or fill with blanks if u-cnt1 less than the current zero terminated length of the string.

```
: Z$-RHT                    runword                ( u-cnt1 c-addr u-cnt2 --- )
```
Make the zero terminated string c-addr u-cnt2 u-cnt1 long preserving the current values to the right of u-cnt1 or fill with blanks if u-cnt1 less than the current zero terminated length of the string.

```
: Z$-CONCAT-             runword              ( s_addr s_cnt c-addr u-cnt --- )
```
Concat source addr and cnt to zero terminated string object

```
: Z$-<CONCAT>              runword              ( saddr scnt c-addr u-cnt --- )
```
Concat source addr and cnt to zero terminated string object.

```
: Z$-BLANK                                                ( c-addr u-cnt --- )
```
Blank zero terminated string object. Zero terminated string specific method definition.

```
: Z$-SQUEEZE               runword                        ( c-addr u-cnt --- )
```
Remove blanks from zero terminated string object.

```
: Z$-LOWER                                                ( c-addr u-cnt --- )
```
Lowercase zero terminated string object.

```
: Z$-UPPER                                                ( c-addr u-cnt --- )
```
Uppercase zero terminated string object.

```
: Z$-TRIM                  runword                        ( c-addr u-cnt --- )
```
Remove leading and trailing blanks from zero terminated string object.

```
: Z$-TRAILING                runword                      ( c-addr u-cnt --- )
```
Remove trailing blanks from zero terminated string object.

```
: Z$-LEADING                runword                       ( c-addr u-cnt --- )
```
Remove leading blanks from zero terminated string object.

```
: Z$-PATH?                 runword                  ( c-addr u-cnt --- flag )
```
Does the zero terminated string contain a file path.

```
: Z$-DRIVE?                runword                  ( c-addr u-cnt --- flag )
```
Does the zero terminated string contain a drive reference.

```
: Z$-<LEN>                                                ( c-addr --- u-cnt )
```
Return count of zero terminated object.

```
: Z$-<NULL?>            runword                          ( c-addr --- flag )
```

Return true if zero terminated object is null else false.

```
: Z$-<UNC?>                runword                              ( c-addr --- flag )
```
Return true if zero terminated string qualifies as a valid UNC path

```
: Z$-<-PATH>               runword                              ( obj obj_cnt --- )
```
remove path from file name Zero terminated file name object method

```
: NEW-INSTANCE             runword                          ( bpe #bytes obj --- )
```
ODL Create a new instance with #bytes and bpe Object Class specific method - <default>

```
: MAT-NEW-INSTANCE         runword                          ( bpe #bytes obj --- )
```
ODL Create a new instance with #bytes and bpe Object Class specific method - <default>

```
: NEW-ITEM                 runword                           ( #bytes obj --- )
```
ODL Make a new item of obj with #bytes size Object Class specific method - <default>

```
: ZFILENAME-<PATH>         runword                            ( c-addr --- )
```
Remove file name from file object leaving only the file path. Zero terminated file name object specific method - <path>

```
  DEFER CHARLOWER                                           ( addr cnt --- )
```
Locale sensitive lower case

```
  ASSIGN LOWER TO-DO CHARLOWER
```

```
   DEFER CHARUPPER                                          ( addr cnt --- )
```
Locale sensitive upper case

```
   ASSIGN UPPER TO-DO CHARUPPER
```

4.16 Zero terminated string classes

```
  Type: CLASSZSTR:
```
Class template for zero terminated string type: definitions. Use only when defining a new zero terminated string type:

```
CLASSZSTR: Methods
     Instance methods
     File Methods
       1.0 Modifiers
           <-ext>                                              ( --- )
           <+ext>                                        ( addr cnt --- )
           <squeeze>                                           ( --- )
       2.0 Inspectors
           <wild?>                                        ( --- flag )
           <path?>                                        ( --- flag )
           <drive?>                                       ( --- flag )
           <unc?>                                         ( --- flag )
     Standard String methods
```

```
  TYPE: CLASSZFILE:
```
Class template for zero terminated file name string type: definitions. Use only when defining a new zero terminated string type:

```
  TYPE: Z$-VAR: INHERITS CLASSZSTR:
```

Zero terminated string structure. Z$-VAR: requires a size on the parameter stack and therefore can only be used as a static instance or a structure member instance. Class defintions that require a size parameter stack input during execution can not be used as local instances. To use as a local instance, create a new type defintion using the class as a member instance and then use the new type def as the local instance.

```
Z$-VAR: Members
        classzstr: zstrdata
Z$-VAR: Usage
 Static instance - 8 Z$-var: zMyVar
 Member instance example
   TYPE: Mytype:
         24 z$-VAR: zMyvar
   END-TYPE
Z$-VAR: Methods
     Instance methods
     1.0 Modifiers
     2.0 Inspectors
         <default>                                          ( --- addr )
            default method has not been over loaded. The default
            instance method of a class is to supply the data address
            of the instance.
     See CLASSZSTR: for more methods
     Class Methods
         <default>                                          ( --- addr )
```

```
TYPE: zBUFFER: INHERITS CLASSZSTR:
```

MAX_PATH byte size zero terminated buffer. `zBUFFER: members`

```
       MAX_PATH Z$-VAR: zstrdata
```

```
zBUFFER: Methods
<count> method overloaded to use MAX_PATH
<to>    method overloaded to use MAX_PATH
```

4.17 96bit and 128 bit data

```
T-VAR-<FETCH>              runword                         ( t1 addr --- )
```

Triad variable store

```
: T-VAR-<FETCH>            runword                         ( addr --- t1 )
```

Triad variable fetch

```
TYPE: T-VAR:
```

Structure for 96bit signed value

```
T-VAR: Methods
     Instance methods
     1.0 Modifiers
     1.1 Arithmatic
         <add>                                                ( t1 --- )
         <inc>                                                  ( --- )
         <sub>                                                  ( --- )
     1.2 Assignment
     1.3 Sign
         <negate>                                               ( --- )
         <sign?>                                                ( --- )
     2.0 Inspectors
         <fetch>                                              ( --- t1 )
         <default>                                            ( --- t1 )
         <count>                                          ( --- addr cnt )
         <sizeof>                                            ( --- n )
         <addr>                                           ( --- addr )
         <ptr>                                            ( --- addr )
         <&ptr>                                           ( --- addr )
```

`: Q@` `runword` `( addr --- n0 n1 n2 n3 )`
Fetch 128bit value at addr. Store format [LSB.MSB] quad_fetch.

`: Q!` `runword` `( n0 n1 n2 n3 addr --- )`
Store 128bit value at addr. Store format [LSB.MSB] quad_store

`TYPE: Q-VAR:`
128bit signed value `Q-VAR: members`

```
L-VAR: x0
L-VAR: x1
L-VAR: x2
L-VAR: x3
```

4.18 Vector address types

`TYPE: VECTOR:`
32bit CFA `VECTOR: members`

```
L-VAR: addrcfa
```

4.19 Arrays of 8-Bit Unsigned Values

`TYPE: CHAR(3):`
Char array 3 long `CHAR(3): members`

```
C-VAR: x0
C-VAR: x1
C-VAR: x2
CLASS \ Class methods
INST  \ instance methods
:M <to> INST-COUNT CHAR(*)-[to] ;M  structure-method
```

`TYPE: CHAR(4):`
Char array 4 long `CHAR(3): members`

```
        C-VAR: x0
        C-VAR: x1
        C-VAR: x2
        C-VAR: x3
        CLASS \ Class methods
        INST  \ instance methods
       :M <to> INST-COUNT CHAR(*)-[to] ;M  structure-method
```

4.20 Export List

Ini API

```
  API-EXPORT (POINTER):
  API-EXPORT VALUE:
  API-EXPORT WORD-VALUE:
  API-EXPORT BYTE-VALUE:
  API-EXPORT C-VAR:
  API-EXPORT CHAR(3):
  API-EXPORT CHAR(4):
  API-EXPORT <C-VAR:
  API-EXPORT W-VAR:
  API-EXPORT W<VAR:
  API-EXPORT WORD:
  API-EXPORT L-VAR:
  API-EXPORT INT:
  API-EXPORT R-VAR:
  API-EXPORT S-VAR:
  API-EXPORT D-VAR:
  API-EXPORT B-VAR:
  API-EXPORT bytes:
  API-EXPORT cstring:
  API-EXPORT zstring:
  API-EXPORT (FIELD):
  API-EXPORT J-VAR:
  API-EXPORT A-VAR:
  API-EXPORT I-VAR:
  API-EXPORT sADDRLEN:
  API-EXPORT OBJID:
  API-EXPORT CLASSZSTR:
  API-EXPORT CLASSZFILE:
  API-EXPORT Z$-VAR:
  API-EXPORT T-VAR:
  API-EXPORT Q-VAR:
  API-EXPORT zBuf:
  API-EXPORT zBUFFER:
  API-EXPORT VECTOR:
  API-EXPORT uint:
  API-EXPORT lptstr:
  API-EXPORT hmenu:
  API-EXPORT hbitmap:
  API-EXPORT handle:
  API-EXPORT HWND:
```

```
API-EXPORT WPARAM:
API-EXPORT LPARAM:
API-EXPORT dword:
API-EXPORT long:
API-EXPORT char:
API-EXPORT byte:
API-EXPORT r-int:
API-EXPORT sword:
API-EXPORT Z$-<ANSI_UPPER>                                    ( obj --- )
API-EXPORT Z$-<ANSI_LOWER>                                    ( obj --- )
API-EXPORT Z$-REMOVE-EXTRA-BLANKS                           ( zstr> --- )
API-EXPORT CHARLOWER                                      ( addr cnt --- )
API-EXPORT CHARUPPER                                      ( addr cnt --- )
```

5 CCS System Extensions

5.1 System Wordlists

5.1.1 Definition management

One of the difficult tasks in software engineering is to manage multiple desperate programers, each with their own style and creative abilities. Naming conventions of definitions and the application interface contract are very problematic in a team enviroment. To limit the conflict between programmers and the misinterpretation of interface contracts the following rules apply:

- Defitions marked as '**' do not form part of the interface and can be changed without notice by the author.
- Definitions marked "***" form part of the 'system' interface and must have a correct stack picture comment and glossary comment. The author can make changes to the definition only with full audit and notification procedures.

Using predefined system wordlists allows for a clear understanding from the user of the definition and gives the user the ability to debug the code without knowing the intimate module construction of the author.

The Hidden wordlists are applicable to definitions that are potential system dependencies or extensions to the current system. Definitions that are possible name conflicts must be placed in a module. Interface defintion must be exposed via export functions and must have names that do not conflict with the underlying system. The export list must be published separately and can be renamed by the implementor to avoid name conflicts.

```
#256 BUFFER: Pocket     \ -- addr
```
Scratch buffer.

```
: LATEST-NFA    \ -- nfa
```
Point to the last defintion's Name Field

```
WORDLIST CONSTANT wid**
```
Two star temporary wordlist during compilitaion

```
WORDLIST CONSTANT wid***
```
Three star temporary wordlist during compilitaion

```
WORDLIST CONSTANT widHidden
```
System Hidden word list not to be used without consent from the owner.

5.1.2 Hidden wordlists and search order

```
: EXPOSE-**     \ --
```
Expose the ** wordlist.

```
: EXPOSE-***    \ --
```
Expose the *** wordlist.

```
: EXPOSE-HIDDEN \ --
```
Expose the hidden wordlist.

```
: [OLDFORTH]    \ --
```
Search order for running CCS system. [imm]

`: [FORTH]  \ --`
Set the search order for compiling CCS. [imm]

`: [PLUG]  \ --`
Set the search order for a plug. [imm]

`: [PRIVATE]  \ --`
Set the search order to make definitions Private. [imm]

`: [FUNCTIONS]  \ --`
Set the search order for compiling Windows functions. [imm]

`: **  ( --- )`
Compile time directive to move last definition from current to two_star private wordlist. When HIDE-** is executed all definitions in the two_star word list will be moved to the CCS hidden wordlist. [imm]

`: ***  ( --- )`
Compile time directive to move last definition from current wordlist to three_star private wordlist. When HIDE-*** is executed all definitions in the three_star word list will be moved to the hidden wordlist. [imm]

`: HIDE-**  \ --`
Compile time directive to hide all ** definitions.

`: HIDE-***  \ --`
Compile time directive to hide all *** definitions.

`: HIDE-WORD  \ {definition} --`
Place this word in the Hidden vocabulary.

`: name>forth  \ nfa --`
Move the given word to the `Forth` vocabulary.

`: last>forth  \ nfa --`
Move the last word found to the `Forth` vocabulary.

`: RUNWORD  \ --`
Used during compilation to unhide a previously hidden word.

`: API-EXPORT  \ {definition} --`
Export the next inline definition to FORTH vocabulary.

`: last>***  \ --`
Move the last word found to the *** vocabulary.

`: API-MODULE  \ {definition} --`
Export the next inline definition to *** vocabulary.

5.1.3 Wordlist Helpers

`: WID-PRINT  \ thread nfa -- flag`
Print definition while walking wordlist

`: WID-WORDS  \ wid --`
Print words in wid wordlist

`: **-(XT-FIND)  \ addr cnt --- xt flag|false`
Find the xt of the definition at addr cnt in the hidden wordlist two-stars

`: ***-(XT-FIND) \ addr cnt --- xt flag|false`

Find the xt of the definition at addr cnt in the hidden wordlist three-stars

```
: HIDDEN-(XT-FIND)      \ addr cnt --- xt flag|false
```

Find the xt of the definition at addr cnt in the hidden wordlist

5.2 Error and Abort Messages

5.2.1 Message numbers

```
 4         3         2         1
--------- --------- --------- ---------
32   28   24   20   16   12   8    4
0000 0000 0000 0000 0000 0000 0000 0000
```

```
VARIABLE NEXT-ERR **    \ -- addr
```

Global variable with next error number

```
: #CCS-ERROR    ( <name> --- n )
```

Define next error number. Error text can be accessed using CCS-ERR-MESSAGE

5.2.2 Static Buffers

```
CREATE Abort$   \ -- addr
```

Buffer for abort message

```
CREATE AbortFilename$   \ -- addr
```

Buffer for file name where error occured

```
CREATE AbortWordName$   \ -- addr
```

Buffer for FORTH word name where error occured

```
CREATE AbortCaption$    \ -- addr
```

Buffer for caption bar when error is displayed

```
CREATE AbortId$ \ -- addr
```

Buffer for error number message

```
CREATE bufException$    \ -- addr
```

Buffer for exception error message

```
CREATE Exception$       \ -- addr
```

Buffer for exception string

```
CREATE bufUserError$    \ -- addr
```

Buffer for user error message

5.2.3 Abort Buffer Access

```
: CLR-ABORT-FNAME$      ( --- )
```

Clear AbortFilename$ text buffer

```
: >ABORT-FNAME$ ( adr cnt --- )
```

Assign addr cnt to AbortFilename$ text buffer

```
: +ABORT-FNAME$ ( addr cnt --- )
```

Append string to AbortFilename$ text buffer

5.2.4 AbortWordName$ Text Buffer Access

```
: CLR-ABORT-FUNC$       ( --- )
```
Clear AbortWordName$ text buffer

```
: >ABORT-FUNC$  ( addr cnt --- )
```
Assign addr cnt to AbortWordName$

```
: +ABORTFUNC$   ( addr cnt --- )
```
Append string to AbortWordName$

5.2.5 Abort$ Text Buffer Access

```
: CLR-ABORT$    ( --- )
```
Clear Abort$ text buffer

```
: >ABORT$       ( adr cnt -- )
```
Assign addr cnt to Abort$ text buffer

```
: +ABORT$       ( adr cnt -- )
```
Append string to Abort$ text buffer

5.2.6 AbortCaption$ Text Buffer Access

```
: CLR-ABORT-CAPTION$    ( --- )
```
Clear AbortCaption$ text buffer

```
: >ABORTCAPTION$        ( adr cnt -- )
```
Assign addr cnt to AbortCaption$ text buffer

5.2.7 AbortId$ Buffer Access

```
: CLR-ABORT-ID$         ( --- )
```
Clear AbortId$

```
: >ABORT-ID$    ( adr cnt -- )
```
Assign addr cnt to AbortId$

```
: +ABORT-ID$    ( addr cnt --- )
```
Add string at addr cnt to AbortId$ buffer.

5.2.8 Exception$ Buffer Access

```
: CLR-EXCEPT$   ( --- )
```
Clear Exception$

```
: CLR-EXCEPT-BUF$       ( --- )
```
Clear bufException$

5.2.9 User Error Message Buffer Access

```
: CLR-USER-BUF$ ( --- )
```
Clear user error buffer

5.2.10 General Abort and Error Buffer Access

```
: CLR-ABORT-BUF ( --- )
```
Clear all abort message buffers

```
DEFER DISP-ABORT        \ --
```
The error display vector

```
: CONSOLE-DISP-ABORT    ( --- )
```
Print the ERROR buffers to the current I/O console

5.3 Guard the Parameter stack

`8 cells constant /guard \ -- n`
The amount by which the stack is guarded.

`: [GuardSP \ -- ; R: -- oldS0 oldSP`
Reserves guard space on the data stack and resets the data stack pointers as if the stack was empty.

`: GuardSP] \ -- ; R: oldS0 oldSP --`
Restore the stack condition saved by `[GuardSP`.

5.4 Literal Lists

`: [LIT-LIST \ --- hival`
Open Literal list. Must use LIT-LIST] to close list. [imm]

`: LIT-LIST] \ hival --- | n0....nx x`
Close literal list. [imm]

5.5 Miscellaneous

`: (COMP-INLINE-STRUCTURE) \ size -- addr ; -- addr cnt`
Compile time helper definition for inline structures. Returns address of the structure during compile time. Returns the address and count of the structure in the image during run time.

`: XT>NFA ( xt --- nfa )`
Convert xt to name field address. If non FORTH xt then return pointer to INVALID-XT$.

`: XT>NFA \ xt --- nfa`
Convert xt to name field address. If non FORTH xt then return pointer to INVALID-XT$.

6 CCS String functions

The CCS application requires a large number of string functions which are not present in most Forth systems. The file *StringCCS.fth* contains these words.

`code scan \ caddr u char -- caddr2 u2`
This version does NOT handle tabs.

`code SKIP \ c-addr u char -- 'c-addr 'u MPE.0000`
This version does NOT handle tabs.

`: -LEADING ( c-addr1 u-cnt1 --- c-addr2 u-cnt2 )`
Modify a string address/length pair to ignore any leading space

`: -TRAILING \ c-addr u1 -- c-addr u2 17.6.1.0170`
Modify a string address/length pair to ignore any trailing space

`: $-TRIM ( c-addr u-cnt --- c-addr' u-cnt' )`
Trim string at addr cnt and return revised addr and cnt.

`: LWC ( char --- char' )`
ASCII Lowercase char on stack

`: LOWER ( c-addr u-cnt --- )`
Lowercase string at address cnt. Operates only on character values ASCII A to ASCII Z

`: STR-END= ( s_addr s_cnt t_addr t_cnt --- bool )`
Compare sub string address count with the end sub string of the target string for the sub string count.

`: STR-START= ( s_addr s_cnt t_addr t_cnt --- bool )`
Compare sub string address count with the start sub string of the target string for the sub string count.

`: WPLACE ( c-addr1 u c-addr2 --- )`
Copy the string described by c-addr1 u to a wcounted string at the memory address described by c-addr2.

`: LAPPEND \ c-addr u $dest --`
Add string c-addr u to the end of 32-bit counted string $dest.

`: LPLACE \ c-addr1 u c-addr2 --`
Copy the string described by c-addr1 u to a 32-bit counted string at the memory address described by c-addr2.

`: -BYTE= ( c-addr u-cnt char --- addr|false )`
Find first occurence of a byte value equal to char. Searches for char from addr-1 do not xam char at addr. minus_byte_equal

`: BYTE= ( c-addr u-cnt char --- addr|false )`
Find first occurence of a byte value equal to char. byte_greater_than

`: POS-JMP ( addr cnt sub_addr sub_cnt step --- addr|false )`
Return the start addr of sub_addr sub_cnt in addr cnt advancing the match with step. Return false if not found. pos_jump

`: REMOVE-XTRA-BLANKS \ addr cnt --- addr' cnt'`
Remove repeating xtra blanks, leaving at least 1 blank, from single byte string at addr cnt return result addr cnt within the original buffer space

```
: REPLACE-CHAR                                    ( addr cnt new_char old_char --- )
```
Replace occurrences of old char with new char in string.

```
: ISOLATE-STR RUNWORD     ( c-add u-cnt start_char end_char --- c-addr2 u-cnt2 )
```
Return the address and count of the first sub string in string at address count.

```
: N>$                                                          ( n --- addr cnt )
```
Convert n to decimal string

```
: ZSTRBUF$   ( zstr --- addr u )
```
Return buffer start address and the size of the buffer in number of character units

```
: ZSTRCHARS  ( zstr --- u )
```
Return number of characters units stored at zstring

```
: ZSTRCHAR#>ADDR   ( zstr index# --- addr true|false )
```
Convert character index number to character address. Index# is an option base 1 character position.

```
: ZSTREND-  ( zstr --- )
```
Remove last character from zero terminated string

```
: ZSTR-REMOVE-EXTRA-BLANK                                      ( zstr -- bool )
```
Scan zero terminated string for first occurance of repaeting blanks. Remove the second blank and return true. If no repeating blanks was found return false.

```
: ZSTR-#BLANKS-LEADING                                         ( zstr --- u )
```
Return number of leading blanks at zstr

Number of words in a zero terminated text string

```
: ZSTR-#WORDS                                                ( zstr --- #words )
```
Return number of words at zstr.

```
: ZSTR>-GET-MACRO                                  ( zmacro> zbuf> --- err|false )
```
Get the current value of the text macro. A valid existing text macro could have a leading and trailing % character.

Index

>

?

[

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

V

W

X

Z