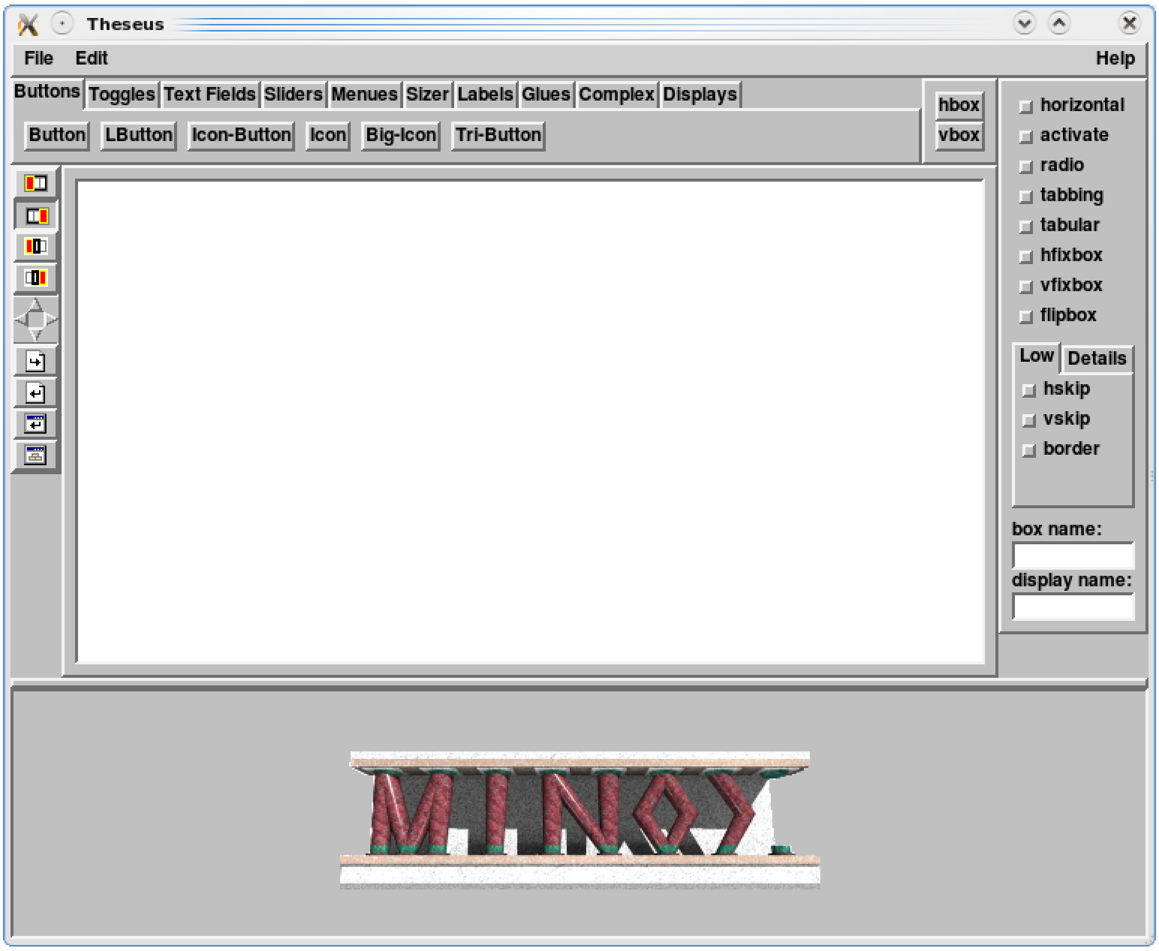
Minos and Theseus

GUI Display and Designer in Forth



Bernd Paysan & Stephen Pelc

Published by Microprocessor Engineering

Minos and Theseus
User manual
Manual revision 4.4
31 August 2009

Software
Software version 4.40

For technical support
Please contact your supplier

For further information:
Bernd Paysan
email: bernd.paysan@gmx.de
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 **Licensing and other matters** 1

1.1 Copyright 1
1.2 License terms 1
1.2.1 Default terms 1
1.2.2 VFX Forth 1
1.3 LGPL exemption, support and upgrades 1

2 **Theseus GUI designer** 3

2.1 Introducing Theseus 3
2.2 Compiling Minos and Theseus 4
2.3 Launching Theseus 4
2.4 Calculator example 6
2.5 Concepts for menus 8
2.6 Menu example 9
2.6.1 Top level menu 9
2.6.2 Adding second level menus 10
2.6.3 Reviewing the design 11
2.6.4 Making it pretty 11
2.7 Adding a menu without a panel 11
2.8 Navigating Theseus 11
2.8.1 At the top 11
2.8.2 Editing modes 11
2.8.3 Dialog editor 12
2.8.4 Box creator 13
2.8.5 Box inspector 13
2.8.6 Object inspector 14
2.9 Adding actions 14

3 **BerndOOF Object Oriented Forth** 15

3.1 What is Object Oriented Programming? 15
3.1.1 The Class Concept 15
3.1.2 Binding: Late or Early? 17
3.1.3 Objects as Instance Variables 18
3.1.4 Tools and Application Examples 22
3.2 The Complete BerndOOF Description 23
3.3 Formal Syntax 27
3.4 Porting the OOP packages 27

4 **Portability layer for VFX Forth** 29

4.1 Blocks 29
4.2 Floating point 29
4.3 Floored division 30
4.4 Miscellaneous math 30
4.5 Headerless words 30
4.6 Memory access with address increment 31
4.7 Memory allocation 31
4.8 Overrides 32

4.9 Miscellaneous tools 32
4.10 Interpretive structures 35
4.11 Application startup chains 35
4.12 MODULE from bigFORTH 36
4.13 Operating system dependencies 36

Index 37

1 Licensing and other matters

1.1 Copyright

The code for Minos and Theseus was written by, and is maintained by, Bernd Paysan. The copyright of the code is held by Bernd Paysan for all versions.

1.2 License terms

1.2.1 Default terms

Unless otherwise ngotiated, Minos and Theseus are released under the terms of the GNU GENERAL PUBLIC LICENSE v3 contained in the file GPLv3 in the main *Minos* folder.

1.2.2 VFX Forth

The effect of this license is that you do not have to provide the code in the *Minos* folder with your application. However, should anyone ask for it, you **must** supply it. You can, however, purchase an LGPL exemption, technical support, and upgrades.

1.3 LGPL exemption, support and upgrades

For details please contact Bernd Paysan or MicroProcessor Engineering Ltd. The contact details are at the front of this manual.

2 Theseus GUI designer

2.1 Introducing Theseus

Minos and Theseus are the answer to the question "Is there something like Visual BASIC or Delphi in Forth?". Basically, these GUI systems contain two components: a library with a wide variety of elements for drawing a graphical user interface; e.g. windows, buttons, edit-controls, drawing areas, etc. and an editor to select and combine the elements with the mouse by drag-and-drop or click-and-point actions. You then insert code that acts when buttons are pressed.

It isn't really necessary to paint graphical user interfaces together, in the same way as it isn't for text, e.g. TeX versus Open Office. Many typesetting functions are more semantic than visual, e.g. a piece of text is a headline or emphasized instead of written in bold 18 point Garamond or 11 point Roman italic. All this is true for user interface design too, and to some extent much more. It's not the programmer who decides which font and size to use for the User Interface - that's up to the user, as are the colour of buttons and texts.

To lay out individual widgets, more abstraction than defining position, width and height makes sense. Typically buttons are arranged horizontally or vertically, perhaps with some space between them. The size of buttons must follow the containing strings, and should conform to a consistent aesthetic, e.g. each button in a row has the same width.

Such an abstract model, related to TeX's boxes and glues model, performs quite well even without a visual editor. The programmer isn't responsible for typesetting the buttons and boxes. This approach is quite usual in Unix. Motif and Tcl/Tk use neighborhood relations, Interviews uses boxes and glues. I decided for boxes and glues, since it's a fast and intuitive solution, although it needs more objects to get the same result. Boxes form the components. The space between them is defined by glues.

These concepts contradict somehow with a graphical editing process, since the editors I know don't provide abstract concepts such as "place left of an object" or place in a row, but positions.

Theseus works with a relationship model. It does not work in terms of the x,y,w,h positioning model familiar to Windows programmers. Instead you specify the relationship between items in Theseus, and then Minos displays the items.

Theseus is designed to create Minos dialogs. Each dialog is derived from a window class. You can create dialogs by composing boxes (layout managers) and widgets together, add actions to the widgets, add variables and methods to the derived window classes and so on.

Minos has four sets of classes:

- **Widgets**: "window gadgets", these are the basic objects like buttons, labels, icons and text fields.
- **Displays**: these widgets are the drawing area of other widgets. They know how to draw to X or to the parent display. It is also possible to create new displays that, e.g., draw into an OpenGL widget or send drawing events over the network, to copy drawing messages to more than one parent display and so on.

- **Boxes**: or layout managers. Although each object in Minos has coordinates and size, formating is done by a surrounding layout manager. These either form horizontal boxes, and align their contents from left to right, or vertical boxes, and align their contents from top to bottom. Each widget has a horizontal and a vertical glue, which are used to compute the look of the widgets.
- **Actions**: these are the objects (Forth words) that link data and widgets together. An action knows in which state it is, and what to do when the state changes. There are several types of actions, for simple buttons, toggle buttons, text fields, and sliders.

Each dialog is hierarchically composed from widgets, boxes, and displays (e.g. viewports). Each active object, thus each button, toggle button, slider, and text field has an associated action, a name, and other attributes.

2.2 Compiling Minos and Theseus

If you have not been supplied with a precompiled version of Theseus or you have upgraded the Forth kernel, you will have to rebuild Theseus. The incantations are specific to the host Forth. To run Theseus as it is is designed to be run, you should create an executable with a precompiled version of Minos. Note that this may modify the start-up behaviour and command-line processing of your Forth.

For bigForth, build *xbigforth*:

```
bigforth
 include startx.fs
```

For VFX Forth, build *xvfxlin*:

```
vfxlin
 include startx.fs
 save xvfxlin
```

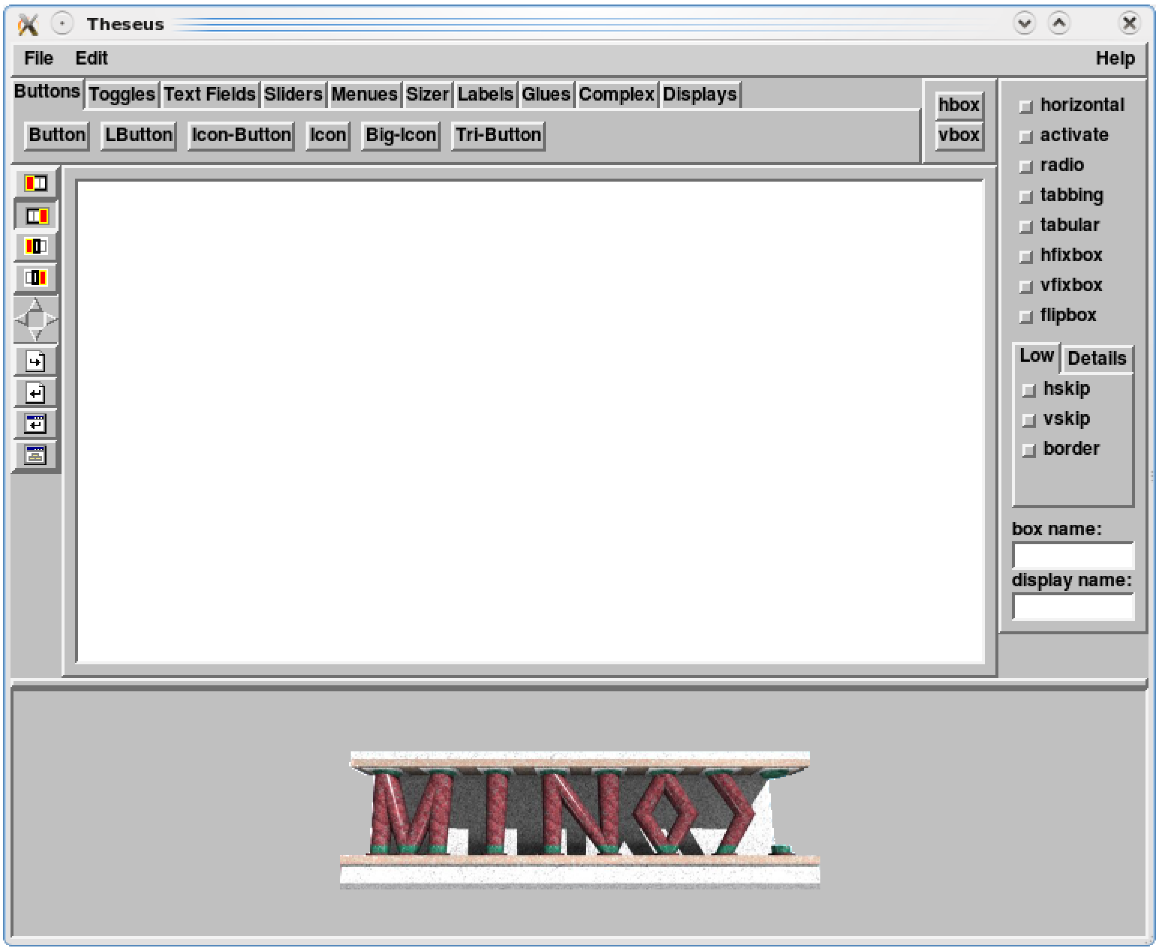
You can now run the binary, which will appear in a window. We usually refer to this binary as <*xbin*>.

2.3 Launching Theseus

In most cases, Theseus is compiled from source code every time - it only takes a fraction of a second. Theseus will start when the code has compiled.

```
<xbin> theseus.fs
```

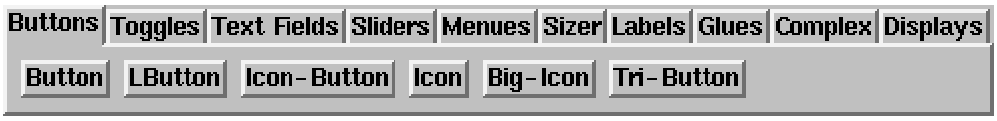
This will leave you with the following display.



You start a new design using the top menu bar.

The file menu allows you to load and save designs. The edit menu has only two entries, for a new dialog and a new menu window.

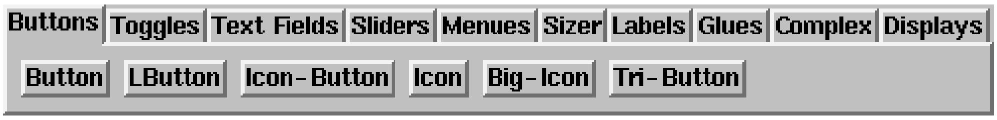
Once you have selected a menu or dialog to buid, you can then select what you are going to put in it using the widget bar below the menu.



The widget bar contains groups of widgets. To insert a widget into a dialog, select the group and click on a button. The corresponding widget is inserted in the current position (affected by the mode switches).

Rather than go into tedious detail about every button, we are now going to take you step by step through the process of building a simple calculator. After that, we'll build a menu system.

2.4 Calculator example

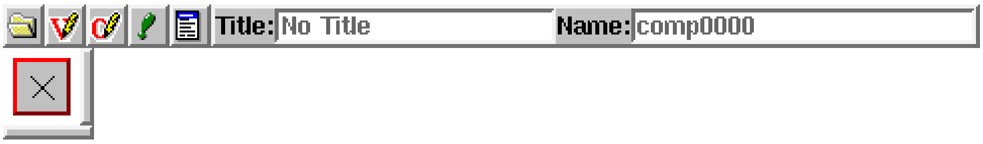


The picture above shows what we are going to build. There's a canned example *calc.m* if you want to cheat. It really is worth going through this example yourself.

Formatting buttons and text fields is done by the system, and is thus not the task of the programmer (you), who only has to specify the logical arrangement.

The project is hierarchically arranged. The topmost hierarchy are the dialog windows. These windows understand two additional methods, open and modal-open which allows you to create both non-modal and modal dialogs. The user then creates a framework of horizontal and vertical boxes inside the dialog. These boxes are filled with contents and glues. Glues are what we call the space between items. Glue can be modified (by you) too.

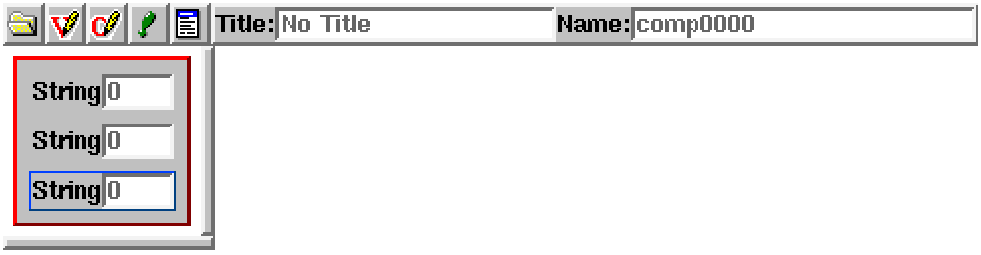
Launch Theseus. Create a "New Dialog" with the "Edit" menu. This is the dialog bar at the start of the project:



Title: No Title
Name: comp0000

Each object has a title and a name. The title is usually displayed at the top of the object as a caption. If the title field shows "No Title", there will be no caption. The name is the name of the Forth object that will be generated when Theseus generates code for you. For the moment, we will not bother much with titles or names, but they will be important when we get to the menu example.

Next we need three entries that contain a label and an editable number. Click in the box in the new dialog that has a cross in it. The cross indicates an empty area. Select "Text Fields" in the widget bar, and then click "Infonumber" three times. You should see:



Title: No Title
Name: comp0000

String 0
String 0
String 0

When you select each field in turn, you will see that the box inspector has replaced the Minos logo. Edit three of the four fields

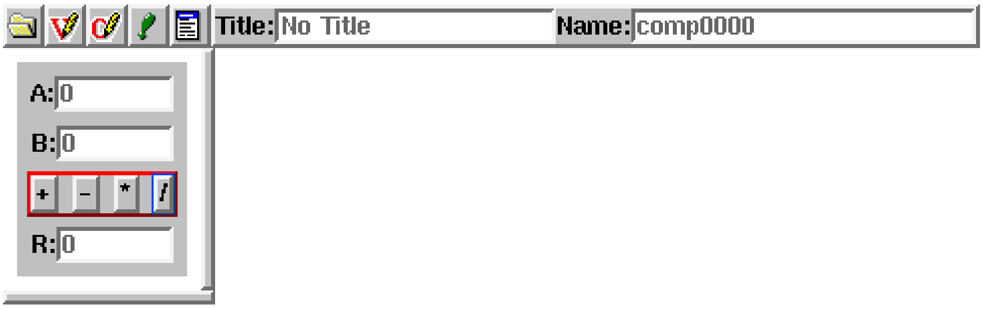
- **Infotextfield name** - this is the name (in Forth) of the object. We need the name in order to get at the data in the number field. Call these `a#` and `b#` for the inputs, and `r#` for the result.

- **Number** - the initial value that will appear in the edit box.
- **string** - this is the label at the left of each entry. Suitable strings are *A:*, *B:* and *R:*.

Beneath the two input fields the operation buttons should be arranged one aside each other. A horizontal box (hbox) does the job, with four buttons in it.

1. First click to select the second input box, the one labelled B.
2. At the left of the main Theseus window, hover the mouse over the top four icons in the vertical bar. Tooltips will appear to show their action. Select the one whose tip says "Add after current object".
3. Select hbox (near upper right) to make the box.
4. Click the "Add last in box" icon.
5. Click into the new box and click off the **vskip** button on the right hand side of the main Theseus window.
6. Click "Buttons" on the widget bar, and then click "Button" on the next line four times to put the buttons in the hbox.
7. Select each button in turn and label it for one of the primary operations +, -, *, /. After selecting it, you can press <Esc> to clear the text field first and then just type the text.

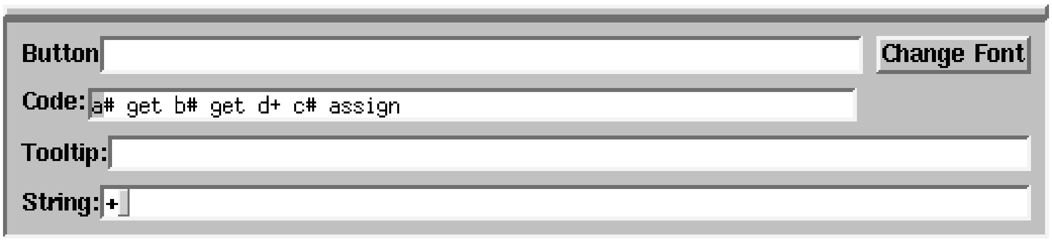
Now you should see something like this:


Title: No Title
Name: comp0000
A: 0
B: 0
R: 0

You have already given object names to the input and result boxes. Now we must add the action code in the **Code:** field of each button. The code looks as follows, for +, -, *, and /.

```
a# get  b# get  d+  r# assign
a# get  b# get  d-  r# assign
a# get  b# get  d*  r# assign
a# get  b# get  drop ud/mod  r# assign  drop
```

We will discuss the gory details of accessing object data later. For the moment, we'll just note that `get` and `assign` are methods applied to the objects `a#`, `b#` and `r#`. You should have a button inspector display like the one below.



Button:

Code: a# get b# get d+ c# assign

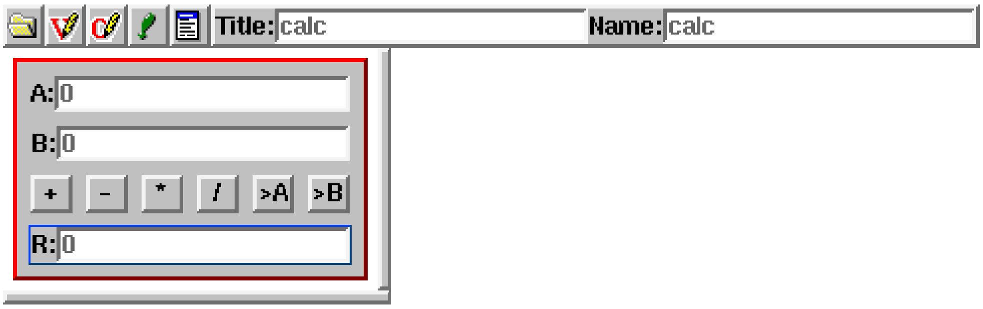
Tooltip:

String: +

It may be useful to take the result and copy it to one of the input buttons for reuse. Thus two additional buttons are required, and to make it nice, all buttons should have the same size (with "tabbing" box style). The dialog should have a title, and a name, too. The added code for the `>A` and `>B` buttons is:

```
r# get  a# assign
r# get  b# assign
```

Now you should see something like this:



Title: calc Name: calc

A: 0

B: 0

R: 0

Save the application. You must provide the extension for the file name. By convention, Theseus uses a *.m* extension for its auto-generated files.

If your particular host Forth permits it, you can click the **Run application** button and make your calculator do some calculations.

2.5 Concepts for menus

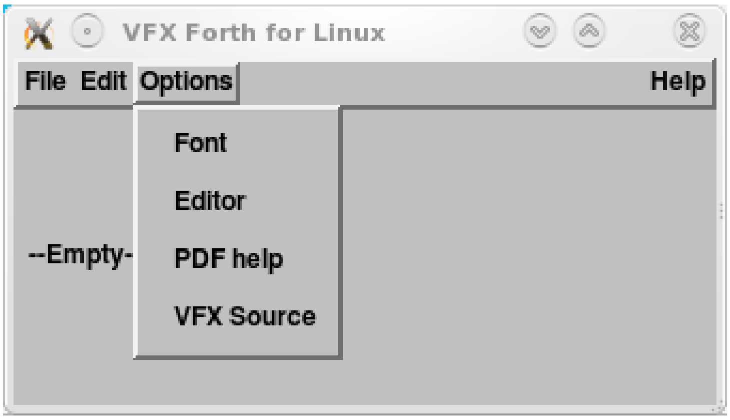
Remember that Minos builds things from boxes and glues. A menu is just a collection of labelled boxes that do things when pressed. In Minos terms, a top-level menu entry that just runs another menu is a **menu-title**. An entry that actually does something is a **menu-entry**. A menu entry (not at the top level) that launches another menu is a `sub-menu`.

A top-level menu is built either as a menu bar using an dialog, or as a "New menu window", which is an `box` containing menu items and a panel for the application's display. All other menus are built with dialogs containg menu items.

To link them together, the **menu-title**s point to the dialogs containg menu items and so on. Note that initially you only want to see the top-level menu, so all the dialogs for sub-menus have their "Show dialog" buttons (see the left-hand buttons on each dialog editor) set to "not shown" (red cross rather than green mark). This prevents all the menus being visible at application start-up.

2.6 Menu example

The example here was generated during the port of Minos and Theseus to VFX Forth for Linux.



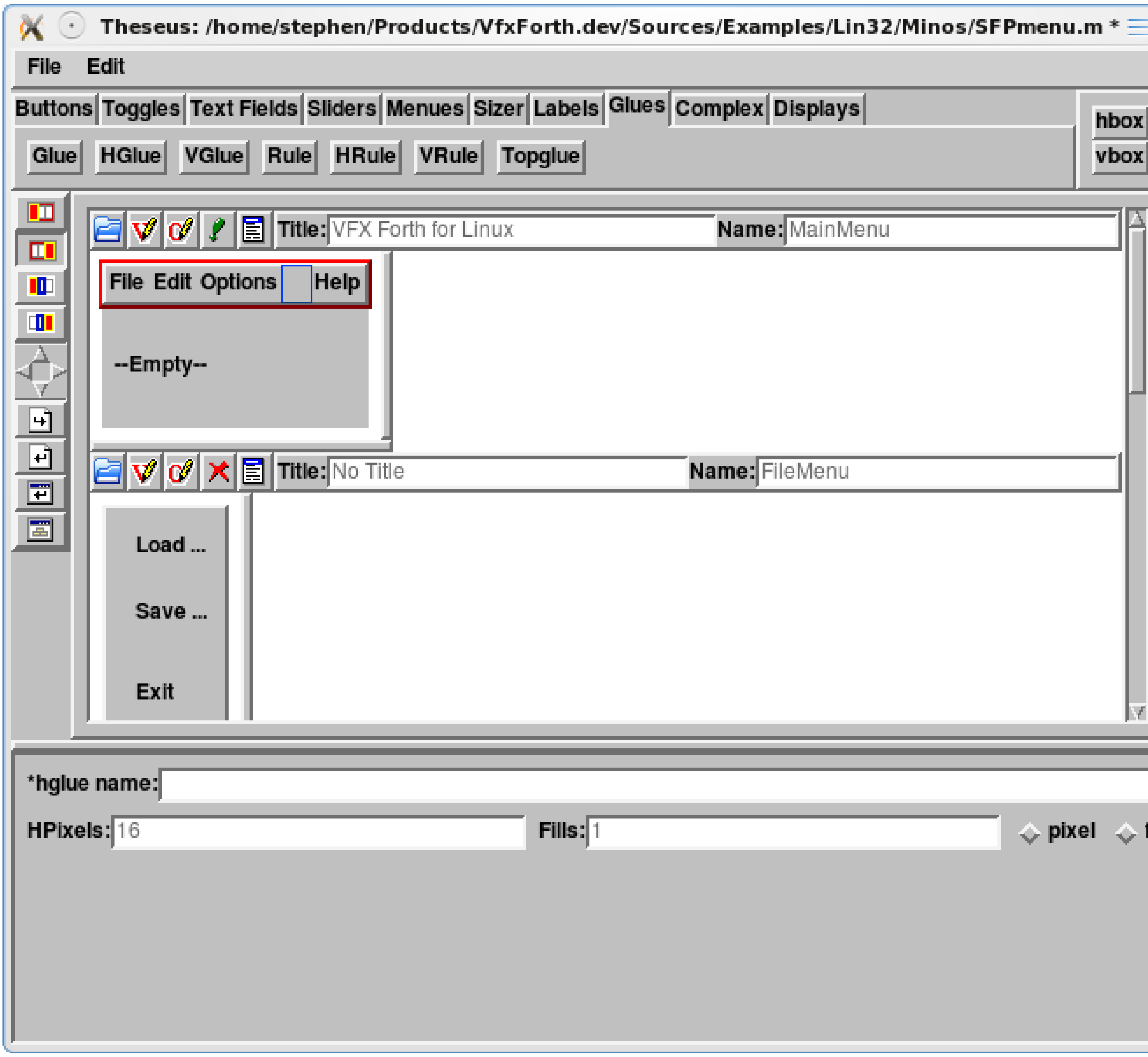
2.6.1 Top level menu

Start Theseus and then

1. Select **Add last in box** (the default) and
2. Add a **New Menu Window** from the **Edit** menu.
3. Select the box portion at the top, and click the **border** button to make the menu more distinct.
4. In the dialog bar (a menu window is a special case of a dialog), give the dialog a title (caption), and a name for the Forth object. We used `MainMenu`, because that's what we were prototyping then.

You will see a horizontal box at the top and an empty panel below it. The box will contain the top-level menu, and the panel will contain the other parts of the application. We are now ready to build the top-level menu. The picture below shows the Theseus display midway through the process.

1. Select **Menus** on the widget bar, and add three **Menu-Title**s.
2. Select **Glues** on the widget bar, and add one **HGlue**
3. Go back to **Menus** and add another **Menu-Title**.
4. Select each menu entry in turn and edit the **Menu-Title name** and **String** fields. The names are the names of Forth objects. Since this is the main menu and the first entry is for a File menu, we name it `mmFile` and enter **File** as the string. We will edit the **Menu** field when we have built the File menu itself.
5. Select the **HGlue** by selecting the gap between the third and fourth menu titles. Check the **fill** button to keep the **Help** menu at the right. If you don't like this layout you can remove the **HGlue** by selecting it with a shift-left-mouse-button click.



Theseus: /home/stephen/Products/VfxForth.dev/Sources/Examples/Lin32/Minos/SFPmenu.m *

Title: VFX Forth for Linux Name: MainMenu

Title: No Title Name: FileMenu

*hglue name:

HPixels: 16 Fills: 1

2.6.2 Adding second level menus

Next, we need to add the four menus. These are vertically oriented dialogs containing **Menu-Entry** components. The process is as follows.

1. Create a new dialog. It should not have a title, but it needs a name. Again this is a Forth object. For the File menu, we called it `FileMenu`.
2. Give the dialog a border and change the **Show Dialog** button on the dialog bar to a red cross. This stops the dialog being displayed until you actually select it.
3. Add your **Menu-Entry** items. Each one needs a Forth name in the **Menu-Entry name** field and some text in the **String** field.
4. To make the menu entry do something when selected, you can put a code fragment in the `Code` field. For the moment leave this field blank. See the section "Adding actions" below for more details.

2.6.3 Reviewing the design

Click the **Run application** button and press the menu entries. The menus appear as they will appear in your final application.

At first glance the boxes are too deep - there's far too much white space, and the items could be grouped by the use of separators. We'll do this in the next section.

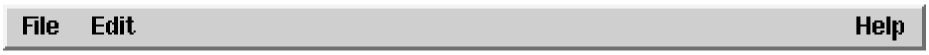
After reviewing a design, remember to close the test design otherwise you'll find the screen cluttered with examples!

2.6.4 Making it pretty

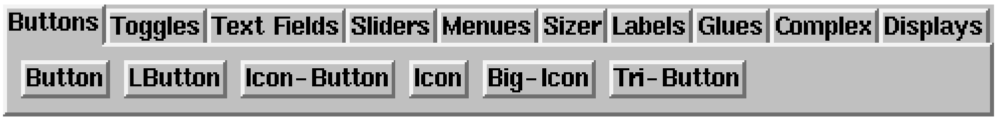
2.7 Adding a menu without a panel

2.8 Navigating Theseus

2.8.1 At the top

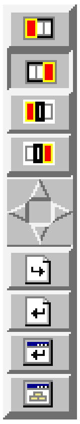


The menu bar contains a file menu, an edit menu and a help menu. From the **Edit** menu, you select a new dialog or menu.



The widget bar contains groups of widgets. To insert a widget into a dialog, select the group and click on a button. The corresponding widget is inserted in the current position (affected by the mode switches).

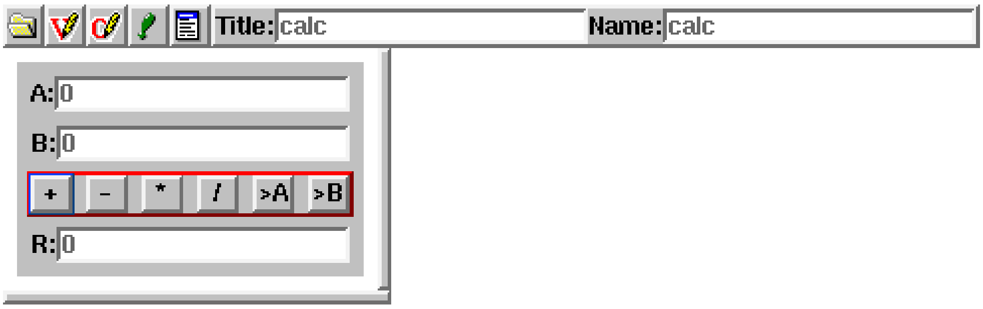
2.8.2 Editing modes



The editing modes are separated into three groups:

1. Insert modes:
   - Add object first in current box
   - Add object last in current box
   - Add object before current box
   - Add object after current box
2. Navigation through nested boxes, sets the active box. The reason why there is a navigator is that sometimes boxes are unselectable because they have no border and the inside boxes fill them completely, leaving no outer space to select with a mouse click.
   - Move one box up in hierarchy
   - Move one box to the left/up
   - Move one box to the right/down
   - Move to the first child
3. Short cuts:
   - Load dialog
   - Save dialog
   - Try dialog
   - Save as module

2.8.3 Dialog editor


Title: calc
Name: calc
A: 0
B: 0
R: 0

The dialog editor shows a navigation bar for each dialog to edit. The dialog itself is resizable with the split bar below and on the side, to see how it looks resized.

The navigation bar consists of an icon to open/hide the dialog, to open/hide the declarations field, to open/hide the code field, to select whether to show the dialog, a dialog menu, a dialog title and a dialog name. It is important to give every dialog a name, since dialogs without names can't be saved. The dialog name is the name of the derived class, you can refer to this class in your code.

The declaration field contains variable and method declarations of the dialog class. The code field contains method definitions. The dialog edit field contains the dialog itself.

Use the mouse for selection.

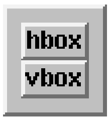
- [Edit] A simple mouse click opens the inspector of that object, with focus on the text/code/name field (left/middle/right mouse button).
- [Cut&Paste] Shift-mouse} clicking left cuts the object to the cut stack, clicking middle or right pastes from the cut stack.

- [Try] Cintrol-mouse clicking makes the object react as in the dialog, but without executing code.

You can delete a complete dialog using **Dialog Menu > Cut Dialog** in the dialog's navigation bar.

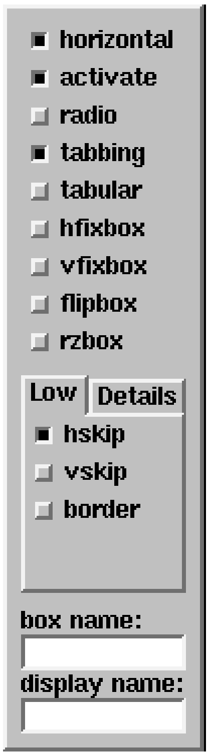
When you are happy with a dialog, you can fold it to consume less Theseus display space using the **Dialog Editor** button. Clicking this button again will unfold (restore) the view.

2.8.4 Box creator



The box creator has two buttons to create horizontal and vertical boxes. Boxes are simple layout managers, that arrange containing objects one after the other. Boxes are created as normal objects, so they go to the same places where a normal object would go. They inherit the settings of the parent object, so these settings have to be changed using the box inspector.

2.8.5 Box inspector



☒ horizontal
☒ activate
☐ radio
☒ tabbing
☐ tabular
☐ hfixbox
☐ vfixbox
☐ flipbox
☐ rzbox

☒ hskip
☐ vskip
☐ border

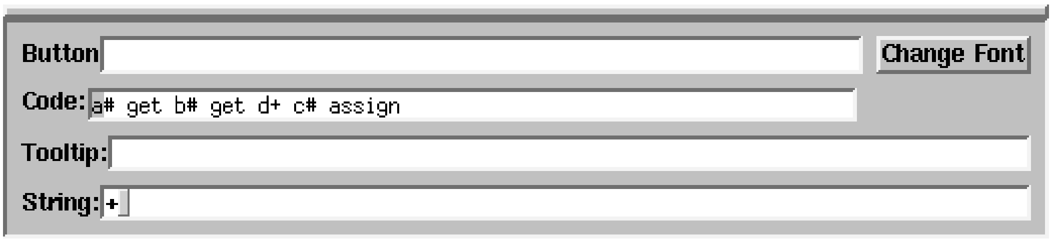
box name:
display name:

The properties of the current box can be changed in the box inspector on the right hand side of the Theseus screen:

- The **horizontal** switch changes the direction of the box
- The **activate** switch changes the selection behavior: active boxes contain one single active object, navigation with <tab> is possible.

- The **radio** switch enables deselection on click, thus only one switch inside such a box may be active at a time.
- The **tabbing** switch changes the layout: all objects except glues in tabbed boxes have the same size.
- The **hfixbox** shrinks the box to the minimal size, no growing
- The **vfixbox** shrinks the box to the minimal size, no growing is possible in vertical direction
- The **flipbox** hides the box when active
- The **hskip** box or slider (with **Details** activated) adds horizontal skips between objects
- The **vskip** box or slider (with **Details** activated) adds vertical skips between objects
- The **border** box or slider (with **Details** activated) adds a shadow to the box (raised or sunken).

2.8.6 Object inspector



Code: `a# get b# get d+ c# assign`

String: `+`

The object inspector contains the informations of the current object. The fields depend on the class of the object, however, some fields are common between objects.

- The **name** field selects the name of the object, this name is used in code to refer to this object
- The **string** field is the string the object displays
- The **code** field is the code that is executed on clicks
- The **tooltip** field is the tooltip that is shown when the mouse is over the object (an empty string means no tooltip)
  There are other fields for properties specific to each type of widget.

2.9 Adding actions

3 BerndOOF Object Oriented Forth

The OOP package documented here here was written by Bernd Paysan for his *bigforth* system. Since then it has been ported to other Forth systems. The package is now known as *BerndOOF* for short. The source code is in the file *oof.fs*. Minos and Theseus rely on *BerndOOF*.

3.1 What is Object Oriented Programming?

The buzzword of the late 80s and 90s in the IT industry was without doubt "object oriented". No operating system, no application, and certainly no programming language, that isn't object oriented. Forth isn't excluded as publications like Dick Pountain's *Object-Oriented FORTH* show clearly.

Ewald Rieger had ported Pountain's OOF to bigFORTH and gave it to me on the Forth-Tagung in 1991 to look at it. Since this OOF lacked several features, I completely rewrote it, to make this interesting programming paradigma available for bigFORTH. This system is in use since 1992, and has proved to be useful even in the rough world of real-time programming. Ewald Rieger uses it to control an automatic chromatography system.

The encapsulation of data and algorithm to form an object has proven useful, more proven essential, especially for changing hardware configurations, as they are typical for many tasks in practise.

To give you an impression of what is possible with object oriented programming, and how to do it, the following is an introduction using a small example. The sources are in the file *oofsampl.fth*. I use a small collection of data types that are known: integers, lists, arrays, and pointers. Object oriented programming hides behind a jargon that, after a closer look,has some similatities with concepts like modularity and definition of clean interfaces.

3.1.1 The Class Concept

The core principle of object oriented programming is to encapsulate data and the procedures that operate on the data into an object. Ideally the procedures (**methods**) that access an object are the only way to operate on the data. The interface to the object then consists of the names of the methods (**messages**) and the parameters that are sent with the message. Since many objects return results, the stack is used conventionally for the **message passing** and the method is called like a Forth word - with a detour over the object, that manages the encapsulation.

It would be a waste to program each method for each single object; especially since many objects have the same or similar structure and are only distinguished by the data itself. Such similar objects are summed up as a **class**. A class is so to speak a template (or a form) for an object; after **instantiation** a real object is created from a class with space for data, and the methods common for all objects of that class.

Often you need only minor modifications to a class to obtain a new one, and therefore you use **inheritance** to create a **sublcass** - a derivative. Additional variables and modified or new methods are just appended to the class.

All objects of a class and its subclasses have a common message protocol, thus they understand the same messages and react similarly. The differences in detail are called **polymorphism**. So may each graphical object have the method **draw_me**, but one object may draw a circle, another a point or rectangle.

If you emphasise the common protocol to all objects of a class hierarchy, you create the protocol separated from the implementation of the subclasses and call the class, which contains only protocol but no implementation, an **abstract data type**. Such a data type is the class data presented in the following listing:

```
Memory also Forth
object class data     \ abstract data class
  cell var ref        \ reference counter
public:
  method !  method @  method .
  method null   method atom?   method #
how:
  : atom? ( -- flag ) true ;
  : # ( -- n ) 0 ;
  : null ( -- addr ) new ;
class;
```

Here I must add that in *BerndOOF* all classes are eventually originated from the same parent class, the class object. Also, classes and objects aren't only used for operating on data, but also for creating new subclasses and instantiation of objects. Therefore a class is just an object without a data area. A class can create new subclasses using the method `class`.

The description of a class consists of two parts: a declaration of variables and methods, and the implementation of the methods. All variables, all polymorphic and externally accessible methods must be declared; helper methods could be declared optionally. In the implementation part, undeclared methods are automatically declaired as `EARLY` (private).

The example creates a cell sized variable called `ref`, in the private area of the class that isn't visible from the outside, but can be inherited (thus corresponds to **protected:** in C++). `public:`, thus publically available are the six methods `!`, `@`, `.`, `null`, `atom?`, and `#`, which are used to store, read, and display the value, to create a **null** object, the query whether the object is atomic or composed, and the number of sub objects if the latter is the case.

The last three methods are already implemented, since they are the same for all simple objects. The unimplemented methods cannot be executed, more precisely they lead to `abort"`. They must be implemented in real data types, like in the following data type `integer:`.

```
data class int
  cell var value
how:
  : ! value F ! ;
  : @ value F @ ;
  : . @ 0 .r ;
  : init ( data -- ) ! ;
  : dispose
    -1 ref +!  ref F @ 0<=
    IF super dispose THEN ;
  : null ( -- addr ) 0 new ;
class;
```

Here I create a new class in the same manner and allocate a (private) variable `value`. The two methods store and fetch (`!` and `@`) access `value`. As an interface this is sufficient. The `F` before the words changes the interpretation from the object vocabulary to the normal vocabulary, thus the normal Forth kernel words `!` and `@` are used. The method `.` is easy to understand too. Here the `@` however is interpreted as the access method.

I must say a few words about the methods `init` and `dispose`: the `init` method is called in the creation of the object and is used to initialise it. Here in the example I initialize `value` with a number on the stack. The `dispose` method removes an object from the dynamic memory management. If you modify this method, you must (unlike in C++) explicitely call the `dispose` method of the parent class with `super dispose`. I dispose only if the reference counter is zero or negative, meaning that there are no further references. Otherwise, the reference counter is just decremented.

The `null` method now has the meaning as expected: it creates an object with the value 0 (dynamically) and leaves its address on the stack. The word `new` is, like `dispose`, a method. Without additional information (thus without class or object), the method of the current class is used.

3.1.2 Binding: Late or Early?

Let's take a step back and look at how methods are called. How much must be defined at compile time, and what has to be resolved at runtime, and then should not produce an error?

As long as it is clear which method of which class is executed, as with `super dispose`, it will be resolved at compile time and create a direct call to the specified method. This is called **early binding**. It's for sure the fastest method, but unfortunately it doesn't allow for polymorphism.

Often it is not clear in advance which subclass the object belongs to when you want to send a method to it. You have to find the address of the method at runtime, then. A search in the dictionary is prohibitively inefficient, as well as a (possibly even sequential) search over a numerical key isn't what I call run time efficiency.

In *BerndOOF* therefore, each object contains a pointer to a jump table as its first element. The jump table contains the addresses of all methods. This doesn't only guarantee a response time independent of the number of methods (after all, Forth should still remain a real time language), it is also quite fast. Especially, since this approach is so easy to code, a simple macro can be directly inserted in the caller's code.

What's prevented, or at least made much more difficult, with this approach is **multiple inheritance**. Crossing a new child class from several parent classes is problematic anyway. Methods and variables with the same names must be renamed if they are not inherited from the same grandparent class, and the offsets of variables in the object change (and so must also be determined at runtime). Alternatively, the compiler must ensure in advance that the necessary space for the mixed class is already reserved in the parent classes. This is a space versus speed tradeoff that cannot be done with a one-pass compiler, and creates difficulties even in complex systems.

A very important aspect is the real time properties of the created code. Thus the run times must be known to the programmer. In turn, this only leaves completely deterministic approaches for binding. Only then the programmer does not lose control of what he writes. C++ does not have this property, and therefore is only of limited use for real time applications.

3.1.3 Objects as Instance Variables

Quite often a straight-forward class hierarchy is good enough. Appropriate abstract data types allow us to circumvent real multiple inheritance issues. In the case of an emergency, you can reach a sort of `multiple inheritance` by copying the sources.

What is necessary though is to have objects as instance variables in other objects, as well as by pointer and direct reference. This can be shown using lists as example, since they need pointers in their implementation:

```
forward nil
data class lists
public:
  data ptr first
  data ptr next
  method empty?
  method ?
how:
  : null nil ;
  : atom? false ;
class;

| lists class nil-class
how:
  : empty? true ;
  : dispose ;
  : . ." ()" ;
class;

| nil-class : (nil
(nil self Aconstant nil
nil (nil bind first
nil (nil bind next
```

Here, we first create an abstract data class for lists; this needs as both pointer to first and rest of the list as data. Since both may normal data, also **dot pairs** are allowed as in Lisp. Would the rest of the list have been a list again, the type isn't necessary; that creates a pointer to the object of the current declared class. The phrase `lists ptr next` won't work, since the class `lists` isn't completely defined at this point and therefore can't be executed.

Additionally to these pointers you also need a few methods: a list could be empty, so you should be able to ask for that. Also, it would be quite useful to display the first element (with `?`).

A null-list is the empty list, also called `nil`. Since this is a list, it must be declared later, therefore I create a forward reference, which is resolved with the later definition of `nil`.

Empty lists differ from ordinary lists quite significantly. They always return true to `empty?`, there's only one of them, and this one certainly may not deleted. It displays a pair of parentheses.

Now I create an element of the class of empty lists, and the address of this element (put on the stack with the method `self`) finally is called `nil`. Both the first as the next element of the empty list is again the empty list. That prevents crashes when a program runs over the end of the list.

The method `bind` allows to bind object references to an object pointer. The object pointer `first` of the object `(nil` behaves, after being bound, exactly like the object that it is bound to, thus `(nil` itself. This is more interesting with real lists:

```
lists class linked
how:
  : empty? false ;
  : init ( first next -- )
    dup >o 1 ref +! o> bind next
    dup >o 1 ref +! o> bind first ;
  : ? first . ;
  : @ first @ ;
  : ! first ! ;
  : . self >o '(
    BEGIN
      emit ? next atom? next self o> >o
      IF ." . " data . o> ." )" EXIT THEN
      bl empty?
    UNTIL
    o> drop ." )" ;
  : # next # 1+ ;
  : dispose
    -1 ref +! ref F @ 0> 0= IF
      first dispose next dispose super dispose
    THEN ;
class;
```

A linked list certainly isn't empty. On creation, I bind the references `first` and `next`; appropriate object addresses must have been put on the stack. At binding, I increment the reference counters of the objects - now another pointer points to them. To make them the current object, I push them on the object stack, thereby with `ref` their reference counter is addressed, and not the one of the list. The object stack isn't a real stack; only the topmost element is put into a register, the rest is on the return stack.

The methods `@`, `!` and `?` refer to the first element of the list; they are only passed through. No complex pointer arithmetic is necessary, the name of the reference is sufficient.

To print a list, I must walk through the list. Before the first element, I open a parenthesis, otherwise the elements are separated by blanks. The current first element of the list is displayed. If the next element is an atom, it must be printed as dot pair; the list ends then. The list also ends when the next element is the empty list. Afterwards, only the parenthesis has to be closed, and the blank is dropped from the stack.

Surprising is the recursion in `#`, which competes the length of a list. It simply computes the length of the rest of the list, increments the result and and finishes. As soon as the list terminates with nil or an atom that has length=0, the recursion terminates. Here you first see a clear advantage of object oriented programming; it makes lots of `IF..ELSE..THEN` for decisions unnecessary and therefore eases recursions.

At deletion of a list, both parts of the list and the node itself have to be deleted. Here also no case decision is necessary, and the termination question, which is often forgotten in recursive programs, isn't asked here.

Now we just need element objects for the list. We already have numbers, but strings would be nice, too. Here they are:

```
int class string
how:
  : !    ( addr count -- )
         value over 1+ SetHandleSize
         value F @ place ;
  : @    ( -- addr count ) value F @ count ;
  : .    @ type ;
  : init ( addr count -- )
    dup 1+ value Handle! ! ;
  : null S" " new ;
  : dispose
          ref F @ 1- 0> 0=
          IF value HandleOff THEN
          super dispose ;
class;
```

We derive the class string from `int`. I use its instance variable value as handle, as pointer to a movable memory area. There, the string is stored as counted string. When storing a new string, the size of the memory block must be adusted; at the first time, it must be allocated, and freed at deletion. All the rest is self-explaining, I hope.

Very useful is the pointer class. You can directly create pointer variables, but you can't insert them into e.g. a list.

```
data class pointer
public:
  data ptr container
  method ptr!
how:
  : !    container ! ;
  : @    container @ ;
  : .    container . ;
  : #    container # ;
  : init ( data -- ) dup >o 1 ref +! o> bind container ;
  : ptr! ( data -- ) container dispose init ;
  : dispose
    -1 ref +! ref F @ 0> 0=
    IF container dispose super dispose THEN ;
  : null nil new ;
class;
```

Analoguous to the list I create a pointer instance variable `(pointer)`; then there's the method `ptr!`, which is used to assign a new object. The methods `@`, `!`, `.` and `#` are fed through to the container. The `init` method binds a passed object to the pointer. The method `ptr!` first releases the previous object, and afterwards stores the new object. I certainly care about reference counting here.

Deletion of a pointer object also means that one pointer less points to the object (and it eventually has to be deleted); afterwards, the pointer is deleted.

Analoguous to a pointer you can create a whole array of pointers:

```
data class array
public:
  data [] container
  cell var range
how:
  : !         ( <value> n -- ) container ! ;
  : @         ( n -- <value> ) container @ ;
  : .         [ # 0
              ?DO emit I container . , LOOP
              drop ." ]" ;
  : init      ( data n -- ) range F ! bind container ;
  : dispose   -1 ref +! ref F @ 0> 0=
              IF  # 0
                  ?DO I container dispose LOOP
                  super dispose
              THEN ;
  : null      ( -- addr ) nil 0 new ;
  : #         ( -- n ) range F @ ;
  : atom?     ( -- flag ) false ;
class;
```

Similarly to the method `new` you create a new array of objects with `new[]` which contains elements of this class. The array really is an array of pointers, you can assign other objects at any position of the array with `bind[]`. The array index for accessing an array variable is expected on the stack.

3.1.4 Tools and Application Examples

The list packet still isn't very easy to use. I've written a few small tools that eases the use - but certainly won't make up a complete Lisp or something like that out of it:

```
: cons         linked new ;
: list         nil cons ;
: car          >o lists first self o> ;
: cdr          >o lists next self o> ;
: print        >o data . o> ;
: ddrop        >o data dispose o> ;
: make-string  string new ;
: $"           state @ IF
                 compile S" compile make-string exit
               THEN
               '"' parse make-string ; immediate
```

The words `cons` and `list` help to create a list. `Cons` concatenates two objects on the stack to a list (TOS as next, thus should be a list; NOS as first element of the list). `List` takes an object and together with `nil` creates a list out of it.

`Car` and `cdr` should be known from Lisp; they return first with respect to the rest of the list.

`Print` calls the output method of an object.

`Ddrop` finally removes and deletes an object.

`Make-string` is the string constructor, analogous to `list`.

`$"` constructs a string constant.

As example how to create a list with these tools:

```
$" Dies" $" ist" $" ein" list cons $" Test" list cons cons ok
dup print (Dies (ist ein) Test) ok
pointer : test ok
test . (Dies (ist ein) Test) ok
test # . 3 ok
```

3.2 The Complete BerndOOF Description

The interface to object oriented programming in *BerndOOF* divides into three parts:

- Tools to manage objects, which are themselves not object related, and the classes from which all other classes and objects are derived,
- Tools to create instance variables and methods,
- methods of the root class, to create new classes, instances, handling of object pointers and similar things.

Only the words of the first item above are directly accessible from the `FORTH` vocabulary. The words of the second item are only available during declaration of a class, and the words of the third item are not words in the traditional Forth sense, but are methods of objects.

BerndOOF uses a coherent way to manage classes: classes are objects, although with class global instance variables, that are just used to create and manage new classes and objects. Classes are also used to send messages to objects whose address is stored in an object pointer, and that need explicit context (because it's not the current defined object), thus are also used as a sort of type casting.

`Vocabulary oo-types      \ --`
All the words that are used to declare classes and implement methods are in the vocabulary `OO-TYPES`. `OO-TYPES` must be topmost of the search order during class definition, since otherwise conflicts would arise. For example : is defined in `OO-TYPES`, the current `PUBLIC` thread, as well as in `FORTH`.

`: ^ ( -- o )`
Returns the pointer o to the current object.

`: o@ ( -- addr )`
Returns the address of the method table of the current object.

`: >o ( o -- ) ( OS -- o )`
Moves the pointer to object *o* to the object stack. The object thereby becomes the current object. Attention: the previously used object is pushed on the return stack, object stack accesses therefore must be balanced with other return stack accesses like `DO ... LOOP`s and `>R` and `R>`.

`: o> ( -- ) ( OS o -- )`
Pops the pointer to the current object from the object stack. The previously used object is restored from the return stack and becomes the current object.

`: static   ( "<name>" -- ) \ oof- oof`
Creates a variable that is common to all the objects of a class. This variable is cell sized and created uninitialized as pointer.

`: method   ( "<name>" -- ) \ oof- oof`
Declares a method. Methods declared like this are late bound, if it's not specified in the context which class is used.

`: early    ( "<name>" -- ) \ oof- oof`
Declares an early bound method. You can't change such a method in a subclass, if you want to use the same name again, you have to declare the early method again.

`: var ( size "<name>" -- ) \ oof- oof`
Creates an instance variable of *size* bytes length.

`: defer    ( "<name>" -- ) \ oof- oof`
Declares an object specific method, that can execute object specific actions. Execution tokens are assigned with `IS`. This is e.g. useful to assign callbacks.

`: dynamic  ( -- )  ['] Dalloc alloc ! ;  dynamic`
Dynamic object creation in the heap on `NEW`. This is the default behaviour.

`: static  ( -- ) ['] Salloc alloc ! ;`
Static object creation in the dictionary on `NEW`. You can compile object structures, preserve them with `SAVE` and reuse them after program load, as long as the objects themselves don't use other functions to allocate dynamic memory.
Each object consists of variables and methods that have to be declared. The methods afterwards need to be implemented. Visibility of variables and objects to the outside can be selected. Private methods and variables are only visible inside the class and subclasses, externally visible methods and variables have to be declared `public`.

BerndOOF separates declaration and implementation of classes. Both together form the definition of a class.

`: bind ( o "<name>" -- )`
Binds the object *o* to the object pointer <*name*>. The object *o* must be derived from the class *name* or a subclass thereof.

`: how:  ( -- ) \ oof- oof how-to`
Changes from the declaration to the implementation part. In this part, you initialise static variables, and implement methods.

`: class; ( -- ) \ oof- oof end-class`
Ends the implementation of a class.
The management of classes and objects is task of the classes themselves. Therefore, the root class `OBJECT` provides some methods and class global variables. These can be separated into the following groups:

- Class browser
- Subclass creation
- Memory management, instance creation
- Binding

`: ptr ( "<name>" -- ) \ oof- oof`
Declares an object pointer, which must point to the currently declared class or a subclass, and is initialized with `BIND`.

`: asptr ( class "<name>" -- ) \ oof- oof`
Casts a pointer created with PTR to the currently declared class, and declares the casted pointer as *name*.

`: : ( "<name>" -- ) \ oof- oof colon`
Implements the method `name`. You end the implementation with `;`.

`Create object  ( ... "<method>" -- ... )  immediate  0 (class`
The root of all object classes. Executes `method` resp. compiles it dynamically bound in the context of the current object.

```
cell var  oblink       \ create offset for backlink
```
First instance variable: points to the method table of an object.

```
static    parento      \ -- addr
```
Points to the parent class.

```
static    childo       \ -- addr
```
Points to the last derived subclass.

```
static    nexto        \ -- addr
```
Points to the next old subclass of the parent class.

```
static    method#      \ -- addr
```
Number of the methods and static variables in bytes.

```
static    size         \ -- addr
```
Number of bytes used as dynamic memory

```
static    newlink      \ -- addr
```
Points to a list of all sub-objects which consume memory in the object. Used for the internal memory management.

```
method    init ( ... -- ) \ object- oof
```
Initializes an object with the given parameters. `INIT` is also called for all objects that are declared as instance variables, in the order of declaration, but first for the main object. `INIT` is a polymorphic method.

```
method    dispose ( -- ) \ object- oof
```
Frees the object's memory space. `DISPOSE` is a polymorphic method.

```
early     class ( "name" -- ) \ object- oof
```
Starts declaration of a subclass called `name`.

```
early     new ( -- o )  immediate  \ object- oof
```
Creates a nameless object (instance) of the current class.

```
early     new[] ( n -- o )  immediate  \ object- oof new-array
```
Creates an array of nameless objects of the current class with *n* elements.

`early     : ( "<name>" -- ) \ object- oof define`

Creates an object under the name `name`.

`early     ptr ( "name" -- ) \ object- oof`

Creates a pointer to an object of the current class (or subclass) under the name `name`. The pointer is empty at first, use `BIND` to assign an object to the pointer.

`early     asptr ( o "name" -- ) \ object- oof`

Casts a pointer created with `PTR` to the current class and creates the casted pointer under the name `name`.

`early     [] ( n "name" -- ) \ object- oof array`

Creates an array of objects with *n* elements under the name *name*.

`early     ::  ( "name" -- )  immediate \ object- oof scope`

Binds method *name* of the current class early. Invoked directly in the implementation part of a class, it inherits methods from other classes, and thus allows a limited form of multiple inheritance. The method must be defined in a common parent class. Only the code address of the method is inherited.

`early     class? ( o -- flag ) \ object- oof class-query`

Early Method: Checks class relationship. *Flag* is only true when object is in the upward derivation chain of the object that executes `CLASS?`.

`early     goto  ( "name" -- )  immediate \ object- oof`

Used for end (tail) recursion. The method *name* is called directly, without pushing a return address (or eventually the old object class) onto the return stack.

`early     super  ( "name" -- ) immediate \ object- oof`

Binds method *name* of the parent class early. `SUPER` is used to modify inherited behaviour, and to access the original behaviour in the modified method. You can use `SUPER` repeatedly to access methods higher up the inheritance chain.

`early     self ( -- o ) \ object- oof`

Returns the address of the object.

`early     bind ( o "name" -- )  immediate \ object- oof`

Stores the address object in the pointer variable *name* The object must belong to the pointer's class or a subclass thereof.

`early     link ( "name" -- addr )  immediate \ object- oof`

Creates a reference to the object pointer *name*, thus an address where object pointers can be stored with `!`.

`: public:  ( -- )  ;  \ empty stub for VFX`

Switches to public declaration. All further methods and variables are visible interfaces to the object.
For debugging porposes, there is the `DEBUGGING` object. It contains further methods which are helpful for debugging. Not all methods are implemented yet.

`object class debugging  ( ... "name" -- ... )`

This is a helper class, that provides necessary tools to debug objects. Otherwise like `OBJECT`.

`early words  ( -- )`

Lists all the words in the public and private vocabulary.

`early m'  ( "name" -- xt )`

Finds the xt of a method or object variable.

`early see ( "name" -- )`

Decompiles *Name*

`early view ( "name" -- )`

Opens the editor at the declaration of *name*.

`early trace' ( ... "name" -- ... )`

Traces the method *name*.

3.3 Formal Syntax

```
declaration ::=
  <parent> CLASS <object>
  {[ private: | public: ] <creator> <selector> }
  [ HOW: {: <method> <coding>  ; }]
CLASS;
<creator> <selector> ::=
  STATIC <static> | METHOD <method> | EARLY <method> |
  <number> VAR <var> | <object> ( : | [] | PTR ) <instance>
<parent> ::=
  OBJECT | <object>
<creation> ::=
  <object> (: | PTR) <instance> |
  <number> <object> [] <instance>
<coding> ::=
  <word> <coding> |
  { <instance> } <selector> <coding>
```

3.4 Porting the OOP packages

All the host dependencies are isolated in separate directories for each host system other than *bigForth*, e.g. the VFX Forth code is in *Minos/vfx-minos*. Use one of the port folders as the basis for yours if you need to port Minos to another host.

4 Portability layer for VFX Forth

The code in *Minos/vfx-minos/VFXharness,fth* contains the main Forth host dependencies required to port Minos and Theseus to VFX Forth for Linux.

In order to allow some redefinitions to be sensitive to whether the Minos or VFX Forth notation is to be used, the flag `MinosMode?` is provided. If you want to revert to the original VFX Forth behaviour, clear this flag.

```
1 value MinosMode?      \ -- flag
```
Some words can operate in Minos mode or in the native host mode.

4.1 Blocks

The Minos/Theseus editor supports blocks. Block support is taken from a library file.

4.2 Floating point

```
: ans-float     \ --
```
Set VFX Floats to ANS mode, in which ’.’ is both the double number separator and the floating point separator.

```
: mpe-float     \ --
```
Set VFX Floats to MPE mode, in which ’,’ is the double number separator and ’.’ is the floating point separator.

```
include %lib%/Ndp387.fth
```
Compile floating point library.

```
: float ;       \ --
```
Indicates that floats have been compiled.

```
 : fx$          \ F: f -- ; -- caddr len
```
Convert a floating point number to ASCII text. This still needs furthr processing according to the required presentation mode to insert a decimal point and so on.

```
Code f>r        \ F: f -- ; R: -- f
```
Transfer a float to the return stack.

```
Code fr>        \ R: f -- ; F: -- f
```
Transfer a float from the return stack.

```
code fr@        \ R: f -- f ; F: -- f
```
Copy a float from the return stack.

```
: fm*           \ F: f -- f’ ; n --
```
Multiply a float by an int.

```
: fm/           \ F: f -- f’ ; n --
```
Divide a float by an int.

```
: fm**          \ F: f -- f’ ; n --
```
Raise a float to the power of the int.

```
: f>fs          \ F: f -- ; -- fs
```
Convert a native float to its 32 bit form on the data stack.

```
: fs>f          \ fs -- ; F: -- f
```

Convert a 32 bit float on the data stack to a native float.

4.3 Floored division

Many graphics operations in Minos require floored division. Because ANS Forth permits Forth systems to default to either symmetric or floored division, a set of operations that always use floored division is defined.

```
: /modf ( n1 n2 -- rem quot )
```

Floored version of `/MOD`.

```
: /f ( n1 n2 -- quot )
```

Floored version of `/`.

```
: modf ( n1 n2 -- rem )
```

Floored version of `MOD`.

```
: */modf ( a b c -- rem quot )
```

Floored version of `*/MOD`.

```
: */f ( a b c -- n )
```

Floored version of `*/`.

```
Synonym m/mod fm/mod  ( d n -- rem quot )
```

This name is widely used, so we force it to be floored.

4.4 Miscellaneous math

```
$7FFFFFFF Constant mi   \ -- n
```

Returns MAXINT, the largest positive integer.

```
Synonym ud/mod mu/mod ( ud1 u2 -- urem udquot )
```

Rename to preserve Minos code base.

```
: d*     ( ud1 ud2 -- udprod )
```

Unsignsigned multiply of two doubles to produce a third.

```
: 0max  0 max ;
```

A micro-optimisation for `0 MAX`.

```
: 0min  0 min ;
```

A micro-optimisation for `0 MIN`.

```
: 8*  ( n -- 8*n ) 3 lshift ;
```

A micro-optimisation for `8 *`.

```
: 3*  ( n -- 3*n ) dup 2* + ;
```

A micro-optimisation for `3 *`.

4.5 Headerless words

```
: |  ( -- )  ;
```

Used in the form below to indicate that the word can be headerless. If the host Forth does not support headerless words, this becomes a `NOOP`.

```
  | : <name> ...  ;
```

4.6 Memory access with address increment

`: c@+            \ caddr -- char caddr'`

Fetch a char/byte and increment the address.

`: w@+            \ addr -- w addr'`

Fetch a 16 bit word and increment the address.

`: @+             \ addr -- x addr'`

Fetch a cell and increment the address.

`: c!+            \ b addr -- addr'`

Store a char/byte and increment the address.

`: w!+            \ w addr -- addr'`

Store a 16 bit word and increment the address.

`: !+             \ x addr -- addr'`

Store a cell and increment the address.

`: wextend        \ w -- w'`

Sign extend a 16 bit word to a cell.

`: cextend        \ b -- b'`

Sign extend an 8 bit byte to a cell.

`: cx@            \ addr -- sb`

Fetch a byte and sign extend it.

`: wx@            \ addr -- sw`

Fetch a 16 bit word and sign extend it.

`: wx@+           \ addr -- sw addr'`

Fetch a 16 bit word, sign extend it and increment the address.

4.7 Memory allocation

Minos uses its own versions of memory access words. Most of these words simply `THROW` on error. These words were inspired by the Mac OS functions.

`vocabulary memory       \ --`

Vocabulary holding the memory access words.

`: NewPtr ( len -- addr )`

Allocate a block of memory.

`: DisposPtr ( addr -- )`

Free a mmory block.

`: NewFix  ( root len n -- addr )`

Allocates a new element of length *len* from a pool specified by the cell-sized variable at *root*. If the pool has no free elements, *n* new elements will be created and added to the pool.

`Variable Masters         \ -- addr`

The root of a pool.

`: NewMP ( -- MP )`

Allocate a new master pointer, referred to as *mp* in the stack comments for other words.

`: NewHandle ( len -- mp )`

Allocate a new dangling memory area, which is pointed to by by the returned master pointer.

`: DisposHandle ( addr -- )`
Free a a dangling memory area and associated master pointer.

`: Handle! ( len mp -- )`
Allocate a block of *len* bytes and associate it with *mp*.

`: SetHandle ( addr mp -- )`
Set the given master pointer.

`: HandleOff ( addr -- )`
Free the block whose pointer is at *addr*.

`: Hlock ( mp -- )`
Lock the pool. A dummy in most implementations.

`: Hunlock ( addr -- )`
Unlock the pool. A dummy in most implementations.

`: SetHandleSize ( mp size -- )`
Resize the pool.

`: GetHandleSize ( mp -- size )`
Get the size of the pool.

4.8 Overrides

The words in this section override the VFX Forth versions

`variable Seed   \ -- addr`
A dummy variable used to satisfy references.

`: RANDOM        \ n1 -- n2`
Generate a random number *\i{n2) in the range 0..n1-1. The VFX Forth word that directly performs this is `CHOOSE`.

4.9 Miscellaneous tools

`: pin ( x n -- )`
Store x in the nth stack slot. The inverse of `PICK`.

`: \G postpone \ ; immediate`
A documenting comment used by gForth.

`: ?EXIT         \ --`
Equivalent to `IF EXIT THEN`.

`: 8aligned ( n1 -- n2 )`
Align *n1* to a boundary of eight.

`: F  ( "<name>" -- )`
Compiles `name` with `FORTH` as the first vocabulary in the search path.

`synonym AVariable Variable`
A variable holding an address which may need relocation.

`synonym AValue Value`
A value holding an address which may need relocation.

`synonym A, ,`
For an address which may need relocation.

`synonym AConstant Constant`
An address which may need relocation.

`synonym ALiteral Literal`
An address which may need relocation.

`synonym Patch Defer`
Equivakent to `Defer`.

`synonym << lshift`
Equivakent to `lshift`.

`synonym >> arshift`
Equivakent to `arshift` which is as `rshift`, but performs an arithmetic right shift.

`synonym toss previous`
Equivakent to `previous`.

`synonym extend s>d`
Equivakent to `s>d`.

`: cont  ( addr -- )  >r ;`
Causes a branch to *addr*. When that code `EXIT`s, execution resumes after the `cont`.

`: push  \ addr --`
Save the contents of *addr* on the return stack, execute the rest of the word and then restore the contents of *addr*.

`: &     \ -- addr`
Return the address of the data area of word rather than its xt.

`: 0>= 0< 0= ;`
Equivakent to `0 >=` or `0< 0=`.

`: 0<= 0> 0= ;`
Equivakent to `0 <=` or `0> 0=`.

`: u>= u< 0= ;`
Equivakent to `u< 0=`.

`: u<= u> 0= ;`
Equivakent to `u> 0=`.

`: rdrop  postpone r>  postpone drop ; immediate`
Equivakent to `R> DROP`.

`: i'    \ --`
Use inside `DO ... LOOP` and friends to return the loop limit.

`: +i'   \ n -- flag`
Increment the loop limit by *n* and return true if the index is now greater than the limit. The comparison is performed using circular arithmetic.

`: ith   \ addr -- x`
Use inside `DO ... LOOP` and friends to return the contents of the *fo`{I}`th cell in an array.

`: list> ( thread -- element )`
Execute the rest of the word for each element of the given list.

```
-1 cells Constant -cell \ -- -4
```

The ngative size of a cell.

```
: over2         \ a b c -- a
```

Equivalent to `2 PICK`.

```
: toupper ( char -- char’ )
```

Convert char to upper case.

```
: tolower ( char -- char’ )
```

Convert char to lower case.

```
: \needs        \ "name" --
```

If `name` is not defined, interpret the rest of the line, otherwise ignore it. Usually used in the form:

```
  \needs foo  include foobar.fth
```

```
: onlyforth  ( -- )  only forth ;
```

Equivalent to `ONLY FORTH`, setting the basic search order.

```
: perform       \ ??? addr -- ???
```

`EXECUTE` the xt held at *addr*.

```
: forward       \ "name" --
```

Declares a forward reference. Will be replaced by `DEFER`.

```
: forward?  ( xt -- flag )
```

Return true if the *xt* is of a forward referenced word. OBSOLETE.

```
: : ( "name" -- )
```

Redefinition of `:` to cope with forward references in a very rough way. OBSOLETE.

```
: recursive     \ --
```

Used inside a colon definition to make the word visible for direct recursion.

```
: +bit ( addr n -- )
```

Set bit *n* in the bit array starting at *addr*.

```
: -bit ( addr n -- )
```

Clear bit *n* in the bit array starting at *addr*.

```
: bit@ ( addr n -- flag )
```

Test bit *n* in the bit array starting at *addr*.

```
: ,0"   ( -- )
```

Lay a zero terminated string. The end of the string is not aligned.

```
  ,0" <text>"
```

```
synonym 0" z"   \ -- ; -- addr
```

Compile a zero terminated string. At run-time the address of the first character is returned.

```
: >len ( addr -- addr u ) dup zstrlen ;
```

The equivalent of `COUNT` for a zero-terminated string.

```
: 0place ( caddr len addr -- )
```

Save the string *caddr/len* as a zero-terminated string at *addr*.

```
: safe/string ( c-addr u n -- c-addr’ u’ )
```

protect /string against overflows.

4.10 Interpretive structures

These structures only operate in the context of a single line.

`: [IFUNDEF]     \ "name" --`
Equivalent to `[undefined] name [if]`.

`: [IFDEF]       \ "name" --`
Equivalent to `[defined] name [if]`.

`: [DO]  ( n-limit n-index -- )`
Interpreted version of `DO`.

`: [?DO] ( n-limit n-index -- )`
Interpreted version of `?DO`.

`: [+LOOP] ( n -- )`
Interpreted version of `+LOOP`.

`: [LOOP] ( -- )`
Interpreted version of `LOOP`.

`: [FOR] ( n -- )`
Interpreted version of `FOR`.

`: [NEXT] ( n -- )`
Interpreted version of `NEXT`.

`: [I]`
Interpreted version of `I`.

4.11 Application startup chains

Minos requires two startup chains to which actions can be added. The first uses the VFX Forth `ColdChain` mechanism. This chain is executed in compilation order, i.e. the first word added is executed first. The second chain, anchored by `MainChain`, is executed so that the last word added is executed first. The `MainChain` allows outermost operations to initialise themselves before earlier operations are initialised.

`variable MainChain      \ -- addr`
Anchors the list of operations.

`: withChain:    \ anchor --`
Starts a `:noname` word that is executed as part of the chanin whose anchor is given.

`: Cold:         \ --`
Starts a `:noname` word that is executed as part of `ColdChain`. The first word added is executed first.

`: Bye:          \ --`
Starts a `:noname` word that is executed as part of `ExitChain` when VFX Forth finishes. The lasr word added is executed first.

`: Main:         \ --`
Starts a `:noname` word that is executed as part of `MainChain`. The last word added is executed first.

`: WalkMainChain \ --`
Execute all the actions anchored by `MainChain`.

4.12 MODULE from bigFORTH

`#16 cells buffer: ModCurrents   \ -- addr`
Holds the value of *\fo{CURRENT when a module is created.

`variable pMC    \ -- addr`
Holds a pointer into `ModCurrents` above.

`: initPMC       \ --`
Initialise `pMC` above.

`: +Module       \ --`
Save the owner of a new module.

`: -Module       \ --`
Restore the owner of the previous module.

`: widOwner      \ -- wid`
Return the wid of the owning module.

`: findExport    \ caddr len -- xt`
Find the given word in the module, i.e. in the `CURRENT` wordlist.

`: createExport  \ caddr len --`
Perform `CREATE` on the string, making the word in the owning wordlist.

`: MakeExport    \ caddr len --`
Create a new definition iwhich behaves like an existing one. The new definition is in the owning vocabulary and is searched for in the `CURRENT` wordlist.

4.13 Operating system dependencies

`char : constant pathsep \ -- char`
The separator between items in a list of paths. This is a colon for Unix-based operating systems, but varies for others, e.g. a semi-colon is used in Windows.

`: .NextToken    \ --`
Display the next token in the input stream.

`: terminali/o ( -- )`
Use the same output device as when the code was compiled. Be careful!

`: my.s ( ... -- ... )`
A horizontal display version of `.s`.

`: (~~) ( in line source -- )`
Display the source location using `terminali/o`.

`: ~~ ( -- )`
Compile code so that the source location is displayed at run time.

`: [~~]          \ --`
Display the current source location.

Index

!

!+ 31

%

%lib%/ndp387.fth 29

&

& 33

(

(~~) 36

*

*/f 30
*/modf 30

+

+bit 34
+i' 33
+module 36

,

,0" 34

-

-bit 34
-module 36

.

.nexttoken 36

/

/f 30
/modf 30

:

: 25, 26, 34, 36
:: 26

<

<< 33

>

>> 33
>len 34
>o 24

?

?exit 32

@

@+ 31

[

[+loop] 35
[?do] 35
[] 26
[~~] 36
[do] 35
[for] 35
[i] 35
[ifdef] 35
[ifundef] 35
[loop] 35
[next] 35

^

^ 23

\

\g 32
\needs 34

|

| 30

~

~~ 36

0

0" 34
0<= 33
0>= 33
0max 30
0min 30
0place 34

3

3* 30

8

8* 30
8aligned 32

A

a, 33
aconstant 33
aliteral 33
ans-float 29
asptr 25, 26
avalue 32
avariable 32

B

bind 24, 26
bit@ 34
buffer: 36
bye: 35

C

c!+ 31
c@+ 31
cextend 31
childo 25
class 25, 26
class; 24
class? 26
cold: 35
constant 34
cont 33
createexport 36
cx@ 31

D

d* 30
defer 24
dispose 25
disposhandle 32
disposptr 31
dynamic 24

E

early 24
extend 33

F

f 32
f>fs 29
f>r 29
findexport 36
float 29
fm* 29
fm** 29
fm/ 29
forward 34
forward? 34
fr> 29
fr@ 29
fs>f 30
fx$ 29

G

gethandlesize 32
goto 26

H

handle! 32
handleoff 32
hlock 32
how: 24
hunlock 32

I

i’ 33
init 25
initpmc 36
ith 33

L

link 26
list> 33

M

m’ 26
m/mod 30
main: 35
mainchain 35
makeexport 36
masters 31
memory 31
method 24
method# 25
mi 30
minosmode? 29
modf 30
mpe-float 29
my.s 36

N

new 25
new[] 25
newfix 31
newhandle 31
newlink 25
newmp 31
newptr 31
nexto 25

O

o> 24
o@ 23
object 25
onlyforth 34
oo-types 23
over2 34

P

parento 25

`patch` 33
`perform` 34
`pin` 32
`pmc` 36
`ptr` 25, 26
`public:` 26
`push` 33

R

`random` 32
`rdrop` 33
`recursive` 34

S

`safe/string` 34
`see` 27
`seed` 32
`self` 26
`sethandle` 32
`sethandlesize` 32
`size` 25
`static` 24
`super` 26

T

`terminali/o` 36
`tolower` 34
`toss` 33
`toupper` 34
`trace'` 27

U

`u<=` 33
`u>=` 33
`ud/mod` 30

V

`var` 24, 25
`view` 27

W

`w!+` 31
`w@+` 31
`walkmainchain` 35
`wextend` 31
`widowner` 36
`withchain:` 35
`words` 26
`wx@` 31
`wx@+` 31