cspForth for Linux

This is a first release of cspForth - a development snapshot - for the annual general meeting of the German Forth-Gesellschaft e.V. at Kloster Roggenburg, in April 2008.

Copyright (C) 2008 Manfred Mahlow (manfred.mahlow@forth-ev.de)

Introduction

cspForth is a 32 bit Forth System for Linux on x86 PC-Systems. It came into being as a test bed, to evaluate the benefits of **using OOP concepts with Forth**. When looking for a small, easy to understand and easy to modify Forth System, I found that Reva Forth by Ron Aron was close to my needs, even though it did not comform to the ANS Forth Standard.

cspForth supports on demand loading of source code modules, importing functions from shared libraries and creating standalone applications. All this was inherited from Reva 6.0 but modified with respect to my OOP interests.

cspForth has two special features, that make it different from most other Forth-Systems:

- cspForth supports two search orders, the well known search order for vocabularies and wordlists and an extra search order used for classes, objects and interfaces.

- cspForth supports preludes. A prelude is a Forth word, that is assigned to another Forth word in a way, that it is executed (by the outer interpreter) before the word, it's assigned to, is executed or compiled.

This two features are the basis for my way of implementing OOP concepts in Forth. The result is an easy to use OOP Toolkit with a syntax in the spirit of Forth, that is available as a loadable source code module.

The basic underlaying idea is, to use implicit context switching to assign methods to objects. An object, before executed or compiled, creates and activates its class specific search order to give access to its methods. A method found in this context, switches back to the default search order (or to another one), before it is executed or compiled itself. The context switching is done by **c**ontext **s**witching **p**reludes.

I found that using OOP concepts with Forth can be very appealing, especially when combined with on demand loading of source code modules.

cspForth is distributed to let you gain your own OOP experience with Forth and to possibly trigger some discussion. It is released under the '**Artistic License**'. Please see the enclosed LICENSE file.

cspForth is work in progress. Please keep in mind that it was made as an OOP test bed and not as a production system.

Installing cspForth

cspForth is distributed as **cspForth-jjmm.tgz** or **cspForth-jjmm.zip** archive file. Unpack the archive file in a directory of your choice. This will create a directory named csp4th-jjmm with several subdirectories and files in it. We will call it the cspForth directory from now on.

In the cspForth directory you will find a file named csp4th. This is the cspForth executable. It should work out of the box with most current Linux distributions. If not, you have to build it from the source (see Building cspForth).

Note:

- You can install cspForth on an USB-Stick for mobile use or to test it with a Linux Live System from CD or DVD.

Building cspForth

Coming soon.

Starting cspForth

The cspForth executable is called csp4th. You will find it in the cspForth directory. It can be invoked with or without options. Options may appear multiple times and in any order. When csp4th is invoked without any option it simply starts up, displays a startup message and waits for terminal input. If invoked with options, it starts up without a startup message and processes the options.

Syntax:

```
csp4th [-e 'words ...'] [-n 'module name'] [-t 'module name'] [filename] [--]
```

Options:

-e 'words ...'	causes 'words ..' to be interpreted
-n 'module name'	loads the file 'module/name.4th' from the cspForth source code library if it is not already loaded
-t 'module name'	loads the file 'module/name.4th' from the cspForth source code library
filename	includes the file 'filename'
–	stops processing of command line options

Before processing any option, cspForth interprets the hidden file '**.appnamerc**' in the users home directory, if it exits. 'appname' is the name of the cspForth executable. The default is '**csp4th**'. This file may be used to configure cspForth during startup. It must contain valid Forth code. *This file, if it exists, should only be editable by the user!*

Notes:

- As long as the cspForth directory is not in your path, you have to enter the relative or absolute file name to invoke the cspForth executable.

- On a Linux System that does not support the /proc file system, you must enter the absolute file name to invoke the cspForth executable. Otherwise it will not startup properly. This is also the case, if you try cspForth on an Ubuntu Live System.

- cspForth does not support command line editing and command line history of itself. To get both you can call the cspForth executable via the rlwrap utility.

Using Modules

cspForth supports to create and use source code library files, called modules. A module is a text file holding valid Forth code. It has the file name extension '.4th' and is stored in one of two module directories. The names of this directories are hold in the variables dir(1) and dir(2). The defaults are 'user' and 'share' in the cspForth directory. You may change this (with care) to your needs. The default search order for modules is dir(1), dir(2).

You can write your own modules, give it the file name extension '.4th' and store it in one of the module directories. Use the 'share' directory for modules that would be of interest for other users too.

Modules can be loaded once or multiple times.

```
needs module name
```

loads the module 'module name' and registers it as loaded. Any further 'needs module name' is then handled like noop.

```
take module name
```

loads the module 'module name' without registering it and can be used to load a module again and again.

The module name is the file name of the related module relativ to the module directory but without the file name extension and with the file name separator characters '/' substituted with spaces, e.g. 'needs String Array' will load the file 'String/Array.4th' from the modules directory dir(1)='<cspForth directory>/user' or dir(2)='<cspForth directory>/share'.

Note:

Text following a 'needs ...' or 'take ...' phrase in the same line, must be separated from the 'needs ...' or 'take ...' phrase by two or more spaces.

Getting Help

cspForth has a context sensitive help system, implemented as a loadable module. The module can be loaded with '**needs help**'. It adds the word '**h**' to the dictionary.

To get help for a word <name> enter '**h <name>**' on the command line.

If <name> is found in the current search order, its glossary entry is displayed. Otherwise a global help file is searched for an entry for the word <name>. If one is found, it is displayed. If not, an error message is displayed.

Without entering a <name> you will get some help or info for the current context.

Using OOP Concepts

Loading the Modules

```
   needs oop ( -- )
```

Loads the OOP module, if it´s not yet loaded. It also loads all other modules required by the OOP module.

After loading the OOP module the system status is as follows :

```
needs oop  ??

  Stack: (0)
Current: oop
Context: oop  oop  forth  root
--------------------------------------------------------------------
 ok
```

- the **data stack** is unchanged
- the current **compilation context** is the vocabulary **oop**
- the current **search context** is the **vocabulary search order** with the vocabulary **oop** on the top

Note: *The vocabulary **oop** should be used as the "root" context for OOP-based definitions.*

Not to Get Stuck

The three most important words of the OOP module are '**..**' , '**??**' and '**???**' . They will help you not to get stuck in the class hierarchy and are available in any context.

```
   .. ( -- )
```

Switches back from a class, object or iface context to the vocabulary search order, that was left, when the class, object or iface context was activated.

Notes:

- *.. is the default prelude that is assigned to methods.*

- *.. is a noop-word in the vocabulary search order.*

```
   ?? ( -- )
```

Displays the current system status and the words of the top wordlist in the current search order.

```
   ??? ( -- )
```

Same as '**??**' but lists all words of all wordlists in the current search order, if it´s a class, object or iface context.

Creating Classes

```
class <name> ( -- )
```

Creates a new class in the current compilation context, using the next word from the input stream as its name.

```
<name> definitions ( -- )
```

Makes the new class the current compilation context.

A new class may inherit from another class.

```
<class> inherit ( -- )
```

The object specification of a class may be an explicit memory request

```
self allot ( u -- )
```

or a list of instance variables (references to objects of other classes)

```
<class1> this <name1> ( -- )
<class2> this <name2> ( -- )
```

or a mixture of both.

An object specification, when finalised, should be sealed explicitly,

```
self seal
```

although it is sealed implicitly when the first instance is created.

The next step is to create methods to be applied to the objects of the class.

```
m: <name> ( x y z oid -- u v )
```

starts a colon definition for a method. '**m:**' has, compared with the Forth word '**:**', the additional affect, to make the created word a context switching one, by assigning it the word '**..**' as context switching prelude.

'**oid**' is the object identifier of the calling object. The programer is responsable to handle it in the methods definition. Sometimes this makes the code complicated, especially when using do loop constructs. Then it's possible to get the 'oid' out of the way by using an **object stack**. This gives a little runtime overhead but can make coding much easier.

To use the object stack, start the stack diagram of the method definition with '**(O**'. This compiles a word, that pushes the oid to the top of the object stack (**TOOS**), when the method is executed.

```
m: <name> (O x y z -- u v )
```

Inside the method definition you can fetch the 'oid' from the TOOS to the top of the data stack (**TOS**) with the word '**this**'. Its a context switching word. Its prelude is the word '**self**', i.e. it gives access to the methods class context.

Note:

- A method definition is, like any colon definition, terminated with the Forth word '**;**'. For methods, the word ';' has an extra compile time semantics: If the word, preceeding ';' is a context switching one, switching to a class or iface context, then the default prelude of the method is overwritten with this word, i.e. the method will switch to that context.

A method can also be created as an alias of an existing Forth word. This is done with

```
method ' <name1> alias <name2>
```

or

```
m' <name1> alias <name2>
```

Creating Objects

To create an instance of a class use the word '**new**'.

```
<class> new <object> ( -- )
```

This creates a variable with <class> as its prelude. A fixed number of address units is assigned to the variable. It was explicitly or implicitly specified with '**u self allot**' in the object specification of <class>. The object, when executed, returns the address of this memory on the TOS. We call it the object identifier 'oid'.

Additional memory may be allocated with the '**init**' method (see the modules '**Buffer.4th**' and '**String.4th**' as examples).

A new object should always be initialized, before it is used.

```
<object> init ( x y z -- u v )
```

The '**init**' method must be explicitly defined in the objects class by the programer. The name '**init**' should always be used for the initialization method. An '**init**' method should at least set initial values at the objects memory.

Using Objects

An object can be any combination of data and methods, you can imagine and implement as a class. You can implement variables, datatypes, data structures, functional building blocks and whole applications. See the predefined classes and the examples in the module directories.

An object, when executed, returns its object identifier '**oid**' on the top of the data stack (TOS). It's the address of the objects data memory. In interpret or compile mode an object gives access to the public instance variables and methods of the class it belongs to, to be executed or compiled.

Using Shared Libraries

cspForth supports to import functions from shared libraries. The (object oriented) library interface can be loaded with

```
  needs libs ( -- )
```

To import a function from a shared library one must first create and initialize a library object to get access to the library

```
        lib new <name> ( -- )    " <filename>" <name> init

  e.g.:  lib new libc ( -- )     " libc.so.6" libc init
```

A function from the library can then be imported with

```
        <name> import <function> ( input parameters -- output parameters )

  e.g.:   libc import time ( a|0 -- sec )
```

or

```
        <name> import <function> as <alias> ( input parameters -- output parameters )

  e.g.:   libc import time as os_time ( a|0 -- sec )
```

Note: *The stack diagram is parsed to determine the number of input and output parameters.*

For further information use the help system in the contexts '**lib**' and '**libs**'.

Interfacing GTK+

Based on the OOP Module, it's relatively easy to use the GTK+ Libraries to create Graphical User Interfaces with cspForth. GTK+ has an object oriented design, although it's written in C. It's class hierarchy can be mapped directly to a corresponding class hierarchy in cspForth and using the same names for classes and properties further simplifies translation from C to Forth.

GTK+ is a really big and powerful toolkit. So I have taken the approach to only implement a minimal subset of properties and methods per class. Further properties and methodes can be added later, when needed for an application. Classes are implemented as loadable source code modules, so one can always only load those classes, that are really needed for an application.

cspForth comes with a lot of predefined classes. Most of the simpler GTK+ Widgets are already implemented and others will follow and, you are invited to contribute. A widget is created as an instance of its class and must be initialized explicitely, with its 'init' method, before any further use.

You will find examples for most of the predefined classes, that will help you figure out, how to use the GTK+ libraries. For the main class hierarchy you will already find glossary entries in the context sensitive help system (after a module has been loaded). For most other classes, and for the examples, help texts are not yet written. Please have a look at the module and example files in the modules directory dir(1), which defaults to <cspForth directory>/user.

See the file hello.1.4th and the files GtkToplevel/example.x.4th to get started. To execute all examples at one go, start cspForth and type

```
take gtk examples
```

Note:

cspForth is work in progress and that's much more the case for the GTK+ Interface. Details may change. You have been warned.

Saving cspForth

You can save the current state of a running cspForth system as a **'turnkey' application** to an executable file. For further information see the help text for '**save**'.