cspForth for Linux

A first development snapshot of cspForth was released for the annual general meeting of the German Forth-Gesellschaft e.V. at Kloster Roggenburg, in April 2008. ***Now the second snapshot csp4th-sn02-080816.tar.gz is available and should be used.*** *The documentation was updated accordingly.*

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 and modified with respect to my needs.

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 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 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 LICENSE file enclosed in the cspForth archive file.

cspForth is work in progress. Please keep in mind that it came into beeing as a test bed and not as a production system.

Installing cspForth

cspForth is distributed as **cspForth-*.tar.gz** or **cspForth-*.zip** archive file. Unpack the archive file in a directory of your choice. This will create a directory named csp4th-* 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

The cspForth archive comes with a prebuild executable. Rebuilding the executable requires the ***FASM assembler***, version 1.62 or later, the ***GNU C Compiler*** and the ***make*** utility. To rebuild the executable start a terminal in the *cspForth directory* and enter the following commands on the command line:

```
make clean ; make all
```

This will delete the old executable and create the new one. Thats all.

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:

```
[/path/]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. *For security reasons, this file, if it exists, should only be editable by the user, it belongs to!*

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 may also be 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.

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

Using Modules

cspForth supports to create and use source code 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 ignored.

```
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 a *space*, 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 utility, 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 search order to the vocabulary search order, that was left, when the class, object or iface search order 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 search order.

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 object is created.

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

```
m: <name> ( i*x oid -- j*x )
```

starts a colon definition for a method. The Forth word **m:** has, compared with the word **:** (colon), 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 i*x -- j*x )
```

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 one. Its prelude is the word **self**, i.e. it activates the *class search order*, the method belongs to.

Note:

- A method definition, like any colon definition, is terminated with the Forth word **;** (semicolon). For methods, the word semicolon has an extra compile time semantics: If the word, preceeding the semicolon 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 the class <class> as its prelude. The memory, needed for the object,is assigned to the variable. It was specified by the object specification of the class <class>. The object, when executed, returns the address of this memory on the top of the data stack (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. The respective method must be defined by the programer and should always be called **init**.

```
<object> init ( i*x -- j*x )
```

The *init method* should at least initialize the objects memory.

Using Objects

An object can be any combination of data and methods. 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 to the 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 utility in the contexts **lib** and **libs**.

Interfacing GTK+

Based on the OOP Module it was relatively easy to implement a **GTK+-API** for **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.

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 methods 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 (you are invited to contribute). A widget is created as an instance of its class and must be initialized explicitly 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. [1)] To execute all this examples at one go, start cspForth and enter

```
take gtk examples
```

on the command line.

For the main class hierarchy you will already find glossary entries with the context sensitive help utility (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 files in the modules directory *dir(1)=<cspForth directory>/user*. See the files **hello.1.4th** and **GtkToplevel/example.x.4th** to get started.

If you are familiar with the German Language you may want to read the article *Widgets zum Anfassen - GUI-Skripting mit Forth und GTK+* in the Forth-Magazin Vierte Dimension 2/2008.

Saving cspForth

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

Warning

cspForth** is **work in progress** and that's much more the case for the GTK+ Interface. **Details may change !

FAQ | Questionary

(this section created because this Wiki seems to lack the concept of a 'discussion page.')

Quote: *The basic underlying idea is to use implicit context switching to assign methods to objects. Before being executed or compiled, an object 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 context switching preludes.*

Preludes look like an interesting concept, but more in-depth description is in order. In Forth, normally STATE and setting global variables are frowned upon. It would seem that setting up a temporary search order is several orders of magnitude more invading than this. Could some examples be given, please? — *marcel hendrix [mailto:mhx@iae.nl] 2008/08/18 22:33*

1. Concerning preludes:

I presented the prelude concept on the euroForth98. The title of the paper was: PRELUDE and FINALE. Implicit context switching based on pre- and post-executed words. Appendix 1 of the paper is outdated. The syntax has changed since then. Today I only use pre-executed words (preludes) and no post-executed ones any more. *(Phone and e-mail are outdated in the paper!)*

2. Concerning STATE … :

I know about the discussion concerning STATE-smart words but I'm not shure what's your concern here. If it's ok for you to use a search order of wordlists and vocabularies, then it should probably also be ok to temporarily change the search order (by context switching words).

Manfred Mahlow [mailto:manfred.mahlow@forth-ev.de] 2008/08/19

Thank you for the reference to the paper. However, the details there aren't sufficient to answer my question above. E.g., the given implementation of `final` as `FORTH` would be wrong if I do something like `ALSO ASSEMBLER variable ape 3 ape !` (The paper also notes this, but doesn't give the details how this is / could be fixed).

The problem with `STATE` (as you of course know), is that it can be used to write words that execute differently depending on its value. When `foo` is `STATE` dependent, this ultimately can lead to `: ape foo ;` not being the same as `: ape [ ' foo ] literal EXECUTE ;`. I was wondering if the potential for this class of problems might be even higher with the prelude concept.

Do you really set and reset the search order at run-time for each word? That would be quite an overhead.

— *marcel hendrix [mailto:mhx@iae.nl] 2008/08/20 22:02*

Do you really set and reset the search order at run-time for each word? …

No, it's only done at interpret time by the outer/text interpreter and does not add a run-time overhead. This might become more obvious when looking at the specification of the Forth text interpreter in DPANS94.

DPANS94, paragraph 3.4 defines the Forth text interpreter of a standard system as follows:

```
Text interpretation (...) shall repeat the following steps until either the
parse area is empty or an ambiguous condition exists:

a) Skip leading spaces and parse a name (see 3,4,1);

b) Search the directory name space (see 3.4.2). If a definition name matching
   the string is found:

   1. if interpreting, perform the interpretation semantics of the definition
      (see 3.4.3.2), and continue at a);
   2. if compiling, perform the compilation semantics of the definition (see
      3.4.3.3), and continue at a);

c) If a definition name matching the string is not found, attempt to convert
   the string to a number ...
```

To support preludes, the subsection b) of the text interpreter definition is changed to:

```
b) Search the directory name space (see 3.4.2). If a definition name matching
```

```
the string is found:

1. if the definition has a prelude, perform the execution semantics of the
   prelude definition;
2. if interpreting, perform the interpretation semantics of the definition
   (see 3.4.3.2), and continue at a);
3. if compiling, perform the compilation semantics of the definition (see
   3.4.3.3), and continue at a);
```

So, when a prelude is assigned to a word/definition, the word/definition gets an extra semantics, that could be called prelude semantics. It's the execution semantics of the preludes definition.

The problem with STATE ...

Assigning a prelude to a word/definition **foo** will not make it STATE-smart but it will however have the effect, that **: ape1 foo ;** will not have the same effect as **: ape2 [' foo] literal execute ;** at compile-time, because when **ape1** is compiled, the prelude is executed before the compile time semantics of **foo** is executed. When **ape2** is compiled, the prelude of **foo** is not executed.

Until now it's my experience, that this is a minor problem, when using context switching preludes.

Thank you for the reference to the paper. However ...

The code from Figure 3 in the paper for the euroForth98 has several limitations

1. The top entry of the search order is changed from the current context to `ascii` to `forth`, when a variable of the type ascii is created or manipulated.
2. When the vocabulary `ascii` is executed as the prelude of an ascii-variable, other vocabularies are still in the search order, so that non-ascii words could accidently be used as ascii methods.
3. If the vocabulary `ascii` is added to the search order with `ascii also`, it's method `variable` could hide the word `variable` in the vocabulary `forth`.

All this could lead to "high quality bugs".

The goal of the code of that days was to briefly demonstrate the basics of using context switching preludes to bind methods to objects. The intention was to trigger some discussion concerning OOP with Forth. The limitations of that code have been overcome by introducing a second search order for classes, interfaces and objects, that does not interfere with the default search order for vocabularies and wordlists. You may want to have a look at the module `oop.4th` in the subdirectory `share` of the <cspForth directory>.

Manfred Mahlow [mailto:manfred.mahlow@forth-ev.de] 2008/08/22

---

[1] *I recently started to write a tutorial for the cspForth GTK+ API. You can find it here.*

Discussion