# Cocoa for VFX64

Content:

## What is this?

Cocoa-for-VFX64 is an extension for a Mac like experience. It includes an interface to the ObjC runtime to deal with Cocoa. Cocoa is the preferred GUI interface for the Mac at the moment.
Cocoa-for-VFX64 also adds  interfaces and functionality not Cocoa related: for instance Quartz interface for 2D drawing, which uses a 'normal' procedural C API. Access to more Apple/Mac technology like CoreAudio, CoreVideo, QuickTime etc. might be available on request.
And some of the additions are Forth related: my pet words and preferred (re)definitions.

## What not to expect!

This extension is not a MacForth, Mach2, Mops etc. like environment. All the ingredients to make one, are there though. But it's outside the capabilities of this author. It is also not an Object Oriented extension to Forth aka OOF.

A tutorial is not included for Cocoa. But in short: Cocoa is the interface to the OS, written in a dynamic (late) binding class based object oriented language: Objective C.

A lot of information can be found on the web at Apple's developer site and others. Some links to support and info in Appendix A. Examples described in Appendix B could be useful for it's usage. In addition, the documentation sections in the source and example files try to explain the how and why.

Much of coco-vfx64 origins and naming choices can be found here:
1984 MacForth Manual
1985 Mach2 Manual

## A quick tour:

The mac folder contains subfolders in which you can find the source files grouped to their specific tasks. The most important:

| | |
|---|---|
| doc | - Documentation |
| utils | - General utilities |
| system | - OS specifics |
| image | - Quartz drawing |
| resources | - The NIB files repository |
| bundling | - App bundle creator |
| cocoa | - The ObjC bridge, the Cocoa interface |
| hotcoco | - Cocoa demo's |

Additional folders with examples and utilities

## Set up:

This demo package mirrors my own setup. Not necessarily your preferred setup, but it should help you with this demo. The essential files for the Cocoa interface can be found in the cocoa folder. The other files are for support and comfort. Together they ought to work out of the box.

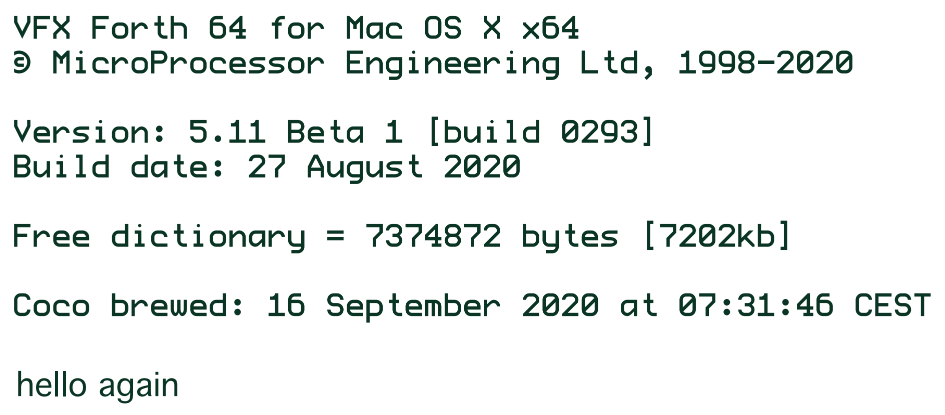
Cocoa-for-VFX64 folder:
Copy the contents of this folder to the VfxForthOsx64/Lib/OSX64/Cocoa folder of your VFX package.

In case you don't have an editor set for use with VFX64, see VFXmanual how. Make sure the command line tools of your editor are installed. Follow the editor's installation instructions.

The main program will be coco-vfx64 which has to be made first:
launch VFX64 in VfxForthOsx64/bin and type or paste at prompt:

```
c" %load_path%" $ExpandMacros dup stripfilename setmacro vfxpath
include %vfxpath%/Lib/Osx64/Cocoa/new-coco64.bld
```

Launch coco-vfx64 in VfxForthOsx64/Bin
You should see something like this:



```
VFX Forth 64 for Mac OS X x64
© MicroProcessor Engineering Ltd, 1998-2020

Version: 5.11 Beta 1 [build 0293]
Build date: 27 August 2020

Free dictionary = 7374872 bytes [7202kb]

Coco brewed: 16 September 2020 at 07:31:46 CEST

 hello again
```

You'll notice an icon is added in the Dock, this represents the Cocoa or NSApp part of coco-vfx64. Every GUI item you add will be dealt with by that side of coco-vfx64. It's the ObjC Runtime's responsibility.

coco-vfx64 is not case sensitive. Upper case is shown here for clarity. BTW all names for the external functions, libraries, frameworks, selectors, classes and so forth are case sensitive when initially declared. After declaration you can use their Forth counterparts with whatever case you prefer.

## Cocoa interface implementation for VFX64:

The Cocoa event-handler run by the application's NSApplication instance, must run on the main thread. Rather than interleave Forth and the ObjC runtime (for instance as an event driven Forth), they are kept apart.

Most Forth's are not fuzzy. So a new task (in VFX64 a thread) is created which will run Forth i.e. QUIT. The main thread, will run the ObjC Runtime. It will execute the NSApplication 'run' method until quitting the application. From within Forth you can now interact with the ObjC Runtime via sending messages to existing classes and extending it by adding classes of your own. The initiating process is HOT.COCO and can be run by STARTER.

The main task is named MAIN. The new task which runs Forth, does that by pretending to be MAIN. Hence it's name: IMPOSTOR.
It inherits all MAIN's user settings including I/O. So, out of the box coco-vfx64 runs in your preferred shell in Terminal as if it is VFX64.

In general the ObjC Runtime and Forth run next to each other independently. There is no explicit event handling or event servicing you have to take care of. It's hands free...
Of course you can interfere when necessary.

## A variation on the GUI version:

As said in Set up, an extra icon represents the app part of our NSApp. This is caused by the default initialization. As of Mac OS X 10.9 Mavericks, you can run NSApp as an accessory as well. This means coco-vfx64 does not appear in the Dock and has no menubar. But you can create windows and other GUI items and run them as you do with the regular application version.

```
NSApp @  /ACC       \ turns coco-vfx64in to a accessory
NSApp @  /APP       \ turns coco-vfx64 in to the default regular application
```

See cocoa/vfxtococoa64.fth for the alternative way to initialize our NSApp.

Some handy GUI interactions:

Load a file:

Type LOAD-FILE at Forth prompt
A Finder dialog ("KiteBox") appears
Choose whatever and click load button
You could navigate to mac/hotcoco and load vfxmenu.fth
It adds an application menu to coco-i4.

Note: It’s no problem at all, to use coco-vfx64 without a menubar. I do without most of the time.

Edit a file:

Type EDIT-FILE at Forth prompt
A Finder dialog appears
Choose whatever and click load button
Editor is launched when not there already
Chosen file opens in new window

Loading from editor:

Type LOAD-EDIT at Forth prompt
Most recent file brought in to the editor with EDIT-FILE is loaded

Loading from Clipboard:

Type LOAD-SCRAP at Forth prompt
Will attempt to load whatever TXT in the Clipboard

## The interface to foreign functions and the ObjC runtime:

Calling external library functions and ObjC Runtime methods are using the same technique based on VFX64 Extern: family of words. A calling system is created in usage hopefully familiar to Forth users. A colon like definition is made, but only the colon-name-stackpicture is necessary:

```
: MYWORD ( n1 n2 -- n3 )
```

That's all. The body of the word or the code executed by the word is already defined in a library of course. Colon itself is not used, but colon-variations are used to distinguish the type of word to create: FUNCTION: COCOA: etc.

The stackpicture or stackmap between parentesis is essential in the word creation. The calling mechanism uses the provided stackpicture to pop the required parameters from the Forth stack(s) and to sent them to the foreign function or method. The return values(s) are pushed on the Forth stack(s).

All coco -* distributions use this interface. Unlike the 32bit Forth systems, the 64bit versions use a second stackpicture for the fp doubles. They will be passed separately from the integer parameters in their own registers, popped from the fp stack.

The interfaces are relocatable after SAVE etc.
No problems to use VFX64 Extern: next to FUNCTION:

See mac/system/function.fth, mac/cocoa/cocoafunc64.fth and mac/doc/c(-rationale.txt.

## The interface to foreign libraries:

FRAMEWORK ( spaces<name.framework> -- )
-- creates 'name' and finds and opens related framework
executing 'name' will make 'name' the current searched framework
Frameworks are a Mac way of accessing libraries, mostly umbrella libraries

FUNCTION: ( spaces<name> <parameterlist> -- )
-- create name, an imported library function name. When executed it expects its parameters on the Forth stack as indicated by the parameterlist and a call is made to the external library function. The parameterlist is similar to a Forth stackpicture. FP doubles are passed via a second stackpicture.

ASCALL: ( spaces<forthname> spaces<name> <parameterlist> -- )
-- similar to FUNCTION: but this external library functions will be called via a Forth name.

GLOBAL: ( spaces<name> -- )
-- create name, an imported library function or variable name. When executed a pointer to this funtction or variable is returned.

Example usage:

```
FRAMEWORK System.framework
System.framework

FUNCTION: abs ( n1 -- n2 )

FUNCTION: acos ( -- ) ( F: r1 -- r2 )
```

## The fun

OSX comes with many libraries installed. Some need Cocoa, others don't.
Have a look in: /System/Library/Frameworks/

Some of which may be of special interest:

| | |
|---|---|
| Accelerate.framework | -- Scientific computing, blas, vectors etc. |
| OpenGL.framework | -- 3D graphics |
| OpenCL.framework | -- Parallel computing using the graphic chips |

Many audio and video processing related frameworks.

Either install Xcode, Apple's development environment, or the CommandLineTools. This will take care of adding the header files to the frameworks. A lot, if not all, information can be found in those files!

The headers for the frameworks are in:
-- Xcode 4 and up (OSX 10.8 - macOS 10.13) :
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.WHATEVER.sdk/System/Library/Frameworks

Installing the command-line tools will add the appropriate header files to /System/Library/Frameworks
Open Terminal and execute:   xcode-select --install

Note: the latest developer tools do not install /usr/include anymore.
This folder can be found in
/Library/Developer/CommandLineTools/SDKs/*.sdk/usr/include

-- Prior versions of Xcode installed the headers in
/System/Library/Frameworks only.

## The interface to existing classes and methods:

COCOACLASS ( <name> -- )
-- like a constant, will cache the class id from name. When executing 'name', return its id.

COCOA: ( <name> <params...> -- )
-- Similar to FUNCTION: but for ObjC messages. Strips the @ character in front of the method name if there. The @ character could be used as 'Cocoa-call' identifier. Not necessary. During execution the selector for the method is send to the receiver.
The parameter list is parsed to create code to pass the parameters with the selector to the receiver.

SUPERCOCOA: ( <name> <params...> -- )
-- Similar to COCOA: but sending the message to receiver's superclass.

COCOA-STRET: ( <name> <params...> -- )
-- Similar to COCOA: but needs an extra structure pointer for the receivers return values.

SUPERCOCOA-STRET: ( <name> <params...> -- )
-- Similar to COCOA-STRET: but for receivers superclass.

COCOA-FPRET: ( <name> <params...> -- )
-- Similar to COCOA: but receiver returns a float.

## Example usage:

```
COCOACLASS NSProcessInfo
COCOA: @processInfo    ( -- NSProcessInfo-object )
COCOA: @operatingSystemVersionString ( -- NSStringRef )
COCOA: @getCString: ( buffer -- ret )

PAD DUP 40 ERASE                 \ setup pad
NSProcessInfo @processInfo       \ get processInfo for current process
@operatingSystemVersionString    \ get OS version as a NSString
@getCString: DROP                \ use pad to convert info into C string
PAD ZCOUNT TYPE                  \ yeah
```

Further examples can be found in the hotcoco folder, see Appendix B.

Note: swizzling-test.fth shows how to change existing method implementations. For the curious and the brave

## The ObjC class making mechanism:

NEW.CLASS ( class-id <spaces name> -- )
-- will setup for a new class called name with class-id as super class

ADD.METHOD ( implementation types name class -- )
-- add instance-method to class under construction

ADD.CLASSMETHOD ( implementation types name metaclass -- )
-- add class-method to class under construction

ADD.IVAR ( type name class -- )
-- add instance-variable to class under construction

ADD.CLASS ( class-id -- )
-- finish building process

## Example usage:

```
\ what the method will do implemented as Forth callback
:NONAME ( rec sel -- ret )    IMPOSTOR'S ." That's all!" CR 0 ;
CB( _int _int )CB-int *simple

\ the arguments type encodings
: TYPES   0" V@:" ;

NSObject NEW.CLASS simpleClass                  \ setup for class building
*simple TYPES Z" simple"  simpleClass ADD.METHOD        \ add a method
simpleClass ADD.CLASS                           \ finish building class

COCOA: @simple ( -- ret )             \ make method usable
simpleClass @alloc @init VALUE me     \ create an instance of simpleClass
me @simple DROP                       \ etc.
```

Further examples can be found in the hotcoco folder, see Appendix B.

Note: the method implementation, here the Forth callback, does have the receiver and selector parameters! Normally you should put them in the parameter list from the callback as the leftmost parameters (exception is when dealing with structure return 'stret' methods). The ObjC runtime passes them round during the method evocation process.
With COCOA:  they're implied, you should _not_ put them in the parameter list, it's done for you.

## The multitasker:

The coco-vfx64 multitasker is a mix of Mach, POSIX and GCD calls.
Grand Central Dispatch (GCD) is optimized for multitasking on multicore CPU's.  Read the source file to see what POSIX calls not to use.

## The interface to tasks, main words:

MAIN ( -- task )
-- task record main task. Runs the ObjC Runtime.

IMPOSTOR ( -- task )
-- task record main task. Runs Forth.

TASK ( spaces<name> -- )
-- create a task record with name. Room for thread handle, xt  and UP.
Leaves record address when executed

ACTIVATE ( task -- )
-- create a thread which will run the code between ACTIVATE and ;
Thread starts immediately. Task record will be filled in during creation.

DONE ( task -- )
-- terminate task. Task record will be cleared.

More words can be found in VfxForthOsx64/Lib/Osx64/MultiOsx64.fth and mac/system/task-control.fth

## Example usage:

```
TASK mytask
VARIABLE accu
: BUMP ( -- )   accu off begin 1 accu +! 1000 ms again ;
: GO ( -- )   mytask ACTIVATE bump ;
```

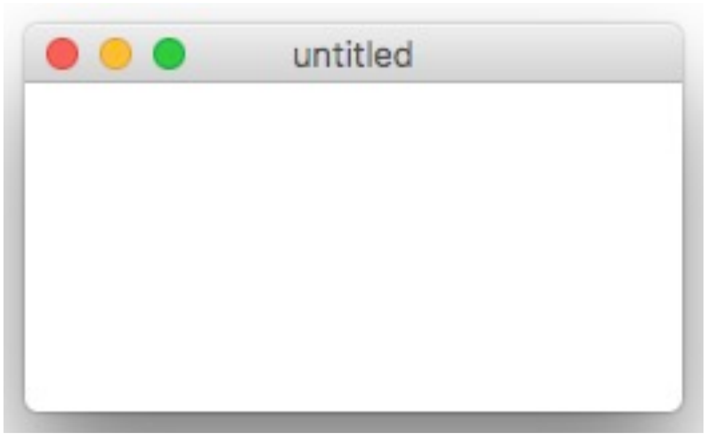
See Appendix B for more (GCD) examples.

The interface to windows, main words:
NEW.WINDOW ( spaces<name> -- )
-- create a window record with name. Fills in record with default settings. Leaves record address, wptr4, when executed

ADD.WINDOW ( wptr4 -- )
-- create actual window and show on screen. Uses settings found in window record.

More words can be found in the cocoa/nswindow64.fth and cocoa/window-utils.fth files.

Example usage:
NEW.WINDOW WIN
WIN ADD.WINDOW



The contentView:
A window has a default contentView. This is the placeholder for the content i.e. text, web pages, control items like buttons etc. You can replace the default view or add subviews. Appendix B has examples for webView, documentView and  widgets (button and slider):
-- runweb-test.frt includes cocoawebview.frt
-- shellview.bld includes sfshellview.fth
-- custom-control.fth

All contentView setting and initialising should be done on the main thread, retrieving information can be done from other threads as well.

MAKE.CONTENTVIEW ( viewref wptr4 -- )
-- executes on the main thread, sets a window's contentView.

The interface to Core Graphics, Quartz 2D drawing:
Include the appropriate cg64.bld load file found in the image/quartz folder
The API to Core Graphics is not an ObjC API!

Note: all coordinates, colours etc. are floating point.

Some (essential) words:
/GWINDOW ( wptr4 -- )
-- initialises windows graphic context

CGCONTEXT ( wptr4 -- cgcontext )
-- retrieves graphic context for drawing etc.

MOVE.TO ( cgcontext -- ) ( F: x y -- )
-- move pen to point. Floating point coordinates!

LINE ( cgcontext -- ) ( F: x1 y1 x2 y2 -- )
-- draw line between points x1y1 and x2y2

DOT ( cgcontext -- ) ( F: x y -- )
-- draw dot at point xy

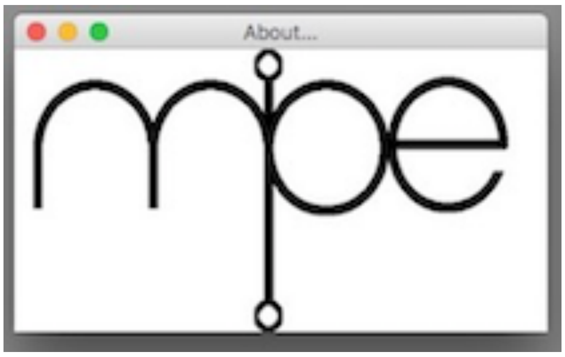
WIPE ( wptr4 -- )
-- clear window

Example usage:

```
NEW.WINDOW ABOUTWIN
S" ABOUT..."    ABOUTWIN W.TITLE

ABOUTWIN DUP ADD.WINDOW /GWINDOW

Z" mpelogo.jpg" ABOUTWIN DRAWPIC
```



See Appendix B for more examples.

## Navigating the file system:

Waypoints are used to find your way during includes. They set the working directory to predefined paths.
Predefined waypoints are: VFX, MAC, Home, and Desktop. Of course you can change them to fit your situation in utils/waypoints.fth

## Example usage:

```
PUSHPATH              \ save current working directory
MAC                   \ make mac working directory
INCLUDE sound/sc.frt  \ load from here
POPPATH               \ restore and continue
```

## Extending CB( ... )CB-

The sequence :NONAME ... CB:  is used a lot, so FORTHCALL: is added which does the same in one word.

## Compiling in callbacks:

...

## SIGINT exceptions:

Optional is to include system/mysigint.fth which installs a handler to catch SIGINT signals thrown at coco-vfx64. For instance by hitting the CMD. keys. See the installer file, for what to expect and what _not_ to expect.

## Turnkey proof:

The Cocoa interface must be turnkey proof. The word /COCOALINKS takes care of relinking all used classes and methods to their proper class id's and message selectors.

## Missing features:

Although very useful in its current form, there's no doubt room for improvements and additions. There's rudimentary adding of GUI objects like menu's and controls programmatically. However these are still more easily build with Apple's Interface Builder. Problem with this application is the ever changing layout. A description how to use it, would be outdated the moment you read this. However an attempt is made in Appendix C.

## OSX and macOS special cases:

OSX 10.9 Mavericks introduced a new feature: App Nap. It is used by the OS to put an application to sleep under certain circumstances.
The NSApp part of coco-vfx64, the main thread/task, is not doing anything for us when not having any GUI or main thread related things going on. Apparently the OS sees this as the whole application idling, and puts the app napping. With consequences for Forth.
The code to deal with this and more info can be found in no-nap64.fth

OSX 10.10 Yosemite spills some debug information in our repl when putting the file navigation box (kite-box) on the screen. This debug information is directed at Apple engineers while developing the OS. Someone forgot to set/reset a switch... This seems to be fixed in macOS 10.12 Sierra.
The temporary fix can be found in yosemite.fth

OSX 10.11 El Capitan caused a rewrite of the ObjC interface related to creating windows on a secondary thread. The initialization of NSWindows and subclasses should be done on the main thread.
This avoids weird live update behaviour when resizing a window.
When you need to execute methods on the main thread, you can use FORMAIN which is shorthand for the NSObject message performSelectorOnMainThread:withObject:waitUntilDone:. Your method will be queued on the main event-loop. Similar is the word PASS which will take a Forth xt with any number of parameters and queues it on the main event-loop. Usage concerns UI related code and working with NSViews and its subclasses. Those are better done on the main thread. Prior to EL Capitan, this wasn't an issue! See forthclass.fth and elcapitan64.fth

macOS 10.12 Sierra has added more NSWindow specific messages to the preferred-on-the-mainthread list. They concern some of the property settings. See sierra64.frt for more information.

macOS 10.13 High Sierra has not shown any cause for changes regarding the OS and ObjC runtime yet. A few words were added to GOES.FORTH to fix a weird glitch when turn-keyed.

macOS 10.14 Mojave and mac OS 10.15 Catalina have not shown any cause for changes regarding the OS and ObjC runtime yet. See appearance.fth in hotcoco for the Dark and Light Mode settings.

All the OS specific patches are included last in the mac-vfx64.bld load file. Default is to have them all in. So far they're harmless in OS versions which don't need them.

## Appendix A: Some support and info

Useful with Cocoa:
Tutorials from Cocoa Dev Central for Cocoa and ObjC
Mike Ash blog for Cocoa, ObjC and Swift, very informative
StackOverflow with tags like cocoa, osx, objective-c
Apple obviously, but grab info while links are still valid (even this one!), computer-rage has its origins here...

Take care when in Apple:
The graphic system works with CGFloats, not integers! The size depends on the application being 32 or 64bit. Thus SFLOATs in SwiftForth and VFX, DFLOATs in iForth64 and VFX64.

In general, when passing structures to external functions, structures of up to 4 members are passed by their individual values. So not a pointer to the structure. Thus a rect structure (CGRect or NSRect) is passed as 4 CGFloats, i.e. 4 parameters in stead of 1.

Point and size structures are returned as two CGFloats on the F: stack. Rectangle structure values are returned in a passed-in pointer to a rect structure. The infamous hidden structure parameter used by C compilers! You have to provide this pointer and fetching it's contents yourself.

Cocoa provides COCOA-STRET: to help with the latter.
See nswindows.frt and window-utils.frt for usage: @frame etc.

In the procedural CoreGraphics, FUNCTION: is used for externals. When encountering a case of a returned rect structure, put the return-structure as the left most in-parameter in the declaration.
C declaration:
CGRect CGContextGetPathBoundingBox(CGContextRef c);
Forth declaration:
FUNCTION: CGContextGetPathBoundingBox ( CGRect CGContextRef -- ret )

For passing rectangle structure members in coco-vfx64, see Appendix D. These CGFloats are passed via the C stack, a trick is shown how.

You'll see int<->fp mucking around, but much less than in the 32bit coco versions.
When in Rome...

## Multithreading and the GUI:

Check this Apple Threading Guide. Still it says windows can be created on a secondary thread here, while in OSX 10.11 El Capitan and up you better not.

## Coming from other Unix/Linux:

Porting UNIX/Linux Applications to OS X
Porting Drivers to OS X

## More information on the Cocoa interface for VFX64 on the Mac:

A lot of files... Are they all necessary?
Actually no, they're not. Only two(!) files are essential.
You could use these two for a Cocoa interface in your own words...
cocoacore.fth for the system calls to communicate with the ObjC Runtime.
cocoalaunch-main-thread.fth for the integration of the ObjC Runtime.

## Take care with Cocoa:

To quote from Apple’s Porting UNIX/Linux Applications to OS X document:

“You should be careful when writing code using Cocoa, because the same constructs that make it easy for the developer tend to degrade performance when overused. In particular, heavy use of message passing can result in a sluggish application.

The ideal use of Cocoa is as a thin layer on top of an existing application. Such a design gives you not only good performance and ease of GUI design, but also minimises the amount of divergence between your UNIX code base and your OS X code base.”

Obviously, the same applies for a Forth application and a Cocoa GUI interface.

## Portability:

The Cocoa interface runs on several 32 and 64 bit Forth systems nearly unaltered. This means the code doesn't grab deep in the system. Unlike coco-sf and coco-vfx, there is no system specific ObjC caller in coco-vfx64.

## Alternatives:

1. iMops is a continuation of the legendary Neon development system on the classic Mac. It's 64bit Intel native and Cocoa based.

2. For graphical output, say plotting functions, one doesn't need this Cocoa interface. AquaTerm provides a Cocoa plot-window and easy to use API.

3. Not Cocoa but Carbon (the previous Apple GUI) is MFonVFX. For legacy Mac Forth code or development for the now or near future, very worthwhile! Note: a 64bit future version should not be counted on.

## Appendix B: Examples

### Cocoa in action, hotcoco folder:

simple-example.fth
Prints process name using Cocoa calls.

nsalertpanel.fth
Show modal alert box, using a deprecated way of doing this. Notice how the ObjC runtime complains in the Forth console.

nsalert.fth
A correct way to display alerts.

nsalert-expanded.fth
Elaborated version, adding special features.

simple-class.fth
Use Forth to build an ObjC class.

delegate-test.frt
Change event-handling using delegation, create a subclass for delegation.

self-delegate-example.fth
Change event-handling using NSWindow instances itself as delegate.

swizzling-test64.fth
Change event-handling by changing method implementation, not real method swizzling, but somewhat similar.

cursor.fth
Hide and show the mouse cursor. Very useful with presenting graphics.

mouselocation64.fth
Get mouse coordinates using NSEvent.

cocoa-focus.fth
Several kiosk modes for focussing on work, hiding stuff when projecting etc.

launches64.fth
Launch the default applications when opening files or URL’s.

simplemenu.fth
Create and add a rudimentary menubar without the use of a NIB.

vfxmenu.frt
Add a menu to coco-vfx64, using NIB file and actions defined in a Forth created ObjC class.

included-menu.fth
Add an -Included- menu to coco-vfx64 menubar programmatically.

appearance.fth
Set appearance mode for GUI elements in macOS 10.14 Mojave.

testcontroller.fth
Build a controller using a NIB file and its actions defined in a Forth created ObjC class.

custom-control64.fth
Programmatically build and show a controller object without using a NIB file. Interesting is how you can add methods to an existing class.

mycontroller.fth
Programatically set GUI element from a NIB file. Also shows the preferred loading from a NIB file.

mybuttonstagged.fth
Another GUI example with a NIB file. It uses tagged UI items for simplified action access.

runweb-test.fth
Browse the web with WebKit. Uses a so called HUD navigation bar.

shellview64.bld
Includes necessary files to create a Cocoa console for coco-vfx64, a toy application.

hello.fth
Relaunch coco-vfx64.

Simple Quartz in action, image/quartz folder:
cg64.bld
Initial load file for some necessary Quartz functionality.

cgcontext-draw-test.fth
Simple example, splashing dots or lines in a window.

cgcontext-squares-task-test.fth
A task draws coloured squares in a borderless window.

cgimage-slideshow-test.fth
Perform a slide show. Provide your own image set!!

cgpath-test.fth
Trivial exercise using CG paths.

randomwalks.frt
Plot our random steps in a window.

Grand Central Dispatch examples in system and scheduling folder:
gcd-timers.64fth
Use GCD for many independent timers.

cause-gcd64.fth
Cause words to execute at given times. Famous scheduler from STEIM.

fork64.fth
Parallel execution on multicore machines using GCD.

spawn.fth
Not GCD but POSIX. Spawn a word in an anonymous task/thread.

realtime-priority64.fth
Tread priority setting. In pursuit of controlling the Mach scheduler.

GDB and LLDB debugging in mac and system folders:
applescript.fth
Apple script interface.

mach-task.fth
Mach kernel task and thread stuff.

gdb.fth
AppleScript to launch gdb with coco-i4 attached.

lldb.fth
AppleScript to launch lldb with coco-i4 attached.

gdb-debug.ldr and lldb-debug.bld load-files.
Also see the info in mac/doc/debugger folder.

# Appendix C: How to create GUI items with Interface Builder and Forth

As an example create a menu for coco-vfx64. Two things are needed: a nib file describing the menu's and a Forth source file implementing the actions called by some of the menu's.

vfxmenu.fth (in hotcoco) is a Forth source file, wherein an ObjC class is created which acts as a delegate class to deal with some menu functions: VFXMenuClass.
Note: no need to assign it as a NSApplication delegate, the OS takes care.

The following instance methods are added to this class:

```
    menuQuit:
    menuOpenFile:
    menuLoadFile:
    menuLoadEdit:
    menuVerbose:
    menuAbort:
    menuHelp:
    menuHelp2:
    menuHelp3:
    menuHelp4:
```

These methods will serve as so-called IBActions for the menu handling.
For now the connection between this class and the actual menu items is established with Interface Builder.

Better would be to do this programmatically.
The reason is that IB changes and possibly could turn in to ObjC usage only... The disappearance of IB as a Xcode independent program since Xcode v. 4 being a point in case. The problem is not the integration in to Xcode, but the added dependence on (source) code!

In IB3 you add methods to a class object, by just filling in some names in text fields. There's no ObjC in sight, very convenient if you're not using ObjC.

With the IB component in Xcode4 and later you need a source .m or header .h file to add methods to a class object. So how do you do this with non C derived languages?

Until a better way is found, use some cut and paste:

1. Create file named after your delegate class. Give it .h extension.
   example: VFXMenuClass.h
2. Add the template header stuff, with the wanted action prototypes.
   example:

```
//
//  VFXMenuClass.h
//

#import <Cocoa/Cocoa.h>

@interface VFXMenuClass : NSObject <NSApplicationDelegate>

- (IBAction)menuQuit:(id)sender;
- (IBAction)menuOpenFile:(id)sender;
- (IBAction)menuLoadFile:(id)sender;
- (IBAction)menuLoadEdit:(id)sender;
- (IBAction)menuVerbose:(id)sender;
- (IBAction)menuAbort:(id)sender;
- (IBAction)menuHelp:(id)sender;
- (IBAction)menuHelp2:(id)sender;
- (IBAction)menuHelp3:(id)sender;
- (IBAction)menuHelp4:(id)sender;


@end
```

   Save it.

( obvious in your own implementations
  change VFXMenuClass to your class name
  change all method names between (IBAction) and (id)
  don't forget all names should end with a colon :
  and delete all declarations you don't need )

3. Launch Xcode
   In File menu hit New, choose File and then choose the appropriate template from the presented dialog.
   Here:  main menu
   Follow instructions and a file mainmenu.xib will be created.

   An Interface Builder worksheet will be presented. If not, go to File menu and choose Open... select the just created mainmenu.x

At the left you'll see a list of icons representing the objects you use. The middle part is the worksheet.
At the top right you see a series of small icons, representing amongst them the Identity Inspector and the Connections Inspector, which you'll need later on.
The bottom right has a series of icons representing the libraries of things to add to your project. You select the object library.

4. In File menu hit Add Files... and select your .h file navigating with the presented kite-box.

5. From the object library, drag an object icon (blue cube) to work field. This will be the delegate class.

6. Select blue cube which is now in the list on the left and select the Identity Inspector.
   Fill in the name of your delegate class in the class name field.
   Here:  VFXMenuClass

8. Select the blue cube icon, now named after your delegate class and select the Connections Inspector. You should see the added 'received actions'.

9. Now construct/adapt your GUI, here the main menu. Play around selecting, dragging , moving/copying, deleting the objects in the worksheet and/or manipulate the Main Menu object in the list at the left.

   NewApplication menu: — leave as is.
   File menu: — rename New to Open File...  using the Identity Inspector, add Cmd O as shortcut.
   — rename Open...  to Load File..., add Cmd L as shortcut.
   — rename Open Recent to Load from Editor, add Cmd I as shortcut.
   — copy the horizontal spacer between Open File... and Load File...
   — rename Close to Verbose.
   — delete the rest.
   Edit menu: — leave as is or keep what's necessary for you.
   Format menu: — rename as Misc and keep one menu item named Abort. add <esc> as shortcut.
   View menu: — delete it.
   Windows menu — leave as is.

Help menu — copy NewApplication Help 3 times (4 help items total).
— rename them top to bottom:
VFX Reference Manual
Cocoa for VFX Info
ANS Forth Standard
Forth Handbook

Assign the appropriate actions to the menu items.
Click on the NewApplication menu to have the menu's drop down. Now select the blue cube called Menu Class in the object list at the left. Go to the Connections Inspector and Ctrl click (hold button down) the MenuQuit: action and drag the appearing connection line to the Application quit menu and let go.
Connection established. Do this for all the actions defined in the VFXMenuClass.
The menu items we didn't touch, call their default actions when hit.
So the font setting is taken care of by the OS, we don't need to assign its actions. Others only do something if they have a context for their actions.

10. When done, save and export to a nib file.
    example: vfxmenu.nib

Note: when you load this nib file in Forth, you could see a message like:
[QL] Can't get plugin bundle info at file://localhost/Applications/Xcode.app/Contents/Library/QuickLook/SourceCode.qlgenerator/

Don't worry, the nib stuff is loaded alright. Nib files created with IB3 don't have this, it's something related with Xcode4:

/Xcode.app/Contents/Library/QuickLook is missing in a default installed Xcode4.

Solve this if you have Xcode3 still around.
Copy the (Xcode 3 version) Xcode.app/Contents/Library/QuickLook directory to (Xcode 4 version) Xcode.app/Contents/Library/

## Appendix D: Compiler Quirk

It looks like there's a bug or issue with the Apple compilers wrt Rectangle structures: NSRect and thus CGRect. According the 64bit AMD ABI one can pass the parameters in their fp registers. In stead Apple passes them in memory through the stack. This means we'll have to use a little trick in iForth to force the calling system to push the rectangle members up the C stack.

Trick: make sure there are enough  parameters inside the ( ... )  to fill the registers. Use dummies if necessary. Then add the rect members. This will ensure that the rect members will be passed through the stack. The ABI says there are 6 registers for the 6 first calling parameters, the rest follows on the C stack.

Remember that COCOA: and family have the 2 hidden parameters RECEIVER and SELECTOR added. So there are 4 registers left to fill before the C stack is used.

As an example:
setFrame:

One expects it to be:
```
COCOA: @setFrame: ( -- ret ) ( F: x y w h -- )
```
But that doesn't work.
The solution used in coco-i4:
```
COCOA: setFrame: ( dummie1 dummie2 dummie3 dummie4 x y w h -- ret )
: @setFrame: ( -- ret ) ( F: x y w h -- )    4 COCOA.DFIX setFrame:  ;
```
See FIX.WINDOW and ADD.WINDOW.

CGRectangles have the same issue, fill up with dummies before adding the rectangle fp values. See mac/image/quartz/quartz-cgdrawing.fth

ABI-call-extras64.fth for the implementation and more info:
DFIX for regular C calls.
COCOA.DFIX for Cocoa calls.

BTW no problems/issues with 2 DFLOATS containing structures like NSPoint/CGPoint and NSSize/CGSize or the 4 member NSColor/CGColor structures. They're passed using the FP registers.

## Appendix E: Turnkey an Application bundle

A toy application as crash course: VFX-APP
VFX-APP is a coco-vfx version build by including new-toy64.bld found in the mac parent folder. Step by step, it shows how to create an app bundle and how to launch without Terminal involved.

Another example of a turnkey process is explained in new-maps64.bld.
A bundled application is build which opens a webkit page linked to the Open Street Map url: an alternative to Apple's Maps.

See the read_me_first.txt in mac parent folder, how to proceed.

Note: none of the created app bundles are signed or notarized.
See doc/code-signing

## Postface:

The Cocoa ObjC interface was developed with the following in mind:
-- Similar in usage as other interfaces in MacForth and Mach2. Think of things like NEW-XXX ADD.XXX etc.
-- No hacks, no shortcuts. Should survive OS upgrades.
-- Try to stay close to ObjC names and procedures. Makes Apple examples easy to follow and to implement.
-- Easy installable on several Darwin Forth systems: iForth, VFX, SwiftForth
Breaking the rules at times is inevitable.

The end result looks like an OOF extension. However, it should be noted that the interface deals with the ObjC Runtime and not with the Forth system!

The Cocoa interface (2008-2020) runs in CarbonMacForth, iForth, iForth64, SwiftForth, VFX, MFonVFX and VFX64. Current version is ObjC 2 Runtime compatible and runs in Mac OSX Snow Leopard 10.6 up to and including macOS 10.15 Catalina.

Many thanks to Doug Hoffman, Charles Turner, Bruno Degazio, Marcel Hendrix, Ward McFarland, Stephen Pelc, Leon Wagner, Marc Simpson and George Kozlowski for their much valued and appreciated contributions.

Anyway, feedback is very welcome and I love to hear about usage.

Huissen September 2020 (include cocoawebview.fth run the proper MAPS)

## License