# iMops 2 internals (in progress)

N. S.

November 2016

## Contents

**1 Objective-C** **3**
- 1.1 introduction . . . 3
- 1.2 NON-OO basis of the OO system of ObjC . . . 3
- 1.3 Two words . . . 4
- 1.4 ObjC Instantiation . . . 4
- 1.5 Implementation of ObjC Method . . . 5
  - 1.5.1 addendum on parallel execution . . . 6
- 1.6 How to Install your Implementation . . . 7

**2 Under the hood of iMops** **7**
- 2.1 Stacks . . . 7
  - 2.1.1 Data Stack . . . 7
  - 2.1.2 Floationg Point Number Stack . . . 7
  - 2.1.3 Return Stack . . . 8
- 2.2 Register definition . . . 8
  - 2.2.1 Integer registers (GPRs) . . . 8
  - 2.2.2 GPR Usage . . . 8
  - 2.2.3 Floating Point Number registers . . . 9
- 2.3 iMops Dictionary . . . 9
  - 2.3.1 Two parts dictionary . . . 9
  - 2.3.2 Word List . . . 9
- 2.4 XT and Relocatable address . . . 9

**3 iMops and Object Orientation** **10**
- 3.1 NEON model . . . 10
- 3.2 Reference and vTable Binding . . . 11
- 3.3 Inheritance and Instance variables . . . 12
- 3.4 Public Instance Variable and Static Instance Variable . . . 13
  - 3.4.1 Public Instance Variable . . . 13
  - 3.4.2 Static Instance Variable . . . 13
  - 3.4.3 Curly braces syntax on Ivar . . . 14
- 3.5 Public Object as an ivar of another Object . . . 15
- 3.6 Temporary(Local) Object . . . 15
  - 3.6.1 A bit more about CLASSINIT: . . . 16
- 3.7 Reference, Heap Object and Garbage Collector . . . 16
  - 3.7.1 Heap object reference . . . 16

3.7.2 Reference Ivars and Temporary reference . . . 18
3.8 RECURSE for Method . . . 18

**4 Using and Building Dynamic Library** **19**
4.1 Public Library . . . 19
4.1.1 introduction . . . 19
4.1.2 ObjC-Framework . . . 19
4.1.3 Framework and FrwkCall . . . 20
4.1.4 Framework and ObjC-Framework . . . 20
4.2 Private Library . . . 20
4.2.1 introduction . . . 20
4.2.2 Library . . . 20
4.3 Building Framework . . . 21
4.3.1 Export a Word . . . 21
4.3.2 Initializing Dynamic Library . . . 22
4.3.3 GUI Elements in a Dynamic Library . . . 22
4.3.4 Garbage Collection in a Dynamic Library . . . 22

**5 Installing a Stand Alone Application** **22**
5.1 Differences of Installed Application from iMops Environment. . . . 22
5.1.1 No Code Generator by Default . . . 22
5.1.2 Don't Save Absolute Memory Address as a Value . . . 23
5.2 Miscellaneous Notes . . . 23
5.2.1 Special Initialization . . . 23
5.2.2 Event Loop . . . 23
5.2.3 Timer . . . 23

**6 Features peculiar to iMops** **24**
6.1 Local Section . . . 24
6.2 Method Declaration . . . 24
6.3 Changes in Method Call Compilation . . . 25
6.4 Getting Information . . . 25
6.4.1 LOCATE and others . . . 25
6.4.2 Words, Classes, Objects and WordsWith . . . 26
6.5 Special Prefixes for Values and Locals . . . 26

**7 Code Generation** **27**
7.1 Machine Code Generator . . . 27
7.2 Inline Definition and Execute . . . 28
7.3 Header Data . . . 29
7.3.1 Word of Data type . . . 29
7.3.2 Executable . . . 30

**8 Mach-O Executable File Format and Virtual Memory Mapping** **30**
8.1 Executable File Format . . . 30
8.2 (Virtual) Memory Mapping of iMops Development Environment . . . 31
8.3 Memory Mapping of an Installed Application . . . 32
8.4 Memory Mapping of a Dynamic Library . . . 32

# 1 Objective-C

## 1.1 introduction

iMops is a Cocoa application, and designed to build Cocoa applications. The reason why it is not a Carbon Application like PowerMops is that most of Carbon GUI functions are 32bit only, while iMops is 64bit only. Objective-C (objC) language itself is designed simple, I think. But Cocoa (the main part is AppKit) is huge and complicated library. The PDF documents on AppKit and Foundation frameworks have totally 6660 pages. In spite of the volume, the descriptions of classes or methods are sometimes too brief.

The GUI class system of iMops is intended to hide objC-bridge part and make it possible to write a program without care for Cocoa details. But, in view of the richness of Cocoa library, large part of the task to write iMops class library will have to be handed to users (as far as iMops can get any users). So, it will be necessary to give some descriptions on the objC-bridge part of iMops.

## 1.2 NON-OO basis of the OO system of ObjC

In order to know the fundamental mechanism of OO support in Objective-C language, books to be read are "Objective-C Runtime reference" and "The Objective-C 2.0 programming Language". You can find them on Apple's Developer Library site.

The method binding in Objective-C is dynamic (late) binding. ObjC uses some plain C functions to bind the selector to the method at runtime. Those are: **objc_msgSend()** (for normal methods), **objc_msgSend_stret()** (for methods whose formal return value is a data structure). The numbers and types of parameters of those functions are indefinite because they can call various methods. The syntax of method call is:

```
receiver selector param1 param2 ... objc_msgSend
```

Two implicit parameters, "receiver" and "selector", must be added to the formal (explicit) parameters of the method.

**Receiver** may be either objC-object pointer (instance method) , objC-class pointer (class method) or objC-superclass data structure pointer (sending message to super: not implemented in iMops).

**Selector** is a registered objC-method selector pointer, which can be got by a word defined with a word "`ObjC_Selector`" in iMops (see next section).

To fix the indefiniteness of the number of the parameters of the binding function before its execution, iMops provides a word "`ObjCMethod`". The syntax is similar to that of SYSCALL:

```
ObjCMethod  BinderName { rec sel p1 .. %fp1 -- ret }
```

"`BinderName`" can be considered as a normal word name. The significant part of the definition is to inform the compiler of the number and types (fp or integer) of the parameters. I think, it may be a good convention to give such a "`BinderName`" that the number and types of parameters can be inferred from it.

By the way, the number of FP parameters in syscall or ObjCMethod is restricted within 8. Normally 8 seems to be enough. But this spec (bug) should be fixed someday.

The function "`objc_msgSend_stret()`" corresponds to a trick in Mac OS X ABI. That is: When the formal return value of a function is a data structure, not the pointer, the data field of the struct should

be passed to the function as the left most parameter via the pointer. Then the return value will be set in the passed data field as the side effect.

iMops word that corresponds to "`objc_msgSend_stret()`" is **ObjCMethStret**. Typical form of the definition will be:

```
ObjCMethstret BinderforStRetMeth { ^ret rec sel p1 ... -- }
```

The pointer of the data structure to be returned should be passed as the **first** parameter.

However, as for 64bit functions (that is, in our case), a struct return value whose length falls within 16 bytes will be returned as values by using packed one or two registers (those are pushed onto the data stack using one or two cells in iMops.) On details of ABI for C-function call, see Conceptual ABI documentation.

## 1.3 Two words

iMops defines two bridge words in order to make it simple to use objC classes, objects or methods. Those are **ObjC_Class** and **ObjC_Selector**. The usage is :

```
ObjC_Class theClassName "NSSome"

ObjC_Selector theSelectorName "objcSelector:name:"
```

These code should be put in execution mode, like value or variable declaration. `theClassName` and `theSelectorName` will become the names of ObjC class and selector in iMops respectively. You can give `theClassName` or `theSelectorName` part as you like, while the string between double quotations, which is the ObjC class or selector name, must be one of the correct objC class/selector names which are case-sensitive. `ObjC_Class` and `ObjC_Selector` are special `CREATE-DOES>` words. A word defined through one of those words will get the external data pointer at runtime and store the copy in its own `CREATE`d data area. From the second call on, the pointer will be simply fetched from the data area.

**Remark**
You can see in a source code file "COCOA_STUFF" in "new_intelport" folder, a lengthy list of Cstrings and ZVALUEs for class and selector pointers, and long objC initialization word defined. `ObjC_Class` and `ObjC_Selector` bind those three (Cstring, ZVALUE and initialization) into one.

## 1.4 ObjC Instantiation

To create ObjC object, you can use generic allocation method. The selector pointer is stored in a zvalue "**genAllocSel**" at launch time of iMops, so that programmer can use it at any time. The method is a class method, and takes no parameters other than receiver and selector, and returns the pointer of a newly allocated objC instance of the receiver class. So the code will be like:

```
ObjC_Class NSWindowClass "NSWindow"

NSWindowClass genAllocSel 0-1msgSend ( -- objcInst )
```

Then, the pointer of the newly allocated window instance will be put on stack. "`0-1msgSend`" is a "`ObjCMethod`" word already defined in iMop nucleus. "0-1" means getting no parameter and returning one integer value.

Generally, objC object need to be initialized before use. There is a generic initialization method for every class, and the pointer of the selector is stored in a zvalue "**genInitSel**". The method is an instance method, so the receiver should be the newly allocated instance. It returns the pointer of initialized object. So the code including allocation will be:

```
ObjC_Class NSWindowClass "NSWindow"
0 zvalue mywindow
NSWindowClass genAllocSel 0-1msgSend GenInitSel 0-1msgSend -> mywindow
```

or like that. This code corresponds to following in ObjC.

```
NSWindow* mywindow = [ [NSWindow alloc] init];
```

iMops code is relatively lengthy. But you could see what ObjC compiler is doing if adding square brackets to iMops code and replacing selectors by their true selector names:

```
[[NSWindowClass alloc 0-1msgSend] init 0-1msgSend] -> mywindow
```

Generic init method will initialize the instance variables of the object by reasonable values, but not by practically usable values. So most of objC classes have special initialization methods. Usually, initialization of an object will be done via such methods.

As for instance method, you can send message to the object stored in a zvalue, like

```
mywindow ShowinSel 0-0msgSend \ show mywindow
```

(though just after sending init message the window size is (0 , 0). )

Source code files, "COCOA_STUFF", "WindowClass", "FileClass" or "MenuClass" can be taken as sample code for ObjC message sending.

Fundamental reference books for Cocoa programming are Application Kit Framework Reference and Foundation Framework Reference.

You can see a table of ObjectiveC Classes, Selectors and Message binding functions already defined in iMops, in a text file "predefinedObjC" in "source" folder.

## 1.5 Implementation of ObjC Method

In Cocoa programming, we sometimes need to implement Objective-C method. Especially, implementing delegate method is very important for event handling.

Generally speaking, there are two ways to modify the standard behavior of an objC object on a method. One is to create objC subclass and implement subclass' method that may contain message sending to SUPER. Another is to use a delegate method. A way to create objC class on iMops is not implemented yet (that will be implemented some day, though). However we can make use of the delegate method way.

Delegate in objective C means an instance independent from the delegated instance, and add some behavior to the event handling of the delegated instance. When certain event concerning the delegated object is happen, some corresponding message is sent to the delegate object by OS. By implementing delegate method, we can add some behaviors to the event handling by the delegated object. iMops is implementing `applicationWillFinishLaunching:` and `applicationWillTerminate:` delegate methods to read/save console text file. Delegate methods of this type are called **Notification**. Additionally,

iMops uses delegate methods to implement control action (Menu and Button) and window closing handling.

Method implementation from iMops is defined as a callback. There are three pairs of words in iMops to define a callback.

One is `@IMPLEMENTATION`–`@END`. This is a kind of mimicry of Objective-C code block definition. The syntax is

```
@IMPLEMENTATION myObjCMethod (( rec sel p1 -- ret ))
[ Mops code ]
@END
```

`(( -- ))` expression is not a stack comment. This is the parameter definition. But the parameters are entered not as named parameters but as stack parameters. And "`[ Mops code ]`" should put the return value on the stack, if any.

Another pair is `:CALLBACK`–`;CALLBACK`. This pair is a synonym of `@IMPLEMENTATION`–`@END`. In defining Carbon or CoreFoundation callback, `:CALLBACK`–`;CALLBACK` word pair will look more suitable.

memoir

In contrast to "`:CALLBACK`" in PowerMops,
`' NewXXHandlerUPP ' DisposeXXHandlerUPP`
preamble is not necessary in iMops. Those New-Dispose function pairs were necessary for Transition Vector technology, by which a callback function could have its own data field (though it is normally the global data area for the fragment where the function is implemented). But those New-Dispose function pairs were, IIRC, doing nothing already since PEF Carbon stage, where the transition vector technology didn't seem to be used. However, similar technology now has been introduced in Mac OS X, as Block-Closure feature and GCD (Grand Central Dispatch). Those could be said to be a revived transition vector!

Callbacks defined by the two word pairs above are supposed to be executed in the main thread. But Mac OS X is inherently preemptively multithreaded, so that some callback may be executed parallel to the main thread of an application. Another word pair is for parallel execution. That is `:MT-CALLBACK`–`;CALLBACK`. The syntax is same as above. A callback word defined with `:MT-CALLBACK`–`;CALLBACK` will set its own integer/fp stacks on entry. Stack space is 100 cells for each.

Generally, data should not be passed through data stack between callbacks even if both callbacks are executed in the main thread. For ordinary Mac applications are event driven, so that the programmer cannot control the execution order of event handler callbacks.

### 1.5.1 addendum on parallel execution

It is difficult to make a general rule to decide what method or function should be executed in main or subsidiary thread. But from some experience, AppleEvent-like features, say, launch application, open document etc., seem to be executed parallel to normal event handlers like activate window, keydown, menu selected etc.. From this point of view, when an AppleEvent handler you implement will invoke some internal event, and the handler of the latter event is also implemented by you, you should define at least one of them with `:MT-CALLBACK`.

`:MT-CALLBACK` word is simply a multithreading capable callback, and not necessarily executed multithreaded. So you can use `:MT-CALLBACK` word for implementing any Objective-C methods or C functions in your application, if maximun 100 cells datastacks are enough for it. But then relatively large memory area allocated for data stacks of the main thread (80kb) will be left unused. It would be memory inefficient. Moreover, there are two restrictions in `:MT-CALLBACK` word except for stack spaces.

1. `:MT-CALLBACK` word shouldn't DIE. Of course, all of your words in your application generally shouldn't DIE in running. But when a `:MT-CALLBACK` word DIEs, iMops will crash immediately. For, in contrast to main thread callback, `:MT-CALLBACK` doesn't set return address to which process should return on DIE.

2. `:MT-CALLBACK` word shouldn't contain interpretation like string EVALUATE. For interpretation word always checks stack overflow while the base address of the bottom of the data stack is always set at that for the main thread. As the result, interpretation in `:MT-CALLBACK` word will DIE as stack overflow, then crash.

These may be really not restrictions in a normal application. But it would be better to mix main and subsidiary threads callbacks.

### 1.6 How to Install your Implementation

Callback word name can be decided as you like. It need not to be same as the method or function name to be implemented.

For method implementation, syscall function "`class_addMethod()`" or "`class_replaceMethod()`" will be used. If the method you are implementing is empty, use the formar. If the method you are implementing has already its content and you need to replace it with your implementation, use the latter. For implementation of a delegate method, `class_addMethod()` is usable. By `class_addMethod()`, you can also add a new method to an objectiveC class. In order to get the function pointer of your callback, use a special prefix word "`@IMP>`", instead of "`[']`".

You can find more detailed description on that in Apple's documentation, Objective-C Runtime Programming Guide and Objective-C Runtime Reference.

## 2 Under the hood of iMops

### 2.1 Stacks

In iMops, Data stack and FP stack are both allocated in the system stack frame for the main thread of iMops process. Return stack is just the system stack frame. As the result, the absolute value of return stack pointer has no meaning, for it will depend on the system state at the time of the event handler call. However, the relative value has definite meaning while our process (words) is running.

System stack frame is allocated at very high part of the virtual memory space and grows downward. Our three stacks also grow downward. So the top item of the stack is at the lowest memory address among all the stack items.

#### 2.1.1 Data Stack

8192 cells (1 cell = 8 bytes) are allocated for Integer Data Stack for the main thread. That will be large enough for normal programming. Data stack next to the FP stack in the virtual memory. Data stack Pointer is stored in RBP (R5) register, named rDSP in iMops.

#### 2.1.2 Floationg Point Number Stack

2048 cells are allocated for FP stack. All FP numbers are supposed to be double (8bytes) numbers. So widths of FP cell and Integer cell are same. The area of FP stack begins from the bottom of the stack frame, that is, it is in the highest part of the frame memory. It is small compared with integer stack. But since address data or flag value is always pushed on integer data stack, the duty of FP stack is generally lighter than integer data stack. FP stack pointer is stored in R10, named rFSP in iMops.

### 2.1.3 Return Stack

The rest of the stack frame is for return stack. MacOS X allocates the main stack frame for an application, the size of which is about 8 MB. Data stack and FP stack use totally 80KB. So almost all of the stack frame is for the return stack. Return stack is used also for external function calls as C-function's stack frame. But as far as our word is running, it is used as a normal Forth type return stack. Current return stack pointer is stored in, naturally, the system stack pointer register RSP.

## 2.2 Register definition

### 2.2.1 Integer registers (GPRs)

X86_64 has 16 GPRs.

Some registers are reserved for special purposes. As described above, Data stack pointer, FP stack pointer and Return stack pointer are stored in GPR5(RBP), GPR10 and GPR4(RSP) respectively. Additionally, the beginning addresses of Data dictionary and Code dictionary are stored in GPR3(RBX) and GPR9 respectively.

GPR11 is reserved for the loop counter register and GPR12 for the loop limit register. Three registers GPR15, 14, 13 are for first three locals from left. GPR8 stores the current iMops object address in method executions. The rest, that is, GPR0(RAX), GPR1(RDX), GPR2(RCX), GPR6 (RSI) and GPR7 (RDI) are used to do essential calculations of data stack items.

In the first implementation, GPR7 is used as a temporary storage in shortage of registers for manipulating stack items. But now, the roll has been swapped with GPR0.(This explains strange names of some words desfined in ix-compileX') Ordinary stack item cache registers are now RDI, RSI, RDX and RCX. RAX is reserved for a temporary need of register. RAX is also used for the address reference in absolute address calls (that is, external calls).

### 2.2.2 GPR Usage

| GPR | volatility | System usage | in iMops |
|---|---|---|---|
| GPR0(RAX) | yes | the first return value on call | various |
| GPR1(RDX) | yes | the third input parameter on call | the third stack item cache register |
| GPR2(RCX) | yes | the fourth input/the second return on call | the fourth stack item cache |
| GPR3(RBX) | no | none | the base address of the data dictionary |
| GPR4(RSP) | no | the stack frame pointer | the return stack pointer |
| GPR5(RBP) | yes | the frame base pointer | the data stack pointer |
| GPR6(RSI) | yes | the second input on call | the second stack item cache |
| GPR7(RDI) | yes | the first input on call | the first stack item cache |
| GPR8 | yes | the fifth input on call | the current object base address |
| GPR9 | yes | the sixth input on call | the base address of the code dictionary |
| GPR10 | yes | none | the FP stack pointer |
| GPR11 | yes | none | the loop counter |
| GPR12 | no | | the loop limit |
| GPR13 | no | | the third locals |
| GPR14 | no | | the second locals |
| GPR15 | no | | the first locals |

### 2.2.3 Floating Point Number registers

For FP number calculations, lower halves of 128bit XMM vector operation registers are used. X87 FPU registers are left unused. This is because MacOS X's function call convention requires XMMs as FP number parameter passing registers. There are 16 XMM registers. XMM0-5 are used for ordinary data operations, XMM6-7 are for temporary use. XMM8-15 are for FP locals. But input/output parameter cache is restricted within 4 registers.

## 2.3 iMops Dictionary

### 2.3.1 Two parts dictionary

iMops dictionary is separated into two parts, Code dictionary and Data dictionary. Both are allocated in heap memory through a system function `posix_memalign()` at start up time. Normally code dictionary is allocated in lower memory.

The beginning address of code dictionary is stored in a `ZVALUE`, `CODE_START`. The dictionary space at start up is iMops kernel code size + 5 MB. The beginning address of the current code dictionary space is represented by a `ZVALUE`, `CDP`.

The address of the start of data dictionary is stored in a `ZVALUE`. `DATA_START`. The dictionary space at start up is iMops kernel data size + 1 MB. The beginning address of the current code dictionary space is represented by a `ZVALUE`, `DP`.

Code/Data space at start up time will be always 5/1 MB, respectively. So, if you need more dictionary space (really?), then load a part of the code on iMops and do `SAVE-DIC`, and replace the executable file in iMops.app package with newly saved dictionary. Code/Data space will be newly set to be 5/1 MB. Then load the rest part of you code.

Word `HERE` pushes the current beginning address of **Data Dictionary** (DP) on to the stack. So, in contrast to some classical forth, you shouldn't Compile-Execute code at `HERE`.

### 2.3.2 Word List

Publicly defined words are mutually linked in 8 way threads. Header data are stored in code dictionary area.

Methods and Instance variables are linked within each class. The lists are independent from public word list.

Mops have not had a custom wordlist feature for a long time. But that seems practically unnecessary for ordinary Mops programming, probably thanks to OOP feature.

## 2.4 XT and Relocatable address

In iMops, `xt` of a word is implemented by using the offset from the "origin" of the code dictionary to the address of the end of the handler code field of the word. This offset will never be changed once compilation is done. As the result, `xt` is a relocatable address. So you can store `xt` as if it is a plain value.

`XT` itself is not pointing a memory address. To recover the address, you can do

```
( xt ) ABS-CODE> \ in compile mode
or
( xt ) CODE_ORIGIN + \ in execution mode
```

This word `ABS-CODE>` simply adds the address value of the start of the code dictionaly (it is one register addition, so very quick).

**note**
But note that absolute xt address is not the function pointer. That points to the header area of the word, not to the content code itself. If you need to get the beginning address of the real machine instruction, few calculations are necessary. That is,

```
( xt ) ABS-CODE> 4+ @X ABS-CODE>
```

Then you will get the code pointer of the word represented by the input xt. But there should be no need to do such a thing in ordinary Mops programming, of course.

The reverse transformation is

```
( code-dic-addr ) REL-CODE> \ in compile mode
or
( code-dic-addr ) CODE_ORIGIN - \ in execution mode
```

By this, you get a relocatable address in code dictionary. ("REL" stands for Relative or Relocatable.)

The data dictionary is allocated different place from the code dictionary. When the address is in the data dictionary, you should use instead

```
( data-dic-addr ) REL-DATA> \ in compile
( data-dic-addr ) DATA_START - \ in execute
( relocatable-data-addr ) ABS-DATA> \ in compile
( relocatable-data-addr ) DATA_START + \ in execute
```

Note that these REL/ABS words never check if the value is valid rel/abs data/code address.

However, if you keep `xt` only for `EXECUTE` or `COMPILE,`, which should be normal, then you can use an `X-Addr` class instance, which will free you from considering internal details of Mops.

# 3 iMops and Object Orientation

## 3.1 NEON model

iMops' OO features are simulating PowerMops, which is said to be based on "NEON Model". Standard syntax for message sending is

```
parameters ... SELECTOR: Object
```

Additionally,

```
parameters ... Object SELECTOR: class_as> theClass
```

should work, too. [1]

The latter type of phrase can be used for forcing the class of the method. Even when the `Object` is of a subclass of `theClass`, the method of `theClass` will be executed there. The class of "`Object`" will usually be "`theClass`" or its subclass. But "`Object`" in this context can be of quite different class from `theClass` or even not an object. Even then the "`SELECTOR:`" will be bound to the method of "`theClass`" without any compiling error message. (So you should be careful, of course.)

These are ways for early (static) binding.

[1] However, "parameters ... Object SELECTOR: theClass" syntax support is dropped.

**note**
Note that `class_as>` is designed only for early binding. It can't be used for selecting a superclass to be bound in a multiple inheritance context. For the superclass selection in a multiple inheritance, you should define an appropriate method in the class using `super>`.

Late binding is also supported, of course. It will be useful when
1) the class of the object that gets message is still indefinite or dynamically changed at runtime, and/or
2) the class or method to be needed there is not defined yet.

iMops equips a method cache mechanism similar to that of PowerMops for the standard late binding. The cache mechanism makes execution time roughly half (supposing that the method check fails 10 times in average until the true method is found. The longer the method list becomes, the more the cache mechanism becomes effective.). So, the execution of the late binding method in iMops is quick enough like in PowerMops. Even the slowest late bind is fast and light enough for GUI classes.

In method definition, `SELF` can be a receiver of messages. This invokes corresponding method already defined in the class on the current object. Note that this way of message sending causes early-binding, not late-binding. If you need late-binding to `SELF` (for example, the real method definition should be postponed to the subclass), use `[SELF]` (or `[ SELF ]`) as the receiver of the message.

## 3.2 Reference and vTable Binding

Object reference is an extension of the message sending system in PowerMops. iMops also has it. A reference is a kind of `VALUE` in that it gets "`->`" or "`to`". But it is supposed to store the address of an Instance, and has its own header data that contains class information.

Message to a reference will be dynamically bound to the corresponding method, however, not through method search but through offset calculations and fetch from vtable. This way of binding is efficient in that the number of methods in the class is irrelevant to the execution speed. The execution speed is roughly twice compared with standard late binding with cache. Compared with early binding, vtable late binding adds one immediate value addition and 2 memory fetches and loses the stack item cache mechanism usable for normal calls.

By pushing an object address to a reference at runtime, your word can dynamically change objects to which messages are sent, with small overhead.

For vTable binding, a reference must be declared with its class. And the class must have the method corresponding to the message. Message-sending to a reference is complied using the reference's class data. However, a reference can store an instance of the subclass of the reference's class. When an object put in a reference at runtime belongs to a subclass of the class of the reference, corresponding method of the subclass will be called.

For creating reference, add "`Ref`" at the head of the declaration. The syntax of reference declaration is same as that of object declaration except for the "`Ref`".

```
\ String Object reference. A subclass instance of STRING is ok.
Ref String myStringRef

\  String Object reference -- need to be strictly of STRING class.
Ref String StringRef2 no_subclasses

\ Following can store an object of any class.
Ref Any AnyObjectRef
```

References can be ivars or temporary objects. The declaration syntax is same as that of public reference — add `ref` prefix to the declaration. For a bit more descriptions on the reference system, see

below (3.7)

## 3.3 Inheritance and Instance variables

(This section is mostly n.s's personal opinion.)

iMops supports Multiple Inheritance. That will become a powerful tool when used appropriately. Mops' multiple inheritance was introduced as a complement of original NEON's single inheritance. It takes the first superclass as, so to speak, the default superclass. The second and after superclasses are supplements for the first superclass. Inheritance is somewhat static and hierarchical system. It takes Class as a data abstraction, and extends or modifies it through subclassing.

On the other hand, compositing various objects into one as instance variables in a class is another way to create an object with new features. Compositing ivars is somewhat synergetic and combinatoric system. The effect is case-by-case. The ivars that are mutually independent objects are internally related with each other through message sendings, which build a class that has quite high level of functionalities. It resembles to the phenomenon that plural atoms (ivars) are composited to be a new molecule which has some unique character.

From my own experience, application programming in PowerMops tends to be the latter atoms-to-molecule approach. I guess programming in iMops will have similar tendency. It is because an ivar in Mops is not a mere slot or data field to store numeric attributions of the instance. Setting an ivar in a class means, in Mops, to embed the functionalities of the ivar's class in the class. That is, an ivar in Mops is "a class embedded in the class" or "a class instance embedded in another class instance". This character has been turned out to make ivars very powerful programming utilities. By the nature of the ivars described above, the feature to build a new class can be seen as a MetaClass system. For binding various objects as ivars into one class means, in Mops, binding all method of those ivars' classes with the class. Defining an ivar of an object in Mops means not merely to modify the data structure but also to bundling methods of the ivar's class not necessarily belonging somewhere in the inheritance tree of the class that has the ivar. Moreover, by using reference ivars, the combination can be changed dynamically. In this respect, Mops class is also MetaClass. I think, Mops programmers have done a kind of Aspect Oriented Programming for 20 years, though without the concept.

There are at least two points of view on the combination of objects. One is based on the structure, another is based on, so to speak, various "features". Standard arguments on OO have moved from the structure considerations to the features side. However inheritance is planed principally based on the structure, at least in the origin. In spite of that, many OOP continued to use inheritance to bind various features. So, on the other hand, it has been realized that there is some impedance mismatch between modern OOP arguments and inheritance, and some important new features have been invented to "avoid" inheritance. It may be good and fun to think about abstract features combination. However, for robust and exact execution, structural thinking is still necessary even in modern OOP, I think.

Anyway, Mops supports both approaches and so does iMops, of course, too. And non-Object ivar "`BYTES`" is also implemented. `BYTES` is similar to Forth array. However `BYTES` can live only in the private data of an object. `BYTES` is not a true object but still has its class, "`OBJECT`". For example,

```
:CLASS aClass super{ Object }
16 BYTES NON-OBJ-IV
...
```

Then, an instance of `aClass` will have an ivar having 16byte length named "`NON-OBJ-IV`". If you write `NON-OBJ-IV` in a method definition, it will put the address of the ivar in the object on the stack.

## 3.4 Public Instance Variable and Static Instance Variable

### 3.4.1 Public Instance Variable

Instance variables in Mops object are private by default. "Private" here means those are not accessible from the outside of the class definition or the subclass definition. Generally, such ivars seem to be called "Protected". Since all ivars are unconditionally inherited to subclasses in Mops, there is no way to set more severely restricted privatization of ivars. But on the other hand, any general way to access to the value in ivars from outside like dot syntax, is not supported in Mops.

In order to make instance variables accessible from outside, you should use Public Instance Variables. How to make public instance variables is simply to declare `PUBLIC` in ivar definition. For example,

```
:CLASS ACLASS super{ OBJECT }
   var v1
   PUBLIC
    var v2
    string string1
   END_PUBLIC
   var v3
:m ...
....
ACLASS O1
```

Then, the object `O1` will have two public ivars, `v2` and `string1`. `v1` and `v3` will become private ivars of `O1`. As an alternative syntax, `PUBLIC{ ... }` is also supported in iMops. That is,

```
:CLASS ACLASS super{ OBJECT }
   var v1
   public{
    var v2
    string string1
   }
   var v3
:m ...
....
ACLASS O1
```

To access to a public ivar, a special syntax is used: that is, for example

```
get: IVAR> v2 in O1
```

Then `get:` message are sent to a public ivar `v2` of `O1`.

### 3.4.2 Static Instance Variable

Static instance variable is instance variable of a class. While ordinary instance variables are owned by a object, static instance variables are shared by objects that belong to the class or its subclasses. So static ivar may be called "Shared Instance Variable".

```
:CLASS ACLASS super{ OBJECT }
   var v1
   STATIC
   {
    var V2
    string String1
   }
   var v3
:m ...
....
ACLASS O1
ACLASS O2
```

Then, `O1` and `O2` will share two objects, `V2` and `String1` . Static ivar can be either public or private. When it is private, it is accessible only through method in the class. When it is public, the syntax of the access uses "`IVAR>`" and "`IN_CLASS`"

```
:CLASS aClass super{ OBJECT }
   var v1
    STATIC
    {
 PUBLIC
      var v2
 END_PUBLIC
      string string1
    }
   var v3
:m ...
;CLASS

GET: IVAR> v2 IN_CLASS aClass
```

### 3.4.3 Curly braces syntax on Ivar

`RECORD{`, `PUBLIC{` and `STATIC{` declarations on Ivars using curly braces can be nested. But they can not be crossed. For example,

```
RECORD
{
 Var v1
   PUBLIC{
    Var v2
}
    String str
    String str2
    }
```

Then, only `v2` will become a public ivar, and `str` and `str2` will be included in `RECORD` structure[2]. On the other hand, `PUBLIC` – `END_PUBLIC` declarations can be crossed with curly bracket declarations.

```
RECORD
{
 Var v1
  PUBLIC
    Var v2
}
    String str
    String str2
  END_PUBLIC
```

Then, `RECORD` structure will include just `v1` and `v2`, and `v2`, `str` and `str2` will be public ivars.

## 3.5 Public Object as an ivar of another Object

You may want in some cases to make a public object (instance) an instance variable of another public object. In such cases, you can use reference ivar. Reference can be declared as an ivar in a class definition according to quite same syntax as reference declaration. A reference ivar doesn't have its content object at creation time of the instance. You can put the content object via method.

## 3.6 Temporary(Local) Object

iMops has the temporary object functionality like PowerMops. Temporary objects mean iMops class instances living only within one word definition (local object in the ordinary pragmatics). Such objects must be declared at the beginning of the word definition that uses the temprary objects. How to use them is same as in PowerMops.

[2]Crossing of `RECORD` with `STATIC` doesn't have practical meaning very much. Because spaces for static and non-static ivars will be allocated in mutually quite different places.

```
: aWord
TEMP
 {
    VAR TempVar
    STRING TempStr
 }
  ....
;
```

`TEMP{ .... }` syntax is also supported.

When your word has named parameters and/or locals, too, temporary object declaration must be **after** the locals declaration (but before the beginning the content definition, of course). Temporary objects are allocated in the return stack frame and just under (higher in memory, though) the locals frame. Therefore, the compiler need to know the size of local frame offset before calculating the addresses of temporary objects.

`CLASSINIT:` method of each class will be called on every temporary object orderly before the execution of the body of the word, then, after the execution of the word body, `RELEASE:` method will be called, and finally stack frame will be dumped. When your temporary object will allocate something in heap memory area and keep the pointer in its data area, `RELEASE:` method should Free the heap area, or memory leak will happen.

Different from PowerMops, Register Object feature is not supported in iMops. Additionally, message sending to locals is not supported (use normal explicit late binding synax).

In the implementation of PowerMops, temporary objects are instance variables of a generic global class. In iMops, they are ivars of the word having those temporary objects.

### 3.6.1 A bit more about CLASSINIT:

A method named "`CLASSINIT:`" has a special meaning like in PowerMops. The message is to every object recursively when it is created. `CLASSINIT:` message is sent to all the ivars (including ivar's ivar...) and super classes. So "`CLASSINIT: SUPER`" is not necessary in the definition of the subclass' `CLASSINIT:`. However, the message is NOT sent to ivar's superclasses.

`CLASSINIT:` method cannot take the parameter from the stack.

## 3.7 Reference, Heap Object and Garbage Collector

### 3.7.1 Heap object reference

Object reference system supports heap objects and garbage collection. The garbage collector mechanism of iMops is simple and primitive. It could be seen as something like delayed release of heap objects. But it surely does some useful works.

By

```
ref String string1

... new> string1
```

you get a string object wholly in heap. After execution of the `new>`, you can send any message to `string1` for calling method of the class. You can treat "`string1`" as if it is a normal object in the message sending respect. In order to be able to get "`new>`", the reference shouldn't be declared with "`Any`" class, naturally.

Then, if you do

```
ref string string2

... string1 -> string2 ... \ increment the reference count
```

the heap object stored in `string1` is now referenced from `string2`, too, so that it gets referenced from two references. `string2`'s class can be "`Any`" (pseudo class for reference creation that can contain an object of any class.). As long as "`new>`"ed object is passed between references, the reference count is maintained by Mops system. By doing `new>` to `string1` again, another heap object is created in `string1`, the reference relation between the previously-created heap object and `string1` is cut off.

```
 ... new> string1
\ a new object is created and stored
\ and the ref count of the previously stored object decremented.
```

Now you have two heap objects of string class, each of which is referenced by just one reference.

Reference can, of course, store normal object.

```
String string3

 ... string3 -> string1 \ decrements the ref count.
```

Now, `string1` stores the base address of the `string3`, and previously stored heap object loses one reference. A heap object that is not referenced anywhere is collected by the garbage-collector. The object will, in the end, get "`release:`" message and its entire object data area will be freed at garbage-collecting time. Garbage collection will happen in idle time.

As for an indexed object (Array, collection classes), the limit number is not necessary to declare the reference.

```
ref array ArrayRef
```

The `ArrayRef` can store an array with any limit.

But if you do `new>` to the reference, then the array size is necessary, of course, at `new>` time.

```
  ... 10 new> ArrayRef ...
```

`release>` is a special prefix or message to detach a reference from the stored object and put `nil` to the reference.

```
 ... release> string2 \ decrements the ref count.
```

Then, the contents of `string2` is cleared and the heap object previously stored in `string2` loses one reference, so that it may be sent to the garbage pool .

But as for an ordinary object, "`release> string1`" will never release an object `string3` nor send it to the garbage pool, of course.

### 3.7.2 Reference Ivars and Temporary reference

One or more ivars of an object can be references. To put a public object to a reference ivar, you have to define an accessor method for that. Although a reference ivar can be a public ivar, you cannot do `->` by message sending syntax for a public ivar since `->` is not a method selector.

By defining some special initialization method to do `new>` to a reference ivar, your object can have a heap object ivar. Even the object that has heap object ivars can be a heap object created through reference. But note that `release:` method of the class that defines heap object reference ivars should do `release>` to all such references. Otherwise memory leaks may happen.

As for temporary object reference, iMops 1.0 extends the feature a little. That is, a "`new>`"ed temporary reference keeps the life up to just after return. For example, a word like following will work.

```
: createString ( -- ^string )
 temp{ ref string tempString }
  new> tempString
  [ put: and manipulate the contents of tempString ]
  tempString \ return
;

: aCaller .. createString -> retString .. \ 'retString' is a reference.
```

The content of the returned string created in `createString` can be passed as a string object through the stack. If `retString` is a publicly defined reference, the returned string will get a long life. If `retString` is a temporary reference of `aCaller`, the string will keep the life until the return of `aCaller`.

Since the garbage collector will not interrupt nor intercept the currently running process, the string object can be passed on the data stack during one process (normally one event handler).

By the way,

```
: anotherCaller ... createString drop ...
```

will be ok. The dropped string object will be garbage-collected.

**note**
However, a temporary **object can't** be returned. The data areas of all temporary objects are released on return. Returning an object described above works only for `new>`ed temporary **references**.

## 3.8 RECURSE for Method

The word "`RECURSE`" won't work in a method definition. So when `RECURSE` is necessary in method, call the method itself by sending message to `SELF`. For example,

```
...
:m fib:  ( n -- n' )
    DUP 1 > IF DUP 1- fib: self SWAP 2- fib: self + THEN ;m
...
```

**note**
The method above calculates Fibonacci sequence up to the $n$th term. Although Fibonacci sequence is often picked up as a simple example of recursive function, implementing it as a recursive function like above is not practical. Hint: transformation $f_{n-2}, f_{n-1} \Rightarrow f_{n-1}, f_n$ can be implemented in forth simply as "`TUCK +`".

# 4 Using and Building Dynamic Library

## 4.1 Public Library

### 4.1.1 introduction

Public library here means code and data resources which are usable from any application running on Mac OS X. That includes dynamic libraries (dylibs) and frameworks. It can be divided into 2 categories, system services which are provided by Mac OS X and the others. MacOS libraries are located in the directories /System/Libraries/Frameworks/ and /usr/lib/ (invisible). The other libraries may be in /Libraries/Frameworks/ or $home/Libraries/Frameworks/ directory. (Libraries in the lastly named folder are usable only from your account.)

As for OS services, most of the main frameworks[3] are already loaded into private memory of iMops. So you usually don't need to load any other framework.

However, you may get "symbol not found!" error message on calling some function in some cases. First, check the spelling of the function name (Case Sensitive!). If it cannot stop the error message, then you should load an optional source code "libCalls" (in a folder /source/iMops_ext/) on iMops by executing

```
need libCalls
```

Libcalls provides 4 main words, `ObjC-Framework`, `Framework`, `Library` and `FrwkCall`.

### 4.1.2 ObjC-Framework

`ObjC-Framework` is for Objective-C classes and methods. When Objective C Class or Method corresponding to a word declared with `ObjC_Class` or with `ObjC_Selector` is not included in already loaded frameworks, the word returns 0, so that your code fails to work. In such a case, you can declare with this word the framework that contains the Objective C classes or methods you need. `ObjC-Framework` defines a word whose execution loads the framework identified by the file path. For example,

```
ObjC-Framework WebKit "WebKit.framework/WebKit"
WebKit \ execute
```

That is, the syntax is:

```
ObjC-Framework word-name "file path to the framework"
```

At the beginning and ending of the file path, double quotation marks are required. By execution of the word name, the framework (WebKit.framework in this case) will be searched in public frameworks folders and, if found, loaded into the private memory of iMops. After that, the framework becomes linkable, that is, Objective C Classes or Methods defined in this framework become accessible through words defined with `ObjC_Class` or `ObjC_Selector` in iMops. If you need the framework in your application to be installed, execute it at the starting up word.

Loading a framework increments the reference count of the framework, so take care your application doesn't load an already loaded framework. `ObjC-Framework` also sets the "current framework" referred

[3] At present, the frameworks or dylibs automatically loaded on launch of iMops are:
System.framework, CoreServices.framework, CoreFoudation.framework, Cocoa.framework, ExceptionHandling.framework, Quartz.framework and libobjc.dylib.
Some of them are Umbrella frameworks.

to in the next section(4.1.3). There need some adjustments among framework declaration words to avoid duplicated loads (see section 4.1.4)

### 4.1.3 Framework and FrwkCall

`Framework` declaration defines a named framework for function calls thereafter. The declaration sets the framework as the current framework. External functions declared with `FrWkCall` are supposed to belong to the framework. For example,

```
Framework glut.framework
FrwkCall glutTeapot { %magn -- }
```

Then, `glut.framework` will be set as the current framework and an external function `glutTeapot` will be supposed to belong to the `glut.framework`. `Framework` declaration requires a framework name, as a rule, with .framework extension.

`FrwkCall` declaration is quite similar to `Syscall`. The only difference between them is that `FrwkCall` needs the current framework to be set before.

### 4.1.4 Framework and ObjC-Framework

Some Objective C frameworks contain not only Objective C Classes and Methods but also functions or procedures. Even when your code already declared an Objective C framework with `ObjC-Framework`, if the framework contains a function you want to call, you may need to declare the same framework with `Framework` again to set it to be the current framework for `FrwkCall`s. In such a case, you should use the framework name declared with `ObjC-Framework` before, also in the later `Framework` declaration. For example (although not good example since Foundation.framework is always loaded at launch),

```
 ObjC-Framework Foundation "Foundation.framework/Foundation"
.....
Framework Foundation \ Shouldn't add .framework extension to the name.
Frwkcall NSDecimalRound { ^res ^num scale mode -- }
```

Then, duplicated loads of the framework will be avoided.

## 4.2 Private Library

### 4.2.1 introduction

Private Library is supposed to be put in an application bundle package. Concretely, it should be in

`applicationName.app/Contents/Frameworks/`

folder. That is, Private Library is owned by one application package. It is convenient in that the version management is generally unnecessary.

Private Library includes Private Framework that may contain its own data resources and Private Dynamic Library that usually consists only of the executable file.

To load a private library, a word `Library` is provided.

### 4.2.2 Library

`Library` is almost same as `Framework`. But `Library` supposes that the declared framework is a private framework. The syntax for `Library` is same as that for `Framework`. `Library` sets the framework declared

with it as the current library. Use `FrwkCall` to define an external call function belonging to the private framework.

```
Library mydylib.framework
frwkcall func1-in-lib { in -- out }
```

`mydylib.framework/mydylib` will be searched in `/Frameworks/` folder in the application package.

## 4.3 Building Framework

### 4.3.1 Export a Word

Since version 1.2, iMops can build framework or dynamic library. The syntax to define a word to be exported is similar to that in PowerMops. That is, the definition must begin with `:ENTRY` and finish with `;ENTRY`. A word defined in such a way will be entered into the external symbol table of the dynamic library to be built.

However, the way of input/output notation of `:ENTRY` word of iMops differs from that of PowerMops. In iMops, the input/output notation uses (( and )) pair, not curly bracket, and inputs or outputs are passed through the Data (and/or FP) stack.

```
:entry  squareSum  (( x y -- z ))  dup * swap dup * + ;entry

:entry fsquareSum  (( %x %y -- %z )) fdup f* fswap fdup f* f+ ;entry
```

A parameter name beginning with % mark indicates that it is a floating point number parameter and comes through FP stack. Other names represent integer parameters. Although both Integer/FP parameters came in through stacks, it is still possible to define locals or named parameters also in `:entry`-defined word.

```
:entry  squareSum2  (( x y -- z )) \ equivalent to squareSum above
 { x y -- z }
  x x * y y * +
;entry
```

Note that the parameter notation with a double parentheses pair is still required in this case.

This parameter notation system is same as that of callback in iMops. In fact, the content of `:ENTRY` is essentially same as that of `:MT-CALLBACK`.

At present, there is a restriction concerning the maximum number of input/output parameters, described in following Table 1.

Table 1: Restrictions on the number of parameters

| Parameters | Max. |
|---|---|
| Integer inputs | 6 |
| FP inputs | 8 |
| Integer output | 2 |
| FP outputs | 2 |

Additionally, if you want to make your dynamic library generally linkable, the word should be conformed to general requirements for a library function (usually, C function specs.).

The name of a word to be exported is case-sensitive, in contrast to ordinary Mops words.

### 4.3.2 Initializing Dynamic Library

A dynamic library built with iMops has its init routine. When you define a word "`FRWK-INIT`" with `:f - ;f` pair in your dynamic library, it will be called on load of the library.

```
:f FRWK-INIT  [your initializing routine]  ;f
```

You can do anything you think necessary for initializing your library, for example, to load nib resource for your library into memory. But you shouldn't, naturally, try to process too heavy task on load of your library.

When your dynamic library uses Objective C classes or methods in any internal words, call a word `Prepare_ObjC` in the definition of `FRWK-INIT`. Then, all Objective C classes and selectors that are preloaded in iMops environment will be set at the time of the initialization.

### 4.3.3 GUI Elements in a Dynamic Library

You can define and send `new:` message to an instance of an iMops GUI element class in your dynamic library. But it is not recommended. Especially, Button or other control objects will not work as you may expect. This is mainly because Control classes use `xt EXECUTE` for the control action — the action xt is supposed to be in the main application area, not in a dynamic library. So the action xt of a control object must be one that represents a word stored in the main application dictionary. (An internal use of `xt EXECUTE` in a dynamic library shouldn't cause any trouble. The trouble described above comes from the fact that an event handling callback is executed in the main application even when the event happens on an object defined in a dynamic library.)

### 4.3.4 Garbage Collection in a Dynamic Library

A dynamic library has its own garbage pool but, naturally, doesn't call an event loop. So, a dynamic library doesn't have the chance of its own garbage collection. If you use `Object_List` or `new>` of a reference in your dynamic library, export a word that calls `?COLLECT_GARBAGE` and repeatedly call the word at some appropriate timing from your main application.

# 5 Installing a Stand Alone Application

iMops can build a stand alone application, a.k.a. a turnkey. The process has been called "Install" in Mops. " Install..." item of Utilities menu is for that. By adding .app extension to the folder name so to create the package, you get your application which can be launched by double clicking on the icon.

## 5.1 Differences of Installed Application from iMops Environment.

### 5.1.1 No Code Generator by Default

An installed application doesn't have the extensible dictionary. In addition, from version 1.2b on, code generator part[4] (about 210kb) will be chopped off in installing process. The code generator part includes

[4] "Code Generator Part" above means, in source code files, `ixCGX, ix-nodesX, ix-wordheaderX, ix-allocatorX, ix-CompileX, ixcg2X, ix-ASMX, ix-FPCompileX, ixcg3X, ix-dictionaryX, ixcg4X, ix-condX,` and `ixcg5X`. All files are in /source/new_intelport/ folder.

the compiler and interpreter, so an installed application shouldn't compile or interpret (`EVALUATE`) any source code text in running. When your installed application tries to execute "`EVALUATE`" in some word, it will crash immediately. (Even before version 1.2b, it is true because an installed application doesn't call the initialization word of the code generator.)

However, when your application needs interpreter or even compiler, check "With Code Generator?" checkbox at the left-bottom part on the Install dialog window. When you installed your application with the checkbox on, your application includes the code generator part and initialize it on launch. An installed application with code generator is essentially an iMops developing environment although without iMops menus and the console window.

### 5.1.2 Don't Save Absolute Memory Address as a Value

The virtual memory address where code or data dictionary is located will be quite different in an installed application from in the developing environment. So you shouldn't store a pointer in dictionary in a variable and use the value after installation. When your application to be installed needs to store such a pointer, use `REL-DATA>` or `REL-CODE>` described in section 2.4. Or you can use a `DicAddr` class instance, by virtue of which you don't need to care whether the pointer belongs to code dictionary or data dictionary.

## 5.2 Miscellaneous Notes

### 5.2.1 Special Initialization

An installed application calls a word "`PRIVATE-INIT`" before starting up word. The content of the word is initially NULL. But `PRIVATE-INIT` is a `FORWARD` declared word, so you can define the content by `:f`-definition.

### 5.2.2 Event Loop

Your application to be installed doesn't need to call the main event loop from the start up word. After executing the start up word, the main event loop (an objective C method, `NSApp run`) will be automatically called. Your application can call your custom event loop from the start up word, if you need to do that. But then the start up word will not return, so that any event loop timer will not be automatically installed. By calling `Install_Timer` before entering into the event loop, the default timer for iMops Garbage Collection will be installed.

### 5.2.3 Timer

iMops creates and installs the default event loop timer on launch also in an installed application. For the timer is used for the garbage collection.

However, only one timer is allowed in one event loop and the default timer has very long interval (10 seconds). So, if your application needs more frequent timer calls, for example, to draw animated pictures in the main event loop, then you may want to create your timer and install it. For that, iMops provides a `zvalue __defaultTimer`.

If you need your customized event loop timer: In your `PRIVATE-INIT` or start up word, call `CFRelease` on the value in `__defaultTimer`, then create your CFRunLoopTimer instance and put the object's pointer into `__defaultTimer`. Then, your timer will be installed in the main event loop.

Note that, if your application uses an `Object_List` class instance or `new>` to a reference, don't forget repeatedly to call a word `?COLLECT_GARBAGE` in some interval (some seconds will be enough) from your timer process.

# 6 Features peculiar to iMops

## 6.1 Local Section

PowerMops has its Local Section feature. iMops' Local Section feature is similar to, but somewhat different from that. iMops' Local section is, say, a local section interpreted as a module.

Local Section can have named input parameters and local variables. But the upper limit of the total number of locals is only 3. If you declare more than 3 locals at the beginning of a local section, iMops will abort the compilation with an error message. **(The restriction on the number of locals of a section has been removed since ver. 1.22! The upper limit is now same as in a normal word (127 in theory).)** And internal words can't have its own locals, naturally. However, a local section can have its Internal Objects and Internal Values and Variables, instead.

Internal Objects works like temporary Objects of the local section. They will be initialized on entry to the section, and released on exit from the section. Internal Variables and Values will not be initialized on entry, so they will work like the state variables of the section. If you need to make the section a "function", you should manually initialize them in some words in the section.

Internal Objects, Values and Variables are really allocated in the dictionary like normal public ones. But they cannot be accessed from the outside of the section. By that, they will work like local variables/objects in a word.

A local section begins with

```
LocalSection theMainWord { in1 in2 \ loc1 -- }
```

"`theMainWord`" is the name of this section and the only exported word name of the section. True locals declaration of the section must be there.

This local section ends with the definition of "`theMainWord`" :

```
:LOC theMainWord \ { in1 in2 \ loc1 -- }
[do something calling inner words, objects, values or variables]
;LOC
```

Words, values, variables and objects defined or declared between them become inner or private ones of the section. They can be callable in the section according to the definition/declaration order, like normal words.

Inner words can use locals of the section. But they cannot have their own locals. If you need more locals in a section, you can define inner values and use them with runtime initialization.

## 6.2 Method Declaration

In iMops, you can postpone (NOT the word, `POSTPONE`) some of method definitions. In a class definition,

```
:CLASS aClass super{ ... }
...
:METHOD> doSomething1:
:METHOD> doSomething2:
```

will add methods `doSomething1:` and `doSomething2:` whose contents are `NULL`. These methods can be bound in compilation after the declaration. The methods are early-binding, not late-binding, so the runtime penalty will be negligible.

Before their executions, you should define the contents by:

```
:IMPLEMENT-METHODS-OF aClass
:IMP doSomething1:   .... doSomething2: SELF .... IMP;
:IMP doSomething2: { a b \ c -- }  .... IMP;
IMPLEMENTATION-END;
```

In the implementation environment, you can access ivars of `aClass` by their names, or send a message to `SELF` or `SUPER` like in the class definition of `aClass`.

## 6.3 Changes in Method Call Compilation

From version 1.1, the way to compile message sendings has been changed.

Classical way of early binding in Mops uses a parse (tick). A selector parses an object, extracts the class data of the object and binds the method in the class.

Contrary to that, iMops doesn't use a parse of an object name. Instead, iMops uses a selector ID stack. In compiling a selector, whose name ends with colon, the ID of the selector (8byte hashed value) is pushed onto the selector ID stack. But there the method binding doesn't happen yet. An object called after it triggers the method search and binding.

In calling an object:
1) if the selector ID stack is not empty, then the object pop the stack item and searches and binds the method corresponding to the selector in its own class;
2) if the selector ID is empty, then the base address of the object is pushed to the data stack.

"`IVAR>`" and "`CLASS_AS>`" still parse the next ivar name and class name, respectively. But they are somewhat exceptional cases in method bindings.

The maximun depth of the selector ID stack is 7. At the end of word definition, the ID stack will be cleared. When there are left any selector ID in the ID stack at the clearance time, a warning "`Selector Unbalanced`" will be displayed.

All of the standard Mops message sending syntaxes should work as before.

**Remark**
As the syntax is not changed, it can be said that the meaning of message sending code in Mops has never been changed since the beginning. But the implementation semantics is surely changed in iMops. In traditional NEON OO model, Selector is a real object in the real implementation. For it is Selector that inspects an object which receives the message, and that decides what to do there based on the result of the inspection. That is, Selector needs to know and knows the object, the class and the method to be called there. While, an Object (Instance) is a mere name in the usual message sending. This is quite different from the meaning of message sending code where an Instance is the receiver of the Selector and should know the method. My a bit quirk trial described above is to match the implementation meaning with the code face meaning. But as you know, the implementation meaning (semantics) is irrelevant for writing the code that has the clear and definite meaning. Why do we need to know anything about vtable or other concrete binding mechanism to program with OO? But knowing some details of the implementation may help some when we infer which part of the code a bug creeps in.

## 6.4 Getting Information

### 6.4.1 LOCATE and others

iMops distribution contains all source code files of the iMops development environment. By `LOCATE`, you can see the definition of a word in a source code file.

```
LOCATE ( "<spaces>ccc" -- )
```

LOCATE will open a special window in which the definition code will be displayed and pointed. The source code text in the window will not be editable. But the path to the file will be printed on the iMops console so that you can find the file to edit, if you need to do that.

However, concerning forward-defined word, LOCATE will point the FORWARD-declaration part of the word, which is usually useless to know the semantics of the word. So, another word DEFINITION> is defined. The syntax to use DEFINITION> is similar to LOCATE.

```
DEFINITION> ( "<spaces>ccc" -- )
```

DEFINITION> will point the current content (:f-definition) of a forward defined word. As for words other than forward-defined, DEFINITION> is same as LOCATE.

A method of a class is not a word in iMops, so that LOCATE or DEFINITION> will not work for a method. In order to see definitions of methods, you can use MLOCATE.

```
MLOCATE ( "<spaces>class-name" "<spaces>method-name" -- )
```

For example,

```
MLOCATE String get:<enter>
```

will point the definition of GET: method of STRING class on the special window for location. In addition, the class name to which the method really belongs and the method name will be printed on iMops console.

### 6.4.2 Words, Classes, Objects and WordsWith

An ANSI forth standard word WORDS is not defined by default. But by loading an optional source file "inspectors", four inspection words, WORDS, Classes, Objects and WordsWith become usable.

WORDS displays a list of all defined word names with type and xt info in a special window. Classes displays a list of all defined class names with the super class information. Objects displays a list of all defined instance names with the class information. Both class and object are formally categorized as WORD, so WORDS list includes Classes and Objects.

A Mops utility word by Douglas Hoffman, WordsWith can search a word from the partial name. For example,

```
WordsWith  fe<enter>
```

will display a list of words each of whose names contain the string fragment "FE" as their parts.

## 6.5 Special Prefixes for Values and Locals

In Mops Values and Locals have much in common in their syntax for the usages. To store a numerical value on the stach into a VALUE or LOCAL variable, a prefix -> or TO will be put before the variable name. To add or subtract a value to/from the stored value of a VALUE or LOCAL, prefix ++> or --> is available, respectively (Note: neither +TO nor -TO is defined in Mops.)

In addition to those standard prefixes, some other prefixes are also defined in Mops for utility (they have been defined in PowerMops since ago, but not clearly documented). Following presfixes should work for ZVALUE (64bit value) and Locals:

```
AND>  ( x -- ) \ Operates AND on the content value with a stack item
OR>  ( x -- )  \ Operates OR on the content value with a stack item
XOR>  ( x -- ) \Operates XOR on the content value with a stack item
NEG>  ( -- )  \ Changes the sign of the content value
NOT>  ( -- )   \ Operates bit-wise NOT on the content value
```

Note, though `NEG>` and `NOT>` work also for 32bit value, the other operations can work only with 64bit value.

# 7 Code Generation

## 7.1 Machine Code Generator

iMops' code generator produces optimized X86_64 machine code. Most of low level forth primitives are inlined. Optimization of iMops CG is neither complete nor perfect, but reasonably aggressive. Machine code part of every word definition is 16byte aligned, which fits to the code cache. Many (though not all) of Moves are deleted when the results are not affected by the deletion. DROP may delete preceding operations to produce the dropped value. Data stack items are, as described above, cached in registers if possible. And some more micro optimizations are also included. For example,

```
: lit-square-add  4 dup * 3 dup * +  ;
```

will be compiled as (GAS like syntax)

```
mov 0x19 %rdi
ret
```

That is, arithmetics on constants will be calculated at compile time.

Additionally, following forth code

```
: square-add  dup * swap dup * +  ;
```

will be compiled as

```
imul %rdi %rdi
imul %rsi %rsi
add %rsi %rdi
ret
```

In interpretation, the values of `RDI` and `RSI` are taken from the top and the second top items of the data stack respectively, and the result value of `RDI` is pushed to the stack. However,

```
: call-square-add  3 4 square-add ;
```

will be compiled as

```
mov 3 %rsi
mov 4 %rdi
call square-add
ret
```

The result value in `RDI` will be pushed into the stack in interpretation of `call-square-add`. Parameter passing of call of normal word is done through registers in many (most) cases.

Implicit inlining is not implemented. But by using explicit inline definition

```
: square-add'  inline{ dup * swap dup * + }  ;
: call-square-add'  3 4 square-add'  ;
```

word `call-square-add'` will be compiled as

```
mov 0x19 %rdi
ret
```

Generally, when many registers are usable, the key for good code generation is good register allocation algorithm. In theory, we can use all 16 GPRs freely in our code space. But iMops CG didn't take such a radical way. As a forth based environment, it kees stacks. So we reserved special registers for stack pointers. Moreover, registers for first three locals and object base address are also reserved. In this respect, iMops' register allocation is rather conservative.

In general theory of register allocation for compiler, randomly accessible variables are supposed as operands. However, in forth-like environment we can suppose stack data structure on operands data. iMops' register allocation algorithm in fact makes use of stack data structure to decide whether to allcate register or memory cell to a node.

iMops CG maintains a data list for the stack node graph. (see file nodeGraph.pdf) Compilation is at first done on the list, which does not generate real machine code yet. Normally, real code generation begins only when the real code address comes to be necessary, especially for jump or call. Call, conditional or loop triggers real machine code compilation and finishes one basic block. Deletions, contract or coalescing of nodes and register allocation on effective nodes are done within one basic block.

After coalescing nodes, registers are allocated to nodes from Top-Last. If two different nodes don't interfere with each other, those node can be allocated with same register. iMops CG has 4 registers to be freely allocatable. Generally speaking, when there are more than 4 stack items living, some nodes must be pushed on to memory cells.

iMops CG uses only the lower level simple machine operations. The code to be generated is RISC-like except for two operands restriction and memory operands.

## 7.2 Inline Definition and Execute

iMops' inline definition stores xts of words of which the definition consists. So all words in an inline definition must be already defined and registered in the dictionary by the inline definition time. Therefore, inline defined word cannot have locals or temporary object, for local somethings are not registered in the main dictionary. Additionally, inline definition can't include literals. If you need a literal number in your inline definition, define it as a `CONSTANT` in advance. If you need a string literal in you inline definition, define it as a `SCON` in advance.

A merit of this way of inline definition compared with classical string evaluation one is that the execution semantics is fixed at the definition time, not postponed to the call compiling time. On the other hand, string Evaluate should still work. For "late bind" of words, it could be used.

Inline defined word is, in reality, a list of xts which will be executed when the word is called. However, inline defined word itself can be `EXECUTE`d through its xt. All words except for :f-defined ones have glue code for xt execution. The word body of inline defined word is compiled in the glue code area. So xt execute is ok also for inline defined word whose real content is a mere list of xts. As its corollary, a word defined with `:f` can not be inline defined, for it has not glue code field.

## 7.3 Header Data

Each word has in dictionary the header data field that is relatively long. The field contains informations for compiling the call of the word.

The dictionary entry of a word begins the link field. The word name string follows it. After 4byte aligned, there is handler code(2byte) and flags(2byte) fields, which are totally 4 bytes long. The rest of header data is depend on the type of the word.

### 7.3.1 Word of Data type

Data type words include `VARIABLE`, `VALUE`, `CONSTANT` and other `CREATE`'ed words. In addition, Classes, Objects(Instances) and Object References are also categorized as ones of data type. The handler code is `$ ACxx`. xx part depends on the species(See Table 2).

Except for `Class`, there is a 4 byte field after 4byte alignment. In a `CONSTANT`, the constant value itself is stored in the field, which will be compiled as a literal value into machine code. In the other data types, the field contains the relocatable address of the data field in data dictionary where the word's own data will be stored.

`Class` is not a Data Type in proper sense, although it can have its own data field (static ivars). Just only for the purpose of categorization, OO constructs are put in the Data Type family.

Table 2: Handler Code for Data Type Words

| Species | handler code (hex) |
|---|---|
| CREATE | AC04 |
| VARIABLE | AC04 |
| ZVARIABLE | AC04 |
| FVARIABLE | AC04 |
| UVALUE | AC01 |
| VALUE | AC03 |
| ZVALUE | AC44 |
| FVALUE | AC48 |
| CONSTANT | AC02 |
| ZCONSTANT | AC42 |
| FCONSTANT | AC46 |
| SCON | AC05 |
| CSTRING | AC45 |
| Class | AC1D |
| Object | AC0B |
| Reference | AC0A |

### 7.3.2 Executable

Executable words include Colon definitions, `:CODE`-definitions, `:NONAME`-definitions, Methods, words declared with `FORWARD`, `DEFER` or `CREATE-DOES>` and external calls (`TOC-CALL`, `SYSCALL`, `FrWkCall`, `ObjC_CLASS` or `ObjC_SELECTOR`). The handler code is `$ AExx`. xx part is like following:

Table 3: Handler code for Executables

| Species | handler code (hex) |
|---|---|
| : | AE00 |
| :CODE | AE00 |
| :NONAME | AE01 |
| CREATE DOES> | AE04 |
| FORWARD | AE10 |
| Method | AE40 |
| ObjC_CLASS | AE44 |
| ObjC_SELECTOR | AE44 |
| SYSCALL | AE80 |
| FrWkCall | AE80 |
| TOC Call | AE81 |
| ObjC Method Binder | AE82 |
| Callback | AE83 |

Framework declared with `Framework` or `library` has the pseudo-executable handler code `AE85`, which is not executable. (By the way, `ObjC-Framework` is a `CREATE-DOES>` word.)

These handler code data are used to identify the word type on calling or initialization.

2byte fileld after handler code is Flags area. The address at the end of the flags area of a word data is the absolute xt of the word. Offset from `CODE_ORIGIN` to there is its true xt.

4byte fileld at the absolute xt stores the displacement to the glue code. This displacement will be used by `EXECUTE`. An external call word cannot be Tick-`EXECUTE`d since it doesn't have glue code while it may use registers to pass the parameters. `FORWARD` declared word doesn't have its glue code, too, but the displacement points the code body and we don't need to care for the parameter cache for `FORWARD`-declared word, so that it can be Tick-`EXECUTE`d.

After the 4byte displacement field, 4byte code offset data is stored. The data is the offset from `CODE_ORIGIN` to the beginning of the executable's machine code body.

After the code offset data field, there is a 20 bytes long word data area where the information of the stack item cache registers (input/output) and of locals is stored.

The machine code begins after that with 16byte alignment.

# 8 Mach-O Executable File Format and Virtual Memory Mapping

## 8.1 Executable File Format

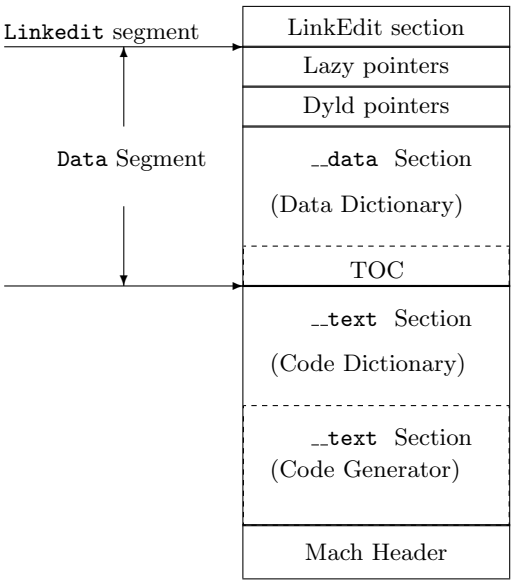
Mach-O executable file consists of three regions: 1. Header, 2. Load commands, 3. Data; Header and Load commands parts are needed for Mach-O file format. Those data are in the first page (the size

of one page is 4kb) in a executable file. As for the details of those data, see OS X ABI Mach-O File Format Reference on Apple's developer site.

The Data region of Mach-O file includes the text segment, data segment and linkedit segment. TEXT segment contains usual machine instructions and some data. The segment will be mapped into an executable but not writable memory area. DATA segment is for normal data, for example, variables, values, string data which are rewritable. The segment will be mapped into an writable but not executable area. LinkEdit segment contains external symbol tables.

iMops' dictionary is separated into two parts, the code and the data. So it is natural that the code dictionary is stored in the TEXT segment and the data dictionary in the DATA segment of the Mach-O executable file. The TEXT segment of iMops executable consists of only one section, that is, the text section whose content is the code dictionary.

The DATA segment consists of three sections, the data section, dyld section and lazy symbol pointers section. The content of the data section is the data dictionary of iMops. The dyld (dynamic link editor) section contains 7 function pointers (naturally 8bytes each) that are set up on loading of the executable by loader. (The detection of this mechanism was the key for building Mach-O executable file with Mops.) The functions are for dynamic linking, one of which is necessary for early system calls. Lazy symbol pointers section contains function pointers of the early system calls. Each of those function pointers are set at the first call time by the binding helper stub which is coded with `:CODE`-definition in iMops.




## 8.2 (Virtual) Memory Mapping of iMops Development Environment

The dictionary of development environment must be extensible. So the code and data dictionaries are allocated in the application's heap memory. All code and data stored in the executable file are copied into the respective part of the heap dictionaries in the setup process of iMops. And the execution of iMops, in reality, runs through the code in the heap dictionary, not in the loaded executable. The

executable file itself is loaded in the lower part of the memory (begins at $ 1000, which is standard for a normal Mac application), while the heap dictionaries are usually at higher place of the memory.

The value of `CODE_ORIGIN` is set to be the beginning address of the set up codes, that is, at the starting address of the page next to the end of the code generator part.

## 8.3 Memory Mapping of an Installed Application

An installed application usually doesn't need the extensible dictionary and code generator. So the code generator part will be chopped off in the installing process, and the execution in an installed application runs on the loaded executable. As the result, `CODE_ORIGIN` becomes equal to `CODE_START`.

The executable file is loaded at the address $ 1000 in the virtual memory. The first page (4kb) is for Mach-O header and commands, and the code dictionary begins at the second page. The executable file is loaded contiguously in the virtual memory for the application.

If you check the "With Code Generator?" checkbox in installing, the memory mapping of the built application is similar to iMops development environment.

## 8.4 Memory Mapping of a Dynamic Library

Dynamic libraries are usually loaded at higher address than the main application that uses the library functions. The dynamic library built by iMops is only for 64bit application since it uses 64bit machine instruction. So the preferred address for a library to be loaded is set at beyond 4 bytes address. That may prevent a 32bit application from loading a dynamic library built with iMops.

The preferred load address area begins from the address `$2A00000000` and is partitioned by 1MB step. That is, when you write 0 in the text field titled "Load area" on the dialog window, the preferred load address of the library will be set at `$2A00000000`. When you write 1 there, `$2A00100000`. And so forth. The step value in "Load area" pane should be a number between 0 and 100.

The preferred load address of a framework is a mere preferred address for the framework to be loaded and doesn't mean that the framework is preferred to any other code or data concerning the load place. The framework may be loaded at quite different address when there is something already loaded within the preferred load area of the framework. However, it shouldn't cause any trouble in linking against the framework.

(Uncompleted)