# FORCE Development

## 1. F1 → Gen-2

### Definition

Generation 2 has the following goals and features:

(1) Test framework

(2) FORCE, FORCEX, FORCEY, FORCEZ and FORCEN in final state

(3) ELF module in final state

(4) Native assembler for IA64

(5) Native assembler for IA32 (80386) 32+ (80486, Pentium extensions), IA16 (80286), IA8 (8086)

(6) Cross assembler for ARM32 and ARM64 → Raspi3B, Raspi4

(7) 8/16/32/64 bit FORCE compilers for all 3 modes.

(8) F2 (REPL/RECL in one application) capable of running on Linux

### Task List

#### Goal 1 → see «The Test Framework»

Elaborate a test framework for unit and integration tests.

This probably requires its own DSL(s) and thus its own vocabulary or -ies.

Use it in all tests described in the other goals below.

The final goal is to append the tests to the end of each *definition*.

#### Goal 2

Rewrite initial FORCE and split up into /force/64/FORCE, /force/64/FORCEX, /force/64/FORCEY, /force/64/FORCEZ and /force/64/FORCEN.

Write tests for all definitions

#### Goal 3

Rewrite /system/os/EFF (Executive File Format) templet with implementation /system/os/linux/ELF.

Write tests

#### Goal 4

Rewrite /force/64/asm/intel/Forcembler

Write tests for each statement and parameter combination

Rewrite /force/64/asm/intel/MacroForcembler as implementation of /force/64/asm/Forcembler templet

Write tests for all definitions

Make sure that every module importing any kind of assembler is tagged as "non-portable" (i. e. «contains machine code»)

#### Goal 5

Extract /force/32/asm/intel/Forcembler and its MacroForcembler from IA64-Forcembler

Derive tests for 32-bit-Forcembler from 64-bit-Forcembler

Extract /force/16/asm/intel/Forcembler and its MacroForcembler from IA64-Forcembler

Derive tests for 32-bit-Forcembler from 64-bit-Forcembler

Extract /force/8/asm/intel/Forcembler and its MacroForcembler from IA64-Forcembler

Derive tests for 32-bit-Forcembler from 64-bit-Forcembler

#### Goal 6

Write /force/32/asm/arm/Forcembler and its MacroForcembler

Write tests for each statement and parameter combination

Write /force/64/asm/arm/Forcembler and its MacroForcembler

Write tests for each statement and parameter combination

#### Goal 7

Rewrite Compiler + depedencies into /force/64/cmp/Compiler and other /force/64/cmp modules

Write tests

Extract /force/32/cmp/Compiler

Write tests

Extract /force/16/cmp/Compiler

Write tests

Extract /force/8/cmp/Compiler

Write tests

#### Goal 8

Rewrite Memory-linux as /system/mem/Memory templet with implementation /system/64/mem/linux/LinuxMemory  
→ Test code must be taken out of the module, because it requires IO, StringFormat and other dynamic modules

Rewrite IO as /system/io/ConIO templet with implementation /system/64/io/linux/LinuxConIO  
→ static allocation must be changed to dynamic

R

...

Assessment:

Memory is stateful but static, and can thus be used with static allocation.

with this module in place, everything else can be dynamically allocated.

# Vocabulary

A vocabulary is a word list

## Vocabulary Structure

A vocabulary consists of the following segments:

HEAD (the vocabulary header)

DICT (the dictionary segment)

CODE (the code segment)

HEAP (the data segment or heap)

TEXT (the text segment)

DBUG (the debug information segment)

### HEAD: The Vocabulary Header (Segment)

The vocabulary header defines the origin, version and compatibility of the vocabulary, as well as other vocabulary related parameters:

ORIGIN: origin of the code (optional, specified by supplier or by forthload).

PORTABLE: If the module is portable (true) or contains assembly code (false).

CODETYPE: The type of code (binary, direct threaded, indirect thread).

...

### DICT: The Dictionary (Segment)

The dictionary is used to *easily* and *quickly* find FORCE words in the vocabulary. In contrast to FORTH, words are not looked up in MRD (most recently defined) order, but by name. If a name cannot be found, an undefined symbol is created for it. When the name is defined later in the code, its address is registered with the undefined symbol, and thus the symbol becomes defined.

Remaining undefined symbols are assumed to be external symbols which will be resolved by the linker. In the compilation summary, a list of undefined symbols is printed if there are any.

#### The Dictionary Structure

The dictionary consists of a static part (stored in the dictionary segment) and a dynamic part, which is the hash table associated with it. The hash table allows to look up words very quickly in comparison to linear searches, which is particularly beneficial for FORCE. The hash table is not persisted, but can be rebuilt easily and quickly from the static part whenever a vocabulary is loaded.

The static part is a list of word entries with the following structure:

NFA (name field address)

CFA (code field address)

FLG (flags)

##### Name Field Address (NFA)

This address refers to the offset in the text segment where the word's name can be found.

##### Code Field Address (CFA)

This address refers to the offset in the code segment where the word's code can be found, regardless of whether the word is machine code or an address list. The word immediately preceding the code contains the code field length (CFL) — this word consists of a UTF-8 letter whose codepoint represents the length of the code following it; the length of this field can therefore be 1, 2, 3 or 4 bytes long. Note that the PFA may precede the CFL.

##### Parameter Field Address (PFA)

This address, if present according to the STATEFUL flag, points at a data field on the HEAP where the parameter field for the word is found. Words created with the CREATE (create) operation have a PF (even if it is zero bytes long), which includes VARIABLE (var), CONSTANT (const) and VALUE (val).

##### Flags

This field defines a couple of binary attributes for the word, particularly:

STATEFUL: indicates if the cell in the code immediately preceding the CFL is a pointer into the data segment where the word's state can be found.

INLINE: indicates if the code field should be copied instead of called, if possible.

UNINLINEABLE: indicates if the code field cannot be copied and must instead be called.

PRIVATE | PROTECTED | INTERNAL | PUBLIC: defines the word's visibility

...

Notes:

There is no such flag as IMMEDIATE as in FORTH. Words with special compiler semantics must be defined in the COMPILER vocabulary and will be executed by the compiler when they are found there. Thus, the control structure word IF defined in the FORTHX vocabulary does not contain the compiler code, but instead contains the code that needs to be executed when it is met by the interpreter (namely to either skip or interpret the next words in the input stream, until an ELSE or THEN token is encountered).

Small pieces of code (< 16 bytes) are always inlined, if inlineable. In order to inline a bigger word (if possible at all), it can be annotated with the INLINE flag.

### CODE: The Code Segment

The code segment contains the machine code generated by the assembler and compiler, as well as the address lists of indirect threaded execution models.

### HEAP: The Data Segment

The data segment contains static state which is referenced from inside the code. Data segment allows read/write access to its contents.

### TEXT: The Text Segment

The text segment contains text and other literals, among them the names of the words. Text segment allows read access to its contents only.

### DBUG: The Debug Information Segment

The debug information segment contains symbols and other information used by the debugger.

Structure

## Standard Vocabularies

All vocabularies are available in 8, 16, 32, and 64 bit versions for various processor architectures. If at some time in the future, there will be 128 bit processors, the vocabularies will support this , too.

### FORCE

The FORCE vocabulary contains the minimal FORTH base.

### FORCEX

The FORCEX or FORCE extension vocabulary contains extended FORTH / FORCE words in common use

### FORCEY

The FORCEY or FORCE advanced extension vocabulary contains extended FORCE words in less common use.

### FORCEZ

The FORCEZ or FORCE ultimate extension vocabulary contains esoteric FORCE words.

### FORCEN

The FORCEN or FORCE numeric extension vocabulary contains advanced natural numbers operations.

### CLAUSES

An optional compiler vocabulary defining compiler clauses.

### INTELX87 (Intel) (implements FLOAT)

The FORTHF or FORTH floating-point extension vocabulary contains X87 floating point operations.

### INTELMX (Intel) (implements NUMSTR)

The FORTHMX or FORTH streaming SIMD extension vocabulary contains MMX (SSE) operations.

### INTELXM (Intel) (implements NUMSTR)

The FORTHXM or FORTH streaming SIMD extension vocabulary contains XMM (SSE2+) operations.

### FLOATEMU (implements FLOAT)

The FLOATEMU vocabulary contains a floating point emulation for processors without a floating point co-processor.

### NUMSTREMU (implements NUMSTR)

The NUMSTREMU vocabulary contains an emulation of the numeric streaming operations for processors without numeric streaming capabilities.

### LINUX (implements OS and FILE)

The LINUX operating system API

### WINDOWS (implements OS and FILE)

The Microsoft Windows operating system API

### ELF (implements EXECFORM)

Contains an implementation of the ELF (Executable and Linking Format) object file specification predominant on Linux systems.

### PE (implements EXECFORM)

Contains an implementation of the PE (Portable Executable) object file specification predominant on Windows systems.

# Templet Vocabulary

A templet is a functional type of a vocabulary (think trait or interface for a class), for which the build configuration must name an implementation for the current (and, in case of cross-compilation, the target) system.

An implementation vocabulary must mention all templets it implements, as the compiler, binder and loader will check the compliance of vocabularies.

### OS

The OS templet defines operating system operations and is implemented by LINUX, WINDOWS, and others.

### FILE

The FILE templet defines file handling operations and can be implemented by LINUX, WINDOWS, and other. In this case, the templet is a subset of the implementations.

### FLOAT

The FLOAT templet defines the floating point API which is implemented, amoung others, by the INTELX87, INTELMX and INTELXM vocabularies

### NUMSTR

The NUMSTR templet defines the numeric stream operations API which is implemented, among others, by the INTELMX and INTELXM vocabularies.

### INTERPRETER

The INTERPRETER templet contains the interpreter (implementations for various platforms), particularly the REPL.

The interpreter vocabulary contains the code for interpreting words that are not found in the search list. These are mainly numeric, string, char, collection, function and other literals, as well as clauses.

### COMPILER

The COMPILER templet contains the compiler (implementations for various platforms), particularly the RECL, the block statements (IF / THEN / ELSE, DO / LOOP, BEGIN / WHILE / REPEAT / UNTIL / AGAIN etc.), as well as special constructs like ' (tick), and compiler clauses.

FORCE does not support immediate words — the compiler vocabulary seems to be a more flexible and natural approach to solve the problem of compiler semantics in a system that supports multiple vocabularies.

### ASSEMBLER

The ASSEMBLER templet contains the assembler (implementations for various CPUs).

### EXECFORM

The EXECFORM templet specifies the operations required to convert a module into an linkable or even executable program.

# Module

A module is a loadable vocabulary.

## Module Structure

Modules add the following segments to the vocabulary structure:

RELT (relocation table)

LSYM (local symbol table) contains the local or internal symbols

GSYM (global symbol table) references the global or external symbols

DEPT (dependency table)

STRG (string table)

# Clauses

## Introduction

A clause is a contraction of more than one dictionary word into a functional unit. In classic FORTH, there is an unsystematic approach to clauses (1+, 1−, ?DUPIF, ."), but to make use of the operational benefit of clauses (code reduction ⇒ speed enhancement), this would lead to the need for defining a zillion words like 32+, 13−, −15AND, 4<<, 404c+!, … the end of which would come with parsing hex clauses ($13and: does the a belong to the hex number or the and operation? Should we hence restrict hex numbers to uppercase and words to lowercase, or vice versa?)

In FORCE, two adjacent words (including usually literals) can be treated as a unit if a corresponding clause has been defined in the CLAUSES vocabulary. A CLAUSES-aware compiler would then defer punching words until it is clear that they are not belonging to a clause. This sounds complicated, but proved to be easy and straightforward to implement.

## Inventory

The following classes of clauses currently exist:

«condition condop»:   
0= IF, CY UNLESSEVER, U< UNTIL, > WHILE, < DUPIF, < 2DUPIF,

«number numop»:  
1 +, $0a and, 4 <<,

«stringlit stringop»:  
"..."abort, "..."|.,

…

## Why Clauses?

The reason for clauses are:

Readability: 24 and is slightly better readable than 24and

Disambiguation: $24 and is clear, while $24and could be read as $24a nd or $24 and.

Extensibility: 11+ (pseudo assembler code: ADD W, 11) is as valid a case as the more frequent 2+. (ADD W, 2) or the most frequent 1+ (INC W), but this would lead to an inflation of words.

Operation: the clause 0= IF can be assembled into two machine code statements, while the words separately would result in a huge overhead; 1+ generates INC W, while 1 + (not as a clause) would produce something like POP X // ADD W, X

# The Search List

The SearchList defines which words are found in which vocabularies first.

Object orientation adds considerable complexity to this topic, because each class and interface is supposed to live in its own vocabulary, and the direct and transient dependencies of classes among each other form a huge net of interdependencies influencing the search list.

On top of this, different modes of operations require different base setups of the search list.

## Compilation Search List

In the compiler search list, the search stack must include the following vocabularies:

COMPILER

CLAUSES

…

individual class and other target vocabularies

…

Words in the target vocabularies would be seen first, followed by clauses and then the special compiler words like IF, tick a. s. o. as well as the syntax and handling of literals and undefined words.

## Runtime Search List

In the runtime search list, the search stack must include the following vocabularies:

INTERPRETER

FORTH

…

individual class and other executive vocabularies

…

Clauses are not necessary here, because we won't do compilation optimization until we enter the compiler through a word invoking it (:, ]). But INTERPRETER will determine the syntax and handling of literals and undefined words.

## Imports

To make things maintainable, individual first-hand dependencies to other vocabularies have to be specified as imports inside the module.

Imports also have a visibility:

private imports are visible only to the module being compiled

protected imports are also visible to subclasses derived from this class

internal imports are visible to importers in the same package

public imports are visible to all importers

When an import is processed, first the module's dependency table is loaded. All visible (transient) dependencies will then be imported recursively first. After that, the dictionary section of the module will be loaded. This dictionary will be added to the global hash table by adding each word's entry by both its simple name, its composite name (vocabulary name + word name), and its fully qualified name.

### Compile-Time Behaviour

When a module gets compiled, the compiler starts with a base setup of the search-list (COMPILER, CLAUSES [if present]), then pushes the direct dependencies onto the search list in the order specified (i. e. the words in the last imported vocabulary will be seen first).

Compiler must probably also load modules fully

### Run-Time Behaviour

When a module is imported as a runtime extension, the interpreter starts with the current setup of the search list, which will include INTERPRETER and FORTH at the far end, followed by the so far imported vocabularies. The import will then load the full vocabulary, depedencies first, followed by the dictionary, from which the global hashtable is enriched, then appending or loading separately the sections in the module and finally performing relocation.

### On the Fly Source Compilation

If a module is out of date relative to its source, or if only the source file exists, the importer will instead read the source and run it through the compiler.

## Transient Dependencies

# The Interpreter Loop (REPL: Read/Eval/Print/Loop)

# The Compiler Loop (RECL: Read/Examine/Compile/Loop)

# The Test Framework

We want to use the by now classic given-when-then (also known as arrange-act-assert) test method specifying test setup (preconditions, INPUT), test procedure (action, PROCESS) and outcome validation (postconditions, OUTPUT).

## Test / Bracket

The keyword TEST: (with an optional name immediately following on the same line) opens the testing bracket, while TEST; closes it. Everything inside the test bracket is considered test code which will not be part of the module, but runs during compilation — ideally in its own isolated process which also allows for artifical preconditions to be set up, or deferred, after the binary module has been generated, immediately before it is deployed.

We assume a model in which the tests immediately follow each definition

## Given / Arrange

The keyword GIVEN (alias ARRANGE:) introduces a new test. The following words specify the preconditions / test setup.

## When / Act

The keyword WHEN: (alias ACT:) introduces the core section that invokes the testee — normally, there is just one invocation of the testee per test setup, because the invocation itself will cause the test setup to change (if only the stack).

## Then / Assert

The keyword THEN: (alias ASSERT:) introduces the section that tests the outcome. In this part, care must be taken with each assertion to not change the test context in a way that impacts on the following assertions.

## Skeleton Example

A simple example from the test code of dup:

TEST: make sure that DUP exactly duplicates the TOP  
GIVEN 12345678 ( we use a particular stack entry )  
WHEN: dup ( invoke testee )  
THEN: != ( assert that both stack entries are the same )  
TEST;  
  
TEST: DUP should gracefully fail on an empty stack  
GIVEN SP0! ( a clear stack )  
 capture{ ( capture exceptions )  
WHEN: dup ( invoke testee )  
THEN: }capture ( end capturing exceptions)

captured@ ParameterStackUnderflow !=

( make sure captured exception is the expected one )

TEST;

Of course, the examples above are so simple that they could actually be written as:

TEST: 12345678 dup != TEST;  
TEST: SP0! capture{ dup }capture captured@ ParameterStackUnderflow != TEST;

But it is a good habit to always assume that a test about to be written could eventually be more complex than anticipated.

# Classes and Objects

## Object Classes

A class is a Vocabulary with additional functionality.

A class instance holds a static list of all its instances.

…

## Referencing object invocations

Sample code:

String String Hashtable class Dictionary

: add ( key:String val:String self -- ) ... ;

: lookup ( key:String self -- val:String\_Option ) super lookup ;

class;

Dictionary new val dict

Dictionary var dict2

Dictionary new =var dict3

"hello" "world" dict add "good-bye" "cruel world" dict add ;

dict3@ .[

"I" "my" . add

"thou" "thy" . add

"he" "his . add

"she" "her" . add

"it" "its" . add

"we" "our" . add

"ye", "your" . add

"they" "their" . add

].

: translate ( key:String – val:String ) dict get ;

: get ( key:String dict:Dictionary -- val:String )

lookup String Option . any? if . some else

NoTranslationException raise then ;

class Dictionary is a subclass of Hashtable with two type parameters (both String).

It defines two methods, add and lookup, to add a and look up a mapping respectively.

The add method takes two String arguments and a reference to the current object instance (self), but returns nothing.

The lookup method takes a String key and a reference to the current object instance (self), and returns the associated mapping as a String Option.

Dictionary value dict is defined to hold a new instance of Dictionary.

Variable dict2 is defined to be assignable with an instance of type Dictionary. Its initial value is set to the default value, NIL.

Variable dict3 is defined by assigning it an instance of type Dictionary.

Both dict2 and dict3 can hold an instance of class Dictionary or any subclass of it.

The dictionary dict is initialized with 2 key-value pairs. The object to invoke methods on ("current object") is selected with the . keyword. The "current object" is a separate register in a thread.

The dictionary in varibale dict3 is fetched using the global getter dict3@. This dictionary is initialized with 8 key-value pairs using a *with*-Block (.[ ... ].). Inside the *with*-Block, the object instance mentioned immediately in front of it, is available as ., and ]. restores the previous "current object"

The . keyword also has the compile-time effect of specifying the current object type. The second line inside get shows this: String Option . is an expression with a vocabulary argument — this will cause the TOP to be assigned to "current object" of type String Option, while the  
. some later on the line picks up this "current object".

The static function translate will return the value of a mapping in dict by invoking the extension function get on it. Extension functions can be used like methods of the type specified as the last function arguments (here: Dictionary), but they are defined outside of the class proper.

In this case, get is similar to lookup, but instead of returning a String Option, it will return a String and fail by raising a NoTranslationException if a mapping was not found.

The "current object" is preserved through invocations until another one is selected; upon EXIT, the previous "current object" is popped along with the return address. Accordingly, the "current object" is pushed together with the return address upon ENTER.

# Function Argument Definitions

: rot ( x1 x2 x3 -- x2 x3 x1 ) is the classical FORTH function signature with a stack comment that has no compile-time effect. In FORCE, the signature for the same function is:

: rot ( x1:Any x2:Any x3:Any -- x2 x3 x1 )

The function arguments are attached a type (here, all three arguments are of type Any, which represents any type, even a primitive number or an untyped address). To the right of the --, return values already mentionned on the left side don't need to be redeclared.

Furthermore, to the compiler, all arguments or return values without an assigned type are of type Any; so for the rot-Function, the signature can actually stay the same like in classic FORTH — *if* the ARG\_TYPE\_CHECK environment variable / command line argument is set to RELAXED (default). If ARG\_TYPE\_CHECK is set to STRICT instead, the compiler would mark the missing type as an error.

## Special Types and Argument Names

The following special types are available:

Any: represents all possible types, including primitive numbers and generic pointers

para\_generic (as in String\_List, String\_C4\_Map, Any\_Option) represents a generic type as it would be specified by replacing all underscores to spaces.

…

The following argument names have an implicit type:

self (inside a class) is bound to the type of the class itself.

…

…

## Extension Attributes

## Widening Conversions

### Numeric Classes

There are 2 × 5 Integer classes and 2(3) Real classes:

#### Cardinals (C) aka unsigned integers

C1 alias Unsigned Byte «C»[[1]](#footnote-1) (0 … 255)

C2 alias Unsigned Word «W» (0 … 65,535)

C3 alias Unsigned Doubleword «D» (0 … 4,294,967,295)

C4 alias Unsigned Quadword «Q» (0 … 18,446,744,073,709,551,615)

C5 alias Unsigned Octword «O» (0 … 3.4028236692093846346337460743177 × 10³⁸‬)

#### Integers (I) aka signed integers

I1 alias [signed] Byte «B» (−128 … 127)

I2 alias [signed] Word alias Short Int «S» (−32,768 … 32,767)

I3 alias [signed] Doubleword alias Int «I» (−2,147,483,648‬ … 2,147,483,647)

I4 alias [signed] Quadword alias Long Int «L» (−9,223,372,036,854,775,808‬ … 9,223,372,036,854,775,807)

I5 alias [signed] Octword alias Huge Int «H» (−1.70141183460469231731687303715 × 10³⁸‬ … ‬1.70141183460469231731687303715 × 10³⁸)

#### Real (R) aka floating point numbers

R4 alias Float

R8 alias Double

RT alias Tenbyte (on Intel processors since 80386, and before on Intel 80x87 coprocessors)

### String Classes

There are 4 String classes differing mainly in size:

STRING1 alias Short String (max. size 255), the class of most String literals

STRING2 alias Medium String (max. size 65,535)

STRING4 alias Long String (max. size 4,294,967,295‬)

STRING8 alias Huge String (max. size 18,446,744,073,709,551,615‬)

1. from trad. **C**har — today Characters are 32-bit quantities, but the ‹C› remained for unsigned bytes [↑](#footnote-ref-1)