35th EuroForth Conference

September 13-15, 2019

Hamburg
Germany

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 35th EuroForth finds us in Edinburgh for the first time. The two previous EuroForths were held in Bad Vöslau, Austria (2017) and in Edinburgh, Scotland (2018). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there was one submission to the refereed track, which was accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 28 submissions, 20 accepts, 71% acceptance rate. The paper was sent to three program committee members for review, and they all produced reviews. The reviews of all papers are anonymous to the authors: The paper was reviewed and the final decision taken without involving the authors. This year the only submission was co-authored by the program chair; Ulrich Hoffmann served as secondary chair and organized the reviewing and the final decision for that paper. I thank the authors for their papers and the reviewers and program committee for their service.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. These online proceedings (`http://www.euroforth.org/ef19/papers/`) also contain papers and presentations that were too late to be included in the printed proceedings. Also, some of the papers included in the printed proceedings were updated for these online proceedings.

Workshops and social events complement the program. This year's EuroForth is organized by Ulrich Hoffmann.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Ulrich Hoffmann, FH Wedel University of Applied Sciences (secondary chair)
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas, SC3D Ltd.

Contents

The new Gforth Header

Bernd Paysan
net2o

M. Anton Ertl*
TU Wien

Abstract

The new Gforth header is designed to directly implement the requirements of Forth-94 and Forth-2012. Every header is an object with a fixed set of fields (code, parameter, count, name, link) and methods (`execute`, `compile,`, `(to)`, `defer@`, `does`, `name>interpret`, `name>compile`, `name>string`, `name>link`). The implementation of each method can be changed per-word (prototype-based object-oriented programming). We demonstrate how to use these features to implement optimization of constants, `fvalue`, `defer`, `immediate`, `to` and other dual-semantics words, and `synonym`.

1 Introduction

Forth started out with a word header (Fig. 1) that satisfied several requirements. As additional requirements arose, the header was adapted, resulting in the fig-Forth header and eventually the old Gforth header. A number of requirements were tacked onto the existing header design, resulting in a maze of `if`s when text-interpreting a word.

Yet more requirements came along that necessitated extending the header again. At that point Bernd Paysan decided to perform a major redesign of the header, following [Bro84, Tip 8.13]: "Use decision tables". The present paper explains this new design.

We start out by discussing header-related requirements in Standard Forth and in Gforth (Section 2); in particular, we discuss the desire to use the execution token as name token (Section 3). In Section 4 we describe the fields and methods of the new Gforth header, and related implementation issues. Section 5 shows examples of using the header features to implement words that the old header does not support well, such as `to` or `synonym`. Section 6 compares the performance of the old and new header. Finally, we look at related work in Section 7.

In this paper we use "Gforth 0.7" to represent old Gforth and Gforth 1.0 to represent new Gforth. Most of the statements about Gforth 0.7 are also true for several older versions; Gforth 1.0 has not been released yet, but all the code presented here works with Snapshot `gforth-0.7.9_20190829`.

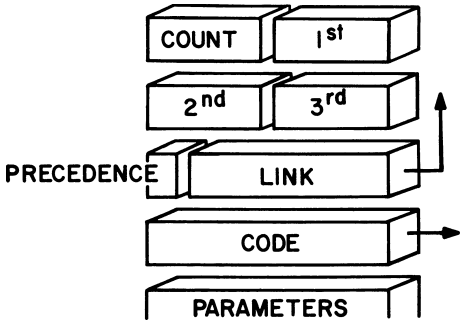


Figure 1: The original Forth word header [Moo74]

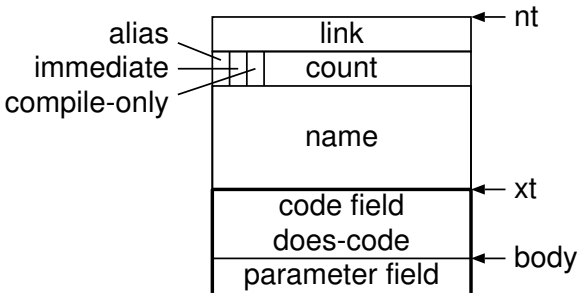


Figure 2: The Gforth 0.7 header

We use the notation `]] a b c [[` instead of `postpone a postpone b postpone c` for compactness and readability. We also use `find-name ( c-addr u -- nt )`. We embed nameless colon definitions inside colon defintions with the syntax `[: ... ;]`; this pushes the xt of the nameless colon definition. The Forth-2012 locals syntax `{: ...-- ... :}` defines the names before `--` as locals in stack effect order and treats the rest as comment.

2 Requirements

The old Gforth header (Fig. 2) is not that far from the original Forth header; it has more flags, a varible-length name field, a two-cell code field, and the link field has moved, but it mostly still has the same fields. It satisfies the following requirements:

*anton@mips.complang.tuwien.ac.at

Execution semantics The code address in the first cell of the *code field* determines which kind (colon definition, constant, etc.) the word is; for children of `create...does>` the *does-code* part points to the code after the `does>`. The body in the *parameter field* contains data (e.g., the value of a constant) or the threaded code of a colon definition.

These fields are all that is necessary for `execute` and `compile,`, so words defined with `:noname` or `noname` have only these fields (indicated by the thick line in Fig. 2).

`immediate` The *immediate bit* makes it possible to differentiate between words with default compilation semantics and immediate words (where the compilation semantics is the execution semantics).

`compile-only` Setting the *compile-only*[1] *bit* removes the interpretation semantics of a word; trying to interpret it produces an error on Gforth 0.7.[2]

`alias` An alias is a word with a different name (or in a different wordlist), but the same xt as another word. They are marked by setting the *alias bit*, and instead of a code field they contain the xt of the word they alias to.

Traverse wordlist e.g., for `words` or for building the hash table; the *link*, *count*, and *name field* are used for that. Words are usually searched through the hash table (a separate data structure), but locals are searched directly by traversing the wordlist.

Colon definition visibility The name of a colon definitions becomes visible only at the end. Gforth inserts (`reveal`s) the new header into the current word list at that time, and therefore does not need a `smudge` bit. We kept this approach in the new header, and therefore hardly mention it in the rest of this paper.

A number of additional requirements were addressed without redesigning the header:

`Compile,` In primitive-centric threaded code [Ert02] each word is `compile,`d to a primitive, usually with an inline argument, e.g., `call` *body* for a colon definition. Different word types have to be compiled to different code. In Gforth 0.7 `compile,` contains a big `case` control structure for this that looks at the contents of the code field.

Dual-semantics words (aka NDCS words [Pel17]) The compilation semantics of words like `s"` and `to` are neither default nor immediate. So these words have an interpretation semantics represented by one xt and a compilation semantics represented by a separate xt. In Gforth 0.7 such words have a special code field, and the two xts are stored in the parameter field.

Gforth uses `name>interpret` and `name>compile`[3] to access interpretation and compilation semantics. In Gforth 0.7 they abstract the following complexity: These words check for this special code field and produce the appropriate xt or ct (compilation token) [Ert98]. `Name>interpret` also looks at the compile-only bit. `Name>compile` also looks at the immediate bit. Both words also look at the alias bit.

`To` Consider `to` *name*: The action performed at run-time depends on how *name* was defined (e.g., with `fvalue` or as a (cell-sized) local). Again, there is a big `case` that looks at the code field of *name* to decide what to do.

As a consequence, in Gforth 0.7 text interpretation performs quite a bit of conditional control flow for every interpreted word. The new header eliminates most of this conditional control flow.

The following requirement is satisfied in the new header, but not reliably in the old header:

`>name` Reliably get from "the" xt of a word to its nt; this is useful in, e.g., a decompiler. Gforth 0.7 implements `>name` with a heuristic that usually works, but can also produce wrong results.

3 XT and NT

Originally, every word had one name and one semantics/action (the execution semantics) which also served as interpretation semantics, and from which either default or immediate compilation semantics were derived. Representing such words through a single address or token is a good idea.

Later, Forth acquired features that turn the relation between "named words" and "semantics" from 1:1 to m:n. Therefore, these separate concepts need separate tokens: execution token (xt) for semantics/actions; and name token (nt) for named words. Yet there is a strong desire among Forthers for a unified token, leading to repeated discussions about the necessity of nts.

[1] aka `restrict`

[2] In Gforth 1.0 all words have interpretation semantics.

[3] Called `name>int` and `name>comp` in Gforth 0.7.

3.1 New Gforth approach

This desire was also strong in the design of the new Gforth header, leading to the following design goals:

- For many words, the nt and the xt representing the interpretation semantics are the same address.
- You can pass an xt to a word that expects an nt, and you will get a plausible result.
- You can pass an nt to a word that expects an xt, and you will get a plausible result.

Most defining words produce words where the interpretation xt=nt. Exceptions are `interpret/compile:` (which defines a dual-semantics word), `synonym` and `alias`. There are other xts associated with a word, but they are usually thought of as being "the xt" of a separate word. E.g., `name>compile` returns "the xt" of `execute` or `compile,` on the top-of-stack (and another xt beneath it).

Words defined with `:noname` and `noname` can also use the xt as nt. `name>string` produces an empty string, and `name>compile` produces default compilation semantics.

Note that, for some words, `name>interpret` is not a noop, and `>name` does not get you back to the nt where you came from. This demonstrates that there is a conceptual difference between nt and xt. This is documented in the stack effect of words:

```
name>interpret ( nt -- xt|0 )
name>compile   ( nt -- w xt )
immediate?     ( nt -- flag )
name>string    ( nt -- c-addr u )
execute        ( ... xt -- ... )
compile,       ( xt -- )
>body          ( xt -- a_addr )
>name          ( xt -- nt|0 )
xt>name        ( xt -- nt )
```

Note that `xt>name` is a no-op; it documents that there is an xt on the stack before, and an nt afterwards.

In the following we present examples that demonstrate the difference. First we define some words:

```
: b ." b" ;
: c ." c" ;
:noname ." d" ; alias d
' b alias e immediate
synonym f b
' b ' c interpret/compile: g
create t
' t alias u
synonym v t
' t ' c interpret/compile: w
```

In the examples, the input is shown in **bold**, while the output is shown in blue.

Here, xt=nt.

```
s" b" find-name ' b = . -1  ok
s" b" find-name execute b ok
' b name>string type b ok
```

But in the following examples, xt≠nt:

```
s" d" find-name ' d = . 0  ok
s" f" find-name ' f = . 0  ok
s" g" find-name ' g = . 0  ok
```

Xts are usable as nts, but the nt is of a different word:

```
' d name>string type  ok
' e name>string type b ok
' f name>string type b ok
' g name>string type b ok
```

`Execute` and `compile,` exhibit the same behaviour for nts as for their interpretation xts:

```
s" d" find-name execute d ok
s" f" find-name execute b ok
s" g" find-name execute b ok
: d1 [ s" d" find-name compile, ] ; see d1
: d1
  ; ok
: f1 [ s" f" find-name compile, ] ; see f1
: f1
  b ; ok
: g1 [ s" g" find-name compile, ] ; see g1
: g1
  b ; ok
```

But `>body` does not maintain this illusion; we decided to make it a simple field access word rather than a method; if you want to get from the nt to the body of the interpretation xt, use `name>interpret >body`:

```
s" u" find-name >body u = . 0  ok
s" v" find-name >body v = . 0  ok
s" w" find-name >body w = . 0  ok
```

There are other xts associated with some words:

```
s" g" find-name name>compile  ok 2[4]
name>string type execute ok 1
name>string type c ok
```

In Gforth, immediacy is a property of a named word, not of the xt:

[4] Gforth~1.0 shows the number of items on the stack after `ok` unless the stack is empty.

```
s" e" find-name immediate? . -1  ok
s" b" find-name immediate? . 0  ok
: h1 e ; b ok
h1  ok
: h2 b ;  ok
h2 b ok
```

For some words, there is a difference between compilation semantics and `compile,`ing the interpretation semantics:

```
e \ interpretation semantics b ok
: j1 e ; \ compilation semantics b ok
j1  ok
: j2 [ ' e compile, ] ;  ok
j2 b ok
g b ok
: i1 g ; c ok
i1  ok
: i2 [ ' g compile, ] ;  ok
i2 b ok
```

3.2 Alternatives

The new Gforth approach is not for every Forth system, but every system has to deal with tokens for named words and tokens for semantics/actions. The difference between these concepts is most pronounced for separated-header systems where the name field and link field of a word are separated in memory from the code field and parameter field (e.g., to make it easy to produce headerless code). Here we explore the options for implementing nt and xt (for interpretation semantics) in this setting; we do not discuss implementing, e.g., dual-semantics words.

XT≠NT

The most straightforward approach is to let nt point to the name field or link field, and let the xt point to the code field or parameter field. You don't get a unified token, and you can only use xts for words that take xts, and only use nts for words that take nts. Getting from the xt to the nt is either slow[5] or requires adding a back pointer to every code field.

NFA as XT=NT

This approach works, but has several disadvantages:

- The name field part is needed whenever an xt is used (e.g., for `execute`), and therefore leaving the name headers away is either much harder or can be used only for applications that don't use xts (e.g., no `execute`, `is`, or `>body`).
- Words that consume an xt, in particular `execute` and `>body`, perform an extra indirection.

Other properties: A synonym has a different xt than its original; the code field can be stored with the name and link fields and separate from the parameter field.

CFA as XT=NT

This does not work with the most obvious way to implement this separated-header concept: If a synonym is implemented as a name that has the same code field as the original, both words have the same nt=xt, and `name>string` will produce the same name. Implementing `words` with `traverse-wordlist` and `name>string` will not show the name of the synonym, but show the name of the original for every synonym. E.g., with the definitions above we would get

```
s" f" find-name name>string type b  ok
```

One can complexify the implementation to avoid these problems, but the result is probably less attractive than the alternatives above.

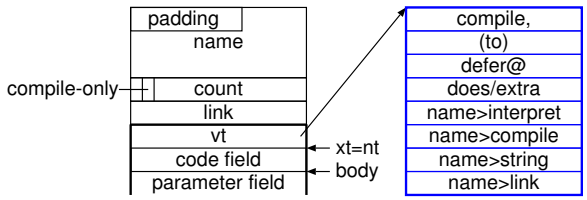


Figure 3: The new Gforth header. The thick lines surround fields present in nameless words.

4 The New Header

4.1 Organization and Fields

Figure 3 shows the new header. The xt/nt points to the code field, and the header structure grows in both directions from there, with variable-length fields (name field and parameter field) at both ends. The fields are:

code field This field contains the *code address* of native code that implements the execution sementics of the word (see also the `execute` method in Section 4.2).[6] The xt directly points to this field.

[5] If the separated headers were stored in an array and if we had the same order in memory for the separated headers and for the separated bodies (and code fields), we could use binary search for the body address in the headers. However, a straightforward implementation of `synonym` means that a header may point to an out-of-order body, so binary search cannot be used.

[6] Note that while NEXT and the threaded code in a colon definition use direct-threaded dispatch, `execute` (and other

vt This *virtual table* field points to a table of implementations for the various methods (see Section 4.2). All method implementations are represented by xts. You can get to this field with `>namevt ( xt -- addr )`.

parameter field This field serves the same purpose as it has since the dawn of Forth: It contains additional data or code, depending on the word type, e.g., the value of a constant, or the threaded code of a colon definition. You can get to this field with `>body ( xt -- addr )`.

Unnamed words have only the fields above (indicated by the thick line in Fig. 3). Named words have the following fields in addition:

count field This (almost) cell-sized field contains the length of the name in bytes. You can get to it with `>f+c ( nt -- addr )`, but it is better to get the count with `name>string ( nt -- c-addr u )` which also works on nameless words.

name field This field contains the name of the word. The arrangement is unusual in having the name before the count field; this is due to the variable-length nature of the name field, and because we already have the variable-length parameter-field at the end of the word header. The actual name starts *count* bytes *before* the count field; before the actual name, there is padding that ensures that the parameter field is maximally aligned (and, as a consequence, the other fields are cell-aligned). You can access the name with `name>string`.

compile-only flag While most flags have been eliminated by using methods, we kept the compile-only (aka `restrict`) flag with a much-reduced role: Gforth now has interpretation semantics for all words, but warns in some cases when the interpretation semantics of compile-only words are used. You can use `compile-only? ( nt -- flag )` to check whether a word is compile-only.

link field This field is just like the link field since the dawn of Forth: it contains the nt of the previous word in the same wordlist, or 0 if there is none. You can get to this field with `>link ( nt -- addr )`, but the method `name>link ( nt1 -- nt2 )` works even on nameless words.

Variations

A possible variation of this organization is to move the code field into the virtual table. This would mean that a word header would be one cell smaller, and, for a fixed virtual table (as for closures [EP18]) would be cheaper to create. The disadvantages would be: Each `execute` and other dispatch through the code field would incur an additional indirection. And we would need an additional virtual table for every primitive (each primitive has a different code address); with 413 primitives in Gforth, this would mean 3717 cells for virtual tables, i.e., 3717 words to break even; Gforth on AMD64 has 3922 named words, and a number of unnamed ones, so, before loading user code, the benefit of this size optimization is small.

To avoid the additional indirection, instead of storing the code address in the virtual table, a copy of the code itself could be stored there. Another way to look at this would be that the virtual table would be prepended to the code; the code would need to be duplicated for different virtual tables. With this variation, the vt field would again become a code field.

4.2 Methods

There are various words that take an nt or xt; the behaviour of many of these words depends on the nt/xt. In the old header conditional/`case` code decided what to do by looking at header flags or at the contents of the code field.

The new header uses an object-oriented design[7]: These words are method selectors, and the method implementations are determined when the word is defined. This section describes the methods; defining them is covered in Section 4.5. Apart from the `execute` method which is implemented in native code and represented by a native-code address, all methods are implemented as Forth code represented by an xt.

`Execute ( ... xt -- ... )`

`Execute` takes an xt and performs the semantics represented by that xt. This has been treated as a method since the dawn of Forth: Different word types have different (native) code addresses in their code field;[8] for some word types (colon definitions, `does>`-defined words), this (native) code then calls threaded code that allows varying the behaviour of the word by writing Forth code.

words going through the code field) use indirect-threaded code dispatch [Ert02].

[7] The design is for a fixed set of methods, not for general object-oriented programming. Mini-OOF is a general object-oriented Forth extension with similar features.

[8] In indirect-threaded code; direct-threaded and native code systems have some differences in details, but they also execute some native code when `execute`ing an xt.

applicable to	generated code	`set-optimizer` by	executed Aarch64 instructions
all words	lit <xt> execute	set-execute	14
`does>` words	does-xt <xt>	set-does>	11
constants	lit <value>	constant	2

Figure 4: Different correct implementations of `compile,` for a word defined with `constant` as shown in Section 5.1

`does ( ... body-addr -- ... )`

For a word defined with `set-does>` or `does>`, the native code pushes the body of the word *body-addr* on the stack, and then executes the xt passed to `set-does>` (or the xt representing the code following the `does>`. This xt is the `does` method of the word.

The `does` method is only used by words defined with `set-does>`/`does>`. Its slot can be used for other purposes by other word types; it is then called `extra`.

`Compile, ( xt -- )`

A classic indirect-threaded Forth system uses `,` to `compile,` an xt. But Gforth (since version 0.6) uses primitive-centric threaded code [Ert02], where different words have to be compiled differently ("intelligent `compile,`"); e.g., a colon definition is compiled to the primitive `call` followed by the body address of the called word.

The significance of `compile,` is that 98.8% of the dynamically executed primitives[9] are in threaded code produced by `compile,`, while only 1.2% of the primitives[10] are invoked through `execute` or a deferred word, which use the `execute` method above. So improving the code generated with `compile,` for a word has a much greater effect than improving its `execute`d code.

In Gforth 0.7, `compile,` is implemented as a `case` control structure that looks at the code address in the code field. In Gforth 1.0, `compile,` is a method, so new word types and their code generators can be added without changing existing code.

Note that `compile,` has to be equivalent to

```
: compile, ( xt -- )
  ]] literal execute [[ ;
```

Many uses of `compile,` rely on this relation between `execute` and `compile,`. Every implementation of the `compile,` method has to satisfy this equivalence, and therefore words like `set-does>` that change what `execute` does for a word also change what `compile,` does for a word. To get an optimizing `compile,` implementation, you have to call `set-optimizer` afterwards.

[9] in https://www.complang.tuwien.ac.at/forth/peep/sorted

[10] 1% calls, 0.2% other primitives

There have been attempts to define the `compile,` for some words in a way that does not satisfy this equivalence, as a shortcut to implementing dual-semantics words, but this does not work correctly in all cases. To implement dual-semantics words, set `name>compile` and/or `name>interpret` (see Section 5.5 and 5.6).

Figure 4 shows the effects that different `compile,` implementations have for compiling a constant as defined in Section 5.1, corresponding to different levels of specialization, resulting in fewer executed instructions for the more specialized code.

`Name>interpret ( nt -- xt )`

The Forth-2012 word `name>interpret` has been in Gforth since 1996 under the name `name>int`, but it used a number of `if`s that looked at various flags and fields to produce the correct xt. In Gforth 1.0, it is a method. For normal words (where nt=xt), it just performs a no-op.

`Name>compile ( nt -- xt1 xt2)`

The Forth-2012 word `name>compile` has been in Gforth since 1996 under the name `name>comp`, and it also used a number of `if`s to do its work. In Gforth 1.0 it is a method; for normal words, it just pushes the xt of `compile,`; for ordinary immediate words, it pushes the xt of `execute`. Examples of other implementations will be shown below.

`(to) ( val xt -- )`

Words defined with, e.g., `value` or `fvalue` can be used with `to`. What to do then depends on how the word was defined.

In Gforth 1.0, `to` and `is` are synonyms, so `to` can also be used for words defined with `defer`.

In Gforth 0.7 `to` *`name`* looks at the code field of *name*, and has a `case` structure for all the types of words known to it. Adding a new type requires changing the code of `to`.

In Gforth 1.0, we have a method `(to)` that performs the actual storing of the value *val* (of any type) into the word specified by xt. E.g., for a word defined with `fvalue` the implementation of `(to)` is (simplified):

```
: fvalue-to ( r xt -- )
  >body f! ;
```

One can see this method as the implementation equivalent to the approach used to specify `to` in the Forth-2012 document[11]: There, `to` first performs the generic part of `to`, and then the word-type-specific `to` *name* semantics; and these semantics are specified at each defining word.

`Defer@ ( xt1 -- xt2 )`

Gforth 1.0 also has several `defer`-like defining words, e.g., the standard `defer` and the per-task `udefer`. Words defined with these words all behave like words defined with `defer`: Running such a word executes the xt stored in it; that xt can be read with `defer@` or `action-of`, and it can be written with `defer!` or `is`.

Gforth 1.0 defines `defer@` as a method, which allows different ways of accessing the xt for the different word types. `Action-of` uses `defer@`. `Is` is a synonym for `to`, which makes `Defer!` a synonym of `(to)` (see above). Therefore we do not need a separate `defer!` method.

`Name>string ( nt -- c-addr u )`

In Gforth words defined with `:noname` and `noname` have no name field. Therefore `name>string` is a method:

- For named words, it just returns the address of the name field and the number of characters.
- For unnamed words, it returns an empty string.

`Name>link ( nt1 -- nt2|0 )`

`Name>link` is a method that provides the functionality of a link field even for (unnamed) words which have no link field.

Grouping

These methods can be divided into the following groups[12]:

- `Name>string` and `name>link` each have only two implementations: One for named words and one for noname words. They are otherwise independent of the other methods. Instead of implementing them as methods, one could also use a *noname* bit in the header, and implement these words with `if`.
- `Name>interpret` and `name>compile` define the semantics of named words. They make sense only for named words.
- `Execute`, `does`, `compile,`, `defer@` and `(to)` (in its role of `defer!`) define the behaviour of an xt, and are relevant for unnamed as well as named words.

These differences are also visible in the stack effect: The first two groups take nts, the last group an xt (or body).

Factoring

For the most part, these words were not introduced for this header design, but already existed earlier, and we just turned them into methods:

word	year	origin
`execute`	1970s	early Forth [Moo74]
`does`	1970s	early Forth
`compile,`	1994	Forth-94
`name>interpret`	1996	Gforth 0.2
`name>compile`	1996	Gforth 0.2
`defer@`	2005	Forth 200x
`name>string`	1996	Gforth 0.2

`(To)` corresponds to the "`to` *name* run-time" semantics factoring in Forth 200x introduced in 2009[13].

Only `name>link` was introduced with this header design.

So most of these words have proven their worth as factors of other words for a long time, but are they also good interfaces for method implementations? In our experience they are, and you can judge for yourself by reading Section 5.

One word that has been questioned is `name>compile`, because it represents compilation semantics with two xts instead of just one. The benefit is that `name>compile` is cheaper to implement for normal words and immediate words (i.e., the vast majority of words). Dual-semantics words require an extra layer (see Section 5.5), but that can be done once for all such words (Section 5.6).

4.3 `Find`

While the new header design does not change our implementation of `find` (we already implemented `find` based on `name>interpret` and `name>compile` in the old header), this is a related topic, and may be of interest to the reader. It also is relevant by showing why we do not want to exploit all the flexibility that the two xts returned by `name>compile` offer.

On success, `find` returns an xt and either 1 (immediate[14]) or −1 (otherwise). `Find`-based

[11] http://forth-standard.org/standard/core/TO

[12] These groups could be considered as *interfaces* or *traits* in object-oriented programming languages.

[13] http://www.forth200x.org/documents/forth09-3.pdf

[14] Actually, assuming that `find` should be usable for user-defined text interpreters, we need a different notion of *immediate* than "compilation semantics = execution semantics": If `find` returns an xt in compile state that the user-defined

user-defined text interpreters either `execute` or `compile,` the xt, depending on `state` and the number returned by `find`.

If `name>compile ( name -- xt1 xt2 )` returns either `execute` or `compile,` as xt2, `find` can be implemented in a way that allows such text interpreters:

```
: find {: c-addr -- c-addr 0 | xt 1/-1 :}
  c-addr count find-name dup if
    dup name>compile >r swap name>interpret
    state @ if drop else nip then
    r> ['] execute = if 1 else -1 then
  else
    drop c-addr 0
  then ;
```

The `state`-dependent part caters for dual-semantics words where the xt returned by `name>interpret` is different from the xt1 returned by `name>compile`. The part afterwards extracts the $1/-1$ from the xt2 returned by `name>compile`.

Our implementations of `name>compile` all heed the restriction mentioned above, and that is probably a good idea for all systems that implement `find`.

4.4 `To` optimization and locals

A simple way to compile `to` *name* is to compile a literal for the xt of *name*, followed by `(to)`. But what we actually do is to resolve the method dispatch already at compile time, and compile the implementation of `(to)` for that xt.

But optimization does not stop there. E.g., in the `fvalue-to` case above, we want to resolve the `>body` during compilation, resulting in compiling the body (instead of the xt) as literal, and compiling `f!`, eliminating the colon definition overhead and the `>body` at run-time. We first define the `compile,` implementation for `fvalue-to` as follows (simplified):

```
: fvalue-to-compile, ( xt -- )
  drop ]] >body f! [[ ;
```

We can use this word with `set-optimizer` (see Section 4.5):

```
: fvalue-to ( r xt -- )
  >body f! ;
' fvalue-to-compile, set-optimizer
```

Another syntax that does not require a name for the `compile,` implementation is to use `opt:`:

```
: fvalue-to ( r xt -- )
  >body f! ;
opt: drop ]] >body f! [[ ;
```

text interpreter should `execute`, `find` should also return 1.

We have also implemented Mecrisp's constant folding mechanism [Koc15] in Gforth [Pay19][15]: Compiling a literal pushes it on a compile-time literal stack; `compile,` implementations can access the literal stack to perform operations on the constants at compile time. In the present case, compiling `>body` takes *name*'s xt from the literal stack and pushes *name*'s body. Before generating actual code, the remaining contents of the literal stack are compiled as literals. In the present case, the body is compiled as literal before compiling `f!`. The end result is that `to` *name* (where *name* is an fvalue) is compiled to the same code as

```
[ ' name >body ] literal f!
```

In most cases the optimization is just nice to have. But there is one case where optimizing `to` *name* is essential: locals. Gforth does not keep the headers of locals around until run-time, so using the xt of a local at run-time would not work. The `(to)` implementation of a (cell-sized value-flavoured) local is called `to-w:`. The `compile,` implementation of `to-w:` is (simplified):

```
: to-w:-opt ( xt-to-w: -- )
  ?fold-to >body @ lp-offset ( offset )
  ]] laddr# [[ , ]] ! [[ ;
```

`Compile,` passes the xt of `to-w:` to `to-w:-opt`. `?fold-to ( xt-to -- xt-name )` drops this xt and moves *xt-name* from the literal stack to the data stack.[16] The rest of `to-w:-opt` computes the *offset* of the local from the locals-stack pointer (first line), and then (second line) compiles code for generating the run-time address of the local (`]] laddr# [[ ,`), and for storing the value there (`]] ! [[`).

Similar optimizations are also used for the `defer@` method (and, based on that, `action-of`).

Alternatives to the scheme above are:

- Keep the headers of locals around for the whole session. Then you can just use an unoptimized or minimally optimized implementation of `to` *name*.
- Have additional methods equivalent to `]] literal (to) [[` and `]] literal defer@ [[`. Then you do not need constant folding to avoid having to deal with the xts of locals at run-time. We used this approach in Gforth before we implemented constant folding.

[15] For a description in English, see news:<2019Aug5.121829@mips.complang.tuwien.ac.at>

[16] The other case (no literal) does not happen when the source code is `to` *name*, but is also handled correctly: it just compiles `to-w:` without optimization and exits `to-w:-opt` without performing the words after `?fold-to`.

4.5 Setting method implementations

By setting the method implementations appropriately, we can define words with capabilities that are not properly supported by the old header format (see Section 5), but how do we set them?

The basic approach is that of an object-oriented system based on prototypes [Bor86] rather than classes. A new word is created by building a new header, copying the methods of an existing one, and then changing individual method implementations.

In practice, the usual approach is to call an existing defining word (which at bottom level works by copying and then changing the behaviour of a pre-existing word, which in turn was created by the cross-compiler), and then modifying the behaviour of the resulting word.

You can change individual method implementations of the most recently defined word with the following words:

setter	stack effect	sets
`set-execute`	`( addr -- )`	code field `compile,`
`set-does>`	`( xt -- )`	code field `does compile,`
`set-optimizer`	`( xt -- )`	`compile,`
`set->int`	`( xt -- )`	`name>interpret`
`set->comp`	`( xt -- )`	`name>compile`
`set-to`	`( xt -- )`	`(to)/defer!`
`set-defer@`	`( xt -- )`	`defer@`
`set->string`	`( xt -- )`	`name>string`
`set->link`	`( xt -- )`	`name>link`

In order to preserve the relation between `execute` and `compile,`, every word that changes what `execute` does also has to change what `compile,` does. `Set-execute` changes the `compile,` implementation to the default code generator for all words. `Set-does>` changes the `compile,` implementation to the default code generator for `create...does>` words. Afterwards, you can change the `compile,` implementation with `set-optimizer` to one that generates faster code. Using `set-optimizer` before `set-execute` or `set-does>` (or `does>`) will not have an effect that survives the `set-execute`/`set-does>`.

If you want to change a method of an older word, you can make the older word take the place of the most recently defined word with

```
make-latest ( nt -- )
```

To avoid confusing mixups of behaviours, the behaviour of a word should not be changed after it has been used; the implementations of the behaviour can still be changed (e.g., for optimization).

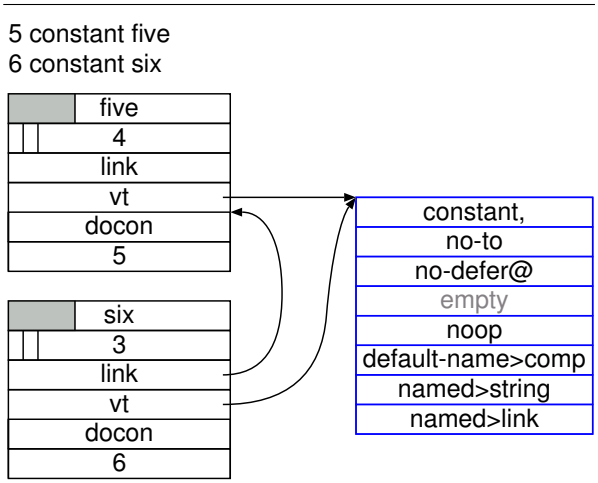


Figure 5: Two words with a shared (deduplicated) virtual table

4.6 Deduplication

Implementing this prototype-based approach is easy if each word has its own vt (virtual table): just copy the vt of the original, and then change it as you please. If you go that way, you can put the virtual table directly into the header instead of in a separate structure.

However, we want to avoid the memory costs of this approach.[17] Therefore we deduplicate the virtual tables: Just before the definition of the next word starts, Gforth checks whether the current word's virtual table is equal to one of the virtual tables that existed before (for words defined earlier). If so, the address of the earlier copy of the virtual table is stored in the current word's vt field, and the current virtual table's memory is reclaimed (see Fig. 5). As a result, we currently only have 117 vts in the Gforth image (for ≈ 4000 words).

You can keep virtual tables in the main dictionary with this scheme (we have done so for a while), but it causes complications in some places. An approach that avoids these complications is to keep the virtual tables in a separate section [Ert16].

Gforth performs deduplication and its reversal (duplication) automatically: If you change a method of a word with a deduplicated vt, Gforth duplicates the vt first. When switching to a new word by defining a new word or with `make-latest`, Gforth deduplicates the vt.

4.7 `Create-from`

Creating a word by starting out with some word, modifying it, and finally deduplicating the vt is

[17] ≈28 000 cells for the Gforth image alone. The memory cost can become a problem even on large machines for code that uses the dictionary for lookup tables, or otherwise creates many words at run-time).

somewhat expensive. This is especially relevant when defining a huge number of words for using a wordlist as a lookup table for data.

A part of that expense could be reduced by using a hash table for deduplication rather than the linear search we use now, but we did not pursue this direction for now. Instead, we introduced

```
create-from ( nt "name" -- )
```

This creates a hidden[18] word with an empty body that has the same method implementations as *nt*. Implementationwise, this means that we just copy the vt address instead of duplicating and later deduplicating the vt.

Gforth uses `create-from` to implement all the common definition words (such as `constant`). But if a word is modified after that (with `immediate`, `does>` or one of the `set-`... words), it incurs the cost of duplication and, later, deduplication.

In combination with `noname`[19], our current implementation still duplicates the vt, changes the `name>string` and `name>link` implementation and eventually deduplicates the vt; we plan to optimize this case in the future.

4.8 Out-of-band data

Gforth keeps some header-related data in other places than the header.

Hash table

Gforth uses a hash table to speed up dictionary searches. This hash table is in `allocate`d memory; it is built on system startup by inserting all the words from the linked-list representation of the wordlists, and is rebuilt in the same way when necessary (e.g., to increase the number of buckets).

The hash table is also needed with the new header design (linear search in a linked list does not become faster with the new header), and our implementation is actually hardly affected by the header change.

Location information

Gforth 1.0 keeps a lot of source code location information around, but it is all out-of-band. In particular, there is an array that contains a location for every dictionary cell. If a dictionary cell is the nt of a word, the corresponding location indicates where the word is defined. If a dictionary cell is some threaded-code cell, the corresponding location indicates the source code for which this threaded-code cell was generated.

[18] You have to `reveal` the word to make it visible.

[19] The next defining word after `noname` produces a nameless word.

4.9 +TO, ADDR

Many Forth systems, including Gforth, also support `+to` *name* (for incrementing *name*) and, in some cases, `addr` *name* (for taking the address of *name*[20]). We will only look at `+to` in the following, but the issue is similar for `addr`.

The most straightforward way to implement `+to` is to have another method `(+to)` and implement `+to` similar to `to`. This leads to similar code at the `+to` definition, but also the various `(+to)` implementations would look very similar to their respective `(to)` implementations; e.g., for `fvalue` they would look as follows:

```
: fvalue-to ( r xt-fvalue -- )
    >body f! ;
opt: drop ]] >body f! [[ ;

: fvalue-+to ( r xt-fvalue -- )
    >body f+! ;
opt: drop ]] >body f+! [[ ;
```

To avoid this code duplication, Gforth employs the following approach: At the start of `TO`/`+TO` text interpretation, the variable `to-style#` is set to indicate which of the two is currently being text-interpreted, then the same code is executed for both words, and in the end, this variable is used to pick the right xt (`f!` or `f+!` in this case) from a table, and then `execute` or `compile,` it. The corresponding code for `fvalue` looks as follows:

```
Create f!-table ' f! , ' f+! ,

: fvalue-to ( r xt-fvalue -- ) \ gforth
    >body f!-table to-!exec ;
opt: drop ]] >body [[ f!-table to-!, ;
```

While using a global variable makes the code smell, variants of this approach have been used in Forth systems for decades, without known problems; in contrast to the `state`-smartness problem [Ert98], the consumer of the value in `to-style#` is not separated from the producer if the `to`-like words parse (as they do in Gforth).[21]

For simplicity of exposition, we ignore this particular twist in the rest of this paper.

[20] `Addr` destroys one of the advantages of value-flavoured words: That they have no aliases and can therefore be allocated to registers, or their accesses reordered wrt memory accesses and accesses to other value-flavoured words. `Addr` exerts this destructive effect already when it *can* be applied to a word, even when it is not actually applied, because there is no guarantee that it will not be applied later. One way to deal with this would be to make it explicit at word definition whether `addr` can be applied to this word, and report an error if `addr` is applied to a word to which it cannot not be applied.

[21] Some other systems have non-parsing variable-setting `to` implementations, and one can produce funny effects with them <news:2017Jan7.150224@mips.complang.tuwien.ac.at>.

5 Examples

This section shows examples of using the `set-...` words to perform things that were much less elegant with the old header. The shown code is usually simplified: It does not show some of the complications for extra features of Gforth that are not the focus of the present work, e.g., `+to`.

5.1 Constant

```
: constant ( x "name" -- )
  create ,
  [’] @ set-does>
  [: >body @ ]] literal [[ ;] set-optimizer
;
```

This first defines `constant` as a `create...does>` word with the `does>` action `@`. The programmer is not allowed to change constants, so our `compile,` implementation (the quotation before `set-optimizer`) compiles the constant's value x as a literal, which has the same effect, but is cheaper than the code that would be produced without the `set-optimizer` part (2 Aarch64 instructions instead of 11).

The actual Gforth implementation of `constant` uses a native-code `docon`. This `docon` has been there since Gforth's inception, and was important for performance before we switched to primitive-centric threaded code (it takes 5 Aarch64 instructions, compared to 10 for the `does>`-based definition above).

5.2 Fvalue

```
: fvalue-to ( r xt-fvalue -- )
    >body f! ;
opt: drop ]] >body f! [[ ;

: fvalue ( r "name" -- ) \ float-ext
  fconstant
  [: >body ]] Literal f@ [[ ;] set-optimizer
  [’] fvalue-to set-to ;
```

`Fconstant` is defined analogously to `constant`; `Fvalue` reuses the `execute/does` part of `fconstant`, but defines a different optimizer that accesses the body at run-time in order to treat changing values correctly; the resulting code performs 3 Aarch64 instructions.

The `to` implementation is set to `fvalue-to`. Its definition and its optimizer are quite straightforward. One non-obvious thing that happens when compiling `to` *name* is that *name*'s xt is compiled as a literal, followed by `>body f!`; constant folding performs the `>body` at compile-time rather than at run-time, so the final code generated when compiling `to` *name* is a literal followed by `f!` (4 Aarch64 instructions, compared to 20 without this optimizer).

5.3 defer

```
: value-to ( x xt -- )
  >body ! ;
opt: ( xt -- ) \ run-time: ( x -- )
  drop ]] >body ! [[ ;

: defer-defer@ ( xt1 -- xt2 )
  >body @ ;
opt: ( xt -- )
  drop ]] >body @ [[ ;

: perform @ execute ;

: defer ( "name" -- )
  create [’] abort ,
  [’] perform set-does>
  [: >body ]] lit-perform [[ , ;]
                          set-optimizer
  [’] value-to set-to
  [’] defer-defer@ set-defer@ ;
```

This example shows setting the `execute/does>`, `compile,`, `to` and `defer@` methods, with `to` and `defer@` having optimizers.

5.4 Default and immediate compilation semantics

```
: default-name>comp ( nt -- xt1 xt2 )
    name>int [’] compile, ;

: imm>comp ( nt -- xt1 xt2 )
    name>int [’] execute ;

: immediate ( -- ) \ core
    [’] imm>comp set->comp ;

: immediate? ( nt -- flag )
    name>compile nip [’] execute = ;
```

The default compilation semantics are to compile the execution semantics, i.e. (in Gforth 1.0) the interpretation semantics. `Default-name>comp` implements the default compilation semantics. Most defining words (e.g., `create` and `:`) use `default-name>comp` as implementation of the `name>compile` method.

`Immediate` changes the compilation semantics to be the same as the execution/interpretation semantics. So this implementation changes the `name>compile` implementation to produce the xt of `execute` (instead of `compile,`) as xt2; so when you `execute` the result, it eventually executes the xt of the word.

`Immediate?` shows how one can determine the immediacy of a word based on `name>compile`. But there is actually rarely a need to check for immediacy. Instead, you can use `name>compile` directly. E.g., a text interpreter in compile state can just perform `name>compile execute` to perform the compilation semantics of a word, without worrying about immediacy. `Postpone` can also be implemented without worrying about immediacy:

```
: postpone ( "name" -- )
  parse-name find-name dup 0= -13 and throw
  name>compile swap ]] literal [[ compile,
 ; immediate
```

5.5 To

`To` is an example of a word that has neither default compilation semantics nor immediate compilation semantics.[22] We first show how to implement `to` directly:

```
: to-int ( v "name" -- )
  parse-name find-name dup 0= -13 and throw
  (to) ;

: to-comp ( compilation: "name" -- )
          ( run-time: v -- )
  parse-name find-name dup 0= -13 and throw
  ]] literal (to) [[ ; immediate

: to-name>comp ( nt -- xt1 xt2 )
  drop ['] to-comp ['] execute ;

synonym to to-int
' to-name>comp set->comp
```

`To-int` has the same interpretation semantics as `to`, and `to-comp` (explained below) the same compilation semantics. We then define `to-name>comp`, which later serves as the `name>compile` implementation for `to`. Finally, `to` is defined as synonym for `to-int`, copying (among others) the interpretation semantics of `to-int`. The compilation semantics is then overwritten with `set->comp`.

Note that `to-comp` resolves the `(to)` method at compile time through the optimizer of `(to)` (not shown here).

5.6 Interpret/compile:

Directly defining a word like `to` is cumbersome, so Gforth has two more convenient ways to define them. First, there is `interpret/compile:`, which has been in Gforth since 1996. You can use it to define `to` as follows:

```
' to-int ' to-comp interpret/compile: to
```

The interface `interpret/compile:` has been designed for the previous header implementation, but can be implemented with the new header implementation relatively straightforwardly:

```
: i/c>comp ( nt -- xt1 xt2 )
    >body cell+ @ ['] execute ;

: interpret/compile: ( i-xt c-xt "name" --)
    defer , lastxt defer!
    ['] defer-defer@ set->int
    ['] i/c>comp set->comp
    ['] no-to set-to
    ['] no-defer@ set-defer@ ;
```

The interpretive behaviour of *name* is like for a `defer`red word, so we implement `interpret/compile:` as inheriting from `defer`. It stores *i-xt* to the first cell of the body with `lastxt defer!`. The `name>interpret` implementation fetches *i-xt* by reusing `defer-defer@`; note that unlike most words, for *name* xt≠nt, so `name>interpret` is not a noop here.

In addition, it stores the *c-xt* in the second cell of the body (with `,`). The compilation semantics is to `execute` *c-xt*, and the `name>compile` implementation `i/c>comp` implements this behaviour.

We inherit from `defer`, so we inherit the `(to)` and `defer@` implementations for `defer` (i.e., `is`, `defer!`, `action-of` and `defer@` would work). We do not want that, so we override these methods with `no-to` and `no-defer@`, which report an error if these words are used on *name*. Note that the interpretation semantics explicitly uses `defer-defer@` rather than the generic `defer@`, because the latter no longer works after the `set-defer@`.

`Interpret/compile:` has been criticized on aesthetic grounds, so Gforth also has code for `compsem:`, which is used like `opt:`. The implementation of `compsem:` is short[23], but would require explaining Gforth features beyond the scope of this paper, so we skip it here.

[22] Standard programs must not tick or `postpone` `to`, making an immediate `state`-smart implementation possible. Other standard words (e.g., `s"`) do not have this restriction. We demonstrate a `to` that works even without this restriction, as an example for this whole class of words, because it also shows the usage of `(to)`.

[23] http://git.savannah.gnu.org/cgit/gforth.git/tree/set-compsem.fs

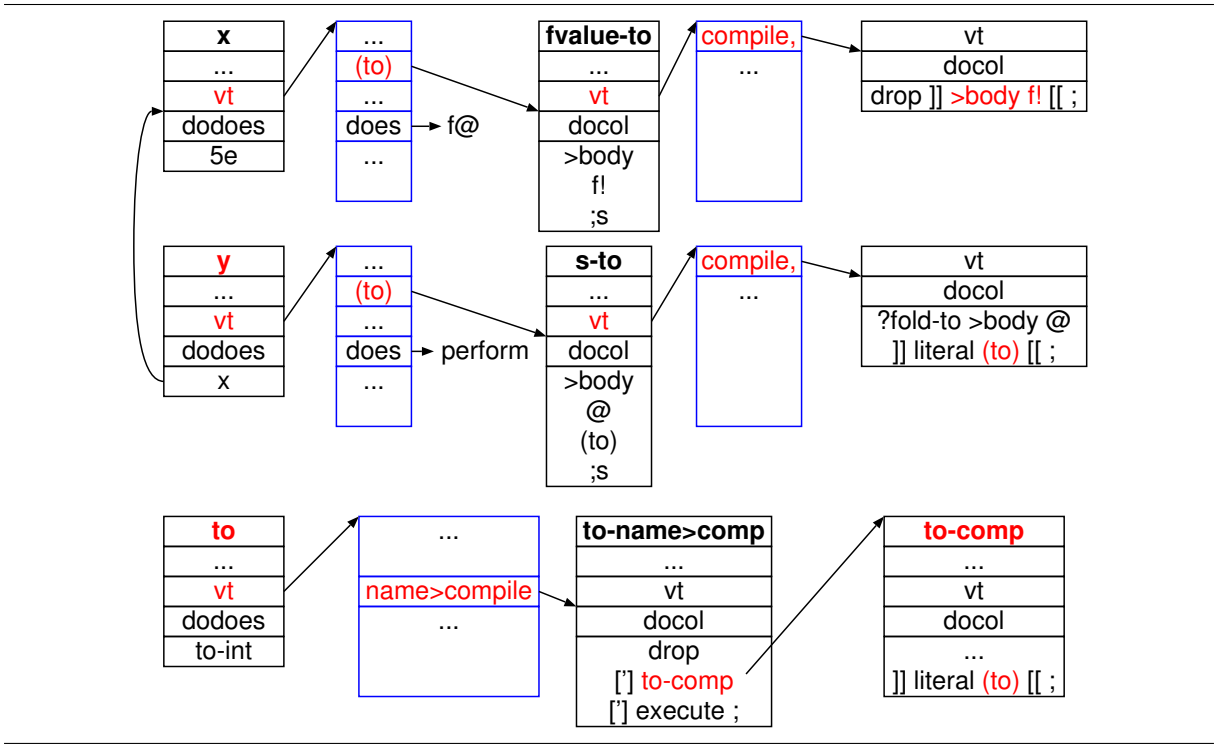


Figure 6: Data structures for the example

5.7 Synonym

```
: s-to ( val nt -- )
  >body @ (to) ;
opt: ( xt -- )
  ?fold-to >body @ ]] literal (to) [[ ;

: s-defer@ ( xt1 -- xt2 )
  >body @ defer@ ;
opt: ( xt -- )
   ?fold-to >body @ ]] literal defer@ [[ ;

: synonym ( "name" "oldname" -- )
  defer
  parse-name find-name dup 0= -13 and throw
  dup lastxt defer!
  compile-only? if compile-only then
  [: >body @ compile,     ;] set-optimizer
  [: >body @ name>interpret ;] set->int
  [: >body @ name>compile   ;] set->comp
  ['] s-to set-to
  ['] s-defer@ set-defer@ ;
```

`Synonym` stores the nt of *oldname* in the body of *name*. The normal way to deal with *name* is through words that use the nt. `name>interpret` gets the nt of *oldname* and gets its xt; likewise for `name>compile`. Text interpretation, ticking and `postpone` work through these words.

The implementation of `(to)` also gets the nt of *oldname* and performs its `(to)` implementation; if *oldname* has `to` ***oldname*** semantics, they will be performed; if not, *oldname*'s `(to)` implementation reports an error. The `(to)` implementation also has an optimizer that computes *oldname*'s nt and compiles its `(to)` implementation (which in turn triggers *oldname*'s `(to)` optimizer). The implementation of `defer@` is analogous.

Note that these things work for arbitrarily deep chains of synonyms, always delegating to implementation of the same method at the next level.

In order to make *name*'s nt also work as xt, `synonym` inherits from `defer`. So, if you `execute` the nt of *name*, this will `execute` the nt of *oldname*; eventually a word with nt=xt is performed. Synonym also has an optimizer for `compile,`ing the nt, which optimizes away the indirection.

5.8 Example

This example presents these words in action. Figure 6 shows the data structures involved in the example.

```
5e fvalue x
synonym y x
: foo to y ;
```

In this code, `to y` performs the compilation semantics of `to` by performing the `name>compile` method and its implementation `to-name>comp`. As a result, the text interpreter `executes to-comp`, which first parses `y` and produces its nt, and then compiles the nt as a literal followed by `(to)`. `(to)`

has an optimizer that takes the preceding literal (`y`'s nt) into account and then `compile,`s `s-to`. This calls the optimizer of `s-to`, which in this case consumes the preceding literal (still `y`'s nt), then fetches the body of `y` (giving `x`'s nt), and compiles that as literal, followed by `(to)`. Again, `(to)`'s optimizer resolves this, this time to `compile,`ing `fvalue-to`. This calls the optimizer of `fvalue-to`, which compiles `>body f!`. `>Body`'s optimizer consumes the preceding literal (`x`'s nt), and compiles `x`'s body as literal. The following `f!` is compiled as-is. As a result, the code for `foo` is the same as produced by directly writing

```
: foo [ ' x >body ] literal f! ;
```

6 Empirical Results

The goal of this work has been design cleanliness and flexibility, not performance. But here we demonstrate that performance does not suffer (and actually benefits a little).

Unfortunately, since the work on the new headers began in 2012, a lot of other changes have been made in Gforth (> 3000 commits), and the header-related changes cannot be isolated with reasonable effort.

Nevertheless, we take various measures to isolate the performance effects: We compiled the last old-header Gforth[24] and a recent new-header Gforth[25] with gcc-4.7.2, a compiler version where both the old and the new version use all the performance features of Gforth. We use `--ss-number=14` to ensure that both versions use almost the same static superinstructions. And finally, instead of measuring a task such as compilation of a complete file, where other features (in particular, IDE features like `locate` and `where`) impact the performance, we look at individual operations.

In the following results, we do not use the simplified implementations shown above, but the more complex implementations in Gforth.

All results are in cycles of user time on a Core i7-4790K (Haswell).

6.1 `Find-name`

Gforth uses an out-of-band hash table that does not use the link field of the header during `find-name`, so the new header structure should have little effect on `find-name` performance. Nevertheless, the text interpreter calls `find-name` once for every word it interprets or compiles, so we present performance numbers here. The following table shows the average run-time of one invocation of `find-name` on one word out of a set of 1602 words (all present in the search order):

cycles	old	new
`find-name`	3184	3855

Apparently the other changes between these version have slowed down `find-name` a little.

6.2 `Name>interpret` and `name>compile`

When text interpreting, the nt produced by `find-name` is then usually processed either by `name>interpret` (when interpreting) or by `name>compile` (when compiling). For the old header, normal words, aliases[26], and `interpret/compile:` words are treated in different paths of a cascade of `if`s spread across a number of words. For the new header, different method implementations are called. So for different word types, the performance may be different; we therefore measure the performance of `name>interpret` and `name>compile` for different word types:

cycles	old	new
`name>interpret` normal	123.8	15.0
`name>interpret` alias	123.3	24.6
`name>interpret` i/c	124.2	24.7
`name>compile` normal	136.4	32.1
`name>compile` alias	138.8	47.4
`name>compile` i/c	143.2	30.1

This part of text interpretation is quite a bit faster with the new header thanks to not having to perform a lot of tests, but in the overall scheme of things, this speedup vanishes in the noise.

6.3 `Compile,`

When compiling, Gforth eventually `compile,`s some xt (e.g., because `name>compile` produced the xt of `compile,` as xt2, and the text interpreter `execute`s that), and that `compile,` eventually compiles a primitive. The path from entering `compile,` to compiling a primitive has changed a lot: With the old header, `compile,` uses an `if`-cascade to decide the word type and what code to generate; the new header uses method dispatch instead. Depending on where in the `if`-cascade a word type is, the performance can vary; in particular, `does>`-defined and `value`-defined words are early in the `if`-cascade; by contrast, the old `compile,` recognizes primitives by excluding all other word types, so the old `compile,` performs the the longest chain of `if`s for them.

In the following, we present the run-time of the whole `compile,`, as well as a number *without code generation*. The latter number does not include the actual code generation of the primitive, nor (in the

[24] commit 617d4a8deccf5f4eefeb236f972171d6f65bb685
[25] commit 045ff553a7c6304015be66533ead115c93866882
[26] Aliases are similar to synonyms; synonyms are not supported in the old header.

new system only) recording which threaded-code location corresponds to which source code location; it does include the time spent on laying down the inline parameters of the primitives that execute the does and value words.

cycles	old	new
`compile,` does	1074	1156
without code generation	153	51
`compile,` value	963	1103
without code generation	151	49
`compile,` primitive	1013	985
without code generation	235	12

In the *without code generation* lines, we can see that the overhead of selecting what to compile has become much smaller. However, that overhead is just a small part of the costs of `compile,`, and the additional work of recording the source/threaded-code correspondence has more than made up for these savings.

6.4 `>Name`

Sometimes it is useful to get back from that xt of a word to the nt; e.g., the decompiler does it in order to print the name of a word. Gforth has a word `>name ( xt -- nt|0 )` that does this or returns 0 if its argument is not an xt. With the new header, the conversion from xt to nt is a noop (but if you want to make it explicit, you can use `xt>name`). But `>name` also does the checking, and Gforth 1.0 uses relatively reliable, but costly heuristics to do that. Gforth 0.7 uses heuristics for both conversion and checking.

cycles	old	new
`' create >name`	1281	2280
`' noop >name`	51971	2314

For the old scheme, `>name` is particularly expensive for primitives like `noop`, while for normal words like `create`, it is not so extreme. For the new headers, the heuristics for non-primitives are more expensive than for the old headers. But these costs have not been a performance bottleneck yet, and we can think of some ways to improve these costs if they ever prove to be a problem.

6.5 Header creation

As mentioned in Section 4.7, creating words is more expensive in Gforth 1.0 if the overhead of duplication and deduplication is incurred, but is fast if that is avoided with `create-from`. We measured this with a benchmark that defines 1,000,000 constants in a wordlist, using different implementations of `constant`. The numbers reported are cycles per created word; for the *new* results, we use snapshot `gforth-0.7.9_20190829` compiled with `gcc-4.9.2`.

		new	
cycles	old	deduplicated	`create-from`
`constant`	2168	9477	1742

This performance difference is not just relevant for applications that use wordlists as a data structure. Loading programs is also affected: We saw a speedup by 30% when loading the OpenGL and Xlib libraries.

7 Related work

In the beginning, Forth stored the length and 3 letters of the name [Moo74]. Fig-Forth supported full-length names, optionally shortened to the length stored in `width` [Tin81]. As a result, it was expensive to get from the name field to the link field (as required repeatedly during name search) and other fields. Smith proposed [Smi80] to reduce this cost by moving the name characters before the count byte, with the link field still pointing to the count byte; In Smith's scheme, the name characters are stored in reverse order, because thanks to `width` they system may store only a part of the name, and that part should be the start of the name. Our new header is similar in putting the variable-length name string before the rest of the header, at a negative offset from the nt address that we use for header accesses; however, we store the name characters in conventional order, because we can use the count to get at the start of the name.

Shaw [Sha88] puts multiple code fields in headers. In addition to the ordinary code field, a word can have a `to` code field, which corresponds to our `(to)` method, but uses a different execution mechanism. Shaw also uses the multiple code field mechanism to get rid of `state`-smart words, by having an optional code field for the compilation semantics (the ordinary code field implements interpretation semantics); it uses the same code field for that as for `to`, using flags to decide if that code field is used for compilation semantics. While there are significant differences from the new Gforth header, Shaw's work is conceptually still closer than all others.

CmForth[27] by Charles Moore and Pygmy[28] by Frank Sergeant implement dual-semantics words by putting a word for the interpretation semantics in the `FORTH` vocabulary and a word for the compilation semantics in the `COMPILER` vocabulary, with the text interpreter searching the appropriate vocabulary for the current use. Mark Humphries also implements dual-semantics words with multiple headers, but he puts them in the same wordlist, with flags that indicate whether the word should be found when looking for a certain semantics.

[27] https://raw.githubusercontent.com/ForthHub/cmFORTH/combined/cmforth.fth

[28] http://pygmy.utoh.org/pygmyforthmanual.html#h1id47

MPE's VFX Forth has `set-compiler` that works like Gforth's `set-optimizer`: It changes what `compile,` does for the preceding word [MPE16, Chapter 19.7.3]. This suggests that VFX implements the intelligent `compile,` in a similar way as the new Gforth header, but to the best of our knowledge this has not been published.

In 2004, Ertl sketched[29] a header structure with an additional field for the intelligent `compile,`, and, for named words, an xt2 field for implementing compilation semantics: for normal words, the xt2 field would contain the xt of `compile,`; for immediate words, the xt of `execute`; and for other words, it would contain something else. However, it requires some additional complexity to implement `find` in a way that supports user-defined text interpreters.

The new Gforth header differs from the 2004 header ideas in that it implements compilation semantics by defining what `name>compile` does for a word. This means that unlike in the 2004 ideas, the xt1 of a compilation token can be different from the xt representing the interpretation semantics, thus making it simpler to implement `find`. In addition, the new Gforth header allows changing what several other words do for the present header, which supports defining `value`-like words, `synonym`s, etc. And it stores the xts for all these method implementations in a separate structure (vt) that is deduplicated.

8 Conclusion

Gforth's old header (based on the original Forth header) leads to complex and inflexible implementations of words like `compile,`, `name>interpret`, `name>compile,`, and `to`; it supports dual-semantics words through an ugly hack; we did not even implement `synonym`, because that would have required adding more special cases.

By contrast, the new Gforth header has a prototype-based object-oriented design that allows extending the behaviour of words like `compile,`, `name>interpret`, `name>compile`, and `to` for individual words. This flexibility makes it relatively easy and compact to implement, e.g., `synonym` such that the created synonyms also work with `to` if the original worked with `to`.

Acknowledgments

We thank the reviewers for their comments, which helped to improve the paper.

[29] https://www.complang.tuwien.ac.at/forth/header-ideas.html

References

[Bor86] Alan Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986. 4.5

[Bro84] Leo Brodie. *Thinking Forth*. Fig Leaf Press (Forth Interest Group), 100 Dolores St, Suite 183, Carmel, CA 93923, USA, 1984. 1

[EP18] M. Anton Ertl and Bernd Paysan. Closures — the Forth way. In *34th EuroForth Conference*, pages 17–30, 2018. 4.1

[Ert98] M. Anton Ertl. `State`-smartness — why it is evil and how to exorcise it. In *EuroForth'98 Conference Proceedings*, Schloß Dagstuhl, 1998. 2, 4.9

[Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002. 2, 6, 4.2

[Ert16] M. Anton Ertl. Sections. In *32nd EuroForth Conference*, pages 55–57, 2016. 4.6

[Koc15] Matthias Koch. Flags, Konstantenfaltung und Optimierungen. *Vierte Dimension*, 31(arm):16–18, 2015. 4.4

[Moo74] Charles H. Moore. Forth: A new way to program a mini-computer. *Astron. Astrophys. Suppl.*, 15:497–511, 1974. 1, 4.2, 7

[MPE16] Microprocessor Engineering. *VFX Forth for x86/x86 64 Linux*, 4.72 edition, 2016. 7

[Pay19] Bernd Paysan. Constant Folding für Gforth. *Vierte Dimension*, 35(2):17, 2019. 4.4

[Pel17] Stephen Pelc. Special words in Forth. In *33rd EuroForth Conference*, pages 37–45, 2017. 2

[Sha88] George W. Shaw. Forth shifts gears. *Computer Language*, pages 67–75 (May), 61–65 (June), 1988. 7

[Smi80] Robert L. Smith. A modest proposal for dictionary headers. *Forth Dimensions*, I(5):49, 1980. 7

[Tin81] C. H. Ting. *Systems Guide to fig-Forth*. Offete Enterprises, Inc., San Mateo, CA 94402, 1981. 7

simple-tester, a testing tool for embedded Forth systems

Ulrich Hoffmann and Andrew Read

EuroForth 2019

Introduction

simple-tester is a very lighweight testing tool designed to assist the devlopment of a Forth system on embedded target. simple-tester's is inspired by the ANS Forth test harness [1]. One innovation is the use of hashing rather than memory to compare actual and expected results.

The need for a simple testing tool

There is a chicken and egg situation with any testing tool that is implemented within the system it is designed to test:

1. if there are bugs in the system then the testing tool itself may not function
2. if the testing tool is complex then it cannot be implemented until the system has largely been completed, so the testing tool is not avilable during the development phase
3. if bugs in the system happen disable charater I/O, then the testing tool will not be able to communicate test diagnosis

For these and other reasons we believe Test Driven Development [2] has not typically been applied to the development of Forth implementations on embedded systems, while the ANS Forth test harness is mainly used for verificaton at the final stage.

The goal of simple-tester is to:

1. allow unit testing on embedded targets with limited resources and
2. allow testing as early on as possible in the lifecycle of new forth systems, even before the system knows how to compile new colon word defintions

Test Driven Development of a Forth system

We present an illustration from seedForth [3] [4], which is an approach to developing an embedded Forth system without a cross-compiler. (Roughly speaking, a tokenizer running on the host compiles source code to a token file which is processed on the target.) In the example below we bring up a series of elementary code words in seedForth and test them.

```
Tstart
  CODE: drop
  T{ 1 2 drop }T 1 ==

  CODE: dup
  T{ 1 dup }T 1 1 ==

  CODE: swap
  T{ 2 1 swap }T 1 2 ==
Tend
```

`CODE:` is part of seedForth, not simple-tester. (Briefly put, it simply "activates" a code word that has been implemented in assembly language on the target.)

`Tstart` is part of simple-tester and we assume that it, like the rest of simple-tester, is already implemented as a code word on the target. `Tstart` initiates testing.

`T{` designates the start of a test. The syntax is the same as the ANS Forth test harness.

`}T` designates the end of the section of code being tested. After `}T` there follows the expected results. This is different syntax to the ANS test harness.

`==` compares the actual results with the expects results and takes action if they do not match. This is new syntax from the ANS test harness, but arguably the post-fix comparison is more Forth-like. (And also avoids the `->` operator that is used for assignements in VFX.)

`Tend` concludes the series of tests.

simple-tester's communication protocol

simple-tester communicates entirely through a single numeric output device. This could be a line of micro LEDs (presenting numbers in binary), a seven-segment hexadecimal display, bytes over a serial line, or some other mechanism.

At the inception of testing `Tstart` sets the internal test counter to zero. At the start of each test, `T{` increments the internal test counter and reports the test number on the numberic output device. At the

conclusion of each test, `==` acts as follows: if the actual results match the expected results, then do nothing. Otherwise halt the system whilst leaving the current test number visible on the numeric output device.

Assuming that all tests conclude sucessfully, then `Tend` displays some magic number, typically `FFFF`, to indicate successful conclusion. On the other hand if any test has failed then `Tend` will not be reached and the sequential number of the failing test will remain on the numeric output device.

This testing protocol was chosen for the following reasons:

1. we leverage hardware on the embedded system for output rather than rely on high-level Forth words such as "dot". Outputting a number to a line of LED's or a seven-segment display can often be accomplished with a single store instruction
2. implementing this communication protocol is very simple and requires minimal code
3. the test number is displayed before the code under test is executed. If the execution of the code causes a system-failure, then the identity of that test will already have been reported
4. assuming that all tests have completed sucessfully, then `FFFF` on the numeric output device is quickly and conveniently noted

This communication protocol is more limited than that of the ANS test harness: the reason for a test failure (different stack count or different actual results) is not reported, and only the first failing test is identified, not the full set. Nevertheless we consider that the advantages listed above are compelling in the situations where we envisage using simple-tester.

Implementing simple-tester

We highlight the key aspects of implementation before walking through the reference implementation in the next setion.

simple-tester is implemented as code words (likely in target system assembly language), rather than as colon definitons. Ideally these code-words should be implemented at a very early stage so that they can be employed to test further code words as they are developed.

We note that the ANS test harness stores the actual results of each test in memory prior to comparison with the expected results. We consider this approach to be less suitable for embedded systems since RAM may be limited. Instead we use a simple hash algorithm to hash both the actual and expected results and compare only the hash totals. Using this approch, our reference implementation requires only two cells of RAM storage.

We recognize that a hash approach may lead to false test passes where the actual and expected results are different but where there is a collision between the hash totals. We don't consider this weakness to be fatal - few testing approaches provide complete coverage of all possible cases, and so a judgement must always be made as to what level of testing coverage is sufficent to provide the required level of assurance.

We do take the precation of using a hash algorithm that is non-symmetric: if the actual and expected test results are the same values but in reversed order, then the test will fail. Of course a more sophisticated hash algorthim than the one we have chosen may be implemented if hash collisions are anticipated to be a problem in any particular situation.

Reference implementation

Although simple-tester is anticipated to be implemented as code words rather than in colon definitions, the reference implementation is given in Forth for ease of communication.

```
\ utility words
\ report the test number to a numeric output device
: T.
    .       \ for gforth testing
;

\ halt the system
: halt
    quit    \ for gforth testing
;

\ compute h1 by hashing x1 and h0
: hash ( x1 h0 -- h1 )
    swap 1+ xor
;

\ hash n items from the stack and return the hash code
: hash-n ( x1 x2 ... xn n -- h )
    0 >R
    BEGIN
        dup 0 >
    WHILE
        swap R> hash >R
        1-
    REPEAT
    drop R>
;

variable Tcount     \ the current test number
variable Tdepth     \ saved stack depth

\ start testing
: Tstart
    0 Tcount !
;
```

```
\ start a unit test
: T{ ( -- )
    Tcount @ 1+ dup T. Tcount !
    depth Tdepth !
;

\ finish a unit test,
: }T ( y1 y2 ... yn -- hy )
    depth Tdepth @ -    ( y1 y2 ... yn Ny )
    hash-n              ( hy )
    depth Tdepth !      ( hy )
;

\ compare actual output with expected output
: == ( hy x1 x2 ... xn -- )
    depth Tdepth @ -    ( hy x1 x2 .. xn Nx )
    hash-n              ( hy hx )
    = 0= IF halt THEN
;

\ signal end of testing
: Tend  ( -- )
    65535 ( 0xFFFF) T.
;
```

Potential applications

simple-tester has been designed first and foremost with the goal of supporting Test Driven Development of Forth systems on embedded targets. simple-tester might also be useful in other situations, for examlple:

1. Test Driven Development of applications on embedded Forth systems. Our experience is that the ANS test harness is not commonly used on embedded systems because of its size and complexity. Where simple-tester is already built in to a new embedded Forth system, then there is no reason why it cannot also be employed application testing

2. power on self testing (POST). For example, in seedForth the Forth system is recompiled from a token file at each power on, and since unit tests are interleaved with the Forth system definition, the system is freshly tested at every restart. This helps ensure that that the system can never be modified without subsequent regression testing, and also verify any hardware upon which the system is reliant

3. as an alternative to the ANS test harness for small application development. Since the reference implemention of simple-tester is provided in Forth it could also be used for application testing on desktops. There may be occasions where the simpler and lighter simple-tester, although more limited in scope than the ANS test harness, might be easier to handle

Conclusion

We have developed a very lighweight tool for supporting Test Driven Development, with a particluar focus on Forth systems in embedded targets. simple-tester is open source and available on GitHub [5]. We welcome correspondence.

Ulrich Hoffmann (FH Wedel University of Applied Sciences), uh@fh-wedel.de

Andrew Read, andrew81244@outlook.com

References

[1] http://www.forth200x.org/documents/html/testsuite.html
[2] https://en.wikipedia.org/wiki/Test-driven_development [3]
http://www.complang.tuwien.ac.at/anton/euroforth/ef18/papers/hoffmann.pdf
[4] https://github.com/uho/preForth
[5] https://github.com/Anding/simple-tester

EuroForth 2019
Internationalisation - a new approach in Forth

Abstract

The unique capabilities of Forth are harnessed in a technique that greatly improves the efficiency of software internationalisation.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

In software, the word "internationalisation" means writing a program in such a way as to separate the function of the program from its country or language specific appearance. The goal is to make it possible to produce either, different language versions of the program, or in our case, to produce a program that can be dynamically switched between different languages. We need this because the normal user of the program might require a foreign language, and our own support engineer might require English. There are cultural aspects to internationalisation (for example the formatting of date, time and currency), but the main part of the work concerns text.

The word "localisation" refers to the process of adding a new language. The aim of the exercise is to make localisation as easy and foolproof as possible.

We have been doing this for many years now, but recently a new technique using the unique capabilities of Forth, has greatly improved the efficiency of the process.

2. The old method reviewed

The previous technique, first described in Euroforth in 1995, was based around the word:

: P" \ Comp: "text" -- ; Run: -- u

The run time action was easy to describe - it simply returned a number that referred to the phrase that in English was "text".

The compilation action was rather more complex.

First, it checked to see if "text" already existed in the set of phrases for English. If so, it just compiled the number of that phrase as a constant. If the phrase did not exist, then it added "text" to the table of English phrases. Then it compiled the new phrase number as a constant.

At the end of the compilation process, the table of phrases was saved to a text file in a directory called "English". (Originally, phrases were saved in a set of Forth "screens" in non-volatile memory!) There was one line in the file for each phrase.

The process of localisation was then to create a matching phrases file, in another directory named after the alternative language. A text editor was selected which both showed line numbers (equal to the phrase number) and allowed two files to be displayed side by side. This made it relatively easy for the translator.

When the user selected a different language, the alternative phrase file was loaded into memory, in a "bi-string" format with both a leading byte count and a double zero terminator. The language text could then be accessed by either a word CTEXT, that returned the address of the counted string, or a word ZTEXT, that returned the address of the zero terminated string. All window text, including menus, were then updated with the new text. Whenever a new dialog was created, during its initialisation, the text was set for each control that required it.

3. Improvements to the underlying system

Over the years, many improvements were made to the P" based system.

a) The storage of phrases was moved to a database table.
b) In addition to standard text phrases, classes of diagnostic phrases and sequences phrases, customised to the machine being controlled, were introduced.
c) When a new phrase was encountered, all supported languages were also updated, using the "text" phrase preceded by the phase number. This made it easy to identify untranslated phrases.
d) A Translators Aid dialog was developed, which isolated the translator from the method of storage, and prevented him from introducing positional errors.
e) English ceased to be special. New "text" was placed into a base language. This meant that text could be altered in English as well as all other languages, without recompilation.
f) Support for counted strings was dropped.
g) P" returned a pointer to the translated string, instead of a phrase number.

4. Limitations of the old method

a) The main limitation of the old method was that all textual objects needed to have their text set, either initially and on change of language, for main windows and menus; or on initialisation, for dialog boxes. In a large program, this accounted for thousands of lines of code.

b) A further limitation was that the original code was devised for ASCII based text. It was extended for "Shift-JIS" for Japanese, and "GB 18030" for simplified Chinese. However, it was very difficult to apply in many other languages with non-standard characters.

5. Big breakthrough no. 1 - dropping Windows!

Several years ago, we came to the conclusion that Windows was no longer a suitable operating system for use with machine automation. It became company policy that all new programs would be written for Linux. From the point of view of internationalisation, the most important improvement here was moving to a native UTF-8 encoding, instead of having to use the "adapted-ASCII" APIs that had been in use with Windows. No more code pages! This dealt with limitation 4b).

6. Big breakthrough no. 2 - GtkBuilder

When we adopted Linux, we also decided to move to GTK+ for our graphical user interfaces.

In my presentation at Euroforth 2017, I described how a control element named in the GTK graphical design program "Glade", could appear automatically as a Forth VALUE, *without any other coding!* This discovery, only possible in Forth, enabled hundreds of lines of code to be removed from a typical program.

It was only later that we realised that a very similar technique could be used to handle text internationalisation, *without any application specific coding*. This would therefore deal with limitation 4a) and remove many hundreds more lines of code.

7. Review of the GtkBuilder process

In order to appreciate the automated internationalisation technique, it is necessary to understand a little bit about how GTK works.

a) The windows, dialog boxes, menus, and all other graphical elements, are designed using a visual tool called "Glade". The information is saved in XML format.

b) Each component is referred to by a name. Components such as labels, titles and menu entries are also given initial text during the design process.

b) The application loads the XML files using functions of an object called GtkBuilder. During the loading process, each component is created and receives a reference number, similar to a "handle" in Windows.

c) When the application uses a graphical component, for example, to set the text of a label, it must use the reference number to identify the component.

d) Any time after loading, a list of all components can be accessed using a GtkBuilder function.

e) We read in XML files not just during execution, but also during compilation, at an early stage. We go through the list and create a Forth VALUE for every component, with the same name as was used in the design tool, and which returns the reference number. This means that during the rest of the Forth compilation process, all components can be referred to by their Glade / Forth VALUE names.

f) Note one big difference from Windows. In Windows, dialog boxes are created and destroyed as necessary. When using GtkBuilder, everything is created when the XML files are loaded, and items such as dialog boxes are shown and hidden, not created and destroyed. Therefore, all graphical components are always accessible.

8. Automating the internationalisation process

When an application is initialising, before any graphical elements have been displayed, but after we have loaded our language phrase table from the database, we need to set the correct phrase into each textual component. We call a word SETPHRASES - simplified listing below:

```
: SETPHRASES { | pslist pobject -- } \ Set language phrases
  PBUILDER gtk_builder_get_objects -> pslist         \ Make list of objects
  pslist g_slist_length 0 ?DO                        \ For all objects
    pslist I g_slist_nth_data -> pobject             \ Get data
    pobject gtk_label_get_type
    g_type_check_instance_is_a IF                    \ It's a label
      pobject SET-LABELPHRASE
    THEN
..... Other object types .....
  LOOP
  pslist g_slist_free                                \ Free list
;
```

This gets the list of objects and checks each one to see if it's a type of object that needs some text (e.g. a label). If so, it calls the function to set text for that object type.

```
: SET-LABELPHRASE { plabel | plabeltxt -- } \ Set language phrase for label
  plabel gtk_label_get_label -> plabeltxt          \ Get text, incl. markup
  plabel plabeltxt SET-WIDGETPHRASE -> plabeltxt   \ Associate with a phrase
  plabel plabeltxt gtk_label_set_label             \ Set language text
;
```

The get_ and set_ functions in GTK are different for each object type, so a different SET-xxxPHRASE word is needed. But all object types use the same base object, so a common word can be used.

```
: SET-WIDGETPHRASE { pwidget ptext | pphrase -- ptext' } \ Widget to phrase
  ptext IF                                    \ Text defined
    ptext C@ TRANSCHAR = ptext                \ Marked for translation
    1+ C@ 0<> AND IF                          \ Not null
      1+ PHRASE? -> pphrase                   \ Get phrase number
      pphrase -1 = IF                         \ Phrase does not already exist
        ptext 1+ ADDPHRASE -> pphrase         \ Add to database
      THEN
      pwidget pphrase SETPHRASE               \ Associate phrase with widget
      pphrase LANGPHRASE                      \ Return language phrase
    ELSE                                      \ This text not to be translated
      ptext                                   \ Return unchanged
    THEN
  ELSE
    ZNULL                                     \ Return a null string
  THEN
;
```

Not all text object should be translated - some might contain symbols. So we mark text that needs translating with a preceding character - usually a hash sign.

Other words such as PHRASE? , ADDPHRASE and LANGPHRASE already exist from our implementation of P".

The interesting bit is SETPHRASE.

```
: SETPHRASE { phandle pphrase -- } \ Associate phrase number with widget
  phandle Z" Phrase" pphrase g_object_set_data
;
```

Because every component in GTK is an object, and all objects can carry a set of key-to-pointer associations, we simply create a key called "Phrase" and link it to the phrase number, which is pretending to be a pointer.

Now we can look at what happens when a user selects a different language. First, we load the set of phrases for the new language, then we call CHANGEPHRASES, which is very similar to SETPHRASES as shown above, except that for each object type, it calls a change function, instead of a set function, for example:

```
: CHANGE-LABELPHRASE { plabel -- } \ Change language phrase for label
  plabel GETPHRASE ?DUP IF                      \ A phrase number defined
    plabel SWAP LANGPHRASE gtk_label_set_label \ Set text in current lang
  THEN
;
```

GETPHRASE simply looks up the phrase number using the key name of the object.

```
: GETPHRASE ( handle---phrase ) \ Get phrase number of widget
  Z" Phrase" g_object_get_data
;
```

Note that none of the above is application specific. Any program that uses GtkBuilder and Forth can be internationalised, simply by including the appropriate files, and adding just just half a dozen words to the application code.

9. Other internationalisation issues

Besides the purely textual issues, we have to resolve some cultural issues, such as formatting of dates.

For example, if we do an SQL query with a date in the result, we need to format that date in a language specific way, before displaying it in, say, a GtkLabel.

The first thing that is needed, is to set the program locale on change of language.

```
2 0 CallProc: on_French_clicked { pbutton puser -- } \ Button "French" click
  ITASK                                  \ Init every thread or callback
  Z" French" CHANGE-LANG                 \ Set all GTK text
  LC_ALL Z" fr_FR.utf8" SetLocale DROP   \ Set program locale
;
```

Now take the date from the SQL result set. First, place it into a Linux "broken down" time structure. Then call the Linux date / time formatting function.

```
: SQLDATE>LOCALE { zsqldate | tm[ tm ] -- z$ } \ Localise SQL date
  zsqldate tm[ SQLDATE>TM          \ Put date into tm structure
  PAD 50 Z" %x" tm[ strftime DROP  \ Format onto PAD
  PAD
;
```

10. An unresolved problem - GtkCalendar

I've just put this in to prove that Forth programmers can still read C code, if they really have to!

There is no easy way to make the GtkCalendar control change language programmatically, because of the way the widget has been written.

It's not a compound widget - the whole calendar is drawn as a single element.
It is locale sensitive, but unfortunately the day and month names for the locale are obtained from the operating system during gtk_calendar_init , which is called only when the first instance of GtkCalendar is first created.

This means that ***any*** calendar control always displays the day and month names that correspond to the locale that was current when the ***first*** calendar control was created. They can't be changed later, even if all calendar controls have been destroyed and recreated.

Possible solutions:
a) Register this as a GTK bug and see if we can persuade the developers to fix it.
I am hoping that by including this note in a conference paper, this might nudge them!
b) Rebuild GTK, fixing the issue ourselves. I can easily fix the code, but we have no experience in the recompilation.
c) Make our own calendar control.
d) Accept the limitation. Remember the locale in use, and save it when closing the application. Then re-select the locale, before doing gtk_init.

In the GTK source code, the problem could be fixed by moving the lookup of
default_abbreviated_dayname[i] and default_monthname[i]
from get_calendar_init to gtk_calendar_draw.

11. Some other enhancements

Several other nice labour-saving things can be done by adding just a few words to the basic SETPHRASES function, for example:

a) If the object is a button, then an icon can be automatically added to the button, according to the base phrase.

```
: SET-BUTTONIMAGE { pbutton | plabeltxt pimage pfilename -- } \ Set image
  pbutton gtk_button_get_label -> plabeltxt    \ Get text of button label
  plabeltxt IF                                 \ Any text set
    plabeltxt C@ IF                            \ A non null string
      ICONSDIR plabeltxt Z+ Z" .png" Z+        \ Construct path to png
      -> pfilename
      pfilename ZCOUNT FILEEXIST? IF           \ There is a matching png
        pfilename gtk_image_new_from_file      \ Create an image
        -> pimage
        pbutton pimage gtk_button_set_image    \ Put image into button
        pbutton TRUE                           \ Always show
        gtk_button_set_always_show_image
      THEN
    THEN
  THEN
;
```

Then, to make *every* button in the program that has, for example, the base phrase "Cancel", display the same cancel icon, all one has to do is place a file "Cancel.png" in the icons directory.

b) If the object is a dialog box, we can automatically make it transient to the main window. This is not possible in Glade, if the main window is defined in a different XML file from the dialog.

```
: SET-DIALOGTRANSIENT { pdialog -- } \ Set transient parent for dialogs
  pdialog MWINDOW gtk_window_set_transient_for
;
```

8. Conclusion

The process of internationalisation has been completely automated. When making a new application, it is only necessary to include the required phrase handling files, and add a couple of words to the initialisation code. A huge amount of work has been saved. This technique is only possible in Forth, with its unique ability to control the compilation action.

NJN
30/7/19

EuroForth 2019
Forth returns to the automotive industry

Abstract

A stretch bending machine for producing automotive components is described, on which the automation system is programmed in Forth.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
R.J. Merrett B.Eng.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

A wide variety of processes are used to produce bodywork parts for vehicles. One of those processes is called stretch bending. A formed metal section (typically a U section) is bent into a complex shape while being simultaneously stretched so as to retain its sectional form without distortion. A typical usage might be the surrounding frame of a car window. The parts have tight tolerances and must be produced in high volumes. The control system of a machine that produces such parts is quite sophisticated.

2. Historical notes

The Euroforth 2003 conference included a visit to a company that made stretch bending machines. The machines at that time used hydraulic proportional valves, each controlled by a dedicated circuit with an IX1 single chip microcontroller. This communicated with a virtual programmable logic controller (VPLC), installed within a personal computer. The PC, VPLC and IX1 were all programmed in Forth.

The mechanical engineering company that made the stretch bending machines had enjoyed a great deal of success, creating machines that were used to produce numerous popular European cars, for all the main manufacturers.

Unfortunately, it was taken over by a much larger company which, a short time later, went bankrupt.

Some of the engineers from the original company went on to form a new company, though it took them some years before they accumulated the resources to offer a complex machine. Finally in 2019 they obtained a contract from Jaguar Land Rover, to build a machine for a new model. In turn, our company was called on to automate the machine. Forth returns to the automotive industry!

3. A truly international industry

Jaguar and Land Rover are historical British companies, and all design work is still carried out in the U.K. However, the owner of JLR is the Indian company Tata Motors. The final assembly of the new model will take place in Slovakia. The engines will be built in the U.K. However, the part that we are concerned with is to be made in Turkey. This means that the automation system must be usable in both English and Turkish.

4. Overview of a stretch bending machine

The material is presented to the stretch forming machine as a continuous strip of preformed steel or aluminium alloy section. The machine cuts a piece to the required length, grasps and then bends it in several planes, simultaneously stretching the section in order to avoid irregularities in the curved surfaces.

Some sections are formed in quite heavy material, and considerable forces are required to perform the stretching and bending actions. This means that the axes cannot be driven by stepper motors which are commonly used on industrial robots. In the past, hydraulic cylinders were used to obtain the necessary force. These days, d.c. servomotors combined with linear actuators are sufficiently powerful. The axes positions are measured by encoders.

In order that the finished parts fit precisely into their assemblies, considerable accuracy is required – the current specification calls for a positional tolerance of 0.01mm over a total axis length of up to 2m. To obtain the correct shape, the stretching and bending must take place over up to seven axes simultaneously, all moving in a co-ordinated manner.

The machines must operate quite rapidly, for example a high volume car might require around 1,000,000 parts per year. Even a company like JLR, which is a relatively low volume luxury manufacturer, produces almost half a million vehicles a year. The machines must be extremely reliable because they form part of a just-in-time production system with no parts storage. Downtime of less than 24 hours can stop a major car plant, with thousands of workers laid off.

5. Some design decisions of the control system hardware and software

a) The machine must be able to produce a variety of different parts, for different varieties of the same basic car - for example, long and short wheelbase models. This is achieved by having a standard machine base, to which are attached different tooling modules. The base, and each item of tooling has different sensor and actuator requirements, and therefore each has its own control panel with PLC I/O modules and servo drives as necessary.

b) To connect the various PLC and servo modules together, a fieldbus is necessary. A decision had been made to use Bosch servo drives, and this determined that the Sercos fieldbus system had to be used.

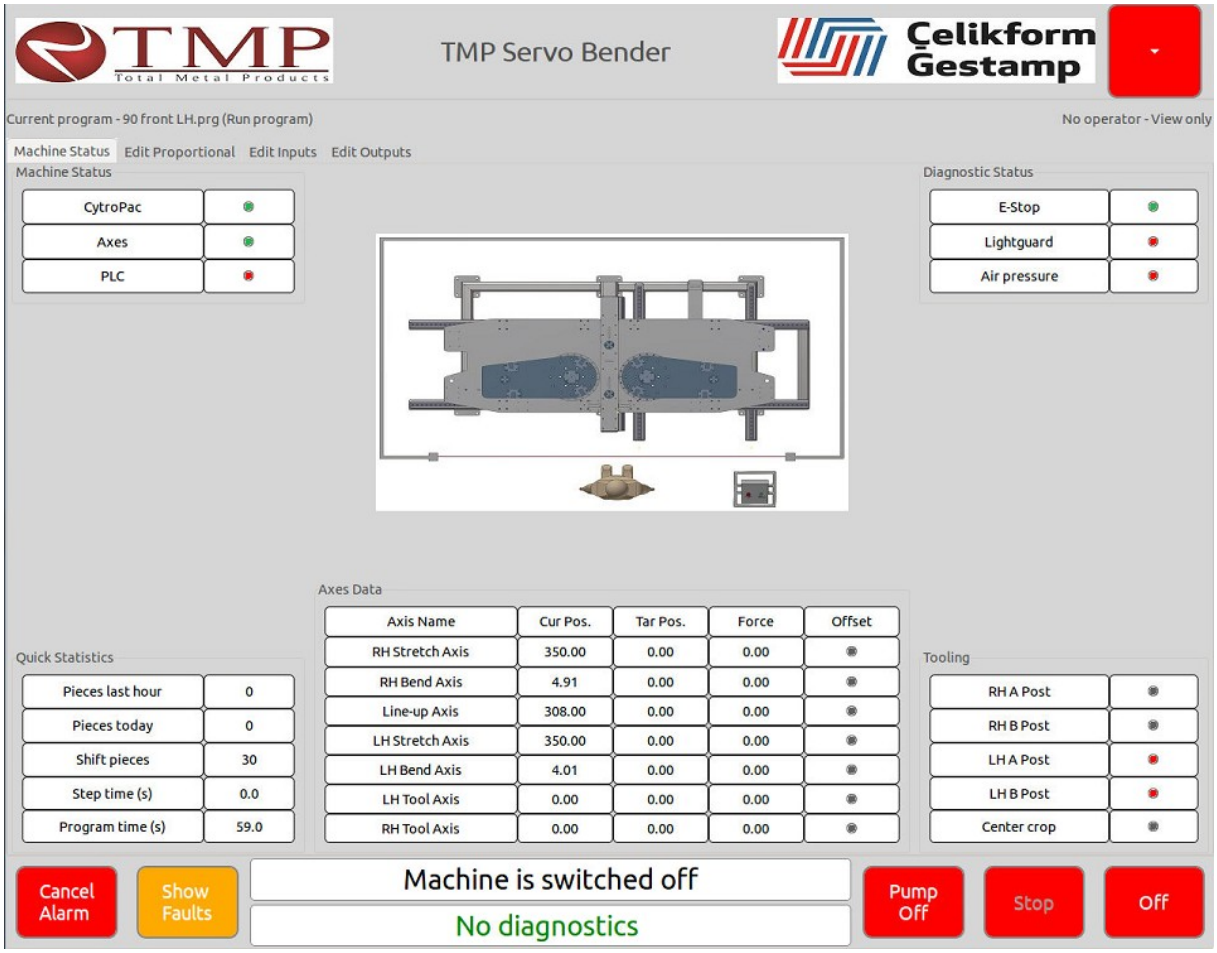
c) In turn, for ease of integration, the Bosch PLC had to be used. This unfortunately meant that IEC61131 had to be used for programming the PLC.

d) Supervisory control, diagnostics, visualisation, performance monitoring reports and statistics, and part profile programming and storage are done by an industrial PC running Ubuntu Linux and programmed in Forth using MPE VFX.

e) In a previous paper at Euroforth, I compared the benefits of programming in Forth, with IEC61131. These remarks still stand, therefore when dividing the control responsibilities between the PLC and the PC, as much of the work as possible is carried out by the PC.

6. Some key features of the software

a) The main visualisation



This is a live display showing the machine status, with an animated plan view of the bending process.

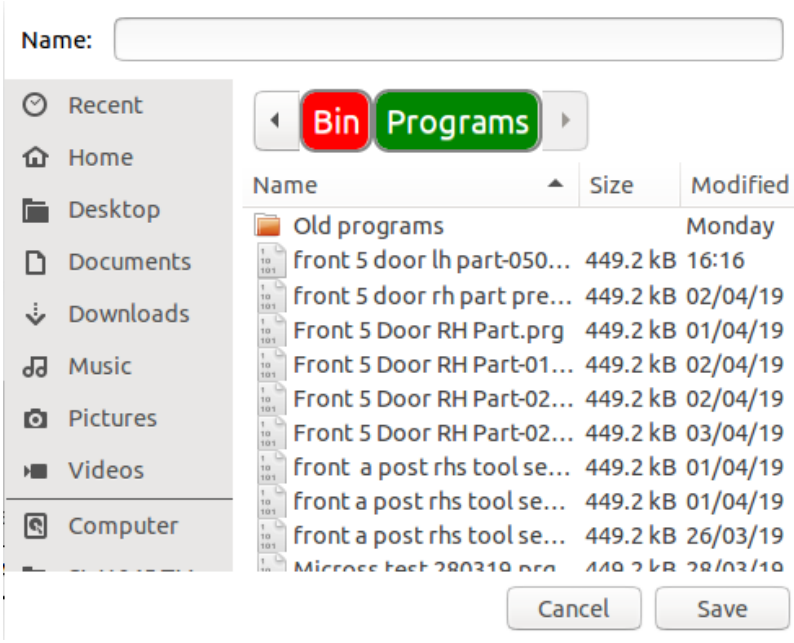
b) High speed communication with PLC

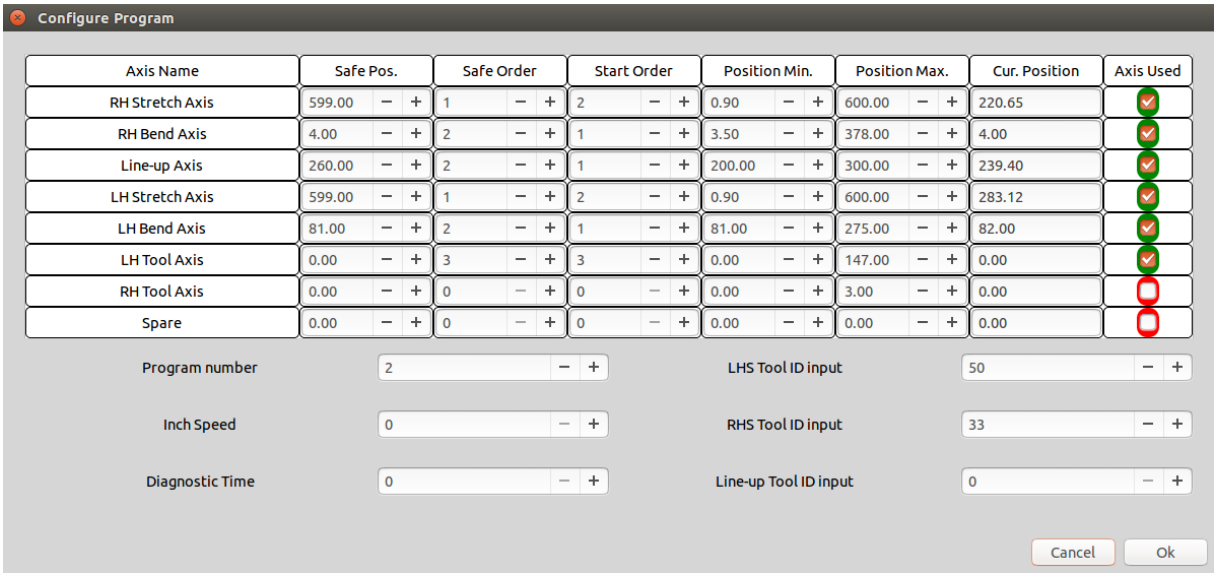
We use our standard method of data exchange from PC to PLC, using fixed format UDP messages over a dedicated Ethernet connection, on a 50ms tick.

c) Diagnostics

As well as the status of the safety system and the air and hydraulic supplies, the correct position of each axis is constantly monitored for tolerances.

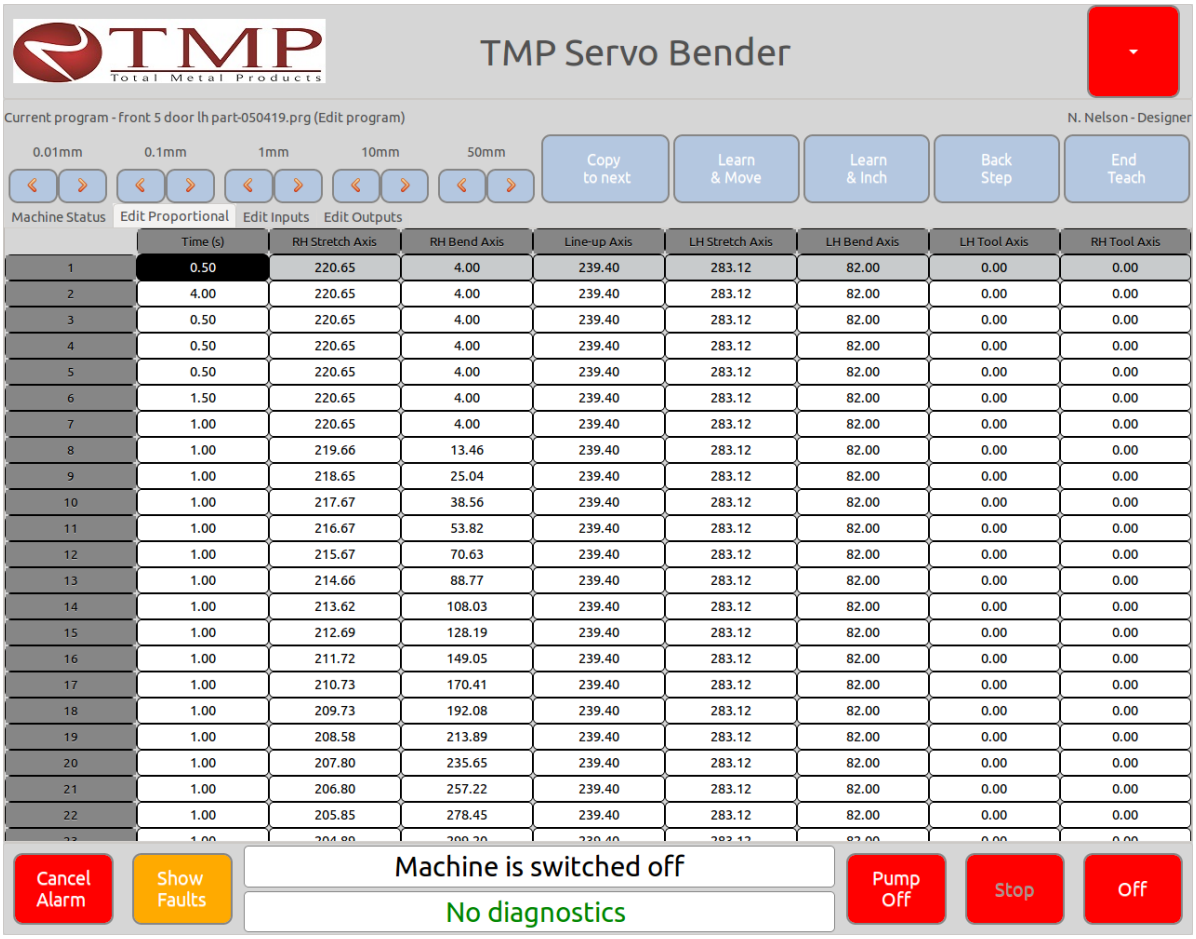
d) Program change





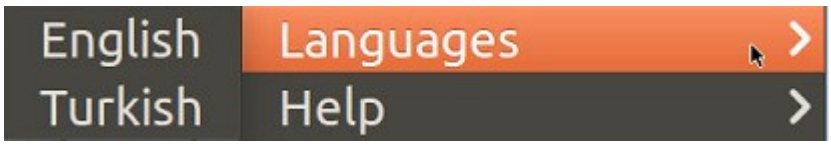
When the tooling needs to be changed, to suit a different model of the vehicle, the fieldbus configuration of the PLC and servo drive network changes, because some elements of the PLC are installed on the changeable tooling. So the system validates that the PLC configuration matches the selected bending program. This was one of the most difficult things to achieve, because the hardware application engineers had not encountered this requirement before.

e) Programming and teach mode



When designing the bending program for a new part, the procedure is to inch each axis step by step to a new desired position, using a "teach mode". This gets the basic shape right, but the bending program still needs fine tuning to account for dynamic operation.

f) Multilanguage support



The machine is installed in a factory just outside Istanbul! So all the on screen functions need to be in Turkish as well as English. See the previous paper "Internationalisation - a new approach in Forth".

7. Some major lessons learned from the project

a) Never overestimate the capabilities of major suppliers.

We were offered a lot of assistance by a team of application engineers from the hardware suppliers, particularly in getting the servo drives to function as required. It took about a week before we realised that by then we knew more about drives than they did!

b) Never underestimate your own engineering capabilities.

Although we had no experience of this type of drive before, we did have a lot of experience in multi-access positioning systems. Sufficient, in fact to realise after a while, that the advice offered by the hardware suppliers might not be the best.

c) Beware of newly introduced products.

The machine included two highly integrated hydraulic compressor packs, which were being used for the first time in the UK. These were fieldbus controlled. You would have thought that a hydraulic compressor was a fairly simple piece of equipment, but it's amazing how complex you can make them with a bit of imagination.

8. If we did it again, we would…

Not go for a new hardware supplier when starting a major new project. The double learning curve was a severe challenge.

9. Conclusion

Once again, Forth has shown its capabilities as an outstanding tool for automating industrial control equipment.

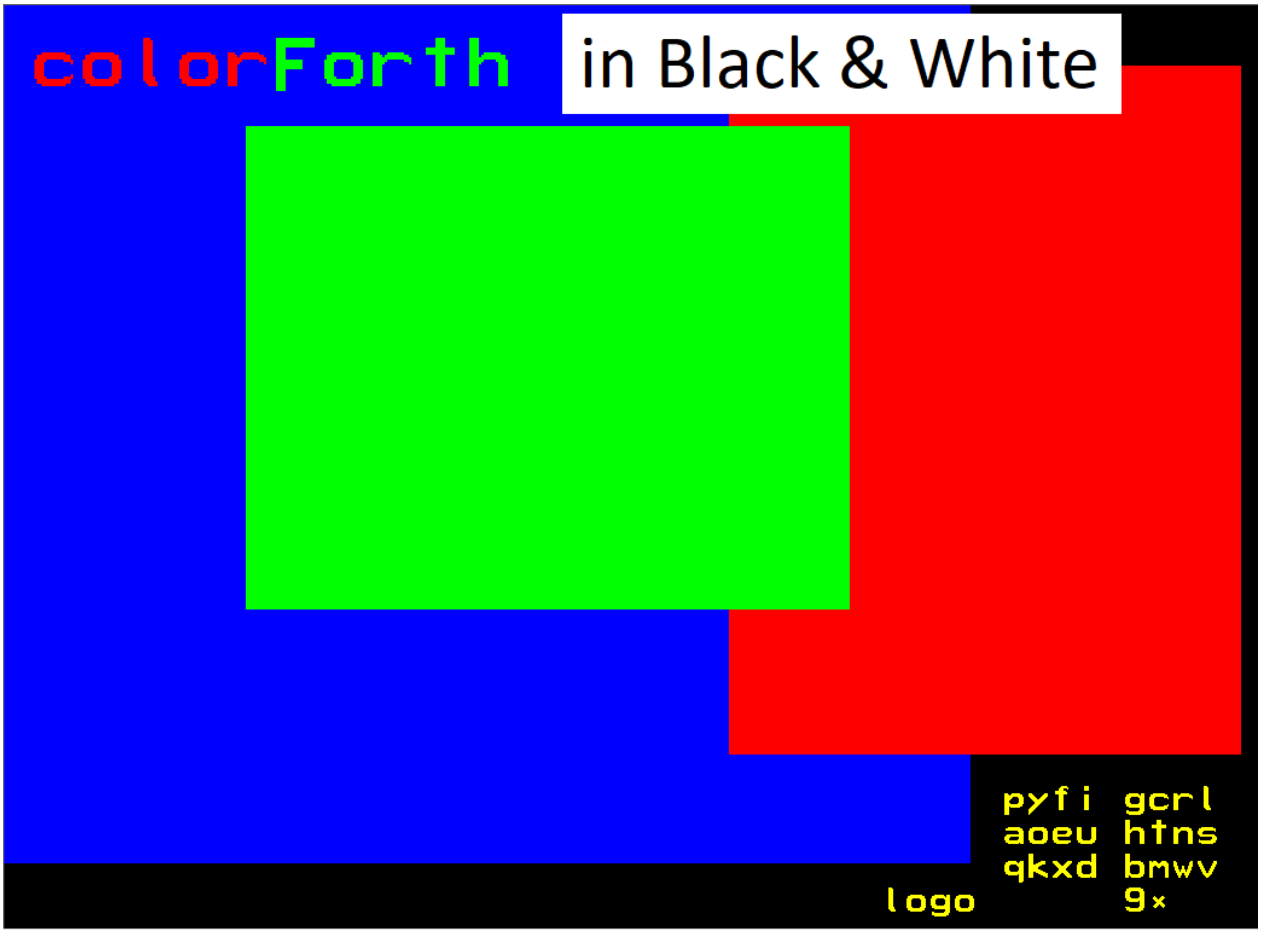
10. References

Industrial control languages: Forth vs IEC61131
N.J. Nelson
Euroforth 2000

A hydraulic servo controller using the IX1 microprocessor
A.C. Wheatley and N.J. Nelson
Euroforth 2003

Internationalisation - a new approach in Forth
N.J. Nelson
Euroforth 2019

NJN RJM
30/7/19



Contents

Summary

colorForth is a dialect of the Forth programming language, both of which languages were invented by Charles H. "Chuck" Moore ; - Forth around 1968, and colorForth in the late 1990's.

In this paper I hope to explain why colorForth is about so much more than just colour.

colorForth uses a 32 bit token as the basic unit of interaction between the computer and human being. Each token has a 28-bit human readable name field and 4 bits of meta-data ("colour"). The token's meta-data field can replace global variables such as STATE, allowing a simpler compiler and a more complex/powerful editor.

Conventional programming environments separate the editor, compiler and interpreter into discrete functional units, whereas colorForth puts them all into a blender and filters the resulting mush into something completely different.

Emergent Properties

One emergent property of the colorForth environment is the Magenta Variable, where setting a new value at run-time actually affects edit-time, by changing the pre-parsed source for the program.

Another is the Blue Token, which controls the behaviour of the editor at edit-time (over and above seeing what you type). This is similar to putting CRs and TABs into a text file, but Blue Tokens are extensible – you can run Forth code at edit-time.

Because the boundaries between editor, compiler and interpreter are blurred, you can choose what you do, and when you do it, much more easily than in a conventional environment. For example, version control could be added using Blue Tokens to retrieve earlier versions of code at edit-time and compile-time.

Shortening the Conceptual Gap

Edsger Dijkstra in his 1968 paper Go To Statement Considered Harmful states that:

"**we should […] do our utmost to shorten the conceptual gap between the static program and the dynamic process […]**", which I interpret as "**shorten the conceptual gap between *source text* and *program execution***". That is, make it as easy as possible for someone reading the source to create a conceptual model of what the program will do when it runs.

To make a program clear and easy to understand, each word should have a name with some mnemonic value, and should do something simple that is hinted at by that name. In this context, a "goto" means that something else happens while this word is executing which is most likely *not* hinted at by the word's name – this is therefore a bad thing.

Giving a Forth word the correct name is of course important, but by adding meta-data, the word (now a token) stores more information – not just what it does, but when it should do it : edit-time, compile-time or interpret-time.

Removing Punctuation

"Shortening the conceptual gap" applies to any computer language – Forth takes it to an extreme by defining a "word" as a sequence of characters surrounded by spaces, and leaving it to the author to decide about almost everything else.

Other languages add a more complicated syntax, restricted keywords and program style guides in order to lock people in to that language. From a Forth perspective these additions are just noise – they do nothing to shorten the conceptual gap between source and program. From a financial perspective these additions increase profit.

When Chuck Moore created colorForth one of his intentions was to use colour to replace punctuation:

becomes:

While the use of colour to replace punctuation is an interesting idea, it ultimately fails as a general-purpose programming language because a surprisingly high percentage of people are colour-blind. According to Wikipedia, red-green color blindness affects up to 8% of males and 0.5% of females of Northern European descent. It also makes it difficult to exchange "pure" colorForth source code in a monochrome text file.

I should point out here that when I work with Forth source in text files (*.f) I use my favourite editor (EditPlus) with a Forth colouring option, so the text appears in colour – but this has absolutely nothing to do with the use of colour in colorForth.

In colorForth, colours are just a *representation* of the "color" of the token, the bottom four bits of the token value. It is very easy to modify the colorForth editor to add conventional Forth punctuation. That is, the meta-data can be used to control what the user sees in the editor, what the compiler compiles or what the interpreter does.

Metadata and Colour

While the name “colorForth”, the coloured representation colorForth and the colourful appearance of the display all emphasise colour (spelled “color” in the USA), in fact the fundamental principles in colorForth go way beyond colour. Colour in this context is just one way of conveying meta-information about a computer program.

For example, conventional Forth uses ‘:’ to indicate the definition of a new Forth word, colorForth uses the colour red together with starting the definition on a new line.

While conventional Forth can have coding style standards that usually specify that colon definitions start on a new line, this not required. In colorForth, red tokens (that start a new word definition) are displayed on a new line automatically. There are some special blue tokens that modify this default behaviour, and this can in any case be changed, if desired, in the NASM source code.

In the cf2019 distribution of colorForth, pressing the F4 function key toggles between colorForth mode and a more conventional Forth display. This is easy to do because the information and meta-information (information about the information) are stored as 32 bit tokens, and can be displayed in any desired way. The F4 function also makes it easier for people who are colour-blind to read the code.

Token Colours

The following colours and their meaning is described below, from file cf2019.nasm :

```
actionColourTable:          ;    * = number
    dd colour_orange        ; 0    extension token, remove space from previous word, do not change colour
    dd colour_yellow        ; 1    yellow "immediate" word
    dd colour_yellow        ; 2  * yellow "immediate" 32 bit number in the following pre-parsed cell
    dd colour_red           ; 3    red forth wordlist "colon" word
    dd colour_green         ; 4    green compiled word
    dd colour_green         ; 5  * green compiled 32 bit number in the following pre-parsed cell
    dd colour_green         ; 6  * green compiled 27 bit number in the high bits of the token
    dd colour_cyan          ; 7    cyan macro wordlist "colon" word
    dd colour_yellow        ; 8  * yellow "immediate" 27 bit number in the high bits of the token
    dd colour_white         ; 9    white lower-case comment
    dd colour_white         ; A    first letter capital comment
    dd colour_white         ; B    white upper-case comment
    dd colour_magenta       ; C    magenta variable
    dd colour_silver        ; D
    dd colour_blue          ; E    editor formatting commands
    dd colour_black         ; F
```

Four Times

There are four logical periods of time in computer programming, starting from the human being and ending up at the computer :

1. Design-time - the human being thinks about how to solve the problem at hand
2. Edit-time - the human being types a program, using the computer running a previously written program (the Editor)
3. Compile-time - and compiles the program, using the computer running a previously written program (the Compiler)
4. Run-time - then asks the computer to run the compiled program, and tests the results

In Forth, there is an outer interpreter that collects words typed by the human being and interprets them.

When developing a program, the four times follow each other in a logical progression, repeating by cycling back to design- or edit-time as required, as in a REPL read–eval–print loop.

In general it is best to concentrate on the earliest possible logical time : a problem solved at compile-time consumes less resources than solving the same problem at run time, likewise a better design-time algorithm or way of approaching a problem can save both compile- and run-time effort.

An example is modular multiplication, calculating A B N **MOD , A taken to the power B modulo N.

Montgomery Multipication (https://en.wikipedia.org/wiki/Montgomery_modular_multiplication) is a design-time improvement that maps A to Montgomery form so that taking the result modulo N can be done by dividing by a power of 2 – this is equivalent to shifting and is very much faster than division. The result is then mapped back from Montgomery form to give a usable result, with better run-time performance.

By being very simple and attaching meta-data to data, colorForth can allow more control over the entire development environment – this could allow a major design-time improvement.

Philosophy : Keep It Simple

It is not easy to define simplicity – it is more of a direction than a goal. Sometimes *adding* complexity in one area can *decrease* the complexity overall. An example is the simple Text Input Buffer in conventional Forth being replaced by a pre-parsing Shannon-Fano encoder in colorForth – but this simplifies the compiler.

From Chuck Moore's book Programming a Problem-oriented Language :

"The Basic Principle

• Keep it Simple

As the number of capabilities you add to a program increases, the complexity of the program increases exponentially. The problem of maintaining compatibility among these capabililties, to say nothing of some sort of internal consistency in the program, can easily get out of hand.

You can avoid this if you apply the Basic Principle.

You may be acquainted with an operating system that ignored the Basic Principle. It is very hard to apply. All the pressures, internal and external, conspire to add features to your program.

After all, it only takes a half-dozen instructions; so why not? The only opposing pressure is the Basic Principle, and if you ignore it, there is no opposing pressure."

I am looking forward to discovering new ways of simplifying the total colorForth system, by adding carefully controlled complexity into certain key areas.

colorForth History

Around 2001 I downloaded Chuck Moores' public domain colorForth from his website and copied on to a 3.5 inch floppy disk. It was not easy to get working – I had to add a new, compatible floppy disk ISA board to make it work.

I was impressed, and wrote the article : colorForth and the Art of the Impossible and presented it at EuroForth 2001. I also had the great good fortune to spend about 45 minutes with Chuck, looking at his colorForth CAD system, OKAD II.

I love working in colorForth – I think it must be something genetic, certainly it appears not to be curable.

I presented another paper at EuroForth 2003 "The colorForth Magenta Variable", and handed out floppy disks with the first distribution of my version of colorForth.

Time marches on, and one of my two PCs still with a floppy disk drive, died. I still have the other one, in the cellar, "just in case". But it became obvious that colorForth needed to be updated to run from a USB stick.

A decade or so later, I presented a paper "Crypto colorForth" at EuroForth 2017 (the video is here), and demonstrated colorForth running from a USB stick. I believe that security and complexity are incompatible in computer software, and that colorForth can be the basis of a very secure operating system (without using files).

Actions, not Words

I strongly recommend that you, dear reader, run cf2019 as a program on a suitable computer. There are two ways of doing this :

1. Copy the binary image file cf2019.img directly onto a USB drive, and boot the computer using this drive.
2. Run cf2019 in a bochs environment under Windows. Double click on the file **go.bat** in the cf2019 distribution to do this.

This is because "the map is not the territory" – both Forth and colorForth provide an interactive environment that is best experienced, rather than discussed.

Downloads

colorForth can be downloaded here, and can be copied to a USB drive to run native on most PCs, or under Bochs for Windows.

Documentation is available here, and is included in the distribution.

Enjoy!

Howerd Oakford 2019 Aug 31

MUTEX (MUTual EXclusion) Mechanism in Hardware

Klaus Schleisiek - kschleisiek at freenet.de

Thanks to µCore's PAUSE signal, mutual exclusion processing can be completely realised in hardware. This gets rid of a very popular source of hard to track errors in complex control systems.

What is mutual exclusion and why do we have to bother?

In complex control systems, there are usually several control loops, each of which is running in its own task. For input signal acquisition, the same A/D converter (ADC) may be shared by several tasks. A/D conversion usually takes several processor cycles and therefore, a MUTEX mechnisam is needed for two things:

a) To prevent any task to touch the ADC while it is converting another task's input signal.

b) To guarantee that a conversion result is actually read by the task that started the conversion previously.

For the sake of discussion I assume that we have an ADC with an integrated eight channel multiplexer. It is connected to a hardware/software interface that allows to store a channel number into its memory mapped ADC register (**ADC_reg**) initiating signal acquisition. When the A/D conversion has finished, **ADC_reg** can be read returning the conversion result of the selected channel. In order to detect when the conversion has finished, we also have flag bit **#adc_ready** in the FLAG register (**FLAG_reg**). In addition we can define **Semaphor**s with methods **lock** and **unlock**.

In the first part of the paper I will show how mutual exclusion is usually implemented in software.

In the second part I will show how the entire MUTEX mechanism can be realised in hardware using µCore's PAUSE signal.

Conventional MUTEX in software

At first some informal word definitions of a typical co-operative multi tasking system:

pause (--) puts the current task to sleep and calls the scheduler in order to give another task the chance to run. Please note that µCore's PAUSE signal input and the Forth word **pause** are completely different things!

Semaphor is a defining word that creates a semaphor. In priciple it is a variable that allows to store true and false.

lock (addr --) as long as the semaphor at addr is true, **pause** is executed. When it is false, it is set to true and execution of the task continues.

unlock (addr --) sets the semaphor at addr to false.

```
Semaphor sema_ADC

: sample  ( channel -- sample )
   sema_ADC lock   ADC_reg !
   BEGIN  pause   FLAG_reg @ #adc_ready and UNTIL
   ADC_reg @   sema_ADC unlock
;

Variable Result
```

And now a task may acquire the analog data of channel 4 in a safe way using the following phrase:

```
... 4 sample Result ! ...
```

MUTEX (MUTual EXclusion) Mechanism in Hardware

MUTEX in hardware

As the Janus-faced side of interrupts, μCore has an additional hardware input signal **pause**, which, when raised, aborts the current instruction, pushes the instruction's program memory address on the return stack, and branches to the pause trap.

interrupt: An event did happen that was **not** expected by the software.

pause: An event did **not** happen that was expected by the software.

In a single task environment, the pause trap just holds an **`exit`** instruction. As a result, the processor would continuously try to execute the instruction that raised the pause signal until some external event makes the instruction executable.

In a multi tasking environment, the pause trap holds a branch to Forth's **`pause`** routine defined above and therefore, the processor can do other things while waiting for the external event to happen.

As before, we need the flag bit **#adc_ready**. But this time, it is not only a flag that can be read by the processor, it becomes the semaphor itself.

Here is some hardware pseudo code (simplified VHDL) that is needed for the MUTEX mechanism:

`read_ADC_reg` is true when the Forth phrase **`ADC_reg @`** is executed.

`write_ADC_reg` is true when the Forth phrase **`ADC_reg !`** is executed.

`ADC_busy` is true while the ADC is converting. Very often this is an output signal of the ADC chip.

The hardware implementation consists of a combinatorial part that feeds the pause input of μCore

```
pause <= true
         WHEN (read_ADC_reg = true AND ADC_busy = true) OR
              (write_ADC_reg = true AND FLAG_reg(#adc_ready) = false)
         ELSE false;
```

and a sequential part that controls the **#adc_ready** bit storing its result on the rising edge of a clock.

```
IF  rising_edge(clk)  THEN
   IF  write_ADC_reg = true  THEN
      FLAG_reg(#adc_ready) <= false;
   END IF;
   IF  read_ADC_reg = true AND pause = false  THEN
      FLAG_reg(#adc_ready) <= true;
   END IF;
   IF  reset = true  THEN
      FLAG_reg(#adc_ready) <= true;
   END IF;
END IF;
```

And now a task may acquire the analog data of channel 4 in a safe way using the following phrase:

```
... 4 ADC_reg !   ADC_reg @ Result ! ...
```

In essence, you can treat the **ADC_reg** similar to a variable without bothering about conversion time needed or mutual exclusion on the software level any more.

Getting Rid of µCore's 2-phase Execution Cycle

Klaus Schleisiek - kschleisiek at freenet.de

µCore_1 did have a two phase instruction execution cycle, because the internal blockRAMs in FPGAs do have an internal address register that needs to be set first before data can be read. Each phase lasts for at least one clock cycle:

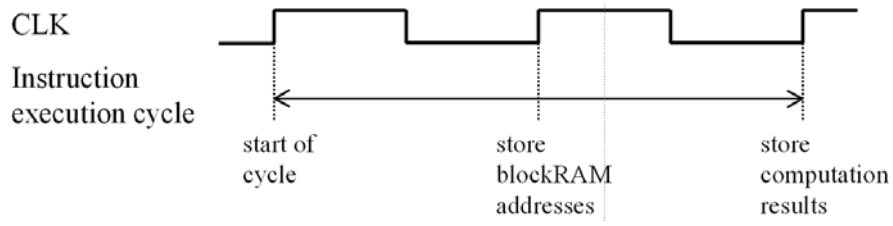
1st phase: All address computations are done for µCore's three memory areas:

1. Program memory
2. Data memory and return stack
3. Data stack

At the end of the 1st phase, these addresses may be registered in the respective memory areas depending on an instruction's need.

2nd phase: Now the memory areas may be read and used for computing results as well as reading the next instruction. At the end of the 2nd phase, results can be stored on the data stack, in the data memory, on the return stack, and in the instruction register.

This is a waste of computing time for the majority of instructions that do not need to read any memory except for the program memory.



I got rid of this unfortunate scheme by splitting up a random blockRAM memory access into two subsequent, indivisible instructions. Along the way, I invented a generic 'instruction chaining mechanism', which can readily be used for indivisible read-modify-write instructions like +!.

Program memory

The trick for the program memory is to get rid of the instruction register. At the end of every µCore_2 execution cycle the next instruction's address gets registered in the program memory's address register. In the next cycle the output data of the program memory constitutes the next instruction. The instruction register of µCore_1 is no longer needed. :-)

Data stack

The trick for the data stack memory is simple: The data stack pointer always points at the the last item pushed into the data stack memory. This implies that its internal address register is set to said item and therefore, it is ready to be read back in the next cycle already. No random access will ever happen, because the data stack memory solely serves as a stack that can only be pushed or poped.

Data memory and return stack

In µCore, both the data memory and the return stack reside in different regions of the same memory. Each memory read must be split up into two instructions: The 1st instruction sets the data memory's address register, and the 2nd instruction can then read the data and operate on it. These two instructions need to be indivisible or else an interrupt may overwrite the memory's address register before executing the 2nd instruction.

Getting Rid of μCore's 2-phase Execution Cycle

The 1st instruction (e.g. @) will be compiled by μCore's cross compiler. When it is executed, it will schedule the 2nd instruction for execution in the next cycle without advancing the program counter.

To this end I invented a general mechanism to chain up a series of indivisible instructions.

The following table lists μCore_2's two-cycle instructions:

instruction	2nd cycle
r>	store memory data into TOR
rdrop	store memory data into TOR
exit, iret	store memory data into TOR
?exit	only executed when TOS /= 0: store memory data into TOR
next	only executed when finishing a FOR ... NEXT loop (TOR = 0): store memory data into TOR
@	store memory data into TOS
+!	write (memory data + NOS) back into memory
I	store the sum of TOR and data memory (2nd return stack item) into TOS
IF	in the 1st cycle, the branch address is dropped, in the 2nd cycle the flag as well

These two cycle instructions constitute about 20 % of the instructions executed by a typical program. Therefore, total throughput of μCore_2 has increased by about 65 % compared to μCore_1.

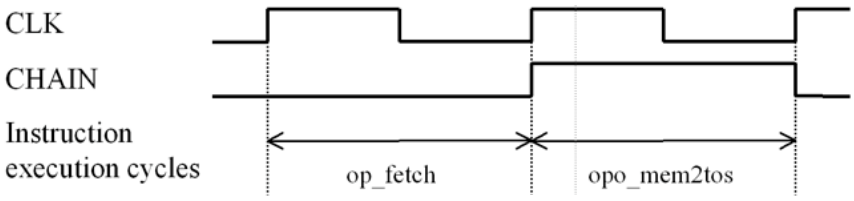
Instruction chaining mechanism

inst is an 8-bit register that delivers the next instruction when **chain = true** (see below).

chain is a flip-flop, which must be set when the next instruction should be read from **inst** instead of the program memory. When **chain** is being set, the program counter will not be advanced.

Interrupts will be suppressed as long as **chain = true**.

Given this mechanism, an arbitrary number of single instructions may be chained up for application specific opcodes.



Execution of @

Reset

After a reset, execution should start at address zero. Therefore, the processor must be set up to execute a noop while fetching the first instruction from address zero. This can be realised as follows:

```
IF  reset = true  THEN
   inst <= op_NOOP;
   chain <= true;
   program_memory_addr <= 0;
END IF;
```

Galois Fields and Forth

Bill Stoddart and John Goldman

September 25, 2019

Abstract

Galois fields are rich finite algebraic structures with applications in cryptography, error correcting codes, experimental design, constraint programming and pattern recognition. We describe some of these fields and the structures related to them known as Latin Squares, which are used in many applications. We describe and implement examples of the polynomial arithmetic that underlies Galois Fields, we describe the automorphisms between different implementations of the "same" field, and we give an implementation of the field used in the Advanced Encryption Standard. As an example application we consider in detail the construction of a pack for the children's card game Dobble, in which there are cards marked with symbols such that any two cards share exactly one symbol. We include mathematical appendices in which we prove, or show how to prove by comprehensive validation, various important properties satisfied by these fields.

1 Introduction

Galois fields are named in honour of the French mathematician Evariste Galois, an inspiring but tragic figure who died at 20 in a duel yet left a huge legacy. He discovered them whilst investigating the properties of polynomials. These fields are rich finite algebraic structures with applications in cryptography, error correcting codes, experimental design, constraint programming and pattern recognition. Each Galois field consist of values $0..p$ along with two operations which are analogous to multiplication and addition. These operations obey the axioms of a mathematical "field", which are also shared by real numbers, but not, totally, by integers, since every element in a Galois

field has an exact multiplicative inverse. Galois fields also support operations analogous to subtraction and division, and division is perfect, there is never a remainder.

This paper is organised as follows. In section 2 we give the axiomatic properties of a Galois field. In section 3 we show how Galois fields with a prime number of elements are implemented. We define Latin squares, and also what it means for two Latin squares to be orthogonal. We show how orthogonal Latin squares can be used to solve an old playing card puzzle. In section 4 we look at fields of size q^n where q is prime, and in particular we implement the field of size 8. We show there are different implementations of this field which are related by an automorphism. In section 5 we implement the Galois field with 256 elements, which is mandated for use in the Advanced Encryption Standard.

In section 6 we look at our main example problem, which is the allocation of symbols to cards in Dobble packs. In section 7 we consider a Forth implementation for the construction of a Dobble pack. In section 8 we conclude.

In the mathematical appendices we prove that modular arithmetic does not generally yield a Galois Field, and we prove the existence of the mutually orthogonal Latin squares associated with each field. Finally we consider the proof of field axioms by exhaustive verification.

2 Galois Field Properties

A Galois field $\{\mathcal{F}\}$ of size p consists of two operations analogous to addition and multiplication, acting on a set of values 0..p and obeying certain axioms. A Galois field will exist whenever p is the power of a prime., e.g. 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, .. 256 ...

The addition and multiplication on a Galois field of size $p=q^n$ where q is prime are those of "polynomial modular arithmetic", which we will explain in due course. In the case where p is prime this reduces to addition and multiplication modular p and we can define the Galois field operations as:

```
: +0 + p MOD  ;          : *0  *  p  MOD  ;
```

We will write the operations of a Galois field as $+_0$ and $*_0$ to distinguish them from normal addition and multiplication. If considering more than one implementation of the same field we will also use the names $+_1$, $*_1$, $+\_$, and $*\_$. The field size will be clear from context.

The axioms of a Galois field are a set of rules that, with one exception, also apply to integers. The exception is that every element has an inverse *within the field*. The integer `n` can be said to have inverse `1/n`, but this inverse is not itself an integer. The inverses of Galois field elements are in the field.

2.1 Axioms

In the following $\mathtt{u}, \mathtt{v}, \mathtt{w}$ are arbitrary values in the field $\mathcal{F}$.

Closure. If `u`, `v` are in the field so are $\mathtt{u} +_0 \mathtt{v}$ and $\mathtt{u} *_0 \mathtt{v}$

Commutativity. $\mathtt{u} +_0 \mathtt{v} = \mathtt{v} +_0 \mathtt{u}$ and $\mathtt{u} *_0 \mathtt{v} = \mathtt{v} *_0 \mathtt{u}$

Associativity. $(\mathtt{u} +_0 \mathtt{v}) +_0 \mathtt{w} = \mathtt{u} +_0 (\mathtt{v} +_0 \mathtt{w})$

Distributivity $\mathtt{u} *_0 (\mathtt{v} +_0 \mathtt{w}) = (\mathtt{u} *_0 \mathtt{v}) + (\mathtt{u} *_0 \mathtt{w})$

Unit property $\mathtt{u} *_0 \mathtt{1} = \mathtt{u}$

Zero properties $\mathtt{u} +_0 \mathtt{0} = \mathtt{u}$ and $\mathtt{u} *_0 \mathtt{0} = \mathtt{0}$

Inverses. If `u` is non-zero it has a multiplicative inverse $\mathtt{u}^{-1}$ in $\mathcal{F}$ such that $\mathtt{u} *_0 \mathtt{u}^{-1} = \mathtt{1}$, and has an additive inverse $\sim\mathtt{u}$ in $\mathcal{F}$ such that $\mathtt{u} +_0 \sim\mathtt{u} = \mathtt{0}$

The existence of inverses means division and subtraction may be defined, with $\mathtt{a} /_0 \mathtt{b} \mathrel{\widehat{=}} \mathtt{a} *_0 \mathtt{b}^{-1}$ and $\mathtt{a} -_0 \mathtt{b} \mathrel{\widehat{=}} \mathtt{a} +_0 \sim\mathtt{b}$

3 Latin Squares, orthogonality, and a combinatorial playing card problem

A Latin square is an $\mathtt{n} \times \mathtt{n}$ array containing `n` different elements with each of these occurring once in each column and once in each row.

Two Latin squares are orthogonal if the pairs of elements formed from the first and second square are all different.

Here are two orthogonal Latin squares, together with the square of pairs formed from them:

```
0 1 2 3   0 1 2 3   0,0 1,1 2,2 3,3
1 0 3 2   2 3 0 1   1,2 0,3 3,0 2,1
2 3 0 1   3 2 1 0   2,3 3,2 0,1 1,0
3 2 1 0   1 0 3 2   3,1 2,0 1,3 0,2
```

A Galois field of size `n` is associated with `n-1` mutually orthogonal Latin squares, with the element at the j,kth position the ith such square given by $\texttt{i} *_0 \texttt{j} +_0 \texttt{k}$. A mathematical proof is given in the appendix.

As an example application of orthogonal Latin squares, and a justification of the term "orthogonal", consider finding a solution to the problem of arranging the 16 picture cards from a pack in a 4×4 square such that in each row and each column we have one Ace, one King, one Queen, one Jack, one Heart, one Diamond, one Club, and one Spade. We can use two orthogonal Latin squares to split the program into two. We use the first Latin square to allocate positions for aces, kings queens and jacks, and the second to allocate positions to hearts diamonds clubs and spades. Under this interpretation the Latin squares above give the shown solution to our picture card problem.

4 Polynomial arithmetic and fields of size 2^n

To implement a Galois field of size $\texttt{q}^\texttt{n}$ where `q` is prime we use addition and multiplication of polynomials with coefficients taken from the field G(q). To explain this fully we consider fields of size $2^\texttt{n}$.

The Galois field G(2) has elements 0,1 with the multiplication:

$$0 *_0 0 = 0,\ 0 *_0 1 = 0,\ 1 *_0 0 = 0,\ 1 *_0 1 = 1$$

so multiplication in this field is equivalent to logical AND. And with the addition we have:

$$0 +_0 0 = 0,\ 0 +_0 1 = 1,\ 1 +_0 0 = 1,\ 1 +_0 1 = 0$$

so addition in this field is equivalent to logical XOR. Also, subtraction modulo 2 is identical to addition modulo 2 so addition and subtraction in the field are identical.

We now consider the polynomial arithmetic required to implement the field G(8), whose elements are the polynomials of order 2 and below. Given that the coefficients of these polynomials are from G(2), i.e. take only the values 0 and 1, we have 8 such polynomials, which are:

$$0\ ,\ 1\ ,\ x\ ,\ x +_0 1\ ,\ x^2\ ,\ x^2 +_0 1\ ,\ x^2 +_0 x\ ,\ x^2 +_0 x +_0 1$$

and these have coefficients that can be encoded in 3 bits as:

$$000\ ,\ 001\ ,\ 010\ ,\ 011\ ,\ 100\ ,\ 101\ ,\ 110,\ 111$$

these polynomials, or more exactly the encoding of their coefficients, will be the members of our implementation of the field G(8). [1]

An example of polynomial addition in this system is:

$$(x^2 +_0 1) +_0 (x +_0 1) = x^2 +_0 x +_0 1 +_0 1 = x^2 +_0 x$$

The bit encoded form of the same calculation, with $\oplus$ representing exclusive or, is $101 \oplus 011 = 110$.

The Forth definition for addition in field G(8), or any field $G(2^n)$, will be

```
: +0  ( n1 n2 -- n3 , n3 = n1 +0 n2 )  XOR ;
```

An example of polynomial multiplication is

$$(x^2 +_0 x +_0 1) *_0 (x^2 +_0 x) = x^4 +_0 x^3 +_0 x^2 +_0 x^3 +_0 x^2 +_0 x$$
$$= x^4 +_0 x$$

We note that the result $x^4 +_0 x$ isn't one of the 8 polynomials of G(8), so we can't use polynomial multiplication as the multiplication operator $*_0$ of our field. We have to reduce the result my taking its modulus relative to a reduction polynomial. This is analogous to modular arithmetic where our multiplication operator for modulus p arithmetic is defined as:

```
: *0 ( n1 n2 -- n3, n3 = n1 *0 n2) * p MOD ;
```

[1] Mathematical note: we treat polynomials as mathematical objects, rather than as expressions denoting mathematical objects. Considered as expressions, polynomials x and x^2 are identical, since with mod 2 arithmetic $x = x^2$. Under the interpretation in which polynomials are mathematical objects they correspond formally to sequences of coefficients $\langle 0, 1, 0 \rangle$ and $\langle 1, 0, 0 \rangle$ and polynomial arithmetic operations are defined on such sequences of coefficients. Representing such sequences of coefficients as bit sequences is an implementation technique.

Just as, for modular arithmetic to yield a field p must be prime, that is a number without factors, our reduction polynomial must also not have factors. It must also be higher order than the polynomials on the field. A suitable polynomial is $x^3 +_0 x +_0 1$.

The polynomial arithmetic we are performing is analogous to our numeric arithmetic. Indeed, we write our numbers in an abbreviated polynomial form; for example 406 is the value of $4x^2 + 0x + 6$ when $x = 10$. We might calculate the quotient and modulus when dividing 406 by 123 as follows

```
      3
123)406
    369
     37
```

The calculation shows us that

$406 = 123 * 3 + 37$

Our division of $x^4 +_0 1$ by $x^3 +_0 x +_0 1$ can be shown as follows (we explicitly include the zero terms in x, x^2 and x^3)

$$\begin{array}{r} x \\ x^3 +_0 x +_0 1 \,\overline{)\, x^4 +_0 0 +_0 0 +_0 0 +_0 1} \\ \underline{x^4 +_0 0 +_0 x^2 +_0 x} \\ x^2 +_0 x +_0 1 \end{array}$$

and shows us that $x^4 +_0 1 = (x^3 +_0 x +_0 1) *_0 x +_0 x^2 +_0 x +_0 1$

The term we are interested in here is the remainder $x^2 +_0 x +_0 1$. this is the result of our Galois field multiplication. In full:

$$(x^2 +_0 x +_0 1) *_0 (x^2 +_0 x) =$$
$$((x^2 +_0 x +_0 1) * (x^2 +_0 x)) \bmod (x^3 +_0 x +_0 1) =$$
$$x^2 +_0 x +_0 1$$

Multiplication in the field G(8) can be defined in Forth as:

```
: *0 ( a b -- a *0 b, multiplication of elements from G(8) )
   (: a b :) 0 ( the polynomial product a*b will be collected on the
   stack, we note XOR being used for addition since we are using
   modulo 2 arithmetic on our polynomial coefficients )
   1 b AND IF a XOR THEN
   a 2* to a
   2 b AND IF a XOR THEN
   a 2* to a
   4 b AND IF a XOR THEN
   ( modulo 2 polynomial product now on stack, we now divide
     it by our reduction polynomial x^3+x+1 )
```

```
DUP 16 AND IF 22 XOR THEN
DUP   8 AND IF 11 XOR THEN ( 11 ~ 1011₂ ~ x³+x+1 ) ;
```

The reduction polynomial is not unique (except in the case of G(4)). For G(8) an alternative reduction polynomial is x^3+x^2+1, and we will use $+_1$ and $*_1$ as the names for the corresponding field operations. We have the following addition and multiplication tables.

$+_0$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

$*_0$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

$+_1$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

$*_1$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	5	7	1	3
3	0	3	6	5	1	2	7	4
4	0	4	5	1	7	3	2	6
5	0	5	7	2	3	6	4	1
6	0	6	1	7	2	4	3	5
7	0	7	3	4	6	1	5	2

Although it is accepted usage to refer, in the singular, of "the field G(8)" we clearly have two different Galois field multiplication operators. However, they may be seen as acting on different encodings of the abstract elements of the field G(8) rather than being operations from two different fields. The following permutation relates the encodings used with reduction polynomial $x^3 +_0 x +_0 1$ to those used with reduction polynomial $x^3 +_0 x^2 +_0 1$

$\mathtt{perm} = \{ 0 \mapsto 0,\ 1 \mapsto 1,\ 2 \mapsto 3,\ 3 \mapsto 2,\ 4 \mapsto 5,\ 5 \mapsto 4,\ 6 \mapsto 6,\ 7 \mapsto 7 \}$

Now for any $a, b \in 0..7$ we have the following "automorphism" relating $+_0$ to $+_1$

$\mathtt{perm}(a +_0 b) = \mathtt{perm}(a) +_1 \mathtt{perm}(b)$

and a similar automorphism relating $*_0$ to $*_1$

$\texttt{perm(a} *_0 \texttt{b)} = \texttt{perm(a)} *_1 \texttt{perm(b)}$

In Forth we can implement this permutation function as:

```
HERE 0 C, 1 C, 3 C, 2 C, 5 C, 4 C, 6 C, 7 C,
: PERM ( 8b -- 8b ) LITERAL + C@ ;
```

and the checks we must do to establish the automorphisms are that for arbitrary I and J from the field

```
I J +0 PERM   I PERM J PERM +1  =
I J *0 PERM   I PERM J PERM *1  =   AND
```

5 Galois fields and encryption

All unary Galois field operations (add x, subtract x, multiply by x, divide by x, additive inverse, multiplicative inverse) have inverses. Thus if such operations are used to encode data, we can always rely on there being inverse operations that can decode it. Also, there are unary operations to take any point in the field to any other point in the field. These properties make Galois field operations useful for scrambling bits during encryption.

The field G(256), with reduction polynomial $x^8 + x^4 + x^3 + x + 1$ is mandated for use in the Advanced Encryption Standard, which (Wikipedia tells us) is "currently the only publicly accessible cipher approved by the National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module".

An implementation of G(256) operations is given below. They follow the same form of polynomial addition and multiplication as we saw for G(8).

```
( Our reduction polynomial x^8+x^4+x^3+x+1 encodes to 100011011_2 )
2 BASE ! 100011011 CONSTANT poly HEX

: +_ XOR ;

: *_ (: a b :) 0
   1 b AND IF a XOR THEN
```

```
   a 2* to a
   2 b AND IF a XOR THEN
   a 2* to a
   4 b AND IF a XOR THEN
   a 2* to a
   8 b AND IF a XOR THEN
   a 2* to a
   10 b AND IF a XOR THEN
   a 2* to a
   20 b AND IF a XOR THEN
   a 2* to a
   40 b AND IF a XOR THEN
   a 2* to a
   80 b AND IF a XOR THEN
  ( modulus 2 polynomial product now on stack, its
      max possible value is < 8000 hex )
   DUP 4000 AND IF [ poly 40 * ] LITERAL XOR THEN
   DUP 2000 AND IF [ poly 20 * ] LITERAL XOR THEN
   DUP 1000 AND IF [ poly 10 * ] LITERAL XOR THEN
   DUP 800 AND IF [ poly 8 * ] LITERAL XOR THEN
   DUP 400 AND IF [ poly 4 * ] LITERAL XOR THEN
   DUP 200 AND IF [ poly 2 * ] LITERAL XOR THEN
   DUP 100 AND IF poly  XOR THEN ;

DECIMAL
```

6 Dobble

Dobble is a card game in which 8 symbols appear on each card, and any two cards have exactly one symbol in common.

We can generalise this to `n` symbols per card, so long as `n-1` is the power of a prime. We illustrate the construction of a Dobble pack using 5 symbols per card, represented by digits 0..9 and letters A , B .. K

We begin by writing down the cards that contain the symbol 0. To see how many such cards there, it will help if we write down at the same time the cards that contain 1. As we write down each 0 card we distribute its symbols between the 1 cards. After we have added the card `05678` we have the following situation, which shows us we need 4 cards that contain 1 (but not

0) in order to distribute the symbols 5, 6 7 and 8 between them, and thus avoid having a card with more than 1 symbol in common with `05678`

```
0 1 2 3 4
0 5 6 7 8   1 5 ? ? ?
            1 6 ? ? ?
            1 7 ? ? ?
            1 8 ? ? ?
```

continuing in this way and adding blocks for cards containing 2, 3 and 4 we arrive at.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 9 A B C   1 6 A E I   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 D E F G   1 7 B F J   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
0 H I J K   1 8 C G K   2 ? ? ? ?   3 ? ? ? ?   4 ? ? ? ?
```

At this point we can see that for a pack with 5 symbols per card we need 21 different symbols and that our pack will contain `5+4*4=21` cards. In general a Dobble pack with `N` symbols per card will use $N+(N-1)^2$ symbols and have $N+(N-1)^2$ cards.

The sub-matrices for the 0 and 1 blocks are complete, and the sub-matrix for the 1 block is the transpose of that for the 0 block (rows of the 0 block have become columns of the 1 block). Now we use the orthogonal Latin squares of the Galois field G(4) to position the elements of the sub-matrix of block 1 in each of the remaining sub-matrices. These Latin squares are:

```
0  1  2  3      0  1  2  3      0  1  2  3
1  0  3  2      2  3  0  1      3  2  1  0
2  3  0  1      3  2  1  0      1  0  3  2
3  2  1  0      1  0  3  2      2  3  0  1
```

The top rows of these Latin squares tell us how to share the row `59DH` of sub-matrix 1 between the sub-matrices 2, 3 and 4.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 ? ? ?   3 5 ? ? ?   4 5 ? ? ?
```

```
0 9 A B C   1 6 A E I   2 ? 9 ? ?   3 ? 9 ? ?   4 ? 9 ? ?
0 D E F G   1 7 B F J   2 ? ? D ?   3 ? ? D ?   4 ? ? D ?
0 H I J K   1 8 C G K   2 ? ? ? H   3 ? ? ? H   4 ? ? ? H
```

The second rows of the Latin squares tell us how to share the row `6AEI`.

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 A ? ?   3 5 ? E ?   4 5 ? ? I
0 9 A B C   1 6 A E I   2 6 9 ? ?   3 ? 9 ? I   4 ? 9 E ?
0 D E F G   1 7 B F J   2 ? ? D I   3 6 ? D ?   4 ? A D ?
0 H I J K   1 8 C G K   2 ? ? E H   3 ? A ? H   4 6 ? ? H
```

the third and forth rows of the Latin squares tell us how to share the the rows `7BFJ` and `8CGK`. We have now shared the symbols `5` to `K` between the rows of the 2 3 and 4 sub-matrices in such a way that we have one symbol in common between any two rows, and because the 0 sub-matrix is the transpose of the 1 sub-matrix, we also have one symbol in common between the rows of the 0 sub-matrix and the rows of the 2 3 and 4 matrices, and we have completed our solution:

```
0 1 2 3 4
0 5 6 7 8   1 5 9 D H   2 5 A F K   3 5 C E J   4 5 B G I
0 9 A B C   1 6 A E I   2 6 9 G J   3 8 9 F I   4 7 9 E K
0 D E F G   1 7 B F J   2 7 C D I   3 6 B D K   4 8 A D J
0 H I J K   1 8 C G K   2 8 B E H   3 7 A G H   4 6 C F H
```

7 Implementing the construction of a Dobble pack in Forth

Given a Galois field `G(p)`, we consider how to implement the construction a Dobble pack with `N=p+1` symbols per card, as outlined above. We allocate space for the Dobble pack, with 1 byte used to represent each symbol. Each run of `p+1` bytes records the symbols on a particular card. Initially we fill the pack with a value that would print as to show an unassigned symbol. [2]

[2]Though not shown here, for printing a pack we set BASE to 72, and character ∼, which is the highest allocated ascii code, corresponds to 71 in this base. This can be verified with: `71 72 BASE ! . DECIMAL`

```
p 1+  CONSTANT N   ( no. of symbols per card )
N  N 1 - DUP * + CONSTANT #PACK ( cards in pack,
and also no. of different symbols used in pack )

CREATE PACK #PACK N * ALLOT
PACK #PACK N * 71 FILL ( fill with a value that will display as ~ )
: CARD ( n -- a, a is addr of card n in PACK ) N * PACK + ;
```

On the same space we impose a different interpretation which allows us to easily access the column vectors and sub-matrices.

```
: V ( i j -- addr ) (: i j :) PACK N +  N N 1- * i * +  j N * + ;
: M ( i j k -- addr,
      addr is address of element (j,k) in matrix i of PACK )
  (: i j k :) PACK  N 1+ +  N N 1- * i * +  j N * +  k + ;
```

We require an implementation of the Latin squares associated with the field.

```
: LAT ( n i j -- k, k is i,jth element of fields nth Latin square)
  >R *0 R> +0 ;
```

We define operations to build the pack. These fill in the first card, store the elements 0 to p onto the cards (completing the constant column vectors), fill in sub-matrix 0, copy the transpose of sub-matrix 0 to matrix 1, and complete the remaining sub matrices using their associated Latin squares.

```
: !CARD0 N 0 DO I PACK I + C! LOOP ;

: !0..p ( store symbols 0 to p onto cards )
  N 0 DO p 0 DO J   J I V C! LOOP  LOOP ;

: !M0 ( --, Set elements of matrix 0 of pack ) N
   N 1- 0 DO N 1- 0 DO
     DUP  0 J I M C!  1+
   LOOP LOOP DROP ;

: M0_Trans_toM1 ( assign transpose of M0 to M1 )
  N 1- 0 DO  N 1- 0 DO
     0 I J M C@   1 J I M C!
```

```
  LOOP LOOP ;

: LATINISE
   N 1- 1 DO
     N 1- 0 DO
       N 1- 0 DO
                    1 I J M C@   K 1+  K I J LAT  J  M  C!
                LOOP
        LOOP
    LOOP ;
```

We now have everything we need to build a pack.

```
: BUILD-PACK !CARD0 !0..p !M0 M0_Trans_toM1 LATINISE ;
```

A quickly implemented debug aid that allows us to look at the memory where the Dobble pack is being constructed is a “Dobble dump", i.e. a `DUMP` modified to use the number of symbols in the pack as its output base.

```
: DDUMP BASE @ >R #PACK BASE !  DUMP R> BASE ! ;
```

To verify our pack we first need an operation to check how many symbols are shared by a pair of cards.

```
: SHARED ( card1 card2 -- n, pre: card1 <> card2
   post: n is no of symbols shared by card1 and card2 of pack )
   (: card1 card2 :) 0
   N 0 DO
     N 0 DO card1 CARD I + C@ card2 CARD J + C@  =
       IF  1+  THEN
     LOOP
   LOOP ;
```

To check the pack we look at every pair of cards and verify that they share exactly one symbol.

```
: CHECK-PACK ( --, report first pair of cards found to differ
by anything other than one symbol )
```

```
0  #PACK 0 DO   #PACK 0 DO
  I J > IF
    I J  SHARED 1  <> IF
     CR ." Cards " I . J . ." differ in"  I J SHARED . ." symbols"
     CR ." Cards are "  I .CARD ." and "  J .CARD  ABORT
   ELSE
     ." ." 1+
   THEN
  THEN
LOOP LOOP CR . ." Checks, pack validated " ;
```

8 Conclusions

Galois fields are rich finite algebraic structures based on modular and polynomial arithmetic and are useful in many applications. Here we have described their properties and shown how they can be implemented. Mathematical properties relating to the existence of such fields and their associated sets of orthogonal Latin squares are are proved in the appendix. We have looked in detail at how the orthogonal Latin squares associated with each Galois field can be used to construct a Dobble pack. Forth proved a versatile programming tool, able to model the complexities of the Dobble pack by overlaying alternative interpretations of the data in a block of memory in a transparent and simple way. Forth's ability to work in a number of bases was made use of: we used binary for representing encodings of polynomials with modulo 2 coefficients, hexadecimal for the compact encoding higher order polynomials, and base 72 output to print the symbols on our cards. This last was useful as it allows us to print the symbols on our cards as numbers, with numbers up to 71 appearing as 71 different single output symbols. This allowed us to concentrate on the theory underlying the implementation, and this is where the beauty of this example lies, rather than in spending time making the representation of cards more attractive. The 71 ascii symbols used were perfectly sufficient to verify our code. We would note, however, that using such a high number base does not work well for input, since it produces conflicts between numbers and the names of Forth operations.

Mathematical Appendix

Modulo p arithmetic where p is not prime.

Arithmetic modulo $\mathtt{p}$ yields a Galois Field if and only if $\mathtt{p}$ is prime. To see why suppose (in order to obtain a contradiction) that modulo $\mathtt{p}$ arithmetic does yield a Galois Field $\mathcal{F}$ when $\mathtt{p}$ is non-prime. When $\mathtt{p}$ is non-prime it has factors, $\mathtt{u}$ and $\mathtt{v}$ say such that $\mathtt{u}, \mathtt{v} \in \mathcal{F}$ and $\mathtt{u} *_0 \mathtt{v} = \mathtt{p}$. Now recall that $\mathtt{u} *_0 \mathtt{v} = (\mathtt{u} * \mathtt{v}) \bmod \mathtt{p}$ and since $\mathtt{u} * \mathtt{v} = \mathtt{p}$ we have $\mathtt{u} *_0 \mathtt{v} = \mathtt{p} \bmod \mathtt{p} = 0$:

But if $\mathtt{u}, \mathtt{v} \in \mathcal{F}$ they have inverses $\mathtt{u}^{-1}$ and $\mathtt{v}^{-1}$ and so:

$(\mathtt{u} *_0 \mathtt{u}^{-1}) *_0 (\mathtt{v} *_0 \mathtt{v}^{-1}) = 1 *_0 1 = 1$

But on the other hand:, by the commutativity and associativity laws of a Galois Field:

$(\mathtt{u} *_0 \mathtt{u}^{-1}) *_0 (\mathtt{v} *_0 \mathtt{v}^{-1})$ = "by comm and ass laws"
$(\mathtt{u} *_0 \mathtt{v}) *_0 (\mathtt{u}^{-1} *_0 \mathtt{v}^{-1})$ = "since u $*_0$ v = 0"
$0 *_0 (\mathtt{u}^{-1} *_0 \mathtt{v}^{-1})$ = "by multiplicative property of 0"
0

and the contradiction is established.

Latin square properties

Given a Galois field $\mathcal{F}$ of size $\mathtt{n}$ with elements $0..\mathtt{n}-1$ there are $\mathtt{n}-1$ mutually orthogonal Latin squares of size $\mathtt{n} \times \mathtt{n}$ such that the value in row $\mathtt{j}$ and column $\mathtt{k}$ of the $\mathtt{ith}$ such square is given by $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k}$ where $\mathtt{i} \in 1..\mathtt{n}-1$ and $\mathtt{j}, \mathtt{k} \in 0..\mathtt{n}-1$.

Proof that each square is a Latin square

By closure properties $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k} \in \mathcal{F}$ which leaves us to prove that for an arbitrary square $\mathtt{i}$ the same element cannot occur twice in any row, or twice in any column.

To show an element cannot occur twice in a row, note that the element at row $\mathtt{j}$ column $\mathtt{k}$ of square $\mathtt{i}$ is $\mathtt{i} *_0 \mathtt{j} +_0 \mathtt{k}$. Assume that the element at row $\mathtt{j}'$ in the same column has the same value, and assume $\mathtt{j} \neq \mathtt{j}'$. Then

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j}' +_o \mathtt{k} \equiv$ "subtracting $\mathtt{k}$ from both sides"
$\mathtt{i} *_o \mathtt{j} = \mathtt{i} *_o \mathtt{j}' \equiv$ "dividing both sides by $\mathtt{i}$"
$\mathtt{j} = \mathtt{j}'$ "contradicting our assumption"

So the only way the element at row $\mathtt{j}$ and col $\mathtt{k}$ of an arbitrary square can be equal to the element at row $\mathtt{j}$ and col $\mathtt{k}'$ of that square is if $\mathtt{j} = \mathtt{j}'$, i.e. if they are the same square.

To show an element cannot occur twice in a column, assume that the element at row $\mathtt{j}$ and col $\mathtt{k}$ of square $\mathtt{i}$ is the same as that at row$\mathtt{j}$ and col $\mathtt{k}'$, and assume $\mathtt{k} \neq \mathtt{k}'$. Then

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j} +_o \mathtt{k}' \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}$ from both sides"
$\mathtt{k} = \mathtt{k}'$ "contradicting our assumption"

Proof that any two distinct squares are orthogonal.

For the squares $\mathtt{i}$ and $\mathtt{i}'$ to be orthogonal we need the pair of values:

$(\mathtt{i} *_o \mathtt{j} +_o \mathtt{k}, \mathtt{i}' *_o \mathtt{j} +_o \mathtt{k})$

to be distinct from

$(\mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}', \mathtt{i}' *_o \mathtt{j}' +_o \mathtt{k}')$.

To obtain a contradiction we will assume these pairs of values are not distinct, i.e.

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}'$ (1) and $\mathtt{i}' *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i}' *_o \mathtt{j}' +_o \mathtt{k}'$ (2).
Then from (1)

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} -_o \mathtt{i} *_o \mathtt{j}' -_o \mathtt{k}' = \mathtt{0} \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}' +_o \mathtt{k}'$ from each side of (1)"
$\mathtt{i} *_o (\mathtt{j} -_o \mathtt{j}') +_o \mathtt{k} -_o \mathtt{k}' = \mathtt{0}$ (3)

"also similarly from (2)"
$\mathtt{i}' *_o (\mathtt{j} -_o \mathtt{j}') +_o \mathtt{k} -_o \mathtt{k}' = \mathtt{0}$ (4)
"Now subtracting (4) from (3) and factorising"
$(\mathtt{i} -_o \mathtt{i}') *_o (\mathtt{j} -_o \mathtt{j}') = \mathtt{0} \equiv$ "since $\mathtt{i} \neq \mathtt{i}'$"
$\mathtt{j} = \mathtt{j}'$

Now substituting $\mathtt{j}$ for $\mathtt{j}'$ in (1) we obtain

$\mathtt{i} *_o \mathtt{j} +_o \mathtt{k} = \mathtt{i} *_o \mathtt{j} +_o \mathtt{k}' \equiv$ "subtracting $\mathtt{i} *_o \mathtt{j}$ from each side
$\mathtt{k} = \mathtt{k}'$

Square Roots

In section 4 we give multiplication tables for possible implementations of G(8), and it can be seen from these that every element has an exact square root.

More generally this is true for any field $G(2^n)$, and this comes from the observation that since, in modulo 2 arithmetic, addition is synonymous with subtraction, this property will also hold for polynomial arithmetic on polynomials with modulo 2 coeffiecients.

To ensure that every element of such a field has a square root it is sufficient to show that for any p, q in the field: $p^2 = q^2 \Rightarrow p = q$

Proof. We begin with the assumption that $p^2 = q^2$

$p^2 = q^2 \equiv$ "subtracting q^2 from each size"
$p^2 -_o q^2 = 0 \equiv$ "factorising"
$(p -_o q) *_o (p +_o q) = 0 \equiv$ "since $p +_o q = p -_o q$"
$(p -_o q) *_o (p -_o q) = 0 \equiv$ "property of additive inverse"
$p = q$ □

Proof of field properties

We don't attempt a mathematical proof that the polynomial arithmetic we have described yields the operations of a Galois field. However, for any particular field that we implement we can prove the axioms hold by verifying them for all combinations of values in the field. The code for this verification just needs a single parameter, p, which gives the size of the field.

For these tests we use +_ and *_ as our operation names. The names of our actual field operations might be $+_0$ and $*_0$ or $+_1$ and $*_1$, so we use these to define +_ and *_ before loading the field tests.

The most complex tests are to establish the existence and uniqueness of inverses. Here is the code for checking the multiplicative inverse axiom.

```
: (*INV) ( n -- f, pre: n ∈ 1 .. p-1
 post: f true iff n has a unique  multiplicative inverse)
 (: n :) 0 ( count of inverses )
  p 1 DO  I  n   *_   1 = IF 1+ THEN LOOP  1 = ;

: *INV-TEST  ( -- f, true if every element of 1 .. p-1 has a unique
```

```
multiplicative inverse in arithmetic modular p)
  TRUE  p 1 DO
    I  (*INV) NOT IF  DROP FALSE LEAVE THEN
  LOOP  ;
```

Given that we already know, from established mathematical results, that the field axioms will hold, the real benefit of verifying them is as a check that our field operations have been correctly implemented.

Forth Projectional Editing

Ulrich Hoffmann <uh@fh-wedel.de>

Abstract

Projectional editing is an alternative way to handle programs and data. Instead of starting with text based source code it is centered around internal program/data structures and so called projections create editable representations that allow to modify the internal structures. In the Forth context memory seems to be the appropriate internal data structure. Different editors interpret memory content in specific ways and allow the user to modify it in an appropriate fashion. A hex and a stack editor are described and other editors are proposed. The idea of Forth projectional editing gives a general view to program and data handling that allows to classify techniques used in different Forth systems.

1 Introduction

Projectional editing [1] is an approach to edit programs in a programming language that does not rely on manipulating source code and then scan, parse, translate it to object code. Instead a projectional editor presents a pleasantly *editable representation* of some internal structure (often an abstract syntax tree). Editing this representation results in appropriate modification of the internal data structure, see figure 1. Instead of source code, the internal *abstract representation* is the original artifact and all other representations are produced from it: To persist a storage representation, to run an executable representation is generated. For editing appropriate editable representations are created.

The process of building editable representations from internal data is called *projection* as it might only extract some important aspects of the internal data structure while leaving others untouched. Different projections for the same structure can exist, allowing to edit it in different ways. As an example, a decision table could be edited in a textual representation as an array initialization or it could be presented to the user in tabular form visualizing a decision table similar to those used in a design document.

Projectional editing can be used with great benefit when using domain specific languages

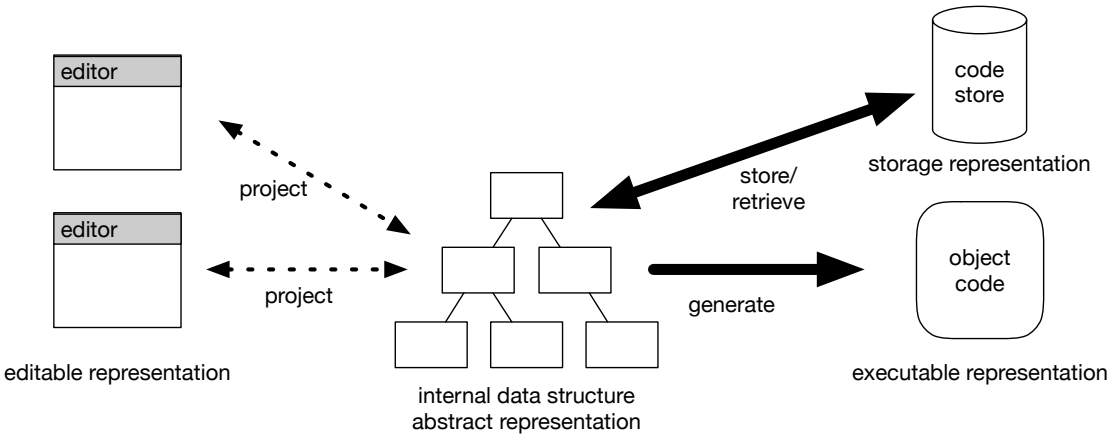


Figure 1: Projectional Editing

(DSLs). They often deal with specific aspects of a problem domain that might have special visual representations. Projectional editing can provide familiar visualization for (parts of) the DSL.

2 Forth Projectional Editing

Forth is especially strong when creating DSLs. Its approach however is not the traditional one building an abstract syntax tree for the internal representation of programs and generate code from this. Instead code generation — even native code generation — typically takes place in a single pass directly starting from Forth source code. This is to support the Forth's interactive nature.

In order to apply projectional editing ideas to Forth a suitable internal representation has to be identified. As Forth is strongly memory oriented (`@ ! , ALLOT MOVE` ...) it seems to be reasonable to use memory as the internal data structure and start different projections from there. Forth programmers are used to take some memory area, represent their data structures in this memory and later on interpret (project) the memory area in this specific way by using only suitable operators on it.[1]

The classical Forth block editor already interprets memory in a specific way: `BLOCK` returns a memory address of a typically 1 KB large memory area that is interpreted as 16 lines of 64 characters without line breaks. Forth programmers edit their source code in these blocks using the block editor and the system interprets and compiles that program by means of the `LOAD` operator. (The block/buffer subsystem transparently handles storing and retrieving blocks.)

2.1 Hex editor

The first projection to look at is a very general one that interprets data in memory as just bytes. This leads to a hex editor. This editor is similar to the `DUMP` utility that can be found in many systems but in addition memory is not just displayed but the hex dump becomes editable. The hex editor and dump utility have the same interface `( c-addr u -- )` and the editor is just a Forth word that can be invoked both interactively and also from within a running program. It returns to the calling word when the editor is exited. Figure 2 shows a sample hex editor session: The editor view is split into two main parts: the hex dump and the character dump. The TAB key switches between the two. The cursor is initially placed in the hex dump and can move by means of the cursor keys. Appropriate data can be entered and directly modifies the represented memory.

A binary editor for Forth blocks can simply be defined as

```
: hedit ( u -- )
    block 1024 hex-edit update ;
```

[1] Type safety is assured by the programmer not the compiler.

```
$ sf hexedit.fs

( hex-editor loaded. Usage: c-addr u HEX-EDIT  ) ok

Create conference  'E' c,  'u' c,  'r' c,  'o' c,  'F' c,  'o' c,  'r' c,  't' c,  'h' c,
conference 30 hex-edit

00003CB44  45 75 72 6F 46 6F 72 74 68 08 68 65 78 2D 65 64   EuroForth.hex-ed
00003CB54  69 74 63 65 2A 00 0F 00 4F 14 00 00 2A 00         itce*...O...*.
```

Figure 2: A sample hex-editor session

If a Forth system implements its stacks in main memory then the stacks also become editable using the hex editor, e.g.:

```
10 20 30 40  sp@ 4 cells hex-edit
```

will start the hex editor on the top most 4 stack items:

```
0BFFFFA80 28 00 00 00 1E 00 00 00 14
00 00 00 0A 00 00 00 (.............
```

The exact layout of the stack in memory is of course system specific.

For this however a different projection of memory might be more appropriate. This leads to the stack editor.

2.2 Stack Editor

The stack editor (sample session in figure 3) displays an editable stack representation with each stack item on a line of its own and allows to interactively modify the stack. Each item ist shown in its character, unsigned hexadecimal, unsigned decimal and signed decimal representation. The stack editor items can be cut/copied and pasted (Ctrl-X, -C, -V) or replaced by items that are the result of a Forth fragment entered on the items line.

Like the hex editor also the stack editor is just a word that can be invoked when appropriate and resumes execution of the caller when exited. So it can be inserted in source where appropriate and be used as an interactive alternative for `.S` debugging.

2.3 Other editors

Adopting the idea of Forth Projectional Editing (i.e. projections from memory to editable representations) in other areas would enable a large selection of possible editors. Examples might be:

- **A Variable Editor**
  This could represent a `VARIABLE` defined word similar to the stack editor in different ways and interactively allow for appropriate changes. Depending on the programmer intended variable type different representations might be reasonable. For example a flag might be shown as a toggle that can be flipped or enumeration data could show and allow to select one of the possible values.
- **A User Area Editor**
  In a multi tasking system the user area is a collection of task specific variables. The idea of the variable editor could be extended to editing the entire user area along with the user variable name and its content.
- **A Structure Editor**
  While a variable editor would allow for editing just a single cell, a structure editor could project an editable representation of an entire structure defined by `BEGIN-STRUCTURE`, `FIELD:`, etc. By introspection it could display the field names and provide appropriate variable editors for each of the fields in the structure.

```
> gforth stackedit.fs
10 20 30 40 50 -1 stack-edit

  0: '?' $FFFFFFFFFFFFFFFF  #18446744073709551615                  -1
  1: '2'               $32                    #50                  50
  2: '('               $28                    #40                  40  42
  3: '.'              $1E                     #30                  30
  4: '.'              $14                     #20                  20
  5: '.'               $A                     #10                  10

up/down: select line   DEL   Ctrl-X, -C , -V    Forth words leaving one item
```

Figure 3: A sample stack-editor session

- **A Wordlist Editor or Dictionary Editor**
  Forth systems typically use a system specific way to organize their dictionary. A projectional editor for the dictionary or a single word list would allow to manipulate the dictionary, i.e. change the order of definitions, adapt spelling or word names, change immediacy of words, possible removing definitions, and others. This way the dictionary becomes the central internal data structure. Also the search-order could be subject of a Search Order Editor.

- **A Word Definition Editor**
  A Forth decompiler typically recreates Forth source code from the memory representation of a definition in the Forth dictionary. A Word Definition Editor could based on a decompiler project a definition in dictionary to an editable representation in source code (tokens or text) and so allow for changing definitions directly in the dictionary.

- **A Screen Editor with line and screen terminators**
  As mentioned before the traditional way to represent Forth source code in memory is 16 by 64 characters in blocks. Source code could be represented in different ways with handling of line terminators (and possible screen terminators) and a screen editor would perform the appropriate projection to user editable source code.

Editors for other data structures seem very well to be possible. The main idea here is to see all of theses editors as a projection from their memory representation to an appropriate editable representation.

3 Related work

Since Martin Fowler's blog article [1] in 2008 some systems have been developed around projectional editing for DSLs. Most prominent is Jetbrain's Meta Programming Systems (MPS) [2] that allows for defining domain specific languages along with appropriate projectional editors. It also supports projectional editing for (parts of) contemporary languages (Java, Javascript, XML, C, ...).

In Forth context the Jupiter Ace [3] home computer includes a Word Definition Editor and follows the *code is the source* paradigm.

The ForthOS [4] system uses 4 KB screens with a source representation of 80 x 25. The Enth [5] system has a screen editor (CodeEd) that handles line terminated source code in fixed sized 1 KB blocks.

HolonForth [6] stores word definitions along with meta data in a database and includes an integrated editing environment that structures projects in a hierarchical way. For editing source code is projected to a full screen editor. Machine code is generated from the internal data base representation.

ColorForth tokenizes word names on entry to 32 bit items and stores these in screens. The colorForth editor generates an editable representation of this token sequence and allow for interactive manipulation of the tokenized source code. The colorForth compiler used the token sequence to generate machine code.

4 Conclusion and future work

Projectional editing gives a different view of programming language editing based on internal abstract representations. It opens additional possibilities for handling programs.

Although Forth systems have never explicitly used projectional editing its ideas are well present in several Forth systems and the idea of projecting memory to specific editors has many interesting applications that complement Forths interactive nature.

Hex- and Stack editors have been implemented as Forth-200x standard programs. Other edi-

tors as mentioned in section 2.3 seem not to be more difficult to implement but some of them probably need system specific details. Among the proposed editors a Structure Editor seems to be the most useful. One can also envision mixed editors that use different editors for different parts of memory.
Is the map the territory? You decide.

> Forth is stacks, words, and blocks; start there.
>
> Jeff Fox [8]

References

[1] *ProjectionalEditing*, M. Fowler, `martinfowler.com/bliki/ProjectionalEditing.html`, 2008
[2] *Meta Programming System*, Jetbrains, `jetbrains.com/mps`
[3] *Jupiter Ace*, Wikipedia, `en.wikipedia.org/wiki/Jupiter_Ace`
[4] *ForthOS*, Wikipedia, `sources.vsta.org/forthos/`
[5] *Enth Flux aha colorForth*, Sean Pringle, `www.ultratechnology.com/enthflux.htm`, 2000
[6] *Holonforth*, Wolf Wejgaard, `holonforth.com`
[7] *ColorForth*, Charles Moore, `colorforth.github.io`
[8] *Forth is stacks, words, blocks*, Jeff Fox, `www.ultratechnology.com/forth2.htm`

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8063 1441
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

The VFX Forthv5.1 kernel incorporates dual-behaviour words and recognisers. This paper discusses our experience over the last year with these changes. Dual-behaviour words are a standards-compliant solution to needing words that have separate interpretation and compilation behaviour. Previous papers called these words NDCS words (non-default compilation semantics). Recognisers are a fashionable solution to providing a user-extensible text interpreter. Our experience converting two OOP packages to use recognisers is discussed.

Introduction

VFX Forth v5.1 was the first VFX version to provide dual-behaviour words and recognisers. An application of 1.34 million lines of Forth source code was converted to VFX 5.1 in 3.5 days and ran first time.

We adopted dual-behaviour words because they fix the problems previously solved by state-smart words. As implemented in VFX, dual-behaviour words are standard-compliant. The vast majority of the application conversion involved rewriting state-smart string-defining words. The application is a commercial one, and uses a large number of string types.

We adopted recognisers because they are fashionable and have one technical possibility that is important in large applications. It is currently impossible to persuade Forth programmers to use just one OOP package. Thus, if we are to reuse library code, we must learn how to manage multiple OOP packages. Recognisers provide a partial solution to this problem, but considerable attention to wordlist and naming is also required.

Dual-behaviour words

The classical Forth interpreter loop below is replaced by a new loop whose essential change is to distinguish between words with dual behaviour, defined in the ANS and Forth-2012 standards as “Non-Default Compilation Semantics” (NDCS for short). The term NDCS was liked by nobody, and words with this behaviour are now referred to as dual-behaviour words.

Traditional FIG-Forth interpreter

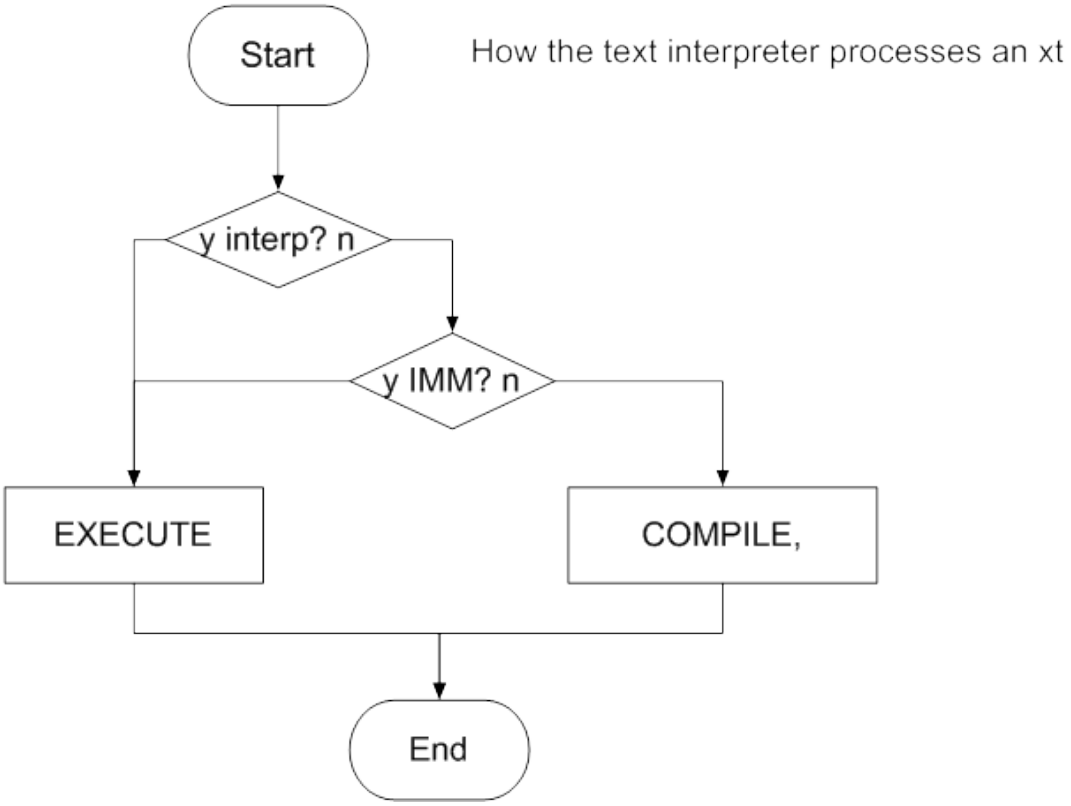
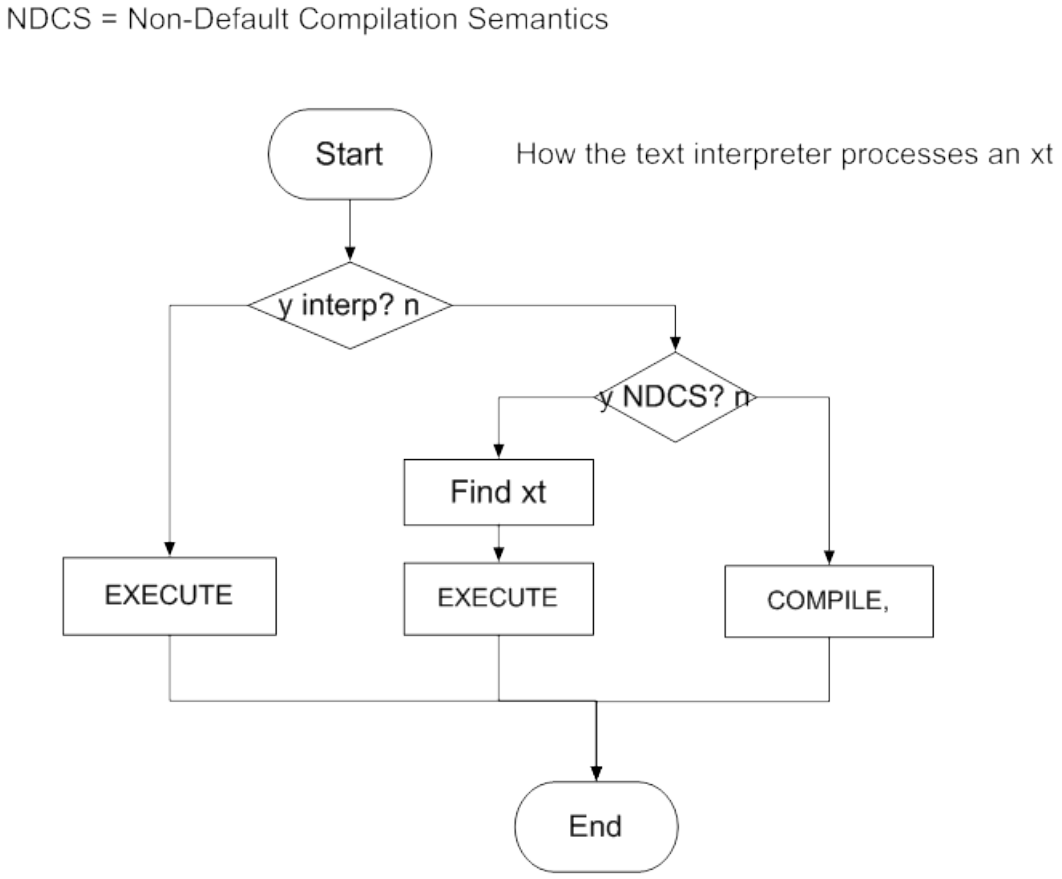


Illustration 1: Classical Forth interpreter loop

```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup immediate?
    if  execute  else  compile,  then
  then
;
```

The classical Forth interpreter loop has been used to describe the operation of Forth for over three decades now. It has been a useful model for many people. People regularly claim that they need to write a custom interpreter and that not all Forth systems permit this in a portable manner. We will see that a minor change to the loop and its associated structures brings it in line with Forth 2012 and expands the interpreter's facilities to take advantage of the Forth 2012 description of Forth words' action or behaviour or semantics. What we now call DUAL words are words such as **`IF`** that have separate interpretation and compilation behaviour, referred to in the standard as "non-default compilation semantics".

Dual-behaviour loop




```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup immediate? if
      execute
    else
      dual?
      if  dual,  else  compile,  then
  then
;
```

The picture illustrates a Forth interpreter/compiler loop that has been modified to cope with separated interpretation and compilation actions.

We also need a small number of new words that enable the loop to be constructed portably:
`IMMEDIATE?  Xt -- flag` ; return true if the word is immediate
`DUAL?  Xt -- flag` ; return true if the word has non-default compilation semantics
`DUAL,  i*x xt -- j*x` ; like **`COMPILE,`** but may parse.

In order to finish up, we need to understand what the word labelled DUAL`,` actually does. It finds the word that performs the non-default compilation semantics and then **`EXECUTE`**s it. The next picture shows the loop using the definition of **`IMMEDIATE`** words as having the same interpretation and compilation semantics.

The significant change is the introduction of a dictionary header flag, DUAL, which indicates that a word has non-default compilation semantics.

Since IMMEDIATE words are dual-behaviour words by definition, the interpreter processing of an xt can in theory be reduced to the one below

```
: process-xt    \ i*x xt -- j*x
  state @ 0 = if
    execute
  else
    dup dual?
    if  dual,  else  compile,  then
  then
;
```

The immediate flag has disappeared because all immediate words have non-default compilation semantics. They are immediate if the DUAL xt is the same as the for interpretation xt. The definition of immediate is more complicated in standards-speak, but comes to the same thing. An alternative implementation strategy may be to keep a separate immediate flag, but we should not hide the basic idea that immediate words have non-default compilation semantics.

The conventional immediate flag in a word's header becomes the DUAL flag, set for all words that have non-default compilation semantics. Comparison of the interpretation xt and the DUAL xt gives us a basis for the word **`IMMEDIATE?`** The word **`DUAL,`** just hides the system-specific action of obtaining the DUAL action from an xt.

Here's a potential way of building DUAL words. They illustrate a conventional **`IF`** … **`THEN`** pair. The word **`DUAL:`** modifies the previous word to have the following non-default compilation semantics – it defines a nameless word and sets system-specific flags and data.

```
: IF      \ C: -- orig ; Run: x --
\ This is the traditional interpretation behaviour
  NoInterp  ;
dual: ( -- orig )  s_?br>,  ;     \ conditional forward branch

: THEN     \ C: orig -- ; Run: --
\ This is the traditional interpretation behaviour
  NoInterp  ;
dual: ( orig -- )  s_res_br>,  ;  \ resolve forward branch
```

To produce an interpreted version, the interpretation behaviour is simply replaced by the new version. The next example shows how a contentious notation such as **`S"`** and friends becomes non-contentious.

```
: S"            \ Comp: "ccc<quote>" -- ; Run: -- c-addr u
\ Describe a string. Text is taken up to the next double-quote
\ character. The address and length of the string are
\ returned.
  [char] " parse >syspad
;
dual: ( -- ) postpone (s") ",  ;
```

Dual-behaviour words are discussed in more detail in my EuroForth 2018 paper "Implementing DUAL words"

Experience

While building the VFX kernel and system, we had few problems because we knew what we were looking for. When recompiling the CCS application, the majority of the conversion effort went into converting the many, many string-type handling words to the new dual-behaviour format. None of it was difficult, only tedious.

Recognisers

Recognisers are currently being promoted as a way to provide a user-extensible and user-definable text interpreter. While this is a worthy goal, it isn't enough on its own for systems that already provide a good set of facilities. However, realising that software development tools are not part of the tech industry but of the fashion industry, we bowed to the wind. The recogniser proposals are fluid and not enough people have implemented a heavy application using them.

Instead of treating an interpreter as a tool for finding words, numbers and undefined actions, recognisers provide a list or table or chain of parsers that identify a particular type of element. Once the type has been identified, type data is passed to one of three processing elements for interpretation, compilation or postponing.

The parser may return more than one form of type data. For example, a word recogniser can return separate type data for normal, immediate and dual-behaviour words. In the example below the words starting with **`r:`** are three-element type tables holding the interpretation, compilation and postpone xts.

```
: rec-find \ addr u -- xt r:word | r:fail
\ *G Searches a word in the search order (wordlist stack).
\ ** The xref utility code is contained inside the dictionary
\ ** search code.
  search-context dup 0= if
    drop r:fail
  else
    0< if                   \ -- xt
      dup ndcs?
      if  r:ndcs  else  r:word  then
    else
      r:immediate
    then
  then ;
```

Similarly, you can provide one or more parsers for numeric literals. It's a matter of taste and existing code. If the list of parsers is made extensible, additional word types and literal types can be added at will, and just as importantly, can be removed at will. This facility, together with disciplined use of wordlists and vocabularies is important to enable us to cope with multiple OOP package.

The various recogniser proposals can be found at:

http://amforth.sourceforge.net/pr/Recognizer-rfc-D.pdf
http://amforth.sourceforge.net/pr/Recognizer-rfc-C.pdf
http://amforth.sourceforge.net/pr/Recognizer-rfc-B.pdf
http://amforth.sourceforge.net/pr/Recognizer-rfc.pdf

Experience

In the build of VFX Forth, the only issue was the partitioning of the returned type data. Because we already had an integrated integer handler, we made no distinction between integer types. However, the various floating point packages are installed separately.

We then converted two of the OOP packages supported by MPE. CIAO (C Inspired Active Objects) was the MPE OOP package designed to ease integration with Windows and C++. ClassVFX is the OOP package used by Construction Computer Software in their Candy application:
https://constructioncomputersoftware.com/solutions/solution-candy
The Candy application consists of 1.34 million lines of Forth source code.

The OOP ports revealed that I do not yet understand how to apply **POSTPONE** actions to the result of the dotted notations used by both packages. The most recent set of proposals RFC-D above proposes that the postpone action can be formalised so that the same action can be used by all **POSTPONE** actions. It is just too early to believe that all compound parsers will be able to work this way. One claim for this approach is that it saves memory. We tested this in the VFX kernel and found that the unified action implementation saved 50 bytes in a system of 250k bytes and more. With base-level desktop systems starting with 1Gb of RAM, saving 50 bytes is not a good rationale for standardisation of implementation. We have not yet moved recognisers into our embedded kernel.

The latest recogniser proposal depends on a “stack” proposal that crept in late. That proposal is inadequate for MPE’s OOP requirements as we need to add new parsers at both ends of the tables.

In our recogniser experience to date, recognisers have caused no problems at all and have enabled us to remove a hook or two.

Conclusions

Dual-behaviour words have caused us no problems except for finally having to get rid of any state-smart words in all applications. We have found the notation described in past papers to be easy to use and understand.

Recognisers do enable user-extension of the text interpreter, but the proposals are not yet ready for standardisation. The main reason for this is that although a considerable number of systems have implemented what has been in the proposals, very few systems have pushed the boundaries beyond words and numeric literals. A few string literal proposals have been made, but these little more than handling address and length pairs.

Acknowledgements

Anton Ertl has tested my understanding of Forth standards for many years.

Bernd Paysan has confirmed my belief that the ingenuity of Forth programmers to break common belief must never be underestimated.

My belief that all standards contain bugs has sustained me over many years.

Construction Computer Software encouraged and sponsored the Community edition of VFX v5.

UI5

a robust HTML5-based user interface for VFX5

Gerald Wodni
gerald.wodni@gmail.com

13.09.2019

Outlook

History

- **2013** FATE "Forth Advanced Template Engine"
- **2014** Flink "Forth Link"
- **2016** f - packages
- **2018** redis

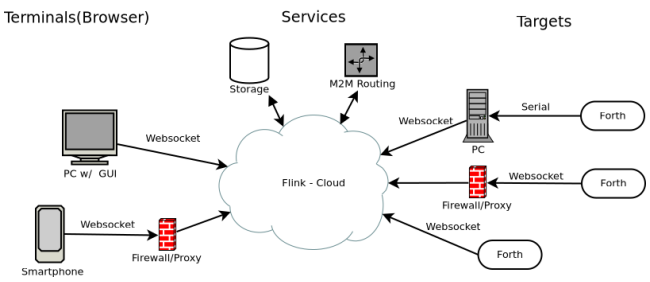
History

- **2013** FATE "Forth Advanced Template Engine"
- **2014** Flink "Forth Link"
- **2016** f - packages
- **2018** redis
- **2019** UI5 ← *you are here*

History

- **2013** FATE "Forth Advanced Template Engine"

Flink



History

- **2013** FATE "Forth Advanced Template Engine"
- **2014** Flink "Forth Link"

Forth on a Server

Highly customizeable compiler with bare metal access

- Protect host from Forth
- Secure other instances

History

- **2013** FATE "Forth Advanced Template Engine"
- **2014** Flink "Forth Link"
- **2016** f - packages

Forth on a Server

Highly customizeable compiler with bare metal access

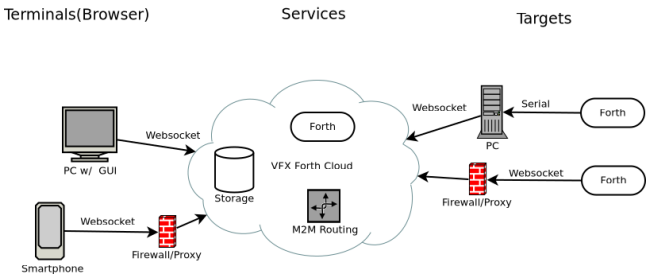
- Protect host from Forth ✗
- Secure other instances

Forth on a Server

Highly customizeable compiler with bare metal access

- Protect host from Forth ✗
- Secure other instances ✗

VFX Forth Cloud



Forth ~~on a Server~~ in a container

Highly customizeable compiler with bare metal access ~~on a server~~ in a container

- Protect host from Forth
- Secure other instances

HTML5

- **HTML** Elements
- **CSS** Styling
- **JS** Interaction

Forth ~~on a Server~~ in a container

Highly customizeable compiler with bare metal access ~~on a server~~ in a container

- Protect host from Forth ✓
- Secure other instances

index.html

```
<main>
    <section id="terminal">
        <vfx-terminal></vfx-terminal>
    </section>
    <section id="help">
        <h2>Help</h2>
        <p>
            There is help, that needs to be found ;)
        </p>
    </section>
</main>
```

Forth ~~on a Server~~ in a container

Highly customizeable compiler with bare metal access ~~on a server~~ in a container

- Protect host from Forth ✓
- Secure other instances ✓

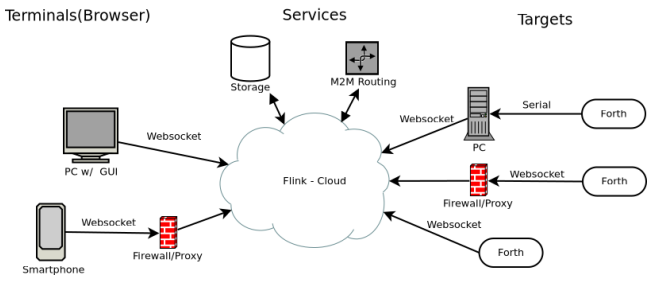
ui5.css - theming

```
:root {
    --main-color: #333;
    --main-background: #FFF;
}

body.theme-night {
    --main-color: #FFF;
    --main-background: #333;
}

main {
    color: var(--main-color);
    background: var(--main-background);
}
```

Original Flink



ui5.css - scaffolding

```
/* +-----------------+
 * |        | HEADER |
 * |        +--------+
 * | ASIDE  | CONTENT |
 * |        +--------+
 * |        | FOOTER |
 * +--------+--------+
 */
body {
    grid-template-columns: var(--aside-width) 1fr;
    grid-template-rows: var(--header-height) 1fr var(--footer-height);
    grid-template-areas: /* note how this text documents the layout */
        'aside header'
        'aside main'
        'aside footer';
}
```

js/main.js - modules

```
/* navigation by hash */
import './vfx-navigation.js';

/* websocket connector */
import './vfx-connector.js';

/* custom elements */
import './vfx-terminal.js';

/* application specific */
import './night-theme.js';
```

app.fth - websocket

```
: app-ws
  s" 101 Switching Protocols" http-status
  s" Upgrade: websocket" http-header
  s" Connection: Upgrade" http-header
  SVIdata Get-Sec-WebSocket-Accept
    s" Sec-WebSocket-Accept" http-header-value
  s" Sec-WebSocket-Protocol: ui5" http-header
  crlf

  mysid gen-handle @ mywss open-gio \ open
  [io mywss setio
    (.cold)
    ['] websocket-quit catch >r
    s0 @ sp! r>
  io] ;
' app-ws add-route-get /ws
```

js/night-theme.js

```
/* before interacting with the DOM wait for it to be fully parsed */
document.addEventListener( "DOMContentLoaded", () => {

  /* querySelector works by getting the first match of a CSS path */
  document.querySelector( "header button[name='night-mode']" )
  .addEventListener( "click", (evt) => {

    /* toggles theme-night class, all else is done by CSS */
    document.body.classList.toggle("theme-night");

  });

});
```

Demo

http://cloud.vfxforth.com

js/vfx-terminal.js

```
const template = document.createElement('template');
template.innerHTML = `
  <style>
    .terminalWrapper .input { color: var( --input-color, #FFF ); }
  </style>
  <div class="terminalWrapper">
    VFX Terminal <button>Connect</button>
    <input name="source" type="text" autofocus/>
  </div>`;
class VfxTerminal extends HTMLElement {
  constructor() {
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.appendChild( template.content.cloneNode( true ) );
    shadow.querySelector("button").addEventListener( "click", ... );
  }
  write( text ) { this.source.insertAdjacentHTML('beforebegin', text ); }
}
customElements.define( 'vfx-terminal', VfxTerminal );
```

Future

- Provide more than the full VFX-"Window"-Experience in UI5
- Migrate AIDE to UI5

app.fth - routing

```
: app-home
    s" %APP%/index.html" s" text/html;encoding=utf8" http-file-type ;
' app-home add-route-get /

: app-css
    s" %APP%/ui5.css" s" text/css" http-file-type ;
' app-css add-route-get /ui5.css

: app-favicon
    s" %APP%/favicon.ico" s" image/x-icon" http-file-type ;
' app-favicon add-route-get /favicon.ico
```

Participate!

app.fth - chunked

```
: app-chunked
    s" 200 OK" http-status
    s" Transfer-Encoding: chunked" http-header
    s" Content-Type: text/plain;encoding=utf8" http-header
    crlf
    s\" Hallo du\n" http-chunk
    s" Wie geht's?" http-chunk
    http-chunk-end ;
' app-chunked add-route-get /chunked
```

CloudCalypse

building a social network on top of net2o,
and importing your existing data

ἀποκάλυψις ■ uncovering
cloud[o]calypse ■ σύννεφο καταστροφή

Bernd Paysan

EuroForth 2019, Hamburg

Photo: Ralph W. Lehmkecht

Centralized/Federated/P2P?

Centralized
- \+ good funding, robust hardware and attack protection
- – lacks privacy, honeypot, captive, EOL at whim of CEO
- – diverse global censorship, possible toxic business model

Federated
- \+ not captive, small business models
- ± regional censorship (nodes blacklisted, e.g. Lolicon Mastodon nodes)
- – poor funding, underpowered hardware/attack protection
- – lacks privacy, EOL of nodes at whim of node admin

Peer2Peer
- \+ Full control over your node, good privacy
- \+ Development funding? Otherwise cheap
- ± non–existend censorship (attracts censorship refugees)
- – Full responsibility for your node

Motivation

Right to data portability

Art. 20 GDPR

1. The data subject shall have the right to receive the personal data concerning him or her, which he or she has provided to a controller, in a structured, commonly used and machine-readable format and have the right to transmit those data to another controller without hindrance from the controller to which the personal data have been provided, where:
   (a) the processing is based on consent pursuant to point (a) of Article 6(1) or point (a) of Article 9(2) or on a contract pursuant to point (b) of Article 6(1); and
   (b) the processing is carried out by automated means.
2. In exercising his or her right to data portability pursuant to paragraph 1, the data subject shall have the right to have the personal data transmitted directly from one controller to another, where technically feasible.
3. The exercise of the right referred to in paragraph 1 of this Article shall be without prejudice to Article 17. That right shall not apply to processing necessary for the performance of a task carried out in the public interest or in the exercise of official authority vested in the controller.
4. The right referred to in paragraph 1 shall not adversely affect the rights and freedoms of others.

6 Years after Snowden

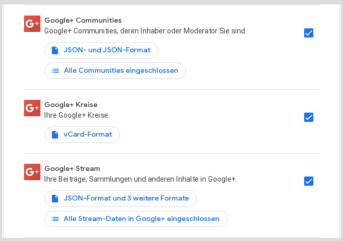
What changed?

Politics
- EU parliament wants upload filters
- EU parliament taxes the link (instead: "right")
- EU parliament wants filtering "terrorist contents"
- Germany wants a Cyberadministration like CAC (Medienstaatsvertrag)
- Tor– and VPN–ban planned
- Seehofer plans reveal–your–password law
- And he wants Alexa's "tapes" & disable E2E encryption [2]

Progress
- The ECHR ruled that GCHQ's dragnet surveillances violates your rights
- net2o becomes more and more usable

Google+ JSON Takeout

https://takeout.google.com/settings/takeout



Cloud[o]Calypse

something went terminally wrong in a cloud [3]

Clouds failing
- Microsoft bought github (install gitlab-ee)
- Dropbox dropped Linux client (except ext4 unencrypted)
- Facebook: Cambridge Analytica scandal+many API holes [4]
- Google+ closed due to possible leak (both users affected)
- tumblr deleted all porn (remains: 1% contents)
- Can't date on Tinder, dating violates Facebook's policy

Root causes
- toxic ad–based revenue model
- user+password authentication
- your data is on someone else's computer

Google+ JSON Takeout

```
~/Downloads/Takeout/Stream in Google+/Beiträge> cat '20181101 - +++ #net2o Import von
Google+_ Avatare_.json'
{
  "url": "https://plus.google.com/+BerndPaysan/posts/P6CiHfAJgpy",
  "creationTime": "2018-11-01 17:51:40+0100",
  "updateTime": "2018-11-01 17:51:40+0100",
  "author": {
    "displayName": "Bernd Paysan",
    "profilePageUrl": "https://plus.google.com/+BerndPaysan",

"avatarImageUrl": "https://lh3.googleusercontent.com/a-/AN66SAyasgoOyZqe
kQqoDpoFmrBKAll3N1-jLFUel43iAg\u003ds64-c",
    "resourceName": "users/114020517704693241828"
  },

"content": "\u003cb\u003e+++ \u003c/b\u003e\u003cb\u003e\u003ca rel\u003d\"nofollow\
class\u003d\"ot-hashtag bidi_isolate\" href\u003d\"https://plus.google.com/s/%23net2o/posts
```

Ad–based business = toxic?

- Incentive to keep you on the site
- Best way to keep you: controversial discussion
- 👎 (no dislikes to force you to comment if you don't agree)
- Incentive to make you easily manipulated
- Worst "fake news" are indeed the ads themselves
- Incentive to gather all kinds of information to target ads to you
- Make you post about your private things
- Incentive for participants to do influencer marketing

Beware: applies to journals and TV, too

Google+ JSON Takeout

```
~/Downloads/Takeout/Stream in Google+/Beiträge> cat '20181101 - +++ #net2o Import von
Google+_ Avatare_.json'
{
  "url": "https://plus.google.com/+BerndPaysan/posts/P6CiHfAJgpy",
  "creationTime": "2018-11-01 17:51:40+0100",
  "updateTime": "2018-11-01 17:51:40+0100",
  "author": {
    "displayName": "Bernd Paysan",
    "profilePageUrl": "https://plus.google.com/+BerndPaysan",

"avatarImageUrl": "https://lh3.googleusercontent.com/a-/AN66SAyasgoOyZqe
kQqoDpoFmrBKAll3N1-jLFUel43iAg=s64-c",
    "resourceName": "users/114020517704693241828"
  },

"content": "<b>+++ </b><b><a rel=\"nofollow\" class=\"ot-hashtag bidi_isolate\" href=
"https://plus.google.com/s/%23net2o/posts\" >#net2o</a></b><b> Import von Google+: Avatar
```

Facebook JSON takeout

```
~/Downloads/Facebook/posts> cat your_posts.json
{
  "status_updates": [
    {
      "timestamp": 1539297571,
      "attachments": [
        {
          "data": [
            {
              "media": {
"uri": "photos_and_videos/videos/10000000_1829816733782306_
2429950629012045824_n_10215835485911416.mp4",
                "creation_timestamp": 1539297649,
                "media_metadata": {
                  "video_metadata": {
                    "upload_timestamp": 0,
```

Twitter JSON takeout

```
~/Downloads/Twitter> cat tweet.js
window.YTD.tweet.part0 = [ {
  "retweeted" : false,
"source" : "<a href=\"https://mobile.twitter.com\" rel=\"nofollow\">Twitter Lite</a>",
  "entities" : {
    "hashtags" : [ ],
    "symbols" : [ ],
    "user_mentions" : [ {
      "name" : "daimbag101",
      "screen_name" : "marco_keule",
      "indices" : [ "0", "12" ],
      "id_str" : "3353806857",
      "id" : "3353806857"
    }, {
      "name" : "Karl Lauterbach",
      "screen_name" : "Karl_Lauterbach",
```

Blogger Atom feed takeout

```
~/Downloads/Takeout/Blogger/Blogs/Bernds Blog> cat feed.atom
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xmlns:blogger='http://schemas.google.com/blog
2018'>
  <id>tag:blogger.com,1999:blog-40816824579095739 2</id>
  <title>Bernds Blog</title>
  <entry>
<id>tag:blogger.com,1999:blog-408168245790957392.post-9403809673276532 6</id>
    <blogger:type>POST</blogger:type>
    <blogger:status>LIVE</blogger:status>
    <author>
      <name>Bernd</name>
      <uri>https://plus.google.com/114020517704693241828</uri>
      <blogger:type>BLOGGER</blogger:type>
    </author>
    <title>Nach Suzhou</title>
```

Things needed for import

- JSON parser, XML parser, HTML parser
- JSON/XML schemas for all those exports
- HTML to Markdown converter
- Downloader for missing parts (e.g. avatar photos)
- Temporary secret keys for all those other authors

Social Networks in net2o

Texts	as markdown
Images	JPEG, PNG
Movies	mkv/webm
Timeline	Chat log with link to DVCS project
Posting	DVCS project, keeping data+comments together
DVCS project	Chat log with link to patchsets/snapshots
Reshare	Fork+added posting+log message in own timeline
Comment	Fork+added posting+pull request
Likes	Chat log messages directly in DVCS project

Status

- \+ Bulk importer for Google+
- – Bulk importers for Facebook/Twitter/Blogger/etc.
- \+ Use avatars to display users's ID
- \+ Markdown renderer
- – Album viewer
- – Movie player
- – Key handover to contact in net2o world (temporary keypair)
- \+ Mark imported keys as not trustworthy

The non–technical problems

- Get your contacts over to net2o
- How to make a social network a nice place?
- Funding of net2o?

net2o: things to do

The incomplete list of incomplete things:

1. Shorter connection setup with NAT traversal
2. Two–stage DHT to separate identity from queries
3. Complete the payment system (+ add support for ticket systems)
4. GUI–only operation must be possible
5. Markdown to presentation converter

Literatur & Links

Bernd Paysan	*net2o fossil repository* https://net2o.de/
Stefan Krempl	*Innenminister Seehofer wünscht sich digitale Wanzen im Wohnzimmer* https://heise.de/-4449720
Information is beautiful	*World's Biggest Data Breaches & Hacks* https://informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/
Marvin Strathmann	*Hallo Mark, viel Spaß mit Deinen Erinnerungen auf Facebook 2018* https://heise.de/-4254681