Compatible Structures in ANS Forth

David N. Williams

Version 1.0.1
July, 2000

Abstract

We present a Structure word set for ANS Forth, and provide a portable implementation in the file `cstruct.fs`. A major aim is compatibility with structures in C libraries. The idea for making structures nestable with reusable field labels is taken from Randolph Peters' Pocket Forth implementation [1], and we implement some early binding ideas stressed in a Forth Scientific Library implementation [2]. We implement both left and right syntaxes for structure data references.

Contents

1 Introduction

Not only does Forth lend itself to a variety of structure schemes, but simple schemes can be very effective for the application at hand. The elegant implementation in Anton Ertl's Gray parser generator [3] is a good example of this occasional approach, where the implementation is precisely suited to its use.* More elaborate schemes, such as that in the Forth Scientific Library (FSL) project [2] and those in this document, can be both ANS Forth portable and reasonably efficient at runtime.

Taken as a whole, the Structure word set in this document is not simple. It is designed to be compatible with C structures, with the aim of supporting the use of C/POSIX libraries. Actually, the discussion is more elaborate than the code, and the code would be a good deal simpler if it did not cover bit-fields. If stripped to Peters' elegant core design, it becomes fairly simple and effective for Forth-centric applications.

We attempt to control the complications a bit in the implementation file `cstruct.fs` by dividing it into sections, with flags for conditional compilation of various features.

*According to Ertl, inspired by earlier work possibly by John Hayes.

Being compatible with C structures does not mean that the syntax has to be derived from C, but it does fix the layout of structure data. For example, the POSIX standard function `localtime()` converts the output of the POSIX time function into year, month, day, hours, minutes, etc., and actually stores into a `struct tm`. If this function is called from Forth (glue for that is not discussed here), the Forth structure words have to know how to build and access a standard `struct tm`.

This is a bit of a problem, because standard C programs [4, 6.5.2.1] are not allowed to assume much about the packing or alignment of the memory layout of structures, although the ordering of structure elements in memory has to be the same as in the structure declaration [4, 6.5.2.1], [5, p. 213].†

The approach we attempt for layout is based on that of the GNU CC compiler [7], which uses implementation-dependent macro expressions and constants to parametrize the underlying system. As far as we can see, even the major system not (yet) covered by GNU, namely Mac OS, can be handled this way. The GNU source indicates that their scheme also covers other languages.

†The ordering of structure elements in particular structure *declarations* in standard C/POSIX libraries is however *not* prescribed [6, p. 551]; a POSIX standard program is not supposed to declare the structures in the standard library independently, but is expected to get them directly or indirectly from the system header files.

Although we intend to cover all standard kinds of C structure fields for the GNU machines, our attempt is not quite comprehensive. For example, we allow bit-fields and arrays as structure elements, but we don't include any data access words for them. But mainly, we can't claim a comprehensive understanding of the GNU structure layouts, which are not definitively documented in any one place, as far as we know. The GNU source is a complex body of knowledge, and even with a lot of embedded documentation, is not exactly your normal bedtime read. Although somewhat C literate, we are not C expert. We hope we have understood the essential part of it. Our fundamental understanding is that, with the exception of bit-fields, any structure layout can be reproduced by specifying a minimum structure alignment, a structure size minimum roundup, and the sizes and alignments of all basic data types (`char`, `short`, `int`, `long`, ...), plus pointer types, which we call "atomic" data types.[‡]

We state the algorithm we actually follow in Section 4.

[‡]We use the word "alignment" to mean a number of address units of which the address of a data block in memory is a multiple, like 1, 2, 4, 8, ... In standard C [4, 3.1,3.4], bytes are not necessarily 8 bits, but as far as alignment is concerned are effectively address units. We always mean 8 bits when we say "byte". The number of bits in an address unit is system dependent.

Except for bit-fields, the system interface is in the file `machine.fs`. The version we provide works for the Amiga 3000 (Motorola 68030), NeXT (Motorola 68040), and Macintosh (Motorola 680x0). What we aim at there is a sufficient parametrization, not a direct mapping of the GNU macros and constants for system parameters.

After some testing with a GNU CC compiler on a NeXT with results that surprised us, we decided to take seriously the warnings in Kernighan and Ritchie [5, pp. 150, 213] that bit-fields are especially implementation dependent in standard C. They should be rare, but do occur in several Berkeley UNIX header files. Although there seem to be none among the required structure elements in POSIX structures, there also seems to be no guarantee that they do not occur legally as nonrequired elements in particular implementations of required structures.

Our approach is to make no attempt to map the GNU parameters that affect bit-fields, such as `PCC_BIT_FIELD_TYPE_MATTERS`, but to make Forth bit-fields rich enough to reproduce any possibility, and leave it up to the user to test what a particular system does and supply the appropriate Forth syntax. In other words, for bit-fields the Forth syntax may vary with the underlying system, while for "normal" structure elements the syntax is system independent.

The implementation here addresses several issues:

1. *Nesting of structures and unions to any depth.* This is not uncommon in Forth (at least for structures), e.g., the FSL implementation [2] does it as does the Forth-83 implementation in Dick Pountain's book on object-oriented Forth [8].

2. *Orthogonality of field names.* I.e., allowed use of identical names in different structures.

For these two issues, we adopt the solution in Randolph Peters' implementation in Pocket Forth [1]. We translate that part of his scheme to ANS Forth, with only cosmetic modifications. Nesting of structures is essential for C layout compatibility. Orthogonality of field names is not; but it's nice to have, and should make it easier to translate between C and Forth structure definitions. Peters' is the only Forth implementation of reusable field names that we know about.

We do not implement independence of field names from other Forth names that might be found first in the search order. That would not be hard to do, say, by searching only a tokens word list when a field name is wanted.

3. *Field typing.* We elaborate aspects of Peters' scheme and the typing scheme in the FSL [2] implementation, which are similar in spirit, and both of which lend themselves to alignment. Peters includes unstructured data fields of any size and substructures, and the FSL also has `integer:`, `float:`, and

`array:` fields, plus unions. Besides these, we include bit-fields and many of the scalar C types, which we call atomic types. We do not make the signed/unsigned distinction, leaving that to the user; and we include only a single pointer type, which is taken to have the same size and alignment as `void*` (also the same as `char*`) in standard C [4, 6.1.2.5]. The focus for us is on sizes and alignments. Having a single generic pointer type should be sufficient in that respect for many systems. For systems where that isn't true, we make it possible for the user to define his own atomic types, with their own sizes and alignments.

4. *Binding of structure data references.* The FSL scheme pays attention to this issue, the point being that a straightforward elaboration of the syntax for variables to access data in structure instances means extra overhead, both in code size and execution time. And that on the other hand structure and substructure field names are fixed, so the overhead can be avoided in a word definition where the structure instance is known at compile time by precomputing and compiling the field address (early binding). Dick Pountain [9] discusses that, too.

Since this is likely to be a common situation, we provide some explicit early binding operators in a Structure Extensions word set, including `]@`, `]!`, `]c@`, `]c!`, and a few others. This kind of peephole optimization is a minor

issue for ordinary variables, and a portable implementation is unlikely to make them more efficient. If written in assembly language, such words could be implemented for explicit optimization of ordinary variables as well. For structures, even a portable optimization can be significant.

Otherwise the binding is late, corresponding to the direct generalization of normal syntax for variables. For array fields one might want to mix the two, binding the address of an array field early, and an index into the array late.

5. *Data reference syntax.* There are two basic syntax choices for referring to data in a structure or union instance, where the field names are mentioned before or after the instance name, and also two basic orderings for the names in each. Peters puts the field names before, ordered from deeper structure nesting on the left to shallower on the right, which makes a natural chain, including the parent structure itself furthest to the right. That is expressive for his examples, (reproduced later), and is also natural for implementing his substructure scheme. The FSL scheme follows the C style of putting field names after the instance name, ordered towards deeper on the right, with the parent structure furthest to the left, again a natural chain including the parent. We reject the other two, "unnatural" orderings, somewhat arbitrarily, because syntactic conventions could probably be discovered that would make

them seem natural.

Surely neither of the remaining "natural orderings" optimizes expressiveness in all situations, so we cop out and do both. We do adopt the field prefix mode for the more primitive words in our list. This interacts with the next issue.

6. *Action of named structure instances.* Should the execution of a named structure instance simply leave an instance pointer, or should it resolve nested field names and leave a field address and/or size, or fetch a value according to its data type, or perform a method, or what? A general purpose scheme should at least not hinder any of these possibilities. Clearly whether field names precede or follow the naming of an instance interacts with this question.

We provide a named-instance defining word `typeof` intended for use with `DOES>` to make defining words with various actions. We also define two generic words of that sort, `{}structof` and `structof{}`, designed for use with address and data access operations to be described later.

2 Structure of structure data

In the interest of focus, we describe here the layout of structure data and type information that we have in mind.

2.1 Structure instances

A structure instance contains the actual data of a structure, and possibly more information. Instances may be either named or unnamed. The essential kernel of either kind of instance is the structure data itself, which we understand to exclude type information. This is the part whose layout should be C compatible. We call the address of such a memory block of pure data the `sda`, for "structure data address". If we speak of a structure pointer, we mean the `sda`. Substructure instances are presumed to contain only pure data.

As a rule of thumb in our discussion, unnamed structure instances contain only pure data, and named instances (in the sense of named Forth words) contain one extra piece of information, a pointer to the structure type information. That pointer is called an `stype`. Although substructures have named identifiers, substructure instances are not "named structures" in the Forth sense, and as we said above contain only pure data.

Layout of a typical named structure instance:

```
stype
field 1 data
⋮
field n data
```

The address of `field 1 data` is the `sda`. Alignment for compatibility with the underlying system is understood. There is a subtlety here. The named structure instance is typically made with `CREATE`, and `stype` occupies the first cell in its data field, at the Forth-aligned `dfa`. Then `field 1 data` is not necessarily located one cell after the `dfa`, because the structure alignment may not be consistent with that. Structure data access words have to take this into account.

A typical unnamed instance would omit the `stype`.

2.2 Structure type definitions

The structure type definition contains the information about the layout and sizes of the structure fields. We implement the structure type pointer, or `stype`, as the `dfa` of a structure type word.

Layout of a structure type definition (implementation dependent):

```
          code field (CREATE'd action leaves the dfa)
stype:    structure data size  (including padding)
          class  (1 for structures)
          structure alignment
          #fields
          field 1 parameters
          ⋮
          field n parameters
```

In this discussion the "field" in `field 1`, etc., is used in the sense of a C structure element. In discussions of standard C, "field" is sometimes used as a synonym for "bit-field" [5, p. 149], a practice we avoid.

The data field arrangement, following the code field, varies only a little from Peters' "type definition table". We call it instead the "structure table", reserving "type definition" for more generic data typing. In other words, our `stype` is the address of the structure table. Including explicit information on the number of fields rather than a table termination signal is an implementation detail. Although the layout of the structure table is

an implementation detail, we like keeping the ordering the same as for the storage of the structure data.

The field parameters in the structure table allow the construction of field offsets from the beginning of pure structure data, including substructure nesting. As long as they satisfy this function, their order and content are implementation details.

Layout of structure field parameters (implementation dependent):

```
field identifier
field offset
field type pointer
```

2.3 Structure field classes

We require six classes of fields, organized in this implementation as follows:

field	class	type pointer
unstructured data	0	`ustype`
structure	1	`stype`
atomic data type	2	`adtype`
array	3	`atype`
union	4	`utype`
bit-field	5	`bftype`

The class numbering is implementation dependent. The numbering here reflects our personal implementation priority, with levels of conditional compilation in mind. An important use of the class number is to indicate nesting termination.

The unstructured data class should not be included in C compatible structures. Taken together with just the structure class, it can provide a simple, standalone Forth structure facility with full nesting and independent field identifiers, where the user keeps track of primary data types and sizes. This kernel is our ANS translation of Peters' implementation, and there is an option in `cstruct.fs` to compile just that much.

The atomic data types are the standard C types, `char`, `short`, `long`, `int`, `long double`, `float`, `char*`, etc. Each has a type definition pointed to by an `adtype`, described in Section 3.

Arrays are made of elements all of the same kind (including especially size), which may belong to any of the six classes except bit-fields. To be C compatible, the array elements should not be unstructured data.

Unions are made of elements whose storage space overlaps, with size and alignment large enough to accommodate the largest. The elements may belong to any of the six classes, except that unstructured data should not be included in C compatible unions.

As Kernighan and Ritchie express it [5, pp. 148, 213], a union is just a structure with all elements offset by zero from the beginning, with alignment accommodating the biggest alignment of any element, and with size big enough to hold any element.

There will be more words later about bit-fields than we would have wished.

In this implementation, each structure field parameter occupies one cell, which means 12 bytes for each field in 32-bit environments. Since we expect that most applications will either involve relatively few structure tables, or will correspond to an industrial strength environment when there are many, this may not be excessive. On the other hand, restricting structure instances to 64K bytes is likely to be more than adequate, in which case 16 bits for each of the first two parameters should suffice, which would reduce the overhead to 8 bytes.

3 Type structures

In Section 2 we introduced atomic data, arrays, unions, and bit-fields as classes of structure fields. In C they are also basic data objects (except for bit-fields, which occur only in structures), with unions on much the same logical footing as structures.

During implementation, we found ourselves driven to a typing scheme with a partial type data structure shared by all types, including each of the atomic data types. In this section we lay out the type data pointed to by `adtype`, `atype`, `utype`, and `bftype`, as well as that pointed to by the unstructured field type pointer, `ustype`. The first three fields, i.e., `size`, `class`, and `alignment`, are shared by all types, including that for structures already described in Section 2. The size in the first field is measured in address units.

Here are the type information layouts. For completeness, we include the `stype` layout given in the Section 2:

```
ustype: size  (of unstructured field)
        class  (0)
        alignment  (1)
```

```
stype:  size  (including padding)
        class  (1)
        alignment
        #fields
        field 1 parameters
        ⋮
        field n parameters

adtype: size  (of atomic data)
        class  (2)
        alignment

atype:  size  (#elements× size of type at type pointer)
        class  (3)
        alignment  (of element type at type pointer)
        #elements
        type pointer  (to element type)
```

```
utype:  size  (rounded max of union field sizes)
        class  (4)
        alignment  (max of union field alignments)
        #fields
        field 1 parameters
        ⋮
        field n parameters

bftype: size  (of 0, 1, or 2 containers in bytes)
        class  (5)
        alignment  (of container type)
        size  (of container type in bits)
        bit offset  (in first container field)
        #bits  (in bit-field)
```

Although we include array fields in structures and unions, we implement neither array data objects nor accessors for array data. C allows multiple array indices, which it treats by having arrays of arrays. We take that to be

a matter of access, irrelevant for the `atype`, which does not record how many indices might be used to index the data.*

The union definition table pointed to by `utype` has exactly the same form as a structure definition table, with a different interpretation of size and alignment as indicated in the table above, and with offsets in the field parameters all set to zero.

We found the implementation of bit-fields a major project. The idea is to save space by packing more than one bit-field or partial bit-field into a system storage unit. This is explained further in the Section 4.

*The `atype` layout here could in principle be used for an implicit multiple index scheme by letting the element type pointer be another `atype`, etc., in a chain ending with a non-`atype`, corresponding to the last index. All of the alignments in the chain would be the same, that of the final array element type. Seems more complicated than we're likely to need.

4 Layout algorithm

Here is the algorithm we follow for structure and union layout, aimed at compatibility with the GNU scheme for parametrizing systems, as long as there are no unstructured fields, and with a certain exception for bit-fields. There is no distinction between signed and unsigned types. It is up to the user to handle that at the point of field access, when it is an issue. Byte-ordering is irrelevant at the layout level—that affects field data access only. Whether bit-allocation for bit-fields is from left to right or right to left within an embedding integer field is irrelevant at the layout level for the same reason.

4.1 Normal fields

The rules in this subsection are mostly for everything except bit-fields.

1. Each atomic data type has a size in address units and an alignment in address units, which is system dependent. These alignments are often expressible in terms of only a few parameters, but each is specified independently in our scheme at the level of the data type structure. We include a single, generic pointer type as an atomic data type, not distinguishing among pointers to different data type instances. Other atomic data types can be supplied by the user.

2. Each structure field has a size, an alignment, and an offset from the beginning of the structure.

3. The size of a structure is the sum of the sizes of its fields, plus any padding between fields to achieve alignment of the later field, plus a padding at the end to round up the size to a multiple of the structure alignment. The contribution of bit-fields to the size will be discussed later. GNU also includes a system dependent `ROUND_TYPE_SIZE` macro, which seems to be defined only for the Intel 80960. We have omitted this. It would occur in the words `}struct` and `}union`.

4. The size of a union is the maximum of the sizes of its fields, rounded up in the same way as the size of a structure. The contribution of bit-fields is again special.

5. The alignment of a structure or union is the maximum of a minimum, system prescribed alignment for structures and unions, and the alignments of all of its fields except bit-fields, which count as integral atomic types for the purpose of alignment.

6. Each structure or union field consists of an atomic data type, a structure, a union, an array, a bit-field, or an unstructured field.

7. An array has elements all of the same type (which implies the same size and alignment), which can be any of the atomic data types, a structure, a union, or an array.

8. The size of an array is the number of elements times the size of one element. The alignment of an array is the alignment of any element.

9. The raw alignment of an unstructured field is one address unit. Other alignments may be forced, but are recorded only implicitly in the offset of the field, not in the unstructured field type instance.

4.2 Bit-fields

For bit-fields, we have already mentioned that we do not attempt to map the GNU macros. In the C Standard [4, 6.2.1.2], named bit-fields are associated with one of the `int` types (unsigned or signed), and cannot have a width of more bits than that type. Our reading of the standard is that the actual container size can be anything big enough, and does not have to be built from a sequence of `int` sizes. The bit-field is allowed to overlap a container boundary if there are too few bits available for packing in it, or not, depending on the implementation. The alignment of the container is unspecified. Unnamed bit-fields with only a container type and a width specified can be used for padding; and an unnamed bit-field of width zero forces packing to end in the current sequence of bit-fields, if any, with the next bit-field starting a new container. The syntax for bit-field access is like that for an `int`. Whether the policy for bit-field layout when embedding would overlap a con-

tainer boundary has to be the same for structures and unions is undefined in the C Standard.*

GNU CC admits other integral types for bit-fields, such as `char`, `long`, etc.

The approach we take in Forth makes possible the layout of a sequence of contiguous bit-field declarations starting with any alignment, any number of bits of initial, unnamed padding, any width of named bit-fields with any unnamed padding in between, embedded in any commensurate total number of address units with any unnamed bit-padding at the end that is also commensurate with the total number of address units.†

We adopt the spirit that the container type not only limits the maximum size of a bit-field, but also has size and alignment implications for the container. Flexibility of container size and alignment is achieved by allowing non-`int` container types.

---

*The terms "unspecified" and "undefined" have a technical meaning in the C Standard, roughly the following. "Unspecified" is for correct language constructs and data, and means the standard "explicitly imposes no requirements" (not the same as imposing no explicit requirements) [4, 3.17]. "Undefined" is for nonportable or erroneous situations, and means the standard "imposes no requirements" [4, 3.16].

†As far as we understand, this is possible in GNU CC.

1. Any atomic type is allowed for bit-field embedding. We call this the container type or the type of the container field. Container types with the same number of bits and the same alignment are not distinguished from each other. The spirit is that the container type should be of integral type, i.e., `char`, `int`, etc.

2. The width of a named bit-field must be nonzero, while that of an unnamed bit-field may be zero.

3. When nonzero the width of a bit-field may not be larger than that of its container type. As many bits as possible are allocated from the unallocated bits in the container field of any immediately preceding bit-field with the same container type; and if a nonzero width is left over, a new container field of the same type is started for the remaining bits. That is, we require the bit-field to overlap an embedding boundary in such a case. If there is an immediately preceding bit-field of different container type, there is no embedding in the preceding container field; and a new container field is started (with the alignment of its type) for all of the bits in the bit-field. This paragraph applies only to nonzero widths. It implies that such bit-fields have either one container field, or two container fields of the same type when it straddles an embedding boundary. We require that structure and union bit-field elements have the same layout; i.e., if a bit-field in a structure overlaps, requiring two container units, it also overlaps and generates two container

units in a union. The offset of the first container field is zero in a union.

4. A zero-width bit-field with any container type prevents further embedding into any immediately preceding field, and aligns the initial offset for the next structure or union field according to the container type. It does not allocate a new container field.

5. Unnamed bit-fields survive in the structure or union table. This would not be necessary just to get the major effect of padding the offsets of succeeding named bit-fields within their container fields, and the offsets of those container fields and of other succeeding named fields within the structure or union; but we think it's a good idea to record the unnamed fields in the structure or union type information.

6. Both named and unnamed bit-fields may occur in any order, adjacent to each other or isolated among other fields.

5 Overview of structure words

This overview is a brief functional description. Stack patterns and other specifications can be found in the implementation file `cstruct.fs`.

5.1 Structure word set

```
struct{  }struct  union{  }union
n-aligned  unstruct  field  array-field  bit-field  bit-pad
cchar  cwchar  cint  cshort  clong  cpointer  cllong
cfloat  cdouble  cldouble
/type  /align  make-type-instance  typeof
make-atomic-type  make-array-type  make-unstruct-type
>sfa  >sfo  >sfa&type  >sfo&type
```

5.2 Description

Here is an example borrowed from Peters (see `examples.fs`) except that his word `field` corresponds to our `unstruct`, and his word `struct` corresponds to our `field`:

```
struct{
 12 unstruct first
 16 unstruct last
}struct name.struct

struct{
 2 unstruct month
 2 unstruct day
 2 unstruct year
}struct date.struct

struct{
 name.struct field name
 date.struct field doa
 12 unstruct mrn
 64 unstruct precis
}struct pt.struct
```

The code above defines structure type words `name.struct`, `date.struct`, and `pt.struct`. We are not particularly advocating the `.struct` naming convention. If we were to do so, it would probably be a `.s` convention for struc-

tures and `.u` for unions. The words `unstruct` and `field` between `struct{` and `}struct` absorb the type information that precedes them, define or look up identifying tokens for the field names that follow them (`first`, `last`, `month`, `day`, `year`, `name`, etc.), and build a sequence of field definition parameters on the stack. The word `struct{` initiates the sequence, and `}struct` creates a structure type word and compiles the sequence of field parameters from the stack into a structure table in the structure type word's data field, preceded by the other information described in Section 2.2.

When executed, the field names leave their identifying tokens, called `id`'s, on the stack; and the structure type words leave their `dfa`'s, that is, their `stype`'s.

Here is a structure type including one C `char` field and one field with an array of 10 C `long`'s:

```
struct{
 cchar field sue
 10 clong array-field george
}struct harry.struct
```

And here is one containing two arrays of `harry.struct` structures:

```
struct{
 15 harry.struct array-field arthur
 20 harry.struct array-field marie
}struct harry-arrays
```

The examples above are included in `shotype.fs`, along with union versions and bit-field examples, to illustrate a browser for structure and union types implemented there.

The word `n-aligned` is mainly a factor in the field constructors `field`, `array-field`, and `bit-field`; but it can be used explicitly when alignment of an unstructured field is wanted (the field constructor `unstruct` does no alignment). For example, if one wanted the field

```
2 unstruct year
```

in `date.struct` above to have an alignment of four, one could say:

```
2 4 n-align unstruct year
```

This is tricky, because it not only has the effect of saying

```
4 unstruct year
```

but also adjusts an implicit alignment deeper on the stack. The best policy is to avoid explicit use of `n-aligned` if possible, and use the implicit minimum alignment of structures, plus padding included directly in the size of the unstructured field.

The word `bit-pad` inserts unnamed bit padding.

The atomic type words `cchar` ... `cldouble` represent most of the scalar GNU CC types. Some of these are not standard C.

The word `/type` converts any of the six type structures into its data size in bytes, and `/align` converts them to the sizes of their alignments. For bit-fields, the data size is that of the 0, 1, or 2 container fields.

The words `make-unstruct-type` and `make-array-type` are used in this implementation by `unstruct` and `array-field` as factors that build type words on the fly. They could also be used explicitly to make unstructured type and array type words to be used with `field`, dispensing with `unstruct` and `array-field`. The word `make-atomic-type` is intended to let the user cover C implementation-dependent gaps, for our example, in our pointer type coverage. The `make-` style of nomenclature is borrowed from Anton Ertl's Gray [3].

The word `make-type-instance` is used to allocate type instances.

A number of words like `>sfo` do exactly the same thing when operating on structure or union types. To save names, we take the attitude in such

cases that a union is just a kind of structure.

The word `>sfo` converts a structure or union type and a sequence of `id`'s for nested substructures that resolves to an atomic or unstructured or array field, such as `last name` for the structure type `pt.struct` in the example above, into the offset of the field from the `sda` of an instance. Here "`sfo`" stands for "structure field offset". For bit-fields, the offset of the first container field is returned. The word `>sfo` can also convert a truncated nesting, such as just `name` with `pt.struct`. Examples of the syntax are given in Section 6.

The word `>sfa` does the analogous thing, but takes an `sda` as well as an `stype` as input, producing the address of the field instead of the offset. For bit-fields, that is the address of the first container field.

The words `>sfo&type` and `>sfa&type` also leave the type pointer. They evolved from a factorization of Peters' implementation, which returned sizes instead of types.

Nesting in the `id` chains resolved by these words cannot go deeper than an `id` for one of the primitive types: unstructured, atomic data, or bit-field. It is also stopped by an array type. Although an array may have structure elements, a new chain would have to be started to access any nesting in those, after indexing into the array. As we said earlier, we do not implement array access.

See `cstruct.fs` for more details about the Structure word set.

5.3 Implementation note

Our implementation makes a type instance for every unstructured field, array field, and bit-field in a structure type definition. We indicated above that `unstruct` and `array-field` can be eliminated by using explicitly defined types with `field`. That would reduce the type overhead if there were several unstructured fields or arrays of the same type.

In the case of bit-fields, implicit type generation helps us track the arbitrary bit offsets they can have in their containers. We have not been tempted to try to reduce that overhead.

6 Overview of structure extension words

As in the previous section, we given only a functional overview here, without stack effects. Details can be found in `cstruct.fs`.

6.1 Structure Extension word set

```
{  }#
{}structof  {}structof/
structof{}  }&  }&/
]&  ]&/  ]@  ]!  ]c@  ]c!  ]2@  ]2!  ]execute
```

6.2 Description

The words `{` and `}#` simply count the number of parameter stack elements between them, and could be of general utility.* Although we indicate at the end of this section how `}#` can be used explicitly for early binding of field offsets, we use it mainly as an implementation factor in the `DOES>` part of the structure instance defining words `{}structof`, `{}structof/`, and `struct{}`.

*We think that words like { should be declared officially to have at-will meanings.

The first two are for left field syntax words, and the third is for right field syntax. (The same three words are used to define union instances.)

To illustrate, we continue with our translation of Peters' example. In the `{}structof` variant, we create an instance of the structure `pt.struct` with name `}patient` like this:

```
pt.struct {}structof }patient
```

To retrieve the address of the `first` field in the `name` substructure in the structure instance `}patient`, we would say:

```
{ first name }patient
```

The address of the `name` substructure is returned by

```
{ name }patient
```

and that of the `}patient` structure data (i.e., the `sda`) by

```
{ }patient
```

A right field variant would be

```
pt.struct structof{} patient{
```

In the left field variant the structure word `}patient` does the address calculation. In the right field variant we use an explicit field closing operator to do that, a variation on Julian Noble's array syntax [9], with which it fortunately does not conflict:

```
patient{ name first }&
```

The addresses of the `name` field and the `patient{` structure pointer would be given by:

```
patient{ name }&
patient{ }&
```

The left syntax example in the file `examples.fs` actually uses the defining word `{}structof/`, which builds instances that leave the field size as well as the address. This is very much like Peters' word `new.struct`, the effective differences being that `new.struct` does not require an opening `{` for the `id` list of an instance it creates, and that its `id` lists have to resolve all the way to a primitive field.

The right syntax example in `examples.fs` gets the same effect by using the closing word `}&/` with a structure instance `patient{` defined using `structof{}`.

The words beginning with `]` are the closing words for early binding mentioned in Section 1. Here is a sample phrase that could be used in a word definition to compile the address of the `last name` field of the `}patient` structure instance as a literal:

```
[ { last name }patient ]&
```

These words all start by switching to compilation mode, where they compile a simple action based on a computation stacked from interpretation mode. They are best understood directly from the implementation in the file `cstruct.fs`, and from the examples in `examples.fs`.

In version 1.0 of this word list, we included early-binding words for structure field offsets, for example, `]o`. We left them out of this version because they are redundant, and because the offsets are most often wanted for computing addresses, which is already covered. The same effect can be achieved by using a structure type word with `>sfo` or `>sfo&type` between `[` and `]`. For example, to compile the offset of the `last name` field from the `sda` of `}patient` into a word definition, we would use the structure type directly in the phrase:

```
[ { last name }# pt.struct >sfo ] literal
```

To compile the offset of the `name` substructure, we would use:

```
[ { name }# pt.struct >sfo ] literal
```

The syntax `{ ... }#` can be eliminated by giving the number of `id`'s explicitly:

```
[ last name 2 pt.struct >sfo ] literal
```

and

```
[ name 1 pt.struct >sfo ] literal
```

GNU free documentation license

0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images

composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical

measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using

public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History"section. You may omit a network location for a work that was published

at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified

Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections

entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) `YEAR YOUR NAME`.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being `LIST THEIR TITLES`, with the Front-Cover Texts being `LIST`, and with the Back-Cover Texts being `LIST`. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being `LIST`"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

References

[1] Randolph M. Peters, June 14, 1993: `Struct`.

[2] "High order data structures V1.9", January 18, 1995: `structs.txt`, `structs.seq`.

[3] Anton Ertl, "Gray", release 4, August 8, 1994: `gray4.tar.gz`. See also: struct.fs, structs.html.

[4] *American National Standard for Programming Languages–C, ANSI/ISO 9899-1990.*

[5] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition, (Prentice Hall, Englewood Cliffs, New Jersey, 1988).

[6] Donald A. Levine, *POSIX Programmer's Guide*, (O'Reilly & Associates, Inc., Sebastopol, California, 1991).

[7] Free Software Foundation, *Using and Porting the GNU Compiler Collection (GCC)*, "Target Description Macros": "Storage Layout", "Layout of Source Language Data Types".

[8] Dick Pountain, *Object-Oriented Forth: implementation of data structures*, (Academic Press, New York, 1987).

[9] Julian V. Noble, *Scientific Forth*, (Mechum Banks Publishing, Ivy, Virginia, 1992), pp. 105, 106.