23rd EuroForth Conference

September 14-16, 2007

Schloß Dagstuhl, Germany

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 23rd EuroForth finds us in Schloss Dagstuhl for the third time. The three previous EuroForths were held in Schloss Dagstuhl, Germany (2004), in Santander, Spain (2005), and in Cambridge, England (2006). Information on earlier conferences can be found at the EuroForth home page (`http://dec.bournemouth.ac.uk/forth/euro/index.html`).

Since 1994, EuroForth has a refereed and a non-refereed track.

For the refereed track, one paper was submitted, and none was accepted (0% acceptance rate). For more meaningful statistics, I include the 2006 numbers: seven submissions, three accepts, 43% acceptance rate. The paper was sent to three program committee members for review, and they produced three reviews. This year, none of the program committee members has submitted a paper. I thank the author for his paper, and the reviewers for their reviews.

Two papers were submitted to the non-refereed track in time to be included in these proceedings. In addition, these proceedings include the slides for four talks that were presented at the conference without being accompanied by a paper. Workshops and social events complement the program.

We are grateful to Klaus Schleisiek and Gudrun Zaretzke for organizing this year's EuroForth.

Anton Ertl

Program committee

Sergey N. Baranov, Motorola ZAO, Russia (secondary chair)
M. Anton Ertl, TU Wien (chair)
David Gregg, University of Dublin, Trinity College
Ulrich Hoffmann, Heidelberger Druckmaschinen AG
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Reuben Thomas

Contents

Programme 5

Non-refereed papers

M. Anton Ertl: Gforth's libcc C Function Call Interface 7
Angel Robert Lynas, Bill Stoddart: A Reversible Computing Approach to Forth Floating Point 12

Presentations

David Guzeman (slides), Stephen Pelc (talk):
Imagine a Sea of Processors 23
Bernd Paysan: Audio GUI: MINOS@work 34
Federico de Ceballos: A Framework for Data Structures in a Typed Forth 37
M. Anton Ertl: How to Deal with Context 43

Programme

Friday, 14.9.

14:30	Coffee & cake / check-in
15:00	Opening of 23rd euroForth conference / organisation of workshops
15:55	A.M.Ertl: Gforth's libcc C Function Call Interface
16:40	B.Stoddard: A Reversible Computing Approach to Forth Floating Point
17:20	Workshop: Distributed Forth
18:00	Dinner
19:00	Workshop: Forth Open Source Library
22:00	Cheese plate im Weinkeller

Saturday, 15.9.

7:30-8:45	Breakfast
9:15	S.Pelc: IntellaSys (Presentation)
10:00	S.Pelc: Implementing programs on 24 or more cores (Discussion)
10:35	K.Schleisiek: Microcore 2.0 (Discussion)
11:20	B.Paysan: Forth activities of the past year
12:15	Lunch
13:30	Workshop: I18N
15:30	Coffee & cake break
16:00	Workshop: Formal Stack Orientated Virtual Machines Type Safe Forth
18:00	Dinner
19:30	"Purple Pool" im Musikzimmer Georgia Charlotte Hoppe (Clarinet) Uli Sobotta (Horn) Frauke Wessel (Sax)
22:00	Cheese plate im Weinkeller

Sunday, 16.9.

7:30-8:45	Breakfast
9:00	Impromptu Talks: Federico de Ceballos: Structures in a Typed Forth Anton Ertl: Context Klaus Schleisiek: Dying Forth
11:00	Presentation of workshop results Presentation of informal workshops: Microcore 2.0 part II Gforth packaging I18N part II
12:15	Lunch
13:30	Closing rituals
	Decision: "Core people are those who have attended euroForth in the previous 5 years".
14:00	End of 23rd euroForth conference
14:00	Beginning of 4th day
15:30	Coffee & cake break
18:00	Dinner in a nearby village

Monday, 17.9.

7:30-8:45	Breakfast
8:45	4th day
12:15	Lunch

Gforth's libcc C Function Call Interface

M. Anton Ertl*
TU Wien

Abstract

A major problem in our earlier proposal for a C interface was that a part of the interface was not portable between platforms. The libcc interface solves this problem by using a C compiler and its `.h`-files. The `.h`-files contain knowledge about the specific platform, and the C compiler automatically inserts the necessary conversions between Forth and C types. In this paper we describe the libcc implementation and interface. We also discuss how a Forth-C interface might be standardized.

1 Introduction

The programming interfaces of many useful libraries are defined as collections of C functions and C data structures, so being able to call C functions is a very useful capability for a Forth system.

In an earlier paper [Ert06], we discussed general design issues for a function call interface and designed a C interface based on these ideas that was intended to be implemented using foreign function call libraries like the ffcall libraries[1] or libffi.

In the meantime, we have explored a new implementation approach for foreign function calls that makes it possible to eliminate the non-portable C part of the declaration in most cases. In this paper we describe this implementation approach (Section 3) and the resulting C interface (Section 4). We also mention some of the issues in standardizing a C interface (Section 5).

2 Portability

Portability is a central problem addressed in our earlier work, and it has led to our new implementation approach, so we revisit it here.

The primary form of portability that we are interested in is in being able to use the same Forth program on several platforms (e.g., Linux-i386, Linux-AMD64, MacOS X, and Cygwin), even if the Forth program calls C functions. Portability between Forth systems is another issue that we discuss in Section 5.

The platform portability problem is that equivalent C functions (same name, same functionality) have different argument and return types on different platforms. A typical example is the POSIX function

```
off_t lseek(int fd, off_t offset,
            int whence);
```

On some ancient platforms, this function is instead defined as follows:

```
long lseek(int fd, long offset,
           int whence);
```

However, even if we use the official POSIX definition, `off_t` can be different things on different platforms: either `long long` or `long`; what's worse, it can even be different things on the same platform, depending on the way that `lseek()` etc. are compiled[2].

In our earlier work we propose to deal with this problem by having a platform-dependent C part in the function declaration, e.g.:

```
c-types lseek int longlong int -- longlong
```

We suggested that these C parts could be generated automatically out of the `.h`-files, reducing the amount of per-platform work needed.

How does C avoid this problem? Primarily by having `.h`-files with standard names, that contain (among other things) platform-specific definitions, e.g., of types like `off_t` (that's why automatioc generation of C parts out of `.h`-files would help.

C has another benefit: the C compiler knows more about data types of parameters than a Forth compiler (at least if the function prototype is `#included`); this allows it to insert the necessary type conversions transparently.

3 Implementation approach

Instead of implementing a C parser that translates `.h`-files into Forth, libcc uses the existing C compiler's knowledge of C and its ability to read and

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1] `http://www.haible.de/bruno/packages-ffcall.html`

[2] Whether _FILE_OFFSET_BITS is set to 64 or not.

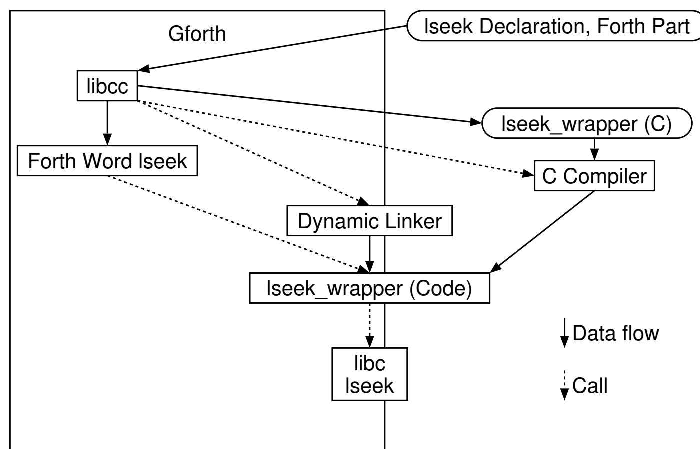


Figure 1: A foreign function call for `lseek` with libcc

understand `.h`-files, and to insert the necessary type conversions.

For each Forth part of a foreign function declaration, libcc generates a wrapper function in C that accesses the cells and floats on the Forth stacks and passes them as parameters to the target C function (e.g., `lseek()`), then calls the target function, and finally takes the return value and pushes it on the appropriate Forth stack. The wrapper function is later compiled by a C compiler, which automatically inserts the necessary conversions between the Forth types and the C types, and also knows about the C calling convention on the particular platform.

Finally, the wrapper function and the target function are linked dynamically, and when the Forth program calls the word corresponding to the target function, it calls the wrapper function, which in turn calls the target function. All the wrapper functions can be called through a simple indirect call, without needing a library like libffi.

This approach is depicted in Fig. 1.

A major advantage of this approach is that any tricks that the `.h`-files play with C macros (e.g., renaming C functions, as is usually done for `lseek()`) take full effect in the wrapper function, so it will call exactly the function that we were interested in.

Disadvantages of this approach are:

- It requires a C compiler at run-time.
- The actual call of the C compiler is quite target-specific (in particular because it is necessary to generate a dynamically linkable binary). Fortunately that's just one small piece of code per platform.
- The compilation of a wrapper function is much more expensive than similar operations necessary with ffcall and libffi. We are exploring batching and caching to reduce this cost, and will report on that in more detail in future papers.

4 The libcc interface

This section describes the libcc interface, in particular the differences from our earlier interface proposal [Ert06]. This section gives an overview, mainly through the `lseek` example. For a complete specification, read the current version of the Gforth manual.[3]

4.1 Declarations

As in our earlier work [Ert06], the declarations for a target C function consist of a C part and a Forth part. The Forth part is exactly as described in our earlier work, but the C part is completely different: it consists of lines of C code prefixed with `\C`; the programmer has to ensure that the C code declares the target C function (and its parameter and return types). A complete set of declarations for two `lseek`-calling words can look as follows:

```
\c #define _FILE_OFFSET_BITS 64
\c #include <sys/types.h>
\c #include <unistd.h>
c-function lseek lseek n n n -- n
c-function dlseek lseek n d n -- d
```

In our implementation the `\C` declarations are copied to the generated C code in front of the wrapper function, making the target function known to the C compiler, which can then generate the right type conversions for the parameters and return value.

[3] `http://www.complang.tuwien.ac.at/forth/gforth/cvs-public/`

In this example we have two Forth parts, resulting in two words, both calling the C function `lseek()`, but with different stack effects: the Forth word `lseek` has the stack effect `( n n n -- n )`, whereas `dlseek` has the stack effect `( n d n -- d )`.

4.2 Calls

Calls work exactly as described in our earlier work: The programmer has to push the arguments left-to-right on the stack; the argument and return value types are determined by the Forth part of the declaration and are independent of the concrete C types used by the function. Example:

```
fd @ 0. SEEK_SET dlseek -1. d= if
  ... \ error handling
then
```

4.3 Variadic Functions

To call C functions that can take a variable number and variable types of arguments, like `printf()`, the programmer can declare several words, each with a desired parameter set, and then call these words by name. E.g.:

```
\c #include <stdio.h>
c-function printf-nr printf a n r -- n
c-function printf-rn printf a r n -- n
s\" n=%d r=%f\n\0" drop -5 -0.5e printf-nr
s\" r=%f n=%d\n\0" drop -0.5e -5 printf-rn
```

One problem here is that the C compiler does not know the desired type of the varargs and does not insert conversions to this type; this can lead to portability problems. One solution to this problem is to perform the conversion explictly at the C level. E.g.:

```
\c #define printfll(s,ll) \
\c             printf(s,(long long)ll)
c-function printfll printfll a n -- n
```

Here, we define a macro `printfll` that converts the second parameter explicitly to the `long long` type, and then calls `printf()`. The Forth word then calls `printf` through this macro.

4.4 Variables, Constants

C variables and constants can be converted to Forth words with this interface, by treating them like functions:

```
\c #include <unistd.h>
\c #define seek_set_macro() (SEEK_SET)
c-function SEEK_SET seek_set_macro -- n
```

However, this method is quite heavyweight, both in the amount of typing and in memory consumption, so we may introduce a lighter-weight way to import C constants and variables in the future.

4.5 Calling C function pointers

Programmers can also call C functions for which they have C function pointers, not the name:

```
\c typedef int (* fun1)(int);
\c #define call_fun1(par1,fptr) \
\c             ((fun1)fptr)(par1)
c-function call-fun1 call_fun1 n func -- n
```

The Forth word `call-fun1` now works similar to `execute`, but instead of execution tokens it takes C function pointers of this particular type. The programmer has to define one such call word for each function pointer type he wants to call.

4.6 Callbacks

Conversely, the programmer sometimes wants to generate C function pointers for Forth words, because he has to pass them to some C function directly or through a data structure; this is usually known as callback (the library calls back into the application program through this function pointer).

We have not implemented callback support in libcc yet, and have not fixed the interface yet.

4.7 Future work

The main other missing piece of the C interface is dealing with structs and unions: accessing their fields and passing them to or returning them from functions. Field access is particularly hairy, for the following reasons:

- Each structure constitutes a separate name space, and we have to find a practical way to map C field names to Forth.
- On fetching from or storing into fields we have to convert between the C type and the Forth type. In general the C type is platform-dependent, so if we want to be portable, we cannot use some general fetching/storing words, but have to use one that is specific to the field.

5 Standards

In the long run, it would be nice to be able to port programs that call C libraries between Forth systems. In order to do that, we need to standardize the C interface. However, the interface outlined above is implementation-specific in the C part of

the declarations, so standardizing on it completely is not going to happen. In the following, we explore various levels of standardization that might be more successful. Note that this discussion is not tied to the interface described in the rest of the paper, but applies to any C interface.

5.1 Calls only

The most important portability issue is the calls of the foreign functions themselves, because they tend to be distributed across a large amount of code, and changing the calls would be a lot of work and error-prone. So the word name, parameter order, parameter Forth types and return value Forth type of the word for calling a particular C function should be the same across Forth systems.

However, while the word name and parameter order can be standardized for all calls, the parameter and return value Forth types are specific to each particular function-calling word, so standardizing that means standardizing either the words themselves, or having a standard way to specify these types, i.e., the Forth part of the declaration.

5.2 Specific libraries

For widely-used libraries and APIs, one could standardize the Forth interfaces of the calls. The amount of work required for that could be reduced by just specifying what C type corresponds to what Forth type. E.g., for the POSIX.1-2001 API one could specify that all integer types would be single cells in Forth, with the exception of a few (e.g., `off_t`) that would be double cells.

In order to be able to use these words portably, it would also be helpful to specify a way in which the interface would become available; e.g., what file needs to be `required`, and maybe in which wordlist these words reside afterwards.

One potential problem in that respect is that some APIs have a lot of optional extensions (similar to ANS Forth). One would have to find ways to specify these extensions.

5.3 Forth part of the declaration

While such an approach is probably satisfactory to programmers using these libraries, each Forth system would need its own files with the declarations of all these C-function words. By standardizing the Forth part of the declaration, these parts could be shared between the Forth systems, which should reduce the effort for all involved parties, and also reduce the number of bugs encountered by an individual user.

5.4 C part of the declaration

As mentioned above, standardizing the C part is hard, because it depends on the way that the Forth system implements the C interface (e.g., compare the way this is done in this paper with that in our earlier work [Ert06]), and because it can depend on the platform (e.g., consider the two different C parts for `lseek()` in our earlier work).

What might be possible to address the Forth system issue is to specify the several syntaxes appropriate for different implementation approaches. A particular Forth system would interpret one of them and ignore all the others. The C part of an interface definition might then contain several ways to specify the same thing, and all Forth systems that implement at least one of these ways could work with that.

That solution has its problems, but it is probably hard to find consensus on anything stronger.

6 Related Work

SWIG (Simplified Wrapper and Interface Generator)[4] is a tool for interfacing scripting languages to C. It works in a way similar to libcc. However, after a superficial investigation of SWIG we decided against using SWIG and for implementing libcc directly in Forth; a still-remembered reason was that we heard that adapting SWIG for a new language would require more than 1000 lines of code; currently libcc has 484 lines.

7 Conclusion

The libcc interface uses actual C code for the C part of a C function declaration; this makes it possible to make the C part platform-independent in most cases, because it can just `#include` a standard `.h`-file that contains the platform-specific C code; so the whole interface to most C functions can be completely portable between platforms.

The libcc interface offers a number of other advantages over the interface proposed earlier [Ert06], such as more capabilities when dealing with variadic functions, and the ability to call C function pointers without needing special support.

The implementation of the libcc interface works as follows: For every target function it generates a wrapper function in C, compiles it and dynamically links it into Gforth; the wrapper function is trivial to call, and performs all the stack accesses and (implicitly through the C compiler) argument conversions.

The standardisation of a C interface between Forth systems can be divided into several parts,

[4] `http://www.swig.org/`

with the actual calls probably the easiest to find consensus on, and the declarations being progressively harder.

References

[Ert06] M. Anton Ertl. A portable C function call interface. In *22nd EuroForth Conference*, pages 47–51, 2006.

A Reversible Computing Approach to Forth Floating Point.

Angel Robert Lynas and Bill Stoddart
Formal Methods and Programming Research Group
University of Teesside, UK

Abstract

We describe an implementation of floating point numbers in Reversible Forth using immutable references, and outline the advantages and disadvantages of this approach for the user and system.

Via a probabilistic algorithm example which requires a large number of random trials to create a sufficient sample size, we demonstrate the disadvantage of a naive program with regard to garbage creation, and its semi-automatic resolution using inbuilt features of the reversible virtual machine.

This allows otherwise prohibitively memory-intensive operations to be split into manageable pieces, interim results being saved while garbage is collected after each stage.

Keywords: Forth, Floating Point, Reversible Computing, Garbage Collection

1 Introduction

Forth works particularly well as a means of manipulating values that fit into a cell. Such values may be held as an item on the parameter stack, manipulated by the core stack operations such as `DUP` and `SWAP`, stored in standard (cell sized) constants and variables, and moved to and from the return stack to provide an alternative form of temporary storage. When floating point extensions are added, the situation is not quite so convenient: we now have values that may not fit in a cell. Most implementations use a separate stack to hold these values, they require their own stack operations, and (although we generally think of Forth as untyped) they require their own special variables and constants.

A cleaner approach, which need have only a marginal effect on speed of execution, is to use an "immutable reference semantics". A reference to a value of any kind will fit into a cell, and if the item referenced is immutable, we can hide the difference between references and the values they reference, leaving the application programmer free to to think of such references as being the values referenced. This approach can be applied to any

immutable data type, for example complex numbers. We also use it in our sets package, as described in a previous EuroForth article[8]. More generally immutable references are important in a number of languages, including Java and Python.

The disadvantage of our approach is that it generates garbage. This leads to the second theme of our paper, which is to describe how we handle garbage by exploiting our reversible version of Forth, the Reversible Virtual Machine (RVM)[1]

The rest of the paper is organised as follows. In Section 2 we introduce simple examples which use both integer and floating point types, and we introduce the RVM dispensation for local variables. In section 3 introduce our main working example, a probabilistic algorithm for the calculation of π. We give a naive version of the algorithm, which ignores collection of garbage, and then show how this is "wrapped" in a program structure which preserves the result of the computation and then collects any garbage by reverse execution. In Section 4 we look at memory issues: we describe the memory organisation of the RVM and we calculate the memory requirements for our probabilistic algorithm, and see that for a given level of accuracy it may be necessary to exceed the memory capacity of our virtual machine. In Section 5 we present an algorithm that takes as input a required level of accuracy, calculates the sample size, and, if the computation space requirements exceed the available memory, will collect garbage during the computation. In Section 6 we draw our conclusions.

2 Simple floating point examples

We can define a constant to give the value of π with

```
3.14159 CONSTANT PI
```

and then print the value of `PI` with:

```
 PI F. <enter> 3.14159 ok
```

Here the Forth interpreter has been designed to leave a reference on the stack when it encounters a floating point literal in the input stream, and the definition of `F.` has been written to take a reference to a floating point number. However, since we never see, at this level, a floating point number as such, and all appropriate operations are written to use references to floating

[1]The RVM for Linux on the i386 Intel platform and its associated documentation is available from www.scm.tees.ac.uk/formalmethods

point values, we can safely think of the reference as *representing the floating point value*.

As a second example consider the following program which calculates the maturity value of a capital sum invested at a given annual rate of compound interest.

```
: MV ( r1 n r2 -- r3, calculate the maturity value of an in-
vestment of r1 for n years at annual compound interest of r2% )
  100. F+  100. F/ ( convert % rate to yearly ratio )
  SWAP >R SWAP ( P: ratio capital, R: years )
  R> 0 DO ( ratio capital )
    OVER F*
  LOOP NIP ;
```

Here we see Forth stack manipulation operations being used to manipulate both floating and integer values. To find the maturity value of 1000 currency units invested for two years at 5% we could use:

```
1000. 2 5. MV F. <enter> 1102.5 ok
```

We present this value again using a local variable approach. In our dispensation of Forth we use `VALUE` as a defining word *within definitions* to create local variables. Locals defined within `(: .. :)` brackets[2] are initialised from values which are in the stack at the point the stack frame is declared (generally at the start of the definition). Those values are matched against the declared locals from *left to right* (unlike the Forth Standard approach). Locals declared in the body of the procedure are initialised from stack values created by code which follows the stack frame. We also need to state how many stack cell are returned by the definition.

```
: MV ( r1 n r2 -- r3, calculate the maturity value of an in-
vestment of r1 for n years at annual compound interest of r2% )
  (: VALUE CAPITAL  VALUE YEARS  VALUE RATE :)
    RATE 100. F+  100. F/ VALUE RATIO
    YEARS 0 DO
      CAPITAL RATIO F* to CAPITAL
    LOOP
  CAPITAL 1LEAVE ;
```

Here we see that the same defining word can be used for both integer and real variables. We still need to distinguish integers and floats when it comes

[2]Our original work on local variables from 1985[7] uses { } brackets to enclose variable lists. However, since the RVM makes extensive use of sets, and these brackets are the standard math set brackets, we now reserve them for this purpose.

to performing operations on them however: if asked to calculate `1. 2. +` the system will add the references to the floating point values given, a totally meaningless computation.

The disadvantages of using a reference semantics in this context are, firstly a slight speed penalty associated with accessing data values via references, and secondly that we need to manage the memory locations in which the floating point values are stored: we need a garbage collector.

Our method of collecting garbage, previously suggested by Henry Baker[1] is based on the use of a reversible machine. Such machines, which preserve information at each step, are of theoretical interest because they minimise the absolute energy requirements of a computation[3, 5].

3 Example application: a probabilistic algorithm for the calculation of π

.

As a working example for the rest of the paper we present a probabilistic algorithm for estimating of the value of π. Consider Figure 1. If we choose a random value for x, uniformly distributed in the interval -1, to 1, and a random value for y, also uniformly distributed in the interval -1, to 1 we obtain a random point within the given square. The probability that this point will fall within the circle is given by the ratio of the areas of the circle and the square, namely $\pi/4$.

A simple way to estimate $\pi/4$ is therefore to choose n points, record the number S_n that fall within the circle, and calculate the estimate

$$\pi/4 \approx S_n/n$$

We use our random number generator `RAND`, which gives values in the range `0..MAXINT`, to provide a word that gives uniformly distributed floating point values within a given range `r1` to `r2`. We also make use of the word `S>F`, which converts a single precision integer to a floating point value.

```
: FUNIFORM ( r1 r2 -- r3,
  r3 is chosen from the uniform distribution between r1 and r2)
  (: VALUE r1  VALUE r2 :)
    RAND S>F VALUE frand
    RAND_MAX S>F VALUE frmax
    r1  r2 r1 F-  frand frmax F/  F*  F+
  1LEAVE ;
```

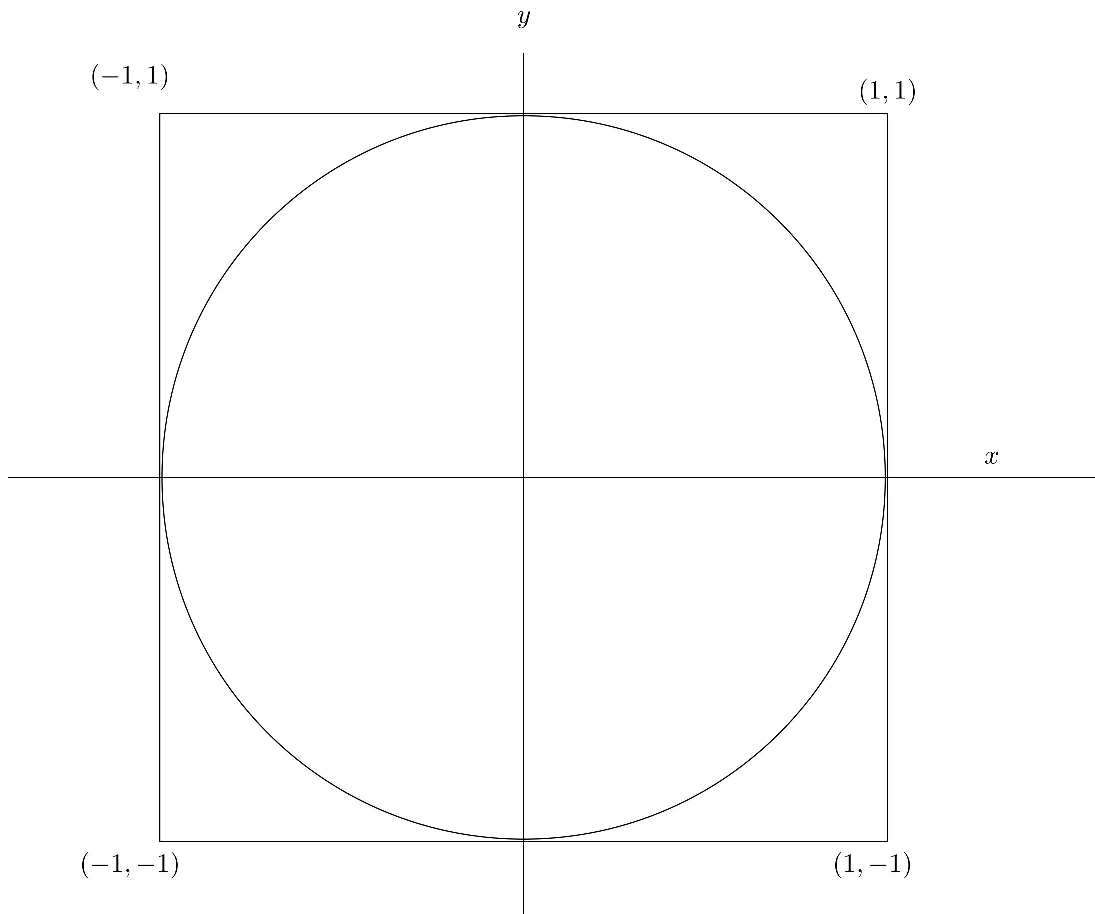


Figure 1: A circle of area π inscribed within a square of area 4.

Thus `-1. 1. FUNIFORM` will yield a random value between -1. and 1.

We can now code a naive word to estimate $\pi/4$. As input we give the sample size n on which the estimate is to be based. On each iteration of the loop we generate a random point (x, y) and calculate whether it is within the circle according to whether $x^2 + y^2 < 1$. If it is we add one to the count of "successes" S_n. Finally we calculate an estimate of $\pi/4$ using a floating point division of S_n/n.

```
: PI/4_0 ( n -- r, n<20000 and r is an estimate of pi/4 based on
a sample size n. Values of n>20000 may cause problems )
  (: VALUE n :)
    0 VALUE Sn  NULL VALUE x  NULL VALUE y
    n 0 DO
      -1. 1. FUNIFORM to x
```

```
    -1. 1. FUNIFORM to y
    x x F*  y y F*  F+ 1. F<
    IF Sn 1+ to Sn THEN
  LOOP
  Sn S>F  n S>F  F/
1LEAVE ;
```

We have called this program naive because it leaves garbage and because its sample size is limited by the memory configuration of the RVM. We first deal with the garbage by wrapping the definition within a programming structure which is particular to reversible computing. The effect of this structure was first described in terms of reversible Turing machines by Lecerf in 1963[6] and was independently given by Bennett[2], who also related it to the physics of reversible computation. Within the RVM it is formulated as follows. Let *prog* represent Forth code whose stack effect is to generate a floating point value. Then the structure:

`<RUN` *prog* `FLOAT>`

will run *prog*, preserve the floating point value created, reverse execution and thus undo all the other effects of *prog* (including removing the garbage that *prog* created) and finally leave the value created by *prog* on the stack. With this structure we can code a refinement of `PI/4_0` called `PI/4_1` which returns the same value as `PI/4_0` but collects garbage.

```
: PI/4_1 ( n -- r, as PI/4_0, but garbage is collected )
  <RUN DUP PI/4_0 FLOAT> NIP ;
```

this program is still limited in the sample size it can handle, and we now turn our attention to calculating the approximate sample size required for our algorithm to (probably) give us some given degree of accuracy.

4 Memory Issues

In our immutable references approach each floating point value created during a computation remains in memory until it is removed by reverse computation. The most general way to hold such memory references is on a heap, but the fixed size and atomic (unnested) nature of floating point values allows us to manage them more efficiently as a stack. Unlike the floating point stack of most implementations, items are only dropped from our floating point stack during reverse execution. The floating point stack is, in effect, a large floating point array, and this form of organisation places an absolute limit

on the number of floating point values that can be accommodated before a program needs to reverse.

The program developed so far is thus strictly limited in the sample size it can handle, and we now turn our attention to calculating the approximate sample size required to give us some given degree of accuracy with a given degree of confidence. The conclusions will be that much larger samples be needed than can be accommodated within the allocated floating point stack memory of the RVM.

We are interested in knowing the number of trials required to obtain an estimate of π which is accurate to within plus or minus some error e with (say) 95% confidence.

Note that if p is the probability of a particular trial falling within the circle, and S_n, the number of trials that fall within the circle when n trials are performed, then S_n follows a binomial distribution with mean $n * p$ and variance n*p*(1-p). For large n this binomial distribution approximates to the normal distribution with the same mean and variance.

Now in general, if X is a random variable with mean μ and variance σ^2 then $a * X$ is a random variable with mean $a * \mu$ and variance $(a * \sigma)^2$, thus S_n/n will have mean p and variance $p * (1 - p)/n$ and accordingly also standard deviation $\sqrt{p * (1 - p)/n}$. A normally distributed random variable has probability of just over 0.95 of being within ± 1.96 standard deviations of its mean. Thus to have a probability of 0.95 of S_n/n being within $\pm e$ of p we choose n to obtain:

$$e > 1.96 * \sqrt{\frac{p * (1 - p)}{n}}$$

Recalling that $p = \pi/4$ we have

$$e > 1.96 * \sqrt{\frac{\frac{\pi}{4} * (1 - \frac{\pi}{4})}{n}}$$

assuming a positive square root and squaring both sides

$$e^2 > 1.96^2 * \frac{\frac{\pi}{4} * (1 - \frac{\pi}{4})}{n}$$

multiplying each size by $\frac{n}{e^2}$

$$n > \left(\frac{1.96}{e}\right)^2 * \frac{\pi}{4} * \left(1 - \frac{\pi}{4}\right)$$

This tells us the minimum sample size for use with our random algorithm when a certain degree of accuracy is required, but this result depends on π

the value we are trying to calculate! However, if we insert a low estimate of π, say 3.1, we will obtain a high estimate of n since if we treat π as a variable the term $\pi * (4 - \pi)$ has a single maximum where $\pi = 2$.

The following word calculates the value of n as a double number.

```
: ?TRIALS ( e -- d, calculate no of trials d required to have
95% conf of obtaining result within pi-e to pi+e. We use the
crude (and low) estimate of 3.1 for pi in this calculation,
giving a higher value for d than is strictly required )
  (: VALUE e :)
    1.96 e F/ DUP F* ( (1.96/e)^2 )
    3.1 ( crude estimate of pi, but referred to as pi in stack
    comments )
    4. F/ ( (1.96/e)^2 pi/4 )
    1. OVER F- ( (1.96/e)^2 pi/4 1-pi/4 )
    F* F* F>D
  2LEAVE ;
```

Some example calculations showing the number of trial required for various degrees of accuracy are:

```
0.01 ?TRIALS D. 6474 ok
0.001 ?TRIALS D. 647493 ok
0.0001 ?TRIALS D. 64749356 ok
0.00001 ?TRIALS D. 6474935678 ok
```

And we note that the last value is beyond the range of 32 bit integers. We see that our algorithm is not a practical way to calculate a very accurate value. On the other hand it is a simple method of obtaining a rough estimate for π. Since the value of π has been known to over a million places for more than 30 years, the value of this algorithm is purely didactic, suggesting quick solutions to a range of problems where similar techniques can be employed.

5 Estimating π to a given level of accuracy

Our next refinement divides the calculation up into a number of separate experiments, each of which can be accommodated within the memory limitations of the RVM, assumed to be able to handle a sample size of 10000 before needing to reverse.

We use some global variables as an alternative to declaring a stack frame. Their longevity will give us increased visibility in case any debugging is required.

```
NULL VALUE PI_TOTALS  10000 CONSTANT INNER-LOOP
NULL VALUE OUTER-LOOP
```

In the following word the estimates of $\pi/4$ produced by each invocation of `PI/4_1` each use the maximum allowable sample size. These estimates are added together and kept in `PI-TOTALS`. The final total is divided by the outer loop count to find the average estimate of $\pi/4$.

```
: PI/4_2 ( n -- pi,
  produce an n estimate sample of pi/4. If n is too
  large for a single pass the calculation is split
  into a number of stages. Garbage from the outer
  loop is not collected yet )
  INNER-LOOP / 1+ to OUTER-LOOP
  0. to PI_TOTALS
  OUTER-LOOP 0 DO
    I SRAND ( seed pseudo random number generator )
    INNER-LOOP PI/4_1 PI_TOTALS F+ to PI_TOTALS
  LOOP PI_TOTALS OUTER-LOOP S>F F/ ;
```

In the above definition, we see that we are using a pseudo-random number generator. We must seed this with a different value for each iteration. The use of pseudo-random numbers makes our algorithm heavily dependent on the performance of the particular pseudo random number generator employed. Using gnu C library routine `rand`, for example, it is not possible to reliably obtain a result accurate to more than 4 or 5 significant figures.

We now present our final program, It takes as input a required level of accuracy. From this is calculates the required sample size, and will issue an error if more than `UMAXINT` trials are required. Otherwise is uses the `<RUN .. FLOAT>` wrapper to encapsulate the previous program and thus collects the garbage that program leaves. Finally it multiplies the estimate of $\pi/4$ by 4 to give an estimate of π.

```
: ESTIMATE_PI ( e -- r, produce a probabilistic estimate of pi
  which has probability 0.95 of being accurate to within an
  error of +/- e. All garbage is collected. If more than UMAXINT
  trials are required an error will be reported)
  <RUN
    DUP ?TRIALS ABORT" Too many trials needed by ESTIMATE-PI"
    PI/4_2
  FLOAT> NIP 4. F* ;
```

6 Conclusions

An immutable reference semantics allows us to deal neatly with data types, such as floating point, that cannot be held in a cell of memory. The three qualifications to this approach are that the data referenced must be immutable (so it is not a suitable approach for use with arrays), there will be a small speed penalty incurred, and the referenced values remain in memory as garbage until removed by reverse computation.

The availability of reverse execution allows us to collect garbage at no additional cost, and an appropriate program structure, particular to reversible computing, is available to do this in our Reversible Virtual Machine. Where a program has more extensive memory requirements than can be accommodated by a "one pass" algorithm, the application programmer must split its execution into stages, each of which returns information required for the final result, and each of which collects its own garbage. Thus garbage collection is best thought of as "semi-automatic" in our approach. A more general approach to re-use of memory resources during reversible computation is studied in [4]. Unlike the method given there, our approach, which is specific to a certain class of problems, requires virtually no additional execution overhead.

Since the value of π has been known to over a million places for more than thirty years, the value of the algorithm presented in this paper is purely didactic, suggesting simple solutions to a range of problems where similar techniques can be employed. As an example of such a problem consider a circle of radius r_1 with a second circle of radius r_2 constructed with its centre on the circumference of r_1. The problem is to find what proportion of each circle lies within the other.

References

[1] H G Baker. The Thermodynamics of Garbage Collection. In Y Bekkers and Cohen J, editors, *Memory Management: Proc IWMM'92*, number 637 in Lecture Notes in Computer Science, 1992.

[2] C Bennett. The Logical Reversibility of Computation. *IBM Journal of Research and Development*, 6, 1973.

[3] C Bennett. The Thermodynamics of Computation. *International Journal of Theoretical Physics*, 21 pp 905-940, 1982.

[4] C Bennett. Time-space Trade-offs for Reversible Computation. *SIAM Journal of Computing*, 18, 1989.

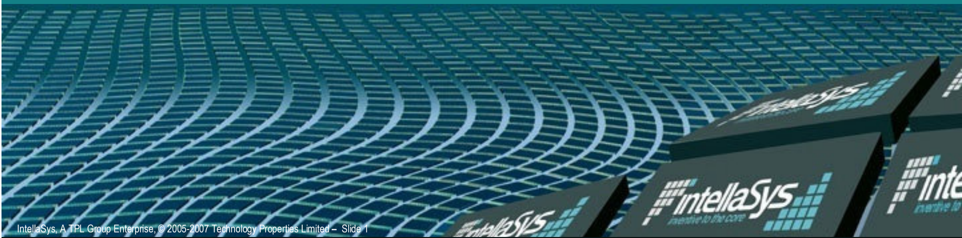
[5] R P Feynman. *Lectures on Computation.* Westview Press, 1996.

[6] Yves Lecerf. Machines de Turing Reversibles. *Comptes rendus de l'Académie Française des Sciences*, 257(1963), 1963.

[7] W. J. Stoddart. Readable and Efficient Parameter Access via Argument Records. *The Journal of Forth Application and Research*, 3(1), 1985.

[8] W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In M. A. Ertl, editor, *18th EuroForth Conference Proceedings*, September 2002. On-line proceedings.

Imagine a SEA™ of Processors



Remember when…

- TPL began development and marketing of proprietary product technology in 1988 – first licensee was Harris Corp in 1989
- IntellaSys was formed by TPL in 2003 to develop, build and sell high-speed multi-core processor arrays
- Alliacense was formed by TPL in 2004 to sell & manage proprietary product technology – Intel purchased the first license in 2005
- SEAforth-24 wins Portable Design Editor's Choice award for 2006

IntellaSys

The Towers – Cupertino City Center

- ~100,000 sq ft
- 200+ employees
- 8 Design Centers
  - Cupertino, CA
  - Irvine, CA
  - Tempe, AZ
  - Castle Rock, CO
  - Redding, CA
  - Cincinnati, OH
  - Vienna, Austria
  - Chennai, India
- International Sales Offices
  - Lugano, Switzerland
  - Taipei, Taiwan

You've probably been reading about us

2006 Press Coverage of SEAforth

Remember when…

- Embedded processors were just I/O “bit bangers?”
- When “high-speed serial link” meant 9600 baud?
- When the most complex algorithm chips had to run was calculating a CRC?
- Airliners had propellers?

Today’s processors…

- Still do all the I/O bit-banging of the old days, BUT NOW THEY HAVE TO DO…
- High-speed serial data links at GIGABIT rates
- Multiple algorithms at the SAME time
- Algorithms like H.264 and MPEG-II for HighDef TV

… and they do it in environments where…

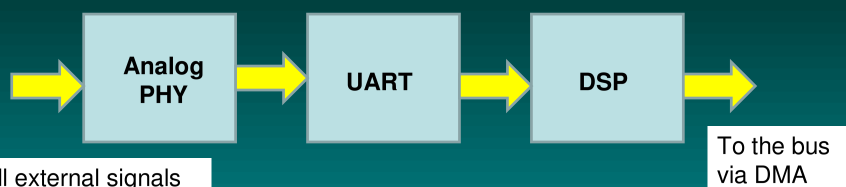
- Standards are changing faster than the chips
- Consumer demands dictate LOW cost with a bare minimum number of chips
- Applications are mobile and require low power

Problem with current embedded processors is the Processor Speed

- 400MHz CPU is far too slow
- Slow processor forced to pass all data and code over a single bus prevents the CPU from doing the I/O in a simple, programmed manner
- If you could speed the processor (and bus) up sufficiently the need for specialized I/O would largely go away

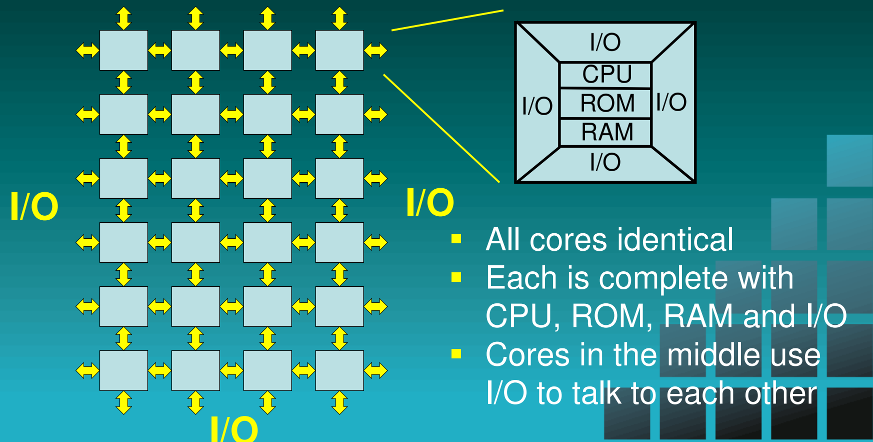
What is the REAL I/O Problem?

Typical Input requiring DSP



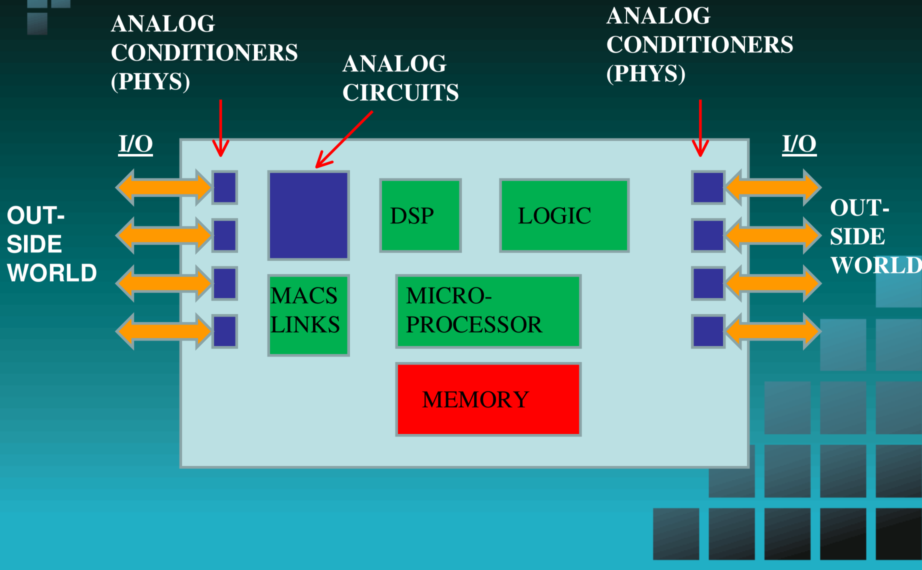
If the processor were fast enough to do the UART function AND act as a DSP AND do all of the other processing for all the other signals, there would be no need for a custom chip

There are no processors that fast, but… what if you put a whole bunch of them on one chip??

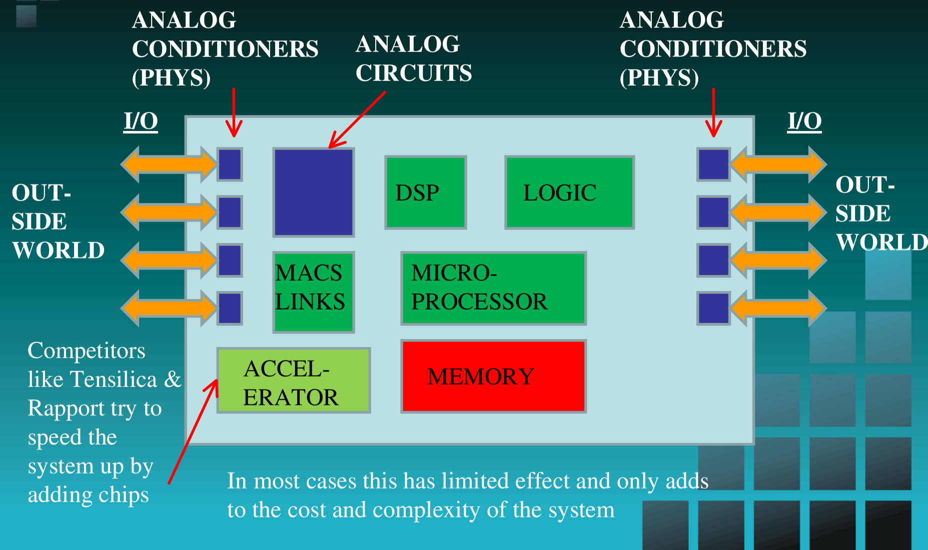


- All cores identical
- Each is complete with CPU, ROM, RAM and I/O
- Cores in the middle use I/O to talk to each other

System From the Top

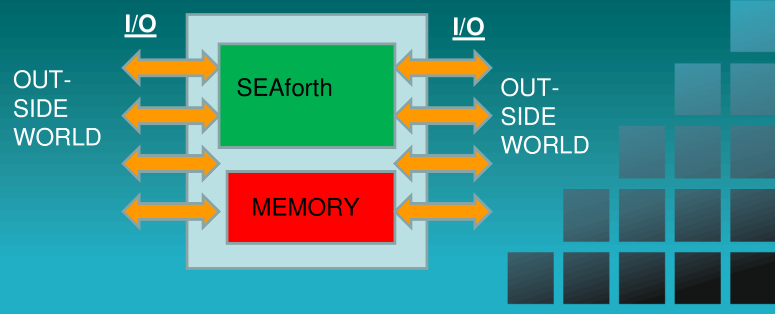


Competitors Try to Speed Up the System



SEAforth Goals

- Replace all chips except RAM
- Increase Performance
- Dramatic cost reduction
- Universal off-the-shelf parts



SEAforth Super Performance

- Each core runs at 1GHz… one Billion VentureForth instructions per second
- 24 cores = 24 Billion instructions per second
- No bus bottleneck because there is no bus
- No shared memory bottleneck because there is no shared memory
- Cores cooperate on tasks… use several as a DSP function, a handful for MP3 or H.264
- I/O pins are programmable for max flexibility

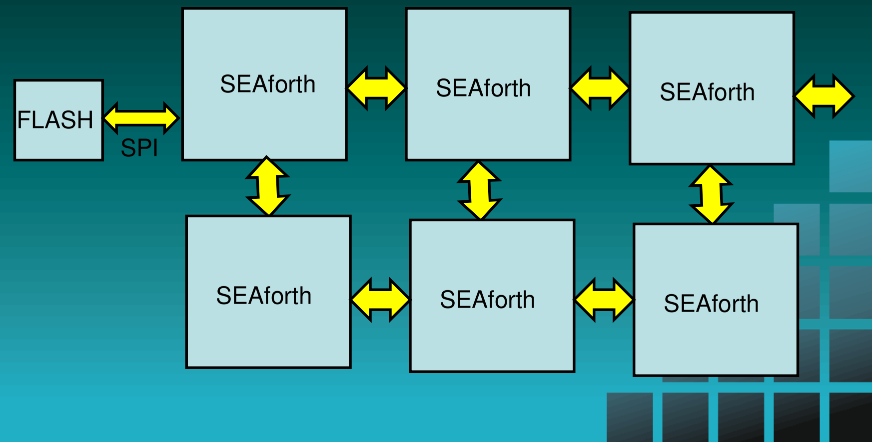
Programs in RAM

FLASH ⟷ SPI ⟷ SEAforth

- Customer applications code and data run from each core's RAM
- SEAforth loads code from FLASH during boot
- Ripple loads all of the cores with their appropriate code

- Feature changes occur in software in FLASH memory chips
- Application changes made WITHOUT mask changes / $$$ charges
- Same SEAforth chip can function as the System on a Chip (SOC) for customer's entire product line

Expanding the SEAforth Array



What is a core processor?

- 18-bit dual stack oriented engine – a Turing machine
- Runs VentureForth, a 32 instruction RISC language
- Each word contains four instructions
- Instruction execution time approx 1ns
- 64 - 128 words of ROM containing BIOS. Each core processor has its own version of the BIOS
- 64 - 128 words of RAM containing program and data
- Clock (each core processor has its own 1GHz ring oscillator)
- Four communications (and I/O) registers

Cores execute code from 3 places

- ROM BIOS
  - Default I/O configuration & drivers
  - Math and other “helper” routines
  - Routing routines for message and data
  - Each core has BIOS unique to it
- RAM
  - Applications code & data
- Ports
  - Boot from I/O port and pass data from core to core by executing transport code and then sending data
  - Run subroutines resident on a neighbor
  - Share data between neighbors

What can you do with 64 - 128 Words ?

- Each data word is 18 bits
- Each instruction word has FOUR instructions
  - 64 words = 256 instructions
  - 128 words = 512 instructions
- Executing from ports means sharing code
- Complex algorithms spread data over multiple cores
  - Matrix stores each row in separate core
- Incredible Code Packing Density
  - Random Number Generator … ONE instruction
  - FIR Filter – 22 instruction words
  - iDCT (Inverse Discrete Transform – 16 words

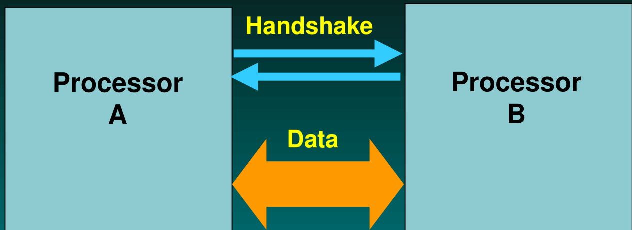
Processor cores used for…

- Algorithms and subroutines – dedicated processors mean no context switching time
- DSP functions like FFT and DFT
- “Interrupts” – assign a processor to each
- I/O ports driven by their processor which programs them to be SPI, SD, I^2C, a Real Time Clock or anything else you can think of
- Manage communication routes for communicating with other processors – be a “smart wire”

High-speed node to node communications

- Cores are constantly passing data back and forth, even sharing code
- Key to SEAforth efficiency is our straightforward way to do that
  - Nearest neighbors only
  - Extremely fast … 2ns per word transferred
- Multi-core algorithms require fast data transfers for video processing

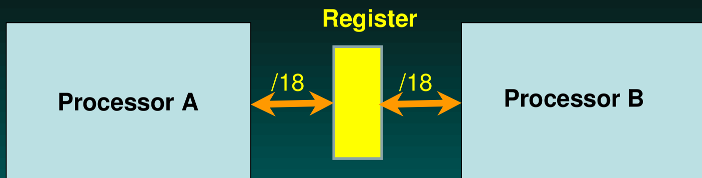
Traditional way of transferring data



- Loop on handshake lines until Processor B ready to receive data
- Send data
- Loop on handshake lines until Processor A ready to send next word
- More time spent in handshake loops than in data transfer

IntellaSys, A TPL Group Enterprise, © 2005-2007 Technology Properties Limited – Slide 21

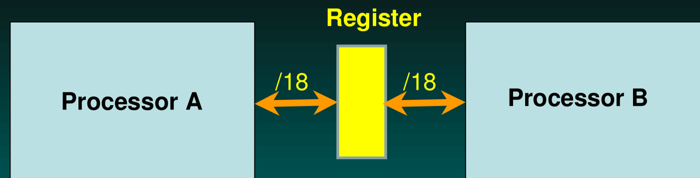
SEAforth data transfers



- Processor A assumes B is always ready for data… does Send data, Send data, Send data
- Processor B assumes A is always sending fresh data… does Get data, Get data, Get data
- 18-bit transfers every 2ns (if processors are actually ready)
- No performance penalty in sharing code and memory
- Ideal for dataflow type applications

IntellaSys, A TPL Group Enterprise, © 2005-2007 Technology Properties Limited – Slide 22

SEAforth - low power by design



- Processor B sleeps, ZERO power, until A has fresh data to transfer
- Processor A sleeps, ZERO power, until B is ready for more data
- Typically 2/3 of nodes are sleeping at any instant and dissipating ZERO power
- At next instant, still 2/3 of nodes are sleeping, but it will be completely different set of nodes
- 9mW per core while running, .04mW while sleeping

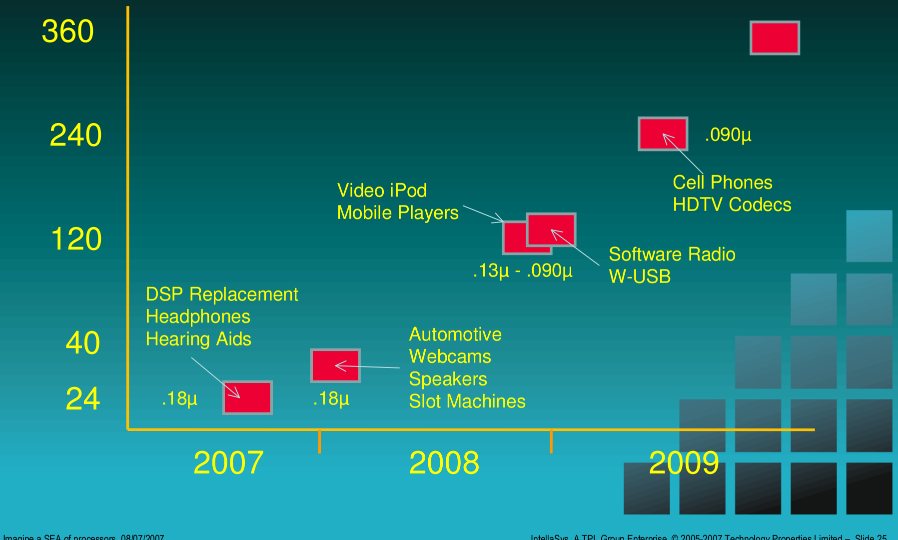
IntellaSys, A TPL Group Enterprise, © 2005-2007 Technology Properties Limited – Slide 23

SEAforth Family

- SF-24A … 24 cores w/64 words each of RAM and ROM, low cost part. Sampling via thumb drives in Q3 - 2007
- SF-48A … 48 cores w/128 words each of RAM and ROM and extensive I/O. Sampling via “thumb” drives in Q1 - 2008
- SF-IF1 Universal Interface Chip... USB, RS-232, SD, Memory Stick, and Flash drivers plus AES engine and two clock timers. Certified by USB and Sony (Memory Stick), etc. Available August. Provides customers with way to program and manage Flash on their production boards.

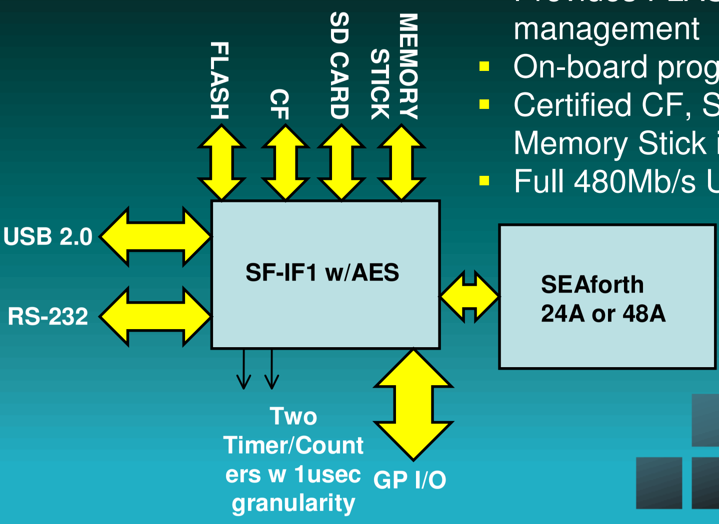
IntellaSys, A TPL Group Enterprise, © 2005-2007 Technology Properties Limited – Slide 24

SEAforth Roadmap



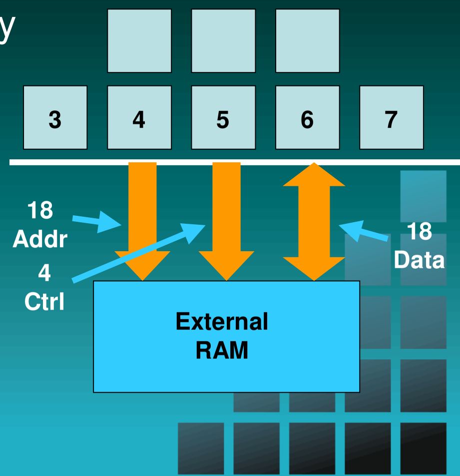
Using the SF-Universal Interface

- Provides FLASH management
- On-board programming
- Certified CF, SD, and Memory Stick interfaces
- Full 480Mb/s USB 2.0



SEAforth external memory

- 3 cores dedicated to handle external memory (static or dynamic)
- 1 Mword or 16 Mwords with paging
- 6 ns access
- Supplements internal RAM in each core
- Used for buffers
- Can also be used for large GP I/O interface



Analog functionality

- Two combination A/D and D/A converters
- Each can dynamically switch between A/D and D/A modes
- Can be used to xmit / receive wireless signals up to 20MHz
- 9-bit D/A drive inexpensive speakers and earphones directly
- 18-bit A/D for audio applicationsTw

Programming SEAforth

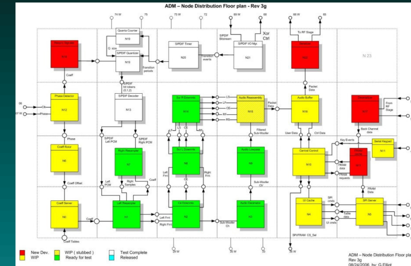
- VentureForth – RISC version of Forth
- 32 Instructions
- Four instructions per word
- microNEXT takes advantage of a four instruction "cache" by jumping back to the first instruction in a word implementing a fetch-less loop with spectacular performance in data pumps and when executing code directly from neighbors
- Portable Code
  - Can be moved from core to core, or from group of cores to group of cores
  - Can be moved from one chip to another
  - Replace two 24-core parts with one 48-core part

Software Modules for the SEAforth Library

- MP3 decode
- LCD screen drivers
- RF bidirectional links
- TCP/IP stack
- JPEG/GIF/PNG display
- H.264 codec (decode)
- ANS Forth interpreter
- Scriptable GUI generator

Building Applications

- Hook software modules together with glue code
- Library software modules
  - H.264, MP3, etc
- Build your own modules for your unique requirements
- With SEAforth , modules contain code AND virtual cores… move modules from chip to chip, from application to application, WITHOUT writing drivers and porting hardware
- Eliminates need for floor planning



Distributed Digital Media™

- SEAforth chips and code that implement a wirelessly connected consumer environment encompassing home theater and mobile devices
- Low cost, single chip implementations
- Reference designs
  - Complete circuitry and software code for demos
  - Initially provided as VentureForth files ready to download onto SEAforth universal evaluation board
  - May be put into physical prototype later
- Initial List *(but more coming)*
  - Wireless home theater speakers
  - Universal bidirectional remotes
  - Mobile video/audio players

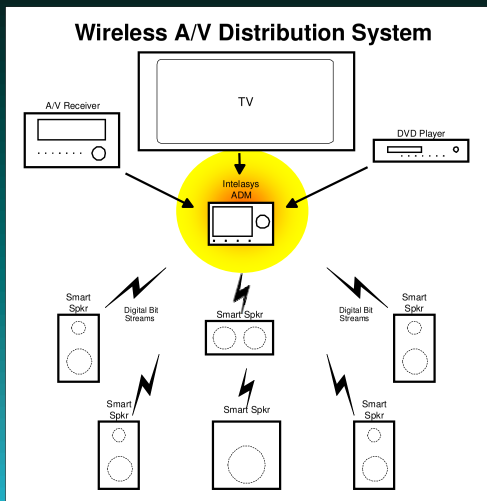
Wireless A/V Receiver

Distributed Digital Media™

- Power amplifiers and large power supply eliminated from A/V receiver
- Provides wireless bidirectional links to six speakers
- Also does MP3 decodes and TCP/IP access to file server

Typical Installation

- 7 SEAforth chips used, one in A/V receiver and one in each speaker
- Speakers extract the appropriate stream
- System can auto calibrate the room and move sweet spot



The SEAforth Advantage

- Low-cost single chip solution to extending wireless capability to speakers… one at A/V receiver and one at each powered speaker
- Eliminates crossover networks at speakers
- Adds MP3 player capability to receiver
- Adds TCP/IP capability to receiver for remote file server access
- Adds bidirectional RF link for SEAforth based remote control

Universal Remote Control

Distributed Digital Media™

- Bidirectional RF link to A/V receiver and home theater components
- Navigate content on RC screen – TV does not have to be ON
- Stereo surround sound headsets driven by RC

Mobile Video Player

Distributed Digital Media™

- H.264 Decoder
- MP3 decoder
- LCD color display
- USB 2.0
- RF link for downloading from file server
- Scriptable GUI

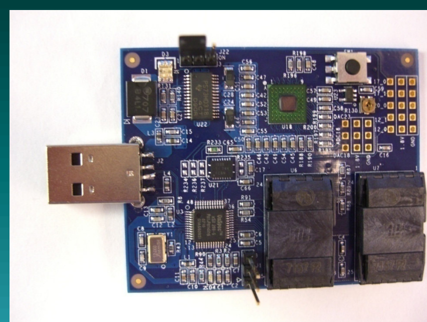
SEAforth Tools
T-18 Compiler / Simulator

- Free download from web right now
- Will be the basis of the SEAforth IDE
  - Write and simulate code on your PC
  - Download to SEAforth hardware platforms like Evaluation thumb drives and evaluation boards
- Develop and debug code today
- Access libraries of VentureForth functions and modules – FIR, DCT, FFT, etc
- Build menus and splash screens
- Windows GUI to be added January 2008
- MATLAB to be added Q1 - 2008

Evaluation Thumb Drive

- SEAforth chip + FLASH + USB Interface
- Plug into laptop and drive with T18
  - Compile code on T18
  - Load code into Thumb FLASH
  - Start SEAforth chip
  - Simple I/O and real time profiling
- Code development platform
  - Every member of your team has a thumb
  - Portable, on-the-go development
  - Adjunct to T18 software simulator
- Available Q3, 2007 (24 core)

Prototyping Boards



- Provide FLASH management plus USB interface
- SOME I/O available
- Low cost
- Customers use for demo purposes
- Derived from prototype of thumb drive

Evaluation Board

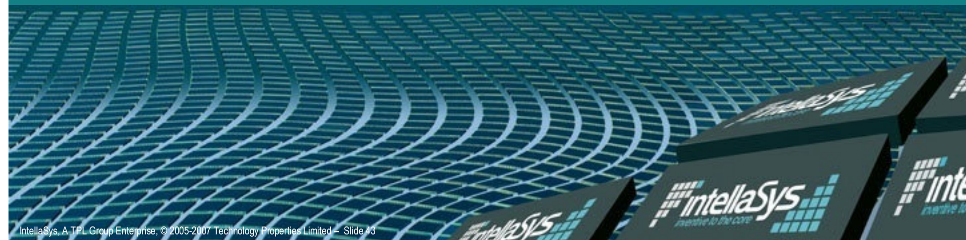
- Hardware prototyping tool
- On-board FLASH loaded and controlled from laptop hosting T18 via USB
- Complete platform for a wide range of applications
  - Four SEAforth chips
  - External RAM
  - Extensive analog support
  - Full-color LCD 640x480 VGA color screen
  - LCD touch screen
- Runs demo applications downloaded from web
- Available late Q4 - 2007

SEAforth applications vary widely

- Consumer apps like audio headsets, wireless home theater speakers, portable video players, remote controllers, etc
- Automotive applications like multiple bus interfaces, sensor drivers, dashboard controllers, etc
- Military/Aerospace applications like phased-array radars, satellites, remote sensor reporting
- Medical applications like hearing aids
- RF applications like direct RF conversion in the GigaHertz range

Imagine a SEA™ of Processors

www.intellaSys.net



Audio GUI: MINOS@work

A presentation

Bernd Paysan[1]

[1]Zetex GmbH

EuroForth 2007

Class–Z Amplifier

What I've been working on at Zetex in the last two years:

- Digital class–D amplifier system with feedback
- System consists of two chips: digital modulator + analog feedback
- The digital chip has many features and needs rather complex software to control it
- System developers aren't programmers, and therefore need a GUI. Must run under Windows (customer requirement). Must also run under Linux (my requirement).

Outline

1. Background
   - Class–Z Amplifier
   - Interfacing the Hardware
2. Implementation
   - Requirements
   - Hands on
   - Solutions
   - Graphic Equalizer
3. Things Learned

Interfacing SPI

- Audio input is via I²S or TDM[1], add a SPDIF decoder to the board, and you are done (SPDIF from PC or CD player)
- Register input is via SPI. Most MPUs have it, desktop PCs don't.
- FTDI has a chip to interface USB to SPI (FT2232C).
  - This comes with a driver and a library (both on Linux and Windows)
  - Basically is a serial port chip with a free programmable bitbang mode

[1]Time Division Multiplex

Requirements

- GUI access to all features
- Direct access as well
- Scriptable
- Save and restore state
- Obtain state from hardware
- Must run without hardware attached for demonstration issues

Solutions

State Remember everything written to the device in a list

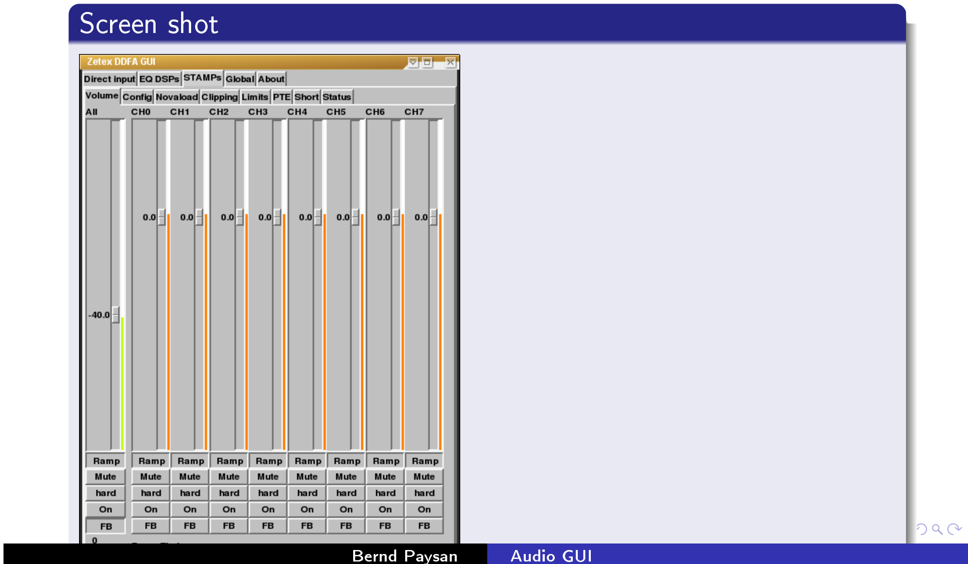
Run without hardware Bypass library, read from the device list (0 if none)

MINOS Added flag actor for bit in a bit field (very useful to represent hardware registers)

Performance Block readouts from the device, cache information, update periodically

Hands on

Screen shot



Graphic Equalizer

Most complicated part

- EQ DSP provides up to 14 4th order biquad filters, 12 are used
- Individual filters are calculated to have xdB gain/attenuation at target frequency, and constant Q (so that the gain is $\frac{x}{2}$dB at half or twice the frequency
- Equalizer coefficients are not independent of each others
- Approximative solution: Linear equation system to be solved
- Visualization: Simulate impulse response, FFT the result, an draw it on a double logarithmic scale

Things Learned

- Good exercise to debug Theseus
- Added features to MINOS (like scale factor to sliders and bit–wise state for toggle buttons)
- Components were a good idea (each register is a component)
- Alternative to Theseus–based UI might be to create the GUI out of a formal register description (in Forth syntax)
- Cross–plattform development can work even close to hardware

A FRAMEWORK FOR DATA STRUCTURES IN A TYPED FORTH

Federico de Ceballos

Universidad de Cantabria
federico.ceballos@unican.es

September, 2007

Strong Forth as a typed Forth

In Strong Forth the interpreter knows the types associated with the elements in the stack. The compiler takes care that a word's behavior is consistent with its declared specification. This has to be unique.

```
SWAP ( SINGLE SINGLE -- 2ND 1ST )
```

Because every word has a declared input diagram, overloading is possible.

```
SWAP ( DOUBLE DOUBLE -- 2ND 1ST )
```

If you do need complete control, you can use a cast.

```
CHAR 0 .                  \ prints 0
CHAR 0 CAST SINGLE .      \ prints 48
```

Strong Forth's type hierarchy

The language comes with several families of types:

```
SINGLE     INTEGER          UNSIGNED
                            SIGNED
                            CHARACTER
           LOGICAL          FLAG
           TOKEN
           MEMORY-SPACE
           FILE

DOUBLE     INTEGER-DOUBLE   UNSIGNED-DOUBLE
                            SIGNED-DOUBLE
           CONTROL-FLOW
           DATA-TYPE        STACK-DIAGRAM
           DEFINITION       COLON-DEFINITION

FLOAT
```

Strong Forth's type hierarchy (2)

```
SINGLE     ADDRESS          DATA
                            CONST
                            PORT
                            CODE
                            CADDRESS        CDATA
                                            CCONST
                                            CPORT
                                            CCODE
                            SFADDRESS       SFDATA
                                            SFCONST
                                            SFCODE
                            DFADDRESS       DFDATA
                                            DFCONST
                                            DFCODE

DOUBLE     FAR-ADDRESS      CFAR-ADDRESS
                            SFFAR-ADDRESS
                            DFFAR-ADDRESS
```

Forth's advantages

Usability.

Interactive development.

Efficiency (both in terms of development time and generated code).

Possibility of accessing low-level resources.

Little constraints.

Intimate knowledge of your development environment.

Typeless. (A language feature, not a bug.) ;-)

Variations from the Strong Forth model

Strong Forth is aimed to embedded systems with multiple address spaces, we are targeting a PC with a single address space.

Strong Forth tries to keep the Forth *flavour* as much as possible, we plan to take advantage of the new possibilities offered.

The idea of having different base pointers in order to navigate through different data sizes is abandoned.

Data definitions in Forth

```
VARIABLE FIRST

2VARIABLE SECOND

FVARIABLE THIRD

SFVARIABLE FOURTH

DFVARIABLE FIFTH

CREATE SIXTH 3 CELLS ALLOT

23 VALUE THIS


CLASS POINT
   VARIABLE X
   VARIABLE Y
END-CLASS

POINT BUILDS MY-POINT
```

Data definitions in Forth

```
23 FIRST !

SECOND 2@

1.23E THIRD F!

1.23E FOURTH SF!

1.23E FIFTH DF!

1 2 3  SIXTH  TUCK !  CELL+ TUCK !  CELL+ !


45 TO THIS


12 MY-POINT X !
14 MY-POINT Y !

14 12 MY-POINT  TUCK X !  Y !       \ An error !!!
```

Basic data types

The Strong Forth system provides the following basic data types from which the rest are derived:

```
SINGLE    A single-cell data type, used to hold a number or an
          address.

DOUBLE    A double-cell data type, used to hold a double number or
          else a couple of single-cell items working together.

FLOAT     A data type wide enough to hold a floating point number.
          In a 16-bit system with 64-bit reals, a FLOAT would
          occupy four cells.
```

In addition, we are using the following basic type:

```
TRIPLE    A data type three cells wide, used to hold three single-
          cells items working together.
```

User data types

```
CHAR        1 BYTE

INT         1 CELL
UINT        1 CELL (UNSIGNED)
LONG        2 CELLS
ULONG       2 CELLS (UNSIGNED)

REAL        1 FLOAT
COMPLEX     2 FLOATS
VECTOR      3 FLOATS
QUATERNION 4 FLOATS
```

User data types

```
C&        An address to a read-only element.
D&        An address to an element (derived from the type above).

C[]       An address to an array of read-only elements, together
          with the number of elements.
D[]       An address to an array of elements, together with the
          number of elements (derived from the type above).

C[,]      An address to a two dimensional array of read-only
          elements, together with the number of rows and columns.
D[,]      An address to a two dimensional array of elements,
          together with the number of rows and columns (derived
          from the type above).
```

C& points to an element that can be read. D& points to an element that can also be modified. D& is derived from C&, as you can do with a normal pointer everything you would do with a pointer to constant plus a few new things.

Categories

All categories allocate enough memory to hold a value of the given type.

```
VAR       No action associated with the word. When it is executed,
          its address is returned.

AUTO      When the word is executed, its address is returned and
          the @ word is applied to it. After the definition, the !
          word is applied to its address.

CONST     When the word is executed, its constant address is
          returned and the @ word is applied to it. After the
          definition, the ! word is applied to its address as if
          it were not constant.
```

CONST words cannot be used inside a struct. AUTO words are not initialised inside a struct.

Manipulators

A manipulator is a state smart word that fetches the next word in the input stream (that has to be an instance of one of the categories given above) and

```
SIZEOF    Returns the size of the object measured in address
          units.

ADDR      Returns the starting address of the object.

TO        Applies the word ! to the object.
```

Examples:

```
+10 CONST INT 5*2

VAR INT FIRST
-5 AUTO INT SECOND

SIZEOF FIRST                  \ returns 4 in a 32-bit system
+10 FIRST !   FIRST @         \ returns +10
+10 TO SECOND   SECOND        \ returns +10
+10 ADDR SECOND !   SECOND    \ returns +10
```

One dimensional arrays

A vector is a collection composed of a fixed number of elements of the same base type. Each of the elements can be used to hold a piece of data and the vector can be manipulated as a whole.

```
10 [] INT DISTANCES   \ defines the vector

SIZEOF DISTANCES      \ returns 40 in a 32-bit machine

DISTANCES .           \ prints the value of all elements
```

The following words should be provided for an array of any of the basic types:

```
SIZE  ( d[] -> type -- uint )
      Returns the number of elements in the array.

@     ( d[] -> type uint -- 2nd )
      Fetches one of the elements in the array. An exception is thrown is
      the index is equal or greater than the number of elements in the
      array.

!     ( type d[] -> 1st uint )
      Sets one of the elements in the array. An exception is thrown is the
      index is equal or greater than the number of elements in the array.
```

One dimensional arrays

```
HEAD  ( c[] -> type uint -- 1st )
      Returns an array with some of the first entries in the original
      array. If the number is greater than the entries in the array, the
      whole array is returned.

TAIL  ( c[] -> type uint -- 1st )
      Returns an array with some of the last entries in the original array.
      If the number is greater than the entries in the array, the whole
      array is returned.

-HEAD ( c[] -> type uint -- 1st )
      Returns an array in which some of the first entries in the original
      array have been removed. If the number is greater than the entries in
      the array, an empty array is returned.

-TAIL ( c[] -> type uint -- 1st )
      Returns an array in which some of the last entries in the original
      array have been removed. If the number is greater than the entries in
      the array, an empty array is returned.

CLONE ( c[] -> type -- d[] -> 2nd )
      Returns a new array with the same data as the given one.
```

Two dimensional arrays

The following words are modelled after their one dimensional counterparts:

```
SIZE  ( d[,] -> type -- uint uint )

@     ( d[,] -> type uint uint -- 2nd )

!     ( type d[,] -> 1st uint uint )

HEAD  ( d[,] -> type uint -- 1st )

TAIL  ( d[,] -> type uint -- 1st )

-HEAD ( d[,] -> type uint -- 1st )

-TAIL ( d[,] -> type uint -- 1st )

CLONE ( d[,] -> type -- 1st )
```

In addition, we have the following word:

```
ROW   ( d[,] -> type uint -- d[] -> 2nd1 )
      Returns an array with one row of the original array.
```

Strings

A string can be defined as a one dimensional array of characters as in:

```
50 [] CHAR ADDRESS
```

However, as the Forth language allows for direct use of strings, some convenience can be provided.
The keyword STRING defines a constant array of chars with the length of its initialiser:

```
" This is some text" STRING MY-TEXT
```

Structures

```
STRUCT POINT-2D
     VAR INT X
     VAR INT Y
END

POINT-2D  A basic type with an associated size of 8 bytes (in a 32
          bit machine).

X         ( C& -> POINT-2D -- 1ST -> INT)

          The returned address is the same as the parameter.

Y         ( C& -> POINT-2D -- 1ST -> INT)

          The returned address is the parameter incremented in 4
          bytes.
```

Using structures

```
STRUCT DATE
   VAR UINT DAY
   VAR UINT MONTH
   VAR UINT YEAR
END

: ! ( UINT UINT UINT D& -> DATE ) >R
    R@ YEAR !  R@ MONTH !  R> DAY ! ;
: @ ( C& -> DATE -- UINT UINT UINT ) >R
    R@ DAY @  R@ MONTH @  R> YEAR @ ;

: 00 ( UINT ) <# # # #> TYPE ;
: . ( C& -> DATE ) @ ROT 00  '/' EMIT  SWAP 00  '/' EMIT  . ;

15 9 2007 CONST DATE TODAY

TODAY MONTH .              \ prints 9
```

Derived structures

A new structure can be a subtype of another type. In this case, the newer one has the size of the old one (augmented with the size of new components) and can use any word that apply to the old one (unless a new overloaded one is defined).

```
STRUCT POINT-2D
     VAR INT X
     VAR INT Y
END

DERIVED POINT-2D POINT-3D
     VAR INT Z
END

: . ( C& -> POINT-2D ) DUP X .  Y . ;
: . ( C& -> POINT-3D ) DUP .  Z . ;
```

Privacy matters

In an OOP language, the following concepts may be available:

```
PRIVATE    A method that can be used only inside the class.

PROTECTED  A method than can be used inside the class and also
           inside derived classes.

PUBLIC     A method without privacy constraints.

FRIEND     An external method that can access all method in the
           class.
```

In Forth, the use of word lists or packages caters for all these possibilities and then some more.

Future lines of work

The distinction between normal manipulators (that increment the data pointer) and manipulators inside a structure (that increment the structure size) can be complemented with USER manipulators (the increment the offset into the user data space).

```
USER VAR INT UNO

USER AUTO LONG DOS
```

If it were possible to derive structures from a TAGGED one, the tab could be used at runtime to choose the exact function that should be called. This modification is not trivial.

```
TAGGED FIRST
     VAR INT X
END

DERIVED FIRST SECOND
     VAR INT Y
END
```

How to deal with context

M. Anton Ertl
TU Wien

How to write hex.

```
: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

How to write hex.

```
: hex.
  hex u. ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

How to write hex.

```
: hex.
  hex u. decimal ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

How to write hex.

```
: hex.
  base @ >r hex u. r> base ! ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

How to write hex.

```
: hex.-helper
  hex u. ;

: hex.
  base @ >r ['] hex.-helper catch r> base ! throw ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

How to write hex.

```
: hex.
  base @ { oldbase }
  TRY
    hex foo \ now the hex is placed correctly
    0       \ value for throw
  RESTORE
    oldbase base !
  ENDTRY
  throw ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

Get and Set?

GET-CURRENT SET-CURRENT

Has certain benefits, but not for context problem

Context wrapper

```
: hex.
  ['] u. $10 base-execute ;

: foo
  ... hex. ...
  ... .    ... ;

decimal foo
hex foo
```

Other contexts

```
outfile-execute
infile-execute
```

Other languages: Postscript

```
<< /base 16 >> begin ... end
```

Other languages: Lisp

Dynamically scoped variables

Other languages: Unix environment variables

```
env LANG=DE_at.utf-8 gforth
```