26th EuroForth Conference

September 24-26, 2010

Haus Rissen
Hamburg
Germany

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 26th EuroForth finds us in Hamburg for the first time. The three previous EuroForths were held in Schloss Dagstuhl, Germany (2007), in Vienna, Austria (2008) and in Exeter, England (2009). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

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

For the refereed track, two papers were submitted, and both were accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 11 submissions, 7 accepts, 64% acceptance rate. Each paper was sent to at least three program committee members for review, and they all produced reviews. One refereed paper was co-authored by me (the primary program committee chair); Ulrich Hoffmann served as acting program chair for this paper, and these reviews are anonymous for me. The other paper was co-authored by a program committee member, and the reviews of that paper are anonymous to him as well. I thank the authors for their papers, the reviewers for their reviews, and Ulrich Hoffmann for serving as secondary chair.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. In addition, the printed proceedings include slides for talks that will be presented at the conference without being accompanied by a paper and that were submitted in time.

These online proceedings also contain late presentations that were too late to be included in the printed proceedings. Also, some of the presentations included in the printed proceedings were updated to reflect the slides that were actually presented. Workshops and social events complement the program.

This year's EuroForth is organized by Klaus Schleisiek and Ulrich Hoffmann.

Anton Ertl

Program committee

Sergey N. Baranov, Motorola ZAO, Russia
M. Anton Ertl, TU Wien (chair)
Ulrich Hoffmann, FH Wedel University of Applied Sciences (secondary chair)
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart, University of Teesside
Reuben Thomas, Adsensus Ltd.

Contents

Refereed papers

M. Anton Ertl, David Kühling: `ABI-CODE`: Increasing the Portability of Assembly Language Words 5
Campbell Ritchie, Bill Stoddart: A Compiler which Creates Tagged Parse Trees and Executes them as FORTH Programs 15

Non-refereed papers

S. N. Arhipov, N. J. Nelson: Securing a Windows 7 Public Access System Using Forth 31
James Bowman: J1: A Small Forth CPU Core for FPGAs 43
Manfred Mahlow: Using Glade to Create GTK+ Applications with FORTH 47

Presentations

Andrew Haley: Forth Concurrency for the 21st Century 53

Late presentations

Klaus Schleisiek: uCore progress (with remarks on arithmetic overflow) . 60
Bernd Paysan: net2o: vapor → reality 64

Late papers

Gerald Wodni, M. Anton Ertl: The Forth Net 68

`ABI-CODE`: Increasing the portability of assembly language words

M. Anton Ertl*
TU Wien

David Kühling

Abstract

`Code` words are not portable between Forth systems, even on the same architecture; worse, in the case of Gforth, they are not even portable between different engines nor between different installations. We propose a new mechanism for interfacing to assembly code: `abi-code` words are written to comply with the calling conventions (ABI) of the target platform, which does not change between Forth systems. In the trade-off between performance and portability, `abi-code` provides a new option between `code` words and colon definitions. Compared to `code` words, the `abi-code` mechanism incurs an overhead of 16 instructions on AMD64. Compared to colon definitions, we achieved a speedup by a factor of 1.27 on an application by rewriting one short colon definition as an `abi-code` word.

1 Introduction

`Code` words are not portable between Forth systems, even between Forth systems running on the same architecture[1]. The main reason for that is that there are no standard registers for the stack pointers.

For Gforth[2], the situation is even worse: Because it uses GCC to build its inner interpreter, and GCC decides the register allocation on its own, `code` words are not even portable between Gforth installations[3] and engines (in particular, not between `gforth` and `gforth-fast`).

In this paper, we describe the new `abi-code` facility of Gforth that allows writing code in assembly language that is portable between different Gforth installations and engines. If other Forth systems implement this facility, too, it could enable porting assembly language code to other Forth systems running on the same platform.

2 Basic Idea

2.1 `abi-code`

`Abi-code` words are called according to the calling convention of the platform, passing and returning the stack pointers through parameters. The calling convention is usually described in the application binary interface (ABI) documentation of the platform, leading to the name `abi-code`.

The data-stack pointer is passed as first parameter, and is returned as result. An address to a memory cell containing the FP-stack pointer is passed as second parameter, and the FP-stack pointer is returned by storing the changed value in this memory cell; if the FP stack is not accessed, the second parameter can be ignored. In C terms, an `abi-code` word has the following prototype:

```
Cell *word(Cell *sp, Float **fp_pointer)
```

The stack layout and the sizes of the stack items are also relevant: In Gforth both data and FP stack grow towards lower addresses, and the sizes of the items on the stack are the same as in memory (i.e., `1 cells` and `1 floats`).

Here is an example of using `abi-code` on Linux-AMD64[4]:

```
abi-code my+  ( n1 n2 -- n3 )
\ SP passed in rdi, returned in rax
lea rax,[rdi+8] \ new sp in result reg
mov rdx,[rdi]   \ get old tos
add [rax],rdx   \ add to new tos
ret             \ return from my+
end-code
```

To make our examples easier to read, we present them in Intel syntax (destination first) rather than the syntax of the Gforth assembler. You can find a version of this example in Gforth syntax (so you

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

[1]We will use this notion of portability between Forth systems running on the same architecture in the rest of this paper.

[2]Gforth is a fast and portable Forth implementation. It achieves portability by for creating the machine code of the primitives with a C compiler.

[3]In particular, Gforth 0.6.2 compiled with one version of GCC is not necessarily compatible with the same Gforth version compiled with another version of GCC; also, Gforth 0.6.2 configured with explicit register allocation (`--enable-force-reg`) is not necessarily compatible with the same Gforth version configured without this option. These problems should be less frequent in Gforth 0.7.0, because there explicit register allocation is tried by default.

[4]Unfortunately, Windows uses a different convention on AMD64

can try it out) in the Gforth manual (development version[5]).

Most calling conventions pass the parameters in registers, but IA-32 calling conventions usually pass them on the architectural stack, and therefore require slightly more overhead:

```
abi-code my+
mov eax,4[esp] \ sp in result reg
mov ecx,[eax]  \ tos
add eax,#4     \ update sp (pop)
add [eax],ecx  \ sec = sec+tos
ret            \ return from my+
end-code
```

And here is an example of an FP `abi-code` word on Linux-AMD64:

```
abi-code my-f+
mov  rdx,[rsi]       \ load fp
fld  qword ptr[rdx] \ r2
add  rdx, 8          \ update fp
fadd qword ptr[rdx] \ r1+r2
fstp qword ptr[rdx] \ store r
mov  [rsi],rdx       \ store new fp
mov  rax,rdi         \ sp in result reg
ret                  \ return from my-f+
end-code
```

Here we have some extra overhead, because the FP stack pointer `fp` is passed in and out in a memory location; also, here we do not need to update the data stack pointer `sp`, so moving `sp` to the result register requires a separate instruction.

Unlike ordinary `code` words, `abi-code` words need a special routine to invoke them out of a threaded-code inner interpreter. The definition of `abi-code` in `gforth-fast` on Linux-AMD64 is equivalent to the following:

```
: abi-code ( "name" -- )
  create also assembler
;code ( ... -- ... )
\ doabicode routine
mov   [r15],r14    \ 1)store TOS to memory
mov   rdi,r15      \ 2)sp to 1st arg reg
movsd [r12],xmm8   \ 1)store FP TOS
lea   rax,[r9+10H] \ 3)body of name
mov [rsp+6b0H],r12 \ 2)store fp in memory
lea rsi,[rsp+6b0H] \ 2)2nd arg: fp address
call  rax          \ 4)call name's body
mov rdx,[rsp+6b0H] \ 2)load fp
mov   r14,[rax]    \ 1)load TOS
mov   r15,rax      \ 2)copy sp to sp reg
movsd xmm8,[rdx]   \ 1)load FP TOS
mov   r12,rdx      \ 2)copy fp to fp reg
NEXT               \ 5)threaded dispatch
end-code
```

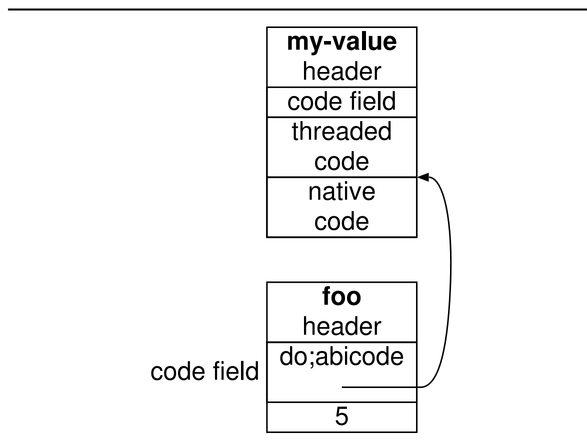


Figure 1: Code field layout for `;abi-code`-defined words

The `doabicode` routine consists of the following components (the numbers in the comments above refer to the component numbers):

1. Saving the tops of the stacks to memory to comply with the memory-stack convention of `abi-code` words; and loading the tops of the stacks back into registers afterwards. This is needed because `gforth-fast` keeps the tops of both the data stack and the FP stack in registers (r14 and xmm8).

2. The FP stack pointer is stored in memory, and argument registers are set up. And after the call the new stack pointers are moved to the registers of sp (r15) and fp (r12).

3. The address of the called routine is computed (it starts at the body address, which is computed by adding 2 cells (10H) to the CFA in r9.

4. The actual call.

5. Invoke the next primitive (NEXT).

Gforth also contains a primitive `abi-call` that is used for invoking `abi-code` words (primitive-centric code [Ert02][6]); it has the same components except that the address of the called routine is found as immediate argument of the primitive, not through the CFA.

This overhead will probably be a little lower in native-code compilers, but most of the components will be there too.

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

[6] In primitive-centric threaded code every non-primitive (colon definition, constant, etc., and `abi-code` words) is compiled to a primitive followed by an immediate argument.

2.2 ;abi-code

;abi-code is to ;code what abi-code is to code. The routine after ;abi-code is passed a third parameter: the body address of the word for which the routine provides the behaviour. In C terms, the routine has the following prototype:

```
Cell *word(Cell *sp, Float **fp_pointer,
           char *body)
```

Here is an example of using ;abi-code on Linux-AMD64:

```
: my-value ( w "name" -- )
  create ,
;abi-code ( -- w )
  \ sp in rdi, address of fp in rsi
  \ body address in rdx, temp reg: rcx
  lea rax,[rdi-0x8] \ new sp in return reg
  mov rcx,[rdx]     \ load value from body
  mov [rax],rcx     \ store value on stack
  ret
end-code

5 my-value foo
```

Unlike for ;code routines, we cannot use the start address of this routine as code address in the CFA of an indirect-threaded Forth. Instead, we have to use a solution like the one we use for does>-defined words: The code address points to a routine do;abicode that calls ;abi-code routines. It finds the address of the routine to call in the cell right after the CFA (see Fig. 1). Gforth has two cells for the xt field, the second being used for does>-defined and ;abi-code-defined words. The do;abicode routine and the ;abi-code-exec primitive contains components similar to the doabicode routine shown above.

3 Discussion

3.1 Compared to code

The main disadvantage of abi-code words compared to code words is that they have additional calling overhead. We will look at the performance difference resulting from this overhead in Section 4.

The advantage of the abi-code approach is that it provides a simple and stable interface.

For programmers the advantage of that stability is that their assembly language words are portable between Gforth engines (gforth, gforth-fast, gforth-itc), and portable across installations (in particular, indepedent of the GCC version used). If abi-code was implemented by other Forth systems, abi-code words could be written to be portable across systems.

For the system implementor, the advantage of the stable interface is that the system is not tied to using the same register assignments internally forever. It can change, e.g., the number of stack items in registers, or move the data stack pointer to a different register if that results in faster code on a new processor.

The simplicity helps programmers by not having to learn and remember top-of-stack registers that the system may use internally; and it helps the system implementor by not having to teach that to programmers. Indeed, Mitch Bradley once commented that he went back from keeping the TOS in a register to keeping all stack items in memory in order to provide a simpler interface to the users. Abi-code provides the same benefit at a lower cost: we pay the additional overhead only when executing an abi-code word, not in the whole system.

3.2 Why use the ABI?

Why did we choose to use the ABI? Couldn't the same benefits not be achieved with another approach?

The major reason we chose to use the ABI is that it is easy to get GCC to generate an ABI call. There are some other benefits, however:

- We can use this facility to call functions written in non-assembly languages that conform to the ABI; these functions would have to be written to access the stacks, though. Indeed, we will probably modify the implementation of the libcc C interface [Ert07] to use this mechanism rather than the less refined calling mechanism it uses now.
- For each platform, the ABI is already given, so the system implementor does not need to decide what the interface should be. As long as there is only one system involved, this is not a particular advantage, but as soon as several systems implement a common interface, they would have to standardize on a common interface for each platform they support, and as anybody witnessing a standards process knows, that tends to be rather time-consuming. And that's the best case; instead, each vendor might go their own way on a new platform, so the advantage of a common interface would disappear.

There is also a disadvantage to using the ABI: ABIs often require passing the stack pointers in ways that are suboptimal. E.g., in our AMD64 example, the data stack pointer is passed in in a different register than it is passed out, and on IA-32 it is even passed in through memory; and the limitations of common ABIs have led us to pass the

FP stack pointer through memory in any case (see Section 3.4).

3.3 Alternatives

An alternative would be to have conventional `code` words, but supply macros that switch from the system's register-and-stack setup to a fixed register setup and back, just like the `NEXT` macro invokes the next word, whether the system is direct-threaded, indirect-threaded or subroutine-threaded.

In terms of overhead for a given interface this approach would save relatively little compared to an `abi-code`-like implementation: Only the computation of the call target, the call itself and the return would be eliminated.

One reason why we did not choose this approach is that we have no good way to get the register allocation for Gforth's engines out of GCC and into these macros; in contrast, with `abi-code` GCC does the setup of the call and the restoration for us.

Similarly, we could use `code` words, but abstract away from the concrete register allocation of a Forth system and the concrete implementration of the stack by providing macros for accessing the stacks and/or for the logical registers (e.g., stack pointers, temporary registers); if several systems implement the same macros, code may be portable between them even if their register allocation differs.

A problem with this approach is that some systems keep the top-of-stack in a register and others in memory. On most architectures you cannot use a memory access wherever you can use a register, so it would be tricky to set up the macros such that they can be implemented on all systems. Moreover, we again cannot use this approach for Gforth, because we have no automatic way to get the actual register allocation out of GCC and into these macros. And, to achieve true portability, Forth vendors would have to agree on the macros (and these may be architecture-specific, e.g., how many temporary registers are usable by `code` words), whereas someone else has already standardized the ABI.

Another approach that might be implementable in Gforth would be to do the setup and call in C code with `asm()` statements. This would allow us to use an arbitrary interface, not limited by the ABI.

A problem of this approach is that there is no guarantee that GCC plays along; it could run into a situation where it cannot allocate registers and would then fail to build Gforth. Or it could produce an abysmal register allocation that slows down Gforth significantly (that actually happens often enough without us playing such games).

The benefit of this approach over `abi-code` words does not appear to be big enough to merit the effort of implementing it.

3.4 What parameters and how to pass them

Gforth has four stacks visible to the engine: data, return, FP, and locals stack. Moreover, there is the instruction pointer (IP). Which of these pointers should be passed to the called word?

We decided to pass only the data and FP stack pointers (sp and fp), because these are the stacks normally used for dealing with general-purpose and floating-point data. The other stacks and IP are typically used for implementing Forth-system internal stuff like control flow or locals. Most users of `abi-code` will probably not want to implement such words; passing and returning them would increase the cost of executing every abi-code word, so we decided not to pass them.

How do we pass these two stack pointers and how do we return them? At first we passed sp and fp as parameters, and returned them in a struct, leading to the following prototype:

```
struct ac_ret {Cell *sp; Float *fp;};
struct ac_ret word(Cell *sp, Float *fp);
```

However, when we looked at the generated code, we found that this is implemented inefficiently on most platforms: The calling convention on most platforms returns a struct by storing it to memory before returning and loading it from memory in the caller (*pcc calling convention*). So, with two stack pointers in the struct this costs two stores and two loads. And, what's more, the programmer would have to write these stores and have to deal with the target address for this struct.

There is at least one platform (Linux-AMD64), where the standard calling convention passes small structs in registers, and on such platforms this could be the most efficient way of passing the stack pointers, but unfortunately GCC generates inefficient code (redundant stores) even on that platform.

So overall, while this could be an efficient method, in practice it isn't. And on most platforms it is cumbersome to use.

Therefore, we decided to switch to the currently-used way to pass the stack pointers:

```
Cell *word(Cell *sp, Float **fp_pointer);
```

The downside here is that fp is passed and returned in a cumbersome way; but at worst this leads to as many loads and stores as returning a struct on platforms with the pcc calling convention, and in most cases (no FP stack access, or unchanged FP stack depth) it will have fewer loads and/or stores. This approach is also easier to learn and to use for programmers, especially for words that don't access the FP stack.

We also considered several other approaches, but did not implement them:

- Put both pointers in memory and pass pointers to them; as a variation, put them in a structure in memory and pass one pointer to that. The main advantage would be that both stack pointers would be passed in the same way, leading to a cleaner interface. The disadvantage is that this approach is less efficient and requires more loads (and usually stores) than our chosen approach.

- Another, mostly orthogonal option would be to have different words for different usages: E.g., we could have `abi-code-sp`, where only sp is passed and returned (avoiding the overhead of storing fp to memory and loading it back); and maybe `abi-code-fp`, where only fp is passed and returned (in the same way that sp is passed now). This would increase the efficiency, but it would also increase the complexity, implementation and documentation effort of the interface, so we decided not to take this approach for now.

- A reviewer suggested passing the word's arguments and returning the result directly as defined in the calling convention, rather than through the Forth stack. The called routine could then also be called from C in the familiar way without having to set up a memory area for the stack and passing a pointer to that. This would require generating a wrapper that automatically translates between the Forth arrangement and the calling convention. We have done such a thing for calling C functions in the libcc C interface [Ert07], and this approach could also be used here.

  But we don't think that this would be very useful, for the following reasons: 1) In a C interface we usually want to call pre-existing C routines, whereas here the typical usage will be to write new assembly code for this specific problem, so the exact kind of parameter passing convention does not make a big difference. 2) Most calling conventions provide no good way to pass back several results. 3) The wrapper would probably incur extra overhead; e.g., transferring the parameters from the Forth stack to the C stack on IA-32, where it is just as hard to access. 4) One could not implement words such as `roll` with such a mechanism.

For `;abi-code`, we have to pass the body address of the child word in addition to sp and fp. We just pass it as extra parameter.

3.5 Other Forth systems

`Abi-code` solves a problem of Gforth. Would there be a benefit to implementing `abi-code` in other Forth systems? Yes:

- Code using `abi-code` would be portable between Forth systems (on the same platform), unlike code using `code`. This could also be achieved by agreeing on a standard interface to `code` words for each platform, but reaching such an agreement can be a long and arduous process. For the ABI somebody else went through that process, so if we use that, we save ourselves that effort. As for the disadvantages, using the ABI leads to more overhead when executing `abi-code` words. What's worse, on some architectures there are different ABIs for different operating systems, so `abi-code` words do not necessarily port to other operating systems, even on the same architecture and Forth system, unlike `code` words on most systems.

- The infrastructure used for implementing `abi-code` can also be used for calling functions in other languages that use the ABI. However, most systems already have a more convenient C interface that does not require the called function to access Forth stacks. Still, given that these Forth systems implement the ABI for the C interface, it should be easy to implement `abi-code` on them.

There are additional requirements to make code portable across Forth systems: The stacks have to grow in the same direction in the systems (in Gforth all stacks grow downwards).[7] And the stack items have to have the same size and format; that's not a problem for cells, but different Forth systems use different FP formats/sizes on IA-32 (64-bit vs. 80-bit floats).

4 Performance

4.1 Benchmarks

The benchmarks are written for Gforth. We measure both `gforth-fast --no-dynamic` (direct threaded code) as well as the default `gforth-fast` (with various optimizations). Other systems use different implementation techniques, with different effects on performance, so take these results with a grain of salt.

We compare different ways to implement `1+`. This word is so short that its cost is relatively minor compared to the overheads of the various implementation techniques, so the overheads should dominate.

[7] We could get around that requirement by having macros for accessing the stack at a certain depth.

Also, we can implement `1+` both as simple words, or through defining words. We complement this micro-benchmark with a result from an application (Section 4.4).

We compare four different ways of defining simple words:

primitive Primitives come with Gforth, and Gforth knows quite a bit about them, in particular, how to use them in dynamic superinstructions [RS96, PR98, EG03a]. And Gforth can also perform other optimizations on them [Ert02, EG04, EG05]. These optimizations do not include combining a sequence of `1+ ... 1+` into `n +`, however.

code-def A `code` definition. Gforth knows very little about such words, so these are executed as direct-threaded code.

abi-code-def `Abi-code` definitions are usually executed through a primitive `abi-call`; all Gforth optimizations can be applied to this primitive, but the called routine is executed as-is.

colon-def A simple colon definition. Gforth does not perform inlining (yet). Colon definitions are invoked through the primitive `call`. But `gforth-fast` optimizes the body of the colon definition with a static superinstruction [Ert02] for the sequence `lit +`.

We also compare the corresponding four ways of defining `1+` through defining words:

field-def Using the built-in field definition word `+field`; children of this word are compiled to a primitive `lit+`, which has all the usual optimizations applied. This primitive uses a literal constant in the threaded code, so it has a little more overhead than the primitive `1+`.

;code-def To maintain the primitive-centric code [Ert02] in Gforth, the child of such a word cannot be compiled directly to threaded code like code-def. Therefore Gforth uses a primitive `lit-execute` to invoke it, adding some overhead; in particular, there is an additional indirect branch (from the primitive to the code after `;code`). Moreover, the other indirect branch will always be mispredicted in some of our benchmark setups: those where we use dynamic superinstructions and run on CPUs with BTBs.

;abi-code-def Children of a `;abi-code` word are executed through a primitive `;abi-code-exec` similar to `abi-call`.

```
' 1+ alias primitive
\ add    rbx,0x8 \ increment IP
\ add    r14,0x1 \ increment TOS (gcc way)
\ next primitive or NEXT

code code-def
 add rbx,0x8     \ increment IP
 inc r14         \ increment TOS
 jmp [rbx-0x8] \ NEXT
end-code

abi-code abi-code-def
\ ABI: SP passed in rdi, returned in rax
 mov rax,rdi        \ sp into return reg
 inc QWORD PTR[rdi] \ increment TOS
 ret
end-code

: colon-def 1 + ;

\ indirect definitions through defining a
\ defining word

1 0 +field field-def drop
\ add r14,[r9+0x10] \ >body @ +
\ add rbx,0x8       \ increment IP
\ NEXT

: my-field1 ( n -- )
    create ,
;code ( n1 -- n2 )
 \ sp=r15, tos=r14, ip=rbx, cfa=r9
 add rbx,0x8       \ increment IP
 add r14,[r9+0x10] \ >body @ +
 jmp [rbx-0x8]     \ NEXT
end-code
1 my-field1 ;code-def

: my-field2 ( n -- )
    create ,
;abi-code ( n1 -- n2 )
 \ sp in rdi, returned in rax,
 \ addr of fp in rsi, body address in rdx
 mov rcx,[rdx] \ fetch increment from body
 mov rax,rdi   \ sp into return reg
 add [rdi],rcx \ add increment to TOS
 ret
end-code
1 my-field2 ;abi-code-def

: my-field3 ( n -- )
    create ,
  does> ( n1 -- n2 )
    @ + ;
1 my-field3 does>-def
```

Figure 2: Benchmark definitions (Intel syntax for assembly)

does>-def Children of `does>` words are compiled to be invoked using the primitive `does-exec` (which is similar to a sequence of `lit` and `call`).

Figure 2 shows the definitions of these words for the Linux-AMD64 platform.

The benchmark consists of a loop that contains a sequence of these implementations of `1+`. We measure a loop with a sequence of 23 `1+`s, and subtract the time for a loop with 3 `1+`s. This gives the time for executing 20 `1+`s without the loop overhead or startup effects. Note that these micro-benchmarks are unrealistic in their branching behaviour and therefore give unrealistic branch prediction results.

4.2 Machines

The performance of these benchmarks is influenced strongly by how well indirect branches are predicted and by the cost of mispredictions when they happen. Therefore we measure the performance on two different CPUs:

Athlon 64 X2 4400+ This processor has a branch target buffer (BTB), which predicts (to the first order) that each indirect branch jumps where it jumped to the last time it was performed. The misprediction penalty is around 12 cycles.

Core 2 Duo E8400 This processor has a history-based indirect-branch predictor that is usually more accurate than a branch target buffer. The misprediction penalty is around 12 cycles.

All of these CPUs implement a return stack, so the returns at the end of `abi-code` words are predicted correctly.

We also vary the options used with the `gforth-fast` engine:

no-dynamic This is direct-threaded code.

default All optimizations are on. In particular, dynamic superinstructions benefits everything except `code-def` and `;code-def`; static superinstructions benefit `colon-def`; and static stack caching benefits `abi-code-def` and `;abi-code-def`.

4.3 Results

Figure 3 shows the results for direct-threaded code, and Fig. 4 shows the results for optimized code. For both machines, we show instructions, some data about branches and branch mispredictions, and cycles. The metric we actually care about on a particular platform is the cycles, but the other metrics are also interesting, because they help explain the cycle counts that we see, and can help understand what performance to expect on other machines or for other benchmark settings.

For cycles and instructions, the count per executed word (implementation of `1+`) is shown; branches and branch mispredictions are scaled up by a factor of 10, for two reasons: to make their size better visible; and to reflect the approximate cost of branch mispredictions in cycles.

Cycles and Instructions

The instruction counts are the same between the machines, because the same binaries are executed on both machines.

For threaded code (Fig. 3), we see that the primitive has a similar cyle and instruction counts as `code-def`; actually, the instruction count and cycle count is slightly better for the hand-written code word compared to the gcc-generated primitive.

Executing the `abi-code` word is more expensive by 13 instructions and 8–9 cycles; for the optimized code, the difference is 11 instructions and 4–7 cycles. This means that one will still use `code` words where their portability disadvantage is acceptable and the number of dynamically executed instructions in the word is relatively small on average (several dozen instructions or less).

The colon definition performs a similar number of instructions (and cycles) as the `abi-code` word for this micro-benchmark, but that's because `1+` is such a tiny word. For words with more functionality, a colon definition in a threaded-code Forth will require more instructions (and cycles) compared to a primitive or code word by a factor of 5-10 in many cases, whereas the `abi-code` word will only have the same 11–13 instructions of overhead as for this benchmark, not an overhead proportional to the functionality.

The instruction counts for `field-def`, `;code-def`, `;abi-code-def`, and `does>-def` are slightly higher than for the corresponding simple words (they fetch the increment from memory), but are otherwise similar to their corresponding simple words.

Branches and Mispredictions

Branch mispredictions have a strong influence on the cycle count, and the mispredictions in these micro-benchmarks are not representative of typical applications, so we have measured the number of branches and branch mispredictions, and present the results here.

For the branches, the Core 2 can count indirect branches (and their mispredictions), whereas the Athlon 64 can count taken branches (and their mispredictions). For these benchmarks, both

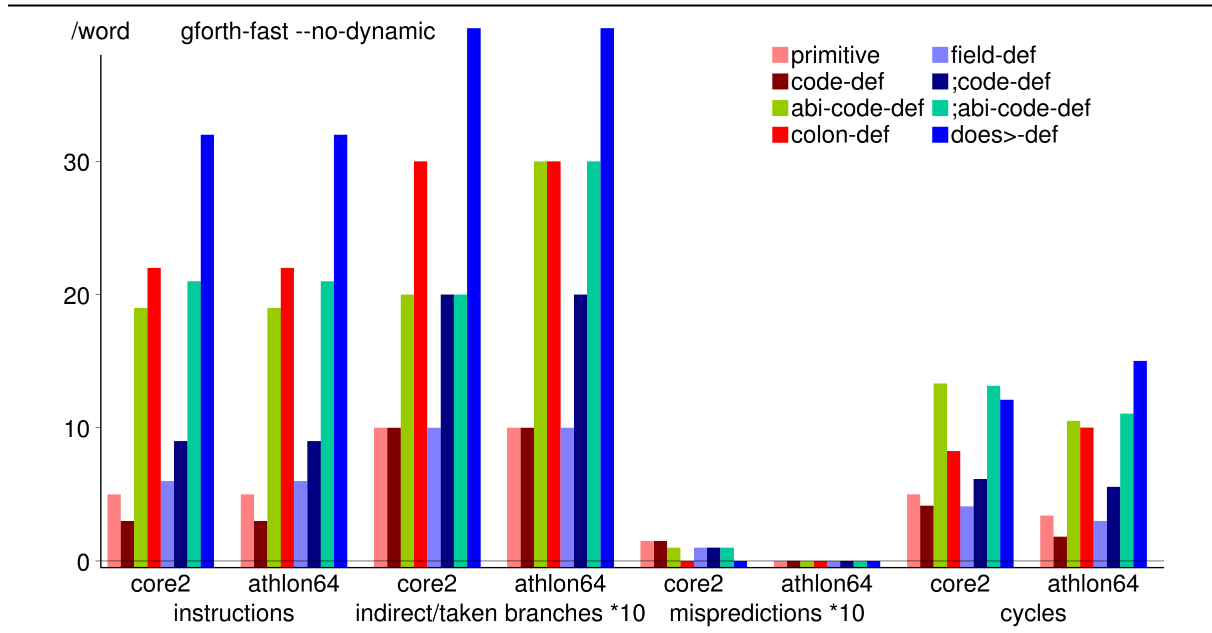


Figure 3: Performance results for `gforth-fast --no-dynamic` (direct threaded code)

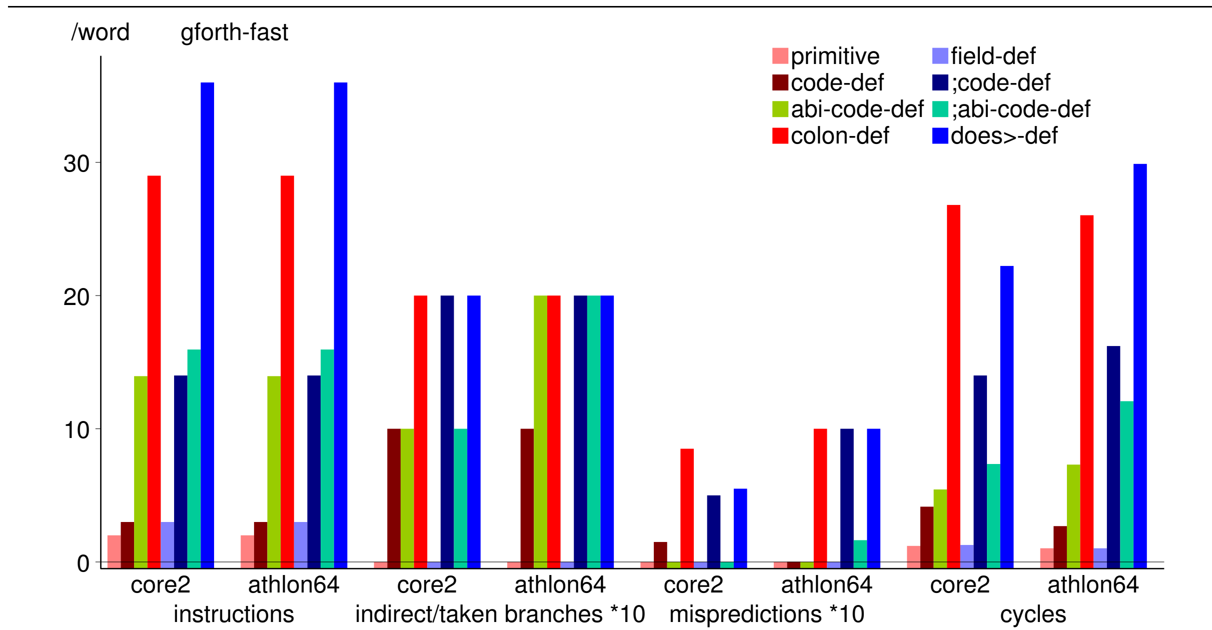


Figure 4: Performance results for `gforth-fast` default (with optimizations)

measurements result in the same branch counts in most cases, except for the `abi-code-def` and `;abi-code-def` cases: There the `ret` from the called word is counted as taken branch by the Athlon 64, but not as indirect branch by the Core 2. These returns are always predicted correctly by the return stack on both CPUs, so this difference does not affect the misprediction counts. The mispredictions differ between the CPUs, because they use different branch predictors.

Does the number of mispredictions differ systematically between `code` words and `abi-code` words?

One difference is that `code` words cannot use the dynamic superinstruction optimization used for Gforth primitives, typically leading to more mispredictions than for primitives (but only partially in our micro-benchmark). For `abi-code` words, dynamic superinstructions can be applied to the `abi-call` primitive; and the indirect call inside this primitive will be well predictable using BTBs and more sophisticated predictors, because each instance of `abi-call` will always call the same code.

Even for this micro-benchmark the Core 2 behaves mostly as expected (for the optimizing

Gforth, see Fig. 4): The branch prediction accuracy is worse for `code-def` than for `abi-code-def`, resulting in a similar cycle count for both of these words.

For threaded code the situation is different: There primitives, `code`, and `abi-code` words all have to perform an indirect branch at the end of the word, and that branch will often ($\approx 50\%$ with a BTB) be mispredicted in real applications. Moreover, because there is only one replica of `abi-call` in a threaded-code system, the indirect call inside `abi-call` will also often be mispredicted if different `abi-code` words are used in the inner loop.

You may notice that the branch prediction accuracy on the Athlon 64 is better on this microbenchmark for threaded code than for the optimized version. That is an artifact of this microbenchmark; real-world code behaves differently [EG03b, EG03a].

4.4 Application performance

In a Mandelbrot set calculation program we replaced a short colon definition (7 words, straight-line code), with an `abi-code` word containing 11 MIPS instructions[8]. This resulted in a speedup by a factor of 1.27 on a 336MHz Ingenic XBurst Jz4720 running `gforth-fast --dynamic`. However, as with `code` words, this approach is only cost-effective if a significant part of the run-time is spent in one or a few words.

5 Related work

The classical Forth way to define words in assembly language is `code`...`end-code`. It has the disadvantage of being system-specific, or worse, in the case of Gforth, installation-specific.

Modern Forth systems also provide a C interface. The main use of this interface is to call libraries that have been developed independently, but it can also be used to call C functions written specifically for a Forth application; and it can be used to call such functions written in assembly language. However, these functions usually have to be compiled or assembled separately before loading the Forth system[9], in contrast to defining words in assembly language at the appropriate places in a Forth source file with `abi-code` and `code`.

New Micros' Max-Forth for the 68hc11 has a word called `code-sub` where the definitions have to end with an `rts` (return from subroutine) rather than a `jmp next` [Dum]. This avoids the need to hard-code the address of `next` and therefore increases the portability of hand-assembled machine code (there was not enough space for a Forth assembler). The implementation uses a run-time routine like Gforth does, but which is less elaborate than `doabicode` (no adjustment to an ABI necessary).

Looking beyond Forth, the Java Native Interface (JNI) [Lia99] shares a number of similarities with `abi-code`. It allows Java to call functions through an interface based on the calling conventions (ABI) combined with additional conventions. The called functions are portable across Java VM implementations, and even across platforms, if written in a portable language like C. There are also differences: JNI functions are compiled separately, and they are usually not written in assembly language.

6 Conclusion

`Abi-code` allows programmers to write assembly language words that work across Gforth engines and versions. If other Forth systems implement `abi-code`, too, they work even across Forth systems.

These words use the standard calling convention (ABI) of the platform, so they are easy to implement in Forth systems that are implemented with the help of a C compiler (like Gforth).

The price we pay for these advantages is an overhead of 11–13 instructions on AMD64 (4–9 cycles on current implementations) when invoking `abi-code` words. However, compared to colon definitions `abi-code` words can provide quite a bit of speedup (a factor of 1.27 by replacing one colon definition in one example application), at the cost of being architecture-specific. So `abi-code` provides a new option between colon definitions and `code` words in the tradeoff between performance and portability.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, which helped improve the paper.

References

[Dum] Randy M. Dumse. *User Manual Max-FORTH*. New Micros.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[8] `http://mosquito.dyndns.tv/freesvn/trunk/nanonote/forth/mandelbr.fs`

[9] Exception: Bernd Paysan used Gforth's libcc interface to generate the C code from Forth code upon loading, and that C code is compiled and linked right away.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.

[EG05] M. Anton Ertl and David Gregg. Stack caching in Forth. In M. Anton Ertl, editor, *21st EuroForth Conference*, pages 6–15, 2005.

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

[Ert07] M. Anton Ertl. Gforth's libcc C function call interface. In M. Anton Ertl, editor, *23rd EuroForth Conference*, pages 7–11, 2007.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

A Compiler which Creates Tagged Parse Trees and Executes them as FORTH Programs

Campbell Ritchie and Bill Stoddart

Formal Methods and Programming Research Group, Teesside University, Middlesbrough, England

Abstract

Most compilers use separate scanning and parsing, and scan their input from left to right. Lynas and Stoddart (2008) demonstrated an alternative technique where an input string is split into multiple sub-expressions, divided at each operator.

We demonstrate an expression parser which extends this work. It runs in two passes. The first pass scans code left-to-right or right-to-left depending on the associativity of the operator sought, creating pointers to strings which together represent a parse tree. At the end of the first pass, it uses lexical analysers to infer the type of each token, or find the type in a lookup table.

The types follow the theory of the B language, building up more complex types as if with the powerset ($\mathbb{P}$) and Cartesian product ($\times$) operators. The "parser" operations, whose titles all begin with "P" assemble the elements into a postfix model of the parse tree, with the operators tagged with a _ character.

The second pass can be run together with the first, or later, allowing the intermediate representation of the code, as a tagged parse tree, to be inspected. The output can be executed as FORTH code; each operator tagged with a _ is matched by a corresponding operation which tests the types supplied, and leaves the type and postfix representation on the stack; the latter is identical to the FORTH representation of the original expression.

We discuss possible uses of such a compiler, and possible problems about its efficiency, since it runs in quadratic time.

Keywords

FORTH, compiler, recursion, parse tree, postfix notation, type tagging.

Address for Correspondence

C Ritchie, Formal Methods and Programming Research Group, Rm P1.10, Teesside University, Borough Rd, Middlesbrough, England TS1 3BA.

work@critchie.co.uk and *bill@tees.ac.uk*

Introduction

Many programming languages are written in infix notation (e.g. "1 + 2"), but most computers execute the contents of a stack, which is most easily expressed in postfix notation (e.g. "1 2 +"). Both languages used for training only (e.g. "VSL" = very simple language (Bennett (1996)) and commercial production languages (e.g. Java™ (Gosling *et al* (2005)) are compiled by conversion to an intermediate form in postfix notation. Since FORTH programs are written in postfix notation, a compiler which produces its intermediate representation in a form similar to FORTH syntax can use a FORTH virtual machine as its back-end. This allows one to write a compiler consisting only of its front-end.

Lynas and Stoddart (2008) introduced such a compiler, originally for teaching undergraduates. This parses an expression differently from most compilers. Instead of scanning the code and dividing it into tokens, their compiler scans the code for operators and connectives, splitting the expression into sub-expressions at each connective. After the code has been subdivided at every operator, it is held in pointers to strings, which together represent a parse tree. Later, a second pass of operations is used to rejoin the strings along with the operators. This resultant string is identical to FORTH syntax for that expression, and can be executed as FORTH code.

Lynas and Stoddart (2008) introduced operations called by an operator followed by an underscore character; for example, the +_ -_ *_ and /_ operations handle addition, subtraction, multiplication, and division respectively. The first pass of compilation produces a parse tree, which can be tagged for types; Lynas and Stoddart's example shows "1 + 2.5" being converted to

```
" 1" INT " 2.5" FLOAT +_
```

by the first pass of compilation.

The values INT and FLOAT are constants representing integer and floating-point numbers, and the other two values are Strings containing the operands. The second pass executes the values on the stack as a FORTH program. The +_ operation requires four values on the stack representing a parse tree. It compares the types, and places two values on the stack: a pointer to the output String

```
“ 1 S>F 2.5 F+”
```

which is itself executable as FORTH code, and the constant FLOAT denoting the type of the result. This implementation technique permits one to use polymorphic operators, providing different operations for different types of operand.

Rather than passing through the code from left to right, separating it into tokens, this parsing technique seeks operators in the code and splits an expression into sub-expressions. It scans the code from left to right for a right-associative operator, and *vice versa*.

We call programs such as +_ tagged operations. We showed plans to use tagged operations, which can be executed as FORTH code last year (Ritchie and Stoddart, 2009). At the end of the first stage of parsing, the operator, along with an underscore "_" is added to the intermediate representation. This produces an output which can be executed as the second stage of parsing. Rather than using integers to represent types, we use strings, allowing types of arbitrary complexity to be declared. This same grammar as we used before, expanded by the addition of Boolean expressions, is shown as an appendix to this paper.

The Structure of this Paper

We first look at sets, and how our typing theory uses sets. Then we describe how this compiler has grown from earlier work, and the three operations used in the parser, the “P” operations which splits the text around an operator, and two tagged operations, ⇔_ and +_. Then follows a section about creating and manipulating sets, and the results of using this parser. Finally, we discuss its potential use, and planned future work.

Sets And Types

Following the type theory of the formal specification and development languages Z and B (Abrial, 1996), we regard each value as having the type of the set to which it belongs. A whole number is a member of the integer set $\mathbb{Z}$ expressed in FORTH as "INT", and a decimal fraction a member of the set $\mathbb{R}$ for real numbers called "FLOAT" in FORTH. It is possible to use the constructor $\mathbb{P}$ for power-set to produce a set whose members are themselves sets; this can be expressed as "INT POW". Similarly the Cartesian Product operator × can be used to produce a set or ordered pairs; for example the ordered pair (1, 2) or $1 \mapsto 2$ is a member of the set $\mathbb{Z}\times\mathbb{Z}$, called "INT INT PROD". Using these constructors, one can create set types of arbitrary complexity, which the set package introduced to RVM FORTH by Stoddart and Zeyda (2002) handles.

Methods

We maintain the convention of Lynas and Stoddart (2008) of appending an underscore to the operator, to give the name of an operation which tests the appropriateness of the typing for that operator. There is however a new problem; we are using arbitrarily complex types, and compound types, in addition to the built-in types "INT" "FLOAT" and "STRING". These cannot be readily expressed as constants, but can be incorporated in Strings which denote those types in the final FORTH code, and which can be compared and manipulated with simple String operations.

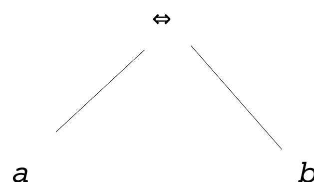
Compiling the Expression Grammar

The grammar is shown as an appendix. It consists of a series of recursive equations, starting with the lowest-precedence operator ⇔ and gradually increasing precedences. The lowest-precedence operator appears in this line:

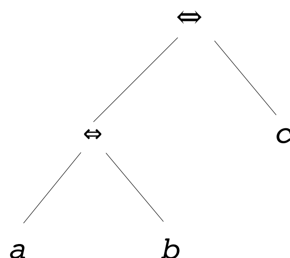
$$E = E \Leftrightarrow E_1,\ E_1$$

. . . meaning that the Strings forming E (for expression) consist of strings from E followed by the ⇔ symbol and a string from E_1, as well as any strings in E_1. Note this grammar permits left recursion.

The equivalence ("iff") symbol ⇔ takes two values, to test for equivalence. It is a binary, infix, left-associative operator. The expression $a \Leftrightarrow b$ can be parsed to form this parse tree:-



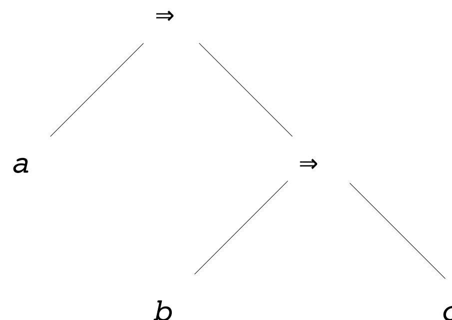
. . . and the expression $a \Leftrightarrow b \Leftrightarrow c$ gives this tree:



In the case where several operators are included in an expression, one splits the expression on the lower precedence operators first. In the case of a left-associative symbol, where there is more than one operator with the same low precedence, the rightmost operator is used to split first (for

right-associative operators, one splits first on the leftmost symbol). The values in *a b* and *c* will themselves be expressions from lower lines in the grammar, which are passed on to parsers for expressions from the next line of the grammar.

The second lowest-precedence connective is the implication symbol ⇒, which is a binary infix, right-associative operator. The expression $a \Rightarrow b \Rightarrow c$ gives this parse tree:



The final output from a parser (P_E) for a string from “E” depends whether the string contains the operator from that line or not. If it does, the right sub-expression is parsed as an E_1, and the left sub-expression as an E; otherwise the whole expression is regarded as an E_0 and passed unchanged to the E_0 parser. The final output from an string *e*, a member of E can be expressed as

$P_E(e ⁀ \text{“}\Leftrightarrow\text{”} ⁀ e_1) = P_E(e) ⁀ P_{E1}(e_1) ⁀ \text{“ }\Leftrightarrow\text{”}$ or $PE(e) = P_{E1}(e)$

. . . where P_E and P_{E1} are parsing operations for strings *e* and e_1 which are members of E and E_1. This maintains the form shown by Lynas and Stoddart (2008). This can be expressed as an intermediate representation from the first pass of parsing; for example, $P_E(e ⁀ \text{“}\Leftrightarrow\text{”} ⁀ e_1)$ gives

“" $P_E(e)$" " T" " $P_{E1}(e_1)$" " U" ⇔_”

This is the result of parsing the left string *e* followed by its type “T” followed by the the result of parsing the right string e_0 and its type “U” followed by the tagged operator ⇔_. The output is a string including the "straight" quote marks, which can be executed as FORTH code. “T” and “U” must both be Boolean type for the ⇔ operator.

The maplet operator ↦ creates ordered pairs and accepts operands of any type. As an example, “f ↦ b”, where the types of f and b are foo and bar respectively would be parsed to give this intermediate representation: “" f" " foo" " b" " bar" ↦_”. This can itself be executed as FORTH code to leave “f b ↦” as the postfix form of the expression, and its type, “foo bar PROD”, on the stack.

The “lsplit” and “rsplit” Operations and the First Stage of Lexical Analysis.

The first stage parsing is to analyse the input string representing the expression into its operator and two arguments or operands, left and right. This is done with operations called lsplit and rsplit; lsplit explores the expression from right to left, looking for left-associative operators, and rsplit explores from left to right, to find a right-associative operator. Each requires two values on the stack, the text to be analysed and a sequence containing the operators sought at the present level of precedence.

The lsplit operation is shown here. It uses the following values internally as local variables: the cardinality of the sequence, the end of the string, a loop count index, and a value as a place-holder for the operator string. These are in addition to the two arguments. It also uses the prefix? function, which tests whether a string is a prefix of another, endaz which finds the end of a 0-terminated string, myazlength which determines its length, and the bracket-avoider-for-lsplit

function, which skips backwards over any text in brackets.

The essence of the operation is two nested loops. The outer loop counts backwards from the end of the text (skipping any text in brackets) and the inner loop compares each value in the sequence in turn to see whether the operator sought has been found; starting from the current value of “end” this would appear to be a prefix to the string.

The original value of “op” is 0 (*null*); whenever a matching prefix is found, “op” is replaced by that value, and both loops terminate. The left sub-expression can be terminated simply by placing a null (\0) character in the location of the “end” pointer with the C! instruction, and the right sub-expression by adding the length of the “op” string to the current “end” pointer.

```
: lsplit ( s seq -- s1 s2 op )
    ( s is a string in the form "a + b" and seq is a sequence of strings e.g. )
    ( ["+", "-"], s1 is the string as far as the operator, s2 after it, and op )
    ( is the operator. If op is null = 0, the value below it must be regarded as a )
    ( nonsense value and deleted from the stack. )
    ( This function skips over any text in “”, (), [] and {} )
( string seq already on stack ) 0 0 0 0 ( 6 values now on stack )
(: string seq end op count size :)
string endaz to end ( One of the 0s gone )
seq CARD to size    ( Second 0 gone      )
BEGIN
    end string >= op 0= AND
WHILE
    0 to count
    end bracket-avoider-for-lsplit to end ( Skip text in brackets etc. )
    BEGIN
        size count > op 0= AND ( Not reached start of string, nor found op )
    WHILE
        count 1+ to count      ( Go through potential operators )
        seq count APPLY end prefix?
        IF
            seq count APPLY to op
        THEN
    REPEAT
    end 1- to end    ( Count backwards to start of string )
REPEAT
op
IF
    end 1+ to end    ( Terminate string at op )
    0 end C!
    end op myazlength + to end ( Move forward length of op )
THEN
string end op
;
```

Note that if no “op” string is found, a 0 will be left on the stack. In that case, the “left” value will be the original text unchanged, and the “right” value must be discarded as a “nonsense” value. The rsplit operation, which is used for right-associative operators, is very similar but slightly simpler, because there is no need to seek the end of the string before starting the two loops.

In this example, passing “*f* ⇔ *b*” and the sequence STRING [" ⇔" ,] to lsplit places the following three string values on the stack:

f   *b*   ⇔

. . . whereas passing “*f* * *b*” and the sequence STRING [" +" , " -" ,] would produce the result

*f * b*   *???*   null

. . . i.e. the original input unchanged, an undefined or nonsense value, and null (= \0).

Passing “1+2-3-4” and the sequence STRING [" +" , " -" ,] however, splits the input at the second minus, leaving this result on the stack:

```
1+2-3     4    -
```

In all cases where there are several left-associative operators which can be found at the same precedence, the input is split at the rightmost symbol, meaning the left string is parsed recursively; similarly the right string (second value on the stack) is parsed recursively if there are several right-associative operators found with rsplit.

The Parser Operations

The parser operations are similar to one another; an example is shown.

```
: Pequiv ( s -- s1 )
(: text :) text equivalence lsplit
VALUE left VALUE right VALUE op
op
IF
    left RECURSE right Pimplies op bar-line AZ^ AZ^ AZ^
ELSE
    left Pimplies
THEN
;
```

. . . where “equivalence” is defined as STRING [" <=>" , " ⇔" ,] (allowing “<=>” as a synonym for “⇔”) and bar-line a string containing “_\n”[1]. The parser splits the “text” into three parts, “left” “right” and “op”. Passing “f ↦ b” as input results in “f” being parsed recursively, “b” being passed to a $P_{implies}$ parser, and the results being catenated (using the AZ^ operation) with the operators and “_\n”. If no operator is found, only the “left” sub-expression represents a real value, which is passed to the next parser in the sequence. In the case of a right-associative operator, rsplit is used, and the recursion is applied to the “right” sub-expression Eventually the recursion reaches a stage where the expression can be subdivided no more; then the parsers return the text and its type with the necessary quotes and spaces included. For example “f” would be parsed to “" f" " foo"” and the type is sought in a lookup table. This has the corollary that variables must be declared in advance, so their type can be entered into the lookup table. The parser above requires one string as input; for “f ⇔ b”, it returns “" f" " foo" " b" " bar" ⇔_” onto the stack.

It is possible to create versions of the parsers which call the operations directly, e.g. with the ⇔_ instruction, and these will call both phases of compilation together.

The Tagged Operators as Operations

The ↦_ operation as a Simple Example

The output from the parsing operations can be passed to a FORTH virtual machine; the output “" f" " foo" " b" " bar" ↦_” puts the four values f foo b bar onto the stack and calls the ↦_ operation. The ↦_ operation has the simplest typing of any; it accepts any type except *null*, and is shown below:

1 Here, \n is the line-feed character 0x10.

```
: ↦_ ( s1 s2 s3 s4 -- ss1 ss2 )
    (: l-value l-type r-value r-type :)
    " ↦" l-type r-type check-types-not-null
    l-value sspace AZ^ r-value AZ^ "  ↦" AZ^ l-type sspace AZ^ r-type AZ^
           "  PROD" AZ^
;
```

The check-types-not-null operation simply checks that null has not been passed as a type because ↦ has no restriction about its types of operand. Then, ↦_ takes the two operand values "l-value" and "r-value", catenating them with spaces and ↦. Then is catenates the two type values "l-type" and "r-type" with "PROD" and the appropriate spaces, leaving those two values on the stack. The equivalence operator ⇔ causes the ⇔_ operation to be invoked; this uses the check-types-for-booleans operation, which tests that both operands have a Boolean type. Almost every tagged operation is similar to ↦_, except those for unary operators which only take one value and one type, and those where the output may differ with the type of input.

Every "check" operation emits an error message and then calls ABORT; this means only one error message is displayed, even if there are several errors.

The +_ Operation as a More Complex Example

As described by Lynas and Stoddart (2008), arithmetical operations may take different input types and produce different output. In a simpler version which accepts only integer numbers, the +_ operation is very similar to ↦_. In the version which accepts floating-point numbers as well, it may be necessary to change the type of the argument with the S>F operator, and prefix the + with an F. The complex version follows:

```
: +_ ( s1 s2 s3 s4 -- ss1 ss2 )
    (: l-value l-type r-value r-type :)
    "  +" VALUE op op l-type r-type check-types-for-arithmetic
    l-type " FLOAT" string-eq r-type " FLOAT" string-eq OR
    ( Either or both is float )
    IF
        "  F+" to op
        l-type " INT" string-eq
        IF  ( Add S>F as appropriate )
            l-value "  S>F" AZ^ to l-value
        ELSE
            r-type " INT" string-eq
            IF
                r-value "  S>F" AZ^ to r-value
            THEN
        THEN
        " FLOAT" to l-type
    THEN
    l-value sspace AZ^ r-value AZ^ op AZ^ l-type
;
```

The check-types-for-arithmetic operation confirms that both types are "INT" or "FLOAT". On checking whether either operand is a "FLOAT", the operator is changed to " F+", and whichever of the operands is an INT has " S>F" appended. Also the type to return is changed to "FLOAT". This operation will take " 1" " INT" " 1.34" " FLOAT" +_ and return " 1 S>F 1.34 F+" and the type "FLOAT".

Sets and Types

A set expression such as {1, 2, 3} can be implemented in FORTH by executing the code

```
INT { 1 , 2 , 3 , }
```

where each token is a FORTH operation. The `INT` provides the type of the set, by placing a pointer to an empty set of `INT`s on the stack, which is opened by the { operation. Each number is placed on the stack in turn, and added to the current set by the comma operator, and the } operation completes the set construction, leaving a new reference to the whole set on the stack.

Sets may contain individual values, or pairs, or sets. {{1, 2}, {3}} is an example of a set of sets, which is represented in FORTH as

```
INT SET { INT { 1 , 2 , } , INT { 3 , } , }
```

and the following is an example of a set of pairs (called a “relation”) from Strings to integers:

```
{“Bill” ↦ 2673, “Campbell” ↦ 2680, “Dave” ↦ 2680}
```

The set of its left-hand elements ({“Bill”, “Campbell”, “Dave”} is called its Domain, and the set of its right-hand elements ({2673, 2680}) is its Range. It can be translated into FORTH as

```
STRING INT PAIR { " Bill" 2673 ↦ , " Campbell" 2680 ↦ , " Dave" 2680 ↦ , }
```

It is possible to retrieve a value from the range of that relation, which we are calling “r”, in infix notation by writing r(“Bill”) which translates to FORTH postfix notation as

```
r " Bill" APPLY
```

If that relation is inverted and the value 2680 applied, there are two possible results, “Campbell” and “Dave”; the choice can be made non-deterministically and on a reversible FORTH implementation the choice can be altered on backtracking.

Sequences can be represented similarly, but using square brackets [...] instead of curly braces {...}; in terms of sets, a sequence is regarded as a relation from integers to another type, T (INT T PROD). In the case of a sequence, its domain is equal to the set of consecutive integers up to its cardinality c, expressed as $1 \ldots c$[2]. It is possible to have relations from integers to T whose domain does not consist of consecutive numbers, and which do not represent sequences. All the set operations can be applied to sequences. As an example where this might be useful, one can compare two sequences of the same type, s and t; s is a prefix of t, if s is a subset of t and the union of s and t equals t ($s \subseteq t \land (s \cup t = t)$).

Operations to Create a Set

The operations to create a set are used in the following order:

`{_`	Puts “{” on the stack (twice) and 0 (= null) because the type is not yet known.
`1`	Puts the value 1 on the stack, and the string “INT” being its type.
`,_`	Catenates the type { 1 and , to leave “ INT { 1 , ”, and changes the type on the stack to “INT”.
`2`	Puts the value 2 and its type “INT” onto the stack.
`,_`	Checks the “INT” is the correct type and catenates the previous value with 2 and , to produce “ { 1 , 2 , ”.
`3`	Puts the value 3 and its type “ INT” onto the stack.
`}_`	Checks the remaining “{“ matches “}”, and that the type “INT” is the same as before, and catenates 3 comma and } to leave “ INT { 1 , 2 , 3 , }” and the type “INT SET” on the stack. This resultant code can be executed as a FORTH instruction.

Since the type of variable is checked, we restrict sets to homogeneous sets, i.e. those which only contain one kind of element.

2 We are using 1, not 0, for the number of the first element in the sequence.

Other Set Operations and Typing

In the case of set union and intersection and difference, the types must be checked that the two operands are the same sort of set, i.e. each shows its type as “T POW”. Relational overriding, using the ⊕ operator replaces values in a relation on the left by values in the same domain in the right operand; overriding of *s* by *t* is written as $s \oplus t$ and can be implemented in FORTH as “s t OVERRIDE”. So each operand must be a relation of the same type, e.g. “S T PROD”.

The typing for other operations can be more difficult. For example the domain restriction operation, using the ◁ operator requires the left operand be a set of type “T” and the right operand be a relation from “T” to a type “U”. So R ◁ S returns a relation from S of all those elements whose domain is included in the set R, and if the type of R is “T POW”, the type of S must be “T U PROD POW”.

Results

We demonstrate a compiler for expressions which can be simply constructed with a recursive architecture. Each component is relatively simple, the most complicated one being a version of rl-lex which can distinguish “-” as a binary or infix operator, for subtraction, from “-” as a unary prefix operator, which occupies 41 lines when comments are excluded. The operation of the compiler can easily be seen by running a FORTH virtual machine. The expression can be fed onto the stack, followed by the name of the operation to compile it, and the output (highlighted in pale grey) can be seen with the .AZ command; feeding this output back to FORTH e.g. with “copy-and-paste” initiates the second pass of compilation, which produces the type “INT” and the postfix expression 1 2 3 * + 4 /, which evaluates to 2.

```
" 1 + 2 * 3 / 4" Pexpression .AZ " 1" " INT" " 2" " INT" " 3" " INT" *_
" 4" " INT" /_
+_
ok
" 1" " INT" " 2" " INT" " 3" " INT" *_ ok....
+_ ok..
" 4" " INT" /_ ok..
.AZ INTok.
.AZ 1 2 3 * + 4 /ok
1 2 3 * + 4 / . 2 ok
```

More complicated expressions can also be analysed. This set expression produces the intermediate result highlighted in grey, and the second output “INT POW” and “INT { 1 , 2 , 3 , }”:

```
" {1, 2, 3}" Pexpression .AZ {_
" 1" " INT" ,_
" 2" " INT" ,_
" 3" " INT" }_
{_ ok...
" 1" " INT" ,_ ok...
" 2" " INT" ,_ ok...
" 3" " INT" }_ ok..
.AZ INT POWok.
.AZ INT { 1 , 2 , 3 , }ok
INT { 1 , 2 , 3 , } .SET {1,2,3}ok
```

Similarly, nested and bracketed expressions can be compiled, for example:

```
" (1 + 2) * 3 / 4" Pexpression .AZ " 1" " INT" " 2" " INT" +_
" 3" " INT" *_
" 4" " INT" /_
ok
" 1" " INT" " 2" " INT" +_ ok..
" 3" " INT" *_ ok..
" 4" " INT" /_ ok..
.AZ INTok.
.AZ 1 2 + 3 * 4 /ok
1 2 + 3 * 4 / . 2 ok
```

This expression also evaluates to 2.

Discussion

"The primary criterion for a parsing algorithm is that it must be efficient." (Bennett 1996, page 80).

Unfortunately, for each stage, it is necessary to traverse the String representing the expression at each of these stages. As described humorously by Spolsky (2001), traversing a null-terminated (or ASCIIZ) String takes a time proportional to the length of the String. Also, the number of traversals is roughly proportional to the number of operators, which again depends on the String's length. Our compiler must therefore run in quadratic time (O(n^2) complexity). So its utility for compiling long programs must be limited, but performance will be better if a large program can be divided into small functions or operations. It may be possible to enhance the FORTH RVM by adding persistent memory, allowing the output from the first pass of compilation to be retained for use by the second pass.

We have, however, demonstrated a compiler for expressions which the writer and reader can simply understand, and which can easily be expanded to complicated expressions. This compiler has the unusual feature that it recursively seeks operators or connectives, rather than going through the text from left to right. It is quite easy to examine and interpret the code, which makes this technique a potential teaching and research tool. Compilers created with automated tools, e.g. yacc (Johnson, 1975) and lex (Lesk 1975) create much code which is difficult to understand at first reading, and does not lend itself to didactic use.

This compiler is suitable for expansion. For example, it would be easy to add Boolean expressions, including conjunction disjunction and implications. It would also be possible to add more operators of different precedences to the grammar, and intersperse parsers to accommodate those operators.

The grammar, as we have written it, easily permits arbitrary recursion both to left and right. This can be seen in the parse trees, and can be seen where the keyword RECURSE appears in the parsers.

Further Work

We plan to add control structures, to implement assignments, loops and selection (if-then-else blocks). These will necessitate Boolean values to control their flow. For assignment, it will be necessary to declare variables before use, so a technique to add variables and their types to a lookup table is needed. Since Böhm and Jacopini (1966) demonstrated that programs of arbitrary complexity can be assembled from elements of sequence, selection and iteration, these control structures are sufficient to build a language capable of any operations. As well as these, additional control structures, including choice and guards, can take advantage of the reversible virtual machine described by Stoddart Lynas and Zeyda (2010).

We shall need a parsing method for Strings, using the opportunity for nesting "smart" quotes provided by Unicode support. It will be necessary for such quotes to be nested in pairs; this following example, which quotes Milne (1926) shows such nesting:

> " "In Which Pooh Goes Visiting and gets Stuck in a Tight Place" by A A Milne
> includes the following:
> "Pooh . . . said that he must be going on. "Must you?" said Rabbit politely.
> "Well," said Pooh, "I could stay a little longer if . . . " " "

It is easy to count from end to end of such a String, until both opening and closing quotes

have been identified.

We hope to implement higher order functions, including λ expressions; these may require both a definition of the function and insertion of its input and output types into a lookup table. Some functions may be implementable as sets of ordered pairs from input to output.

It may also be possible, after a full language is written, to bootstrap the compiler by rewriting it in the new language.

Conclusion

We have demonstrated a two-pass compiler for a rich expression language; one can execute the two passes separately or together. It supports strongly-typed sets and sequences, following the conventions of B. This compiler is made up of small, mostly simple modules which are assembled to form a parse tree, and uses operations called after the operators, tagged with a _ character, to complete the compilation.

Since parse trees have a structure very similar to the postfix notation used in FORTH, it is simple to convert a parse tree to FORTH code which can be executed directly.

References

Abrial J-R 1996. *the B Book* Cambridge: Cambridge University Press.

Bennett J P 1996. *Introduction to Compiling Techniques. A First course using ANSI C, lex and yacc* 2/e (the McGraw-Hill International Series in Software Engineering) Maidenhead: McGraw-Hill

Hehner Eric C R 1981. *Bunch Theory: a Simple Set Theory for Computer Science.* Information Processing Letters **12**(1): 26-30

Böhm C, Jacopini G. *Flow diagrams, Turing machines and languages with only two formation rules*, Communications of the Association for Computing Machinery **9(5)**: 366-371

Gosling J, Joy B, Steele G and Bracha G 2005, *The Java Language Specification* 3/e *(Java Series)* Upper Saddle River NJ: Prentice-Hall, also available at http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html Accessed 15th September 2010

Hehner Eric C R 1993. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Berlin: Springer-Verlag. A more recent edition is available at http://www.cs.toronto.edu/~hehner/aPToP/ accessed 17th June 2010.

Johnson S C 1975. *Yacc—Yet Another Compiler-Compiler.* Comp. Sci. Tech. Rep. No 32, Murray Hill NJ: AT&T Bell Laboratories, July 1975.

Lesk M E 1975. *Lex—A Lexical Analyzer Generator.* Comp. Sci. Tech. Rep. No 39, Murray Hill NJ: AT&T Bell Laboratories, October 1975.

Levine J R, Mason T, Brown D 1992. *lex & yacc* 2/e. Sebastopol CA: O'Reilly & Associates, Inc.

Lynas A R, Stoddart W J 2008. Using Forth in a Concept-Oriented Computer Language Course, in ed. *A* Ertl, Proceedings of the 25th EuroForth Conference, Wien, *pages* 7-19. Available at http://www.complang.tuwien.ac.at/anton/euroforth/ef08/papers/proceedings.pdf, accessed 15th June 2010.

Milne A A, 1926. *Winnie the Pooh* London: Methuen & Co Ltd., and other publishers.

Ritchie C, Stoddart W J 2009. *Formulating Type Tagged Parse Trees as Forth Programs*. in *ed.* A Ertl, Proceedings of the 25th EuroForth Conference, Exeter, *pages* 11-22. Available at http://www.complang.tuwien.ac.at/anton/euroforth/ef09/papers/proceedings.pdf, accessed 16th June 2010.

Spolsky J 2001, *Back to Basics* http://www.joelonsoftware.com/articles/fog0000000319.html accessed 15th June 2010, also available as *Back to Basics, in* Spolsky J 2004, *Joel on Software.* Berkeley CA: Apress, *pages* 5-15

Stoddart W J, Lynas A R, Zeyda F 2010. *A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages.* Electronic Notes in Theoretical Computer Science **253(6)**: 33–56, also available at http://www.www.elsevier.com/locate/entcs accessed 17th June 2010

W. J. Stoddart and F. Zeyda. Implementing Sets for Reversible Computation. In *ed.* A Ertl, 18th EuroForth Conference Proceedings, 2002. On-line proceedings, available at http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/ accessed 21st June 2010.

Appendix

The expression grammar is given here, in bunch notation (Hehner, 1981 and Hehner, 1993). The comma is an operator representing bunch union, including all members of both bunches which are its operands. As an example, if "A" means the bunch of all strings representing arithmetic expressions and "A_1" means the bunch of all strings representing terms in arithmetic, and "+" means joining or catenating two strings around a + sign (with or without spaces), etc., we can regard the top line in an arithmetic grammar as

$$A = A \text{ "+" } A_1,\ A \text{ "–" } A_1,\ \ A_1$$

. . . i.e. the bunch of strings constituting arithmetic expressions followed by + followed by an arithmetic term, AND arithmetic expressions followed by – followed by a term, AND arithmetic terms. There may be white-space between the sub-expressions and the operator. Another definition of "arithmetic term" or A_1 is an arithmetical expression which does not contain + or – as its lowest-precedence operator.

The following abbreviations are used:

- L A comma separated list of expressions. In this case, the underlined comma , is used to represent a literal comma rather than bunch union. L = L, E, L
- E An expression
- B A boolean expression
- P An expression representing a pair
- S An expression representing a set
- W An expression representing a string or a set
- A An arithmetic expression (expression representing a number)
- N Numeric literal
- $ String literal
- I An identifier

Details of the grammar of some non-terminals, e.g. string and numeric literals, are omitted below.

E	=	B "⇔" B_1 ,	E_1			
E_1	=	B_2 "⇒" B_1 ,	E_2			
E_2	=	B_2 "∧" B_3 ,	B_2 " ∨" B_3 ,	E_3		
E_3	=	"¬" B_3 ,	E_4			
E_4	=	E "∈" S ,	E "∉" S ,	E_4 "=" E_5 ,	E_4 "≠" E_5	
E_5	=	A "<" A ,	A "≤" A ,	A ">" A ,	A "≥" A ,	E_6
E_6	=	S "⊆" S ,	S "⊈" S ,	S "⊂" S ,	S "⊄" S ,	E_7
E_7	=	E_7 "↦" E_8 ,	E_8			
E_8	=	E_8 "\" E_9 ,	E_8 "∪" E_9 ,	E_8 "∩" E_9 ,	E_8 "⊕" E_9 ,	E_9
E_9	=	S_3 "◁" S_2 ,	S_3 "◁-" S_2 ,	E_{10}		
E_{10}	=	S_2 " ← " E ,	W "⁀" W_1 ,	S2 "▷" S_3 ,	S_2 "-▷" S_3 ,	S_2 "↓" A ,
			S_2 "↑" A ,	E_{11}		
E_{11}	=	A "+" A_1 ,	A "-" A_1 ,	E_{12}		
E_{12}	=	A_1 "*" A_2 ,	A_1 "/" A_2 ,	E_{13}		
E_{13}	=	"-" A_3 ,	E_{14}			
E_{14}	=	N ,	$,	I ,	F ,	"{" L "}" ,
			"[" L "]" ,	"(" E ")" ,	λ	

B	=	B “⇔” B_1 ,	B_1				
B_1	=	B_2 “⇒” B_1 ,	B_2				
B_2	=	B_2 “∧” B_3 ,	B_2 “ ∨” B_3 ,	B_3			
B_3	=	“¬” B_3 ,	B_4				
B_4	=	E “∈” S ,	E “∉” S ,	E_4 “=” E_4 ,	E_4 “≠” E_4		
B_5	=	A “<” A ,	A “>” A ,	A “≥” A ,	A “≤” A,	B_6	
B_6	=	S “⊆” S ,	S “⊈” S ,	S “⊂” S ,	S “⊄” S ,	B_7	
B_7	=	“true” ,	“false” ,	I ,	I “(” L “)” ,	“(” B “)”	
S	=	S “\” S_1 ,	S “∪” S_1 ,	S “∩” S_1 ,	S “⊕” S_1 ,	S_1	
S_1	=	S_2 “⊲” S_1 ,	S_2 “⊲-” S_1 ,	S_2			
S_2	=	S_2 “←” E ,	S_2 “⁀” S_3 ,	S_2 “⊳” S_3 ,	S_2 “⊳-” S_3 ,	S_1 “↑” A ,	
			S_2 “↓” A ,	S_3			
S_3	=	I ,	F ,	“{” L “}” ,	“[” L “]” ,	“(” S “)” ,	λ
W	=	S_1 “←” E ,	W “⁀”m W_1 ,	S_2 “⊳” S_3 ,	S_2 “⊳-” S_3 ,	S_1 “↑” A ,	
			S_2 “↓” A ,	W_1			
W_1	=	I ,	F ,	“{” L “}” ,	“[” L “]” ,	“(” W “)” ,	λ
A	=	A “+” A_1 ,	A “-” A_1 ,	A_1			
A_1	=	A_1 “*” A_2 ,	A_1 “/” A_2 ,	A_2			
A_2	=	“-” A_2 ,	A_3				
A_3	=	I ,	F ,	N ,	“(” A “)”		
λ	=	$\underline{\lambda}$[3] I • E					

3 Here the underlined $\underline{\lambda}$ is used to represent a literal λ in the text, rather than a λ expression.

EuroForth 2010

Securing a Windows 7 Public Access System Using Forth

S.N. Arhipov Mg.Sc.Eng. apx@micross.co.uk
N.J. Nelson B.Sc C.Eng. M.I.E.E. njn@micross.co.uk
Micross Automation Systems, Units 4-5, Great Western Court, Ross-on-Way,
Herefordshire HR9 7XP, Tel. +44 1989 768080, Fax. +44 1989 768163.

Abstract

Very often in industrial conditions the real time program system is used by one or more operators, who have computer qualification and experience only in using this program. Therefore, it is very important, that this special group of users is allowed to use only this program and not allowed to use all other programs in the computer. At the same time it is necessary to keep multi user environment and allow the administrator to use all system opportunities. Herewith, only the administrator can switch between desktops, it should be quick and should continue executing the programs. In this article the Forth programmatical technique of disabling some functional features of Windows 7 is described.

Introduction

`TRACKNET` is a universal software package for tracking and control of work in commercial laundries. The software consists of two parts. The first is an operator interface program which runs on a network of Personal Computers (PCs) under the Windows operating system. It is a Windows application and therefore it can be run simultaneously with all business software. The `TRACKNET` operator interface program works with authorised users, who, accordingly, can have two roles: as an operator and as `MICROSS` administrator. These roles are specified and registered in the `TRACKNET` program and while using this program an operator has some functional limitations and also has no opportunity to switch to another program application or to press `Ctrl+Alt+Del` keys, because this action is intercepted and forbidden by the `TRACKNET` program.

Nevertheless, in the Windows Vista and Windows 7 operating systems it is impossible to intercept `Ctrl+Alt+Del`. Experiments with keyboard hooking, using `SetWindowsHookEx()` function, and the `WM_HOTKEY` message trapping code injection into the main windows procedure did not give a positive result because of the hook procedure

```
LRESULT KeyboardProc(...)
{
    if (Key == VK_CTRLALTDEL) return 1;
    return CallNextHookEx(...);
}
```

and hot key catching in Windows main procedure

```
LRESULT CALLBACK NewWindowProc
        (HWND  hWnd,  UINT  uMsg,   WPARAM  wParam,  LPARAM
```

```
lParam)
{
  if (uMsg == WM_HOTKEY)
     if (lparam == MAKELONG(MOD_CONTROL | MOD_ALT, VK_DELETE))
         return 1;
  return CallWindowProc(OldWindowProc, hWnd, wParam, lParam);
}
```

is activated later than `Ctrl+Alt+Del` typing event happens, an operating system sets focus on Windows log off screen.

Therefore, it was decided to use a desktop switching technique, creating a new desktop and run the `TRACKNET` program on it in order not to lock into program such system keys as `Alt+Esc`, `Alt+Tab`, `Ctrl+Shift+Esc` etc., but to isolate one process from another. Other processes continue running on the "`Default`" desktop and the screen saver runs on the "`Screen-saver`" desktop. All these desktops must be locked until the `TRACKNET` process runs on a new desktop and does not finish, or an authorised user switches back to the "`Default`" desktop.

Figure 1. Winlogoff desktop before and after commands disabling.

However, the logoff process runs on the "`Winlogoff`" desktop, which always switches on in Windows 7 and Windows Vista when the user presses `Ctrl+Alt+Del`. In this case, when creating a new desktop, it is necessary to change the background and disable all commands the "`Winlogoff`" desktop, except for pressing the “`Cancel`” button, and resetting the "`Winlogoff`" desktop to initial state, when an authorised user switches on the "`Default`" desktop. The initial state of the logoff screen and its transformation are illustrated in Figure 1.

Startup program overview

Program `TrStarter.exe` manages two processes: creating, if it does not exist, a new executable process `TRACKNET.EXE` in a new desktop, and disable all desktop commands: “`Log off`”, “`Lock this computer`”, “`Change a password...`”, “`Start Task Manager`”, “`Switch User`” as well as disable “`Shut down`” and “`Ease of access`” buttons (Figure 1.) on the "`Winlogoff`" desktop. On starting the program

TrStarter.exe a new desktop is switched on with a new background image backgroundDefault.jpg from a specially created directory Currentdirectory\DESKTOPRES. Now any user has an opportunity to work solely with TRACNET program.

All another programs continue running on the "Default" desktop. Only a user with corresponding privileges, using "File\Switch desktop" command, can switch the desktop back without closing the TRACNET program. Next time when a starter program tries to return the focus to a new desktop, it can recognize if the TRACNET program is running or not and correspondingly not create a new process. In the TRACNET program, with the help of the "File\Exit" command, the start up desktop takes the focus and TRACKNET is closed (Figure 2.).

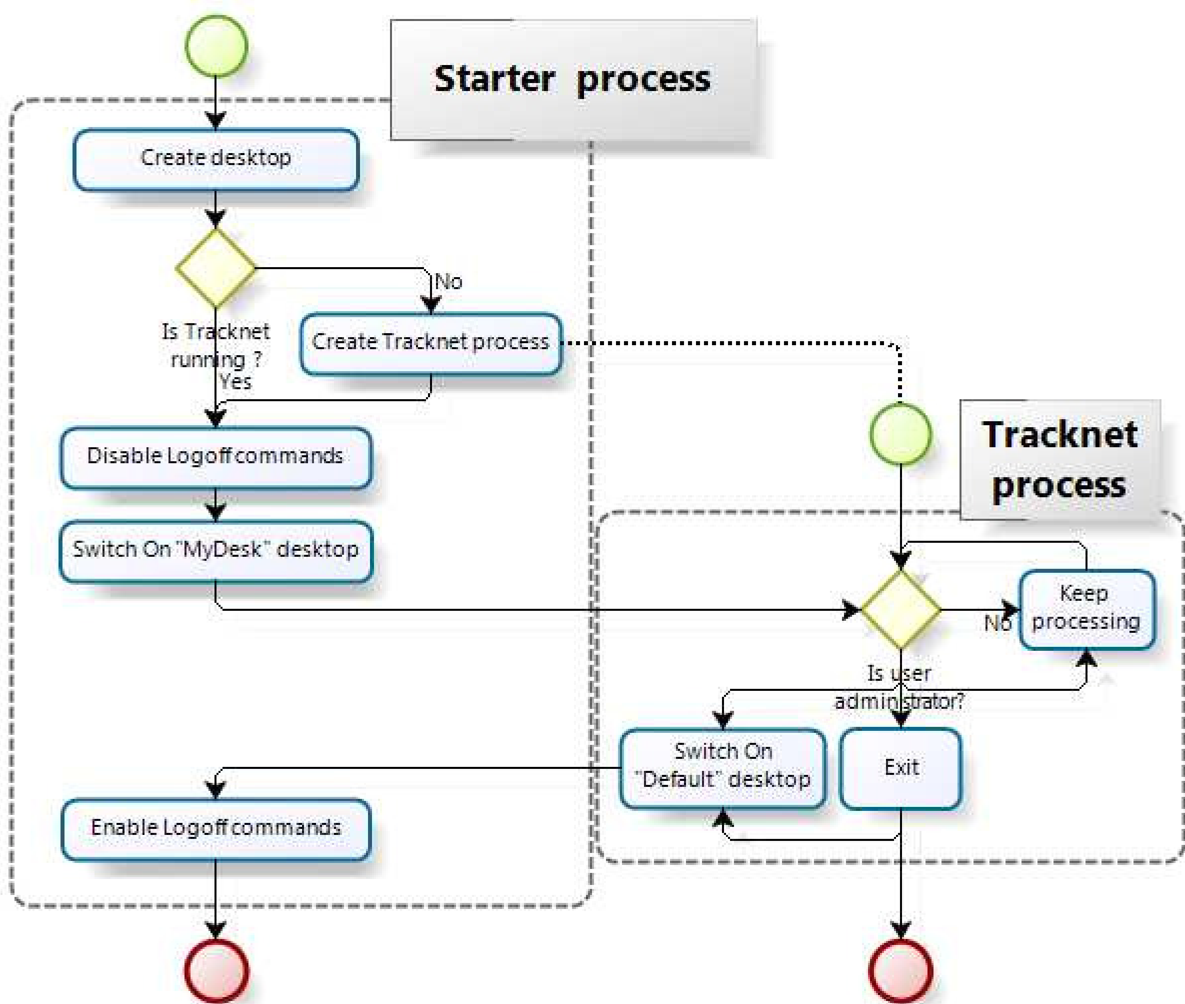


Figure 2. Starter TrStarter.exe and Tracknet.exe processes activity diagram.

TRACNET is a program which is implemented using ProForth for Windows V2.100 and which already has been successfully commercially used more than ten years [1]. TrStarter starter program is implemented using VFX Forth for Windows (4.41. 29 March 2010) [3] in order to execute the TRACKNET program in the operating systems Vista or Windows 7 [2] .It uses WinApi 32 functions for processes, windows registry, files and directories management.

C structures mapping in Forth

To create a new process it is necessary to use two structures. The first of them is `STARTUPINFO`, which specifies a window station, desktop, standard handles, and the appearance of the main window for a process at the time of creation. The table below specifies this structure in Forth and `C` notations, and fields, which are used in a process creation, are represented as comments in the third column of the table below.

Forth notation	*C definition*	*Used fields*
STRUCT STARTUPINFO	typedef	
	struct _STARTUPINFO {	
DWORD field SI.CB	DWORD cb;	\ Size of structure
DWORD field lpReserved	LPTSTR lpReserved;	
DWORD field SI.LPDESKTOP	LPTSTR lpDesktop;	\ Name of desktop
DWORD field lpTitle	LPTSTR lpTitle;	
DWORD field dwX	DWORD dwX;	
DWORD field dwY	DWORD dwY;	
DWORD field dwXSize	DWORD dwXSize;	
DWORD field dwYSize	DWORD dwYSize;	
DWORD field dwXCountChars	DWORD dwXCountChars;	
DWORD field dwYCountChars	DWORD dwYCountChars;	
DWORD field dwFillAttribute	DWORD dwFillAttribute;	
DWORD field dwFlags	DWORD dwFlags;	
WORD field wShowWindow	WORD wShowWindow;	
WORD field cbReserved2	WORD cbReserved2;	
DWORD field lpReserved2	LPBYTE lpReserved2;	
DWORD field hStdInput	HANDLE hStdInput;	
DWORD field hStdOutput	HANDLE hStdOutput;	
DWORD field hStdError	HANDLE hStdError;	
END-STRUCT	}	
	STARTUPINFO,	
	*LPSTARTUPINFO;	

The second structure that needed to create a new process is `PROCESSINFORMATION`. It contains information about the newly created process and its primary thread. In the table below this structure is specified in Forth and `C` notations. This structure is used as an output parameter of the `CreateProcess()` function in order to use process handle closing time.

Forth notation	*C language definition*
STRUCT PROCESSINFORMATION	typedef struct _PROCESS_INFORMATION {
DWORD field hProcess	HANDLE hProcess;
DWORD field hThread	HANDLE hThread;
DWORD field dwProcessId	DWORD dwProcessId;
DWORD field dwThreadId	DWORD dwThreadId;
END-STRUCT	}
	PROCESS_INFORMATION,
	*LPPROCESS_INFORMATION;

Entry function of starter program is

```
: RUN ( --- ) { | si[ STARTUPINFO ] pi[ PROCESSINFORMATION ]
res }
;
```

, where local variables `si[` and `pi[` are defined. Structure `si[` is initialised in a Forth program by setting values into two fields:

```
STARTUPINFO si[ SI.CB !                \ Set size of structure
```

```
DESKTOPNAME si[ SI.LPDESKTOP !        \ Set name of desktop
```

The function RUN creates a new desktop with the help of function `CreateDesktop()`, using parameter values in Forth notation:

Forth notation	*C notation*
\	`HDESK CreateDesktop(`
`DESKTOPNAME` \	`in LPCTSTR lpszDesktop,` \ The name of the desktop to be created.
`NULL` \	`LPCTSTR lpszDevice,` \ Reserved; must be NULL.
`NULL` \	`DEVMODE * pDevmode,` \ Reserved; must be NULL.
`0` \	`in DWORD dwFlags,` \ The access to the desktop
`DESKTOP_SWITCHDESKTOP` \	`in ACCESS_MASK dwDesiredAccess,` \ The access to the desktop
`NULL` \	`in LPSECURITY_ATTRIBUTES lpsa` \ A pointer to a structure
`CreateDesktop` \	`)`

After a new desktop and startup information has been created TRACKNET process is created on a new desktop.

Process creation and monitoring

Using Win32-based Spy++ (SPYXX.EXE) utility, it is possible to view the system's processes, threads, windows, and window messages, but impossible to view the name of a desktop, or the name of both the desktop and window station for those processes. The starter program was executed in two different ways – on a default and a new separated desktop, and the results can be seen on Figure 3. When the starter program is running on a new desktop (right window), but Spy++ is running on a default desktop, it is possible to view only a process and its threads, but the actual desktop and windows of process are invisible. Anywhere Spy++ gives useful information about system's object names and its identification. It is important information, that can be used during testing of the process of creating new desktop.

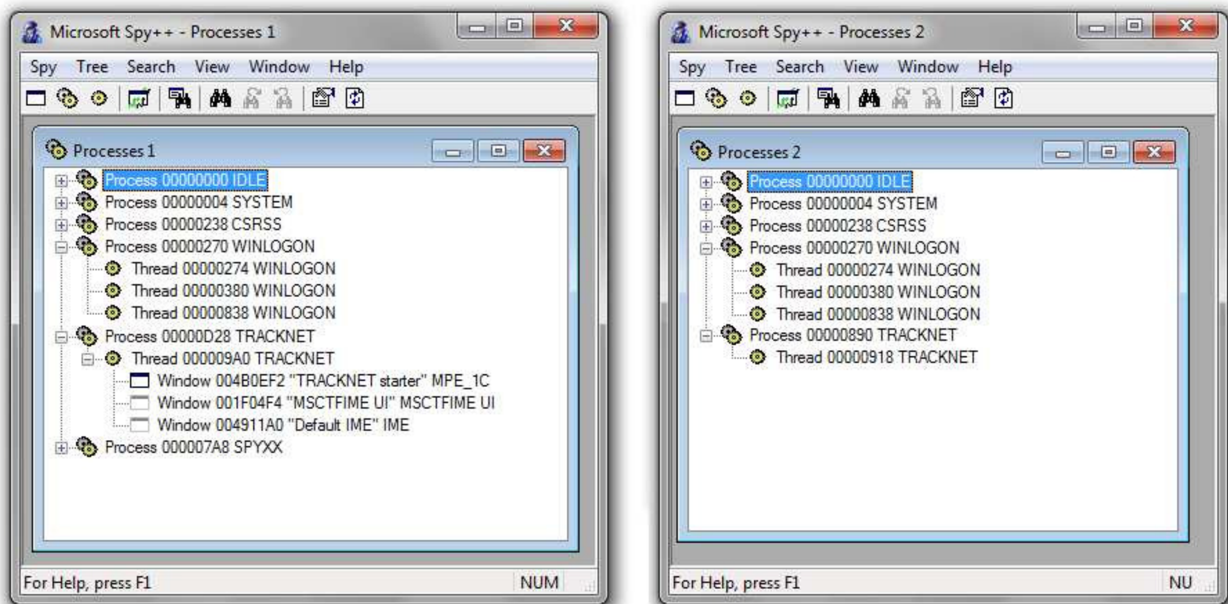


Figure 3. Starter TrStarter processes, threads and windows when running on "Default" desktop (left window) and when running on new desktop (right window).

The first step of the starter program `TrStarter` is to identify if the `TRACKNET` process is running or not. Forth function

```
: CHECKISTRACKNETRUNNING
   { | res processhandle modulehandle -- f }
;
```

examines all processes that are running (Figure 6.) and compares the names of the modules with the name `TRACKNET.EXE`. If there are no such names in the module name list then the function returns the value `false`, and for the rest, value `true`.

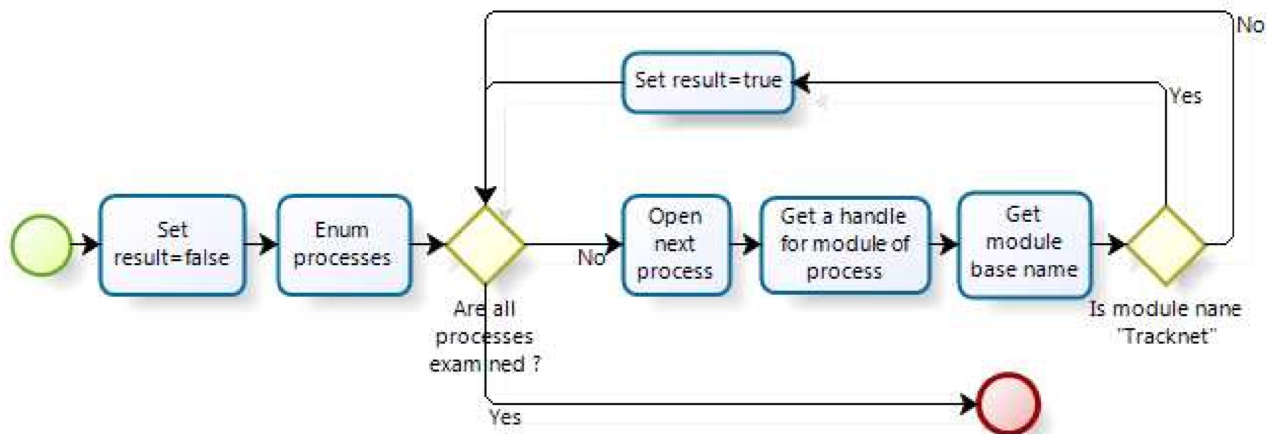


Figure 4. Activity diagram of function `CHECKISTRACKNETRUNNING`

There are three local variables in `CHECKISTRACKNETRUNNING` function. Result variable `res` is a temporary value storage. In the loop, variable `processhandle` sequentially sets the values, which are the handles of the all processes running in the system. Variable `modulehandle` sets a value, which is a handle of the module of each process. There may be many modules in one process, but function `CHECKISTRACKNETRUNNING` examines only first module in each process and therefore there is only one loop for looking over all processes.

The fist used WinApi function has the signature

```
BOOL WINAPI EnumProcesses
 (out    DWORD    *pProcessIds,in    DWORD    cb,out    DWORD
*pBytesReturned);
```

and it retrieves the process identifier for each process object in the system. The use of this function is necessary in three objects:

- an array size constant, correspondent to input parameter `cb`

```
4096 CONSTANT SIZEOFARR
```

- a pointer to an array of process identifiers, correspondent to the output parameter `pProcessIds`

```
CREATE ARROFPROCIDS SIZEOFARR ALLOT
```

- the number of bytes returned in the array of process identifiers, correspondent to the output parameter `pBytesReturned`

```
VARIABLE NUMBEROFBYTESRETURNED
```

In this case calling of `EnumProcesses()` function in Forth notation is:

```
ARROFPROCIDS  SIZEOFARR  NUMBEROFBYTESRETURNED  EnumProcesses
DROP
```

When all system processes are enumerated it is possible to examine every process cyclically using the function

```
HANDLE OpenProcess ( in DWORD dwDesiredAccess,
                     in BOOL bInheritHandle,
                     in DWORD dwProcessId);
```

that opens an existing local process object and returnes the handle to the process according to its identifier. Function

```
BOOL EnumProcessModules(in HANDLE hProcess,out HMODULE
*lphModule, in DWORD cb, out LPDWORD lpcbNeeded);
```

retrieves a handle for each module in the specified process.

In every loop before process opening, into the variable, that is defined as

```
VARIABLE PROCESSID
```

the identifier of current process is loaded

```
ARROFPROCIDS I CELLS+ @ PROCESSID ! \ Set the current process ID
```

Using the current process identifier, which is the `PROCESSID` variable, calling the `OpenProcess()` function returns an open handle to the specified process.

```
PROCESS_QUERY_INFORMATION PROCESS_VM_READ OR \ The access to the process
FALSE                                        \ Processes do not inherit this handle
PROCESSID @                                  \ Current process ID
OpenProcess -> processhandle
```

Continuing a loop, the function `EnumProcessModules()` retrieves a list of handles of process modules

```
processhandle          \ A handle to the process from OpenProcess()
ADDR modulehandle      \ An array that receives the list of module handles
SIZEOFDWORD            \ The size of the array, in bytes\
NUMBEROFBYTESNEEDED    \ The number of bytes required to store handles in the array
EnumProcessModules -> res \ If the function succeeds, the return value is nonzero.
```

There are two specifics calling this function in the Forth program. The first of them is the use of the second parameter `modulehandle`, witch must be specified by type `HMODULE *`. Basically, it is an address of variable `modulehandle` and in Forth program the `ADDR` word is used. The second specific aspect is the using of the third parameter. In the function signature it is defined as `DWORD` type number, but it is not a count of elements in an array, but textually it is the size of the module handle address – simply constant

```
 4 CONSTANT SIZEOFDWORD
```

Last WinApi function in the loop is

```
DWORD GetModuleBaseName( in HANDLE hProcess,in HMODULE
hModule,out LPTSTR lpBaseName,in DWORD nSize);
```

that retrieves the base name of the specified module. The handle of module is loaded in a local variable `modulehandle` and the handle of process is loaded in local variable `processhandle`

Therefore, calling this function

```
processhandle       \ A handle to the process that contains the module.
```

```
modulehandle          \ A handle to the module.
MODULEBASENAME        \ A pointer to the buffer that receives the base name of the module
MODULEBASENAMESIZE    \ The size of the buffer, in characters
GetModuleBaseName DROP
```

retrieves the base name of the specified module into the output parameter `MODULEBASENAME`. Its corresponding parameter in the function signature has `LPTSTR` type. In the Forth program this parameter is defined as a buffer with length `1024`:

```
1024 CONSTANT MODULEBASENAMESIZE
CREATE MODULEBASENAME MODULEBASENAMESIZE ALLOT
```

Likewise, the received value of the current module name is compared with the "`TRACKNET`" string and if they are equal then function `CHECKISTRACKNETRUNNING` returnes the value `true`.

Windows registry editing

When a new desktop has been created, it is necessary to disable all commands (Figure 1.) except "`Cancel`". This command disabling technique is based on Windows registry key editing. The data structure of Windows registry structure is represented in Figure 5. It is a hierarchical structure of keys and their sub keys. Every key can have many or no values at all; every key value is a structure with three attributes – name of value, type of value and data that is stored into the key value.

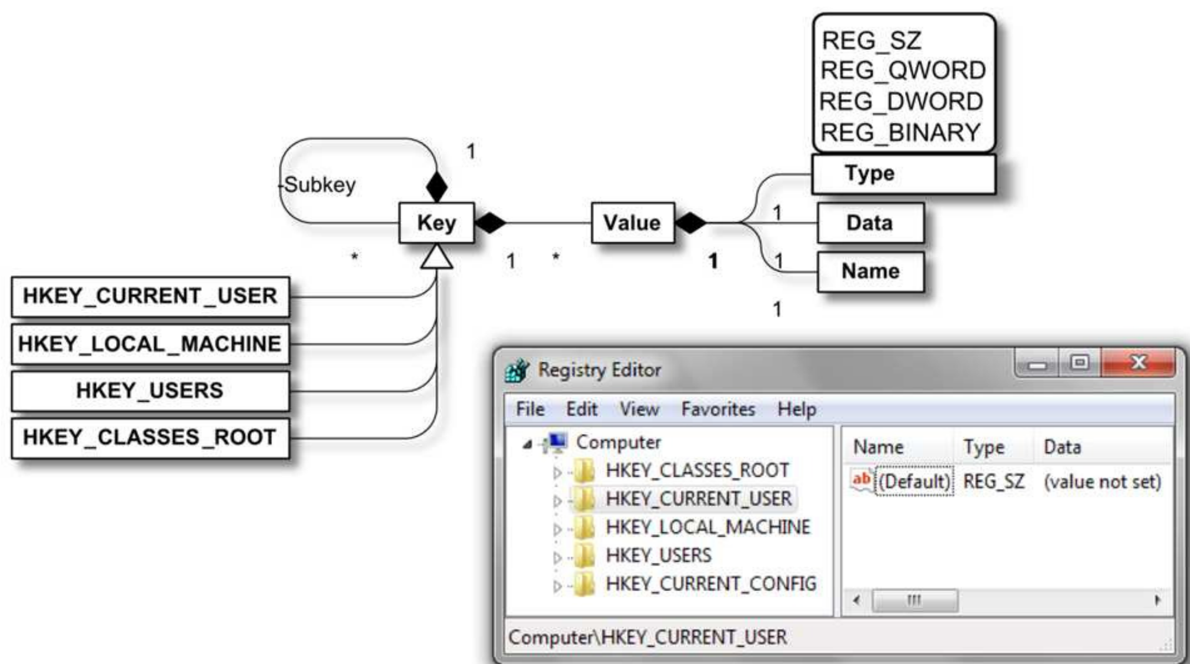


Figure 5. Structure of Windows registry

In order to disable a command on the logoff window, it is necessary in `HKEY_CURRENT_USER` root for `Software \Microsoft\Windows\CurrentVersion\Policies\Explorer` key to create a value with a corresponding name type and data as seen in table below.

Name of key value	*Type of key value*	*Data of key value*	*Disable action*
NoLogoff	DWORD	1	“Logg off”
NoClose	DWORD	1	“Shut Down”

But for the
`Software\Microsoft\Windows\CurrentVersion\Policies\System`
key it is necessary to create a value according to the table below.

Name of key value	*Type of key value*	*Data of key value*	*Disable action*
DisableLockWorkstation	DWORD	1	“Lock this computer”
DisableChangePassword	DWORD	1	“Change a password...”
DisableTaskMgr	DWORD	1	“Start Task Manager”
HideFastUserSwitching	DWORD	1	“Switch User”

In order to enable any command, the corresponding data of key value must be deleted or set to zero. It is possible to do it manually using the `regedit.exe` editor. But there are four WinApi functions for key management (Figure 6.) and two WinApi functions, with the help of which it is possible to manage values for the given key.

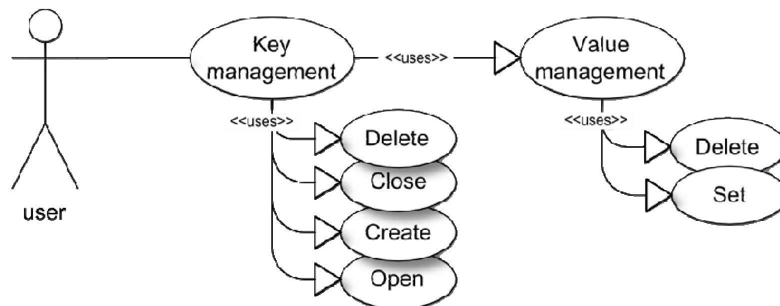


Figure 6. Functionality of Windows registry keys and values

The starter program uses such registry functions as:
- `RegOpenKeyEx()` - opens the specified registry key,
- `RegCloseKey()` - closes a handle to the specified registry key,
- `RegSetValueEx()` - sets the data and type of a specified value under a registry key,
- `RegDeleteKeyValue()` - removes the specified value from the specified registry key and sub key.

In the disable command the sequence of calling functions is shown on Figure 7. This is the activity sequence of the Forth word

```
: REGKEYSETVALUE ( predefinedkey keyname valuename value --- )
  { predefinedkey keyname valuename value | res }
;
```

, which set the value into the registry key. It receives input values into local variables
- `predefinedkey`, that must be a predefined constant `HKEY_CURRENT_USER`,
- `keyname`, that has one of two string values “`...\Explore`” or “`...\System`”,
- `valuename` – is one of the strings from the first column from the tables above,
- `value` – must be 1 to disable a command.

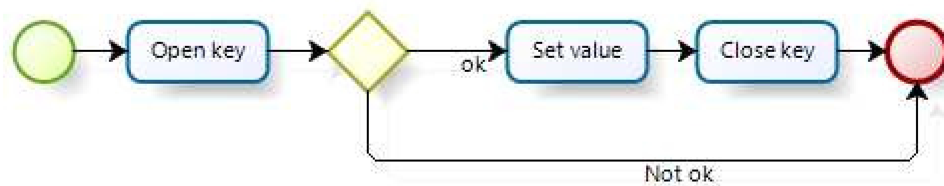


Figure 7. Activity diagram of function, that disable command

The fourth function `REGKEYSETVALUE` tries to open the key using function `RegOpenKeyEx()` with input parameter values `predefinedkey` and `keyname` (table below). If the function fails, the return value is a nonzero error code.

Forth notation	*C language signature*
\	LONG WINAPI RegOpenKeyEx(
predefinedkey\	in HKEY hKey, \ handle to registry key = HKEY_CURRENT_USER
keyname \	in LPCTSTR lpSubKey, \ The name of the registry subkey to be opened.. f
0 \	DWORD ulOptions, \ must be zero
KEY_SET_VALUE\	in REGSAM samDesired,\ A mask that required to create, delete, or set a registry value
PHKEYRESULT \	out PHKEY phkResult \ A pointer to a variable that receives a handle to the opened key
RegOpenKeyEx \	);
-> res	

Key value setting is executed calling function `RegSetValueEx()` in Forth notation, that can be seen in table below.

Forth notation	*C language signature*
	LONG WINAPI RegSetValueEx(
PHKEYRESULT @ \	in HKEY hKey, \ A handle to an open registry key
valuename \	in LPCTSTR lpValueName, \ The name of the value to be set
0 \	DWORD Reserved, \ must be zero
REG_DWORD \	in DWORD dwType, \ int type of data
ADDR value \	in const BYTE * lpData, \ The data to be stored
4 \	in DWORD cbData \ The size of the data to be stored
RegSetValueEx \	);

The use of `RegSetValueEx()` function has some particular features. The function fourth parameter `dwType` is predefined and has a value from the set – `REG_BINARY`, `REG_DWORD`, `REG_QWORD` etc. depending on the type (Figure 5.) of the value data to be stored. In the case of the command's disability it uses only value 1, that has type `int` or predefined value `REG_DWORD`. The second particular feature is that there are many of data types which it is possible to store in the registry key value. Therefore in the general case it is specified as `const BYTE *lpData` – pointer to first byte of memory segment with storage data. In Forth notation the `ADDR` word is used, which is the address of a variable value, that is received as an input parameter of the `REGKEYSETVALUE` word. The last parameter value of `RegSetValueEx()` function is simply `4` – the size of `int` type.

Function `RegCloseKey()` (table below) closes the opened registry key, using its handle as the parameter.

Forth notation	*C language signature*
PHKEYRESULT @ RegCloseKey	LONG WINAPI RegCloseKey(in HKEY hKey \ A handle to an open registry key);

Switching the "Default" desktop, program TRACKNET back on enables all logoff commands. It deletes key values NoLogoff, NoClose DisableLockWorkstation, DisableChangePassword, DisableTaskMgr, HideFastUserSwitching from the registry using function RegDeleteKeyValue (table below).

Forth notation	*C language signature*
predefinedkey keyname valuename RegDeleteKeyValue	LONG WINAPI RegDeleteKeyValue(in HKEY hKey, \ A handle to an open registry key. in LPCTSTR lpSubKey, \ The name of the registry subkey in LPCTSTR lpValueName \ The registry value to be removed from the key);

Desktop background changing

Using function CopyFile() a new background image, placed in a reserved folder and has the same name as original desktop image backgroundDefault.jpg, is copied into the folder C:\Windows\System32\oobe\Info\backgrounds and replaces the original background with the new one. Switching "Default" desktop back on, the original background image is restored from the reserved folder, replacing the image in its own folder. Function CopyFile() signature and its usage is represented in table below.

Forth notation

```
LOGINIMAGENAME    \ The name of an existing login image file from reserved folder.
LOGONINFOIMAGE    \ The name of the new login image file in C:\Windows\System32\oobe\Info\backgrounds
FALSE             \ Overwrite if file already exist
CopyFile
```

C language signature

```
BOOL CopyFile(
in  LPCTSTR lpExistingFileName,
in  LPCTSTR lpNewFileName,
in  BOOL    bFailIfExists);
```

In order to make the copying process successful, it is necessary to set corresponding permissions for the MICROSS administrator role. It is passible to do manually, using the Application Data Property editor for changing file ownership and security settings. But these steps can be executed programmatically.

Setting administrator permissions

To enable administrators to take ownership of shares is possible, using the command-line utility TakeOwn, but to manage security settings of files and folders is possible, using command-line tool Icacls. In order to use them, the Win32 function ShellExecute() is

used in a program. This function signature and its usage are represented in the tables below.

Forth notation	*C language signature*

`0`
`Z" open"`
 	
`Z" cmd.exe"`
 	
`Z" /c takeown /f C:\im.jpg"`
 	
`Z" "`
`0`
`ShellExecute`
	`HINSTANCE ShellExecute(`
	`in HWND hwnd,` \A handle to the owner window
	`in LPCTSTR lpOperation,` \A verb, that specifies the action \ to be performed: „edit", „open", „print" etc.
	`in LPCTSTR lpFile,` \The object on which to execute \ the verb.
	`in LPCTSTR lpParameters,` \ The parameters to be passed \ to the application
	`in LPCTSTR lpDirectory,` \ Pointer to working directory
	`in INT nShowCmd` \The flag how to display \ application
	`);`

Forth notation

```
\ Applaing stored DACLs to files in specified directories for Administrators
  0
  Z" open"
  Z" cmd.exe"   \ open cmd.exe and execute Icacls utility
  Z"  /c Icacls C:\im.jpg /grant BUILTIN\Administrators:(F)  "
  Z" "
  0
  ShellExecute
```

Conclusions

In result, the program `TrStart` allows to start any program in separated desktop and to restrict some users from all others program's execution, including task manager and all logoff commands. It is possible to tune the `TrStart` program in order to configure concrete user permissions. For users with the appropriate privilege it is enable to switch on default desktop by the new "`Switch desktop`" menu entry. The program `TrStart` allow return to the new desktop without restarting the base program, if it is already running.

Information sources

[1] N.J. Nelson, "Experiments in real time control in Windows using Forth", 20th euroForth Conference, Dagstuhl Castle, November 19th-22th 2004.

[2] Yochay Kiriaty, Laurence Moroney, Sasha Goldshtein, Alon Fliess "Introducing Windows® 7 for Developers", Microsoft Press, 2010.

[3] Microprocessor Engineering Limited, VFX Forth for Windows Native Code ANS Forth Implementation, User manual 4.41 29 March 2010

J1: a small Forth CPU Core for FPGAs

James Bowman
Willow Garage
Menlo Park, CA
jamesb@willowgarage.com

***Abstract*— This paper describes a 16-bit Forth CPU core, intended for FPGAs. The instruction set closely matches the Forth programming language, simplifying cross-compilation. Because it has higher throughput than comparable CPU cores, it can stream uncompressed video over Ethernet using a simple software loop. The entire system (source Verilog, cross compiler, and TCP/IP networking code) is published under the BSD license. The core is less than 200 lines of Verilog, and operates reliably at 80 MHz in a Xilinx Spartan®-3E FPGA, delivering approximately 100 ANS Forth MIPS.**

I. INTRODUCTION

The J1 is a small CPU core for use in FPGAs. It is a 16-bit von Neumann architecture with three basic instruction formats. The instruction set of the J1 maps very closely to ANS Forth. The J1 does not have:

- condition registers or a carry flag
- pipelined instruction execution
- 8-bit memory operations
- interrupts or exceptions
- relative branches
- multiply or divide support.

Despite these limitations it has good performance and code density, and reliably runs a complex program.

II. RELATED WORK

While there have been many CPUs for Forth, three current designs stand out as options for embedded FPGA cores:

MicroCore [1] is a popular configurable processor core targeted at FPGAs. It is a dual-stack Harvard architecture, encodes instructions in 8 bits, and executes one instruction in two system clock cycles. A call requires two of these instructions: a push literal followed by a branch to Top-of-Stack (TOS). A 32-bit implementation with all options enabled runs at 25 MHz - 12.5 MIPS - in a Xilinx Spartan-2S FPGA.

b16-small [2], [3] is a 16-bit RISC processor. In addition to dual stacks, it has an address register A, and a carry flag C. Instructions are 5 bits each, and are packed 1-3 in each word. Byte memory access is supported. Instructions execute at a rate of one per cycle, except memory accesses and literals which take one extra cycle. The b16 assembly language resembles Chuck Moore's ColorForth. FPGA implementations of b16 run at 30 MHz.

eP32 [4] is a 32-bit RISC processor with deep return and data stacks. It has an address register (X) and status register (T). Instructions are encoded in six bits, hence each 32-bit word contains five instructions. Implemented in TSMC's 0.18μm CMOS standard library the CPU runs at 100 MHz, providing 100 MIPS if all instructions are short. However a jump or call instruction causes a stall as the target instruction is fetched, so these instructions operate at 20 MIPS.

III. THE J1 CPU

A. Architecture

This description follows the convention that the top of stack is T, the second item on the stack is N, and the top of the return stack is R.

J1's internal state consists of:

- a 33 deep $\times$ 16-bit data stack
- a 32 deep $\times$ 16-bit return stack
- a 13-bit program counter

There is no other internal state: the CPU has no condition flags, modes or extra registers.

Memory is 16-bits wide and addressed in bytes. Only aligned 16-bit memory accesses are supported: byte memory access is implemented in software. Addresses 0-16383 are RAM, used for code and data. Locations 16384-32767 are used for memory-mapped I/O.

The 16-bit instruction format (table I) uses an unencoded hardwired layout, as seen in the Novix NC4016 [5]. Like many other stack machines, there are five categories of instructions: literal, jump, conditional jump, call, and ALU.

Literals are 15-bit, zero-extended to 16-bit, and hence use a single instruction when the number is in the range 0-32767. To handle numbers in the range 32768-65535, the compiler follows the immediate instruction with `invert`. Hence the majority of immediate loads take one instruction.

All target addresses - for call, jump and conditional branch - are 13-bit. This limits code size to 8K words, or 16K bytes. The advantages are twofold. Firstly, instruction decode is simpler because all three kinds of instructions have the same format. Secondly, because there are no relative branches, the cross compiler avoids the problem of range overflow in `resolve`.

Conditional branches are often a source of complexity in CPUs and their associated compiler. J1 has a single instruction that tests and pops T, and if $T = 0$ replaces the current PC with the 13-bit target value. This instruction is the same as `0branch` word found in many Forth implementations, and is of course sufficient to implement the full set of control structures.

ALU instruction have multiple fields:

field	width	action
T'	4	ALU op, replaces T, see table II
$T \rightarrow N$	1	copy T to N
$R \rightarrow PC$	1	copy R to the PC
$T \rightarrow R$	1	copy T to R
dstack $\pm$	2	signed increment data stack
rstack $\pm$	2	signed increment return stack
$N \rightarrow [T]$	1	RAM write

Table III shows how these fields may be used together to implement several Forth primitive words. Hence each of these words map to a single cycle instruction. In fact J1 executes all of the frequent Forth words - as measured by [6] and [7] - in a single clock cycle.

As in the Novix and SC32 [8] architectures, consecutive ALU instructions that use different functional units can be merged into a single instruction. In the J1 this is done by the assembler. Most importantly, the `;` instruction can be merged with a preceding ALU operation. This trivial optimization, together with the rewriting of the last call in a word as a jump, means that the `;` (or `exit`) instruction is free in almost all cases, and reduces our measured code size by about 7%, which is in line with the static instruction frequency analysis in [7].

The CPU's architecture encourages highly-factored code:

- the call instruction is always single-cycle
- `;` and `exit` are usually free
- the return stack is 32 elements deep

B. Hardware Implementation

Execution speed is a primary goal of the J1, so particular attention needs to be paid to the critical timing path. This is the path from RAM read, via instruction fetch to the computation of the new value of T. Because the ALU operations (table II) do not depend on any fields in the instruction, the computation of these values can be done in parallel with instruction fetch and decode, figure 1.

The data stack D and return stack R are implemented as small register files; they are not resident in RAM. This conserves RAM bandwidth, allowing `@` and `!` to operate in a single cycle. However, this complicates implementation of `pick` and `roll`.

Our FPGA vendor's embedded SRAM is dual-ported. The core issues an instruction read every cycle (port a) and a memory read from T almost every cycle (port b), using the latter only in the event of an `@` instruction. In case of a memory write, however, port b does the memory write in the following cycle. Because of this, `@` and `!` are single cycle operations[1].

In its current application - an embedded Ethernet camera - the core interfaces with an Aptina imager and an open source Ethernet MAC using memory mapped I/O registers. These registers appear as memory locations in the \$4000-\$7FFF range so that their addresses can be loaded in a single literal instruction.

[1]the assembler inserts a `drop` after `!` to remove the second stack parameter

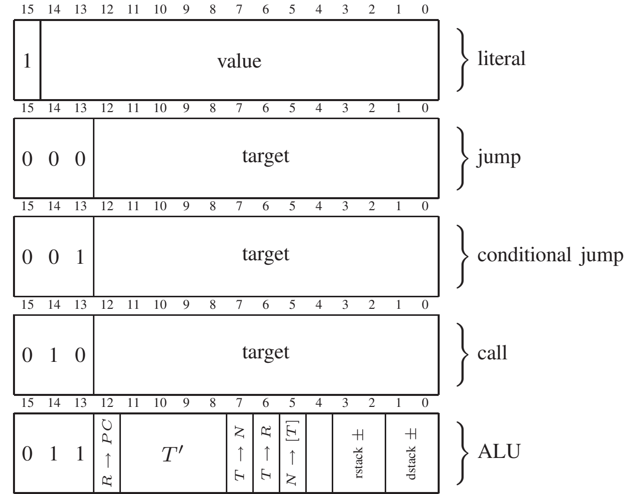


TABLE I: Instruction encoding

code	operation
0	T
1	N
2	$T + N$
3	$T \texttt{and} N$
4	$T \texttt{or} N$
5	$T \texttt{xor} N$
6	$\sim T$
7	$N = T$
8	$N < T$
9	$N \texttt{rshift} T$
10	$T - 1$
11	R
12	$[T]$
13	$N \texttt{lshift} T$
14	`depth`
15	$N \texttt{u<} T$

TABLE II: ALU operation codes

word	T'	$T \rightarrow N$	$R \rightarrow PC$	$T \rightarrow R$	dstack $\pm$	rstack $\pm$	$N \rightarrow [T]$
`dup`	T	•			+1	0	
`over`	N	•			+1	0	
`invert`	$\sim T$				0	0	
`+`	$T + N$				-1	0	
`swap`	N	•			0	0	
`nip`	T				-1	0	
`drop`	N				-1	0	
`;`	T		•		0	-1	
`>r`	N			•	-1	+1	
`r>`	R	•		•	+1	-1	
`r@`	R	•		•	+1	0	
`@`	$[T]$				0	0	
`!`	N				-1	0	•

TABLE III: Encoding of some Forth words.

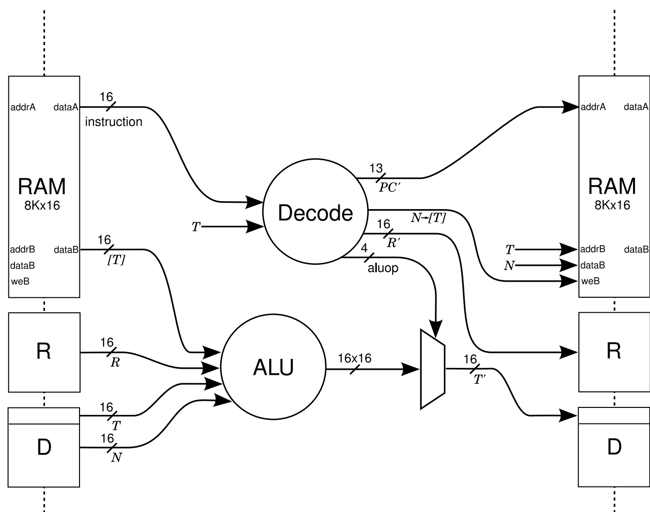


Fig. 1: The flow of a single instruction execution. ALU operation proceeds in parallel with instruction fetch and decode. Bus widths are in bits.

C. System Software

Because the target machine matches Forth so closely. the cross assembler and compiler are relatively simple. These tools run under gforth [9]. The compiler generates native code, sometimes described as subroutine-threaded with inline code expansion [8].

Almost all of the core words are written in pure Forth, the exceptions are `pick` and `roll`, which must use assembly code because the stack is not accessible in regular memory. Much of the core is based on eforth [10].

D. Application Software

The J1 is part of a system which reads video from an Aptina image sensor and sends it as UDP packets over Ethernet. The PR2 robot running ROS [11] uses six of these cameras, two in stereo pairs in the head and one in each arm.

The main program implements a network stack (MAC interface, Ethernet, IP, ARP, UDP, TCP, DHCP, DNS, HTTP, NTP, TFTP and our own UDP-based camera control protocol), handles I^{2}C, SPI, and RS-232 interfaces, and streams video data from the image sensor.

The heart of the system is this inner loop, which moves 32 bits of data from the imager to the MAC:

```
begin
    begin MAC_tx_ready @ until
    pixel_data @ MAC_tx_0 !
    pixel_data @ MAC_tx_1 !
    1- dup 0=
until
```

IV. Results

The J1 performs well in its intended application. This section attempts to quantify the improvements in code density and system performance.

Static analysis of our application gives the following instruction breakdown:

instruction	usage
conditional jump	4%
jump	8%
literal	22%
call	29%
ALU	35%

An earlier version of the system used a popular RISC soft-core [12] based on the Xilinx MicroBlaze®architecture, and was written in C. Hence it is possible to compare code sizes for some representative components. Also included are some tentative results from building the same Forth source on MicroCore.

component	MicroBlaze	J1	MicroCore
	code size (bytes)		
I^{2}C	948	132	113
SPI	180	104	105
flash	948	316	370
ARP responder	500	122	–
entire program	16380	6349	–

The J1 code takes about 62% less space than the equivalent MicroBlaze code. Since the code store allocated to the CPU is limited to 16 Kbytes, the extra space freed up by switching to the J1 has allowed us to add features to the camera program. As can be seen, J1's code density is similar to that of the MicroCore, which uses 8-bit instructions.

While J1 is not a general purpose CPU, and its only performance-critical code section is the video copy loop shown above, it performs quite well, delivering about 3X the system performance of the previous C-based system running on a MicroBlaze-compatible CPU.

V. Conclusion

By using a simple Forth CPU we have made a more capable, better performing and more robust product.

Some directions for our future work: increasing the clock rate of the J1; using J1 in other robot peripherals; implementing the ROS messaging system on the network stack.

Our source code and documentation are available at: `http://www.ros.org/wiki/wge100_camera_firmware`

VI. Acknowledgments

I would like to thank Blaise Glassend for the original implementation of the camera hardware.

References

[1] K. Schleisiek, "MicroCore," in *EuroForth*, 2001.
[2] B. Paysan. `http://www.jwdt.com/~paysan/b16.html`.
[3] B. Paysan, "b16-small – Less is More," in *EuroForth*, 2004.

[4] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in *Canadian Conference on Electrical and Computer Engineering*, pp. 518–521, 2007.
[5] E. Jennings, “The Novix NC4000 Project,” *Computer Language*, vol. 2, no. 10, pp. 37–46, 1985.
[6] D. Gregg, M. A. Ertl, and J. Waldron, “The Common Case in Forth Programs,” in *EuroForth*, 2001.
[7] P. J. Koopman, Jr., *Stack computers: the new wave*. New York, NY, USA: Halsted Press, 1989.
[8] J. Hayes, “SC32: A 32-Bit Forth Engine,” *Forth Dimensions*, vol. 11, no. 6, p. 10.
[9] A. Ertl, B. Paysan, J. Wilke, and N. Crook. http://www.jwdt.com/~paysan/gforth.html.
[10] B. Muench. http://www.baymoon.com/~bimu/forth/.
[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “Ros: an open-source robot operating system,” in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, (Kobe, Japan), May 2009.
[12] S. Tan. http://www.aeste.my/aemb.

Using Glade to create GTK+ Applications with FORTH.

euroFORTH 2010 - manfred.mahlow@forth-ev.de

September 8, 2010

Abstract

When talking about GUI development with FORTH, one of the most expressed desires is to have an IDE or a graphical editor. Some years ago, when I wrote an object-oriented GTK+ interface for cspForth, an IDE or graphical editor was not my concern. But, when I recently decided to port the GTK+ interface to MINFORTH, I remembered that often expressed desire and had a look at Glade and found that Glade and FORTH can be quite good companions.

What is Glade ?

Glade is a graphical user interface builder for GTK+. It's neither an IDE nor a code editor. It allows to design graphical user interfaces saved as XML files. The XML files describe the layout of the GUI, the properties of the widgets and the signal handling. Since version 3.5.0 two file formats are supported, an older one to be used with Libglade and a newer one for Libgtk.

Using a GUI created with Glade is not a tough task. The usual steps to write the required program code are

1. Decision what library to use, Libglade or Libgtk.
2. Initializing GTK+.
3. Creating a GladeXML or a GtkBuilder object per widget hierarchy.
4. Loading the widget hierarchies from the XMF file into this objects.
5. Reading widget identifiers from the GladeXML or GtkBuilder objects.
6. Writing signal handlers.
7. Assigning signal handlers to widgets.
8. Displaying the GUI.
9. Starting the GTK+ main loop.

A GUI created with Glade can be used with any program language, as long as the language gives access to the required Libgtk or Libglade functions.

Glade and FORTH

FORTH and Glade can be great companions. Let's have a quick look at a small example, written for MINFORTH 1.5(p).[1]

An Example ...

The Graphical User Interface

Our starting point is a GUI created with Glade. The newer GtkBuilder format is used. The GUI consists of two widget hierarchies, the main application window (window1 in Fig. 1) and a modal dialog (dialog1 in Fig. 2).

Fig. 1 and Fig. 2 show the same Glade instance. The only difference is the selected widget. In Fig. 1 it's window1, in Fig. 2 it's dialog1.

The main window is a GtkWindow with only one child, which is a GtkLabel (label1). The dialog is a GtkDialog widget and has some more children, packed into container widgets. A GtkImage (image1) and a GtkLabel (label2) are packed into a GtkHBox (hbox2) which is packed into the dialogs internal GtkVBox (dialog-vbox1) and two GtkButton widgets (button1, button2) are packed into the dialogs internal GtkHBox (dialog-action_area1).

All packing has been done and all widget properties have been set with Glade. The only thing that can not be done when using FORTH, is to assign signal handlers with Glade. This is only possible when using the program language C.

The GUI specification is stored in XML format (Fig. 3) to be used with the GtkBuilder object defined in Libgtk.

[1] MINFORTH 1.5(p) is MINFORTH 1.5 (for LINUX) with some additional hooks and minor extensions for the object-oriented GTK+ interface.

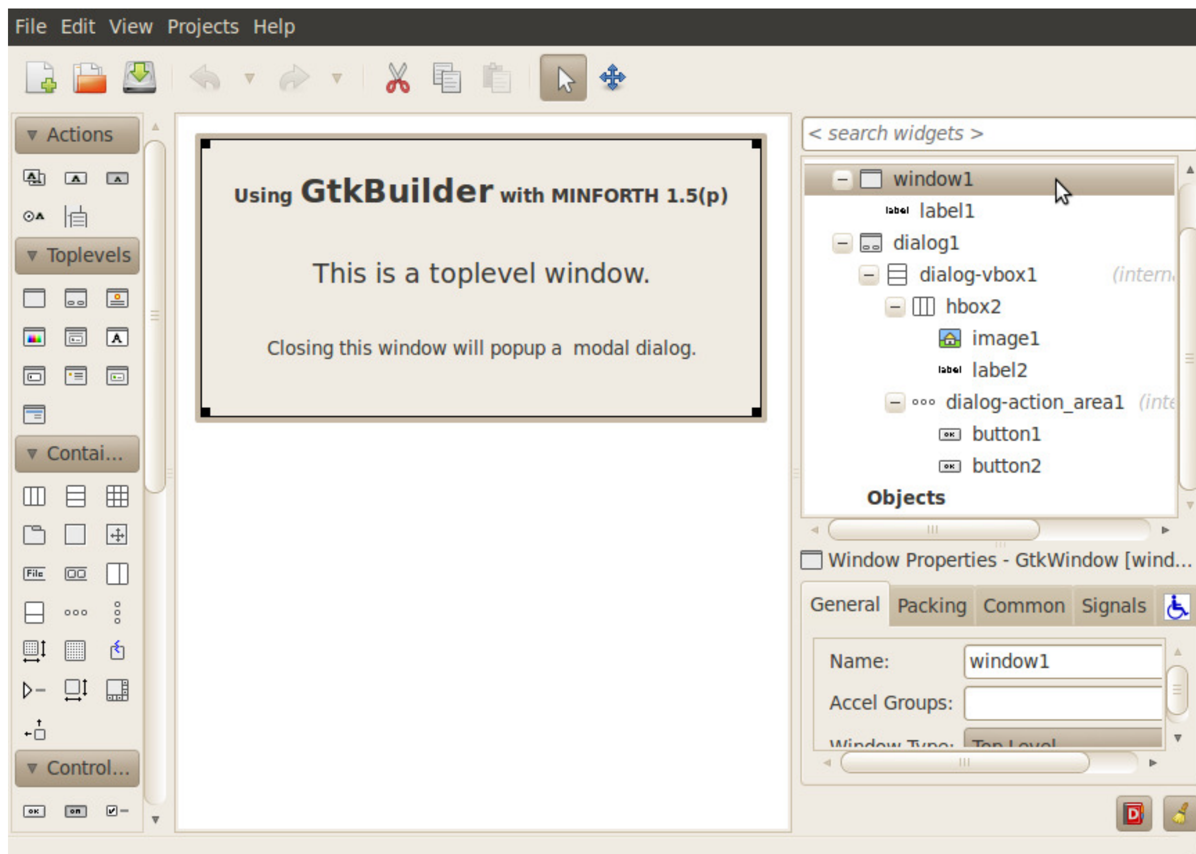


Figure 1: Glade GUI editor, window1 selected

The GTK+ Interface

To transform the GUI into a GUI application, some program code is required. We'll use an object-oriented GTK+ interface here.

The GTK+ classes are mapped to classes of the same name in FORTH. Classes are loaded on demand as required.

A widget is an instance of its widget class. The methods to create and initialize a widget and to modify its properties are defined in its class.

Widget properties are implemented as instance variables. Property and method names are choosen to be close to the corresponding GTK+ names.

The FORTH Code

Now lets have a look at the program code in Fig. 4.

The required classes are loaded in line 8 to 11.

A String object is needed for the name of the XML file, a GtkBuilder object to load the GUI specification from the file and a GtkDialog and a GtkWindow object to get access to the GUIs main window and to the dialog. The objects are created in line 15 to 18.

In line 18 and 19 two signal slots are created to be used to connect signals and signal handlers to the main window(window1).

A first signal handler is defined in line 23. It is called when a 'destroy' signal is received by the main window (window1). It's very simple here. Its only task is to terminate the GTK+ event processing by leaving the GTK+ main loop. It's called with two parameters. Both are not used here.

The second signal handler is defined in line 25 to 27. It is called, when the main window(window1) receives a 'delete-event' signal from the window manager. Its task is to open the dialog1, wait for a button press, destroy the dialog when a button is pressed and return a button-specific value. A 'destroy' signal will be send to the main window (window1) if FALSE is returned. This is the case if the 'QUIT' button (button2) was pressed. Otherwise no 'destroy' signal will be emitted and the 'delete-event' signal will have no further effect.

In line 26 the GtkBuilder object (builder) is initialized from the XML file with the GUI specification for the dialog (dialog1). All widgets of the dialogs widget hierarchy are created here. Then, in line 27, the GtkDialog object for dialog1 is initialized by reading its widget identifier from the builder object and the dialog is shown

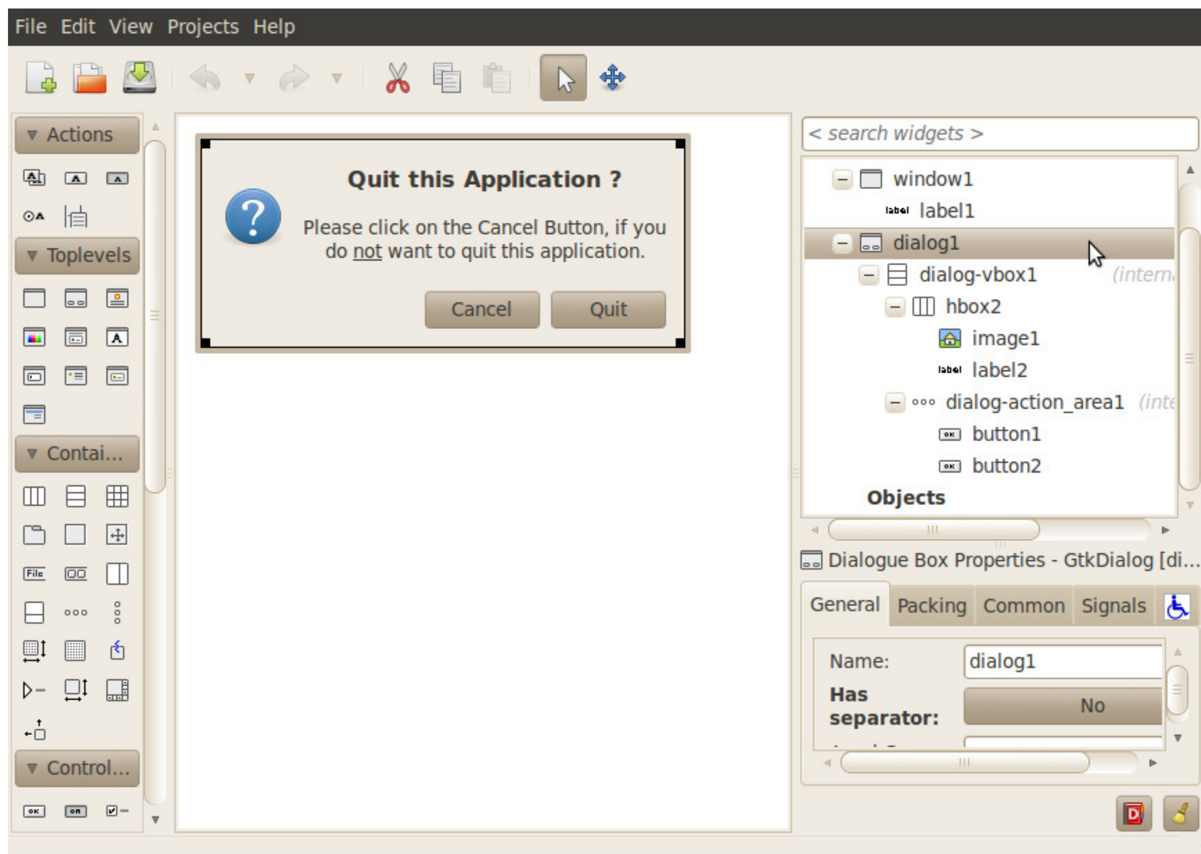


Figure 2: Glade GUI editor, dialog1 selected

to the user (dialog1 run), waiting for the user to press one of the dialog buttons.

In line 29 to 34 the word to start the application is defined.

In line 30 the GtkBuilder object (builder) is initialized from the XML file with the GUI specification for the main window (window1). All widgets of the main windows widget hierarchy are created here.

In line 31 the GtkWindow object for the main window (window1) is initialized by reading its widget identifier from the builder object.

Line 32 connects the on.destroy signal handler from line 23 to the 'destroy' signal at window1, using the signal slot cb.destroy and line 33 connects the on.delete-event signal handler from line 25 to the 'delete-event' signal at window1, using the signal slot cb.delete.

The code in line 34 makes the main window (window1) visible on the computer display and, finally, the application is started in line 36.

Two modes of event processing are supported here. If MINFORTH runs in a terminal window, the GTK+ events are processed in the background while waiting for terminal input, to preserve FORTHs interactivity. Otherwise a GTK+ main loop is entered for event processing.

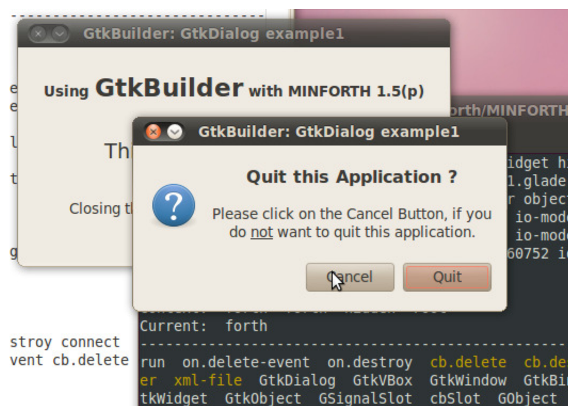


Figure 5: The running GUI example.

Up and Running

Fig. 5 shows the running application with the dialog waiting for user response after the close button of the main window was clicked.

The Benefit of using Glade

The advantage of creating a GUI with Glade instead of creating it from source code is, that no code needs to be written to create the widgets, to set the widget properties and to pack the widgets into container widgets to get the

```
 1 <?xml version="1.0"?>
 2 <interface>
 3   <requires lib="gtk+" version="2.16"/>
 4   <!-- interface-naming-policy project-wide -->
 5   <object class="GtkWindow" id="window1">
 6     <property name="border_width">24</property>
 7     <property name="title" translatable="yes">GtkBuilder: GtkDialog example1</
property>
 8     <property name="resizable">False</property>
 9     <property name="window_position">center</property>
10     <child>
11       <object class="GtkLabel" id="label1">
12         <property name="visible">True</property>
13         <property name="label">&lt;b&gt;Using &lt;span size='xx-
large'&gt;GtkBuilder&lt;/span&gt; with MINFORTH 1.5(p)&lt;/b&gt;
14 
15 
16 &lt;span size='x-large'&gt;This is a toplevel window.&lt;/span&gt;
17 
18 
19 Closing this window will popup a  modal dialog.
20 </property>
```

Figure 3: First twenty lines of Glades XML output.

```
 1 \  euroFORTH-2010/GtkBuilder/example1.mf
 2 \ -------------------------------------------------------------------------------
 3 \  GtkBuilder              OOP Library for MINFORTH                       MM-100801
 4 \ -------------------------------------------------------------------------------
 5 \                 Copyright (C) 2010 manfred.mahlow@forth-ev.de
 6 \
 7 \ -------------------------------------------------------------------------------
 8    requires String
 9    requires GtkBuilder
10    requires GtkWindow
11    requires GtkDialog
12 
13    forth definitions decimal
14 \ -------------------------------------------------------------------------------
15    String     new xml-file
16    GtkBuilder new builder
17    GtkDialog  new dialog1
18    GtkWindow  new window1             GSignalSlot new cb.destroy
19                                       GSignalSlot new cb.delete
20 
21    s" euroFORTH-2010/GtkBuilder/example1.glade" xml-file !
22 
23    : on.destroy ( data oid -- ) 2drop gtk main_quit ;
24 
25    : on.delete-event ( event data oid -- f ) 2drop drop
26      xml-file @ s" dialog1" builder init
27      dialog1 init_from_builder  dialog1 run  dialog1 destroy ;
28 
29    : run ( -- )
30      xml-file @ s" window1" builder init
31      window1 init_from_builder
32      ['] on.destroy 0 window1 signal destroy cb.destroy connect
33      ['] on.delete-event 0 window1 signal delete-event cb.delete connect
34      window1 show_all ;
35 
36    run term? [if] cr ?? [else] gtk main bye [then]
37 \ -------------------------------------------------------------------------------
38 \ Last revision: MM-100903
```

Figure 4: The FORTH code for the GUI created with Glade.

desired layout. Widgets that do not need to be manipulated by the program require no code at all.

Another advantage is that the GUI can be changed aesthetically and widget properties can be changed, without the need to change the code. The only restriction is not to change the widget names.

To see the benefit of using Glade take a look at Fig. 6 and 7. It's the listing of the code that is required to create the same small GUI example as in Fig. 4 but without using Glade.

Obviously the difference is significant and can be expected to be much more significant when writing real applications instead of small examples.

And - the same is true when using Libglade instead of Libgtk. The advantage of Libgtk is, that Libglade is not required at runtime.

References

[1] Andrew Krause. Foundations of GTK+ Development. Apress, 2007.

[2] Matthias Warkus. Das GTK+/GNOME Entwicklerhandbuch. dpunkt.verlag, 2008.

[3] http://www.gtk.org/documentation.html

```
 1 \  euroFORTH-2010/GtkDialog/example1.mf
 2 \ ---------------------------------------------------------------------------------
 3 \  GtkDialog example1      OOP Library for MINFORTH                   MM-100801
 4 \ ---------------------------------------------------------------------------------
 5 \               Copyright (C) 2010 manfred.mahlow@forth-ev.de
 6 \
 7 \ ---------------------------------------------------------------------------------
 8   requires GtkToplevel
 9   requires GtkDialog
10   requires GtkHBox
11   requires GtkImageFromStock
12   requires GtkLabel
13
14   forth definitions decimal
15 \ ---------------------------------------------------------------------------------
16   GtkDialog   new dialog1
17   GtkHBox     new  hbox2
18   GtkImage    new   image1
19   GtkLabel    new    label2
20   String      new    markup2
21   GtkLabel    new  label1
22   String      new  markup1
23   GtkToplevel new window1          GSignalSlot new cb.delete-event
24                                    GSignalSlot new cb.destroy
25
26   176 chars markup1 init
27   s" <b>Using <span size='xx-large'>GTKBuilder</span> " markup1 !
28   s" with MINFORTH 1.5(p)</b>" markup1 +!cr
29   s" " markup1 +!cr
30   s" <span size='x-large'>This is a toplevel window.</span>" markup1 +!cr
31   s" " markup1 +!cr
32   s" Closing this window will popup a modal dialog" markup1 +!
33
34   118 chars markup2 init
35   s" <b>Quit this Application ?</b>" markup2 !cr
36   s" " markup2 +!cr
37   s" Please click on the Cancel Button, if you" markup2 +!cr
38   s" do <u>not</u> want to quit this application." markup2 +!
39
40   : destroy ( data oid -- ) 2drop  gtk main_quit ;
41
42   : delete-event ( event data oid -- f )
43     nip nip \ event and data are not used here
44     s" GtkDialog example1" dialog1 init  GtkWindow @ dialog1 transient-for !
45     6 dialog1 border-width !  dialog1 resizable no
46
```

Figure 6: FORTH code to create the GUI without using Glade, page 1.

```
47     false 6 hbox2 init dialog1 vbox pack_start_defaults
48     s" gtk-dialog-question" image1 from-stock dialog-size init
49     hbox2 pack_start_defaults  6 image1 xpad !
50     markup2 @ label2 init hbox2 pack_end_defaults
51     label2 use-markup yes  label2 justify center  12 label2 ypad !
52
53     s" gtk-cancel" -4  dialog1 add_button
54     s" gtk-quit" false dialog1 add_button
55     false dialog1 default-response !  dialog1 run  dialog1 destroy ;
56
57   : run ( -- )
58     s" GtkDialog example1" window1 init
59     24 window1 border-width !  window1 position center
60
61     markup1 @ label1 init window1 add
62     label1 use-markup yes  label1 wrap yes  label1 justify center
63
64     ['] delete-event 0 window1 signal delete-event cb.delete-event connect
65     ['] destroy 0 window1 signal destroy cb.destroy connect
66
67     window1 show_all ;
68
69  run term? [if] cr ?? [else] gtk main bye [then]
70 \ ---------------------------------------------------------------------------------
71 \ Last revision: MM-100903
```

Figure 7: FORTH code to create the GUI without using Glade, page 2.

Forth concurrency for the 21st century

Andrew Haley

Where are we going?

Shared-memory multiprocessors are the dominant technology for servers and desktops

Today we have four cores per die

According to Moore's law, in ten years we'll have 100 cores per die

But they may not be very much faster than the cores we have today

We need language support so that normal human beings can program these beasts

Where are we?

Moore's law has not been cancelled: every 18 months, the number of transistors per unit area doubles

However, clock speeds have not been increasing for several years, and if anything have got slightly slower

Performance can still be increased with new pipeline and cache designs, but not by much.

There seems to be a 4 GHz barrier

Where were we?

Chuck Moore's Forth multi-tasking design, from early 1970s:

Round-robin scheduler

Non-preemptive

Because of the lack of preemption, this design is very easy to use, because

You don't have to lock data structures unless there is a PAUSE or I/O

However

This design doesn't work for shared-memory multiprocessors, where there are several cores working on the same memory at the same time

Where are we?

Alternatively, lock-free data structures using CompareAndSet

But almost no-one in the world knows how to program them, and even those few people make mistakes

The principal difficulty is that synchronization primitives such as CompareAndSet work on only a single word, and this often forces a complex and unnatural structure on algorithms

Even a lock-free queue is an order of magnitude more complex

Lock-free structures are not composable either

Where are we now?

Almost no concurrency support in the Forth language standard

Some Forths use language support from OS: POSIX threads

GET and RELEASE primitives using mutexes for shared data structures

Difficult and unreliable to program, and doesn't scale well: deadlocks, races, etc.

The heart of the problem is that no-one knows how to organize and maintain large systems that rely on locking

Locks are not *composable*

In summary

Locks are hard to manage effectively

CompareAndSet operates on only one word at a time, resulting in complex algorithms

It is difficult to compose multiple calls to multiple objects into atomic units

redhat

Transactions and Atomicity

Wouldn't it be nice if we could say

```
begin-atomic
  x @ if  x foo  then
  true y !
end-atomic
```

Everything between **`begin-atomic`** and **`end-atomic`** is in an uninterruptable transaction – as long as we don't do any I/O

The code in **`foo`** also executes as part of this transaction

Programming this model would be just like the “old” Forth round-robin multitasker

redhat

Transactional memory

For every atomic block, there are two possibilities

The transaction *commits*, so its results become visible outside the atomic block

The transaction *aborts*, and it leaves the program's state unchanged

If the Transactional Memory (TM) system detects a collision between transactions, it aborts one or more of them and re-executes those that have failed

This process of re-execution is not visible to the program

redhat

Transactions and Atomicity

In a simple single-core system, **`begin-atomic`** and **`end-atomic`** don't have to do anything except ensure that no task switch occurs

In the case of a round-robin scheduler, they don't have to do anything at all

redhat

Types of Transactional Memory system

Consistent and Inconsistent

- Inconsistent TMs can lead to e.g. segfaults and exceptions

Fine-grained and coarse-grained

- Fine-grained TMs work on cells. Each time a cell in memory is accessed, the TM makes sure no other transaction has altered the cell since this transaction began
- Coarse-grained TMs work on entire objects in memory, but Forth has no idea what an object is

TM for Forth must be *consistent* and *fine-grained*

redhat

Types of Transactional Memory system

- Deferred or direct update
  - In a *direct-update* TM, writes are done immediately to memory. The system must record the original value of an object so that if a transaction has to be aborted it can be restored
  - In a *deferred-update* TM, the system updates an object in a location private to the transaction, and only writes the real object when the transaction succeeds

redhat

TL2

- Uses commit-time locking and a global version clock
- Fine-grained, consistent, deferred-update

redhat

TL2

- Dice, Shalev, and Shavit, Transactional Locking II, DISC 2006
- The front-running Software Transactional Memory system – IMO
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.811&rep=rep1&type=pdf

redhat

TL2

- Uses commit-time locking and a global version clock
- Fine-grained, consistent, deferred-update
- A global version-clock is incremented once by each transaction that writes to memory, and is read by all transactions.
- Every cell in memory has a corresponding lock that contains a version number
- Transactions start by reading the global version-clock and validating every location read against this clock
- This guarantees that only consistent memory views are ever read

TL2

- Writing transactions maintain a *write set.* This is the set of (address,value) pairs to be committed at the end of the transaction
- Writing transactions also maintain a *read set.* This is the set of addresses that have been read during the transaction
- Writing transactions need a read set but read-only ones do not

TL2

- Low-Cost Read-Only Transactions
  - Sample the global version-clock
  - Run through a speculative execution
- This is very fast

TL2

- Writing Transactions
  - Sample the global version-clock
  - Run through a speculative execution
  - Lock the write-set
  - Increment global version-clock
  - Validate the read-set
  - Commit and release the locks

TL2

- Low Contention Global version-clock Implementation
  - Tricky, but works. Read the paper

STM in Forth

Save the top N elements of the data stack (on the return stack)

Execute the transaction

If the transaction committed, throw away the saved stack items

If the transaction aborted, restore the top N elements of the data stack (from the return stack) and retry the transaction

If the transaction threw an exception, throw away the saved stack items and re-throw the exception

STM in Forth

We only need two new primitives, **`begin-atomic`** and **`end-atomic`**

But for good read-only transaction performance, we also want **`begin-readonly-atomic`**

`Begin-atomic` selects a wordlist that contains transactional versions of `:` `;` `@` `!`

Partial-word writes such as **`c!`** are hard but can be defined by using transactional `@` and `!`

Don't use words such as **`cmove`** in a transaction

Don't do any I/O in a transaction

STM in Forth

```
'transaction @ if
    execute  // We're already in a transaction
else
    'transaction !
    10 depth min n>r  // Save 10 cells
    begin  nr@ drop
        ['] do-transaction catch
        dup retrytx = while
            drop  repeat
    throw
    nrdrop
    0 'transaction !
then
```

STM in Forth

STM is going to be the only game in town

Although atomic transactions have been used in databases forever, Software Transactional Memory is very new. The key papers only date from a few years ago

Future processors will have hardware support for TM

GCC will soon have STM, but it's going to be a while before it's in Standard C

STM is not even on Java's radar

The **`atomic`** Forth primitives scale beautifully from the smallest embedded system to the largest multi-CPU server

Forth could be one of the first languages with STM support.

STM in Forth

There is no law that says Forth must be trailing-edge!

Questions?

uCore progress

with remarks on arithmetic overflow

Klaus Schleisiek

SEND Off-Shore Electronics GmbH, Hamburg

ks @ send.de

State of uCore affairs

- uCore 1.x is finished.
- Version 1.65 defines a very rich instruction set. Because of its co-design environment, it can be safely reduced to match application needs.

Progress during the past year:

- New application using Lattice LFXP2-17E as a "single chip controller"
- Cross-compiler interprets CONSTANTS.VHD and therefore, uCore is a co-design environment building upon one single source
- Step instructions for UM/MOD and FM/MOD executing in #bits+2 cycles
- Meticulous overhaul of all arithmetic overflow conditions
- Deterministic results on overflow
- Simplification of the bit-wise writable register mechanism

Co-design environment

- In a hardware / software co-design environment, both the hardware and the software can be simulated in a unified environment.
- Changes in the hardware should directly modify the software as well. Otherwise, hardware and software may deviate, working with inconsistent processors models.
- For uCore this means that the Forth cross-compiler must derive its "knowledge" about the architecture and the instruction set from uCore's VHDL specification.
- This is defined in CONSTANTS.VHD

VHDL code interpretation

Therefore, CONSTANTS.VHD has to be loaded into Forth before loading the cross-compiler itself

```
include vhdl.fs
include ../uCore/constants.vhd
include microcore.fs
include disasm.fs
include constants.fs
```

VHDL code to be interpreted

Various constants that define compiler switches, bus widths, register addresses, and control values:

```
CONSTANT with_mult        : STD_LOGIC := '0';
CONSTANT data_width       : NATURAL := 24;
CONSTANT flag_reg         : INTEGER := -2;
CONSTANT mark_start       : byte := "00110011";
```

... and instructions:

```
--ALU POP \ and PUSH
CONSTANT op_ADD     : inst_group := "000";
CONSTANT op_ADC     : inst_group := "001";
CONSTANT op_SUB     : inst_group := "010";
CONSTANT op_SSUB    : inst_group := "011";
CONSTANT op_AND     : inst_group := "100";
CONSTANT op_OR      : inst_group := "101";
CONSTANT op_XOR     : inst_group := "110";
CONSTANT op_NIP     : inst_group := "111";
```

VHDL code interpretation

In addition, the VHDL source has to be beefed up by additional words, which control Forth interpretation. These words all start with –– turning them into comments for the VHDL compiler.

```
--VHDL --~ ------------------------------------------------------------
-- SST100 - constants.vhd
-- ---------------------------------------------------------------------

LIBRARY IEEE;
USE     IEEE.STD_LOGIC_1164.ALL;
USE     work.functions.ALL;

PACKAGE constants IS
--~--  \ when loading the Forth cross-compiler, code between "--~" up to "--~--" will be skipped.
CONSTANT version          : NATURAL := 1100;
CONSTANT with_mult        : STD_LOGIC := '1'; -- '1' when FPGA has hardware multiply resources
CONSTANT data_width       : NATURAL := 24;    -- data bus width
--~
... more "VHDL only" code
--~--
--BRA POP
CONSTANT op_ALWAYS : inst_group := "000"; -- ELSE, REPEAT
CONSTANT op_QZERO  : inst_group := "001"; --      ?dup IF
CONSTANT op_SIGN   : inst_group := "010"; --     0< 0= IF
CONSTANT op_NSIGN  : inst_group := "011"; --        0< IF
CONSTANT op_ZERO   : inst_group := "100"; --           IF
CONSTANT op_NZERO  : inst_group := "101"; --        0= IF
CONSTANT op_NOVL   : inst_group := "110"; --     ovfl? IF
```

cross-compilation consequences

- Now e.g. `op_ZERO` has been compiled into the Forth dictionary as a constant holding the instruction's bit pattern and we can use it to define a code compiler for the target system

  ```
  op_ZERO Brn: 0=BRANCH ( f addr -- )
  ```

  which later on will be used appropriately by `IF`, `WHILE`, and `UNTIL`.
- This way any change in the VHDL code will automatically be ported to the cross-compiler.
- In addition: When an instruction has been removed from the VHDL code, because it is not needed in the application, the cross-compiler will get a hiccup when it has not been removed there as well.

unsigned Division

At first I started with unsigned division `um/mod`, because it is easy.
Three instructions are needed:

`op_UDIVS`	sets up the parameters
`op_DIVS`	the basic divison step executed once per bit
`op_UDIVL`	corrects the result and cleans up the stacks

`op_DIVS` holds its parameters in `TOS`, `NOS`, `TOR`, and the status register and therefore, it is fully interruptible.

```
: um/mod ( ud u -- urem uquot )
   udivs data_width times divs udivl ;
```

signed Division

Signed division was a problem for a long time. Until it ocurred to me that the obscure high level definition of `fm/mod` based on `um/mod` can be translated into VHDL quite easily.

```
: fm/mod  ( d n -- rem quot )
   dup >r   abs >r   dup 0< IF  r@ +  THEN
   r>  um/mod   r@ 0<
   IF  negate over IF  swap r@ + swap 1-  THEN
   THEN  rdrop ;
```

Therefore, signed division just needs two more instructions and, unfortunately, two more bits in the status register to remember the signs of the arguments

op_SDIVS	does the "intro" code
op_DIVS	identical to the "unsigned" step instruction
op_SDIVL	does the "correction" code

Overflow

- Once signed division was defined "in hardware" it was possible to set the overflow bit of the status register without run time penalty.
- Division overflow is quite complex adding a considerable amount of logic.
- Remains the `*` operation. Very often this is implemented as

```
: * ( n1 n2 -- n3 )   um* drop ;
```

  which delivers a misleading result in case of an overflow.
- Therefore, two more primitives have been added when multiply hardware is available implementing `m*` as a single cycle instruction

```
: * ( n1 n2 -- n3 )   m* multl ;
```

- Debugging was tricky.
  Reducing the data width to 8 bits allowed to do a complete test of all possible cases in about 15 minutes. This uncovered multiplication bugs as well.

? what to do on overflow ?

- Now the overflow bit of the status register is set/reset in a mathematically correct way.

So what?

The result is bogus nevertheless.

- We can branch depending on the overflow using

ovfl? IF	which could be a single cycle branch instruction
?ovfl	which is a conditional call to a fixed address on overflow

  but that adds runtime overhead and even if the program knows there was an overflow, the programmer may not know what to do.
  After all, the "division by 0" blue-screen of Windows is not really a meaningful result.

Returning a well defined result

- On overflow, we can return the "smallest" or the "largest" number that can be represented, i.e. $8000 or $7FFF in a 16 bit system instead of misleading bit patterns. E.g.:

+n 0 /	returns $7FFF
-n 0 /	returns $8000

- This leaves some cases, which are not so obvious:

0 0 /	I decided that it should return zero.
$8000 negate	If it returns $7FFF, e.g. the high level code for `fm/mod` does not work any more. So we better leave it at $8000, although it is intuitively as wrong as it can be.

- Does a commercial controller with "controlled overflow" exist?

Not much has been published about arithmetic overflow!

Setting register bits

- A more efficient mechanism to realize "bit-wise writable registers" has been found.

`5 Ctrl-reg !`	sets bits 0 and 2 of the memory mapped control register without affecting the other bits of the register.
`5 invert Ctrl-reg !`	resets bits 0 and 2 without affecting the other bits.

- The sign-bit of the number stored into the register determines whether the number will be ored (sign-bit=0) or anded (sign-bit=1) with the content of the register.
- Compared to the previous mechanism, the code is more readable, more efficient, and bit_0 of the register can be used as a flag as well.

net2o: vapor → reality

Bernd Paysan

EuroForth 2010, Hamburg

Looking for a challange

- As presented last year on EuroForth, the challange I'm looking at is a clean slate reimplementation of "the Internet"
- My previous company managed to sell me with my team instead of firing us—so the planned sabbatical doesn't happen
- This means it will take more time, but on the other hand it has to be simpler and more compact
- This talk is partly status report and much more a list of things to do
- IETF discussions about strategic internet development indicate that I'm on the right track

Outline

1. Motivation
   - No sabbatical, but also no real challange
   - Recap: Requirements
2. Recap: Topology
   - Recap: Packet Header
3. Implementation Status
   - Data Structures
   - Working Stuff
4. Todo-List
   - Flow Control
   - Cryptography
   - Browser

Recap: Requirements

Scalability Must work well with low and high bandwidths, loose and tightly coupled systems, few and many hosts connected together over short to far distances.

Easy to implement Must work with a minimum of effort, must allow small and cheap devices to connect. One idea is to replace "busses" like USB and firewire with cheap LAN links.

Security Users want authentication and authorization, but also anonymity and privacy. Firewalls and similar gatekeepers (load balancers, etc.) are common.

Media capable This requires real-time capabilities, pre-allocated bandwidth and other QoS features, end-to-end.

Transparency Must be able to work together with other networks (especially Internet 1.0).

Switching Packets, Routing Connections

- Similar to MPLS, packets should run through a switching network, not through routers
- Routing is a combination of DNS (name resolution) and routing calculation (destination lookup)

Physical Route

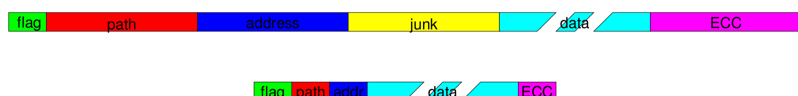
- Take first n bits of target address and select destination
- Shift target address by n
- Insert bit-reversed source into address field

Starting Point

- As starting point, I first implement net2o using UDP as transport layer
- UDP offers a reaonable interface for a single server that handles many connections without crazy Unix overhead
- For start, IPv4 only; IPv6 requires more work (no fragmented packets possible)
- Two parts: Packet server and command generator/interpreter

Recap: Packet Header

	Size
Flags	*2*
Path	*2/8*
Address	*2/8*
Junk	*0/8*
Data	*32/128/512/2k*
ECC	*L1 dependent*



Switching

- Use a hash for "switching" IP-Addresses: Hash value equals prefix
- Hash collissions resolved with longer prefixes
- Prefix granularity: Byte

  MSB=0 Direct routing entry
  MSB=1 larger prefix, look at next byte for more data

Shared Memory

- Map from address to connection context
- Connection context (will) contain
  - real addresses
  - file handles
  - cryptographic keys
  - authentication information
  - and other status information (a lot of that still unimplemented)
- Event queue for received packets

Working Testcase

Server loop

```
init-server
server-loop
```

Debugging output

```
init-client
s'' localhost'' net2o-udp insert-ipv4
          constant lserver
net2o-code s'' This is a test'' $, type
          '!' char, emit cr end-code
cmdbuf cell+ 0 lserver sendA
```


Commands

- UTF-8 encoded commands: Simple commands are 0-7F, one byte, complexer commands take more bytes
- Commands packet into 8 byte chunks
- 8 byte literals (e.g. addresses) and strings embedded into the command structure
- Command assembler allows seamless commands within Forth code
- Discussion: offsets to literals as UTF-8 code?

Flow Control

- UDP offers no quality of service
- TCP/IP flow control is horribly broken, assumes no buffers—reality are buffers everywhere, filled up completely by TCP/IP (causing horribly lags)
- Idea: PLL-based flow control, tries to prevent buffers filling up
- “Fast start:” Send first few packets out as fast as possible, to measure actual data rate

Cryptography

- Ellyptic Curve Cryptography code for the assymmetric part (much faster than RSA, a lot stronger per bit)
- Wurstkessel as symmetric cryptography and hashes
- Ubiquituous encryption is very important for network neutrality!

Presentation/Browser

- Typesetting engine
- Embedding of images, audio, and video—but please no plugins!
- Properly secured scripting (needs to be simple enough for that!)

Summary

- There is already a little bit of code
- A lot more work for long dark winter evenings
- After completion of reference implementation: RFC, IETF discussions, presentations at larger network-related conferences

For Further Reading I

Bernd Paysan
Internet 2.0
http://www.jwdt.com/~paysan/internet-2.0.html

The Forth Net

Gerald Wodni* M. Anton Ertl †

September 24, 2010

Abstract

CPAN and PECL are impressive ways of sharing custom libraries. Projects are discussed, hosted and downloaded. Their dependencies are clear (no need to search across the web) and also downloaded at once. There is no such web portal for Forth — until now.

1 Introduction

When working on the SWIG - Gforth Extension, we designed a platform independent file format for C interfaces called FSI[1]. Creating such files requires SWIG[2] to be installed and some understanding of the C interface as well as the library. Thats when we thought about having a central place to put FSI files, which are just downloaded and compiled using a normal C compiler (much likelier to meet at the end users system than SWIG). When hosting such libraries on a website, users also want to share code examples, host projects built on top of these libraries, as well as discussing about libraries and projects.

So instead of creating a FSI host and exchange website, we created a Forth portal capable of more than that. We want developers to be able to share their projects, get some feedback, explain the usage of their work and define dependencies to other projects. Users on the other hand should be allowed to browse through all projects, find related projects and download the source code.

2 Related Work

Sourceforge[3] provides easy creation of projects, but the relation between them is not always obvious. Downloading requires human interaction and could be cumbersome if you have to look up dependencies by yourself. Access is granted by using OpenID[4], so if one already precesses an OpenID, no registration is required.

The Comprehensive Perl Archive Network(CPAN)[5] supplies developers with their own web space. Read access is publicly available, write access is only allowed to the author. The registration process is human driven, one is approved as contributer after filling out a registration form and wait for up to 3 weeks. Using modules is easy as the download process will inform you about all dependencies, and allow you to download them at once.

The PHP Extension Community Library(PECL)[6] is similar to CPAN but registration works via normal email confirmation form.

Forth also has a website for sharing libraries called Forth Library Action Group (FLAG)[7]. It is operated by a steering committee which manages the accounts. Every library's "champion" is responsible for keeping his stable release up to date and available through FLAG.

3 Features

In order to attract users, and fit into the social web, we used some Web2.0 techniques, and tried to simplify processes on the website. We also considered related websites and picked up some of their ideas.

Login No registration is required, login is done with OpenID[4], so becoming a user of the Forth net is a matter of seconds. If one owns no OpenID, he is free to choose from many existing providers, or even become provider himself[8].

Projects Every user who is logged in can create a new project. To point out this feature and make people contribute, the "Create" menu is visible at all times. Project names are only allowed to contain alphanumerical letters and minus '-', that way they can directly be used as part of a URI or as definition names in Forth.

Tags As a hierarchical system of categorization can never quite serve the description of a project and sometimes make it even harder to find because one thought of it to be in another category, we only use tags. The author can assign Tags that fit his project, if a tag is not within the database, it will be created as soon as requested. To avoid a big amount of tags, they

*TU Wien; gerald.wodni@gee.at

†TU Wien; anton@mips.complang.tuwien.ac.at

are only allowed to contain letters and numbers and are case insensitive. Popular tags like Forth Systems are marked as popular by the administrators and get better rankings, so users are encouraged to use them.

Personalization When dealing with lots of users within a comment section, its hard to remember who is who. Small avatars allow quick association of replies, to use the social web again, we included Gravatar[9]. It allows users to host multiple avatars at a central place and make use of their fast content distribution network. Once a user has logged on to the Forth net and enters his email address, the MD5 checksum of it used to reference his image on Gravatar. If none is set, Gravatar supplies a random geometric pattern using the email address as seed.

URIs Instead of using old fashioned URIs with lots or parameters, e.g.: `/index.php?display=cont&user=42&si=...` pretty URIs are used: `/projects/the-Forth-net`. This way users and visitors quickly realize how the URI works and could easily link to them.

Every user has his own profile site where projects managed by him are displayed and other users can send him private messages.

4 Conclusion

The Forth net aims to be the de facto standard for sharing forth libraries some day. By using established Web2.0 technologies such as OpenID and Gravatar, the threshold of becoming a project maintainer is much lower than in other networks where contact to the hoster needs to be made first. Using pretty-URIs, search engines and users can easily see the link between the URI and the content and refer to the homepage.

5 Further Work

5.1 fget

Instead of letting the user struggle with keeping his local library copies up to date and resolve any dependencies to others, a download manager – working titled "fget" – could do this for him. The web server will have special access features with no markup for this sole purpose.

5.2 Crawler

To minimize the effort for developers, a crawler could collect the most up to date version of a project from a given URI. As several security issues become relevant this feature will only be allowed to users who have been approved by the administrators.

References

[1] Gerald Wodni. SWIG - Gforth Extension (Bachelor Thesis), 2010.

[2] David M. Beazley et al. Simplified Wrapper and Interface Generator (SWIG). URL `http://www.swig.org`.

[3] SourcForge. URL `http://sourceforge.net`.

[4] OpenID. URL `http://openid.net`.

[5] Comprehensive Perl Archive Network. URL `http://www.cpan.org/`.

[6] The PHP Extension Community Library. URL `http://pecl.php.net/`.

[7] Forth Library Action Group. URL `http://soton.mpeforth.com/flag/`.

[8] OpenID Explained. URL `http://openidexplained.com/`.

[9] Gravatar. URL `http://gravatar.com`.