Ficl – an embeddable extension language interpreter

Forth for the rest of us

John Sadler
john_sadler@alum.mit.edu

1 Introduction

Glue languages like PERL, tcl, and Python are popular because they help you get results quickly. They make quick work of problems that are often tedious to code in C or C++, and they can work with code written in other languages. People often miss this last benefit because PERL or Python usually run standalone. Tcl, on the other hand, was designed from scratch as an extension language and is relatively simple to insert into another program. All of these languages were designed to work with mainstream operating systems, so they require lots of memory, a file system, and other resources that are commonplace on a modern PC or workstation. In this article, I'll describe an interpreter that has a system interface similar in spirit to tcl, but is specifically designed for embedded systems with minimal resources. The syntax is ANS Forth, so I've called it ficl, or Forth-Inspired Command Language.

Ficl is Forth, but you don't have to be a rabid Forth zealot to use it. In fact, you can do useful work without knowing any Forth at all. On the other hand, you can learn enough in a half-hour to do useful work. Read on, and I'll show you. (From here on, I'll use the words Forth and ficl interchangeably.)

Why not write your own command interpreter? While tempting, this approach has its disadvantages. For one thing, you usually wind up with something that has a line-oriented syntax where the command is at the beginning and any arguments come afterwards, and you can do one command per line. This is enough in many cases, and ficl can do this for you, too. But what if your customer asks for a behavior that's not hard-coded in the interpreter? What if it takes too long to transmit the command stream to run the product? How about diagnosing field failures? Ficl is a small but complete programming language, not just a command interpreter, so you can hack together a prototype for that new behavior. If communication overhead is too high, consider downloading common groups of commands as stored programs, or "words" in Forth, and you can invoke them with a single alias. Because ficl is interactive, you can use it to help troubleshoot problems. One more advantage of using ficl is that you can find several good tutorials for free on the Web (see the references), so you save time on documentation and training: you only need to document your extensions to the language.

As I implied above, people who do not program for a living can do useful work with ficl. Your friendly neighborhood electrical engineer can bench test new hardware quite easily using ficl, saving your time to add features or fix your own bugs (not that there will be any). You can use a minimal ficl system to do rapid prototyping on new hardware,

getting new features to a demonstrable state normally in much less time than would be required with a compiled language.

2 Using the language

2.1 Enough ficl syntax to make you dangerous

In the examples that follow, you may wish to download the ficlwin executable and play with the language. Ficlwin has some simulated hardware that will be useful later. Ficlwin runs under Windows 95 or NT.

First Rule of ficl: use spaces to separate everything. Forth is very simple-minded about parsing its input: it looks for space-delimited tokens, and tries to interpret them one by one.

Second Rule of ficl: if it's not a word, try to make it a number. If that doesn't work, it's an error. A word is a named piece of code (like a function or subroutine) that may also own some data. Words are organized into a list called a dictionary. For each token in the input stream, the interpreter tries to find a word in the dictionary with the same name. If successful, the interpreter will execute the word. Otherwise, the interpreter attempts to convert the token to a number. If this fails, you get an error message. By the way, the Second Rule means that you can do Evil Things like redefining your favorite number.

Third Rule of ficl: Words find their arguments on a stack. The interpreter pushes numbers onto the stack automatically. The language does not have the equivalent of a function prototype, so Forth programmers use comments to show the state of the stack before and after execution of a word. For example:

```
+     ( a b -- c )
```

indicates that the word "+" consumes two values from the stack (a and b) and leaves a third (c, the sum). By the way, an open paren followed by a space tells the interpreter to treat everything up to the next close paren as a comment. You can comment to the end of a line with a backslash character followed by a space.

Here's how you add two numbers in Forth (if you've ever used an RPN calculator, you will be in familiar territory):

```
2 3 +  .
```

The interpreter pushes the number 2 and the number 3, then executes the word "plus" and finally executes the word "dot". This prints the top value on the stack, which I hope is five.

There are really two stacks in Forth: one stores return addresses and the other stores parameters. We'll refer to the parameter stack simply as "the stack" from now on.

Ficl is not case sensitive. Word names can be up to 255 characters long, but only the first 31 characters will be stored. All words are stored in a linked list called the dictionary. There exist words to allot space in the dictionary so that (other) words can have arbitrary size data areas.

Ficl has these main data structures:

- A virtual machine stores one execution context, and would typically map to a thread.
- Each virtual machine has two stacks, one for parameters and the other for return addresses.
- A word binds a name to code, and optionally, data.
- The dictionary is a list of all words of the system.

See Section 5 for more ficl syntax.

2.2 Porting ficl

As shown in Example 1, it only takes a few lines of code to hook ficl into your system: initialize the ficl system data structures with a call to ficlInitSystem, and create one or more virtual machines using ficlNewVM. After that, you simply feed blocks of text to the virtual machine from an I/O device, a file, or stored strings using ficlExec. You can tear down memory allocated to ficl with ficlTermSystem.

Example 1: Code fragment to initialize and run FICL

```
FICL_VM *pVM;
ficlInitSystem(10000); /* create a 10,000 cell dictionary */
pVM = ficlNewVM();

for (;;)
{
    int ret;
    gets(in);
    ret = ficlExec(pVM, in);
    if (ret == VM_USEREXIT)
    {
        ficlTermSystem();
        break;
    }
}
```

Ficl requires an ANSI C compiler and its runtime library to build and execute. Porting to a new CPU involves editing two files: sysdep.c and sysdep.h. The header file contains macros that control build properties of ficl, and macros that insulate the implementation from differences among compilers. Ficl interfaces to the operating system via four functions:

- FiclMalloc and ficlFree map closely to the standard C malloc and free functions, but they act as a choke-point for ficl memory management in case your OS has specialized functions for this purpose or (gasp!) you've rolled your own.
- ficlLockDictionary provides exclusive access to the dictionary in multithreaded implementations. If you only intend to make one ficl Virtual Machine, this function can be empty.
- ficlTextOut is like standard C's puts, except that it takes a virtual machine pointer and one additional parameter to indicate whether to terminate the string with a newline sequence.

Because ficl is a 32 bit Forth, the language requires some 64-bit math. There are two unsigned primitives in sysdep.c that handle this. One function multiplies two 32-bit values to yield a 64-bit result, and the other divides a 64-bit value by a 32-bit value to return a 32-bit quotient and remainder. These are usually simple to implement as inline assembly for a 32 bit CPU (see the Intel 386 example in the source). I was too lazy to come up with a generic version in C. If you're lazy too, you can kludge these functions to use only the low 32 bits of the 64 bit parameter and be safe – so long as you avoid multiplying and dividing really big numbers!
Memory requirements of the code vary by processor. The dictionary is the largest RAM-resident structure. The word-set that comes with the source requires fewer than 1000 cells or 4K bytes. Stacks default to 128 cells (512 bytes) each, so you can fit a useful implementation into 8K bytes RAM plus code space (which can be in ROM).
Use testmain.c as a guide to installing the ficl system and one or more virtual machines into your code. You do not need to include testmain.c in your build. The source package includes a Win32 executable that will help you get a feel for the language.

2.3 Roll you own extensions in C

You can extend the language with words that are specific to your application, written in C, in Forth, or in a mixture of C and Forth. Use the ficlBuild function to bind a C function to a name in the dictionary. Functions that implement ficl words take one parameter: a pointer to a FICL_VM. This pointer refers to the running virtual machine in whose context the word executes. The files words.c and testmain.c have (literally) hundreds of examples of words coded in C. Example 2 shows a function that interfaces ficl to the Win32 "chdir" service.

Example 2: example Ficl/C interface function

```
/*
** Ficl interface to _chdir (Win32)
** Gets a newline (or NULL) delimited string from the input
** and feeds it to the Win32 chdir function...
** Usage example:
**    cd c:\tmp
*/
static void ficlChDir(FICL_VM *pVM)
{
    FICL_STRING *pFS = (FICL_STRING *)pVM->pad;
    vmGetString(pVM, pFS, '\n');
    if (pFS->count > 0)
    {
        int err = _chdir(pFS->text);
        if (err)
        {
            vmTextOut(pVM, "Error: path not found", 1);
            vmThrow(pVM, VM_QUIT);
        }
    }
    return;
}
```

```
/* Here's the corresponding ficlBuild call...
** ficlBuild("cd",       ficlChDir,    FW_DEFAULT);
*/
```

To write a new ficl word in Forth, follow the examples in softcore.c: embed the source in a string constant, and feed the string to ficlExec once you have a virtual machine created. You'll also find some examples of words coded in a mixture of C and Forth in words.c (see `evaluate`, for example). Because ficlExec calls can be nested, you can invoke ficlExec from within a function that implements a word and feed it a string argument, effectively mixing the two languages.

3 Using ficl: an example

Here's a quick example of ficl at work. Suppose we have a simple target board that has a block of 8 LEDs, 8 DIP switches, an 8-bit analog to digital converter , and an 8 bit DAC. (Conveniently, this is what ficlwin simulates).Let's start by setting up an address map:

```
hex
10000 constant leds
10002 constant switches
20000 constant adc
20002 constant dac
```

What happened? The first line tells ficl that we're going to be writing numbers in hexadecimal. The next four lines set up named constants for the registers we want to use. When we invoke one of these constants (by typing its name), it pushes its value.

```
: !oreg  ( value -- )   leds c! ;
: @adc   ( -- value )   adc c@ ;
: !dac   ( value -- )   dac c! ;
: @ireg  ( -- value )   switches c@ ;
```

A constant pushes its value when invoked, and a variable pushes its address. The ficl word @ (the at-sign) fetches the contents of an address and puts the 32 bit value on the stack. ficl also has c@ to fetch a byte, and w@ to fetch 16 bits. Likewise, ! w! and c! store a value at an address. The syntax is (value address --), meaning that the operations consume a value and an address from the stack. The lines above create ficl words that hide the width of the registers they use by wrapping the fetch or store operation. Note: ficlwin implements the above four words so that you can try the rest of the code in this example verbatim.

```
variable led-shadow   0 led-shadow !
: !leds  ( value -- )   dup !oreg led-shadow ! ;
0 !leds
```

The first line above creates a variable to track the LED state. (Real hardware engineers often find it too expensive or too bothersome to add a readback capability to digital output registers.) The word on the second line writes the LED register and updates the shadow variable. The last line forces the LEDs off.

```
: toggle-led  ( led -- )
    1 swap lshift    \ make a bit-mask for the LED
    led-shadow @ xor \ toggle the bit in the shadow reg
    !oreg            \ now update the LED and shadow
;
```

This word toggles an LED by index (0..7). Lshift is equivalent to C's << operator.

```
: adc-loop
    begin
        @adc dup !dac 100 msec
    255 = until
;
```

The above word loops, writing the ADC value back to the DAC until the ADC value reaches 255. After each ADC sample, there is a 100 millisecond pause (msec is implemented in ficlwin with the Win32 Sleep function).
These are extremely simple examples, but they give a feel for the accretive process one uses to bring up hardware with ficl. I invite you to try them with ficlwin.

4 Where to find more information

Web sites:
Skip Carter's Taygeta site has lots of Forth archives, and is the Web home of ficl:
http://www.taygeta.com/ficl.html (latest Ficl information and downloads)
http://www.taygeta.com/forth.html (Check the Forth Literature links for downloadable books)
Or you can read the Draft Proposed American National Standard (and save yourself $300) – it's the most readable language standard I've seen. You can actually understand it.
http://www.taygeta.com/forth/dpans.html
The Forth Interest Group (loads of tutorials, interpreters, and other info):
http://www.fig.org/fig.html
A web-based introduction to Forth using an intuitive graphical notation:
http://forth.org/forth_intro/stackflo.htm
Another web-based Forth intro:
http://astro.pas.rochester.edu/Forth/forth.html

5 Ficl Quick Start

`.s`	Displays the contents of the stack non-destructively
`.`	Pops the top of stack and displays it
`Hex`	Set number base to hex (decimal is the default)
`Decimal`	Set number base to decimal
Arithmetic	
`+ - * / ( a b -- c )`	Pop a and b off the stack, perform the operation, and push the result. Example: `3 2 + ( leaves 5 on the stack)`
`Negate ( x -- -x )`	Change the sign of the value on top of the stack
Stack Operations	
`Dup ( x -- x x )`	Copy the top of stack
`Swap ( x y -- y x )`	Swap the top two cells
Logic	
`True ( -- -1 )`	
`False ( -- 0 )`	False is always zero in Forth, and true can be defined as `false invert` (all bits set logical and bitwise operations. Logical operations view any non-zero value as true, as

`And or xor  ( x y -- z )`	Perform BITWISE operations on two arguments and push the result
`Invert  ( x -- ~x )`	One's complement the top of stack
`< > <> =` `( x y -- flag )`	Perform comparison (<> means "not equal") and push true or false
Fetch and Store	
@ w@ c@ (address -- value)	Fetch 32, 16, or 8 bits respectively from address and leave the result on the stack. Valu padded to 32 bits
`! w! c!` `( value address -- )`	Store 32, 16, or 8 bits respectively of value at address. Note: if you think "@" and "!" a how about "*" for "dereference" and "&" for "address of"?
Creating New Words	

`Constant ( x -- ) "name"`	Creates a new word that pushes its value (x) when executed. Example:
	`0x10000 constant ram-base`
	`ram-base @  ( fetches 32 bits from address 0x10000 )`

`Variable ( -- ) "name"`	Creates a new word that pushes its address when executed. *Example*
	`variable v  ( creates a new variable named v )`
	`0 v !  ( set v to zero )`

Colon definitions	You can define a new word in terms of existing words using colon and semicolon, like `: v++  1 v +! ;  ( create new word named v++ )` Now when you execute `v++`, it increments `v`. But wait, there's more. In addition to chaining together words you can also use control colon definitions…
Iteration	*Example* `: testloop limit index do ( insert code here ) loop ;`

Conditionals	*Example*
	`: signum  ( x -- sign )`
	`\ push -1 if negative, 1 if positive, else 0`
	`  dup  ( -- x x )`
	`  0< if drop -1  ( -- x -1 )`
	`  else`
	`  0= if 0 else 1 endif`
	`  endif`
	`;`

.