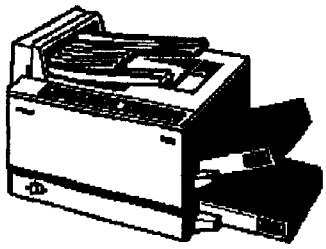

The PostScript™ Column

Paul Snow

Insights into PostScript

Continuing our progress towards embedded systems programming in a PostScript-compatible interpreter, I think we should address the foundation of PostScript. (For those of you tuning in late, this is not the twilight zone! Catch the previous article in the Newsletter, Vol.2 No.4 pg.5) In this article we will look at implementing PostScript-style control structures, names, and dictionaries. These concepts will form the basis for implementing a PostScript mini interpreter in Forth. We will also identify the major differences between Forth and PostScript, and nail down a word list that we will use to merge the two. And just to rankle and annoy, we'll toss in some C as well.

Legalities & Moralities

Before we dive in, let's explore our legal and moral position in this endeavor. In 1986 when Cliff Click and I starting working on a PostScript-compatible interpreter, it was not absolutely crystal clear that we were working within a proper moral/legal context, though I felt strongly that we were. Happily we can now (as of December of '90) lay all such nagging doubts to rest. In the latest edition of the Red Book[1] (which is no longer red, but mostly white and a light burnt orange) on page 10 Adobe gives permission to do the following (and I quote:)

- ✓ write programs in the PostScript language;
- ✓ write drivers to generate output consisting of PostScript language commands;
- ✓ write software to interpret programs written in the PostScript language; and
- ✓ copy Adobe's copyrighted list of commands to the extent necessary to use the PostScript language for the above purposes.

What Adobe goes on to say we *can't do* is call the results of our work **PostScript™**, or anything which might be confused with the PostScript trade mark. (Printware, a printer manufacturer, once called their interpreter "PrintScript". Adobe objected, and Printware changed their interpreter's name to "PrintStyle") So we have to call it something else. [Editor's Note: it's *your* turn, Mr. Reader: write Paul Snow or send him a GEnie message with your suggestions for a suitable name for this project!]

[1]From the Introduction of the PostScript Language Reference Manual, second edition by Adobe Systems Inc., Addison-Wesley, Dec 1990.

To Catch a Rabbit

While I have my new Red Book open, you may be interested what Adobe's current stance on Forth is. (Warning, this is a *rabbit* and has nothing to do with the subject at hand!) On page 23 Adobe says: "As with all programming languages, the PostScript language builds on elements and ideas from several of the great programming languages. The syntax most closely resembles that of the programming language Forth. It incorporates a postfix notation in which operators are preceded by their operands. The number of special characters is small and there are no reserved words."

The only other "great programming language" Adobe mentions is Lisp. This is the strongest endorsement I have ever read of Forth by anyone other than dyed-in-the-wool Forthers.

And so we find ourselves on firm legal and moral ground, programming in one of the *great programming languages,* inspired with the quest, and ready to tackle all problems at hand. So, let's do it!

Control Structures

PostScript differs significantly from C and Forth in how it implements control and data structures. We'll look at C, Forth and PostScript code and point out some differences, and write some Forth code to implement some simple PostScript control structures.

PostScript, like Lisp, makes heavy use of lambda objects to implement control structures. (A lambda object is a term I'm borrowing from Lisp to refer to a subroutine with no name.) The significance of this statement can sneak past you if your programming background is largely in C. For example in C you would code an **if** statement:

```
if(A == B)
  {
  printf(" A is equal to B!");
  }
```

The same kind of statement is coded in PostScript as:

```
A B eq { (A is equal to B!) print } if
```

A C programmer tends to think in terms of syntactical structures such as statements and declarations. This kind of thinking obscures important differences between C and PostScript. In this example, the critical difference is *not* that the { ... }'s appear after the **if** in C and before the **if** in PostScript, but that the **if** in PostScript is a bonafide operator, and the { ... } 's delimit a lambda object.

The Forth programmer will not be so easily mislead because she will tend to think in terms of data stack manipulations and functions. In fact, before the Forth programmer has finished reading these last two paragraphs a thought balloon has probably formed above her head containing Forth code that looks something like:

```
: { 0 [COMPILE] LITERAL [COMPILE] IF HERE
  >R ; IMMEDIATE

: } EXIT [COMPILE] ENDIF R> [COMPILE]
  LITERAL ; IMMEDIATE

: PS-IF SWAP IF EXECUTE ELSE DROP ENDIF ;
```

This code will allow the Forth programmer to code the above example in one of two ways. The first is the Forth "Standard" way, and the second relies on the Forth code in the thought balloon ...

Standard Forth Syntax:

```
A @ B @ = IF ." A IS EQUAL TO B!" ENDIF
```

PostScript-Style Syntax:

```
A @ B @ = { ." A IS EQUAL TO B!" } PS-IF
```

The balloon code given relies on a branch around the lambda object that could be avoided with a little bit of effort.

Now let's look at a PostScript **repeat** loop. This control structure takes a count and a lambda object and executes the lambda object that many times. Coding a simple example in our three languages yields:

In the C Language:

```
int i;
for(i=0;i++;i<10)
{
printf("*");
}
```

In Forth:

```
10 0 DO ." *" LOOP
```

And in PostScript:

```
10 {(*) print} repeat
```

Note that though all three examples print ten asterisks, the control structures used are very different. For example, should I wish modify the above code to produce a function that executes its parameter ten times, the result would be:

In the C Language:

```
(void) repeat10(*f())
{
int i;
for(i=0,i++;i<10)
{ f(); }
}
```

In Forth:

```
: REPEAT10
  >R 10 0 DO R@ EXECUTE LOOP
  R> DROP ;
```

And in PostScript:

```
/repeat10 {10 exch repeat} def
```

Note further that the Forth and PostScript implementations take into account the possibility that each call to the function may require parameters. So long as those parameters have been pushed onto the data/operand stack, both the Forth code and PostScript code will work fine.

The very interesting observation to make from the examples above is that PostScript's virtual machine does not branch. In fact *none* of PostScript's control structures require branching, conditional or otherwise. PostScript implements all normal control flow via calls and conditional calls to lambda objects. Whereas this is very similar to Lisp, PostScript lambda objects are implemented as arrays rather than lists. Since PostScript maintains a length with each array pointer, no end of subroutine marker or instruction is required (in a later article I may explore a microcontroller based on PostScript's control style; benefits of such a controller include easy virtual memory, simple effective memory caches, Really RISC (RRISC) processor design, etc.)

The most difficult PostScript control operators to implement are **stop** and **exit**. These operators have fairly complex behaviors similar to a smart **EXIT** word in Forth that works within **DO ... LOOP**s. As interesting as these operators are, I will put off addressing them to some later date.

PostScript Style Names

Function naming is another very unique aspect to PostScript. I implied above that all of PostScript's control structures are implemented via lambda objects including subroutines. In C and Forth, subroutines (ie: words) have a single name by which they can be referenced. Not so with PostScript. In fact, PostScript does not have any named functions or operators in the C or Forth sense. Instead, PostScript binds name objects to entities such as procedures, numbers, and strings through special hash tables called dictionaries.

Dictionaries are hash tables of key/value pairs. Given a key and a dictionary, the value for the key is found by hashing the key into the dictionary to yield the value pair.

Even though any PostScript object can be used as a Key in a dictionary, the most useful objects to use as keys are names. Names in PostScript are independent objects that are guaranteed to each represent a unique string. Nothing more or less. Names are lent meaning via dictionaries and the dictionary stack. When the PostScript interpreter needs the definition of a name, the dictionaries on the dictionary stack are searched in order, from the top of the stack to the bottom. When an entry is found in one of the dictionaries for the name in question, the value is used as the value for the name.

This is not unlike many Forth systems with their vocabulary stacks. The critical difference is that in PostScript a name's value is defined by the dictionary stack at run time, whereas with Forth names are defined at compile time. To implement PostScript-style

naming in either C or Forth, you need the following functions:

str2name (string -- name)
Hashes a the given string into the name table. If the string has not been encountered before, than a new entry in the name table is built. A pointer or offset to the entry in the name table is returned as the name.

name2str (name -- string)
A pointer to the string corresponding to the given name is returned.

dict (integer -- dict)
Creates a hash table of a given max length. To anyone writing this code from scratch, I would advise you to build dynamically sizable dictionaries. I would also advise you to build a header into the beginning of the dictionary giving its max length (even if it is dynamic) and other data about the dictionary.

lookup (key -- addr)
Returns the address of the entry in the dictionary corresponding to the given key. Should this key be undefined in the dictionary, the address should point to an empty slot in the dictionary which may be filled if so desired. If there is no more room in the dictionary (you implemented static dictionaries, or you are out of memory) then the address would be null, -1, odd -- whatever passes for an impossible address in your system.

def (key tag value tag dict --)
The key used with the dictionary can be anything, an integer, a float, whatever. To have reasonable performance the key should be a name. The value is also tagged so later you can treat it as an address or a number, or a procedure, etc. **def** adds the key value pair to the given dictionary. Since every dictionary has a header describing itself, a tag for the dictionary is not necessary.

To Sum Up...

In the previous column we covered the advantages of a PostScript shell for embedded systems, and now we have begun to dig into building one. I have a early version of a PostScript shell that I will be making available on GEnie. It is written in Fifth, and will run on IBM PC's and compatibles. (I will make sure our shareware version of Fifth is also be available.) For those of you that might like to move it to FPC or TIL, the code is largely Forth-83 compatible, but does rely on Fifth's "compile on demand" capablility.

If you have any PostScript questions, or a suggestion on *what to name* this PostScript based embedded system language we've been talking about, then I can be reached at:

The Software Construction Company, Inc.,
2900-B Longmire, College Station, TX 77845
or on GEnie by P.SNOW1

Paul Snow is an accomplished programmer and entrepreneur. His previous works include a unique 32-bit Forth for IBM-PC's called *Fifth*, and a clone of the Adobe PostScript controller language known as *X-Script*. Paul is a frequent contributor to Forth meetings and publications. He lives the life of a country squire with his family on the pleasant plains of College Station, Texas, near his alma mater (Texas A&M University).

continued from page 3:

I'll also still be writing the ANS Forth reports, and maybe I'll even have time for a technical article or two. Thank you for the opportunity to do something useful, and to leave what seems to be a positive mark in our profession.

Best of Luck to all!

George Shaw has been programming in Forth for 10 years. He has been around the Forth community for awhile: having been a referee on the Forth-83 standard, chairman of the Asilomar conference twice, chairman of the Silicon Valley or North Bay chapters of FIG for the last four years, a member of the ANS Forth committee, and the founder and chairman of ACM SIGForth. In his spare time he builds houses and is a consulting software engineer via his company, *Shaw Laboratories*. He also has been trained as a psychotherapist which he uses to prevent himself and the people around him from going insane while trying to keep up.

Coming Attractions

If you like **PostScript™** you'll like what's ahead in upcoming issues of the Newsletter!

We'll have another interesting PostScript column by our resident expert, Paul Snow.

There'll be a review of **PSTutor™** , a software program for the IBM-PC which can teach you how to write code in PostScript like a pro.

We hope to have additional articles on PostScript by well known authors for your enjoyment.

Forth-oriented articles include a superb hardware project by Peter Grabienski entitled: "FLIP-FLOP: A Stack-Oriented Multiprocessing System".

All this, and a few surprises for you! Stay posted!