New Loops: The Commodore 128 Basic Stack

Jim Butterfield
Toronto, Ontario

...Some programmers have a knack forgetting themselves tangled up in code...

Even if you're just a Basic person, you've likely heard about the "stack". That's the place where the computer leaves temporary notes for itself... how to get back from a subroutine or interrupt, plus other small bits of data.

From the Basic end, the stack holds two items of interest: information on live FOR/NEXT loops, and information on active subroutines. Whenever your program commands FOR. . . or GOSUB.. . the computer notes the command location and the current line number. That's so it will know where to come back to when it encounters a NEXT or RETURN as the case may be, and so it can reinstate the line number in case an error notice is needed.

That's all that is noted for a GOSUB, since all the program needs to know is how to RETURN. The FOR command puts away lots more data, however. It gives the variable identity, the step value, the loop limit, and whether the loop is counting up or down (say, with a STEP -1). That way, NEXT can modify the loop variable by the right amount and then test to see whether to go around the loop again.

All this adds up to: seven items on the stack to log a GOSUB, and eighteen items to record a FOR. These items are reclaimed later - at least they should be. RETURN wipes out the GOSUB entry, and NEXT will kill the FOR entry when the loop has been exercised the proper number of times.

On earlier Commodore machines, this stuff went onto the "hardware" stack. That's the area from 506 going down to 320 (hex $O1 FA down to $0140), or about 186 locations. That means that you can nest about ten FOR/NEXT loops, or you can go about 24 subroutines deep. Try to go further and you'll get an ?OUTOFMEMORY error, which is puzzling to beginners since there seems to be lots of memory left.

Most programmers get ?OUTOFMEMORY because of sloppy subroutine handling. Within a subroutine, they forget to RETURN and instead leap directly back into the main code with a GOTO statement. The used part of the stack never gets restored, and eventually the program runs out of space. The fastest demonstration of this is the one-line program, 100 GOSUB 100, which will bomb faster than you can say EBCDIC.

The C128

The 128 has a much bigger stack - over 500 bytes - and it's reserved purely for Basic activity. This stack logs not only FOR (eighteen bytes per item) and GOSUB (five bytes per item) but also the new command LOOP (five bytes per item). Even though this stack is much bigger, you can still fill it up quickly with foolish programming.

The new stack is implemented in software, not hardware. It's located in bank 0 in the area from hexadecimal 09FE down to 0800. There's a pointer stored at address 125 and 126 (hex 7D and 7E) which indicates the last location in use. The stack fills from the top down; if the stack is empty, the pointer will say $09FF, if it's nearly full the pointer will have dropped to the low 800's.

Here's a detailed rundown of what goes on the Basic stack for each type of entry:

FOR - hex $81 to indicate a FOR entry;

loop variable address	(two bytes);
increment floating value	(five bytes);
increment sign, 01 or FF	(one byte)
variable limit, floating	(five bytes);
line number	(two bytes);
return address	(two bytes).

GOSUB - hex $81) to indicate a GOSUB entry;	
line number	(two bytes);
return address	(two bytes).

DO - hex $EB to indicate a DO entry;	
line number	(two bytes);
return address	(two bytes).

Mechanics

It's useful to get a feeling for the workings of these. When a Basic program executes a FOR, GOSUB, or DO, the appropriate entry is placed on the Basic stack. Extras: FOR searches to see if the stack contains a previous FOR entry with the same variable name; if so, it strips the stack back to that point and then makes the entry... but it can only look through FOR entries. Example: a sequence such as FORD... GOSUB.. . FORJ would work badly; the new FORJ wouldn't catch the previous FORJ that had been started outside the subroutine.

DO may also have an extra: If WHILE or UNTIL is part of this statement, Basic will check and if necessary skip ahead to the appropriate LOOP statement. Note that although a FOR/NEXT must be executed at least once, a DO/LOOP may have its contents skipped entirely - a straight hop from the DO to the LOOP with everything in between ignored.

Now we come to the other end of these constructs. A NEXT will cause all earlier FOR entries to be searched for a matching variable name, although the computer will not search across a GOSUB or DO entry. When the right entry is found, the variable is "stepped" and tested for within range. If it's in range, Basic goes back; if not, the stack entry is scrapped and the program proceeds.

A RETURN will cause a scan of the entire stack... the most recent GOSUB entry found will be honoured and the stack stripped back to that point. Note that if FOR loops or DO structures had been opened within the subroutine, they will be scrapped upon RETURN.

A LOOP action depends on whether or not the command is followed by a WHILE or UNTIL. Assuming the LOOP statement is found to be active, it will scan the entire stack. The most recent DO entry found will be honoured and the stack stripped back to that point. Note that if FOR loops or subroutines had been opened within the DO structure, they will be scrapped upon LOOP.

Sample Program

Let's write a C128 program to allow names to be entered, sorted and listed. We'll look at the stack, commenting on some of the workings. Here's the program:

PROGRAM: LOOPER

```
100 DIM N$(100)
110 DO
120 PRINT
130 PRINT " 1 -ENTER NAMES"
140 PRINT "2 - LIST NAMES"
150 PRINT "3 - QUIT"
160 PRINT
170 INPUT " YOUR CHOICE ";C
180 ON C GOSUB 200,300,400
190 GOTO 170
200 INPUT "NAME ";X$
210J=N
220 DO WHILE J>0
230     K=J-1
240     IF X$>N$(K) THEN EXIT
250     N$(J) = N$(K)
260     J = K
270 LOOP
280 N=N+1
290 N$(J) = X$
300 FOR J =0T0  N-1
310     PRINT N$(J)
320 NEXT J
330 RETURN
400 END
500 LOOP
```

Enter the program; you might like to play with it, entering names and seeing them come out in alphabetic order. In a moment, we'll change it in order to allow ourselves to look around.

You should know that the array N$(..) uses element zero as the first item. Thus, if N (the number of items) equals 3 the array goes from N$(0) to N$(2).

The DO ... LOOP that extends from lines 220 to 270 is different from the comparable FOR ... NEXT, and usefully so. Here's why. As we put each item into the table, we compare it with the existing items. But the first time through, there's nothing in the table to compare with. If I coded

```
FOR J = N-1 TO 0 STEP -1
```

. . . I'd be stuck on the first item (where N equals 0), since a FOR/NEXT insists on exercising its contents at least once. No such problem with the DO/LOOP, which skips over the intervening code when the first item is encountered.

We're about to look into the mechanics a little. If you don't like tinkering with the works, or if you feel threatened by hexadecimal numbers, you might prefer to skip this section.

We want to analyze the stack, just to show that it's there and can be viewed whenever you like. Let's put a STOP command in at line 305:

```
305 STOP
```

RUN the program once again; call option 1 and enter any name. When the program stops, command MONITOR.

Now we'll examine the Basic stack pointer. Command M7D7E and look at the line that results. The first two bytes should be E3 09, meaning that the stack pointer is down to 09E3. Good - we'll look at the stack with command M9E39FF.

Let's review what we know of the program's "state". The first thing that happened is that line 110 executed a DO. When we selected option 1, line 180 performed a GOSUB. Line 220 performed another DO, but the LOOP at line 270 cancelled it. The next thing that happened, just before the STOP at 305, was that a FOR loop was opened by line 300.

So we have a FOR entry inside a GOSUB entry inside a DO entry. Let's see if we can find them on the stack.

Address 9E3 contains a value of 81 - that's the FOR flag. You can see that the variable is at address 0410 (variables are kept in Bank 1). Floating point numbers are hard to read, even if you know the secret, but beyond them we can see 2C 01 for line number 012C - that's hex for 300.

Eighteen locations along, at 9F5, we see the 8D flag signalling a GOSUB entry. Again, we could read the line number entry as hex 00B4 or 180. Five more locations along, at 9FA, we see EB for the LOOP item, with line number 6E for line 110.

It's all there, and you can look at it any time you get fuddled over loops. Now return to Basic with command X, and then delete line 305.

The Untangler

Some programmers have a knack for getting themselves tangled up in code. I get calls that sound like this: "I'm seven subroutines deep, and now I've found a situation where 1 want to give up and return to the menu. How do I get out?"

The proper answer is: you should never have gotten yourself into that mess. Start over, use variable flags to give return signals, and next time do it right.

In the past, I've taken pity on some of these unfortunates by digging out a SYS or suggesting a brief machine language routine to set things to rights. At least these people have learned (sometimes the hard way) that you can't just go back to the menu with a GOTO or you'll leave a messy stack and eventually get an OUTOFMEMORY message.

But with the 128, there's an easier way out. If you have only one DO/LOOP active, you can clean out all subroutine and FOR/NEXT entries just by going to the LOOP.

Examine the sample program. Note that a DO appears very early in the coding, and that the corresponding LOOP can never be reached. Here's the trick: if we ever do get to that LOOP - assuming no other DO's are open - we'll immediately be transported back to the menu and the stack will be neatly trimmed. It's a little like the magic word FROBOZZ that transports you back to the vault.

Here's a simple example. Suppose in the above program, we've picked option l and are asked to enter a name. At this point, we say something like, "Gosh ... 1 don't really want to enter a name after all". It would be nice to abort back to the menu.

Suppose we say, "OK, if the user types an asterisk character instead of a name we'll go back to the menu". We might try a new line at 205 which says something like:

```
IF X$ = " * " GOTO 120
```

But if we do so, we'll eventually have problems, since we have never cleared away the subroutine call from line 180.

The solution is simple. Enter line 205 as:

```
205 IF X$= " * " GOTO 500
```

When an asterisk is entered, we'll go to the LOOP statement at line 500. This will search for the last corresponding DO, finding it at line 110. The Basic stack will be trimmed back to that point, in this case removing the subroutine entry, and the program will resume by printing the menu.

It's a simple example, and doesn't trim much from the stack. But once you understand the principle, you can use it more generally.