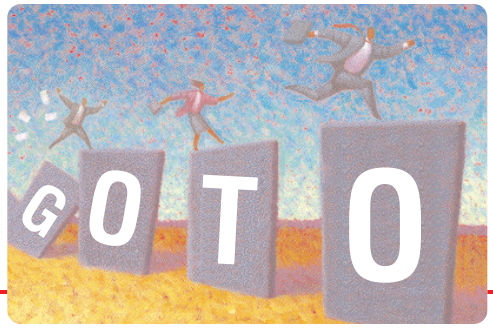

Leaving GOTO Behind

by Thomas E. Kurtz
Co-inventor of BASIC

Line numbers were part and parcel of the first BASIC developed at Dartmouth in 1964. They were used for two purposes: as targets of GOTO or GOSUB statements, and for creating and making changes to programs. We had no WYSIWYG screen editors in those days, and had to use teletypewriters. To change a line, one had to type the entire line, with changes, but with the same line number.

Over the next ten or so years, two things happed. First, screen editors came into being, thus allowing one to change a part of a line without retyping the entire line. Second, we learned about the perils of undisciplined use of GOTO statements and structured programming was invented. This was about the time the term "spaghetti code" came about to describe exceptionally bad examples of GOTO statements. Around 1975 or so we ceased using line numbers at Dartmouth, and what a relief.

Other personal computer versions of BASIC made line numbers optional, but retained their use as targets of GOTO statements.

The purpose of this report is to show how easy it is, most of the time, to toss line numbers into the dust bin. There are two steps. The first is to eliminate the use of GOTO and GOSUB statements. When this has been done, the second step is to remove the line numbers with a simple utility program (such as DO UNNUM,) since line numbers no longer appear in the program proper.

Although only simple two-way choice structures and general loop structures are needed for completely structured programming, we shall consider several types of choice structures, several types of loop structures, and subroutines.

Simple Choice Structures

```
100 IF x >= 3 THEN 200
110 LET result$ = "less"
200 ...
```

can be handled as

```
IF x < 3 then LET result$ = "less"
```

If there are two actions, as in

```
100 IF x >= 3 THEN 130
110 LET result$ = "less"
120 GO TO 140
130 LET result$ = "more"
140 ...
```

translates to

```
IF x < 3 then
   LET result$ = "less"
ELSE
   LET result$ = "more"
END IF
```

This has one big advantage – beside getting rid of the GOTO statements and the line numbers – the action for the "true" condition immediately follows it. That is, what happens when "x < 3" takes place in the next line!

We learned early on that even using GOTO statements and line numbers, careful construction of choice structures allowed automatic indenting to reveal the structure, as with

```
100 IF x >= 3 THEN 130
110    LET result$ = "less"
120    GO TO 140
130 LET result$ = "more"
140 ...
```

Structured choice statements allow indentation trivially!

Multiple Choice Structures

The old BASIC **ON** statement allowed multiple branching.

```
100 ON case GOTO 200, 300, 400
200 REM Here for the first case
210 GOTO 500
300 REM Here for the second case
310 GOTO 500
400 REM Here for the third case
410 GOTO 500
500 ...
```

can be written as

```
SELECT CASE case
CASE 1
   REM Here for the first case
CASE 2
   REM Here for the second case
CASE 3
   REM Here for the third case
END SELECT
...
```

Loop Structures

The following is common:

```
100 REM Start of a loop
110 REM More of the loop
200 REM End of the loop
210 IF case <= 10 then 100
220 ...
```

can be expressed as

```
DO
   REM Start of a loop
   REM More of the loop
   REM End of the loop
LOOP while case <= 10
```

If you need to exit from the middle of the loop, as in

```
100 REM Start of a loop
110 REM More of the loop
150 IF case > 10 then 220
200 REM End of the loop
210 GOTO 100
220 ...
```

you can use

```
DO
   REM Start of a loop
   REM More of the loop
   IF case > 10 then EXIT DO
   REM End of the loop
LOOP
```

We need not discuss the FOR-NEXT loop as that was the one structured programming construct in the original BASIC.

Subroutines

The old original subroutine structure

```
200 GOSUB 500
210 ...
500 REM Start of the subroutine
510 ...
520 RETURN
```

can be expressed using the CALL and SUB statements as

```
CALL TheSubroutine
...
SUB TheSubroutine
...
END SUB
```

Beside getting rid of the GOSUB statement and the line numbers, there are two other advantages: first, the subroutine is "invoked" by name rather than by line number, and it is now possible to add parameters:

```
CALL TheSubroutine (3, x)
...
CALL TheSubroutine (4, y)
...
SUB TheSubroutine (a, b)
...
END SUB
```

The subroutine can be invoked from more than one place without having to use LET statements to give values to the arguments. Using GOSUB statements, one would have had to to write

```
180 LET a = 3
190 LET b = x
200 GOSUB 500
205 LET x = b                    ! This is the result
210 ...
280 LET a = 4
290 LET b = y
300 GOSUB 500
310 LET y = b                    ! This is the result
320 ...
500 REM Start of the subroutine
510 ...
520 RETURN
```

More Complicated Cases

More complicated cases arise, a simple example of which might be:

```
100 IF x < y then 200
110 IF x = y then 300
120 REM ComputationA
130 GOTO 400
200 REM ComputationB
300 REM ComputationC
400 REM Completion
```

If you try to turn this into a combination of two simple choice structures, you cannot avoid duplicating the code in Computerion C, which is common to two different possibilities.

```
IF x < 100 then
   REM ComputationB
   REM ComputationC
ELSE if x = 100 then
   REM ComputationC
ELSE
   REM ComputationA
END IF
REM Completion
```

Avoiding duplication of code was a virture in the early days of extremely small memories, but it is not longer a virtue today. If Computation C is just one or two lines, just duplicate them. If Computation C has many lines, turn it into a subroutine

```
IF x < 100 then
   REM ComputationB
   CALL ComputationC
ELSE if x = 100 then
   CALL ComputationC
ELSE
   REM ComputationA
END IF
REM Completion
...
SUB ComputationC
...
END SUB
```

Incidentally the type of code illustrated just above is the most complicated problem I have ever run into converting thousands of lines of GOTO-code (whether in BASIC or in Fortran) into structured code.

Thomas E. Kurtz, May 19, 1999

Comments or questions to: tom@truebasic.com