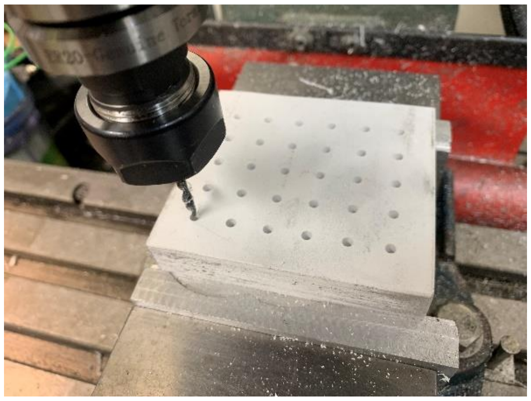
Using Variables, Computation, GOTO, and IF-THEN-ELSE G-code Commands, Version 2.2

By R. G. Sparber

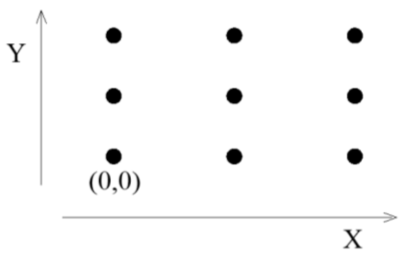


Different tasks require different tools. This is why a well-equipped shop maintains a wide variety of tooling. The same goes for our knowledge of G-code. Often the automatically generated G-code gets the job done just fine. But once in a while, some manual adjustments must be made. And then there are cases where the best solution requires pure hand coding.

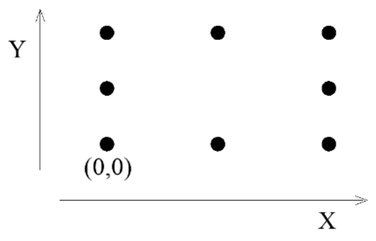
A powerful category of "tooling", when hand coding, involves logical control commands. This article discusses the use variable, computation, GOTO, and IF-THEN-ELSE commands along with the concept of user-defined variables. An example is presented that uses these commands to drill an array of holes.

My Centroid G-code information was taken from the M-Series Operator's Manual dated 9/14/16 chapters 11, 12, and 13. The intent is not to duplicate the entire manual, only to pick out and expand on the immediately relevant bits.

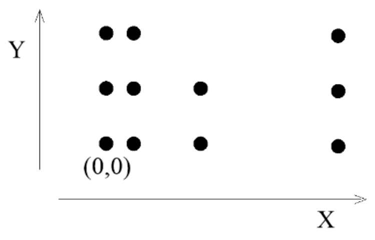
[1] This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



The example presented here drills a 3 by 3 array of holes with an X-axis spacing of 1.000 inch and Y-axis spacing of 0.500 inches. These values are easily changed. Not very exciting.



But I can add a single line of code that will selectively prevent any hole from being drilled. For example, I can skip drilling the hole at X = 1.000 Y = 0.500.

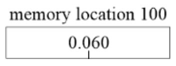


Given this total freedom to place holes, I could increment the spacing of the holes along the X-axis exponentially. Notice how the X-axis spacing increases as we move along this axis. I have kept the Y-axis spacing constant although I did eliminate one hole in the top row.

Given the computational power of G-code plus the logical tests presented here, you are only limited by your imagination. Sure this can be done with CAD and a G-code generator but it is so much faster and more flexible with hand coding.

G-code Elements I Will Use

Centroid gives me 50 user definable memory locations, 100 through 149. A "#" before the number identifies it as one of these locations. I can assign the number 0.060 to location 100 by writing

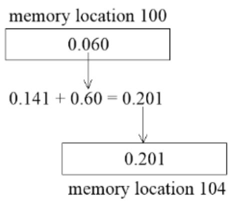


#100 = 0.060

Then I can use #100 like any other number. For example

#104 = 0.141 + #100

says to take 0.141 and add it to the value stored in memory location 100. Place the result into memory location 104.



#104 = 0.141 + #100
= 0.141 + 0.060
#104 = 0.201

I can embed arithmetic in commands by using [*square brackets*]. For example

Z [0.141 + #100]

says to take 0.141 and add it to the contents of memory location 100. Use the result to set the Z position.

Z [0.141 + #100]
Z [0.141 + 0.060]
Z [0.201]

which is the same as saying Z 0.201. The cutter will move along the Z axis to 0.201 inches above our Z = 0.000 point.

We also get a few control logic commands.

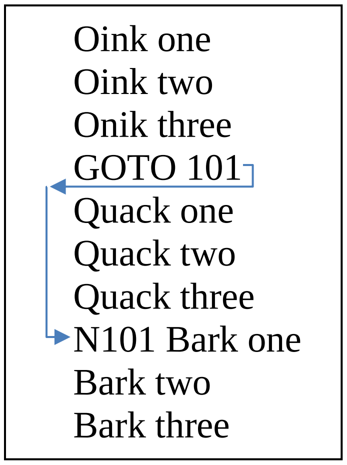
By using GOTO, we can jump to any defined line number. For example,

```
GOTO 101
```

When the program reaches this line of G-code, it will jump to line 101 which is identified by having "N101" at the beginning of the line.

```
N101 G1 F1.0 Z 0.100
```

Line numbers are optional. If you are not going to jump to a given line, no need to number it.



```
Oink one
Oink two
Onik three
GOTO 101
Quack one
Quack two
Quack three
N101 Bark one
Bark two
Bark three
```

For example, we would execute Oink one, two, and three. Then GOTO 101 would tells the interpreter to jump to line 101. Quack one, two, and three would be skipped. Bark one would be the next line of G-code executed. Two and three would follow.

Line numbers can also be in any order. So line number 200 can come before line number 100. They are just the way the interpreter identifies unique lines of code.

Often we need to jump to different lines depending on current conditions. That is where IF-THEN-ELSE comes in. The command format is

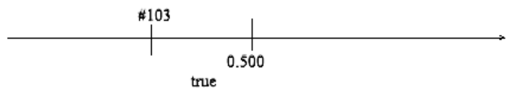
IF <expression> THEN <execute if true> ELSE <execute if false>

An "expression" is something that crunches on one or more values and produces a true or false result. So, when I write

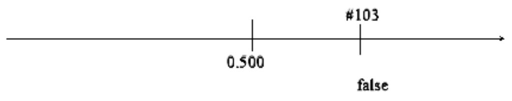
[#103 LE 0.500]

The square brackets tell the interpreter that logical or computational work must be done.

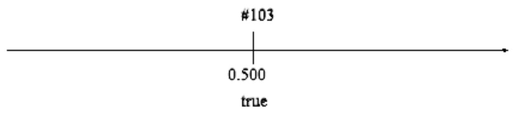
I am telling the machine to compare the contents of memory location 103 with the value 0.500. Those two letters tell the machine how to do the comparison.
LE means Less than or Equal to. So when the contents of memory location 103 is less than or equal to 0.500, this bit of code generates a true result. Otherwise, out pops "false".



The contents of memory location 103 is to the left of 0.500. This means it is less than 0.500 so [#103 LE 0.500] will evaluate to being true.



In this next example, the contents of memory location 103 is to the right of 0.500. This means it is greater than 0.500. [#103 LE 0.500] will evaluate to being false.

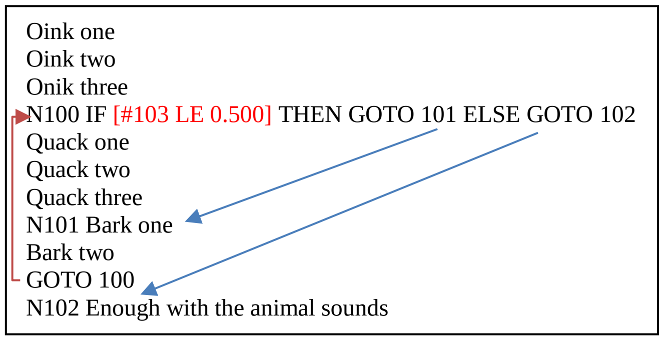


Our final example is when the contents of memory location 103 equals 0.500. Recall that the "E" in LE means "equal to". So when the contents of memory location 103 equals 0.500 the logic evaluates to true.

I can write

N100 IF [#103 LE 0.500] THEN GOTO 101 ELSE GOTO 102

Lots going on here. First off, this is line number 100 because it starts with “N100”. Then I have my IF-THEN-ELSE command. The <expression> is [#103 LE 0.500] which will be either true or false depending on the number stored in memory location 103. If true, we GOTO line 101. Otherwise, it is off to line 102.

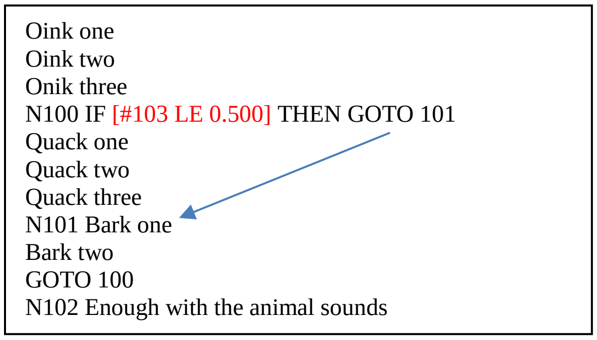


```
Oink one
Oink two
Onik three
N100 IF [#103 LE 0.500] THEN GOTO 101 ELSE GOTO 102
Quack one
Quack two
Quack three
N101 Bark one
Bark two
GOTO 100
N102 Enough with the animal sounds
```

We execute Oink one, two, and three before reaching line 100. Depending on the value stored in memory location 103, we either jump to line 101 or line 102.

If [#103 LE 0.500] evaluates to true, the jump is to line 101. There, Bark one and two execute before we jump to line 100.

If [#103 LE 0.500] evaluates to false, we jump to line 102. Notice that Quack one, two, and three never execute.



```
Oink one
Oink two
Onik three
N100 IF [#103 LE 0.500] THEN GOTO 101
Quack one
Quack two
Quack three
N101 Bark one
Bark two
GOTO 100
N102 Enough with the animal sounds
```

The ELSE <execute if false> part is optional. Without ELSE GOTO 102, a false result will drop us down to the next line and Quack one, two, three, Bark one, Bark two are executed before we go back to line 100. Note that line 102 is never executed.

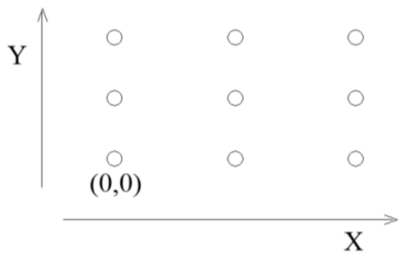
The last command I will introduce is both of supreme importance and essentially not there. It is the semicolon. This character tells the G-code interpreter to ignore what is coming next. It is of supreme importance because it lets me add comments as needed to explain what is going on to other humans. It is essentially not there because the G-code interpreter ignores the text and the machine sees nothing.

```
G0 Z 0.100; RETURN TO THE SAFE PLANE
```

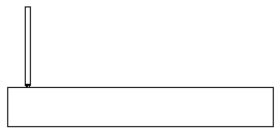
These comments will be painfully obvious while you write the G-code and invaluable when you have been away from it for a few months. Others will depend on these comments too as they try to understand what was written.

Let's Do Something Useful

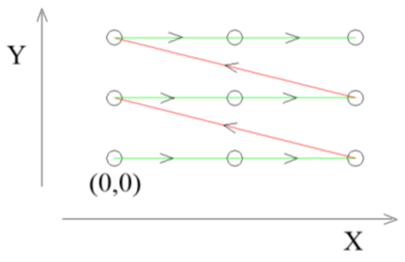
So much for theory. Time to write a G-code program that is useful: drill an array of holes.



The first hole will be in the lower left corner. When viewed on the mill table, this is left front. This hole is defined as the origin which means X = 0.000 and Y = 0.000.



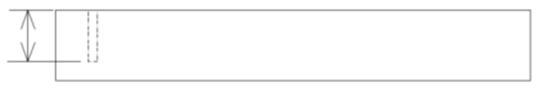
I will also lower the drill until it just touches the surface and then set Z = 0.000.



The program will drill a hole at the origin, then move along the X-axis drilling holes until it completes the row of 3 holes. Then it will move back to X = 0.000 while moving up to the next row along the Y-axis. Again, holes will be drilled along the X-axis. This pattern will repeat until all holes have been drilled.

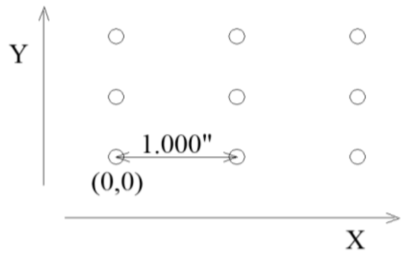
The Actual Code, Illustrated

;Drill an Array of Holes
`G20 G90 G64 G40`; sets up machine
;define parameters



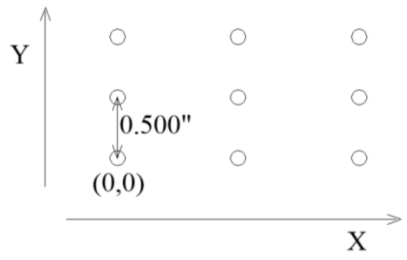
`#100 = 0.100`; depth of each hole
Memory location 100 contains the depth of each hole, 0.100 inches.

Side view.

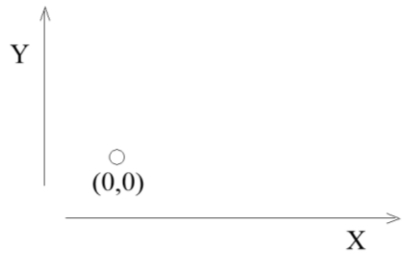


`#101 = 1.000`; X increment
1.000 inches is the distance between holes along the X-axis.

Top view.

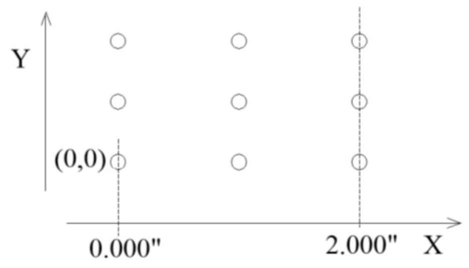


`#102 = 0.500`; Y increment
0.500 inches is the distance between holes along the Y-axis.

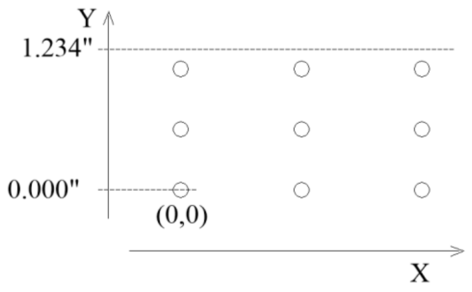


`#103 = 0.000`; X position initialized to the origin
Current position along X axis.

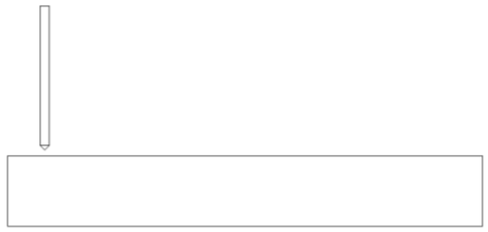
`#104 = 0.000`; Y position initialized to the origin
Current position along Y axis.



`#105 = 2.000`; distance limit along the X axis

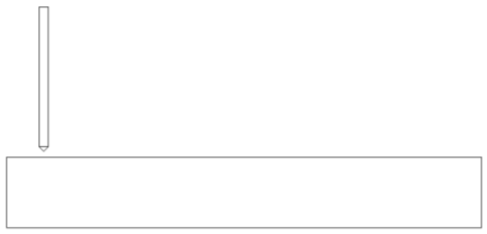


`#106 = 1.234`; distance limit along the Y axis
Note that the distance limit can be beyond the last hole in the column or at the last hole.

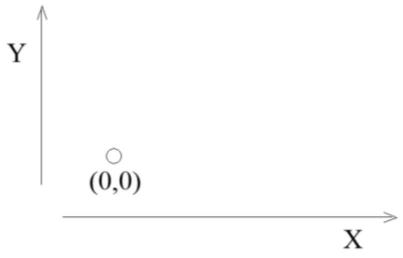


`#107 = 0.050`; safe plane
Side view.

`#108 = 5.0`; drilling feed rate
;end of parameters
Now we start moving the cutter around.



`G0 Z #107`; go to the safe plane

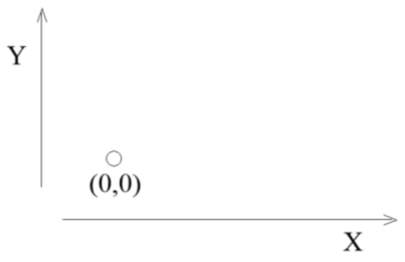


`G0 X0.000 Y0.000`; then go to the first hole at front left corner which has been defined as the origin. Z0.000 is at the surface of the stock

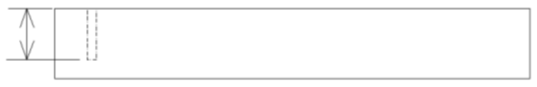
;drilling holes and moving The start of the logic.

`N100 IF [#103 LE #105] THEN GOTO 101 ELSE GOTO 102`
; test X position
;if at end of line along X-axis, start next row
Our current position along the X-axis is stored in memory location #103 is 0.000. We compare it to our X-axis limit which is stored in memory location #105. It is 2.000. When we check if "0.000 is Less than or Equal to 2.000", the answer is true. This directs us to the first GOTO and we jump to line 101.

`N101 G0 X #103 Y #104`; move to new hole location

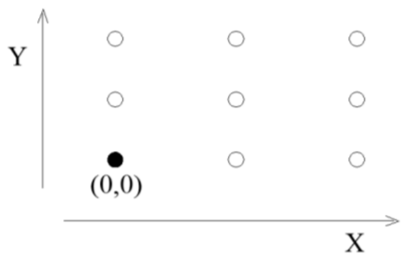


Our current position along the X-axis is 0.000. Along the Y-axis the current position is stored in memory location #104 and has the value 0.000 too. Therefore, this line is interpreted as X0.000 Y0.000. That G0 says to move to these coordinates at maximum speed. Since we were already at (0,0), nothing moves this time around.



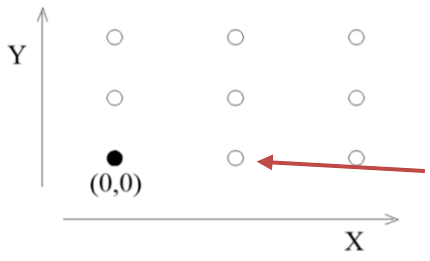
`G1 F #108 Z [-#100]`; drill down through the stock to a depth found in memory location 100 and at the feed rate found in memory location 108
Memory location #108 contains 5.0 so we will drill down at 5.0 inches per minute. Memory location #100 contains 0.100. The square brackets tell the interpreter there is a calculation needed. I am taking the contents of #100 and multiplying it by -1. The result is Z -0.100. Recall that I set Z 0.000 at the surface of the stock. So -0.100 inches is below the surface. The entire line is saying to drill down at 5.0 inches per minute to a depth of 0.100 inches.

`G0 Z #107`; return to the safe plane



Our first hole has been drilled as indicated by the black dot.

`#103 = #103 + #101`; increment X axis pointer
#103 contains the value 0.000. #101 is our increment along the X-axis and is 1.000. This line is saying to take 0.000, add 1.000 and then save the result in #103. In other words, advance the pointer for the X-axis to the next hole. We are drilling the first row of holes so the Y-axis value does not change yet.



To recap, we drilled our first hole at (0,0) and then updated the X-axis pointer to our next hole which will be at
X = 1.000 and Y = 0.000.

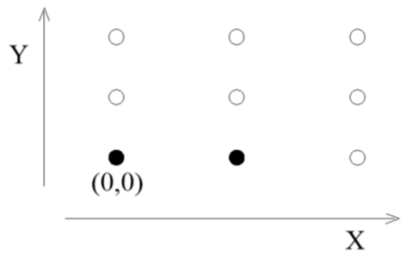
`GOTO 100`; return to test of X-axis pointer

`N100 IF [#103 LE #105] THEN GOTO` **`101`** `ELSE GOTO 102`
; test X position
;if at end of line along X-axis, start next row
This time #103 contains 1.000 so we are asking if 1.000 is Less than or Equal to 2.000. Since it is true, we again jump to line 101.

`N`**`101`** `G0 X #103 Y #104`; move to new hole location
Our X-axis pointer equals 1.000. Our Y-axis pointer is still 0.000. Therefore, this line is interpreted as X1.000 Y0.000. We move to these coordinates at maximum speed.

`G1 F #108 Z [-#100]`; drill down through the stock to a depth found in memory location 100 and at the feed rate found in memory location 108
As on the first hole, we drill down at 5.0 inches per minute to a depth of 0.100 inches.

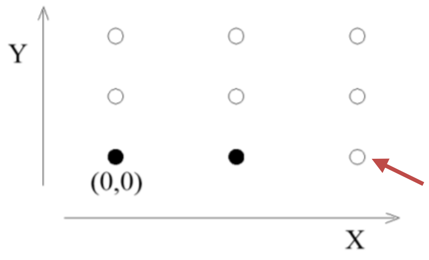
`G0 Z #107`; return to the safe plane



Our second hole is now complete.

`#103 = #103 + #101`; increment X axis pointer

#103 contains the value 1.000. #101 is our increment along the X-axis and is 1.000. This line is saying to take 1.000, add 1.000 and then save the result in #103.



In other words, advance the pointer for the X-axis to the next hole position at 2.000. We are drilling the first row of holes so the Y-axis value has not changed yet.

`GOTO 100`; return to test of X position

`N100 IF [#103 LE #105] THEN GOTO` **`101`** `ELSE GOTO 102`

; test X pointer

;if at end of line along X-axis, start next row

This time #103 contains 2.000 so we are asking if 2.000 is Less than or ***Equal*** to 2.000. Since it is true, we again jump to line 101.

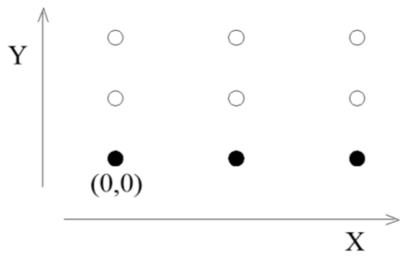
`N`**`101`** `G0 X #103 Y #104`; move to new hole location

Our current position along the X-axis is 2.000. Along the Y-axis the current position is 0.000. Therefore, this line is interpreted as X2.000 Y0.000. We move to these coordinates at maximum speed.

`G1 F #108 Z [-#100]`; drill down through the stock to a depth found in memory location 100 and at the feed rate found in memory location 108

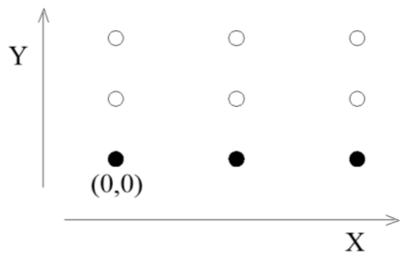
We drill down at 5.0 inches per minute to a depth of 0.100 inches.

`G0 Z #107`; return to the safe plane



All 3 of our holes in the first row have now been drilled.

`#103 = #103 + #101`; increment X axis pointer



#103 contains the value 2.000. #101 is our increment along the X-axis and is 1.000. This line is saying to take 2.000, add 1.000 and then save the result in #103. In other words, advance the pointer for the X-axis to the next hole at 3.000. But wait, there is no hole needed at X = 3.000!

`GOTO 100`; return to test of X pointer

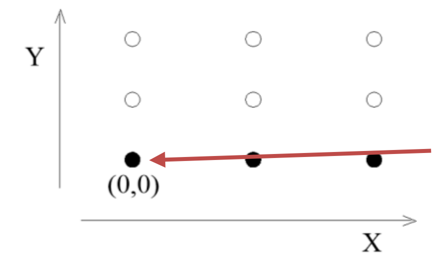
`N100 IF [#103 LE #105] THEN GOTO 101 ELSE GOTO` **`102`**

; test X pointer

;if at end of line along X-axis, start next row

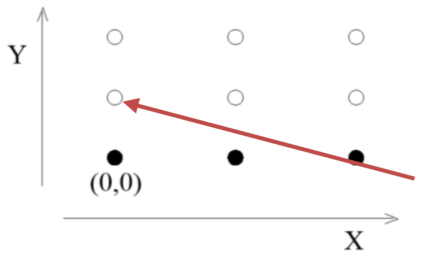
This time #103 contains 3.000 so we are asking if 3.000 is Less than or Equal to 2.000. Since it is false, we jump, for the first time, to line 102. No hole will be drilled at X = 3.000. Instead, we detected that the end of the row has been reached.

;if at end of line along X-axis, start next line



`N102 #103 = 0.000`; reset X-axis pointer to zero
No movement yet.

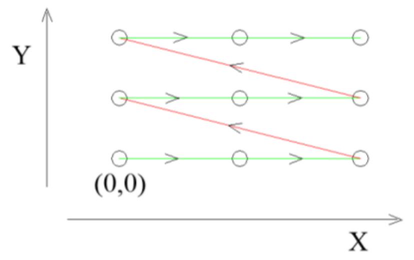
```
#104 = #104 + #102; increment Y-axis pointer
```



Our current Y-axis position is 0.000 which is stored in #104. Our Y-axis increment between holes is 0.500 and is stored in #102. We add 0.000 to 0.500 and store the result into #104. So #104 contains 0.500.
This advances our Y-axis pointer to the second row. No movement yet.

```
IF [#104 LE #106] THEN GOTO 100; if not done with rows,
continue sequence
```

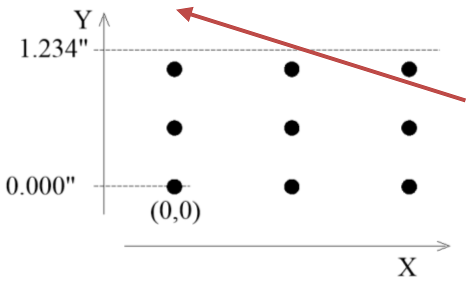
Before we drill a hole, the program checks to see if this next row will be within the specified Y-axis range. #104 is our current Y-axis pointer value and equals 0.500. #106 is our Y-axis limit which is set to 1.234. The code is testing if 0.500 is Less than or Equal to 1.234. Since it is true, we go to line 100.



Recall that line 100 is the start of the code that drills a row of holes. When the last hole has been drilled, the X-axis pointer value is incremented to a value larger than the X-axis limit. This causes the program to advance the Y-axis pointer value to the next row while it resets the X-axis pointer value to 0.000. Then the cycle repeats.

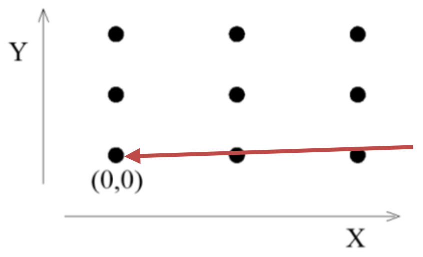
After the last row has been drilled, our Y-axis pointer is advanced to 1.500. Then we return to our Y-axis test:

```
IF [#104 LE #106] THEN GOTO 100; if not done with rows,
                                  continue sequence
```



This time, we are testing if 1.500 is Less than or Equal to 1.234. Since it is not, we drop down to the next line of code rather than going to line 100.

```
G0 X0.000 Y0.000; otherwise, done so
return to origin
```



I chose to return the cutter to the origin.

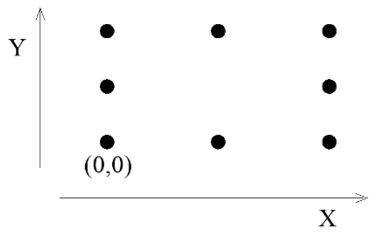
`M00; stop for operator` Done!

Just the Code

On my mill, spindle speed is manually controlled so I have not included related commands in this code.

```
;Drill an Array of Holes
G20 G90 G64 G40; sets up machine
;define parameters
#100 = 0.100; depth of each hole
#101 = 1.000; X-axis increment
#102 = 0.500; Y-axis increment
#103 = 0.000; X-axis pointer initialized to the origin
#104 = 0.000; Y-axis pointer initialized to the origin
#105 = 2.000; distance limit along the X-axis
#106 = 1.234; distance limit along the Y-axis
#107 = 0.050; safe plane
#108 = 5.0; drilling feed rate
;end of parameters
G0 Z #107; go to the safe plane
G0 X0.000 Y0.000; then go to the first hole at front left corner
     which has been defined as the origin. Z0.000 is at the
     surface of the stock
;drilling holes and moving
N100 IF [#103 LE #105] THEN GOTO 101 ELSE GOTO 102;test X-axis
     pointer
;if at end of line along X-axis, start next row
N101 G0 X #103 Y #104; move to new hole location
G1 F #108 Z [-#100]; drill down through the stock at the
     specified feed rate
G0 Z #107; return to the safe plane
#103 = #103 + #101; increment X-axis pointer
GOTO 100; return to test of X-axis pointer
;if at end of line along X-axis, start next line
N102 #103 = 0.000; reset X-axis pointer to zero
#104 = #104 + #102; increment Y-axis pointer
IF [#104 LE #106] THEN GOTO 100; if not done with rows, continue
     sequence
G0 X0.000 Y0.000; otherwise, done so return to origin
M00; stop for operator
```

Removing a Single Hole

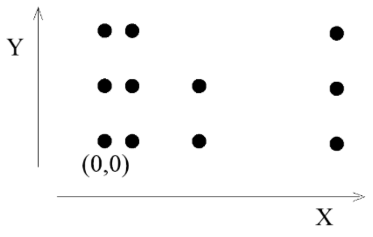


We can remove a single hole by using what we have learned so far. Our IF-THEN-ELSE statement can identify the X-axis and Y-axis pointer value pair that we want to skip.

```
IF [#103 EQ 1.000] THEN GOTO 300; see if X-axis pointer is
     1.000. If so, check Y-axis pointer value
GOTO…; continue normal program
N300 IF [#104 EQ 0.500] THEN GOTO 301; see if Y-axis
     pointer is 0.500. If so, skip drilling this hole but do
     increment pointer(s)
GOTO…; continue normal program
```

I'll leave it to the reader to weave this bit of logic into the program. You will need to label an existing line N301.

Exponential Hole Spacing



We have a line of code that deal with the X-axis pointer:

```
#103 = #103 + #101; increment X-axis pointer
```

It takes the constant stored in #101 and adds it to the current X-axis pointer value to create the next pointer value.

I could write

```
#103 = #103 * #101;
```

but there is a hitch. If the first X-axis pointer value is 0.000, then we will be stuck there since any number multiplied by 0 equals zero. So let's change the program so the first hole is at (1.000,0.000). This moves the origin to the left by 1.000 but has no effect on the relative position of the array.

Our first X-axis pointer value will be 1.000. No reason to bother the first Y-axis pointer so I left it at 0.

With #103 starting off at 1.000 and I will set #101 equal to 1.500, then the X-axis pointer values will be

1.000 the initial value
1.000 * 1.500 = 1.500
1.500 * 1.500 = 2.250
2.250 * 1.500 = 3.375

Looking at the spacing of these holes, we have

1.500 – 1.000 = 0.500 between the first and second holes
2.250 – 1.500 = 0.750 between the second and third holes,
3.375 – 2.250 = 1.125 between the third and fourth holes.

The hole spacing is going up exponentially.

I'll leave it to the reader to weave this bit of logic into the program.

Acknowledgments

Thanks to Karl Harnish for his many suggestions to improve clarity.
Thanks to Tony Foale for suggesting advanced examples at the start of the article.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with just "Subscribe" in the subject line. If you are on this list and have had enough, email me "Unsubscribe" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org