Conditional G Code “best practice”

This tutorial should be read in conjunction with the information provided by the Duet 3D wiki and in particular the sections on

G Code https://docs.duet3d.com/User_manual/Reference/Gcodes

Reprap Firmware Object Model https://docs.duet3d.com/en/User_manual/RepRapFirmware/Object_Model

G-Code Meta Commands https://docs.duet3d.com/User_manual/Reference/Gcode_meta_commands

Whilst the subject of this page is "best practice" when writing conditional G Code, the author does not claim that the contents are in fact "best practice"
Rather it is presented to demonstrate that there are many possible configurations spanning many types of applications. Therefore, if one is planning on creating conditional g-code (especially if those files are to be shared), then consideration should be given to making the code work as expected with minimum modification by as many people as possible.

G-Code is a well-known format designed to provide movement and other instructions to machines such as 3D printers, CNC cutting machines, milling machines etc.
In the realms of 3D printing it is typically created a "slicer" and transferred to the printer for execution on a line by line basis.
The machine will try to execute the command given regardless of circumstances or limitations.
This makes G-Code almost universally usable only on the machine for which it was created, or other machines which are configured in exactly the same way.
Likewise, there is little to no ability to "react" to variable circumstances during the execution of the code.

Conditional G-Code provides the facility to carry out different commands based on data retrieved from the firmware and configured hardware.
This data may be in the form of axis positions, temperatures, or any other information available through the RepRap Firmware object model.
The object model is a hierarchical structure which contains "objects" to which values are assigned by the firmware.
These objects may be queried during the execution of a G-Code file and the values returned can determine the path of the G-Code from that point.

This provides a very powerful tool which allows for not only the ability to carry out jobs which may not have been previously possible, but also to compartmentalize code into "macros" or "sub-programs" which can be executed (or not) based on the information gathered from the object model. Furthermore it allows the creation of G-Code which is portable across machines of varying configurations and sizes.

In order to take advantage of the power of conditional G-Code requires the user to follow certain programming rules, which include the use of correct syntax and structure.
Additionally, "best practice" requires the user to (wherever possible), not make assumptions about the configuration of the machine that the code will run on.
Instead, you should try to replace any hard-coded values with calculated values.
All code should also be heavily commented in order that anyone else who may look to use it can easily follow the intention of the author.

The "conduit" between the G-Code file and the object model are the G-Code Meta Commands.
These are essentially a subset of the C++ programming language which provides for code flow that is not necessarily sequential and allows the firmware to recognize and parse the returned data into values that would otherwise need to be hard coded.

The object model has a hierarchical structure which can be "drilled down" upon to access various data that may be common to different parts of the machine.
For example in a typical 3 axis machine, all axes have a "current position" which is unique to that axis.
However the "path" to that position can be represented in a common manner.

move.axes["n"].machinePosition , where "n" is the axis number starting from zero.
i.e X=0, Y=1, Z=2 etc
The square braces [] represent an array which allows the same structure to areas of the machine which have multiple objects. (Tools, extruders, GPIO pins etc)

The following is a list of suggested "best practice" methods which may help you and others make use of the power of the conditional G Code structure.
It is simply opinion, and is put forward mainly for those that plan on sharing code they have written, or on using shared code.

1. Wherever a piece of code may be re-used in different gcode files should be compartmentalized into macros.
   This not only allows the easy re-use of verified code (thus reducing mistakes), but also means that if a change needs to be made it only need be done in one place.
   All other files that use that code will then run the updated code when called.
   Macros can themselves call other macros, thus extending the re-use of code.

2. Wherever possible do not assume that the code will be run on a machine that is configured identically.
   For example when probing the bed you should base your positions on the bed sizes in the configuration rather than a specific X/Y position.
   Say we have a 300mm bed and we want to probe 25mm in from the edges in the X direction and in the center of the Y axis travel
   Whilst it may be convenient to simply use a fixed point such as G1 X275 Y175, what happens if the code is subsequently run on a printer with a smaller (200mm) bed?
   We could instead query the object model for the minimum and maximum travel points and base our position on that
   G1 X{move.axes[0].max - 25} Y{(move.axes[1].max - move.axes[1].min) / 2}
   The curly braces {} are required so that the firmware can differentiate the conditional code and parse it as if it were a simple hard coded value.
   BUT
   Whilst our conditional code will indeed move to a point 25mm inside our configured maximum travel, how do we know that the probe is over the bed at that point?
   The machine may be configured with enough area outside of the print area to change tools, perform nozzle wiping etc.
   Also some probe moves like G30 are allowed to place the probe outside the printable area.
   For the code to be "portable", we may need instead to rely on points which configuration defines as being able to be probed.
   G1 X{move.compensation.probeGrid.xMax - 25} for example.
   We may also need to compensate for the probe offset in both X & Y if our intent is to probe in the bed centre then we need to determine the size of the bed
   M208 tells us axis minima and maxima.
   We know the X0 & Y0 are going to be the minimum printable area, but the X/Y maxima may well be well outside the bed.

We know from from M557 what the min and max probe points are, and we know from G31 what the probe offsets are.
From that, we can reasonably assume that the bed maximimum can be defined by ***(ProbeMinimum - ProbeOffset + ProbeMaximum)***
So to move to the X & Y max bed points, we would use.

G1 X{move.compensation.probeGrid.xMin - sensors.probes[0].offsets[0] + move.compensation.probeGrid.xMax} ;X probe offset used

G1 Y{move.compensation.probeGrid.yMin - sensors.probes[0].offsets[1] + move.compensation.probeGrid.yMax} ;Y probe offset used

And it follows that the centre of the bed would be those points divided by two.
But if we want to actually place the probe over the centre of the bed we must subtract the probe offset
The math will work whether the offset is positive or negative because adding a negative is the same as subtracting a positive.

There is a limit to the number of characters that any single line of G Code can contain (160 Characters).

Therefore it may be necessary to break code down into multiple lines as above, or assign values to variables* to reduce the line length

****At the time of writing, user variables have not yet been implemented.***

var ProbeXpoint = {move.compensation.probeGrid.xMin - sensors.probes[0].offsets[0] + move.compensation.probeGrid.xMax} ; declare an X variable and assign a value to it

var ProbeYpoint = {move.compensation.probeGrid.yMin - sensors.probes[0].offsets[1] + move.compensation.probeGrid.yMax} ; declare a Y variable and assign a value to it

G30 X{ProbeXpoint} Y{ProbeYpoint}

Once a variable has been declared once, it may be re-used and assigned different values using the "set" function.

set ProbeXPoint = 120 ; set the X point to 120

set ProbeYpoint = 60 ; set the Y point to 60

G30 X{ProbeXpoint} Y{ProbeYpont} ; probe at X120 Y60

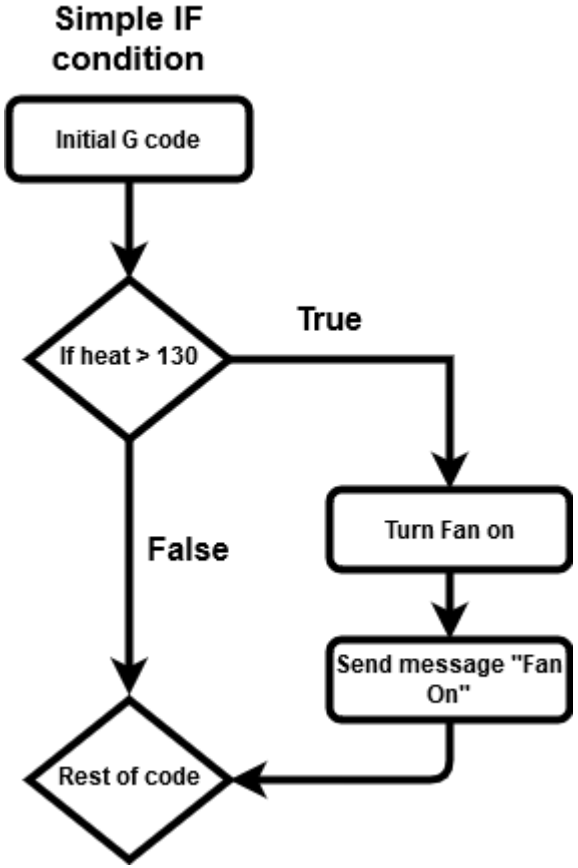
3. If the code you are writing is designed specifically for a certain kinematics type, it should include a mechanism to prevent it executing on another type of machine to avoid possible damage.
   This will avoid code applicable only to a Delta being run on a cartesian or CoreXY printer for example.

```
;check if the machine is a printer and abort if not
if state.machineMode != "FFF" ; != means “not equal to”
        abort "Job cancelled, machine is not printer" ; This line is only executed if printer does not equal FFF
; code after here will only be executed on FFF machines (Not CNC, Laser etc)
;now check if the machine is cartesian and abort if not
if move.kinematics.name != "cartesion" ; != means "not equal to"
        abort "Job canceled, machine is not cartesian"
;code after this point will only be executed on a cartesion printer
;run one piece of code if it's on a Delta and another for others.
if move.kinematics.name = "delta"
        G1 X0 Y0 ; move to centre of bed on delta
else
        G1 X{(move.axes[0].max - move.axes[0].min) /2} Y{(move.axes[1].max - move.axes[1].min) / 2} ; move to centre of bed on non deltas
;continue with normal code
G1 Z0.2 ; move to bed
```

4. At the heart of conditional Gcode is the ability to react to…. (drum roll) Conditions!
   IF, WHILE, ELSE, ELIF
   Code may be "nested" and contain several levels of conditional directions to take.
   The code executed based on the condition may carry out a task, or it may even abort the whole print.

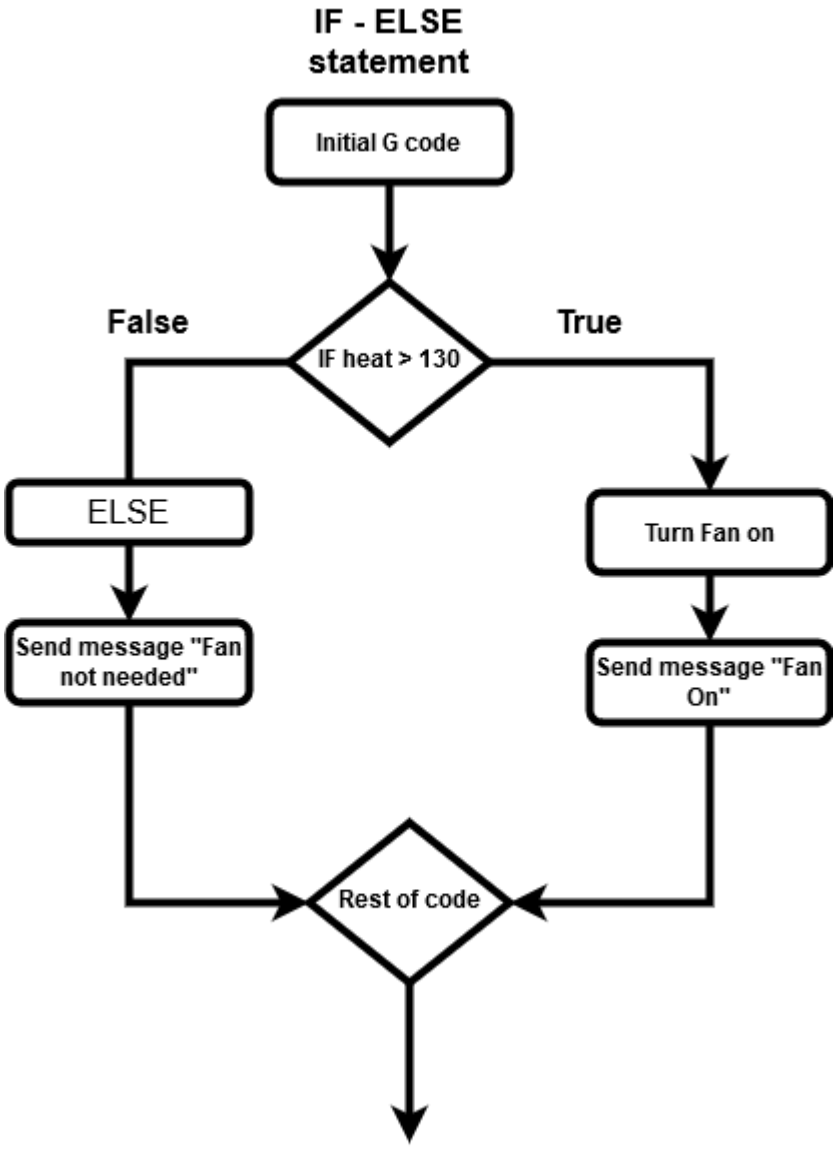
Simple IF statement.

If answer is TRUE do this… Otherwise carry on with rest of code as normal.



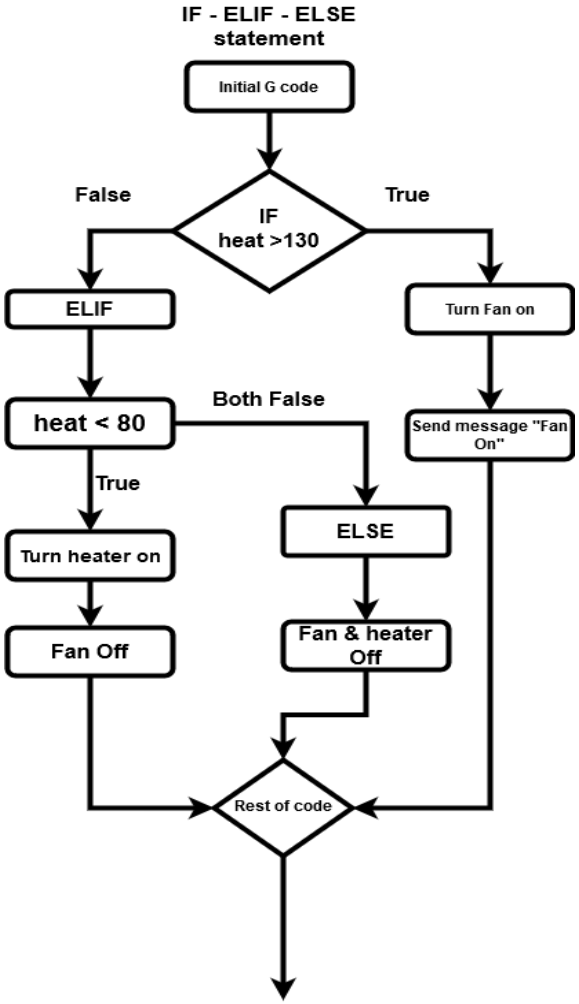
IF / ELSE statement

If answer TRUE, do this. If FALSE do that..
THEN carry on as normal.



IF / ELIF /ELSE statement

If the answer to question 1 is TRUE do this..
Otherwise ask question 2.
If answer to question 2 is TRUE do that..
Otherwise (both questions must be FALSE) do the other thing
THEN (If we haven't aborted)
Carry on as normal



The code that will be executed after a conditional construct such as IF, ELSE, WHILE etc is determined by the level of indentation from the previous line. All standard code has no indentation and will always be executed.

To assist others in understanding your intentions, comments should be added before and after such areas to indicate the start and end.

```
;Move back and forward in a "wipe" motion stepping across 1mm on each pass for 5mm, then back again.
G1 X100 Y100 ; move to preparation point
;begin loop
while iterations < 10 ; create a loop that will repeat 10 times.  Note: Iterations is zero based
    if mod(iterations+1,2)=0 ; check if this loop is an odd or even number using MOD function remembering iterations starts at zero. so we add 1 to make the first loop an odd number
    ;start of code to be run on odd runs
    echo "This is an odd number loop"
    ;begin code to work out start point
    if iterations < 5
        G1 X{10 +  iterations} F1800 ; for first five passes, start at X10 and move 1mm across at each pass (10,11,12,13,14)
    else
        G1 X{19 - iterations} F1800 ; for last five passes, begin at X14 and move back 1mm at each pass (14,13,12,11,10)
    ;end code to work out start point
    G1 Y60 ; move down to wipe
    ;end of code to be executed on odd runs
else
    ;start of code to execute on even runs
    echo "This is an even number loop"
    ;begin code to work out start point
    if iterations < 5
        G1 X{10 +  iterations} F1800 ; for first five passes, start at X10 and move 1mm across at each pass (10,11,12,13,14)
    else
        G1 X{19 - iterations} F1800 ; for last five passes, begin at X14 and move back 1mm at each pass (14,13,12,11,10)
    ;end code to work out start point
    G1 Y10
    ;end of code to be executed on even runs
;end of loop
```