

Lauri Solin (Joel Kontas), xyplotter project (makeblock xy-plotter)

Embedded systems project course

Helsinki Metropolia University of Applied Sciences

Degree ICT engineer

Degree Programme smart systems

Project Report

Date 31.10.2018

**Contents**

[1 LASER SAFETY TIP 2](#_Toc528877616)

[2 Goal of the project 2](#_Toc528877617)

[3 Project plan 2](#_Toc528877618)

[3.1 The original project plan 2](#_Toc528877619)

[3.2 What actually happened 3](#_Toc528877620)

[3.3 Workload distribution 7](#_Toc528877621)

[3.3.1 Lauri Solin 8](#_Toc528877622)

[3.3.2 Joel Kontas 8](#_Toc528877623)

[4 implementation 8](#_Toc528877624)

[4.1 general description and flowcharts 8](#_Toc528877625)

[4.2 algorithms and explanations 16](#_Toc528877626)

[4.2.1 forlooping Bresenham (modern version, not allowing diagonal moves) 16](#_Toc528877627)

[4.2.2 RIT-interrupt driven Bresenham (demoed to the teacher) 19](#_Toc528877628)

[4.3 problems and solutions 23](#_Toc528877629)

[5 bugs, missing features and future development 24](#_Toc528877630)

[6 Outcome 25](#_Toc528877631)

[References 25](#_Toc528877632)

Appendices

Appendix 1. Title of the Appendix

Appendix 2. Title of the Appendix

# LASER SAFETY TIP

When you debug this project with MCUXpresso, into the plotterdevice, you should have the laser safety glasses on, because laserpin is ON by default in the beginning. AND, the MCUXpresso debugging stops into an initial breakpoint in the beginning, so the laser is ON in the beginning.

The laser will be turned OFF inside the real\_calibration\_task when the scheduler has started that task.

It may be possible that you can turn off the MCUXpresso initial debugging breakpoint, but I never remembered where you can turn it off from.

Also, currently the real\_calibration\_task accepts laser power commands, but I think it would be advisable NOT to use those laser power commands UNTIL calibration task has finished. Reason is because if you turn ON the laser, and send M28, you cannot any longer change laser settings during the steppercalibration and limitdetection phase… Only after the stepcounting is finished, will you be able to again send and receive mDraw commands.

# Goal of the project

The goal of the project for minimum viable product was to have pencil plotting mode and servo actuation working at the end of the course. The planned deadline of project would have been 12.10.2018. We did have a backup plan for a “soft deadline” in our group, so that we had capability to work on the project also during the autumn vacation from 13.10. - 21.10.2018. Then, we planned to be ready at the end of the autumn vacation, but the teacher pushed back the project deadline by another week also.

It appears, that our group managed to make the minimum viable product somewhat correctly, and we also implemented laser mode at the end. There were some problems using plotter 1, though, as will be seen later.

# Project plan

## The original project plan

At the beginning of the project, we had to setup version control, and some kind of backup for our sourcefiles. I had earlier experience with GitLab source control. So, we decided to use GitLab again in this plotterproject. But I had to re-acquaint myself with that software again, because I forgot most of git commands. The positive of GitLab is that you can have private repos for free unlike GitHub.

We had to also begin studying about Gcodes in general, and how to make the GcodeParser in visual studio with C++. We didn’t have any knowledge about Gcodes and neither did we know anything about mDraw. We intended to debug mDraw and try to find out what commands happen when you do something inside mDraw GUI.

The plan was to do all the GcodeParser testing within visual studio, and use filereading and filewriting to verify that the parser works correctly. We had planned to finish GcodeParser by 16.9.2018.

The other major hurdle in the beginning was to debug the connection link with mDraw and the C++ program.

Then, our other ongoing goals were also to stay up-to-date with the ARM processors lab assignments especially lab4, lab6, and lab7. All those labs had useful topics for our project such as stepper motors and SCtimer pulsewidth modulation.

The minimum viable product was intended to be able to plot with steppers constant speed, and have pencil servo working. We planned to test the code in the future mostly with the plotteremulator board, at least initially, and basically hope that there would not be serious bugs with the real plotter device. Stepper acceleration and

lasermode were to be implemented last if we had enough time. Planned deadline was 12.10.2018.

## What actually happened

We had some communication problems with Joel Kontas, sadly. I tried to reach him thru Skype and whatsapp, and I told him that we should come to the school so we can program there together with the plotteremulator, but he never did want to come to school for some reason. So, it was very difficult to organize the workload between the two of us, because I always had the plotteremulator board. And it was tough to plan remote work schedule because Joel didn’t always respond very quickly to the messages. So, then I just decided to do this project basically alone.

I developed the GcodeParser first in visual studio as a simple main() program, to test if it was able to parse Gcode. We used a struct called “CommandStruct.h” to transmit data out from the parser. We decided early on to use C++ string as the input just to make everything simpler to code as opposed to C-style char arrays.

We had couple of problems with the tokenizing portion of our GcodeParser. We tried to use stringstreams to tokenize a single C++ string delimited by spacebars, into a vector of strings. In this way, the parser could simply compare C++ strings individually easily to known values. This is a relatively common idiom for parsers to first tokenize and then compare the words.

The problem was that in MCUXpresso environment the LPCXpresso ran out of flash memory (program size). This mean that I had to basically hand-code the tokenizing function, so it no longer uses stringstreams. We managed to shrink the code footprint such that the GcodeParser only includes <string> and <vector>, and this fixed the problem, so our code builds successfully in LPCXpresso. There currently also exists a refactored version of the tokenizing function called tokenize\_input\_refactored, which also should work (was unit tested), but it is not in-use currently.

The first testing was done by having two separate gcode.txt files with 100 lines of Gcode, where Gcodes were on each line. Each text file contained either all legal inputs, or all illegal inputs. Then the test program reads from text file and if the results were as expected, then it writes the line number and the gotten results to an output text file. This was effective and simple way to test the parser. Later, I managed to study how to implement actual unit tests for C++ native unit testing framework, but the results were mostly unchanged, so it was little bit wasted effort on my part in that sense.

Later, we read the updated plotter documentation pdf, and it described the extra Gcodes that we should implement (M11 limitquery, M5 save values etc..). Then we realized our parser also needed to implement those extra commands. Then we simply kept updating the GcodeParser inside the main project in MCUXpresso. It worked well.

Overall, our parser was pretty “defensively coded”, so we prepared quite well for erroneous inputs, but other groups had “leaner and more computationally efficient” parsers that still worked well. Our parser was computationally a little bit slow because of C++ new and delete operations under-the-hood with vector and C++ strings, and extensive checking of the inputs. Inside MCUXpresso, we turned our GcodeParser into a separate class for clarity purposes.

Our parser was delayed because of the stringstream tokenizing problem and my personal desire to self-study how to implement real life unit tests in C++ test framework.

Communication link between mDraw and C++program was then easy to debug once we had implemented the usb-project files as a base for our project. Lab7 exercises provided useful background information for me in order to understand the usb-drivers. We simply used breakpoint to debug the C-style buffer that was gotten with the usb-drivers, so we knew what mDraw sent.

A small problem we originally had in parse\_task was that we ditched some Gcodes instead of gathering the incoming Gcodes into a “stream of Gcode”. We fixed that with a C++ string, where we appended more incoming Gcode at the end, and we kept parsing the substring from the beginning of the “stream of Gcode” until a delimiting newline is found. Then we could process that gotten substring of Gcode in the parser, delete the substring, and keep going.

The problem in the IDE was that the ITM\_print was totally bugged and gave garbled and jumbled printouts in the ITM console. The ultimate reason for wrong printouts was never discovered, but the substrings of Gcode were always the correct values I think, because certain breakpoints never triggered when we debugged. We didn’t seem to have missing Gcode problems later-on either.

For the simulator testing phase, we had pursued two paths of development for the Bresenham algorithm. Firstly, we started with the Bresenham algorithm as it was described in Wikipedia article. It was a for-looping Bresenam algorithm.

But, after googling I also found John Bresenham’s original publication of that Bresenham line algorithm from the year 1965.

*(IBM Systems Journal; vol4; no1; year1965**; “algorithm for computer control of a digital plotter”; J.E. Bresenham)*

*(*[*https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwj6wYeSpLPeAhXHjywKHS5iCdcQFjAAegQIARAC&url=https%3A%2F%2Fpdfs.semanticscholar.org%2Fc443%2Fc0b5f74f75d87193bc373cdc5b0b61cf28fd.pdf&usg=AOvVaw3yFjSzJj2nA1qogWxZLi7u*](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwj6wYeSpLPeAhXHjywKHS5iCdcQFjAAegQIARAC&url=https%3A%2F%2Fpdfs.semanticscholar.org%2Fc443%2Fc0b5f74f75d87193bc373cdc5b0b61cf28fd.pdf&usg=AOvVaw3yFjSzJj2nA1qogWxZLi7u)*)*

After having studied the IBM journal about Bresenham, I decided that we could pursue that second path of development for Bresenham such that the iterating portion of the algorithm would be interrupt-driven, instead of for-looping iteration. The article had very good pictures and a table-of-values at the end, which I used as help in understanding the Bresenham algorithm.

I did a mock prototype in visual studio where I coded a fake “interrupt handler” which was just a regular function, and then I used some global variables for the Bresenham algorithm and the coordinates. Then, in the main(), I just initialized some coordinates, added some testcases for ending coordinates and expected algorithm output, and then I called the “interrupt handler” in a while-loop.

Then, I just debugged the algorithm, so I could code the RIT-interrupt handler better inside MCUXpresso. Both Bresenham versions were tested in plotteremulator and they seemed to work quite fine. But, the RIT-driven Bresenham version was the version that I demoed to Keijo on 31.10.2018.

There were three versions of the RIT interrupt handler, first two were the original interrupt handlers, first one was for RIT-driven Bresenham (which was demoed). The second one was a forlooping version Bresenham which allows diagonal motormoves (this version of ISR is commented-out currently). And the third one (there is some octant checking inside that ISR) is a forlooping Bresenham, which does not allow diagonal motormoves, but instead the diagonal motormove is translated into a sequence of one horizontal followed by one vertical move. This will be later explained better in the algorithm portion of this report.

Pencil servo actuation code was basically copy-pasted/recoded from the lab7 exercise solution, so it worked well in the simulator and later also with plotterdevice.

I decided to implement real\_calibration\_task, in such a way that it is the first “user-defined task”, that is running (cdc-task is also running, of course). Calibration task also initializes all pins in the beginning. Then there is also an EventGroup, on which the execute\_task and parse\_task are waiting.

When the calibration task has finished executing, then it notifies the EventGroup and allows other tasks to unblock. The usage of an EventGroup, prevents lots of unnecessary pointer checking for nullpointer.

Later, I also received the information tip from the teacher that we should make the usb-interrupt priority less important than the RIT interrupt priority, and this was done also. Then, I also downgraded by usb-cdc task into the same priority as the other tasks such as execute\_task and parse\_task.

I had some major inaccuracy problems near the end of the project when I used plotter1. But, when I debugged the identical code on 29.10.2018 with all three plotters, the code seemed to be accurate with plotter2 and plotter3.

The lasermode worked with plotter3 very accurately when I demoed the code to the teacher on the real deadline 31.10.2018. This was in stark contrast to the lasermode with plotter1 when I tried that lasermode during the autumn vacation. The laser itself worked powerfully in the plotter1, but the stepping was totally inaccurate and missing steps.

We didn’t have enough time to implement acceleration ramping and deceleration ramping. We concluded through testing that a viable minimum speed was 1000pps for the RIT-interrupt driving. There was also some stabilizing vTaskDelay(2) added after G1 command finishes plotting. It may be unnecessary delay.

With 1000pps you could basically stop immediately if you hit the limitswitch, and you could make relatively smooth stepping when Gcode plotting sharp-corner figures (where one axis has to stop and switch direction, like the tip of a triangle). It seems that the final day lasermode plotting seemed to verify that the 1000pps works well as a minimum speed because the laserplot of Metropolia logo was very accurate.

## Workload distribution

Most of the work was done by the author (Lauri Solin).

### Lauri Solin

Lauri Solin (author) did most of the work, including planning, programming, testing, and documentation and debugging. Overall, I would say that I did the lion’s share of the work in this project in terms of planning, dedication, effort and effective work.

### Joel Kontas

He did some debugging of GcodeParser in the early stages of the project and maybe also helped with planning. He did find our tokenizer stringstream bug, so that was valuable. He also maybe did some mDraw debugging if I remember correctly.

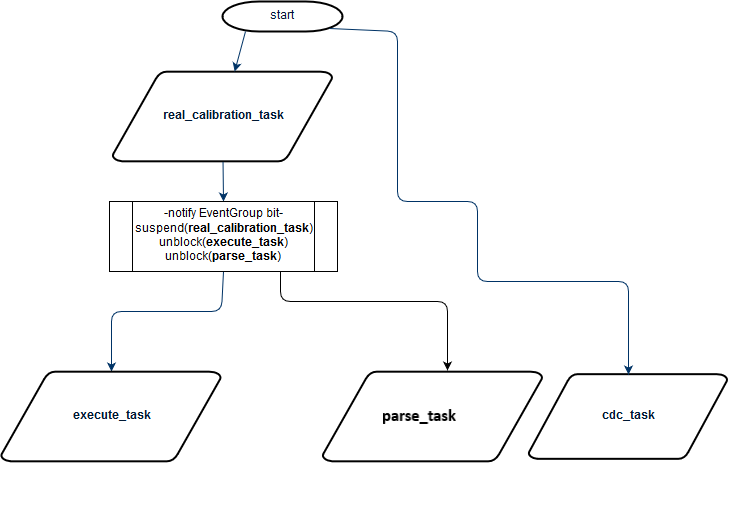
# implementation

## general description and flowcharts

Here is a diagram of the FreeRTOS tasks that are being used.

The main architecture is a simple consumer-producer design. Commands are communicated through a queue.

If limitswitch contact is detected while the plotter is plotting (i.e. the plotter-piece is moving due to G28 command, or G1 command), then the program should stop (execute\_task and parse\_task task-loops stop, and tasks block themselves). Currently the constant speed is 1000 pps so there aren’t much problems when stopping.



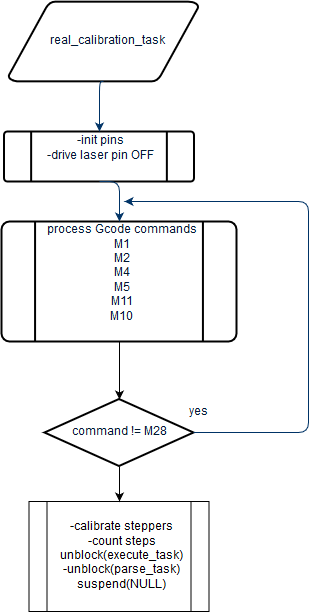
Notes about the real\_calibration\_task

* initialize pins, and switch the laser OFF
* probably you should not enable laser power at this stage, yet (wait until calibration task is done)
* limitswitch detection was implemented with std::swap and swap-by-value of the DigitalIoPins, it seems to work quite well, and the added benefit is that pointer-swapping is unnecessary. So, you can keep the limitpointers pointing at the same pin objects. DigitalIoPin objects should define themselves only by their own datamembers (port and pin) so swapping the datamembers by value works well.
* Calibration task allows pencilservo calibration and certain mDraw commands in the beginning, before M28 is received.
* M28 breaks out of mDraw command polling loop, and initiates steppercalibration and stepcounting
* After stepcounting, update currentCoords, originCoords, and xFullStepPerMM and yFullStepPerMM ratios
* Ending position for the plotter after steppercalibration should be the originCoords
* Suspend itself and unblock execute\_task and parse\_task
* NOTE!:: the plotter-piece must not be touching any of the limits when powering up the device and starting the steppercalibration phase and stepcounting phase.
* NOTE!:: the current code is written for microstepping mode

This means that e.g. when the plotter-piece has found the expectedYmaxlimit => then, a for-loop is run in the opposite direction for 360 fullsteps, so that it clears the limit trigger

This is not a whole lot of steps in the microstepping mode, but it is useful to remember if you change the steppingmode. The amount is controlled by const int variable g\_STEPS\_FROM\_EDGE. Ideally, this amount should always be enough steps such that you are able to clear away from the limitswitch trigger.

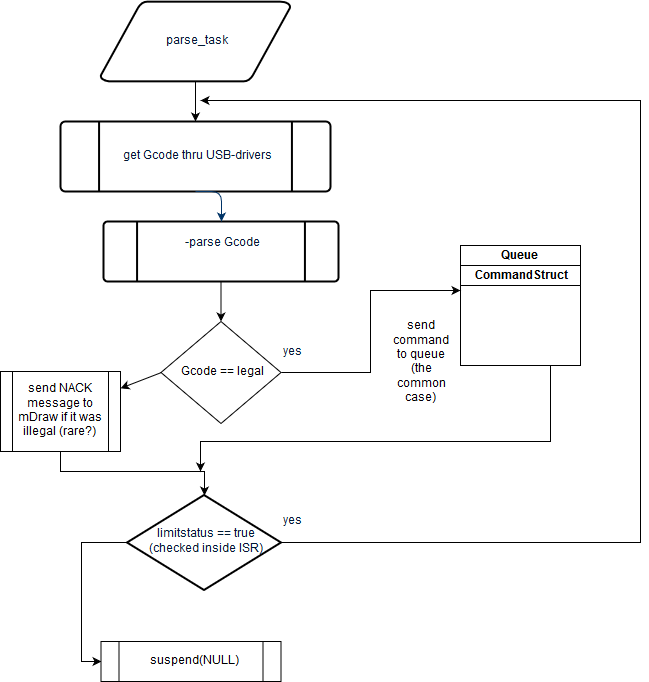
Here is basic flowchart for real\_calibrate\_task



Notes about parse\_task

* Queue is currently size 20, but parse\_task doesn’t respond to Gcodes unless they are illegal. Queue size was 20, because in my earlier version I had the parse\_task responding with usb-responses. But, with the real\_plotter there is always some “real-world lag” from stepper moving or servo actuation etc… so that the bigger queue size probably won’t be so helpful.
* In regular case Gcodes are legal => then the Gcode usb-response is given by the execute\_task after that command was executed, so effectively queue length should be one because of that
* g\_limitStatusOK is a global bool variable that controls if the parse\_task-loop keeps running (value is written to inside RIT\_ISR)
* g\_limitStatusOK is written to inside ISR and also polled inside ISR when plotter-piece is moving, and also this variable is polled in the task-loops as a condition.

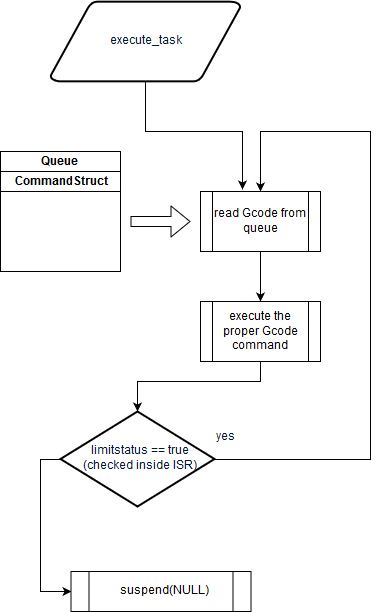
Here is basic flowchart for parse\_task.



Notes about the execute\_task

* there exists a global PlotterSettings object which keeps track of saved mDraw settings like plotWidth, plotHeight, plotSpeed etc…
* PlotterSettings datamembers are modified with certain mDraw commands like M5
* g\_limitStatusOK is a global bool variable that controls if the execute\_task-loop keeps running (value is written to inside RIT\_ISR)
* CommandStruct xCoord and yCoord are stored as integers but their units are hundredthsOfMillimetres (so that xCoord == 102,55mm == 10255 as integer datamember)
* Separate xStepRatio and yStepRatio are used to convert millimetres into fullsteps (there is only a small difference typically, as expected)
* plotter coordinates are stored in global int variables g\_curX and g\_curY (currentCoords are incremented/decremented inside RIT-ISR)
* originCoords will be (int g\_OriginX == 0, int g\_OriginY == 0)
* pre-processor defines (#define useLoopingBresenham) can select between RIT-driven Bresenham or forlooping Bresenham, but the demoed version was the RIT-driven Bresenham (so, that you comment-out that define, to use the RIT-driven Bresenham)…
* in the older versions, I used the currently commented-out RIT interrupt handler function, as the actual forlooping Bresenham version. I think that old piece of code would no longer work, because I recoded the forlooping Bresenham to disallow diagonal motormoves.

Here is basic flowchart for execute\_task



## algorithms and explanations

I will describe the currently existing two versions, forlooping Bresenham (don’t allow diagonal moves version) and the RIT-interrupt driven Bresenham version (demoed to the teacher and plotted with laser also).

### forlooping Bresenham (modern version, not allowing diagonal moves)

The basic structure of forlooping Bresenham algorithm together with its helper functions was coded based on the Wikipedia article.

([https://en.wikipedia.org/wiki/Bresenham%27s\_line\_algorithm#All\_cases](https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm%23All_cases))

The pseudocode was easy to translate to C++ code.

RIT interrupt will be utilized to occur for each fullstep, such that the steps are initialized inside for-loops (plotLineLow, plotLineHigh functions; initialize RITstart with 2 halfsteps).

The Wikipedia solution of the Bresenham was complemented with an easy helper function, that makes it simpler to execute the motormoves.

int getOctant(bool a, bool b, bool c)

The helper-function above returns the active octant for that executable G1 command. The getOctant function is based on the tabulated results section of Bresenham’s IBM journal paper (p.4 in the online pdf version). The first three columns from the leftside in the table, are used as Boolean inputs into the getOctant function to determine the active octant for the lineplot.

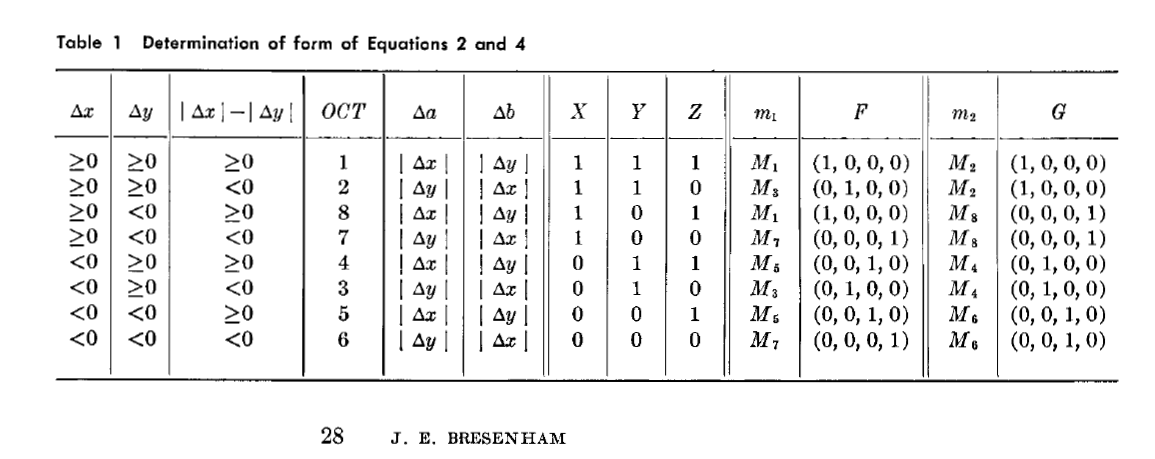


Figure 1 tabulated section used to code Bresenham helper functions

The active octant is also used to compute the dirXpin and dirYpin directions correctly, before each lineplot is done. This can be seen in the tabulate section by cross-referencing the three columns together; oct-column, m1-column, and m2-column.

Note that the m1-column values and m2-column values were simply translated into g\_m1parameter integer variable, and g\_m2parameter integer variable in the C++ code.

The current forlooping Bresenham was coded to allow only straight motormoves in a sequential manner, to hopefully avoid timing problems in motorpulsing inside RIT-ISR.

This was motivated by the fact, that the original Bresenham algorithm in the IBM paper, would have had simultaneously timed motorpulses in the diagonal m2Motormove (45degree plot). In real world this would require precisely timed simultaneous step pulses to both stepYpin, and stepXpin. (these would still be halfpulsed with the RIT interrupt)

In the modern forlooping Bresenham these diagnonal motormoves were converted into sequential straight motormoves as shown in the picture in orange and blue vectors. The numbers near the origin indicate each octant from 1 to 8.

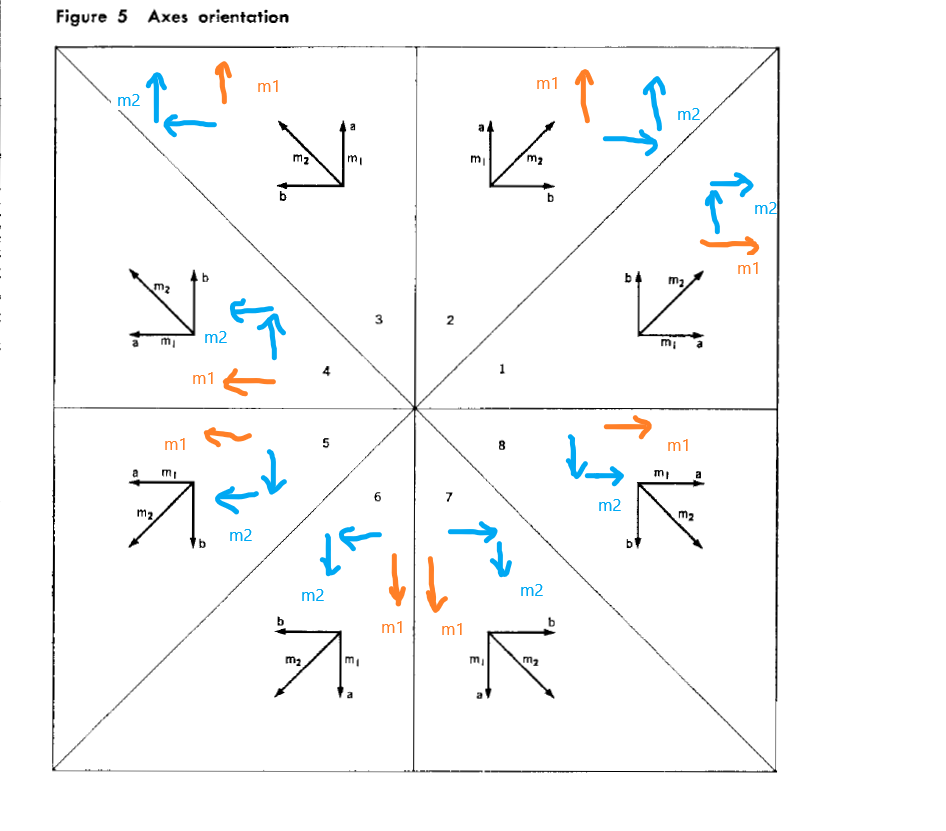


Figure 2 shows m1Motormoves and m2Motormoves in each octant, blue color is the m2Motormove, and orange is the m1Motormove

The forlooping Bresenham version was tested with plotteremulator board and it seemed to work somewhat OK. But this version was only tested with plotter1 on the real device and did not seem to fix the inaccuracy problem with that plotter.

Note that there is that difference that this sequential straight move version, translates each diagonal fullstep m2Motormove into two sequential straight fullsteps. The sequentiality of the straight moves, is controlled by the forloop. But the coordinate increment/decrement is done inside the RIT interrupt handler, as well as the stepPin pulsing also.

### RIT-interrupt driven Bresenham (demoed to the teacher)

This version uses RIT-driven Bresenham and was tested to work well with plotteremulator board and seemed to work accurately with real plotter2 and plotter3. Lasermode was also demoed and seemed to verify practical accuracy of the plotting.(?)

The RIT-driven Bresenham allows diagonal m2Motormoves.

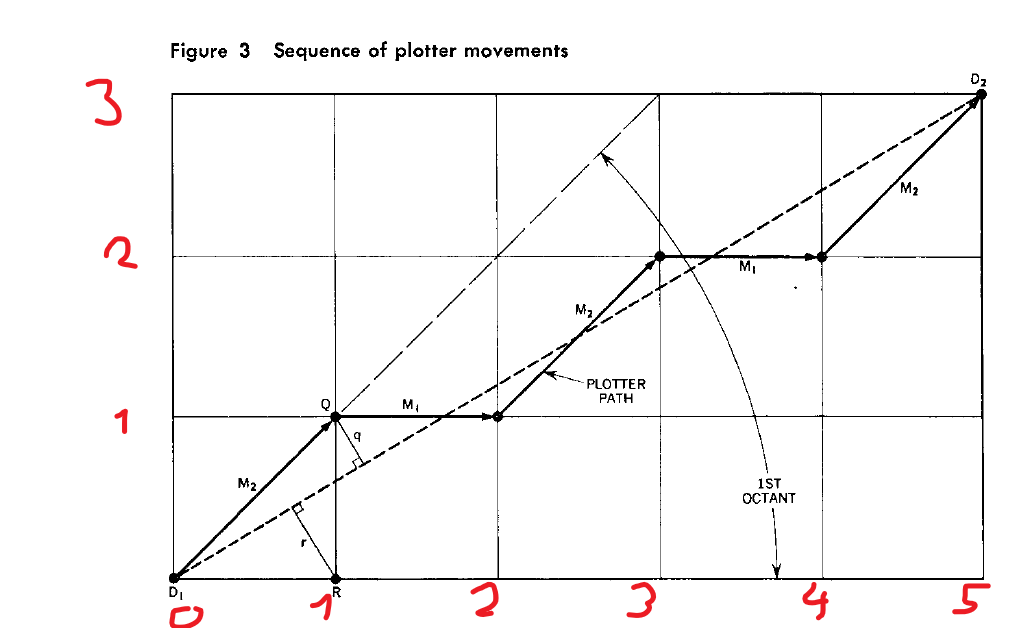
The picture below shows a good example diagram plot from Bresenham’s paper, from (0,0) source-point into (5,3) destinationpoint, using diagonal motormoves and straight motormoves.

Figure 3 example Bresenham lineplot where dx = +5, and dy = +3

The corresponding timing diagram for the given example plot above, could be something like below. Note how both step pins are supposed to be pulsed at the same time with m2Motormove.

I did check this same example plot with a logicanalyzer with the LPCXpresso, with a test plot task. The timing diagram was quite similar to this hand-drawn image, but I forgot to take the screenshot. Also, in real life step pulsing there seems to be some small amount of lag between stepX.write(pulsestate) and stepY.write(pulsestate). This seems to be inevitable because the RIT interrupt handler takes about 35-50 microseconds to execute completely. (that was measured with the cyclecounter register when debugging with breakpoints)

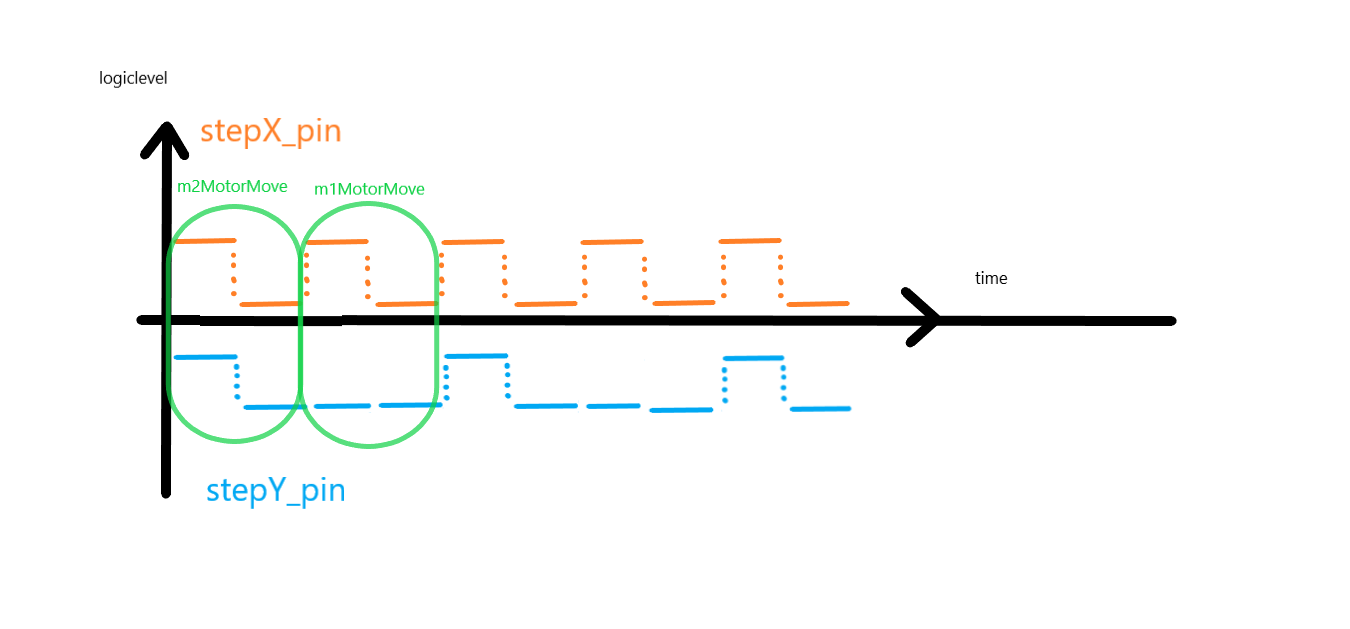


Figure 4 timing diagram for corresponding Bresenham lineplot example (dx= +5, dy= +3)

Bresenham algorithm was mathematically defined by set of four major equations. I used the tabulated section for the Bresenham helper functions also. Those helper functions are used in pre-computation per each lineplot.

(Eq1 is a recursive relation)

Eq1:

Eq2:

Eq3:

Eq4:

(Eq4 affects dirXpin and dirYpin)

(Eq2 affects whether or not swap the driving axis)

Eq1 and Eq3 are iterated by RIT-ISR, whereas the other equations are pre-computed for each lineplot.

The tabulated section determines the values all variables in precomputation.

Note that in the Eq1 where there is and *,* those variables are actually absolute values inside the iteration, as can be seen from the table below. If swap is required then the driving axis simply changes, otherwise x-axis remains the driving axis.

Note also that octant must be determined based on the “raw dx” and “raw dy” values. Raw dx, would be the actual amount of difference in X-axisSteps between source and destination.

Whereas the third column from the left, uses the absolute values from those “raw dx” and “raw dy” (before any kind of swapping is done). So, in conclusion the octant is determined with the first three leftmost columns and will utilize that raw dx and raw dy values and their respective absolute values.

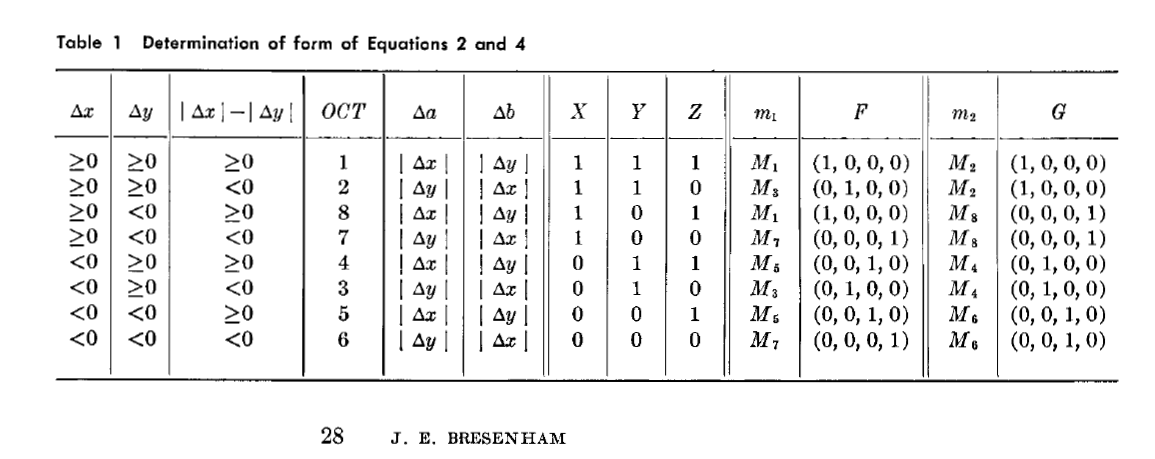


Figure 5 table shows how the required Bresenham variables are precomputed

m1Parameter and m2Parameter can also be seen clearly in the diagram below.

Note there are only four choices for m1Parameter and four choices for m2Parameter.

* m2parameter is seen circled in blue, and they control m2Motormoves (diagonal)
* m1parameter is seen circled in orange, and they control m1Motormoves (straight)
* these parameters determine the dirX and dirY pins for each lineplot, as can be seen from the diagram below.

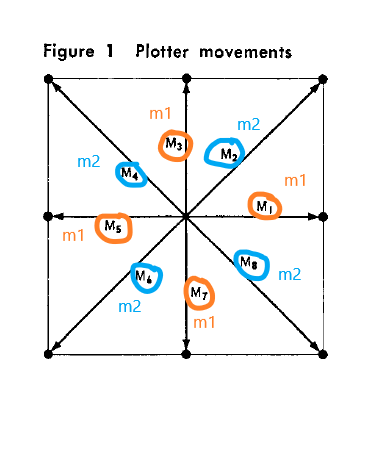


Figure 6 plotter m1Motormove, m2Motormove and the respective m1parameter and m2parameter

## problems and solutions

I had a weird version control disaster during the week after the autumn vacation. I had apparently forgotten to stage some commits into the currently worked branch. Then, I tried to create a new branch and checkout into that one, and keep working from there. I lost maybe two days’ worth of changes, because of this version control disaster. So, I just recoded those changes the next day (there weren’t too many changes, and I basically remembered what to do).

After that disaster, I kept also making local zip archives of the project on different part of the hard drive just to be sure. With git workflow it is always best to commit often, it seems…

There were major accuracy problems with plotter1, which did not appear to happen with plotter2 and plotter3. I tried to debug that inaccuracy issue with plotter1 for an entire week, I could not find a good fix for the plotter1 inaccuracy. Even with very small pps value like pps 500, there were still inaccuracies with plotter1.

By luck, I just decided to test the same code with all real plotters because on the evening 29.10.2018, I was testing some debug solutions with plotter2 in “korppi-classroom”. The plot appeared to be unusually accurate with plotter2, so it sparked my interest to test the same code with all the plotters.

# bugs, missing features and future development

The most puzzling bug was that the same code works well with plotter2 and plotter3 but fails miserable in stepping accuracy with plotter1. The true cause is currently unknown, and I don’t have a fix for plotter1 inaccuracy.

Occasionally (rarely) there is also a “bug” that mDraw crashes in the middle of a plot, but it is currently unknown if these are just mDraw’s own crashes or if they started with my code. I think that sometimes this “bug” can be induced if you increase the mDraw plottingspeed to the maxSpeed value, and then save the plottingspeed value and start plotting.

Missing feature/bug is that I configured apparently the axes in the wrong way in the C++ code, because in mDraw if you want to plot e.g. text like the Metropolia logo, you must plot the mirror image in mDraw, so the real plotted image is drawn correctly.

For future development there could be the addition of acceleration ramping maybe with increments/decrements of 100-200pps such that with each step it increments that amount, and then runs at some constant pps value. You would have to check before accelerating if you are running enough amount of fullsteps, before deciding to accelerate, and otherwise run the amount of fullsteps at constant 1000pps.

Another small quick fix could be to remove existing vTaskDelay(2) commands from refactored\_BresenhamInterruptAlgorithm function and plotLineGeneral functions. Then, you would see if the plot accuracy is still preserved, and also the delay from G1 commands is reduced.

The difficult part about acceleration while also plotting with pencil seems to be as follows

* ideally, you should accelerate, & decelerate only the least amount steps possible, and run the maximum amount possible at some high constant speed (such as when you get the first G1 command, and then you keep getting lots of G1 commands in a row). So, for theoretical purposes you should have G1 commands inside “command blocks” all-together, separated by M4 pencilservo commands. Then, you would theoretically know the “total plot distance” of the “command block”. So, then you would theoretically only accelerate with the first G1 commands, and decelerate at the last G1 commands.
* But, the problem is that if the G1 commands are making sharp turns, then you could still miss steps, because if you don’t slow down enough when making a sharp turn, it could be inaccurate (e.g. the tip of a triangle).
* So, in the end maybe you have to approach it pessimistically and always follow these cases
  + For small amount of fullsteps per lineplot, plot at constant speed which is also easily stoppable speed, and startable speed
  + For bigger amount of fullsteps per lineplot, be prepared to accelerate, const speed, and decelerate for each G1 command.

# Outcome

Overall, everything went pretty much better than expected

Perhaps with more time I would have tried to have the simplified acceleration ramping.

Sadly, I was not able to get the code working very well with plotter1 because of the inaccuracies.

# References

1. IBM systems journal; vol4; no1; 1965; J.E.Bresenham; “algorithm for computer control of a digital plotter”
2. English Wikipedia article Bresenham line algorithm

**Appendix (source files , code snippets, and reference)**

Useful source files:

* Mcuxpresso main plotter project
* Visual studio bresenham RIT interrupt handler mock prototype
* Unit tested GcodeParser was already returned to oma assignments box (but that version was a slightly older parser version, which didn’t have all the Gcode commands implemented). The mcuxpresso will have the true GcodeParser version of course
* Pdf version of the IBM journal for Bresenham will also be provided for teacher’s reference, if he wants to read the code.