Welcome to SAC – The last stepper-motor interface you'll ever need.

SAC is the rocket science of stepper-motor control. It does all the heavy lifting with respect to timing, signaling, acceleration, deceleration, step count, micro-stepping and more. It is the missing link in any linear/rotary solution, providing the bridge between a user control application and the electronics that drive modern stepper motors. All SAC needs are some simple instructions, everything else is taken care of.

There are many companies providing hardware solutions in the motion market. Many more willing to build applications that will accommodate all your logic, and then some. There is however, a vast chasm that separates the two. SAC pulls that gap together with a platform-independent interpreter/controller that will take simple ASCII command strings and provide the signaling needed by the driver. A simple yet elegant solution whose time is now. Increasingly, stepper motors are being used for both rotary and linear motion. Their use would find immense new markets if the customer had convenient access to their capabilities. As it stands, without an engineering department capable of micro-coding solutions in a cost-effective manner, the market is restricted to large organizations with vast budgets, or hobbyists willing to tinker. SAC aims to turn the $2billion stepper motor market into a $20 billion concern by making the use of the motors as simple as the controls on your car radio. With the emerging markets in industrial automation and robotics, there is nothing in the way.

SAC is the newest addition to our suite of motion solutions. Pared down from its bigger siblings to fit on a single, inexpensive chip, SAC offers performance and functionality previously available only on systems many times larger and more costly. What really makes SAC shine though, is its simple interface design. Whether talking to an Android app on your smartphone, or a central command and control system, the intuitive instructions lend to quick profile planning. Program the command queue and set autorun for unattended applications or network and coordinate with synchronized motion for multi-axis, operator-controlled implementations. How imaginative can you be?

SAC is a combination platform of interpreter/driver providing stand-alone user control of a single stepper motor. All activity associated with the motion of a stepper motor, acceleration, deceleration, cruise, continuous motion, are all available to the operator with simple commands through a variety of communications options. Just tell SAC what you want done, be it cruise at 60 RPM for 3 minutes or loop 180 degrees back and forth every 3 seconds, the possibilities are endless.

While SAC was designed from the ground up as a stand-alone, single-axis controller, its real strength comes from its synergistic relationship with its siblings and mother. ‘mother’ is an open-source front end for SAC or multiple SACs. With a robust RS485 network providing the platform for communications, mother is the command, control and coordination center. mother will function as single point of contact for all SAC in the network, or simply as an adjunct processor for a single unit. In either configuration, the sum is greater than the parts.

Originally designed for industrial and commercial robotic systems, SAC hardware is lightweight and compact allowing for installation alongside or nearby the motor if needed. Requiring just two wires for network command and control, its presence is unobtrusive as well. According to specifications, the network will support in excess of 200 nodes, all coordinated by mother, thus the n-degrees of freedom sought in today’s advanced systems is now a reality.

SAC can be command driven from a user terminal or have its motion managed externally through mother or the user’s choice of application. Interfacing with SAC is not rocket science. What SAC does is translate

motion commands into timing profiles that are then fed into a controller which produces the electrical signals to drive the motor. Completely configurable, the designer even has the choice between a non-linear harmonic curve and a constant acceleration linear ramp for velocity planning. Motion can be programmed into a loop within the command buffer, or an array of command references can be executed continuously. Seamless transitions from one profile to the next help to minimize disruptive jerk associated with speed changes.

SAC is a multi-process state machine. It has five operational states: ACCEL, CRUISE, DECEL, STOP and STANDBY. In ACCEL state the motor is being accelerated (up or down) to a new target velocity from the previous, then current, velocity. This is an important concept as many systems will seek the simpler (mathematically) path of returning each command to zero and beginning the next command from there. At high acceleration rates this may be indistinguishable from continuous motion. Incurring these additional ramps however, can significantly increase processing time, especially is applications where many short vectors are required (describing a circle or arc for example). It a truncated command mode (see discussion later) there may never be enough time to reach a target velocity before having to execute the deceleration phase of the command. SAC offers the option of seamless streaming by GAP testing and the retention of inertia from one command to the next thereby staying within the pull-in and pull-out torque curves. The top speed attained is command n-1 is held as the starting speed for command n through further acceleration to present yet another, higher speed to command n+1. The prudent profile planner can use this capability to ramp up to any target velocity, through any changes necessary along the path without the loss of productivity in having to bring the system back to zero. This is just one feature inherent in SAC that makes it the most diverse interpreter/controller available today.

The CRUISE state supplies steps to the motor continuously while keeping track of those already consumed by acceleration and the run state until now. Having already calculated the number of steps required to terminate the current command, SAC will exit the CRUISE state at the appropriate moment prior to its next phase.

The next phase is either DECEL or STOP depending on the number of steps required for deceleration. Mentioned now, it will be discussed in detail later, there is no DECEL state except as a transition to STOP. If there is a command in the queue waiting to be processed as a seamless continuation of the current command, the current command is profiled with zero deceleration. The CRUISE state is run to completion consuming all steps assigned and is abandoned at its achieved velocity for the subsequent command to assume. If there is a change in velocity, up or down, that will become the acceleration ramp/curve for the incoming command and be charged against its step total.

Regardless, DECEL or not, the state following is STOP. Now this is not what it may seem and as such there is probably some confusion in mistaking this state for STANDBY. The STOP state must be passed through at the end of each command processed as there are housekeeping tasks that need to be performed. Every command will have a RUN, CRUISE or DECEL state (or a combination thereof, or all three) and a STOP state. If there is no command in the queue, then the systems will enter the STANDBY state, waiting for instructions.

Within SAC there are three distinct processing environments, held together with several support functions. These are the message processor, the command processor and the motion engine. The message processor handles all communications and provides the HMI/MMI. It parses network and terminal traffic and takes action accordingly, feeding the queue, proctoring configuration or simply chatting with the neighbor.

The command processor manages the queue. It breaks down the command profile and builds it backup taking into consideration all of the system parameters, the previous command and the next command. This is the workhorse of SAC.

Last (and most definitely not least) is the motion engine. This is where all the heavy lifting is done. The motion engine controls all the internal clocks, interrupts and signaling. This is where the rubber hits the road. Commands are translated into timing sequences which in turn are converted to electrical signals to be

passed on to the motor driver. All profile calculations pertaining to cycle timing, acceleration, deceleration, adjustments to account for microstepping, clock frequencies, motor characteristics, its all done in the motion engine. This is what makes SAC what it is.

What SAC is not. SAC is not a CNC or 3D printer controller. It can however, control as many axes, rotary heads, filament feeders as you wish. Simply break the motion commands into their constituent vectors and send those commands off to SAC. mother has functions built in that do the calculations for x-y control. SAC is also not a toy. While it tries to be user friendly, one must realize that there's a lot of deterministic processing going on that precludes verbosity. Relative to the timing sequences necessary to bring a motor up to speed with microstepping and the accompanying ramp calculations, the time required to send one character to a display is forever. Often there is no direct feedback of a command having been issued although discreet verification is possible through command line requests. The operator must be aware that once machines are in motion, damage and/or serious personal injury are very real possibilities if the machine doesn't behave as expected. Know your commands, know this manual and above all, know where the emergency stop is.

Mostly the operator/designer is left alone to make their own mistakes. SAC is not forgiving. As it is said, computers will do what you tell them to do, not what you want them to do. Care must be taken that every command falls within the operational limits of your particular configuration with respect to loading, inertia, stability, etc. There is too much computational overhead for SAC to check the validity of operator input.
In fact, communicating with a human takes so much time (relative) that it is reduced to its barest minimum. In debug mode, you simply cannot function in real-time as the xmit times required to send screen data exceed cycle times in all but the slowest speed cases. There are exceptions where SAC will intervene and these will be noted as they arise. For now consider the time required to load a new command and continue uninterrupted operation. Depending on the state left by the last command, this may or not be possible. SAC performs a check between the end of the last command and the start of the new command. If there is reasonable assurance that uninterrupted streaming will not risk a motor stall, processing will continue where it left off, otherwise the new command will have to accelerate from slow speed. If you try microstepping at high speed, this could be a problem and as commands take different times to load, it would be fruitless to offer guidelines without specific operational parameters. If in doubt, you can always program a quick bounce down to a slower speed. For planning purposes, commands take between 8 and 40 microseconds to load. The more time-consuming loads are associated with changes in acceleration as these involve some floating point math. There are timing loops and console displays in debug but one must consider the overhead added by debug itself. A typical command load involving steps and speed takes about 16 microseconds. Note that the time to process for a command as reported under debug is the time taken to receive the serial input, parse the data and load it into the command buffer, NOT in presenting the data to the motion engine. The input/parse process is interruptible by the pulse generator so it's not an issue.

DISCLAIMER

This document was prepared by the engineer that designed and built SAC. His familiarity with the product will no doubt result in many an invalid assumption and oversights that will make the text and command structure appear confusing. This is not intentional. If there are questions and/or comments, please address them to apengineering@yahoo.com. Individual replies may be forthcoming, otherwise look to revisions which incorporate your concerns. The current release of this document can be downloaded from

github.com/DKWatson/SAC

Happy stepping.

(this is a document in process, as in not yet finished)

Discussion

The essence of motion: start – accelerate – cruise – decelerate – stop, all related by position and time.

Throughout this document, written and compiled over the life of the program, there are references used interchangeably. In most cases the alternate is parenthesized at first use, most cases.

Run and cruise both refer to the state between accel and decel.

Accel = acceleration = acc, decel = deceleration = dec

CLI can either be the engine, the command line interpreter, or the command , command line instruction.

Single letter commands/attributes/parameters often bear little or no resemblance to that which they define/describe. The primary consideration in assigning identifiers is to minimize communications overhead, not ambiguity. In most cases, they are chosen such that accidental keyboard entry is not a part of the equation.

SAC has been tested against the 1,001 things it was designed to do and about half of the million things it wasn't. It is entirely possible that you may attempt something that it was never designed for or tested against, in which case all we can say is that the results are unknown. If they prove to be useful, we'd love to hear from you. We have already added descriptions of functionality that have been 'stumbled' upon. SAC is a complicated piece of software that utilizes several 'tricks' and code-sharing techniques to enable it to fit within the confines of the hardware platform. At times this has resulted in unexpected additional functionality, mostly in parallel with design, but a bonus regardless as an option. Not all of these have been documented. One example of this is the (L/I) command (no qualifiers) which was laid in place to terminate a loop. As it happened, the steps necessary to close off a loop and restore the state machine to an otherwise 'normal' operation are quite complex and the result was that (L/I) can in fact be used as a universal terminator (with grace).

BTW, with prejudice and grace, and/or their derivatives, are used quite frequently to differentiate between an immediate stop, do not pass Go, do not collect $200 (prejudice) and a stop that allows the current command to complete, perhaps with deceleration (grace). The former should be equated with 'emergency stop'.

Curve and ramp generally refer to the same thing which is an accel/decel profile. When plotted against time, one will be a curve and the other a (nearly) straight line.

Network and terminal differentiate between communication protocols. Network connections are packets, terminal connections are not.

Velocity (speed) – the change of position over time, perhaps most commonly thought of as miles per hour (mph). In the case of stepper motors, where the primary motion is rotational, this is commonly referred to as radians per second. This measurement cans be translated many ways and converted from rotational to linear whence it becomes inches per minute or millimeters per second. As these translations are application specific, we'll stick with more conventional notation and one perhaps more familiar – revolutions per minute (RPM). Regardless of which doctrine you subscribe to, it will all resolve to pulses per second, defined by the period or cycle count.

Acceleration – the change of velocity over time or the rate at which the speed changes. If we go back to the car analogy and examine comparison specs, there will often be '0-60 in x seconds'. This is a measure of the vehicle's ability to accelerate from a standing start (0) to 60mph in a given amount of time. If for example, the starting speed is 0 and 60mph is reached after 10 seconds, we say that the car has accelerated at a rate of 6mph/sec (final velocity minus initial velocity, all divided by the time). The key point is that the bigger the number, the quicker it gets up to speed.

The primary acceleration ramp used by SAC is a Harmonic curve, easing away from a resting stop and approaching straight-line towards the target. This approach was selected for general use as it minimizes jerk and for profile planning purposes it's the least cumbersome mathematically from start to target the speed is increased/decreased by the value specified as acceleration or deceleration. The target speed can be specified as a cycle in microseconds offering the highest resolution available, the accel/decel ramps are a means to get there.

The Harmonic curve is expressed mathematically as 1/n where n goes from 1 to infinity. It is divergent which means basically that there is no definitive solution to its calculation. What makes it practically applicable as an acceleration profile is that when values of n are small, the result is comparatively large. In our use, everything is translated into a value which represents in microseconds, the duration of the cycle between pulses – the period of the pulse train.

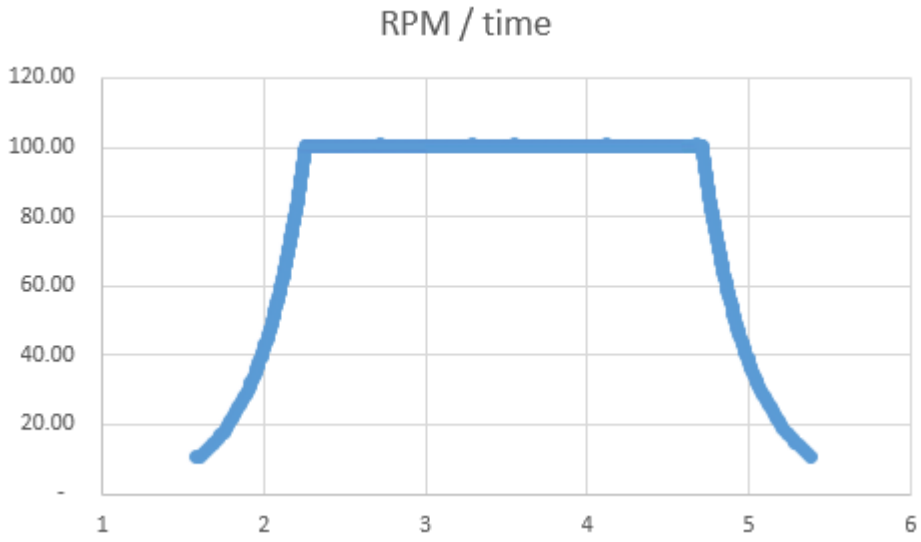
For computation purposes in using the Harmonic curve, n represents velocity in revolutions per minute (RPM), a value understood by many and as you will see, very useful in its range here. One RPM of the ubiquitous 1.8° stepper (200 steps per revolution) results in a cycle time of 300,000 microseconds.
Re-writing the equation we get:

$$cycle = 300{,}000/RPM$$

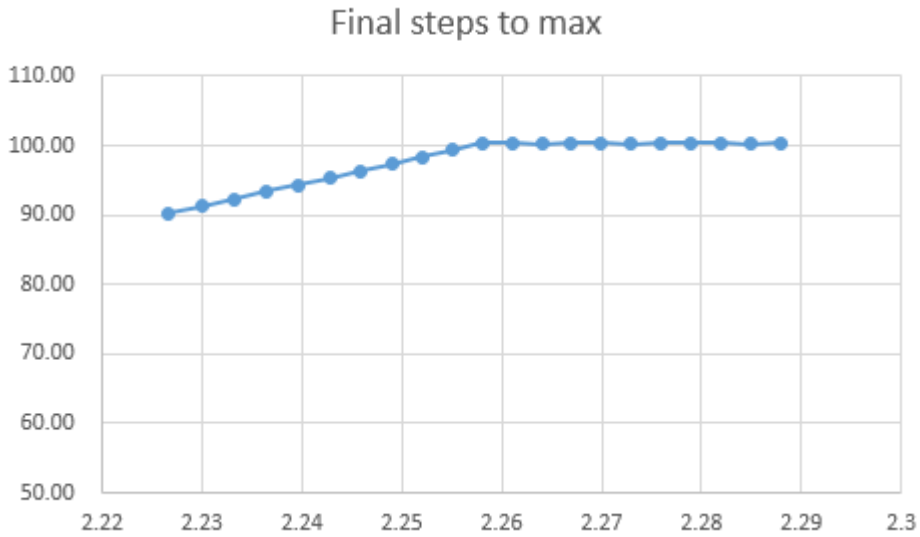
This indeed, lends itself naturally to stepper motor acceleration as a low speed, when torque is greatest, the incremental steps are the largest, thereby passing through the speed ramp quickly. As velocity increases and the pull-in torque curve becomes an issue, the incremental steps are much kinder. In fact, it was using a Harmonic acceleration profile that we obtained our fastest ramp to date, achieving in excess of 2,400 RPM with a NEMA23 motor, 24V 0.6A in less than 1.5 seconds.

From a profile planner's perspective the Harmonic curve is also quite easy to work with. The acceleration from Va to Vb is going to take (Vb – Va) steps (divided by the acceleration factor). In most of our applications we leave the Harmonic acceleration at the default of 1.0 so a path from slow start of 10RPM to 100RPM will take 90 steps. Attaining higher speeds will require a much flatter curve but this can be achieved is steps.

One note to take away from this early discussion is a reminder that all internal calculations are performed with the cycle value in microseconds. For convenience the option exists to enter velocity values in RPM. This however, should only be used for speeds up to 600RPM. Beyond this 'magic' number, velocity values would better be entered in terms of their cycle times as the RPM resolution falls off pretty quick.



This is a plot of scoped data from the harmonic curve, constant velocity increment/decrement ramps.



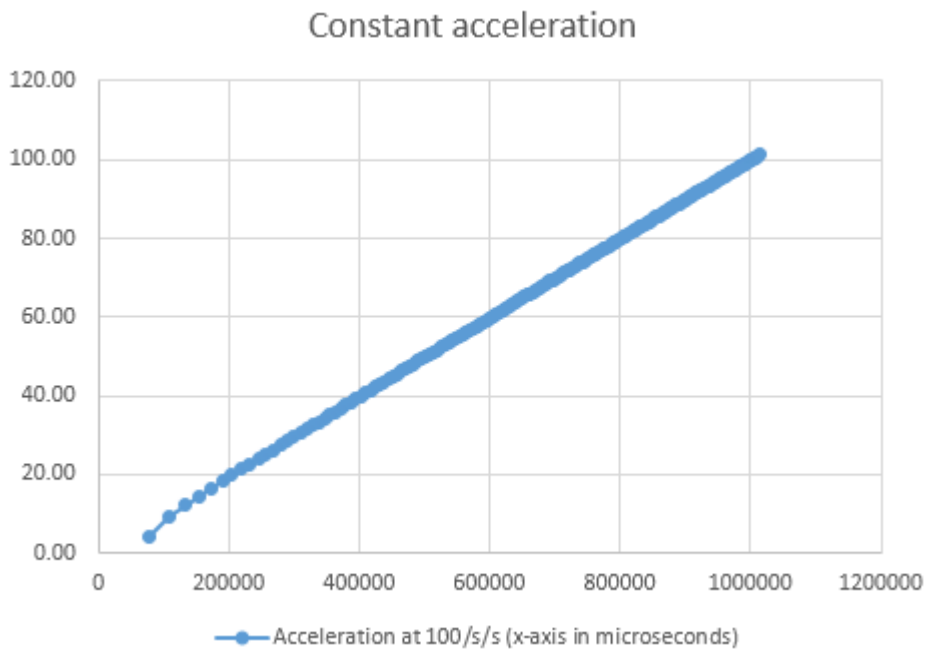
A zoom in to the last steps before target shows that the transition involves a virtually unnoticeable discontinuity.

There are those who tout the constant acceleration curve as the preferred implementation. As a straight-line accelerator you end up with wildly divergent behavior at the 'ends'. To achieve any reasonable speed within a reasonable period of time you must leap away from the start. As an example. To reach 300RPM inside one revolution (200 steps), you will already have reached 100RPM after 22 steps. The same acceleration that will take you to 100RPM in one revolution will take more than 9 to reach 300. One advantage to this algorithm however, is that since you are dealing with adjustments to the timer counter directly, you are already at the highest resolution available for precision in speed control. The suite of formulae discussed by David Austin in 2004 and the application note AVR446 released by Atmel later, provide for some capability much more difficult to deal with in a variable acceleration model.

The undeniable advantage in using the linear algorithm however, is in the case of synchronizing the motion of multiple motors. If you fast-forward to the discussion of Bresenham's algorithm, you will note why we chose not to use it. Consistent with this decision is the need for multiple axes to achieve their respective targets as close to the specified time as possible (bearing in mind we're still dealing with discreet values in an analog application). The linear algorithm defines its acceleration as radians per second per second. One radian per second is approximately 9.55RPM. 100RPM is about 10.47 rad/sec (10.47 is the default acceleration value when in linear programming mode) so an acceleration of 10.47 suggests that from a zero start we will achieve 100RPM in one second. Now suppose this is the x-axis and to describe the line we want

at say 30 degrees, we need to move the y-axis at sine(30) times the velocity of the x-axis. This is fine once we're at cruising speed but the ramp up to the target will describe a portion of the line as well and we need to ensure that this is straight and at 30 degrees also. With the linear algorithm, we simple multiply the acceleration by sine(30) and we achieve the y-axis target velocity at the same time as the x-axis achieves its target, thereby ensuring perfection, something not possible when using the Bresenham algorithm (used by most).

We will drift back in and out in our discussion of motion, to the acceleration and deceleration profiles. It is the singular event which differentiates controllers. Getting a grip on it early is useful. Download the discussion papers listed in the references for a more in depth description as well as some useful formulae.



Velocity profile of a constant acceleration curve – Taylor series approximation

Velocity profiles for stepper are time based rather than event based, hence the direct manipulation of the timer counter values allows for more precise control over ramp characteristics. The most immediate example of this is in syncing multiple axes to achieve their target velocities at the same time. With the constant acceleration model you deal with a straight-line function, the Harmonic curve is a polynomial. So with all this in mind, SAC offers up both. For general control, easing away from stop and quick rise to target, Harmonic curve. For acceleration sync, the linear algorithm. One final note regarding the linear algorithm is that, all else aside, it takes a bit better that 52 microseconds to compute, setting its limit to about 300RPM with 16x microstepping (see microstep).

There are times when a combination may be the best solution and a flag is used in the command queue to differentiate between profiles. When choosing a command to xRef, or when looping, the motion engine automatically switches plans depending on the command. When you look at the queue (below) you can see which commands use which plan.

```
SAC>m2000u6t50000
SAC>q
Q       dir     RPM     cycle   steps   acc     dec     pulse   DeltaT
=============================================================================
1       CW      100     3000    200     1.00    1.00    0       0
2       CW      5       60000   200     1.00    1.00    0       0
3   +   CW      5       60000   200     1.00    1.00    0       0
4   +   CW      8       37500   200     1.00    1.00    0       0
5   +   CW      9       33333   200     1.00    1.00    0       0
6   +   CW      10      30000   200     1.00    1.00    0       0
7   +   CW      11      27272   200     1.00    1.00    0       0
8   +   CW      20      15000   200     1.00    1.00    0       0
9  *+   CW      20      15000   4000    10.47   10.47   0       0
10 *+   CW      120     2500    4000    10.47   10.47   0       0
11 *+   CW      20      15000   400     10.47   10.47   0       0
12 *+   CCW     20      15000   400     10.47   10.47   0       0
13      CW      200     1500    200     0.20    0.20    0       0
14      CW      200     1500    2000    0.20    0.20    0       0
15      CCW     200     1500    2000    0.20    0.20    0       0
16      CW      200     1500    2000    0.20    0.20    6       50000
17      CW      100     3000    200     1.00    1.00    0       0
18      CW      100     3000    200     1.00    1.00    0       0
19      CW      100     3000    200     1.00    1.00    0       0
20      CW      100     3000    200     1.00    1.00    0       0
=============================================================================
Q is at 17      * linear OFF    + pulse OFF
SAC>
```

Command queue

The example of a command queue shown above with the last command to have been entered, m2000u6t50000. This command is in queue position 16 and inherits velocity, acceleration, deceleration and ramp from the previous command as they were omitted therefore defaulted.

Jerk – the change of acceleration over time. Constant acceleration and constant velocity typically translate into smooth motion. There are however, two points along this curve (when plotted against time) when this can never be the case: at the beginning and the end of acceleration, be it positive or negative (deceleration). At start-up, acceleration must go from 0 to whatever and then return to 0 once the target velocity has been reached. The amount of jerk will have long term negative effects on equipment, materials and passengers. Gentle transitions can minimize these and prolong system life.

Infinite Acceleration – the change of velocity from zero. Further to the discussion of jerk, as acceleration is a measure of the change of velocity over time and, mathematically, anything divided by zero is infinity, when begin any motion we experience infinite acceleration. In theory our initial speed, regardless of how small it may be, is achieved instantaneously – zero time – therefore infinite acceleration. In reality though, there is a small amount of time and while the acceleration may not be infinite it is very large. All motors handle infinite acceleration as they must. Some get off the mark, others spin their wheels. Under a variety of different loading scenarios, the same motor will likely demonstrate a variety of performances. These are difficult to formulate and empirical data is typically used for practical planning purposes. Suffice it to say that there is a speed (V' – V prime) until which we can safely accelerate from zero. Beyond that we spin our wheels or in the case of stepper motors, stall.

Slow speed – a transition point for timing. The action of a stepper motor is the result of receiving signals from a driver alternately charging coils. The rate at which the coils are charged/discharged determines the velocity. The driver uses a pulse stream supplied by the controller for this operation. The controller's primary function is to provide this stream and therefore calculate and time the delays between transmission of pulses. If we take a quick look at the benchmark speed of 1RPM with a 200 step per revolution motor, we calculate that a pulse must be generated every 300 milliseconds to maintain this. As speed increases, this timing cycle becomes shorter, at 100RPM it's 3 milliseconds and at 1,000RPM it's 300 microseconds. Once we include microstepping it becomes shorter still. Bottom line here is that we set up timing such that we can accommodate as wide a range of velocities as possible. A physical/electrical limit with modern drivers is that they like to see a triggering pulse for at least 1 microsecond followed by a rest of another microsecond, that's a 2 microsecond period or a frequency of 500,000Hz. So timers are set to count in microseconds using a 2MHz clock. With a 16-bit timer, the maximum count is 32,768 microseconds which, as a delay, equates to about 9RPM. We set our slow speed at 10. What that means simply is that when speeds are 10RPM or less they are no longer controlled by the system timer but instead are polled by an onboard function and thus handled internally, differently. What needs to be known by the designer is that in slow speed ranges there are no acceleration/deceleration ramps, all are infinite. Slow speed is also the default stopping speed. All deceleration to stop will terminate at 10RPM. Additionally, slow speed is the change of direction speed. To transition from clockwise to counter-clockwise motion (or vice-versa) the velocity curve will pass through slow speed. This can be programmed out but in doing so, consider jerk.

Timing

A note on timing. All oscillators are made to a specification that will typically include a +/- tolerance. What this means is that there is never a guarantee that one second will be exactly 1 million ticks of a 1MHz signal. This also varies according to environment. Temperature for example, can have a dramatic effect on oscillators. There are several options available to the operator to fine tune the timing with SAC. Two persistent parameters, intohead and xclk allow for the tweaking of the cycle time for overall system timing. What needs reminding is that the cycle time is calculated from the velocity with the formula 300,000/RPM. This would indicate the 60RPM has a cycle time of 5,000 us and is in fact the value that will be used. 59RPM has a cycle time of 5,084 us and 61RPM, 4,918 us. With all due consideration to factors that can cause a variance, somewhere between a value of 4,918 and 5,084 you will realize a velocity of 60RPM. Further, remember that the velocity is based on the cycle time, NOT the speed entered in RPM. So for precise control, on-site testing will reveal what this value is and then the onus is on the profile planner to program the velocity not with RPM, but by using the direct entry of (V/c)(C/c)nnn which records the velocity as a cycle delay and not RPM. One entry will cause the other to be calculated so bear in mind that the LAST update to be made is the one that will be used.

Profile Planning

Remember this – THERE ARE NO DECELERATION CURVES, except for stopping. All profile planning will consist of describing motion that starts, transitions and eventually stops. The start and the transitions are handled by acceleration, be it positive or negative. As a planner it is vital to know how many steps will be required to execute each command. In a real-time environment, without necessarily the benefit of knowing what the next command might be, the only way of calculating this is with a run to stop. That is, a defined acceleration, a cruise state followed by a ramp to zero or some other terminating velocity. This is the easiest method for profiling and with fast or no accel/decel ramps, is probably okay, notwithstanding the wear and tear caused by jerk. In other cases you'll want a smooth transition from one state to the next. In these cases, the transition, or ramp, from the current velocity to the next velocity will be treated as an acceleration to a new target velocity. The number of steps required to make the transition are readily available to the planner via the calculator.

Getting back to the no deceleration curves, it does mean that if you want a graceful transition from V1 to V2, decel V1 with 999 and V2 will adjust for the starting velocity and accelerate from there. This however, fails if V2 is below SLOW SPEED as there is no accel or decel within that range. For a graceful transition, program V1 with the desired ramp as the decel-to-stop will run down to SLOW SPEED and V2 can pick up from there.

Communications

Before getting into detail, first a general note about communications. The serial transmission of data is a very time and compute intensive function for the duration of the operation. At the preferred rate of 38,400 bits per second (0.2% error), transmission is every 26 microseconds. That's 234 us per character, or the equivalent of about 3500 CPU instructions. As you can readily surmise, it doesn't take much before the processor can be overcome by handling communication requests instead of performing other functions. Now the implementation of these links in SAC are such that normal operational activities will take precedence, however this means that attempts to communicate may result in lost data or corrupt packets, forcing a resend thereby adding to the problem. It is with these understandings in mind that consideration for operational integrity take priority over unnecessary traffic.

> **NOTE:** With the exception of the pulse commands (+/-) all terminal commands to SAC must end with a CR and/or LF character(s). Failure in this will result in the system waiting...

SAC supports three modes of communications: RS485, USB and TTL serial. The RS485 link is the intelligent network connection and will packetize data with a header, footer and CRC using the Dallas/Maxim inverse polynomial. The USB and serial links are for 'dumb' terminals that have no mechanism for entering non-printable characters. These terminals can still communicate across the network at the expense of no error checking.

By default, SAC is configured for a USB console. This will provide terminal connectivity via the USB port or the TTL level Tx/Rx screw posts. The auxiliary port then becomes the network connection in a multi-SAC installation. Note here that there are completely different command responses based on the connection model as full screen displays will not be passed across the command network. Alternatively, SAC can be configured to provide terminal access via the auxiliary port in which case network traffic would default to the USB/TTL connection. Which configuration to employ is up to the individual. Terminal traffic is likely to be heavier than network traffic and would therefore be better handled by the USB/TTY port. Either port or both will accommodate a RS485 connection and there is a separate connection for transmit enable that will be used by both. Bear in mind that RS485 is implemented as half duplex.

SAC, by design, generates NO network traffic. Operational considerations dictate that this policy be adopted by the operator as well. Both networking modes will accept command traffic from a central controller (mother or other) and respond simply with ACK (ASCII 6) if the CRC checks. Again, the onus is on the operator to ensure that the data sent is useful.

Before we begin this discussion remember, no SAC command begins with a number. Numbers at the beginning of a command (byte values between 0 and 255) are network node addresses and will be stripped off the message and placed in the packet header. Any number at the start of a message will be parsed as a network address. Be sure of what you send. Having said that, read on.

The native USB port is primarily for system uploading. It can however, be used as a fully functional control console for SAC. In network mode however, it's hold on the hardware serial port is relinquished to the RS485 network. When operating as a stand-alone controller the node addresses have no effect and should be avoided as the operator console is obviously the sole destination and other messages will be ignore.

In all network modes, the zero address references the general call or broadcast method such that every node connected will receive a copy. Commands are issued as nnn(M/m)xxx as an example where nnn is the

network node address, M/m or other is the SAC command followed by its parameter. When nnn is omitted, it becomes a local command in stand-alone operations.

All network transmissions are packetized as per the following:

- Byte 0 - STX (ASCII 2)
- Byte 1 – Destination address
- Byte 2 – Origin address
- Byte 3 – data
- …
- Byte n – data
- Byte n+1 – ETX (ASCII 3)
- Byte n+2 – CRC*
- Byte n+3 - \0 (ASCII 0)

*CRC can be switched off if the overhead in calculation/transmission becomes an issue

Additionally, all packets are acknowledge either with ACK (ASCII 6) or NAK (ASCII 21) which also may be switched off.

Please consult with your communications engineer or representative for assistance with network design.

Under normal conditions, with SAC operating as a stand-alone controller connected to a nearby PC or other smart device (mother?), serial communications over the USB port have been tested to a baud rate of 250,000 with no data loss. Your results may vary.

It is worth mentioning that both communication port terminal posts are exposing TTL level transmit and receive signals and could therefore connect to any network infrastructure with the appropriate adapter. The interface marked *term* is connected to a hardware UART which can handle the higher speeds and data throughput. The *ntwk* interface is implemented in software with SAC and is limited to 38,400 baud.

Recalling that prefixing a command with a number will result in its interpretation as a network address, separating that address from the text will indicate that what follows is merely a message, not a command. So where 85m3000a2d1 would be a motion command for 3000 CW steps with accel 2 and decel 1, sent over the network to node 85, 85 hello is simply a text message. The same can be accomplished with a dot (period), whichever is most convenient. In the same vein of convenience, the backslash (/) will automatically direct the subsequent text to be sent as a response to the originator of the previously received message, unless it was a general call. This 'last message received' address is preserved until changed by a message from a different source. The space/dot requirement still applies.

Final reminder, the port options with SAC allow for either to be the network connection with the other as a terminal. The network connection is presumed to be a multi-drop RS485 network and as such the data is packetized with CRC and ACK/NAK. The terminal port can use any media/protocol capable of handling TTL level RxTx. There is a transmit enable signal available on a terminal post (see hookup diagram) that is active for both ports. All Rx connections are biased with internal pull-up resistors.

- **RS485** default network
  At present SAC is primarily interested in networking via a twisted pair, multi-drop, peer-to-peer RS485 network using a custom packet protocol. This can be established over the hardware or software serial port. The default baud rate for both ports is 38400 and the software port should not exceed this. The hardware port however, has performed satisfactorily over short distances at rates as high as 1 Mbps.

- **BTLE**
  At the time of this writing, BTLE mesh networks are still in their infancy. We will continue to monitor development in that arena with an eye to incorporate any useful functionality in futures software release. The network connectivity provided by SAC is TTL Rx/Tx so the media is not a hindrance. At present, our experience with Bluetooth does not encourage us to recommend its use in any electrically noisy environment.

  Presently however, with the addition of a blue tooth module, any blue tooth enabled device (Android smartphone, portable PCs, etc) can easily connect as the terminal device.

- **LoRa**
  As with BTLE, we watch and wait. The flexibility built in to SAC communications was in preparation for future advancements. Of all the wireless technologies, we believe LoRa to be key in future wide-area control networks.

- **VHF/UHF/2.4GHz**
  As with the other wireless technologies, we are continuing research. The use of these bands, unlicensed, severely restricts their range at any other than extremely low baud rates. With any of these technologies however, their implementation is generally just a matter of a new software release.

Functional Specifications

Note: In all command discussions, most can be invoked using either the upper or lower case and are denoted as (A/a). A command reference such as (A/a)(B/b) implies that AB, Ab, aB and aa are all valid sequences. If the command is case-sensitive (most system commands) the option will not be noted. There are no spaces in any of the command sequences, contrarily a space designates the end of the command so anything after is simply ignored.

The three primary arguments to any motion (aside from the number of steps) are velocity, acceleration and deceleration, a total of 32 characters and 14 syllables. When discussing them collectively we'll use VAD.

Digging up the Greek symbol mu to represent micros is time consuming and doesn't work with displays anyhow, so convention has it that we use u instead. So us does not refer to you and I and others, it means microseconds.

- ### Profiles

  When planning a profile there are many behavioral characteristics to be considered. The following represents a notebook of ideas being put to paper as the thoughts occur.

  GAP testing can be very confusing at the beginning, and clears up only a little further on. For many it may be easier to turn it off and leave it off. For the foolhardy, it provides a shortcut to profile planning that can ease that burden. Think of it essentially as keep your foot on the gas for a bit longer than the end of the last command just in case. If a new command comes in during this brief period (determined by the cycle time of the last pulse generated) it is assumed that the commands are all part of the same profile. Without GAP testing, the 'current velocity' would have to be reset. When the calculations are performed to determine the number of steps required to achieve a target velocity and that target velocity happens to be the current velocity because we are still within the GAP, there is zero acceleration.

  Now if you want to return to start without changing out the basic constructs of your profile, adjust the deceleration to get at least one step. If decel is at 999, it is a trigger to stop cruising with noe decel thereby leaving the current velocity intact. To get at least one decel step you need to plan for a value other then 999.

  Under normal circumstances, the accel and decel ramps when using the Harmonic curve and values of 1.0, are equal in steps to the difference in velocities. To achieve zero steps you simply need to see that the value equals or exceeds that number. When dealing with truncated profiles (profiles where there are not enough steps in the plan to achieve the target velocity), this value needs to be adjusted by the accel value if we're programming decel or decel for accel. Confused yet? Try this. You plan motion for 200 steps at 100RPM from a standing start (10RPM) with accel and decel values defaulted at 1.0. So that's 100 – 10 = 90 steps for accel and 90 again for decel making the profile 90-20-90 = 200 steps (accel-cruise-decel). Now lets say we want to slow the accel ramp down to 0.5. Our calculation going in says we need (100 – 10) / 0.5 = 180 steps to accelerate then after cruise, 90 steps (unchanged from the first) to decelerate. This is a total of 270 steps. Whoops, we only want to move for 200 though. Enter the truncation zone! Once it has been determined that we need help, we do a quick truncation calculation by normalizing the ramps according to the step total. So 200 steps is the total and 270 is required for ramping. We divided 200 /270 to get 0.74074.. and use this to normalize the ramps, retaining their respective ratios. The accel ramp becomes 133 steps and decel 67, with nothing left over for cruising. Fair enough, steps, a.k.a. rotation, translates to linear movement and precision dictates that we move the exact number of prescribed steps.

  Ok, so we're good so far? Now for the tricky bit. Let's change the profile a bit to have the repeated plan return to zero (10RPM) at the end of each sequence. As we saw from the first, a plan with accel and decel set to 1.0 would give us a 90-20-90 profile. Now however, what we want is a ramp up to

speed and then a restart. This is akin to a sawtooth wave if the accel ramp is too shallow to attain the target velocity. There are actually several ways to do this depending on exactly what you're after. Just remember that with GAP testing, a profile of 200-0-0 will automatically self-adjust to 0-200-0 as the current velocity remains as the target velocity from the previous command and no acceleration is deemed necessary. So to return to zero we need to program a deceleration and we want to do this so it is exactly one and only one step. Going back to the above example, we'll change accel to 0.4 and decel to 999 which results 225 and 0 steps. The truncation for this is simple as it will just cut back to 200-0-0. As we noted however, this will automatically self-adjust to 0-200-0 if we execute the next iteration within the GAP, which we don't want. We want to restart therefore we need that one decel step to bring us back to zero (10RPM). Recall the formula for step calculation, (V2 – V1) / accel (or decel) and remember we're dealing in integer results. Delta V is 90. Divide this by any number between (dV/2) + 1 and V (46 and 90) and you get 1. Anything larger than 90 the answer is zero, 45 or less and the answer's 2 plus.

That only works however, in normal operations. In the truncation zone we need to tread a bit differently. Let's look at the normalizing calculation again:

(steps_acc + steps_dec) / steps_tot

If we actually use one step and then normalize it, it will instantly become zero as integer math truncates. We need a value that normalizes to 1. Without going through the math from first principles, if we multiply the accel factor by the target velocity, we get the decel value that will result in exactly one step. As there is a range of values that will give the same results, avoid using those that are right at the border. Remember the old saying, “Two plus two equals five, for large enough values of two.”

What has been detailed above pertain to the Harmonic curve. The procedure is similar when using the linear ramp except calculations all assume a zero start so to get the number of steps between V1 and V2 you must calculate the steps from zero to each and then subtract. The formula for the number of steps to attain velocity V using the linear ramp is:

steps = (V * V) / (2 * alpha * acceleration)

where alpha is the step angle expressed in radians (0.0314159 for a 1.8° motor).

The internal buffer in SAC will hold 30 commands. You'd be surprised what can be done with a few commands but remember what SAC is and isn't. It is a controller. It does all the heavy lifting towards having a stepper motor do when we tell it to and when. It is not a command unit. It is intended as the missing link between the intelligence and the motor and really wants to sit there and process a stream of commands. The buffer exists to allow for burst transmissions so as no to tie up either end unnecessarily. To aid in the synchronization of command there are hardware and software flags signaling the beginning and ending of each command. See ops/soft_ops for details.

- **CLI**

  The command line instruction is the user interface to SAC. There are four types of commands available to the operator (see Appendix C for a summary of all commands):

  - Motion
  - Unary
  - Line
  - System

Motion commands all begin with the letter ‘m’, upper case or lower case, referred to in this document as (M/m) – either/or. Following this will be 0 to many qualifiers. All commands beginning with the letter M/m will cause motion

Unary commands are commands that can be entered singularly and span the entire range of commands. Unary commands may or not effect motion, mostly they alter the parameters that will effect motion.

Line commands are the catch-all for everything else that’s not a system command. These range from displaying the command buffer, to showing the current motor speed. Some line commands have options and are referenced individually, while other are in the collection preceded by (C/c).

Systems commands (prefaced by ‘$’) are, in general, state-altering commands. What this really means is that you’ll need a cold or warm reboot for most of these to take effect. Most of them deal with the persistent parameters stored in EEPROM, but there are reset and restore commands, commands to change the ramp profile, set the system to sync, etc. Again, see Appendix C for a summary or the individual commands for detail.

- **Cycle processing**

  The timing of pulses to the stepper motor regulates the velocity at which it rotates. This rotational velocity in turn quite often, drives a lead screw or some other device that will make the conversion to linear motion. Regardless, it will be necessary from time to time to have as precise a control over the velocity as possible. All else aside, this resolution is constrained by the delays inherent in electronic switching. In the case of the more modern drivers this is one microsecond.

  A simple formula worth remembering is that with a typical 1.8 degree motor (200 steps per revolution), a rotational velocity of 1RPM requires a pulse every 300,000 microseconds (60/200). Thus for any speed given in RPM simply divide it into 300,000 and you get your required period in microseconds (i.e. 100RPM = 300,000/100 = 3,000 microseconds). The only problem with this is when you examine the difference between say 100RPM (3000us) and 101RPM (2970us integer math), you readily notice the lost granularity of 30us. No so as reality would have it. The use of RPM or radians/second or inches per minute are all conveniences for humans. The bare metal approach deals strictly with microseconds and this can be programmed directly. When entering a velocity command, instead of v100 denoting 100RPM, use vc3000 which signifies you want velocity recorded as a cycle period of 3000 or whatever value. This is as fine as it gets. It applies only to the cruising speed as the accel/decel ramps must be calculated separately.

- **Accel/decel**

  The shortcut keys work at the command line in either ramp plan and reflect the defaults that have been configured. The factory settings are 1.0 for the S-curve and 10.47 for the linear ramp so entering (A/a) with no argument at the command line will set the session default for acceleration to either 1.0 or 10.47, depending on the current ramp mode. The decel shortcut (D/d) works the same. Bear in mind that these default values of 1.0 and 10.47 can be changed or restored.

- **Command buffer**

  The command buffer is a circular FIFO buffer that will hold up to nn motion definitions. Almost all motion originated by SAC comes from the command buffer. The exceptions to this are those commands which do not require any specific detail such as the single pulse command +/-. All looping and referencing relate to positions in the command queue. Typing (Q/q) at the SAC prompt will display the command queue.

- **Queue**

  The queue displays the contents of the command buffer. Its pointers indicate the current command being executed and the next available position. The queue is seeded upon a cold start with the last preserved values as written to EEPROM with the Q save (Q/q)(S/s) command. This does not indicate however, that they are in the process chain. The entries can be used in any xRef command or in loops. The greatest value of the queue is that the individual entries can be edited. At any time the queue can be reset (Q/q)(R/r) in which it will, as in the case of a restart or cold boot, be restored to the last saved snapshot of values. Alternately, the queue can be cleared (Q/q)(C/c) which will set every entry to the session defaults. Clearing the queue has no effect on the values saved in EEPROM which can yet be restored with Q reset.

```
SAC>q
Q       dir     RPM     cycle   steps   acc     dec     pulse   DeltaT
=================================================================
1       CW      120     2500    600     1.00    1.00    0       0
2    s  CCW     120     2500    800     2.00    2.00    0       0
3       CW      60      5000    1200    2.00    2.00    0       0
4  * s  CW      60      5000    1000    10.47   10.47   0       0
5  *+   CW      60      5000    1000    10.47   10.47   0       0
6  *+s  CW      60      5000    2000    10.47   10.47   0       0
7  *+   CW      80      3750    600     10.47   10.47   0       0
8   +   CCW     140     2142    1200    1.00    1.00    0       0
9       CW      140     2142    1200    1.00    1.00    0       0
10      CW      20      15000   400     1.00    1.00    0       0
11      CW      100     3000    200     1.00    1.00    0       0
12      CW      100     3000    200     1.00    1.00    0       0
=================================================================
Q is at 11      * linear OFF    + pulse OFF     s sync
SAC>
```

  The above image shows another view of the queue with the various parameters and flags, each of which is editable.

  If you enter a motion command it will be stored in the next available position and a flag will be raised indicating that there is a pending command it which case it will automatically be processed in order. When the last command entered has been processed, motion will stop. At any time the queue can be displayed by entering the CLI (Q/q) and as suggested earlier, commands can be recalled via the xRef command (see xRef for details).

  By entering the command (Q/q)(P/p) you enter into the queue program mode where each entry can be edited, without consequence, and all parameters can be altered. In much the same manner by which commands are entered at the command line, the queue entries, in program mode, are changed by prefixing the attribute with the queue position, thus a command of 8m2000 will change the total number of steps for queue position 8 to 2000. These are full steps. If in microstep mode these steps will automatically be converted to as many pulses, regardless of whether the command is set for full step or pulse. If the (P/p) qualifier is added to the command, the value entered will be in pulses and you will see the flag '+' indicating such.

  The queue is a very powerful process, exposing all the parametric attributes of the motion command. It is perhaps the most used tool in profile planning so an intimate familiarity with it is advised.

  (X/x) will exit the program mode and return to the SAC prompt. From here, individual commands can still be edited by via the CLI (Q/q)nn... where nn represents the queue position followed by whichever parameter values are to be changed. In theory, continuous motion could be described

simply by continually editing the single command. If you program a loop, any change will be reflected as soon as the pointer reaches that position again. This is most useful when set in continuous mode and is the basis for using a quadrature encoder as a throttle. The only change which requires stopping and a reset of course, is if you alter the microstep value.

If a queue position is programmed with a deceleration of 999 (zero ramp) and then selected a the only entry in a xRef loop, it effectively becomes a continuous motion command. The difference being of course, is that the parameters can be changed whilst in motion. Caution must be exercised however, if you change for CW to CCW. With no deceleration and the absolute velocity unchanged, there will be no acceleration calculated either, simply a change of direction will be executed. This will happen immediately upon completion of the current cycle and may result in the motor stalling if the speed is too great (hence the earlier discussion pertaining to ½ V'). The least that will occur is significant jerk that may not be desirable.

Spend some time with the queue and get comfortable with it. You will quickly ascertain its value. Refresh your understanding of the potential communication bottlenecks and you will see that by utilizing a command queue structure that is already in place, the commands transmitted to SAC can be reduced to just a few characters thereby minimizing overhead.

- **Throw-away**
  There are two hidden command positions in the queue , position zero and position COMBUFLGTH+1. Position zero holds the detail for the continuous command. It cannot be used in a loop or programmed directly. When you see it you'll understand what it is and then you can ignore it. COMBUFLGTH is the length of the command queue, at this writing it is 30. Position 31 is used as a one-shot throw-away command that is used internally to execute commands that you don't want added to the queue. Any motion command can utilize position 31 by prefixing the CLI instruction with X/x. If there are any parameters not specified, the session defaults will be used. Qualification of any parameter will not however, alter the session defaults. If you look at the companion app, there is a list for programming 'favourite' commands and these are typically one-shot.

- **EEPROM**
  The on-board EEPROM holds the values of all the persistent parameters and will retain that data even in the event of power loss. When configuring SAC, either through the menu or direct command line instructions, the values are updated directly into EEPROM.

- **System defaults**
  There are factory settings for all the configurable parameters. They are hard coded and cannot be changed. They are however, reflected in EEPROM and these are the values loaded as system defaults at startup or reset. They can be modified through the configure command and will persist until reconfigured. As an option, they can also be restored to the factory settings.

- **Session defaults**
  As command processing progresses, parameter values get changed depending on the profile. These changes generally remain in effect until the next command which may or not alter them. These are the session defaults. Entering (C/c)(F/f) at the command line will display the values currently in uses. Most of these can be changed via unary commands or through the motion command line. Refer to the individual command for detail.

- **Continuous**
  Think of continuous as a fan control. At a given speed the motor will rotate until instructed to stop. In this mode the encoder functions as a throttle. See encoder for more detail.

- **Loop**
  Looping can be accomplished in two ways. The loop start and loop finish commands each require queue positions as parameters. Once engaged, the command processor will cycle from start to finish, increasing the queue position with each command completion. If the finish position is lower in the queue than the start, the processor will wrap around to one and continue counting up. When the finish position is reached, a jump is made to the start position and the process repeats until the number of loops specified is complete. In the case where there has been no number of loops specified, looping will continue until a command in received to stop.

  Alternately, one can use the xReference array. This is a one dimensional array holding as values, queue positions. It is a zero-indexed array hence its kick-off is at xRef[0]. The entries can be any valid queue position, in any order and/or number. The array terminator must be null. When the loop is initiated, either with a specified number of iterations or continuously, it will execute to command referenced is its [0] position, proceeding to the command in [1] and so on until it reaches the null entry at which point it will restart from [0].

- **Autorun**
  The autorun parameter is only accessible through the configure screen or the CLI $(C/c)(A/a). It has a value that is either zero (factory set) or non-zero between one and the maximum length of the command queue. If the value is non-zero, SAC will execute two commands on startup. The first is ls1 which indicates the beginning of a loop at queue position 1. The second is lsn where n is the non-zero value of autorun and defines the end of the loop. This provision allows for the programming of the queue and a setup whereby SAC can operated unattended. One word of caution, there is no memory of the physical position of the motor. If the previous session was terminated without consideration of an automatic restart (i.e. power loss), such may initiate motion that may exceed physical travel limits. The onus is on the operator to ensure that all physical plant is in an appropriate 'home' position prior to a start or restart.

- **Single pulse**
  Single pulses (not to be confused with single steps) are useful in precision work and can be stacked for very tight control of motion. There is no timing associated with a single pulse, nor does the clock or microstepping come into play in any meaningful way. A single pulse is just that. A single step on the other hand, if the driver were configured for x16 microstepping, would comprise 16 pulses. A single pulse is one. With the encoder switched to follow mode, it causes a single pulse to be generated. In practical application, a 100 pulse quadrature encoder will require two full revolutions to cause a 1.8° motor complete just one. Configured for x16 microstepping would then require 32 complete rotations of the encoder for one motor rotation.

  When not operating in a continuous mode, an attached encoder will generate single pulses. See encoder for more detail.

  Programmatically, SAC will generate single pulses in response to +/-.

- **Micro-stepping**
  Contrary to popular belief (and widespread advertising), microstepping does not increase the resolution of a stepper motor. A stepper motor's resolution is fixed according to the windings and magnets that it's made with and typically are 1.8 degree (200 steps per revolution) or 0.9 degree (400 steps per revolution). This can be changed with gearing, not electronics. The stepper motor can only make stops at full steps. What microstepping does do is ramp the voltages applied to the coils so there is a smoother transition from one step to the next. In practice, this is as good as increased resolution but if your motion, translated from rotational to linear say with a 1mm pitch lead screw and a 1.8 degree stepper, gives you 0.005mm resolution, that's it. You stop on 0.995, 1 or 1.005mm,

there is nothing in between. (Technically there is but the holding torque is so small at the intermediate steps it can't be relied on – get away from the theory and get practical). Motion between the steps however, is much smoother with far less jitter.

The implemented theory of stepper motor motion is the repetitive turning on and off of the current to the coils With microstepping, this current is fractionalized according to where it sits along the sin/cos curve between full steps. If we consider full steps to be at 0 (sin = 0) and 90 (sin = 1) then a half step would be a 45 (sin = cos = 0.707), so to hold a position half-way between steps we apply equal 70.7% current to each coil. One could normalize this, it doesn't matter. Any other microstep value and you begin to see that the current to one coil dominates and it becomes almost impossible to sustain this position with any force. As has been mentioned, microstepping does result in much smoother motion, but the lack of ability to state with certainty that a motion command will hold a stop position in between full steps is counter to the philosophy of a reliable solution. We support the use of microstepping for motion, but not in determining the stopping of a command. Commands within SAC all proceed to the full stop position. (There is of course a hidden command to override this).

Furthermore, our work with a variety of motors and drivers has resulted in our use of microstepping to be limited to x2. We have found no benefit in granularity greater than that. For increased resolution we choose gearing or alternate translations thereby maintaining, or even increasing applied torque. Each implementation may require a unique solution.

The biggest issue with microstepping is the consumption of bandwidth, which has just increased by the microstepping factor. Under full step a 1.8 degree motor requires a pulse every 1000 microseconds to sustain a rotational velocity of 300RPM. At 16x microstepping, this period is now 62.5 microseconds, creating a very small window for other activity. A couple things to consider: microstepping was introduced to smooth out motion at slow speeds and full torque is needed to attain higher speeds anyhow. SAC has a configurable option called auto-micro that will utilize microstepping much like an automobiles transmission. At slow speeds, full microstep, at high speed – full step. This accomplishes three things: 1) at low speeds you get the benefit of smoother motion, 2) at high speeds you get full available torque and 3) at all speeds you keep the cycle time in a window that pleases the processor.

- ## SAC in a multi-axis environment

One of the motives in developing SAC was to get away from that which compromises most multi-axis controllers – a single clock.

In 1962 a gentleman by the name of Jack Bresenham, then working at IBM, developed an algorithm to determine which points of an n-dimensional raster should be selected in order to form a close approximation to a straight line between two points. This is known as Bresenham's Algorithm, the method used by virtually every multi-axis controller to issue pulses to stepper motor drivers.

In a fixed grid it is necessary to decide whether to place a pixel on this line or wait until the next. In this manner, close examination will reveal that lines on displays are staircased. The minor axis follows the lead axis. As can be seen in Figure 1, the minor axis, shown here as Channel 0, is tethered to the lead axis, Channel 1, and as such must skip a step periodically to compensate for different axis speeds which equates to distance over time. Both examples depict a transit of (240,200) so there is a 6:5 step ratio. Bresenham's algorithm will have these lock-stepped to the clock of the lead axis as can is shown is the display.

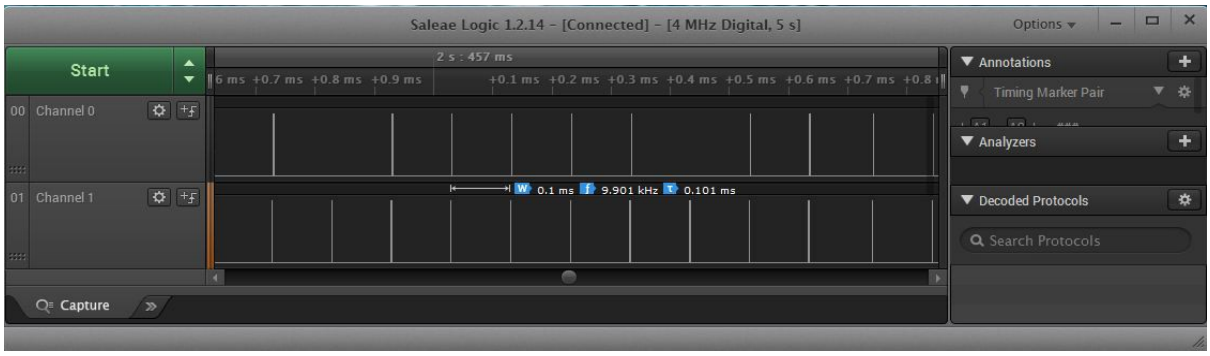


Figure 1 - Bresenham's Algorithm

We are not constrained to a fixed raster when driving axis motors. While their respective motion is dependent on each other, it is a time dependency rather than a position dependency.

In Figure 2 we see SAC. Necessitated by the swing to IoT and the need to distribute control across the network, we needed to decouple the dependency of one axis on another. Using independent time-sliced algorithms, we see that we now have two continuous bit streams driving the separate axes. This results in smoother stepping and axis independence.

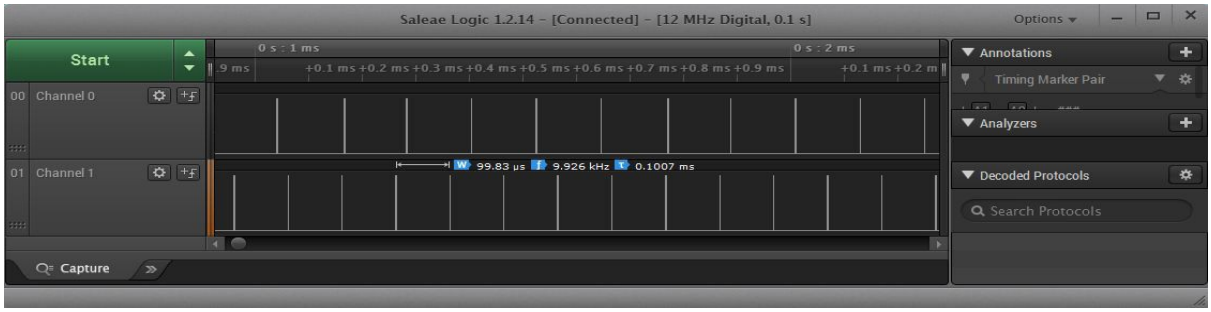


Figure 2 - SAC

Our single axis, TD (time derivative) controllers network via I2C to provide up to 127 axes of control. Incorporating Mother, our hybrid multi-axis controller, leads to an unprecedented 762 degrees of freedom per network.

There's another fundamental issue I've noticed with most OTC G-code interpreters – the tendency to specify the acceleration profile for each axis. In a 3-axis machine (router, mill, etc.) it is useful to be able to define the Z-axis parameters separately as often the characteristic vertical motion of the tool is vastly different than that along the X-Y plane.

Acceleration should be calculated with due consideration to the torque curve(s) of the motor(s) driving the axes. Due consideration must also be given to maintaining sufficient chipload (don't forget that your chips are your greatest heat sink) and moving in a straight line. It's back to a conundrum similar to Bresenham's algorithm. If we maintain a constant acceleration for both the X and Y axes, the only time they are actually in sync is if the toolpath is exactly 45 degrees. The purpose of acceleration, all other considerations aside, is to attain a prescribed (maximum) velocity. To maintain sync, and traverse that straight line, this (maximum) velocity must be achieved simultaneously along both axes. If the same acceleration is used, the minor axis will achieve the desired velocity sooner than the major axis, thereby inscribing a line with a slope dx/dy that is different from the desired plot. We again revisit the alternate philosophies of pulse driven versus time-sliced motion.

mother does several calculations prior to task execution. Considered to be fundamental is the toolpath velocity. In all but straight X or Y transitions, this is a vector product of the two. There is a

cosine function applied to the appropriate axis to adjust it velocity such that the velocity along the toolpath is maintained according to the desired calculation. During this pre-process period, we also describe the acceleration of the minor axis as a function of the major axis (major axis acceleration being defined by the operator) ensuring that both attain the prescribed (maximum) velocity at the same time resulting in true straight lines.

Velocity Sidebar

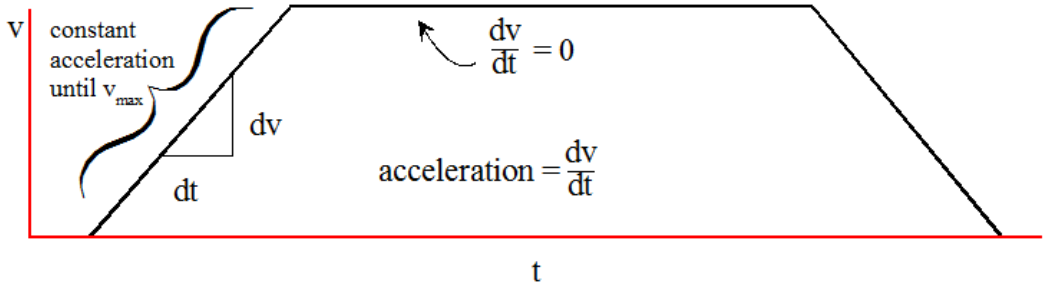
If we specify velocity as a function of the axis, which most G-code interpreters do, we are in fact describing a variable that is the vector product of both axes. Say for example that the velocity is specified as 1 for both the X and Y axis. Now if motion is constrained to either of these only, the toolpath velocity is 1. If however, the toolpath is not an orthogonal motion, the velocity equals the square root of the sum of the squares, so the actual velocity along the toolpath of a 45 degree cut would be 1.414 (the square root of 2). This is great if you want some free speed and the individual axes are already max'ed out. Not so good however, if your velocity was calculated (as it should be) based on the cutting speed of the material and the diameter of the toolbit. Now you're overclocked by 41% and subject to potentially catastrophic failure. What should happen is that the maximum velocity is specified in accordance with proper engineering practice and then the major axis is calculated to be a function of the angle/slope of the toolpath – a simple cosine. The velocity vector along the minor axis is then the sine of the angle or, perhaps mathematically simpler,

$$\sqrt{V_{max}^2 - V_{major}^2}$$

Therefore, if V_{max} is 1, the angle is 60°, V_{major} is V_{max} * cos(60) = 0.866. V_{minor} then is V_{max} * sin(60) or V_{max} * cos(90 – 60) or $\sqrt{V_{max}^2 - V_{major}^2}$ = 0.5 ($0.866^2 + 0.5^2 = 1$).

If we drift back to basic calculus momentarily, we may recall that a derivative defines the slope of a curve while the integral defines the area under the curve. The slope of a curve is better known as m in the equation y = mx + b or (y2-y1)/(x2-x1). Remember too that dx is simply x2-x1 and of course dy, y2-y1. Now in calculus-speak, dx/dt where x is velocity becomes dv/dt which is (v2-v1)/(t2-t1) which is to say the change of velocity with respect to time, otherwise known as acceleration. Similarly, the change of position (dx or dy) with respect to time is what we term velocity.

So, distance is a change in position. The first derivative of position/distance with respect to time is velocity and the first derivative of velocity with respect to time is acceleration.



This is the trapezoidal velocity curve that you see in any discussion about acceleration, and it's perfectly valid. Constant acceleration gives us a nice smooth ride but more importantly it's easy to use in calculations, in knowing how many steps it will take to get up to speed and again back down. This is a calculation that can be done in real-time and results in accurate positioning.

In tests SAC, driving a series 57 motor (that's NEMA 23 for those not yet metricized), microstepped at 16x runs continuously at 33.3kHz with the processor (Atmel MEGA328P) loading at 10.3%. That includes acceleration/deceleration, jerk compensation, on-the-fly speed change, direction change and enhanced g-code processing. Full step speeds up to 2,200 RPM at 24V drawing only 300mA have also been attained under controlled temperatures.

In a single interpreter/controller environment, say 3D printing for example, Bresenham's algorithm is fine. From a business perspective, we are moving into a multi-axis robotic world where processors will outnumber the actuators and giving each its own identity, coupled only by a heartbeat, is paramount. There will be enough data passed among them, no need to add pulse tracking to the chaos.

Command Detail

- ## Motion

The motion commands are the basis for all signal generation. Programmatically, the records in the command queue contain all the information necessary to produce that motion. Those values not supplied directly will use either the session or system defaults.

All motion has been reduced to the specification of the number of full steps and the timing of the delay between steps. The former will translate to distance depending on lead screw, pulley and belt, rack and pinion or how you choose to convert rotary motion to linear, if in fact that is your choice. The latter of course, describes the velocity. The onus is on the profile planner for the calculations and conversions although many front-end applications will carry that load.

Any parameter that can impact motion can be programmed through the motion command. The exceptions to this are the system commands. As has been/will be discussed, system commands often require a reboot and as such are dealt with separately. When entering a motion command it is important to know that, except for the number of steps, all other parameters will become the new session parameters. Thus an entry that changes the acceleration to 1.2 for example, will result in the acceleration being 1.2 for each successive command unless explicitly modified in a new motion command or via the CLI.

Those parameters that can be included in a motion command are:

- number of steps
- direction of motion
- velocity
- acceleration
- deceleration
- ramp profile
- start wait
- sync
- continuously
- pulse plus

Motion commands begin with the letter (M/m) followed immediately (or not) by the minus ('-') sign to indicate counterclockwise motion, the absence of which indicates clockwise motion. This is the minimum command and without further qualification will cause acceleration to the default target speed and continuous motion thereafter. The immediate qualifier would be the number of steps to be taken, either until the next command of the deceleration profile to stop. Repeating, again, there are no deceleration profiles except to stop. Any transitions need to be planned as accelerations with the ensuing commands.

Each command, with the exception of continuous motion commands, will create an entry in the command queue, a FIFO circular buffer that allows stacking commands for use in loops (see looping and xReference). There are several parameters (listed above) associated with each command entry and can be filled with session/system defaults or programmed as well. Each command exists as a unary CLI which will alter the session default but for brevity, can be included in the motion command. So the syntax for motion:

M/m[-][nnn][(V/v)[(C/c)]nnn][(A/a)[nnn]][(D/d)[nnn]][(L/l)(1/0)][(U/u)nn][(T/t)nn][(f/p)]

Each of these is discussed in detail under separate headings, but briefly, square brackets indicate options while parenthesis present an either/or choice. The n's need replacing with appropriate numeric values. Following M/m[-][nnn], the remaining parameters can be listed in any order.

- **Accel**

  Acceleration is perhaps the most important state that the motion engine passes through. Proper planning will result in achievable target speeds with minimal negative impact.

  There are many considerations in planning an acceleration profile but they all come down to time and speed. Most production requirements will not accommodate a leisurely rise to target velocity – time is money. Most machines however, will not start and stop as the Executives would like. All factors which affect acceleration are a complete study unto themselves and will not be covered in this document save to say they should be regarded carefully in your planning.

  Having done all that, there are only a couple of decisions to make: use the harmonic curve or the linear ramp, and what value to assign. A slow, flat profile will generally allow for attaining higher speeds if this is the goal. In light load scenarios, a fast ramp or infinite start may be the solution. With either selection, the lower the parameter value, the slower, flatter the ramp. Be aware that at some point the ramp will be flat enough that with the total number of steps there won't be sufficient time to attain the target velocity. This may also influence planning.

  When calculating acceleration and the step count, understand that the acceleration curve/ramp will complete the total number of steps only if it is starting from stop and the command is not truncated. With the former, transitions are handled automatically from the current velocity to the new target velocity. In certain situations where the two may in fact be the same, there will be no acceleration and the step total will be given over to cruise/decel in its entirety. In the latter case, velocity will increase until such time as the command must be terminated or enter the decel state. In either instance, what is most important is that the step total be preserved.

  For precision, acceleration is a floating point number with two positions available to the right of the decimal. This is for values up to 100 at which point only whole numbers are permitted. Given the nature of the formulae to calculate acceleration values, by that point you would be looking at a near vertical curve. With either solution, entering a value of 999 will trigger a zero ramp scenario and proceed directly to the target velocity.

  Accel is a unary command, a motion parameter as well as a persistent parameter. As a unary command it can be entered at any time without affecting the current operation. This will change the session default which will then be attached to all subsequent commands until changed or reset. A reset will restore the system default which is modifiable through the configure panel of via a system CLI. It can also, as mentioned, be included as a parameter in the motion command in which case it becomes effective immediately for that command as well as the new session default.

  The simple syntax in every case is

  (A/a)[nn[.nn]]

  Absence of any qualifier will return the session default to the system default. As a system CLI, the command is

  $Cdnn.nn

  for the harmonic curve default and

  $Cfnn.nn

  for the linear ramp.

- **Decel**

  From a purely mathematical perspective, deceleration is treated the same as acceleration. The one major difference being is that there is no automatic nature to the decel curve or ramp – it simply exists or it doesn't, either state must be programmed. If there is a non 999 value for deceleration a profile will be generated whereby motion proceeds to stop. The specification of 999 as the parameter value eliminates deceleration and is used thusly for accelerated transitions to subsequent commands. In this latter case the step total will be comprised entirely of acceleration (if any) and/or cruise (run).

  As with accel, decel is unary, system and motion parameter. The unary and motion syntax is:

  (D/d)[nn[.nn]]

  while the system CLI for the harmonic curve is

  $Cenn.nn

  and

  $Cgnn.nn

  for the linear ramp.

- **V**

  V/v is the target velocity. For convenience and familiarity we use RPM. Motion velocity however, is controlled by the frequency of the pulse stream being issued to the driver and this is defined in microseconds. There are simple relationships that exist between all the common variables. One RPM translates to 3.3r Hz requiring a pulse period of 300,000 microseconds. One radian per second is approximately 9.55RPM. 200 pulses per second is 60RPM (full step with a 1.8 degree motor) and so on. Regardless of how you choose to define speed, internal to SAC we deal with microseconds. The speed you enter (in RPM) representing a target velocity is converted immediately to a counter delay. Following acceleration, this value is used to cycle the pulse stream so you can think of speed in any manner you wish, at the bare metal its microseconds. This also is the finest resolution for speed control so being aware of it doesn't hurt.

  Velocity is a unary command as well as a command line option and is utilized simply by preceding the integer value by V or v. As mentioned, this will then be converted and used as the target cycle counter. The rpm value itself is used with the accel/decel ramps to bring the motor speed to within transition range of the target. For more precise control, you can enter to cycle counter directly by preceding a long value with (V/v)(C/c), so 'vc42567' represents at target velocity of 7.0477 RPM. If you list the Q you will see the cycle of 42567 but the RPM will display 7. Remember that RPM is an integer and used only for ramping up to speed. Vc4567 will display 65RPM where v65 will display a cycle of 4615. 4567 is actually 65.689RPM but gets truncated to 65 as an integer. Both are correct. The cycle value is what will be used for the target speed.

  There is a persistent parameter, expressed only as a cycle time, that is used to seed the command queue upon reset (to avoid zero values). This is factory defaulted to 3000. As a system default this can be changed through the configure menu or via the CLI $Cbnnn.

- **Loop**

  Given that commands have streamed in from one of the interface connections, or have been programmed (see Q programming) you have a full command buffer. You can initiate a continuous execution of the saved commands by using the loop function. Loop requires a start and a finish and they can be any valid Q position so long as they're not the same. The function counts positively and

will loop around at the top end of the buffer (it is in fact a circular buffer). If you wish to loop the entire buffer, simply enter any two Q positions such that the finish is one less than the start.

To program a loop, enter (L/l)(S/s)nn where nn is the valid Q position such as LS9. Similarly the finish is entered as (L/l(F/f)nn. To terminate the loop, enter L/l with no qualifier, the loop will end gracefully at the completion of the current command. The loop terminator can be used as well to stop (gracefully) any other continuous motion. Loop can be preempted and subsequently terminated by a continuous command. If however, the continuous command is preempted by an X command (see xRef), completion of this will return control to the loop. X preempting direct will suspend the loop only for the duration of the interrupting command.

The loop may also be triggered to run for only a prescribed number of iterations by setting (L/l) (N/n)xx to some value (xx) other than zero and up to a maximum of 65,535. The counters will reset to zero upon completion or termination of the loop. A point to remember when programming a loop is that one the command processor has a valid start and finish position from the queue (in either order), the loop will begin (notwithstanding being held in check by sync) so a non-continuous loop count, if desired, must be set prior to the second/final command which establishes the loop itself.

Finally, a loop beginning with the start at Q position one can be programmed to run automaically at startup. See autorun for details.

- **xRef**

The xReference command (X/x) allows for the preemptive execution of any command in the command queue. It will terminate a continuous operation but return to a loop at the next command to be executed when interrupted. If however, the continuous command has interrupted another process, be it a loop or regular command processing, execution will resume at the next command from the point of interruption by continuous. See continuous for more information.

In addition to the preemptive command, there is an Xref array accessible through the (X/x)(P/p) command (X program). Any command in the queue may be placed in the array, in any order and as many times as you wish (obviously until the end of the queue). The array is null terminated and once programmed, it can be set to execute in a loop with the command (L/l)nn where nn is the number of iterations from 1 to 65,535. Setting nn to zero will cause a continuous loop to execute.

The operator must ensure that the array is terminated properly for the loop function. A NULL (ASCII 0) needs to be inserted as the final command in the array.

When in program mode, entries are made as nn,xx where nn is the zero-indexed position in the xRef array and xx is the Q position of the command. After each entry the screen display will reflect the array, here showing as well the termination byte.

```
x_ 0 1
x_ 1 3
x_ 2 7
x_ 3 5
x_ 4 0
```

- **Continuous**

Continuous rotation is engaged when a motion command is entered with NULL or zero steps and/or not the minus sign indicating CCW direction. The continuous command is pre-emptive and will terminate a loop upon exit with (S/s). Exiting a loop with (N/n) will cause the loop to commit to one more command before terminating. If the command buffer is queued with incoming and continuous is terminated with (N/n) (the ‘next’ qualifier) motion will continue through the queue. The caveat to

this is that if continuous has interrupted a loop and then itself interrupted by an xRef command, motion will return to the loop upon completion of the X command. The reason for this is that xRef is intended to be interrupt-and-return, continuous is not. Continuous is a one-shot, throw-away command that is targeted really as an on/off switch. Command_buffer[0] was set aside for this purpose.

Flagging a zero step motion, all parameters are written to buff[0], xRef is set to zero and the xRef and continuous flags raised - xRef to deal with the command queue and continuous to intercept motion run_state. This is a pre-emptive command and will be invoked immediately. It leaves the queue untouched and will return to interrupted motion (i.e. if a loop was in progress). As a throw-away command, all parameters are trashed at completion if they are invoked with the command line. If you want them to persist, invoke them separately and prior to the motion command. Example:

mv120a.8d1.2

creates a throw-away command for continuous motion at 120RPM, with acceleration 0.8 and deceleration 1.2. Upon completion system defaults will still be say 100RPM, accel 1.0 and decel 1.0. If however, you enter

v120
a.8
d1.2
m

the VAD parameters are established as system defaults and will persist. With a normal Qdump, buff[0] is not visible. As it is a throw-away, its contents are meaningless anyhow. It is visible under debug, just so you know it's there.

- **S/s**
  Stop. Either with prejudice (emergency) or with grace. ‘S’ terminates all operations immediately. ‘s’ (lower case) sends the current command immediately to its deceleration ramp which should minimize jerk if the situation allows for it. In either case, the command pointer is reset so no further commands are available for execution (but still in the queue).

- **N/s**
  Terminates the current command with prejudice (‘N’) or grace (‘n’) and proceeds to the next command in the queue, if there is one.

- **Internal/external clock** default internal
  Configurable as a persistent parameter, the internal clock flag, ‘C’ (upper case) in the configuration options, is set to 1 if using the internal clock or 0 is using and external source (see pin connections for T1). In most cases the internal clock is appropriate. There are however, instances which require coordinated time and synchronization that can only be achieved through an external clock. mother provides such a signal as well as the synchronization. These can of course, be provided by any external process. It is assumed that the clock frequency will be 1MHz as all timing is in microseconds. If the external clock source is another frequency, enter that value through the configure screen as well, option ‘c’ (lower case). Both the clock flag and external frequency are configurable through the CLI (C/c) command.

- **ops/softs_ops** default off
  All motion commands will trigger a hardware flag to logic 1 when the operation begins and reset to zero at completion. This signal can be used by other applications for synchronization. A programmable persistent option (default OFF) is the soft_ops packet containing either DC1 (0x11) or DC2 (0x12) depending on whether the operation has started or stopped (respectively). The reason this is optional and by default is OFF, is because of the GAP test. Almost certainly the soft_ops

transmission will force the ensuing command to accelerate from STOP. (See GAP for more information).

From the configure menu or as a system command, soft_ops can be switched off(0) or on(1) as a persistent default. Also as a system command ($On) soft_ops can be turned on, n = 1, or off, n = 0, for the duration of the session.

- **GAP** default on
  GAP is a time measurement between the end of the command just completed and the beginning of the next queued command. Measured against the current cycle timing and the processing of the incoming command parameters, a decision is made as to whether operations can continue seamlessly without risk of stalling. With no prior knowledge of system loading or other configuration specifics, this decision is based purely on timing. As this is agreeably quite arbitrary, there is a simple CLI (G/g) that toggles GAP testing on or off. Check command defaults ((C/c)(F/f)) for current status.

  Many operations require speeds that rarely exceed V' and as such there is really no need for GAP testing. There are times however, when excess loads, otherwise unforeseen, may cause a motor to stall with infinite acceleration expected at too high a speed. In these instances it may be operationally advantageous to maintain motion, even if restarting from zero and accelerating back to the target velocity. These are operational considerations and beyond the scope of the system design save for the option to turn it on or off.

  GAP is used by default as the timer to reset startup velocity at standby. If GAP is turned off, the programmed velocity will remain at the previous command level. If that previous command terminated with a deceleration ramp of 999 (0 steps) the in-play speed will still be whatever the target of that command reached. By design, the subsequent (next) command will accelerate from that last speed to the new target. If, for example, they are the same, there will be zero acceleration as the target speed immediately becomes the current speed. Caution must be exercised that this infinite acceleration is acceptable and will not cause any disastrous results. The onus again is on the operator to ensure that the entire profile is well conceived.

- **sync** default off
  All commands have a sync option. At the motion command level, S/s will set the sync flag. The queued command will be suspended until a sync release is received, either a soft "y" from the terminal or network or a high to low transition of the sync terminal. From a queue edit perspective, Q/qnnS/s will toggle sync on or off for position nn.

- **T/t start wait** default 0
  Intended for use with sync, every command has the programmable option of delaying its start by the specified number of microseconds. In a multi-SAC deployment, commands can be queued and held in check with sync, then started at various times after t0 based on their respective delays. Delta_t uses a 32 bit placeholder so the maximum delay is 2^32 microseconds or just over 71 minutes.

- **Encoder**
  There are posts labeled EnA and EnB which are for a quadrature encoder. The encoder has two modes: follow and throttle.

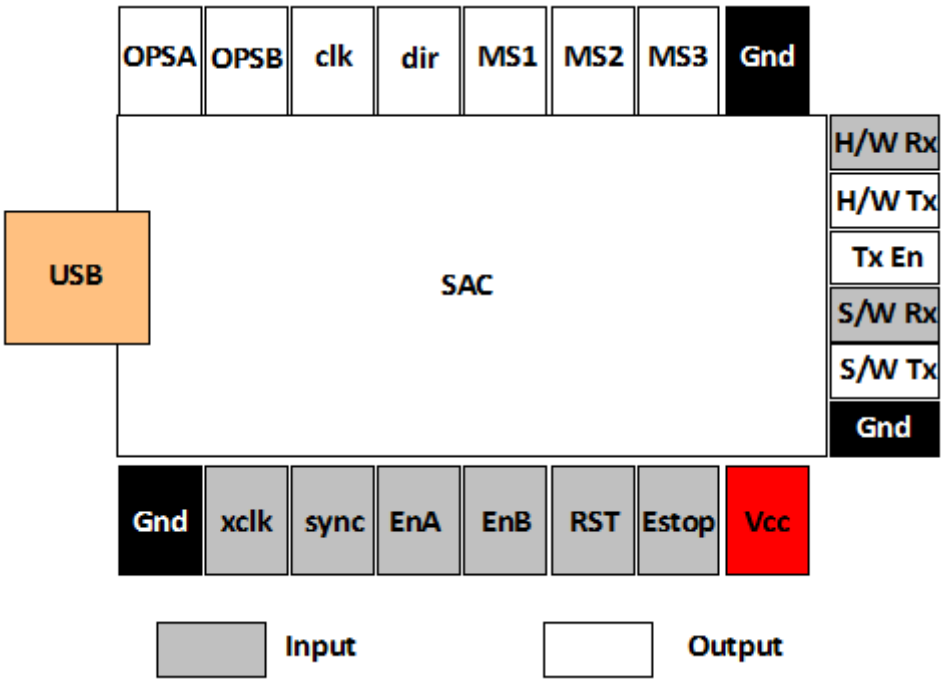
  In follow mode, the encoder causes a pulse to be generated in the same direction of rotation. Thus the motor 'follows' the encoder. This pulse will only be generated if otherwise the motor is not engaged.

  Throttle mode is intended for use with continuous motion. If the motor is in motion, a turn clockwise will increase speed and clockwise, decrease. These speed changes will not affect the direction of

motion, to do that a new command must be issued. If the motor is in STANDBY mode, the encoder has no effect. It is recommended that GAP be switched off when in throttle mode.

Encoder function automatically responds to the current state of motion. There is no setting for this.

- **Terminal Connections**



OPSA and OPSB are the hardware flags that reflect the start and finish of a command cycle. OPSA goes to logic 1 at the beginning and resets to 0 at the end. OPSB is the inverse.

clk and dir provide the cycle pulse and the direction to the stepper motor.

MS1-3 are the configuration pins for the stepper motor driver, currently configured for the Allegro A4988 and TI DRV88xx.

H/W RX/Tx share the in-built USART with the USB connection

S/W Rx/Tx are software controlled

Tx En is set to logic 1 when SAC is transmitting and cleared to 0 otherwise. Intended use is for half-duplex RS485 network connections.

xclk is the input for and external clock signal

sync will clear the sync flag when pulled to ground.

EnA and EnB are the inputs for a quadrature encoder.

RST is the bail-out factory reset that needs to be held low during a cold boot

Estop is biased to logic 1 and if pulled to ground will terminate with pejudice.

Configure

CAUTION – There are no checks on the validity of the data you enter. Using this facility is with the understanding that invalid data may cause the unit to convulse into an irrecoverable state. Most persistent parameters are system defaults. Some have session defaults and a further few are implemented in line commands. These are noted where applicable. For system defaults to become effective you must at least perform a warm reboot ($R).

Entering the system command $C will bring up a screen similar to this:

```
Config:
A autorun       0       a sprf          200
B ramp          0       b cycle         3000
C clock         1       c xfreq         1000000
D                       d acc           1.00
E network       0       e dec           1.00
F cw            0       f l_acc         10.47
G dir           CW      g l_dec         10.47
H ihd           67      h ops           100100
I xclk          0       i soft_ops      0
J ustep         1       j ACK           1
K pulse         30      k CRC           1
L slow          10      l Enc           0
M d_id          85      m hw            38400
N r_id          2       n sw            38400
SAC>
```

Configure Menu

Flags want a zero or one. In reality, any non-zero number will be dealt with as a one anyhow.
8-bit values will accept any number between zero and 255. Reality creeps in again and allows you to enter any number but it will be stored as modulo 255. You have been forewarned,
16-bit values, well there's only one and the reason it's there is because if you're using 0.9 degree motors (or a gearbox) the full steps per revolution is greater than 255.
32-bit values can hold some really big numbers, or floating point values. K?

Let's examine each and make sure we know what we're changing. Reality suggests that until you actually try to invoke the motion engine, these values can be changed again and again. So, make your changes, see what you've got and then carry on. Bear in mind that unless you 'brick' the unit, you can always restore the factory settings and start over.

From this screen, values can be changed by entering the single letter associated with the parameter followed by the new value (no spaces). Outside from this screen, a command line instruction (CLI) prefixed with $C will garner the same result. In the following, command line instructions reference parameters as they are indexed as of this writing. Be sure to check the configuration of the unit you are modifying to see that the indices are correct as different software releases may have different values.

- **autorun**
  The default of zero will simply allow SAC to start and present the operator with a command line. Any non-zero value, up to including the length of the command buffer (any value greater than this will be pared back to the maximum), will cause a loop to be initiated at startup. The loop will process commands preserved in the queue from the last Q save (Q/q)(S/s) beginning at position one and looping at the position identified by the autorun parameter. This value can also be modified via a CLI $CAnn where nn is the loop-back queue position.

- **ramp**
  Ramp defines the acceleration, discussed elsewhere so not here. This value simply determines what default ramp will be in use at startup. The default use of the Harmonic profile is indicated with ramp set at zero while a setting of one indicates Linear. This is of course programmable at the command level. A display of session defaults (C/c)(F/f) will reveal which ramp is currently being used. Additionally, when the command queue is displayed (Q/q) all commands using the Linear ramp profile are displayed with an asterisk while those without employ the Harmonic curve. These are individually editable, as are all values in the queue. See Q program for details.

- **clock**
  As the nature of stepper motor motion is in varying the time delay between pulses sent to the driver hardware, there must be a clock. The source of this heartbeat is often a crystal or ceramic resonator and, as with all things electronic, is subject to manufacture within tolerances. What this boils down to is that a 20MHz resonator, +/- 0.001% or 10PPM, is guaranteed to give you just that and not exactly 20MHz. In stand-alone applications this is generally not an issue. When multiple units are operating as a collaboration however, that variance can potentially accumulate and lead to catastrophic failure.

  It is with this in mind that SAC is equipped to time itself based on an external clock. There is a separate terminal post to attach a source and assuming the devices share a common ground all that's needed is to flip the software switch via the configure menu or through the CLI $CC(0/1), where 1 (default) implies the use of the internal clock and zero, the external.

  All timing resolution is in microseconds and as such it is assumed that any external clock source would be 1MHz. If this is not the case, you will notice in the configure menu the option to input another value for this. As you might imagine it is essential that the external source is properly detailed.

- **network**
  The network flag indicates which port will function as the network connection thereby establishing the terminal port as well. If set at zero, the default, the network port will be the software controlled posts and the terminal will utilize the USB/hardware components. When set to one, the USB will become the network interface and thus only speak in packets, communication with a 'dumb' terminal is effectively cut off from this port at that moment. If you do not have an intelligent network connection or a terminal connected to the software port, you have in fact 'bricked' your device. This is recoverable with a hard reset but in doing so every setting will be restored to the factory default and the command queue will be reset. Be aware of your network configuration prior to making potentially disastrous changes.

  To recover from a network block, locate the hard reset post. This is an open drain input held high by an internal resistor. Shunting this to ground during a cold reboot will force a factory reset. Often simple contact with the connection will provide a sufficient ground plane.

- **cw**
  Depending on the driver and the manner in which it is wired, the direction pin will result in clockwise motion if presented with a logic 0 or 1. Simply change this setting to reflect reality.

- **dir**
  When initializing the device and later when entering motion commands, there are many defaults, one of which is rotation direction. With no other information, this is the direction that motor will turn when issued a default command. From the factory this direction is clockwise. If you wish you can change it to counterclockwise by adjusting this setting.

You will note that on the configure menu the letters CW or CCW are displayed as their binary value will depend on the entry made above for cw. Simply follow that lead with a zero or one, not the letters, depending on whether you need CW or CCW default motion.

- **ihd**
  Interrupt overhead is meant to take care for the hidden time consumed in servicing the call for a pulse to be sent to the motor driver. It is important to understand that the deterministic nature of real-time embedded systems guarantees that a service request is honoured within a finite period of time. As is the nature of all processors upon receipt of an interrupt, the current instruction being processed is first completed and the the jump is made to the service routine. Instructions can vary from one to three clock cycles and there is little chance of knowing exactly when the interrupt will occur. Add to this environmental factors such as heat which will cause a clock's output to vary.

- **xclk**
  This has the opposite effect of ihd. If an external clock source is being used that results in timing cycles that are just too quick and there is no possible means of adjusting the source, this value can be tweaked to slow the cycle down. Xclk and ihd will cancel each other if they have the same value as one is added to the clock counter and the other subtracted, thus avoiding negative numbers and doubling the range of adjustment to +/- 255. One should be drawn down to zero before the other is increased.

- **ustep**
  As a default, some drivers either have no microstep capabilities of they are selected via jumpers and/or switches on the device itself. Whichever the case, this persistent parameter can be set to reflect that or simply as the default on a programmable device.

  In other instances, software adjustments to the microstep value are available and programmable through the configure menu or via a command line instruction. Note here that any reference to microstepping in terms of configuration, must be according to the binary power. Valid settings therefore are:

Setting	Microstep
0	1
1	2
2	4
3	8
4	16

  and possibly

5	32
6	64
7	128

  depending on the device.

  Please acquaint yourself with the relevant issues concerning microstepping, torque and speed limitations prior to adjustment of these settings.

  There are three terminal posts available for programmable drivers labeled MS0 – 2. These should be considered as the three least significant bits in a byte giving us values from B000 to B111. These values correspond to the logic levels required by the driver and are represented internally as the exponent for microstep calculations. There are two configurations pre-programmed: device_id 0 is for the Allegro A4988 and device_id 1 is the TI DRV8825. Device_id 2 is available for other drivers but needs to be programmed. Programming is done with a hidden command through the CLI configure option. The command syntax is $CUnx where n is the binary exponent that results in the microstep multiplier (i.e. 0 → 2^0 = 1 = Full Step, 1 → 2^1 = 2 = ½ step, 2 → 2^2 = 4 = ¼ step, etc.) and x is the

decimal equivalent of the binary value represented by the logic level of the three pins required by the driver to result in the specific microstep.

Example: The Toshiba TB6600 specifies M1 = H, M2 = L and M3 = L to give ¼ step (x4). Thus our binary exponent is 2 which is n. The pin logic, if we map M1 to M0, M2 to M1 and M3 to M2 we get LLH or 001 which is decimal 1 which is x. So the command to register that mapping in $CU21, and so on. There are eight placeholders for pin mapping that cover all the three bit combinations. It is important that these values are mapped correctly as there is no feedback from either the motor of the driver that would indicate whether the resultant motion corresponds correctly to the number of pulses processed.

If you have a SAC with a built on driver, all of this or course is moot except for ensuring that the microstep setting is correct. All the pin mapping is done already and the result should be transparent unless switching between full step and pulse in the treatment of motion values.

- **pulse**
  Each driver has its own specification for the duration of the incoming pulse necessary to trigger an event. Most modern drivers (A4988, DRV8825, TB6600) all require 1 microsecond. Some older drivers specify longer periods, the TB6560 for example wants 30us. This setting allows for adjustment to the duration of the pulse.

- **slow**
  See the discussion on acceleration and more specifically, infinite acceleration. Regardless of the chosen starting speed, there will be this brief period of zero to something, hence the term infinite acceleration. With this comes jerk (the change of acceleration with respect to time) which can have disastrous effects on equipment. It is however a reality and must be dealt with. The default starting speed, defined here as slow, is also the speed to which a return to zero or stop is directed as well as the speed at which a change of direction will occur unless specifically programmed otherwise. It has been chosen for default as 10RPM, as 30,000us clock pulse.

  The reasons for using 10RPM are twofold. Firstly, ramping up from zero, or one (there really is no zero speed) has empirically shown no benefit and adds only to the time to target. In fact at speeds less than 10RPM there is no inherent acceleration, they are all treated as infinite jump-to targets.

  Secondly, speeds above 10RPM are dealt with using internal timers with 16-bit counters operating at two MHz effectively limiting the count to 32,768us. At 10RPM and below pulse timing is handled by a long timing function that can extend for years. There are two distinct and different algorithms employed depending on the target velocity and that change-over point happens to be at 10RPM.

- **d_id**
  Device ID. All network traffic is via variable length packets. Each packet has, as part of its header, a destination address and a source address. The source address is the device ID of the originator while the destination address is that of the other device. A default ID will be assigned and in the case of a stand-alone unit this is not an issue. In a networked environment though, address conflicts are not desired and as such each device should be assigned a unique ID. These are 8-bit addresses and as such can range from 0 – 255. Zero however, is the general call or broadcast channel and 1 is reserved for mother or some other command and control master. 2 and 3 are the non-printable control characters for STX and ETX used in packet transmission and are therefore unavailable as well. In general it would be better to avoid the use of addresses below 10.

- **r_id**
  Driver ID. For the purpose of run-time configuration of microsteps, the driver ID must be known. At the time of this writing the only two drivers pre-configured are the A4988 (id = 0) and the DRV8825

(id = 1). There is provision for a third driver (id = 2) and requires that the pin configurations be programmed. See ustep for details.

- **sprf**
  Steps per revolution – full. Ignoring any microstep, this is the full step count for one revolution, typically 200 for a 1.8° motor or 400 for a 0.9° motor.

- **cycle**
  Every motion command must have as input the number of steps to be generated. That is the only requirement, All other parameters can rely on preset defaults. Target velocity is one and as has been discussed elsewhere in this document, this equates to a cycle time in microseconds. Remembering the simple relationship between cycle time and RPM (RPM = 300,000/cycle), 3000 (the default) is equal to 100RPM. This of course is dynamic and is used to seed the command queue at startup. It may in some cases be the basis of all motion and as such is a configurable persistent parameter.

- **xfreq**
  As alluded to earlier in the internal/external clock discussion, this is the CPP (configurable persistent parameter) for the frequency of the external clock source.

- **acc**
  A default value for acceleration along the Harmonic curve. This is a floating point value with two digit precision up to 100 and integer values after that. That translates to 0.01 to 99.99 and 100 to 999. Recall that in all cases for acceleration and deceleration, 999 is the key value to indicate no ramp to be used. With a value of 999 motion will initiate or terminate at the target velocity. The onus is on the profiler to ensure that this is a workable option. Remember as well that acceleration, if not zero (999) will occur from the in-place velocity so terminating a motion command with zero (999) deceleration followed by a command with a non-zero acceleration will in fact produce a smooth transition from the existing velocity to the new target, as in fact what it was designed for. The acceleration steps are charged against the incoming command.

- **dec**
  As with _acc but for deceleration.

- **l_acc**
  As above but for acceleration along the Linear ramp. Read the discussion on acceleration and deceleration for more information.

- **l_dec**
  As with l_acc but for decelaration.

- **ops**
  This value is displayed as binary and represents the two hardware connections that are triggered at the beginning and end of each command. Under normal circumstances what you will see as this value is 100100 or decimal 36. If, for some reason you wish to turn these flags off, simply set this value to zero. Later it can be turned back on by writing the value 36 back in.

- **soft_ops**
  This flag sets the default as to whether soft ops messages are transmitted. There is a session setting for this as well.

- **ACK**
  When enabled, the default, and upon receipt of a network packet, SAC will respond with either ACK (ASCII 6) or NAK (ASCII 21). An ACK will be sent in case of the packet simply having been received

unless there is a CRC byte attached. If there is a CRC, it will be check and if valid proceed to respond with ACK. Otherwise a NAK will be returned. No acknowledgement is made one way or the other in the case of a general call (broadcast to address 0). The onus is on the originator to react to a negative response, no further action will be taken by the intended recipient.

ACK consumes processor cycles as it builds its response packet and then waits for the transmission to complete. For a variety of reasons this may not be necessary. The response can be disabled/enabled as a persistent default via the configuration screen or toggled with a system command (session default) $K(0/1) for off/on.

- **CRC**
  Using the Dallas/Maxim inverse polynomial, an 8-bit CRC is calculated for the body of the text (header and footer excluded). In response by a recipient of a CRC check, the originator will know if the transmission was without error. A negative response may initiate a retransmit attempt, that is the prerogative of the operator. In some networks the lines are clean and this overhead is unwarranted. For that reason the option exists to switch the CRC calculation and inclusion within the packet, on or off. From the configuration screen or through CLI config, the system default (persistent parameter) can be changed, or temporarily as the session default with the system command $H(0/1).

  As the recipient, the CRC will be checked if it is there. At the packet level the byte at the end of the body is ETX (ASCII 3) and the CRC is to follow. If the CRC is present, the next byte will be non-zero and will therefore be confirmed as a valid CRC (or not). If however, the byte is zero, which is the array terminator, CRC validation will be bypassed – no operation intervention.

- **hw**
  Sets the baud rate for the hardware serial port. Settings up to 2Mbps. 250k is the first useful speed with 0.0% error. Beware the speed vs. distance tradeoff.

- **sw**
  Sets the baud rate for the software serial port. 38400 is the chosen speed for the software connection as there is only 0.2% error. From a communications perspective, SAC packets are quite small and as such are less likely to be affected by baud rate errors. In any event, 57k6, 115k2 and 230k4 should always be avoided when transmitting more that a handful of bytes.

System Commands

System commands differ from operational commands in that they alter the underlying state of the processor. Often they will require a restart or at least a re-initialization of system variables. It is intended that system commands are entered in a non-operational or stand-by/stasis mode for safety purposes. To ensure this, all activity will be terminated upon entering a system command. System commands are prefixed with a '$'.

- **$L** Linear ramp vs. modified S-curve. By default the system start-up is modified S-curve. Entering $L1 will switch to Linear Ramp, $L0 will revert.

- **$R** is the system reset, or warm boot. It will reload the persistent parameters (PPs) from EEPROM thereby returning all values to their defaults. Additionally, non-persistent variables are recalculated to reflect any changes that may have occurred since the last restart. This is necessary in the case of configuring some of the PPs and dependent values are only set at boot.

  System reset will NOT affect the command queue. If there is a command structure already programmed into the queue, it will remain intact. If you wish to reset the queue, you can do that through a line command.

- **$C** will bring up the configure screen that displays all of the PPs. The operator will be given the option to adjust any or all of these values until exiting (X/x). There is no error checking to ensure that viable entries have been made. The onus is on the individual. These changes will be made directly without prompt or confirmation. The display however, will be refreshed (in terminal mode) so a visual check may be made.

  $C can be used as a CLI (Command Line Instruction) as well if you append the character identifier (as you would from the config screen) in addition to the new parameter value. As an example, $CA8 would set the autorun loop end value to eight.

  All configuration changes are written directly to EEPROM.

- **$E** will display the EEPROM values that underlie the current configuration. If you are in debug mode, this command will send a HEX dump of the entire EEPROM to the screen.

- **$F** sets all motion commands to be treated as full step values, irrespective of the microstep setting. Execution of this command sets the full step flag (see $P).

- **$H** toggles the CRC transmit flag and requires an explicit setting ($H0) to switch it off. Any other value will leave it on which is the default. The CRC check is the mechanism by which commands are rejected if they fail, suggesting an error in communications and an invalid receipt. There is no other validity check and turning this off may have serious negative effects on your operations. In some implementations however, this may be considered unnecessary overhead, thus the option.

- **$K** Acknowledge (ASCII 6) or not (ASCII 21), ACK/NAK. As with CRC, this can be explicitly turned off with $K0. If on (default), each received command (network commands only) will be acknowledged if they pass the CRC check. If CRC checking is switched off then a properly formatted network packet will result in a positive acknowledgment.

- **$O** There are two hardware terminals that carry signals indicating the commencement and completion of a command. $O controls a software option that will broadcast a DC1 (ASCII 17) packet at the start of a command cycle and a DC2 (ASCII 18) packet at the end. Typically, 0/1 are the values to turn this feature off or on.

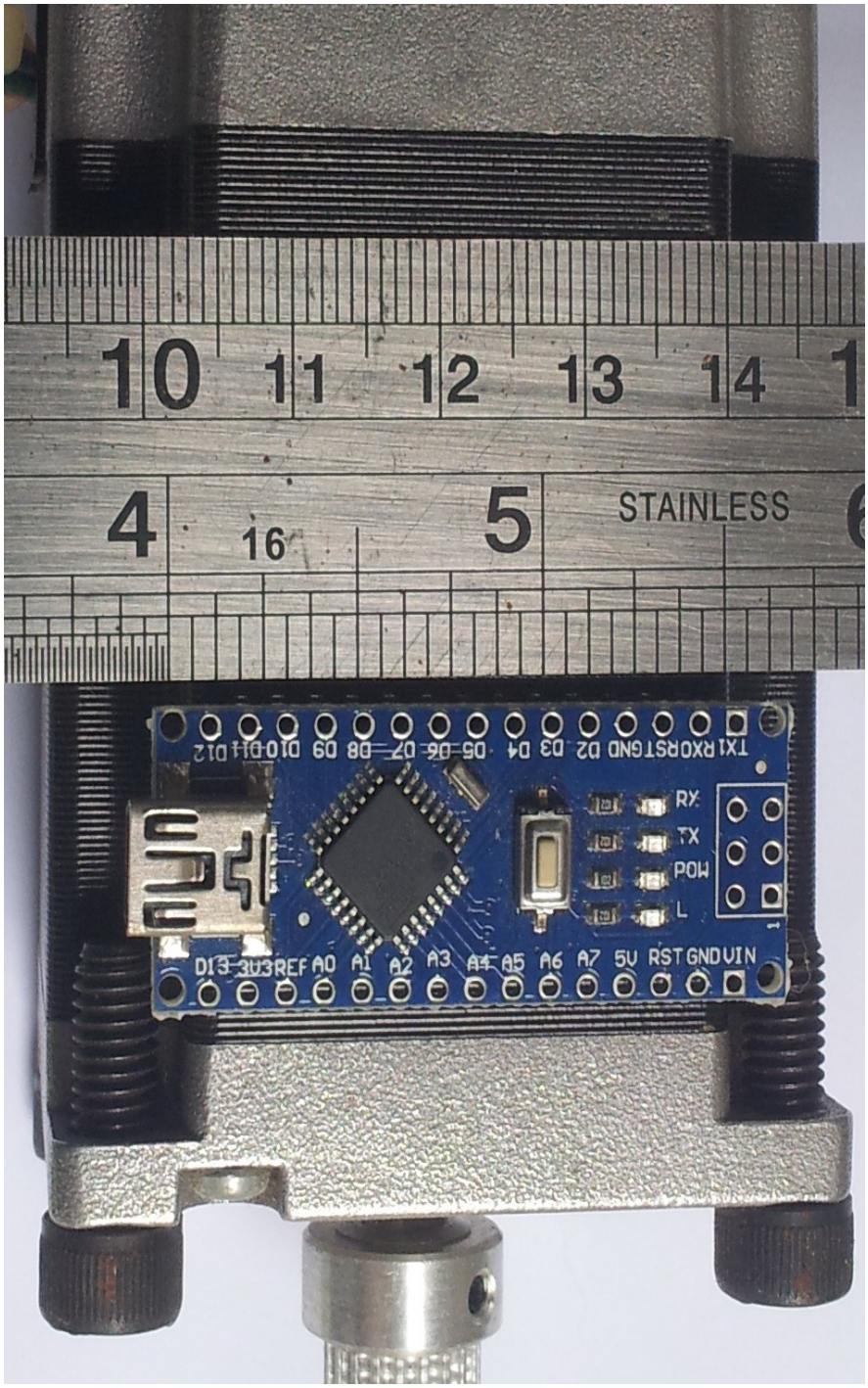
- **$P** Paired with $F, will clear the full step flag and motion commands will treat values as pulses rather than steps.

APPENDIX A

References

1. "Generate stepper-motor speed profiles in real time" by David Austin (Embedded Systems Programming, January 2005, www.embedded.com/56800129).
2. AVR304: Half Duplex Interrupt Driven Software UART, Application Note 0941C-AVR-04/08, Atmel Corporation, www
3. AVR306: Using the AVR USART on tinyAVR and megaAVR devices, Application Note 1451C-04/2016, Atmel Corporation,
4. AVR446: Linear speed control of stepper motor, Application Note 8017A-AVR-06/06, Atmel Corporation, www

APPENDIX B



SAC working model (less driver) on a NEMA 23 motor
17.7mm x 43.5mm (0.7” x 1.7”)

SAC will sit in a standard 32 pin socket. At the time of this document, the preferred motor drivers are the Allegro A4988 or the Texas Instruments DRV 8880/8825. They are essentially pin replaceable and have a 15.3mm x 20.7mm (0.6” x 0.8”) footprint. With decoupling capacitor and connectors, SAC occupies less than 50x50 mm (2 inches square).

APPENDIX C

Command Summary

M/m[d](nnn)[(A/a)nn][(D/d)nn][(V/v)[C/c]nn][(T/t)nn][(L/l)n][(U/u)n]

Motion line command.

d – Either blank, indicating positive, for clockwise motion or "-", negative, for CCW.

nnn – the absolute number of steps to process, inclusive of accel/decel.

(A/a)nn – acceleration

(D/d)nn – deceleration

(V/v)[C/c]nn – velocity[cycles]

T/t – start wait in microseconds

L/l – if 1 use linear acceleration profile, 0 for Harmonic profile

S/s – sets sync flag for the command

U/u – tags on n pulses (not steps) to the CRUISE cycle

+ Generates one step CW

- Generates one step CCW

A/a[nn]

Causes the session acceleration default to be set to nn. If nn is omitted, the session default will revert to the system default.

D/d[nn]

As with acceleration.

L/l[(S/s)nn][(F/f)nn][(N/n)]

Loop start (S/s), finish (F/f), number (N/n) where nn is the queue positions of start/finish. If qualifiers omitted, signal to end loop following the completion of the next command.

V/v[nn][(C/c)xxx][(D/d)]

Set the session velocity default to nn RPM or as a cycle delay of xxx microseconds.

D/d will display the current velocity in RPM and cycle delay.

S (upper case) Stop with prejudice. Terminates all motion immediately.

s (lower case) Stop with grace. Terminates all motion with the current session decel profile.

N (upper case) Terminates current command immediately. If there is a command pending in the queue, proceed.

n (lower case) Terminates the current command with the command decel profile. If there is a command pending in the queue, proceed with next.

B/b Debug. Toggle in and out of debug mode.

C/c(a) Line command where a is the command

c – Display current configuration

f – Display current session defaults and flag status

G/g Toggle in and out of GAP testing

Q/q(a) Line command for the command queue. With no qualifiers displays the command queue on the terminal screen.

P/p – enter Q program mode

R/r – reset the queue, seed all queue entries with system defaults

nn – Q position to be followed by Q program directives (See Q program)

X/x[nn][D/d][L/l][P/p]

Line command, xReference. Will execute queue position nn.

D/d – display xRef array

L/l – begin loop

P/p – program xRef array

y (lower case) Terminal command to drop the sync flag

Z/z Undo

$ System Command Processor

System Command Processor

All system commands are preceded by a dollar sign, solely to safeguard against accidental discharge.

$C Configure

With no qualifiers, enters the configure screen.

With qualifiers it functions as a system level line command for configuring SAC. This allows for remote configuration without the overhead if the display screen. See the discussion on configure for details.

Any change to the configuration should follow with a cold or warm reboot. Failure to do so will result in unpredictable behavior. Remember, a warm reboot ($R) leaves the command queue untouched.

$E EEPROM

Display the contents of EEPROM, the persistent parmeters.

$H CRC check

If explicitly set to zero, will switch off CRC creation (detection is automatic).

$K ACKNAK

As with CRC, if explicitly set to zero, will switch off ACK/NAK transmission

$L(0/1) Linear ramp

Set (1) linear ramp mode or clear (0) to return to Harmonic. Session accel/decel defaults will automatically be adjusted.

$n (lower case) Encoder – toggles between follow and throttle mode. See session defaults for current setting, EnT (throttle) or EnF (follow).

$On Soft ops

The hard-wired operations flag is set at the start of every command and cleared upon termination. Soft ops will issue a DC1/DC2 (start/end) packet over the network at the same time as the hard flag is triggered if set to one.

$R Reset – will execute a warm reboot. All system variables reset and/or recalculated. Does not affect the command queue. Should be performed after any change to the configuration.

Miscellaneous

The bail-out option. If all else fails, re-write the EEPROM and restore the system to factory settings.

When performing a cold start and with a terminal connection (will not work over the network line), you will notice between the version release data and the first SAC> prompt, approximately a one second delay. During this wait period SAC is looking for any incoming signal from the terminal. If it receives one it will go into factory restore mode. If you have bricked the device and lost your terminal connection it will be necessary to perform a hard reset. See network under Configure for hard reset instructions.

Hidden command to override full step. By default all commands are processed as full steps, multiple pulses for a single step if in microstepping mode. In session, this can be changed (toggled) by issuing the system command $U. As a persistent parameter, the configure command u(0/1) will set the system default.

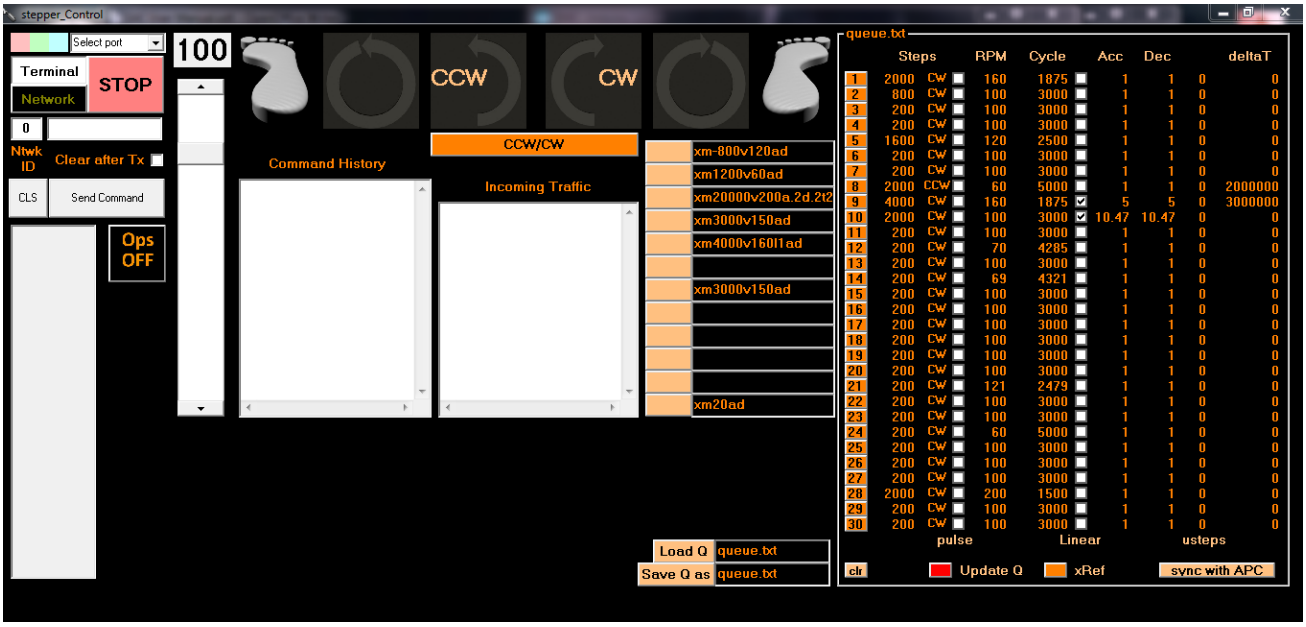
An option around this is to leave the system in full step mode and add pulses to the CRUISE state, adjusting for the additional microsteps. This is accomplished by including (U/u)n in the motion command, n being the number of pulses to add. Remember that in the motion engine, steps are converted to pulses so 1000 steps as described by the m1000 command becomes 8000 pulses with x8 microstepping. This is a difference from m1001 which becomes 8008 pulses. M1000Un however, becomes 8000 + n pulses. The value n is a byte so in fact can range as high as 255, room enough to play with this capability. This will not work with truncated commands as they never achieve CRUISE state.

Estop

Hard-wired and biased high, the emergency stop will terminate all motion with a momentary grounding of this terminal post.

COMPANION APP

Screen Shot



While discussion of the companion app is outside the scope of this document, you can visualize the controls exposed for convenience. This app is free to download from Github.