Programming: Command Set

Programming: Command Set

Commands are grouped by function - with the following notations:

#	Refers to an integer number
exp	Refers to an expression or signed integer
var	Refers to a variable
com	Refers to a communication channel

A=exp Set absolute acceleration

Acceleration must be a positive integer within the range of 0 to 2,147,483,648. It defaults to zero so you must set it to something to
ever get motion. A typical value is 100. If left unchanged during a motion, the same value will govern the acceleration as well as the
deceleration to form either a triangular or trapezoidal velocity motion profile. It can be changed at any time during or between motions.
The value set does not get acted upon until the next ‘G’ command is issued. If your motor has a 2000 count encoder (sizes 17 & 23),
multiply your desired acceleration in rev/sec2 by 7.91 to arrive at the number to set A to. With a 4000 count encoder (sizes 34, 42 & 56)
you will need to multiply by 15.82. These constants are a function of the motors PID rate. If you lower the PID rate, you will need to raise
these constants.

The following code shows how you could use variables to set motion values to real-world units and have the working values scaled for
motor units.

```
a=100           ‘Acc in rev/sec*sec
v=1             ‘Vel in rev/sec
p=100           ‘Pos in revolutions
GOSUB10         ‘Initiate motion

C10             ‘Motion routine
  A=a*8         ‘Set Acceleration
  V=v*32212     ‘Set Velocity
  P=p*2000      ‘Set Position
  G             ‘Start
RETURN          ‘return to call
```

V=exp Set maximum permitted velocity

Use the ‘V’ command to set a limit to what velocity the motor can accelerate to. That limit becomes the slew rate for all trajectory based
motion whether in position mode or velocity mode. The value defaults to zero so it must be set before any motion can take place. The
new value set does not take effect until the next ‘G’ command is executed. If your motor has a 2000 count encoder (sizes 17 & 23),
multiply your desired velocity in rev/sec by 32212 to arrive at the number to set V to. With a 4000 count encoder (sizes 34, 42 & 56)
you will need to multiply by 64424. These constants are a function of the motors PID rate. If you lower the PID rate, you will need to

raise these constants.

D=exp Set relative distance for position move

The “D=” command will allow you to specify a relative distance, instead of an absolute position. The number following is encoder counts
and can be positive or negative.

The relative distance will be added to the current position, either during or after a move. It is added to the desired position rather than
the actual position so as to avoid the accumulation of small errors due to the fact that any servo motor is seldom exactly where it should be.

P=exp Set absolute position for move

The “P=” command allows the setting of an absolute end position. The number following is encoder counts and can be positive or negative.
The end position can be set or changed at any time during or at the end of previous motions.

G Go, start motion

The “G” command does more than just start motion. It can be used dynamically, during motion to create elaborate profiles. Since the
SmartMotor allows you to change Position, Velocity and Acceleration during motion, “on-the-fly”, the “G” command can be used to
trigger the next profile at any time.

S Abruptly stop motion in progress

If the “S” command is issued while a move is in progress it will cause an immediate and abrupt stop with all the force the motor has to offer.
After the stop, assuming there is no position error, the motor will still be servoing.

X Decelerate to stop

If the “X” command is issued while a move is in progress it will cause the motor to decelerate to a stop at the last entered “A=” value.
When the motor comes to rest it will servo in place until commanded to move again.

O=exp Set/Reset origin to any position

The “O=” command (using the letter ‘O’, not the number zero) allows the host or program not just to declare the current position zero,
but to declare it to be any position, positive or negative. The exact position to be re-declared is the ideal position, not the actual position
which may be changing slightly due to hunting or variable loading.

OFF Turn motor servo off

The “OFF” command will stop the motor from servoing, much as a position error or limit fault would.

MP Position Mode

Position mode is the default mode of operation for the SmartMotor. If the mode were to be changed,

however, the “MP” command would
put it back into position mode. In position mode, the ‘P#’ and ‘D#’ commands will govern motion.

Binary Data Transfer

The ASCII based command string format, while convenient, is not the fastest way to communicate data. It can be burdensome when
performing multi-axis contouring. For that reason a special binary format has been established for the communication of trajectory
critical data such as Position, Velocity and Acceleration. These 32 bit parameters are sent as four bytes following a code byte that
flags the data for a particular purpose. The code bytes are 252 for acceleration, 253 for velocity and 254 for position. As an example,
the following byte values communicate A=52, V=-1 & P=2137483648.

A=52	252 000 000 000 052 032
V=-1	253 255 255 255 254 032
P=2137483648	254 127 255 255 255 032

For further expediency, the commands can be appended with the ‘G’ command to start motion immediately. Two examples are as
follows (the ASCII value for ‘G’ is 71):

P=0 G	254 000 000 000 000 071 032
V=512 G	253 000 000 002 000 071 032

This data formatting is not supported by SMI - is is intended for those users writing specialized interfaces.

MV Velocity Mode

Velocity mode will allow continuous rotation of the motor shaft. In velocity mode the programmed position using the “P” or the “D”
commands is ignored. Acceleration and Velocity need to be specified using the “A=” and the “V=” commands. After a “G” command
is issued, the motor will accelerate up to the programmed velocity and continue at that velocity indefinitely. In Velocity Mode as in
Position Mode, Velocity and Acceleration are changeable on-the-fly, at any time. Simply specify new values and enter another “G”
command to trigger the change. In Velocity Mode the velocity can be entered as a negative number, unlike in position mode where
the location of the target position determines velocity direction or sign. If the 32 bit register that holds position rolls over in velocity
mode it will have no effect on the motion.

MT Torque Mode

In Torque Mode the motor shaft will simply apply a torque independent of position. The internal encoder tracking will still take place,
and can be read by a host or program, but the value will be ignored for motion because the PID loop is inactive. To specify the amount
of torque, use the “T=” command, followed by a number between -1023 and 1023.

T=exp Set torque value, -1023 to 1023

In Torque Mode, activated by the “MT” command, the actual torque value can be set with the “T=”

command. The following number or
variable must fall in the range between -1023 and 1023. The full scale value relates to full scale or maximum torque.

Brake Commands (where optional brake exists):

BRKRLS	Brake release
BRKENG	Brake engage
BRKSRV	Release brake when servo active, engage break when inactive.
BRKTRJ	Release brake when running a trajectory, engage under all other conditions. Turns servo off when the brake is engaged.

Many SmartMotors are available with power safe brakes. These brakes will apply a force to keep the shaft from rotating should the
unit lose power. Issuing the 'BRKRLS' command will release the brake and 'BRKENG' will engage it. There are two other commands
that initiate automated operating modes for the brake. The command 'BRKSRV' engages the brake automatically, should the motor stop
servoing and holding position for any reason. This would include loss of power, but also consider a position error, limit fault,
over-temperature fault, etc..

Finally, the 'BRKTRJ' command will engage the brake in not only all of the previously mention circumstances, but also at any time the
motor is not performing a trajectory. In this mode, rather than the motor servoing when it is at rest, the motor will be off, and the brake
will be holding it in position, perfectly still. As soon as another trajectory is started, the brake will release. The time it takes for the brake
to engage and release is on the order of a few milliseconds.

The brakes used in the SmartMotors are zero-backlash, extremely long life devices. It is well within their capabilities to operate interactively
within an application. You do not, however, want to create a situation where the brake would be repeatedly set during motion. That would
reduce the brake's life.

The Language:

Program commands are like chores, whether it is to turn on an output, set a velocity, start a move or whatever. A 'program' is a list of these
chores. When the programmed SmartMotor is powered-up or its program is reset with the 'Z' command, its will execute its program starting
at the top and progressing to the bottom.

RUN Execute stored user program

Resetting the motor to run the program erases all variables and mode changes and provides a fresh start.
Alternatively, you can use the 'RUN' command to start the program in which case, the state of the motor is unchanged.

RUN? Halt program if no RUN issued

In the event you do not want your downloaded program to execute upon power-up, you need to make the 'RUN?' command the first in the
program. That will prevent the program from starting when power is applied, but it will not prevent the

program from executing when the
motor sees the ‘RUN’ command come from a host over the RS-232 channel.

Once your program is running, there are a variety of commands that can redirect program flow and most of those can do so based on
certain conditions. How you set up these conditional decisions determines what your programmed SmartMotor will do, and exactly how
“smart” it will actually be.

GOTO# Redirect program flow
C# Subroutine label, C0-C999

The most basic command that redirects program flow, without inherent conditions, is the ‘GOTO#’. Labels are placed in the program as
numbers preceded by the letter ‘C’, like “C10” or “C320”. If you put that same number at the end of a ‘GOTO’ command like “GOTO10”
or “GOTO320”, then the program flow will be redirected to the location of that label. Labels can range from 0 to 999 and you can likewise
use as many as a thousand of them.

END End program execution

If ever you want your program to end, simply use the ‘END’ command and execution will stop right there. The ‘END’ command can also
be sent by the host to intervene and stop a program running within the motor. You never “erase” the program in the SmartMotor, you
only download a new one. If you do not want one then download a program with only the ‘END’ command.

GOSUB# Execute a subroutine
RETURN Return from subroutine

Just like the ‘GOTO’ command, the ‘GOSUB’ command will redirect program execution to the included label, but unlike the ‘GOTO’ command,
execution will return to the location of the ‘GOSUB’ upon completion of the subroutine, signified by the execution of the command ‘RETURN’.
If you encapsulate a piece of code into a subroutine, that code can be called upon repeatedly and from all different areas of the rest of the
program. Organizing your code into multiple subroutines is a good practice. In addition to being called by the program, you can call them
from a host, while the program is otherwise not running. Under this circumstance, it is like having many different programs.

The commands that can conditionally direct program flow to different areas operate on a constant (like ‘1’ or ‘25), a variable (like ‘a’ or
‘al[5]’) or a function involving constants and/or variables (a+b or a/25). Only one operator can be used in a function. A list of those
operators is as follows:

+	Addition
-	Subtraction
*	Multiplication
/	Division
==	Equals (use two =)
!=	Not equal
<	Less than

>	Greater than
<=	Less than or equal
>=	Greater than or equal
&	Bitwise AND (see appendix A)
|	Bitwise OR (see appendix A)

WHILE, LOOP

The most basic looping function is a ‘WHILE’. The ‘WHILE’ is followed by an expression that determines whether the code between the
‘WHILE’ and the following ‘LOOP’ will execute or be passed over. While the expression is true, the code will execute. An expression is
true that is non-zero. If the expression results in a “zero” then it is false. The following are valid ‘WHILE’ structures:

```
WHILE 1        ‘1 is always true
  UA=1         ‘Set output to one
  UA=0         ‘Set output to zero
LOOP           ‘Will loop forever

a=1            ‘Initialize variable a
WHILE a        ‘Starts out true
  a=0          ‘Set a to zero
LOOP           ‘This never loops back


a=0            ‘Initialize variable a
WHILE a<10     ‘a starts less
  a=a+1        ‘a grows
LOOP           ‘Will loop back 10x
```

The task or tasks within the “WHILE” loop will execute as long as the function remains true.

The ‘BREAK’ command can be used to break out of a ‘WHILE’ loop, although that somewhat compromises the elegance of a ‘WHILE’
statement’s single test point, making the code a little harder to follow. The ‘BREAK’ command should be used sparingly or preferably
not at all in the context of a ‘WHILE’.

If you want a portion of code to execute only once based on a certain condition then use the “IF” statement.

IF, ENDIF

Once the execution of the code reaches the ‘IF’ structure, the code between that ‘IF’ and the following ‘ENDIF’ will execute only when
the condition directly following the ‘IF’ command is true. For example:

```
a=UAI   ‘Variable a set 0/1
a=a+UBI          ‘Variable a 0/1/2
IF a==1 ‘Use double = test
  b=1   ‘Set b to one
ENDIF   ‘End IF
```

Variable ‘b’ will only get set to one if variable ‘a’ is equal to one. If ‘a’ is not equal to one, then the program will continue to execute with the command after the “ENDIF”.

Notice also that the SmartMotor language uses a single equal sign (=) to make an assignment, like where variable ‘a’ is set to equal
the logical state of input ‘A’. Alternatively, a double equal (==) is used as a test, to query whether ‘a’ is equal to 1 without making
any change to ‘a’. These are two different functions. Having two different syntaxes has further reaching benefits.

ELSE, ELSEIF

The ‘ELSE’ and ‘ELSEIF’ commands can be used to add flexibility to the ‘IF’ statement. What if you wanted to execute different
code for each possible state of variable ‘a’. You could write the program as follows:

```
a=UAI          ‘Variable ‘a’ set 0/1
a=a+UBI        ‘Variable ‘a’ 0/1/2
IF a==0        ‘Use double ‘=’ test
  b=1          ‘Set ‘b’ to one
ELSEIF a==1
  c=1          ‘Set ‘c’ to one
ELSE           ‘If not 0 or 1
  d=1          ‘Set ‘d’ to one
ENDIF          ‘End IF
```

There can be many ‘ELSEIF’ statements, but at most one ‘ELSE’. If the ‘ELSE’ is used, it needs to be the last statement in the
structure before the ‘ENDIF’.

You can also have ‘IF’ structures inside ‘IF’ structures. That is called “nesting” and there is no practical limit to the number of
structures you can nest within one another.

SWITCH, CASE, DEFAULT, BREAK, ENDS

Long, drawn out ‘IF’ structures can be cumbersome, however, and burden the program, visually. In these instances it can be
better to use the ‘SWITCH’ structure. The following code would accomplish the same thing as the previous program:

```
a=UAI          ‘Variable ‘a’ set 0/1
a=a+UBI        ‘Variable ‘a’ 0/1/2
SWITCH a       ‘Begin SWITCH
CASE 0
  b=1          ‘Set ‘b’ to one
  BREAK
CASE 1
  c=1          ‘Set ‘c’ to one
  BREAK
DEFAULT        ‘If not 0 or 1
  d=1          ‘Set ‘d’ to one
  BREAK
ENDS           ‘End SWITCH
```

Like a rotary switch directs electricity, the ‘SWITCH’ structure directs the flow of the program. The ‘BREAK’ statement then jumps the code execution to the code following the associated ‘ENDS’ command. The ‘DEFAULT’ command covers every condition other than those listed. It is optional.

TWAIT Wait during trajectory

The ‘TWAIT’ command pauses program execution while the motor is moving. Either the controlled end of a trajectory, or the abrupt end of a trajectory due to an error, will terminate the ‘TWAIT’ waiting period. If you had a succession of move commands without this command, or similar waiting code between them, the commands would overwrite each other because program advances, even while moves are taking place. The following program has the same effect of the ‘TWAIT’ command, but has the added virtue of allowing you to program other things during the wait, instead of just waiting.

```
WHILE Bt        ‘While trajectory
LOOP            ‘Loop back
```

WAIT=exp Wait (exp) sample periods

There will probably be circumstances where you will want to pause program execution for a specific period of time. Time, within the SmartMotor, is tracked in terms of sample periods. Unless otherwise programmed with the ‘PID#” command, the sample rate is about 4KHz. “WAIT=4” would wait about one second. “WAIT=1000” would wait for about one quarter of a second. The following code would be the same as “WAIT=1000”, only it will allow you to execute code during the wait if you place it between the ‘WHILE’ and the ‘LOOP’.

```
         CLK=0  ‘Reset CLK to 0
WHILE CLK<1000         ‘CLK will grow
          IF UAI==0      ‘Monitor input A
            GOSUB911   ‘If input low
          ENDIF‘End the IF
         LOOP   ‘Loop back
```

The above code example will check if input ‘A’ ever goes low, while it is waiting for the the ‘CLK’ variable to count up to 1000.

STACK Reset the GOSUB return stack

The “stack” is where information is held with regard to nesting of subroutines (nesting is when one or more subroutines exist within others). In the event you direct program flow out of one or more nested subroutines, without executing the included ‘RETURN’ commands, you will corrupt the stack. The ‘STACK’ command resets the stack with zero recorded nesting. Use it with care and try to format your program such that it is totally unnecessary.

The last example serves as a possible use of the ‘STACK’ command. If you were nested into one or more subroutines and an emergency occurred, the program could issue the ‘STACK’ command and then ‘GOTO’ a label back at the beginning. Using this method instead of a reset, would retain the states of the variables and allow further specific action with regard to the emergency.