

# jStepper

An Arduino style C++ Library for complex control of multiple stepper motor movements with individual speed, synchronization, and linear acceleration profiles. The library offers a complete systems approach for applications such as 3D printers, CNC machines, and robotics.

## FEATURES

1. Supports up to 3 stepper motors per instance of the library.
2. Multiple instances limited to the number of available 16-bit timers.
3. High speed stepping up to 20,000 PPS on three concurrent motors.
4. Very low speed stepping down to one step every 72 minutes.
5. Accurate step pulse timing with minimal timebase jitter.
6. Constant speed mode (square profile).
7. Real time linear acceleration (trapeziod & triangle profiles).
8. Synchronize motors to arrive at the destination at the same time.
9. Primitive functions for setting direction, driver enable, single stepping, & end-stop detection.
10. Built-in homing function. Interrupt driven / non-blocking.
11. Interrupt driven step generation and planning operates in the background (multi-tasking).
12. Library allocates resources dynamically for multiple instances.
13. User programmed timer interrupt callback.
14. Fast I/O functions available to your program. See the jsio.h file.

## THEORY OF OPERATION

Each instance of the library uses one of the ATmega 16-bit timers. In the case of the ATmega 2560 there are 4 16-bit timers. The timer (1, 3, 4, 5) is selected in the template structure during initialization. Each instance must use a different timer than any previous instance.

The linear acceleration profiles are based on information from this source:

<https://www.embedded.com/design/mcus-processors-and-socs/4006438/Generate-stepper-motor-speed-profiles-in-real-time>

The math for calculating linear acceleration is well published but in embedded systems with modest performance CPU's it is not possible to do divisions and floating point math in real time. This library utilizes a lookup table scheme which maintains a high degree of accuracy while accomplishing the equivalent of two square root calculations and floating point division in about 3 microseconds.

## INSTALLATION

1. Download the latest library version from: <https://github.com/johnny49r/jStepper> as a .ZIP file. It will probably be named jStepper-master.ZIP.
2. Place the ZIP file in the Arduino/libraries folder.
3. Launch the Arduino IDE and goto Sketch → Include Library → Add .ZIP Library...
4. Select the ZIP file in your Arduino/libraries folder. The IDE should create a folder called 'jStepper-master' and unpack all files in that folder. Your sketch should also contain a new directive '#include <jStepper.h>'. If not you will need to add it manually.
5. The library can also be added manually by creating a new folder in your Arduino/libraries folder and extract the contents of the ZIP file into the new folder. Close the IDE and restart. You should now find the library is available to add to your sketch.

## VERSION HISTORY

### V-1.01 10/03/2018

- Added homing direction options to the jsMotorConfig structure. This allows homing in the opposite direction of the 'origin'.
- Small changes to the homeMotor() function to improve repeatability of endstop detection.
- Added function setStepsPerUnit() and getStepsPerUnit() which allows changing movement geometry on the fly. Also needed to support g-code command M92.
- Added new command planMoves() which handles all the movement calculations, acceleration profiles, and motor synchronization. This can be called while motor(s) are running to prepare for the next movement (runMotors). This avoids the overhead of doing the planning at execution time. Additionally the runMotors() now has a new argument which dictates if the planner needs to be invoked from runMotors().
- Other small improvements and code cleaning.

### 1.0 Initial release.

## USAGE

1. The library include directive (#include <jStepper.h>) should have been added when the IDE added the library. If not add this manually to the sketch or the main header file in your program.
2. Create an instance of the library. Example: **jStepper jstep0;**
3. Create a template structure. Example: **jsMotorConfig mGroup0;** See the jsconfig.h for details of the configuration.
4. Initialize the structure items which define the hardware pin assignments for driver signals and other related geometry.

5. Pass the structure to the library in the 'begin' function. Example: ***jstep0.begin(mGroup0);***
6. The library is now initialized for a group of 3 motors and is ready to accept commands.
7. If more than 3 motors are to be controlled, another instance of the library and associated template can be added. Example: ***jStepper jstep1; jsMotorConfig mGroup1;***
8. Pass the second structure to the second library instance using its 'begin' function. Example: ***jstep1.begin(mGroup1);***

## **PUBLIC CONSTANTS**

### **Motor Constants**

MOTOR\_0  
 MOTOR\_1  
 MOTOR\_2  
 MOTOR\_ALL        *// Set all motors in group to the same state*

MOTOR\_DIRECTION\_IN – defined as moving away from the origin.  
 MOTOR\_DIRECTION\_OUT – defined as moving towards the origin.

MOTOR\_ENABLE  
 MOTOR\_DISABLE

### **Timer Constants**

TIMER\_SEL\_1        *// available timer assignments*  
 TIMER\_SEL\_3  
 TIMER\_SEL\_4  
 TIMER\_SEL\_5

TMR\_1\_CMPA        *// enumerations for user interrupt callbacks*  
 TMR\_1\_CMPB  
 TMR\_1\_CMPC  
 TMR\_1\_OVF  
 TMR\_3\_CMPA  
 TMR\_3\_CMPB  
 TMR\_3\_CMPC  
 TMR\_3\_OVF  
 TMR\_4\_CMPA  
 TMR\_4\_CMPB  
 TMR\_4\_CMPC  
 TMR\_4\_OVF  
 TMR\_5\_CMPA  
 TMR\_5\_CMPB  
 TMR\_5\_CMPC  
 TMR\_5\_OVF

### **Error Constants**

ERR\_NONE - No error

ERR\_BAD\_PARAM - *Parameter is not the expected type*

ERR\_INVALID\_MOTOR - *Bad motor number – see MOTOR\_n above*

ERR\_DISABLED - *Motors can't move if disabled*

ERR\_IN\_ENDSTOP - *Motor can't move further if inside endstop*

ERR\_ENDSTOP\_NOT\_FOUND - *Homing error – no endstop detection found*

ERR\_POSITION\_UNKNOWN - *Position has not been established (homing needed)*

ERR\_OUTSIDE\_BOUNDARY - *Requested move is outside the preset MIN/MAX boundaries*

ERR\_MOTORS\_RUNNING - *Motors are currently executing (busy),*

ERR\_NULL\_PTR - *External structure or function pointer is NULL*

## **PUBLIC MEMBER FUNCTIONS**

### ***begin( \*jsMotorConfig)***

Imports a configuration template structure of type jsMotorConfig (see public types).

Initializes hardware I/O and dynamically sets timer register and timer interrupt handler assignments.

Returns: error code:

ERR\_NULL\_PTR - *bad template structure.*

ERR\_BAD\_PARAM - *timer select is not valid.*

### ***setDirection(uint8\_t motorNum, uint8\_t direction)***

Sets the desired direction for the requested motor. Use constants:

MOTOR\_DIRECTION\_IN – *move away from origin.*

MOTOR\_DIRECTION\_OUT – *move toward origin.*

### ***getDirection(uint8\_t motorNum)***

Returns the direction for the requested motor (see setDirection()).

### ***setEnabled(uint8\_t motorNum, bool enable)***

Sets the driver enable signal to T/F for the requested motor.

Returns: Nothing.

*move toward origin.*

### ***getEnabled(uint8\_t motorNum)***

Returns true if the motor driver for the requested motor is enabled.

### ***setSpeed(float speed0, float speed1, float speed2)***

Sets the speeds (in mm/sec) for all three motors. Motor speeds are initially defined in the jsMotorConfig structure (see begin()) but are overridden with setSpeed.

Default values are set in the jsMotorConfig structure passed in the begin() function.

Returns: Nothing

### ***getSpeed(uint8\_t motorNum)***

Returns the speed for the requested motor (float).

### ***setMaxSpeed(float maxSpeed0, float maxSpeed1, float maxSpeed2)***

Sets the maximum speed allowed for all three motors. If a setSpeed() value exceeds the maximum speed, it will be limited to the maximum speed.

Default values are set in the jsMotorConfig structure passed in the begin() function.

Returns: Nothing.

### ***getMaxSpeed(uint8\_t motorNum)***

Returns the maximum speed limit for the requested motor (float).

### ***setAcceleration(float accel0, float accel1, float accel2)***

Sets the acceleration/deceleration values for all three motors. If synchronized motor movements are used, the acceleration value will only be valid for the motor with the longest travel distance. Other motors will be speed matched and acceleration may or may not be applied (see runMotors).

Default values are set in the jsMotorConfig structure passed in the begin() function.

Returns: Nothing.

### ***getAcceleration(motorNum)***

Returns the acceleration value for the requested motor (float).

### ***setPosition(uint8\_t motorNum, float newPosition)***

Sets the absolute position for the given motor. This overrides the current position regardless of actual position.

Returns: ERR\_OUTSIDE\_BOUNDARY if the new value exceed the maximum position.

### ***getPosition(uint8\_t motorNum)***

Returns the current absolute position for the requested motor (float).

### ***setPositionMode(uint8\_t positionMode)***

Sets the coordinate system to relative or absolute mode. Uses constants MODE\_RELATIVE or MODE\_ABSOLUTE. Default = RELATIVE. If in relative mode, movement calculations are based on the last move. In absolute mode movement calculations are based on the origin.

Returns: Nothing.

### ***setMinPosition(float minPosition0, float minPosition1, float minPosition2)***

Sets the minimum position for each motor. The minPosition can be negative, zero, or positive.

Default values are set in the jsMotorConfig structure passed in the begin() function.

Returns: Nothing

### ***getMinPosition(uint8\_t motorNum)***

Returns the minimum position setting for the requested motor.

### ***setMaxPosition(float maxPosition0, float maxPosition1, float maxPosition2)***

Sets the maximum allowable position for each motor. The maxPosition can be negative, zero, or positive.

Default values are set in the jsMotorConfig structure passed in the begin() function.

Returns: Nothing

### ***getMaxPosition(uint8\_t motorNum)***

Returns the maximum position setting for the requested motor.

### ***isPositionKnown(uint8\_t motorNum)***

Returns true if the current position is valid for the requested motor. Position becomes valid when the motor is homed (see homeMotor()). This value can also be set/reset using the setKnownPosition() function.

### ***setKnownPosition(uint8\_t motorNum, bool mValid)***

Sets the motor position valid state for the requested motor. This overrides any prior settings made during the homeMotor() function.

### ***runMotors(float newPos0, float newPos1, float newPos2, bool mSync, bool plan)***

Starts one to three motors running to the new position(s). If ***mSync*** = true the motor speeds are synchronized to match the motor with the longest move duration. This ensures that all motors start and end together. This is true even if all motors have different speeds and acceleration values set. Motors that have acceleration specified may not use acceleration after speeds are recalculated.

If ***plan*** = true the planMoves() function is called inside the runMotors() function. If false, the user must call the planMoves() function prior to runMotors().

Note: This command returns immediately and the user program is not blocked. The user program can check if the operation is complete using isRunning() function.

Returns possible error codes:

ERR\_MOTORS\_RUNNING – indicates previous runMotors() or homeMotor() is still busy.

ERR\_OUTSIDE\_BOUNDARY – new position is outside boundaries of setMaxPosition().

ERR\_DISABLED – motor driver(s) are not yet enabled.

ERR\_POSITION\_UNKNOWN – motor position has not been validated.

### ***planMoves(float pos0, float pos1, float pos2, bool mSync)***

This function is normally called by runMotors() to plan the moves. The user can call this function in advance of a move to preplan the next move and avoid any overhead incurred in this function.

This function does the work of calculating the geometry of the move, generate acceleration profiles, and synchronize motor speeds.

### ***homeMotor(uint8\_t motorNum, uint16\_t homingSpeed)***

Forces the requested motor to locate the home endstop detector. When the motor has reached the endstop its position is validated.

Note: This command returns immediately and the user program is not blocked. The user program can check if the operation is complete using isPositionKnown() function.

Returns possible error codes:

ERR\_MOTORS\_RUNNING – indicates previous homeMotor() or runMotors() command is still busy.

### ***stepMotor(uint8\_t motorNum)***

Issues one step pulse to the requested motor in the direction set using `setDirection()` function.

Return possible error codes:

ERR\_DISABLED – motor driver(s) are not yet enabled.

ERR\_INVALID\_MOTOR – not a valid motor.

### ***quickStop(uint8\_t motorNum)***

Forces an immediate halt of the requested motor. Stepping will stop and the driver will be disabled.

Returns: Nothing.

### ***atMinEndStop(uint8\_t motorNum)***

Returns true if the MIN endstop signal is active. The active state is set in the `jsMotorConfig` structure (see `begin()`).

### ***atMaxEndStop(uint8\_t motorNum)***

Returns true if the MAX endstop signal is active. The active state is set in the `jsMotorConfig` structure (see `begin()`).

### ***addTimerCallback(uint8\_t isrVect, void \*callback)***

Allows the user access to unused timer interrupt vectors. This is somewhat like the Arduino `attachInterrupt()` function. The `isrVect` is one of the constants:

TMR\_1\_CMPA  
TMR\_1\_CMPB  
TMR\_1\_CMPC  
TMR\_1\_OVF  
TMR\_3\_CMPA  
TMR\_3\_CMPB  
TMR\_3\_CMPC  
TMR\_3\_OVF  
TMR\_4\_CMPA  
TMR\_4\_CMPB  
TMR\_4\_CMPC  
TMR\_4\_OVF  
TMR\_5\_CMPA  
TMR\_5\_CMPB  
TMR\_5\_CMPC  
TMR\_5\_OVF

‘***callBack***’ is a pointer to an external function. Example:

```
addTimerCallback(TMR_3_CMPA, &yourISRhandler);
```



### ***setStepsPerUnit(uint16\_t su0, uint16\_t su1, uint16\_t su2)***

Sets the motor steps per unit of movement. Example: *setStepsPerUnit(100, 100, 120)*. If the units are in millimeters, this sets motors 0 & 1 = 100 steps / MM and motor 2 = 120 steps / MM.

This command overrides the initial values set in the jsMotorConfig structure (see begin()).

### ***getStepsPerUnit(uint8\_t motorNum)***

Returns the steps per unit setting for the requested motor.