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-taksing).
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

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***;
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

<u>Motor Constants</u>

MOTOR_0
MOTOR_1
MOTOR_2
MOTOR_ALL // Set all motors in group to the same state

MOTOR_DIRECTION_IN
MOTOR_DIRECTION_OUT

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 - M*otor 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 if bad template structure.

ERR_BAD_PARAM if 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.

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 setting made during the homeMotor() function.

runMotors(float newPos0, float newPos1, float newPos2, bool mSync)

Starts one to three motors running to the new position(s). If ***mSync*** is 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.

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 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.

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_MOTORS_RUNNING – indicates previous homeMotor() or runMotors() command is still busy.

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 acces 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);

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

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