PowerStep Library Documentation

Marwan Buhdima, Alberts NV

19/07/2018

Abstract

This is a documentation for the POWERSTEP01 library. The original Library is available on `github.com/cerimis/powerstep01`. This is a documentation for a modified version of the original library.
This modified library can be found on `github.com/0xD9D0/powerstep01`

Hilight on modifications

- Added static variable SSPin : array of SS pins used for devices. This allows direct communication to each stepper driver separately on SPI.
- Modified all member functions to accomodate writing to specific device
- Modified initializers (Begin function for Powerstep).
- Added public member function SetMaxSpeed to class POWERSTEP01
- Added public member function setminspeed to class POWERSTEP01
- Added public member function maxspeedcalc to class POWERSTEP01
- Added public member function minspeedcalc to class POWERSTEP01
- Added public member function setAcc to class POWERSTEP01
- Added public member function calcacc to class POWERSTEP01
- Added public member function setDec to class POWERSTEP01
- Added public member function calcdec to class POWERSTEP01
- Added public member function selectMicrosteps to class POWERSTEP01 : this member function maps the existing function to the one we use in code.
- Added public member functions SetfullSpeed, FSCalc to class POWERSTEP01. Added an alias for SetFullSpeed which is setThresholdSpeed
- Added public member function free() as an alias to CmdHardHiz in class POWERSTEP01
- Added public member function GoUntil to class POWERSTEP01 : constant speed movement

- Added public member function getMark to class POWERSTEP01 : this function returns the position of the mark.
- Added public member function IsBusy to class POWERSTEP01 : this function is a public alternative to the private member function IsDeviceBusy

Tips on usage

- A test code is attached with the library to show the proper initialization steps.
- Each object can handle up to 7 different stepper drivers.
- Before using the library you should modify powerstep01_target_config.h to edit the initial values for each device.
- You must pass an array of SSPins used to the object constructor along with the number of devices.
- To All testing have been done on IHM03A1 board from ST.

Main Code

Below is the code added to the file powerstep01.cpp. All functions have a small brief that indicates their functionality in addition to the inputs and outputs handled by the function.

Code added to powerstep01.cpp

```

/******************************************************//**
* @brief Sets maximum speed of the stepper in the
* MAX_SPEED register. if a command tried setting a speed more than this the motor
* will simply run at this speed. The value is 0x041 on power up
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecond steps per second
* @retval none
*********************************************************/
void POWERSTEP01::setMaxSpeed(uint8_t DeviceId, unsigned long stepsPerSecond)
{
unsigned long integerSpeed = maxSpeedCalc(stepsPerSecond);
CmdSetParam(DeviceId, POWERSTEP01_MAX_SPEED, integerSpeed);
return;
}


/******************************************************//**
* @brief calculates the speed by converting the value
* from steps/s to a 10 bit value. the maximum for 10 bits is 0x03ff
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecond steps per second
* @retval unsigned long steps per tick
**********************************************************/
unsigned long POWERSTEP01::maxSpeedCalc (unsigned long stepsPerSec)
{
unsigned long temp = ceil(stepsPerSec* .065536);
if (temp > 0x000003FF) return 0x000003FF;
else return temp;
}


/*******************************************************//**
* @brief Sets minimum speed of the stepper in the
* MIN_SPEED register.for any move or goto function where no speed is specified
* this value will be used
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecond steps per second
* @retval none
**********************************************************/
void POWERSTEP01::setMinSpeed(uint8_t DeviceId, int speed){

CmdSetParam(DeviceId, POWERSTEP01_MIN_SPEED, MinSpeedCalc(speed));
}

/*******************************************************//**
* @brief The value in the MIN_SPEED register is [(steps/s)*(tick)]/(2^-24) where
   tick is
* 250ns (datasheet value)- 0x000 on boot.
* Multiply desired steps/s by 4.1943 to get an appropriate value for this register
* This is a 12-bit value, so we need to make sure the value is at or below 0xFFF.
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecond steps per second
* @retval unsigned long steps per tick
**********************************************************/
unsigned long POWERSTEP01::MinSpeedCalc(float stepsPerSec){

float temp = stepsPerSec * 4.1943;
if( (unsigned long) long(temp) > 0x00000FFF) return 0x00000FFF;
else return (unsigned long) long(temp);
}


/*******************************************************//**
* @brief Configure the acceleration rate, in steps/tick/tick. There is also a DEC
   register;
* both of them have a function (AccCalc() and DecCalc() respectively) that convert
* from steps/s/s into the appropriate value for the register. Writing ACC to 0xfff
* sets the acceleration and deceleration to 'infinite' (or as near as the driver can
*  manage). If ACC is set to 0xfff, DEC is ignored. To get infinite deceleration
* without infinite acceleration, only hard stop will work.
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecondpersecond steps per second per second
* @retval none
*******************************************************/
void POWERSTEP01::setAcc(uint8_t DeviceId, unsigned long stepsPerSecondPerSecond)
{
unsigned long integerAcc = accCalc(stepsPerSecondPerSecond);
CmdSetParam(DeviceId, POWERSTEP01_ACC, integerAcc);
return;
}


/******************************************************//**
* @brief The value in the ACC register is [(steps/s/s)*(tick^2)]/(2^-40) where tick is
* 250ns (datasheet value)- 0x08A on boot.
* Multiply desired steps/s/s by .137438 to get an appropriate value for this register.
* This is a 12-bit value, so we need to make sure the value is at or below 0xFFF.
* &param[in] stepsPerSecondPerSecond steps per second per second
* @retval unsigned long 12 bit acc
*******************************************************/
unsigned long POWERSTEP01::accCalc(unsigned long stepsPerSecPerSec)
{
unsigned long temp = stepsPerSecPerSec * 0.137438;
if(temp > 0x00000FFF) return 0x00000FFF;
else return temp;
}

/******************************************************//**
* @brief Configure the acceleration rate, in steps/tick/tick. There is also a DEC register;
* both of them have a function (AccCalc() and DecCalc() respectively) that convert
* from steps/s/s into the appropriate value for the register. Writing ACC to 0xfff
* sets the acceleration and deceleration to 'infinite' (or as near as the driver can
*  manage). If ACC is set to 0xfff, DEC is ignored. To get infinite deceleration
* without infinite acceleration, only hard stop will work.
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecondpersecond steps per second per second
* @retval none
*******************************************************/
void POWERSTEP01::setDec(uint8_t DeviceId, unsigned long stepsPerSecondPerSecond)
{
unsigned long integerDec = decCalc(stepsPerSecondPerSecond);
CmdSetParam(DeviceId, POWERSTEP01_DEC, integerDec);
return;
}


/******************************************************//**
* @brief The calculation for DEC is the same as for ACC. Value is 0x08A on boot.
* This is a 12-bit value, so we need to make sure the value is at or below 0xFFF.
* Multiply desired steps/s/s by .137438 to get an appropriate value for this
    register.
* This is a 12-bit value, so we need to make sure the value is at or below 0xFFF.
* &param[in] stepsPerSecondPerSecond steps per second per second
* @retval unsigned long 12 bit dec
**********************************************************/
unsigned long POWERSTEP01::decCalc(unsigned long stepsPerSecPerSec)
{
unsigned long temp = stepsPerSecPerSec * 0.137438;
if(temp > 0x00000FFF) return 0x00000FFF;
else return temp;
}

/*******************************************************//**
* @brief This sets the full speed register to a certain speed
* the full speed is the speed that if surpassed the motor will cease microstepping
* and go to full-step
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] stepsPerSecond steps per second
* @retval none
**********************************************************/
void POWERSTEP01::setFullSpeed(uint8_t DeviceId, float stepsPerSecond)
{
unsigned long integerSpeed = FSCalc(stepsPerSecond);
CmdSetParam(DeviceId, POWERSTEP01_FS_SPD, integerSpeed);
}

// Alias for setFullSpeed
void POWERSTEP01::setThresholdSpeed(uint8_t DeviceId,float stepsPerSecond)
{setFullSpeed(DeviceId,stepsPerSecond);}

/*******************************************************//**
* @brief the value in the FS_SPD register is ([(steps/s)*(tick)]/(2^-18))-0.5 where
    tick is
* 250ns (datasheet value)- 0x027 on boot.
* Multiply desired steps/s by .065536 and subtract .5 to get an appropriate value
    for this register
* This is a 10-bit value, so we need to make sure the value is at or below 0x3FF.
* &param[in] stepsPerSecond steps per second
* @retval unsigned long steps per tick
**********************************************************/
unsigned long POWERSTEP01::FSCalc(float stepsPerSec)
{
float temp = (stepsPerSec * .065536)-.5;
if( (unsigned long) long(temp) > 0x000003FF) return 0x000003FF;
else return (unsigned long) long(temp);
}


/*******************************************************//**
* @brief This function maps the input from
* the main function code to the selectstpeMode function to modify the microstepping
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* &param[in] microSteps this parameter can be 1,2,4,8,16,32,64,128
* @retval none
*********************************************************/
void POWERSTEP01::setMicroSteps(uint8_t deviceId, int microSteps){

switch (microSteps)
{
case 1:
SelectStepMode(deviceId, STEP_MODE_FULL);
break;
case 2:
SelectStepMode(deviceId, STEP_MODE_HALF);
break;
case 4:
SelectStepMode(deviceId, STEP_MODE_1_4);
break;
case 8:
SelectStepMode(deviceId, STEP_MODE_1_8);
break;
case 16:
SelectStepMode(deviceId, STEP_MODE_1_16);
break;
case 32:
SelectStepMode(deviceId, STEP_MODE_1_32);
break;
case 64:
SelectStepMode(deviceId, STEP_MODE_1_64);
break;
case 128:
SelectStepMode(deviceId, STEP_MODE_1_128);
break;
default:
SelectStepMode(deviceId, STEP_MODE_FULL);
}
return;
}

/******************************************************//**
* @brief Issues PowerStep01 Go Until command
* @param[in] deviceId (from 0 to MAX_NUMBER_OF_DEVICES-1 )
* @param[in] action ACTION_RESET or ACTION_COPY
* @param[in] direction movement direction
* @param[in] speed in 2^-28 step/tick
* @retval None
*********************************************************/
void POWERSTEP01::CmdGoUntil(uint8_t deviceId,
motorAction_t action,
motorDir_t direction,
uint32_t speed)
```

```
{
SendCommand(deviceId,
(uint8_t)POWERSTEP01_GO_UNTIL | (uint8_t)action | (uint8_t)direction,
speed);
}

/*******************************************************//**
* @brief  Sets the number of devices to be used
* @param[in] nbDevices (from 1 to MAX_NUMBER_OF_DEVICES)
* @retval TRUE if successfull, FALSE if failure, attempt to set a number of
* devices greater than MAX_NUMBER_OF_DEVICES
***********************************************************/
bool POWERSTEP01::SetNbDevices(uint8_t nbDevices)
{
if (nbDevices <= MAX_NUMBER_OF_DEVICES)
{
numberOfDevices = nbDevices;
return TRUE;
}
else
{
return FALSE;
}
}
```

Listing 1: Blink.ino