API Documentation for Serial Terminal Commands

Content

1. Introduction

1.1. Physical Configuration of the Serial Link

The serial terminal application requires a specific configuration for the serial link to function correctly. The only allowed configuration is as follows:

- **Baud Rate:** 115200
- **Data Bits:** 8
- **Parity:** None (N)
- **Stop Bits:** 1 (8N1 configuration)

Make sure your serial interface is set to this configuration to ensure proper communication between devices. This configuration is critical for the correct operation of the commands and to avoid data transmission errors.

1.2. General Command Structure

Examples

Provide examples of how to use the command, including different options and arguments.

1. **Basic Usage:**
2. COMMAND_NAME SUBCOMMAND ARGUMENT
3. COMMAND_NAME SUBCOMMAND
4. **With Options:**
5. COMMAND_NAME SUBCOMMAND -f ARGUMENT
6. COMMAND_NAME SUBCOMMAND -v ARGUMENT

Return Values

Describe what the command returns upon successful execution. Include data types and possible values.

Errors

List possible error messages or codes that the command might return. Provide explanations and potential solutions or troubleshooting steps.

Related Commands

- **RELATED_COMMAND_1**: Brief description.
- **RELATED_COMMAND_2**: Brief description.

Notes

Include any additional information that might be helpful, such as performance considerations, limitations, or compatibility notes.

1.3. General Command Output Format

All commands follow a standard output format. The output consists of optional values and strings, each separated by a new line (*\r\n*). The final line of the output indicates the success or failure of the command:

- **Success**: The command outputs

```
OK\r\n
```

as the last line.

- **ERROR**: The command outputs

```
FAIL\r\n
```

as the last line.

Allowed Output Types:

- **INTEGER NUMBERS**: Whole numbers without any decimal point.
- **HEXADECIMAL NUMBERS**: Prefixed with **0X**, representing numbers in **BASE 16**.
- **FLOAT NUMBERS**: Numbers with a decimal point.
- **STRINGS**: Natural language text.

Output Format Example Output:

```
123\r\n
0x1A3F\r\n
45.67\r\n
Example string output\r\n
OK\r\n
```

Or, in case of an error:

```
Invalid argument input\r\n
FAIL\r\n
```

2. Console Library and Library Dependencies

To use the serial terminal application and its commands, ensure that the following libraries are installed and properly configured in your development environment:

- **LIBRARY 1:** Brief description of its purpose and version requirements.
- **LIBRARY 2:** Brief description of its purpose and version requirements.
- **LIBRARY 3:** Brief description of its purpose and version requirements.

Ensure that these libraries are compatible with your system and are up-to-date to avoid any compatibility issues.

1.4. Registering Custom Commands

Developers can extend the functionality of the console by registering their own commands using the

```
CONSOLE_RegisterCommand()
```

function. This function allows you to add new commands to the console interface. By registering a command handler and passing the command name. The console processor parses all user arguments given by the user and passes them in a decoded string array to the command handler function.

Function Prototype

```
int CONSOLE_RegisterCommand(ConsoleHandle_t h, char* cmd, char* help,
CONSOLE_CommandFunc func, void* context);
```

Parameters

- **H:** A pointer to the the instance of the console processor which has been previously generated by calling the CONSOLE_CreateInstance() function. The handle stores all relevant information about the runtime of the console processor.
- **CONTEXT:** A void* pointer that can be used to pass context-specific data to the command function. This can be NULL if no context is needed.
- **CMD**: A char* representing the name of the command. This is the string users will type to invoke the command.
- **HELP**: A char* providing a brief description of the command. This text is displayed when users request help.
- **FUNC**: A function pointer which represents the command function. This function is invoked whenever a user enters the command name given by the cmd argument.

Return Value

- **0**: Indicates successful registration of the command.
- **-1**: Indicates an error occurred during registration.

Function Prototype of a command handler

The command function prototype must follow the following format and the argument list must match with the described one in this documentation. A command handler function in this format can be passed as pointer when calling the CONSOLE_RegisterCommand function.

```
int MyCommandHandler(int argc, char** argv, void* context);
```

Parameters

- **ARGC**: A integer value which gives the number of parsed arguments to the command handler. All inputs separated by a space or tab are split into an argument and so eacah input after the command itself is stored into its own argument element in the argv array in case it is separated by a space.
- **CONTEXT:** A void* pointer that can be used to pass context-specific data to the command function. This can be NULL if no context has been registered before by the CONSOLE_CommandRegister function.
- **ARGV**: A char* array (char**) representing the list of string arguments of the command input. This is the string list users will type as arguments when invoking the command.

Return Value

- **0**: Indicates successful execution of the command.
- **!0**: Indicates an error occurred during execution.

Example Usage

Here is an example of how to register a custom command and how the number of arguments passed to the function can be printed. This is a quite simple example but it can be used as entry point to create own more complex commands. The implementation is only limited by the capability of the used stdlib because all other input and output functionality is kept in the execution context of the function handler as long as it has not been returned to the console processor by leaving the function with a return statement. The execution of such a function is blocking per design; to write asynchronous behavior, one must implement asynchronous data flow mechanisms in its design and needs also some commands to poll for the state of the asynchronous command. Such polling is performed by the user via a console command again.

```
int MyCommandFunction(int argc, char** argv, void* context) {
    // Implement the command logic here
    printf("My custom command executed with %d arguments!\n", argc);
    return 0;
}

int main() {
    // imagine the console handle has been created elsewhere
    if (CONSOLE_RegisterCommand(h, "mycommand", "Executes my custom
command", MyCommandFunction, NULL) == 0) {
        printf("Command registered successfully.\n");
    } else {
        printf("Failed to register command.\n");
    }
    // do something else here
    …
    return 0;
}
```

This example demonstrates how to define a command function and register it with the console. Ensure that the command name is unique and descriptive to avoid conflicts and provide clear help text for users.

1.5. Registering Alias Commands

Developers can extend the functionality of the console by registering their own alias which maps a short command to a longer variant with all its arguments defined in the alias string, using the

```
CONSOLE_RegisterAlias()
```

function. This function allows you to add new alias commands to the console interface. Additional Arguments after the alias command are parsed and passed to the original command and so it’s a simple copy and forwarding algorithm behind an alias command input. The alias can also be defined via the console at runtime with the following call. An alias can also point to an alias and also the commands are substituted properly.

```
alias [cmd] [full cmd with args]
```

3. The Capability Command

1.6. Presence of commands

The capability function prints an array of bits in which each bit represents the presence of a command, subcommand, command flag/switch or a group of commands.

The bits can be checked using the

```
capability
```

command, which returns a bit array indicating the presence of implemented commands. These feature bits are defined as follows and can be counted from the right (0-indexed)

Capability name	Bit position
CAP_HAS_SPINDLE	0
CAP_HAS_SPINDLE_STATUS	1
CAP_HAS_STEPPER	2
CAP_HAS_STEPPER_MOVE_RELA	3
CAP_HAS_STEPPER_MOVE_SPEED	4
CAP_HAS_STEPPER_MOVE_ASYNC	5
CAP_HAS_STEPPER_STATUS	6
CAP_HAS_STEPPER_REFRUN	7
CAP_HAS_STEPPER_REFRUN_TMOUT	8
CAP_HAS_STEPPER_REFRUN_SKIP	9
CAP_HAS_STEPPER_REFRUN_ENABLED	10
CAP_HAS_STEPPER_RESET	11
CAP_HAS_STEPPER_POSITION	12
CAP_HAS_STEPPER_CONFIG	13
CAP_HAS_STEPPER_CONFIG_TORQUE	14
CAP_HAS_STEPPER_CONFIG_THROVERCURR	15
CAP_HAS_STEPPER_CONFIG_POWERENA	16
CAP_HAS_STEPPER_CONFIG_STEPMODE	17
CAP_HAS_STEPPER_CONFIG_TIMEOFF	18
CAP_HAS_STEPPER_CONFIG_TIMEON	19
CAP_HAS_STEPPER_CONFIG_TIMEFAST	20
CAP_HAS_STEPPER_CONFIG_MMPERTURN	21
CAP_HAS_STEPPER_CONFIG_POSMAX	22
CAP_HAS_STEPPER_CONFIG_POSMIN	23
CAP_HAS_STEPPER_CONFIG_POSREF	24
CAP_HAS_STEPPER_CONFIG_STEPSPERTURN	25
CAP_HAS_STEPPER_CONFIG_CANCEL	26

Example

If the capability command returns

```
1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,…\r\n
OK
```

, the second bit from the right (0-indexed) states the presence of spindle status command. Which means that this command is implemented as described in this document and can be called by the user via the console!

4. Command Overview Spindle

1.7. Subcommand Start

Command Name:

spindle start

Description

The spindle can be started by the user with the start subcommand. There is an additional RPM argument required which can be negative or positive. The command is formatted as integer base 10. In case the RPM value is less than the minimum RPM or greater than the maximum one, the value is capped and the spindle gets started with the capped value. With the spindle status command, the user can get the current state and RPM of the spindle.

Syntax

spindle start [RPM]

Parameters

- **RPM**: This is the integer base 10 number which can be positive or negative and which is required to start the spindle with the given RPM.

Options/Flags

The command has no additional flags

Preconditions

There are no preconditions to start or stop the spindle. In case the command is called multiple times without a stop command, the RPM value is adapted and the command succeeds as well.

Required Feature Bit

The `CAP_HAS_SPINDLE` feature bit is required for this command to be available.

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibSpindle**: must be fully integrated to get this command.
- **PWM Generator (Timer)**: There is a Timer required with two separate PWM channels to control the external H-Bridge.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Spindle Library.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions:** The user must provide the user specific functions for changing PWM duty cycle, enabling or disabling the PWM timer and to change the direction of the spindle by changing PWM outputs.

1.8. Subcommand Stop

Command Name:

spindle stop

Description

The stop command is used to stop the spindle in case it has been started. The command can be called multiple times and should always succeed

Syntax

spindle stop

Options/Flags

The command has no additional flags

Preconditions

There are no preconditions to start or stop the spindle. In case the command is executed even if the spindle stopped already, the command succeeds without doing anything else.

Required Feature Bit

The `CAP_HAS_SPINDLE` feature bit is required for this command to be available.

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibSpindle**: must be fully integrated to get this command.
- **PWM Generator (Timer)**: There is a Timer required with two separate PWM channels to control the external H-Bridge.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Spindle Library.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions:** The user must provide the user specific functions for changing PWM duty cycle, enabling or disabling the PWM timer and to change the direction of the spindle by changing PWM outputs.

1.9. Subcommand Status

Command Name:

spindle status

Description

The status command is used to gather some information about the current state of the spindle. It returns the current PWM state (turning or not) and the current RPM requested by the user which is clipped to stay within the minimum and maximum RPM.

Syntax

spindle status

Output Format

The command returns an integer value with 0 or 1 in the first line, the RPM requested by the user as integer base 10 value with optional negative minus prefix in the second line and a “OK” string as last line.

```
1\r\n
1800\r\n
OK\r\n
```

Required Feature Bit

The `CAP_HAS_SPINDLE` feature bit is required for this command to be available.

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibSpindle**: must be fully integrated to get this command.
- **PWM Generator (Timer)**: There is a Timer required with two separate PWM channels to control the external H-Bridge.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Spindle Library.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions:** The user must provide the user specific functions for changing PWM duty cycle, enabling or disabling the PWM timer and to change the direction of the spindle by changing PWM outputs.

5. Command Overview Stepper

1.10. Subcommand Reset

Command Name:

stepper reset

Description

This command resets any error state and forces the stepper controller back to its initial state. All further commands to bring up the step controller to working movement needs to be executed again after this command.

Syntax

stepper reset

Preconditions

There are no preconditions to reset the stepper. This command must be accepted always and in each state.

Required Feature Bit

The `CAP_HAS_STEPPER_RESET` feature bit is required for this command to be available.

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the user specific function for changing the HW Reset Pin of the stepper IC.

1.11. Subcommand Reference

Command Name:

stepper reference

Description

The reference command is required at least once on power up to find the reference mark of the system and therefore to calculate absolute position requests properly. This command has multiple options which can be implemented to change the behavior of the reference run.

Syntax

stepper reference [FLAGS]

Options/Flags

- *-t*: [OPTIONAL] Flag which passes an additional timeout in seconds. The flag requires an integer behind the -t option separated by a space.
- *-e*: [OPTIONAL] Flag which forces the power output to be enabled after the reference run has been completed. This option is an implicit step which is required by a separate command in case this flag has not been specified.
- *-s*: [OPTIONAL] Flag which skips the reference run and sets the initial position on the current physical location of the spindle.

Preconditions

This command requires an error free system. The command can be executed multiple times but it fails if the system is in error state or an error happens while reference run performs.

Required Feature Bit

The `CAP_HAS_STEPPER_REFRUN` feature bit is required for this command to be available. The flags are separated with additional Capability bits as follows:

- *-t*: CAP_HAS_STEPPER_REFRUN_TMOUT
- *-e*: CAP_HAS_STEPPER_REFRUN_ENABLED
- *-s*: CAP_HAS_STEPPER_REFRUN_SKIP

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the user specific function for enabling the power states and generating the PWM signals to step the library. Additionally, the user must provide the communication function to interact with the stepper driver IC.

Reference run flow

The reference run moves slowly to the end switch until it hits the switch. It then stops and sets the initial reference position. In case the spindle already hits the end switch when starting the reference run, it first has to move at least 2mm away from the switch and then starts the regular process. The reference run must handle errors from the stepper IC as well.

1.12. Subcommand Position

Command Name:

stepper position

Description

The position command returns the absolute position of the system in mm as float number. The position is valid in case the system is not in error state.

Syntax

stepper position

Preconditions

This command requires an error free system. The command can be executed multiple times in all states except the error state

Output Format

The command returns an float value of the position in mm

```
100.0000\r\n
OK\r\n
```

Required Feature Bit

The `CAP_HAS_STEPPER_POSITION` feature bit is required for this command to be available

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the communication function to interact with the stepper driver IC.

1.13. Subcommand Status

Command Name:

stepper status

Description

The status command returns multiple values in the console. The first value is the internal state machine state as integer hexadecimal coded. Values of the state machine can be found in the state diagram at the end of this document. The second value describes the status of the stepper driver which lists the internal error bits in an integer value hexadecimal coded. Each bit is coded as follows:

- 0: DIRECTION bit
- 1: HIGH-Z Driver Output bit
- 2: NOTPERF_CMD bit
- 3: OVERCURRENT_DETECTION bit
- 4: ONGOING bit
- 5: TH_SD bit
- 6: TH_WARN bit
- 7: UVLO Undervoltage Lockout bit
- 8: WRONG_CMD bit

The last value is an integer decimal coded value which can be 0 or 1. This value represents an asynchronous pending operation (movement). Which can be aborted with the cancel command.

Syntax

stepper status

Preconditions

The command can be executed multiple times in all states.

Output Format

The command returns an float value of the position in mm

```
0x4\r\n
0x1\r\n
0\r\n
OK\r\n
```

Required Feature Bit

The `CAP_HAS_STEPPER_STATUS` feature bit is required for this command to be available

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the communication function to interact with the stepper driver IC.

1.14. Subcommand Cancel

Command Name:

stepper cancel

Description

The cancel command is used to stop movement in case it has been started asynchronously with the stepper move command and the optional -a flag. This asynchronous movement allows the user to enter other commands while the stepper is moving. This asynchronous operation is then canceled with the cancel command in case the user wants to stop the movement before it reaches its final position.

Syntax

stepper cancel

Preconditions

The command can be executed multiple times in an error free and ready to move system. It returns success even if there is no movement ongoing any more.

Required Feature Bit

The `CAP_HAS_STEPPER_CANCEL`feature bit is required for this command to be available

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the communication function to interact with the stepper driver IC.

1.15. Subcommand Move

Command Name:

stepper move

Description

The move command is used to move the spindle with the stepper motor to a given absolute position. The controller must always check that the requested position stays within the mechanical range, other wise it must fail. The movement can be speed regulated and is able to execute the movement asynchronously. The position is given as integer or float number separated by a dot. The value can be negative in case the relative flag is supported! Is the speed larger or lower than the mechanically supported speed limits, it is capped to the next possible and valid value without an error

Syntax

stepper move <POSITION> [FLAGS]

Options/Flags

- *-s*: [OPTIONAL] Flag which passes an additional speed argument in mm/min. The flag requires an integer behind the -s option separated by a space.
- *-r*: [OPTIONAL] Flag which instructs the controller to move relative by the given offset in mm instead to move to the given absolute position in mm.
- *-a*: [OPTIONAL] Flag which executes the movement asynchronously and immediately returns to the console.

Preconditions

The command can only be executed in an error free and ready to move system. In case the command has been issued with the asynchronous flag, it has to fail until the movement has been done when the user issues the command again multiple times while the stepper is still moving! In case it is executed synchronously, it must operate accordingly and returns after the position has been reached.

Required Feature Bit

The `CAP_HAS_STEPPER` feature bit is required for this command to be available. For the additional flags, the following feature bits are required:

- *-s*: CAP_HAS_STEPPER_MOVE_SPEED
- *-r*: CAP_HAS_STEPPER_MOVE_RELA
- *-a*: CAP_HAS_STEPPER_MOVE_ASYNC

Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the communication function to interact with the stepper driver IC. And also, the PWM generation functions to move the stepper via the stepper driver. All GPIOs and additional platform functions must be provided.

1.16. Subcommand Config

Command Name:

stepper config

Description

The config command allows to change runtime parameters of the stepper ecosystem dynamically without recompiling the Firmware. The parameters are set volatile so after each stepper reset command the parameters are lost!

One parameter is a high priority parameter because it is used to enable or disable the output drivers and so this is part of the general stepper handling and part of the bring up process.

Syntax

stepper config <PARAM_NAME> [-v <PARAM_VALUE>]

Options/Flags

- *-v*: [OPTIONAL] Flag is used to change the parameters value by giving an additional value. The format depends on the parameter.

Preconditions

The command can only be executed in an error free and ready to move system. Some parameters can not be changed while the output drivers are enabled, See possible parameters chapter!

Possible Parameters

The stepper config can provide the maximum of the following parameters. Not all parameters can be changed in all modes of operation. The parameters, its formats and the preconditions are listed below:

Parameter name	Format	Changeable while active
torque	Integer	yes
throvercurr	Integer	yes
powerena	Integer	yes
stepmode	Integer	yes
timeoff	Integer	no
timeon	Integer	no
timefast	Integer	no
mmperturn	Float	yes
posmax	Float	yes
posmin	Float	yes
posref	Float	yes
stepsperturn	Integer	yes

Required Feature Bit

The `CAP_HAS_STEPPER_CONFIG` feature bit is required for this command to be available. For all further parameters a separate capability bit is used to decode the presence as follows:

`Capability name`	Parameter name
`CAP_HAS_STEPPER_CONFIG_TORQUE`	torque
`CAP_HAS_STEPPER_CONFIG_THROVERCURR`	throvercurr
`CAP_HAS_STEPPER_CONFIG_POWERENA`	powerena
`CAP_HAS_STEPPER_CONFIG_STEPMODE`	stepmode
`CAP_HAS_STEPPER_CONFIG_TIMEOFF`	timeoff
`CAP_HAS_STEPPER_CONFIG_TIMEON`	timeon
`CAP_HAS_STEPPER_CONFIG_TIMEFAST`	timefast
`CAP_HAS_STEPPER_CONFIG_MMPERTURN`	mmperturn
`CAP_HAS_STEPPER_CONFIG_POSMAX`	posmax
`CAP_HAS_STEPPER_CONFIG_POSMIN`	posmin

`CAP_HAS_STEPPER_CONFIG_POSREF`	posref
`CAP_HAS_STEPPER_CONFIG_STEPSPERTURN`	stepsperturn

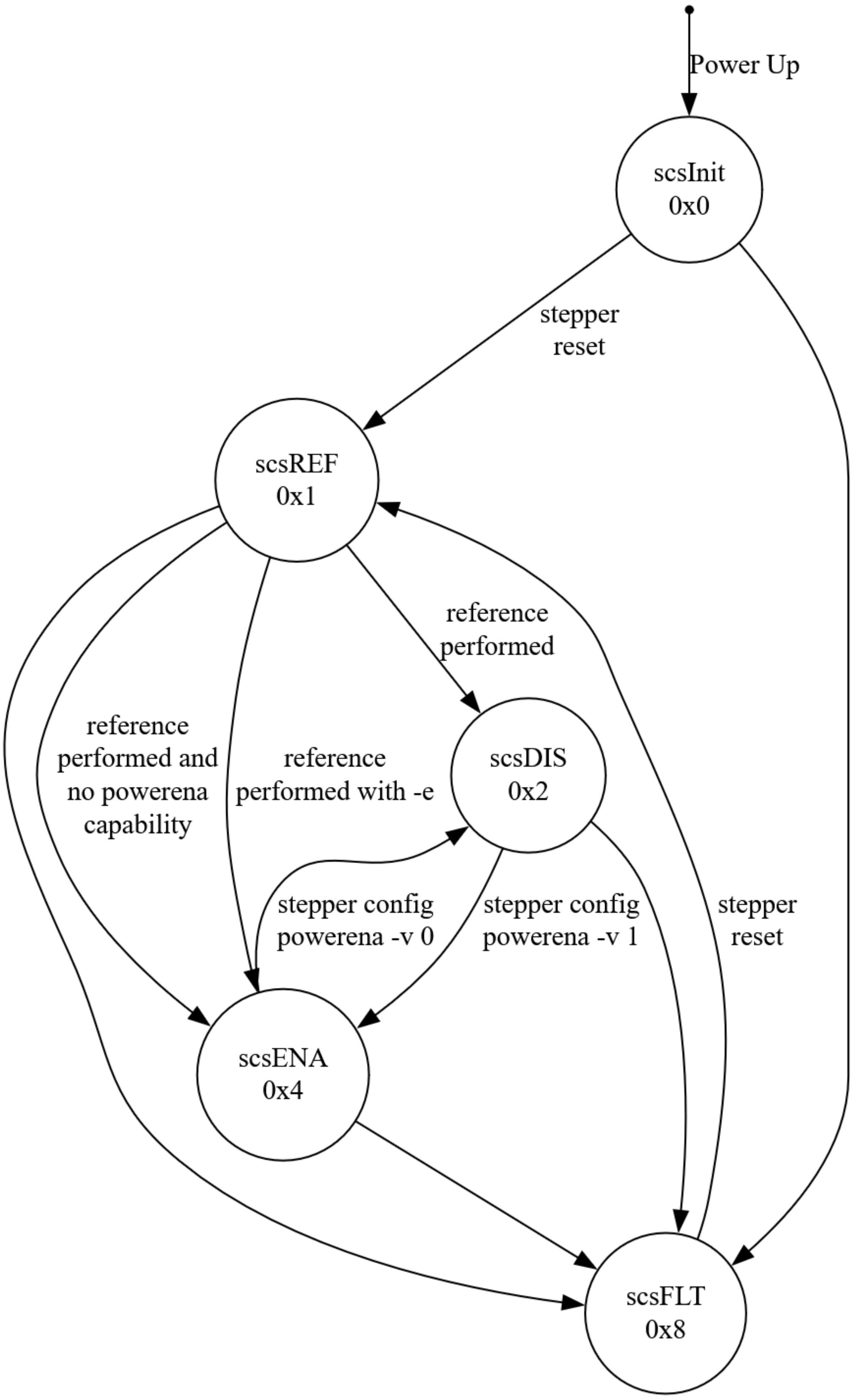
Implementation-Specific Dependencies

The following List is an overview of additional dependencies or considerations specific to the implementation of this command. This includes:

- **LibL6474**: must be fully integrated to provide this command.
- **FreeRTOS**: A OS is required to provide the basic Control- and Dataflow Objects for the Console and the Stepper Controller.
- **LibRtosConsole**: The Console library is an additional dependency for the spindle library.
- **Library-Specifc-Functions**: The user must provide the communication function to interact with the stepper driver IC.

6. Bring-Up Sequence and state machine

The bring-up sequence involves transitioning through a series of states to initialize the system. The state diagram below illustrates these states and the transitions between them.


Power Up
scsInit 0x0
stepper reset
scsREF 0x1
reference performed
scsDIS 0x2
reference performed and no powerena capability
reference performed with -e
stepper config powerena -v 0
stepper config powerena -v 1
stepper reset
scsENA 0x4
scsFLT 0x8

1.17. States and Transitions

1. **scsINIT**: The initial state when the system is powered on.
    - **Transition to scsREF**: Triggered by *stepper reset*.

2. **scsREF**: The system is setting up necessary components and prepares everything for the reference run.
    - **Transition to scsDIS**: Triggered by *stepper reference*
    - **Transition to scsENA**: Triggered by stepper reference -e or in case there is no *stepper config powerena -v 1* command implemented!

3. **scsDIS**: The system is ready to start movement but the ouptut drivers are disabled.
    - **Transition to scsENA**: Triggered by *stepper config powerena -v 1*
    - **Transition from scsENA**: Triggered by *stepper config powerena -v 0*

4. **scsENA**: The system is actively performing movement or is ready to do so.
    - **Transition to scsDIS**: Triggered by *stepper config powerena -v 0*

5. **scsFLT**: The system is not ready because an error is active
    - **Transition to scsIDLE or scsREF**: *Triggered by stepper reset*

Additional notes

The current state and the error causes can be viewed by entering *stepper status* in case the command has been implemented. A reference run must be possible from within scsREF, scsDIS and scsENA!

7. Attachments

State diagram code

```
digraph finite_state_machine {
                    //Define the nodes/states in the system and their style here


                    /*These is the entry node for the system.
                     * They are small, solid black circles.
                     * listed on the same line because they all have the same properties
                     */

                    node [shape=point,label=""]ENTRY;

                    //This line defines a new node style: the circle

                    node [shape=circle];

                    /* All the nodes defined here will be circles. Additional attributes defined
                     * for each node will be added on to the attributes listed above.
                     */

                     scsINIT[label="scsInit\n0x0"];                                    //Label attributes are placed inside the node
                     scsREF[label="scsREF\n0x1"];             //Newlines are allowed in labels
                     scsDIS[label="scsDIS\n0x2"];
                     scsENA[label="scsENA\n0x4"];
                     scsFLT[label="scsFLT\n0x8"];
                     //Below are all of the definition of the edges that connect the nodes

                     //Edges can have labels too

                     ENTRY->scsINIT [label="Power Up"];
                     scsINIT->scsREF [label="stepper\nreset"];
                     scsDIS->scsENA [label="stepper config\npowerena -v 1"];
                     scsENA->scsDIS [label="stepper config\npowerena -v 0"];
                     scsENA->scsFLT;
                     scsDIS->scsFLT;
                     scsREF->scsFLT;
                     scsREF->scsDIS [label="reference\nperformed"];
                     scsREF->scsENA [label="reference\nperformed with -e"];
                     scsREF->scsENA [label="reference\nperformed and\nno powerena\ncapability"];
                     scsINIT->scsFLT;
                     scsFLT->scsREF [label="stepper\nreset"];

}
```