|  |  |
| --- | --- |
| **DESI Positioner Firmware Revision 6.23**  DESI-1710-v23 | **desilogo.png** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rev** | **Change Notes** | **Submitted** | **Checked** | **Date** |
| 13 | Added Cmd 29 spreadsheet | MS |  | 2022-04-05 |
| 14 | Make consistent with 5.31; Added execution times  Made some corrections, put them in red text. | hdh |  | 2022-04-05 |
| 15 | need\_double\_pulse defaults FALSE | hdh |  | 2022-04-10 |
| 16 | Added description for Cmd 62 | hdh |  | 2022-04-11 |
| 17 | Added Command 55 | hdh |  | 2022-04-15 |
| 20 | Added short ramp spinup; simplified command structure | hdh |  | 2022-10-07 |
| 23 | Fixed bugs in 6.22; Simplified Cmd 59 | hdh |  | 2023-02-24 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Firmware Developers: H. Heetderks (LBL), I. Gershkovich (UofM), M. Schubnell (UofM)

**Overview**

This document describes the functionality of the DESI fiber positioner firmware loaded into the Cortex STM32F103 microcontrollers on the fiber positioner boards. Version 1.0 is based on the original ‘Open Loop’ firmware, developed by H. Heetderks at LBNL. Later versions show changes as new commands are added and some older commands are removed. Please refer to the document ‘Open Loop Description.doc’ for a conceptual overview of the firmware and a detailed description of the basic movement commands.

To aid in the understanding of the STM32F CAN registers, Chapter 24, Controller area network (bxCAN), of the STM32F1xx hardware description manual, RM0008, has been included in DESI-1710.

**Other Relevant Documents**

**DESI-doc 948 UMICH Fiber Positioner Circuit Electrical Documentation** (schematic and layout of fipos electronics board)

**DESI-doc 2764 DESI Positioner Bootloader** (This document describes the functionality of the DESI positioner bootloader. It contains the Keil development environment project folder for building and using the bootloader as well as all materials from the bootloader reviews.)

**Bootloader**

Flash memory in the STM32F103 begins at 0x08000000 in the memory space. The first 0x104C bytes contain a bootloader and that memory area is protected and cannot be modified without first using JTAG to remove the memory protection. The positioner firmware starts at address 0x08002000 and is about 0x6A40 bytes in length. On reset, execution starts at 0x08000000 implementing the bootloader. This code starts with a 2 second delay during which the user can start the bootload process by sending a CAN message with an 8 byte unlock code. (Note: for devices with failed hardware clocks, the soft clock is 9 times slower, so the wait time will be 18 seconds.) If it does not receive the command that initiates the firmware upload process within two seconds, or receives any other command, the microcontroller jumps to the start of the main firmware application. Once it enters the firmware application mode it responds to the commands detailed in this document. Please see the related DESI document 2764 for bootloader usage and commands. Starting with release v5.0, the microcontroller checks if its external oscillator circuit is stable after jumping into the firmware and reconfigures the clock to use the internal oscillator in case of an instability.

2022-09-16 Bootloader bug:

*Bootloader 3.8 which is installed on all positioners currently in the DESI instrument has a bug which limits the maximum size of firmware which can be downloaded to 0x7CFE bytes. A detailed description of the bug is as follows:*

The bootloader in the fipos has a buffer of 4,000 words, i.e. 16,000 bytes.   After it gets everything set up, the downloading code sends 4,000  Cmd 132's, each with one word of four bytes, thus filling up the buffer.

When the buffer is full, we pause sending Cmd 132's, and the bootloader in the fipos writes the buffer of data into FLASH.  This can go on for how many buffer fulls are needed to download the code.  The last buffer is typically not filled completely and so there is a special check as the buffer is being filled to see if all of the code has been received and if so it switches some pointers around to terminate the process and set up to pad the remainder of the buffer with zeros.

That is where a bug arises.

Line 139 “WriteRxBuffer(uint32\_t Start\_Address, uint16\_t  currentp)”    is the function which writes the buffer of data into FLASH.  It takes the value of *currentp* to be the number of words to be written.  *currentp* is the count of words over the entire code to be transferred,  so for the first buffer full,  *currentp* is 3999 and the write to memory works fine.  But for the second buffer full, currentp is 7999 and so the code runs off the end of the buffer and the write to FLASH fails.

In line 337, instead of currentp which is the packet count over the whole download, we should use *p* which is a count that starts at zero for each new buffer. Meanwhile, if the second buffer is not full, the pointers are switched when the end of the full download is reached and the FLASH write works properly.  So the bottom line is that we can use the current bootloader to install firmware of up to two buffers worth, or 0x7CFE bytes.

**Command Summary**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 1 | 01 | stay\_alive | stop the 60 second countdown to go into Standby Mode | 0 | 0 / 1 | 1,100/  1,200 |  |
| 2 | 02 | set\_currents | set motor currents | 8 | 0 / 8 | 1.100 |  |
| 3 | 03 | set\_spinup/down ramps | Sets the spin ramps for the legacy spinup/down algorithm | 8 | 0 / 8 | 1,100/  1,200 |  |
| 4 | 04 | not used | Was Load movetable (one axis / line) | 0 | 0 | N/A | removed\* |
| 5 | 05 | Set\_cruise&creep\_rates | These give the effective cruise speed, and with the creep period, the creep s | 8 | 0 / 8 | N/A |  |
| 6 | 06 | set\_creep\_periods | With Cmd 5, sets creep speed |  | 0 / 8 | 1,200 |  |
| 7 | 07 | not used | was *execute\_move \_table* | 0 | 0 | N/A | removed\* |
| 8 | 08 | not used | Was send calc value of checksum | 4 | 5 | N/A | removed\* |
| 9 | 09 | get\_temperature | report temperature (raw) | 0 | 2 | 9,000 |  |
| 10 | 0A | get\_CAN\_address | report CAN address | 0 | 2 | 1,200 |  |
| 11 | 0B | get\_firmware\_version | report firmware version | 0 | 2 | 1,200 |  |
| 12 | 0C | get\_device\_type | report device type | 0 | 1 | 1,100 |  |
| 13 | 0D | get\_movement\_status | report movement status | 0 | 1 | 1,400 |  |
| 14 | 0E | get\_current\_monitor\_vals | report current monitor values | 0 | 8 | 9,000 |  |
| 15 | 0F | get\_bootloader\_version | report bootloader version | 0 | 2 | 1,100 | Fails in current fw |
| 16 | 10 | set\_duty\_fiducial | set fiducial duty cycle | 2 | 0 | 1,100 |  |
| 17 | 11 | not used | Was read silicon ID (lower) | 0 | 0 | N/A | removed |
| 18 | 12 | not used | Was read silicon ID (upper) | 0 | 0 | N/A | removed |
| 19 | 13 | read\_sid\_short | read silicon ID (short version) | 0 | 8 | 1,200 |  |
| 20 | 14 | write\_CAN\_address | write CAN address to flash | 2 | 0 | 24,000 |  |

Table 1: Firmware Commands 1-20

\* legacy movetable commands are documented in the appendix

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 21 | 15 | read\_CAN\_address | report CAN address | 0 | 2 | 1,100 | removed |
| 22 | 16 | not used | Was send the received value of the silicon id (lower) | 8 | 0 | N/A | removed |
| 23 | 17 | not used | Was send the received value of the silicon id (upper) | 0 | 0 | N/A | removed |
| 24 | 18 | not used | Was send the received value of the silicon id (short) | 0 | 0 | N/A | removed |
| 25 | 19 | set\_device\_type | set device type to positioner or fiducial | 1 | 0 | 25,000 |  |
| 26 | 1A | load\_movetable | load movetable (two axes per line) | 8 | 0 | 1,000/  Last:1,300 |  |
| 27 | 1B | execute\_movetable | Execute/arm movetable | 1 | 0 / 7 | 1,200 |  |
| 28 | 1C | get\_checksum | report value of checksum and flags | 0 | 8 | 1.200 |  |
| 29 | 1D | not used | Was set Cruise Rate and Spin Ramps | 6 | 0 | N/A | removed |
| 30 | 1E | not used | Was set\_currents\_legacy | 8 | 0 | N/A | removed |
| 31 | 1F | not used | Was set\_motor\_parameters  \_legacy | 3 | 0 | N/A | removed |
| 32 | 20 | set\_cruise&creep\_amt | Set the steps to go | 8 | 0 / 8 | 1,200 |  |
| 33 | 21 | set\_up&execute\_move | Legacy cmd sets up cruise and creep for both motors and executes the mov | 2 | 0 | 1,200 | Modified so  It always executes |
| 34 | 22 | not used | was *execute move set up by Cmd 33* | 0 | 0 | N/A | removed |
| 35 | 23 | not used | was *flash\_leds\_legacy* |  |  | N/A | removed |
| 36 | 24 | not used | Was get\_bootloader\_version\_alt | 0 | 0 | N/A | removed |
| 37 | 25 | not used | was get\_firmware\_version\_alt | 0 | 0 | N/A | removed |
| 38 | 26 | set\_need\_double\_pulse | enables need\_double\_pulse on SYNC to execute a move table | 1 | 0 | 1,100 |  |
| 39 | 27 | get\_runtime | reports elapsed processor running time | 0 | 4 | 1,200 | new |
| 40 | 28 | enter\_stop\_mode\_sync | enter Stop Mode (exit via SYNC) | 0 | 0 | N/A |  |

Table 2: Firmware Commands 21-40

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 41 | 29 | not used | Was enter\_stop\_mode\_CAN | 0 | 0 | N/A | removed |
| 42 | 2A | software\_reset | processor is reset (from software) | 0 | 0 | 2,700,000 |  |
| 43 | 2B | Not used | enter bootloader mode (Branch to 0x08000000 to run bootloader) | 0 | see description | see description | Fails in fw 6.23 |
| 44 | 2C | dump\_n\_bytes | dump N bytes from memory or register | 6 | 8 | varies |  |
| 45 | 2D | get\_fw\_flash\_checksum | calculate and report simple checksum over firmware flash memory | 0 | 2 | 300,000 |  |
| 46 | 2E | get\_sync\_status | report status of the SYNC line | 0 | 1 | 1,400 |  |
| 47 | 2F | get\_system\_clock | read back value of configured System Core Clock (HCLK) | 0 | 1 | 1,200 |  |
| 48 | 30 | not used | Was set\_fid\_pwm\_frequency | 0 | 0 | N/A | removed |
| 49 | 31 | not used | Was get\_fid\_pwm\_frequency | 0 | 0 | N/A | removed |
| 50 | 32 | DC\_current\_test | set DC Current Test and Measurement | 6 | 8 X 8 | 27,000,000 |  |
| 51 | 33 | not used |  |  | 0 | N/A |  |
| 52 | 34 | enter\_stop\_mode | Enter stop mode (exit via POR only) | 0 | 0 | N/A |  |
| 53 | 35 | not used |  |  | 0 | N/A |  |
| 54 | 36 | set\_fid\_timed | timed fiducial operation | 4 | 0 | 1,500 |  |
| 55 | 37 | Report fidcial duty cycle. | Shows value of duty\_cycle | 0 | 8 | 1,400 |  |
| 56 | 38 | not used |  |  | 0 | N/A |  |
| 57 | 39 | not used |  |  | 0 | N/A |  |
| 58 | 3A | not used |  |  | 0 | N/A |  |
| 59 | 3B | store\_params | writes 46 parameters to flash | 0 | 8 | 27,000 |  |
| 60 | 3C | make\_stored\_params\_  default | read 46 parameters from flash and set as defaults | 0 | 0 | 1,400 |  |

Table 3: Firmware Commands 41-60

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 61 | 3D | read\_defaults\_RAM | report default parameters from RAM | 0 | 8 x 23 | 6,000 |  |
| 62 | 3E | read\_defaults\_FLASH | report default parameters from FLASH | 0 | 8 x 23 | 6,000 |  |
| 63 | 3F | Show\_inter\_priorities | Shows priority for 7 interrupts | 0 | 8 x 7 | 4,000 |  |
| 64 | 40 | not used | Was read memory | 0 | 0 | N/A | removd |
| 65 | 41 | write\_to\_memory | write to RAM memory or register. | 8 | 4 | 1,300 |  |
| 66 | 42 | can\_dump\_short | Returns one CAN message with CAN\_MCR and CAN\_ESR | 0 | 8 | 1,300 |  |
| 67 | 43 | not used | Was high\_power\_mode | 0 | 0 | N/A | removd |
| 68 | 44 | not used | Was setup\_vibramove | 0 | 0 | N/A | removd |
| 69 | 45 | not used | Was execute\_vibramove | 0 | 0 | N/A | removd |
| 70 | 46 | clear\_move\_table | resets all motor and fiducial activity | 0 | 0 | 1,200 |  |
| 71 | 47 | can\_dump\_long | like Cmd 66 but more data | 0 | 8 x 9 | 4,000 |  |
| 72 | 48 | get\_interrupt\_reg | reports the values of all interrupt registers | 0 | 8 x 14 | 4,500 |  |
| 73 | 49 | set\_ack\_move | set or reset Ack\_Move | 1 | 0 | 1,100 |  |
| 74 | 4A | set\_ABOM | set or reset Automatic Bus Off Management | 1 | 0 | 1,100 |  |
| 75 | 4B | not used | Was Set\_Fast\_HCLK | 1 | 0 | N/A | removd |
| 76 | 4C | Set\_Verbose | When Verbose is TRUE, various diagnostic CAN messages are generated | 1 | 0 | 1,100 |  |
| 77 | 4D | not used |  |  | 0 | N/A |  |
| 78 | 4E | not used |  |  | 0 | N/A |  |
| 79 | 4F | not used |  |  | 0 | N/A |  |
| 80 | 50 | Set\_first\_phase\_jump | Initial jump for short ramp spinup/down | 8 | 0 / 8 | 1,400 |  |

Table 4: Firmware Commands 61-80

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 81 | 51 | Set\_short\_spin\_ramp\_parameters\_motor\_0 | Set four total rotation values for motor\_0 | 8 | 0 / 3X8 | 1,200/  2,000 |  |
| 82 | 52 | Set\_short\_spin\_ramp\_parameters\_motor\_1 | Set four total rotation values for motor\_1 | 8 | 0 / 3X8 | 1,200/  2,000 |  |
| 83 | 53 | not used |  |  | 0 | N/A |  |
| 84 | 54 | not used |  |  | 0 | N/A |  |
| 85 | 55 | not used |  |  | 0 | N/A |  |
| 86 | 56 | Set\_Theta\_0 = 0 | Sets the initial value of Theta\_0 to 0 | 0 | 0 | 1,300 |  |
| 87 | 57 | Record\_Theta\_0\_CW\_up\_0 | Dumps the values of Theta\_0 recorded during a CW spinup via CAN | 0 | varies | varies |  |
| 88 | 58 | Record\_Theta\_0\_CW\_down\_0 | Dumps the values of Theta\_0 recorded during a CW spindown via CAN | 0 | varies | varies |  |
| 89 | 59 | Record\_Theta\_0\_CCW\_up\_0 | Dumps the values of Theta\_0 recorded during a CCW spinup via CAN | 0 | varies | varies |  |
| 90 | 5A | Record\_ Theta\_0\_ CCW\_down\_0 | Dumps the values of Theta\_0 recorded during a CCW spindown via CAN | 0 | varies | varies |  |

Table 4: Firmware Commands 81-90

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Command** | | **Command** | **Description** | **Data bytes** | **Resp-onse bytes** | **Approxi-mate**  **Execution**  **Time**  **(u-sec)** | **Notes\*** |
| **Nr.** | **Hex** |
| 128 | 3D | select\_mode | Starts the bootload process if the correct 8 byte code is given | 8 | 8 |  |  |
| 129 | 3E | send\_codesize |  | 4 | 0 or 8 |  |  |
| 130 | 3F | send\_number\_of\_parts |  | 4 | 0 or 8 |  |  |
| 131 | 40 | get\_bootloader\_status |  | 0 | 1 or 8 |  |  |
| 132 | 41 | send\_packet |  | 8 | 0 or 8 |  |  |

Table 5: Firmware Commands 128 - 132 (Bootloader Commands)

**Command and Data Structure**

This section describes each of the commands implemented in the firmware. Each CAN command sent out by the petalcontroller consists of a the 29-bit CAN message identifier ( formed from positioner ID <<8 + Cmd Nr.) and up to 8 bytes of data. The 8 least significant bits of the identifier specify the command number.

Note that in version 5.20 and above, when the processor is not executing a command or running motors, it drops into Sleep Mode where it draws approximately 2.8 milli-Amperes.

Any command sent with a number marked “not used” in Table 4 will be ignored by the hardware.

Outgoing CAN frame format:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Device ID  [28:8] | Cmd Nr.  [7:0] | Data Byte 0  (MSB)  [7:0] | Data Byte 1  [7:0] | Data Byte 2  [7:0] | Data Byte 3  [7:0] | Data Byte 4  [7:0] | Data Byte 5  [7:0] | Data Byte 6  [7:0] | Data Byte 7  (LSB)  [7:0] |

Each command can be sent out to all devices, all positioners only, or all fiducials only by using broadcast IDs

all devices: 20000 (0x4e20)

positioners only: 20001 (0x4e21)

fiducials only: 20002 (0x4e22)

If the command initiates a response from the positioner, the format of the response CAN message consists of an identifier field and a variable number of data bytes up to a maximum of 8, as indicated below. The MS bit of the return message CAN ID is set to ‘1’ so as to not confuse a return message with a command from the petal controller.

Response data format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ID +  0x10000000  [28:0] | byte 0 (LSB)  [7:0] | byte 1  [7:0] |  |  | byte N (MSB)  [7:0] |

NOTE: *positioner id in all response CAN messages has a ‘1’ in the MSB of the CAN identifier.*

The two motors controlling the fiber positioning are typically referenced as PHI motor and THETA motor. The THETA motor controls the movement about the central axis. The THETA motor is referenced as motor\_1 (M1) and the PHI motor as motor\_0 (M0).

The direction of motor rotation is denoted as CW (clockwise) or CCW (counterclockwise). Clockwise motion proceeds from the top to the right when viewing the fiber tip head-on.

**Motor Control Parameter Commands**

Commands 2, 3, 5, 6, 80, 81, and 82 set parameters which control the operation of the motors. They all have similar structure and operation. For most of the commands, the first four data bytes returned are parameters for motor\_0 (phi), and the final four bytes are the corresponding parameters for motor\_1 (theta).

With the exception of Command 6, they all take 8 data bytes and set either 8 one byte parameters, or 4 two byte parameters. If the data byte(s) corresponding to a given parameter is zero, then that parameter is not changed. Each of them will report back the values of the parameters set by the command if Verbose & 2 (set by Command 76) is true.

If the command is sent with all data bytes equal to zero, then the current values of all parameters are reported (provided Verbose & 2 is TRUE), but no values are changed. In the rare instance where it is desired to set parameters to zero, sending the command with data[0] == 255 will cause all parameters to be set to zero.

**Detailed Command Descriptions**

**Command 1 (0x01) – stay\_alive**

When a positioner is reset either by POR or a Command 42 (Software Reset), a timer is set up with the RTC which puts the device into Stop Mode after a 55 to 65 second count down. Once stopped, there is no way to exit from this other than a power cycle RESET operation. Command 1 disables the 60 second countdown, and thus must be sent within 55 seconds of turn-on to use the positioner. The data field of the CAN message is left empty.

Until Command 1 is received, the positioner will ignore all other commands

If Verbose & 1 is TRUE (default at power on), then the positioner returns the following message of four data bytes starting with data[0]:

data[0] (value of Stand\_By\_Flag when the command was received),

data[1] (value of low order byte of the real time clock when the command was received),

data[2] (value of Stand\_By\_Flag after it has been set to zero),

data[3] (value of low order byte of the real time clock after Stand\_By\_Flag) was set to zero),

**Command 2 (0x02) – set\_currents**

The set\_currents command is used to set the current parameters for the various stages of motor motion. Each argument is 1 byte long and expressed as a percentage (0-100).

Command 2 sets 8 motor operating currents as follows:

data[0] sets SpinUpCurrent\_0

data[1] sets SpinDownCurrent\_0

data[2] sets CruiseCurrent\_0

data[3] sets CreepCurrent\_0

data[4] sets SpinUpCurrent\_1

data[5] sets SpinDownCurrent\_1

data[6] sets CruiseCurrent\_1

data[7] sets CreepCurrent\_1

If Verbose & 2 is TRUE, then Command 2 returns the 8 parameter values as follows, starting with data[0]:

SpinUpCurrent\_0, SpinDownCurrent\_0, CruiseCurrent\_0, CreepCurrent\_0,

SpinUpCurrent\_1, SpinDownCurrent\_1, CruiseCurrent\_1, CreepCurrent\_1,

If any of the data bytes sent with Command 2 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 2 may be read out without changing anything by sending Cmd 2 with all data bytes equal to zero

**Command 3 (0x03) – set\_spin\_ramps**

The set\_spin\_ramps command is used to set the speed at which the spin up/down stages of motion are performed when using the legacy spinup and spindown algorithms. The command is sent with 8 arguments, each 1 byte in length.

data[0] sets CW\_SpinUpRamp\_0 (default is 12).

data[1] sets CCW\_SpinUpRamp\_0 (default is 12).

data[2] sets CW\_SpinDownRamp\_0 (default is 12).

data[3] sets CCW\_SpinDownRamp\_0 (default is 12).

data[4] sets CW\_SpinUpRamp\_1 (default is 12).

data[5] sets CCW\_SpinUpRamp\_1 (default is 12).

data[6] sets CW\_SpinDownRamp\_1 (default is 12).

data[7] sets CCW\_SpinDownRamp\_1 (default is 12).

What is now called the spin ramp, used to be called spin\_period. The change in 6.23 is that it is possible to set a different value for CW vs CCW, spin up vs. spin down, and motor\_0 vs. motor\_1.

If Verbose & 2 is TRUE, then Command 3 returns the 8 parameter values as follows, starting with data[0]:

CW\_SpinUpRamp\_0, CCW\_SpinUpRamp\_0, CW\_SpinDownRamp\_0, CCW\_SpinDownRamp\_0, CW\_SpinUpRamp\_1, CCW\_SpinUpRamp\_1, CW\_SpinDownRamp\_1, CCW\_SpinDownRamp\_1

If any of the data bytes sent with Command 3 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 3 may be read out without changing anything by sending Cmd 3 with all data bytes equal to zero

**Command 5 (0x05) – set\_cruise\_and\_creep\_rates**

This sets 8 values which are the step size in tenths of a degree of the motor rotation for the motor and direction indicated. The step during cruise occurs with each timer interrupt, 18,000 times per second. For creeps, the step occurs every 18,000 / Cruise\_Period times per second.

The command is sent with 8 arguments, each 1 byte in length.

data[0] sets CW\_CruiseStep\_0 (default is 33).

data[1] sets CCW\_ CruiseStep\_0 (default is 33).

data[2] sets CW\_Creep\_Step\_Size\_0 (default is 1).

data[3] sets CCW\_Creep\_Step\_Size\_0 (default is 1).

data[4] sets CW\_CruiseStep\_1 (default is 33).

data[5] sets CCW\_ CruiseStep\_1 (default is 33).

data[6] sets CW\_Creep\_Step\_Size\_1 (default is 1).

Data[7] sets CCW\_Creep\_Step\_Size\_1 (default is 1).

If Verbose & 2 is TRUE, then Command 5 returns the 8 parameter values as follows, starting with data[0]:

CW\_CruiseStep\_0, CCW\_CruiseStep\_0, CW\_Creep\_Step\_Size\_0, CCW\_Creep\_Step\_Size\_1, CW\_CruiseStep\_1, CCW\_CruiseStep\_1, CW\_Creep\_Step\_Size\_1, CCW\_Creep\_Step\_Size\_1

If any of the data bytes sent with Command 5 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 5 may be read out without changing anything by sending Cmd 5 with all data bytes equal to zero

**Command 6 (0x06) – set\_creep\_periods**

This sets 2 values which range between 1 and 255 and which together with the Creep\_Step\_Size set in Command 5 above, determine the creep rate. The rate of rotation of the motor shaft during a creep is equal to the Creep\_Step\_Size \* 18,000 / Creep\_Period / 10 degrees per second.

The command is sent with 2 arguments, each 1 byte in length.

data[0] sets CreepPeriod\_0 (default is 2).

data[1] sets CreepPeriod\_1 (default is 2).

If Verbose & 2 is TRUE, then Command 6 returns the 2 parameter values as follows, starting with data[0]:

CreepPeriod\_0, 0, 0, 0,

CreepPeriod\_1, , 0, 0, 0,

The spacing out of the return data is to preserve consistency with the practice established in the commands above of putting parameters for motor 0 in the first four bytes of the return message, with those for motor 1 in the second four bytes.

If any of the data bytes sent with Command 6 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 6 may be read out without changing anything by sending Cmd 6 with all data bytes equal to zero

**Command 9 (0x09) – get\_temperature**

This command is used to request the 12-bit ADC value associated with the temperature sensor. It returns 2 bytes and is sent with an empty data field. Note that in the 5.0 positioner software, commands which report values measured by one of the ADC’s give the value which was measured by the previous operation of the ADC. I.e. the value given is late by one ADC cycle. This was corrected in Rev. 5.25 and following.

*Response data format:* data[0]: Temperature [0:7]  
 data[1]: Temperature [8:11]]

**Command 10 (0x0A) – get\_can\_address**

This command is used report the device positioner ID. It returns 2 bytes and is sent with an empty data field.

*Response data format:* data[0]: CAN address [0:7]  
 data[1]: CAN address [8:15]

**Command 11 (0x0B) – get\_fw\_version**

This command is used to request the version of the firmware running on the positioners. It returns 2 bytes and is sent with an empty data field.

*Response data format:* data[0]: FW minor release number  
 data[1]: FW major release number

**Command 12 (0x0C) – get\_device\_type**

This command is used to request the currently configured device type. It returns 1 byte and is sent with an empty data field.

*Response data format:* data[0]: Device\_Type[0:7](fiducial = 1, positioner = 0)

**Command 13 (0x0D) – get\_move\_status**

Command 13 returns 1 byte and is sent with an empty data field. A positioner responds to this command while moving (counting down commanded creep/cruise steps) and also while idle. It is used to show whether the positioner motors are in operation or not, and whether a move table has been armed by command 27 and is waiting for a SYNC to start the movement.

If no move table has been loaded, or if there is a move table loaded but it has not been armed with command 27, then it returns ‘0’

If a move table has been loaded and armed by command 27, then it returns ‘2’

If a move table has been loaded and then allowed to time out then it returns ‘4’ the first time it is sent and returns ‘0’ on subsequent reads.

If one or motors is moving, it returns ‘1’.

If it is in a pause specified by the move table, it returns ‘0’.

To determine if a motor is moving, use Command 47. If it does not return ‘8’, then either a motor is moving or it is in a pause.

*Response data format:* data[0]: Movement\_Status[0:7] (Stopped = 0; Moving = 1; Armed = 2; Timed out = 4)

**Command 14 (0x0E) – get\_currents**

This command is used to request the 12-bit ADC measured from the current monitors 1 (Theta) and 2 (Phi). It returns 8 bytes and is sent with an empty data field. The positioner responds to this command while moving (counting down commanded creep/cruise steps) and also while idle. Note that in the 5.0 positioner software, commands which report values measured by one of the ADC’s give the value which was measured by the previous operation of the ADC. I.e. the value given is late by one ADC cycle. This was corrected in Rev. 5.25 and following.

*Response data format:* data[0]: IMON\_1[0:7]  
 data[1]: IMON\_1[8:11]  
 data[2]: 0  
 data[3]: 0  
 data[4]: IMON\_2[0:7]  
 data[5]: IMON\_2[8:11]  
 data[6]: 0  
 data[7]: 0

**Command 15 (0x0F) – get\_bl\_version**

This command is used to request the version of the bootloader running on the positioners. It returns 2 bytes and is sent with an empty data field.

*Response data format:* data[0]: Bootloader\_Minor\_Release[0:7]  
 data[1]: Bootloader\_Major\_Release[0:7]

Note: This command will return 00 00 in fw 6.23 because the allocated memory now exceeds the fixed location where the bootloader version was stored.

The bootloader version can alternatively be read by sending Command 42 followed after one second by Command 128 (with no data)

* + This will return 8 data bytes: (with the posID in the message ID as usual)
  + 42 6F 6F 74 46 57 (Major Ver. Nr.) (Minor Ver. Nr.)

**Command 16 (0x10) – set\_fiducial\_duty**

This command is used to set the fiducial duty cycle (216=100 %, 0 = off). This command is executed immediately upon receipt, and is used to turn the fiducial ON and OFF.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Duty Cycle  [15:8] | Duty Cycle  [7:0] |  |  |  |  |  |  |

**Command 19 (0x13) – read\_sid\_short**

This command is used to read the abbreviated version of the unique silicon id. The command is sent without an argument in the data field and returns the unique 64 bits constructed from the contents of the UID register using an algorithm supplied by STMicrosystems.*Response data format:* data[0]: SID[0:7]  
 data[1]: SID[8:15]  
 data[2]: SID[16:23]  
 data[3]: SID[24:31]  
 data[4]: SID[32:39]  
 data[5]: SID[40:47]  
 data[6]: SID[48:55]  
 data[7]: SID[56:63]

**Command 20 (0x14) – write\_can\_address**

This command is used to write a new CAN ID to a positioner's flash memory. Prior to using it, Command 19 must be used to read the positioner’s short silicon ID, and then Command 24 is used to write the short silicon ID back to the positioner and thus arm it to receive the new CAN ID.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CAN address  [15:8] | CAN address  [7:0] |  |  |  |  |  |  |

**Command 21 (0x15) – read\_can\_address**

This command is used to read the first location of page 61 in flash memory where the positioner CAN address is stored. The command is sent without an argument in the data field and returns the positioner CAN address in 2 bytes.

*Response data format:* data[0]: CAN\_Address [0:7]  
 data[1]: CAN\_Address [8:15]

**Command 24 (0x18) – write\_short\_version\_silicon\_ID**

This command is used to send the short version unique ID to a positioner and if it is correct will enable the target device’s write\_CAN\_address flag.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SID  [63:56] | SID  [55:48] | SID  [47:40] | SID  [39:32] | SID  [31:24] | SID  [23:16] | SID  [15:8] | SID  [7:0] |

**Command 25 (0x19) – set\_device\_type**

This command is used to set a device up as a fiducial or positioner. Once this command is sent, its setting is applied immediately and also stored in flash to be used as a default for when the device is power cycled or reset.The general broadcast ID for all devices is 0x4e20. To address all positioners only, an additional broadcast ID is used: 0x4e21. To address just all fiducials, the broadcast ID 0x4e22 is used. This command also updates the device so that it responds to the appropriate category broadcast ID, in addition to the general broadcast ID.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Device Type  [7:0] |  |  |  |  |  |  |  |

Device type:

1 - fiducial (any value 1 ... 255) will set the device type to 'fiducial'.

0 - positioner

**Command 26 (0x1A) – load\_movetable**

This is similar to Command 4 in Version 5.0, except that it uses the full 8 bytes of the CAN message to implement both a theta and a phi move in one table entry.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data[0] | data[1] | data[2] | data[3] | data[4] | data[5] | data[6] | data[7] |

data[0]: bit\_7 -- Immediate Execute (note 1)

bit\_6 -- Immediate Execute (note 1)

bit\_5 -- Phi Move

bit\_4 -- 1: Phi Cruise if ‘1’, creep if ‘0’

bit\_3 -- 1: Phi CCW if ‘1’, CW if ‘0’

bit\_2 -- Theta Move

bit\_1 -- 1: Theta Cruise if ‘1’, creep if ‘0’

bit\_0 -- 1: Theta CCW if ‘1’, CW if ‘0’

data[1]: Theta Move Steps [19:16], Phi Move Steps [19:16] (note 2)

data[2]: Phi Move Steps [15:8] (note 3)  
 data[3]: Phi Move Steps [7:0]

data[4]: Theta Move Steps [15:8] (note 3)

data[5]: Theta Move Steps [7:0]

data[6]: Post Pause [15:8] (note 4)

data[7]: Post Pause [7:0]

Notes:

(1) If both data[0] bit\_7 AND data[0] bit\_6 are FALSE, then the move table entry will be executed immediately. If either data[0] bit\_7 OR data[0] bit\_6 is TRUE, then the entry will be written to the move table and execution will occur following sending of Command 27.

(2) These will typically be used only when using reduced Cruise speeds or creeps greater than 19.4 degrees.

(3) Two bytes of steps can Cruise 641 degrees, or Creep 19.4 degrees

(4) These two bytes specify a delay in milliseconds. The execution of the move table is stopped in a while() loop but the CAN Rx interrupt is not disabled, so other commands will be received and executed

**Command 27 (0x1B) – execute\_movetable**

This will execute the Command 26 movetable without checking the checksum after SYNC is received, or if data[0]==TRUE, then immediately without waiting for SYNC. If data[0]==FALSE, then it waits for SYNC. If data[0]==TRUE, it executes immediately.

Note that at version 5.17 and later, there is no checking of bit sum match by the fipos software.

Commands 13,14, 42, and 70 may be executed while the motors are operating.

If AckMove is TRUE, then when all motor operations specified by the move table are complete, a response is sent with the standard positioner ID as the message ID, and 7 data bytes:

count, '\_', 'L', 'i', 'n', 'e', 's' where “count” is the number of move table lines which were sent to the positioner motors. Ack\_Move defaults FALSE. It is set by Cmd 73.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data[0] |  |  |  |  |  |  |  |

**Command 28 (0x1C) – get\_checksum**

This will report the value of bit\_sum without changing anything in the processor state. It is used in lieu of Command 8 (used in Version 5.0) with a move table loaded by Command 26. Bit\_sum is formed from a weighted sum of the data bytes used to generate the move table as follows:

bit\_sum = bit\_sum + (CAN\_RxMsg.data[0] + 65536\*CAN\_RxMsg.data[1] + 256\*CAN\_RxMsg.data[2] + CAN\_RxMsg.data[3] + 256\*CAN\_RxMsg.data[4] + CAN\_RxMsg.data[5] + 256\*CAN\_RxMsg.data[6] + CAN\_RxMsg.data[7] + 26)

The four high order data bytes report the status of other parameters related to the motor move process as shown.

*Response data format:*

data[0]: bit\_sum[7:0]

data[1]: bit\_sum[15:8]

data[2]: bit\_sum[23:16]

data[3]: bit\_sum[31:24]

data[4]: MoveCancel count

data[5]: MoveCancel\_Flag

data[6]: Ack\_Move

data[7]: bit\_sum\_match

**Command #32 – set\_cruise\_and\_cw\_creep\_amounts\_legacy**

This command is used to set the motor movement amounts in legacy mode.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **positioner id**  **[28:8]** | **command id: 32**  **[7:0]** | Cruise Steps to Go M0  [15:8] | Cruise Steps to Go M0  [7:0] | Creep Steps to Go M0  [15:8] | Creep Steps to Go M0  [7:0] | Cruise Steps to Go M1  [15:8] | Cruise Steps to Go M1  [7:0] | Creep Steps to Go M1  [15:8] | Creep Steps to Go M1  [7:0] |

**Command #33 – set\_up\_move\_legacy**

This command is used to set the movement flags. Four bytes have been added to allow long moves. The move that has been set up will immediately be executed.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **positioner id**  **[28:8]** | **command id: 33**  **[7:0]** | Shadow Flag M0  [7:0] | Shadow Flag M1  [7:0] |  |  | Cruise Steps to Go M0  [23:16] | Creep Steps to Go M0  [23:16] | Cruise Steps to Go M1  [23:16] | Creep Steps to Go M1  [23:16] |

Command 33 Execute Flags

* bit 7 -- 1 if want a CW spin-up
* bit 6 -- 1 if want a CW cruise
* bit 5 -- 1 if want a CW spin-down
* bit 4 -- 1 if want a CCW spin-up
* bit 3 -- 1 if want a CCW cruise
* bit 2 -- 1 if want a CCW spin-down
* bit 1 -- 1 if want a CCW creep back to the hard stop
* bit 0 -- 1 if want a CW creep forward

**Command 38 (0x26) – set\_need\_double\_pulse**

Enables need\_double\_pulse on SYNC to execute a move table. If sent with data[0] equal to '1', then need\_double\_pulse = `1` (so it is active). If data[0] is 0, then a single pulse on the SYNC line will execute the move table.

Need\_double\_pulse is FALSE on power up.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data[0] |  |  |  |  |  |  |  |

data[0]: need\_double\_pulse (1= need double pulse, 0 = single pulse is OK)

**Command 39 (0x27) – get\_runtime**

Reports the elapsed processor running time in increments of one second since the processor was reset

*Response data format:*

data[0]: RTC\_CNTL LSB

data[1]: RTC\_CNTL MSB

data[2]: RTC\_CNTH LSB

data[3]: RTC\_CNTH MSB

**Command 40 (0x28) – enter\_stop\_mode\_sync**

This command causes the microcontroller to enter the low-current Stop Mode. The data field of the CAN message is left empty. To exit Stop Mode (and resume normal operation) the hardware ‘SYNC’ line is set high by the petal controller, causing an external interrupt to be triggered and waking the processor from Stop Mode.

**Command 42 (0x2A) – soft\_reset**

This command initiates a software RESET of the microcontroller. The processor is reset by setting bit2, SYSRESETREQ, in register SCB\_AIRCR.

This command can be used to initiate the bootloader process. Send Command 42, then delay 500 m-sec and send Command 128 with the boot unlock code (0x4D, 0x2E, 0x45, 0x2E, 0x4C, 0x65, 0x76, 0x69) Use of Cmd 42 eliminates the need to control turn-on of the positioner 7.5 Volt power and then ensure that Cmd 128 is sent within the 2 second window after the power supply comes up.

See the bootloader documentation for details about the firmware upload process (DESI-2764).

**Command 43 (0x2B) – enter\_bootloader\_mode**

This command is inoperative and should not be used.

**Command 44 (0x2C) – dump\_n\_bytes**  
This command dumps a specified portion of the microcontroller memory, sending CAN messages, each of which contain two words read from the memory. It is sent with a starting address argument (S\_Addr) and an argument that specifies the number of bytes to be read out (N\_bytes). The starting address should be on a word boundary, and the number of bytes read should be modulo four. The response sends back a message showing the command line data, and then sends messages containing two 4 byte words in each message until the specified number of bytes have been transmitted. Care must be taken in the selection of the address range, as reading some of the RESERVED portions of RAM may cause the device to become non-responsive and need to be power-cycled.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| S\_Addr [31:24] | S\_Addr [23:16] | S\_Addr [15:8] | S\_Addr  [7:0] | N\_bytes  [15:8] | N\_bytes  [7:0] |  |  |

*Response data format:* data[0]: first word [7:0]  
 data[1]: first word [15:8]  
 data[2]: first word [23:16]  
 data[3]: first word [31:24]

data[4]: first word [7:0]  
 data[5]: first word [15:8]  
 data[6]: first word [23:16]  
 data[7]: first word [31:24] data[0]: third word [7:0] ……..

data[n]: memory value [n\*8:n\*8+7]

**Command 45 (0x2D) – get\_fw\_checksum**

This command calculates a checksum over the region of flash memory where the firmware is stored. It sends the lowest two bytes onto the CAN bus. See the bootloader documentation for a detailed memory map (DESI-2764). The command is sent out without an argument and returns two bytes.

Positioner firmware Rev 5.0 returns 0xECF5

Positioner firmware Rev 5.20 returns 0x…..

Positioner firmware Rev 5.25 returns 0xD0B6

Positioner firmware Rev 5.31 returns 0x76D9

Positioner firmware Rev 6.0 returns 0x12D7

Positioner firmware Rev 6.23 returns 0x0B32

*Response data format:* data[0]: FW\_Checksum [0:7]  
 data[1]: FW\_Checksum [8:15]

**Command 46 (0x2E) – get\_sync\_status**

This command reads back the status of the SYNC signal line and returns one byte (1 if the SYNC line is high and 0 if it is low). The command is sent out without an argument.

.

*Response data format:*  
  
 data[0]: Sync\_Status [0:7]

**Command 47(0x2F) – get\_system\_clock**

This command reads back the frequency of the system core clock in MHz. Devices with fully functional external oscillators will return 72 (MHz). If during startup the microcontroller detects an unstable external oscillator circuit, it will switch to the internal oscillator and this command will return 64 (MHz). The command is sent out without an argument and returns one byte. Positioners with firmware 5.15 and above will return 8 unless a motor operation is in process, in which case they will return 72 or 64.

*headerResponse data format*:  
  
 data[0]: System\_Clock [0:7]

**Command 50 (0x32) – current\_test**

Command 50 sets the six motor phases to either GND or 7.5V as specified by the command data. The command is sent with 6 data bytes. When the command is sent, it puts the specified pattern of DC voltages on the 6 motor coils and then executes the send current values function once every 3.6 seconds for a total of 7 measurements of the current drawn from the 7.5 volt supply by each motor switch. The command is sent with 6 data bytes which correspond to the 6 motor coils as follows:

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data[0]  M0\_Phase\_A | data[1]  M0\_Phase\_B | data[2]  M0\_Phase\_C | data[3]  M1\_Phase\_A | data[4]  M1\_Phase\_B | data[5]  M1\_Phase\_C |  |  |

If the data byte is FALSE, the corresponding coil is connected to ground. If TRUE it is connected to 7.5 volts.

For software Rev 5.24 and earlier, the first current measurement should be discarded because it represents the current before the coils were energized. After approximately 27 seconds, the DC is turned off and the positioner returns to its normal operating state.

*Response data frame format:*

The positioner responds with seven CAN messages, each with two current values as shown:

data[0]: theta\_current [7:0] data[1]: theta\_current [15:8]

data[2]: theta\_current [23:16] data[3]: theta\_current [31:24]

data[4]: phi\_current [7:0] data[5]: phi\_current [15:8]

data[6]: phi\_current [23:16] data[7]: phi\_current [31:24]

**Command 52 (0x34) – enter\_stop\_mode**

Enters Stop Mode with no way to get out except power on reset. Sent with no data.

**Command 54 (0x36) – set\_fid\_timed**

Timed Fiducial Operation. Data bytes data[0]:data[1] specify the duty cycle, data[2]:data[3] specify number of seconds after which it will automatically turn off. Note that during fiducial operation, commands 9, 14,16, 42, and 70 are recognized. Note also that the operation time is modulo 10 seconds, so that the actual time is as follows:

on\_time setting Actual time of operation

1. 0
2. 5 +/- 1 second
3. 5 +/- 1 second

10 15 +/- 1 second

11 15 +/- 1 second, etc

Similarly as with motor operation, the processor HCLK remains at 72 MHz rather dropping to 8 MHz.

Cmd 54 is recognized only by devices set to be fiducials, i.e with device\_type == TRUE. The command is ignored by devices for which device\_type == FALSE.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data[0]  duty cycle [8:15] | Data[1]  duty cycle [7:0] | Data[2]  on\_time [8:15] | Data[3]  on\_time [7:0] |  |  |  |  |

**Command 55 (0x38) – report\_fiducial\_duty\_cycle**

This command is used to read back the value of duty\_cycle, the parameter which controls the on time of the LED in a fiducial. The command is sent without an argument in the data field and returns 8 bytes.

*Response data format:* data[0]: 00 if duty\_cycle == 0. 01 if duty\_cycle != 0  
 data[1]: 00  
 data[2]: 00  
 data[3]: 00  
 data[4]: duty\_cycle[7:0]  
 data[5]: duty\_cycle[15:8]  
 data[6]: 00  
 data[7]: 00

**Command 59 (0x3B) – store\_default\_parameters\_in\_FLASH**

Writes the current values for the SYNC format (single or double pulse), motor currents, cruise speed, spinup rate, and creep rate into the FLASH memory at address DEFAULT\_PARAMETERS (Page 65).

The list of parameters is saved to FLASH in the following order:

SpinUpCurrent\_0 SpinDownCurrent\_0

CruiseCurrent\_0 CreepCurrent\_0

SpinUpCurrent\_1 SpinDownCurrent\_1

CruiseCurrent\_1 CreepCurrent\_1

CW\_SpinUpRamp\_0 CCW\_SpinUpRamp\_0

CW\_SpinDownRamp\_0 CCW\_SpinDownRamp\_0

CW\_SpinUpRamp\_1 CCW\_SpinUpRamp\_1

CW\_SpinDownRamp\_1 CCW\_SpinDownRamp\_1

CW\_CruiseStep\_0 CCW\_CruiseStep\_0

CW\_Creep\_Step\_Size\_0 CCW\_Creep\_Step\_Size\_0

CW\_CruiseStep\_1 CCW\_CruiseStep\_1

CW\_Creep\_Step\_Size\_1 CCW\_Creep\_Step\_Size\_1

CreepPeriod\_0 CreepPeriod\_1

first\_CW\_spinup\_0\_phase\_incr first\_CCW\_spinup\_0\_phase\_incr first\_CW\_spindown\_0\_phase\_incr first\_CCW\_spindown\_0\_phase\_incr first\_CW\_spinup\_1\_phase\_incr first\_CCW\_spinup\_1\_phase\_incr first\_CW\_spindown\_0\_phase\_incr first\_CCW\_spindown\_1\_phase\_incr

CW\_spinup\_0\_rotation CCW\_spinup\_0\_rotation

CW\_spindown\_0\_rotation CCW\_spindown\_0\_rotation

CW\_spinup\_1\_rotation CCW\_spinup\_1\_rotation

CW\_spindown\_1\_rotation CCW\_spindown\_1\_rotation

need\_double\_pulse Ack\_Move

ABOM Verbose

Responds with a CAN message containing the number of attempted writes of words to FLASH, and the number of words successfully written, which is 0x005C for th6.23 firmware.

*Response data format:* data[0]: Number\_of\_attemped\_FLASH\_writes  
 data[4]: Number\_of\_successful\_FLASH\_writes

I.e. the 8 byte response after a successful execution of Cmd 59 is: 0x5C 00 00 00 0x5C 00 00 00.

**Command 60 (0x3C) – make\_stored\_parameters\_defaults**

Reads the current values from FLASH which were written by Command 59 and writes them into RAM as the default values.

Note that when a positioner or fiducial is powered or undergoes a software reset, the values of the default parameters are those programmed into the version 6.23 firmware. To use the parameters stored in FLASH for motor operations, Command 60 must be executed. The revised parameters will then continue to be operative until the parameters are re-written or the device is re-powered or is otherwise reset.

**Command 61 (0x3D) – read\_defaults\_RAM**

Reports the currently set values of the default parameters listed above under Command 59 as they are recorded in the RAM. These are the values which will be implemented when a motor operation is executed.

Command 61 returns the values in CAN messages of 8 bytes as follows. The first item listed, SpinUpCurrent\_0, is returned in data[0] through data[3], of the first response message, and the rest follow in order

*Response data format: (This is the same as is shown for Command 59)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Message 1 | data[3:0]: | SpinUpCurrent\_0 | data[4:7]: | SpinDownCurrent\_0 |
| Message 2 | data[3:0]: | CreepCurrent\_0 | data[4:7]: | CruiseCurrent\_0 |
| Message 3 | data[3:0]: | SpinUpCurrent\_1 | data[4:7]: | SpinDownCurrent\_1 |
| Message 4 | data[3:0]: | CreepCurrent\_1 | data[4:7]: | CruiseCurrent\_1 |
| Message 5 | data[3:0]: | CW\_SpinUpRamp\_0 | data[4:7]: | CCW\_SpinUpRamp\_0 |
| Message 6 | data[3:0]: | CCW\_SpinDownRamp\_0 | data[4:7]: | CW\_SpinDownRamp\_0 |
| Message 7 | data[3:0]: | CW\_SpinUpRamp\_1 | data[4:7]: | CCW\_SpinUpRamp\_1 |
| Message 8 | data[3:0]: | CW\_SpinDownRamp\_1 | data[4:7]: | CCW\_SpinDownRamp\_1 |
| Message 9 | data[3:0]: | CW\_CruiseStep\_0 | data[4:7]: | CCW\_CruiseStep\_0 |
| Message 10 | data[3:0]: | CW\_Creep\_Step\_Size\_0 | data[4:7]: | CCW\_Creep\_Step\_Size\_0 |
| Message 11 | data[3:0]: | CW\_CruiseStep\_1 | data[4:7]: | CCW\_CruiseStep\_1 |
| Message 12 | data[3:0]: | CW\_Creep\_Step\_Size\_1 | data[4:7]: | CCW\_Creep\_Step\_Size\_1 |
| Message 13 | data[3:0]: | CreepPeriod\_0 | data[4:7]: | CreepPeriod\_1 |
| Message 14 | data[3:0]: | first\_CW\_spinup\_0\_phase\_incr | data[4:7]: | first\_CCW\_spinup\_0\_phase\_incr |
| Message 15 | data[3:0]: | first\_CW\_spindown\_0\_phase\_incr | data[4:7]: | first\_CCW\_spindown\_0\_phase\_incr |
| Message 16 | data[3:0]: | first\_CW\_spinup\_1\_phase\_incr | data[4:7]: | first\_CCW\_spinup\_1\_phase\_incr |
| Message 17 | data[3:0]: | first\_CW\_spindown\_0\_phase\_incr | data[4:7]: | first\_CCW\_spindown\_1\_phase\_incr |
| Message 18 | data[3:0]: | CW\_spinup\_0\_rotation | data[4:7]: | CCW\_spinup\_0\_rotation |
| Message 19 | data[3:0]: | CW\_spindown\_0\_rotation | data[4:7]: | CCW\_spindown\_0\_rotation |
| Message 20 | data[3:0]: | CW\_spinup\_1\_rotation | data[4:7]: | CCW\_spinup\_1\_rotation |
| Message 21 | data[3:0]: | CW\_spindown\_1\_rotation | data[4:7]: | CCW\_spindown\_1\_rotation |
| Message 22 | data[3:0]: | need\_double\_pulse | data[4:7]: | Ack\_Move |
| Message 23 | data[3:0]: | ABOM | data[4:7]: | Verbose |

**Command 62 (0x3D) – read\_defaults\_FLASH**

Reports the currently set values of the default parameters listed above under Command 59 as they are recorded in FLASH memory. These are NOT the values which will be implemented when a motor operation is executed, but rather are simply stored values in the FLASH memory. Execution of Command 60 will write these stored values into the RAM in place of the default values and then they will be used for subsequent motor operations.

Command 62 returns the values in CAN messages of 8 bytes as follows. The first item listed, SpinUpCurrent\_0, is returned in data[0] through data[3], of the first response message, and the rest follow in order.

*Response data format:*

Same as that of Command 61.

**Command 65 (0x41) – write\_to\_memory**

Write to RAM memory or register. Write data[7:4] to register or RAM location addressed by data[3:0].

Enter the bytes MSB first, and it reads out the MSB first. This reads back what you wrote. Note that in the case of writing to registers, what you read is often not what you wrote.   
Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| data[0]  address  [31:24] | data[1] address  [23:16] | data[2] address  [15:8] | data[3] address  [7:0] | data[4]  data to  write  [31:24] | data[5] data to  write  [23:16] | data[7] data to  write  [15:8] | data[7] data to  write  [7:0] |

*Response data format:* data[0]: LS byte of the word in RAM or register that was written to  
 data[1]: ….data[2]: …..data[3]: MS byte of the word in RAM or register that was written to

**Command 66 (0x42) – can\_dump\_short**

Short CAN Info Errors Dump. Gives 1 CAN messages reporting: | CAN\_MCR, CAN\_ESR |

Note: This command returns 8 bytes. The last one is REC. The next to last is TEC.

*Response data format:*

Message 1 data[3:0]: CAN\_MCR data[7:4]: CAN\_ESR

See Command 71 for a listing of the register names

**Command 70 (0x46) – clear\_move\_table**

Stops positioner all motor or fiducial operation. Clears Move Table. Resets fiducial duty cycle to zero. This command is sent with no data bytes.

**Command 71 (0x47) – can\_dump\_long**

Extended CAN Information Dump. This is sent with no data bytes. It is similar to Cmd 66 except that it sends a total of 9 CAN messages with additional data.

*Response data format:*

Message 1 data[3:0]: CAN\_MCR data[7:4]: CAN\_MSR

Message 2 data[3:0]: CAN\_TSR data[7:4]: CAN\_RF0R

Message 3 data[3:0]: CAN\_RF1R data[7:4]: CAN\_ESR

Message 4 data[3:0]: CAN\_TDL0R data[7:4]: CAN\_TDL1R

Message 5 data[3:0]: CAN\_TDL2R data[7:4]: CAN\_TDH0R

Message 6 data[3:0]: CAN\_TDH1R data[7:4]: CAN\_TDH2R

Message 7 data[3:0]: CAN\_RI0R data[7:4]: CAN\_RI1R

Message 8 data[3:0]: CAN\_RDL0R data[7:4]: CAN\_RDL1R

Message 9 data[3:0]: CAN\_RDH0R data[7:4]: CAN\_RDH1R

CAN\_MCR: CAN master control register

CAN\_MSR: CAN master status register

CAN\_TSR: CAN transmit status register

CAN\_RF0R: CAN receive FIFO 0 register

CAN\_RF1R: CAN receive FIFO 1 register

CAN\_ESR: CAN error status register

CAN\_TDLxR: CAN mailbox data low register (x=0,1,2)

CAN\_TDHxR: CAN mailbox data high register (x=0,1,2)

CAN\_RIxR: CAN RX FIFO identifier register (x=0,1)

CAN\_RDLxR: CAN RX FIFO mailbox data low register (x=0,1)

CAN\_RDHxR: CAN RX FIFO mailbox data high register (x=0,1)

**Command 72 (0x48) – read\_interrupts**

This is sent with no data bytes and returns 14 CAN messages showing the contents of all interrupt enable, pending, and priority registers which control the operation of the Nested Vectored Interrupt Controller at the lowest operational level. The register values returned are shown below:

*Response data format:*

Message 1 data[3:0]: NVIC\_ISER0 data[7:4]: NVIC\_ISER1

Message 2 data[3:0]: NVIC\_ICER0 data[7:4]: NVIC\_ICER1

Message 3 data[3:0]: NVIC\_ISPR0 data[7:4]: NVIC\_ISPR1

Message 4 data[3:0]: NVIC\_ICPR0 data[7:4]: NVIC\_ICPR1

Message 5 data[3:0]: NVIC\_IABR0 data[7:4]: NVIC\_IABR1

Message 6 data[3:0]: NVIC\_IPR0 data[7:4]: NVIC\_IPR1

Message 7 data[3:0]: NVIC\_IPR2 data[7:4]: NVIC\_IPR3

Message 8 data[3:0]: NVIC\_IPR4 data[7:4]: NVIC\_IPR5

Message 9 data[3:0]: NVIC\_IPR6 data[7:4]: NVIC\_IPR7

Message 10 data[3:0]: NVIC\_IPR8 data[7:4]: NVIC\_IPR9

Message 11 data[3:0]: NVIC\_IPR10 data[7:4]: NVIC\_IPR11

Message 12 data[3:0]: NVIC\_IPR12 data[7:4]: NVIC\_IPR13

Message 13 data[3:0]: NVIC\_IPR14 data[7:4]: NVIC\_IPR15

Message 14 data[3:0]: CRTC\_ALRL data[7:4]: RTC\_ALRH

**Command 73 (0x49) – Set Ack\_Move**

Sets or resets Ack\_Move depending on the value of the single data byte sent with the command. If Ack\_Move is TRUE, then when a Cmd 26 move table has completed execution, a response is sent with the standard positioner ID as the message ID, and 7 data bytes:

count, '\_', 'L', 'i', 'n', 'e', 's' where “count” is the number of move table lines which were sent to the positioner motors. Ack\_Move defaults FALSE.

**Command 74 (0x4A) – Set ABOM**

Sets or resets bit\_6 in the CAN\_MCR register, ABOM (Automatic Bus Off Management), depending on the value of the single data byte sent with the command. According to the STM32F103 manual, if ABOM is TRUE, then the positioner will recover from a Bus Off state automatically following 128 X 11 data periods of recessive state of the CAN bus. If ABOM is FALSE, the 128 X 11 data periods of recessive state are required, along with a software command from the positioner software. ABOM defaults TRUE.

Note: In actual operation, when ABOM is true, the REC counts down by 3 and the TEC counts down by 1 following each error free CAN message. But once in Bus Off, the positioner never recovers unless it is RESET.

**Command 76 (0x4C) – Set Verbose Flag**

Sent with the three LSBits of data[0] set as follows:

If Verbose & 1, then Command 1 responds with the four byte message described above.

If Verbose & 2, then Commands 2, 3, 5, 6, 32, 80, 81, and 82 respond with the current values of the parameters they control.

If Verbose & 4, then a diagnostic message is sent with each service of the RTC interrupt, and a pattern of pulses is put onto the “D1” test output on the fipos PCB during each interrupt service.

**Command 80 (0x50) – Set first phase jump for short ramp spin up and spin down**

This command is part of the set up of the short ramp spin up process. For the short ramp spinup/down, the number of degrees of rotation of the motor rotor is specified along with an initial phase jump of the motor magnetic field direction as specified by this command 80. The software then calculates a linearly increasing rotation rate for the magnetic field direction such that at the end of the spin up, the total field rotation (including that specified by command 80), is equal to the specified value, See Commands 81 and 82.

Each first phase increment is specified as a one byte number which sets the number of degrees of rotation of the applied magnetic field in the motor during the first pass through the TIMER interrupt at the start of the spinup/dowm

The command is sent with 8 data bytes which specify ther initial phase jumps as shown below.

data[0] first\_CW\_spinup\_0\_phase\_incr

data[1] first\_CCW\_spinup\_0\_phase\_incr

data[2] first\_CW\_spindown\_0\_phase\_incr

data[3] first\_CCW\_spindown\_0\_phase\_incr

data[4] first\_CW\_spinup\_1\_phase\_incr

data[5] first\_CCW\_spinup\_1\_phase\_incr

data[6] first\_CW\_spindown\_1\_phase\_incr

data[7] first\_CCW\_spindown\_1\_phase\_incr

If Verbose & 2 is TRUE, then Command 80 returns the 8 parameter values as follows, starting with data[0]:

first\_CW\_spinup\_0\_phase\_incr, first\_CCW\_spinup\_0\_phase\_incr,

first\_CW\_spindown\_0\_phase\_incr, first\_CCW\_spindown\_0\_phase\_incr,

first\_CW\_spinup\_1\_phase\_incr, first\_CCW\_spinup\_1\_phase\_incr,

first\_CW\_spindown\_1\_phase\_incr, first\_CCW\_spindown\_1\_phase\_incr

If any of the data bytes sent with Command 80 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 80 may be read out without changing anything by sending Cmd 80 with all data bytes equal to zero.

If Command 80 is sent with data[0] == 255, then all 8 parameters are set to zero.

**Command 81 (0x51) – Specify the Rotor rotation for the short ramp spin up and calculate spin parameters for motor\_0**

This command is part of the set up of the short ramp spin up process. For the short ramp spinup/down, the number of degrees of rotation of the motor rotor is specified by Command 81 along with an initial phase jump of the motor magnetic field direction as specified by command 80. The software then calculates two parameters: the number of spinup steps, and the amount of the linearly increasing magnetic field phase angle. The phase increment at each timer interrupt step is equal to the number of the pass through the TIMER interrupt times this calculated value.

The command is sent with 8 bytes which specify four values of total spin up angle as shown below:

Note this specifies the total rotation of the motor rotor, including the effect of any non-zero value from command 80.

Not also that the specified rotation is that of the motor, so that if the rotation of the output shaft is specified, the number has to be multiplied by 337.35943 before it is used in Command 81

CW\_spinup\_0\_rotation = 256 \* data[0] + data[1]

CW\_spindown\_0\_rotation = 256 \* data[2] + data[3]

CCW\_spinup\_0\_rotation = 256 \* data[4] + data[5]

CCW\_spindown\_0\_rotation = 256 \* data[6] + data[7]

Then the number of required spin steps is calculated:

nr\_of\_CW\_spinup\_0\_steps = (int)( (float)(CW\_spinup\_0\_rotation - 10 \* first\_CW\_spinup\_0\_phase\_incr) \* 2 / CW\_CruiseStep\_0 - 0.5 )

nr\_of\_CW\_spindown\_0\_steps = (int)( (float)(CW\_spindown\_0\_rotation + 10 \* first\_CW\_spindown\_0\_phase\_incr) \* 2 / CW\_CruiseStep\_0 - 0.5 )

nr\_of\_CCW\_spinup\_0\_steps = (int)( (float)(CCW\_spinup\_0\_rotation - 10 \* first\_CCW\_spinup\_0\_phase\_incr) \* 2 / CCW\_CruiseStep\_0 - 0.5 )

nr\_of\_CCW\_spindown\_0\_steps = (int)( (float)(CCW\_spindown\_0\_rotation + 10 \* first\_CCW\_spindown\_0\_phase\_incr) \* 2 / CCW\_CruiseStep\_0 - 0.5 )

Then the phase increment is calculated;

CW\_spindown\_phase\_0\_incr = (float)(CW\_spinup\_0\_rotation - 10 \* first\_CW\_spinup\_0\_phase\_incr) \* 2 / (nr\_of\_CW\_spinup\_0\_steps \* (nr\_of\_CW\_spinup\_0\_steps + 1) )

CW\_spindown\_phase\_0\_incr = (float)(CW\_spindown\_0\_rotation + 10 \* first\_CW\_spindown\_0\_phase\_incr) \* 2 / ( nr\_of\_CW\_spindown\_0\_steps \* (nr\_of\_CW\_spindown\_0\_steps + 1)

CCW\_spinup\_phase\_0\_incr = (float)(CCW\_spinup\_0\_rotation - 10 \* first\_CCW\_spinup\_0\_phase\_incr) \* 2 / (nr\_of\_CCW\_spinup\_0\_steps \* (nr\_of\_CCW\_spinup\_0\_steps + 1) )

CCW\_spindown\_phase\_0\_incr = (float)(CCW\_spindown\_0\_rotation + 10 \* first\_CCW\_spindown\_0\_phase\_incr) \* 2 / ( nr\_of\_CCW\_spindown\_0\_steps \* (nr\_of\_CCW\_spindown\_0\_steps + 1)

If Verbose & 2 is TRUE, then Command 81 returns three CAN messages each with 8 data bytes which each give four parameter values.

The first message reports the values which were set for the spinup rotations as follows:

data[0] : CW\_spinup\_0\_rotation >> 8, data[1] : CW\_spinup\_0\_rotation & 0xFF,

data[2] : CW\_spindown\_0\_rotation >> 8, data[3] : CW\_spindown\_0\_rotation & 0xFF,

data[4]: CCW\_spinup\_0\_rotation >> 8, data[5]: CCW\_spinup\_0\_rotation & 0xFF,

data[6]: CCW\_spindown\_0\_rotation >> 8, data[7]: CCW\_spindown\_0\_rotation & 0xFF);

The second message shows the calculated values for the number of spinup steps:

data[0] : (int)nr\_of\_CW\_spinup\_0\_steps>>8, data[1] : (int)nr\_of\_CW\_spinup\_0\_steps & 0Xff

data[2] : (int)nr\_of\_CW\_spindown\_0\_steps>>8, data[3] : (int)nr\_of\_CW\_spindown\_0\_steps & 0xFF,

data[4] : (int)nr\_of\_CCW\_spinup\_0\_steps>>8, data[5] : (int)nr\_of\_CCW\_spinup\_0\_steps & 0xFF, data[6] : (int)nr\_of\_CCW\_spindown\_0\_steps>>8, data[7] : (int)nr\_of\_CCW\_spindown\_0\_steps & 0xFF);

The third message shows the calculated values of the phase increment:

data[0] : (int)(CW\_spinup\_phase\_0\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CW\_spinup\_phase\_0\_incr \* 10000) & 0Xff

data[0] : (int)(CW\_spindown\_phase\_0\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CW\_spindown\_phase\_0\_incr \* 10000) & 0xFF,

data[0] : (int)(CCW\_spinup\_phase\_0\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CCW\_spinup\_phase\_0\_incr \* 10000) & 0Xff

data[0] : (int)(CCW\_spindown\_phase\_0\_incr \* 10000)>>8 & 0Xff

data[0] : (int)(CCW\_spindown\_phase\_0\_incr \* 10000) & 0xFF);

Note: The phase increment is a floating point number. To report the value, it is multiplied by 10,000 and then transmitted in the CAN message as a two byte integer. Therefore, divide these numbers by 10,000 after taking them out of the CAN message.

If any of the data bytes sent with Command 81 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 81 may be read out without changing anything by sending Cmd 81 with all data bytes equal to zero.

If Command 81 is sent with data[0] == 255, then all 8 parameters are set to zero. It is necessary to be able to reset the value of the specified rotations, because if the specified value is not zero, then the timer interrupt will use the short ramp spin up algorithm instead of the original algorithm..

**Command 82 (0x52) – Specify the Rotor rotation for the short ramp spin up and calculate spin parameters for motor\_1**

This command is part of the set up of the short ramp spin up process. For the short ramp spinup/down, the number of degrees of rotation of the motor rotor is specified by Command 82 along with an initial phase jump of the motor magnetic field direction as specified by command 80. The software then calculates two parameters: the number of spinup steps, and the amount of the linearly increasing magnetic field phase angle. The phase increment at each timer interrupt step is N times this calculated value.

The command is sent with 8 bytes which specify four values of total spin up angle as shown below:

CW\_spinup\_1\_rotation = 256 \* data[0] + data[1]

CW\_spindown\_1\_rotation = 256 \* data[2] + data[3]

CCW\_spinup\_1\_rotation = 256 \* data[4] + data[5]

CCW\_spindown\_1\_rotation = 256 \* data[6] + data[7]

Then the number of required spin steps and the phase increments are then calculated as in Command 81 above.

If Verbose & 2 is TRUE, then Command 82 returns three CAN messages each with 8 data bytes which each give four parameter values.

The first message reports the values which were set for the spinup rotations as follows:

data[0] : CW\_spinup\_1\_rotation >> 8, data[1] : CW\_spinup\_1\_rotation & 0xFF,

data[2] : CW\_spindown\_1\_rotation >> 8, data[3] : CW\_spindown\_1\_rotation & 0xFF,

data[4]: CCW\_spinup\_1\_rotation >> 8, data[5]: CCW\_spinup\_1\_rotation & 0xFF,

data[6]: CCW\_spindown\_1\_rotation >> 8, data[7]: CCW\_spindown\_1\_rotation & 0xFF);

The second message shows the calculated values for the number of spinup steps:

data[0] : (int)nr\_of\_CW\_spinup\_1\_steps>>8, data[1] : (int)nr\_of\_CW\_spinup\_1\_steps & 0Xff

data[2] : (int)nr\_of\_CW\_spindown\_1\_steps>>8, data[3] : (int)nr\_of\_CW\_spindown\_1\_steps & 0xFF,

data[4] : (int)nr\_of\_CCW\_spinup\_1\_steps>>8, data[5] : (int)nr\_of\_CCW\_spinup\_1\_steps & 0xFF, data[6] : (int)nr\_of\_CCW\_spindown\_1\_steps>>8, data[7] : (int)nr\_of\_CCW\_spindown\_1\_steps & 0xFF);

The third message shows the calculated values of the phase increment:

data[0] : (int)(CW\_spinup\_phase\_1\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CW\_spinup\_phase\_1\_incr \* 10000) & 0Xff

data[0] : (int)(CW\_spindown\_phase\_1\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CW\_spindown\_phase\_1\_incr \* 10000) & 0xFF,

data[0] : (int)(CCW\_spinup\_phase\_1\_incr \* 10000)>>8 & 0xFF,

data[0] : (int)(CCW\_spinup\_phase\_1\_incr \* 10000) & 0Xff

data[0] : (int)(CCW\_spindown\_phase\_1\_incr \* 10000)>>8 & 0Xff

data[0] : (int)(CCW\_spindown\_phase\_1\_incr \* 10000) & 0xFF);

Note: The phase increment is a floating point number. To report the value, it is multiplied by 10,000 and then transmitted in the CAN message as two bytes of an integer. Therefore, divide these numbers by 10,000 after taking them out of the CAN message.

If any of the data bytes sent with Command 82 is zero, then the value of that parameter is not changed, but the current value of the parameter still appears in the report if Verbose & 2 is TRUE.

Thus the values of the parameters addressed by Command 82 may be read out without changing anything by sending Cmd 82 with all data bytes equal to zero.

If Command 82 is sent with data[0] == 255, then all 8 parameters are set to zero. It is necessary to be able to reset the value of the specified rotations, because if the specified value is not zero, then the timer interrupt will use the short ramp spin up algorithm instead of the original algorithm.

**Temporary Test Commands for the Evaluation of the Short Ramp Spinup/down Algorithms**

To support testing of the short ramp spin up/down algorithms, code has been added to the TIMER ISR for the motor\_0 spinup and spindown. An array of 2048 elements is defined for each of the four cases, CW spinup, CW spindown, CCW spinup, and CCW spin down. The total motor rotation and the total number of passes through the interrupt are also recorded.

Note that the added code is included for motor\_0 only. Motor\_1 has the same algorithms, but with no readout.

**Command 86 (0x56) – Sets motor\_0 rotor positioner to zero**

To aid in interpretation of the output of Commands 87, 88, 89, and 90, Theta\_0 and the counts of the total motor rotation are set to zero.

**Command 87 (0x57) – Report Results of CW short ramp spinup**

The command is sent with no data bytes.

While the motor is doing the spinup, the following are recorded in the structure for each pass through the TIMER ISR:

Index\_MSB, Index\_LSB, Theta\_0[Index]\_MSB, Theta\_0[Index]\_LSB,

Total\_Theta\_0\_rot\_MSB, Total\_Theta\_0\_rot\_LSB, Total\_passes\_MSB, Total\_passes\_LSB

Then, if Command 87 is sent, it reads the values in the structure and returns a CAN message for each pass through the TIMER interrupt service routine. This gives a detailed description of the profile of the spinup of the motor.

**Command 88 (0x58) – Report Results of CW short ramp spindown**

Similar action to that of Command 87.

**Command 89 (0x59) – Report Results of CCW short ramp spinup**

Similar action to that of Command 87.

**Command 90 (0x90) – Report Results of CCW short ramp spindown**

Similar action to that of Command 87.

**Appendix A: Bootloader Related Commands**

The firmware uploading process is also governed by the petal controller. Updated firmware can be sent by the petal controller to all devices via CAN.

Note: A bug in the bootloader code in the positioner firmware limits the size of the S/W object file which can be download to 0x7CFE bytes. To limit the code size, we have eliminated less used commands, eliminated repeated code where the hit in processing speed can be tolerated, and we have shown that we can tolerate speed reduction using a cosine table which covers only 90 degrees and uses trigonometry functions to cover the full angle range. Given the danger of attempting to modify the bootloader code, there is currently no plan to correct this. It is still possible to install code of arbitrary size into positioners not installed in a petal via JTAG or SWD.

The FW upload process works as follows:

1. Positioner/fiducial power is switched on
2. Within 2 seconds after power-up the microcontroller is in upload mode'. Sending command 128 (with special 8 byte code) during this time will initiate the FW upload process.
   1. If this command (with correct code) is not received within 2 seconds, the bootloader jumps to the main firmware application
   2. If command 128 is received without the special code, the bootloader sends out its version number before jumping to the main application
3. After entering upload mode the control code must send the code size, number of parts, and contents of hex file. Broadcast addresses are accepted.
   1. The targeted microcontroller counts any errors related to expected part number, packet number, write operation count, and checksum.
   2. A delay longer than 10 seconds prior to receiving the code size or number of parts will result in a timeout (causing the controller to jump out of upload mode and run the application, sending out an error message)
4. After the upload the control code sends a verification request message to boards using unique CAN addresses.
   1. boards respond, one by one, with either ‘OK’ or ‘ERROR’
   2. a pause that is longer than 10 seconds after receiving all packets but prior to receiving the verification request command results in a timeout (after which the device will wait for a power reset and send out an error message)
   3. boards that respond with ‘OK’ automatically jump to running the main application
   4. boards that respond with ‘ERROR’ (or don’t respond) are then reprogrammed individually

Note that the current bootloader code cannot handle the case of loading firmware into positioners with failed high speed external oscillators.

**Command 128 (0x80) – select\_mode**

This command is used to initiate firmware upload mode. If after a positioner/fiducial is powered up this command is received without the special code below in the data field, any other command is received, or 2 seconds pass without activity, the device will switch to running the main firmware application. If command 128 is received with the special code then the device will continue with the firmware upload process. It will send the response below to indicate that it has entered bootloader mode.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0x4D  [7:0] | 0x2E  [7:0] | 0x45  [7:0] | 0x2E  [7:0] | 0x4C  [7:0] | 0x65  [7:0] | 0x76  [7:0] | 0x69  [7:0] |

*Response data format (if command 128 with special code is received):* data[0]: 'B' 0x42  
 data[1]: 'o' 0x6F  
 data[2]: 'o' 0x6F  
 data[3]: 'T'' 0x54  
 data[4]: 'M' 0x4D  
 data[5]: 'o' 0x6F  
 data[6]: 'd'' 0x64  
 data[7]: 'e 0x65

*Response data format (if command 128 with anything other than the special code):* data[0]: 'B' 0x42  
 data[1]: 'o' 0x6F  
 data[2]: 'o' 0x6F  
 data[3]: 't' 0x74  
 data[4]: 'F' 0x46  
 data[5]: 'W' 0x57  
 data[6]: BL\_Major\_Release  
 data[7]: BL\_Minor\_Release

**Command 129 (0x81) – send\_codesize**

This command is used to tell the positioners/fiducials the size of the hex firmware file (in 32-bit words) that will be transferred in bootloader mode. If there is a pause that is longer than 10 seconds between the receipt of this command and entry into bootloader mode, the microcontroller will send out a ‘timeout’ CAN message and switch to running the firmware application.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Code size  [31:24] | Code size  [23:16] | Code size  [15:8] | Code size  [7:0] |  |  |  |  |

*Response data format (if command 129 was not received within 10 seconds of receiving command 128):* data[0]: 129 0x81  
 data[1]: 'T' 0x54

data[2]: 'i' 0x69  
 data[3]: 'm' 0x6D  
 data[4]: 'e' 0x65  
 data[5]: 'o' 0x6F  
 data[6]: 'u' 0x75  
 data[7]: 't' 0x74

**Command 130 (0x82) – send\_number\_of\_parts**

This command is used to tell the positioners/fiducials how many parts the firmware has been broken up into. If there is a pause that is longer than 10 seconds between the receipt of this command and code size, the microcontroller will send out a ‘timeout’ CAN message and switch to running the firmware application.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Number of  Parts  [31:24] | Number of  Parts  [23:16] | Number of  Parts  [15:8] | Number of  Parts  [7:0] |  |  |  |  |

*Response data format (if command 130 was not received within 10 seconds of receiving command 128):* data[0]: 130   
 data[1]: 'T'

data[2]: 'i'  
 data[3]: 'm'  
 data[4]: 'e'  
 data[5]: 'o'  
 data[6]: 'u'  
 data[7]: 't'

**Command 131 (0x83) – get\_bootloader\_status**

This command is used to request a bootloader programming status from positioners/fiducials after a firmware hex file has been transmitted. Each positioner or fiducial is polled by their unique id. If the bootloader received all expected CAN messages and no errors occurred, the bootloader will respond with a 1 in data byte 0. If there was a mismatch between what the bootloader was expecting to receive and what had been transmitted an error message will be returned. This command is sent with an empty data field. The microcontroller will timeout after 10 seconds if this command is not received, at which point the ‘Waiting’ message will be sent out followed by the positioner’s current error counts. The device will wait for power to be reset.

*Response data format (if command 131 was not received within 10 seconds of last 132 command):* data[0]: 131  
 data[1]: 'W' 0x57

data[2]: 'a' 0x61  
 data[3]: 'i' 0x69  
 data[4]: 't' 0x74  
 data[5]: 'i' 0x69  
 data[6]: 'n' 0x6E  
 data[7]: 'g' 0x67

*Response data format (if no errors were encountered):* data[0]: 0x01

*Response data format (if errors were encountered):* data[0]: 0x00  
 data[1]: Error Count [0]

data[2]: Error Count [1]  
 data[3]: Error Count [2]  
 data[4]: Error Count [3]  
 data[5]: Error Count [4]  
 data[6]: 0x00  
 data[7]: 0x00

**Command 132 (0x84) – send\_packet**

This command is used to transfer all packets of the firmware hex file (as 32-bit words) from the petal controller to the selected fiducials/positioners. Each packet is sent with a part number, a packet number, and an XOR checksum value. If a pause in transmission that is greater than 10 seconds occurs, the microcontroller will send out a ‘Waiting’ message and wait for the power to be reset.

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Part  Number  [7:0] | Packet  Number  [15:8] | Packet  Number  [7:0] | Packet  [31:24] | Packet  [23:16] | Packet  [15:8] | Packet  [7:0] | Checksum  [7:0] |

*Response data format (if command was not received within 10 seconds of last command):* data[0]: 132  
 data[1]: 'W'

data[2]: 'a'  
 data[3]: 'i'  
 data[4]: 't'  
 data[5]: 'i'  
 data[6]: 'n'  
 data[7]: 'g'

**Appendix B : Legacy Commands**

**Command 4 (0x04) – set\_up\_move**

Note: This command has been removed from FW versions 5.17 on. It was used to load the legacy move table structure. The command is documented here in case devices with FW revision 5.0 (used during commissioning and during Year 1) need to be operated.

Each execution of this command loads a single row into a move table. Each move table row, when executed, causes a single axis, theta or phi, to move CW or CCW by a specified amount in Creep or in Cruise. The move table entry also includes two bytes which specify an optional delay time (Post-Pause) following the motor move. Execution of the lines in the move table will occur after the conditions of Cmds 7 and 8 below are satisfied.

The first byte of the move table entry (Mode) sets the choice of axis, direction, and speed of the move, and specifies whether the entry is the last line in the move table or if the entire table consists of a single move which is to be executed immediately without having to meet the requirements of Cmds 7 and 8. The meaning of the bits in the Mode byte is as follows:

Data[0] bit\_7 Reserved; set to ‘0’  
Data[0] bit\_6 Reserved; set to ‘0’  
Data[0] bit\_5 1: End of move tables If bit\_4 and bit\_5 are both ‘0’, then it is a single move  
Data[0] bit\_4 1: Move entries will be sent which will execute immediately  
Data[0] bit\_3 1: Pause only, 0: move as specified  
Data[0] bit\_2 1: Theta move, 0: Phi move  
Data[0] bit\_1 1: Cruise, 0: Creep  
Data[0] bit\_0 1: Rotate CCW, 0: CW

Data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Mode[7:0] | Move Steps  [23:16] | Move Steps  [15:8] | Move Steps  [7:0] | Post-Pause  Byte 1 (ms)  [15:8] | Post-Pause  Byte 0 (ms)  [7:0] |  |  |

*Notes:*

*(1) A bug in the Rev 5.0 firmware requires that adjacent entries in a move table alternate between theta and phi moves, or no move will occur. When the entries are alternated, then both motors will run at the same time when the execution gets to the complimentary entry. The work around to this is to pair each desired move with a dummy move of zero length of the other axis. This has been corrected in Rev 5.15 and above.*

*(2) In the documentation for Rev 5.0 FW Data[0] bit\_2 the settings for Theta and Phi are incorrect. In all FW versions bit\_2 must be set for a Theta move.*

**Command 7 (0x07) – execute\_table\_legacy**

Note: This command has been removed from FW versions 5.17 on. The command is documented here in case devices with FW revision 5.0 (used during commissioning and during Year 1) need to be operated.

This command is used to execute the movetable loaded with command 4 upon its receipt (rather than waiting for a SYNC signal). The data field of the CAN message is left empty.

**Command 8 (0x08) – get\_move\_table\_status**

Note: This command has been removed from FW versions 5.17 on. The command is documented here in case devices with FW revision 5.0 (used during commissioning and during Year 1) need to be operated.

This command is sent with 8 bytes of data which are a checksum over the data bytes loaded into the move table calculated by the petal controller. On receipt of this command, the positioner replies with 5 data bytes. The first 4 are the checksum as calculated by the positioner. Data[4] is a Status Response Code as follows:

0x01: Move table received, checksum matched, move table will be executed when SYNC (hard or soft) is received

0x02: Move table received checksum mismatch, move table will be cleared without anything being executed

0x03: No move table or incomplete table. Move table stack will be cleared and ready for a new table to be sent.

After Cmd 8 has been sent with a successful 01 response, the motor move will start after receipt of Cmd 7, or receipt of a SYNC pulse.

In Rev 5.0 firmware, a partially filled or not executed move table will remain in the memory indefinitely. In Rev 5.15 and above, any unexecuted table will be cleared after 40 seconds with no move table commands being sent. I.e., the 40 second timer is reset each time another move table command is sent.

*NOTE: If a move table is successfully verified with Cmd 8, then additional sendings of Cmd 8 with data of zero will not change anything. Sending Cmd 8 again with the correct bit sum, or anything other than zero in the data field will cause the move table to be erased. Sending Cmd 8 with data zero for the case of a partially filled or filled move table before it has been verified with the correct bit sum will cause the table to be erased.*

Data frame format sent with Cmd 8:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Sum [31:24] | Bit Sum [23:16] | Bit Sum [15:8] | Bit Sum [7:0] |  |  |  |  |

Response data frame format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Sum [31:24] | Bit Sum [23:16] | Bit Sum [15:8] | Bit Sum [7:0] | Status  Respons  Code |  |  |  |