# AN4290
# Application note

# L647x, L648x and powerSTEP01 family communication protocol

Enrico Poli

## Introduction

The L6470, L6472, L6474, L648x and powerSTEP01 devices provide advanced features and high programmability. The devices are controlled by a host microcontroller through a fast SPI interface compliant with the daisy chain configuration.

This document describes how the SPI communication protocol is implemented and it gives some suggestions about the application design.

The features of the different parts and the current control algorithms (voltage mode driving, predictive current control and auto-adjusted decay) are not investigated in this document. Please refer to the respective application notes.

# Contents

# 1 SPI communication interface

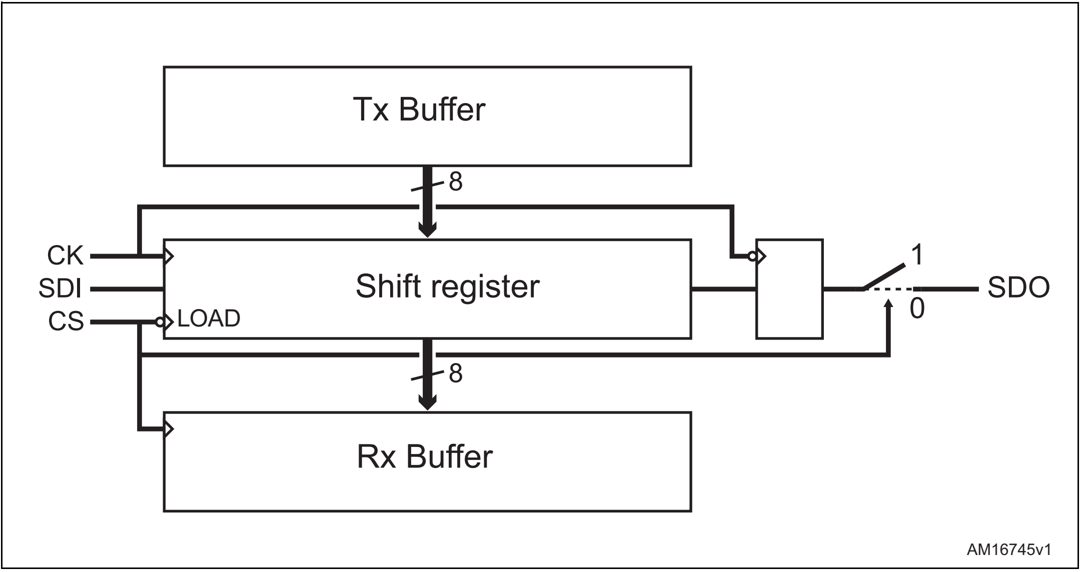
The devices (always slave) can be driven by an MCU (always master) sending commands through an 8-bit SPI interface. The 8-bit shift register of the device is kept enabled while $\overline{CS}$ input is forced low. During this time, at every raising edge of the serial clock (CK) the SDI input is stored into the shift register. At CK falling edges the SDO output is updated according to the last bit of the shift register (see *Figure 1*).

When the $\overline{CS}$ input is raised, the device catches the shift register content and interprets its value as a command or an argument of the command received previously.

All the bytes are transmitted starting from the most significant bit.

At every transmission cycle, that is the time between the falling and raising edge of the $\overline{CS}$ line, the number of bytes transmitted by the master is equal to those received.

**Figure 1. SPI interface structure**



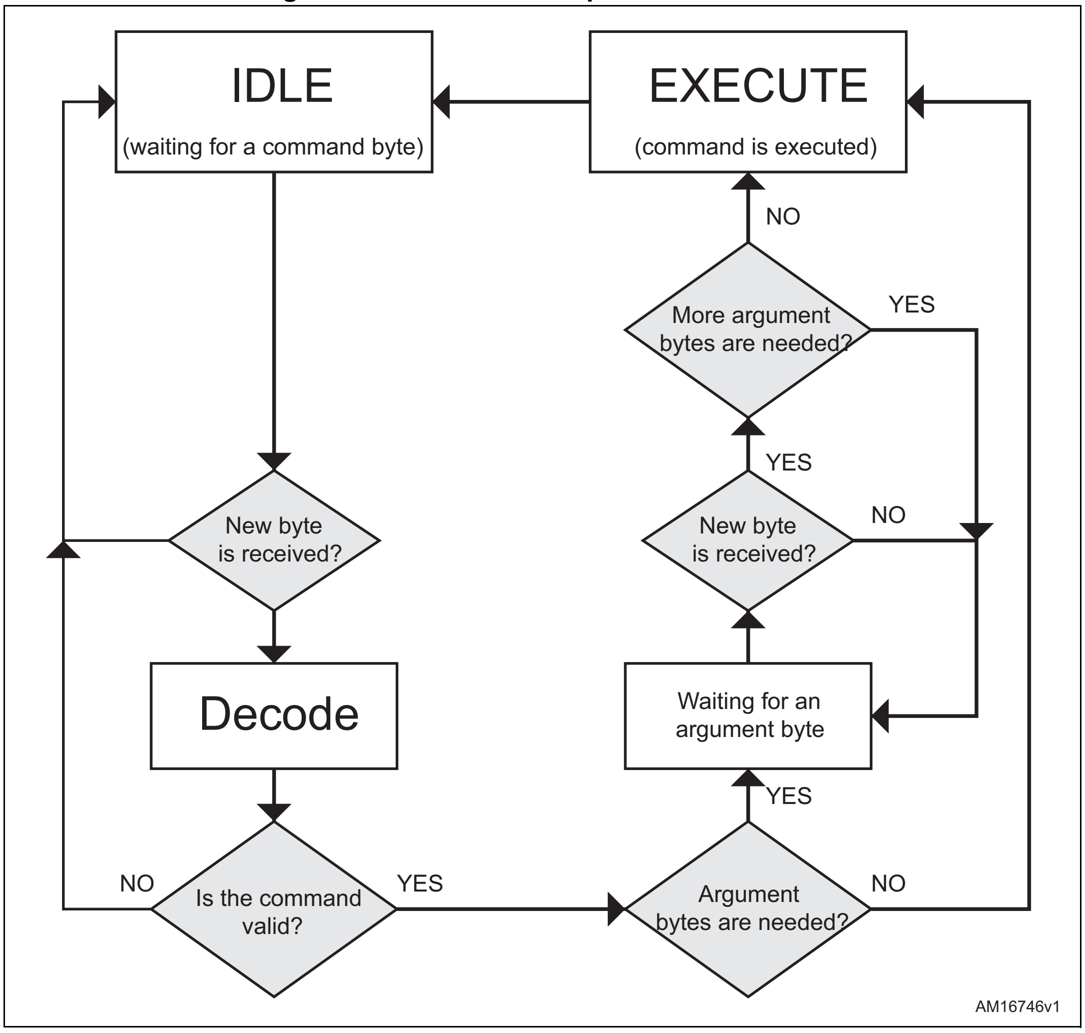
## Communication protocol

The communication protocol is based on single byte commands that can be followed by a command argument up to 3 bytes long.

Part of the information needed to execute the target operation could be embedded into the command byte, that is the target direction for the Run command, and the argument provides the extra data as well as the target position of the GoTo command.

When a command requires an argument, it has to be transmitted starting from the most significant byte. The command is not completed and it is not executed until all the argument bytes are received by the device. It is not possible for the transmission of a command to be aborted once it is started; the command has to be completed and a new command can be used. The flow chart of the communication protocol is shown in *Figure 2*.

**Figure 2. Communication protocol flow chart**



By default the response byte of the device is h00 (hexadecimal format). Some commands, for example those used to read the value of a register, generate a response from the device up to 3 bytes long which is transmitted during the following transmission cycles.

When a command returns a response, it is transmitted during the following transmission cycles starting from the most significant byte.The MCU transmits a byte (command or argument) for each response byte that is transmitted.

The number of required argument bytes and the number of returned response bytes for each command are listed in *Table 1* (the zero value implies that no argument is requested or that no response is generated).

**Table 1. Command list**

| Command | Argument length [byte] | Response length [byte] | Notes |
|---|---|---|---|
| NOP | 0 | 0 | |
| SetParam | According to the target register | 0 | Some registers are read only. Some registers can be set in specific conditions only. |
| GetParam | 0 | According to the target register | |
| Run(1) | 3 | 0 | |
| Move (1) | 3 | 0 | Available only if the motor is stopped. |
| GoTo(1) | 3 | 0 | Available only if a previous command is not under execution (not BUSY). |
| GoTo_DIR(1) | 3 | 0 | Available only if a previous command is not under execution (not BUSY). |
| GoHome(1) | 0 | 0 | Available only if a previous command is not under execution (not BUSY). |
| GoMark(1) | 0 | 0 | Available only if a previous command is not under execution (not BUSY). |
| GoUntil(1) | 3 | 0 | |
| ReleaseSW(1) | 0 | 0 | |
| StepClock(1) | 0 | 0 | Available only if the motor is stopped. |
| SoftStop(1) | 0 | 0 | |
| HardStop (enable in L6474) | 0 | 0 | |
| SoftHiZ(1) | 0 | 0 | |
| HardHiz (disable in L6474) | 0 | 0 | |
| ResetPos(1) | 0 | 0 | |
| ResetDevice(1) | 0 | 0 | |
| GetStatus | 0 | 2 | |

1. Available in L6470, L6472, L648x and powerSTEP01 devices only.

## Reading and writing registers (GetParam and SetParam commands)

All the device registers can be read using the GetParam and written using the SetParam command. While the reading is always available, the writing of the registers is bound to the state of driving of the motor.

In particular:

– Some registers are read only.

– Critical configuration registers can be modified only when the power stage is disabled.

– Some parameters related to the speed profile and the positioning can be modified when the motor is stopped (L6470, L6472, L648x and powerSTEP01 only).

The SetParam command byte is described in *Table 2*. It includes the address of the target registers in five low significant bits (ADDR[4] to ADDR[0]).

**Table 2. SetParam byte command**

| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | ADDR[4] | ADDR[3] | ADDR[2] | ADDR[1] | ADDR[0] |

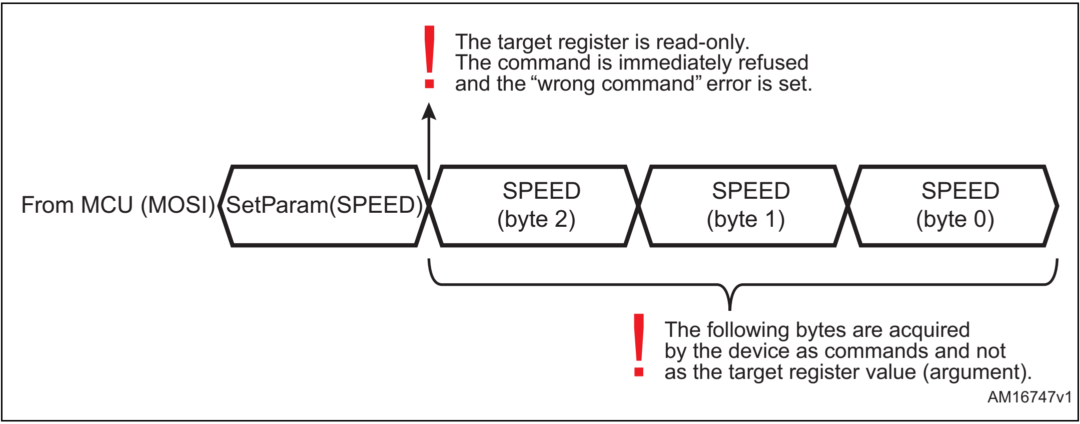
The command byte has to be followed by an argument providing the new register value (most significant byte first) and does not generate any response byte (standard response bytes are returned if no previous response is pending). The byte length of the argument depends on the dimension of the target register. For example a 12-bit register requires a 2-byte argument and a 4-bit register requires a single byte argument. Some bits of the argument could be ignored according to the structure of the target register.

As a consequence, the SetParam is the only command composed of a variable number of bytes (command byte + 1, 2 or 3 argument bytes according to the target register). The *Table 4* lists the number of argument bytes for each device register.

Some registers of the device are read only or can be only written in particular conditions (see *Table 4*).

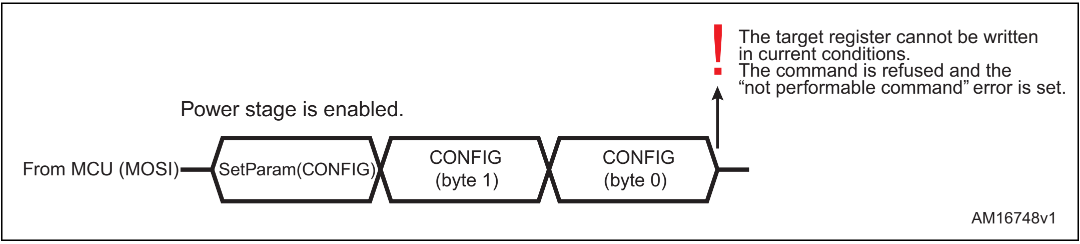
If the host microcontroller sends a SetParam command targeting a read only or a not existing register (the address value does not match any register) the command byte is ignored and the wrong command failure flag in the STATUS register is latched (in the L648x and powerSTEP01 devices the wrong command and the not performable command failure flags correspond to the same bit). In this case the failure occurs as soon as the command byte is received, so no argument byte must be sent as shown in *Figure 3*.

Figure 3. Attempt to write a read only \ not existing register


The target register is read-only.
The command is immediately refused and the "wrong command" error is set.
From MCU (MOSI)
SetParam(SPEED)
SPEED (byte 2)
SPEED (byte 1)
SPEED (byte 0)
The following bytes are acquired by the device as commands and not as the target register value (argument).
AM16747v1

If the host microcontroller attempts to write a writable register in a forbidden condition (CONFIG register when the power stage is enabled) the command is ignored and the not performable command failure flag in the STATUS register is latched (in the L648x and powerSTEP01 devices, the wrong command and the not performable command failure flags correspond to the same bit). In this case the failure occurs as soon as the last argument byte is received as shown in *Figure 4*.

Figure 4. Attempt to write a register in a forbidden condition


The target register cannot be written in current conditions.
The command is refused and the "not performable command" error is set.
Power stage is enabled.
From MCU (MOSI)
SetParam(CONFIG)
CONFIG (byte 1)
CONFIG (byte 0)
AM16748v1

The GetParam command byte is described in *Table 3*. It includes the address of the target registers in five low significant bits (ADDR[4] to ADDR[0]).

Table 3. GetParam byte command

| **Bit #** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | ADDR[4] | ADDR[3] | ADDR[2] | ADDR[1] | ADDR[0] |

The command does not need any argument and generates a response byte sequence containing the target register value which is transmitted starting from the most significant byte. The length in bytes of the response depends on the dimension of the target register. For example a 12-bit register generates a 2-byte response and a 4-bit register generates a single byte response. All the response bits, which do not match the target register, are forced to zero. The *Table 4* lists the number of response bytes for each device register.

**Table 4. Number of arguments or response bytes of the SetParam \ GetParam commands according to the target register**

| Register | Number of bytes | Notes |
|---|---|---|
| ABS_POS | 3 | |
| MARK | 3 | |
| SPEED | 3 | Read only |
| EL_POS | 2 | Writable when in HiZ only |
| ACC | 2 | Writable when stopped only |
| DEC | 2 | Writable when stopped only |
| MAX_SPEED | 2 | |
| MIN_SPEED | 2 | Writable when stopped only |
| FS_SPD | 2 | |
| KVAL_HOLD(1) | 1 | |
| KVAL_RUN(1) | 1 | |
| KVAL_ACC(1) | 1 | |
| KVAL_DEC(1) | 1 | |
| INT_SPEED(1) | 2 | Writable when in HiZ only |
| ST_SLP(1) | 1 | Writable when in HiZ only |
| FN_SLP_ACC(1) | 1 | Writable when in HiZ only |
| FN_SLP_DEC(1) | 1 | Writable when in HiZ only |
| K_THERM(1) | 1 | |
| TVAL_HOLD(2) | 1 | |
| TVAL_RUN(2) | 1 | |
| TVAL_ACC(2) | 1 | |
| TVAL_DEC(2) | 1 | |
| T_FAST(2) | 1 | Writable when in HiZ only |
| TON_MIN(2) | 1 | Writable when in HiZ only |
| TOFF_MIN(2) | 1 | Writable when in HiZ only |
| ADC_OUT | 1 | Read only |
| OCD_TH | 1 | |
| STALL_TH(1) | 1 | |
| STEP_MODE | 1 | Writable when in HiZ only |
| ALARM_EN | 1 | Writable when stopped only |
| GATECFG1(3) | 2 | Writable when in HiZ only |
| GATECFG2(3) | 1 | Writable when in HiZ only |

Table 4. Number of arguments or response bytes of the SetParam \ GetParam commands according to the target register (continued)

| Register | Number of bytes | Notes |
|---|---|---|
| CONFIG | 2 | Writable when in HiZ only |
| STATUS | 2 | Read only |

1. L6470 and L6480 only.
2. L6472, L6474 and L6482 only.
3. L6480 and L6482 only.

During the response transmission the device is still operative and can receive and execute new commands (see *Figure 5*). If a command, requiring a new response, is received before the end of the transmission of the current response, the new response replaces the previous one (see *Figure 6*).

Figure 5. Response transmission management

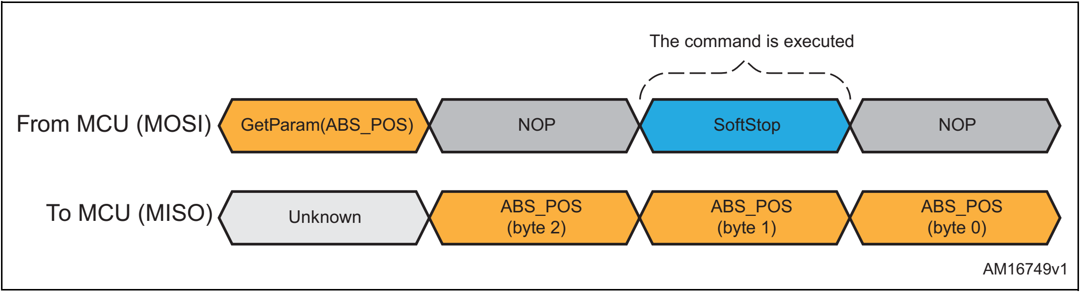
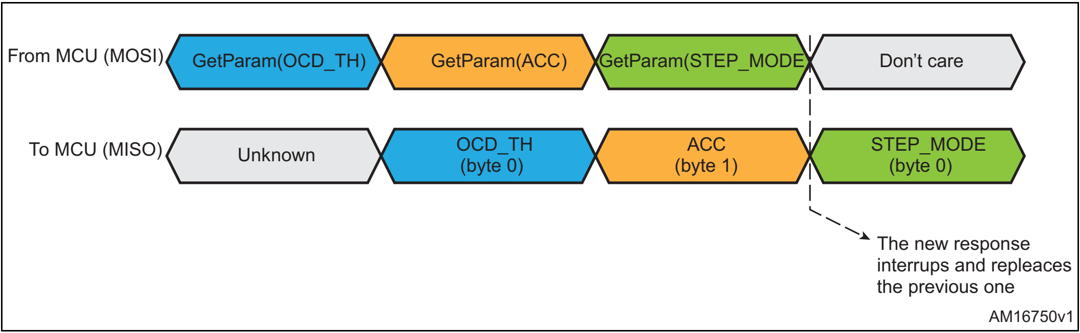


Figure 6. Response aborting



In case of registers, which can autonomously change their value such as the ABS_POS register, the response corresponds to the value of the register after the GetParam command has been received and decoded. If the register value changes later, the response is not updated accordingly.

## Releasing the failure flags (GetStatus command)

All the failure flags in the STATUS register of the device are latched when the respective alarm condition is triggered. In this way, the occurrence of a temporary failure is stored into the status register and the user is warned about it. The flags can be released through the GetStatus command.

The command byte is described in *Table 5*.

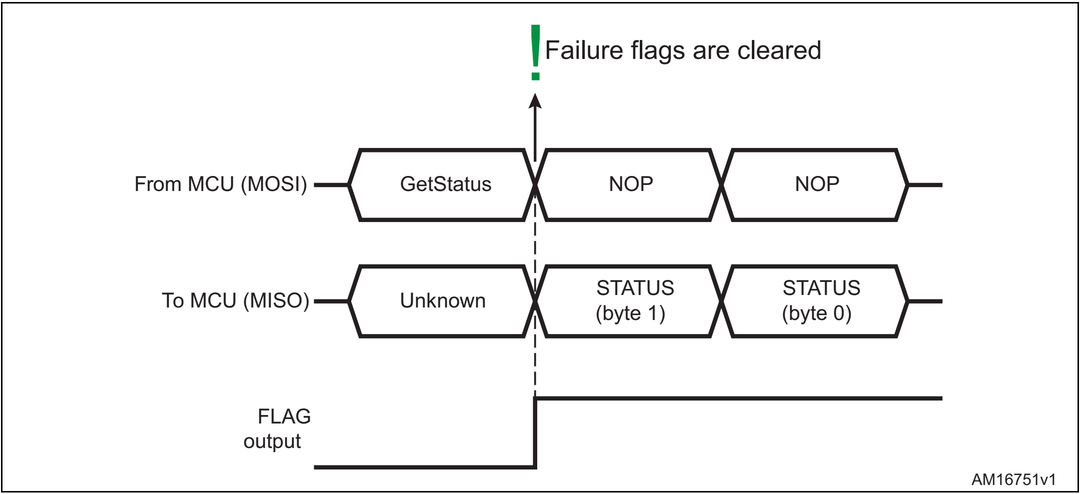
**Table 5. GetStatus byte command**

| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

The command does not need any argument and generates a response byte sequence containing the STATUS register value (most significant byte first) and releases all the latched failure flags. The response is generated before releasing the flags, so the returned STATUS value still contains all the failure information.

The failure flags are cleared as soon as the GetStatus command byte is received as shown in *Figure 7*.

**Figure 7. GetStatus command and failure flag clearing**



The response corresponds to the value of the STATUS register after the GetStatus command has been received and decoded. If the register value changes later, the response is not updated accordingly.

The STATUS register can also be read using the GetParam command; in this case the failure flags are not released.

## No operation command (NOP)

The NOP command is all zero byte (as shown in *Table 6*), which does not perform any action.

**Table 6. NOP byte command**

| Bit # | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

This dummy command byte allows the master to generate a transmission cycle without the slave (or some of the slaves in case of daisy chain configuration) to perform any operation.

## Motion commands (L6470, L6472, L648x and powerSTEP01 only)

The motion commands are used to send a request to the motion engine which is integrated into the device. Each command requires a fixed number of argument bytes. Some motion commands can be executed only if particular conditions are satisfied (see *Table 1 on page 5* for details), otherwise the command is refused and the not performable command failure flag in the STATUS register is latched (in the L648x and powerSTEP01 devices, the wrong command and the not performable command failure flags correspond to the same bit).

# 2 Daisy chain

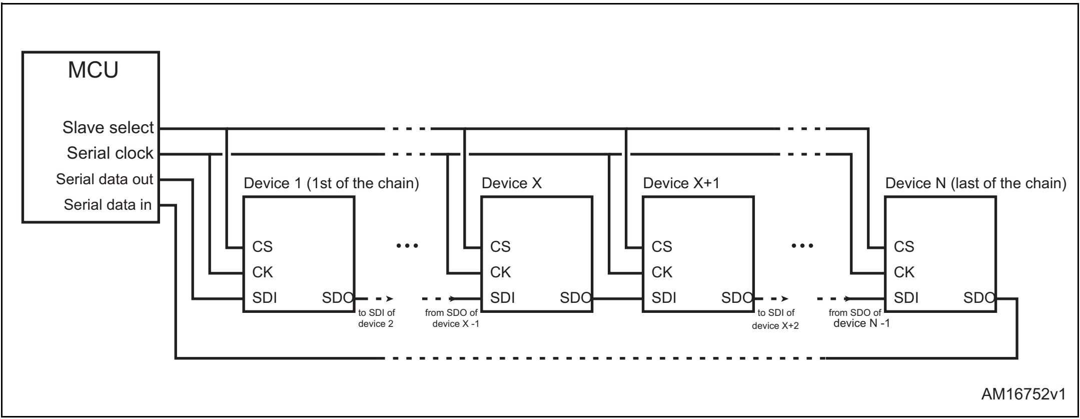
The device is compatible with the daisy chain architecture allowing the MCU to drive multiple devices with a single SPI interface.

The daisy chain architecture is obtained as follows:

– Master serial clock line is connected to the CK input of all the slaves.
– Master slave select line is connected to the $\overline{\text{CS}}$ input of all the slaves.
– Master serial data output (MOSI) is connected to the SDI input of the first slave of the chain.
– The SDO output of each slave is connected to the SDI input of the next slave, last slave of the chain excluded.
– Master serial data input (MISO) is connected to the SDO output of the last slave of the chain.

The connection diagram of the configuration is shown in *Figure 8*.

**Figure 8. Daisy chain connection diagram**

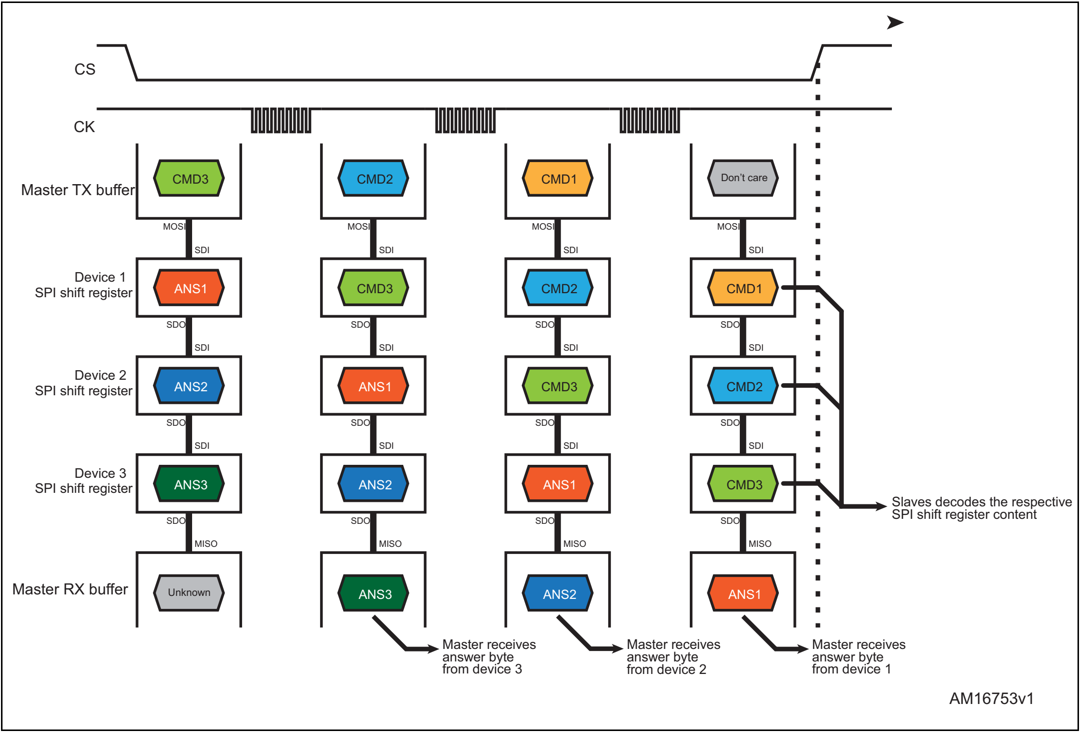


In this configuration the chain of slaves acts as a single slave with an SPI device of a number of bytes. At each communication cycle, when the master needs to transmit/receive a byte from/to a slave, the master must fill all the shift registers of the slaves before raising the $\overline{\text{CS}}$ line.

The devices are addressed according to the position of the byte in the communication cycle: the first byte transmitted by the master is received by the last device of the chain; the second one is received by the last-but-one slave and so on down to the last transmitted byte which is received by the first slave of the chain. The response bytes from the device chain are addressed to the same way: first byte received by the master has been transmitted by the last device of the chain; the second one has been transmitted by the last-but-one slave and so on down to the last received byte which has been transmitted by the first slave of the chain.

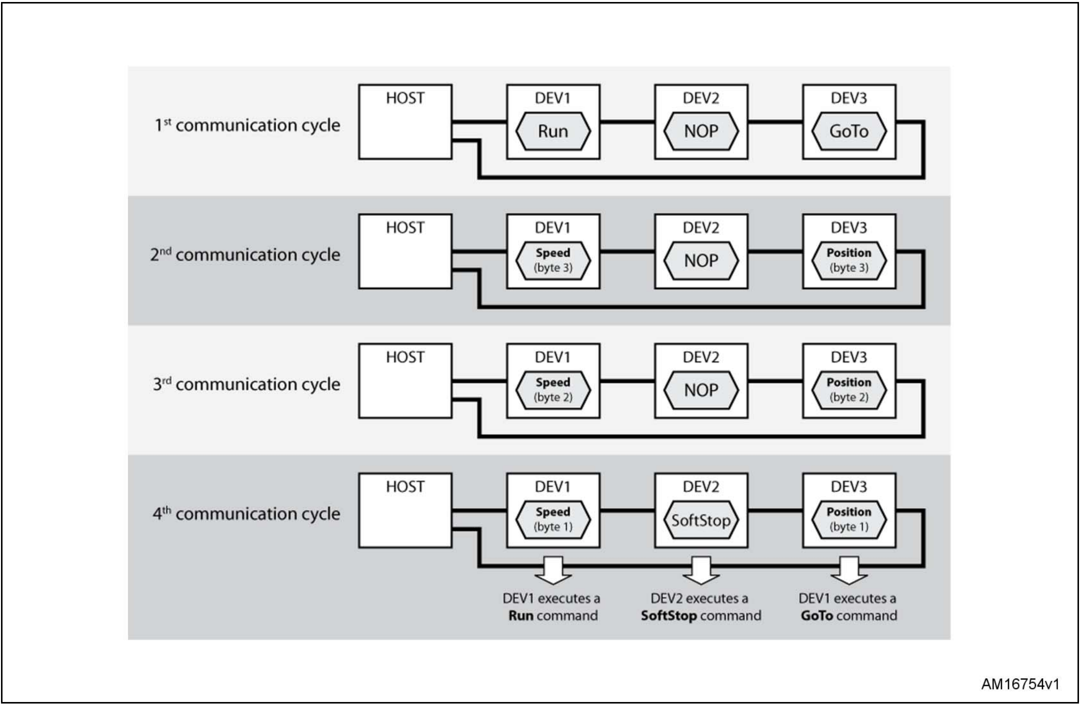
See *Figure 9* as example of a daisy chain configuration with 3 slaves.

Figure 9. Time diagram of a 3 slave daisy chain


CS
CK
Master TX buffer
Device 1 SPI shift register
Device 2 SPI shift register
Device 3 SPI shift register
Master RX buffer
CMD3
CMD2
CMD1
Don't care
ANS1
ANS2
ANS3
Unknown
MOSI
SDI
SDO
MISO
Slaves decodes the respective SPI shift register content
Master receives answer byte from device 3
Master receives answer byte from device 2
Master receives answer byte from device 1
AM16753v1

The daisy chain configuration also allows a perfect synchronization in the execution of the commands by the slave devices. Considering that all the devices acquire the respective transmitted byte at the same time (raising edge of the $\overline{\text{CS}}$ line), the different devices can start the execution of a command at the same time as shown in *Figure 10*.

Figure 10. Command synchronization example



In theory, the number of slaves that the MCU can drive using the daisy chain configuration is unlimited; in practice the maximum number of devices connected to the same SPI depends on the clock skew.

The number of slaves also limits the communication speed because every time a byte has to be transmitted to a device, the whole N slave chain has to be filled transmitting N - 1 extra bytes.

# 3 Revision history

Table 7. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 11-Apr-2013 | 1 | Initial release. |
| 30-Mar-2015 | 2 | Replaced title *on page 1* by new title “L647x, L648x and powerSTEP01 family communication protocol”.<br>Replaced “easySPIN™ family” by “L6474”, “dSPIN™ family” by “L6470 and L6472”, and “cSPIN™ family” by “L648x and powerSTEP01” in whole document.<br>Minor modifications throughout document. |