Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

Firmware Version V1.11

TMCL™ FIRMWARE MANUAL

TMCM-1310

1-Axis Stepper
Closed Loop Controller / Driver
3 A RMS / 48 V
ABN and SSI Encoder Input
18 GPIOs
USB, EtherCAT®

EtherCAT

TRINAMIC Motion Control GmbH & Co. KG
Hamburg, Germany

www.trinamic.com

TRINAMIC
MOTION CONTROL

Table of Contents

1 Features

The TMCM-1310 is a single axis stepper motor controller/driver standalone board with closed loop support. For communication an USB interface and EtherCAT®* are provided. The module supports motor currents up to 3A RMS and supply voltages up to 48V nominal. The module offers inputs for one incremental a/b/n (TTL, open-collector and differential inputs) or absolute SSI encoders (selectable in software). There are dedicated stop switch inputs, 8 general purpose inputs, and 8 general purpose outputs.

MAIN CHARACTERISTICS

Bipolar stepper motor driver
- Up to 256 microsteps per full step
- High-efficient operation, low power dissipation
- Dynamic current control
- Integrated protection: overtemperature and undervoltage
- stallGuard2™ feature for stall detection (for open loop operation)

Encoder
- Encoder input for incremental a/b/n (TTL, open-collector and differential inputs) and absolute SSI encoders (selectable in software)

Interfaces
- USB 2.0 full-speed (12Mbit/s) communication interface (mini-USB connector)
- EtherCAT LINK IN and LINK OUT (RJ45)
- Dedicated STOP_L / STOP_R inputs
- Up to 8 multi-purpose inputs (+24V compatible, incl. 2 dedicated analog inputs)
- Up to 8 multi-purpose outputs (open-drain, incl. 2 outputs for currents up to 1A)

Software
- TMCL™ remote (direct mode) and standalone operation with memory for up to 1024 TMCL commands
- Closed-loop support
- Fully supported by TMCL-IDE (PC based integrated development environment)

Electrical data
- Supply voltage: +12V… +48V DC
- Motor current: up to 3A RMS (programmable)

Mechanical data
- Board size: 110mm x 110mm, height 26.3mm

Please refer to separate TMCM-1310 Hardware Manual for additional information.

* EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.

TRINAMIC Features – Closed Loop Mode

The TMCM-1310 is mainly designed to run 2-phase stepper motors in closed loop mode. It offers an automatic motor load adaption in positioning mode, velocity mode, and torque mode, which is based on encoder feedback and closed loop control software for analysis, error detection and error correction.

The closed loop mode operation combines the advantages of a stepper driver system with the benefits of a servo drive. Thus, the TMCM-1310 is able to satisfy ambitious requirements in reliability and precision and can be used in several industrial demanding applications.

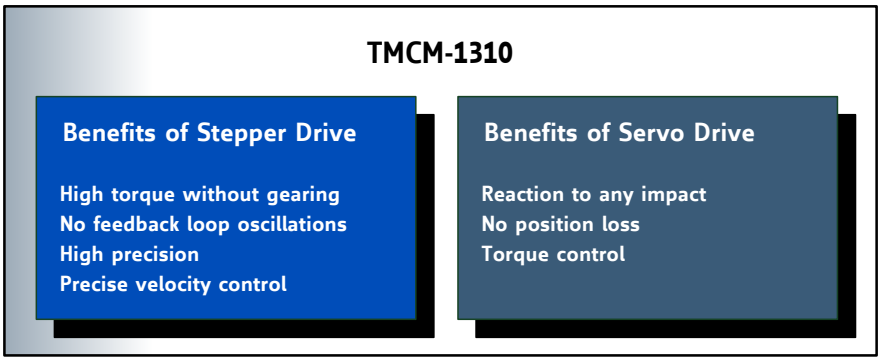

Figure 1.1 TMCM-1310 characteristics in closed loop mode

The TRINAMIC Closed Loop Mode operation

- prevents the motor from stall and step loss caused by too high load or high velocity.
- adapts the current amplitude to each motor load which is within the ranges predetermined by motor and controller/driver board characteristics.
- achieves a higher torque output than in open loop mode.
- guarantees a precise and fast positioning.
- enables velocity and positioning error compensation.

Using the TMCM-1310, energy will be saved and the motor will be kept cool.

2 Overview

The software running on the microprocessor of the TMCM-1310 consists of two parts, a boot loader and the firmware itself. Whereas the boot loader is installed during production and testing at TRINAMIC and remains untouched throughout the whole lifetime, the firmware can be updated by the user. New versions can be downloaded free of charge from the TRINAMIC website (http://www.trinamic.com).

The TMCM-1310 can be used as an EtherCAT slave device. The whole communication with the EtherCAT master follows a strict master-slave-relationship. Via the TMCL mailbox motor parameters are written and/or read using TRINAMICs TMCL protocol.

The firmware of this module is related to the standard TMCL firmware with a special range of values. The TRINAMIC Motion Control Language [TMCL] provides a set of structured motion control commands. Every motion control command can be given by a host computer or can be stored in an EEPROM on the module to form programs that run standalone. For this purpose there are not only motion control commands but also commands to control the program structure (like conditional jumps, compare and calculating).

Every command has a binary representation and a mnemonic. The binary format is used to send commands from the host to a module in direct mode, whereas the mnemonic format is used for easy usage of the commands when developing standalone TMCL applications using the TMCL-IDE (IDE means *Integrated Development Environment*).

There is also a set of configuration variables for the axis and for global parameters which allow individual configuration of nearly every function of the module. This manual gives a detailed description of all TMCL commands and their usage.

3 Communication via EtherCAT

3.1 SyncManager

The SyncManager enables consistent and secure data exchange between the EtherCAT master and the local application, and it generates interrupts to inform both sides of changes. The SyncManager is configured by the EtherCAT master. The communication direction is configurable, as well as the communication mode (buffered mode and mailbox mode). The SyncManager uses a buffer located in the memory area for exchanging data. Access to this buffer is controlled by the hardware of the SyncManager.

A buffer has to be accessed beginning with the start address, otherwise the access is denied. After accessing the start address, the whole buffer can be accessed, even the start address again, either as a whole or in several strokes. A buffer access finishes by accessing the end address, the buffer state changes afterwards. The end address cannot be accessed twice inside a frame. Two communication modes are supported by SyncManagers, the *buffered mode* and the *mailbox mode*.

3.1.1 Buffered Mode

The buffered mode allows both sides, EtherCAT master and local application, to access the communication buffer at any time. The consumer gets always the latest consistent buffer which was written by the producer, and the producer can always update the content of the buffer. *The buffered mode is used for cyclic process data.*

Data transfer between EtherCAT master (PC etc.) und slave (TMCM-1310) is done using the dual port memory of the ET1100 EtherCAT-IC on the slave. The buffered mode allows writing and reading data simultaneously without interference. If the buffer is written faster than it is read out, old data will be dropped. The buffered mode is also known as 3-buffermode. One buffer of the three buffers is allocated to the producer (for writing), one buffer to the consumer (for reading), and the third buffer keeps the last consistently written data of the producer.

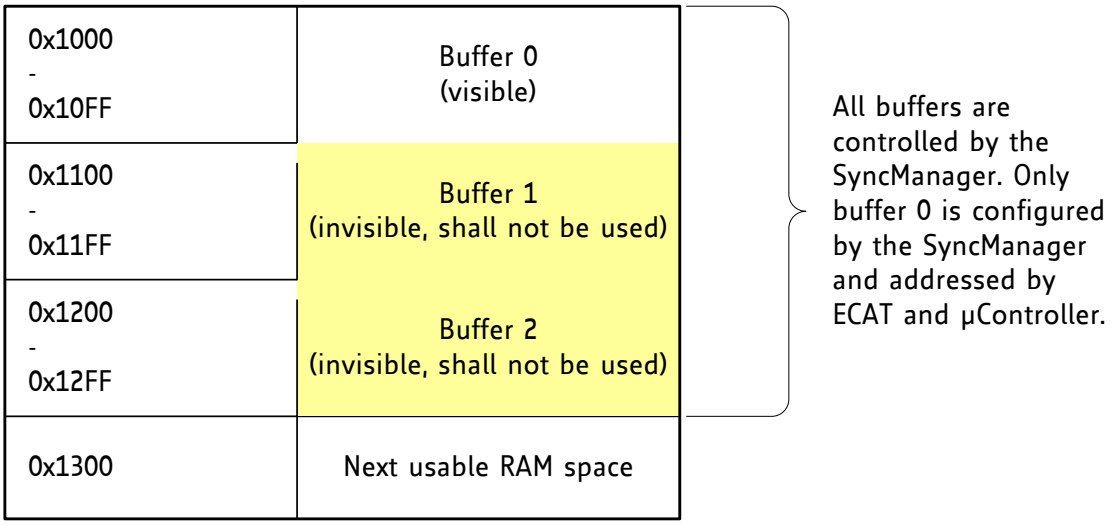

Figure 3.1 SyncManager buffer allocation

As an example, Figure 3.1 demonstrates a configuration with start address 0x1000 and length 0x100. The other buffers shall not be read or written. Access to the buffer is always directed to addresses in the range of buffer 0.

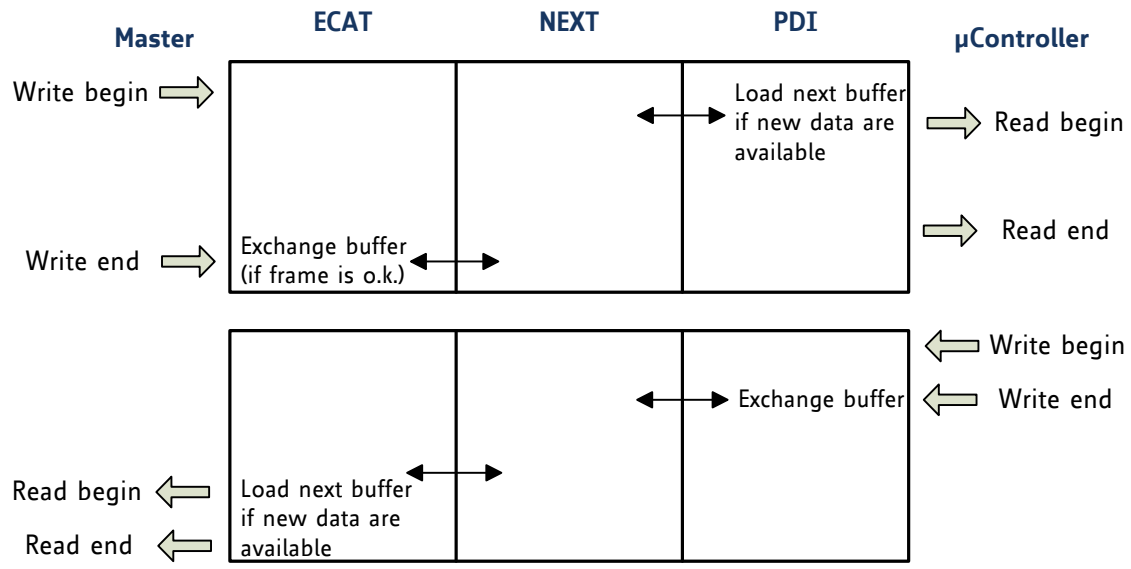

Figure 3.2 SyncManager buffered mode interaction

3.1.2 Mailbox Mode, used for TMCL-Applications

The mailbox mode implements a handshake mechanism for data exchange, so that no data will be lost. Each side, EtherCAT master or local application will get access to the buffer only after the other side has finished its access. The mailbox mode only allows alternating reading and writing. This assures that all data from the producer will reach the consumer. The mailbox mode uses just one buffer of the configured size. At first, after initialization/activation, the buffer (mailbox, MBX) is writeable. Once it is written completely, write access is blocked, and the buffer can be read out by the other side. After it was completely read out, it can be written again. The time it takes to read or write the mailbox does not matter. The mailbox mode is used for the application layer protocol.

Via the mailbox motor-parameters of the TMCM-1310 can be written/read using the TMCL protocol.

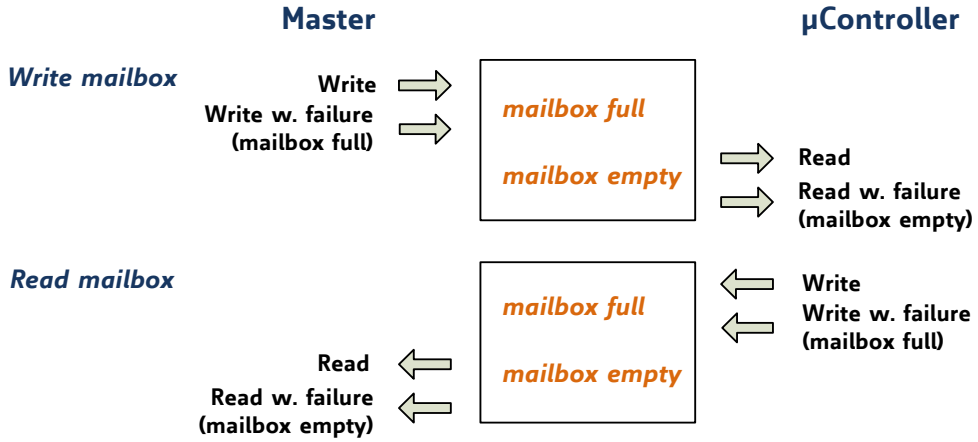

Figure 3.3 SyncManager mailbox interaction

3.2 EtherCAT Slave State Machine

The EtherCAT slave state machine has four states, which are shown in Figure 3.4. After power ON the slave state machine is in the *Init state*. In this situation mailbox and process data communication is impossible. The EtherCAT master initializes the SyncManager channels 0 and 1 for the communication via mailbox.

While changeover from *Init state* to *Pre-Operational state* the EtherCAT slave checks the correct initialization of the mailbox. Afterwards mailbox communication is possible. Now, in the *Pre-Operational state* the master initializes the SyncManager channels for the process data and the FMMU channels. Furthermore adjustments are sent, which differ from the default values.

While changeover from *Pre-Operational state* to *Safe-Operational state* the EtherCAT slave checks the correct initialization of the SyncManager channels for the process data as well as the adjustments for the Distributed Clocks. Before accepting the change of state, the EtherCAT slave copies actual input data into the accordant DP-RAM array of the EtherCAT slave controller. In the *Safe-Operational state* mailbox and process data communication are possible, but the slave holds its outputs in a safe situation and actualizes the input data periodically.

Before the EtherCAT slave changes the state to *Operational* it has to transfer valid output data. In the *Operational state* the EtherCAT slave copies the output data from the EtherCAT master to its outputs. Process data communication and mailbox communication are possible now.

The *Bootstrap state* is only used for updating the firmware. This state is reachable form the *Init state*. During *Bootstrap state* mailbox communication is available over *File-Access over EtherCAT*. Beyond this mailbox communication or process data communication is not possible.

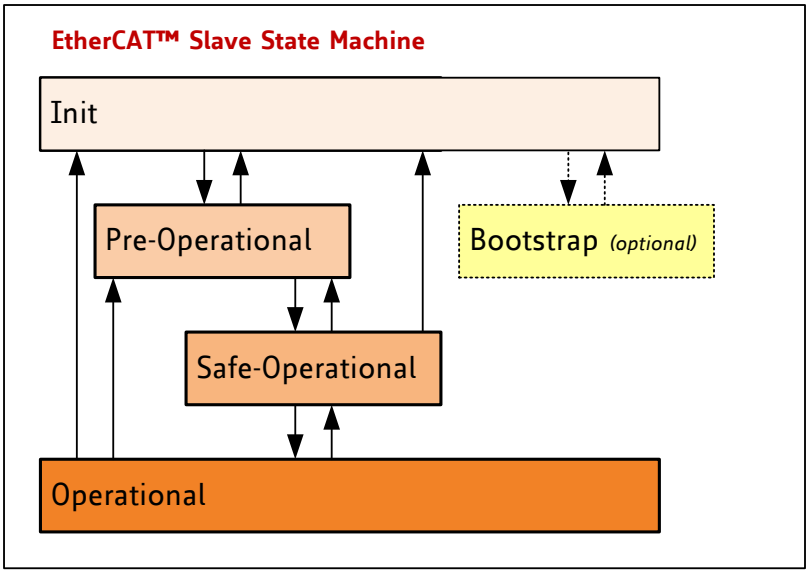

Figure 3.4 EtherCAT™ slave state machine

State / state change	Services
Init	- No communication on application layer - Master has access to the DL-information registers

Init to Pre-Operational	- Master configures registers, at least:
	  • DL address register
	  • SyncManager channels for mailbox communication
	- Master initializes Distributed Clock synchronization
	- Master requests *Pre-Operational* state
	  • Master sets AL control register
	- Wait for AL status register confirmation

Pre-Operational	- Mailbox communication on the application layer - No process data communication

Pre-Operational to Safe-Operational	- Master configures parameters using the mailbox:
	  • e.g., process data mapping
	- Master configures DL Register:
	  • SyncManager channels for process data communication
	  • FMMU channels
	- Master requests *Safe-Operational state*
	- Wait for AL Status register confirmation

Safe-Operational	- Mailbox communication on the application layer - Process data communication, but only inputs are evaluated. Outputs remain in safe state
Safe-Operational to Operational	- Master sends valid outputs - Master requests *Operational state* (AL Control/Status) - Wait for AL Status register confirmation
Operational	- Inputs and outputs are valid
Bootstrap	Recommended if firmware updates are necessary - State changes only from and to *Init* - No Process Data communication - Mailbox communication on application layer, only FoE protocol available (possibly limited file range)

THREE LEDS DISPLAY THE ACTUAL ACTIVITY:

Green LED	Description	
EtherCAT0 LINK OUT state	OFF	No link.
	blinking	Link and activity.
	single flash	Link without activity.
EtherCAT LINK IN state	OFF	No link.
	blinking	Link and activity.
	single flash	Link without activity.
EtherCAT RUN state	OFF	The device is in state INIT.
	blinking	The device is in state PRE-OPERATIONAL.
	single flash	The device is in state SAFE-OPERATONAL.
	ON	The device is in state OPERATIONAL.
	flickering (fast)	The device is in state BOOTSTRAP.

3.3 EtherCAT Firmware Update

For firmware updates the EtherCAT state machine of the slave has to be switched to *Bootstrap state*. The *file access over EtherCAT* protocol (FoE) is used.

THE TWO MAILBOXES FOR DATA TRANSFERS HAVE THE FOLLOWING PARAMETERS:

- Data output buffer: Start-address: 4096, length: 268 byte
- Data input buffer: Start-address: 4364, length: 40 byte

3.4 Process Data

In standard configuration for data transfer the following buffers are used (slave view):

DATA OUTPUT BUFFER / ETHERCAT MASTER -> SLAVE DATA TRANSFER

Data output buffer: Start-address: 4096(0x1000), length: first 8 bytes

<table>
<tr><th>Start address</th><th>End address</th><th>Data type</th><th>Data value / contents</th></tr>
<tr><td>0x1000</td><td>0x1003</td><td>UNSIGNED32</td><td>Controller Mode
<table><tr><th>Bit</th><th>Description</th></tr><tr><td>0</td><td>No operation</td></tr><tr><td>1</td><td>Position mode</td></tr><tr><td>2</td><td>Velocity mode</td></tr><tr><td>3</td><td>Torque mode</td></tr></table></td></tr>
<tr><td>0x1004</td><td>0x1007</td><td>SIGNED32</td><td>Value
(position / velocity / current)</td></tr>
</table>

DATA INPUT BUFFER / ETHERCAT SLAVE -> MASTER DATA TRANSFER

Data input buffer: Start-address: 4216(0x1078), length: first 44 bytes

Start address	End address	Data type	Data value / contents
0x1078	0x107B	SIGNED32	Target Position Command for read out: GAP 0
0x107C	0x107F	SIGNED32	Actual position Command for read out: GAP 1
0x1080	0x1083	SIGNED32	Virtual actual position Command for read out: GAP 233.
0x1084	0x1087	SIGNED32	Encoder Position Command for read out: GAP 209
0x1088	0x108B	SIGNED32	Target velocity Command for read out: GAP 2
0x108C	0x108F	SIGNED32	Actual velocity Command for read out: GAP 3
0x1090	0x1093	SIGNED32	Measured velocity Command for read out: GAP 131
0x1094	0x1097	UNSIGNED32	Status word Command for read out: GAP 18)
0x1098	0x109B	UNSIGNED32	Error flags Bit 0: Target reached Bit 1: Velocity reached Bit 2: Closed loop Bit 3: Position mode Bit 4: Velocity mode Bit 5: Torque mode Bit 6: Home switch Bit 7: Left stop switch Bit 8: Right stop switch Bit 9: Undervoltage Bit 10: Overvoltage Bit 11: Overtemperature
0x109C	0x109F	SIGNED32	Scaler Command for read out: GAP123
0x1100	0x1103	SIGNED32	Delta / torque Command for read out: GAP 14
0x1104	0x1107	SIGNED32	Gamma Command for read out: 230

All numbers are stored in little endian format. (least significant byte is stored at the lowest address)

3.5 TMCL Mailbox

The TMCM-1310 slave module supports the TMCL protocol in direct mode. The communication follows a strict master-slave-relationship. Via the TMCL mailbox motor-parameters can be read and/or written.

3.6 Binary Command Format

Every command has a mnemonic and a binary representation. When commands are sent from a host to a module, the binary format has to be used. Every command consists of a one-byte command field, a one-byte type field, a one-byte motor/bank field and a four-byte value field. So the binary representation of a command always has seven bytes.

TRANSMIT AN 8-BYTE COMMAND:

Bytes	Meaning
1	Module address
1	Command number
1	Type number
1	Motor or Bank number
4	Value (*MSB first!*)

Every time a command has been sent to a module, the module sends a reply.

RECEIVE AN 8-BYTE REPLY:

Bytes	Meaning
1	Reply address
1	Module address
1	Status (e.g. 100 means *no error*)
1	Command number
4	Value (MSB first!)

3.7 Status Codes

The reply contains a status code.

THE STATUS CODE CAN HAVE ONE OF THE FOLLOWING VALUES:

Code	Meaning
100	Successfully executed, no error
2	Invalid command
3	Wrong type
4	Invalid value
5	Configuration EEPROM locked
6	Command not available
8	Parameter is password protected

4 Operation with USB Interface

In direct mode and most cases the TMCL communication over USB follows a strict master/slave relationship. That is, a host computer (e.g. PC/PLC) acting as the interface bus master will send a command to the TMCL-1310. The TMCL interpreter on the module will then interpret this command, do the initialization of the motion controller, read inputs and write outputs or whatever is necessary according to the specified command. As soon as this step has been done, the module will send a reply back over USB to the bus master. Only then should the master transfer the next command.

4.1 Binary Command Format for USB Interface

When commands are sent from a host to a module, the binary format has to be used. Every command consists of a one-byte command field, a one-byte type field, a one-byte motor/bank field and a four-byte value field. So the binary representation of a command always has seven bytes. When a command is to be sent via USB interface, it has to be enclosed by an address byte at the beginning and a checksum byte at the end. In this case it consists of nine bytes.

THE BINARY COMMAND FORMAT FOR USB IS AS FOLLOWS:

Bytes	Meaning
1	Module address
1	Command number
1	Type number
1	Motor or Bank number
4	Value (MSB first!)
1	Checksum

The checksum is calculated by adding up all the other bytes using an 8-bit addition.

CHECKSUM CALCULATION

As mentioned above, the checksum is calculated by adding up all bytes (including the module address byte) using 8-bit addition. Here are two examples to show how to do this:

in C:

```
unsigned char i, Checksum;
unsigned char Command[9];

//Set the "Command" array to the desired command
Checksum = Command[0];
for(i=1; i<8; i++)
   Checksum+=Command[i];

Command[8]=Checksum; //insert checksum as last byte of the command
//Now, send it to the module
```

4.2 Reply Format

Every time a command has been sent to a module, the module sends a reply.

THE REPLY FORMAT FOR USB IS AS FOLLOWS:

Bytes	Meaning
1	Reply address
1	Module address
1	Status (e.g. 100 means "no error")
1	Command number
4	Value (MSB first!)
1	Checksum

The checksum is also calculated by adding up all the other bytes using an 8-bit addition. Do not send the next command before you have received the reply!

4.2.1 Status Codes

The reply contains a status code.

The status code can have one of the following values:

Code	Meaning
100	Successfully executed, no error
101	Command loaded into TMCL program EEPROM
1	Wrong checksum
2	Invalid command
3	Wrong type
4	Invalid value
5	Configuration EEPROM locked
6	Command not available

4.3 Standalone Applications

The module is equipped with an EEPROM for storing TMCL applications. You can use TMCL-IDE for developing standalone TMCL applications. You can load them down into the EEPROM and then it will run on the module. The TMCL-IDE contains an editor and the TMCL assembler where the commands can be entered using their mnemonic format. They will be assembled automatically into their binary representations. Afterwards this code can be downloaded into the module to be executed there.

5 The ASCII Interface

There is also an ASCII interface that can be used to communicate with the module and to send some commands as text strings. The ASCII format can be used with the USB interface, only.

PROCEED AS FOLLOWS

- The ASCII command line interface is entered by sending the binary command 139 (enter ASCII mode).
- Afterwards the commands are entered as in the TMCL-IDE. Please note that only those commands, which can be used in direct mode, also can be entered in ASCII mode.
- For leaving the ASCII mode and re-entering the binary mode enter the command BIN.

5.1 Format of the Command Line

As the first character, the address character has to be sent. The address character is *A* when the module address is 1, *B* for modules with address 2 and so on. After the address character there may be spaces (but this is not necessary). Then, send the command with its parameters. At the end of a command line a <CR> character has to be sent.

EXAMPLES FOR VALID COMMAND LINES

```
AMVP ABS, 1, 50000
A MVP ABS, 1, 50000
AROL 2, 500
A MST 1
ABIN
```

These command lines would address the module with address 1. To address e.g. module 3, use address character *C* instead of *A*. The last command line shown above will make the module return to binary mode.

5.2 Format of a Reply

After executing the command the module sends back a reply in ASCII format. The reply consists of:

- the address character of the host (host address that can be set in the module)
- the address character of the module
- the status code as a decimal number
- the return value of the command as a decimal number
- a <CR> character

So, after sending AGAP 0, 1 the reply would be BA 100 -5000 if the actual position of axis 1 is -5000, the host address is set to 2 and the module address is 1. The value 100 is the status code 100 that means *command successfully executed*.

5.3 Commands Used in ASCII Mode

The following commands can be used in ASCII mode: ROL, ROR, MST, MVP, SAP, GAP, STAP, RSAP, SGP, GGP, STGP, RSGP, RFS, SIO, GIO, SCO, GCO, CCO, UF0, UF1, UF2, UF3, UF4, UF5, UF6, and UF7.

SPECIAL COMMANDS WHICH ARE ONLY AVAILABLE IN ASCII MODE

- BIN: This command quits ASCII mode and returns to binary TMCL mode.
- RUN: This command can be used to start a TMCL program in memory.
- STOP: Stops a running TMCL application.

5.4 Configuring the ASCII Interface

The module can be configured so that it starts up either in binary mode or in ASCII mode. ***Global parameter 67 is used for this purpose*** (please see also chapter 11.1).

Bit 0 determines the startup mode: if this bit is set, the module starts up in ASCII mode, else it will start up in binary mode (default).

Bit 4 and Bit 5 determine how the characters that are entered are echoed back. Normally, both bits are set to zero. In this case every character that is entered is echoed back when the module is addressed. Character can also be erased using the backspace character (press the backspace key in a terminal program).

When bit 4 is set and bit 5 is clear the characters that are entered are not echoed back immediately but the entire line will be echoed back after the <CR> character has been sent.

When bit 5 is set and bit 4 is clear there will be no echo, only the reply will be sent. This may be useful in RS485 systems.

6 TMCL Commands

6.1 Motion Commands

These commands control the motion of the motor. They are the most important commands and can be used in direct mode or in standalone mode.

Mnemonic	Command number	Meaning
ROL	2	Rotate left
ROR	1	Rotate right
MVP	4	Move to position
MST	3	Motor stop
RFS	13	Reference search
SCO	30	Store coordinate
CCO	32	Capture coordinate
GCO	31	Get coordinate

6.2 Parameter Commands

These commands are used to set, read and store axis parameters or global parameters. Axis parameters can be set independently for the axis, whereas global parameters control the behavior of the module itself. These commands can also be used in direct mode and in standalone mode.

Mnemonic	Command number	Meaning
SAP	5	Set axis parameter
GAP	6	Get axis parameter
STAP	7	Store axis parameter into EEPROM
RSAP	8	Restore axis parameter from EEPROM
SGP	9	Set global parameter
GGP	10	Get global parameter
STGP	11	Store global parameter into EEPROM
RSGP	12	Restore global parameter from EEPROM

6.3 Control Commands

These commands are used to control the program flow (loops, conditions, jumps etc.). It does not make sense to use them in direct mode. They are intended for standalone mode only.

Mnemonic	Command number	Meaning
JA	22	Jump always
JC	21	Jump conditional
COMP	20	Compare accumulator with constant value
CSUB	23	Call subroutine
RSUB	24	Return from subroutine
WAIT	27	Wait for a specified event
STOP	28	End of a TMCL program

6.4 I/O Port Commands

These commands control the external I/O ports and can be used in direct mode and in standalone mode.

Mnemonic	Command number	Meaning
SIO	14	Set output
GIO	15	Get input

6.5 Calculation Commands

These commands are intended to be used for calculations within TMCL applications. Although they could also be used in direct mode it does not make much sense to do so.

Mnemonic	Command number	Meaning
CALC	19	Calculate using the accumulator and a constant value
CALCX	33	Calculate using the accumulator and the X register
AAP	34	Copy accumulator to an axis parameter
AGP	35	Copy accumulator to a global parameter
ACO	39	Copy accu to coordinate

For calculating purposes there is an accumulator (or accu or A register) and an X register. When executed in a TMCL program (in standalone mode), all TMCL commands that read a value store the result in the accumulator. The X register can be used as an additional memory when doing calculations. It can be loaded from the accumulator.

When a command that reads a value is executed in direct mode the accumulator will not be affected. This means that while a TMCL program is running on the module (standalone mode), a host can still send commands like GAP and GGP to the module (e.g. to query the actual position of the motor) without affecting the flow of the TMCL program running on the module.

6.6 Interrupt Commands

Due to some customer requests, interrupt processing has been introduced in the TMCL firmware for ARM based modules.

Mnemonic	Command number	Meaning
EI	25	Enable interrupt
DI	26	Disable interrupt
VECT	37	Set interrupt vector
RETI	38	Return from interrupt

6.6.1 Interrupt Types

There are many different interrupts in TMCL, like timer interrupts, stop switch interrupts, position reached interrupts, and input pin change interrupts. Each of these interrupts has its own interrupt vector. Each interrupt vector is identified by its interrupt number. Please use the TMCL included file *Interrupts.inc* for symbolic constants of the interrupt numbers.

6.6.2 Interrupt Processing

When an interrupt occurs and this interrupt is enabled and a valid interrupt vector has been defined for that interrupt, the normal TMCL program flow will be interrupted and the interrupt handling routine will be called. Before an interrupt handling routine gets called, the context of the normal program will be saved automatically (i.e. accumulator register, X register, TMCL flags).

There is no interrupt nesting, i.e. all other interrupts are disabled while an interrupt handling routine is being executed.

On return from an interrupt handling routine, the context of the normal program will automatically be restored and the execution of the normal program will be continued.

6.6.3 Interrupt Vectors

The following table shows all interrupt vectors that can be used.

Interrupt number	Interrupt type
0	Timer 0
1	Timer 1
2	Timer 2
3	Target position reached
15	stallGuard2
21	Deviation
27	Left stop switch
28	Right stop switch
39	Input change 0
40	Input change 1
41	Input change 2
42	Input change 3
43	Input change 4
44	Input change 5
45	Input change 6
46	Input change 7
255	Global interrupts

6.6.4 Further Configuration of Interrupts

Some interrupts need further configuration (e.g. the timer interval of a timer interrupt). This can be done using SGP commands with parameter bank 3 (SGP <type>, 3, <value>). Please refer to the SGP command (paragraph 6.8.9) for further information about that.

6.6.5 Using Interrupts in TMCL

To use an interrupt the following things have to be done:

- Define an interrupt handling routine using the VECT command.
- If necessary, configure the interrupt using an SGP <type>, 3, <value> command.
- Enable the interrupt using an EI <interrupt> command.
- Globally enable interrupts using an EI 255 command.
- An interrupt handling routine must always end with a RETI command

The following example shows the use of a timer interrupt:

```
    VECT 0, Timer0Irq  //define the interrupt vector
    SGP 0, 3, 1000     //configure the interrupt: set its period to 1000ms
    EI 0               //enable this interrupt
    EI 255             //globally switch on interrupt processing

//Main program: toggles output 3, using a WAIT command for the delay
Loop:
    SIO 3, 2, 1
    WAIT TICKS, 0, 50
    SIO 3, 2, 0
    WAIT TICKS, 0, 50
    JA Loop

//Here is the interrupt handling routine
Timer0Irq:
    GIO 0, 2           //check if OUT0 is high
    JC NZ, Out0Off     //jump if not
    SIO 0, 2, 1        //switch OUT0 high
    RETI               //end of interrupt
Out0Off:
    SIO 0, 2, 0        //switch OUT0 low
    RETI               //end of interrupt
```

In the above example, the interrupt numbers are used directly. To make the program better readable use the provided include file *Interrupts.inc*. This file defines symbolic constants for all interrupt numbers which can be used in all interrupt commands. The beginning of the program above then looks like the following:

```
#include Interrupts.inc
    VECT TI_TIMER0, Timer0Irq
    SGP TI_TIMER0, 3, 1000
    EI TI_TIMER0
    EI TI_GLOBAL
```

Please also take a look at the other example programs.

6.7 ASCII Commands

Mnemonic	Command number	Meaning
-	139	Enter ASCII mode
BIN	-	Quit ASCII mode and return to binary mode. This command can only be used in ASCII mode.

6.8 Commands

The module specific commands are explained in more detail on the following pages. They are listed according to their command number.

6.8.1 ROR (rotate right)

With this command the motor will be instructed to rotate with a specified velocity in *positive* direction (increasing the position counter).

Like on all other TMCL modules, the motor will be accelerated or decelerated to the speed given with the command. The speed is given in microsteps per second (pps). For conversion of this value into rounds per minute etc. please refer to chapter 0, also.

The range is -327.678.000... +327.679.999.

Internal function: first, velocity mode is selected. Then, the velocity value is transferred to axis parameter #2 (*target velocity*).

Related commands: ROL, MST, SAP, GAP

Mnemonic: ROR 0, <velocity>

Binary representation:

INSTRUCTION NO.	TYPE	MOT/BANK	VALUE
1	don't care	<motor> 0*	<velocity> -327.678.000... +327.679.999

*** Motor number is always 0 as the module support just one axis**

Reply in direct mode:

STATUS	VALUE
100 – OK	don't care

Example:

Rotate right, velocity = 10000
Mnemonic: ROR 0, 10000

Binary:

Byte Index	0	1	2	3	4	5	6	7
Function	Target-address	Instruction Number	Type	Motor/ Bank	Operand Byte3	Operand Byte2	Operand Byte1	Operand Byte0
Value (hex)	$01	$01	$00	$00	$00	$00	$27	$10

6.8.2 ROL (rotate left)

With this command the motor will be instructed to rotate with a specified velocity in *positive* direction (increasing the position counter).

Like on all other TMCL modules, the motor will be accelerated or decelerated to the speed given with the command. The speed is given in microsteps per second (pps). For conversion of this value into rounds per minute etc. please refer to chapter 0, also.

The range is -327.678.000... +327.679.999.

Internal function: first, velocity mode is selected. Then, the velocity value is transferred to axis parameter #2 (*target velocity*).

Related commands: ROR, MST, SAP, GAP

Mnemonic: ROL 0, <velocity>

Binary representation:

INSTRUCTION NO.	TYPE	MOT/BANK	VALUE
2	don't care	<motor> 0*	<velocity> -327.678.000... +327.679.999

*** Motor number is always 0 as the module support just one axis**

Reply in direct mode:

STATUS	VALUE
100 – OK	don't care

Example:

Rotate left, velocity = 10000
Mnemonic: ROL 0, 10000

Binary:

Byte Index	0	1	2	3	4	5	6	7
Function	Target-address	Instruction Number	Type	Motor/ Bank	Operand Byte3	Operand Byte2	Operand Byte1	Operand Byte0
Value (hex)	$01	$02	$00	$00	$00	$00	$27	$10

6.8.3 MST (motor stop)

The motor will be instructed to stop.

Internal function: the axis parameter *target velocity* is set to zero.

Related commands: ROL, ROR, SAP, GAP

Mnemonic: MST 0

Binary representation:

INSTRUCTION NO.	TYPE	MOT/BANK	VALUE
3	don't care	<motor> 0*	don't care

*** Motor number is always 0 as the module support just one axis**

Reply in direct mode:

STATUS	VALUE
100 – OK	don't care

Example:

Stop motor 0
Mnemonic: MST 0

Binary:

Byte Index	0	1	2	3	4	5	6	7
Function	Target-address	Instruction Number	Type	Motor/ Bank	Operand Byte3	Operand Byte2	Operand Byte1	Operand Byte0
Value (hex)	$01	$03	$00	$00	$00	$00	$00	$00

6.8.4 MVP (move to position)

The motor will be instructed to move to a specified relative or absolute position or a pre-programmed coordinate. It will use the acceleration/deceleration ramp and the positioning speed programmed into the unit. This command is non-blocking – that is, a reply will be sent immediately after command interpretation and initialization of the motion controller. Further commands may follow without waiting for the motor reaching its end position. The maximum velocity and acceleration are defined by axis parameters #4 and #5.

UNITS AND RANGE

Open loop: the range of the MVP command is 32 bit signed (-2.147.483.648... +2.147.483.647). The unit is microsteps.

Closed loop: the range of the MVP command is 32 bit signed (-2.147.483.648... +2.147.483.647). The unit is encoder steps.

Positioning can be interrupted using MST, ROL or ROR commands.

THREE OPERATION TYPES ARE AVAILABLE:

- Moving to an absolute position in the range from -2.147.483.648... +2.147.483.647.
- Starting a relative movement by means of an offset to the actual position. In this case, the new resulting position value must not exceed the above mentioned limits, too.
- Moving the motor to a (previously stored) coordinate (refer to SCO for details).

Please note, that the distance between the actual position and the new one should not be more than 2.147.483.647 (2^{31}-1) microsteps resp. encoder steps. Otherwise the motor will run in the opposite direction in order to take the shorter distance.
When moving to a coordinate, the coordinate has to be set properly in advance with the help of the SCO, CCO or ACO command.

Internal function: A new position value is transferred to the axis parameter #0 (target position).

Related commands: SAP, GAP, SCO, CCO, GCO, MST

Mnemonic: MVP <ABS|REL|COORD>, 0, <position|offset|coordinate number>

Binary representation:

INSTRUCTION NO.	TYPE	MOT/BANK	VALUE
4	0 ABS – absolute	<motor> 0*	<position> -2.147.483.648... +2.147.483.647
	1 REL – relative		<offset> -2.147.483.648... +2.147.483.647
	2 COORD – coordinate		<coordinate number> 0... 20

*** Motor number is always 0 as only one motor is involved**

Reply in direct mode:

STATUS	VALUE
100 – OK	don't care