Thorlabs APT Motor Controllers

Host-Controller Communications Protocol

Date: 21-March-2006
Revision: A
Document Ref.No.:

1. Purpose and Scope

This document describes the low-level communications protocol and commands used between the host PC and motor controller units within the APT family. The information contained in this document is intended to help third party system developers to write their own applications to interface to the Thorlabs range of motor controllers without the constraints of using a particular operating system or hardware platform. The commands described here are those which are necessary to control motor movement; there is an additional set of commands, used for calibration or test, which will not be detailed as these are not required for the external system developer.

2. Electrical interface

The electrical interface within the APT motor controllers uses a Future Technology Devices International (FTDI), type FT232BM USB peripheral chip to communicate with the host PC. This is a USB2.0 compliant USB1.1 device. While the overall communications protocol is independent of the transport layer (for example, Ethernet or serial communications could also be used to carry commands from the host to the controller), the initial enumeration scheme described below is specific to the USB environment.

3. Device Enumeration

The APT Server PC software supplied is designed to work with a number of different types of motor controller. The purpose of the enumeration phase is for the host to establish what devices are present in the system and initialise the GUI accordingly. Initially this is done by enumerating the USB devices connected to the system and reading the serial number information contained in the USB device descriptor.

For the Thorlabs range of controllers, this serial number is an 8-digit decimal number. The first two digits (referred to as the prefix) describe the type of controller, while the rest of the digits make up a unique serial number. By extracting the prefix, the host can therefore establish what type of hardware is connected to the system.

In most cases, specifically with benchtop controllers, the USB serial number contains sufficient information for the host to know the exact type of hardware is connected. There is a range of other controller products where several controller cards (without their own individual USB peripheral chip) can be plugged into a motherboard and it is only the motherboard that has USB connectivity. These are generally referred to as a card slot type of system (for example, the BSC103 controller). In these systems, a second enumeration state is carried out; however, this second state is done within the protocol framework that will be detailed in this document.

For the motor controller types, the USB prefixes can be the following:

USB S/N	Type of product	Thorlabs code
20xxxxxx	Legacy single channel stepper driver	BSC001
25xxxxxx	Legacy single channel mini stepper driver	BMS001
30xxxxxx	Legacy dual channel stepper driver	BSC002

35xxxxxx	Legacy dual channel mini stepper driver	BMS002
40xxxxxx	Single channel stepper driver	BSC101
60xxxxxx	OptoSTDriver (mini stepper driver)	OST001
63xxxxxx	OptoDCDriver (mini DC servo driver)	ODC001
70xxxxxx	Three chanel card slot stepper driver	BSC103
80xxxxxx	Stepper Driver T-Cube	TST001
83xxxxxx	DC servo driver T-Cube	TDC001

Of these listed above, only the BSC103 (serial number prefix 70) is a card slot type of controller.

4. Overview of the Communications Protocol

The communications protocol used in the Thorlabs controllers is based on the message structure that always starts with a fixed length, 6-byte *message header* which, in some cases, is followed by a variable length *data packet*. For simple commands, the 6-byte message header is sufficient to convey the entire command. For more complex commands, for example, when a set of parameters needs to be passed on, the 6 byte header is not enough and in this case the header is followed by the data packet.

The header part of the message always contains information that indicates whether or not a data packet follows the header and if so, the number of bytes that the data packet contains. In this way the receiving process is able to keep tracks of the beginning and the end of messages.

Note that in the section below describing the various byte sequences, the C-type of notation will be used for hexadecimal values (e.g. 0x55 means 55 hexadecimal) and logical operators (e.g. | means logic bitwise OR). Values that are longer than a byte follow the Intel little-endian format.

5. Description of the message header

The 6 bytes in the message header are shown below:

Byte:	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
Meaning if no data packet to follow	message ID		param1	param2	source	dest
Meaning if data packet to follow	message ID		datapacket length		source | 0x80	dest

The meaning of some of the fields depends on whether or not the message is followed by a data packet. This is indicated by the most significant bit in byte 4, called the source byte, therefore the receiving process must first check if the MSB of byte 4 is set.

If this bit is not set, then the message is a header-only message and the interpretation of the bytes is as follows:

message ID: describes what the action the message requests

param1:	first parameter (if the command requires a parameter, otherwise 0)
param2:	second parameter (if the command requires a parameter, otherwise 0)
source:	the source of the message
dest:	the destination module

The meaning of the source and destination bytes will be detailed later.
If the MSB of byte 4 is set, then the message will be followed by a data packet and the interpretation of the header is the following:

message ID:	describes what the action the message requests
datapacket length:	number of bytes to follow after header Note: although this is a 2-byte long field, currently no datapacket exceeds 255 bytes in length.
source | 0x80:	the source of the destination logic ORed with 0x80
dest:	the destination module

The source and destination fields require some further explanation. In general, as the name suggests, they are used to indicate the source and destination of the message. In non card-slot type of systems the source and destination of messages is always unambiguous, as each module appears as a separate USB node in the system. In these systems, the when the host sends a message to the module, it uses the source identification byte of 0x01 (meaning host) and the destination byte of 0x50 (meaning "generic USB unit"). (In messages that the module sends back to the host, of course the source and destination bytes are swapped.)

In card-slot type of systems, there is only one USB node for a number of sub-modules, so this simple scheme cannot be used. Instead, the host sends a message to the motherboard that the sub-modules are plugged into, with the destination field of each message indicating which *slot* the message must be routed to. Likewise, when the host receives a message from a particular sub-module, it knows from the source byte which slot is the origin of the message.

Numerically, the following values are currently used for the source and destination bytes:

```
0x01        Host controller (i.e control PC)
0x11        Rack controller (i.e. motherboard in a card slot system)
            or comms router board
0x21        Bay 0 in a card slot system
0x22        Bay 1 in a card slot system
0x23        etc.
0x24        etc.
0x25        etc.
0x26        etc.
…
0x2A        Bay 9 in a card slot system
0x50        Generic USB hardware unit
```

In slot-type systems the host can also send messages to the motherboard that the sub-modules are plugged into (destination byte = 0x11). In fact, as a very first step in the communications process, the host must send a message to the motherboard to find out which slots are used in the system.

Note that although in theory this scheme would allow communication between individual sub-modules (the source of the message could be a sub-module and the destination another one), current systems do not use this option.

6. General message exchange rules

The type of messages used in the communications exchange between the host and the sub-modules can be divided into 4 general categories:

(a) Host issues a command, sub-module carries out the command without acknowledgement (i.e. no response is sent back to the host).

Typically, these are commands which require no information from the sub-module, for example setting the digital outputs to a particular state.

(b) Host issues a command (message request) and the sub-module responds by sending data back to the host.

For example, the host may request the sub-module to report the state of the digital inputs.

(c) Following a command from the host, the sub-module periodically sends a message to the host without further prompting.

These messages are referred to as *status update messages*. These are typically sent automatically every 100 msec from the sub-module to the host, showing, amongst other things, the position of the stage the controller is connected to. The meters on the APT User GUI rely on these messages to show the up-to-date status of the stage.

(d) Rarely – error messages, exceptions. These are spontaneously issued by the sub-module if some error occurs. For example, if the power supply fails in the sub-module, a message is sent to the host PC to inform the user.

Apart from the last two categories (status update messages and error messages), in general the message exchanges follow the SET -> REQUEST -> GET pattern, i.e. for most commands a trio of messages are defined. The SET part of the trio is used by the host (or, sometimes in card-slot systems the motherboard) to set some parameter or other. If then the host requires some information from the sub-module, then it may send a REQUEST for this information, and the sub-module responds with the GET part of the command. Obviously, there are cases when this general scheme does not apply and some part of this message trio is not defined. For consistency, in the description of the messages this SET->REQUEST->GET scheme will be used throughout.

Note that, as the scheme suggests, this is a master-slave type of system, so sub-modules never send SET and REQUEST messages to the host and GET messages are always sent to the host as a destination.

COMMAND REFERENCE

The sections below detail the messages used for motor controller operations. Note that the source and destination fields are not filled in as these vary depending on the originator and target of the message.

IDENTIFY

Sent to instruct hardware unit to identify itself (by flashing its front panel LEDs).

(#define MGMSG_MOD_IDENTIFY 0x0223)

Tx Message bytes:

0x0223	0x00	0x00	Source	Dest

No response.

HARDWARE INFORMATION

Sent to request hardware information from the controller.

SET: N/A
REQUEST:

(#define MGMSG_HW_REQ_INFO 0x0005)

Tx Message bytes:

0x0005	0x00	0x00	Source	Dest

Response as follows:-

GET:

(#define MGMSG_HW_GET_INFO 0x0006)

Rx Message bytes:

0x0006	0x0054	Source | 0x80	Dest

Followed by an 84-byte data packet:-

[Equivalent C/C++ structure]

```
// Rx
typedef struct _HWINFO
{
  DWORD dwSerialNum;   // Unique 8 digit serial number.
  char szModelNum[8]; // Alphanumeric model number.
  WORD wHWType;        // Hardware type ident (see #defines below).
  DWORD dwSoftwareVersion;// Software version encoded as follows ((BYTE)Major << 16) +
((BYTE)Interim << 8) + (BYTE)Minor.
  char szNotes[64];    // Arbitrary alphanumeric info string.
  WORD wNumChannels;   // Number of channels of operation

} HWINFO, *LPHWINFO;

// Hardware type idents.
#define P_HW_RACK_USB         0x01
```

```
#define P_HW_RACK_ETHERNET      0x02
#define P_HW_MOTOR              0x10
#define P_HW_PIEZO              0x12
#define P_HW_POWERMETER         0x13
#define P_HW_NANOTRAK           0x14
#define P_HW_CTRL6MOTOR3CHAN    0x20
#define P_HW_CTRL6PIEZO3CHAN    0x22
```

STATUS UPDATES

Sent to start status updates from the embedded controller.

SET:

(#define MGMSG_HW_START_UPDATEMSGS 0x0011)

Tx Message bytes:	0x11	Update Rate (0x02 for ODC001)	0x00	Source	Dest

REQUEST: N/A

Status update messages are received with the following format:-

GET:

>>>FOR STEPPER CONTROLLERS<<<

(#define MGMSG_MOT_GET_STATUSUPDATE 0x0481)

Rx Message bytes:	0x0481	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet:-

[Equivalent C/C++ structure]

```
// Rx
typedef struct _MOTSTATUS
{
  WORD wChannel;       // Channel ident.
  long lPosition;      // Position in microsteps.
  long lEncCount;      // Encoder count (for encoded stages).
  DWORD dwStatusBits;  // Status bits (see #defines below).

} MOTSTATUS, *LPMOTSTATUS;

// Motor specific status bit locations.
#define P_MOT_SB_CWHARDLIMIT          0x00000001     // CW hardware limit switch (0 -
no contact, 1 - contact).
#define P_MOT_SB_CWHARDLIMIT_MASK     0xFFFFFFFE
#define P_MOT_SB_CCWHARDLIMIT         0x00000002     // CCW hardware limit switch (0 -
no contact, 1 - contact).
#define P_MOT_SB_CCWHARDLIMIT_MASK    0xFFFFFFFD
#define P_MOT_SB_CWSOFTLIMIT          0x00000004     // CW software limit switch (0 -
no contact, 1 - contact).
#define P_MOT_SB_CWSOFTLIMIT_MASK     0xFFFFFFFB
#define P_MOT_SB_CCWSOFTLIMIT         0x00000008     // CCW software limit switch (0 -
no contact, 1 - contact).
#define P_MOT_SB_CCWSOFTLIMIT_MASK    0xFFFFFFF7

#define P_MOT_SB_INMOTIONCW           0x00000010     // Moving clockwise (1 - moving, 0
- stationary).
#define P_MOT_SB_INMOTIONCW_MASK      0xFFFFFFEF
```

```
#define P_MOT_SB_INMOTIONCCW          0x00000020    // Moving counterclockwise (1 - 
moving, 0 - stationary).
#define P_MOT_SB_INMOTIONCCW_MASK     0xFFFFFFDF
#define P_MOT_SB_JOGGINGCW            0x00000040    // Jogging clockwise (1 - moving, 
0 - stationary).
#define P_MOT_SB_JOGGINGCW_MASK       0xFFFFFFBF
#define P_MOT_SB_JOGGINGCCW           0x00000080    // Jogging counterclockwise (1 - 
moving, 0 - stationary).
#define P_MOT_SB_JOGGINGCCW_MASK      0xFFFFFF7F

#define P_MOT_SB_CONNECTED            0x00000100    // Motor connected (1 - connected, 
0 - not connected).
#define P_MOT_SB_CONNECTED_MASK       0xFFFFFEFF
#define P_MOT_SB_HOMING               0x00000200    // Motor homing (1 - homing, 0 - 
not homing).
#define P_MOT_SB_HOMING_MASK          0xFFFFFDFF
#define P_MOT_SB_HOMED                0x00000400    // Motor homed (1 - homed, 0 - not 
homed).
#define P_MOT_SB_HOMED_MASK           0xFFFFFBFF

#define P_MOT_SB_INTERLOCK            0x00001000    // Interlock state (1 - enabled, 0 
- disabled)
#define P_MOT_SB_INTERLOCKMASK        0xFFFFEFFF
```

>>>FOR DC SERVO CONTROLLERS – E.G. ODC001<<<

(#define MGMSG_MOT_GET_DCSTATUSUPDATE 0x0491)

Rx Message bytes:	0x0491	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet:-

[Equivalent C/C++ structure]

```
// Rx
typedef struct _DCMOTSTATUS
{
  WORD wChannel;        // Channel ident (see #defines below).
  long lPosition;       // Position in encoder counts.
  WORD wReserved1;
  WORD wReserved2;
  DWORD dwStatusBits;  // Status bits (see #defines above).

} DCMOTSTATUS, *LPDCMOTSTATUS;

// Channel idents.
#define P_MOD_CHAN1          0x01    // channel 1
#define P_MOD_CHAN2          0x02    // channel 2
#define P_MOD_CHAN3          0x04    // channel 3
#define P_MOD_CHAN4          0x08    // channel 4
```

MGMSG_HW_STOP_UPDATEMSGS

Sent to stop status updates from the controller

SET:

(#define MGMSG_HW_STOP_UPDATEMSGS 0x0012)

Tx Message bytes:	0x0012	0x0000	Source	Dest

No response.

REQUEST: N/A
GET: N/A

REQUEST STATUS UPDATE

Used to request a status update for the specified motor channel. This request can be used instead of enabling regular updates as described above. Not implemented in all controllers.

>>>FOR STEPPER CONTROLLERS<<<

REQUEST:

(#define MGMSG_MOT_REQ_STATUSUPDATE 0x0480)

Tx Message bytes:

0x0480	Chan Ident	0x00	Source	Dest

GET:

See above details on:-

MGMSG_MOT_GET_STATUSUPDATE

>>>FOR DC SERVO CONTROLLERS – E.G. ODC001<<<

REQUEST:

(#define MGMSG_MOT_REQ_DCSTATUSUPDATE 0x0490)

Tx Message bytes:

0x0490	Chan Ident	0x00	Source	Dest

GET:

See above details on:-

MGMSG_MOT_GET_DCSTATUSUPDATE

LIMIT SWITCH PARAMS

Used to set the limit switch characteristics of a controller, consistent with which stage the controller is configured to drive.

SET:

(#define MGMSG_MOT_SET_LIMSWITCHPARAMS 0x0423)

Tx Message bytes:	0x0423	0x0010	Source | 0x80	Dest

Followed by a 16-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTLIMSWITCHPARAMS
{
  WORD wChannel;      // Channel ident (see #defines earlier).
  WORD wCWHardLimit;  // Clockwise hardware limit switch operation (see #defines
below).
  WORD wCCWHardLimit; // Similarly for counter clockwise hardware limit switch.
  long lReserved;
  long lReserved;
  WORD wReserved;

} MOTLIMSWITCHPARAMS, *LPMOTLIMSWITCHPARAMS;

// Hardware limit switch mode definitions.
#define P_MOT_SWITCHIGNORE    0x01    // Ignore switch or switch not present.
#define P_MOT_SWITCHMAKES     0x02    // Switch makes on contact.
#define P_MOT_SWITCHBREAKS    0x03    // Switch breaks on contact.
// KAD 06-12-05 - Set upper bit to swap CW and CCW limit switches in code.
// Both wCWHardLimit and wCCWHardLimit structure members will have the upper bit
// set when limit switches have been physically swapped.
#define P_MOT_SWITCHSWAPPED   0x80    // bitwise OR'd with one of the settings above...
```

REQUEST:

(#define MGMSG_MOT_REQ_LIMSWITCHPARAMS 0x0424)

Tx Message bytes:	0x0424	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_LIMSWITCHPARAMS 0x0425)

Rx Message bytes:	0x0425	0x0010	Source | 0x80	Dest

Followed by a 16-byte data packet (see structure for Set message above):-

PHASE POWER PARAMS (STEPPER CONTROLLERS ONLY)

Used to set the stationary and moving phase powers for the specified motor channel. This is set as a percentage of full power in steps of 10% or 20% depending on the controller type (does NOT apply to DC Servo controllers such as the ODC001). The normal default settings are 20% of maximum power for motor stationary and 100% for motor moving. The settings are generally specific to a particular payload application.

SET:

(#define MGMSG_MOT_SET_POWERPARAMS 0x0426)

Tx Message bytes:

0x0426	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTPOWERPARAMS
{
  WORD wChannel;      // Channel ident (see #defines earlier).
  WORD wRestFactor;   // Phase power at rest (1 to 100%).
  WORD wMoveFactor;   // Phase power while moving (1 to 100%).

} MOTPOWERPARAMS, *LPMOTPOWERPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_POWERPARAMS 0x0427)

Tx Message bytes:

0x0427	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_POWERPARAMS 0x0428)

Rx Message bytes:

0x0428	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet (see structure for Set message above):-

VELOCITY PROFILE PARAMS

Used to set the trapezoidal velocity parameters for the specified motor channel, in position steps/sec for velocity or position steps/sec/sec for acceleration.
For stepper controllers the position steps are micro-steps and for DC servo controllers encoder counts (see notes below).

SET:

(#define MGMSG_MOT_SET_VELPARAMS 0x0413)

Tx Message bytes:

0x0413	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTVELPARAMS
{
  WORD wChannel;      // Channel ident (see #defines earlier).
  long lMinVel;       // Minimum (start) velocity in pos. steps/sec (for ODC001 see
notes below).
```

```
  long lAccn;          // Acceleration in position pos. steps/sec*sec (for ODC001 see
notes below).
  long lMaxVel;        // Maximum (final) velocity in pos. steps/sec (for ODC001 see
notes below).

} MOTVELPARAMS, *LPMOTVELPARAMS;
```

Notes:-
1. At this time lMinVel is always set to zero.

2. For LM629 based controllers (e.g. ODC001 Cube Controller) the velocity = encoder counts/sample period (similarly for acceleration). Refer to National Semiconductor LM628/LM629 data sheet for details on setting velocity and acceleration parameters.

REQUEST:

(#define MGMSG_MOT_REQ_VELPARAMS 0x0414)

Tx Message bytes:

0x0414	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_VELPARAMS 0x0415)

Rx Message bytes:

0x0415	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet (see structure for Set message above):-

JOGGING PARAMS

Used to set the velocity jog parameters for the specified motor channel, in position steps/sec for velocity or position steps/sec/sec for acceleration.
For stepper controllers the position steps are micro-steps and for DC servo controllers encoder counts (see notes below).

SET:

(#define MGMSG_MOT_SET_JOGPARAMS 0x0416)

Tx Message bytes:

0x0416	0x0016	Source | 0x80	Dest

Followed by a 22-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTJOGPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  WORD wJogMode;       // Jogging mode (see #defines below).
  long lJogStepSize;   // Jog step size in pos. steps (single step mode)
```

```
  long lJogMinVel;     // Minimum (start) velocity in pos. steps/sec (for ODC001 see 
notes below).
  long lJogAccn;       // Acceleration in position pos. steps/sec*sec (for ODC001 see 
notes below).
  long lJogMaxVel;     // Maximum (final) velocity in pos. steps/sec (for ODC001 see 
notes below).
  WORD wJogStopMode;  // Stop mode. See #defines for MGMSG_MOT_MOVE_STOP.

} MOTJOGPARAMS, *LPMOTJOGPARAMS;

// Jog mode definitions.
#define P_MOT_JOGCONTINUOUS   0x01    // Move continuously while jog signal present.
#define P_MOT_JOGSINGLESTEP   0x02    // Move one jog step (lJogStepSize) when jog 
signal detected.
```

Notes:-
1. At this time lMinVel is always set to zero.

2. For LM629 based controllers (e.g. ODC001 Cube Controller) the velocity = encoder counts/sample period (similarly for acceleration). Refer to National Semiconductor LM628/LM629 data sheet for details on setting velocity and acceleration parameters.

REQUEST:

(#define MGMSG_MOT_REQ_JOGPARAMS 0x0417)

Tx Message bytes:

0x0417	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_JOGPARAMS 0x0418)

Rx Message bytes:

0x0418	0x0016	Source | 0x80	Dest

Followed by a 22-byte data packet (see structure for Set message above):-

GENERAL MOVE PARAMS

Used to set the general move parameters for the specified motor channel. At this time this refers specifically to the backlash settings.

SET:

(#define MGMSG_MOT_SET_GENMOVEPARAMS 0x043A)

Tx Message bytes:

0x043A	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTGENMOVEPARAMS
{
```

```
  WORD wChannel;       // Channel ident (see #defines earlier).
  long lBLashDistance; // Backlash correction distance (in pos. steps, 0 = no backlash
correction).

} MOTGENMOVEPARAMS, *LPMOTGENMOVEPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_GENMOVEPARAMS 0x043B)

Tx Message bytes:

0x043B	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_GENMOVEPARAMS 0x043C)

Rx Message bytes:

0x043C	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet (see structure for Set message above):-

MOVE JOG

Sent to start a jog move on the specified motor channel.

(#define MGMSG_MOT_MOVE_JOG 0x046A)

Tx Message bytes:

0x046A	Chan Ident	Direction	Source	Dest

```
// Jog direction definitions.
#define P_MOT_JOG_CW          0x01    // Clockwise.
#define P_MOT_JOG_CCW         0x02    // Counterclockwise.
```

No response on initial message, but upon completion of jog move controller responds as follows:-

(#define MGMSG_MOT_MOVE_COMPLETED 0x0464)

Rx Message bytes:

0x0464	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet described by the same status structures (i.e. MOTSTATUS and MOTDCSTATUS) described in the earlier STATUS UPDATES section.

Note for 2 channel units there will be 2 status structures (one for each channel) appended to the message header i.e. 28 data bytes total.

HOMING PARAMS

Used to set the home parameters for the specified motor channel. These parameters are stage specific.

SET:

(#define MGMSG_MOT_SET_HOMEPARAMS 0x0440)

Tx Message bytes:

0x0440	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTHOMEPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  WORD wDirection;     // Direction of home (see #defines below).
  WORD wLimSwitch;     // Limit switch for zero reference (see #defines below).
  long lHomeVelocity; // Homing velocity in pos. steps/sec (for ODC001 see notes
below).
  long lOffsetDist;    // Origin offset distance (in pos. steps) from limit switch.

} MOTHOMEPARAMS, *LPMOTHOMEPARAMS;

// wDirection definitions.
#define P_MOT_CW              0x01    // Clockwise.
#define P_MOT_CCW             0x02    // Counterclockwise.

// wLimSwitch definitions.
#define P_MOT_CWHARD          0x01    // Clockwise hardware switch.
#define P_MOT_CCWHARD         0x04    // Counterclockwise hardware switch.
```

REQUEST:

(#define MGMSG_MOT_REQ_HOMEPARAMS 0x0441)

Tx Message bytes:

0x0441	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_HOMEPARAMS 0x0442)

Rx Message bytes:

0x0442	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet (see structure for Set message above):-

Notes:-
1. For LM629 based controllers (e.g. ODC001 Cube Controller) the velocity = encoder counts/sample period (similarly for acceleration). Refer to National Semiconductor LM628/LM629 data sheet for details on setting velocity and acceleration parameters.

MOVE HOME

Sent to start a home move sequence on the specified motor channel (in accordance with the home params above).

(#define MGMSG_MOT_MOVE_HOME 0x0443)

Tx Message bytes:

0x0443	Chan Ident	0x00	Source	Dest

No response on initial message, but upon completion of home sequence controller responds as follows:-

(#define MGMSG_MOT_MOVE_HOMED 0x0444)

Rx Message bytes:

0x0444	Chan Ident	0x00	Source	Dest

MOVE RELATIVE PARAMS

Used to set the relative move parameters for the specified motor channel. The only significant parameter at this time is the relative move distance itself. This gets stored by the controller and is used the next time a relative move is initiated.

SET:

(#define MGMSG_MOT_SET_MOVERELPARAMS 0x0445)

Tx Message bytes:

0x0445	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTRELMOVEPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  long lRelDistance;   // Signed relative distance to move (in pos. steps).

} MOTRELMOVEPARAMS, *LPMOTRELMOVEPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_MOVERELPARAMS 0x0446)

Tx Message bytes:

0x0446	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_MOVERELPARAMS 0x0447)

Rx Message bytes:

0x0447	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet (see structure for Set message above):-

MOVE RELATIVE

Used to start a relative move on the specified motor channel (using the relative move distance param above).

(#define MGMSG_MOT_MOVE_RELATIVE 0x0448)

Tx Message bytes:

0x0448	Chan Ident	0x00	Source	Dest

No response on initial message, but upon completion of relative move the controller responds as follows:-

(#define MGMSG_MOT_MOVE_COMPLETED 0x0464)

Rx Message bytes:

0x0464	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet described by the same status structures (i.e. MOTSTATUS and MOTDCSTATUS) described in the earlier STATUS UPDATES section.

Note for 2 channel units there will be 2 status structures (one for each channel) appended to the message header i.e. 28 data bytes total.

MOVE ABSOLUTE PARAMS

Used to set the absolute move parameters for the specified motor channel. The only significant parameter at this time is the absolute move position itself. This gets stored by the controller and is used the next time an absolute move is initiated.

SET:

(#define MGMSG_MOT_SET_MOVEABSPARAMS 0x0450)

Tx Message bytes:

0x0450	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTABSMOVEPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  long lAbsPosition;   // Absolute position to move (in pos. steps).

} MOTABSMOVEPARAMS, *LPMOTABSMOVEPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_MOVEABSPARAMS 0x0451)

Tx Message bytes:

0x0451	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_MOVEABSPARAMS 0x0452)

Rx Message bytes:

0x0452	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet (see structure for Set message above):-

<u>MOVE ABSOLUTE</u>

Used to start an absolute move on the specified motor channel (using the absolute move position param above).

(#define MGMSG_MOT_MOVE_ABSOLUTE 0x0453)

Tx Message bytes:

0x0453	Chan Ident	0x00	Source	Dest

No response on initial message, but upon completion of absolute move the controller responds as follows:-

(#define MGMSG_MOT_MOVE_COMPLETED 0x0464)

Rx Message bytes:

0x0464	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet described by the same status structures (i.e. MOTSTATUS and MOTDCSTATUS) described in the earlier STATUS UPDATES section.

Note for 2 channel units there will be 2 status structures (one for each channel) appended to the message header i.e. 28 data bytes total.

<u>MOVE AT VELOCITY</u>

Sent to start a move at constant velocity (as specified by VELOCITY PARAMS earlier) on the specified motor channel.

(#define MGMSG_MOT_MOVE_VELOCITY 0x0457)

Tx Message bytes:	0x0457	Chan Ident	Direction	Source	Dest

```
// Move direction definitions.
#define P_MOT_CW        0x01    // Clockwise.
#define P_MOT_CCW       0x02    // Counterclockwise.
```

No response on initial message, and move does not complete until an MGMSG_MOT_MOVE_STOP message is issued (see below).

STOP MOVE

Sent to stop any type of motor move (relative, absolute, homing or move at velocity) on the specified motor channel.

(#define MGMSG_MOT_MOVE_STOP 0x0465)

Tx Message bytes:	0x0465	Chan Ident	Stop Mode	Source	Dest

```
// Stop mode definitions.
#define P_MOT_STOP_IMMEDIATE   0x01    // stops the move immediately
#define P_MOT_STOP_PROFILED    0x02    // stops the move in a controlled deceleration.
```

The controller stops the motor moving and responds as follows:-

(#define MGMSG_MOT_MOVE_STOPPED 0x0466)

Rx Message bytes:	0x0466	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet described by the same status structures (i.e. MOTSTATUS and MOTDCSTATUS) described in the earlier STATUS UPDATES section.

Note for 2 channel units there will be 2 status structures (one for each channel) appended to the message header i.e. 28 data bytes total.

POSITION COUNTER

Used to set the 'live' position count in the controller (not implemented in all controller variants).

SET:

(#define MGMSG_MOT_SET_POSCOUNTER 0x0410)

Tx Message bytes:	0x0410	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTPOSPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  long lPosition;      // Position counter value in microsteps (stepper units) or
encoder counts (DC units).

} MOTPOSPARAMS, *LPMOTPOSPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_POSCOUNTER 0x0411)

Tx Message bytes:

0x0411	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_POSCOUNTER 0x0412)

Rx Message bytes:

0x0412	0x0006	Source | 0x80	Dest

Followed by a 6-byte data packet (see structure for Set message above):-

<u>DC SERVO PID PARAMS</u>

Used to set the PID loop parameters of the DC servo control loop. These settings apply to LM628/629 based servo controllers (only ODC001 at this time).

Refer to data sheet for National Semiconductor LM628/LM629 for details on setting these PID related parameters.

SET:

(#define MGMSG_MOT_SET_DCPIDPARAMS 0x04A0)

Tx Message bytes:

0x04A0	0x0014	Source | 0x80	Dest

Followed by a 20 byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _DCMOTPIDPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  long lProportional;  // Proportional constant - LM629 kp param [0x0000 to 0x7FFF]).
  long lIntegral;      // Integral constant - LM629 ki param [0x0000 to 0x7FFF]).
  long lDifferential;  // Differential constant - LM629 kd param [0x0000 to 0x7FFF]).
  long lIntegralLimit; // Summed integration maximum limit - LM629 il param [0x0000 to
0x7FFF]).
  WORD wFilterControl; // Specifies the filter parameters to update - LM629 = 'filter
control' word.

} DCMOTPIDPARAMS, *LPDCMOTPIDPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_DCPIDPARAMS 0x04A1)

Tx Message bytes:

0x04A1	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_DCPIDPARAMS 0x04A2)

Rx Message bytes:

0x04A2	0x0014	Source | 0x80	Dest

Followed by a 20-byte data packet (see structure for Set message above):-

POTENTIOMETER PARAMS

Sets the potentiometer control params for ODC001 and OST001 Cube Controllers. These parameters determine the motor speeds achieved when using the sprung potentiometer on the front panel of the Cube Controller.

SET:

(#define MGMSG_MOT_SET_POTPARAMS 0x04B0)

Tx Message bytes:

0x04B0	0x0016	Source | 0x80	Dest

Followed by a 22 byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTPOTPARAMS
{
  WORD wChannel; // Channel ident.
  WORD wZeroWnd; // Pot ADC value (e.g. range 0 - 128 for ODC001 & OST001)
  long lVel1;  // In pos. steps/sec (for LM629 based controllers see note below).
  WORD wWnd1;  // Pot ADC value (e.g. range wZeroWnd to 128) – NOT USED BY OST001
  long lVel2;  // In pos. steps/sec (for LM629 based controllers see note below) – NOT
USED BY OST001
  WORD wWnd2;  // Pot ADC value (e.g. range wWnd1 to 128) - NOT USED BY OST001
  long lVel3;  // In pos. steps/sec (for LM629 based controllers see note below)– NOT
USED BY OST001
   WORD wWnd3; // Pot ADC value (e.g. range wWnd2 to 128) – NOT USED BY OST001
  long lVel4;  // In pos. steps/sec (for LM629 based controllers see note below) – NOT
USED BY OST001.

} MOTPOTPARAMS, *LPMOTPOTPARAMS;
```

REQUEST:

(#define MGMSG_MOT_REQ_POTPARAMS 0x04B1)

Tx Message bytes:

0x04B1	Chan	0x00	Source	Dest

	Ident			

GET:

(#define MGMSG_MOT_GET_POTPARAMS 0x04B2)

Rx Message bytes:	0x04B2	0x0016	Source | 0x80	Dest

Followed by a 22-byte data packet (see structure for Set message above):-

Notes:-
1. For LM629 based controllers (e.g. ODC001 Cube Controller) the velocity = encoder counts/sample period. Refer to National Semiconductor LM628/LM629 data sheet for details on setting velocity and acceleration parameters.

AUDIO VISUAL MODES

Used to set the Audio Visual settings for the ODC001 and OST001 Cube Controllers. At this time only the operation of the Cube front panel LED is covered. Future variants of the Cube may include a buzzer for audible warnings.

SET:

(#define MGMSG_MOT_SET_AVMODES 0x04B3)

Tx Message bytes:	0x04B3	0x0004	Source | 0x80	Dest

Followed by a 4 byte data packet:-

[Equivalent C/C++ structure]
```
// Tx
typedef struct _MOTAVPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  WORD wModeBits;      // Mode enable bits (see #defines below).

} MOTAVPARAMS, *LPMOTAVPARAMS;

// LED mode enable bit definitions – Bitwise OR to combine modes.
#define P_MOT_LED_IDENT       0x0001  // Flash LED on ident (1 - enable, 0 - disable).
#define P_MOT_LED_LIMITSWITCH 0x0002  // Flash LED on hardware limit switch (1 -
enable, 0 - disable).
#define P_MOT_LED_BUTTONMODECHANGE    0x0004  // Flash LED when button mode is changed
(1 - enable, 0 - disable).
#define P_MOT_LED_MOVING      0x0008  // Illuminate LED when motor is moving (1 –
enable, 0 – disable).
```

REQUEST:

(#define MGMSG_MOT_REQ_AVMODES 0x04B4)

Tx Message bytes:	0x04B4	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_AVMODES 0x04B5)

Rx Message bytes:	0x04B5	0x0004	Source | 0x80	Dest

Followed by a 4-byte data packet (see structure for Set message above):-

PANEL BUTTON MODES

Used to set the front panel button settings for the ODC001 and OST001 Cube Controllers.

SET:

(#define MGMSG_MOT_SET_BUTTONPARAMS 0x04B6)

Tx Message bytes:	0x04B6	0x000E	Source | 0x80	Dest

Followed by a 14 byte data packet:-

[Equivalent C/C++ structure]

```
// Tx
typedef struct _MOTBUTTONPARAMS
{
  WORD wChannel;       // Channel ident (see #defines earlier).
  WORD wMode;          // Button operating Mode (see #defines below).
  long lPosition1;     // Preset position for button 1 in pos. steps.
  long lPosition2;     // Preset position for button 2 in pos. steps.
  WORD wTimeOut1;      // Button press timeout in ms (used when holding buttons down
for home or teaching positions).
  WORD wTimeOut2;      // Reserved for future use.

} MOTBUTTONPARAMS, *LPMOTBUTTONPARAMS;

// Button operating modes.
#define P_MOT_BUTTON_JOG      0x01    // Buttons used for jogging (using the jog
settings described earlier).
#define P_MOT_BUTTON_POSITION 0x02    // Buttons used for preset positions.
```

REQUEST:

(#define MGMSG_MOT_REQ_BUTTONPARAMS 0x04B7)

Tx Message bytes:	0x04B7	Chan Ident	0x00	Source	Dest

GET:

(#define MGMSG_MOT_GET_BUTTONPARAMS 0x04B8)

Rx Message bytes:	0x04B8	0x000E	Source | 0x80	Dest

Followed by a 14-byte data packet (see structure for Set message above).