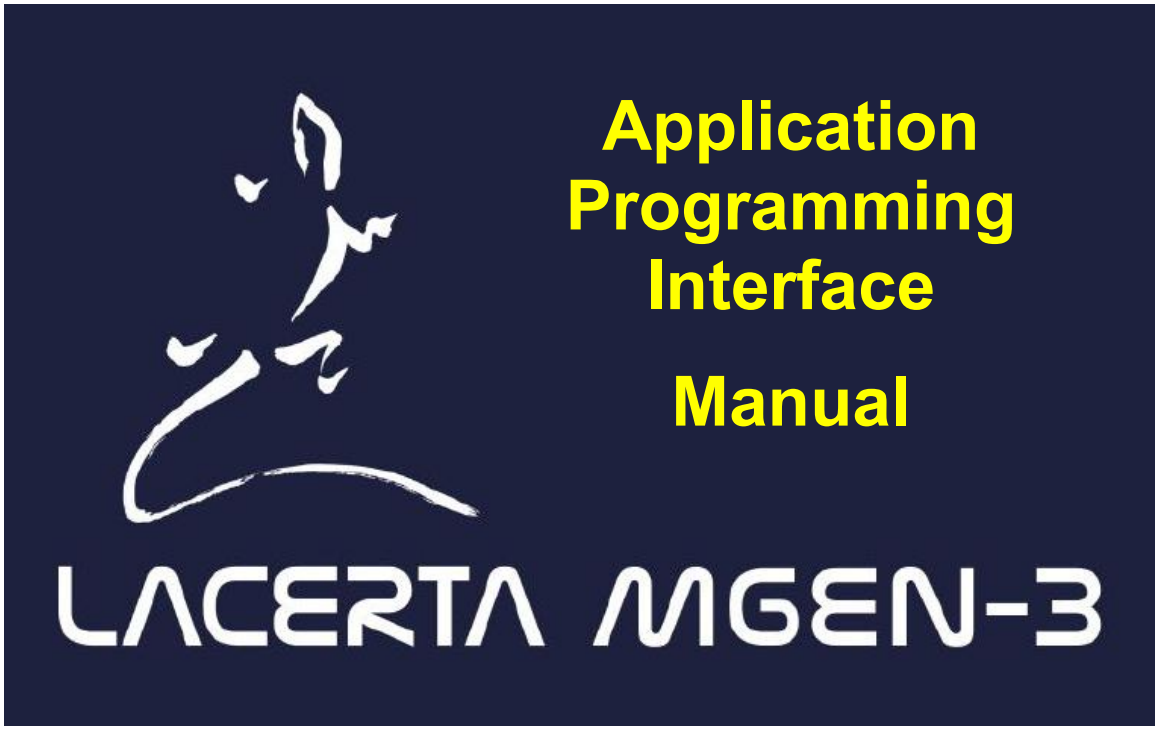


Version 1.04

Table of Contents

1. Release types of API

MGen-3 API's version number always indicate the HC Firmware's version that it is able to control. Backward compatibility is mostly supported (using a newer API than the Fw.), the protocol / API is intent on handling the opposite case too.

The API is available as the following:

- DLL for Windows, 32-bit application
- DLL for Windows, 64-bit application
- Source code for Windows

The API packs are released as a ZIP file. On the main directory of it you can find the header file(s) for the DLL releases you need to include. In the `x86\` and `x64\` directories there are the libraries and executable DLLs you need to use for compilation and execution, each for a Win32 and a Win64 executable.

If you use the DLL release, you either can do it from a C or a C++ program. While the C++ interface uses a class for the instances of MGen-3, there are C functions that do the same, of course you have to create and destroy the handles for the instances instead of having it done by the constructor/destructor of the C++ class.

Source code form is for the ones who need to implement the API in an other environment than Windows. In order to have it compiled well, some files must be extended with the required info/data. You need to provide the FTDI D2XX driver's interface on your system as the source code uses it.

The different type of APIs share the same interface function names. The differences between them are detailed in the below sections.

1.1. API as a DLL for Windows

The most convenient method for using the API on a Windows OS is using one of the DLLs. By this your application should not be modified when a bugfix/patch/new release is available for the API.

The API uses the D2XX driver for FTDI ICs so the driver must be installed on the target system as well. (Available at https://www.ftdichip.com/Drivers/D2XX.htm)

There is always a separate DLL for Win32 and Win64 applications with separate library files. You should choose it by your application's type. In this document there is no differentiation between them.

A DLL release consists of the following parts:

- The executable file: `MG3lib.dll`

  This file must be in your application's working path or located in Windows system's search paths.

- Library file: `MG3lib.lib`

  This is the library file for Microsoft C/C++ you need to include when compiling your application.

- Header file(s): `MG3lib.h`

  You need to include this header into your C or C++ source code to access the API. The other header files next to this file will be included automatically.

1.2. API as a source code

The source code API (only C++) is somewhat different to the DLL version. All the required source and header files are located in the `source\` directory of the API pack. The headers in the main directory of the API pack must not be used!

When the source code API compiles, you will have a class named `cMG3`, which is similar to the `CMG3lib` class of C++ DLL API but with all the sub-functions and low-level implementations. The usage is the same as for the C++ DLL API class. (See 2.2.1).

There are some functions available in the source code that are not exact part of the API but may be useful if someone wants to implement special features. For example the "external sky emulation" feature is only available from the source code, though not yet supported by detailed information. (It is still part of the development features.)

The source code in its release form compiles for Windows environment. It uses the FTDI's D2XX driver API so you must provide it for your application. (Download at https://www.ftdichip.com/Drivers/D2XX.htm) These include the `Windows.h` header file.

Porting the source code to an other system is up to the developer. (Not tested yet on any other system.) Requirements and help notes on this:

- The system has to have an available sw. interface of FTDI D2XX driver
- In `mg3_acc_basictypes.h` you must define some types
- Compiler or OS-specific modifications may be made as well

2. About the interfaces

There is a function naming convention for MGen-3 API: function names start with `MG3_…` that do not communicate with the MGen-3 HandController (doing other management) or are special ones. Function names start with `MG3cmd_…` that do at least one messaging cycle with it. These functions may take longer to execute due to many reasons but they have timeouts. Details are given for each function in this document.

Only some specific functions may be named uniquely that are not present for all API forms.

2.1.C++ interface

For a C++ program using the DLL, a class named `CMG3lib` is provided, which is meant to handle one instance of MGen-3. If you define such object no communication is done or device is opened but the required data structures are allocated and initialized.

After creating the object, you need to call `MG3_Open()`. This function will try connect to the HandController and retrieve data like protocol version and command list. If succeeded, other API functions can be used.

Note that the functions that need communication with the device will call `MG3_Open()` if there is no open connection yet. Consider handling the opening error codes of `MG3_Open()` for the other API functions or call the opening function preceding the other to make sure the device is ready for use.

Note: currently the device selection is automatic, the first MGen-3 HC device will be opened that is available on the system.

`MG3_Close()` of your objects must be called before closing your application. Though, the destructor also does the required jobs. When you want to open a connection to a other (or unknown) device, you have to call `MG3_Close()` before calling `MG3_Open()` again.

2.1.1. Example

```
#include "MG3lib.h"

int main(int argc, char* argv[])
{
    CMG3lib hc;                 // the object is constructed for a MGen-3 instance

    int res = hc.MG3_Open();    // connecting to the HC   (not required)
    if(MG3_OK == res)
    {
        // you can use the API here… e.g.:
        res = hc.MG3_ESC_Control(200);
        // …
        hc.MG3_Close();         // close the connection   (not required)
    }
    return 0;
    // the object is destructed, it would close the connection automatically
}
```

2.2.C interface

For a C program the same functions are available but these are global functions instead of class members. The difference is about the constructing and destructing of an object that refers to a connection with a MGen-3 HC.

Instead of the constructor there is the function `Create_MG3_handle()` that return a `MG3HANDLE` type value. You need to store this value and provide for the C API fuctions (always as the first parameter) to refer to the active instances. This 1st extra parameter is not shown during the introduction of the functions in later sections.

You need to use `MG3_Open()` and `MG3_Close()` the same way as described in the previous section (0).

Instead of the destructor you need to call `Release_MG3_handle()` to free allocated resources from the system.

2.2.1. Example

```
#include "MG3lib.h"

int main(int argc, char* argv[])
{
    int res;
    MG3HANDLE hc;               // the object will be referred by this handle

    hc = Create_MG3_handle();   // create a new handle
    if(NULL == hc) return 1;    // check for successful creation of it

    res = MG3_Open(hc);         // connecting to the HC    (not required)
    if(MG3_OK != res) return 2;

    // you can use the API here… e.g.:
    res = MG3_ESC_Control(hc, 200);
    // …

    MG3_Close(hc);              // close the connection    (not required)
    Release_MG3_handle(hc);     // destruct the handle
    return 0;
}
```

3. Common info

3.1. Return values

The return value of all common interface functions is type of `int`. These mean error codes (unless otherwise noted) and therefore the value of zero means “no error”. There may be special non-zero return value codes for some functions that mean no error as well but indicate some specific state of the system.

Error codes are defined in `mg3_retval.h`. `MG3_OK` is defined for no error.

Human-readable strings (ANSI) can be requested by the function `MG3_FormatResult()` for an error code. For example a communication fault can be reported by the FTDI driver at anytime, this error must not be related directly to an API function but a low level failure. But displaying it in your application may help to understand the exact fault of the API call, without handling all the possible error codes for all your API calls.

For C DLL API this function is straightforward. For C++ DLL and source API this function is static for the class, so you should call it as `CMG3lib::MG3_FormatResult()` or `cMG3::MG3_FormatResult()`.

3.2. Implemented for

At the detailed descriptions of the API functions there will be such a text box indicating which running modes of the HC is the function is implemented:

BOOT + Firmware

The functions that do not communicate with the HC actively are not marked, these are available for any modes.

If you call an API function at an unsupported running mode of the HC `MG3_IF_COMMAND_UNSUPPORTED` is returned, or if the API function checks for the required mode, the return value is straightforward, for example `MG3_BOOT_MODE_REQUIRED`.

3.3. Handling different API + Fw. versions

An API being used and the loaded DLL's version should be equal. Though your application might be able to load a DLL with different version too, these cases are "not supported", problems may occur.

It is more hard to ensure that a Firmware running on the HC is new enough for an API. These cases are handled, API will return the proper error code.

For the API functions that uses a structure to specify or receive data, you always have to set the `strSize` member of the structure before the call. Always use the known size of the structure (`sizeof(…)`). API will use this info not to query and write data that would overflow your structure's memory space and will also disable using some parameter accidentally that you didn't want to specify. (Only full structure sizes are allowed that were released by an API version.)

If you need flawless work with the MGen-3 HC device, ask the user to allow the update of the Firmware if it was older than the API. You can use the API functions to execute the update processes from your application. (Of course the update may need a valid user key file for a given Fw. version but that's the user's responsibility to provide it.)

4. Management functions

The following sections will describe all the available API functions and their detailed usage. It is assumed that the object/handle of a MGen-3 device is already created without error.

Management function types:

- Do something through the FTDI IC without communicating the device's Firmware
- Communication channel: opening (initialize) or closing it

Common return values will not be listed for each function, as their meaning is straightforward. The list below summarizes these:

`MG3_OK`	Success, the function has run without error.
`MG3_FT230_ERROR`	Some low level error occured when using FTDI driver for the communication. May be due to an invalid state of the HC, for example if it was powered down (timeout error).
`MG3_NO_FTDI_HANDLE`	The object / handle has not been created well so comm. is impossible. (The object must tried to be re-created to recover from this error.)
`MG3_INVALID_HANDLE`	For C API, if an invalid handle value is passed to a function, this is returned.
`MG3_FAILED_TO_OPEN_DEVICE`	The FTDI driver couldn't open its device. Probably there is no proper MGen-3 HC device available.

4.1.MG3_API_version

```
int MG3_API_version();
```

Returns the version number of the loaded DLL API (executable).

Being a special function the defined return value codes do not apply here.

Remarks

This DLL version equals the Firmware version that the DLL is fully able to control. Older Firmwares are mostly supported and the functions that does not need some new feature of a newer Firmware will also run ok.

Firmware number is a hexadecimal number in the format of 0xAABB, where AA is the main and BB is the sub version (for a Firmware ver. of AA.BB.C - C is only a patch number, AA.BB Fw.s are functionally equivalent).

Example: for the DLL released for the Firmware ver. 1.04.0 the return value is 0x0104 (decimal 260).

4.2.MG3_get_FT_serial, MG3_get_FT_description

```
int MG3_get_FT_serial(const char** pstr);
int MG3_get_FT_description(const char** pstr);
```

To query some info of the FTDI IC used for communication. The serial number and description are provided.

Parameters

`pstr`	A pointer to a variable that will hold the pointer to the zero-terminated C string after return. For an error the variable will point to an empty string.

Return value

Default.

Remarks

These info of the FTDI IC is externally modifiable with a tool, do not use it for serious identification purposes. Though the serial number is kept unique during manufacturing.

The FT serial ID is 8 character long and is formatted as: "`M3S00999`", where `00` stands for the index of the series manufactured (decimal from 1) and `999` stands for an instance index in the series (decimal from 1).

The description is currently always set to the string "`MGen-3 HandController`" at production time.

4.3.MG3_ESC_control

```
int MG3_ESC_control(int pulse_ms);
```

This function pulses the ESC button's line directly. Can be used to turn-on the HC from the power-down state like the user pushed the button for a defined time length.

Parameters

`pulse_ms`	The minimum length of the pulse, in milliseconds unit.

Return value

Default.

Remarks

The function call is blocking, returns only after the pulse signal is turned off. The calling thread is idle until.
Note that if the host system is heavily loaded, the pulse length may be some longer than expected.

To turn-on the HC the recommended pulse length is 200 ms.

To enter into BOOT mode the recommended pulse length is 2000 ms.

To enter into BOOT mode with dimmed brightness the recommended pulse length is over 3000 ms.

If you had no open connection to the FTDI IC when calling this function, the device is opened only for the time of signaling. No communication takes place with the HC.

4.4.MG3_RESET_control

```
int MG3_RESET_control(int pulse_ms);
```

This function pulses the RESET line of the microcontroller (MCU) of the system directly. Do not use it unless it is not critical!

Parameters

`pulse_ms`	The minimum length of the pulse, in milliseconds unit. The shortest 1 ms pulse is enough to reset the MCU.

Return value

Default.

Remarks

WARNING: As the MCU and so the whole system will be reset, there may be data loss on the SD card's filesystem or improper data stored in the device's non-volatile memory (for storing the HC's settings). Avoid using this function, only do it if you are sure that the HC is "frozen" and there seems no other way to recover from this state.

The reset pulse is ineffective if the device is powered down. Nothing will happen, the HC must not be sleeping when applying a reset pulse.

After such an external reset when the HC was already turned on will restart in BOOT mode automatically.

If you had no open connection to the FTDI IC when calling this function, the device is opened only for the time of signaling. No communication takes place with the HC.

4.5.MG3_PollDevice

```
int MG3_PollDevice();
```

Polls for an available MGen-3 HC to be opened but does not open connection to it.

Return value

MG3_OK	A device is found, calling MG3_Open() will work too (if the device's state doesn't change until).
MG3_FAILED_TO_OPEN_DEVICE	This means that there was no available device found. It's not an error as the function's purpose is only polling/detection.

Remarks

Polling for a device is useful when you only want to know if a HC is connected to the system or not. MG3_Open() will not only open the communication channel but does actively communicate as well.

4.6.MG3_Open

BOOT + Firmware

```
int MG3_Open();
```

Opens and initializes the communication with a MGen-3 HandController device.

Return value

MG3_FAILED_TO_OPEN_DEVICE	Can't connect to the HC device. Could be due to various causes, the most common may be that there was no proper device available for opening.
MG3_CANT_READ_PROTOCOL	Connected physically but failed to read protocol/version info or the available command list. Connection is closed.

Remarks

During initialization the protocol version, actual device mode and the list of the available commands and parameters are queried so you don't need to handle the Firmware/protocol version, the API will automatically report an error if a feature is not available on the device due to version mismatch.

MG3_Open() is automatically called by the API functions whenever a communication is to be done but the connection is not opened yet. (So calling this function may not be critical to the application.)

4.7.MG3_Close

```
int MG3_Close();
```

Closes the communication with the MGen-3 HandController device if it was opened.

Return value

MG3_OK	Success, the communication is closed. Another application may access the device through the FTDI driver from now.
MG3_FT230_ERROR	Error is reported by the FTDI driver when releasing the device.

Remarks

The device is automatically released when an object/handle is destroyed/released. Calling this function before exiting your application is not critical. Use the function if your app. does not need to control the device anymore and lets other programs to access it.

4.8.MG3_IsOpen

```
int MG3_IsOpen();
```

Returns a non-zero value (-1) if this object/handle is connected to a MGen-3 HC device.

The return value should not be considered as a MG3_... error code.

4.9.MG3_IsAlive

BOOT + Firmware

```
int MG3_IsAlive(int comm);
```

Tests for the connected HC being still alive and returns non-zero value if it's true.

The return value should not be considered as a MG3_… error code.

Parameters

comm	Non-zero if you also want to apply some communication with it.

Remarks

If the parameter is non-zero, two special NOP (no operation) commands are sent to the HC and their answers are checked. If succeeds, the HC must be ready for other commands.

If the parameter is zero, only the connection to the FTDI IC is tested. There may be that the HC would not respond to a command.
Use this mode if you want to test for the state but don't want to cause extra overhead and delay due to these calls. (Ca. 10 ms time is consumed by one test with communication.)

4.10. MG3_IsBoot

```
int MG3_IsBoot();
```

Returns whether the HC is running in BOOT mode or not. For BOOT mode running a non-zero value is returned.

The return value should not be considered as a MG3_... error code.

Remarks

Note that the function is not issuing communication (returns immediately) so it does not know if the HC changed its running mode, except if it was done through an API call.

If there is no open communication to the HC device the function returns zero. The function will NOT try to open it with MG3_Open() automatically.

4.11. MG3_GetCurrentMode

BOOT + Firmware

```
int MG3_GetCurrentMode( int* pboot,
                        int* pver,
                        int* pexver );
```

Gets the latest running mode of the HC device that the API knows, without doing any communication.

Parameters

`pboot`	Points to a variable to hold the value if the HC is running BOOT mode (non-zero).
`pver`	Points to a variable to hold the protocol version number.
`pexver`	Points to a variable to hold the extra version number.

Return value

`MG3_OK`	Always.

Remarks

If the HC is running in BOOT mode, *pboot is non-zero and *pver shows the running BOOT version in hexadecimal format: 0x00AB for boot version A.B. `*pexver` is reserved.

If the HC is running in Firmware mode, *pboot is zero and *pver shows the Frmware's version in hexadecimal format: 0xAABB for Fw. Ver. AA.BB. `*pexver` is 1 (for final release version).

You can pass NULL as a parameter to skip getting that number.

If the connection is not open to the HC or failed before, zero values are set for each variable pointed. (*pver getting zero indicates this clearly.)

Note that this function returns the *latest known state* of the HC that may have been changed since. To query the actual state (with some communication and so overhead and possible delay) use `MG3cmd_ReadDeviceMode()`.

5. Command functions

It is assumed that the object/handle of a MGen-3 device is already created without error.

Such functions do actively communicate with the MGen-3 HC. The calls are blocking and may have longer delays with a timeout. These functions' names start with `MG3cmd_…` .

It is common for all command functions that:

- connection is opened and initialized if not yet done;
- handles mode change signals (BOOT and Firmware and power-off).

As mode changes are handled you may observe a long running time (ca. 3 seconds) but you don't need to care about implementing and the communication will be flawless.

Common return values (beyond the ones at 0) will not be listed for each function. There are quite many possible error codes, for example the protocol-level errors when sending a command message or receiving the answer. The list below is incomplete but contains some of these too that are more probable:

`MG3_INVALID_COMMAND`	The HC doesn't know the command code. (The API tries to avoid to issue a command that is not present on the HC device, this error is reported directly by the HC.)
`MG3_UNKNOWN_PARAMETER`	The command does not know a parameter code sent. (The same applies here as for the previous error above but for a parameter of a command.)
`MG3_CANT_READ_PROTOCOL`	`MG3_Open()`'s error, see at 4.6. Only returned if the connection was closed before.
`MG3_INVALID_ANSWER`	The answer for the command is invalid. Must be due to some serious error when the protocol's byte streams are violated.
`MG3_IF_COMMAND_UNSUPPORTED`	The HC does not support the command code. The API checks for the allowed command codes and if it is missing from the ones queried at initialization time, returns this error. No comm. has taken place.
`MG3_IF_PRM_UNSUPPORTED`	Same as above but for a parameter of a command. No comm. has taken place.
`MG3_WRONG_ARGUMENT_GIVEN`	An API function was called with wrong / invalid arguments. No comm. has taken place.

It is possible that the MGen-3 HC changes running mode between two API calls, for example it may be powered down. This case the second call will return a special error code indicating this state. Be prepared for this return value when using an API function frequently, like the Virtual UI functions.

MG3_SYS_POWEROFF	The HC has been powered down. No more communication is possible but the connection is not automatically closed, you may use some of the maintenance functions.
MG3_SYS_ERROR	An unknown system message is received from the HC that the (older) API does not know. Communication should be reset or closed after such.

For functions that access a variable (option, control parameter etc.) of the Firmware (in most of the cases accessible from the UI too) there are other possible return values as well.

MG3_VARIABLE_CANT_CHANGE	A variable's value can't be changed now, it's prohibited by the Firmware. The HC must be in some operation mode (e.g. autoguiding active) that disables the value change.
MG3_VARIABLE_SIZE_MISMATCH	The Variable's size is different to the specified one. Not probable for the dedicated API functions.
MG3_VARIABLE_NOT_FOUND	Variable doesn't exist. Also not probable for dedicated API funcs.

5.1.MG3cmd_ReadDeviceMode

BOOT + Firmware

```
int MG3cmd_ReadDeviceMode( int* pboot,
                           uint16* pver,
                           uint16* pextra );
```

Reads the actual running mode and version numbers of the HC device.

Parameters

pboot	Points to a variable to hold the value if the HC is running BOOT mode (non-zero).
pver	Points to a variable to hold the protocol version number.
pexver	Points to a variable to hold the extra version number.

Return value

Default.

Remarks

In contrast to MG3_GetCurrentMode() this function queries the mode at the moment with a cost of some communication but mode changes since the last query are handled internally.
Interpretation of the returned values are the same.

5.2.MG3cmd_ReadMPUID

BOOT + Firmware

```
int MG3cmd_ReadMPUID( uint64* pchipid
                      uint8* puniqueid );
```

Reads the Unique Identifier and chip ID of the embedded Microcontroller.

Parameters

`pchipid`	Points to a variable to hold the chip ID. Currently this is constant with the value of 0x00000002A1120E00.
`puniqueid`	Points to an array to hold the raw UID. The buffer must be at least 16 bytes long.

Return value

Default.

Remarks

The Unique Identifier (UID) is unique among the MCUs and therefore can be used to identify an instance of MGen-3 HC.

UID is returned in its raw form as 16 bytes. This means that is can't be interpreted as a readable C string. (This ID is read and converted to a string by the function `MG3cmd_ReadDeviceInfo()`, which reads other info as well.)

If you need to convert the raw UID to a C string, replace all zero bytes to a space character and care about the terminating zero.

5.3.MG3cmd_ReadPCBID

BOOT + Firmware

```
int MG3cmd_ReadPCBID( uint8* ppcbid );
```

Reads the Hardware ID number of the HC device. This is also called the PCB ID.

Parameters

ppcbid	Points to a byte to hold the PCB ID.

Return value

Default.

Remarks

Hardware ID numbers identify the PCB versions that the Firmware and BOOT programs handle. You don't need to deal with it, though there may be some information later for some API function that are related to this.

The ID is of 4 bits, upper 4 bits are zero. The first released PCB ID has a value of 0x0D and the newer ones get a lower index (0x0C is the latest at the time of API/Fw. 1.04).

5.4.MG3cmd_ReadDistributor

BOOT + Firmware

```
int MG3cmd_ReadDistributor( char* pnamebuf );
```

Reads the name of the distributor company of this MGen-3 instance.

Parameters

pnamebuf	Points to the char buffer to hold the string. Must be at least 32 characters long that includes space for the terminating zero.

Return value

Default.

5.5.MG3cmd_ReadDeviceInfo

BOOT + Firmware

```
int MG3cmd_ReadDeviceInfo( mg3devinfo* pdi );
```

This function calls all the info query functions as all-in-one. The result is stored in a structure.

Parameters

pdi	Points to the structure to hold the data.

Return value

As for the sub function calls.

Remarks

mg3devinfo structure members:

ft_serial	String retrieved by MG3_get_FT_serial(). Zero-terminated.
ft_descr	String retrieved by MG3_get_FT_description(). Zero-terminated.
is_boot prot_ver extra_ver	Values by MG3cmd_ReadDeviceMode().
ChipID UID	Values by MG3cmd_ReadMPUID(). Raw UID is converted to a valid zero-term. C string.
PCB_pins	Value by MG3cmd_ReadPCBID().
distr_name	String retrieved by MG3cmd_ReadDistributor(). Zero-terminated.

5.6.MG3cmd_ReadFWInfo

BOOT + Firmware*

```
int MG3cmd_ReadFWInfo( int n,
                       mg3fwinfo* pfwi );
```

Reads info about a Firmware on the HC.

Parameters

n	Index of the Firmware, valid from 0 to 7 (only the lower 3 bits are used).
pfwi	Points to a structure to receive the data in.

Return value

Default.

Remarks

The HC is able to store maximum 8 Firmware entries, info for these entries can be retrieved by the function in BOOT mode. Currently there is only the main “Astro Firmware” available and is located at the index 0.
In Firmware mode the only one to retrieve data for is itself and the value of parameter n must be zero.

`mg3fwinfo` structure members:

error	Zero if the data below is valid, there is a Firmware at the given index. Non-zero otherwise.
flags	The type flags etc. of the Fw.
version	Version of the Fw. (0xAABB for ver. AA.BB)
npages	Number of flash pages (512 bytes) used by the Fw.
descr	Name of the Firmware. (C string)

The patch subversion number is the upper 4 bits of the `flags` value. For example you can format the full version number as:

```
printf(“%x.%02x.%1x”, fwi->version >> 8, fwi->version & 0xff, fwi->flags >> 12);
```

If a Firmware entry is not valid or the Firmware is corrupted, the `error` member’s

value shows the cause (in BOOT mode only):

1..7	Firmware's main data is corrupted. (Various causes.)
8..13	Checksum error. (Various causes.)
15	No Firmware entry is at the given index.

5.7.MG3cmd_ChangeRunningMode

BOOT + Firmware

```
int MG3cmd_ChangeRunningMode( int change_to );
```

Changes running mode of the HC with safe state transition. (E.g. stopping the system normal way from Firmware mode.)

Parameters

change_to	MG3MODE_TURNOFF if you want to power-down the HC.
	MG3MODE_BOOT if you want to go to BOOT mode.
	MG3MODE_FIRMWARE if you want to start the Firmware.

Return value

MG3_CMD_EXEC_ERROR	Can't execute the mode change now.
MG3_IF_PRM_UNSUPPORTED	BOOT program version below 1.2 does not support power-down issued as an external command. This error code will be returned.

Remarks

When a mode change has been started and this function returned MG3_OK indicating the success, the connection is automatically closed to the device (MG3_Close() called). If the required mode was power-down, no more command function will succeed until the device is turned on manually or by calling MG3_ESC_control().

BOOT and Firmware modes can be chosen from both modes. So it possible to restart the Firmware/BOOT when the Firmware/BOOT is already running. To avoid such ensure that the device is in the other mode using MG3cmd_ReadDeviceMode().

As a workaround for the power-off from BOOT program below version 1.2, you may try the following, this will navigate back and power-off the device "manually":

- send at least 4 ESC button-down events (keyups are not required);
- then send at least 10 Down events;
- then send at least 4 Left events;
- at last send one Up event and one SET event.

5.8.Real-time clock

MGen-3 HC embeds a Real-Time Clock (RTC) unit with battery that stores the actual date and time for the HC.

Time is always meant in UTC as being an astronomical device. Leap years are handled automatically.

Resolution of the clock is seconds, at this level of precision you can read or set the RTC value. Though the Firmware uses millisecond sub-precision for timestamps in log files etc.

At the version of 1.04, only BOOT mode implements the change of the RTC value through the USB interface.

5.8.1. MG3cmd_ReadClock

BOOT

```
int MG3cmd_ReadClock( mg3clock* pc );
```

Reads the RTC value.

Parameters

`pc`	Points to a structure to receive the RTC data.

Return value

Default.

Remarks

`mg3clock` structure members:

`year`	Year of the date. A value of 2000 must be added to this.
`month`	Month of the year [1..12]
`day`	Day of the month [1..31]
`hour`	Hours [0..23]
`min`	Minutes [0..59]
`sec`	Seconds [0..59]

5.8.2. MG3cmd_SetClock

BOOT

```
int MG3cmd_SetClock( mg3clock* pc );
```

Sets the RTC value.

Parameters

pc	Points to a structure holding the new RTC data.

Return value

Default.

Remarks

mg3clock structure members are the same as for MG3cmd_ReadClock() function.

The effect of giving an invalid date or time value is undefined.

Note that setting the clock needs stopping the RTC for a short time. Calling this function often (by the values read from the device) may lead to undue shifting of time value. Set the clock only if it's really necessary.

5.9. Virtual User Interface

Virtual user interface is something like controlling the device remotely like you were controlling it through the User Interface: pushing buttons and watching the display. These functions are meant to support the implementation of such Virtual UI.

MGen-3 supports this either in Firmware or BOOT mode and as mode changes are handled by the API internally, you don't need to care about this. Full control of the device is available.

Button events (both pushing and releasing) are processed by the HC's UI so you need to provide all remote events to the HC to access all the functionality of the buttons.

The API function `MG3cmd_InsertButtonEvent()` does this job, it adds an event to the system's input queue. You should call this function at the time when a button event occurs on your remote interface, including all button release events.

Button auto repeat is NOT covered by this interface, you should re-send button down events if it is held down and auto repeat gets active in your system.

For example remote controlling by a PC, you should receive all keydown/keyup events for some buttons and map and forward them to the HC. Auto repeat is done by your OS or should be done by your application.

Note that the buttons on the HC are also able to generate button events while your Virtual UI is sending its events and these will work coincidentally.

The visual feedback is done through the LCD display and the LEDs state of the buttons. These data are retrieved (updated) by the API function `MG3cmd_UpdateDisplay()`. The current state of the display and LED states are returned, that was at the time of the call. API handles the data transmission and decoding, you need only to provide a buffer for the display content and for LED states.

Note that calling this function very often will somewhat slow down the HC as the display contents need to be compressed and sent for each call. Though the compression method includes that only the display changes are transmitted (compared to the state of the previous call) and so the data may be of low amount, the compression takes up CPU time. (BOOT mode runs on lower system clock speed and does not include buffer for the previous display content and therefore it will slow down more than the Firmware.)

For example remote controlling by a PC, you should call this function repetitively to receive the latest display content and show it on your remote system. The time between calls should be not too short, the recommended value is ca. 50 ms. You need to provide information if you call this API function "for the first time" so that the full display contents are received into your buffer. Later calls can be only for the display changes as long as your buffer's content has not changed.

5.9.1. MG3cmd_InsertButtonEvent

BOOT + Firmware

```
int MG3cmd_InsertButtonEvent( int bcode,
                              int keyup );
```

Adds a button event to the HC's button input queue.

Parameters

bcode	The button's code. Defined macros can be found as MG3BTN_... in mg3_acc_struct.h .
keyup	Non-zero if the event is a key-up / release event instead of a key-down / push. You can use e.g. MG3BTN_KEYUP here.

Return value

MG3_SYS_POWEROFF MG3_SYS_ERROR	As this function may be called frequently and the HC can be powered down through the UI, be prepared for these common return values.

Remarks

For information about the usage read the main info.

5.9.2. MG3cmd_UpdateDisplay

BOOT + Firmware

```
int MG3cmd_UpdateDisplay( uint16* fb,
                          uint8* leds,
                          int fullfrm );
```

Updates the display contents in the array `fb` to the actual display state of the HC. Also receives the actual state of the button LEDs.

Parameters

`fb`	The frame buffer must be provided externally for the HC object. Length must be at least 240 * 320 = 76800 16-bit words.
`leds`	Point to an array to hold the LED states, must be 12 bytes long.
`fullfrm`	Set to non-zero if your buffer in `fb` does not contain display data yet (practically for the first call of this func.). In this case the full frame content is queried. Otherwise only the changes are transmitted and the updates are done faster for most of the calls. Set to non-zero if your display looks weird for any reason (not probable) and you want it to be fully refreshed.

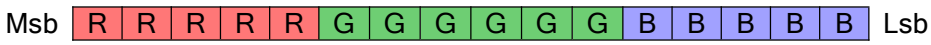
Return value

`MG3_SYS_POWEROFF` `MG3_SYS_ERROR`	As this function must be called frequently and the HC can be powered down through the UI, be prepared for these common return values.

Remarks

For more information about the usage read the main info.

The frame buffer `fb` is 240 x 320 sized (width x height; top left pixel is at index 0), each element is a 16-bit RGB pixel value with format of 5-6-5 bits:



A recommended example C code to convert a pixel value (at position (x,y)) into 3x 8-bits RGB value:

```
uint8 R, G, B;
uint16 px = fb[y*240 + x];
R = (px >> 11) * 8; R += (R >> 5);
G = ((px >> 5) & 0x3f) * 4; G += (G >> 6);
B = (px & 0x1f) * 8; B += (B >> 5);
```

For the LEDs state, values are received in pairs of bytes: the 1st byte is the brightness value (0 to 9 as set in the HC), the 2nd byte is the color code for each button LED. The order of these pairs is: ESC, SET, Left, Right, Up, Down.

Color codes are for ESC and SET:

0	red
1	purple
2	blue

(Color codes are always 0 (red) for the direction buttons.)

Example: ESC and SET are lit with a brightness value of 7, ESC is blue, SET is purple and the direction LEDs are lit with a value of 3, except Right which is lit at a value of 9. The following 12 bytes are returned:

7	2	7	1	3	0	9	0	3	0	3	0
ESC		SET		Left		Right		Up		Down	

5.10. Calibration data

The AutoGuider needs to know the direction and speed where the mount moves the telescope for the autoguider signals and so how the sky would be moved virtually. This data is called “calibration (data)”. It consists of two vectors, one for both axes (RA and DEC). This 2D vector is meant to be in the coordinate system of the Camera’s sensor, having an X and Y component. The vector’s direction shows in which direction the guide star is moving when that axis is signaled and the vector’s length shows the speed of it in pixels/second unit.

The RA and DEC vectors should be orthogonal, though you can set these vectors to any value/direction. The internal calibration procedure automatically calculates orthogonal pair of vectors.

5.10.1. MG3cmd_ReadCalibration

Firmware

```
int MG3cmd_ReadCalibration( mg3calib* pc );
```

Reads the current (active) calibration data.

Parameters

`pc`	Points to a structure to receive the data.

Return value

Default.

Remarks

`mg3calib` structure members:

`strSize`	MUST BE filled prior to the API call.
`rax` `ray`	X and Y component of the RA vector (32-bit float value).
`decx` `decy`	X and Y component of the DEC vector.

If there was no calibration data in the HC, both vectors get zero.

If the DEC axis had been disabled, the DEC vector gets zero.

5.10.2. MG3cmd_SetCalibration

Firmware

```
int MG3cmd_SetCalibration( mg3calib* pc );
```

Sets the current calibration data for the given one.

Parameters

pc	Points to a structure holding the data.

Return value

MG3_CMD_EXEC_ERROR	Some of the given vector values are invalid. The vector don't change in HC.

Remarks

mg3calib structure members:

strSize	MUST BE filled prior to the API call.
rax ray	X and Y component of the RA vector (32-bit float value).
decx decy	X and Y component of the DEC vector.

To clear the calibration data set RA vector's length lower than 1e-6 (practically zero).

To specify a disabled DEC vector its length must be lower than 1e-6.

Note that changing the calibration vector is allowed at anytime, even during an active autoguiding!

5.11. Dithering control

Dithering is a method to lower the effect of the remaining fixed pattern noises present on the preprocessed light frames. The method is about moving the telescope “randomly” between the exposures so the fixed pattern gets smoothed/canceled on the final image. MGen-3 features Dithering and the API enables full control over it:

- read or set the parameters of Dithering;
- trigger a new Dithering process or stop the current one and read the status of it.

5.11.1. MG3cmd_ReadDitheringPrms

Firmware

```
int MG3cmd_ReadDitheringPrms( mg3dithparams* pd );
```

Reads the dithering parameters.

Parameters

`pd`	Points to a structure to receive the data.

Return value

Default.

Remarks

`mg3dithparams` structure members:

`strSize`	MUST BE filled prior to the API call.
`enable`	Non-zero if Dithering feature is enabled.
`radius`	Radius (or size) of the max. Dithering distance (in pixels unit).
`period`	How many exposures are taken in series without a new Dithering position.

No important remarks on this.

5.11.2. MG3cmd_SetDitheringPrms

Firmware

```
int MG3cmd_SetDitheringPrms( mg3dithparams* pd );
```

Sets the dithering parameters.

Parameters

pd	Points to a structure holding the data.

Return value

Default.

Remarks

mg3dithparams structure members:

strSize	MUST BE filled prior to the API call.
enable	Same. The variable is changed only if this value is ≥ 0.
radius	Same. The variable is changed only if this value is ≥ 0.
period	Same. The variable is changed only if this value is > 0.

Enabling / disabling of Dithering is safe along with the other variables. Dithering is enabled only after all the other parameters have been changed or is disabled before the other parameters are changed.

5.11.3. MG3cmd_ReadDitheringState

Firmware

```
int MG3cmd_ReadDitheringState( int* pdst );
```

Reads the current state of the Dithering process.

Parameters

`pdst`	Points to the variable receiving the status value.

Return value

Default.

Remarks

The status value is a 8-bit status byte:

ag	dt	-	-	-	-	S	S

Bit 'ag' indicates if AutoGuiding is active (1).

Bit 'dt' indicates if Dithering is enabled (1).

Value 'S' shows the state of Dithering process:

0: It is currently inactive or has ended the process started in the past.
1: It is currently active and is displacing the guiding star or is still measuring the star's final position as the end of this process.
2-3: reserved

5.11.4. MG3cmd_DitheringControl

Firmware

```
int MG3cmd_ReadDitheringControl( int trigger );
```

Triggers a new Dithering process or stops it if it was active.

Parameters

`trigger`	Set to non-zero to trigger a new Dithering. Set to zero to finish the movement of a Dithering.

Return value

`MG3_CMD_EXEC_ERROR`	Only if `trigger` was non-zero. Can't start a Dithering. (Of any causes: no Camera, no guide stars, not autoguiding, already dithering or dithering is disabled.)

Remarks

Triggering does the same as it was signaled by the AutoExposure in the Firmware. It generates a new autoguiding center position by the Dithering parameters and starts the repositioning of it.

If `trigger` was zero, the function never returns with `MG3_CMD_EXEC_ERROR`. Stopping a dithering can be always signaled, even if it was not active. To wait for the end of the Dithering process call `MG3cmd_ReadDitheringState()` until the state bits gets zero.

It is allowed to trigger a Dithering process if the AutoGuiding is active, independent of that the AutoExposure could also trigger a Dithering process. Users should avoid using both at the same time, there might be some confusion between the synchronization of these processes.

5.12. Guider setup data

"Guider setup" parameters are those data that the MGen-3 HC can not know automatically but are helpful to do or display something well. These are:

- the effective focal length of the guiding optics;
- the actual autoguiding speed of the mount, as a fraction of the sidereal speed.

5.12.1. MG3cmd_ReadGuiderSetup

Firmware

```
int MG3cmd_ReadGuiderSetup( int* pfoclen_mm,
                            float* pagspeed );
```

Reads the guider setup parameters.

Parameters

`pfoclen_mm`	Points to a variable to receive the focal length value of the guiding optics. Unit is millimeter.
`pagspeed`	Points to a variable to receive the autoguiding speed (ratio).

Return value

Default.

Remarks

If any of the function arguments is NULL, that value won't be read.

5.12.2. MG3cmd_SetGuiderSetup

Firmware

```
int MG3cmd_SetGuiderSetup( int foclen_mm,
                           float agspeed );
```

Sets the guider setup parameters.

Parameters

`foclen_mm`	Same. The variable is changed only if this value is > 0.
`agspeed`	Same. The variable is changed only if this value is > 0.

Return value

Default.

Remarks

No important remarks on this function.

5.13. Imaging parameters

Imaging parameters are the gain and exposure time used by the Camera.

5.13.1. MG3cmd_ReadImagingPrms

Firmware

```
int MG3cmd_ReadImagingPrms( int* pgain,
                            int* pexpo );
```

Reads the actual imaging parameters used by the Camera.

Parameters

`pgain`	Points to a variable to receive gain value. It is a code, not a scalar. For MGen-3 Camera gain interval is 0 (low, 1x) to 9 (high, 8x);
`pexpo`	Points to a variable to receive exposure time in milliseconds unit. Valid from 1 to 4000.

Return value

Default.

Remarks

If any of the function arguments is NULL, that value won't be read.

5.13.2. MG3cmd_SetImagingPrms

Firmware

```
int MG3cmd_SetImagingPrms( int gain,
                           int expo );
```

Sets the actual imaging parameters used by the Camera.

Parameters

gain	Same. The variable is changed only if this value is ≥ 0.
expo	Same. The variable is changed only if this value is > 0.

Return value

Default.

Remarks

New imaging parameters will be applied only after the ongoing exposure has been taken. Therefore after setting new imaging parameters, you most probably will see the next incoming frame with the old params. (as the exposure has not been interrupted) and only the later ones will be taken with the new parameters.
Of course the new values can be read back right after setting them.

5.14. AutoGuiding related

AutoGuiding parameters define how the AutoGuiding must be "executed", independently for the two axes.

The main part is a PI controller. The proportional (P) component reacts for the sudden changes in offsets, while the integral (I) component compensates the low frequency remnant of the periodic error. The optimal coefficient/value of these components depend much on the characteristics of the mount's tracking error and somewhat depend on the actual seeing conditions. For a given mount a fixed P+I value may be said to be optimal, though MGen-3 offers an automatic trimming mode of these parameters, so they are optimized continuously during guiding. This optimization is called the "auto trim" mode.

There are other functions that are related to AutoGuiding, those are listed here as well.

5.14.1. MG3cmd_ReadAGPrms

Firmware

```
int MG3cmd_ReadAGPrms( mg3agparams* pag,
                       int ax );
```

Reads the AutoGuiding parameters of an axis.

Parameters

`pag`	Points to a structure to receive the data for the axis.
`ax`	Zero to read RA's parameters. Non-zero to read DEC's parameters.

Return value

Default.

Remarks

`mg3agparams` structure members:

`strSize`	MUST BE filled prior to the API call.
`mode`	AutoGuiding mode. Zero means manual mode (fixed parameters), one means "auto trim" mode. DEC axis works always in manual mode (0).
`Prop`	Proportional coefficient of the PI controller.
`Integr`	Intergal coefficient of the PI controller.

`I_enable`	Non-zero if the integral component is enabled. (Otherwise the controller is only of a P type.) For DEC axis integral component is always disabled (0).
`Tol`	Tolerance value, in pixel units.
`filter`	Averaging filter value allowing more rare corrections for the axis. Only available for DEC axis.

Tolerance value should be kept at zero if the mount reacts to the autoguiding signals well, does not exhibit much delay or imprecise / non-linear movements. For a low quality mount guiding is very hard to done well, too much quasi-random movements would be done ruining the final image. For this reason the user may set the tolerance level greater than zero to avoid correction singals as the guiding star is treated to be “near enough” the required position.

5.14.2. MG3cmd_SetAGPrms

Firmware

```
int MG3cmd_SetAGPrms( mg3agparams* pag,
                      int ax );
```

Sets the AutoGuiding parameters of an axis.

Parameters

pag	Points to a structure holding the new data for the axis.
ax	Zero to set RA's parameters. Non-zero to set DEC's parameters.

Return value

Default.

Remarks

mg3agparams structure members:

strSize	MUST BE filled prior to the API call.
mode	Same. The variable is changed only if this value is ≥ 0.
Prop	Same. The variable is changed only if this value is ≥ 0.
Integr	Same. The variable is changed only if this value is ≥ 0.
I_enable	Same. The variable is changed only if this value is ≥ 0.
Tol	Same. The variable is changed only if this value is ≥ 0.
filter	Same. The variable is changed only if this value is > 0.

The table here indicates which of the parameters are variable (●), constant or dontcare (-) for the axes and modes: (other params. may be changed but will have no effect on guiding)

	RA manual	RA "auto trim"	DEC
Prop	●	●	●
Integr	●	●	-
I_enable	●	enabled	disabled
Tol	●	-	●
filter	1	1	●

5.14.3. MG3cmd_ReadLastFrameData

Firmware

```
int MG3cmd_ReadLastFrameData( mg3framedata* pfd);
```

Reads data for the last known frame used for AutoGuiding.

Parameters

`pfd`	Points to the structure to control and receive data.

Return value

Default.

Remarks

`mg3framedata` structure members:

`strSize`	MUST BE filled prior to the API call.
`query`	Flags indicating what data need to be read. Flags must defined using `LFD_…` macros.

Members for `LFD_BASIC_DATA`:

`star_present`	Non-zero if there was a star evaluated on the final image.
`ag_enabled`	Non-zero if AutoGuiding is currently enabled.
`frame_idx`	Frame's sequential index since the start of the Camera.
`pos_x` `pos_y`	Center of the star in the Camera's coord. system.
`fwhm`	Full Width at Half Maximum value for the final star. (Must be above of the individual star's FWHM.)
`brightness`	Not used currently, ignore the data.

Members for `LFD_VEC_AGCENTER`:

`ag_center_x` `ag_center_y`	AutoGuiding center position, where the star should be held at. (Camera's coord. system)

Members for `LFD_VEC_AGREFPT`:

`ag_refpt_x` `ag_refpt_y`	AutoGuiding reference point, around which the AutoGuiding center positions will be spread by the Dithering. (Camera's coord. system)

Members for `LFD_VEC_XFORM`:

`ag_cal_ra_x` `ag_cal_ra_y`	RA axis calibration's normal vector. You can use this to derive RA drifts from the Camera-position vectors above. (Equations are shown later.)
`ag_cal_dec_x` `ag_cal_dec_y`	Same for DEC axis.

Members for `LFD_IMAGE`:

`pimage`	Points to an array of `int16` elements to receive the final image contents of the last frame.
`size[2]`	To specify you buffer's max. size and to get the received image's size.

If you query for the last frame's data including the image (`LFD_IMAGE`) often, you may use the `LFD_IGNORE_KNOWN` flag to avoid receiving an already known frame's content again and again causing useless data transmission and slowing down the communication. In this case you have to specify `frame_idx` member prior to the API call and the Firmware won't transmit the frame content if the last frame's index equals to the given one. It is recommended using a static `mg3framedata` structure in which you can have the `frame_idx` member intact (but for first initialize to 0xffffffff), the API call will refresh it whenever a new frame is available. `size[2]` member will be set to zeroes if there were no new frame received at the current call.

When requiring the frame's image (`LFD_IMAGE`), you have to specify the buffer for it by `pimage` and set `size[2]` so that size[0] * size[1] equals your buffer's length (number of `int16` pixels). Both values of `size[2]` must be greater than zero. After receiving the data `size[2]` is set to the used guiding window's size to inform you. (Currently it is constant 32 * 32 pixels.)
The pixel values are normalized but with a maximum factor so that the displayed image would not show extremely high noise. MGen-3 displays this final image content too, zero pixel values are shown as dark gray and maximum (32767) values are shown as white.
Note that it is possible to get negative pixel values.

To calculate RA drift D_{RA} (in pixel units) from the data received use this equation:

$$\overline{pos} = (pos\_x, pos\_y)$$
$$\overline{cal}_{RA} = (cal\_ra\_x, cal\_ra\_y)$$
$$\overline{agc} = (ag\_center\_x, ag\_center\_y)$$
$$D_{RA} = (\overline{pos} - \overline{agc}) \cdot \overline{cal}_{RA}$$

or expressed with scalars

$$D_{RA} = (pos\_x - ag\_center\_x) \cdot cal\_ra\_x + (pos\_y - ag\_center\_y) \cdot cal\_ra\_y$$

Similar must be done for DEC drift but using the DEC calibration's normal vector.

About the Camera's coordinate system and points:

As the final image consists of multiple stars from multiple absolute (and sub-pixel) positions on the image sensor, there is no real origo for the final image. Instead, a virtual origo is defined as the center of the final image's guiding window after the star search. Normally the averaged stars (final star) would be at this position in this window so near the virtual origo. Therefore the position values read by this API function will be close to zero after a star search procedure. This origo stays the same until a new star search has taken place. (AutoGuiding center and reference points may change/be relocated during the usage.)
All these queried points and positions are in this coordinate system. The X and Y axis are equal to the one of the image sensor's and therefore their unit is always pixels.

AutoGuiding reference point is the point around which the current AutoGuiding center positions will be chosen by the Dithering. Reference point only changes at the (re)start of the AutoGuiding if the "set new ref.pt." parameter is specified for it.

AutoGuiding center position is the position that the (final) guiding star should be kept at. This position is changed when a Dithering is triggered.

5.15. Complex functions' interface

In MGen-3's Firmware there are several features / functions that are required for guiding, which use many resources or need more time to run. (Calibration, Star Search etc.) Controlling of the AutoGuider is done mostly using these. Also the API has interface function calls to manage these internal "complex functions".

This kind of "complex functions" run in an asynchronous manner due to their time requirements. You can call for such a function to be started with the given (or default) parameters using an API func. Then you poll for the state or internal return value of the running function using another API function. You may signal the running function to be canceled meanwhile.

Two complex functions can't be run at the same time. There are also some cases when you can't start a function though you may, one of these is when a message box is shown on the LCD. (This may be eliminated in the future.)

The internal return codes are defined in `mg3_iretval.h`. While errors may be shown as a message box on the LCD if the functions are issued from the UI (buttons), this is not the case for calling them through the API. There won't be errors shown on the LCD (neither won't the UI blocked), instead you should rely on the internal return codes.

There's a list of the probably internal return codes with some description, which are not listed for each function:

`ERR_OK`	Success, the complex function has run without error.
`ERR_CANCELED`	You have canceled the running of the function from the API. (The function has exited now.)
`ERR_FUNCTION_IS_DISABLED`	Running of the function is permitted now. For example AutoGuiding is active and you wanted to run a Calibration.
`ERR_CAMERA_NOT_ACTIVE`	Func. Requires the Camera but it is not present.
`ERR_GUIDESTAR_NOT_AVAILABLE`	There's no guide star available currently on the last frame.
`ERR_CAMERA_BUSY`	The Camera/AutoGuider entity is used by some other internal process.

To execute these cpx. functions as a blocking call (requiring the internal result code), you may use `MG3cmd_WaitFunctionResult()` API func that also introduces timeouts. (If you want to implement it yourself, check the source code of this API func. as a how-to-use example.)

Example code for a function execution as a blocking call:

```
mg3func_cal cal;
cal.strSize = sizeof(cal);
…    // you specify the parameters here

int res = hc.MG3cmd_Function_Calibrate(&cal);       // start Calibration
if(MG3_FUNC_BUSY == res) { /* some cpx. func. is already running */ return; }
else if(MG3_OK != res) { /* handle the other errors */ return; }

int fnres;
const int to_s = 8;          // 8 seconds timeout for the cpx. function
const int use_cancel = 1;   // whether to cancel the function if the timeout occurs
res = hc.MG3cmd_WaitFunctionResult(&fnres, to_s, use_cancel);
if(MG3_FUNC_BUSY == res) { /* function is still running… */ return; }
if(MG3_OK != res) { /* handle the error */ return; }

printf("Internal result code = %d\n", fnres);
```

Below there are the API functions that are used to manage the running of a complex function. Starting of the complex functions will be listed one-by-one thereafter as their parameters and usage are different. Naming convention for the starting function is `MG3cmd_Function_…()`. These functions return `MG3_FUNC_BUSY` if they can't be run because some other is already running.

5.15.1. MG3cmd_ReadFunctionState

Firmware

```
int MG3cmd_ReadFunctionState( int* pires );
```

Reads the state and return value of the currently or last run complex function (issued from the API).

Parameters

`pires`	Points to a variable to receive the internal return value, the result of the function run.

Return value

`MG3_FUNC_BUSY`	The last issued cpx. function is still running on the HC.
`MG3_OK`	The function has ended and the return value is stored in `*pires`.

Remarks

Independent of whether the HC runs a cpx. function, this API call will return the result of the one called from the API. The user may have started a function manually by the UI after, this API function will show that the function (called by the API) has ended as this is the case. `MG3_FUNC_BUSY` is returned only if the API has started the running cpx. function.

5.15.2. MG3cmd_CancelFunction

Firmware

```
int MG3cmd_CancelFunction();
```

Signals the currently running function to be canceled.

Return value

MG3_FUNC_BUSY	Normal return value, no error. Just indicating that a function was running at the time of the call.
MG3_OK	There was no function running, nothing has been done.

Remarks

The function always succeeds. (Except for protocol level etc. errors.)

All functions running will be signaled to be canceled by this API call, regardless to that if it was called by the API or not. This has the same effect as pressing ESC by the user when a (blocking) UI function has started by him before. (E.g. Calibration.)

5.15.3. MG3cmd_WaitFunctionResult

Firmware

```
int MG3cmd_WaitFunctionResult( int* pires,
                               int timeout_s,
                               int to_cancel);
```

Waits for the previously started complex function's internal return value to be ready, which means that the function has ended.

Parameters

`pires`	Points to a variable to receive the internal return value.
`timeout_s`	Timeout value in seconds unit, must be higher than zero. After this amount of time the API call returns or if specified (`to_cancel`) tries to cancel the still running function and waits for its end.
`to_cancel`	If zero, the API func. returns immediately after the timeout. If non-zero, the API func. tries to cancel the cpx.function before returning.

Return value

`MG3_FUNC_BUSY`	Means that the cpx. function was still running when this API call returned.
`MG3_OK`	Done, there is the internal result code at `*pires`. Cpx.func. has ended some way.

Remarks

This API function uses `MG3cmd_ReadFunctionState()` to poll for the result periodically and applies a canceling with `MG3cmd_CancelFunction()` if specified and waits for it to be canceled.

If `to_cancel` parameter is zero, the API func. returns `MG3_FUNC_BUSY` in the case of a timeout. `MG3_OK` is returned only if the function has ended before the timeout.

If `to_cancel` parameter is non-zero, the API func. will signal the cancellation of the cpx.function at a timeout and waits until its end. This cancellation must end under 5 seconds and if the function is still running when this 2nd timeout occurs, `MG3_FUNC_BUSY` is returned. Otherwise `MG3_OK` is returned and the internal result code in `*pires` must be the value of `ERR_CANCELED`.

5.16. MG3cmd_Function_StarSearch

Firmware

```
int MG3cmd_Function_StarSearch( mg3func_ss* pss );
```

Starts a new star searching procedure using the specified (or default) parameters.

Parameters

`pss`	Points to a structure holding the parameters.

Return value

Default.

Remarks

`mg3func_ss` structure members:

`strSize`	MUST BE filled prior to the API call.
`min_gain` `max_gain`	The gain interval from which the resulting gain can be picked. Both values must to be ≥ 0 to be applied. Otherwise the min & max values stored in the Fw. will be used.
`min_expo` `max_expo`	The exposure time interval in which the resulting exposure time can be set. Both values must to be > 0 to be applied. Otherwise the min & max values stored in the Fw. will be used.
`prefer_long_expo` Fw. 1.04	Zero if a shorter exposure time is preferred. Non-zero if a longer exposure time is preferred.

Star Searching is performed as the following:

- Min. gain and exposure time is set for first and a frame is taken;
- Stars are located and measured on the frame;
- New gain+expo combo is chosen so that the brightest useful star would be under saturation and a new frame is taken.

This is done at several times until no more considerable change is possible.

After the function has run and there was at least one star found (max. is 100 now), these stars are aligned together at sub-pixel level and are weighted quasi optimally so that the maximum SNR is reached. This is called the “final star” image. At each new frame coming a new “final star” image is calculated and evaluated and AutoGuiding is performed based on this data. (See `MG3cmd_ReadLastFrameData()` for accessing these.)

Internal result / error codes

ERR_OK	The function has run without an error. This does not indicate implicitly that it has found any stars.
ERR_CANCELED	In the case of cancellation of the Star Search, the previous multi star data is restored and the previous final guiding star (if any) will be available again. Gain and exposure time is also restored.

5.17. MG3cmd_Function_Calibrate

Firmware

```
int MG3cmd_Function_Calibrate( mg3func_cal* pcal );
```

Starts a Calibration procedure. If succeeds new calibration data has been created.

Parameters

`pcal`	Points to a structure holding the parameters.

Return value

Default.

Remarks

`mg3func_cal` structure members:

`strSize`		MUST BE filled prior to the API call.
`tries_max`	Fw. 1.04	Maximum number of attempts to move the DEC axis for the first time. Must be set to greater than zero.
`retry_rate`	Fw. 1.04	The ratio between two pulse lengths when moving the DEC axis for the first time. (Limited into [1.0, 2.0].) Only applied if `tries_max` is specified.

Note that Fw. 1.03 does not know the options above but the API call will do the calibration with the Fw.'s default options.

Calibration can be started only when there is a guiding star present. (After Star Search.)

Calibration may take for a long time, depending on the mount and the used autoguiding setup.
Prior to start the Calibration procedure, you may be ensured that proper data is given in Guider setup variables (`MG3cmd_SetGuiderSetup()`). Focal length and AG speed is used to apply just enough guiding signal pulses and so helps the Calibration to finish as soon as possible.

Internal result / error codes

`ERR_OK`	New calibration data has been calculated.
`ERR_CANCELED`	If the cpx.function is canceled, the previous calibration vectors remain.

5.18. MG3cmd_Function_AGStart

Firmware

```
int MG3cmd_Function_AGStart( int new_refpt );
```

Starts the AutoGuiding.

Parameters

new_refpt	Set to non-zero if you want the latest known guiding star position to be the AutoGuiding reference point. (More on this here).

Return value

Default.

Remarks

Starting AutoGuiding for the first time since the power-on of the HC always sets the reference point regardless of the function argument.

AutoGuiding can be started only when there is a guiding star and calibration data present.

Note that many variables (some of the AutoGuiding parameters too) are now allowed to be changed while the AutoGuiding is active. Calling such API function will return error and only some of the variables may be changed. Always stop AutoGuiding when you need to do changes that may affect the guiding (gain and exposure time too).

Internal result / error codes

ERR_OK	AutoGuiding has been started. The next guiding signal is put out at the next available Camera frame.
ERR_AG_ACTIVE	AutoGuiding is already enabled. Nothing was done. (Neither the reference point is changed even if it was specified.)
ERR_CALIBRATION_MISSING	There is no calibration data. It is required for guiding.

5.19. MG3cmd_Function_AGStop

Firmware

```
int MG3cmd_Function_AGStop();
```

Stops the AutoGuiding.

Return value

Default.

Remarks

Stopping AutoGuiding must always succeed, even if it was not active. The possibly active AG signals are stopped immediately.

5.20. MG3cmd_Function_OnePushStart

Firmware

```
int MG3cmd_Function_OnePushStart();
```

Starts the One-push AutoGuiding procedure.

Return value

Default.

Remarks

One-push start covers the calling of Star Search, Calibration and Start AutoGuiding functions.

Stopping guiding is of course done with MG3cmd_Function_AGStop().

If there is the final image's star (still) present, Star Search and Calibration is not re-done after the later calls of this API fn.

Internal result / error codes

Any internal error codes are possible that are present for its sub-functions.

6. BOOT functions

As most of the functions are implemented only by the Firmware, so are some that are only available for the BOOT mode. These are the Firmware management functions.

You may use these functions to update the Firmwares on the HC and Camera directly from your application.

Of course the HC need to be in BOOT mode, use `MG3cmd_ReadDeviceMode()` to check and `MG3cmd_ChangeRunningMode()` to change to BOOT mode if necessary. (Or turn-on the HC into BOOT mode using the call "`MG3_ESC_control(2000);`".)

6.1.MG3cmd_UpdateHCFirmware_File

BOOT

```
int MG3cmd_UpdateHCFirmware_File(
                                const char* keyfn,
                                const char* fwfn
                                int upd_deletable,
                                int upd_mode );
```

Updates the HC's Firmware with the given files and options.

Parameters

keyfn	Filename (with path) of the User Key file to be used. Can be NULL or an empty string if there is no key file required (freeware).
fwfn	Filename (with path) of the released binary firmware file, e.g. LMG3_fw0104_0.bin.
upd_deletable	8 bits of flags indicating which Firmware (out of the 8) is allowed to be deleted if there were not enough space. Yet not used, set it to zero.
upd_mode	Mode flags of the update. (UPD_MODE_... macros) Mostly not used, set it to zero.

Return value

MG3_UPDATE_PROHIBITED	Update is not allowed now. An update is already in progress (directly from the SD card for example).
MG3_UPDATE_INTERNAL_FAULT	Any serious internal error, not probable with valid Fw. data.
MG3_UPDATE_FAILED_WRONG_HS_KEY	Wrong handshaking key is given. (Handshaking is to avoid random entering into update state when the protocol stream is messed up.) Not probable error.
MG3_UPDATE_FAILED_WRONG_USER_KEY	The User Key given is invalid for this Firmware, update is not allowed.
MG3_UPDATE_FAILED	There was some error during the decoding and programming of the Firmware. (Not probable for a valid Fw. data.)
MG3_UPDATE_FAILED_NO_SPACE	No space for the Firmware.

Remarks

The API call is blocking, returns only after the result of the update process is known. Meanwhile the LCD will display that the Firmware is being updated from the USB command interface and the progress bar is visible.

If the function returns an error e.g. when a wrong user key is provided, there may be that the HC stays in update mode and displays some state. (While you can't apply an other Fw. update call.) Check out `MG3cmd_CancelUpdateHC()` for the info how to handle this.

Raw Fw. and key data are read internally from the given files. If you have another type of source for these you can check the source code of this API function how to call `MG3cmd_UpdateHCFirmware()` directly with the raw binary data.

Test upload:

`upd_…` arguments should be set to zero. Though there is a mode you may need, the "test upload" of a Firmware. This means that the Firmware will be decoded and checked just as it was programmed to the flash but the data would not be programmed, the Firmware entries also remain the same. Useful to check if a User Key enables the update of the Firmware or not, or to check the Fw. data's validness.

For this test mode set `upd_mode` to `UPD_MODE_TEST_UPL`.

6.2.MG3cmd_CancelUpdateHC

BOOT

```
int MG3cmd_CancelUpdateHC();
```

Cancels an update process.

Return value

Default.

Remarks

As the API fully implements the update communication, we can say this function is used only to exit the update mode after an unsuccessful update process.
If the update process succeeded, API automatically calls this to quit the update mode and enable updates (e.g. for the Camera) for the forthcoming API calls.

In the case of an error during the update process, HC stays in update mode for more 10 seconds (so that the user can read the error, if shown), then times out and quits it. This function is used to quit update mode immediately and to end showing the state/error.

6.3.MG3cmd_UpdateCamFirmware_File

BOOT

```
int MG3cmd_UpdateCamFirmware_File( const char* fwfn );
```

Updates the MGen-3 Camera's Firmware that is connected to the HC directly.

Parameters

`fwfn`	Filename (with path) of the released binary firmware file, e.g. `LMG3_fw0104_0.bin`.

Return value

`MG3_UPDATE_FAILED_TO_START_CAMERA`	Could not turn-on the Camera.

…and the ones possible for the HC's update.

Remarks

The same applies for the Camera Firmware's update as for the HC Firmware's, about staying in the update mode after an error. Though there is no cancel function in API for the Camera update process currently to exit the update mode and display.

To implement update using raw binary data, check the source code of this function.

There are no user key required and other options available for the Camera in the mean of the Firmware. Only one Firmware can be on it at a time.

6.4.MG3cmd_Cam_Start

BOOT

```
int MG3cmd_Cam_Start();
```

Enables the USB host and enumerates the connected MGen-3 Camera in BOOT mode.

Return value

MG3_UPDATE_FAILED_TO_START_CAMERA	Though this is not a special update command, this code indicates that the Camera was not found.

Remarks

The HC enables the USB host and tries to enumerate the MGen-3 Camera. On success (MG3_OK) the Camera is left turned on waiting for camera boot commands to execute, otherwise USB host is stopped (powered down) automatically.

WARNING: as the Camera start is handled as a Camera Fw. update process, the HC will display this update mode while the Camera is turned on. Similar to the HC Firmware's update there is a 10 seconds timeout, after which the HC stops the camera (disables the USB host) and exits update mode. You need to issue camera commands (MG3cmd_Cam_…) before these timeouts elapse to keep the Camera turned on.

6.5.MG3cmd_Cam_Stop

BOOT

```
int MG3cmd_Cam_Stop();
```

Disables (powers down) the USB host if it was enabled.

Return value

Default.

Remarks

The HC disables the USB host and so the possibly connected MGen-3 Camera is turned off.

6.6.MG3cmd_Cam_ReadDeviceInfo

BOOT

```
int MG3cmd_Cam_ReadDeviceInfo( mg3devinfo* pdi );
```

Reads the device info of the MGen-3 Camera connected.

Parameters

pdi	Points to the structure to receive the data.

Return value

MG3_CMD_EXEC_ERROR	Command couldn't be executed on Camera or has been executed but it returned some error.

Remarks

The structure is the same used for receiving device info data for the HC but not all members can be filled with data. The unused members are filled with zero bytes and are not listed below:

mg3devinfo structure members:

is_boot	As this command must be executed for the Camera in BOOT mode, is_boot must be one.
prot_ver	Protocol version which equals the BOOT version of the Camera as 0x00AB for the version A.B.
extra_ver	Extra version index, not yet used (zero).
ChipID	The MCU's chip ID. Currently this is constant with the value of 0x00000001A11D0A00.
UID	The Unique Identifier (UID) of the MCU in the Camera, similar to the HC. A zero-terminated C string.
PCB_pins	Hardware ID code (byte) of the PCB in the Camera, similar to the one for the HC. (These PCB codes are not meant to match between the used HC and Camera instance.) The lowest 3 bits are valid, the other are set to one for the 8 bits of the code byte.

6.7.MG3cmd_Cam_ReadFWInfo

BOOT

```
int MG3cmd_Cam_ReadFWInfo( mg3fwinfo* pfwi );
```

Reads the Firmware's info of the MGen-3 Camera connected.

Parameters

`pfwi`	Points to the structure to receive the data.

Return value

`MG3_CMD_EXEC_ERROR`	Command couldn't be executed on Camera or has been executed but it returned some error.

Remarks

`mg3fwinfo` structure members:

`error`	Zero if the data below is valid, there is a Firmware at the given index. Non-zero otherwise.
`flags`	The type flags etc. of the Fw.
`version`	Version of the Fw. (0xAABB for ver. AA.BB)
`npages`	Number of flash pages (512 bytes) used by the Fw.
`descr`	Name of the Firmware. (C string) Currently always "`MGen-3 Camera`".

The patch subversion number is the upper 4 bits of the `flags` value as for the HC Firmware flags.