TESCAN Scanning Electron Microscope

SharkSEM Remote Control

Revision 2.0.22
10th December 2015

SEM Models: Vega 3, Mira 3, Maia 3, Lyra 3, Gaia 3, Fera 3, Xeia 3

TESCAN Brno, s.r.o.
Brno, Czech Republic

TESCAN
DIGITAL MICROSCOPY IMAGING

Libušina třída 816/1, Brno, Czech Republic
Phone +420 530 353 411
info@tescan.cz

Table of Contents

Table of Contents 3
Preface 7
Overview 8
  TCP/IP Protocol 8
  Concurrent Clients 9
  Functions 9
  Protocol design 9
Command Processing 11
  Single Command 11
  Single Queue 12
  Multiple Queues 13
  Internal Command Processing 13
Message Structure 15
  Scalar Data Types 15
  Arrays 15
  Compound Types 15
  Storage Rules 16
  Message Header 16
  Message body 18
Session Management 21
Communication 23
    TcpGetDevice 23
    TcpGetSWVersion 23
    TcpGetVersion 24
    TcpRegDataPort 25
Electron Optics 27
    AutoColumn 27
    AutoGun 28
    AutoWD 28
    Degauss 29
    EnumCenterings 30
    EnumGeometries 30
    EnumPCIndexes 31
    Get3DBeam 32
    GetBeamCurrent 33
    GetCentering 33
    GetGeometry 34
    GetGeomLimits 35
    GetIAbsorbed 36
    GetImageShift 36
    GetPCContinual 37
    GetPCFine 37
    GetPCIndex 38
    GetSpotSize 38
    GetViewField 39
    GetWD 39

Set3DBeam ... 40
SetBeamCurrent ... 41
SetCentering ... 41
SetGeometry ... 42
SetImageShift ... 43
SetPCContinual ... 43
SetPCFine ... 44
SetPCIndex ... 45
SetViewField ... 45
SetWD ... 46
Manipulators ... 47
ManipGetConfig ... 47
ManipGetCount ... 47
ManipGetCurr ... 48
ManipSetCurr ... 49
Stage Control ... 50
StgCalibrate ... 50
StgGetLimits ... 50
StgGetMotorized ... 51
StgGetPosition ... 51
StgIsBusy ... 52
StgIsCalibrated ... 53
StgMove ... 53
StgMoveTo ... 54
StgStop ... 55
StgWatchdog ... 55
Input Channels and Detectors ... 57
DtAutoSignal ... 57
DtEnable ... 57
DtEnumDetectors ... 58
DtGetChannels ... 59
DtGetEnabled ... 60
DtGetGainBlack ... 60
DtGetSelected ... 61
DtSelect ... 61
DtSetGainBlack ... 62
Scanning ... 64
ScEnumSpeeds ... 64
ScGetBlanker ... 64
ScGetExternal ... 65
ScGetSpeed ... 66
ScLUTParGet ... 66
ScLUTParSet ... 67
ScLUTSrcGet ... 68
ScLUTSrcSet ... 69
ScScanBitmask ... 69
ScScanLine ... 69
ScScanLitho ... 71
ScScanXY ... 71
ScSetBeamPos ... 73

ScSetBlanker 74
ScSetExternal 74
ScSetSpeed 75
ScStopScan 76
Scanning Mode 78
SMEnumModes 78
SMGetMode 78
SMGetPivotPos 79
SMSetMode 80
Vacuum 81
VacGetPressure 81
VacGetStatus 81
VacGetVPMode 82
VacGetVPPress 83
VacPump 83
VacSetVPMode 84
VacSetVPPress 84
VacVent 85
Airlock 87
ArlGetType 87
Airlock type 1 – manual 88
ArlCloseValve 88
ArlGetStatus 88
ArlOpenValve 89
ArlPump 90
ArlVent 90
Airlock type 2 – motorized 92
Arl2Calibrate 92
Arl2GetStatus 92
Arl2Load 93
Arl2MoveStop 94
Arl2Pump 94
Arl2Recovery 95
Arl2Unload 96
Arl2Vent 96
Detector and shutter manipulation 98
NoseCalibrate 98
NoseGetConfig 99
NoseGetPosition 100
NoseIsBusy 100
NoseIsCalib 101
NoseMoveToMem 101
NoseMoveToPos 102
NoseStop 103
Detector nose guard 105
NGuardGetStatus 105
NGuardLock 106
NGuardTest 107
NGuardUnlock 107
High Voltage 109

HVAutoHeat ... 109
HVBeamOff ... 110
HVBeamOn ... 111
HVEnumIndexes ... 111
HVGetBeam ... 112
HVGetEmission ... 113
HVGetFilTime ... 114
HVGetHeating ... 114
HVGetIndex ... 115
HVGetVoltage ... 115
HVSetIndex ... 116
HVSetVoltage ... 117
HVStopAsyncProc ... 118
SEM Presets ... 119
PresetEnum ... 119
PresetSet ... 119
Chamber Camera ... 121
CameraDisable ... 121
CameraEnable ... 121
CameraGetStatus ... 122
SEM GUI Control ... 124
GUIGetScanning ... 124
GUISetScanning ... 124
Progress Indication ... 126
ProgressHide ... 126
ProgressPerc ... 126
ProgressShow ... 127
ProgressText ... 128
Power Interface ... 129
PowerStateEnum ... 129
PowerStateGet ... 130
PowerStateSet ... 130
Sample Stage Layout ... 132
SmplGetCount ... 132
SmplEnum ... 132
SmplGetHldrName ... 134
SmplGetId ... 134
SmplGetLabel ... 135
SmplGetPosition ... 136
SmplGetShape ... 136
SmplGetType ... 137
SmplSetLabel ... 138
Miscellaneous ... 139
ChamberLed ... 139
Delay ... 139
GetDeviceParams ... 140
GetUPSStatus ... 140
IsBusy ... 141
IsLicenseValid ... 142

Preface

All TESCAN Scanning Electron Microscopes (SEMs) are equipped with remote control capability. It can be used by remote applications like an EDX system, or lithography system or other custom application.

The remote control interface is referred as SharkSEM Remote Control.

There are two protocol generations:

- 1.x.x – SEM models produced in the years 2005 – 2010 (G2)
- 2.x.x – SEM models produced in 2010 and later (G3)

The two protocol revisions are compatible. The main difference is that some functions are not supported in both generations, some other functions are slightly different.

Overview

The microscopes are controlled locally from dedicated PC using special applications provided by TESCAN as a part of standard microscope delivery. However, there are many situations when SEM should be operated remotely (across internet, from a different computer, etc.).

Typical situations when remote control can be used:

- Servicing the device
- Interfacing with EDX, WDX, EBSD systems
- Non-standard extension applications
- Training

To enable remote applications to control the SEM, each modern TESCAN SEM is equipped with SharkSEM Remote Control Interface. This is a purely software interface and it is a part of the PC software, which is the main software controlling the microscope.

There are three electron beam gun types: **field emission**, **LaB_6**, **tungsten** (thermal emission). The Vega 3 system is tungsten (optionally LaB_6) based, all Mira 3 systems have field emission guns. Tungsten FIB is called Vela 3, field emission FIB is called Lyra 3. Fera 3 FIB uses plasma ion source and field emission electron source.

To control the SEM remotely, it is necessary to write a special application. The application should strictly follow the rules declared in this document.

Very narrow subset of remote control functions is also available via special Windows DLL. Please contact TESCAN, a.s. for more details.

The communication protocol is improved continuously. TESCAN always tries to keep the protocol backward compatible, however, 100% compatibility of different releases is not guaranteed.

TCP/IP Protocol

To transport commands and image data between the remote application and the SEM, TCP/IP protocol is used. Currently IPv4 only is supported.

It is necessary to establish one or two TCP/IP connections from remote system to the SEM, there should be direct network connectivity between the two computers. The connections are always established from client to well-known server ports, SEM behaves as a **server** and remote application behaves as a **client**. In the following text, “client” means any remote application and “server” means the SEM main control application.
If direct network connection between the two systems is not available, NAT can be used.

The first connection is called **control connection** and all commands and their return values are passed through this connection.

The second connection is optional. Its purpose is to transfer the image data, so it is called **data connection**. Image data is sent through an independent connection because it helps to minimize the communication latency over the control connection.

Concurrent Clients

The SEM server can serve several clients connected at the same time. To prevent interference among the clients connected, they must synchronize write access to the microscope. If a client wants to change any microscope parameter, it should enter "exclusive mode" of operation first. If the exclusive lock is acquired by a client, the other clients are not allowed to change any microscope parameter.

It is also necessary to acquire the exclusive lock if a client wants to control the image scanning, because there is only one image scanning and acquisition unit.

Functions

Basically there are two sets of functions:

- Core functions
- Extension functions

Core functions are a standard part of each SEM. These functions are described in this document.

Sample functions:

- Set working distance
- Set brightness and contrast
- Move the stage
- Set probe current

Extension functions are available only if the microscope is equipped with an extra hardware parts. These functions are not covered by this document.

For example, FIB Lyra systems are equipped with focused ion beam column. The ion column can be controlled as well, using a set of commands described in the *SharkRemote-2-0-Fib.pdf* document.

Number of functions can be virtually unlimited. Each function is identified by a unique self-explaining string, allowing easy debugging of the client application.

Protocol design

Elementary piece of information sent between the client and the server is called **message**. Sometimes it is referred as **function** or **command**.

Each message consists of **header** and **body**. Header size is fixed, body size is variable. Header contains several mandatory fields like name of the function, body size, control flags, etc. In the body, there is encoded list of arguments passed to the function.

The server does not send any piece of information unless asked by the client. It means that currently there is no notification engine which would tell the client that some event has occurred.

Command Processing

There are several different approaches how the client can be designed in terms of the client – server interface. There is only one execution unit inside the server, but these programming models allow user to design the client application in a simple and straightforward way or a complex powerful way, depending on requirements.

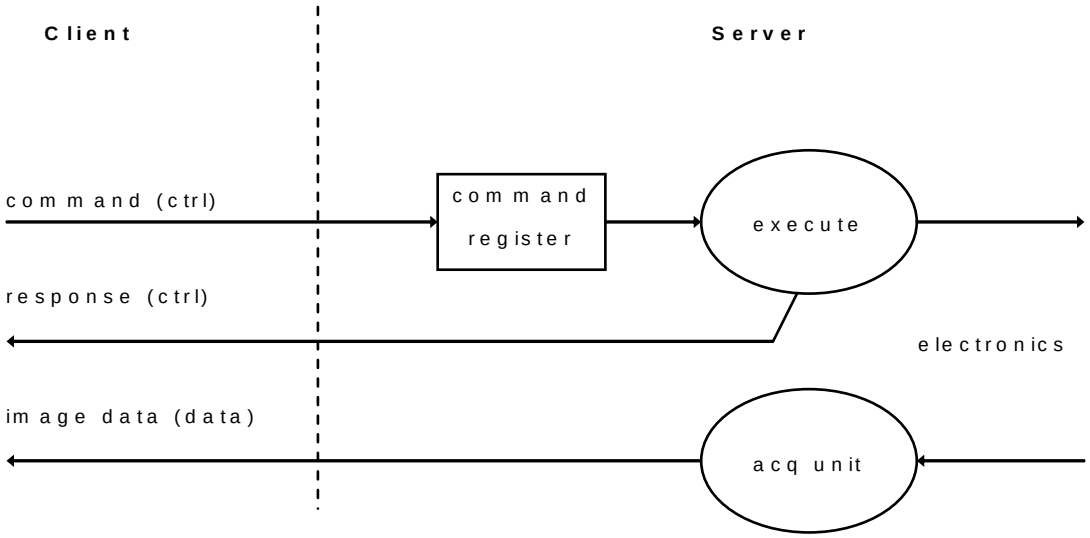
Three models are described:

- Single command
- Single queue
- Multiple queue

Note: for most applications, "single command" mode is the best choice.

Single Command

There is either none or just one command issued by the client at the same time. The server is either idle or processing the only one active command.



There are two TCP/IP connections - bidirectional control connection marked **ctrl** and unidirectional connection marked **data**.

"Send response" flag in the command header is required. See below for detailed description of the flags.

Next command must not be issued until the response from the first command is received.

It is impossible to cancel the command once sent.

The Single Command programming model is very easy to understand and it is not complicated to program the client application. Command execution is fully synchronous, that is, commands are executed one after another.

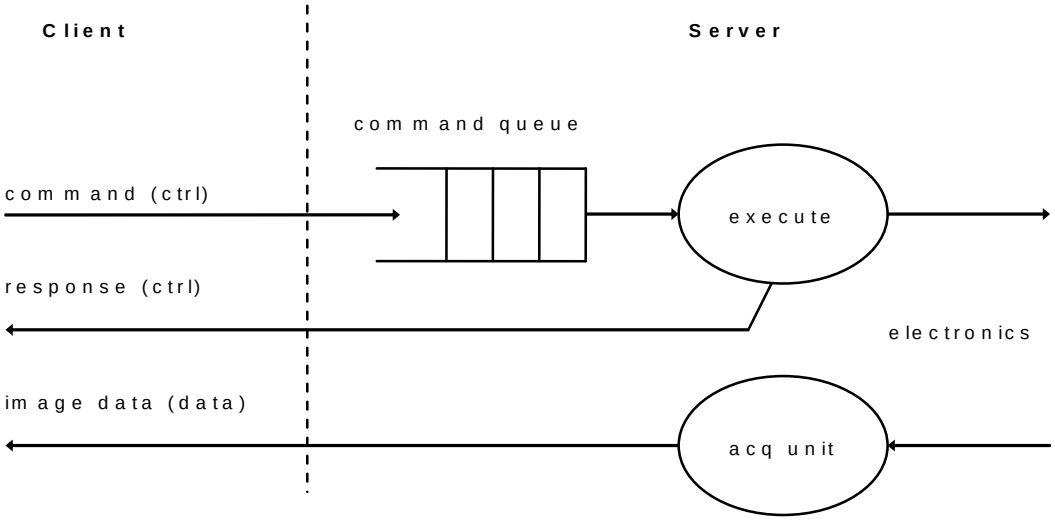
Image acquisition can be done even using this simple model. There are commands for starting and stopping the image scanning. “Start Scanning” command is executed immediately when it is received by the server. As a result, server starts to send continuous series of messages through the data channel. These messages contain image data in their body.

In the meantime (while the scanning is active), other commands may be sent to the server. This allows the client to adjust the conditions under which image is acquired during the acquisition.

Single Queue

This is more powerful solution. It allows the client application to minimize the communication latency.

There is a server-side FIFO command queue. It is possible to issue several command and store them in the server queue. The commands are processed by the server one after another. When a command is executed, response can be sent to let the client know which command has just been finished.



The client must be able to distinguish which command has been executed. To allow that, there is an optional 32-bit value which is sent back in response and which can identify the command.

It is also possible to remove the commands from the queue.

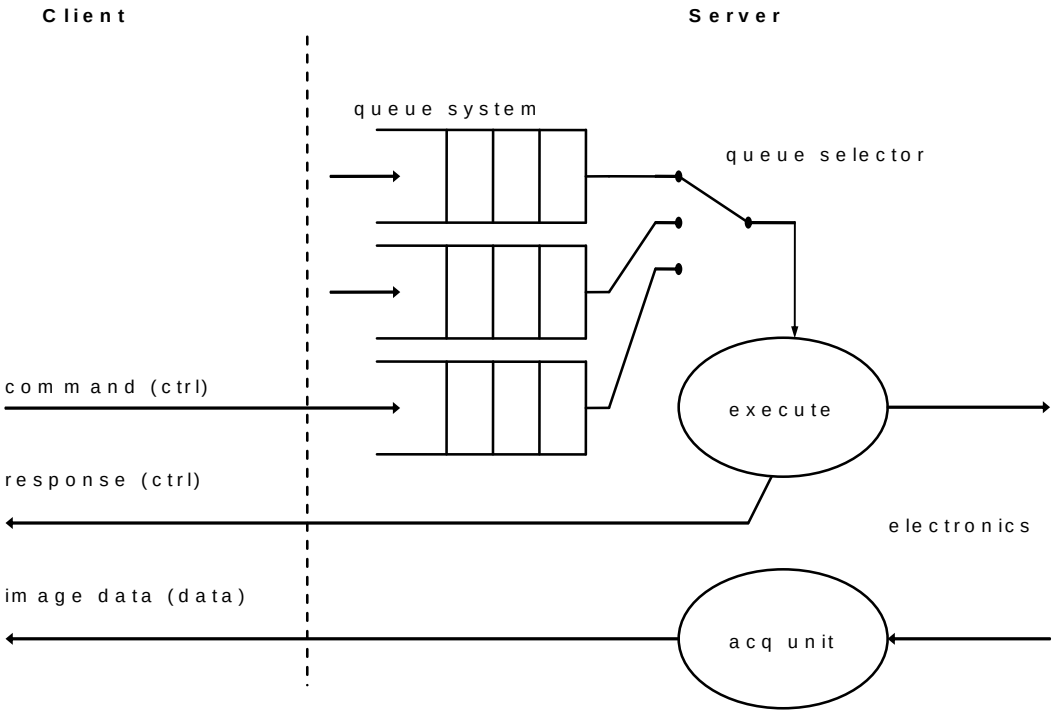
”Send response” flag in the command header is optional.

When command is issued, it is possible to either put it in the end of the queue or to put it into the head of the queue. This is useful if several commands in the queue shall be removed, in this case it is necessary to jump the queue. Another example could be polling the status of the microscope while there is a series of commands pending in the queue.

Multiple Queues

This is similar to the Single Queue model. There are several queues, each of them containing several commands.

All the Single Queue features are also valid for Multiple Queue model. Moreover, it is possible to manage the set of queues, change the active queue, etc. The active queue is selected by a special command.



To be clarified: how the active queue is switched. Next release of this documentation will address this issue.

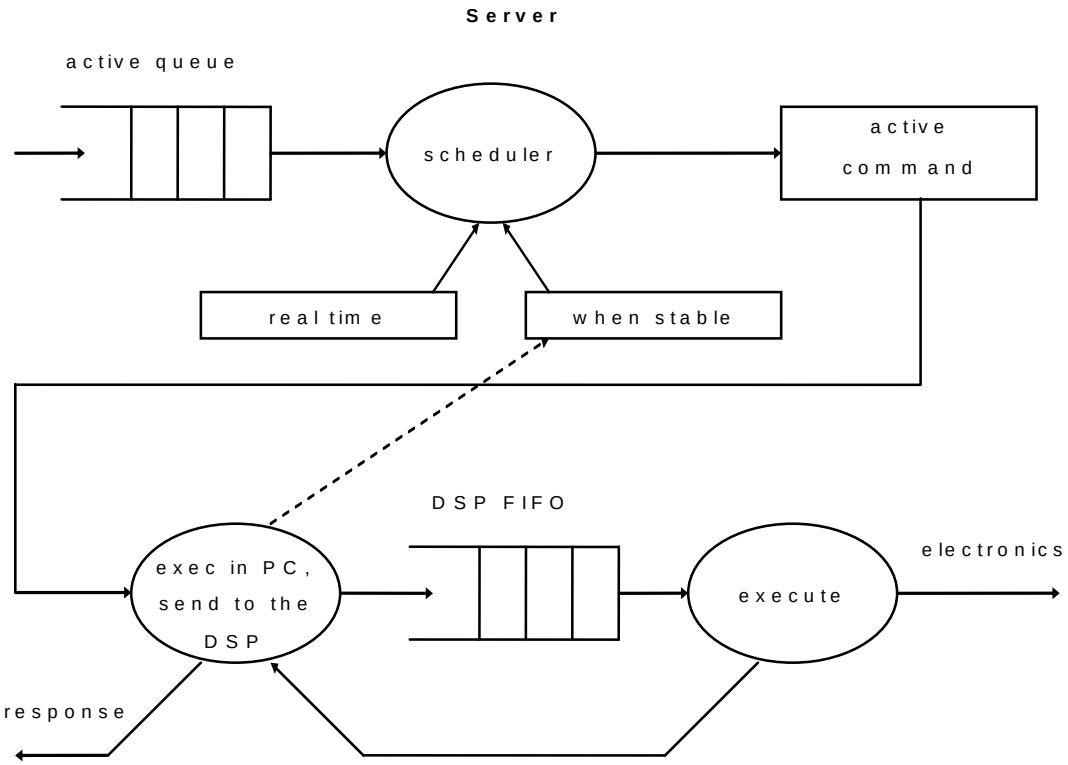
Internal Command Processing

In fact, the execution of command is not as simple as described above. There is a scheduler that controls whether the input command can be scheduled for execution or not.

The scheduler is controlled by “Wait” command flags. These tell the microscope whether it has to wait until all transient effects have disappeared. For example, if working distance is

changed, it takes an arbitrary time until the lens focus is stabilized. If image acquisition command is scheduled for execution after such a slow function, it can be marked as “Wait” command so that it is guaranteed that the system is stable at the moment of execution.

Currently there are four SEM Wait flags – Wait A, Wait B, Wait C and Wait D. FIB systems define three additional Wait flags – Wait E, Wait F, Wait G. Different waiting condition is related with each of them. See *Message Header* for details.



Since there is a DSP controller inside the microscope, the execution of the command is done in two phases. First, it is processed by the server on the PC. As a result, the command is split into several elementary commands for the DSP and then processed. For example, if working distance is set, it is necessary to compute the focus of several lenses in the optics and then send several commands to the DSP board. Finally, the DSP board will process the commands in a serial manner.

Client is not able to influence the commands already scheduled for execution.

Message Structure

In this chapter, we describe the layout of the command header and body.

Scalar Data Types

There are six fundamental data types which can appear in the message header and message body. The types are similar to C language data types.

char
This is an 8-bit wide signed integer. Its range is -128 to 127.

unsigned char
This is an 8-bit wide unsigned integer. Its range is 0 to +255.

short
This is a 16-bit wide signed integer. Its range is -32768 to 32767.

unsigned short
This is a 16-bit wide unsigned integer. Its range is 0 to 65535.

int
This is a 32-bit wide signed integer. Its range is -2147483648 to 2147483647.

unsigned int
This is a 32-bit wide unsigned integer. Its range is 0 to 4294967295.

void
This not a real type, it simply says that there is nothing.

Arrays

All six scalar types can be stored in a variable sized array. For example, **char[]** is an array of characters of variable length. The array size is unlimited, the actual size is sent together with the data. See below for details.

Sometimes **string** is mentioned in the documentation. **String** is a **char[]** array, which contains ASCII characters >=1 and <= 255, the last character is always zero.

Compound Types

There are currently two compound types – **float** and **map**.

float
Floating point type is implemented using zero terminated **char[],** that is, **string**. **Float** values are sent as a string, for example “3.1415e-3” is a valid floating point number. The syntax is equal to syntax recognized by C-language *strtod()* function.

map
Map is an unordered set of (*key*, *value*) pairs. It is encoded as a **string**, each pair on a separate line. The *key* and the *value* are separated by '=' character. *Key* consists of letters, digits, underscores and dots. *Value* may consist of any ASCII characters >= 32. *Key* should be considered as a unique case-sensitive identifier.

Example:

```
speed.1=1.15
speed.2=4.0
speed.3=12
```

The **map** type is very flexible. On the other hand, it is rather complex. It is used mostly for configuration purposes.

Storage Rules

If a string is used, it is always zero terminated.

Storage method is little-endian, that is, least significant byte is stored at the lowest address.

Inside the body of the message, data alignment takes place as described below. The header does not need any alignment.

Message Header

Header is the same for both request and response.

Request is a message sent by a client. *Response* is a message sent by the server.

```
struct tMessageHeader
{
     char               Cmd[16];          // :00
     unsigned int       BodySize;         // :16
     unsigned int       Id;               // :20
     unsigned short     Flags;            // :24
     unsigned short     Queue;            // :26
     char               Resvd[4];         // :28
};
```

Address	Type	Size in bytes	Value
00	char[]	16	Command name
16	unsigned int	4	Body size
20	unsigned int	4	Identification
24	unsigned short	2	Flags
26	unsigned short	2	Queue
28	char[]	4	Reserved

Total size of the header is 32 bytes.

Command name
Unique command name. Command name is a case sensitive, null terminated string. Maximum number of characters is 15, the last character must be followed by 0x00 null terminator. The command name is the same for both request and response.

Body size
Size of the message body in bytes, including the padding (if any).

Identification
This is an optional identification of the command. Server copies the Id from request to response. The Id is also required when message is removed from a queue, the delete call requires Id array as an argument.

Flags
There are several flags that control the message handling.

Request flags

Bit	Value	
0	Send response	
1	Jump	
2-7	Reserved, always 0	
8	Wait (SEM)	A – e-beam scanning
9		B – stage
10		C – e-beam optics
11		D – e-beam automatic procedure
12	Wait (FIB)	E – i-beam scanning
13		F – i-beam optics
14		G – i-beam automatic procedure
15	Reserved, always 0	

Response flags

Bit	Value
0-15	Reserved, always 0

Send response – if set, response is always generated when command is executed. If not set, response is generated only if there is a return value or output parameter.

Jump – if set, the command is put into the head of the FIFO queue. If not set, the command is queued normally, at the end of the FIFO queue.

Wait flags are combined using logical OR. If Wait B and Wait C are set, command is executed when stage has stopped and the electron optics is stable.

Wait A – e-beam scanning – if set, the command is executed only if electron beam scanning is inactive. Usually used with *ScStopScan()* call in the single = 1 mode.

Wait B – specimen stage or detector movement – if set, the command is executed only if stage is idle and all detector travels are idle.

Wait C – e-beam optics – if set, the command is executed only if electron optics is in stable condition.

Wait D – e-beam automatic procedure – if set, the command is executed only if no electron beam automatic procedure is in progress.

Wait E – i-beam scanning – if set, the command is executed only if ion beam scanning is inactive.

Wait F – i-beam optics – if set, the command is executed only if ion optics is in stable condition.

Wait G – i-beam automatic procedure - if set, the command is executed only if no ion beam automatic procedure is in progress.

All reserved bits must be set to zero.

Since protocol version 2.0.9, the wait flags can be also queried explicitly by *IsBusy()* function.

Queue
If no queue or single queue is used, it must be set to zero (queue 0).

If multiple queues are used, this field contains identification of the queue where the command will reside.

Client must always define the queue. If a response is sent, the queue identification is set to the same value as in the request.

Reserved
Set to zero in both request and response.

Message body

Each message is defined by its *function prototype*.

Example

void GetWD(out float wd)

This says: *GetWD* function has no input arguments, no return value and one string as an output value. That is, 32 bytes (header only) are sent in the request and 32 + 4 + N + padding bytes are sent as a response. 4 is the size of a field which contains size of the string, N is number of characters in the string including the terminating null character, whole message is padded to fit the argument size into multiple of four (4-padded).

Return value
Return value is handled the same way as if it was the first output argument.

Input / output type modifier
Each argument is prefixed with modifier which tells the way argument is passed.
in – sent from client to server
out – sent from server to client
inout – sent both ways

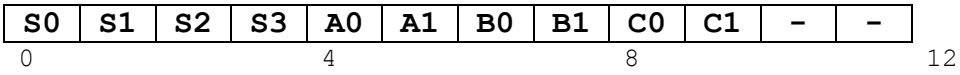
Argument order
Arguments are encoded from left to right, starting with the first parameter if request is sent and with return value if response is sent. First, header is sent, then, all arguments are encoded into the message body one after another.

Argument encoding
If argument of an integral type (fundamental type) is used, it is encoded into the body and 4-padded. That is, integral types always occupy 4 bytes in the message body.

If the argument is an array, first *unsigned int* contains number of bytes in it. This is followed by the array itself, the array is 4-padded at the end. There might be even empty array, zero is valid number of elements. Elements within the array are not padded.

Example

Let's say there is an array of three 16-bit short values. The array is encoded into 12 bytes. First, there is the 32-bit wide array size (**S**). Immediately follows the array itself (elements **A**, **B**, **C**). Since the total size is 10 bytes, 2 bytes of padding are added.



Example

void DtGetGainBlack(in int detector, out float gain, out float black);

Message body sent by the **client**:

Address	Type	Size in bytes	Value
00	Int	4	detector

The size of the message body is 4 bytes.

Message body sent by the **server**:

Address	Type	Size in bytes	Value
00	unsigned int	4	gain (size = 5)
04	char[]	5	gain ("3.14")
09	char[]	3	gain (padding)
12	unsigned int	4	black (size = 4)

16	char[]	4	black (“0.0”)
20	char[]	0	black (padding)

The size of the message body is 20 bytes.

Session Management

First of all, control connection between the client and the server is established. Client must initiate the process by calling TCP *connect()* function. If the server is ready, connection request is accepted immediately.

The default control connection server port is 8300. This can be changed if the port is already used by another application.

Once connected, client may want to check the version of Remote Control Interface by calling *GetVersion()* function. If it is not compatible with the version reported, it must close the connection and notify the operator.

Then, client may want to establish data connection. This is an optional step, if the client does not need to read image data from the server, is may skip it. The data connection **server port** is control connection server port + 1, which is 8301 by default. The data connection **client port** number must be known at the server side prior to establishing the data connection. Server requires this value to be able to match the control and the data connection.

Note: *this has been extended since 1.0.8. See "firewall-friendly mode" described below.*

A special function is provided to pass the client-side data connection port number to the server. Refer to ***TcpRegDataPort*** for details.

Server TCP ports (default setting)

Connection	Port
Control	8300
Data	8301

Then, authentication might be required. This is not supported yet.

Now, client is ready for microscope control. The connection may remain established as long as required. There is no limitation how the client is designed, it must use the functions according to the specification.

If the client calls unexpected function, or if the function arguments are wrong, or if the message is incorrect, it is silently ignored.

The client must read all the data sent by the server. If there is outgoing data pending and server is not able to send them to the client, the client is disconnected by the server. This does not happen immediately, there is approximately 10 s timeout. The exact time depends on the server send buffer size and amount of data produced by the server.

After the client has finished its operation, it must first close the data connection and then close the control connection. If there are some resources associated with the client connection (pending commands, locks acquired, etc.), they are automatically released when the control connection is closed.

Client must properly handle situation when connection is lost. The recommended behavior is to close the connections and tell the user that connection to the SEM was lost.

Without image transfer

Step	Client task
1	Establish control connection (port 8300)
2	Check interface version
3	Authentication
4	Control the SEM, there can be many commands
5	Close control connection

Image transfer required

Step	Client task
1	Establish control connection (port 8300)
2	Check interface version
3	Create data connection descriptor, bind to a local port
4	Call TcpRegDataPort
5	Establish data connection (port 8301)
6	Authentication
7	Control the SEM, there can be many commands
8	Close data connection
9	Close control connection

As described above, the data connection requires special procedure when setting up. Also note that it is important to wait until TcpRegDataPort() call is completed, the server needs to know the client data connection port number before the connection request is issued.

Refer to socket(), connect(), bind(), getsockname() socket functions for more information on how to the data connection.

Firewall-friendly mode
Since the above method of establishing data connection is often blocked by the firewalls, there is an alternative approach which is more reliable.

The main difference is that only single control + data connection is accepted by the server, and thus the server does not need to distinguish between the clients. In this case, *TcpRegDataPort()* function is ignored.

A good practice is still to use the *TcpRegDataPort()* function, and in case of complex network configurations to put the server into single client (firewall friendly) mode.

See *TcpRegDataPort()*for more info.

Communication

Following commands control the TCP communication channel between the client and the server.

TcpGetDevice

Read the microscope device serial number.

Arguments

char[] ***TcpGetDevice****(void)*

return value device S/N

Timing

Executed immediately.

Remarks

TcpGetDevice() example result: "VG1234567US"

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.3 and later

TcpGetSWVersion

Read the SEM software version.

Arguments

char[] ***TcpGetSWVersion****(void)*

return value version string

Timing

Executed immediately.

Remarks

This function reports the software version of remote (SEM) application. Example response is "4.1.19.0".

Should not be confused with *TcpGetVersion()*. *TcpGetVersion()* returns the revision of the SharkSEM communication interface and its function set. *TcpGetSWVersion()* reports the version of remote software. Such information may serve for diagnostic purposes.

The recommended usage is that the client SW first checks the *TcpGetVersion()*, if the version is >= 2.0.9, SEM SW version can be determined using *TcpGetSWVersion()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.9 and later

TcpGetVersion

Read the protocol implementation version.

Arguments

char[] ***TcpGetVersion****(void)*

return value version string

Timing

Executed immediately.

Remarks

Most basic SharkSEM functions are available in all protocol versions. However, the protocol has been growing and new functions appeared.

Beginning with version 1.0.5, it is possible to check the current protocol version by calling this function. If the client plans to call functions 1.0.5 or later, it is recommended to call *TcpGetVersion()* at the beginning of the session.

The version string consists of three decimal numbers separated by dots. Following version strings are valid examples:

"1.0.5"
"1.0.129"
"2.0.0"

Since the *TcpGetVersion()* was not present in the original version of the protocol, the servers 1.0.4 and older do not respond, because the function is not implemented. The client must properly handle the response timeout.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

TcpRegDataPort

Set the originating (source) TCP port for data connection.

Arguments

*int **TcpRegDataPort**(in unsigned int port)*

port	client TCP port number of the data connection
return value	0

Timing

Executed immediately.

Remarks

First, control connection is established. If the client wants to transfer an image, it must also establish TCP data connection.

When the client tries to establish the data connection, the server must be able to match the data connection with the control connection. There are two ways how to achieve that.

Classic approach - registering the source TCP port

Prior to establishing the data connection, server must know the data connection originating (client) port. The port number is transferred from client to server using this function.

The procedure is:

- Establish control connection (server port 8300)
- Bind data connection to local port
- Read back the local port number
- Call *TcpRegDataPort()*
- Establish data connection

The return value is provided to the client to make sure that the server has processed the request and the client may try to establish the data connection.

Firewall-friendly approach
The classic approach has one significant disadvantage. It can hardly pass the firewalls, because most firewalls alter the connection source port.

Since version 1.0.8, the server can be configured the way that it accepts just single connection (or better to say, single pair of control connection and data connection). If the server is configured this way, it is not necessary to call the *TcpRegDataPort()* anymore, because the server matches the data connection and the control connection automatically (because there is only one control connection).

The advantage is that such method easily traverses firewalls, but, on the other hand, it is not possible to use more than one client at the same time.
See also *Session Management* article.

Call Context

Must be called before establishing the data connection.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Electron Optics

This set of functions controls how the electron optics is set up. The most important parameters are scanning mode, magnification, probe current and working distance.

AutoColumn

Start automatic column alignment procedure. Depending on the scanning mode, OBJ Centering and IML Centering are adjusted.

Arguments

*void **AutoColumn**(in int channel)*

channel	input video channel

Timing

Execution time may vary between 10 and 60 seconds (wait D).

Remarks

Channel index is a zero based index of the input channel. Appropriate detector must be selected as an active.

There are several important preconditions. If they are not fulfilled, the column centering may fail.

- Good signal quality
- Good structure on the specimen surface
- Well focused image

As a result, objective centering value is changed. Unfortunately, it is not possible to automatically verify the quality of the automatic alignment result.

See also *DtSelect(), AutoWD(), AutoSignal().*

Call Context

Scanning must be inactive.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

AutoGun

Start automatic gun alignment procedure.

Arguments

*void **AutoGun**(in int channel)*

channel	input video channel

Timing

Execution time may vary between 10 and 30 seconds (wait D).

Remarks

Channel index is a zero based index of the input channel. Appropriate detector must be selected as an active.

Especially the tungsten filament is worn during the SEM operation. It may be useful to re-center the gun if the probe current decreases.

As a result, the electronic gun shift and the electronic gun tilt values are changed.

See also *DtSelect(), AutoSignal()*.

Call Context

Scanning must be inactive.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

AutoWD

Start automatic working distance (WD) adjustment procedure.

Arguments

*void **AutoWD**(in int channel [, in float min_wd, in_float max_wd])*

channel	input video channel
min_wd	minimum allowed WD
max_wd	maximum allowed WD

Timing

Execution time may vary between 10 and 60 seconds (wait D).

Remarks

Channel index is a zero based index of the input channel. Appropriate detector must be selected as an active first, the automatic procedure requires good signal quality and suitable specimen to achieve good result.

If *min_wd* and *max_wd* are specified, the search range is restricted. In some cases, this improves the algorithm reliability.

As a result, WD value is changed.

See also *DtSelect()*.

Call Context

Scanning must be inactive.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Min_wd and *max_wd* parameters are optional, they have been added in SharkSEM v. 2.0.12.

Degauss

Degauss the microscope column.

Arguments

*void **Degauss**(void)*

Timing

Synchronous execution. No command is executed until *Degauss()* is completed. Takes several seconds to finish.

Remarks

During the degauss procedure, all electromagnetic coils in the microscope column are fed with alternating current with decreasing amplitude. The aim is to minimize the magnetic remanence.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

EnumCenterings

Get list of all column centering parameters.

Arguments

map ***EnumCenterings****(void)*

return value list of centering parameters

Timing

Executed immediately.

Remarks

The *map* has the following form:

```
cen.0.name
cen.0.count
cen.0.unit
cen.1.name
…
cen.X.name
cen.X.count
cen.X.unit
```

The key is separated by two dots. The middle number is an index of the centering parameter. Application should not expect that the indexes form a consecutive sequence. The index can be any integer number.

name	human-readable name.
count	1 or 2 (one if only the first value is used, two otherwise)
unit	%, mm, or other physical unit

The exact set of centering parameters depends on the SEM configuration.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

See also *GetCentering(), SetCentering()*

Call Context

Anytime

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

EnumGeometries

Get list of all available scanning transformations.

Arguments

map ***EnumGeometries****(void)*

return value list of geometric transformations

Timing

Executed immediately.

Remarks

The *map* has the following form:

```
geom.0.name
geom.0.count
geom.0.unit
geom.1.name
…
geom.X.name
geom.X.count
geom.X.unit
```

The key is separated by two dots. The middle number is an index of the parameter. Application should not expect that the indexes form a consecutive sequence. The index can be any integer number.

name	human-readable name.
count	1 or 2 (one if only the first value is used, two otherwise)
unit	%, mm, deg, or other physical unit

The standard transformations are beam shift, electronic rotation, beam tilt, tilt correction, dynamic focus. The exact set of geometric transformations depends on the SEM configuration.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

See also *GetGeometry(), SetGeometry()*

Call Context

Anytime

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

EnumPCIndexes

Get list of all PC indexes.

Note: *PC (Probe Current)* is an old name for *BI (Beam Index).* For backward compatibility, remote control still uses the *PC* parameter. The *PC* and *BI* range is usually 1 to 20. The higher is *BI*, the higher is the electron beam current. *PC* uses inverse logic.

Arguments

*map **EnumPCIndexes**(void)*

return value	list of PC indexes

Timing

Executed immediately.

Remarks

The *map* has the following form:

```
pc.1.current
pc.2.current
…
pc.N.current
```

1 is the minimum index, **N** is the maximum index. The list of indexes is a continuous sequence and the corresponding values are increasing or decreasing as well. Currently there are 20 indexes (1 to 20), but this may change in the future.

current is the estimated probe current in pA (picoamps). Application should not expect that the value is accurate, because it depends on the acceleration voltage, gun centering, and other parameters.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

Get3DBeam

Get electronic beam tilt.

Arguments

*void **Get3DBeam**(out float alpha, out float beta)*

alpha	tilt angle along the X scanning axis [deg]
beta	tilt angle along the Y scanning axis [deg]

Timing

Executed immediately.

Remarks

3D Beam is the angle between the optical axis of the column and the beam scanning on the specimen surface.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

GetBeamCurrent

Get calculated beam current.

Arguments

*float **GetBeamCurrent**(void)*

return value	beam current [pA]

Timing

Executed immediately.

Remarks

Calculated SEM beam current. Refer to *SetBeamCurrent()* for more details.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.19 and later

GetCentering

Get column centering parameter.

Arguments

*void **GetCentering**(in int index, out float x, out float y)*

index	index of the parameter – see *EnumCenterings()*
x, y	centering values in X and Y axis

Timing

Executed immediately.

Remarks

Note: it is usually not necessary to read/write the centering from the remote application.

This function returns one or two values (the **y** parameter may not be used in some cases).

The SEM column centering depends on:
1) accelerating voltage
2) scanning mode
3) active detector

Each accelerating voltage range (index) keeps its own centering set. Each scanning mode keeps its centering set as well. Finally, there are some detectors (In-beam) which have an effect on centering as well and must keep an extra set.

See also *EnumCenterings(), SetCentering()*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

GetGeometry

Get image (scanning) geometry parameter.

Arguments

*void **GetGeometry**(in int index, out float x, out float y)*

index	index of the parameter – see *EnumGeometries()*
x, y	one or two values

Timing

Executed immediately.

Remarks

Note: it is usually not necessary to read or adjust the image geometry from the remote application.

This function returns one or two values (the **y** parameter may not be used in some cases).

Unlike the centering, the image geometry does not depend on the accelerating voltage or the scanning mode. There are several limitations, for example it is not possible to use dynamic focus in the FIELD mode.

See also *EnumGeometries(), SetGeometry(), GetGeomLimits()*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

GetGeomLimits

Get min/max limits of *GetGeometry(), SetGeometry()* microscope parameters.

Arguments

*int **GetGeomLimits**(in int index, out float x_min, out float x_max, out float y_min, out float y_max)*

index	index of the parameter – see *EnumGeometries()*
x_min, x_max	limits of the first parameter
y_min, y_max	limits of the second parameter
return value	0 – ok, < 0 – unable to determine the limits

Timing

Executed immediately.

Remarks

There are image geometry parameters either with known or unknown limits. If known, the limits often depend on the actual SEM operating conditions. Examples of known limits are beam tilt or image rotation. Contrary, example of parameter with unknown limits is image shift.

See also *GetGeometry(), SetGeometry()*

Call Context
Anytime

Also Affected
-

Compatibility

1.x.x	2.x.x
no	2.0.22 and later

GetIAbsorbed

Read specimen current.

Arguments
*float **GetIAbsorbed**(void)*

return value	absorbed current in [pA]

Timing
Executed immediately.

Remarks
-

Call Context
Anytime

Compatibility

1.x.x	2.x.x
yes	yes

GetImageShift

Get electronic image shift.

Arguments
*void **GetImageShift**(out float x, out float y)*

x	x shift [mm]
y	y shift [mm]

Timing
Executed immediately.

Remarks

Image Shift is done in the scanning coils of the SEM.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

GetPCContinual

Get probe current (beam current).

Arguments

*float **GetPCContinual**(void)*

return value	PC value (usually 1.0 to 20.0)

Timing

Executed immediately.

Remarks

Refer to *SetPCContinual()*.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.2 and later

GetPCFine

Get probe current fine adjustment. **This function is obsolete**.

Arguments

*float **GetPCFine**(void)*

return value	PC fine value [%]

Timing

Executed immediately.

Remarks
See *SetPCFine()*.

Call Context
Anytime

Compatibility
Provided for backward compatibility only. Please use *GetPCContinual()* instead.

1.x.x	2.x.x
yes	yes - obsolete

GetPCIndex

Get probe current index.

Arguments
*int **GetPCIndex**(void)*

return value PC index

Timing
Executed immediately.

Remarks
Refer to *SetPCIndex()*.

Call Context
Anytime

Compatibility

1.x.x	2.x.x
yes	yes

GetSpotSize

Get electron beam spot size.

Arguments
*float **GetSpotSize**(void)*

return value spot size in [nm]

Timing

Executed immediately.

Remarks

The electron beam spot size is a calculated value. It depends on the scanning mode, working distance, probe current and other parameters. The spot size is twice the microscope resolution (under given conditions).

If the spot size is exactly zero (0.0), it is not defined. This will happen when the spot size cannot be calculated.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
1.0.8 and later	yes

GetViewField

Get current view field.

Arguments

*float **GetViewField**(void)*

return value view field in [mm]

Timing

Executed immediately.

Remarks

View field is the width of scanning window. This is especially important if scanning window shape is different than square.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

GetWD

Get current working distance.

Arguments

*float **GetWD**(void)*

return value	working distance in [mm]

Timing

Executed immediately.

Remarks

Working Distance is a distance between the end of the objective and the surface of the observed specimen. If *Working Distance* is set properly, image should be sharp.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Set3DBeam

Set electronic beam tilt.

Arguments

*void **Set3DBeam**(in float alpha, in float beta)*

alpha	tilt angle along the X scanning axis [deg]
beta	tilt angle along the Y scanning axis [deg]

Timing

Executed immediately.

Remarks

3D Beam is the angle between the optical axis of the column and the beam scanning on the specimen surface. It is usually only a small value (several degrees). The shorter the working distance is, the higher the beam tilt range is. If the required value is too high, it is silently limited.

Some detectors (eg. In-beam) may reset the beam tilt value when selected. Make sure that the *DtSelect()* is called before *Set3DBeam()* function.

Call Context

Available in the *Resolution* or *Depth* scanning modes only.

Also Affected

Setting the 3D Tilt resets the image shift to zero.

Compatibility

1.x.x	2.x.x
yes	yes

SetBeamCurrent

Calculate and set SEM beam current.

Arguments

*void **SetBeamCurrent**(in float beam_current)*

beam_current	beam current [pA]

Timing

Several hundreds of milliseconds until the system becomes stable (wait C).

Remarks

This is an alternative to *SetPCContinual()*. Unlike it, *SetBeamCurrent()* does all necessary calculation and sets the beam intensity according to the required beam current. Consequently, Beam Intensity (BI) value is changed.

Schottky FEG SEMs have good accuracy of *SetBeamCurrent()*. When maintained in a good condition, typically better than 10%. Tungsten SEMs have significantly worse accuracy, because instead exact calculation, there can be only simple mathematical fit.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.19 and later

SetCentering

Set column centering parameter.

Arguments

*void **SetCentering**(in int index, in float x, in float y)*

index	index of the parameter – see *EnumCenterings()*
x, y	centering values in X and Y axis

Timing

Typically 5 ms to finish (wait C).

Remarks

Note: it is usually not necessary to read or adjust the centering from the remote application.

If the values are out of range, they are limited.

See also *EnumCenterings(), GetCentering()*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

SetGeometry

Set image (scanning) geometry parameter.

Arguments

*void **SetGeometry**(in int index, in float x, in float y)*

index	index of the parameter – see *EnumGeometries()*
x, y	geometry values

Timing

Typically 5 ms to finish (wait C).

Remarks

Note: it is usually not necessary to read or adjust the image geometry from the remote application.

If the values are out of range, they are limited.

See also *EnumGeometries(), GetGeometry()*

Call Context

Anytime

Also Affected

Some parameters are related, for example Image Shift will change the 3D Beam value.

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

SetImageShift

Set electronic image shift.

Arguments

*void **SetImageShift**(in float x, in float y)*

x	x shift [mm]
y	y shift [mm]

Timing

Executed immediately.

Remarks

Image Shift is done in the scanning coils of the SEM. The maximum range depends on the scanning mode and other conditions, and thus it is not guaranteed. If the shift value is too high, it is silently limited. It is recommended to read back the image shift value after the change and check whether the required value was accepted.

Call Context

Anytime

Also Affected

Setting the image shift resets the 3D Tilt value to zero.

Compatibility

1.x.x	2.x.x
yes	yes

SetPCContinual

Set probe current (beam current).

Arguments

*void **SetPCContinual**(in float pc)*

pc	PC value (usually 1.0 to 20.0)

Timing

Several hundreds of milliseconds until the system becomes stable (wait C).

Remarks

PC Continual is an extension of PC Index concept. Instead of integer value, floating point value capable of continuous beam current control is accessible. The value is typically between 1.0 and 20.0, 1.0 is the highest current and 20.0 is the lowest one. More information can be received by calling *EnumPCIndexes()* function.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.2 and later

SetPCFine

Set probe current fine adjustment. **This function is obsolete**.

Arguments

*void **SetPCFine**(in float pcfine)*

pcfine PC fine [%]

Timing

Several hundreds of milliseconds until the system becomes stable (wait C).

Remarks

Probe Current Fine is a value in the range -5 % to + 5.0 %. If the value is equal to zero percent, the beam current is defined by the current PC index. If the value is greater than zero, the beam current grows higher, if it is less than zero, the probe current is lower. The function may not be linear in the whole +/- 5.0 range.

The *Probe Current Fine* value is associated with each PC index. The Get / Set functions control the *PC Fine* for the currently selected index only.

PC Fine should be used only for special applications (eg. EDS/WDS probe current tuning), for general probe current control *SetPCIndex()* should be used.

Call Context

Anytime

Compatibility

Removed in version 2.0.2 and later. Please use *SetPCContinual()* instead.

1.x.x	2.x.x
yes	2.0.1 and older

SetPCIndex

Set probe current index.

Arguments

*void **SetPCIndex**(in int index)*

index	PC index

Timing

Several hundreds of milliseconds until the system becomes stable (wait C).

Remarks

Probe Current Index is an index into the microscope PC table. Typically the value is between 1 and 20, 1 is the highest current and 20 is the lowest one. More information can be received by calling *EnumPCIndexes()* function.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

SetViewField

Set view field.

Arguments

*void **SetViewField**(in float vf)*

vf	view field in [mm]

Timing

Typically 5 ms to finish (wait C).

Remarks

View Field is subject to limitation. If the required value is out of the limits, it is automatically adjusted.

View Field is a replacement of traditional *Magnification* function. Since *Magnification* is related to the computer display, remote control works rather with *View Field*.

Call Context

Not available in *Fish Eye* scanning mode.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

SetWD

Set working distance.

Arguments

*void **SetWD**(in float wd)*

wd	working distance in [mm]

Timing

Several hundreds of milliseconds until the system becomes stable (wait C).

Remarks

Working Distance is subject to limitation. If the required value is out of the limits, it is automatically adjusted.

Call Context

Not available in *Fish Eye* scanning mode.

Also Affected

View Field may slightly change when *Working Distance* is modified.

Compatibility

1.x.x	2.x.x
yes	yes

Manipulators

There can be one or more manipulators installed in the SEM. In most cases, just single stage for specimen manipulation is available.

ManipGetConfig

Get manipulator configuration.

Arguments

*map **ManipGetConfig**(in int index)*

index	index of the manipulator – see *ManipGetCount()*
return value	manipulator description

Timing

Executed immediately.

Remarks

The *index* is zero based.

The *map* contains following items:

```
manipulator.index
manipulator.name
```

The **name** is human-readable string.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.10 and later	2.0.1 and later

ManipGetCount

Get number of installed manipulators.

Arguments

*int **ManipGetCount**(void)*

return value	1, 2, …	number of manipulators

Timing

Executed immediately.

Remarks

Number of available manipulators is returned. See *ManipGetConfig()* for more info about specific manipulator. On most devices, the result is 1.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.10 and later	2.0.1 and later

ManipGetCurr

Get currently selected manipulator.

Arguments

*int **ManipGetCurr**(void)*

return value	0, 1, …	currently selected manipulator

Timing

Executed immediately.

Remarks

Returns index of the currently selected manipulator.

See *ManipSetCurr()* for details.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.10 and later	2.0.1 and later

ManipSetCurr

Select manipulator.

Arguments

*void **ManipSetCurr**(in int index)*

index	index of the manipulator

Timing

Executed immediately.

Remarks

Defines which manipulator becomes selected. All subsequent stage *StgXxx()* commands refer to this manipulator.

After microscope initialization, manipulator 0 is automatically selected.

See also *ManipGetCurr()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.10 and later	2.0.1 and later

Stage Control

Most microscopes are equipped with motorized stage. Stage can be fully controlled via remote control interface, there are only limitations related to the stage model. For example, some microscopes do not support motorized movement in Z axis.

StgCalibrate

Calibrate the stage.

Arguments

*void **StgCalibrate**(void)*

Timing

The calibration takes up to one minute, depending on the stage configuration. Wait B flag is set while the calibration is in progress.

Remarks

If the calibration is successful, *StgIsCalibrated()* returns 1 after Wait B reset.

Call Context

All stages and detector noses must be idle (Wait B flag must not be set).

Compatibility

1.x.x	2.x.x
yes	yes

StgGetLimits

Get stage limits.

Arguments

*void **StgGetLimits**(in int limit_type, out float x_min, out float x_max, out float y_min, out float y_max, out float z_min, out float z_max, out float r_min, out float r_max, out float t_min, out float t_max)*

limit_type	0 – device limits, 1 – soft limits
x_min, x_max	minimum and maximum x value [mm]
y_min, y_max	minimum and maximum y value [mm]
z_min, z_max	minimum and maximum z value [mm]
r_min, r_max	minimum and maximum rotation value [mm]
t_min, t_max	minimum and maximum tilt value [mm]

Timing

Executed immediately.

Remarks

Device limits specify the stage travel range in arbitrary axis. Soft limits may be more strict that device limits, for example when there is a detector which prevents travel across the whole device range.

If an axis is not motorized, limits are zero (see also *StgGetMorotized()*).

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.22 and later

StgGetMotorized

Get stage motorization.

Arguments

*void **StgGetMotorized**(out int x, out int y, out int z, out int rotation, out int tilt)*

x, y, z, rotation, tilt 0 – axis is motorized, 1 - axis is not motorized

Timing

Executed immediately.

Remarks

Most microscope models use 5-axis motorization, however, there are instruments with special manipulators. It can also happen that arbitrary axis is switched off.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.22 and later

StgGetPosition

Get current stage position.

Arguments

*void **StgGetPosition**(out float x, out float y, out float z, out float rotation, out float tilt)*

x	absolute x coordinate [mm]
y	absolute y coordinate [mm]
z	absolute z coordinate [mm]
rotation	absolute stage rotation [degrees]
tilt	absolute stage tilt [degrees]

Timing

Executed immediately.

Remarks

If the stage is busy (movement is in progress), current position is updated approximately every 500 ms.

If an axis is not motorized, zero is returned.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	2.0.22

StgIsBusy

Determine if the stage is moving.

Arguments

*int **StgIsBusy**(void)*

return value	0 – stage idle, 1 – stage movement in progress

Timing

Executed immediately.

Remarks

This function can be also used in conjunction with *StgWatchdog()*.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

StgIsCalibrated

Determine if the stage has been successfully calibrated.

Arguments

*int **StgIsCalibrated**(void)*

return value	0 – not calibrated 1 - calibrated

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

StgMove

Start stage movement, specify velocity.

Arguments

*void **StgMove**(in float vx[, in float vy[, in float vz[, in float vrotation[, in float vtilt]]]])*

vx	x velocity [mm/s]
vy	y velocity [mm/s]
vz	z velocity [mm/s]
vrotation	rotation velocity [degrees/s]
vtilt	tilt velocity [degrees/s]

Timing

Executed immediately, Wait B flag is set during the movement.

Remarks

Infinite movement is started. Application must stop it later using *StgStop()* call.

Important: **Stages without position indicators will run infinitely. If the stage is forced to move for a long time after it hits one of the mechanical limits, it can be damaged.**

If less than 5 arguments are passed, the other velocities are zero.

This function is supported by all types of stages. It is recommended to use watchdog timer together with *StgMove()*. Refer to *StgWatchdog()*.

Call Context

All stages and detector noses must be idle (Wait B flag must not be set). If the stage is busy, the command is ignored.

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.10 and later	yes

StgMoveTo

Move the stage to a position.

Arguments

*void **StgMoveTo**(in float x[, in float y[, in float z[, in float rotation[, in float tilt]]]])*

x	absolute x position [mm]
y	absolute y position [mm]
z	absolute z position [mm]
rotation	absolute rotation [degrees]
tilt	absolute tilt [degrees]

Timing

The execution time may vary. The time depends on the distance of the movement. Wait B flag is set while running.

Remarks

The stage position is restricted in several ways, if an invalid position is required, it is silently limited into the allowed range.

If less than 5 arguments are passed, other positions are kept unchanged.

This function is supported only by stages equipped with position indicators.

Call Context

All stages and detector noses must be idle (Wait B flag must not be set). Stage must be calibrated. If the stage is busy or not calibrated, the command is ignored.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

StgStop

Stop the stage movement.

Arguments

*void **StgStop**(void)*

Timing

Executed immediately.

Remarks

If the stage is in motion, it is stopped immediately.

Call Context

Anytime. Make sure that this call does not wait for Wait B flag.

Compatibility

1.x.x	2.x.x
1.0.8 and later	yes

StgWatchdog

Stage safety watchdog timer.

Arguments

*void **StgWatchdog**(in float timeout)*

timeout timeout [s]

Timing

Executed immediately.

Remarks

Stage watchdog timer is a safety feature. When stage movement is started from remote application, it may be hazardous if the remote application crashes or if the connection is lost, because the movement won't stop.

Before starting the movement, application may schedule watchdog timer, which stops the movement when it times out. The application must periodically restart the timer, indicating that it is still alive.

The timer is restarted by calling *StgIsBusy()*. When *StgIsBusy()* reports “in motion” status, timer is internally restarted. When it reports “stopped” status, the timer is killed.

Once *StgWatchdog()* is initiated, application is responsible for calling *StgIsBusy()* periodically, in a shorter time than specified by timeout. The timer is killed either if *StgIsBusy()* reports that the movement has been finished or if *StgStop()* is called explicitly.

Example:

```
StgWatchdog(2.0);           // schedule watchdog timer to 2 s
StgMoveTo(10, 20);          // movement command
while (StgIsBusy()) {       // restart/kill the timer
        if (stop_requested) {  // if stop requested by user
                StgStop();
                break;
        }
        Sleep(1000);        // wait a while
}
```

Call Context

Just before movement is started.

Compatibility

1.x.x	2.x.x
no	2.0.22 and later

Input Channels and Detectors

Each microscope is equipped with at least one image input channel. Each input is assigned one of the detectors. This section describes how detectors are controlled and assigned to input channels.

DtAutoSignal

Invoke automatic brightness/contrast adjustment procedure.

Arguments

*void **DtAutoSignal**(in int channel)*

channel	input video channel

Timing

Execution time may vary between 1 and 10 seconds (wait D).

Remarks

Channel index is a zero based index of the input channel. Brightness and contrast of the active detector is adjusted to get optimal signal.

If *DtAutoSignal()* function is called, appropriate detector must be selected as an active one first.

See also *DtSelect(), DtSetGainBlack(), DtGetGainBlack()*.

Call Context

Scanning must be inactive.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

DtEnable

Enable/disable input channel.

Arguments

*void **DtEnable**(in int channel, in int enable[, in int bpp])*

channel	input video channel index
enable	0 – disable, 1 – enable
bpp	bits per pixel (optional argument) 8 or 16, default is 8

Timing

Executed immediately.

Remarks

This function is called before scanning is started. If input is enabled, image data is sent to the client. If input is disabled, image data is ignored.

Example:

```
DtSelect(0, 1);          // select detector #1 into channel #0
DtEnable(0, 1);          // enable channel #0, 8 bpp
DtAutoSignal(0);         // adjust detector brightness/contrast
ScScanXY(...);           // start scanning

// process image data

ScStop(...);             // stop scanning
```

If 16 bits per pixel is selected, the *ScData()* callback has twice as much bytes in the data buffer than in 8-bit mode.

See also *DtGetEnabled()*.

Call Context

Scanning must be inactive when calling this function.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Protocol versions <= 1.0.5 do not have the bpp argument and use 8 bpp. Protocol versions >= 1.0.6 have optional bpp argument.

DtEnumDetectors

Get list of available detectors.

Arguments

map ***DtEnumDetectors****(void)*

return value	list of detectors

Timing

Executed immediately.

Remarks

Map of detectors is returned.

The *map* has the following form:

```
det.0.name
det.0.detector
det.1.name
det.1.detector
…
det.N-1.name
det.N-1.detector
```

N is number of installed detectors.

name is human-readable detector name.

detector is a numeric detector index provided in *Dt()* functions, eg. *DtSelect()*.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

DtGetChannels

Get number input video channels.

Arguments

*int **DtGetChannels**(void)*

return value — number of available input video channels

Timing

Executed immediately.

Remarks

When channel index is referred, the channel index must be less than the number of available channels.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

DtGetEnabled

Get input channel status (enabled / disabled, bpp).

Arguments

*void **DtGetEnabled**(in int channel, out int enable, out int bpp)*

channel	input video channel index
enable	0 – disabled, 1 – enabled
bpp	bits per pixel (8 or 16)

Timing

Executed immediately.

Remarks

See *DtEnable()* for details.

Call Context

-

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.11 and later	2.0.1 and later

DtGetGainBlack

Get detector gain and black level (contrast/brightness).

Arguments

*void **DtGetGainBlack**(in int detector, out float gain, out float black)*

detector	detector index
gain	gain (contrast) [percent]
black	black level (brightness) [percent]

Timing

Executed immediately.

Remarks

Detector index is a zero based detector index returned by *DtEnumDetectors().*

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

DtGetSelected

Get detector selected into a channel.

Arguments

*void **DtGetSelected**(in int channel, out int detector)*

channel	input video channel index
detector	detector index

Timing

Executed immediately.

Remarks

Returns index of the detector selected into the specified channel.

See also *DtEnumDetectors(), DtGetChannels(), DtSelect().*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.11 and later	2.0.1 and later

DtSelect

Select active detector.

Arguments

*void **DtSelect**(in int channel, in int detector)*

channel	input video channel index
detector	detector index

Timing

Executed immediately.

Remarks

There is an arbitrary detector assigned to each input channel. Since there is an analog cross switch, any detector can be assigned to any input channel.

See also *DtEnumDetectors(), DtGetChannels(),DtGetSelected().*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

DtSetGainBlack

Set detector gain and black level.

Arguments

*void **DtSetGainBlack**(in int detector, in float gain, in float black)*

detector	detector index
gain	gain (contrast) [percent]
black	black level (brightness) [percent]

Timing

Execution time depends on the type of the detector. For standard SE and BSE detectors the typical time is 100 ms. Wait C is set while not ready.

Remarks

Detector index is a zero based detector index returned by *DtEnumDetectors().*

Call Context

Anytime

Also Affected

Objective centering and 3D Beam can be changed if some detectors are selected. If 3D Beam is used, its value should be set up later, after *DtSelect()* is finished.

Compatibility

1.x.x	**2.x.x**
yes	yes

Scanning

Scanning functions control how the electron beam scans over the specimen surface. The most important parameters are scanning rate, window size and scanning method. There are several scanning methods available – most frequently used XY Scan, then Line Scan, Mask Scan and others.

ScEnumSpeeds

Get list of available scanning speeds.

Arguments

*map **ScEnumSpeeds**(void)*

return value scanning speed list

Timing

Executed immediately.

Remarks

Map of speeds is returned.

The map *key* has the following form:

```
speed.1.dwell
speed.2.dwell
…
speed.N.dwell
```

Where **N** is the total number of speed indexes.

The map *value* contains **float** value describing pixel dwell time in microseconds for given speed index.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

ScGetBlanker

Get current beam blanker mode.

Arguments

*int **ScGetBlanker**([in int index])*

index	beam blanker index 0 – electrostatic blanker (default) 1 – magnetic gun blanker
return value	blanking mode

Timing

Executed immediately.

Remarks

There are three possible blanking modes:

0 – beam is always ON, that is, blanker is switched off.

1 – beam is always OFF.

2 – beam is ON only if the image acquisition unit requires it. In the XY scanning mode, beam is OFF during the line and the frame retrace.

Magnetic gun blanker is relatively slow, only modes 0 and 2 are supported.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

The optional *index* argument is supported in version 2.0.3 and later.

ScGetExternal

Get external scanning status.

Arguments

*int **ScGetExternal**(void)*

return value	external scanning status

Timing

Executed immediately.

Remarks

External scanning can either be ON or OFF:

0 – external scanning is OFF.
1 – external scanning is ON.

See *ScSetExternal()* for more information.

Call Context
Anytime

Also Affected
-

Compatibility

1.x.x	2.x.x
yes	yes

ScGetSpeed

Get current scanning speed index.

Arguments
*int **ScGetSpeed**(void)*

return value scanning speed index

Timing
Executed immediately.

Remarks
Scanning speed is always valid, no matter if the scanning is active or stopped. Refer to *ScEnumSpeeds()* for details.

Call Context
Anytime

Compatibility

1.x.x	2.x.x
yes	yes

ScLUTParGet

Get image look-up table parameters.

Arguments
*void **ScLUTParGet**(in int channel, out float lut_min, out float lut_max, out float lut_gamma)*

channel	input video channel
lut_min	left margin [%], default is 0.0
lut_max	right margin [%], default is 100.0
lut_gamma	gamma [-], default is 1.0

Timing

Executed immediately.

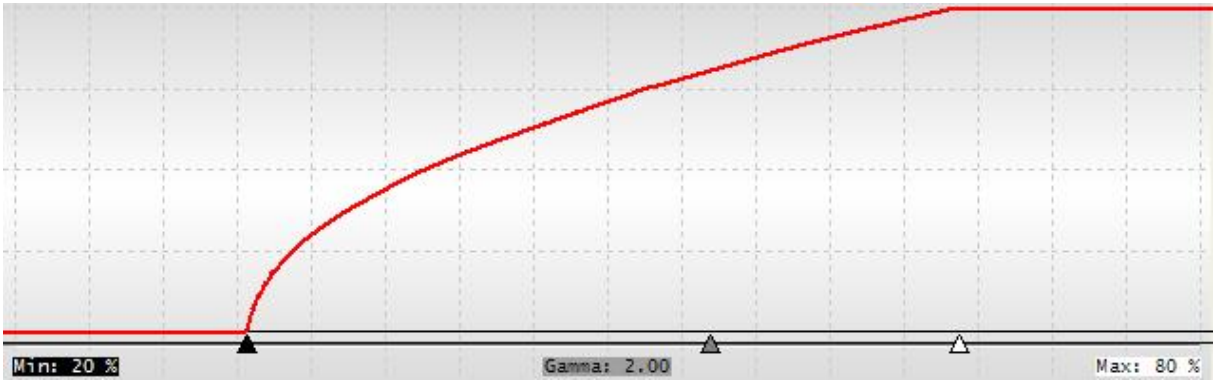
Remarks

Read image look-up table configuration parameters.

Look-up table for each SharkSEM session, as well as Vega (Mira) GUI look-up table, are maintained independently. That is, *ScLUTParGet()* and *ScLUTParSet()* functions access only look-up table which is locally defined for the actual SharkSEM session.

Image look-up table of Vega (Mira) GUI cannot be read or written.

The parameters *lut_min*, *lut_max*, *lut_gamma* define the pixel brightness transformation curve. Example look-up table may look like:



Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.8 and later

ScLUTParSet

Set image look-up table parameters.

Arguments

*void **ScLUTParSet**(in int channel, in float lut_min, in float lut_max, in float lut_gamma)*

channel	input video channel
lut_min	left margin [%]
lut_max	right margin [%]
lut_gamma	gamma [-]

Timing

Executed immediately.

Remarks

Set image look-up table parameters.

See *ScLUTParGet()*for parameter explanation.

See also *ScLUTSrcSet()*.

Call Context

Scanning must be inactive.

Compatibility

1.x.x	2.x.x
no	2.0.8 and later

ScLUTSrcGet

Get the current image look-up table origin.

Arguments

int ***ScLUTSrcGet()***

return value	0 – look-up table is switched off 1 – GUI look-up table is used (default) 2 – SharkSEM session-specific table is used

Timing

Executed immediately.

Remarks

Look-up table parameters are configured via *ScLUTParSet()* function. This function configures the SharkSEM session-specific table.

The remote application can select, whether look-up table is not in action (0), GUI look-up table is used (1), or SharkSEM look-up table is used (2).

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.8 and later

ScLUTSrcSet

Set the image look-up table origin.

Arguments

*void **ScLUTSrcSet**(in int lut_src)*

lut_src	0 – look-up table is switched off 1 – GUI look-up table is used (default) 2 – SharkSEM session-specific table is used

Timing

Executed immediately.

Remarks

See *ScLUTSrcGet()* for details.

Call Context

Scanning must be inactive.

Compatibility

1.x.x	2.x.x
no	2.0.8 and later

ScScanBitmask

This function has been removed due to the redesign of the scanning ramp generator.

Compatibility

1.x.x	2.x.x
yes	no

ScScanLine

Start scanning along a line. Beginning of the line, end of the line, pixel dwell time and number of pixels are provided.

In this mode, electron beam can be deflected to any visible position. This feature is sometimes referred as *Spot Mode*.

Arguments

*int **ScScanLine(***
in unsigned int frameid,
in unsigned int width,
in unsigned int height,
in unsigned int x0,
in unsigned int y0,
in unsigned int x1,
in unsigned int y1,
in unsigned int dwell_time,
in unsigned int pixel_count,
in int single
);

frameid	unique frame id sent back in the data callback
width, height	dimensions of the whole scanning window
x0, y0	beginning of the line
x1, y1	end of the line
dwell_time	pixel acquisition time in [ns]
pixel_count	number of pixels on the line
single	0 – continual scanning, 1 – single frame
return value	0 – ok, < 0 – failed, invalid parameters

*void **ScData(***
in unsigned int frameid,
in int channel,
in unsigned int index,
in int bpp,
in char[] data
);

See *ScScanXY()* for details.

Timing

ScScanLine() is executed immediately. Wait A flag is set during scanning. The flag is cleared when single scan has finished or when ScStopScan is called.

ScData() is called periodically when image data is ready, the period is about 50 ms.

Remarks

ScScanLine()

The microscope detectors and input image channels must be configured before calling *ScScanLine()*.

Width x *height* is the window size related to *View Field* microscope parameter. The line is drawn from (x0, y0) to (x1, y1). X grows from left to right, Y grows from top to bottom. Top left corner has coordinates (0, 0).

Examples:

ScScanLine(0, 1000, 1000, 0, 0, 999, 999, 1000, 2048, 1)
Scan single image, frame id 0, from top-left corner to bottom-right corner, the line consists of 2048 pixels, each pixel acquisition takes 1000 ns.

ScData()
is called to pass the image data to the client.

Call Context

Scanning must be inactive when calling *ScScanLine()*. No automatic procedure should be running. Make sure the wait flags are set properly.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

ScScanLitho

This function has been removed due to the redesign of the scanning ramp generator.

Compatibility

1.x.x	2.x.x
yes	no

ScScanXY

Start scanning over rectangular area.

Arguments

*int **ScScanXY(***
in unsigned int frameid,
in unsigned int width,
in unsigned int height,
in unsigned int left,
in unsigned int top,
in unsigned int right,
in unsigned int bottom,
in int single,
[in unsigned dwell]
);

frameid	unique frame id sent in the data callback
width, height	dimensions of the whole scanning window
left, top, right, bottom	definition of the visible region
single	0 – continual scanning, 1 – single frame
dwell	pixel dwell time in [ns]
return value	0 – ok, < 0 – failed, invalid parameters

*void **ScData(***
in unsigned int frameid,
in int channel,
in unsigned int index,
in int bpp,
in char[] data
);

frameid	frame id
channel	channel index
index	index of the first pixel in the data buffer
bpp	bits per pixel (8 / 16)
data	image data buffer

Timing

ScScanXY() is executed immediately. Wait A flag is set during scanning. The flag is cleared when single scan has finished or when *ScStopScan()* is called.

ScData() is called periodically when image data is ready, the period is about 50 ms.

Remarks

ScScanXY()

Scanning according to the supplied parameters is started, other parameters like scanning speed, input signal configuration, etc., should have already been specified.

Width x *height* is the window size related to *View Field* microscope parameter. *Left*, *top*, *right* and *bottom* describe the visible region over which the beam scans.

If the *dwell* parameter is omitted, current scanning speed is used instead. Refer to *ScSetSpeed()*. The *dwell* parameter is supported since protocol revision 2.0.6. It is recommended that all new client applications use the *dwell* parameter instead of the *ScSetSpeed()* call.

Examples:

ScScanXY(13500, 1000, 1000, 0, 0, 999, 999, 1)
Scan single image, frame id 13500, 1000x1000 pixels, default scanning speed.

ScScanXY(13500, 1000, 1000, 0, 0, 999, 999, 1, 1500)
Scan single image, frame id 13500, 1000x1000 pixels, 1.5 µs dwell time.

ScScanXY(13501, 1000, 1000, 450, 450, 549, 549, 1)

Scan single image, frame id 13501, reduced raster 100x100 pixels from the center.

ScScanXY(13502, 10000, 10000, 4500, 4500, 5499, 5499, 1)
Scan single image, frame id 13502, reduced raster 1000x1000 pixels from the center, digital zoom 10x.

ScData()
is called when image data is sent to the client. If 16-bit scanning is active, one pixel consists of two bytes, the bytes are stored in little-endian order.

Call Context

Scanning must be inactive when calling *ScScanXY()*. No automatic procedure should be running. Make sure the wait flags are set properly.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

The optional *dwell* argument is supported since version 2.0.6.

ScSetBeamPos

Deflect the electron beam, in other words, set beam position. This can be called only if scanning is inactive.

Arguments

*void **ScSetBeamPos**(in float x, in float y)*

x	x position, 0.0 – 1.0
y	y position, 0.0 – 1.0

Timing

Executed immediately.

Remarks

The virtual image size is 1.0 x 1.0. Top left corner has (0, 0) coordinates. The *ScSetBeamPos()* deflects the beam to the given position. The beam remains in this position till either next *ScSetBeamPos()* call is issued or till scanning is started.

Call Context

Scanning must be inactive.

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.6 and later	yes

ScSetBlanker

Set beam blanker operating mode.

Arguments

*void **ScSetBlanker**([in int index,] in int mode)*

index	beam blanker index 0 – electrostatic blanker (default) 1 – magnetic gun blanker
mode	blanking mode

Timing

Executed immediately.

Remarks

Refer to *ScGetBlanker()* for description of the blanking modes.

If the scanning is inactive, blanker is switched on/off immediately. If the scanning is active, blanker mode is changed after the scanning is stopped.

If the beam blanker is not installed, calling this function is legal but has no effect.

The *index* argument is optional. If only single argument appears, index defaults to zero.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

The optional *index* argument is supported in version 2.0.3 and later.

ScSetExternal

Enable/disable external scanning.

Arguments

*void **ScSetExternal**(in int enable)*

enable	enable flag

Timing

Executed immediately.

Remarks

If external scanning is enabled (ON), scanning amplifiers switch their inputs to the external scanning ramp. External scanning is typically used by EDX systems.

External scanning can either be ON or OFF:
0 – external scanning is OFF.
1 – external scanning is ON.

External scanning does not affect internal scanning. This function controls simple switching circuit (multiplexer), it does not start or stop the internal scanning.

Besides the active scanning ramp input, beam blanker input corresponding to the active scanning ramp input is selected.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

ScSetSpeed

Set current scanning speed index.

Arguments

*void **ScSetSpeed**(in int speed)*

speed	required scanning speed index

Timing

Typically 5 ms to execute (wait C).

Remarks

If incorrect index is provided, closest valid index is selected automatically.

Call Context

Anytime.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

There is a difference between 1.x.x and 2.x.x protocol versions. In the 2.x.x version, the scanning speed is not changed if scanning is in progress. Scanning must be restarted to apply the new scanning speed value.

Newly developed client applications should not rely on this function. It is recommended to define the dwell time in each specific scanning function. Since protocol revision 2.0.6, all scanning functions allow specifying arbitrary dwell time.

ScStopScan

Stop scanning.

Arguments

*void **ScStopScan**(void)*

Timing

1 to 20 ms to execute (wait C).

Remarks

If scanning was initialized using one of the scanning functions (*ScScanXY()* or similar), it must be stopped later using *ScStopScan()*. It does not matter if single scan or continual scanning was requested.

It is not guaranteed that the image data has already been sent at the moment this call is completed. Application must use the frame id identifier to distinguish which frame the image data belongs to.

In particular, following command order can be useful.

```
ScScanXY(101, -- first region --, 1);          // single scan of region 101
void ScStopScan(void);                         // wait A set to 1
                              // we get here when the region 101 is finished
ScScanXY(102, -- second region --, 1);         // single scan of region 102
void ScStopScan(void);                         // wait A set to 1
                              // we get here when the region 102 is finished
...
```

Note that the *ScStopScan()* call is required after every command which starts scanning. If wait A was not specified, single scan would be terminated prematurely.

Call Context

Scanning must be active when calling this function.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Scanning Mode

SMEnumModes

Enumerate all available scanning modes.

Arguments

*map **SMEnumModes**(void)*

return value list of scanning modes

Timing

Executed immediately.

Remarks

The map *key=value* has the following form (example):

```
mode.0.name=RESOLUTION
mode.1.name=DEPTH
mode.5.name=WIDE FIELD
```

The total number of scanning modes depends on the SEM model and configuration. Each scanning mode is assigned a number. Some scanning mode numbers may be omitted. That is, the sequence may not be consecutive.

Each map *value* contains **string** value describing name of the scanning mode.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

SMGetMode

Get current scanning mode.

Arguments

*int **SMGetMode**(void)*

return value scanning mode number

Timing

Executed immediately.

Remarks

Refer to *SMEnumModes()* for details.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
yes	yes

SMGetPivotPos

Get pivot point of the active scanning mode.

Arguments

*int **SMGetPivotPos**(out float pivot)*

pivot scanning pivot point position [mm]

return value 0 – ok, < 0 – unable to determine pivot point

Timing

Executed immediately.

Remarks

Scanning pivot point is a virtual point on the SEM optical axis. The electron beam diverges from this point.

Exact position of pivot point depends on the scanning mode and other optical parameters.

The value is relative to the objective pole piece. The orientation is positive below objective pole piece, negative above objective pole piece. The orientation and units are same as for working distance.

Call Context

Anytime

Compatibility

1.x.x	2.x.x

no	2.0.22 and later

SMSetMode

Set current scanning mode.

Arguments

*void **SMSetMode**(in int mode)*

mode	scanning mode number

Timing

Executed immediately. Function following the *SMSetMode()* may be delayed automatically, because automatic degauss of the column may take place and the degauss procedure cannot be interrupted.

Remarks

Only valid scanning modes should be selected. If the scanning mode number is invalid, the behavior is undefined.

Call Context

Scanning must be inactive.

Also Affected

Column centering values are coupled with each scanning mode. It may also happen that view field is changed, because each scanning mode has different magnification limits.

Compatibility

1.x.x	**2.x.x**
yes	yes

Vacuum

VacGetPressure

Read microscope chamber/column/gun pressure.

Arguments

*float **VacGetPressure**(in int gauge)*

gauge	0 – chamber, 1 – column, 2 – gun
return value	pressure in [Pa]

Timing

Executed immediately.

Remarks

Use together with *VacGetStatus()* to obtain complete information about the vacuum conditions in the SEM.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

VacGetStatus

Read microscope vacuum status.

Arguments

*int **VacGetStatus**(void)*

return value	vacuum status

Timing

Executed immediately.

Remarks

The vacuum status is a single integer value. It tells the current vacuum conditions of the microscope.

-1	error – call *VacGetPressure()* to check the pressure
0	ready for operation
1	pumping in progress
2	venting in progress
3	vacuum off (pumps are switched off, valves are closed)
4	chamber open

It takes several minutes to evacuate the microscope. If sufficient vacuum level is not reached in an arbitrary time, the system reports error (-1).

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

VacGetVPMode

Get on/off status of the variable pressure mode.

Arguments

int ***VacGetVPMode****(void)*

return value	0 – high vacuum operation 1 – variable pressure operation

Timing

Executed immediately.

Remarks

This function returns the vacuum operation mode. To find out whether the vacuum conditions allow imaging, call *VacGetStatus()* function.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x

yes	yes

VacGetVPPress

Get variable pressure setpoint.

Arguments

*float **VacGetVPPress**(void)*

return value pressure in [Pa]

Timing

Executed immediately.

Remarks

Does not measure anything, just returns the required chamber pressure setpoint. The value is valid in both high vacuum and variable pressure mode.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	**2.x.x**
yes	yes

VacPump

Evacuate the microscope chamber.

Arguments

*void **VacPump**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

After the pumping procedure start, remote application should check the vacuum status periodically for success/error.

The call has no effect if the microscope is already evacuated.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

VacSetVPMode

Change vacuum mode from variable pressure to high vacuum and vice versa.

Arguments

*void **VacSetVPMode**(vpmode)*

vpmode	0 – high vacuum mode 1 – variable pressure mode

Timing

Executed immediately. May take several seconds to complete the process, may even require user intervention.

Remarks

The vacuum mode change is a complex process. Some microscopes are not capable of doing the mode change automatically and user intervention is required, it may be necessary to change the aperture configuration or to open/close a valve.

Use *VacGetStatus()* to find out if the microscope is ready for operation after the change of the vacuum mode.

Call Context

Anytime

Also Affected

Detectors are enabled / disabled depending on the vacuum mode. HV is switched off (thermal emission) or beam is switched off (field emission) automatically.

Compatibility

1.x.x	2.x.x
yes	yes

VacSetVPPress

Set variable pressure setpoint.

Arguments

*void **VacSetVPPress**(pressure)*

pressure	pressure in [Pa]

Timing

Executed immediately.

Remarks

Set variable pressure setpoint. Depending on the SEM model and configuration, the value is silently limited to a certain range. It may take an arbitrary time before the required pressure in the SEM chamber is reached.

Call Context

Anytime

Also Affected

Some detectors may not function if the pressure is out of their range.

Compatibility

1.x.x	2.x.x
yes	yes

VacVent

Vent the microscope chamber.

Arguments

*void **VacVent**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

Call *VacVent()* when chamber has to be opened. For field emission microscopes, this only allows air (nitrogen) into the chamber. On the other hand, tungsten gun microscopes have no valve between the chamber and the gun. It means that the main high voltage is automatically switched off during the venting procedure.

The call has no effect if the microscope is already vented.

Call Context

Anytime

Also Affected

HV is switched off automatically (Vega\\ only).

Compatibility

1.x.x	2.x.x
yes	yes

Airlock

Airlock (load lock) is a device for fast specimen exchange. The airlock chamber is attached to the main SEM chamber, there is a separating valve between them through which a sample is pushed in and pulled out.

Currently two airlock types are offered, manual and fully automated.

ArlGetType

Determine what type of airlock is installed.

Arguments

*int **ArlGetType**(void)*

return value airlock type

Timing

Executed immediately.

Remarks

The airlock type can be:

0 not installed
1 manual airlock (*ArlXxx* functions)
2 motorizes airlock (*Arl2Xxx* functions)

The motorized airlock has been introduced in version 2.0.5.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
No	2.0.5 and later

Airlock type 1 – manual

Manual airlock is not equipped with automated mechanism for specimen exchange. API described in this chapter is available only if airlock type 1 is installed.

ArlCloseValve

Close the separation valve between the SEM chamber and the airlock chamber.

Arguments

*void **ArlCloseValve**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

The separation valve is closed and the SEM chamber and the airlock chamber are separated.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.7 and later	yes

ArlGetStatus

Read airlock status.

Arguments

*int **ArlGetStatus**(void)*

return value airlock status

Timing

Executed immediately.

Remarks

The airlock status is one of:

-2 airlock is not installed

-1	error
0	ready – separation valve can be opened
1	open – separation valve is opened
2	pumping in progress
3	venting in progress
4	airlock vented, door closed
5	airlock vented, door open
6	vacuum system is switched off

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.7 and later	yes

ArlOpenValve

Open the separation valve between the SEM chamber and the airlock chamber.

Arguments

*void **ArlOpenValve**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

ArlOpenValve() can be called only if both SEM chamber and airlock chamber are evacuated. The effect is that the separation valve is opened and a sample can be inserted to or removed from the SEM chamber.

Call Context

Both SEM and airlock chambers evacuated.

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.7 and later	yes

ArlPump

Pump (evacuate) the airlock chamber.

Arguments

*void **ArlPump**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

ArlPump() evacuates the airlock chamber. The pumping is slower than venting, it takes tens of seconds. Check *ArlGetStatus()* to find out when the pumping is finished.

The call has no effect if the airlock is already evacuated.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.7 and later	yes

ArlVent

Vent the airlock chamber.

Arguments

*void **ArlVent**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

Call *ArlVent()* if the separating valve is closed and the airlock chamber shall be open to air. The venting is fast (several seconds).

The call has no effect if the airlock is already vented.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.7 and later	yes

Airlock type 2 – motorized

Motorized airlock allows fully automated specimen exchange. The SharkSEM API is different from the manual airlock. Functions described in this chapter can only be used if airlock type 2 is installed.

Arl2Calibrate

Calibrate the airlock manipulator.

Arguments

*void **Arl2Calibrate**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

The whole procedure takes tens of seconds, use *Arl2GetStatus()* to find out when the procedure is finished.

This function is not necessary during normal operation. This is actually a variant of *Arl2Vent()* function, unlike *Vent*, is also does calibration of the airlock manipulator.

Call Context

Airlock must be in the "inside" position.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2GetStatus

Get the airlock status.

Arguments

*void **Arl2GetStatus**(out int status, out int detailed_status)*

status	main airlock status (see below)
detailed_status	detailed status (not documented)

Timing

Executed immediately.

Remarks

Status describes the current state of the whole airlock system.

Symbol	Code	Meaning
RALS2_NOT_INSTALLED	-2	Airlock not installed
RALS2_ERROR	-1	Airlock error
RALS2_POS_OUTSIDE	0	Manipulator is outside, ready for command
RALS2_POS_INSIDE	1	Manipulator is inside, ready for command
RALS2_POS_STOPPED	2	Manipulator was manually stopped
RALS2_PROC_PUMPING	10	“Pump” in progress
RALS2_PROC_LOADING	11	“Load” in progress
RALS2_PROC_UNLOADING	12	“Unload” in progress
RALS2_PROC_VENTING	13	“Vent” in progress
RALS2_PROC_RECOVERY	14	“Recovery” in progress

Call Context

Anytime.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2Load

Load the specimen from airlock chamber onto the stage.

Arguments

*void **Arl2Load**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

The whole procedure takes about one hundred seconds, use *Arl2GetStatus()* to find out when the procedure is finished.

Call Context

The airlock must be in the "inside" position before this call (see *Arl2GetStatus()*). The SEM chamber must be pumped (see *VacGetStatus()*). The stage must be calibrated (see *StgIsCalibrated()*).

There must not be specimen on the stage.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2MoveStop

Stop the airlock manipulator.

Arguments

*void **Arl2MoveStop**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

This function is emergency stop function. It can be used, for example, when the user sees that the airlock manipulator is in motion and that there is a risk of mechanical collision.

After this call, *Arl2Recovery()* will be typically called.

Call Context

Anytime.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2Pump

Load the specimen into the airlock chamber and pump the chamber.

Arguments

*void **Arl2Pump**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

In the first step, the specimen is loaded into the airlock chamber. In the second step, the chamber is pumped down. The whole procedure takes tens of seconds. Use *Arl2GetStatus()* to find out when the pumping is finished.

The call has no effect if the airlock is already evacuated.

Call Context

The main SEM chamber should be pumped and SEM vacuum status should be "ready" before this call (see *VacGetStatus()*). The airlock must be in the "outside" position before this call (see *Arl2GetStatus()*).

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2Recovery

Error recovery procedure.

Arguments

*void **Arl2Recovery**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

Arl2Recovery() is used when airlock manipulator crashes, when it is manually stopped, etc.

The procedure vents the main SEM chamber, airlock chamber, and the manipulator is moved to the outside position. The recovery procedure leaves the SEM chamber vented, airlock vented and separation valve closed.

Call Context

In the "inside" position, "outside" position, or in case of "error" status.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2Unload

Unload the specimen from stage, move it into the airlock chamber.

Arguments

*void **Arl2Unload**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

The whole procedure takes about one hundred seconds, use *Arl2GetStatus()* to find out when the procedure is finished.

Call Context

The airlock must be in the "inside" position before this call (see *Arl2GetStatus()*). The SEM chamber must be pumped (see *VacGetStatus()*). The stage must be calibrated (see *StgIsCalibrated()*).

The specimen must be on the stage, the airlock holder must be empty.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Arl2Vent

Vent the airlock chamber and remove the specimen from the chamber.

Arguments

*void **Arl2Vent**(void)*

Timing

This is a non-blocking call, it is executed immediately.

Remarks

In the first step, the airlock chamber is vented. In the second step, the specimen is moved out of the airlock chamber. The whole procedure takes tens of seconds. Use *Arl2GetStatus()* to find out when the procedure is finished.

Call Context

The airlock must be in the “inside” position before this call (see *Arl2GetStatus()*).

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

Detector and shutter manipulation

There exist detectors, for example backscattered electron detector or cathodoluminescence detector, equipped with motorized travel. Such detectors can be inserted or retracted from a remote client application.

Refer also to the *Detector nose guard* chapter, which describes how to check whether the shared space below the SEM objective is available.

Mechanical detector shutters, which prevent sensor damage, eg. ion damage of EDS detector window, are controlled similar way as retractable detectors. Due to that, API described in this chamber can control motorized shutters as well.

As a general term, *nose* describes either detector nose (retractable part) or shutter nose.

The detector (shutter) indexes are zero based. As of 2.0.19, the index assignment is:

0	Standard BSE Detector #1
1	Standard CL Detector #1
2	Standard BSE Detector #2
3	Standard CL Detector #2
4	Standard Shutter #1
5	Standard R-STEM Detector #1

In the future, detectors or shutters can have up to three (X, Y, Z) actuators. Currently, only X axis is used.

NoseCalibrate

Mechanically calibrate the detector (shutter) travel.

Arguments

*int **NoseCalibrate**(in int nose_index)*

nose_index	nose index	
return value	-1	invalid request (bad index, Wait B is set)
	0	calibration was started

Timing

The nose calibration takes up to one minute, depending on the travel distance and its original position. Wait B flag is set during the calibration procedure.

Remarks

If the calibration is successful, *NoseIsCalib()* returns 1 when the procedure is finished.

The calibration is done in a safe manner, i.e. the end switch is located in the outside (retracted) position.

The current state of the calibration procedure can be polled using *NoseIsBusy()* function.

Call Context

All stages and noses must be idle (Wait B flag must not be set).

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseGetConfig

Read configuration of a nose.

Arguments

*map **NoseGetConfig**(in int nose_index)*

nose_index	nose index
return value	nose configuration

Timing

Executed immediately.

Remarks

Set of *key=value* pairs is returned.

Example return value:

```
Name=Standard BSE Detector #1
LimitMin=-45.0
LimitMax=0.0
```

The limits are in [mm]. In the inserted position, the value is zero. In the retracted position, the value is negative. The nose name is a unique string.

This parameter set may be expanded in the future. Client application must handle unknown keys.

If the *nose_index* is invalid, empty string is returned.

Call Context

Anytime.

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseGetPosition

Get detector (shutter) nose position.

Arguments

*void **NoseGetPosition**(in int nose_index, out float x, out float y, out float z)*

nose_index	nose index
x, y, z	manipulator position

Timing

Executed immediately.

Remarks

X, Y, Z position in [mm] is returned.

If the *nose_index* is invalid, all zeroes are returned.

Call Context

Anytime.

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseIsBusy

Determine if nose is in motion.

Arguments

*int **NoseIsBusy**(in int nose_index)*

nose_index	nose index	
return value	-1	invalid request
	0	idle
	1	in motion

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseIsCalib

Determine if nose is calibrated.

Arguments

*int **NoseIsCalib**(in int nose_index)*

nose_index	nose index	
return value	-1	invalid request
	0	not calibrated
	1	calibrated

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseMoveToMem

Move nose to an arbitrary predefined position.

Arguments

*int **NoseMoveToMem**(in int nose_index, in int memory)*

nose_index	nose index	
memory	index of a stored position	
return value	-1	invalid request

0 movement was started

Timing

The execution time depends on the travel distance. Wait B flag is set while in motion.

Remarks

For *nose_index* 0 through 3, and 5 (detectors), there are following predefined positions:

Memory	Detector Position	Typical X value
0	Fully retracted	-90.0
1	Inserted into working position	0.0
2	Not used	-50.0
3	Safe position (retracted so that there is not mechanical conflict with other detectors)	-50.0

For *nose_index* 4 (shutter), there are following predefined positions:

Memory	Detector Position	Typical X value
0	Fully retracted (parking position)	-90.0
1	Fully inserted (sensor covered)	0.0
2	Standby position (sensor not covered, partly retracted)	-10.0
3	Safe position (sensor not covered, mechanically safe)	-30.0

If there is potential mechanical conflict during nose manipulation, the movement request is discarded.

Call Context

All stages and noses must be idle (Wait B flag must not be set). Nose must be calibrated.

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NoseMoveToPos

Move nose to an exactly defined position.

Arguments

*int **NoseMoveToPos**(in int nose_index, in float x[, in float y][, in float z])*

nose_index	detector nose index	
x, y, z	target position [mm]	
return value	-1	invalid request
	0	movement was started

Timing

The execution time depends on the travel distance. Wait B flag is set while in motion.

Remarks

The target position is restricted in several ways, if an invalid position is specified, it is silently limited into the allowed range.

If there is potential mechanical conflict during nose manipulation, the movement request is discarded.

The *y* and *z* arguments are optional.

Call Context

All stages and detector noses must be idle (Wait B flag must not be set). Nose must be calibrated.

Compatibility

1.x.x	2.x.x
No	2.0.6 and later

NoseStop

Stop nose movement. Intended for emergency use only.

Arguments

*void **NoseStop**(in int nose_index)*

nose_index	nose index

Timing

Executed immediately.

Remarks

Immediately stop the nose movement. As a result, calibration may be lost.

If the nose is not in motion, this function is ignored.

Call Context

Anytime.

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

Detector nose guard

Some detectors and other SEM accessories are located under the microscope objective pole piece. The space here is restricted, and thus it is often necessary to insert just single device and to remove the others.

If such devices are equipped with automated motorized movement, it is possible to assign the shared space to specific device (or set of devices), and prevent others from moving in.

There is a token (lock), which is assigned to a device only if it is allowed to enter the shared space below the objective pole piece. If a device fails to acquire the lock, it must not be inserted.

List of devices:

Device	Identifier	Token assigned via SharkSEM
Tescan BSE detector	Standard BSE Detector #1	no
Tescan CL detector	Standard CL Detector #1	no
GIS	Orsay Multi-GIS	no
GIS	Orsay Single-GIS #1	no
Renishaw Raman	Renishaw Raman	yes
EBSD	EBSD Detector	no
Tescan BSE detector	Standard BSE Detector #2	no
Tescan CL detector	Standard CL Detector #2	no
SmarAct Nanomanipulator	SmarAct Nanomanipulator #1	no
SmarAct Nanomanipulator	SmarAct Nanomanipulator #2	no
SmarAct Nanomanipulator	SmarAct Nanomanipulator #3	no
SmarAct Nanomanipulator	SmarAct Nanomanipulator #4	no
Tescan Detector Shutter	Detector Shutter	no
OmniProbe Nanomanipulator	OmniProbe Nanomanipulator #1	yes
Tescan R-STEM detector	Standard R-STEM Detector #1	no
Omni GIS II	OmniGIS II #1	yes

NGuardGetStatus

Read list of devices which occupy the shared space.

Arguments

char[] ***NGuardGetStatus****(void)*

return value list of devices (see below)

Timing

Executed immediately.

Remarks

List of device identifiers is returned. The list can be empty. If there are more devices, each identifier is on a separate line.

Call Context

Anytime

Also Affected

-

NGuardLock

Acquire token.

Arguments

*int **NGuardLock**(in char[] dev_id)*

dev_id	device identifier (zero terminated string)	
return value	-1	invalid request
	0	token cannot be granted
	1	token granted to device *dev_id*

Timing

Executed immediately.

Remarks

The device identifiers are shown in the *List of devices* at the beginning of this chapter.

If the token is granted to the *dev_id* device, the device can be mechanically inserted until the token is returned.

If the token is already owned by the *dev_id* device, this call will succeed.

When the device is mechanically removed from the shared space, token must be returned using *NGuardUnlock()* call.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NGuardTest

Test if token is available.

Arguments

*int **NGuardTest**(in char[] dev_id)*

dev_id	device identifier (zero terminated string)	
return value	-1	invalid request
	0	token cannot be granted
	1	token can be granted device *dev_id*

Timing

Executed immediately.

Remarks

Similar to *NGuardLock()*, but the token availability is only tested, token is not granted.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

NGuardUnlock

Release token.

Arguments

*void **NGuardUnlock**(in char[] dev_id)*

dev_id	device identifier (zero terminated string)

Timing

Executed immediately.

Remarks

When the device is mechanically removed from the shared space, this function returns the token and other devices can subsequently acquire it.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

High Voltage

In this chapter, High Voltage stands for the main accelerating voltage of the microscope. The tungsten gun, LaB_6 gun, and the field emission microscopes use different accelerating voltage power supply unit. They share the same remote control function set and implement commands in a bit different way if necessary.

In the protocol version 1.x.x, *HVSetVoltage()* does not switch the HV indexes automatically and its execution is asynchronous.

In the protocol version 2.x.x, *HVSetVoltage()* can switch the HV index automatically and its execution is synchronous.

Since protocol version 2.0.16, *HVSetVoltage()* and *HVSetIndex()* can be executed either synchronously or asynchronously, depending on the optional second argument.

The reason for using HV indexes is that it is not possible to use a single set of column centering values for the whole HV range (typically 200 V – 30 kV). There are rather several indexes and each of them has its own set of column centering parameters. The per-index HV ranges do not overlap.

HVAutoHeat

Start automatic filament heating procedure. Only supported by tungsten SEMs and LaB_6 SEMs.

Arguments

*void **HVAutoHeat**(in int channel)*

channel	input video channel

Timing

Execution time between 20 and 60 seconds (wait D).

Remarks

This function is supported by tungsten gun SEMs only.

Channel index is a zero based index of the input channel. Appropriate detector must be selected as an active one.

During the automatic heating run, *AutoGun()* and *DtAutoSignal()* procedures are invoked.

The heating level is adjusted only for the current HV index. The other HV indexes remain untouched.

The automatic procedure relies on the specimen, it won't succeed if conductive specimen is not installed or if the detector does not give reasonable image signal.

See also *DtSelect(), AutoGun(), DtAutoSignal(), HVGetHeating().*

Call Context

Scanning must be inactive, no automatic procedure running.
HV must be switched on.
Beam must be switched on.
Valid detector must be selected.
Non-conductive specimen must be installed.

Also Affected

Gun centering, gain and black level of the active detector.

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

HVBeamOff

Turn off the beam.

Arguments

*void **HVBeamOff**(void)*

Timing

Executed immediately.

Remarks

This function switches off the beam. The way how this is done depends on the SEM type.

Tungsten gun: turn off the filament heating.
LaB_6 gun: close the gun valve, decrease the heating.
Field emission gun: close the gun valve.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

HVBeamOn

Turn on the beam.

Arguments

*void **HVBeamOn**(void)*

Timing

Executed immediately.

Remarks

This function switches on the beam. See *HVBeamOff()* for details.

Tungsten gun microscopes require several seconds to stabilize the beam after it has been switched on.

Call Context

If the vacuum is not *ready*, this command is silently ignored. See *VacGetStatus()* how to determine the vacuum conditions.

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

HVEnumIndexes

Enumerate HV indexes.

Arguments

*map **HVEnumIndexes**(void)*

return value HV index map

Timing

Executed immediately.

Remarks

With each index, several microscope parameters associated:

- actual HV value
- set of column centering values
- gain, offset of detectors
- gun heating level and HV bias (tungsten, LaB_6)

If an index is selected by calling *HVSetIndex()*, all above parameters are loaded automatically.

The map *key=value* has the following form:

```
hv.0.min=200
hv.0.max=5000
hv.0.value=1000
hv.1.min=5000
…
hv.N.min=20000
hv.N.max=30000
hv.N.value=30000
```

N+1 is the total number of HV indexes. The typical number of indexes is four, but application must be able to handle different count.

The map *value* element contains accelerating voltage in [V] for given HV index. The *min* and the *max* elements contain minimum allowed [V] and maximum allowed voltage [V] for this index, respectively. All numbers are floating point.

For future compatibility, application must silently ignore any *key* which does not have exactly the above form.

See also *HVGet()*, *HVSet()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Min, *max* introduced in version 1.0.5

HVGetBeam

Get beam status.

Arguments

int ***HVGetBeam****(void)*

return value	-1	filament is blown
	0	beam is off
	1	beam is on

Timing

Executed immediately.

Remarks

Determine the current beam status. It is possible to switch the beam on and off by *HVBeamOn()* and *HVBeamOff()* functions.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

HVGetEmission

Read measured emission current. The result is an approximate current emitted from the SEM filament tip.

Arguments

float ***HVGetEmission****(void)*

return value emission current [μA]

Timing

Executed immediately.

Remarks

First, it is necessary to determine the gun status by *HVGetBeam()*. The emission current value is valid only if the beam is on.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

HVGetFilTime

Get the filament live time since the last exchange.

Arguments

*int **HVGetFilTime**(void)*

return value	filament live time [s]

Timing

Executed immediately.

Remarks

The result is the time elapsed since the last filament exchange. Only the live time (operation time) is counted.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

HVGetHeating

Read filament heating current.

Arguments

*float **HVGetHeating**(void)*

return value	requested heating current [A]

Timing

Executed immediately.

Remarks

Returns the requested heating current. This is an indicative value. Depending on the SEM type, the proper heating current can be between 1.0 A and 3.0 A.

The heating level has separate value for each HV index. *HVGetHeating()* returns the value associated with the current index.

See also *HVAutoHeat().*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

HVGetIndex

Get current HV index.

Arguments

*int **HVGetIndex**(void)*

return value	0, 1, 2, …	HV index
	1000	HV is switched off
	1001	HV change is in progress

Timing

Executed immediately.

Remarks

Current HV status is determined and returned. There can be several HV indexes (typically four). Each index is linked with one predefined accelerating voltage value.

See *HVEnumIndexes()* for general discussion what HV index is.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

HVGetVoltage

Read current accelerating voltage.

Arguments

*float **HVGetVoltage**(void)*

return value	accelerating voltage [V]

Timing

Executed immediately.

Remarks

See also *HVGetIndex(), HVSetVoltage().*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

HVSetIndex

Set HV index and load all associated parameters.

Arguments

*void **HVSetIndex**(in int index[, in int async])*

index	valid HV index
async	synchronous/asynchronous execution (optional) 0 – synchronous (default) 1 – asynchronous

Timing

Several seconds (maximum about 100 seconds if the HV change is significant) until the system becomes stable. In version 1.x.x, or in asynchronous mode, check Wait C flag. In synchronous mode, the function is blocking (subsequent command is executed only after *HVSetIndex()* is finished).

Remarks

HV index is set and all associated parameters are loaded.

See *HVEnumIndexes()* for general discussion what HV index is.

Since many microscope parameters are adjusted by the HV index change, it is recommended to check the operating conditions and the image quality after the function is finished.

Call Context

Anytime

Also Affected

Accelerating voltage, column centering values and detector settings are changed. It is also not guaranteed that view field size, scanning speed and PC settings will persist.

Compatibility

1.x.x	2.x.x
yes	yes

Non-blocking (asynchronous) behavior in 1.x.x.
Blocking behavior from 2.0.0 to 2.0.16.
User-defined behavior in version 2.0.16 and later.

HVSetVoltage

Set accelerating voltage.

Arguments

*void **HVSetVoltage**(in float voltage[, in int async])*

voltage	accelerating voltage [V]
async	synchronous/asynchronous execution (optional) 0 – synchronous (default) 1 – asynchronous

Timing

Several seconds (maximum about 100 seconds if the HV change is significant) until the system becomes stable. In version 1.x.x, or in asynchronous mode, check Wait C flag. In synchronous mode, the function is blocking (subsequent command is executed only after *HVSetIndex()* is finished).

Remarks

In the protocol version 1.x.x, the requested HV value must be inside the range of the currently selected HV index. If the value is outside the range, it will be limited. In the version 2.x.x, the index is switched automatically.

See also *HVEnumIndexes(), HVSetIndex(), HVGetVoltage().*

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

Non-blocking (asynchronous) behavior in 1.x.x.
Blocking behavior from 2.0.0 to 2.0.16.
User-defined behavior in version 2.0.16 and later.

HVStopAsyncProc

Cancel asynchronous procedure (*HVSetVoltage(), HVSetIndex()*).

Arguments

*void **HVStopAsyncProc**(void)*

Timing

Executed immediately. When asynchronous procedure is cancelled, the system requires few more seconds till the Wait C flag is cleared.

Remarks

This function has an effect only if HV asynchronous procedure is in progress. No matter which SharkSEM session invoked this call, the running HV change is stopped immediately.

As a result, the accelerating voltage value may end up somewhere between the original value and the requested value.

Call Context

Anytime.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.16 and later

SEM Presets

SEM preset is a user-defined group of SEM parameters (stage position, scanning mode, detector configuration, …). The presets can only be created by the SEM user. Each preset is identified by unique, user-defined name. SharkSEM can enumerate the presets and activate one of them.

PresetEnum

Get list of available SEM presets.

Arguments

char[] ***PresetEnum****(void)*

return value	string containing list of presets

Timing

Executed immediately.

Remarks

The returned string contains list of preset names. The string is divided into lines, there is one preset name per line. If there are no presets, the string is empty.

Call Context

Anytime

Compatibility

1.x.x	2.x.x
no	2.0.19 and later

PresetSet

Activate preset.

Arguments

void ***PresetSet****(char[] preset)*

preset	name of the preset to be activated

Timing

This call can take arbitrary time, depending on the previous state of the system. Before further processing, check SEM wait flags using *IsBusy()* call, or use conditional command execution.

Remarks

The preset table is searched for the specified preset name. If it is found, the preset is activated. If the preset is not found, the request is ignored.

Call Context

Depends on the SEM operating state. For example, it is not possible to activate preset containing stage movement, if stage is already busy.

Compatibility

1.x.x	2.x.x
no	2.0.19 and later

Chamber Camera

Most SEMs are equipped with chamber camera – chamberscope. Remote client can ask for video from the camera. The video consists of static images sent in regular intervals.

CameraDisable

Stop the chamberscope image stream.

Arguments

void ***CameraDisable****(void)*

Timing

Executed immediately.

Remarks

The SEM stops sending the stream of camera images. See *CameraEnable()* for details.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.6 and later	yes

CameraEnable

Open camera window in the SEM software, enable the chamberscope image stream.

The camera functions have been introduced in protocol version 1.0.6.

Arguments

void ***CameraEnable****(*
in int channel,
in float zoom,
in float fps,
in int compression
)

channel	reserved, must be 0
zoom	zoom (0.05 – 1.0)
fps	frames per second (0.1 – 5.0)

compression	reserved, must be 0

*void **CameraData(***
in int channel,
in int bpp,
in int width,
in int height,
in char[] data
);

channel	channel index (reserved, should be 0)
bpp	bits per pixel (always 8)
width	image width
height	image height
data	image data, width * height bytes

Timing

If the Chamber View module in the SEM software is closed, it may take several seconds till the client receives valid image. If the Chamber View module is running, the first image is sent immediately.

Remarks

CameraEnable()

The camera image size is not fixed, the actual size is sent in the image callback. Client can ask for image with reduced size (if zoom < 1.0).

CameraData()

Image is uncompressed, grayscale, sent in a byte array. The data starts with top-left corner. The camera image is resampled according the client's request. Each image is sent at once, in a single message.

The *CameraData()* messages are sent through the data channel (same as the scanning data).

Call Context

Call *CameraEnable()* only if camera is disabled.

Also Affected

The Chamber View module in the SEM software is opened if necessary.

Compatibility

1.x.x	2.x.x
1.0.6 and later	yes

CameraGetStatus

Get chamber camera status.

Arguments

*void **CameraGetStatus**(in int channel, out int is_enabled,*
out float zoom, out float fps, out int compression)

channel	channel index (reserved, should be 0)
is_enabled	0 – off, 1 – on
zoom	zoom
fps	frames per second
compression	compression mode, always 0

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Compatibility

1.x.x	**2.x.x**
1.0.6 and later	yes

SEM GUI Control

There are a few functions which control the Vega / Mira / Lyra SEM graphical user interface. Generally speaking, the remote application should not modify or use the SEM user interface, this is undesirable. On the other hand, there are situations when some interaction between the remote client and the server UI is eligible and suitable.

GUIGetScanning

Get scanning status of the SEM GUI.

Arguments

*int **GUIGetScanning**(void)*

return value	0/1 flag - SEM live window is scanning

Timing

Executed immediately.

Remarks

Return value:
0 – no SEM live window is currently scanning
1 – there is a live SEM window currently scanning

When combined with *GUISetScanning(),* it is possible to stop the SEM scanning, execute some procedure, and then restore the scanning back to its original state.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

GUISetScanning

Enable / disable the SEM GUI scanning.

Arguments

*void **GUISetScanning**(in int enable)*

enable	enable flag

Timing

Executed immediately.

Remarks

If *enable* is set to one, scanning in the topmost live window is started. If it is zero, scanning is stopped. See *GUIGetScanning()* for details.

Call Context

Anytime

Also Affected

The result of *GUISetScanning(1)* call is that continual scanning is started. The remote client must not issue any scanning requests until the scanning is stopped (either by invoking *GUISetScanning(0)* or *ScStopScan()*).

Compatibility

1.x.x	2.x.x
1.0.5 and later	yes

Progress Indication

External SharkSEM application can indicate its progress in the SEM software. Arbitrary text and progress indicator overlay window is displayed in the Vega / Mira / Lyra software. This feature can be useful, for example, in case of long-taking spectrum acquisition or similar kind of applications.

There are no priorities among the remote sessions, there is only single progress dialog. “The last wins” strategy takes place.

Progress indication has been introduced in version 2.0.11.

ProgressHide

Hide the progress dialog.

Arguments

*void **ProgressHide**(void)*

Timing

Executed immediately.

Remarks

If the client application fails to call this function, *ProgressHide()* is automatically invoked on application disconnect.

Call Context

Anytime.

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

ProgressPerc

Set progress dialog percentage, change progress bar position.

Arguments

*void **ProgressPerc**(in int perc)*

enable	enable flag

Timing

Executed immediately.

Remarks

Only makes sense if progress dialog shows percentage. The progress indicator range is specified in the *ProgressShow()* call. Decreasing the percentage value is not considered user-friendly.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

ProgressShow

Display progress dialog.

Arguments

*void **ProgressShow**(in char[] title, in char[] text, in int allow_hide, in int marquee, in int perc_min, in int perc_max)*

title	progress dialog title
text	progress dialog text
allow_hide	1 – enable hide button, 0 – disable hide button
marquee	1 – marquee mode, 0 – percentage mode
perc_min	in case of percentage, minimum value
perc_max	in case of percentage, maximum value

Timing

Executed immediately.

Remarks

Shows progress dialog. If the dialog is already shown, its configuration is updated. *Title* and *text* are zero-terminated messages encoded in UTF-8 strings.

If *marquee* is set to one, dialog shows activity but does not indicate what proportion of the task is complete. If *marquee* is set to zero, progress dialog shows specific position within arbitrary range. The range is delimited by *perc_min* and *perc_max* values. The initial indicator position is equal to *perc_min*.

Allow_hide controls whether the dialog contains Hide button. If the Hide button is displayed, SEM user is allowed to hide the progress dialog manually.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

ProgressText

Set progress dialog text.

Arguments

*void **ProgressText**(in char[] text)*

text	UTF-8 encoded text

Timing

Executed immediately.

Remarks

Text specifies a new progress dialog text which replaces the original one.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

Power Interface

Power interface API allows an external application to control the power state of the microscope. The power saving capabilities depend on the actual microscope model.

Before switching the power mode, make sure to check what are the available modes supported.

Power interface has been introduced in version 2.0.11.

PowerStateEnum

Get list of power saving modes.

Arguments

*unsigned int **PowerStateEnum**(void)*

return value	power saving modes, bit mask

Timing

Executed immediately.

Remarks

Bit description:

0	normal operating mode
1	stand by
2 – 30	reserved, zero
31	soft off

In normal operating mode, microscope can observe samples.

In mode 1 (stand by), pumping is switched off, some parts of electronics are switched off as well. However, the microscope can be switched back to normal operating mode.

In mode 31 (soft off), all software-controllable supplies are off, including PC. The microscope cannot be resurrected back to normal operating mode without user intervention.

Typically, only tungsten-based systems support mode 31, because field emission guns require permanent HV supply operation.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

PowerStateGet

Get current power state.

Arguments

*int **PowerStateGet**(void)*

return value power state, integer index

Timing

Executed immediately.

Remarks

Value:

0 normal operating mode
1 stand by

For more details, refer to *PowerStateEnum()*. Note that state 31 is not supported, because the PC and the SharkSEM server are inactive in this state.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.11 and later

PowerStateSet

Set power state.

Arguments

*void **PowerStateSet**(int state)*

state requested power state, integer index

Timing

Synchronous execution, may take a few seconds until the operation is completed.

Remarks

Value:

0	normal operating mode
1	stand by
31	soft off

For more details, refer to *PowerStateEnum()*. Once state 31 is entered, the microscope cannot be remotely switched back.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	**2.x.x**
no	2.0.11 and later

Sample Stage Layout

The SEM sample stage has variable layout, depending on the holder selected by user. For example, there can be GSR sample holder, standard 7-position sample holder, TIMA sample holder for mineral samples, etc.

Using this API, the remote application can read all predefined sample positions, their shapes, and also read/write user-defined sample labels.

SmplGetCount

Arguments

*int **SmplGetCount**(void)*

return value	number of predefined samples

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

SmplEnum

Read list of samples, including all details.

Arguments

*map **SmplEnum**(in int coord_system)*

coord_system	coordinate system (reserved, must be 0)
return value	map, list of samples

Timing

Executed immediately.

Remarks

The *coord_system* parameter is reserved for future use, must be zero.

Example output map:

```
layout.name=LPC-00-P (Mineralogy Sample Holder)
sample.0.shape=rectangle
sample.0.width=11.500
sample.0.height=16.500
sample.0.angle=0.00
sample.0.x=40.000
sample.0.y=-41.000
sample.0.r=0.00
sample.0.id=1
sample.0.type=0
sample.0.label=Moon Dust
sample.1.shape=circle
sample.1.diameter=12.700
sample.1.x=-40.000
sample.1.y=-41.000
sample.1.r=0.00
sample.1.id=2
sample.1.type=0
sample.1.label=
```

There is a continuous sequence of samples, indexing starts from zero. Each sample has following attributes:

shape	either **rectangle** or **circle**
width *height* *angle*	if *shape* is **rectangle**, these three values specify the object dimensions. *Width* and *height* are in mm, *angle* in degrees (counterclockwise)
diameter	in case of **circle**, diameter of the circle
x, y, r	absolute position of the sample center (stage coordinates). *X* and *y* are in mm, *r* in degrees
id	sample identification, for example “1”, “2”, “FC” , UTF-8 encoded
type	sample type (integer). For more details, refer to SmplGetType()
label	user-defined sample label, UTF-8 encoded

Client application must silently ignore unknown rows.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

SmplGetHldrName

Get name of the whole sample holder.

Arguments

char[] ***SmplGetHldrName****()*

return value	sample holder name, UTF-8 encoded

Timing

Executed immediately.

Remarks

-

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.13 and later

SmplGetId

Get sample identification string.

Arguments

char[] ***SmplGetId****(in int index)*

index	sample index
return value	sample id, UTF-8 encoded

Timing

Executed immediately.

Remarks

If *index* is out of range, empty string is returned.

Sample label can be user defined, while sample id is fixed and it is not possible to change it. See also *SmplGetLabel()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.13 and later

SmplGetLabel

Get user-defined sample label.

Arguments

char[] ***SmplGetLabel****(in int index)*

index	sample index
return value	user-defined sample label, UTF-8 encoded

Timing

Executed immediately.

Remarks

If *index* is out of range or if the label is undefined, empty string is returned.

See also *SampleSetLabel()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

SmplGetPosition

Get sample coordinates, in the stage coordinate system.

Arguments

*void **SmplGetPosition**(in int index, in int coord_system,*
out float x, out float y, out float r)

index	sample index
coord_system	coordinate system (reserved, must be 0)
x	stage **x** coordinate [mm]
y	stage **y** coordinate [mm]
r	stage **r** coordinate [deg]

Timing

Executed immediately.

Remarks

Only X, Y, R are stored per-sample. Neither stage tilt nor Z is stored, because these manipulator axes are not used for sample navigation.

The stage coordinates can be simply passed to *StgMoveTo()* function.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

SmplGetShape

Get sample shape and dimensions.

Arguments

*void **SmplGetShape**(in int index, out int shape, out float p1, out float p2, out float p3)*

index	sample index
shape	0 - circle, 1 - rectangle
p1, p2, p3	sample metrics, depending on *shape*

Timing

Executed immediately.

Remarks

Currently, only rectangle and circle are supported by stage layouts. Depending on the shape, the parameters have following meaning.

Shape		p1	p2	p3
circle	0	diameter [mm]	-	-
rectangle	1	width [mm]	height [mm]	angle [deg]

Rectangle *p3* parameter (rotation) specifies how much is the rectangle rotated, relatively to its center, counterclockwise.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

SmplGetType

Get sample type. The type specifies the standard sample usage, for example Faraday cup, EDS standard, etc.

Arguments

*int **SmplGetType**(in int index)*

index	sample index
return value	sample type

Timing

Executed immediately.

Remarks

If *index* is out of range, zero is returned.

Sample types

0	any sample
1	Faraday cup
2	EDS standard

The client application must be able to handle unknown sample types.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.13 and later

SmplSetLabel

Set user-defined sample label.

Arguments

*void **SmplSetLabel**(in int index, in char[] label)*

index	sample index
label	user-defined sample label, UTF-8 encoded

Timing

Executed immediately.

Remarks

If *index* is out of range or if the label is undefined, call is ignored. Once set, the label becomes visible in the SEM GUI. Previous label, if any, is discarded.

See also *SampleSetLabel()*.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.12 and later

Miscellaneous

ChamberLed

Switch the chamber illumination LED on and off.

Arguments

*void **ChamberLed**(in int onoff)*

onoff	0 – off, 1 – on

Timing

Typically 10 ms to execute (wait C).

Remarks

If there is a detector sensitive to the LED light, detector recovery time must be considered carefully.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

Delay

Delay execution.

Arguments

*void **Delay**(in int delay)*

time	time in [ms]

Timing

Wait C is set for given period.

Remarks

This function does not delay execution. It only sets Wait C flag. Thus, following commands are delayed only if they have Wait C flag set to one.

When the specified time has elapsed, the Wait C flag is reset.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
yes	yes

GetDeviceParams

Get operating conditions of SEM or FIB. Remote applications can store the whole parameter list for future reference.

Arguments

*map **GetDeviceParams**(in int param_set)*

param_set	parameter set	
	0	common parameters
	1	SEM parameters
	2	FIB parameters
return value	device parameters	

Timing

Executed immediately.

Remarks

Parameter set is a list of *key=value* pairs, each on a separate line. The parameter set is same as in TESCAN image header, except a few items that are missing.

Call Context

Anytime.

Compatibility

1.x.x	2.x.x
no	2.0.6 and later

GetUPSStatus

Get status of microscope UPS (uninterruptible power supply) unit.

Arguments

*int **GetUPSStatus**(void)*

return value status

Timing

Executed immediately.

Remarks

There can be a UPS connected to the microscope. This function queries the state of mains power line and status of the UPS battery.

Following table describes statuses. It is recommended that SharkSEM react to some of the statuses accordingly.

Status	Meaning	Action
-2	unknown	-
-1	UPS not installed	-
0	running on mains, battery OK	-
1	running on battery, battery OK	prepare for termination
2	running on mains, battery low	prepare for termination
3	running on battery, battery low	terminate immediately

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.5 and later

IsBusy

Check if some SEM subsystem(s) are busy.

Arguments

*int **IsBusy**(in unsigned int wait_flags)*

wait_flags wait flag mask

return value 1 – busy, 0 – not busy

Timing

Executed immediately.

Remarks

Each command header may contain “wait flags”, which specify what conditions shall be fulfilled before the command is executed.

It is sometimes useful to explicitly query if some wait condition is fulfilled. This function accepts wait flag mask as an argument, returns status if such condition is currently fulfilled or not.

The list of flags is same as in the SharkSEM message header.

Bit	Value	
0-7	Reserved, must be 0	
8	Wait (SEM)	A – e-beam scanning
9		B – stage
10		C – e-beam optics
11		D – e-beam automatic procedure
12	Wait (FIB)	E – i-beam scanning
13		F – i-beam optics
14		G – i-beam automatic procedure
15-31	Reserved, must be 0	

Note: even if the SEM response is “not busy”, subsequent command may be blocked, because the actual SEM state may change in the meantime.

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.9 and later

IsLicenseValid

Check if specific SharkSEM extension module is licensed.

Arguments

*int **IsLicenseValid**(in char[] module)*

module	module name (zero terminated string)
return value	1 – license valid, 0 – license not valid

Timing

Executed immediately.

Remarks

There exist SharkSEM extensions which require additional license. It can be determined if the license for specified module is valid or not.

Example module name: “SharkSEM RCA”

Call Context

Anytime

Also Affected

-

Compatibility

1.x.x	2.x.x
no	2.0.3 and later