The Scorbot-ER 4U

Function reference and notes for the usbc.dll

By Jacob Cornelius Mosebo

Table of Contents

Introduction.....5
Notes.....6
Concepts of the ER-4 Scorbot.....6
Initialization, Homing and Control.....6
Vectors and Points.....6
Setup - Files and folders.....8
Files needed to use the Scorbot-ER 4U:.....8
Additional files.....8
File versions.....9
Function Reference.....10
Configuration Functions.....10
Initialization.....10
SetParameterFolder.....11
GetParameterFolder.....12
CloseUSBC.....13
GetVersion.....14
IsEmergency.....15
IsTeachMode.....16
IsOnlineOk.....17
GetConfig.....18
ChangeConfig.....19
EnumPeriph.....20
SetJawMetric.....21
SetGripperStatus.....22
SetGripperType.....23
SaveConfig.....24
GetUSBDeviceNumber.....25
USBDeviceArrival.....26
USBDeviceRemoved.....27
USBDeviceQueryRemove.....28
GetMode.....29
MyPumpMessages.....30
WaitUSBCCommunication.....31
Movement Functions.....32
WatchMotion.....32
EnterManual.....33
MoveManual.....34
CloseManual.....35
Time.....36
Speed.....37
SpeedLin.....38
MoveTorque.....39
MoveTorque.....40
MoveJoint.....41
MoveLinear.....42
MoveCircularVect.....43
MoveCircularPoint.....44
MoveSplineJoint.....45
MoveSplineJointTime.....46
MoveSplineLinear.....47
OpenGripper.....48
CloseGripper.....49

Stop....50
Velocity....51
JawPerc....52
JawMetric....53
GetJaw....54
Control....55
WatchControl....56
CloseWatchControl....57
GetConStatus....58
GetMotionStatus....59
WatchHoming....60
Home....61
SetHome....62
GetTPInfo....63
WatchJoint....64
CloseWatchJoint....65
SetJoint....66
Impact....67
Position Functions....68
GetPointsPeriphID....68
ClearPointsAttributes....69
AddPoints....70
DefineVector....71
RenameVector....72
ResetPoints....73
DeletePoint....74
Here....75
Teach....76
SetEncoders....77
SetJoints....78
IsPointExist....79
IsTheBasePosition....80
GetNextPoint....81
GetPointInfo....82
AttachPosit....83
GetCurrentPosition....84
Monitoring Commands....85
ShowEnco....85
CloseEnco....86
ShowXYZ....87
CloseXYZ....88
ShowJoint....89
CloseJoint....90
ShowPositErr....91
CloseEnco....92
ShowHomeSwitches....93
CloseHomeSwitches....94
ShowTorque....95
CloseTorque....96
Input/Output Commands....97
WatchDigitalInp....97
CloseWatchDigitalInp....98
WatchDigitalOut....99
CloseWatchDigitalOut....100
GetDigitalInputs....101

GetDigitalOutputs..... ..102
SetDigitalOutput..... 103
ForceStatusDigitalInput..... 104
GetForceStatusDigitalInput..... 105
EnableDigitalInput..... 106
DisableDigitalInput..... 107
GetDigitalInputEnabledStatus..... 108
GetHomeSwitch..... 109
WatchAnalogInp..... 110
CloseWatchAnalogInp..... 111
WatchAnalogOut..... 112
CloseWatchAnalogOut..... 113
GetAnalogInput..... 114
ForceStatusAnalogInput..... 115
ForceAnalogInput..... 116
GetForceStatusAnalogInput..... 117
SetAnalogOutput..... 118
Appendix A: Managed vs. Unmanaged code..... 119

Introduction

I decided to write this, since i did not find the documentation available to me sufficient and complete, neither is this, but it is a step in the right direction. Should you find any errors or information missing (you probably will), edit this document yourself.

The reference does not describe all functions in the USBC libary, but it is more than sufficient to control the robot and its peripherals. Also much of the functionality given by the libary can be done by the user, using other functions.

Notes

Concepts of the ER-4 Scorbot

Initialization, Homing and Control

In order to use any function except for `Initialization, IsOnlineOk` and few other configuration functions, the robot needs to be initialized with the `Initialization` funtion. This allows the use of all functions that does not result in movement.
The next move might be to turn control on, since the only movement function that will work without control turned on is `MoveTorque` which sets the PWM values to the motor manually.
Even though control is on, most movement functions will still not work, since the robot does not know excatly where it is. Actually `MoveJoint, MoveLinear and MoveManual` will only work partially, the rest will not.
Homing the robot will set all axes in a default position, allow the use of the rest of the movement functions and many callback functions.

Vectors and Points

The robot uses coordinates to move. These are defined in points, which again is defined in vectors.
A vector is an array of points, with a name, and each point has a numbered position in the vector.
Vectors and points can be defined right after the robot is initialized.
Use `DefineVector` to add a vector into the robots memory, and then use either `Teach` or `Here` to define a point in the vector(s). Alternatively points can be loaded into the memory from a file with `AddPoints`.
Additionally the homing process saves a point in a default named vector, with its default position (can later be owerwritten with `SetHome`), this can be seen by saving the points in a file by using `SavePoints`.

Point types

BS_XYZ_AA
This type of point is absolute, it has 5 values, x, y, z, pitch and roll.

REL_XYZ_CRNT
This type of point is relative to the current position, it has the same values as the absolute point.

ABS_JOINT
This type of point is also absolute, but the values defines the angles of the joints instead of coordinates. It also has 5 values, base, shoulder, elbow, pitch and roll.

REL_JOINT_CRNT
This type is relative relative to the current position, values are the same as ABS_JOINT.

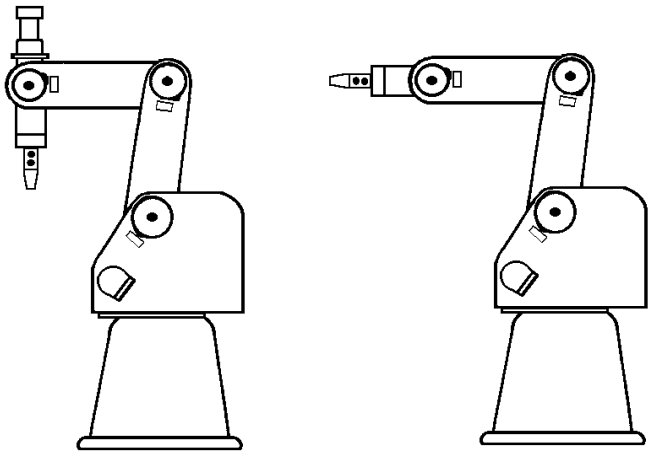
XYZ Values
Axis 1: X-coord in micrometers (ie. 50,000 = 5 cm)
Axis 2: Y-coord in micrometers
Axis 3: Z-coord in micrometers
Axis 4: Pitch angle in degrees / 1,000 (ie. 90,000 = 90 degrees)
Axis 5: Roll angle in degrees / 1,000

It should be noted that (0, 0, 0, x, x) is at the bottom-center of the robot.

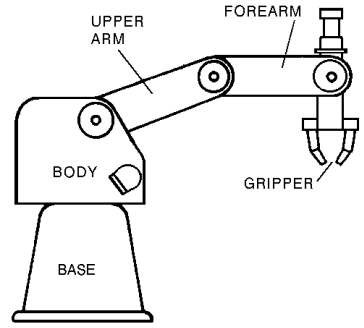
Home is close to (169000, 0, 500000, -63000, 0).

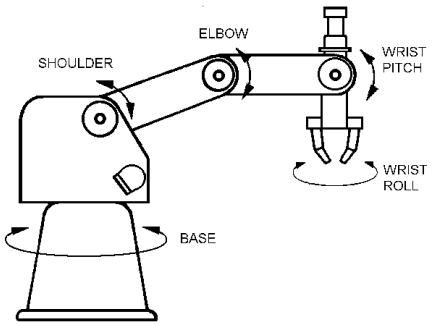
Joint Values

Axis 1: Base/torso angle in degrees / 1000 (ie. 90,000 = 90 degrees)
Axis 2: Shoulder angle in degrees / 1000
Axis 3: Elbow angle in degrees / 1000
Axis 4: Pitch angle in degrees / 1000
Axis 5: Roll angle in degrees / 1000



The left image shows the robot with a pitch angle of -90,000. The right image shows the robot with a pitch angle of 0.





Setup - Files and folders

Files needed to use the Scorbot-ER 4U:

This section focuses on files needed to use the usbc.dll properly.

usbc.dll
What all the fuss is about.
Put this file where your program can find it, such as where your program is, or the windows folder or its system folders.

usbc.ini
Put this together with the usbc.dll.
This tells the usbc.dll where to find er4conf.ini file, and where to put the log file.
It also sets the monitoring value. It tells the delay between callbacks, in miliseconds, for many monitoring functions.

er4conf.ini

This files tells the libary where to find the parameter folders, and the names of the configuration files for the individual axis. The default path for this folder is the same as usbc.ini.

The PAR folder

This folder contains all the configuration files for the individual axes and peripherals. The default location is PAR\ER4u\$Default.
Changing the configuration files can potentially damage the robots motors.

Additional files

These files are not needed to use the robot, but they do provide some usefull information. They are however needed if you should be developing a wrapper for the usbc.dll in managed C++.

usbc.h

This is the header file for the usbc.dll. Here you will find the complete definition for all its functions.

usbcdef.h

This file contains definition for the constants and structures the usbc.dll uses. Which you will need no matter what platform you choose to develope from.

extern.h

This contains the definion for the ErrorInfo class.

error.h

Full definition of errors from the usbc.dll

usbc.lib

Should you be developing in C++ you will need to link to this file.
Also add the `_AFXDLL` to your preprossor definitions, and `/D "_AFXDLL"` to project options.

File versions

Make shure you have the right version of the USBC.dll and ER4conf.ini files. Many problems can be traced back to just having the wrong files.

USBC.dll	452 KB	10-05-2004 17:20
ER4conf.ini	2 KB	10-05-2004 17:29

Function Reference

Configuration Functions

Initialization

Description:

Finds the USB device connected to the robot and initializes the connection between the robot and the computer.

Syntax:

```
bool Initialization( short sMode [, short sSystemtType], CallBackFun fnInitEnd,
       CallBackFun fnErrMessage )
```

Parameters:

`short sMode`

`INIT_MODE_DEFAULT = 0`	selects last used mode (from ini file)
`INIT_MODE_ONLINE = 1`	force online mode
`INIT_MODE_SIMULAT = 2`	selects simulation mode

`[optional] short sSystemType`

`DEFAULT_SYSTEM_TYPE = 0`	let the libary detect the robot type
`ER4USB_SYSTEM_TYPE = 41`	define robot type as ER-4 Scorbot with USB connection

`CallBackFun fnInitEnd`

Any pointer to a function, function must be of type `void function(ConfigData *)` to work properly.
This function is called on succesful initialization.

`CallBackFun fnErrMessage`

Any pointer to a function, function must be of type `void function(ErrorInfo *)` to work properly.
This function is called if an detectable error has occured.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

The functions return value does not indicate a succesful initialization, only a call to `fnInitEnd` indicates a succesfull initialization.
Initialization may not call back at all, it is a good idea to wrap the call to this function around a timer (5 seconds should be enough).
The connection between the robot and the libary is uniquely tied to the process calling this function.
Simulation mode allows you to use the dll, as the robot was online and homed.
Only one connection to the robot is allowed/possible.
Full definition of constants can be found in USBCDEF.h.
Definition for ConfigData can be found in USBCDEF.h.
Definition for ErrorInfo can be found in Extern.h.
`CallBackFun` is a typedef for `(void)*`.

SetParameterFolder

Description:

Set the folder which holds the files with values the motors on the robot.

Syntax:

```
bool SetParameterFolder(char * szFolderName)
```

Parameters:

`char * szFolderName`
: null terminated string that holds the new path

Return value:

Returns true if a new folder has been set.

Notes:

The parameter folders are defined in ERCONF.INI.

GetParameterFolder

Description:

Gets the folder which holds the files with values the motors on the robot.

Syntax:

```
bool GetParameterFolder(char * szFolderName)
```

Parameters:

`char * szFolderName`
a string to be filled with the existing path for the parameter folder

Return value:

Returns true if the string has been filled, false otherwise.

Notes:

The parameter folders are defined in ERCONF.INI.

CloseUSBC

Description:

closes the connection (created with Initialization) to the robot.

Syntax:

```
void CloseUSBC()
```

Parameters:

none

Return value:

none

Notes:

none

GetVersion

Description:

?

Syntax:

```
bool GetVersion(unsigned long * pulVersion)
```

Parameters:

```
ULONG * pulVersion
```
?

Return value:

?

Notes:

```
none
```

IsEmergency

Description:

tells wether the emergency button has activated or not.

Syntax:

```
bool IsEmergency()
```

Parameters:

none

Return value:

Returns true if the emergency button has been activated, false otherwise.

Notes:

none

IsTeachMode

Description:

?

Syntax:

```
bool IsTeachMode()
```

Parameters:

none

Return value:

Returns true if ?, false otherwise.

Notes:

none

IsOnlineOk

Description:

Tells wether the robot is online.

Syntax:

```
bool IsOnlineOk()
```

Parameters:

none

Return value:

Returns true if the robot is online, false otherwise.

Notes:

If under simulation mode, this function returns false.

GetConfig

Description:

Gets the configuration for the currently connection robot

Syntax:

```
bool GetConfig(ConfigData &Config)
```

Parameters:

`ConfigData &Config`
reference to the structure to be filled with config data

Return value:

Returns true if the `Config` object has been succesfully filled, false otherwise.

Notes:

Requires an initialized robot connection.
Definition for ConfigData can be found in USBCDEF.h.

ChangeConfig

Description:

?

Syntax:

```
bool ChangeConfig(char * szDeviceID1, char * szDeviceID2, CallBackFun fnConfigAvail)
```

Parameters:

`char * szDeviceID1`
?

`char * szDeviceID2`
?

`CallBackFun fnConfigAvail`
?

Return value:

?

Notes:

`CallBackFun` is a typedef for `(void)*`.

EnumPeriph

Description:

?

Syntax:

```
bool EnumPeriph( short sDeviceNo, short sSystemType, char * szDeviceID )
```

Parameters:

```
short sDeviceNo
```
?

```
short sSystemType
```
?

```
char * * szDeviceID
```
?

Return value:

?

Notes:

SetJawMetric

Description:

?

Syntax:

```
bool SetJawMetric( short sValue )
```

Parameters:

```
short  sValue
```
?

Return value:

?

Notes:

SetGripperStatus

Description:

?

Syntax:

```
bool SetGripperStatus( bool bIsOpen )
```

Parameters:

```
bool bIsOpen
```
?

Return value:

?

Notes:

SetGripperType

Description:

?

Syntax:

```
bool SetGripperType( short sGripType, short sOutNum, bool bOutState, short sDelay )
```

Parameters:

`short sGripType`
?

`short sOutNum`
?

`bool bOutState`
?

`short sDelay`
?

Return value:

?

Notes:

SaveConfig

Description:

?

Syntax:

```
short SaveConfig()
```

Parameters:

none

Return value:

?

Notes:

GetUSBDeviceNumber

Description:

?

Syntax:

```
bool GetUSBDeviceNumber( int * iDevice )
```

Parameters:

`int * iDevice`
?

Return value:

?

Notes:

USBDeviceArrival

Description:

?

Syntax:

```
void USBDeviceArrival()
```

Parameters:

none

Return value:

?

Notes:

USBDeviceRemoved

Description:

?

Syntax:

```
void USBDeviceRemoved()
```

Parameters:

none

Return value:

?

Notes:

USBDeviceQueryRemove

Description:

?

Syntax:

```
void USBDeviceQueryRemove()
```

Parameters:

none

Return value:

?

Notes:

GetMode

Description:

?

Syntax:

```
short GetMode()
```

Parameters:

none

Return value:

?

Notes:

MyPumpMessages

Description:

?

Syntax:

```
bool MyPumpMessages()
```

Parameters:

none

Return value:

?

Notes:

WaitUSBCCommunication

Description:

?

Syntax:

```
bool WaitUSBCCommunication()
```

Parameters:

none

Return value:

?

Notes:

Movement Functions

WatchMotion

Description:

Sets the callback functions for movement events.

Syntax:

```
CallBackFun WatchMotion( CallBackFun fnMotionEnd, CallBackFun fnMotionStart)
```

Parameters:

`CallBackFun  fnMotionEnd`
: A pointer to a function, the function must be of type `void function(char *)` to work properly. This function is called when a motion has ended.
A `null` value will stop callbacks of this type.

`CallBackFun  fnMotionStart`
: A pointer to a function, the function must be of type `void function(char *)` to work properly. This function is called when a motion has begun.
A `null` value will stop callbacks of this type.

Return value:

Pointer to the previous callback function.

Notes:

`CallBackFun` is a typedef for `(void)*`.
`UCHAR` is a typedef for `unsigned char`

When a callback is made, it is given a pointer to a `char`. The value of this char denotes on what part of the robot the event has occured:
'0'-'7' for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

If both parameters are set with non `null` values, a pointer to the previous `fnMotionEnd` function will be returned.

EnterManual

Description:

Enter manual movement mode. Must be called in order to use the MoveManual function

Syntax:

```
bool EnterManual( short sManualType )
```

Parameters:

```
short  sManualType
       MANUAL_TYPE_ENC = 0 selects movement control by axes
       MANUAL_TYPE_XYZ = 1 select movement control by coordinates
```

Return value:

Returns true if the robot has succesfully entered manual movement mode, false otherwise.

Notes:

The robot must be homed in order to use movement by coordinates

Full definition of constants can be found in USBCDEF.h.

MoveManual

Description:

Manually moves robot.

Syntax:

```
bool MoveManual( unsigned char ucAxis, long lVelocity )
```

Parameters:

`bool ucAxis`
: Axis which to move.

`long lVelocity`
: The speed for the movement

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

In order to use movement by coordinates, the robot must be homed.

The axis parameters use differs depending on movement mode (set by EnterManual).
Under movement by axis:
0 - Base or Torso
1 - Shoulder
2 - Elbow
3 - Wrist-Pitch
4 - Wrist-Roll
5 - Gripper
6 - Unused
7 - Conveyorbelt

Under movement by coordinates:
0 - X
1 - Y
2 - Z
3 - Pitch
4 - Roll
5 - Unused
6 - Unused
7 - Unused

MoveManual does not produce any callbacks.

MoveManual need control to be turned on, in order to function properly.
MoveManual automaticly turns on control, if control is off, but cancels all other funtionality during that call.
MoveManual automaticly enters manual movement mode, if the robot is not in manual movement mode, but cancels all other funtionality during that call (control takes precedence over manual movement mode).

CloseManual

Description:

Exits manual movement mode.

Syntax:

```
bool CloseManual( )
```

Parameters:

none

Return value:

Returns true if movemanully has been succesfully exited, false otherwise.

Notes:

none

Time

Description:

Sets the time future movement should take.

Syntax:

```
bool Time( unsigned char ucGroup, long lTime )
```

Parameters:

`bool ucGroup`
: Axis group to which the time should be applied
'&' for all axes
'0'-'7' for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

`long lTime`
: Time in milliseconds

Return value:

Returns true if time has been succesfully set, false otherwise.

Notes:

If the time set is invalid (extremly low or high), the closest possible speed will be automaticly set.

Speed

Description:

Sets the speed future movement should take.

Syntax:

```
bool Speed( unsigned char ucGroup, long lSpeed )
```

Parameters:

`bool ucGroup`
: Axis group to which the time should be applied
'&' for all axes
'0'-'7' for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

`long lSpeed`
: Speed in percent of max speed

Return value:

Returns true if the speed has been succesfully set, false otherwise.

Notes:

If the speed set is invalid (extremly low or high), the closest possible speed will be automaticly set (0 is not valid).

SpeedLin

Description:

???

Syntax:

```
bool SpeedLin( unsigned char ucGroup, long lSpeedLin )
```

Parameters:

`bool ucGroup`
: Axis group to which the time should be applied
'&' for all axes
0-7 for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

`long lSpeedLin`
: Speed in percent of max speed

Return value:

Returns true if the speed has been succesfully set, false otherwise.

Notes:

???

MoveTorque

Description:

Manually sends PWM values directly to the motors.

Syntax:

```
bool MoveTorque( long * plTorqueArray, short sTorqueArrayDim )
```

Parameters:

`long * plTorqueArray`
Pointer to the array filled with PWM values.

`long sTorqueArrayDim`
Size of `plTorqueArray`. Standard is 8.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

CAUTION: There is no impact protection when using this function, and can potentially burn the motors.

Control must be off while using this function.

MoveTorque

Description:

Manually sends PWM values directly to a motor.

Syntax:

```
bool MoveTorque( unsigned char ucAxis, long lTorque )
```

Parameters:

`unsigned char ucAxis`
: Axis which to moveAxis group to which the time should be applied

`long lTorque`
: PWM value.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

CAUTION: There is no impact protection when using this function, and can potentially burn the motors.

Control must be off while using this function.

MoveJoint

Description:

Moves the robot without care for path

Syntax:

```
bool MoveTorque( char * szVectorName, short sPointNumber
                    char * szVectorNameB, short sPointNumberB )
```

Parameters:

`char * szVectorName`
Name of the vector where to find the position

`short sPointNumber`
Point in the vector which to move to.

`char * szVectorNameB`
Name of the vector where to find the secondary position

`short sPointNumberB`
Point in the vector which to move to.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

if `szVectorNameB` and `sPointNumberB` are set to null and 0 respectively, the robot only moves to the first position.

MoveLinear

Description:

Moves the robot in a linear path

Syntax:

```
bool MoveLinear( char * szVectorName, short sPointNumber
                    char * szVectorNameB, short sPointNumberB )
```

Parameters:

`char * szVectorName`
Name of the vector where to find the position

`short sPointNumber`
Point in the vector which to move to.

`char * szVectorNameB`
Name of the vector where to find the secondary position

`short sPointNumberB`
Point in the vector which to move to.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

if `szVectorNameB` and `sPointNumberB` are set to null and 0 respectively, the robot only moves to the first position.

MoveCircularVect

Description:

Moves the robot in a circular motion

Syntax:

```
bool MoveCircularVect( char * szVectorName, short sThroughPointNumber,
                           short sTargetPointNumber, char * szVectorNameB,
                           short sPointNumberB )
```

Parameters:

`char * szVectorName`
Name of the vector where to find the position

`short sThroughPointNumber`
Point in the vector which to move via.

`short sTargetPointNumber`
Point in the vector which to move to.

`char * szVectorNameB`
Name of the vector where to find the secondary position

`short sPointNumberB`
Point in the vector which to move to.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

if `szVectorNameB` and `sPointNumberB` are set to null and 0 respectively, the robot only moves to the first position.

MoveCircularPoint

Description:

???

Syntax:

```
bool MoveCircularPoint( unsigned char ucGroup, char * szThroughVectorName,
                        char * sTargetVectorName )
```

Parameters:

```
UCHAR ucGroup
```
??

```
char * szThroughVectorName
```
??

```
char * sTargetVectorName
```
??

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

???

MoveSplineJoint

Description:

???

Syntax:

```
bool MoveSplineJoint( char * szVectorName, short sFirstPointNumber,
                          short sLastPointNumber )
```

Parameters:

`char * szVectorName`
??

`short sFirstPointNumber`
??

`short sLastPointNumber`
??

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

???

MoveSplineJointTime

Description:

???

Syntax:

```
bool MoveSplineJoint( char * szVectorName, short sFirstPointNumber,
                          short sLastPointNumber, long lTime )
```

Parameters:

```
char * szVectorName
```
??

```
short sFirstPointNumber
```
??

```
short sLastPointNumber
```
??

```
long lTime
```
??

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

???

MoveSplineLinear

Description:

???

Syntax:

```
bool MoveSplineLinear( char * szVectorName, short sFirstPointNumber,
                           short sLastPointNumber )
```

Parameters:

`char * szVectorName`
??

`short sFirstPointNumber`
??

`short sLastPointNumber`
??

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

???

OpenGripper

Description:

Completely opens the gripper.

Syntax:

```
bool OpenGripper( )
```

Parameters:

none

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

CloseGripper

Description:

Completely closes the gripper.

Syntax:

```
bool CloseGripper( )
```

Parameters:

none

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

Stop

Description:

Stops robot movement.

Syntax:

```
bool Stop( unsigned char ucGroup)
```

Parameters:

```
bool ucGroup
```

Which axis group to stop movements on.
'&' for all axes
'0'-'7' for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

Velocity

Description:

???

Syntax:

```
bool Velocity( UCHAR ucAxis, short sPercent )
```

Parameters:

```
unsigned char ucAxis
```
?

```
short sPercent
```
?

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

JawPerc

Description:

Moves the grippers fingers to a given width in percent of the maximum width.

Syntax:

```
bool JawPerc( short sPercent )
```

Parameters:

`short sPercent`
: Width between the fingers in percent

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

JawMetric

Description:

Moves the grippers fingers to a given width in mm.

Syntax:

```
bool JawMetric( short sValue )
```

Parameters:

`short sPercent`
: Width between the fingers in mm.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

GetJaw

Description:

Gets the width between the grippers fingers.

Syntax:

```
bool GetJaw( short * psPerc, short * psMetricValue )
```

Parameters:

`short * psPerc`
: Pointer to the variable to be set to the idth between the fingers in percent.

`short * psMetricValue`
: Pointer to the variable to be set to the idth between the fingers in mm.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none

Control

Description:

Turns control on or off for a specific axis group.

Syntax:

```
bool Control( unsigned char ucGroupAxis, bool bIsOn )
```

Parameters:

`unsigned char ucGroupAxis`
: Axis group affected.
'A' for robot.
'B' for peripherals.
'&' for all axes.

`bool bIsOn`
: Control status.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

Control must be turned on, to use allmost all movement functions properly functions properly. MoveTorque is the exception.

WatchControl

Description:

Watch control status on the robot.

Syntax:

```
bool WatchControl( CallBackFun fnWatchControl )
```

Parameters:

`CallBackFun fnInitEnd`

Any pointer to a function, function must be of type `void function(unsigned long *)` to work properly.
This function is called when control status changes.
A `null` value will stop callbacks of this type, this has the same effect as `CloseWatchControl()`.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

The callback functions parameter `(unsigned long *)` can alternatively be cast as a `(char *)`, where each bit denotes the status of the corresponding axis.

CloseWatchControl

Description:

Stops watch control status on the robot.

Syntax:

```
bool CloseWatchControl( )
```

Parameters:

none.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none.

GetConStatus

Description:

Gets the control status on a specific axis group.

Syntax:

```
bool GetConStatus( unsigned char ucGroupAxis, bool * pbIsOn )
```

Parameters:

`unsigned char ucGroupAxis`
: Axis group affected.
'A' for robot.
'B' for peripherals.
'G' for gripper movements
'&' for all axes.
'0'-'7' for axis movements

`bool * pbIsOn`
: Pointer to the variable to be set to the control status.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none.

GetMotionStatus

Description:

Gets the motion status on a specific axis group.

Syntax:

```
bool GetMotionStatus( unsigned char ucGroup )
```

Parameters:

`bool ucGroup`

Axis group to which the time should be applied
'&' for all axes
'0'-'7' for axis movements
'A' for robot movements
'B' for peripheral movements
'G' for gripper movements

Return value:

Returns true if the given axis group is in motion, false otherwise.

Notes:

none.

WatchHoming

Description:

Watch the homing process.

Syntax:

```
bool WatchHoming( CallBackFun fnHomingNotif )
```

Parameters:

`CallBackFun  fnHomingNotif`

Any pointer to a function, function must be of type `void function(unsigned char *)` to work properly.
This function is called when the homing process reaches a milestone.
A `null` value will stop callbacks of this type.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

The callback functions parameters tells the status of the homing process.
The values are:
0xff: Homing started
1 - 8: Axis 1 - 8 is being homed
0x40: Homing ended
These values does not indicate succes or failure.

Home

Description:

Homes the robot.

Syntax:

```
bool Home( unsigned char ucGroupAxis, CallBackFun fnHomingNotif )
```

Parameters:

`unsigned char ucGroupAxis`

Axis group affected.
'A' for robot.
'B' for peripherals.
'G' for gripper movements
'&' for all axes (normally used).
'0'-'7' for axis movements

`CallBackFun fnHomingNotif`

Any pointer to a function, function must be of type `void function(unsigned char *)` to work properly.
This function is called when the homing process reaches a milestone.
A `null` value will stop callbacks of this type.
Setting this parameter has the same effect as `WatchHoming()`.

Return value:

Returns true if the homing process was a succes, false otherwise.

Notes:

The callback functions parameters tells the status of the homing process.
The values are:
0xff: Homing started
1 - 8: Axis 1 - 8 is being homed
0x40: Homing ended
These values does not indicate succes or failure.

SetHome

Description:

Sets the home position to the current position.

Syntax:

```
bool SetHome( unsigned char ucgroupAxis )
```

Parameters:

`unsigned char ucGroupAxis`

Axis group affected.
'A' for robot.
'B' for peripherals.
'G' for gripper movements
'&' for all axes (normally used).
'0'-'7' for axis movements

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

Calling this function does NOT force the robot homed status, it only saves the position in a base vector.

GetTPInfo

Description:

???

Syntax:

```
bool GetTPInfo( unsigned char & code, char * szText )
```
USBC_API BOOL GetTPInfo(UCHAR &code, char *szText);

Parameters:

```
unsigned char & code
```
???

```
char * szText
```
???

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

???

WatchJoint

Description:

Watch the values for each joint on the robot.

Syntax:

```
bool WatchJoint( CallBackFun fnWatchJoint )
```

Parameters:

`CallBackFun fnWatchJoint`

Any pointer to a function, function must be of type `void function(RobotData *)` to work properly.
This function is called periodicly.
A `null` value will stop callbacks of this type.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

RobotData is a typedef for `long[8]`.
The interval between each callback is defined in USBC.ini in miliseconds.

CloseWatchJoint

Description:

Stops watch the joints status on the robot.

Syntax:

```
bool CloseWatchJoint( )
```

Parameters:

none.

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

none.

SetJoint

Description:

???

Syntax:

```
bool SetJoint( RobotData * pJointData )
```

Parameters:

`RobotData * pJointData`
???

Return value:

Returns true if the function has been succesfully run, false otherwise.

Notes:

RobotData is a typedef for `long[8]`.

Impact

Description:

???

Syntax:

```
bool Impact( RobotData * pJointData )
```

Parameters:

`RobotData * pJointData`
???

Return value:

Returns true if ?, false otherwise.

Notes:

RobotData is a typedef for `long[8]`.

Position Functions

GetPointsPeriphID

Description:

???

Syntax:

```
bool GetPointsPeriphID( Char * szDeviceID1, CHAR * szDeviceUD2, CHAR * szFileName )
```

Parameters:

```
Char * szDeviceID1
```
???

```
Char * szDeviceID1
```
???

```
Char * szDeviceID1
```
???

Return value:

Returns true if ?, false otherwise.

Notes:

???

ClearPointsAttributes

Description:

???

Syntax:

```
bool ClearPointsAttributes( )
```

Parameters:

none

Return value:

Returns true if ?, false otherwise.

Notes:

???

AddPoints

Description:

Loads points into the robots memory from a file.

Syntax:

```
bool AddPoints( Char * szFileName )
```

Parameters:

`Char * szFileName`
path and filename.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none

DefineVector

Description:

Defines a vector in the robots memory.

Syntax:

```
bool DefineVector( Unsigned Char ucGroup, Char * szVectorName, short sDimension )
```

Parameters:

`Unsigned Char ucGroup`
'A' for robot. (normally used).
'B' for peripherals.
'&' for all axes

`Char * szVectorName`
Vectorname

`short sDimension`
Size of the vector (number of points).

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

RenameVector

Description:

Renames a vector in the robots memory.

Syntax:

```
bool RenameVector( Char * szOldName, Char * szNewName )
```

Parameters:

`Char *  szOldName`
Old vectorname

`Char *  szNewName`
new vectorname

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ResetPoints

Description:

???

Syntax:

```
bool ResetPoints( Unsigned Char ucGroup )
```

Parameters:

```
Unsigned Char ucGroup
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

DeletePoint

Description:

Deletes a vector in the robots memory.

Syntax:

```
bool DeletePoint( Char * szVectorName, short sPointNumber )
```

Parameters:

`Char * szVectorName`
Name of the vector.

`short sPointNumber`
Point in the vector.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

If sPointNumber is set to -1 then all of the vectors are deleted.

Here

Description:

Saves a point in memory defined by the robots current position.

Syntax:

```
bool Here( Char * szVectorName, short sPointNumber, long lPointType )
```

Parameters:

`Char * szVectorName`
: Vectorname for the point to be saved in.

`short sPointNumber`
: Point number in the vector.

`long lPointType`
: Point type.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

Teach

Description:

Renames a vector in the robots memory.

Syntax:

```
bool Teach( Char * szVectorName, short sPointNumber, long * plCoorArray, short
sCoorArrayDim, long lPointType )
```

Parameters:

`Char * szVectorName`
: Vectorname.

`short sPointNumber`
: Point number in the vector.

`long * plCoorArray`
: Array with point information (XYZ/Joint values).

`short sCoorArrayDim`
: Size of `plCoorArray`.
5 is most common.

`long lPointType`
: Point type.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

SetEncoders

Description:

???

Syntax:

```
bool SetEncorders( Char * szVectorName, short sPointNumber, long * plCoorArray, short
sCoorArrayDim, long lPointType )
```

Parameters:

`Char * szVectorName`
???

`short sPointNumber`
???

`long * plCoorArray`
???

`short sCoorArrayDim`
???

`long lPointType`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

SetJoints

Description:

???

Syntax:

```
bool SetJoints( Char * szVectorName, short sPointNumber, long * plCoorArray, short
sCoorArrayDim, long lPointType )
```

Parameters:

```
Char * szVectorName
```
???

```
short sPointNumber
```
???

```
long * plCoorArray
```
???

```
short sCoorArrayDim
```
???

```
long lPointType
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

IsPointExist

Description:

???

Syntax:

```
bool IsPointExist( char * szVectorName, short sPointNumber )
```

Parameters:

```
Char * szVectorName
```
???

```
short sPointNumber
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

IsTheBasePosition

Description:

???

Syntax:

```
bool IsTheBasePosition( Char * szVectorName, short sPointNumber, bool &
bIsTheBasePosition, short & sRelPointNumber )
```

Parameters:

`Char * szVectorName`
???

`short sPointNumber`
???

`bool & bIsTheBasePosition`
???

`short & sRelPointNumber`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetNextPoint

Description:

???

Syntax:

```
bool GetNextPoint( Char * szVectorName, short sCurPointNumber,
       short * psNextPointNumber )
```

Parameters:

`Char * szVectorName`
???

`short sCurPointNumber`
???

`short * psNextPointNumber`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetPointInfo

Description:

Retrieves information from a point in memory.

Syntax:

```
bool GetPointInfo( Char * szVectorName, short sPointNumber,
       RobotData * pEncData, RobotData * pXYZData, long * lPointType )
```

Parameters:

`Char * szVectorName`
: Vectorname.

`short sPointNumber`
: Point in the vector.

`RobotData * pEncData`
: Pointer to object to be filled with point information.

`RobotData *  pXYZData`
: Pointer to object to be filled with point information.

`long * lPointType`
: Pointer long to be filled with the point type.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

AttachPosit

Description:

???

Syntax:

```
bool AttachPosit( Unsigned Char ucGroup, Char * szVectorName )
```

Parameters:

```
Unsigned Char ucGroup
```
???

```
Char * szVectorName
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetCurrentPosition

Description:

Get the robots current position.

Syntax:

```
bool GetCurrentPosition( RobotData * pEnc, RobotData * pJoint, RobotData * pXYZ )
```

Parameters:

`RobotData * pEnc`
: Pointer to object to be filled with point information.

`RobotData * pJoint`
: Pointer to object to be filled with point information.

`RobotData * pXYZ`
: Pointer to object to be filled with point information.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

Possibly only `pXYZ` is filled.

Monitoring Commands

ShowEnco

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowEnco( CallBackFun fnViewEnco )
```

Parameters:

```
CallBackFun fnViewEnco
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseEnco

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseEnco( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ShowXYZ

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowXYZ( CallBackFun fnViewEnco )
```

Parameters:

```
CallBackFun fnViewEnco
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseXYZ

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseXYZ( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ShowJoint

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowJoint( CallBackFun fnViewEnco )
```

Parameters:

```
CallBackFun fnViewEnco
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseJoint

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseJoint( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ShowPositErr

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowPositErr( CallBackFun fnViewEnco )
```

Parameters:

```
CallBackFun fnViewEnco
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseEnco

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseEnco( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ShowHomeSwitches

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowHomeSwitches( CallBackFun fnViewEnco )
```

Parameters:

`CallBackFun fnViewEnco`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseHomeSwitches

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseHomeSwitches( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ShowTorque

Description:

Renames a vector in the robots memory.

Syntax:

```
bool ShowTorque( CallBackFun fnViewEnco )
```

Parameters:

```
CallBackFun fnViewEnco
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseTorque

Description:

Renames a vector in the robots memory.

Syntax:

```
bool CloseTorque( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

Input/Output Commands

WatchDigitalInp

Description:

Watch digital input.

Syntax:

```
bool WatchDigitalInp( CallBackFun fnWatchDigitalInp )
```

Parameters:

`CallBackFun fnWatchDigitalInp`

A pointer to a function, the function must be of type `void function(long *)` to work properly.
This function is called each time digital input changes.
A `null` value will stop callbacks of this type.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseWatchDigitalInp

Description:

Stop watching digital input

Syntax:

```
bool CloseWatchDigitalInp( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

WatchDigitalOut

Description:

Watch digital output.

Syntax:

```
bool WatchDigitalOut( CallBackFun fnWatchDigitalOut )
```

Parameters:

`CallBackFun fnWatchDigitalOut`

A pointer to a function, the function must be of type `void function(long *)` to work properly.
This function is called each time digital output changes.
A `null` value will stop callbacks of this type.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseWatchDigitalOut

Description:

Stop watching digital output.

Syntax:

```
bool WatchDidgitalOut( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetDigitalInputs

Description:

Gets the digital input.

Syntax:

```
bool GetDigitalInputs( Unsigned long * plStatusBitmap )
```

Parameters:

`Unsigned long * plStatusBitmap`
: Binary mapping of inputs.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetDigitalOutputs

Description:

Gets the digital output.

Syntax:

```
bool GetDigitalOutputs( Unsigned long * plStatusBitmap )
```

Parameters:

`Unsigned long * plStatusBitmap`
: Binary mapping of output.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

SetDigitalOutput

Description:

Force the digital output.

Syntax:

```
bool SetDigitalOutput( short sIONumber, bool bIsOn, bool bIsImmediate )
```

Parameters:

`short sIONumber`
: Port number.

`bool bIsOn`
: New output value.

`bool bIsImmediate`
: True if change should happen asap. (Always set to true)

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ForceStatusDigitalInput

Description:

???

Syntax:

```
bool ForceStatusDigitalInput( short sIONumber, bool bIsOn )
```

Parameters:

`short sIONumber`
???

`short bIsOn`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetForceStatusDigitalInput

Description:

???

Syntax:

```
bool ForceStatusDigitalInput( Unsigned long * plStatusBitmap )
```

Parameters:

```
Unsigned long * plStatusBitmap
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

EnableDigitalInput

Description:

???

Syntax:

```
bool EnableDigitalInput( short sIONumber )
```

Parameters:

```
short sIONumber
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

DisableDigitalInput

Description:

???

Syntax:

```
bool DisableDigitalInput( short sIONumber )
```

Parameters:

```
short sIONumber
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetDigitalInputEnabledStatus

Description:

???

Syntax:

```
bool GetDigitalInputEnabledStatus( Unsigned long * plStatusBitmap )
```

Parameters:

```
Unsigned long * plStatusBitmap
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetHomeSwitch

Description:

???

Syntax:

```
bool GetHomeSwitch( long * plStatusBitmap )
```

Parameters:

```
long * plStatusBitmap
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

WatchAnalogInp

Description:

???

Syntax:

```
bool WatchAnalogInp( CallBackFun fnWatchAnalogInp )
```

Parameters:

```
CallBackFun  fnWatchAnalogInp
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseWatchAnalogInp

Description:

???

Syntax:

```
bool CloseWatchAnalogInp( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

WatchAnalogOut

Description:

???

Syntax:

```
bool WatchAnalogOut( CallBackFun fnWatchAnalogOut )
```

Parameters:

```
CallBackFun fnWatchDigitalOut
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

CloseWatchAnalogOut

Description:

???

Syntax:

```
bool CloseWatchAnalogOut( )
```

Parameters:

none.

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetAnalogInput

Description:

???

Syntax:

```
bool GetAnalogInput( short sIONumber, Unsiged Char * ucValue )
```

Parameters:

`short sIONumber`
???

`Unsiged Char * ucValue`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ForceStatusAnalogInput

Description:

???

Syntax:

```
bool ForceStatusAnalogInput( short sIONumber, bool bIsOn )
```

Parameters:

`short sIONumber`
???

`bool bIsOn`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

ForceAnalogInput

Description:

???

Syntax:

```
bool ForceAnalogInput( short sIONumber, unsigned Char ucValue )
```

Parameters:

`short sIONumber`
???

`unsigned Char ucValue`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

GetForceStatusAnalogInput

Description:

???

Syntax:

```
bool GetForceStatusAnalogInput( Unsigned long * plStatusBitmap )
```

Parameters:

```
Unsigned long * plStatusBitmap
```
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

SetAnalogOutput

Description:

???

Syntax:

```
bool SetAnalogOutout( short sIONumber, unsigned Char ucValue )
```

Parameters:

`short sIONumber`
???

`unsigned Char ucValue`
???

Return value:

Returns true if operation was succesful, false otherwise.

Notes:

none.

Appendix A: Managed vs. Unmanaged code

Import functions from a dll

In C# you can fairly easy use a function from an external libary, such as a .dll.

The atrribute DLLImport allow this in C#.

A C++ function might look like this in the header file:

```
__declspec(dllexport) bool someFunction(short number, MyClass * someObj);
```

And in C#:

```
[DllImport("mylibary.dll", EntryPoint = "someFunction",
       CallingConvention = CallingConvention.Cdecl)]
public static extern bool myImportedFunction(Int16 number, IntPtr someObj);
```

Just writing the name of the function in the EntryPoint value might not work, it depends on how the libary is compiled. The real entry point can be found with a disassembler or just some clever work with any text editor. Example. the real entry point for the `GetParameterFolder` function in the usbc.dll is:

```
?GetParameterFolder@@YAHPAD@Z.
```

The callingConvention value ensures compatibility with many different types of libaries. The USBC.dll uses `Cdecl`, but windows libaries and C# default is `ThisCall`. The calling convention tells how the variables are put and removed from the stack, defining the wrong value in this field might cause runtime exceptions.

The garbage collector

The main problem between managed and unmanaged code, comes from the use of a garbage collector (GC). The GC controls the memory of managed program. It frees up memory from unused variables, and moves variables to another place in memory, to possibly improve efficiency of the program.
The problem comes up when using unmanaged code (unmanaged code is anywhere your GC does not control the memory). For instance you might want to pass a variable by reference to another program, but while the other program uses the data, your GC moves it, or even deletes it, since it is not used by your program anymore. This will cause a runtime exception, trying to read/write protected memory, or simply just read the wrong data.
So exceptions has to be made. In C# this is done by marshaling.
Marshaling copies and converts incoming data into managed memory, so the GC can control it. It is important to note marshaling copies the data, so you will not be working with the data the other program references.

External data structures

When the marshaling happens, the GC needs to know the structure of the incoming or outgoing data, this is done runtime, so no matter how you define the structure for the GC, exceptions will be coming runtime, the compiler cannnot pick this up.

A C++ class might look like this:

```
class MyClass
{
public:
       char someLetter;
       int someNumber;
       char someText[32];
```

```
        MyClass()
        {
                someLetter = 'a';
                someNumber = 0;
                someText = "some text.";
        }
};
```

The corresponding C# code could look like this:

```
[StructLayout(LayoutKind.Sequential, Pack = 0, CharSet = CharSet.Ansi)]
private class MyClass
{
        public Int32 lNumber;
              public Byte someLetter;
              public Int32 someNumber;
              [MarshalAs(UnmanagedType.ByValArray, SizeConst = 32)]
              public Char[] someText;
          }
}
```

It should be noted that a C++ char array is an 8-bit array (Ansi), whereas a C# char array is a 16-bit array. So by setting the field someText to byte you could get the raw data, but if you have defined Charset.Ansi as above, the runtime environment will marshal that field to a C# char array (Unicode).

So lets use it. Lets create an instance of this object, and fill it in unmanged code, then read it in managed code.
Since we cannot have that our GC moves the object while it is being used in unmanged code, we need to allocate memory for our object in global memory, where both managed and unmanged code can reach it, but the GC cannot:

```
MyClass myObject = new MyClass();
IntPtr myObjectPtr = Marshal.AllocHGlobal( Marshal.SizeOf( myObject ));
Marshal.StructureToPtr( myObject, myObjectPtr, false );
```

Now that our object is ready and copied to global memory, we can send the pointer to unmanged code, so it can be filled:

```
myImportedFunction(0, myObjectPtr);
```

The object is now filled, but we still need to copy it to managed memory so we can read it:

```
myObject = (MyClass) Marshal.PtrToStructure( myObjectPtr, typeof( MyClass ));
```

The object is now ready to use, how we see fit, but one last thing, we need to clean up after ourselfes:

```
Marshal.FreeHGlobal( myObjectPtr );
```

Callback functions

We have looked at outgoing functions calls, but what about incoming calls.
The C# runtime environment needs to know what kind of call to expect. We need to export a function to unmanaged code. First we must declare the type:

```
[UnmanagedFunctionPointer( CallingConvention.Cdecl, CharSet = CharSet.Ansi )]
private delegate void OnSomeEvent( IntPtr someData);
```

We of course also need a function of this type:

```
private void someEvent( IntPtr someDataPtr )
```

```
{
	MyClass myObject =
		(MyClass) Marshal.PtrToStructure( someDataPtr, typeof( MyClass ));

	//do something with data
}
```

And to make sure the pointer to our function will not be deleted while it is used, this can be done by letting the pointer be an instance variable:

```
private OnSomeEvent someEventFunctionPtr = new OnSomeEvent( someEvent );
```

Now we are ready to deliver our function to unmanged code, and recieve callback later:

```
anotherImportedFunction( someEventFunctionPtr );
```