Overview

The PowerCurve Sensor comes with a DLL interface for software developer that would like to integrate the PowerCurve Sensor directly with an external application. There are two DLL's, **`powercurve32.dll`** for 32 bit applications and **`powercurve64.dll`** for 64 bit applications.

The functions are very easy to use and can be called from any programming language that support DLL's . The recommended approach is to load the DLL with Windows API function **`LoadLibrary`** and to get the address of the functions using the **`GetProcAddress`** function. Using this approach you can control exactly when the DLL will be loaded by your application.

The function prototypes are indicated below using C language syntax, but they can easily be converted to another language as simple data types are used.

API

Function calls will return a result code to indicate if the call was successful or not. You should always check and respond to returned result codes. The following result codes may be returned:

```
resSuccess                                    =  0
// any negative return value is an error
resOnlyOnePowerCurveSensorMustBeConnected     = -1
resSensorNotInitialized                       = -2
resSensorAlreadyInitialized                   = -3
resPowerCurveSensorNotPluggedIn               = -4
resInternalException                          = -5
resInvalidPortNo                              = -6
resInvalidTrainerModelNo                      = -7
```

InitPowerCurveSensor2 Function

```
int __stdcall InitPowerCurveSensor(int UpdateTrainerList);
```

Before communicating with a PowerCurve Sensor a call to `InitPowerCurveSensor` must be made. If `UpdateTrainerList == 1` an attempt will be made to update the list of supported trainers over the Internet.

PowerCurveSensorDone Function

```
int __stdcall PowerCurveSensorDone(void);
```

This function must be called when interfacing with a PowerCurve Sensor is no longer required.

GetLastErrorMessage Function

```
char * __stdcall GetLastErrorMessage(void);
```

If resInternalException is returned from a call, more info can be obtained with this function call which returns a pointer to a null terminated C style string.

GetNumberOfPortsOnSensor Function

```
int __stdcall GetNumberOfPortsOnSensor(void);
```

This function will return the number of ports on the connected PowerCurve Sensor. Currently, it will return 2 or 8 as these are two models of the PowerCurve Sensor available.

GetFirstTrainerModelNo Function

```
int __stdcall GetFirstTrainerModelNo(void);
```

This function returns the first valid trainer model number to be used in other API calls.

GetLastTrainerModelNo Function

```
int __stdcall GetLastTrainerModelNo(void);
```

This function returns the last valid trainer model number to be used in other API calls.

GetTrainerProperties Function

```
int __stdcall GetTrainerProperties(
      int TrainerModelNo,
      char ** pName,
      double * pCalibrationSpeedMPS,
      int * pNumberOfLevels,
      int * pResistanceLevel,
      int * pCalibrationResistanceLevel,
      char ** pPowerCurveImageURL,
      char ** pMainImageURL,
      char ** pResistanceUnitImageURL );
```

This function is used to obtained the properties of a trainer model. Strings are passed as pointers to C style null terminated strings. You must pass a pointer to a `char *` (`char **`) for each string parameter. It is acceptable to pass NULL for any property and in that case this property will not be fetched.

The calibration speed in meters per second is returned in *****`pCalibrationSpeedMPS`. This is the speed the rider must exceed during calibration before coasting until the wheel stops turning.

If a trainer has multiple resistance levels (typical of magnetic trainers) the number of levels is returned in `*pNumberOfLevels`.

The resistance level that the rider must use is returned in `*pResistanceLevel`.

For some trainers with multiple resistance levels, the calibration may be done at a different resistance level. For example, with the Tacx CycleTrack on resistance level 7, the calibration is done at level 4 to provide sufficient coast down time. When that is the case, `*pCalibrationResistanceLevel` will return a value of 4 while `*pResistanceLevel` will return a value of 7.

A URL of an image to the power curve will be returned in `**pPowerCurveImageURL`.

A URL of an image of the trainer will be returned in `**pMainImageURL`.

A URL of an image of the resistance unit of the trainer will be returned in `**pResistanceUnitImageURL`.

ConfigurePort Function

```
int __stdcall ConfigurePort(
      int PortNo,
      int TrainerModelNo,
      double WheelCircumferenceM );
```

This function is used to configure a port. `PortNo` starts at value 1.

`TrainerModelNo` identifies the trainer model number.

The wheel circumference is specified in meters. For example, 2.096 is typical for a 700c 23mm road tire.

GetCalibrationStatus3 Function

```
int __stdcall GetCalibrationStatus3(
      int PortNo,
      int * pStatus,
      int * pCalibrating,
      int * pCalibrationValue);
```

This function will return the calibration status in the `*pStatus` parameter which is a pointer to a variable of type `int`. This is the last known calibration status from the last successful coast down. The following values can be returned:

0 = calibrated
1 = unknown (coast down has not occurred)

You should call this function to validate the user has calibrated the press on force otherwise the watts will not be accurate.

During coast down calibration, `pCalibration*` will return 1. If the rider does not coast down completely to a stop, `pCalibration*` will change from 1 to 0 without completing the calibration.

At the end of a calibration cycle, when the coast down has come to a stop, `pCalibrationValue*` will return a numerical calibration value. Riders should aim to use consistent press on force to always obtain the same or very close calibration value.

GetTrainerWattsAndSpeed Function

```
int __stdcall GetTrainerWattsAndSpeed(
      int PortNo,
      double * pWatts,
      double * pWheelSpeedMPS );
```

This function returns the current watts in `*pWatts` and the current wheel speed in `*pWheelSpeedMPS` on a specific port. If you are not interested in one of the return values, it is acceptable to pass NULL for `pWatts` or `pWheelSpeed`.

GetEffectiveSpeed Function

```
double __stdcall GetEffectiveSpeed(
      double Watts,
      double RiderWeightKg,
      double GradePercent,
      double HeadwindMPS );
```

This function returns the effective speed in meters per second given and input power in watts, rider weight in kilograms, the grade in percent, and a headwind in meters per second. For a tail wind pass a negative value for the headwind.

The function will not only take the grade and wind in consideration, but also adjust the aerodynamic factor based on the size (relative to the weight) of the rider.

It has been “calibrated” to a rider on a typical road bike riding in the “drops”.