BLKQCL SDK API Reference

Data Model

All of these operations apply to strongly-typed objects, with attributes. It's hard to understand the operations in any detail, without understanding those types.

Simple Types

- enum **AlarmType**

```
    enum    class   AlarmType {
        Critical_ACUTCUCommunicationsError,
        Critical_CannotTalkToFPGA,
        Critical_ConfigurationError,
        Critical_LaserOverDriving,
        Critical_LaserOverheating,
        Critical_SystemOverTemperature,
        Error_PulseParameterFault,
        Error_SystemTemperatureNotSettled,
        Error_ThermalControlFault,
        Warning_DetectorSignalTooCloseToSaturationLevel,
        Warning_MirrorTransitioningTooFast,
        Warning_UserDiskspaceLow,
        Information_DetectorAccumulationSaturation,
    };
```

This is an enumeration of persistent error states that must be examined and explicitly 'cleared'.

- Critical_ConfigurationError

  This refers to internal system configuration, and not user settings, and should not happen unless there is data corruption, or serious hardware failure.

- Critical_LaserOverDriving

  This refers to too much electrical current going to one or more of the Lasers.

- Warning_MirrorTransitioningTooFast

  Wavelength Changing Faster Than Mirror Transition Event

- Warning_DetectorSignalTooCloseToSaturationLevel

  This means the detector at least once failed to report a sample for accumulation because the intensity of the light saturated the detector.

- Information_DetectorAccumulationSaturation

  This means that the detector stopped accumulating data in at least one bin to avoid an arithmetic overflow. The data is still valid, but you may wish to detect for a shorter period of time to avoid this.

- Warning_UserDiskspaceLow

This is triggered when the space available to store user data, like Scans is nearly full. The user can clear up space with the PersistenceClear or ScanPersistenceClear APIs, or by setting the user Setting *OnDiskspaceLow* to *DeleteOldestScans*, or setting the DeleteScansOlderThan user setting.

- enum **SensorDataKind**

```
    enum    SensorDataKind {
        Accelerometer,
        ActiveLaser,
        ActiveLaserWaveNumber,
        DetectorTemperature,
        ElectricalBoardTemperature,
        ExternalPressure,
        ExternalTemperature,
        LaserCurrent,
        LaserFiringRecentWaveNumber,
        LaserTemperature,
        LaserVoltage,
        MirrorTemperature,
        OpticsTemperature,
        RangeFinder,
        SystemHumidity,
        SystemTemperature,
        TECPowerConsumptionStats
    };
```

This is an enumeration of sensors (used to select which sensors you want to read back in a ReadSensors() call).

- enum **PowerStateType**

```
    enum    PowerStateType {
        Off,
        Hibernate,
        Suspend,
        ReadyToAcceptCommands,
        ReadyToFire,
    };
```

The difference between ReadyToAcceptCommands and ReadyToFire, is that the lasers and device frame have been driven to the correct temperate set-point, and the laser is ready to operate immediately.

- enum SupportedFeatureType

```
    enum    SupportedFeatureType {
        Accelerometer,
        ExternalPressureSensor1,
        ExternalPressureSensor2,
        ExternalPressureSensor3,
        ExternalPressureSensor4,
        ExternalTemperatureSensor1,
        ExternalTemperatureSensor2,
        ExternalTemperatureSensor3,
        ExternalTemperatureSensor4,
        LaserPointer,
        PowerState_Hibernate,
        RangeFinder,
        ScanDetector,
        StepWaveNumberTransitionsExternallyTriggeredViaWAVESignalSupported,
        TouchScreen,
        TriggerButton,
        UserControllableToggleSwitch_FanA,
        UserControllableToggleSwitch_FanB,
        UserControllableToggleSwitch_SolenoidA,
        UserControllableToggleSwitch_SolenoidB,
        WaveNumberExternallyControlledViaWAVESignalSupported,
    };
```

These are factory set features which can be read back from FactorySettings.

- StepWaveNumberTransitionsExternallyTriggeredViaWAVESignalSupported determines if the WAVE (input) hardware is present and feature supported to allow trigging step transitions (WAVELineTriggered) on this trigger. 'Internal" or "Automatic" stepping is always supported. If used, then ExternallyTriggered Step means each pulse advances to the next wave number step entry. See StepTune() for details. This should NOT be confused with UserSettings. LaserControlMode.

- WaveNumberExternallyControlledViaWAVESignalSupported determines if the WAVE line in is supported, and the feature that the full wave-number control is provided by the analog input (0V corresponds to 600 cm-1, and 2.5 V -> 2238 cm-1, and linear in between). See ExternallyControlledTune() for details. This should NOT be confused with UserSettings. LaserControlMode.

- WaveNumberType

```
    using WaveNumberType = double;
```

A WaveNumber is the reciprocal of wave-length. The units are in cm-1 (1/centimeters). WaveNumberType is a floating point number. The minimal legal value is zero (not inclusive) and there is no upper bound.

However, the BLKQCL device only supports wave numbers between 600.0 and 2438.4. The exact range for your device depends on the particular lasers and configuration. See the GetFactorySettings API for more details.

This entire API is defined in terms of WaveNumbers, and not wave-lengths.

- VoltageType

```
    using      VoltageType   =       double;
```

Always measured in volts.

- PressureType

```
    using      PressureType  =       double;
```

Always measured in millibars.

- DistanceType

```
    using      DistanceType  =       double;
```

Always measured in meters.

- RadiansType

```
    using      RadiansType   =       double;
```

Angle measured in radians.

- DetectorDarkModeType

```
    enum DetectorDarkModeType {
        SAMPLE_WIDTH_DIVIDED_BY_4,
        SAMPLE_WIDTH_DIVIDED_BY_2,
        SAMPLE_WIDTH,
        SAMPLE_WIDTH_TIMES_2,
    };
```

Adjustment to Predark/Postdark sample width. This adjustment is in the form of a factor applied to the user setting SampleWidth, and is applied to both the predark and postdark stages of detection.

- ToggleSwitchType

```
    enum ToggleSwitchType{
        FanA,
        FanB,
        SolenoidA,
        SolenoidB,
    };
```

The Fan-A / Fan-B are 12-volt switches, and the names are somewhat misleading, in that they can be used for anything. The Solenoid-A/B switches are similarly misleadingly named, but are really just 5-volt switches. The names used here are to track the names in hardware schematics etc, so it's clear what switch we mean.

- ToggleStateType

```
    enum ToggleStateType{
        Off,
        On,
    };
```

These toggles are off or on. The exact meaning of off/on depends on the particular hardware these control lines are mapped to.

- LaserControlModeType

```
    enum      LaserControlModeType{
        InternallyControlled,
        ExternallyControlled,
        ExternallyTriggered,
    }
```

These modes only apply to laser operation with MoveTune(), StepTune(), SweepTune(), and ExternallyControlledTune().

These modes control how the laser pulses when performing a tune operation.

In InternallyControlled mode, the pulsing is completely controlled by parameters to the .*Tune() functions, and other settings in UserSettings. Also, in this mode, a signal is sent to the TRIG_OUT line indicating whether or not the laser is currently on (see TRIG_OUTDelayTime).

In ExternallyTriggered, most aspects of the laser pulse are defined by the *.Tune() functions and User Settings (e.g. PulseDuration - as with InternallyControlled) - except that the start of each pulse is governed by rising edge of the TRIG_IN signal. The trailing edge of the TRIG_IN signal has no meaning. It is a user error, and will generate an Alarm, if the external triggers come more frequently than the PulseDuration, or the limits of duty cycle.

This external trigger must trigger for the start of each laser pulse - not just once to begin the laser firing process.

In ExternallyTriggered mode, TRIG_OUT is not supported.

In ExternallyControlled mode, the TRIG_IN signal fully controls the laser pulse. The rising edge of the signal turns the laser on, and the falling edge of the TRIG_IN signal turns off the laser. All pulse settings in UserSettings are ignored in this mode. However, system duty-cycle limits are still operative, and if violated, will generate an Alarm.

In ExternallyControlled mode, TRIG_OUT is not supported.

- duration

This is an ISO-8601 standard duration object (used in XML).

http://en.wikipedia.org/wiki/ISO_8601#Durations

- FolderNameType

```
    Using FolderNameType = String;
```

RegExp: [a-zA-Z0-9\-_ \(\)]*

- FileNameType

```
    Using FileNameType = String;
```

RegExp: [a-zA-Z0-9\-_ \(\)]*\.?[a-zA-Z0-9\-_]*

- VolumeIDType

```
    enum VolumeIDType {
         InternalStorage,
         USBDrive,
         SDCard,
    };
```

InternalStorage is a reserved area on the BLKQCL device. This has limited size, but is always available.

USBDrive refers to any USB-drive, formatted with a FAT partition.

SDCard refers to any SDCard inserted into the optional (not available on all systems) SDCard slot.

- TransferEncoding

```
    enum TransferEncodingType {
         text,
         binary-base64,
    };
```

This really applies more to the XML transfer (SOAP) than any particular binding, which could represent/hide this issue.

But in the raw XML (SOAP) data transferred, *text* means the XML element should be treated as a UNICODE string; *binary-base64* means that the data is base64 encoded binary data (in the raw XML).

- ScanKindType

```
    enum ScanKindType {
         Background,
         Reference,
         Sample
    };
```

This enum is used in the IScanPersistence API.

- ScanIDType

```
    using ScanIDType = uint32_t;
```

This is a PER-DEVICE auto-incremented value (increases monotonically over time).

NOTE - as CLOCK time may not advance monotonically (due to access to NTP server, bad clocks, etc), this value takes precedence for defining temporal relationships between spectra (like most recent reference).

Note at one second per scan, 32-bits won't wrap for 126 years

Note - these values may be shared across folders, so you cannot just increment yourself. You must call Advance().

Also - individual scans could disappear (be deleted).

- StrongTunerIDType

```
    using StrongTunerIDType= String;
```

This is a (unicode) string identifier for a tuner - which persists for the lifetime of that tuner. The string length must not exceed 16 UTF8 characters.

Complex (structured) Types

➢ AppConfigurationType

```
struct  AppConfigurationType {
       Optional<LocalServiceType>  LocalService;

       /**
        *  Directory on disk where App is installed. This is optional for
        *  the case when the app is not installed (a package to install)
        */
       Optional<String>    InstallationRoot;

       /**
        *  Relative Path from the installation root. Typically this is 'html'
        *
        *  It is legal to have an app installed with no GUI (for example,
        *  if it just has a WS interface).
        */
       Optional<String>    HTMLRoot;

       /**
        *  Allow installing an app, but disabling it, so it doesn't have its
        *  service run, or app UI appear in the GUI.
        */
       Optional<bool>  Enabled;

       /**
        *  Arbitrary key-value pairs whose meaning is defined by the app
        */
       Optional<Mapping<String, String>>    Features;

       /**
        *  Key. And is relative path from top of url hierarchy name displayed
        * (e.g. /Tune is where the AppName Tune html is loaded from).
        */
       Optional<String>    AppShortName;

       /**
        *  Generally same as AppShortName, but can be used to override what is
        *  displayed in menu and other places app name is shown
        */
       Optional<String>    AppDisplayName;
};
struct  AppConfigurationType::LocalServiceType {
    /**
     *
     *  If present, when app installed, this is added to systemd automatically.
     *
     *  The service will be enabled and started iff the app is installed and enabled.
     *
     *  This file path is a relative path from the installation root
     *
     *  This file is a normal systemd file, except that it doesn't contain any
     *  absolute pathnames.
     *  It contains the special string [INSTALLDIR] any place where the full path
     *  to a file is needed (typically the ExecStart line)
     */
    Optional<String>    SystemDServiceFileTemplate;
    /**
     * The GUI connects to this address to control the app
     */
    uint16_t            WSPortNumber {};
};
```

- BatteryStatusType

```
    struct  BatteryStatusType {
        bool                            BatteryCapable;
        Optional<bool>                  BatteryPresent;
        Optional<float>                 BatteryPercentCharged;
        bool                            ExternalPowerPresent;
        Optional<bool>                  Charging;
        enum    class   State {
            TooHot,
            NoLongerCharing,
        };
        Optional<Set<State>>            BatteryStates;
    };
```

- TECPowerConsumptionStatsType

```
    struct  TECPowerConsumptionStatsType {
        Mapping<TunerNumberType, CurrentType>   TunerTECCurrent;
    };
```

➢ **UserSettingsType**

```
    struct   UserSettingsType {
        struct LaserPumpingVoltageType {
            enum ProfileType { Variable, Fixed };
            ProfileType Type;
            double FixedVoltage;
            LaserPumpingVoltageType (ProfileType pt);         // variable only
            LaserPumpingVoltageType (ProfileType pt,
                                     VoltageType fixedVoltage
                                                           ); // fixed only
        };
        Optional<Mapping<TunerNumberType, TemperatureType>>
                                               LaserTemperature;
        Optional<Mapping<TunerNumberType, LaserPumpingVoltageSettingType>>
                                               LaserPumpingVoltage;
        Optional<duration>                     PulseDuration;
        Optional<duration>                     PulsePeriod;
        Optional<LaserControlModeType>         LaserControlMode;
        Optional<bool>                         ContinueFiringAfterInterleavedScans;
        Optional<bool>                         AutomaticallyAdjustInterleavedScanLag;
        Optional<duration>                     TRIG_OUTDelayTime;
        struct IdleAutoPowerStateChangesType {
             Optional<duration>  Off;
             Optional<duration>  Hibernate;
             Optional<duration>  Sleep;
             Optional<duration>  LaserNotReady;
        }
        Optional<IdleAutoPowerStateChangesType>IdleAutoPowerStateChanges;
        Optional<bool>                         MonitorDACEnable;
        Optional<CCUDACValueType>              GainDAC;
        Optional<duration>                     SampleDelay;
        Optional<duration>                     SampleWidth;
        Optional<TemperatureType>              DetectorTemperatureSetPoint;
        enum OnDiskspaceLowType {
            Stop,
            DeleteOldScanData
        };
        Optional<OnDiskspaceLowType>           OnDiskspaceLow;
        Optional<Duration>                     DeleteScansOlderThan;
    };
```

- LaserTemperature

  Per tuner laser temperature set-point.

- LaserPumpingProfile

  Per tuner laser voltage settings. Each tuner can be set to a specific voltage, or can be programmed to use the (per tuner) factory set variable voltage profile.

- PulsePeriod

  Represented as an ISO-8601 duration.

  This value can be anything but must respect the duty cycle constraints in FactorySettings. LaserDutyCycleLimit (below). Typically this is around 100 times the PulseDuration, so perhaps 5µs.

- PulseDuration

  Represented as an ISO-8601 duration.

  A typical value might be 50ns.

  This is the duration (sometimes called pulse-width) of a single laser pulse. This value must always be in the range specified by FactorySettings. LaserPulseDurationLimit (within its lower/upper bounds).

  This value – is also constrained by the FactorySettings.LaserDutyCycleLimit (see that for details on limitation).

- LaserControlMode

  See LaserControlModeType.

- TRIG_OUTDelayTime

  This only applies when UserSettings.LaserControlMode is InternallyControlled, and then it is the delay between when the laser after the signal is sent before the laser pulse is fired.

- IdleAutoPowerStateChanges

  Timer based power level transitions. As of system release 0.02, only 'LaserNotReady' state transitions are implemented.

  Specify a very large duration (1 year?) to disable the feature.

  When the system has been idle for 'LaserNotReady' time period, the lasers are automatically turned off, and attempts to keep them at the proper temperature are relaxed. To warm them back up, and assure they are ready to fire, call "SetPowerState (ReadyToFire)"

- AutomaticallyAdjustInterleavedScanLag

  If true, automatically adjust the calibrated lag between forward and backward scanning in InterleavedScan mode.

- OnDiskspaceLow

  When adding scan data via ScanPersistenceAdd - if the system is low on diskspace, raise a fault or just silently delete old scan data.

- DeleteScansOlderThan

  When adding new scan data to a folder, delete unneeded scans older than the given time (now - duration).

- **FactorySettings**

```
    struct  FactorySettingsType {
        struct  TunerFactorySettingsType {
            InterleavedScanLagType                    InterleavedScanLag;
            TemperatureRangeType                      LaserOperationTemperatures;
            WaveNumberRangeType                       LaserWaveNumberRanges;
            VoltageRangeType                          LaserPumpingVoltageBounds;
            WaveNumberToVoltageMapType                LaserVariablePumpingVoltage;
            CurrentType                               LaserMaximumPumpingCurrent;
            duration                                  LaserVoltageSettleTime;
            MirrorDACRangeType                        MirrorMovementRange;
            FrequencyType                             MirrorOperationFrequency;
            FrequencyType                             MirrorResonantFrequency;
            LineSlopeAndOffsetType   MirrorCurrentToDriveVoltageRelation;
            StrongTunerIDType                         StrongID;
            DistanceType                              NominalGrooveSpacing;
            RadiansType                               NominalPhiNeutralAngle;
            PIDType                                   TunerTECControlParameters;
            CurrentType                               TECCurrentUpperBound;
            WaveNumberToMirrorDriveCalibrationTableType
                                     WaveNumberToMirrorDriveCalibrationTable;
        };
        struct  LaserStitchPointType {
             TunerNumberType     LowerTuner;
             TunerNumberType     UpperTuner;
             WaveNumberType      WaveNumber;
        };
        struct  ACUBoardType {
             uint   BoardVersion;
             bool   SupportsFPGAProgramCommand;
             bool   Supports921600BaudSerialLink;
             uint   MirrorSPIPeriod;
             String TargetFPGAVersion;
             bool   TunerColdPlateTEC;
             double SweepLUTStepSizeInWaveNumbers;
        };
        struct  CCUBoardType {
             bool   SupportsFPGAProgramCommand;
             String TargetFPGAVersion;
             uint   TECPWMWidthMax;
        };
        struct      ColdPlateTECType {
              PIDType            PID;
              bool               Enabled;
              duration           PIDLoopCycleTime;
              TemperatureType    SetPoint;
        };
        struct      I2CConfigurationType {
             uint   InterconnectBoardVersion;
        };

        ACUBoardType                        ACUBoard;
        ColdPlateTECType                    ColdPlateTEC;
        CCUBoardType                        CCUBoard;
        CCUDACValueType                     BiasDAC;
        UserSettingsType                    DefaultUserSettings;
        TemperatureRangeType                DetectorTemperatureSetPointRange;
        PIDType                             DetectorTECPIDParameters;
        DetectorDarkModeType                    DetectorDarkMode;
        I2CConfigurationType                    I2CConfiguration;
        DutyCycleType                           LaserDutyCycleLimit;
        DutyCycleType                           LaserDutyCycleJitterAdjustment;
        DurationRange                           LaserPulseDurationLimit;
        LaserStitchPointType                    LaserStitchPoints[];
        TemperatureType                         LaserTemperatureVariationAllowed;
        duration                                LaserTemperatureWarmupTimeout;
        duration                                LightToPostDark;
        duration                                MirrorMoveSmoothingDuration;
        TemperatureRangeType                    OpticsTemperatureRange;
        duration                                PreDarkToLight;
        DurationRange                           SampleDelayRange;
        Set<SupportedFeatureType>               SupportedFeatures;
        TemperatureRangeType                    SystemTemperatureRange;
        Mapping<ToggleSwitchType,ToggleStateType>
                                                ToggleSwitchInitialValues;
        Mapping<TunerNumberType, TunerFactorySettingsType>
                                                Tuners;
    };
```

Note that FactorySettings objects are read-only, and contain many details a typical user won't be concerned with.

Typically, one might only be interested in

- Tuners/ LaserWaveNumberRanges
- Tuners/ LaserVariablePumpingVoltage
- PreDarkToLight & LightToPostDark
- SupportedFeatures
- LaserDutyCycleLimit
  Roughly speaking, this is the maximum percent of time the laser can be pulsing. That means the user settings value
  UserSettings.PulseDuration/UserSettings.PulsePeriod must be <= LaserDutyCycleLimit /100

But with laser pulses longer than 50 ns (UserSettings.PulseWidth > 50ns), this duty cycle limit decreases slightly.

Also note that this limit applies individually to each laser, so if you have a system with multiple laser tuners, doing interleaved scan, you can effectively multiply the total light output by that number of lasers.

- LaserPulseDurationLimit
- LaserTemperatureVariationAllowed
  The BLKQCL system tries to keep the laser at the user-settings-specified temperature. This 'allowed-temperature-variation' - controls how much variation in temperature is allowed before the BLKQCL-Controller faults.

  This number is also used (/3) to be the temperature tolerance we use in warming up the lasers before we allow an operation to start.

- LaserTemperatureWarmupTimeout
  When we start a laser operation, we wait for the lasers to reach the target temperature (within LaserTemperatureVariationAllowed/3 tolerance). This is the maximum amount of time the BLKQCL-Controller will wait for that temperature goal to be reached.

➢ SensorDataType

```
    struct  SensorDataType {
        struct AccelerometerDataType {
            AccelerationType    AngularVelocityX;
            AccelerationType    AngularVelocityY;
            AccelerationType    AngularVelocityZ;
            AccelerationType    AccelerationX;
            AccelerationType    AccelerationY;
            AccelerationType    AccelerationZ;
        };
        Optional<AccelerometerDataType>                   Accelerometer;
        Optional<TunerNumberType>                         ActiveLaser;
        Optional<WaveNumberType>                          ActiveLaserWaveNumber;
        Optional<TemperatureType>                         DetectorTemperature;
        Optional<TemperatureType>                         ElectricalBoardTemperature;
        Optional<PressureType>                            ExternalPressure;
        Optional<TemperatureType>                         ExternalTemperature;
        Optional<Mapping<TunerNumberType, TemperatureType>>LaserTemperatures;
        Optional<Mapping<TunerNumberType, CurrentType>>    LaserCurrents;
        Optional<Mapping<TunerNumberType, VoltageType>>    LaserVoltages;
        Optional<Mapping<TunerNumberType, TemperatureType> MirrorTemperatures;
        Optional<TemperatureType>                         OpticsTemperature;
        Optional<DistanceType>                            RangeFinderDistance;
        Optional<HumidityType>                            SystemHumidity;
        Optional<TemperatureType>                         SystemTemperature;
        Optional<TECPowerConsumptionStatsType>            TECPowerConsumptionStats;
    };
```

- DetectorTemperature
  This is the CCU temperature
- ExternalPressure
  This is the reading of the external pressure sensor. This feature is optional (see SupportedFeatureType:: ExternalPressureSensor)
- ExternalTemperature
  This is the reading of the external temperature sensor. This feature is optional (see SupportedFeatureType:: ExternalTemperatureSensor)
- SystemTemperature
  This is the 'cold plate' temperature
- TECPowerConsumptionStats

This returns the actual (current in the time sense) Current (in the electrons sense) being used for each installed tuner to drive the given tuner to the set-point temperature.

This current usage could be for either cooling or warming.

Please note that GetFactorysettings().TECCurrentLimits can be combined with these above values to produce a percentage of max current

- **SpectrumType**

```
using SpectrumType = Mapping<WaveNumberType, IntensityType>;
```

A spectrum can be thought of as a histogram, or ‘bar chart’, where the ‘x’ axis is wave-number, and the y-axis is intensity.

- PersistenceScanAuxDataType

```
using PersistenceScanAuxDataType = Mapping<String, String>;
```

This is a key-value pair mapping string to string. This API provides no mechanism to handle meta information: it is up to callers to know the underlying ‘type’ of the mapped-to data. And if the data cannot be safely represented as a string, it must be encoded by the caller (e.g. base64).

- **VersionInfoType**

```
    struct      VersionInfoType {
        String      ModelName;
        String      ModelNumber;
        String      FactorySerialNumber;
        String      ACUFPGASoftwareVersion;
        String      CCUFPGASoftwareVersion;
        String      BLKControllerSoftwareVersion;
        String      OSSoftwareVersion;
    };
```

Interfaces

WSDL allows us to logically divide the BLK-15 API into logically related sections. The most important interfaces for users getting started would be:

- IConfiguration
- IDeviceManagement
- ILaserOperation
- IBasicPersistence
- IScanPersistence

IConfiguration

This interface is concerned with persistent state. Information set or received through this interface is preserved across reboots, and is unlikely to require change from use to use.

The interface provides these important operations:

- **GetFactorySettings**

```
GetFactorySettings () -> FactorySettingsType;
```

This returns a great deal of static information (information that will not likely change without sending your device in for service, or unless you do re-calibration).

RETURNS: returns the full set of factory settings (FactorySettingsType)

- **GetUserSettings/ SetUserSettings**

```
GetUserSettings () -> UserSettingsType;
SetUserSettings (UserSettingsType settings) -> void;
```

This retrieves all the user-settable, persistent options which control your BLK-15.

This allows setting any single (or collection of) persistent options.

RETURNS: GetUserSettings returns the full set of user settings (UserSettingsType), and SetUserSettings() returns nothing.

- **ResetToFactoryDefaults**

```
ResetToFactoryDefaults () -> void;
```

This is roughly equivalent to SetUserSettings (GetFactorySettings ().FactoryDefaultsForUserSettings) except that it also resets other things (@todo LIST OTHER THINGS)

- **GetAppConfigurations**

```
GetAppConfigurations () -> Collection<AppConfigurationType>;
```

Retrieve the full set of installed applications and their configuration information. This configuration information is not user-modifiable.

IDeviceManagement

This returns a variety of operational status details which have little to do laser operations, but are more generally associated with generic device management and control.

Where functionality logically belongs in multiple Interfaces, we place it here if it tends to be apply to any networked device, as opposed to laser specific configuration features.

- **GetAlarms / ClearAlarms**

```
GetAlarms () -> AlarmType[];
ClearAlarms (All) -> AlarmType[];
ClearAlarms (AlarmType[] alarmstoClear) -> AlarmType[];
```

The BLK-15 device allows for very complex and powerful end-user control, and some combinations of settings and operations may not be able be possible. Plus, some hardware components may fail over time.

GetAlarms() returns the list of current alarm conditions (hopefully now). And ClearAlarms () tells the device the controller (gui or smart control system) has been informed of the problem, and to attempt to automatically correct the problem.

RETURNS: All these methods returns the current set of AlarmType[] outstanding after the call.

- **GetBatteryStatus**

```
GetBatteryStatus () -> BatteryStatusType;
```

If your device has a battery, this will return information about its charge status.

RETURNS: BatteryStatusType.

- **GetDeviceName / SetDeviceName**

```
GetDeviceName () -> String;
SetDeviceName (String newName) -> void;
```

Retrieve the device name (appears in SSDP/network neighborhood discovery)

RETURNS: GetDeviceName() returns the device name (as it appears in SSDP discovery).

➢ GetPowerState / SetPowerState

```
GetPowerState () -> PowerStateType;
SetPowerState (PowerStatetype newPowerState) -> void;
```

Power states include 'Ready to accept commands', 'poweroff', and 'ready to fire lasers'

RETURNS: **GetPowerState** () returns the current PowerStateType.

➢ GetVersionDetails

```
GetVersionDetails () -> VersionInfoType;
```

Software/Firmware component versions.

RETURNS: the version information etc., see VersionInfoType.

➢ Lock

```
Lock (Optional<String> PIN, Optional<bool> Optional<String> Reason = {},
Optional<Duration> AutoUnlockAfter = {}, Optional<Duration> AutoUnlockAfterIdle =
{}) -> String;
```

The lock function may be used to put the device in a mode where only a particular source of commands will be respected. This can be used to prevent accidental changes to the system by one user while another is performing a critical operation.

Lock () takes an optional PIN (a numeric string).

Lock () does not persist across reboots, so it's always safe to recover an accidentally locked device (where the key was lost) – but powering off – pressing the device power button (click – not long press-hold).

The 'Locked' state can be detected by using the WS API GetCurrentOperation () and looking for the Locked flag, as well as the locker, and reason for locking.

Anyone who knows the PIN may call RecoverLock– to retrieve the Locker-Key.

Lock on an already locked device will cause a lock failure.

The returned Locker-Key must be provided as a SOAP-Header (handled in proxies) for (nearly) every subsequent operation done to the BLK device, until the device is unlocked. (Most) calls which do not have this key will be rejected with a 'System Locked' fault.

Operations which can be done to a Locked device without the Locker-Key SOAP header include:

- Lock
- UnLock
- RecoverLock
- GetCurrentOperation
- GetUserSettings
- GetFactorySettings
- ReadSensors()
- SetPowerState(Shutdown)

RETURNS: String (Locker-Key token).

➢ UnLock

```
UnLock (Optional<String> PIN, Optional<String> LockerKey) -> void;
```

The UnLock() function – undoes the effect of Lock(). If the device was already not locked, this has no effect.

Regardless of how many previous outstanding calls there were to Lock(), a single UnLock() call unlocks the device.

If the last Lock() call provided a PIN, then UnLock() requires EITHER a matching PIN, OR the LockerKey returned from the last Lock() call.

This function will SOAP-FAULT if provided an incorrect PIN or LockerKey. A void response means the system was successfully unlocked.

RETURNS: void

➢ RecoverLock

```
RecoverLock (Optional<String> PIN) -> void;
```

The RecoverLock () function will return the lockKey associated with the given PIN (if any) – if the device is locked. If no PIN provided, but a locker key is provided (SOAP header), then allow that to VALIDATE that the locker key is correct without performing any operation. RecoverKey will SOAP-FAULT if the device is not locked, or the provided PIN doesn't match, and the provided lockerKey does not match.

RETURNS: void

ILaserOperations

Laser operations are the primary functions of the BLK-15 system. These APIs are stateless, in that none of their parameters are preserved, though they are modal in the sense that they drive the behavior of the BLK-15 device – and report back on that state.

- **GetLaserPointerOn / SetLaserPointerOn**

```
GetLaserPointerOn () -> bool;
SetLaserPointerOn (bool on) -> void;
```

This function returns a Boolean indicator of whether or not the laser pointer is on.

This controls the laser pointer, used to aid in visually aligning your laser system.

RETURNS: GetLaserPointerOn() returns bool true if laser pointer is on.

- **GetToggleSwitchState / SetToggleSwitchState**

```
GetToggleSwitchState (ToggleSwitchType: which) -> ToggleStateType;
SetToggleSwitchState (ToggleSwitchType: which, ToggleStateType: state) -> void;
```

This function gets/sets the state of the given switch. These switches are not always available on all instruments, and are mapped to different kinds of physical controls, depending on application. So – don't use this API unless you really know what your device/switch is mapped to.

RETURNS: **GetToggleSwitchState** () returns Open or Closed.

- **ReadSensors**

```
ReadSensors(All) -> SensorDataType;
ReadSensors([SensorKind sensor,]*) -> SensorDataType;
```

This function can read back any of an argument set of sensors (or all sensors).

Importantly – ReadSensors() can be run in one HTTP-connection channel while laser operations are happening in another connection channel, to allow these things to happen in parallel.

OR – you can use a single HTTP connection, and read sensors before and after the laser operation, to guarantee ultra-precise sequencing.

RETURNS: SensorDataType – with the sensors specified in the argument to ReadSensors() supplied.

StopLasers

```
StopLasers () -> void;
```

This function has two behaviors: it aborts any outstanding laser operations, and it turns off the laser.

MoveTune

```
MoveTune (
    WaveNumberType waveNumber,
    LaserOnOffTransitionType duringTransition = LaserOn
) -> WaveNumberType;
MoveTune (
    WaveNumberType[] waveNumbers,
    LaserOnOffTransitionType duringTransition = LaserOn
) -> WaveNumberType[];
```

The Move operation simply turns the laser on, and on a particular *waveNumber*. This is mostly the equivalent of a Step tune with dwellTime 0, start=end, stepSize meaningless, except that the laser is always ON at the end of a move tune.

If your device contains multiple lasers, you may be able to do multiple moves at the same time (sometimes called InterleavedTune) – but the wavenumbers provided in the array must each fall in the wavenumber range for a different tuner (see GetFactorySettings).

Note meaning of *duringTransition* is that if OFF, we force the laser off before the move. If duringTransition is ON, we simply skip that turn-off step. 'duringTransition' ON does not imply turning the laser on before move - just implies NOT turning it off before move.

RETURNS: The measured target wavenumber (WaveNumberType)

- **StepTune**

```
StepTune (
    WaveNumberType start,
    WaveNumberType end,
    WaveNumberDistanceType delta,
    StepTransitionControlModeType stepControlMode = Internal,
    duration dwellTime = PT0S,
    LaserOnOffTransitionType duringTransition = LaserOn
) -> WaveNumberType[]
```

A step tune turns the laser on for each step from *start* to *end* (by *delta* increments).

In stepControlledMode==Internal (the default), the laser stops and waits 'dwellTime' at each increment.

In stepControlledMode==WAVELineTriggered' mode, the dwellTime is ignored (and must be PT0S if provided), and the laser transitions with the rising edge of the WAVE hardware signal line.

And at the end of a step tune, the laser is turned off.

Note – stepControlMode==WAVELineTriggered mode is only available if FactorySettings.SupportedFeatures contains the StepWaveNumberTransitionsExternallyTriggeredViaWAVESignalSupported feature.

RETURNS: The measured target wavenumbers (WaveNumberType).

- **SweepTune**

```
SweepTune (
    WaveNumberType start,
    WaveNumberType end,
    SweepRateType sweepRate,
    unsigned int repeatCount = 1,
    duration interRepeatDelay = PT0S
) -> void
```

A sweep tune turns the laser on and moves it smoothly through the waveNumbers start thru end, at a rate of sweepRate (cm^-1/ms -inverse centimeters per millisecond).

The sweep will be repeated 'repeatCount' times.

A delay after each run (except the last) of 'interRepeatDelay' may be applied to increase the time between end of the preceding sweep and the start of the next. However, if a delay of less than the hardware is capable is applied, the system will just repeat as quickly it's able to.

Note: it is legal for start > end, and if that is the case, the sweep takes place from higher wavenumbers down to lower wavenumbers. However, sweepRate must always be positive.

RETURNS: void; to monitor the progress, use ReadSensors ([LaserFiringRecentWaveNumber]);

- **ExternallyControlledTune**

```
ExternallyControlledTune () -> void
```

This function turns the laser on (subject to the UserSettings LaserControlMode).

The selected WaveNumbers are completed controlled externally via the WAVE analog input.

The 'ExternallyControlled' part of the name refers to control of the wave number, not the laser pulse (see LaserControlMode).

The wavenumber is linearly related to the input voltage, with 0 corresponding to 600cm-1, and 2.5V corresponding to 2238cm-1.

This operation is only available if the WaveNumberExternallyControlledViaWAVESignalSupported feature is in the FactorySettings.FeaturesSupported set.

RETURNS: void

- **Delay**

```
Delay (
    duration delay
) -> void
```

This does nothing but wait the given amount of time. This can be used as part of a pipelined sequence of operations to adjust the timing of operations (since operations run one after the other within a given command channel)

RETURNS: void

- **StepScan**

```
StepScan (
    WaveNumberType start,
    WaveNumberType end,
    WaveNumberDistanceType delta,
    duration dwellTime,
    LaserOnOffTransitionType duringTransition = LaserOn,
    unsigned int scansPerSpectrum = 1,
    duration delayBetweenCoAdds = PT0S
) -> SpectrumType;
```

This function scans all from the *start* to the *end*, in increments of delta.

At each step, it measures laser wave number intensity, and returns those intensities as a spectrum.

dwellTime is the time spent waiting (and measuring/detecting) at each wavenumber step (and must be greater than zero).

If spectrumResultCount > 1, this function captures multiple results with a time gap of *delayBetweenCoAdds* between then, and averages the results.

scansPerSpectrum is sometimes called '# of co-adds'.

Note this delta is both the step size - space between scans, and the resulting bin size of the spectrum results.

RETURNS: The resulting spectrum (SpectrumType).

- **SweepScan**

```
SweepScan (
    WaveNumberType start,
    WaveNumberType end,
    SweepRateType sweepRate,
    WaveNumberDistanceType scanResolution = 10.0,
    unsigned int scansPerSpectrum = 1,
    duration delayBetweenCoAdds = PT0S
) -> SpectrumType;
```

This function scans in the designated range (from start to end, at a rate measured in cm^-1/ms - inverse centimeters per millisecond).

This function returns data in buckets of width scanResolution wave numbers.

If spectrumResultCount > 1, this function captures multiple results with a time gap of *delayBetweenCoAdds* between then, and averages the results.

scansPerSpectrum is sometimes called '# of co-adds'.

RETURNS: The resulting spectrum (SpectrumType).

- **InterleavedScan**

```
InterleavedScan (
        duration singleSpectrumMeasurementTime,
        WaveNumberDistanceType scanResolution = 10.0,
        unsigned int scansPerSpectrum = 1,
        duration delayBetweenCoAdds = PT0S
   ) -> SpectrumType;
```

This function scans all supported laser frequencies, emitting laser radiation, and reading back a measured spectrum (from the detector). This function runs for *singleSpectrumMeasurementTime* and returns data in buckets of width *scanResolution* wave numbers.

Note that 'scanResolution' can be the special value "NaturalBinning" (details depend on language bindings) – which causes the scan to be returned in un-evenly sized bins, which match the low level hardware sensors/bin size.

If scansPerSpectrum > 1, this function captures multiple results with a time gap of *delayBetweenCoAdds* between then, and averages the results.

scansPerSpectrum is sometimes called '# of co-adds'.

RETURNS: The resulting spectrum (SpectrumType).

The returned spectrum contains a map from wavenumbers to intensities. The wave numbers returned represent the midpoint of a bucket (*scanResolution* wide).

IBasicPersistence

IBasicPersistence provides limited persistence support. This can be used by applications which interact with the BLKQCL device to store small amounts of information persistently (across runs), and to access external memory devices (such as USB flash drives or SD cards).

The API does not currently allow for hierarchical folders. You can have as many folders as you want at the top level of storage, and each can contain as many files as you want (as many as fit on the volume selected).

This API provides no security interface (other than security by obscurity - there is no way to enumerate folders), no hierarchical folders, and no meta information (e.g. last-modified) APIs.

This CAN be used to store small amounts of user-preference state specific to an application (not session), and can be used to store larger amounts of data on removable media (such as scan results).

- **PersistenceClear**

```
PersistenceClear (VolumeIDType volume, FolderNameType folder,
Optional<FileNameType> file) -> void;
```

Clear the given folder (if no file given), or file. This returns no indicator, but raises a fault on failure.

RETURNS: void.

- **PersistenceGet**

```
PersistenceGet (VolumeIDType volume, FolderNameType folder, FileNameType file,
TransferEncodingType transferEncoding = text) -> BLOB;
```

Return the data from the given volume, folder, filename combination. This raises a fault if the file is not present. The caller specifies whether the file is read and returned as a ‘string’ – or if its treated as a binary BLOB of data and returned over the wire base64-encoded (raw xml).

RETURNS: Either string or base64 encoded binary, depending on transferEncoding parameter.

- **PersistenceList**

```
PersistenceList (VolumeIDType volume, Optional<FolderNameType> folder) ->
struct Entry { Optional<FileNameType> fileName; Optional<FolderNameType>
folderName;} [], Space {float TotalUsedBytes; float TotalAvailableBytes};
```

For the given folder, this returns the list of all filenames in the folder. Note – this API does NOT support hierarchical folders, but is designed so that we could naturally extend it in that direction, in the future, if needed.

If the folder is omitted, PersistenceList() returns the top level folders in the given volume. Top level volumes must NOT directly contain files: only folders.

This function will fault if given a nonexistent folder, but will return an empty list if the given folder exists, but is empty.

RETURNS: Zero or more entries, each of which has an attribute 'fileName' or 'folderName'. Optionally returns Space element, containing TotalUsedBytes and TotalAvailableBytes.

- **PersistencePut**

```
PersistencePut (VolumeIDType volume, FolderNameType folder, FileNameType file,
TransferEncodingType transferEncoding, BLOB value) -> void;
```

Send the data to the given volume, folder, filename combination. The caller specifies whether the data to be written to the file is interpreted as a Unicode string – or if it's treated as a binary BLOB of data passed over the wire base64-encoded (raw xml).

RETURNS: void

IScanPersistence

This API is layered on top of the IBasicPersistence API, to provide fast, high level access to persistent scan data.

The basic idea is to store captured scan data from this BLKQCL device, and provide fast routines to associate other data (such as analysis data) with those scans, and to track related references etc.

This also quickly computes ratioed scans, while preserving the original raw spectral data.

- **ScanPersistenceAdd**

```
ScanPersistenceAdd (VolumeIDType Volume, FolderNameType Folder, DateTime
ScanStart, DateTime ScanEnd, Optional<String> ScanLabel, Optional<ScanKindType
> ScanKind, Optional<SpectrumType> RawSpectrum,
Optional<PersistenceScanAuxDataType> AuxData, Optional<ScanIDType> Background,
Optional<ScanIDType> Reference) -> ScanIDType;
```

RETURNS: ScanIDType.

This function stores the given Scan() data (presumably just returned from an earlier InterleavedScan()). This API assumes that if all samples – if relative to a reference (the default) – it is relative to the last entered reference scan.

Even if the scan is ratioed, the passed in this API must be given the raw, un-ratioed, unprocessed spectrum.

Note that the ScanKind can be omiited for 'virtual scan' records – which record other sorts of meta information, and just refer to other Scans. For example, if you have analysis which leverages multiple scans, you can record the scans by themselves without analysis data, and then record the 'virtual scan' for the analysis results, and have it refer to its (TBD) BasedOn scans.

Background: If this is a sample or reference, this may be used to control whether it is relative to the the given background. Otherwise, it is ignored.

Reference: If this is a sample, and true, then the sample is relative to the given stored reference. Otherwise it is ignored.

Note that it is an error to pass in a value of Reference with a sample where the used Reference has a different value of Background from the argument Background.

[@ed perhaps don't allow this parameter for samples?]

- **ScanPersistenceAdvance**

```
ScanPersistenceAdvance (VolumeIDType Volume, FolderNameType Folder,
ScanIDType[] ScanIDs, int ByScanID) -> ScanIDType[];

ScanPersistenceAdvance (VolumeIDType Volume, FolderNameType Folder,
ScanIDType[] ScanIDs, duration ByDuration) -> ScanIDType[];
```

RETURNS: ScanIDType[]

For each id, advance it offset by scan id (essentially count of samples), or time offset. This can be negative or positive. In both cases, we round. Going back from any sample by ALOT gets you the first sample.

This always returns valid scan ids, and uses the incoming scanIDs as a hint, so even with no offset, it can be used to find the nearest scanid to the given one.

The only way this can raise an error (fault) is if there are no valid scan ids.

- **ScanPersistenceClear**

```
ScanPersistenceClear (VolumeIDType Volume, FolderNameType Folder,
Optional<ScanIDType> LowerBound, Optional<ScanIDType> UpperBound) -> void
```

RETURNS: void

Clear the specified range of scan data. If the lower or upper bounds are omitted, assume that means from the beginning of time, or end of time (for that folder).

Any attempt to remove scan data, such as references that are still in use in later data, will not be an error, but that part of the request will be silently ignored.

So - it's perfectly safe to delete all data before a certain point in time, and not worry about if the references or backgrounds were in that range.

Data is not automatically deleted when no longer referenced, to external logic must be applied to delete scan data.

- **ScanPersistenceGetAuxScanData**

```
struct AuxScanResult {
      ScanIDType ScanID;
      DateTime   ScanEndTime;
      PersistenceScanAuxDataType AuxData;
};

ScanPersistenceGetAuxScanData (VolumeIDType Volume, FolderNameType Folder,
ScanIDType[] ScanIDs, Set<String> AuxKeys, Optional<float> SmoothingFactor) ->
AuxScanResult[]
```

RETURNS: AuxScanResult[]

This is a shortcut for ScanPersistenceGetScanDetails (), and extracting just the relevant Aux data, and throwing the rest away.

If any of the argument scanIds are missing or invalid, they will be ignored, so the resulting AuxScanResult array will be the same size, or less, than the argument ScanIDs array.

This means the caller CAN just create an array of N..M, where the caller increments values.

EXAMPLE:

```
         ScanPersistenceGetAuxScanData (
               InternalStorage,
               "my experiment",
                [1344,3456],
                ['C1, 'C2']
           ) -->
                 [
                    {ScanEndTime: '2014-12-31 2pm', {C1: 12.3, C2: 9.5 },
                    {ScanEndTime: '2014-12-31 2:15pm', {C1: 12.5, C2: 9.1 },
                 ]
```

- **ScanPersistenceGetFolderSummary**

```
struct  ScanPersistenceFolderSummaryType {
       ScanIDType? FirstScanID;
       DateTime? FirstScanAt;
       ScanIDType? LastScanID;
       DateTime? LastScanAt;
       uint TotalScans = 0;
       ScanIDType? LastBackgroundID;
       DateTime? LastBackgroundAt;
       ScanIDType? LastReferenceID;
       DateTime? LastReferenceAt;
       DateTime NoAdditionsSince;
       DateTime NoRemovealsSince;
};
ScanPersistenceGetFolderSummary (VolumeIDType Volume, FolderNameType Folder) ->
ScanPersistenceFolderSummaryType
```

RETURNS: ScanPersistenceFolderSummaryType

This returns a variety of data about the scan data in a given folder. This mostly is used for performance reasons, as most of the data can be extracted in other ways (except for NoAdditionsSince and NoRemovealsSince, but they are intended to be used to invalidate client side caches).

EXAMPLE:

```
ScanPersistenceGetFolderSummary (
      InternalStorage,
      “my experiment”
 ) -->
       TBD…
```

- **ScanPersistenceGetScanDetails**

```
struct PersistentScanDetailsType {
      ScanIDType ScanID;
      DateTime ScanStart;
      DateTime ScanEnd;
      Optional<String>       ScanLabel;
      Optional<ScanKindType> ScanKind;
      Optional<SpectrumType> RawSpectrum;
      Optional<SpectrumType> ProcessedSpectrum;
      PersistenceScanAuxDataType AuxData;
      Optional<ScanIDType> BackgroundID;
      Optional<ScanIDType> ReferenceID;
};
ScanPersistenceGetScanDetails (VolumeIDType Volume, FolderNameType Folder,
ScanIDType[] ScanIDs) -> PersistentScanDetailsType[];
```

RETURNS: PersistentScanDetailsType []

This returns lots of data about each scan, including the raw scan data, and any processed scan data (after applying backgrounds and/or references).

Note that if the ScanKind is missing (say a virtual scan) the RawSpectrum may also be omitted.

If the data has a Background, and/or reference, the ProcessedSpectrum will be returned with the background subtracted, and Reference applied (ratioed).

If any of the argument scanIds are missing or invalid, they will be ignored, so the resulting PersistentScanDetailsType array will be the same size, or less, than the argument ScanIDs array.

This means the caller CAN just create an array of N..M, where the caller increments values.

- **ScanPersistenceExport**

```
ScanPersistenceExport (VolumeIDType Volume, FolderNameType Folder, String
RequestedContentType, Optional<ScanIDType> LowerBound, Optional<ScanIDType>
UpperBound, Optional<Set<String>> OmitAuxKeys, const
Optional<Sequence<String>>& PreferredAuxKeys) -> BLOB;
```

RETURNS: BLOB (and internetMediaType)

Return a data file (e.g. zipfile, or excel file) suitable for user review with the scan data specified. If the lower or upper bounds are omitted, assume that means from the beginning of time, or end of time (for that folder).

If PreferredAuxKeys is provided, these AUXData items come first in the output display.

- **ScanPersistenceSubset**

```
ScanPersistenceSubset (VolumeIDType Volume, FolderNameType Folder,
Optional<ScanIDType> LowerBound, Optional<ScanIDType> UpperBound,
Optional<ScanKind> OnlyScanKind, Optional<unsigned int> MaxPoints, Optional<
PreferWhichPointsType> PreferWhichPoints) -> ScanIdType[];

ScanPersistenceSubset (VolumeIDType Volume, FolderNameType Folder,
Optional<DateTime> LowerBound, Optional<DateTime> UpperBound,
Optional<ScanKind> OnlyScanKind, Optional<unsigned int> MaxPoints Optional<
PreferWhichPointsType> PreferWhichPoints) -> ScanIdType[];
```

RETURNS: `ScanIdType[]`

Find a subset of points which is representative (for display) from the given set.

Note that `LowerBound` and `UpperBound` MAY refer to illegal values, or values from another folder.

This is NOT an error, and can be used to extend the range of selection. This method selects all items in the provided range THAT are in the given folder.

If OnlyScanKind is provided, only scans of that kind (e.g. reference) will be considered.

If MaxPoints is provided, PreferWhichPoints may also be provided to indicate if points should be evenly selected from the designated range, or if only the most recent 'MaxPoints' should be considered.

Note - ScanPersistenceAdvance() is better to use than adding one or two to the end of a range, because after a given scanID, there could be hundreds or thousands of deleted points, or points in another folder, that would NOT result in real data being returned.

Note also - if you pass in MaxPoints==2, and both a LowerBound and UpperBound, the returned two points will be exactly those LowerBound and UpperBound scanIds (subject to the usual constraints that the points must be valid for the given Volume/Folder).

MaxPoints: This optional parameter is used to limit the number of data returned. If the number of data points exceeds this limit, the controller should return a 'representative sample' of the data.

- **ScanPersistenceUpdate**

```
ScanPersistenceUpdate (VolumeIDType Volume, FolderNameType Folder, ScanIDType
ScanID, Optional<PersistenceScanAuxDataType> AuxData, Optional<String>
ScanLabel) -> void;
```

RETURNS: void

Update either the label, or the AuxData for a particular Scan. Note – no API to incrementally update the AUXData is provided. The caller must get the previous value, and replace it in its entirety.

It is an error to call this method with a ScanID that doesn't already exist in the given Volume/Folder.