We use the MAF (MS Addin Framework) structure for Inspire, but we’d propose that you simply make a class by the same name as our AddIn base, InsrumentConnectAddInView, to serve as your instrument base class. An instrument’s AddIn class is responsible primarily for connection and disconnection (whatever thay may entail) to/from the hardware, for handling a basic set of events, and for maintaining a heartbeat event for unexpected disconnect handling. Beyond that, a given instrument is coded by implementing capabilities.

The Instrument AddIn can implement any of the following capability interfaces:

* Probing
* Scanning
* Driving
* Weather
* Level
* Modeling

The AddIn must override the following definitions from the base

ConnectionAttributes GetConnectionParameters(string forThisModelID)

List<ConnectionAttributes> GetDiscoverableConnectionParameters(string forThisModelID)

bool Connect(ConnectionAttributes inputConnectionAttributes, out ConnectionAttributes connectedConnectionAttributes, out StatusChangedEventArgs conStatus)

bool Reinitialize()

void RegisterConnectionEventHandler(InstrumentConnectionHostEventHandlers handler)

UnitsProfile GetInstrumentUnitsProfile()

bool GetSystemInfo(out List<string> sysInfo)

bool HasSettingsToShow(out List<string> settingsNames)

bool HasHomeCommand()

bool Home()

bool ShowSettings(string settingsName, out InstrumentAction results)

void ApplyUIwithOkButton(InstrumentAction settings, bool closedWithOK)

bool Disconnect();

ProbeAddInView GetProbingInterface()

ScannerAddInView GetScanningInterface()

DriveAddInView GetDrivingInterface()

WeatherStationAddInView GetWeatherStationInterface()

LevelAddInView GetLevelInterface()
ModelAddInView GetModelingInterface()

The AddIn’s event handler, InstrumentConnectionHostEventHandler, is only responsible for the following.

void OnStatusChangedHandler(StatusChangedEventArgs args)

void OnPowerSourceInfo(PowerSource ps)

void OnPopUIwithOkButton(InstrumentAction contents, string okButtonLabel)

void OnUpdate(Measurement data)

void OnNextFeature()

void OnHeartbeat(bool beat)

Each of the capabilities Interfaces has its own event handler for e.g. measurements received, status changes, etc. Here is a closer look at the Probing and Scanning interfaces, with their abstract methods and event handlers:

ProbeAddInView interface

NVector approachPoint

double minimumApproachVectorLength

NUnitVector approachVector

ActionProfiles acProfiles = new ActionProfiles()

string GetProbeName()

double GetTipRad()

bool GetCurrentTarget(out double radiusMM, out string name, out bool isTargetSelectable, out TargetAttributes targAtt)

bool GetLastSelectedSelectableTarget(out double radiusMM, out string name, out TargetAttributes targAtt)

GetSelectableTargets()

bool SelectTarget(string name)

bool CanAcceptMeasureStopMeasureCommands()

ProgrammableButtons GetInitialProgrammableButtons()

bool SetProgrammableButtons(ProgrammableButtons progButtons)

bool ProgrammableButtonsAreActive()

ProgrammableButtons GetInitialProgrammableRemoteButtons()

bool SetProgrammableRemoteButtons(ProgrammableButtons progButtons)

ActionProfiles GetAcquisitionProfiles()

bool SetAcquisitionProfiles(ActionProfiles acquisitionProfiles)

string GetCurrentAcquisitionProfile()

bool SetCurrentAcquisitionProfile(string acProfileName)

bool InstrumentActionButtonPress(InstrumentAction profile, out InstrumentAction updatedProfile)

bool Measure(string acquisition)

bool StopMeasure(string acquisition)

bool QueryPosition(out InstrumentAction pos)

bool HasGraphicViewCameraTransform()

NTransform GetGraphicViewCameraTransform()

void RegisterProbeEventHandler(ProbeHostEventHandlers handler)

And the ProbeHostEventHandlers are responsible for these events

void OnButtonEvent(ProgrammableButton button);

void OnMeasurement(Measurement data);

void OnTipChange(double radiusMM, string name, bool isTargetSelectable, TargetAttributes targAtt);

void OnMeasureProfleChange(string name)

void OnPointScanStarted()

void OnPointScanStopped()

void OnPointProbingStarted()

void OnPointProbingStopped()

ScannerAddInView interface

string GetScannerName()

bool InstrumentHasScannerSwitch()

InstrumentAction GetNRKFilterParameters()

bool SetNRKFilterParameters(InstrumentAction lineScanFilter)

bool SetScanRegion(InstrumentAction region)

void ChangeSettings()

bool CanAcceptBeginEndScanCommands()

bool BeginScan()

bool EndScan()

bool HasGraphicViewCameraTransform()

NTransform GetGraphicViewCameraTransform()

void RegisterScannerEventHandler(ScannerHostEventHandlers handler)

ScannerHostEventHandlers events

void OnScannerSwitchedOn();

void OnScannerSwitchedOff();

void OnScanStarted();

void OnScanStopped();

void OnMeasurement(Measurement data);

void OnUpdate(Measurement data);

Note that Probing and Scanning use the same class for data, Measurement. When the instrument api receives data, it notifies the parent app via OnMeasurement for both Probing and Scanning. Here is the Measurement class.

List<string> info

List<ProbeMeasurement> pointData

NScannedPointCloud cloudData

bool? isPatch

ScanInfo scanInfo

NMesh meshData

public Measurement() // ctor

{

 pointData = new List<ProbeMeasurement>();

 info = new List<string>();

 cloudData = null;

 isPatch = null;

 scanInfo = null;

 meshData = null;

}

Add3dProbeMeas(NVector p3d, NUnitVector approach, DateTime? dateTime = null, double? temp = null, double? press = null, double? humid = null, double? rms = null)

{

 ProbeMeasurement meas = new ProbeMeasurement();

 meas.SetPoint3d(p3d, dateTime, temp, press, humid, rms);

 meas.SetApproachVector(approach);

 pointData.Add(meas);

}

public void Add6dProbeMeas(NTransform p6d, NUnitVector approach, DateTime? dateTime = null, double? temp = null, double? press = null, double? humid = null, double? rms = null, double? rms6dRotation = null)…

Another significant support class is the InstrumentAction. Note its use for button press events, probing acquisition profiles, and scanner paramters. This class is useful because it generalizes instrument actions. It is comprised of arrays of InstrumentActionParameters, which contain not only data and type information, but name information. In this way, an Instrument action contains all the information needed by the ui side to create its presentation to the user. Here is an example of using the InstrumentAction class, defining a measure profile for a targetless scanner. Note that InstrumentActions can be set to not show in the ui, and can be set as read only:

// create the Acquisition

var surfaceScanAc = new InstrumentAction(InstrumentAction.AcType.**SurfaceScan**);

surfaceScanAc.name = @"Area Scan";

// create the parameters (set current values from hardware)

var lmfSurfScan = lmfProfiles[**i**] as AreaScanProfile;

// create the parameters (set current values from hdw)

var speedParams = new ActionParamStringList(InstStrings.**Accuracy**, InstStrings.**Fast**);

speedParams.AddMode(InstStrings.**Fast**, true);

speedParams.AddMode(InstStrings.**Standard**);

speedParams.AddMode(InstStrings.**Precise**);

var scanMode = new ActionParamStringList(@"Scan Mode", @"Continuous");

scanMode.AddMode(@"None");

scanMode.AddMode(@"StopAndGo");

scanMode.AddMode(@"Continuous", true);

// Leica's AreaScanMode parameter removed as of LMF v.1.6.0.5319

scanMode.showMe = false;

// create and set our params that don't come from Leica's

var sendAsCloud = new ActionParamBool(@"Send as Cloud", true);

sendAsCloud.showMe = true;

var intensityFilter = new ActionParamDoubleWithRange(@"Intensity, Reject Below", 0, 0, 100, InstrumentActionParameter.UnitType.**percent**);

intensityFilter.showMe = true;

var qualityFilter = new ActionParamStringList(@"Quality Filter", @"Medium");

qualityFilter.AddMode(@"Off");

qualityFilter.AddMode(@"Low");

qualityFilter.AddMode(@"Medium", true);

qualityFilter.AddMode(@"High");

qualityFilter.showMe = true;

var newFeaturePerRegion = new ActionParamBool(@"New Feature Per Region", true);

newFeaturePerRegion.showMe = false;

var setRegionsButton = new ActionParamButton(@"Define Regions");

var estimatedScanTime = new ActionParamTimeSpan(lmfSurfScan.EstimatedScanTime.Label, lmfSurfScan.EstimatedScanTime.Value);

estimatedScanTime.isReadOnly = true;

var regions = new ActionParamRegions(@"Scan Regions");

regions.showMe = false;

// add the parameters to this Acquisition

surfaceScanAc.parameters.Add(speedParams);

surfaceScanAc.parameters.Add(scanMode);

surfaceScanAc.parameters.Add(sendAsCloud);

surfaceScanAc.parameters.Add(intensityFilter);

surfaceScanAc.parameters.Add(qualityFilter);

surfaceScanAc.parameters.Add(newFeaturePerRegion);

surfaceScanAc.parameters.Add(setRegionsButton);

surfaceScanAc.parameters.Add(estimatedScanTime);

surfaceScanAc.parameters.Add(regions);

// and set all our params to equal theirs

SetOurAreaScanFromLeicas(surfaceScanAc);

// add this Acquisition to the Profiles

acProfiles.acs.Add(surfaceScanAc);

One more example of using the InstrumentAction class, defining a measure profile for a spatially incremented probe scan:

// create the Acquisition

var spatialScanAc = new InstrumentAction(InstrumentAction.AcType.**SpatialScan**);

spatialScanAc.name = InstStrings.**ContinuousDistance**;

// create and set the parameters (set current values from hdw where applicable)

var lmfSpatScan = lmfProfiles[**i**] as ContinuousDistanceProfile;

var scanInc = new ActionParamDoubleWithRange(InstStrings.**Separation**, lmfSpatScan.DistanceSeparation.Value, lmfSpatScan.DistanceSeparation.MinValue, lmfSpatScan.DistanceSeparation.MaxValue, InstrumentActionParameter.UnitType.length);

var stableStart = new ActionParamBool(InstStrings.**StableStartForSMR**, true);

var loopToIterate = new ActionParamBool(@"Loop to Iterate for SMR", false);

var maxTimeSinceLastPtToIterate = new ActionParamBool(@"Stable to Iterate for SMR", false);

// add the parameters to this Acquisition

spatialScanAc.parameters.Add(scanInc);

spatialScanAc.parameters.Add(stableStart);

spatialScanAc.parameters.Add(loopToIterate);

spatialScanAc.parameters.Add(maxTimeSinceLastPtToIterate);

// add this Acquisition to the Profiles

acProfiles.acs.Add(spatialScanAc);