Research Mode API

Contents

Overview

Research mode was introduced in the 1st Gen HoloLens to give research applications not intended for deployment access to key sensors on the device. Research mode for HoloLens 2 retains the capabilities of HoloLens 1, adding access to additional streams. Similarly, to the 1st version, it is possible to gather data from the following inputs:

- *Visible Light Environment Tracking Cameras* - Used by the system for head tracking and map building. They return grey scale 8bit per pixel images.
- *Depth Camera* – Operates in two modes:
- Articulated HAnd-Tracking mode (AHAT), high-frequency (45 FPS) near-depth sensing used for hand tracking. As hands are supported up to 1m from the device, the HoloLens 2 saves power by calculating only "aliased depth" from the phase-based time of flight camera. This means that the signal contains only the fractional part of the distance from the device when expressed in meters.
    - Long-throw, low-frequency (1-5 FPS) far-depth sensing used by Spatial Mapping.
- *Two versions of the IR-reflectivity stream* - Used by the HoloLens to compute depth. These images are illuminated by infrared and unaffected by ambient visible light.

In addition, HoloLens 2 also supports access to the following:

- *Accelerometer* – Used by the system to determine linear acceleration along the X, Y and Z axes as well as gravity.
- *Gyroscope* – Used by the system to determine rotations.
- *Magnetometer* – Used by the system for absolute orientation estimation.

Outline

Research Mode API is based on a light-weight derivation of COM called Nano-COM. Nano-COM refers to an API design pattern which uses IUnknown for object identity and lifetime, but without requiring the COM runtime infrastructure, replacing use of CoCreateInstance with factory functions that return objects initialized with the parameters to those functions. The interfaces only support QueryInterface, AddRef and Release. APIs return HRESULT error codes. DirectX11 and 12 are also a Nano-COM APIs.

API is structured as follows:

- A research mode device is the first object created. This is the API factory object. It is used to:
    - Enumerate available sensors by type,
    - Create sensor objects,
    - Request access consent.
    - Only one sensor per sensor type can be created,
- Sensors provide the following functionalities:
    - Return sensor name and type,
    - Start and stop streaming,
    - In Streaming state wait for and retrieve frames,

    - Return extrinsics matrices that give relative position of the sensor relative to a device attached origin (Rig Origin),
    - Return Rig Coordinate frame GUID that can be used to map the Rig coordinate frame to other perception coordinate frame,
    - Sensors can be cameras or IMUs and both return frames with sensor-specific payload formats.
- Sensor Frames provide:
    - Frame timestamps,
    - Frame sizes,
    - Specialized per-sensor properties and payload formats.

The initialization call should be made only once for all sensors and sensors are not thread safe. Frames should be read from the thread the sensor was opened on. Sensors can share a thread or have a thread each.

Main Sensor Reading Loop

The outline of the main sensor processing loop is:

- Create Research mode device,
- Get the device coordinate frame in which all sensors are positioned. We call this the rigNode and it is identified by a GUID that can be used with the HoloLens perception APIs to map sensor specific coordinates in other HoloLens perception coordinate frames. The following https://docs.microsoft.com/en-us/windows/mixed-reality/coordinate-systems explains Perception coordinate frames.
- Enumerate sensors,
- Get information from sensors:
    - For cameras, Camera object to project / unproject image points to 3d points in the camera coordinate frame,
    - Extrinsics for positioning the sensor with respect to the device rigNode.

The code below shows opening the research mode device, getting sensor descriptors and getting frames from sensors. APIs return HRSULTS that should be checked for errors. Error checking is left out of the code below to make it easier to follow the API calls. Coordinate frame related APIs will be described in a later section.

```
HRESULT hr = S_OK;
IResearchModeSensorDevice *pSensorDevice;
IResearchModeSensorDevicePerception *pSensorDevicePerception;
std::vector<ResearchModeSensorDescriptor> sensorDescriptors;
size_t sensorCount = 0;

hr = CreateResearchModeSensorDevice(&pSensorDevice);

// This call makes cameras run at full frame rate. Normaly they are optimized
// for headtracker use. For some applications that may be sufficient
pSensorDevice->DisableEyeSelection();

hr = pSensorDevice->GetSensorCount(&sensorCount);
```

```
    sensorDescriptors.resize(sensorCount);

    hr = pSensorDevice->GetSensorDescriptors(sensorDescriptors.data(),
sensorDescriptors.size(), &sensorCount);

    for (const auto& sensorDescriptor : sensorDescriptors)
    {
        // Sensor frame read thread

        IResearchModeSensor *pSensor = nullptr;
        size_t sampleBufferSize;
        IResearchModeSensorFrame* pSensorFrame = nullptr;

        hr = pSensorDevice->GetSensor(sensorDescriptor.sensorType, &pSensor);

        swprintf_s(msgBuffer, L"Sensor %ls\n", pSensor->GetFriendlyName());
        OutputDebugStringW(msgBuffer);

        hr = pSensor->GetSampleBufferSize(&sampleBufferSize);

        hr = pSensor->OpenStream();

        for (UINT i = 0; i < 4; i++)
        {
            hr = pSensor->GetNextBuffer(&pSensorFrame);

            if (pSensor->GetSensorType() >= IMU_ACCEL)
            {
                ProcessFrameImu(pSensor, pSensorFrame, i);
            }
            else
            {
                ProcessFrameCamera(pSensor, pSensorFrame, i);
            }

            if (pSensorFrame)
            {
                pSensorFrame->Release();
            }
        }

        hr = pSensor->CloseStream();

        if (pSensor)
        {
            pSensor->Release();
        }
    }

    pSensorDevice->EnableEyeSelection();

    pSensorDevice->Release();

    return hr;
```

The code above shows all sensors read on the same thread. Due to the fact that the GetNextBuffer calls are blocking, each sensor frame loop should be run on its own thread. This allows processing each sensor at its own frame rate.

OpenStream and GetNextBuffer need to be called from the same thread. GetNextBuffer calls are blocking. Sensor frame loops for each sensor should be run on their own thread. This allows sensors to be processed on their own frame rate. The following threading pattern is recommended:

- Main thread manages ResearchMode device and sensors,
- Each sensor has a thread that opens the sensor streams and reads buffers and processes buffers,
- Main thread renders buffers and results

```
SensorLoop(IResearchModeSensor *pSensor)
{
    hr = pSensor->OpenStream();

    while (fRunning)
    {
        hr = pSensor->GetNextBuffer(&pSensorFrame);

        ProcessFrame(pSensor, pSensorFrame, i);

        if (pSensorFrame)
        {
            pSensorFrame->Release();
        }
    }

    hr = pSensor->CloseStream();
}
```

Sensor types

Camera Sensors

- Intrinsics (project/unproject)
- These functions in the camera coordinate space
- Extrinsics returns R, T transform in rig space
- Frames are specialized for camera frames

IMU Sensors

- Extrinsics returns R, T transform in rig space
- Frames are specialized for IMU frames

Sensor Coordinate Frames

Each sensor returns its transform to the rigNode (Rig origin) expressed as an extrinsics rigid body transform. Figure 1 shows camera coordinate frames relative to rig coordinate frame. Note that, on the

HoloLens 2, the device origin corresponds to the Left Front Visible Light Camera. Therefore, the transformation returned by this sensor corresponds to the identity transformation.

The extrinsics transform can be retrieved as follows:

```
IResearchModeCameraSensor *pCameraSensor;
DirectX::XMFLOAT4X4 cameraPose;
// …
// Get matrix of extrinsics wrt the rigNode
pCameraSensor->GetCameraExtrinsicsMatrix(&cameraPose);
```

To map the rigNode (and hence the device) in other HoloLens perception coordinate frames, one can use the Perception APIs.

```
using namespace winrt::Windows::Perception::Spatial;
using namespace winrt::Windows::Perception::Spatial::Preview;

SpatialLocator locator;
IResearchModeSensorDevicePerception* pSensorDevicePerception;
GUID guid;
HRESULT hr = m_pSensorDevice->QueryInterface(IID_PPV_ARGS(&pSensorDevicePerception));
if (SUCCEEDED(hr))
{
    hr = pSensorDevicePerception->GetRigNodeId(&guid);
    locator = SpatialGraphInteropPreview::CreateLocatorForNode(guid);
}
// …
auto location = locator.TryLocateAtTimestamp(timestamp, anotherCoordSystem);
```

Camera sensors expose map / unmap methods to project 3D points in the camera. Figure 3 shows the relationship between 3D coordinates in camera reference frame and 2D image coordinates.

Map / unmap methods can be used as follows:

```
IResearchModeCameraSensor *pCameraSensor;
//…
float xy[2] = {0};
float uv[2] = {0};
float uv_mapped[2] = {0};

for (int i = 0; i <= 10; i++)
{
       for (int j = 0; j <= 10; j++)
       {
           // VLC images are 640x480
           uv[0] = i * 64.0f;
           uv[1] = j * 48.0f;

           pCameraSensor->MapImagePointToCameraUnitPlane(uv, xy);
           // …
           pCameraSensor->MapCameraSpaceToImagePoint(xy, uv_mapped);
           // …
       }
}
```

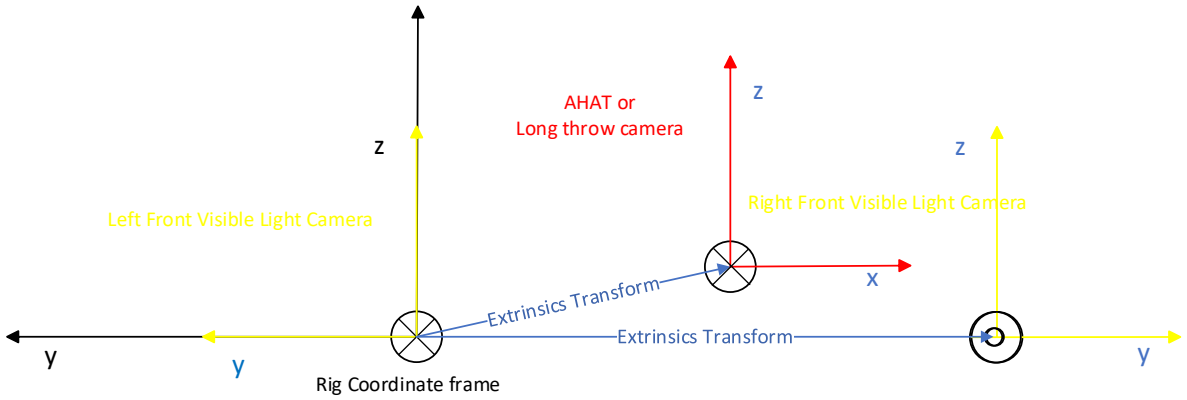


Figure 1 Depth and Front visible light camera coordinate frames relative to rig node coordinate frame. Long throw and AHAT are different modes of the same camera so the extrinsics are the same.

Figure 2 Hololens cameras. Yellow are the VLC cameras and Red is the depth camera

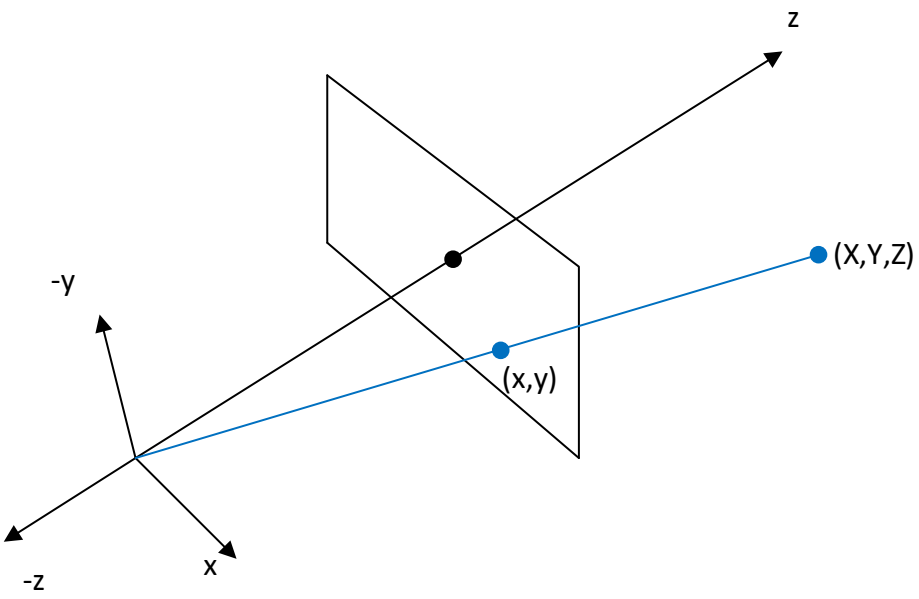


Figure 3 Camera Map Unmap methods convert 3d (X,Y,Z) coordinates in camera reference frame into camera (x,y) image coordinates, and (x,y) image coordinates into (X,Y,Z) direction vectors in camera coordinate frames.

Sensors

The IResearchModeSensor abstracts the research mode sensors. It provides methods and properties common to all sensors:

```
DECLARE_INTERFACE_IID_(IResearchModeSensor, IUnknown, "4D4D1D4B-9FDD-4001-BA1E-
F8FAB1DA14D0")
{
    STDMETHOD(OpenStream()) = 0;
    STDMETHOD(CloseStream()) = 0;
    STDMETHOD_(LPCWSTR, GetFriendlyName)() = 0;
    STDMETHOD_(ResearchModeSensorType, GetSensorType)() = 0;

    STDMETHOD(GetSampleBufferSize(
        _Out_ size_t *pSampleBufferSize)) = 0;
    STDMETHOD(GetNextBuffer(
        _Outptr_result_nullonfailure_ IResearchModeSensorFrame **ppSensorFrame)) = 0;
};
```

- OpenStream puts the sensor in a state in which it produces frames. This has to be called before retrieving buffers,
- CloseStream stops frame capture,
- GetFriendlyName returns a string containing the sensor name,
- GetSensorType returns the sensor type,
- GetNextBuffer returns the next available buffer. It is a blocking call.

Sensors can be of the following types:

```
enum ResearchModeSensorType
{
```

```
    LEFT_FRONT,
    LEFT_LEFT,
    RIGHT_FRONT,
    RIGHT_RIGHT,
    DEPTH_AHAT,
    DEPTH_LONG_THROW,
    IMU_ACCEL,
    IMU_GYRO,
    IMU_MAG
};
```

Each sensor object defines sensor specific interfaces that can by QIed for. These will retrieve sensor specific information.

Sensor Frames

Once a sensor is in streaming mode, sensor frames are retrieved from the sensor with IResearchModeSensor::GetNextBuffer. All sensor frames have a common interface that returns frame information common to all types of frames. The memory containing the frame data in the buffer is owned by the frame object. When the frame interface is released the memory is released with it. Frame interfaces provide methods that can be used to access data contained by the frame.

```
DECLARE_INTERFACE_IID_(IResearchModeSensorFrame, IUnknown, "73479614-89C9-4FFD-9C16-
615BC32C6A09")
{
    STDMETHOD(GetResolution(
        _Out_ ResearchModeSensorResolution *pResolution)) = 0;
    // For frames with batched samples this returns the time stamp for the first sample
in the frame.
    STDMETHOD(GetTimeStamp(
        _Out_ ResearchModeSensorTimestamp *pTimeStamp)) = 0;
};
```

All sensor frame interfaces are listed in Appendix Sensor Frames.

Each sensor has its own frame specialized interfaces.

- All frame types:
    - Frame Times stamps. These are in HostTicks and SensorTicks. HostTicks are CPU times in filetime units and SensorTicks are sensor ticks in nanoseconds.
    - Sample size in bytes.
- Camera frames:
    - All camera frames provide getters for resolution, exposure, gain.
    - VLC camera frames return grayscale buffer.
    - Depth Long throw camera frames contain active brightness buffer, distance buffer and the sigma buffer.
    - Depth AHAT camera frames contain an active brightness buffer and a distance buffer.
- IMU frames contain batches of sensor samples. Each sensor sample is a struct containing a sensor value with its corresponding SocTicks (HostTicks), VinylHupTicks (SensorTicks) and temperature:
    - Accelerometer values are three m/s^2 accelerations,
    - Gyro meter values are three angular speeds in deg/s.

- Magnetometer frames contain magnetometer values.

VLC Frame Payload

VLC frames implement the following interface:

```
DECLARE_INTERFACE_IID_(IResearchModeSensorVLCFrame, IUnknown, "5C693123-3851-4FDC-A2D9-
51C68AF53976")
{
    STDMETHOD(GetBuffer(
        _Outptr_ const BYTE **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetGain(
        _Out_ UINT32 *pGain)) = 0;
    STDMETHOD(GetExposure(
        _Out_ UINT64 *pExposure)) = 0;
};
```

GetBuffer returns a pointer to memory containing frame of gray scale pixels. These are row major byte pixels with values from 0 to 255. The size of the buffer is obtained from the IResearchModeSensorFrame::GetResolution interface of the frame.
Gain is a value from zero to 255 and exposure is in nanoseconds.

The code below shows how resolution, exposure, gain, timestamp and image data are extracted from a VLC frame.

```
void ProcessFrame(IResearchModeSensor *pSensor, IResearchModeSensorFrame* pSensorFrame,
int bufferCount)
{
    ResearchModeSensorResolution resolution;
    ResearchModeSensorTimestamp timestamp;
    wchar_t filename[260];
    const BYTE *pImage = nullptr;
    IResearchModeSensorVLCFrame *pVLCFrame = nullptr;
    HRESULT hr = S_OK;
    size_t outBufferCount;

    pSensorFrame->GetResolution(&resolution);
    pSensorFrame->GetTimeStamp(&timestamp);

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pVLCFrame));

    if (SUCCEEDED(hr))
    {
        UINT32 gain;
        UINT64 exposure;

        pVLCFrame->GetBuffer(&pImage, &outBufferCount);

        // Add code to process frame pixels here.

        swprintf_s(filename, L"%s_%d_ts%d.bmp", pSensor->GetFriendlyName(), bufferCount,
            timestamp.HostTicks);
        // The pixel data is at pImage memory address. Pixels are BYTES from 0-255 and
        // frame is for major.
```

```
        swprintf_s(filename, L"  %S_%d_ts%d.bmp\n", pSensor->GetFriendlyName(),
bufferCount, timestamp.HostTicks);
        OutputDebugStringW(filename);

        hr = pVLCFrame->GetGain(&gain);

        if (SUCCEEDED(hr))
        {
            swprintf_s(filename, L"  Gain %d\n", gain);
            OutputDebugStringW(filename);
        }

        hr = pVLCFrame->GetExposure(&exposure);

        if (SUCCEEDED(hr))
        {
            swprintf_s(filename, L"  Exposure %d\n", exposure);
            OutputDebugStringW(filename);
        }
}
```

AHAT and Long Throw Camera Frame Payload

Depth frames implement the following interface:

```
DECLARE_INTERFACE_IID_(IResearchModeSensorDepthFrame, IUnknown, "35167E38-E020-43D9-898E-
6CB917AD86D3")
{
    STDMETHOD(GetBuffer(
        _Outptr_ const UINT16 **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetAbDepthBuffer(
        _Outptr_ const UINT16 **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetSigmaBuffer(
        _Outptr_ const BYTE **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
};
```

The depth camera frame in Long Throw mode has a depth buffer and a sigma buffer used to invalidate depth pixels, and an active brightness (Ab) buffer. In AHAT mode it only has the depth buffer and the active brightness buffer.

The active brightness buffer returns a so-called IR reading. The value of pixels in the clean IR reading is proportional to the amount of light returned from the scene. The image looks similar to a regular IR image.

The sigma buffer for Long Throw is used to invalidate unreliable depth based on the invalidation mask computed from the depth algorithm. For AHAT, for efficiency purposes, the invalidation code is embedded in the depth channel itself.

Long Throw invalidation

LT has the invalidation codes and confidence embedded in the sigma buffer, which is 8-bit data for every pixel. If the most significant bit (MSB) is set to 1, then the other 7 bits represent the invalidation reason. The invalidation masks are:

- Invalid = 0x80, // MSB
- OutOfBounds = 0xC0, // Outside of active IR illumination mask
- SignalSaturated = 0xA0, // Saturated IR signal
- FilterOutlier = 0x90, // Filter outlier
- EmptySignal = 0x88, // Low IR signal
- MultiPathDetected = 0x84, // Multi-path interference detected
  // (receiving signals from more than one object in the scene)
- OutOfRangeFar = 0x82, // Exceeded max supporting range (set to 7500mm)
- OutOfRangeNear = 0x81 // Exceeded min supporting range (set to 200mm)

AHAT invalidation

For AHAT, invalidation codes are embedded in the depth channel. Pixels with values greater than 4090 are invalid. Invalidation codes are:

- 4095: Outside of active IR illumination mask
- 4093: Low IR signal

Reading AHAT and Long Throw depth frames:
Resolution, exposure, gain and timestamp can be extracted as above. Code below shows how to extract and process buffers from Long Throw and AHAT frames.

```
void ProcessFrame(IResearchModeSensor *pSensor, IResearchModeSensorFrame* pSensorFrame,
int bufferCount)
{
    ResearchModeSensorResolution resolution;
    ResearchModeSensorTimestamp timestamp;
    wchar_t filename[260];
    IResearchModeSensorDepthFrame *pDepthFrame = nullptr;
    const UINT16 *pAbImage = nullptr;
    const UINT16 *pDepth = nullptr;

    // sigma buffer needed only for Long Throw
    const BYTE *pSigma = nullptr;

    // invalidation mask for Long Throw
    USHORT mask = 0x80;

    // invalidation value for AHAT
    USHORT maxValue = 4090;

    HRESULT hr = S_OK;
    size_t outBufferCount;

    pSensorFrame->GetResolution(&resolution);
    pSensorFrame->GetTimeStamp(&timestamp);

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pDepthFrame));
```

```
    bool isLongThrow = (pSensor->GetSensorType() == DEPTH_LONG_THROW);

    if (SUCCEEDED(hr) && isLongThrow)
    {
       // extract sigma buffer for Long Throw
       hr = pDepthFrame->GetSigmaBuffer(&pSigma, &outBufferCount);
       // Add code to process buffer here.
    }

    if (SUCCEEDED(hr))
    {
        // extract depth buffer
        hr = pDepthFrame->GetBuffer(&pDepth, &outBufferCount);
        // validate depth
        for (size_t i = 0; i < outBufferCount; ++i)
        {
             // use a different invalidation condition for Long Throw and AHAT
             const bool isInvalid = isLongThrow ? ((pSigma[i] & mask) > 0) :
                                                  (pDepth[i] >= maxValue));
             if (isInvalid)
             {
                 pDepth[i] = 0;
             }
        }
        // Add code to process buffer here.
    }

    if (SUCCEEDED(hr))
    {
        // extract active brightness buffer
        hr = pDepthFrame->GetAbDepthBuffer(&pAbImage, &outBufferCount);
        // Add code to process buffer here.
    }

    if (pDepthFrame)
    {
        pDepthFrame->Release();
    }
}
```

IMU Frame Payload

IMU frames implement on of the following interfaces:

```
DECLARE_INTERFACE_IID_(IResearchModeAccelFrame, IUnknown, "42AA75F8-E3FE-4C25-88C6-
F2ECE1E8A2C5")
{
    STDMETHOD(GetCalibratedAccelaration(
        _Out_ DirectX::XMFLOAT3 *pAccel)) = 0;
    STDMETHOD(GetCalibratedAccelarationSamples(
        _Outptr_ const AccelDataStruct **ppAccelBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeGyroFrame, IUnknown, "4C0C5EE7-CBB8-4A15-A81F-
943785F524A6")
{
    STDMETHOD(GetCalibratedGyro(
```

```
        _Out_ DirectX::XMFLOAT3 *pGyro)) = 0;
    STDMETHOD(GetCalibratedGyroSamples(
        _Outptr_ const GyroDataStruct **ppAccelBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeMagFrame, IUnknown, "2376C9D2-7F3D-456E-A39E-
3B7730DDA9E5")
{
    STDMETHOD(GetMagnetometer(
        _Out_ DirectX::XMFLOAT3 *pMag)) = 0;
    STDMETHOD(GetMagnetometerSamples(
        _Outptr_ const MagDataStruct **ppMagBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};
```

- Accelerometer frames – Contain linear acceleration along the X, Y and Z axes as well as gravity.
- Gyroscope frames– Contain rotations.
- Magnetometer – Contain absolute orientation estimation.

IMU frames contain batches of IMU samples. Each sample is one of the following:

```
struct AccelDataStruct
{
    uint64_t VinylHupTicks; // Sensor ticks in micro seconds
    uint64_t SocTicks;
    float AccelValues[3]; // In m/(s*s)
    float temperature;
};

struct GyroDataStruct
{
    uint64_t VinylHupTicks; // Sensor ticks in micro seconds
    uint64_t SocTicks;
    float GyroValues[3];
    float temperature;
};

struct MagDataStruct
{
    uint64_t VinylHupTicks; // Sensor ticks in micro seconds
    uint64_t SocTicks;
    float MagValues[3];
};
```

Read a single sample of the IMU frame:
The code below shows how IMU data will be extracted from a single sample of the IMU frame

```
void PrintSensorValue(IResearchModeSensorFrame *pSensorFrame)
{
    DirectX::XMFLOAT3 sample;
    IResearchModeGyroFrame *pSensorGyroFrame = nullptr;
    IResearchModeAccelFrame *pSensorAccelFrame = nullptr;
    IResearchModeMagFrame *pSensorMagFrame = nullptr;
    char printString[1000];
    HRESULT hr = S_OK;
```

```
    ResearchModeSensorTimestamp timeStamp;
    UINT64 lastSocTickDelta = 0;

    pSensorFrame->GetTimeStamp(&timeStamp);

    if (glastSocTick != 0)
    {
        lastSocTickDelta = timeStamp.HostTicks - glastSocTick;
    }
    glastSocTick = timeStamp.HostTicks;

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorAccelFrame));
    if (SUCCEEDED(hr))
    {
        hr = pSensorAccelFrame->GetCalibratedAccelaration(&sample);
        if (FAILED(hr))
        {
            return;
        }
        sprintf(printString, "####Accel: % 3.4f % 3.4f % 3.4f %f %d\n",
                sample.x,
                sample.y,
                sample.z,
                sqrt(sample.x * sample.x + sample.y * sample.y + sample.z * sample.z),
                (lastSocTickDelta * 1000) / timeStamp.HostTicksPerSecond
                );
        OutputDebugStringA(printString);
        pSensorAccelFrame->Release();
        return;
    }

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorGyroFrame));
    if (SUCCEEDED(hr))
    {
        hr = pSensorGyroFrame->GetCalibratedGyro(&sample);
        if (FAILED(hr))
        {
            return;
        }
        sprintf(printString, "####Gyro: % 3.4f % 3.4f % 3.4f %f %d\n",
                sample.x,
                sample.y,
                sample.z,
                sqrt(sample.x * sample.x + sample.y * sample.y + sample.z * sample.z),
                (lastSocTickDelta * 1000) / timeStamp.HostTicksPerSecond
                );
        OutputDebugStringA(printString);
        pSensorGyroFrame->Release();
        return;
    }

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorMagFrame));
    if (SUCCEEDED(hr))
    {
        hr = pSensorMagFrame->GetMagnetometer(&sample);
        if (FAILED(hr))
        {
            return;
```

```
        }
        sprintf(printString, "####Mag: % 3.4f % 3.4f % 3.4f %d\n",
                sample.x,
                sample.y,
                sample.z,
                (lastSocTickDelta * 1000) / timeStamp.HostTicksPerSecond
                );
        OutputDebugStringA(printString);
        pSensorMagFrame->Release();
        return;
    }
}
```

Read all the IMU samples of the IMU frame:
The code below shows how IMU data will be extracted from all the samples of the IMU frame

```
void PrintSensorValue(IResearchModeSensorFrame *pSensorFrame)
{
    DirectX::XMFLOAT3 sample;
    IResearchModeGyroFrame *pSensorGyroFrame = nullptr;
    IResearchModeAccelFrame *pSensorAccelFrame = nullptr;
    IResearchModeMagFrame *pSensorMagFrame = nullptr;
    char printString[1000];
    HRESULT hr = S_OK;
    ResearchModeSensorTimestamp timeStamp;
    UINT64 lastSocTickDelta = 0;

    pSensorFrame->GetTimeStamp(&timeStamp);

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorAccelFrame));
    if (SUCCEEDED(hr))
    {
        const AccelDataStruct *pAccelBuffer;
        size_t BufferOutLength;
        hr = pSensorAccelFrame->GetCalibratedAccelarationSamples(
            &pAccelBuffer,
            &BufferOutLength);
        if (FAILED(hr))
        {
            return;
        }
        for (UINT i = 0; i < BufferOutLength; i++)
        {
            sample.x = pAccelBuffer[i].AccelValues[0];
            sample.y = pAccelBuffer[i].AccelValues[1];
            sample.z = pAccelBuffer[i].AccelValues[2];
            if (glastHupTick != 0)
            {
                lastSocTickDelta =  pAccelBuffer[i].VinylHupTicks - glastHupTick;
                sprintf(printString, "####Accel-%3d-%3d-%3d:
% 3.4f % 3.4f % 3.4f %f %d\n",
                        gBatchCount,
                        i,
                        BufferOutLength,
                        sample.x,
```

```
                        sample.y,
                        sample.z,
                        sqrt(sample.x * sample.x + sample.y * sample.y + sample.z *
sample.z),
                        lastSocTickDelta / 1000 // micro seconds
                        );
            }
            glastHupTick = pAccelBuffer[i].VinylHupTicks;
            OutputDebugStringA(printString);
        }
        gBatchCount++;
        pSensorAccelFrame->Release();
        return;
    }

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorGyroFrame));
    if (SUCCEEDED(hr))
    {
        const GyroDataStruct *pGyroBuffer;
        size_t BufferOutLength;
        hr = pSensorGyroFrame->GetCalibratedGyroSamples(
            &pGyroBuffer,
            &BufferOutLength);
        if (FAILED(hr))
        {
            return;
        }
        for (UINT i = 0; i < BufferOutLength; i++)
        {
            sample.x = pGyroBuffer[i].GyroValues[0];
            sample.y = pGyroBuffer[i].GyroValues[1];
            sample.z = pGyroBuffer[i].GyroValues[2];
            if (glastHupTick != 0)
            {
                lastSocTickDelta =  pGyroBuffer[i].VinylHupTicks - glastHupTick;
                sprintf(printString, "####Gyro-%3d-%3d-%3d: % 3.4f % 3.4f % 3.4f %f
%d\n",
                        gBatchCount,
                        i,
                        BufferOutLength,
                        sample.x,
                        sample.y,
                        sample.z,
                        sqrt(sample.x * sample.x + sample.y * sample.y + sample.z *
sample.z),
                        lastSocTickDelta / 1000 // micro seconds
                        );
            }
            glastHupTick = pGyroBuffer[i].VinylHupTicks;
            OutputDebugStringA(printString);
        }
        gBatchCount++;
        pSensorGyroFrame->Release();
        return;
    }

    hr = pSensorFrame->QueryInterface(IID_PPV_ARGS(&pSensorMagFrame));
    if (SUCCEEDED(hr))
```

```
    {
        const MagDataStruct *pMagBuffer;
        size_t BufferOutLength;
        hr = pSensorMagFrame->GetMagnetometerSamples(
            &pMagBuffer,
            &BufferOutLength);
        if (FAILED(hr))
        {
            return;
        }
        for (UINT i = 0; i < BufferOutLength; i++)
        {
            sample.x = pMagBuffer[i].MagValues[0];
            sample.y = pMagBuffer[i].MagValues[1];
            sample.z = pMagBuffer[i].MagValues[2];
            if (glastHupTick != 0)
            {
                lastSocTickDelta =  pMagBuffer[i].VinylHupTicks - glastHupTick;
                sprintf(printString, "####Mag-%3d-%3d: % 3.4f % 3.4f % 3.4f %d\n",
                        gBatchCount,
                        i,
                        sample.x,
                        sample.y,
                        sample.z,
                        lastSocTickDelta / 1000 // micro seconds
                        );
            }
            glastHupTick = pMagBuffer[i].VinylHupTicks;
            OutputDebugStringA(printString);
        }
        gBatchCount++;
        pSensorMagFrame->Release();
        return;
    }
}
```

Consent Prompts

Any UWP application using Research Mode API for accessing cameras or IMUs must request user consent before opening the streams. Depending on the user input the app should further proceed.

The following steps outline adding required code for consent prompts in the UWP app:

- For enabling user consent for the camera and IMU access, please make sure to declare the following capabilities to the app manifest:

```
<DeviceCapability Name="webcam" />
<DeviceCapability Name="backgroundSpatialPerception"/>
```

- Querying for SensorDeviceConsent interface that implements the consent checks in Research Mode API.

```
hr = m_pSensorDevice->QueryInterface(IID_PPV_ARGS(&m_pSensorDeviceConsent));
```

- Before streams can be opened (OpenStream), consent must have been granted. Consent results are returned with a callback. One way to synchronize consent response with callers of the API is with an event set on the consent callback.

```
ResearchModeSensorConsent camAccessCheck;
HANDLE camConsentGiven;

camConsenGiven = CreateEvent(nullptr, true, false, nullptr);
```

- Register for camera and/or IMU consent callback from the main UI thread of the app.

```
hr = m_pSensorDeviceConsent->RequestCamAccessAsync(CamAccessOnComplete);
```

- Define the callback functions where the user consent will be captured, and an event created for this action is set.

```
void CamAccessOnComplete(ResearchModeConsent consent)
{
    camAccessCheck = consent;
    SetEvent(camConsentGiven);
}
```

- If a worker thread is used, wait on the callback and look for the consent provided by the user and proceed forward.

```
void
CameraUpdateThread(SlateCameraRenderer* pSlateCameraRenderer, HANDLE camConsentGiven, Res
earchModeSensorConsent *camAccessConsent)
{
    HRESULT hr = S_OK;
    DWORD waitResult = WaitForSingleObject(camConsentGiven, INFINITE);

   // wait for the event to be set and check for the consent provided by the user.

    if (waitResult == WAIT_OBJECT_0)
    {
        switch (*camAccessConsent)
        {
        case ResearchModeSensorConsent::Allowed:
            OutputDebugString(L"Access is granted");
            break;
        case ResearchModeSensorConsent::DeniedBySystem:
            OutputDebugString(L"Access is denied by the system");
            hr = E_ACCESSDENIED;
            break;
        case ResearchModeSensorConsent::DeniedByUser:
            OutputDebugString(L"Access is denied by the user");
            hr = E_ACCESSDENIED;
            break;
        case ResearchModeSensorConsent::NotDeclaredByApp:
            OutputDebugString(L"Capability is not declared in the app manifest");
            hr = E_ACCESSDENIED;
            break;
        case ResearchModeSensorConsent::UserPromptRequired:
            OutputDebugString(L"Capability user prompt required");
            hr = E_ACCESSDENIED;
            break;
```

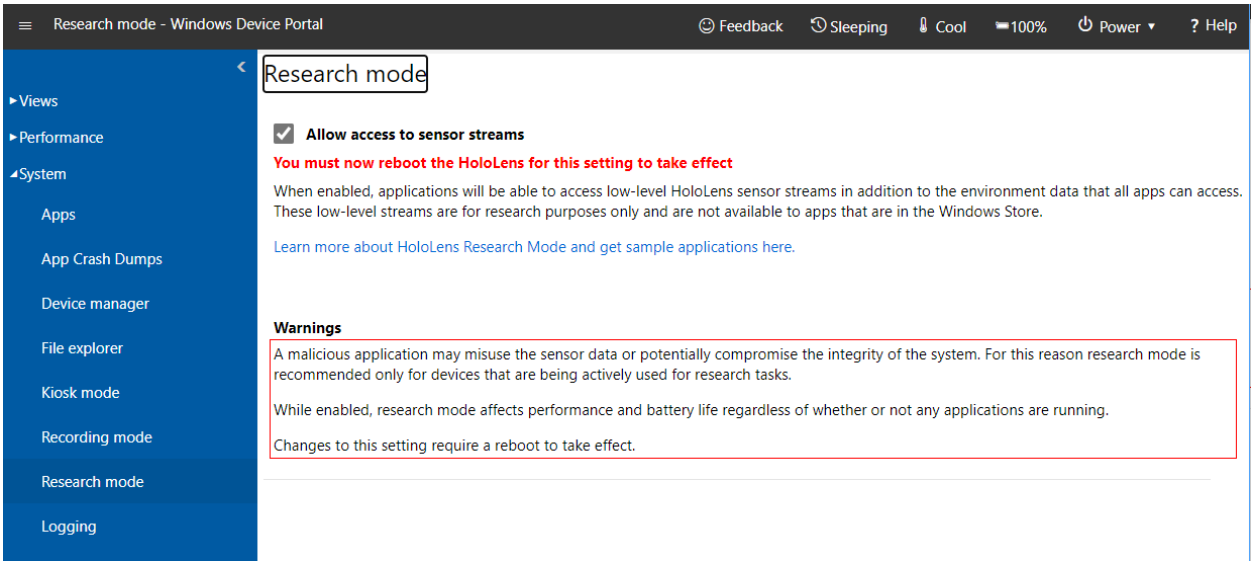
```
        default:
            OutputDebugString(L"Access is denied by the system");
            hr = E_ACCESSDENIED;
            break;
        }
    }
    else
    {
        hr = E_UNEXPECTED;
    }

    if (SUCCEEDED(hr))
    {
         hr = pSlateCameraRenderer->m_pRMCameraSensor->OpenStream();
    }
```

For testing if your application correctly implemented the checks, make sure to verify that the prompts appear for camera and/or IMUs before the app output appears. Also, the prompts appear only on the first use of the application for each user**.** To revoke the access, change the following in Settings:

- go to Settings->Privacy-> Camera→Application and turn the access off for cameras
- go to Settings->Privacy→User Movements→Application and turn the access off for IMUs.

Setup

- Windows 10
    - Desktop Windows 10
    - Insider builds HoloLens 2
      (build 19041.1364.VB_RELEASE_SVC_SYDNEY.200807-1600 or newer)
- Visual Studio requirements
    - Download free copy of Visual Studio 2017 or 2019
    - Install C++, UWP, Windows SDK components
- Turn on Developer mode:
  On your HoloLens 2 device, go to Settings > Update & security > For developers and Turn on Developer mode
- Turn on device portal:
  On your HoloLens 2 device, go to Settings > Update & security > For developers and Turn on device portal
- Enable research mode:
  Plug your HoloLens into your PC, open a browser, in the address bar go to http://127.0.0.1:10080:
    - Hit “Request PIN”. A PIN will show in your HoloLens. Enter that PIN into the browser dialog,
    - Setup a username and password to enter device portal with that HoloLens,
    - Enter the username and password in the browser dialog,
    - From device portal, go to System > Research mode and Check “Allow access to sensor streams”



Required Manifest entries

See example at:
https://github.com/microsoft/HoloLens2ForCV/blob/main/Samples/SensorVisualization/SensorVisualization/Package.appxmanifest

```
  <Capabilities>
    <Capability Name="internetClient" />
    <uap:Capability Name="documentsLibrary" />
    <rescap:Capability Name="perceptionSensorsExperimental" />
    <DeviceCapability Name="webcam" />
    <DeviceCapability Name="wifiControl" />
    <DeviceCapability Name="backgroundSpatialPerception" />
  </Capabilities>
```

API Reference

Device Interfaces

```
DECLARE_INTERFACE_IID_(IResearchModeSensorDevice, IUnknown, "65E8CC3C-3A03-4006-AE0D-
34E1150058CC")
{
    STDMETHOD(DisableEyeSelection()) = 0;
    STDMETHOD(EnableEyeSelection()) = 0;

    STDMETHOD(GetSensorCount(
        _Out_ size_t *pOutCount)) = 0;
    STDMETHOD(GetSensorDescriptors(
        _Out_writes_(sensorCount) ResearchModeSensorDescriptor *pSensorDescriptorData,
        size_t sensorCount,
        _Out_ size_t *pOutCount)) = 0;
    STDMETHOD(GetSensor(
        ResearchModeSensorType sensorType,
        _Outptr_result_nullonfailure_ IResearchModeSensor **ppSensor)) = 0;
```

```
};

DECLARE_INTERFACE_IID_(IResearchModeSensorDevicePerception, IUnknown, "C1678F4B-ECB4-
47A8-B6FA-97DBF4417DB2")
{
    STDMETHOD(GetRigNodeId(
        _Outptr_ GUID *pRigNodeId)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeSensorDeviceConsent, IUnknown, "EAB9D672-9A88-4E43-
8A69-9BA8f23A4C76")
{
    STDMETHOD_(HRESULT, RequestCamAccessAsync)(void
        (*camCallback)(ResearchModeSensorConsent))= 0;
    STDMETHOD_(HRESULT, RequestIMUAccessAsync)(void
        (*imuCallback)(ResearchModeSensorConsent)) = 0;
};
```

Sensor Interfaces

```
DECLARE_INTERFACE_IID_(IResearchModeSensor, IUnknown, "4D4D1D4B-9FDD-4001-BA1E-
F8FAB1DA14D0")
{
    STDMETHOD(OpenStream()) = 0;
    STDMETHOD(CloseStream()) = 0;
    STDMETHOD_(LPCWSTR, GetFriendlyName)() = 0;
    STDMETHOD_(ResearchModeSensorType, GetSensorType)() = 0;

    STDMETHOD(GetSampleBufferSize(
        _Out_ size_t *pSampleBufferSize)) = 0;
    STDMETHOD(GetNextBuffer(
        _Outptr_result_nullonfailure_ IResearchModeSensorFrame **ppSensorFrame)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeCameraSensor, IUnknown, "3BDB4977-960B-4F5D-8CA3-
D21E68F26E76")
{
    STDMETHOD(MapImagePointToCameraUnitPlane(
        float (&uv) [2],
        float (&xy) [2])) = 0;
    STDMETHOD(MapCameraSpaceToImagePoint(
        float(&xy)[2],
        float(&uv)[2])) = 0;
    STDMETHOD(GetCameraExtrinsicsMatrix(DirectX::XMFLOAT4X4 *pCameraViewMatrix)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeAccelSensor, IUnknown, "627A7FAA-55EA-4951-B370-
26186395AAB5")
{
    STDMETHOD(GetExtrinsicsMatrix(DirectX::XMFLOAT4X4 *pAccel)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeGyroSensor, IUnknown, "E6E8B36F-E6E7-494C-B4A8-
7CFA2561BEE7")
{
    STDMETHOD(GetExtrinsicsMatrix(DirectX::XMFLOAT4X4 *pGyro)) = 0;
};
```

Sensor Frames

```
DECLARE_INTERFACE_IID_(IResearchModeSensorVLCFrame, IUnknown, "5C693123-3851-4FDC-A2D9-
51C68AF53976")
{
    STDMETHOD(GetBuffer(
        _Outptr_ const BYTE **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetGain(
        _Out_ UINT32 *pGain)) = 0;
    STDMETHOD(GetExposure(
        _Out_ UINT64 *pExposure)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeSensorDepthFrame, IUnknown, "35167E38-E020-43D9-898E-
6CB917AD86D3")
{
    STDMETHOD(GetBuffer(
        _Outptr_ const UINT16 **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetAbDepthBuffer(
        _Outptr_ const UINT16 **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
    STDMETHOD(GetSigmaBuffer(
        _Outptr_ const BYTE **ppBytes,
        _Out_ size_t *pBufferOutLength)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeAccelFrame, IUnknown, "42AA75F8-E3FE-4C25-88C6-
F2ECE1E8A2C5")
{
    STDMETHOD(GetCalibratedAccelaration(
        _Out_ DirectX::XMFLOAT3 *pAccel)) = 0;
    STDMETHOD(GetCalibratedAccelarationSamples(
        _Outptr_ const AccelDataStruct **ppAccelBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeGyroFrame, IUnknown, "4C0C5EE7-CBB8-4A15-A81F-
943785F524A6")
{
    STDMETHOD(GetCalibratedGyro(
        _Out_ DirectX::XMFLOAT3 *pGyro)) = 0;
    STDMETHOD(GetCalibratedGyroSamples(
        _Outptr_ const GyroDataStruct **ppAccelBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};

DECLARE_INTERFACE_IID_(IResearchModeMagFrame, IUnknown, "2376C9D2-7F3D-456E-A39E-
3B7730DDA9E5")
{
    STDMETHOD(GetMagnetometer(
        _Out_ DirectX::XMFLOAT3 *pMag)) = 0;
    STDMETHOD(GetMagnetometerSamples(
        _Outptr_ const MagDataStruct **ppMagBuffer,
        _Out_ size_t *pBufferOutLength)) = 0;
};
```

Consent Interfaces

```
DECLARE_INTERFACE_IID_(IResearchModeSensorDeviceConsent, IUnknown, "EAB9D672-9A88-4E43-
8A69-9BA8f23A4C76")
{
    STDMETHOD_(HRESULT, RequestCamAccessAsync)(void
        (*camCallback)(ResearchModeSensorConsent))= 0;
    STDMETHOD_(HRESULT, RequestIMUAccessAsync)(void
        (*imuCallback)(ResearchModeSensorConsent)) = 0;
};
```