Contents

[Technical Architecture and Design of Device Hub 2](#_Toc212212185)

[Device Hub Architecture and Design 2](#_Toc212212186)

[Device Driver Design 8](#_Toc212212187)

[Notes on the Device Hub Dome Synchronisation Algorithm 9](#_Toc212212188)

[Introduction 9](#_Toc212212189)

[Notation 9](#_Toc212212190)

[Dome centre 9](#_Toc212212191)

[Coordinates and Conventions 10](#_Toc212212192)

[X, Y, Z Coordinate System 10](#_Toc212212193)

[Roll and Pitch Angle Coordinate System 10](#_Toc212212194)

[Roll Angle 11](#_Toc212212195)

[Pitch Angle 11](#_Toc212212196)

# Technical Architecture and Design of Device Hub

## Device Hub Architecture and Design

The Device Hub application is written in C# using Visual Studio 2019 Community Edition. The ASCOM templates were used to generate the starter code for the local server and for each of the exposed drivers. The structure of the code is a bit different from other local servers due to the choice of Windows Presentation (WPF) for the user interface. WPF applications do not expose a Main entry point as Windows Forms applications do. The application’s entry point is hidden in an App object. The App object is defined by the application definition in App.xaml and App.xaml.cs, its code-behind class. The code-behind class overrides the OnStartup method to start up the local server.

In addition to using WPF, the application is organized using the Model-View-ViewModel (MVVM) architecture. This design allows for separation of the U/I and business objects. The business object classes have no knowledge of or dependence on the user interface. This allows the business object classes (view models) to be completely tested from Unit Test projects without any user interface.

Each Window or UserControl (the main U/I components) is a view. For most views, the code-behind is completely empty. Instead, marshalling data between the view and the viewmodel is accomplished by databinding. Also, instead of using event handlers to perform actions as requested by the U/I, the viewmodels implement ICommand methods that are hooked to the various controls via WPF data binding. Below is a simple example:



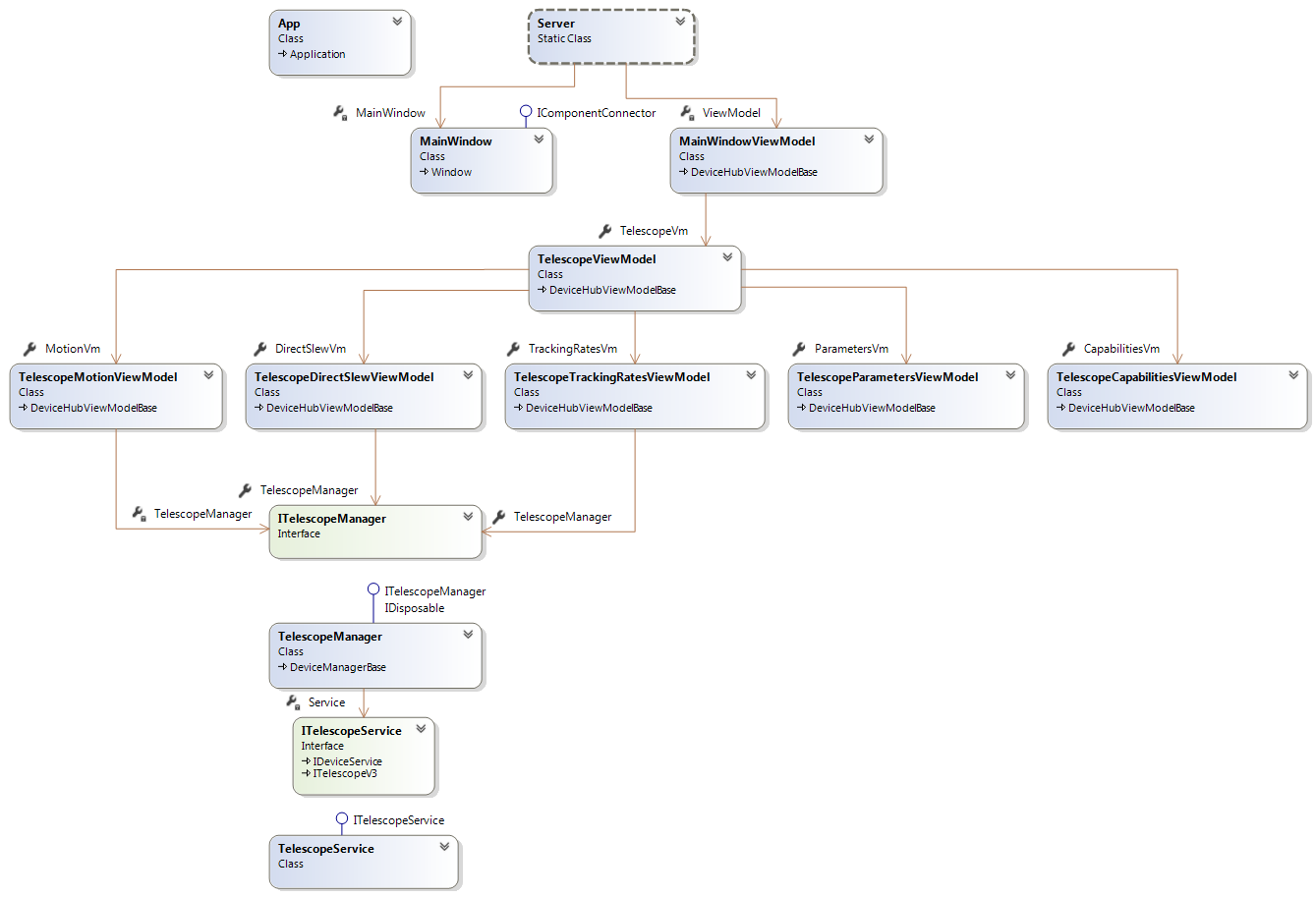
Above is the XAML description of an example button object. The Command attribute is bound to the viewmodel’s StopMotionCommand property. That property is declared as follows:



The RelayCommand class implements the ICommand interface. It resides in the Business Object Classes folder of the DeviceHub project. The class supports definition of an Action object, StopMotion in the example above, and a “can execute” predicate. Although the above example does not define the predicate, most instances of RelayCommand do. These predicates are used, automatically by the WPF data binding logic, to enable or disable the bound control based on the predicate’s return value.

Another consequence of designing for testability is the use of services to provide access to logic that is external to the application classes. Several services have been incorporated into the architecture. Each of the ASCOM device objects is wrapped in a service. In addition, there are services for displaying the activity log, for loading and saving the application settings, and for displaying Windows message boxes and other modal dialogs. These services are injected during the application startup, but mock versions of each service that implement the same interfaces are injected during unit testing. This allows for the creation of unit test classes and methods that allow for thorough, automated testing of each layer of software between the views and the services. This includes the view models and the device managers.

Below is a simple class diagram showing the application skeleton of the classes involved in telescope control:



Unfortunately, the Visual Studio class diagram tool does not provide the ability to define the dependency between the App class and the static Server class. The best that I can do is to simply show the App class. The dependency does exist, however. The App’s instance overrides the base class’ OnStartup method to invoke the Server class Startup method and pass it any command-line parameters.

The Server.Startup method instantiates the MainWindow object and the MainWindowViewModel object and connects them, via the MainWindow’s DataContext property. This allows properties and commands in the viewmodel to be accessible from the view.

The remainder of the startup method is very similar to that which is created by from the LocalServer template.

One generated class that is substantially different is the GarbageCollection class. GarbageCollection is instantiated from the Server class, during startup, and cancelled when the main view is closed. The Device Hub uses the System.Threading.Tasks.Task class to run the garbage collector on a worker thread. At the time of this writing, the garbage collection interval is coded at 10 seconds. However, due to the capabilities of Task objects and their CancellationTokens, the garbage collector can be immediately cancelled without waiting for the garbage collector’s sleep timer to expire. This means that disposed objects are cleaned up more quickly during shutdown.

The MainWindowViewModel instantiates the TelescopeViewModel which in turn instantiates the viewmodels for each of the Telescope control tabs (Motion, DirectSlew, TrackingRates, Parameters, and Capabilities). The TelescopeView.xaml definition defines the view to be used for displaying those viewmodels.

The Resources element of the TelescopeView (which is a UserControl) declares a DateTemplate which tells the WPF rendering engine which view to use for displaying that viewmodel.



The viewmodel instance is bound to the Content property of a ContentControl which is displayed in the appropriate TabItem.



Some of the telescope viewmodels only provide information from the telescope for display others need to affect changes to the state of the telescope. For example, the TelescopeCapabilitiesViewModel holds the values of the Capabilities properties that are read from the device when it is initially connected. The values of these properties are displayed by the TelescopeCapabilitiesView. This viewmodel needs no further interaction with the telescope.

On the other hand, the TelescopeMotionViewModel needs to interact with the telescope to display the current telescope state AND to change property values and call methods on the telescope object that affect the telescope’s state. This interaction is supported by the TelescopeManager. There are also DomeManager and a FocuserManager classes that support access from the viewmodels to those devices. These objects are singleton objects that are created in the MainWindowViewModel and injected into the created viewmodels as constructor arguments. During unit testing of a viewmodel a mock version of the device manager is injected into the VM being tested.

The design of the device managers makes use of the capability of C# to split a class definition over multiple files. So, there is a TelescopeManager.cs file as well as a TelescopeManagerAccess.cs file. Both files define properties and methods for the TelescopeManager class. Where necessary, properties and methods in TelescopeManager.cs use calls in TelescopeManagerAccess.cs to communicate with the device service (more in the next paragraph), rather than communicate directly with the device service. The methods in TelescopeManagerAccess perform checks to ensure that we are connected to the device and to forward the details about the access to the activity logger. The properties and methods in TelescopeManagerAccess are also used by the exposed Telescope device driver to access the served Telescope driver.

The device managers are the most complex classes in the application and their testability is a requirement. They communicate with the ASCOM device objects through a service which wraps the ASCOM calls to the driver. Each service has an abstract definition and two concrete definitions. One concrete definition is for the normal operation of the application and the other is a mock definition for unit testing. For the most part the concrete TelescopeService is a simple wrapper which passes through any property values or method calls to the associated ASCOM property or method. The concrete MockTelescopeService, however has the capability to allow the Unit Testing infrastructure to initialize the service to support each of the individual tests.

The above discussion presents some detail about the responsibilities of the classes that are included in the previous class diagram. The organization of classes for the Dome and Focuser devices are organized in a similar manner.

The above discussion, while fairly detailed, is incomplete. It does not mention how information is propagated between the device managers which generate the data by reading it from the device and the VMs that need to provide the data for display. This job is the responsibility of the Messaging subsystem.

Messages are like events but are more powerful since any class can subscribe to a message and any class can generate a message. Rather than design a Messaging system from scratch, I decided to use one that is part of an open-source project where the author has generously made the source code available.

One thing that I discovered when I started writing WPF MVVM applications is that there are a lot of common needs that they all share. One solution to making an aspiring MVVM developer more productive is to utilize a 3rd party library. One such library is the MVVM Light Toolkit from Laurent Bugnion. This tookit is available as a NuGet package that can be added to a Visual Studio project. The author also makes it available for download from a GitHub repository. After reading the license under which the author makes the code available, I extracted only the classes of the messaging component and incorporated them into an assembly, MvvmMessenger, which is part of the DeviceHub solution. This decouples the Device Hub from dependence on the MVVM Light Tookit and relieves us from the burden of distributing the entire toolkit with the Device Hub.

The TelescopeManager begins communicating with the Telescope (via the TelescopeService) when it is connected. The Capabilities and Parameters information are immediately read from the device and forwarded by means of messages. The following code fragments show how the capabilities properties are read from the device and forwarded to viewmodels that either display the values or use the values to make decisions about how to interact with the device.



This method fragment is from the TelescopeManager. It is called on a worker thread immediately upon successfully connecting with the device. It creates an instance of the TelescopeCapabilites class and initializes it by making calls to the ITelescopeService object to read the capabilities property values from the ASCOM Telescope object. Further down in the code it sends a message where the message payload is a clone of the Capabilities object.

One of the obvious recipients of this message will be the TelescopeCapabilitiesViewModel. Here is the messaging-related code from the VM:



The VM’s constructor registers a subscriber for the message and specifies the message handler method, UpdateCapabilities, in this case. So, when the Telescope Manager sends the message, UpdateCapabilities is called to receive it. The Capabilities object is extracted from the payload and passed to SetCapabilities which updates the class’s Capabilities property on the main User Interface thread.

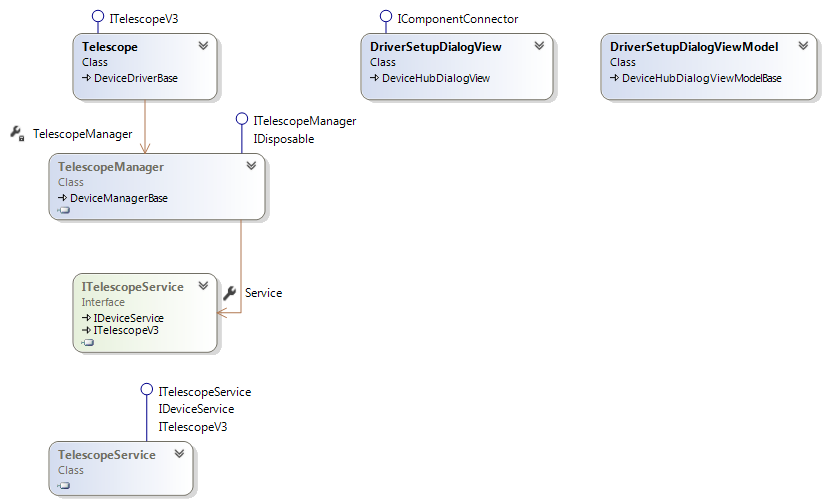
WPF has the same limitation as Windows Forms in that controls can only be updated by code that is running on the same thread that created them. Therefore, views and viewmodels are always created on the main thread and viewmodel properties that are bound to controls in the view can only be updated by code that is running on that thread.

One of the abilities of the Task and TaskFactory classes is to execute the thread method on the thread that is given by the synchronization context. The property Globals.UISyncContext holds a synchronization context from the U/I thread and should be specified for any task that will be updating properties that are bound to the U/I.

All the VM classes derive from a base class that implements the IDisposable interface. The base class’s Dispose method calls an empty, virtual method called DoDispose. Each VM can override DoDispose to allow cleanup to occur. In this case, the DoDispose method simply unsubscribes from any messages that the class previously subscribed to. If it had created any child VM’s they would be Disposed and nulled at this time.

## Device Driver Design

As has been previously discussed, the Device Hub provides a telescope driver, a dome driver, and a focuser driver that other applications can connect to. These drivers sit atop and use the facilities of the Device Hub application. Below is a class diagram of the DeviceHub Telescope driver. The dome and focuser drivers are organized in a similar manner.



The Telescope driver implements the ITelescopeV3 interface and derives from DeviceDriverBase which derives from ReferenceCountedObject. DeviceDriverBase provides convenience methods that are used for data and state validation by the device drivers. It also provides common methods for logging.

The driver communicates with the physical telescope through the same TelescopeManager and TelescopeService objects that the DeviceHub application uses.

The DriverSetupDialogView and DriverSetupDialogViewModel classes are instantiated and used by the Telescope driver’s SetupDialog method. This allows an application that is using the driver the ability to choose which driver will be used for the served telescope and to configure that device.

# Notes on the Dome Synchronisation Algorithm

## Introduction

Patrick Wallce of [TPoint Consulting](https://www.tpointsw.uk/) authored an authoritative [paper](https://www.tpointsw.uk/edome.pdf) in June 2017 on how to calculate the required azimuth and altitude of a dome slit in order for an equatorial telescope to see a desired object. A backup copy of the paper is retained in the ASCOM repository and can be downloaded through this [link](https://github.com/ASCOMInitiative/ASCOMPlatform/raw/main/Drivers%20and%20Simulators/ASCOM%20Device%20Hub/Documents/WallaceDomePointingAlgorithm.pdf) if necessary.

The algorithm described in the paper is now used as the synchronisation engine in Device Hub and its C# code is a translation of C code kindly donated by Patrick.

During implementation it became clear that Patrick’s paper used one set of notation, while his implementation code used another although both are equivalent. These notes provide a bridge between the paper’s notation and comments and variable the names used in the C / C# implementations. In addition, this paper documents helpful insights communicated privately by Patrick.

## Implementation Note

This discussion and the implementation in Device Hub are intentionally constrained to designs where the two mechanical rotational axes intersect in order to reduce the possibility of confusion when measuring and entering coordinates.

## Nomenclature Relationships

|  |  |  |
| --- | --- | --- |
| **Element** | **Wallace Paper Name** | **Computer Code Name** |
| The separation between the mechanical axes at their closest approach.  *This is zero in most mount designs.* | P | yt |
| The distance along the declination axis from the mechanical axis intersection and the point closest to the telescope optical axis.  *This is usually zero for fork mounts and always non-zero for German equatorial mounts.* | q | xt |
| The distance between the declination and optical axes.  *This is frequently zero but can be non- zero when an offset bar is used to mount multiple telescopes side by side.* | r | yo |
| RA or altitude axis | RA or altitude axis | Roll or longitude axis |
| Declination or elevation axis | Declination or elevation axis | Pitch or latitude axis |

## Use in Device Hub

In line with the implementation note above, Device hub only exposes the **q/xt** and **r/yo** parameters, which are by far the most commonly used. The **p/yt** parameter is present in the computational algorithm but is set to zero when computations are made.

## Dome centre

The primary reference point is the geometric centre of the dome sphere. This is the centre of the sphere from which the dome is made. E.g. if the dome is less than a hemisphere, the reference point is not in the plane of the bottom of the dome but lower than this at the centre of the sphere of which the dome section is a part.

If the dome is more than a hemisphere the centre will be above the plane of the bottom of the dome as shown below.

## 

## Coordinates and Conventions

### X, Y, Z Coordinate System

The offset of the telescope mechanical axis intersection from the dome centre is represented as a set of x, y, z coordinates:

* X 🡺 East – west (east positive)
* Y 🡺 North – south (north positive)
* Z 🡺 Up – down (up positive)

In equatorial mounts PHI is the site latitude but for Alt/Az mounts PHI is to 90 degrees regardless of the site latitude.

The code requires the mechanical roll angle which is calculated from the telescope hour angle (local sidereal time minus right ascension) and declination for equatorial mounts azimuth for alt/az mounts.

### Roll and Pitch Angle Coordinate System

Both roll and pitch axis angles are calculated in the range -180.0 to +180.0 degrees (-π to +π radians)

The coordinate system is “right-handed” with **mechanical** roll angle increasing anticlockwise as viewed from space looking down on the north pole (equatorial) or zenith (alt/az). This means that:

* Equatorial mounts: **Mechanical** hour angle is zero for telescope targets due south and positive for telescope targets to the east of south (the more east the more positive).
* Alt/Az mounts: **Mechanical** azimuth is zero to the south and positive for telescope targets to the east of south.

### Roll Angle

In Patrick’s models zero roll angle (hour angle / azimuth) is defined as being when a telescope in the northern hemisphere is pointing due SOUTH in the “Normal” pointing state. So for equatorial mounts, the roll angle in both hemispheres is given by:

* Normal pointing state (PierEast) 🡺 minus hour angle
* Through the pole pointing state (PierWest) 🡺 minus (12.0 minus hour angle)

For alt/az mounts it is: 180.0 minus telescope azimuth in both pointing states

### Pitch Angle

The pitch angle (declination / altitude) is more complex and varies with hemisphere. Zero pitch angle aligns with declination for equatorial mounts and altitude for alt/az mounts.

In the northern hemisphere for equatorial mounts the pitch angle is:

* PierEast 🡺 declination
* PierWest 🡺 180.0 minus declination

e.g. for declination = 20 degrees:

* PierEast = 20 degrees
* PierWest =180 - 20 = 160 degrees

In the southern hemisphere for equatorial mounts the pitch angle is:

* PierEast 🡺 declination
* PierWest 🡺 minus 180 minus declination
  + note that declination is negative in the southern hemisphere

e.g. for declination = minus 20 degrees:

* PierEast = -20 degrees
* PierWest =-180 - (-20) = -180 + 20 = -160 degrees

For alt/az mounts the pitch angle equals telescope altitude.

Offsets