# Guidelines for Driver and Simulator Developers

The ASCOM development servers are available for any developer who wishes to contribute code related to The ASCOM Initiative. The Subversion source code control system makes it easy to collaborate with a team of developers, carry out peer reviews and give other ASCOM developers visibility of your code. It’ll also make sure you code is backed up, both on other developer’s working copies and in the daily tape backup that we run.

## Where to Put Your Code

As a driver/simulator developer, you’ll have read-only access to the entire ASCOM source tree.

You’ll have read/write access to any projects that you’re actively participating in. This way, you have plenty of reference code to look at and you can freely commit changes to your own part of the tree without fear of affecting others.

When you create a new project, we recommend the following naming convention:

* /*Vendor*/*DeviceName*

Your solution file goes in the /DeviceName folder, with sub-folders under that for each project. If you are working on a compound driver based on the LocalServer (singleton) template, then one of your projects will be named *DeviceName*Server, and you’ll have a series of projects named *DeviceType*Driver for each of the devices supported. Example: say you’re working on a Meade LX-200 Classic driver, with support for both the telescope drive and the built-in focuser. You need to implement both the Telescope interface and the Focuser interface and you’ll expose those interfaces by wrapping them in a LocalServer. This will be your project structure:

/Meade/LX200/LX200.sln  
 /LX200 Server/  
 /Telescope Driver/   
 /Focuser Driver/

## Assembly Naming and Versioning

We recommend naming your projects (and the assemblies they produce) along the following lines:

* **ASCOM**.*DeviceName*.*DeviceClass* (example: **ASCOM.LX200.Telescope**)

We recommend that you give your assemblies the same major and minor version as the ASCOM platform that they were compiled against. So if you’re using ASCOM Platform 5.1, then your assembly should have Major Version = 5 and Minor Version = 1. We suggest letting Visual Studio manage the Revision and Build numbers automatically, by placing an asterisk (\*) in the Build number and leaving the Revision number blank.

## Referencing Platform Components (Platform Dependencies)

When developing drivers, we recommend that you do not rely on any assemblies being installed in the Global Assembly Cache. Your driver should be able to build and run without any version of the ASCOM Platform installed. This will enable our build server to build your solution and help keep the codebase consistent and in good working order.

To facilitate this, we recommend checking out the entire ASCOM source tree when working on a driver. Use Project References to reference the ASCOM component projects directly.

## Installing Your Driver

### Checking the Installed Platform Version

This is an area where many driver developers come unstuck – please take care! Naturally, when you install your driver, you want to check that the minimum required version of the ASCOM Platform is installed on the user’s system. The helper components in the ASCOM.Utilities namespace give you an easy way to do this with the PlatformVersion() method, which returns a string containing the platform major and minor version in “major.minor” format. Please do not be tempted to convert this string to a number. If your program is running in a country that does not use the point as the decimal separator, your conversion will fail. If you must convert it to a number, be sure to use the Invariant Culture when making the conversion.

The preferred way to check for a minimum required platform version is to avoid internationalization issues completely by using:  
IsMinimumRequiredVersion(RequiredMajorVersion, RequiredMinorVersion)  
This simply returns ‘true’ or ‘false’.

# Development Servers

There are several servers of interest to ASCOM developers. In the table below, the first two in the list will be essential for you to become familiar with. The others are useful, but you can get by just fine without ever using them.

|  |  |  |
| --- | --- | --- |
| Product | Purpose | URL |
| VisualSVN Server | Subversion source code control | <http://svn.tigranetworks.co.uk/svn/ASCOM> |
| Atlassian Jira | Issue tracking and project management | <http://teamserver.tigranetworks.co.uk:8010> |
| JetBrains TeamCity | Build & Continuous Integration Server | <http://teamserver.tigranetworks.co.uk> |
| Atlassian Fisheye | Source code metrics | <http://teamserver.tigranetworks.co.uk:8060> |
| Atlassian Crucible | Code Reviews | <http://teamserver.tigranetworks.co.uk:8010> |
|  |  |  |

* **VisualSVN**: You’ll use the VisualSVN server on a daily basis as you develop and commit your code.
* **Jira**: This is where you (and your customers/users) create and monitor bugs and improvement requests, project milestones and releases.
* **TeamCity**: The TeamCity build server builds the entire platform each time any new code is committed and checks that everything builds OK. The build happens in an isolated environment. It can also perform unit tests. The standard practice is that you should only commit code that builds with no errors. If your commit breaks the build, then the build server will designate you as the ‘responsible person’ and you’ll be expected to fix things as soon as possible.
* **Fisheye**: Generates metrics, statistics and charts based on the code committed to the source code control server and events that happen in the other servers. Some of this information feeds back into the other servers (e.g. Jira) and helps to provide a comprehensive view of all developer activities. One of the most useful outputs from Fisheye is the ‘activity stream’ which lists everything happening to the code base. You can subscribe to the RSS feed for this activity stream and keep up to date with what’s happening.
* **Crucible**: Supports code reviews. When new code is committed, a formal review can be created. Developers can examine the code and comment on it, perhaps even raising defect reports. In this way, the ‘many eyes’ approach, code quality should be improved and defects detected early while they are still easy to fix. Use of Crucible is optional but highly recommended.

# Ensuring 32-bit and 64-bit Compatibility

From Platform 5.5 onwards, all of the ASCOM core components work on both 32-bit and 64-bit systems. With a little care, your drivers will be able to work on both types of system too.

They key is to set your project’s target architecture to “Any CPU”. The Common Language Infrastructure uses a ‘JIT’ (Just In Time) compiler that actually generates executable code at the moment it is needed, when it is loaded on the end user’s system. So, assemblies that were targeted for “Any CPU” have a chameleon-like ability to change from 64-bit to 320bit and back again, on demand, every time they are used. The JIT compiler is aware of both the CPU architecture and the host process that is loading the assembly – if either is 32-bit then the assembly will be JIT-compiled in 32-bit mode. If both the CPU and the host process are 64-bit, then the JIT compiler works in 64-bit mode. So, by targeting “Any CPU” you get automatic compatibility with both 64-bit and 32-bit systems and programs.

### Why Not Just Use 32-Bit (x86)?

This seems like a smart move, because all systems can run 32-bit programs. However, this is not the best option to choose. While 32-bit programs may call into 64-bit processes, the reverse is not true. A 64-bit process may not call into a 32-bit process. So, the first time your code is loaded by a 64-bit process running on a 64-bit system, it will fail. You can easily demonstrate this by loading your driver from VBScript. The default scripting engine on 64-bit systems runs in 64-bit mode, so your driver will fail when you try to script it. Therefore, do not use ‘x86’, use ‘Any CPU’.

### Debugging on 64-bit Systems

Visual Studio is a 32-bit application. This fact presents some difficulties when developing 64-bit targets. X64 programs can be successfully debugged in Visual Studio, but there will be problems related to COM registration and ASCOM registration. For this reason, it is recommended (by Microsoft) to develop your driver in ‘x86’ mode. Clearly, this advice conflicts with the above advice to use ‘Any CPU’. The solution is to use a number of different build configurations for different situations.

By default, Visual Studio projects and solutions come with two build configurations, called Debug and Release. These names are fairly self-explanatory. However, we can use these configurations to our advantage. We can make the Debug configuration build in x86 mode and the Release configuration build in ‘Any CPU’ mode. Additionally, the configuration manager allows us to control which solution items are built, so we may choose to omit building any non-essential projects during debugging (setup projects are a prime candidate for omission).

 

Two build configurations showing builds for ‘Debug/x86’ and ‘Release/x86’. Note the two setup projects, one for x86 and one for x64. The Visual Studio Configuration Manager can be used to configure a number of build configurations for different situations.

# Setup and Deployment

ASCOM Drivers should always be supplied with an integrated setup application. Drivers which do not do this do not meet the minimum requirements for ASCOM logo usage and may not claim ASCOM compatibility.

## Installer Technology

There are a number of options available to driver developers for creating setup packages.

* Windows Installer
* Inno Setup
* Other third party products

### Windows Installer

Windows Installer is Microsoft’s own technology and offers the best future-proofing and flexibility. This type of setup is easily integrated into Visual Studio and therefore forms a normal part of the build process. The trade-off is that Windows Installer is deceptively tricky to use and can be problematic to troubleshoot when things don’t go as planned.

There are two main approaches to creating Windows Installer packages:

* Visual Studio Professional has integrated tools for creating Windows Installer setup projects, these projects are therefore fully integrated with visual studio’s build system.
* WiX (Windows Installer XML) – WiX is a free, open source product that enables Windows Installer packages to be created based on declarative XML files. These XML documents can be easily managed in source code control systems and the process can be integrated into Visual Studio’s build system. WiX can be used with Express editions of Visual Studio. WiX is the product used in-house by many Microsoft products, including Microsoft Office.

If Windows Installer being used, it is recommended that you create two setup projects, one for x86 and one for x64, per Microsoft guidelines ([How to: Create a Windows Installer for a 64-bit Platform](http://msdn.microsoft.com/en-us/library/cd7a85k9.aspx)).

This web page has some helpful troubleshooting tips and workarounds for known problems:  
<http://msdn.microsoft.com/en-us/library/kz0ke5xt.aspx>

### Inno Setup

Inno is a third-party script-based installer. An Inno setup script generator is provided with the ASCOM Platform, making it quick and easy to get started. Inno has the advantage of being lightweight and easy to use, but it is not easily integrated into Visual Studio and lacks some of the advanced features of Windows Installer.

### Other Tools

Other third party setup technology such as InstallShield is available commercially, but their use is outside the scope of this document.