Solartron Metrology

ORBIT®3 SOFTWARE (for the Microsoft .NET Framework) Manual

502989

1.1 DOCUMENTATION CROSS REFERENCE

502914	Orbit3 Modules manual	Details on installation and electrical requirements.
502990	Orbit3 System manual	Details on installation and electrical requirements for the Orbit Library compatible products

1.2 TRADEMARKS AND COPYRIGHTS

1.3 CONTACT INFORMATION

For updated information, troubleshooting guide and to see our full range of products, visit our website: http://www.solartronmetrology.com

2 TABLE OF CONTENTS

3 INTRODUCTION

3.1 SCOPE

The Orbit®3 Measurement System is a modular measurement system that can be put together quickly, easily and is cost effective. It allows different types of sensors to be easily mixed and integrated on a single network independent of sensor technology. In addition to linear probes and linear displacement transducers, third party sensors can easily be integrated. This, combined with programmable input and output modules for interfacing to external equipment makes the Orbit®3 Measurement System a flexible solution for measurement applications.

Typically an Orbit®3 Measurement System will consist of four elements: windows support Measurements Modules with T-Connectors, Measurement System controllers, power supplies and cables. All of which can be obtained from the same supplier, thus guaranteeing compatibility and accelerating system integration.

The Orbit®3 Measurement System also includes a range of readouts for stand alone measurement systems; these also can be used as a basic interface to a PLC.

This document defines the software protocol of the Orbit®3 Measurement System and provides information and guidance on using the OrbitLibrary.
The information is principally for users of PC systems who wish to develop software applications for use with the Orbit Measurement System.
Orbit can also be used with low level commands using the RS232IM Controller Module. These commands are detailed in the Orbit3 Low Level Command manual (503303), available on request.

This manual should be used in conjunction with the Orbit®3 System manual & the Orbit®3 Module manual.

The measurement and mechanical performance of individual products is detailed in the appropriate sections of the manuals.

Support software is supplied for all versions of Windows, from 7 onwards (both 32 & 64 bit versions).

Examples are provided for C# and C++.

3.2 NAVIGATING THIS DOCUMENT

This is a large document, which is a useful reference when writing Orbit applications. Hyperlinks are included to aid navigation.

To return to the point where you have jumped from, most pdf readers have a ‘Previous Page View’ button, alternatively use the keyboard shortcut 'ALT' + left arrow key.

3.3 TERMS AND ABBREVIATIONS

For terms associated with the Orbit3 measurement system, see the Orbit3 System manual.

3.3.1 Abbreviations

AIM	Analogue Input Module
AGM	Air Gauge Module
DP	Digital Probe (e.g. DP10)
DIOM	Digital Input Output Module
DIOM2	Digital Input Output Module V2
EIM	Encoder Input Module
LE	Linear Encoder
LT	Laser Triangulation Probe
LTA	Laser Triangulation Probe
LTH	High performance Laser Triangulation Probe
DIM	Digimatic Interface Module
WCM	Wireless Connection Module
M	(Orbit) Module
PIE	Probe Interface Electronics
USBIM	USB Interface Module
RS232IM	RS232 Interface Controller
RS485IM	RS485 Interface Controller
ETHIM	Ethernet Interface Controller This term is used throughout the document to cover the ETHIM 2.0 product and the older ETHIM product
ORBIT®	Orbit communication protocol
OSPW	Orbit Support Pack for Windows
WIM	Bluetooth Wireless Interface Controller
4K MODE	Synchronised Measurement at 3096 readings per second
2K MODE	Synchronised Measurement at 1953 readings per second
1K MODE	Synchronised Measurement at 976 readings per second
VBA	Visual Basic for applications
COM	Component Object Model
DLL	Dynamic Link Library
SC1-A	Single channel conditioner

4 SOFTWARE INTERFACING TO ORBIT

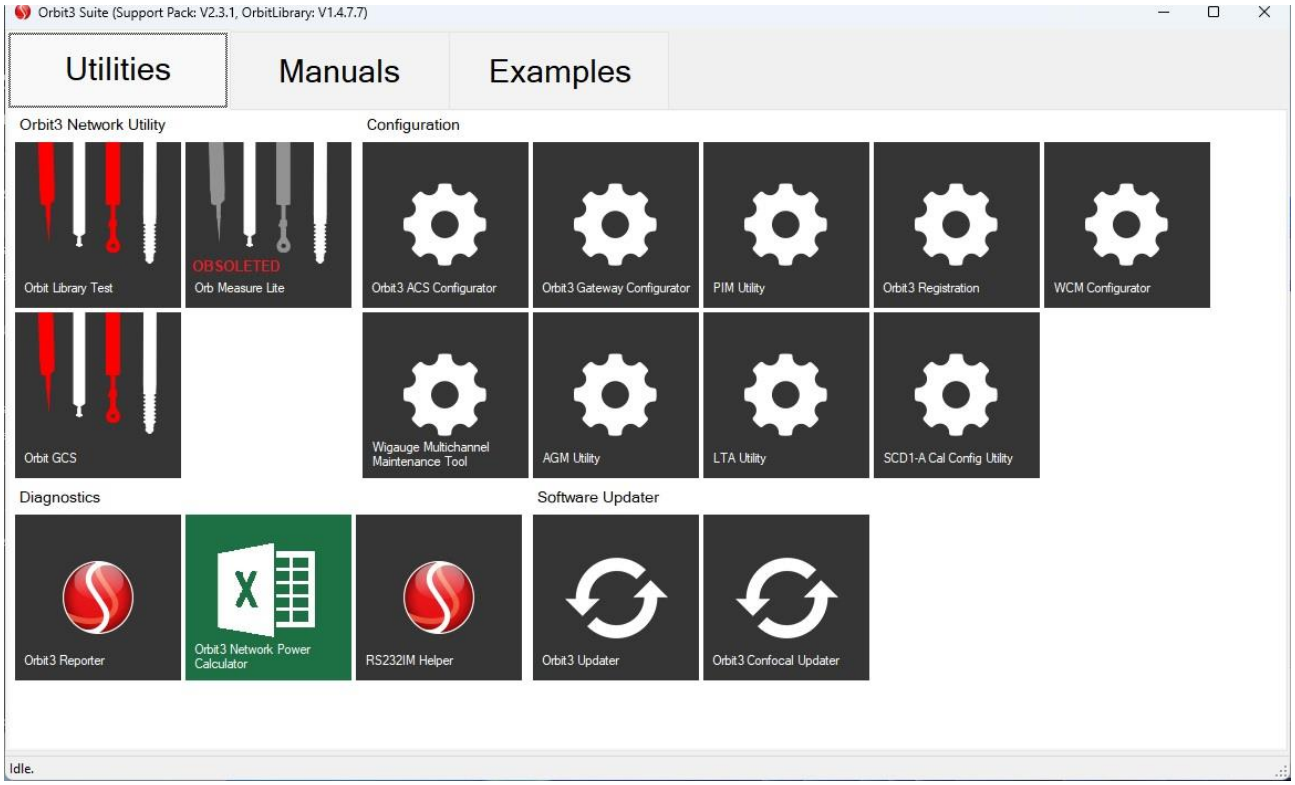
4.1 INTRODUCTION

The Orbit®3 Support Pack for Windows covers interfacing Orbit to a PC with Microsoft Windows Operating System. It contains an Install program which will installs the Orbit drivers, software libraries, applications and manuals on to your hard disc.

Included with Orbit®3 Support Pack for Windows is the Orbit library. This is the simplest and best tool to use to communicate with an Orbit Measurement System.

See the Orbit®3 System manual for installation details, including PC specification and updating Windows drivers.

After installing, a desktop shortcut to the 'Orbit3 Suite' dashboard will appear.
The dashboard looks like this:



4.2 ORBITLIBRARY

The Orbit Library is specifically designed for the Microsoft .NET Framework that is included with all Windows operating systems from Windows 7 onwards.

Using the Orbit Library greatly simplifies the development of Orbit systems since it:

- Provides a more modern object orientated software design.
- Allows the end user to avoid learning about the intricacies of the ‘low level’ Orbit interface. In particular, this:
  - Calls all the necessary functions in the correct order.
  - Handles the timing constraints of different modules and controllers.

- Handles the compatibility of modules and controllers.
- Seamlessly handles dynamic mode.
- Can be easily interfaced to Windows based software using standard high level languages. Example programs are available that illustrate this.

For these reasons, the Orbit Library is ***always*** recommended for use with new designs.

- For a simple C# implementation of the Orbit Library, see the 'Orbit3 CSharp Example' project
- For a full implementation of all the Orbit Library features, see the 'Orbit Library Test' project.

Both are installed with the Orbit Support Pack.

4.2.1 Compatibility

The OrbitLibrary is fully compatible with Windows 7, 8 and 10, both 32 bit and 64 bit versions.
Some of the older Orbit network Controllers are not compatible, see the Orbit Compatibility Roadmap section for details.

4.2.2 OrbitLibrary Code Reference

This provides extensive help on commands and syntax of the Orbit Library.
It is installed to the Orbit3 Support Pack for Windows-Manuals sub-directory.
Note. References to standard C# commands link to the Microsoft MSDN website.

4.2.3 OrbitLibrary Code UML Diagram

This provides a UML (Unified Modelling Language) drawing for the Orbit Library.
It is installed to the Orbit3 Support Pack for Windows-Manuals sub-directory.

4.3 ORBIT LIBRARY TEST

This program is designed to demonstrate the functionality of the Orbit Library. This program is a useful test program in its own right.

- This includes examples of all Orbit Library commands.
- The source code is included with full comments.

See Orbit Library Test

4.4 ORBIT GCS

- Orbit GCS is an advanced gauging tool with multiple measuring dimensions which can perform multipoint or single point measurements at very high precisions when used with Solartron's Orbit Digital Measurement Network. Probes or modules can be algebraically combined with powerful maths and control formulas to display dimensions including roundness, run out etc.
- Dimension values are displayed using bar charts and text readouts on user configurable layout screens. Values are logged to CSV files (free software registration required). Orbit GCS supports up to 150 modules per network. Input and Output via Orbit DIOM modules is supported for taking readings and displaying statuses such as logging readings and error conditions.

- **Orbit GCS is not installed with the Orbit Support Pack for Windows. The base version is available as a free download from the Solartron Website. Once installed it can be launched from the Orbit Suite.**

- Optional Upgrade Packages for the GCS can be purchased from your Solartron Agent for additional functionality including multiple Orbit Network support, storing the measurement configuration of up to 1000 parts, user editable maths formulas ,logging batches of parts in a single log file, Mastering, SPC and user diagrams.

4.5 ORBMEASURE LITE (RETIRED)

- This has been superseded by Orbit GCS which offers the functionality provided by Orb Measure Lite with many new features. The base version of Orbit GCS is a free download. Please see the Orbit GCS section.
- OrbMeasure Lite provided an 'out of the box' Orbit application, limited to 16 Orbit Modules.

4.6 ORBIT3 EXCEL® ADD-IN

The Orbit3 Excel® Add-in enables you to take readings from Orbit Modules forming an Orbit Network and place them in cells of a Microsoft® Excel® spreadsheet.
It is designed to work with standard Solartron Orbit3 Controllers (USBIM, RS232IM, ETHIM 2.0 and ETHIM).
Example spreadsheets for this add-in are included.
The add-in is available from the Solartron website.

4.7 ORBIT3 CODE EXAMPLES

To illustrate how to use OrbitLibrary software in different applications, examples are provided for C# , C++ COM and VBA (Excel COM).
They are installed as part of the Orbit3 Support Pack for Windows.
Both the C# ,and C++ COM examples will only connect to the first available Orbit Network and will ignore any others.
See Example Code - Walk through for more details of the C++ and C# examples.

4.7.1 Excel VBA COM Example

This is intended for use with Excel 97 through to Excel 2003. It may also be run on newer versions, but from Excel 2007 onwards, the .NET Framework interface is automatically used, rather than the COM.
To run the Excel VBA example, the Macro security level in Excel should be set to Medium, i.e. “allowing you to choose whether or not to run potentially unsafe macros”.

4.8 USING ORBIT WITHOUT WINDOWS

Some users require access to non Windows based computers (e.g. PLC – Programmable Logic Controllers). The USBIM MK2, RS232IM MK2 Orbit Controllers can be used to interface to this type of computer. Also, the Ethernet Controller can be used via 'sockets'.
The ‘Low Level’ Orbit protocol that should be adhered to is detailed in the Orbit3 Low Level Command manual (503303), which is available on request.
The RS232IM Helper can be used to assist in writing non Orbit Library programs.

4.9 ORBIT TROUBLESHOOTING

A useful utility for helping to diagnose software problems is the Orbit3 Reporter. The software produces a log file, which can then be sent to your supplier, along with a description of the problem itself, to aid technical support.

5 ORBIT UTILITY PROGRAMS

These programs are all available as part of the installation of the Orbit3 Support Pack for Windows.

5.1 ORBIT3 REGISTRATION

This program is used to register Orbit ETHIMs and RS232IMs. Any ETHIMs (including ETHIM 2.0) or RS232IMs **must** be registered with this program in order to work with the Orbit Library.
See Orbit3 System manual for details.

5.2 ORBIT3 REPORTER

If you are having problems with interfacing with the Orbit3 Measurement System, it is highly recommended to use the 'Orbit3 Reporter'. This program is used to find Orbit Controllers and Modules on the Orbit Measurement System. It retrieves information about the PC set-up configuration, as well as any Orbit Software, Controllers and Modules found.
Before running this program, close all other Orbit related programs.

On exiting the program, the on screen results can be logged to a text file. Choose the appropriate file name & path and click Save.
This file contains useful information about the Orbit3 Measurement System in question and is useful for troubleshooting & diagnosing problems.

5.3 ORBIT3 UPDATER

This application enables the user to update the firmware of Orbit Controllers (including MODIM), Orbit Modules, OrbitACS & PIM products. The Confocal System has a separate updater application, see Orbit3 Confocal Updater.
It may be used in conjunction with the Orbit3 Reporter to ensure that the Orbit components are all running the latest firmware.

5.4 ORBIT3 NETWORK POWER CALCULATOR

This is an Excel spreadsheet used to determine power supply calculations and considerations. Refer to the Orbit System manual for Orbit installations.

5.5 RS232IM HELPER

The RS232IM Helper application is a .NET based application written in C# (C Sharp) that can be used as an aid when writing non Orbit Library based programs.
It displays the serially transmitted and received ASCII bytes for common Orbit commands.
The ‘Low Level’ Orbit protocol that should be adhered to is detailed in the Orbit3 Low Level Command manual (503303), which is available on request.

5.6 ORBITACS CONFIGURATOR

A utility for configuring multiple OrbitACS controllers with their attached Orbit3 modules

5.7 ORBIT3GATEWAY CONFIGURATOR

A utility for configuring MODIM controllers with their attached PLC and Orbit3 networks

5.8 PIM UTILITY

A utility for configuring PIM controllers with their attached PLC and Orbit3 networks

5.9 ORBIT3 CONFOCAL UPDATER

This application enables the user to update the firmware of Confocal Controllers (requires both an Orbit and Ethernet connection).

5.10 WCM CONFIGURATOR

This application enables the user to configure a Wireless Connection Module (WCM) with Wireless Handtool devices. See Orbit3 Module manual for details.

5.11 WHT-M MAINTENANCE TOOL

A utility which reconfigures a WHT-M for replacement probes

5.12 AIR GAUGE UTILITY

This utility enables Air Gauge Modules (AGM) to be mastered from a PC. See the Orbit3 Module manual for details.

5.13 LTA LASER UTILITY

This utility enables LTA Laser triangulation sensors to display readings and diagnostics via a PC. See the LTA user manual (503773) for details.

5.14 SC(D)1 CALIBRATION AND CONFIGURATION SOFTWARE

This utility allows a single channel conditioner module (SC1-A or SCD1-A) to be calibrated and configured via a PC. See the SC(D)1-A user manual (503899) for details.

6 POWER UP CONDITIONS

On power up, the default conditions for all modules are:

- Basic Measurement Mode
- The Orbit Network Speed (Baud Rate) is 187.5 K Baud, (see Orbit Speed).

In addition, on power up default conditions for Digital Probes & AIMs are:

- Resolution defaults to14 bits, (see Programmable Resolution).
- Averaging defaults to 16, this provides:
    - An Electrical Measurement Bandwidth of nominally 100Hz
    - Measurement Register Refresh Rate of 244 updates per second (every 16 x 256µs = 4.096mS).

  (see Programmable Electrical Measurement Bandwidth).

These are the factory defaults for Resolution and Averaging. The user may set their own defaults. See Resolution and Averaging Configuration.

DIOM only:
- Default state on all pins at switch on is INPUTs.

DIOM2 only:
- Default state on all output pins at switch on is de-activated.

DIM only:
- Default state is Read Continuous.

EIM only:
- Default state is x1 quadrature mode and Reference Mark not active.

LT, LTA & LTH Laser Modules
- Default state is laser beam On

AGM only:
- Settings are stored in non volatile memory and are restored on power up.

6.1.1 RS232IM Default Baud Rate
- The default RS232 Baud rate is 9600 Baud.

7 MEASUREMENT MODES

7.1 OVERVIEW

Each of the following reading commands and modes have been developed to accommodate the differing scenarios and challenges that commonly present themselves to manufacturing and metrology systems. Each mode, although seemingly similar, has definitive differences; some of these initially appear subtle. The aim of this document is to provide simple and clear explanations to what each reading mode or command is, the basics of how it does it, and importantly, what each mode can be used for.
Normal readings, ReadBurst Mode, Dynamic Mode (1&2) and Buffered Mode are all capable of retrieving readings in either units of measurement (UOM) or in counts.
The table below shows the advantages and disadvantages for each mode.

Mode	Advantages	Disadvantages
Basic Measurement Mode	• Easy to use	• Slow with more than 1 module • Readings not synchronised
Difference Mode	• Fast	• Only max & min are stored • Not compatible with all module types
Buffered	• Fast • Synchronised Readings • Compatible with all controllers	• 3000 readings max • Not compatible with all module types

ReadBurst	• Fastest non dynamic method • Easy to use • Synchronised Readings • Compatible with all Orbit3 module types & controllers	
Dynamic Mode	• Fast • Synchronised Readings	• Limited to 31 modules max • High speed only • Not compatible with all module types • USBIM Mk2 / LITE & ETHIM 2.0 Only
Dynamic 2	• Fastest • Synchronised Readings • Up to 200 modules • High or Ultra high speed • Compatible with all Orbit3 module types	• USBIM Mk2 / LITE & ETHIM 2.0 Only

Refer to the Reading Rate Comparison Section for more speed based comparisons.

7.2 BASIC MEASUREMENT MODE

Each Orbit Module type has both simple *get reading* commands; *ReadingInCounts* and *ReadingInUnits;* returning the reading in the modules pre-configured unit of measurement.. For modules such as the EIM or DIOM that do not have a specific unit of measure, getting the the ReadingInUnits, will return the ReadingInCounts. Executing these commands simply returns the reading.

NOTE: Iterating through from one module in a network to the next and taking a reading on each module in quick succession is **not** a recommended or precise method of taking synchronised readings. ReadBurst is provided to do this.

7.3 DIFFERENCE MODE

In Difference mode, readings are taken continually and the maximum, minimum, number and sum of readings are stored in the module itself. These values can be read at any time using the appropriate Orbit commands. This mode has the advantage that the modules themselves do the calculations, not by the controlling software. Thus, the Orbit Communication speed does not affect reading rate.

Notes:
The sum and number of readings can be used to simply obtain the mean (average) reading.
For Digital Probe, LT, LTA, LTH, AGM and AIMs, Difference mode is only available in 14-bit resolution.
For Linear Encoder, Difference mode does not return the sum and number of readings.

For Digital Probe, AIMs and AGMs, the Reading Rate during Orbit Difference Mode is 244 updates per second (i.e. every 4.096ms).

When the Module exits from Difference Mode, the measurement conditions prior to entering difference mode are restored.

7.4 BUFFERED MODE

7.4.1 Introduction

Buffered capable Orbit modules have internal memory that can store up to 3000 synchronised readings at a reading rate of 244 reading / second, which can then be retrieved by the Orbit Library.
All measurements are 14 bit, (for DP, LT, LTA, LTH, AGM and AIM only).
Currently buffered capability is restricted to Digital Probes, LT, LTA, LTH, AIMs, AGMs and DIOM.

Buffered mode is ideal for slower, less powerful computers or when dynamic mode is not available.

Buffered Mode has three sub-modes:

- Sync Mode – readings are taken at a pre-defined interval
- Sample Mode – readings are taken under software control
- External Master Mode – readings are taken via an external master.

Refer to the Buffered Mode Example Code - Walk through and the Orbit Library Test example for further illustration of these modes.

When using buffered mode, the module(s) to be included in the collection should be individually selected in sync or sample mode and then enabled. All enabled modules are synchronously started by the Network OrbitBuffered Start method and stopped by Network OrbitBuffered Stop method. Note that after 3000 readings have been stored, no more will be read, since the internal buffer is full.

On the network OrbitBuffered Stop method, the Orbit Library downloads all enabled modules' buffered readings and exposes them via each module's BufferedData property.

7.4.2 Synchronized Mode

Once started, this mode sets all buffered enabled modules to store readings at a pre-defined Interval.

This interval is set by the ModuleBuffered OTUs property before enabling. One OTU (Orbit Timing Unit) is defined as being 102.4uS. The selected interval is required to be not zero and to be a multiple of 40 OTUs.

In order to simplify buffered sync mode, the Orbit Library also provides a ReadingInterval property (in microseconds), which avoids using OTUs. Upon setting the ReadingInterval property, the Orbit Library selects the nearest valid OTUs value and loads the ReadingIntervalErrorInUs property with deviation between the two.

Therefore, we recommend using the ReadingInterval property for all new designs.

Note that different modules can be assigned different Reading Intervals in the same buffered collection.

7.4.3 Sample Mode

Once started, this mode sets all buffered enabled modules to store readings each time the Network OrbitBuffered Sample method is called.
This Sample method is broadcast to all buffered enabled modules, hence these readings are synchronised to each other.

7.4.4 External Master Mode - Using EIM / DIOM2

In a similar way to External Master Mode in dynamic, an Encoder Input Module (EIM) or the DIOM2 (see External Master Mode) can be used to Sample readings in buffered mode in place of the controller.

In order for this to be valid, the following rules apply:

- The EIM / DIOM2 should be at the last module Index of the network
- Select the Network OrbitBuffered MasterAddress to be that of the EIM / DIOM2
- Enable all modules in Buffered Sample Mode

Upon Network OrbitBuffered Start

- EIM only - The EIM waits until its reference mark is passed
- Every TxSample counts, a buffered Sample command is automatically sent by the EIM / DIOM2.

Upon Network OrbitBuffered Stop

- Exits out of buffered external master mode.

After a buffered master collection has been stopped, the Status of the External Master module should be checked to ensure that there has not been a sync/sample gap error (a sample being triggered before the previous sample has finished triggering).

If a sync/sample gap error occurred then the External Master will have stopped triggering buffered samples at that point.
The error is cleared at the point that it is read back.

Note. When a Buffered master collection is stopped on an EIM, the Ref Action settings on the master module are reset to none.

7.5 READBURST

New with the Orbit Library, the ReadBurst command retrieves a single, synchronised block of readings of all contiguously capable modules on the network. ReadBurst is fast and precise and allows easy access to data immediately.
ReadBurst is designed for situations that require taking a set of synchronised readings quickly, processing them and moving on to the next operation. It is designed for many quick bursts of reads.
There is no configuration needed for ReadBurst other than ensuring the ReadBurst capable modules are contiguous from the first address; any non-capable modules will break contiguity, potentially reducing the number of modules included in the collection.

Refer to the ReadBurst Mode section for a code 'walk through'.

7.6 DYNAMIC MODES

7.6.1 Introduction to Dynamic Modes

Dynamic mode provides a method of obtaining synchronized measurements at high speed from Digital Probes, EIM, AGM, AIM, DIOM and other compatible Orbit products. This mode is of importance when measuring moving objects. It gives the user the ability to take high-speed measurements from a set of transducers that are sampled at the same point in time – i.e. simultaneous sampling.
Unlike ReadBurst, dynamic is centred around collecting large sets of data at known or predictable intervals over a period of time, the collected data can then be processed once the collection is complete.
Using the Orbit Library, dynamic modes are *only* available with a USBIM Mk 2, USBIM LITE, ETHIM 2.0 (and later) controllers.

To help you write your application we recommend that you refer to the Dynamic Modes 1 & 2 Code 'Walkthrough', the Orbit3 Code Examples, Orbit Library Test and the *OrbitLibrary Code Reference*.

7.6.2 Introduction to Dynamic 2

Dynamic2 mode was introduced with Orbit3 and the Orbit Library. This mode operates in a similar way to traditional dynamic, but with extra features and improvements:

- Fastest possible reading speeds
- Will run in both High or Ultra high speed
- Any number of modules. Up to 200 modules max.
- Has improved Collection Rate flexibility with the DynamicInterval property
- Compatible with all Orbit3 module types

7.6.3 Why Use Dynamic Mode

In basic measuring mode (non-dynamic), readings are read back from Orbit modules as required. A command to request a reading is transmitted from the controller & the reply from the Orbit module is then received. A finite amount of time is taken with the Orbit communications to transmit and receive data.
A single module on a network works well in this mode, as there is only the one module to communicate to, which gives fast reading rates. However, when there is more than one module on the network, there are several issues to consider:

- Is it important to synchronize readings?
- What reading rate is required?
- What type of Orbit controller is to / can be used? (e.g. USBIM, RS232IM, ETHIM etc.)

Note that Dynamic modes are only available if all modules and the controller are dynamic capable. Whether the desired Orbit Network is 'Dynamic capable' is automatically handled by the Orbit Library.

7.6.4 Collection Rate

A dynamic collection is defined by the rate of time that each *sync (*a collection of readings) is spaced. The Orbit Library supports the traditional dynamic sync rates of 1k, 2k and 4k, as well as the dynamic2custom (Dynamic 2) rate. This dynamic mode, introduced with the Orbit Library, optimises the dynamic collection rate for the current network set up, opposed to the traditional dynamic approach. Dynamic 2 also

provides a settable interval property to further configure a dynamic collection's sync rate.

Table 10. Collection rate details

Rate	Maximum Modules	Time Between Reads	Network Speeds
4K	8	256µS	High
2K	16	512µS	High
1K	31	1024µS	High
Dynamic 2	Network Maximum (200)	Between 256µS and 30 Secs	High UltraHigh

7.6.5 Implementing Dynamic2

To use Dynamic2, select the DynamicRate property to Dynamic2Custom.

The DynamicInterval property, only applicable with Dynamic2, allows the user to configure a custom collection sync rate (between 256µS and 30 Seconds). If zero is selected, Dynamic2 mode runs 'flat out' i.e. as fast as possible.

Apart from when running 'flat out', the MinimumInterval method, only applicable with Dynamic2, returns the minimum valid DynamicInterval allowed for the current set-up.

Apart from the extra properties, using the Orbit Library with Dynamic2 is the same as for using traditional Dynamic.

7.6.6 Dynamic Mode System Constraints

For the DP, LE AIM Modules only, care should be taken to ensure that the reading rate (Averaging) is set high enough for the faster Dynamic collection rates, otherwise loss of information will occur.
Therefore, the Measurement Bandwidth and the Dynamic Collection Rate should be programmed compliant with the table below:

Measurement Bandwidth (Hz)	Collection Rate (Readings per second)
450	4K Mode
420	4K or 2K Mode
320	4K, 2K or 1K Mode
200	
100	
50	
25	
12	
6	

The programming of the correct Bandwidth and Register Output Rate is the responsibility of the user. The Modules will not provide error checking.
See Programmable Electrical Measurement Bandwidth.

7.6.7 Dynamic Data

The readings returned from a collection using the Orbit Library is contained in the OrbitDynamic DynamicData property. This object contains:

- Results of each reading along with any error that occurred.
- CollectionStatus of the dynamic collection as a whole.

It is important to check the CollectionStatus and Reading errors to ensure valid data has been collected.

7.6.8 Dynamic External Master Mode

Dynamic mode can also be configured to have an external master.

By default, syncs are triggered (at the specified collection rate) by the Orbit Controller. Therefore the readings are *time* based.

However, when a network is set into dynamic external master mode; triggering of syncs is handled by the external master, which can be:

- Encoder Input Module (EIM).
  - This module can then trigger syncs that are *angle* based.
  - This is of particular use when measuring rotating parts.
  - Also note that an EIM can also be used with a Linear Encoder. In this case, readings would be *Displacement* based.
- DIOM2 module
  - This module can then trigger syncs that are edge triggered – see External Master Mode

Note that in this mode, the EIM / DIOM2 triggers the syncs. Therefore, it will not be part of the dynamic collection itself.

The external master is selected by setting the OrbitDynamic MasterAddress property to the module address of the EIM / DIOM2 to be used as external master. The EIM / DIOM2 master only triggers sync to modules with lower addresses. Therefore, the EIM / DIOM2 master should have been previously set to be the last addressed module on the Orbit Network.

The EIM / DIOM2 master should have its DynamicMasterMode and TxSync properties set as well as the OrbitDynamic MasterAddress property should be set to the address of the EIM / DIOM2 master.

Note. When Dynamic master completes on an EIM, the Ref Action settings on the Dynamic master module are reset to none.

7.6.8.1 EIM DynamicMasterMode Property

A variety of settings are available to start triggering Syncs from the EIM. The DynamicMasterMode property can be set to Start triggering:

- Immediately (Instant).
- Until a certain number of counts have occurred (HoldOff). The EIM HoldOff property contains the number of counts to hold-off for.
- Until the encoder's reference mark has been passed. EIM reading zeroed (Refaction_Reset).
- Until the encoder's reference mark has been passed. EIM reading preset (Refaction_Preset).

7.6.8.2 DIOM2 DynamicMasterMode Property

At present, the DIOM2 has one setting available to start triggering Syncs. The DynamicMasterMode property can be set to Start triggering:

- Immediately (Instant).

7.6.8.3 TxSync Property

In order to trigger syncs from the dynamic master at a certain multiple of counts, the TxSync property is provided.
(e.g. in Dynamic 4k mode for an encoder with 3,600 counts per revolution, setting TxSync to 10 will produce 1 sync every 10 counts = 1 per degree)

Traditional dynamic rates (2k and 1k) required extra Syncs to transfer a block of data in external master mode. This should be taken account of if using these modes.
The table below shows how many syncs per block are required for different collection rates.

Collection Rate	**Syncs per block**
4k	1
2k	2
1k	4

Dynamic 2	1

Therefore, the TxSync value set will be lower than expected (divided by the Syncs per Block value).
(e.g. in Dynamic 2k mode for an encoder with 3,600 counts per revolution, setting TxSync to 5 will produce 1 block of readings for every 10 counts = 1 per degree)
Use Dynamic2 to avoid this issue.

7.6.8.4 Sync Pulse Rate when using EIM as a Sync source

When using an EIM to generate Sync pulses, the rate is determined by several factors:

- Encoder rotation speed
- Number of encoder pulses per revolution
- Number of encoder pulses required per Sync pulse
- ReadingInterval property if using Dynamic2 mode.

Since the Orbit EIM Dynamic system is based around a minimum time between Syncs, it is possible to violate this timing (e.g. by rotating the encoder too quickly).
For this reason, the calculation below should be made to guarantee reliability.
Maximum speed (Revolutions per second) =

Where P = Encoder pulses per revolution

Example with P = 3600 (Encoder with 3600 pulses / rev), TxSync = 10 and MinSyncGap = 290uS (minimum with EIM in Dynamic master mode)
(Note: if using Dynamic2: MinSyncGap = MinimumInterval() + 34uS
Where the 34us is due to the EIM update rate)

Maximum speed = $1/ [(3600 / 10) * 290*10^{-06}] = 9.57$ revs / Second

If this maximum speed is exceeded, the collection will will automatically be stopped and a 'Sync Timing Violation' error will be assigned to OrbitDynamic status property by the Orbit Library.

Note that this calculation assumes a constant speed throughout. As this is rarely the case in practice, the practical maximum speed will be lower than the calculation suggests.

The EIM will have also detected the timing violation and set its error status, if the EIM is read via the OrbitModule ReadInCounts or ReadingInUnits properties, an ‘Encoder Module Sync Gap Error’will be reported. This can be cleared by using the ModuleStatus UpdateStatus method.

7.6.9 Requirements for Dynamic Mode

The Dynamic measurement system requires a USBIM controller (Mark 2 or later) to be installed.
For module compatibility, refer to Module Compatibility.
For controller compatibility, refer to Controllers & Software.

To be able to use Dynamic mode, ensure that you are running the latest software for modules, controllers and Orbit3 Support Pack For Windows.

Should problems occur, the Orbit3 Reporter should be run to check the firmware / software of the Orbit Measurement system connected.

7.6.10 Hints and Tips on Using Dynamic Mode

- Make sure your software & firmware is up-to-date for running dynamic mode.
- Refer to the Orbit3 Code Examples and Orbit Library Test whilst writing any code.
- See the OrbitLibrary Code Reference.
- If using Orbit3 modules and controllers, the status LEDs will be lit for the duration of the collection on both (modules and controllers) giving a visual indication of correct operation.
- If an AGM module is included within the network for dynamic collection and it has been re-mastered singe being initialised on the network, the dynamic preperation sequence will fail indicating the AGM module index and the fault code “Not in Normal Mode”. If this error is encountered, clearing the error can be performed by either re-initialising the network, or calling the ‘updateInfo’ method the specific AGM module in question.

The example scenarios / schemes illustrate how dynamic mode operates.
Refer to the Orbit3 Code Examples, Orbit Library Test and the *OrbitLibrary Code Reference* for more details

This scenario is used to illustrate measuring the profile of a rotating cam shaft when time triggered readings are required.
In this example, a cam shaft has eight DPs (Digital Probes) measuring various positions. A motor rotates the cam shaft.

The eight DPs should be synchronized together – i.e. they should all take their readings at the same time with no skew.
The readings are desired to be 1 millisecond apart.

Dynamic 2 mode can be used in this particular case to read every 1000µs. Enough readings should be taken to ensure that a complete cycle has taken place.

7.6.11 Dynamic Schemes

7.6.11.1 Dynamic Scheme 1 – USBIM controller as the Sync source

Typical application using an USBIM controller as the Sync source

Refer to the Dynamic Modes 1 & 2 - code 'walk-through'

This scenario is used to illustrate measuring the profile of a rotating cam shaft when angle triggered readings are required.
In this example, a cam shaft has four DPs (Digital Probes) measuring various positions. A motor rotates the cam shaft and a rotary encoder, connected to a EIM (Encoder Input Module), measures the angle of rotation (3600 pulses per rev).

The four DPs should be synchronized together – i.e. they should all take their readings at the same time with no skew, once per degree of rotation.
Enough readings should be taken to ensure that a complete cycle has taken place.

External Master Dynamic mode with Dynamic2 rate can be used in this particular case to trigger syncs from the EIM. With Dynamic2, 3600 counts / revolution and one reading per degree, TxSync should be set to 10 (See EIM TxSync Property).

7.6.11.2 Dynamic Scheme 2 - Encoder as the Sync source

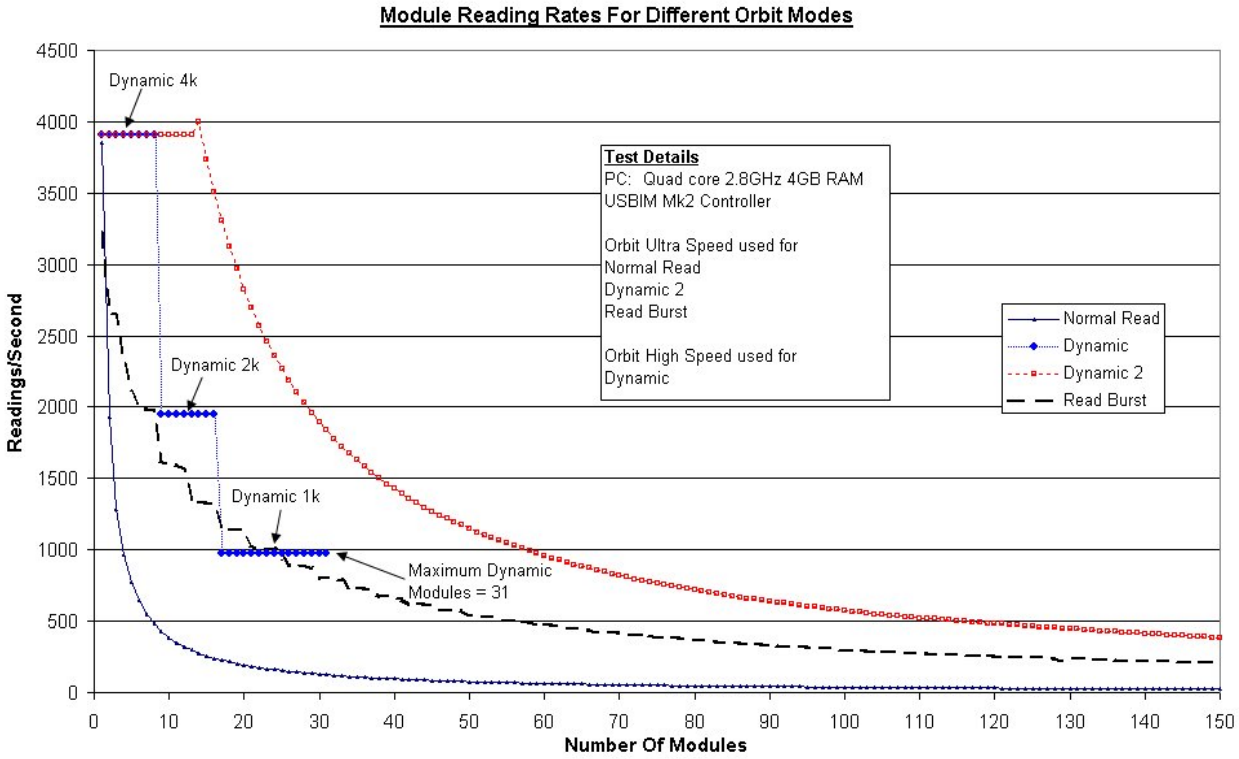
Typical application using an Encoder Input Module (EIM) as the Sync source

Refer to the Dynamic External Master Mode Code 'Walkthrough' for an example.

- If using external master mode with an EIM and the master EIM does not trigger readings, the EIM could be counting in the wrong direction. Confirm that when the encoder is rotated the direction of count is as expected.
  - Note:
    - TxSync is positive (e.g. TxSync = 5) the EIM will be expecting the count to increase when the Encoder is rotated.
    - TxSync is negative (e.g. TxSync = -5) the EIM will be expecting the count to decrease when the Encoder is rotated.

7.7 READING RATE COMPARISON

7.7.1 USBIM MK2 Controller reading rates



The graph shows a comparison of the reading modes available using the USBIM MK2 Controller.
Note that reading rates will vary with differing PC systems.

Example.
From the graph, for 50 Modules with Readburst in Ultra speed mode, each Module has a reading rate of approximately 500 readings per second.
This equates to a combined throughput of 50 * 500 = 25000 readings per second.

- Clearly, for larger networks, the advantages of Readburst and Dynamic2 can be seen.

7.7.2 ETHIM Controller reading rates

The reading rates of the Orbit Ethernet based controllers are very dependent on the amount of other network traffic.
For applications that require high Orbit reading rates it is recommended that other network traffic is minimised.

7.7.2.1 ETHIM 2.0 Controllers

These have similar reading rates to the USBIM MK2 Controller
For Orbit Dynamic applications it is recommended that a dedicated network is used.

7.7.2.2 ETHIM Controller

The graph shows a comparison of the reading modes available using the ETHIM Controller.
Note that reading rates will vary with differing PC systems and networks.

Example.
From the graph, for 50 Modules with Readburst in High Speed mode, each Module has a reading rate of approximately 120 readings per second.
This equates to a combined throughput of 50 * 120 = 6000 readings per second.
Whereas with normal reads, we can only achieve 9 readings per second,
i.e. 50 x 9 = 450 readings per second.

7.7.3 RS232IM MK2 Controller reading rates

The graph shows a comparison of the reading modes available using the RS232IM MK2 Controller.
Note that reading rates will vary with differing PC systems.

Example.
From the graph, for 50 Modules with Readburst in High Speed mode at an RS232 rate of 115200 Baud, each Module has a reading rate of approximately 25 readings per second.
This equates to a combined throughput of 50 * 25 = 1250 readings per second.
Whereas with normal reads, we can only achieve 5 readings per second,
i.e. 50 x 5 = 250 readings per second.

7.7.4 WIM Controller reading rates

The WIM is intended for static measurements of a typical 25 readings per second (due to the latency in Bluetooth communications).
To optimise speeds for multiple probe systems, Readburst is recommended. In this mode it is possible to maintain the 25 readings per probe.
Example. With 8 probes at 25 readings per probe = 8 x 25 measurements per second.

7.8 SUMMARY

When data is needed quickly for processing on the spot, ReadBurst should be used. When data needs to be collected over a period of time for processing after the collection, Dynamic 2 should be used. For use with older, slower hardware, or integrating Orbit 3 hardware with a PLC, Buffered mode is available.

8 ORBIT FEATURES AND COMMANDS

8.1 HOTSWAP

Note: To use this mode in its simplest form requires Orbit3 compatible TCONS and Modules.
However to fully use this mode requires one of the following Orbit Controllers: USBIM MK2, ETHIM 2.0, ETHIM, RS232IM MK2 or RS232IM and the Orbit Library.

Hot Swap is a feature of Orbit3 and is the ability to 'assume' a module's Orbit identity. The identity 'assumed' is stored in the module's T-Con (only written after successfully adding a module using the Add Module, Notify Add Module, or Ping methods). This feature is particularly useful when replacing a module (but leaving the original T-Con in place, which contains the stored / original Orbit identity). The new (replaced) module automatically 'assumes' the original Orbit identity (from the T-Con) on power-up (if compatible**).

This means that after replacing a module, operating software does not need to change as all module identities are effectively the same as they were.

A further feature of Hot Swap mode is to store the module's last known address in the T-Con. This address is 'assumed' on power-up (if compatible**). This means that after

initial set-up, operating software can be written / modified to be a more flexible and simpler design. See FindHotswapped section for more details.

Notes

The module identity returned via BaseModuleID is that of the actual Orbit identity, **not** the 'assumed' one.
The ModuleID is the identity that was used to set up the device (which could be a HotSwapped identity).

In normal operation, the module Orbit identities and addresses are *usually* the same as they were the previous time, hence there is no need to 'assume' the Orbit identity.

** A *compatible module* is the same module type and stroke. For example a 2mm Digital probe is only compatible with another 2mm Digital Probe. Incompatible modules indicate this by flashing the red status LED.
Note that incompatible modules do not 'assume' Orbit identities and addresses.

To clear an incompatible module, simply add this module (using the AddModule or NotifyAddModule methods) to the Orbit network. This will store the module's actual Orbit identity in the T-Con and thus clear the compatibility issue.
See HotSwap.

8.1.1 Using Orbit3 without Hot Swap

Question. What happens if you don't want to use the Hot Swap function?

Answer. You can use the system normally without Hotswap.
Basically it's transparent unless you choose to take advantage of it.

If you don't want to use Hotswap, don't change anything, it'll just work as it always did. Just 'Notify' and 'Set Address' as normal.

Summary

- 'Notify' always responds with the Module's ID irrespective of what is stored in the TCON.
- 'Identify' will return the details of the module and not the details stored in the TCON.

Note. If the module's status LED flashes, (at 4 times per second), due to a HotSwap error, the error condition will be cleared by a 'Set Address' command to the modules ID.

- The most likely cause for the error being the Module being a different type or stroke to the one previously plugged into the TCON.

8.2 FINDHOTSWAPPED

To use this mode requires one of the following Orbit Controllers: USBIM MK2, ETHIM 2.0, ETHIM, RS232IM MK2 or RS232IM and the Orbit Library along with Orbit3 compatible TCONS and Modules.
This function scans the selected Orbit network for hot swap 'assumed' addresses. This allows modules to be communicated to without having to use the Add Module, Notify Add Module, or Ping methods each time.
Note that this feature is one command only.

Note also that the Add Module, Notify Add Module, or Ping methods need to be called once, to initially store the T-Con data (hot swap 'assumed' addresses). This operation would only need to be performed once and could use standard utility programs to action this (avoiding the need to write custom software).

Note that if modules have been altered (e.g. a module has been removed), non contiguous addresses (i.e. there is a gap) will mean that addresses after the gap will be ignored.
For example, if there are modules with T-Con addresses:1,2,3,5,6 then only 1,2,3 will be setup as there is a gap between 3 and 5, i.e. 5 & 6 are not set up.
See FindHotSwapped.

8.3 CLEAR TCONS

To use this mode requires the Orbit Library along with Orbit3 compatible TCONS and Modules.
This function clears the T-Con HotSwap data (that have a module attached).
It should only need be used when rebuilding systems from parts from existing systems, in order to clear conflicting hotswap data.(ie 2 TCONS with the same address data). See Clear TCON Memory.

8.4 PING

To use this mode requires one of the following Orbit Controllers: USBIM MK2, ETHIM 2.0, ETHIM, RS232IM MK2 or RS232IM and the Orbit Library along with Orbit3 compatible TCONS and Modules.

This Orbit3 command interrogates the specified Orbit network to find all modules connected on it.
Each module found will be set-up / added in a process that takes about 20 seconds.
Orbit modules that have already been added (using Add Module, Notify Add Module, or Ping methods) will not be picked up by a subsequent Ping.

Note: that due to the way the OrbitPing command works, the order (address) in which the modules are found may be different each time. Therefore, the order may need to be changed using the 'RemapIndex' and 'ApplyRemap' methods.
See Change Address.

Ping can be used in conjunction with the 'Save' method to quickly configure a system with the modules found.

8.5 READINGINUNITS

To use this mode requires the Orbit Library along with Orbit2 and Orbit3 Modules.

This feature allows Orbit modules to simply return their reading in the required Units Of Measure (UOM). For example, a 2mm Digital probe at mid position would return 1mm, whereas 'ReadingInCounts' would return 8192 counts.

For Orbit3 modules where they operate over a fixed range and can exceed the range (I.e. go over or under range) the reading is limited to each end of the range (ie for 4-20mA AIM, 3mA would return 4, and for a 10mm DP at position 10.2mm it would return 10mm).
If the module is at the end of its range, the Orbit Error code should be evaluated to identify if you are over or under range.

For Orbit3 modules with a fixed measurement range that can be zeroed (eg. Confocal System) it is not possible to limit to the range so on error conditions a reading of NaN (not a number) is returned.

8.6 ORBIT SPEED

The Orbit Network Speed (Baud Rate) can be set or read using the NetSpeed property of the Network.
The available speeds of both Modules and Controllers can be obtained using their AvailableSpeeds properties.

The available speeds will be dependent on both the Orbit Controller and the Module. i.e. if the Module is 'Ultra' speed capable, but the Controller (such as WIM) is only 'High' speed capable, then 'High' speed will be the maximum achievable speed.
The Orbit Library automatically updates the Network speed (and available speeds) depending on the combination of Controller and connected Modules.

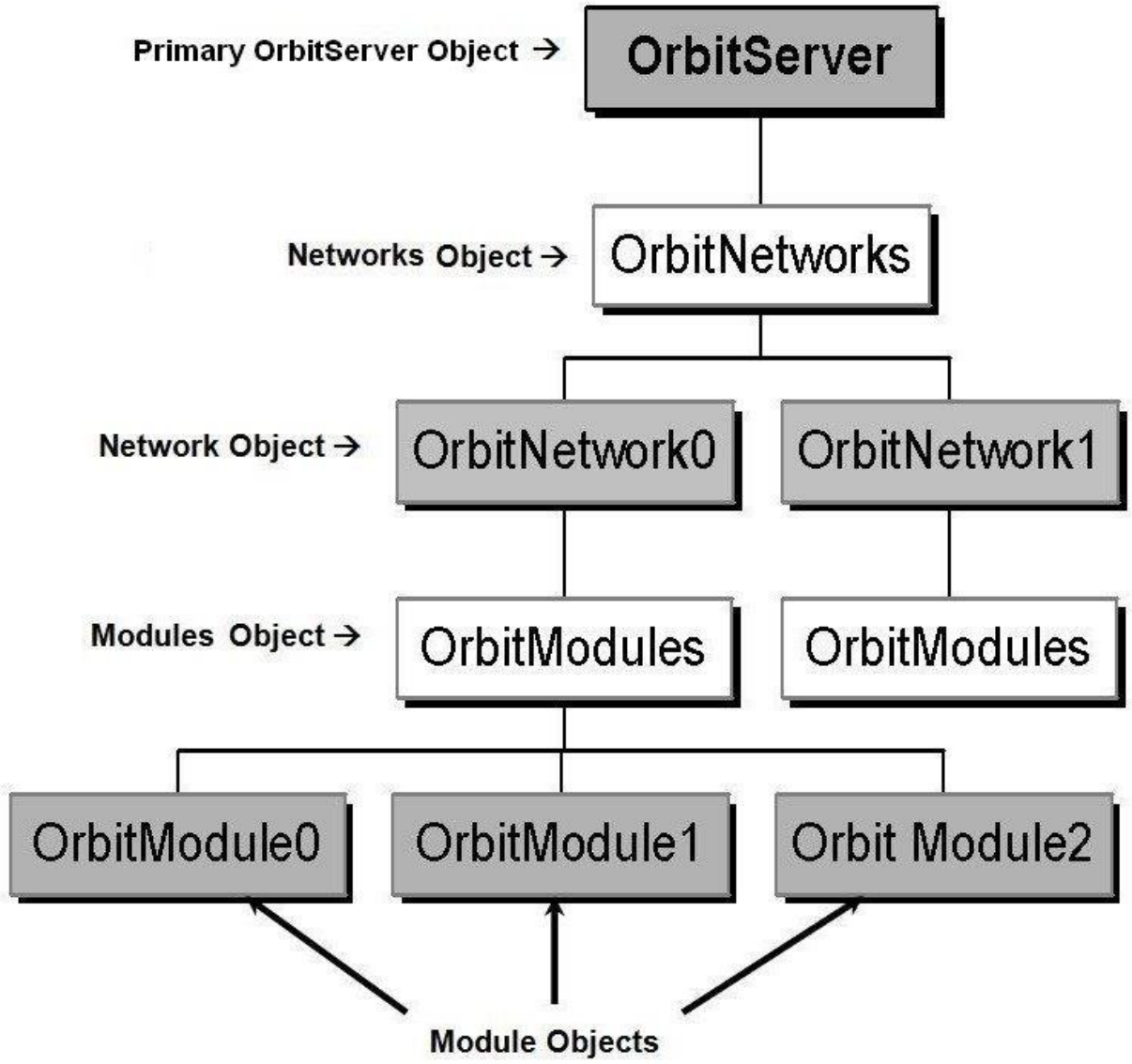
Orbit Network Speed	Baud Rate (Bits/Sec)
Standard	187.5k (Default)
High	1.5M
Ultra	2.25M

Note. For both High and Ultra speeds, ensure that the recommended Orbit cable and power supply configuration is used. See the Orbit 3 System manual for details.

9 ORBIT LIBRARY

9.1 OVERVIEW

The following diagram shows a top-down view of the Orbit Library.
The Primary object is **OrbitServer**; beneath this are other objects that are generated automatically when **OrbitServer** is created**.**
Each object has a set of functions or 'Methods' that are used to initiate specific actions and a set of parameters or 'Properties' that can be read or set.



For a more detailed diagram of the class hierarchy, see OrbitLibrary Code UML Diagram.
Primary Object
Object name: OrbitServer
Orbit Server is the primary Object in the Orbit Library hierarchy.
It contains the whole Orbit Measurement system, connecting to it, disconnecting from it and providing system wide controls, providing the network's object to the users and maintaining it.
On connecting to the OrbitServer, the OrbitNetworks object is created representing all 'discovered' / available Orbit Networks.

9.1.1 Networks Object

Object name: OrbitNetworks
The Networks object exists available networks on the PC, providing information about how many networks are available and providing a simple interface to them and provides the individual Network objects (via an index).

9.1.2 Network Object

Object name: OrbitNetwork.
The Network Objects are a software model of the network channel hardware, providing information about the network and controls (such as speed settings), also providing and Maintaining the Modules Object.
Each Network Object represents a single Orbit network channel.
Individual OrbitNetwork objects can be obtained by accessing the OrbitNetworks object via an index.

9.1.3 Modules Object

Object name: OrbitModules.
Object for containing all the OrbitModule objects currently active on a specific OrbitNetwork.
The Modules Objects maintains the individual module objects providing interfaces for adding and removing modules and for directly interfacing with the individual module objects (for reading and setting of their properties etc).
It manages all the OrbitModule objects currently active on a specific OrbitNetwork.
Individual OrbitModule objects can be obtained by accessing the OrbitModules object via an index.
Note that individual modules are not 'discovered' as they are for Orbit Networks when Orbit Server is connected to – they are added using the methods made available in the OrbitModules object.

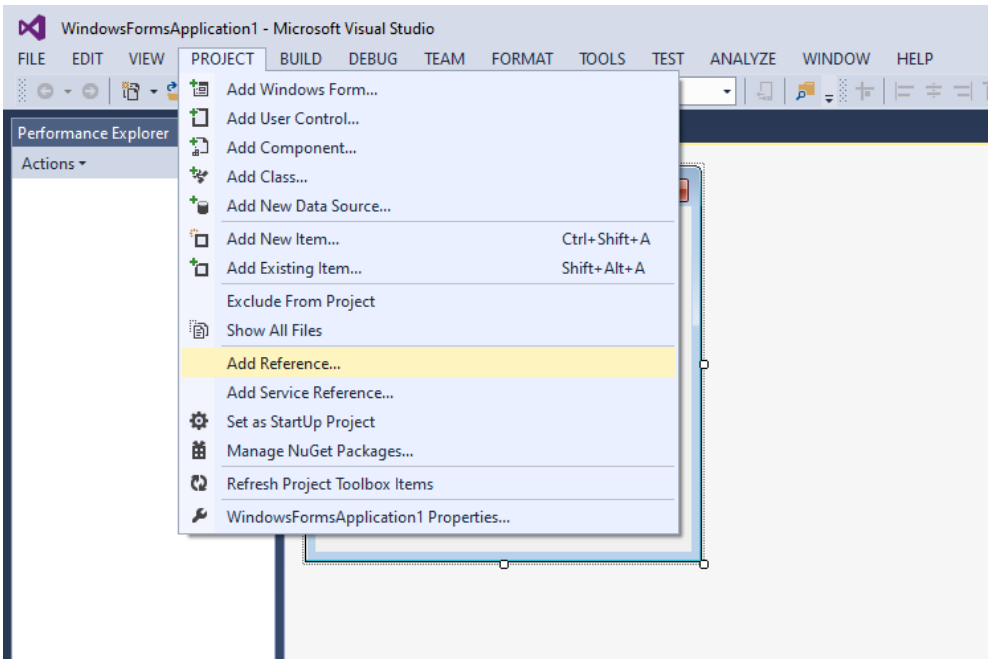
9.1.4 Module Object

Object name: OrbitModule.
Represents a single Orbit module.
The Module Objects provides a model of each individual connected orbit module exposing functions and properties that enable to user to read the modules and set their properties.
Each Module Object represents a single Orbit module.

9.2 REFERENCING THE ORBIT LIBRARY

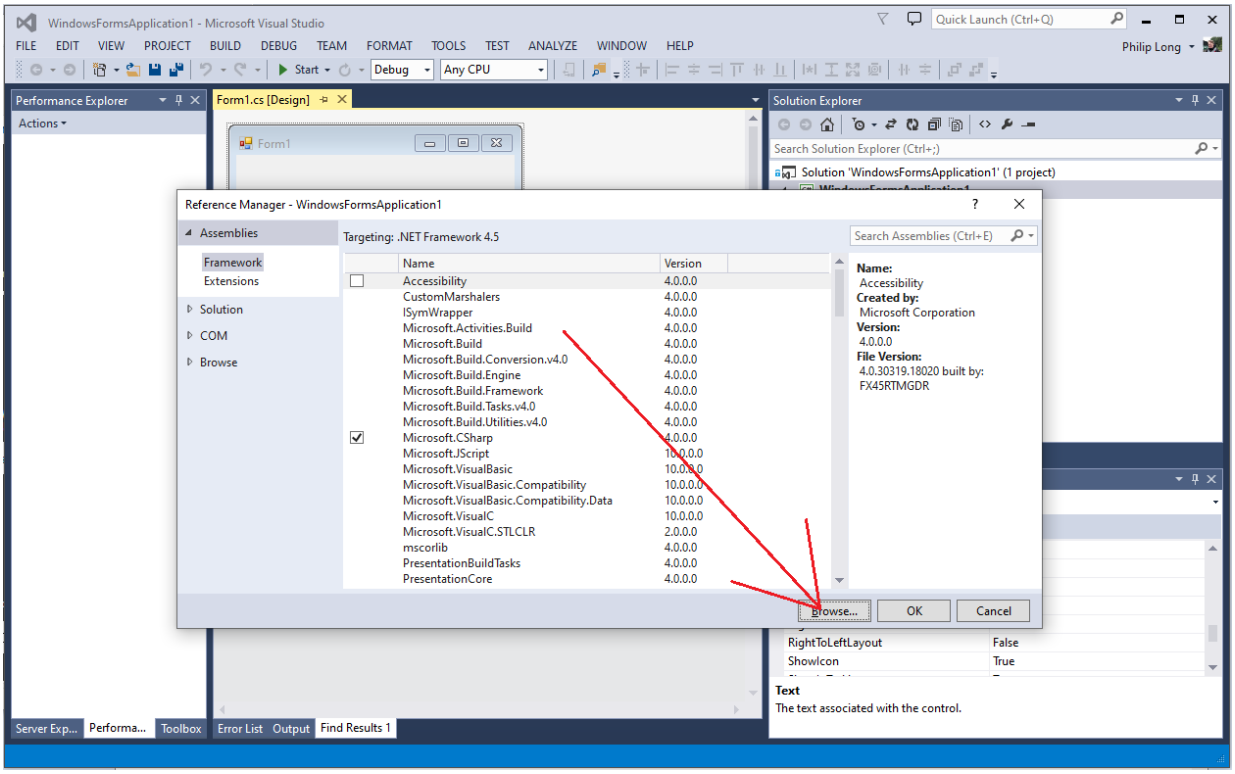
In order to use the Orbit Library a reference to it needs to be created.
To do this with C# Visual Studio Professional 2013 proceed as follows:

Add a Reference by Project->Add Reference
This will bring up the 'Add Reference' window. Select the 'Browse' tab.

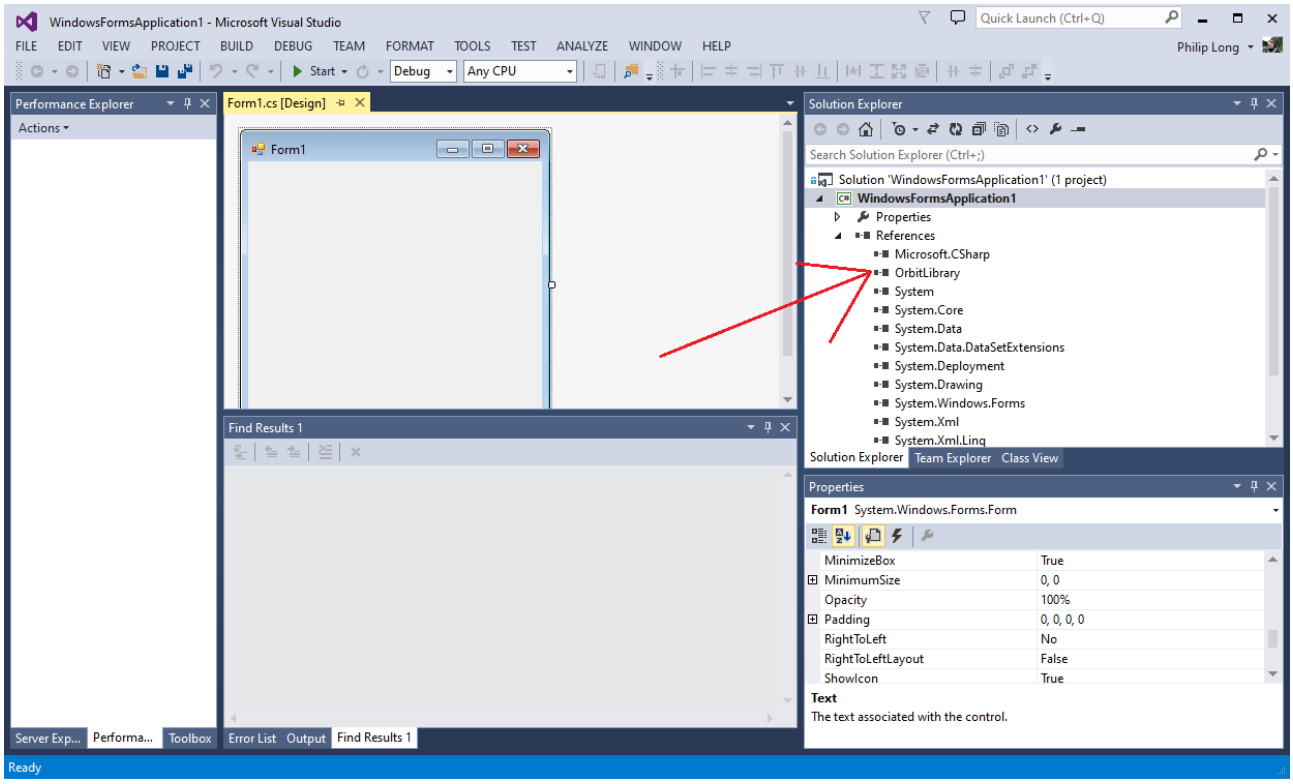


For 64 bit machines, navigate to C:\Program Files (x86)\Solartron Metrology\Orbit3 Support Pack for Windows\Drivers\Library
or for 32 bit machines, navigate to C:\Program Files\Solartron Metrology\Orbit3 Support Pack for Windows\Drivers\Library
(or for either, the location of the Windows folder if not using drive C:)

Select 'OrbitLibrary.dll'. Click OK



In the References section of the Solution Explorer, the OrbitLibrary reference should be visible.



9.3 ORBIT LIBRARY COM INTERFACE

The Orbit Library is designed to work as a COM library to allow interfacing to older, legacy programs that do not support the .NET Framework (e.g. Microsoft Excel 97 VBA).
See COM Interface for more details.

9.4 MIGRATING FROM THE ORIGINAL ORBIT COM LIBRARY

The basic hierarchy of the Orbit Library is similar to that of the Orbit COM in that it uses an OrbitServer, OrbitNetwork(s) and OrbitModule(s).
However, the internal working of the OrbitLibrary is of a different structure.
Therefore, code written using the Orbit COM is not compatible with the Orbit Library.

The following section details a comparison between methods (using C Sharp as an example).

Action	**Orbit Library**	**Orbit COM Library**
Reference Library	Reference: OrbitLibrary.dll	Reference: OrbitCOM.dll
Create Library Instance	OrbitServer Orbit = new OrbitServer()	OrbitServer Orbit = new OrbitServer()
Connect	Orbit.Connect()	Orbit.Connect()
Disconnect	Orbit. Disconnect()	Orbit. Disconnect()
Number of connected networks	Orbit.Networks.Count	Orbit.Networks.Count
Add a module to Orbit Network 0	Orbit.Networks[0].Modules.AddModule()	Orbit.Networks[0].Modules.Add (String ModuleName)
Notify and add a module on Network 0	Orbit.Networks[0].Modules.NotifyAdd()	Orbit.Networks[0].Modules.NotifyAndAdd (String ModuleName)
Get Identity of Module 0 on Network 0	String MyOrbitID = Orbit.Networks[0].Modules[0].ModuleID	String MyOrbitID = Orbit.Networks[0].Modules[0].ModuleID
Read Module 0 on Network 0 in Units of measure	Double MyReading = Orbit.Networks[0].Modules[0].ReadingInUnits	Double MyReading = Orbit.Networks[0].Modules[0].ReadCurrentInUOM
Preset Module 0 on Network 0 to 100 Counts	Orbit.Networks[0].Modules[0].PresetInCounts = 100	Orbit.Networks[0].Modules[0].SetPreset(100)

As can be seen, the Orbit Library is very similar to the Orbit COM Library in terms of its usage.
Other modes of operation (e.g. dynamic) are improved in the Orbit Library and thus there is less similarity here with the Orbit COM Library.
Refer to the Example Code - Walk through and Orbit Library Test for more detailed code examples.

Note that the Orbit Library has extra members available for each class (i.e. modes, controllers and modules) that are provided to produce less complicated code implementation. It also has built in comprehensive error checking and automatically deals with compatibility / legacy issues.

10 EXAMPLE CODE - WALK THROUGH

10.1 OVERVIEW

The Orbit Library provides an accessible way to receive readings with a variety of methods and performance specific modes to cater for a wide range of metrology needs. This section takes the supplied examples included with the Orbit3 Support Pack for Windows, and expands upon them.
See the Examples installed as part of the Orbit3 Support Pack for Windows for full example code listings. Other examples are taken from Orbit Library Test; the project files for this application, including source code listings, are also available within the Orbit3 Support Pack for Windows.
As with any .Net C# library, suitable exception handling should be employed.

10.1.1 COM Interface

A COM (Component Object Model) interface is supplied with the OrbitLibrary, providing a binary interface for the OrbitLibrary.
COM interfaces have bindings for C++, Visual Basic and Delphi applications allowing for easy and accessible implementation of the Orbit Library.
Examples of interfacing to the COM are also included within this document as well: the VBA example Orbit3ExcelCOMExample.xls, and the C++ example Orbit3CppComExample.
Interface to COM objects and methods of creating instances of them, varies from compiler to compiler. The C++ example given is the method for Borland C Builder 5. In general a *create* object will need to be called, rather than new.
More recent C++ compilers do not require the get_Item method and instead abstract it to use array indexers for example Modules->get_Item(0); would be Modules[0]; for example: Visual Studio.

10.2 CONNECTING TO THE ORBIT LIBRARY

The OrbitServer makes up the core of the Orbit Library, allowing software developers to integrate the Orbit3 series into production and research applications. It is through the core OrbitServer instance that all communication with Orbit3 is mediated.

The following sub-sections provide explanations, examples and advice for creating an instance of the OrbitServer and talking with the connected Orbit3 Controllers. Each Controller can have multiple Orbit Modules connected by a serial bus. The controller is responsible for each of these modules, forming a network. A network may only have one controller attached to the bus.
The controllers handle all communication with the Modules and OrbitServer handles all communication with the controllers.
To get data from modules, an instance of the OrbitServer needs to be created and a connection established. The OrbitServer shall then search for any registered (see Orbit3 Registration) and connected controllers. These are made available in the form of OrbitNetwork objects, which in turn, contain OrbitModule objects that represent the Orbit3 hardware probe Modules.

10.2.1 Initialising The OrbitServer

To create an instance of the OrbitServer a reference must be made to *OrbitLibrary.dll* assembly. See Referencing the Orbit Library.
The Orbit Library types must then be imported from the Solartron.Orbit3 namespace.

C# (.NET)

```
using Solartron.Orbit3;
```

C++ (COM)

```
#include "Orbit3CppCOMExample.h"
```

An instance of the OrbitServer must then be created. It is recommended that a reference to the OrbitServer instance is maintained throughout the duration of the application to avoid having to reconnect to the OrbitServer.

C# (.NET)

```
OrbitServer Orbit;
Orbit = new OrbitServer();
```

C++ (COM)

```
OrbitServerPtr Orbit;
Orbit = CoOrbitServer::Create();
```

10.2.2 Connecting to The OrbitServer

- Prior to connecting to the OrbitServer, the AllowOSSuspend property (default = True) should be set False if required.
  - If False, the Orbit Library will inhibit the operating system from entering Idle Suspend while connected to Orbit networks.

To avoid the complexity of searching for and handshaking with the various Orbit Controllers, the Connect() method is provided. On calling this Connect method, OrbitServer runs setup routines and builds an accessible collection of OrbitNetwork objects pertaining to the connected / discovered Orbit Controllers.

C# (.NET)

```
Orbit.Connect();
```

C++ (COM)

```
Orbit->Connect();
```

The connection process is time consuming due to the resets and delays involved in the process to initialise the controllers into a known and predictable state upon each Connect() call.
The OrbitServer.Connected bool is exposed to provide access to the OrbitServer's connection status.

- If unable to connect, the OrbitServer shall throw an exception detailing the fault.
- If the call to Connect() succeeds, OrbitServer.Connected is set true.

Important!

- There can only be a single instance of the OrbitServer connected at any one time on a system. This is enforced through a file handle *mutex* that is created on Connect() and released on Disconnect().
- If another instance of the OrbitServer attempts to connect to Orbit, when another instance has already been connected, the OrbitServer will throw an exception stating that the Orbit Library is already in use.

10.2.2.1 WIM Controllers

With WIM controllers (which operate via Bluetooth), the Connect() method will only find them if already paired (see the Orbit System manual for pairing instructions).

The original method for connecting (prior to June 2017) is no longer implemented. This took the form of an overridden function for the connect method, which automatically paired with any WIMs found, if the FindWIMs parameter was set to true.
The overridden function for the connect method is left in place (purely to not break any existing programs that use it), but the FindWIMs parameter has no effect.

10.2.3 Disconnecting from The OrbitServer

When closing an application, the OrbitServer.Disconnect() should be called to allow the OrbitServer to shutdown and close communications with the Orbit Controllers correctly.

- On disconnecting, the OrbitServer.Connected bool is set false.

C# (.NET)

```
Orbit.Disconnect();
```

C++ (COM)

```
Orbit->Disconnect();
```

10.3 LISTING ORBIT NETWORKS

To access network commands, a collection of OrbitNetwork objects are exposed once an OrbitServer has been connected to. To identify an Orbit Controller, the controller's name and type is exposed, allowing the collection to be iterated.

C# (.NET)

```
for (int networkIndex = 0; networkIndex < Orbit.Networks.Count; networkIndex++)
{
        Console.WriteLine(Orbit.Networks[networkIndex].Name);
}
```

C++ (COM)

```
for (int networkIndex = 0; (networkIndex < Orbit->Networks->Count); networkIndex++)
{
        OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(networkIndex);
        cout << \t" << AnsiString(OrbNet->Name).c_str() << endl;
}
```

See the OrbitLibrary Code Reference for detailed information on the other accessible properties of OrbitNetwork.

10.4 ADDING ORBIT MODULES

Once connected to the OrbitServer, OrbitServer.Networks shall be populated allowing access to controllers and modules. Each OrbitNetwork has a collection of OrbitModules; OrbitNetwork.Modules. By default, a network shall be configured with an empty Modules collection.
The different methods to add, remove and configure Orbit 3 modules are described in the following sub-sections.

10.4.1 Listing Orbit Modules

Modules can be listed in the same manner networks can be listed. The listing below print the ID for each OrbitModule connected to Orbit.Networks[NETINDEX].

C# (.NET)

```
const int NETINDEX = 0;  //Network whose modules shall be listed.

for (int moduleIndex = 0; moduleIndex < Orbit.Networks.Count; moduleIndex++)
{
        Console.WriteLine(Orbit.Networks.[NETINDEX].Modules[moduleIndex].ModuleID);
}
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
for (int moduleIndex = 0; (moduleIndex < OrbNet->Modules->Count); moduleIndex++)
{
        OrbitModulePtr OrbModule = OrbNet->Modules->get_Item(moduleIndex);
        cout << \t" << AnsiString(OrbModule->ModuleID).c_str() << endl;
}
```

See the OrbitLibrary Code Reference for detailed information the other accessible properties of OrbitModule.

10.4.2 Add Module

OrbitModule.Modules.Add() allows a specific Orbit 3 module to be added to a network by passing the exact module ID string of module; for example, '*100C619P19*'.

C# (.NET)
```
Orbit.Networks.[NETINDEX].Modules.AddModule("100C619P19");
```

C++ (COM)
```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->AddModule(WideString("100C619P19"))
```

The Add function has been designed with a input text box in mind. *An implemented example of this can be found within the* Orbit Library Test

10.4.3 Notify Add Module

OrbitNetwork.Modules.NotifyAddModule() will by default block the current thread of execution indefinitely until a module Notifies, the *Esc* key is pressed, or StopNotify() is called. The OrbitServer continuously sends notify commands to the Orbit controller, the controller in-turn will send out notify commands along the bus to each of the Orbit modules. This stops when either a valid ID or an error is returned. When a valid ID is received by the OrbitServer, an address on the network is assigned to the module. Errors received are then reported by the OrbitServer in the form of an exception with a relevant message.
When NotifyAddModule() has been called, an Orbit3 DP for example, will notify when the probe tip's position is physically changed, the DP will then be added to the network.

C# (.NET)
```
Orbit.Networks.[NETINDEX].Modules.NotifyAddModule();
```

C++ (COM)
```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->NotifyAddModule();
```

NotifyAddModule() shall return false if a module is not added to the network. *For details on configuring a NotifyAddModule() call, see the OrbitLibrary Code Reference.*

To stop a network from notifying, a call to OrbitNetwork.Modules.StopNotify() is required. The LibraryTest application has two buttons; one for starting NotifyAddModule(), and one to StopNotify().

10.4.4 Ping

OrbitNetwork.Modules.Ping() shall find all *ping-able* modules on the network and returns the number of new modules found. The found modules are addressed and added to the network.

C# (.NET)
```
int modulesFound = Orbit.Networks.[NETINDEX].Modules.Ping();
```

C++ (COM)
```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->Ping();
```

10.4.5 FindHotSwapped

OrbitNetwork.Modules.FindHotSwapped() shall add all HotSwapCapable OrbitModules to the OrbitNetwork when the TCON that the Orbit 3 Module is attached to has valid entry in memory for the OrbitModule. The function returns the number of new modules added to the network.

C# (.NET)

```
int modulesFound = Orbit.Networks.[NETINDEX].Modules.FindHotSwapped();
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->FindHotSwapped();
```

This command allows an operator to physically hot-swap one Orbit 3 Module for another, then call FindHotSwapped() to rectify any differences in software.
The Orbit 3 Modules must be of the same device type and have the same stroke to qualify as valid entry in a TCON's memory.

When disconnecting from the OrbitServer, TCON memory is not cleared, thus FindHotswapped() can be used once an application reconnects to the OrbitServer to recover the previously addressed OrbitModules.

10.4.6 Delete Module

To remove a specific OrbitModule from an OrbitNetwork, the OrbitModule's index number in the OrbitNetwork.Modules collection can be specified, or the OrbitModule's ModuleID string can be specified. Either variable being passed as parameters to OrbitNetwork.Modules.DeleteModule() will result in the OrbitModule being removed from the OrbitNetwork.

C# (.NET)

```
Orbit.Networks.[NETINDEX].Modules.DeleteModule(MODULEINDEX);
// Or
Orbit.Networks.[NETINDEX].Modules.DeleteModule("100C619P19");
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->DeleteModule(MODULEINDEX);
// Or
OrbNet->Modules->DeleteModule(WideString("100C619P19"));
```

Calling DeleteModule will also clear the OrbitModule's TCON memory. Deleting a module will re-order the indexes, to maintain contiguous addresses. For example, if there are modules set at indexes: 0,1,2,3,4,5,6 and module 4 was deleted, then modules at indexes 5 & 6 would be shifted down by one address to fill the gap.

10.4.7 Clear All Modules

To remove all OrbitModules from an OrbitNetwork's Modules collection, a call to OrbitNetwork.Modules.ClearModules() is required.

C# (.NET)

```
Orbit.Networks.[NETINDEX].Modules.ClearModules();
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->ClearModules();
```

This function will clear the OrbitNetwork of OrbitModules, but leave TCON memory intact, allowing FindHotSwapped() to find HotSwapCapable OrbitModules still attached to the OrbitNetwork.

10.4.8 Clear TCON Memory

OrbitNetwork.Modules.ClearTcons() shall remove all OrbitModules from an OrbitNetwork, as well as clearing the memory on all TCONs attached to the BUS that have an Orbit 3 Module attached.

C# (.NET)
```
Orbit.Networks.[NETINDEX].Modules.ClearTCons();
```

C++ (COM)
```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->ClearTCons();
```

By clearing the TCON memory, finding HotSwapCapable OrbitModules is no longer possible.

10.4.9 Change Address

To change an OrbitModule's address, which is represented by the OrbitModule's index within OrbitNetwork.Modules, setting OrbitModule.RemapIndex to the desired (new) index. This should be carried out on another module (e.g. if two modules indexes are to be swapped).
Once the remapping of addresses and indexes is configured, OrbitNetwork.Modules.ApplyRemap() should be called to carry out the modifications.

C# (.NET)
```
Orbit.Networks.[NETINDEX].Modules[FromModuleIndex].RemapIndex = ToModuleIndex;
Orbit.Networks.[NETINDEX].Modules[ToModuleIndex].RemapIndex = FromModuleIndex;
Orbit.Networks.[NETINDEX].Modules.ApplyRemap();
```

C++ (COM)
```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbNet->Modules->get_Item(FromModuleIndex)->RemapIndex = ToModuleIndex;
OrbNet->Modules->get_Item(FromModuleIndex)->RemapIndex = FromModuleIndex;
OrbNet->Modules->ApplyRemap();
```

This is can be a useful tool when modules are desired to be set-up in a certain order. Once the correct order has been established, the network can be saved using the OrbitNetwork.Save method. This saves the network's modules to an XML file that can be loaded next time by the OrbitNetwork.Load method.

10.4.10 Load and Save Network

Any orbit network can be loaded/saved with the OrbitNetwork.Save and OrbitNetwork.Load Commands.

To Save an orbit network after it has been setup just call save providing a filename that the network information should be saved to, eg:

C# (.NET)
```
MyNetwork.Save(FileName);
```

C++ (COM)
```
MyNetwork->Save(FileName);
```

To Load an orbit network there are 2 options
1)just load the file - if one or more modules is loaded it will return true, otherwise it will return false; note that the module count should be checked to ensure you have all the modules.

```
C# (.NET)
bool ModulesLoaded = MyNetwork.Load(FileName);
```

```
C++(COM)
bool ModulesLoaded = MyNetwork->Load(FileName);
```

2)Load a network with creating a mappings array where the mappings array lists the original module address (the array index) to the current address (array value) with -1 used for the current address of missing items.
C# (.NET)
int[] ModuleIndexMappings;
MyNetwork.Load(FileName, out ModuleIndexMappings);

C++ (COM)
LPSAFEARRAY Mappings;
TOLEBOOL Success = OrbitNet0->Load_2(FileName, &Mappings);

10.5 GETTING MODULE READINGS

There are two calls that are made available to receive a single reading from an OrbitModule, OrbitModule.ReadingInCounts, and OrbitModule.ReadingInUnits. The former returning a reading in Counts, which is a representation of the full scale ($2^{resolution}$ for DP / AIM) of a module. The latter, ReadingInUnits will the return a reading in the base units of measure (e.g. millimetres); of type double. (see OrbitModule.ReadingInUnits property in the *OrbitLibrary Code Reference*.

```
C# (.NET)
        int readingInCounts   = Orbit.Networks.[NETINDEX].Modules[MODULEINDEX].ReadingInCounts;
        double readingInUnits = Orbit.Networks.[NETINDEX].Modules[MODULEINDEX].ReadingInUnits;
```

```
C++ (COM)
        OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
        OrbitModulePtr OrbModule = OrbNet->Modules->get_Item(ModuleIndex);
        int readingInCounts   = OrbModule->ReadingInCounts;
        double readingInUnits = OrbModule->ReadingInUnits;
```

For taking synchronised blocks of readings for an OrbitNetwork's OrbitModules, see ReadBurst. For recording a collection of synchronised readings from any number of modules for processing after the collection has been completed, see Dynamic mode.

When retrieving module readings with the Orbit Library, any hardware errors (e.g. Overspeed Error), generate an exception (see Orbit Error Codes and Error Handling). Under and Over range are dealt with separately (see Module Status).

10.5.1 Configuring Modules

Due to the object orientated nature of the Orbit Library, it is necessary to cast to the modules class to access module specific properties (such as Resolution and Averaging on the DP).

```
C# (.NET)
        OrbitModuleDP ModuleDP = ((OrbitModuleDP)MyNetwork.Modules[ModuleIndex]);
        ModuleDP.Resolution = eResolution.res18Bit;
```

```
C++ (COM)
        OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item( NETINDEX);
        OrbitModulesPtr OrbModules = OrbNet->Modules;
        OrbitModuleDPPtr ModuleDP = (OrbitModuleDPPtr)OrbModules->get_Item(ModuleIndex);
```

```
ModuleDP->Resolution = eResolution_res18Bit;
```

10.5.2 Module Status

With each reading made, OrbitModule.ModuleStatus is updated. Checking this status is important when working with displacement probes (DP/AIM/AGM) which can go outside of their calibrated ranges. When outside of the range, an OrbitModule shall report that it is either UnderRange or OverRange respectfully, represented by OrbitModule.ModuleStatus.Error.

C# (.NET)

```
string moduleID       = Orbit.Networks.[NETINDEX].Modules[MODULEINDEX].ModuleID;
double readingInCounts = Orbit.Networks.[NETINDEX].Modules[MODULEINDEX].ReadingInCounts;
eOrbitErrors ErrorState    = Orbit.Networks.[NETINDEX].Modules[MODULEINDEX]
        .ModuleStatus.Error;
if( ErrorState == eOrbitErrors.OverRange || ErrorState == eOrbitErrors.UnderRange)
{
        Console.WriteLine( moduleID + " Error: " + ErrorState.ToString());
} else
{
        Console.WriteLine( moduleID + " Reading: " + readingInCounts.ToString());
}
```

C++ (COM)

```
OrbitNetworkPtr OrbNet   = Orbit->Networks->get_Item(NetIndex);
OrbitModulePtr OrbModule = OrbNet->Modules->get_Item(ModuleIndex);
AnsiString moduleID      = AnsiString(OrbModule->ModuleID);
double readingInCounts   = OrbModule->ReadingInCounts;
eOrbitErrors ErrorState  = OrbModule->ModuleStatus->Error;
if( ErrorState == eOrbitErrors.OverRange || ErrorState == eOrbitErrors.UnderRange)
{
        AnsiString ErrorString = AnsiString(OrbModule->ModuleStatus->ErrorString);
        cout << moduleID.c_str() << " Error: " << ErrorString.c_str() << endl;
} else
{
        cout << moduleID.c_str() << " Reading: " << readingInCounts << endl;
}
```

Other errors, such as a linear encoder's over speed, or an encoder input module's framing error will remain set until the OrbitModule's status is reset by a call to UpdateStatus().

C# (.NET)

```
Orbit.Networks.[NETINDEX].Modules[MODULEINDEX].ModuleStatus.UpdateStatus();
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbitModulePtr OrbModule = OrbNet->Modules->get_Item(ModuleIndex);
OrbModule->ModuleStatus->UpdateStatus();
```

10.6 READING MODES

The following sub-sections describe the methods implementation for the reading modes provided by the Orbit Library. Each of these modes are designed for use in differing scenarios and situations; a comprehensive guide of when and where to use these modes can be found in the Measurement Modes section.
Note that iterating through an OrbitNetwork.Modules collection and taking a reading with getReadingInUnits/Counts on each module in quick succession is **not** a recommended or precise method of taking synchronised readings.

10.6.1 ReadBurst Mode

The ReadBurst OrbitNetwork command provides an easy, precise and optimised method of returning a collection of synchronised readings from a group of ReadBurstCapable OrbitModules. (see ReadBurst)

A call to OrbitNetwork.ReadBurst() will take a synchronised read block of the capable OrbitModules.
These can be accessed through the OrbitModule.Modules.ReadBurstData object.
The error state of ReadBurst call can accessed though ReadBurstData.Error.

C# (.NET)

```
Orbit.Networks[NETINDEX].Modules.ReadBurst();
OrbitReadBurstResults ReadResults = Orbit.Networks[NETINDEX].Modules.ReadBurstData;
// List Results
for (int Index = 0; Index < ReadResults.NumberOfModules; Index++)
{
        Console.WriteLine(Orbit.Networks[NETINDEX].Modules[Index].ModuleID +
                "\t" + string.Format("{0:0.000000}", ReadResults.GetReadingUOM(Index)) +
                "\t\t" + Orbit.GetErrorString((int)ReadResults.GetReadingError(Index)) +
                "\r\n");//Error status
}
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item(NetIndex);
OrbitModulesPtr OrbModules = OrbNet->Modules;
OrbModules->ReadBurst();
OrbitReadBurstResultsPtr ReadResults = OrbModules->ReadBurstData;
// List Results
for (int Index = 0; Index < ReadResults->NumberOfModules; Index++)
{
        OrbitModulePtr OrbModule = OrbModules->get_Item(Index);
        double ReadingInUOM = ReadResults->GetReadingUOM(Index);
        int ReadingInCounts = ReadResults->GetReadingCounts(Index);
        int ReadingError = ReadResults->GetReadingError(Index);
        cout << AnsiString(OrbModule->ModuleID).c_str() << "\t"
                << AnsiString().sprintf("%+9.5f",ReadingInUOM).c_str() << "\t\t"
                <<  AnsiString().sprintf("%+8i", ReadingInCounts).c_str() << "\t\t"
                << AnsiString(Orbit->GetErrorString( ReadingError)).c_str() << endl;
}
```

The ReadBurstCapable OrbitModules must be contiguous from the first address on the OrbitNetwork. A non-capable OrbitModule that is addressed between two capable OrbitModules will break contiguity, cutting short the number of modules the ReadBurst collection can collect from.

10.6.2 Dynamic Modes 1 & 2

A Dynamic mode collection provides a fast and precise method to collect large numbers of synchronised readings for post processing once a collection is complete (see Dynamic Modes).

All OrbitModules connected to an OrbitNetwork must be capable of Dynamic or Dynamic 2, represented by the OrbitModule fields DynamicCapable and Dynamic2Capable. The OrbitNetwork's DynamicCapable and Dynamic2Capable statuses represent the overall capability of the network, taking into account the capabilities of all connected OrbitModules.

Dynamic mode requires that OrbitNetwork.NetSpeed is set to eNetSpeed.High. Dynamic 2 can operate at either High or UltraHigh speeds.

A Dynamic collection consists of five stages:

- Unconfigured (dynamic disabled)
- Configured (dynamic enabled)
- Prepared
- Running (collecting data)
- Complete (data available).

The Dynamic.Enabled property represents an OrbitNetwork's dynamic enabled status. When Enabled is false, dynamic configuration settings can be modified. Upon changing Enable to *true*, the configuration settings are checked against OrbitNetwork and OrbitModule capabilities and validated; if the current dynamic settings are found to be correct. These configuration settings can no longer be modified; if further modification is required, Enabled should be set back to *false*.

With Dynamic mode now enabled, OrbitNetwork.Dynamic.Prepare() should be called to setup the Dynamic collection. Once prepared, the OrbitNetwork is ready start collecting. OrbitNetwork.Dynamic.DynamicState will have changed from eDynamicState.Inactive, to Prepared.

C# (.NET)

```
// Configure Dynamic Collection (see Examples support pack for examples)
Orbit.Networks[NETINDEX].Dynamic.Enabled = true;
Console.WriteLine("DynamicState before prepare: " +
        Orbit.Networks[NETINDEX].Dynamic.DynamicState.ToString());
Orbit.Networks[NETINDEX].Dynamic.Prepare();
Console.WriteLine("DynamicState after prepare: " +
        Orbit.Networks[NETINDEX].Dynamic.DynamicState.ToString());
```

The server command, Orbit.StartAllDynamic(), starts dynamic mode for all enabled and prepared OrbitNetworks. Upon starting the collection, an OrbitNetwork shall set its DynamicState to Running.

When a dynamic collection is running, Orbit.StopAllDynamic() is available to stop all the dynamic collections in progress (i.e. any OrbitNetwork with a DynamicState of Running). This call is needed to stop a dynamic collection that has been configured with the Dynamic.CollectionSize set to '0': unlimited syncs.
When a collection ends, the DynamicState is set to Complete and collected data is made available.

C# (.NET)

```
// Configure Dynamic
// Enable Dynamic
// Prepare Dynamic
Orbit.StartAllDynamic();
// Give dynamic a chance to read
System.Threading.Thread.Sleep(1000);
Orbit.StopAllDynamic();
// Print number of readings
Console.WriteLine(Orbit.Networks[NETINDEX].Dynamic.DynamicData.ReadingCount.ToString());
```

If CollectionSize has been configured to a non-zero positive number, then the collection will complete once the correct number of syncs has been reached or if StopAllDynamic() is called. In the case of the CollectionSize being reached, the collection shall stop, the DynamicState set to Complete and an Event kicked off to signal that the collection is complete.

C# (.NET)

```
private void StartDynamic()
{
        Orbit.DynamicComplete += new d_DynamicComplete(Orbit_DynamicComplete);
        Orbit.Networks[NETINDEX].Dynamic.CollectionSize = 100000;
        // Configure Dynamic
        // Enable Dynamic
        // Prepare Dynamic
        Orbit.StartAllDynamic();
}
private void Orbit_DynamicComplete()
{
        Console.WriteLine("Dynamic collection complete.");
}
```

C++ (COM)

```
int NumberOfModules;
int NumberOfReadings = 100;
eDynamicRate ReadRate = eDynamicRate_Dynamic2Custom;
double ReadingInUOM;
OrbitNet0 = Orbit->Networks->get_Item(0);
OrbitDynamicPtr NetworkDynamic = OrbitNet0->Dynamic;
OrbitModulesPtr OrbitModules0 = OrbitNet0->Modules;
OrbitNet0->NetSpeed = eNetSpeed_UltraHigh;
NumberOfModules = OrbitModules0->Count;
NetworkDynamic->Enabled = false;
NetworkDynamic->NumberOfModules = NumberOfModules;
NetworkDynamic->DynamicRate = ReadRate;
NetworkDynamic->CollectionSize = NumberOfReadings;
NetworkDynamic->Enabled = true;
cout << "Dynamic State: " << NetworkDynamic->DynamicState;
NetworkDynamic->Prepare();
cout << "Dynamic State: " << NetworkDynamic->DynamicState;
Orbit->StartAllDynamic();
cout << "Dynamic State: " << NetworkDynamic->DynamicState;
while( Orbit->DynamicInProgress == true){Sleep(10);}
cout << "Dynamic State: " << NetworkDynamic->DynamicState;
AnsiString ErrorStatus =  AnsiString(Orbit->GetErrorString(
   NetworkDynamic->DynamicData->CollectionStatus));
cout <<"Collection Error status: " << ErrorStatus.c_str();
int NumberOfReads = NetworkDynamic->DynamicData->ReadingCount;
//Print the Results
AnsiString RowString;
for (int LoopReads =0; LoopReads < NumberOfReads; LoopReads++)
{
        RowString=AnsiString().sprintf("\t%i\t\t", LoopReads);
        for (int LoopModules = 0; LoopModules < NumberOfModules; LoopModules++)
        {
                ReadingInUOM =  NetworkDynamic->DynamicData->get_Item(LoopModules,LoopReads);
                RowString    += AnsiString().sprintf( "%+9.5f\t",ReadingInUOM);
        }
        cout <<RowString.c_str();
}
NetworkDynamic->Enabled = false;
```

To check a collection's error state see the Dynamic.DynamicData.CollectionStatus property.

If a collection completes without errors, the collected data is made available. This data is stored under Dynamic.DynamicData[ModuleIndex, ReadingIndex] for readings in units of measure and Dynamic.DynamicData.GetReadingInCounts(ModuleIndex, ReandingIndex) for counts.

C# (.NET)

```
// Configure Dynamic
// Enable Dynamic
// Prepare Dynamic
// Start Dynamic Collection
// Stop/Finish Collection
int ReadCount   = Orbit.Networks[NETINDEX].Dynamic.DynamicData.ReadingCount;
int ModuleCount = Orbit.Networks[NETINDEX].Dynamic.DynamicData.ModuleCount;
OrbitDynamicData DynData = Orbit.Networks[NETINDEX].Dynamic.DynamicData;
for (int readBlockIndex = 0; readBlockIndex < ReadCount; readBlockIndex++)
{
        Console.WriteLine("\t" + readBlockIndex + "\t");
        for (int moduleIndex = 0; moduleIndex < ModuleCount; moduleIndex++)
        {
                Console.Write(string.Format("{0:0.000000}", DynData[moduleIndex, readBlockIndex])
                        + "\t\t");
        }
        Console.Write("\n");
}
```

10.6.2.1 Dynamic External Master Mode

Dynamic mode can also be configured to have an external master (by default syncs are triggered by the timing rates from the Orbit Controller), but when an OrbitNetwork's Dynamic.DynamicMode is set from eDynamicMode.Normal to External, triggering of syncs is handled by the Dynamic.MasterAddress, usually in the form of an encoder input module. The MasterAddress is set by OrbitModule index, and should therefore be the last addressed OrbitModule in the dynamic collection.

There are a number of modes that can be set for the MasterAddress that determine the device's behaviour.
OrbitModule.TxSync property defines the number of counts the MasterAddress must read until the MasterAddress triggers a sync; all OrbitModules included within the Dynamic collection shall then record a reading. Results are accessed using the same method as eDynamicMode.Normal.
See Dynamic External Master Mode
The following code details setting up dynamic master for an EIM:

C# (.NET)

```
// Configure Dynamic
Orbit.Networks[NETINDEX].Dynamic.MasterAddress = EimIndex;
((OrbitModuleEIM)Orbit.Networks[NETINDEX].Modules[Orbit.Networks
        [NETINDEX].Dynamic.MasterAddress]).DynamicMasterMode =
        eEncoderDynamicMasterMode.HoldOff;
((OrbitModuleEIM)Orbit.Networks[NETINDEX].Modules[Orbit.Networks
        [NETINDEX].Dynamic.MasterAddress]).TxSync = (Int16)100;
((OrbitModuleEIM)Orbit.Networks[NETINDEX].Modules[Orbit.Networks
        [NETINDEX].Dynamic.MasterAddress]).HoldOff = (Int16)500;
// Configure Dynamic
// Enable Dynamic
// Prepare Dynamic
// Start Dynamic Collection
// **Turn the EIM to trigger reads**
// Stop/Finish Collection
// Process Results
```

10.6.3 Buffered Mode

See Buffered Mode section for more details about this mode.
Buffered mode allows OrbitModules to be individually enabled in any order by modifying OrbitModule.Buffered.Enable to *true.*

Once the OrbitNetwork is configured, Buffered mode enabled OrbitModules can be started at the same time with the OrbitNetwork.Buffered.Start() command. The enabled OrbitModules shall then start to take readings at their set intervals (see ModuleBufferedBase.OTUs in the *OrbitLibrary Code Reference*, and buffer them locally on the actual Orbit 3 Module hardware (up to 3000 readings on each Orbit 3 Module).

When Buffered.Stop() is called, each OrbitModule shall stop buffering reads. These readings are then automatically read back by the Orbit /library and can then be retrieved using the OrbitNetwork.Buffered.BufferedData object reference.

Buffered mode has two sub-modes of operation - readings triggered by time intervals; SyncMode, and readings triggered by an external source; SampleMode.

In sync mode, the time interval, Buffered.ReadingInterval, can be configured. Each OrbitModule will then begin to fill a local buffer at the locally set ReadingInterval. Sample mode syncs are triggered from the controller, thus OrbitModules in sample mode will take syncs at the same time.

In the example, next, two modules are set into buffered mode: one to sync mode (reading interval to 0.1 seconds) and the other to sample.
Buffered start and stop are approximately 1 second apart (via sleeps).
Therefore, the sync mode module would have taken 10 readings (0.1s apart) and the sample mode module would have 5 taken readings (via calls to the Sample method).

C# (.NET)

```
int moduleOne = 0;
int moduleTwo = 1;
//set module1 to sync mode, interval every 100000 us = 0.1 seconds
Orbit.Networks[NETINDEX].Modules[moduleOne].Buffered.Mode = eBufferedMode.Sync;
Orbit.Networks[NETINDEX].Modules[moduleOne].Buffered.ReadingInterval = 100000;
Orbit.Networks[NETINDEX].Modules[moduleOne].Buffered.Enable = true;
//set module2 to sample mode
Orbit.Networks[NETINDEX].Modules[moduleTwo].Buffered.Mode = eBufferedMode.Sample;
Orbit.Networks[NETINDEX].Modules[moduleTwo].Buffered.Enable = true;
//start buffered mode...
Orbit.Networks[NETINDEX].Buffered.Start();
Orbit.Networks[NETINDEX].Buffered.Sample();
Orbit.Networks[NETINDEX].Buffered.Sample();
System.Threading.Thread.Sleep(100);
Orbit.Networks[NETINDEX].Buffered.Sample();
System.Threading.Thread.Sleep(200);
Orbit.Networks[NETINDEX].Buffered.Sample();
System.Threading.Thread.Sleep(700);
Orbit.Networks[NETINDEX].Buffered.Sample();
//Stop buffered mode. The time that buffered mode was approx 1 second. This means:
// Sync Mode: 10 readings taken
// Sample Mode: 5 samples were taken
Orbit.Networks[NETINDEX].Buffered.Stop();
//Look at the results. Write them to console.
Console.WriteLine("Module One Results:");
for (int index = 0; index <
        Orbit.Networks[NETINDEX].Modules[moduleOne].Buffered.BufferedData.Length; index++)
{
        Console.WriteLine("\t" + index + "\t" +
Orbit.Networks[NETINDEX].Modules[moduleOne].Buffered.BufferedData[index].Reading.ToString());
}
Console.WriteLine("\nModule Two Results:");
```

```
for (int index = 0; index <
        Orbit.Networks[NETINDEX].Modules[moduleTwo].Buffered.BufferedData.Length; index++)
{
        Console.WriteLine("\t" + index + "\t" +
Orbit.Networks[NETINDEX].Modules[moduleTwo].Buffered.BufferedData[index].Reading.ToString());
}
```

C++ (COM)

```
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item( NETINDEX);
OrbitModulesPtr OrbModules = OrbNet->Modules;
OrbitModulePtr Module0 = OrbModules->get_Item(0);
OrbitModulePtr Module1 = OrbModules->get_Item(1);
Module0->Buffered->Mode = eBufferedMode.Sync;
Module0->Buffered->ReadingInterval = 100000;
Module0->Buffered->Enable = true;
Module1->Buffered->Mode = eBufferedMode.Sample;
Module1->Buffered->Enable = true;
OrbNet->Buffered->Start();
OrbNet->Buffered->Sample();
OrbNet->Buffered->Sample();
Sleep(100);
OrbNet->Buffered->Sample();
Sleep(200);
OrbNet->Buffered->Sample();
Sleep(700);
OrbNet->Buffered->Sample();
OrbNet->Buffered->Stop();
cout <<"Module One Results:" <<endl;
int NumReads = Module0->Buffered->BufferedData->Length;
for (int index = 0; index < NumReads; index++)
{
        cout <<"\t" << index << "\t" +
                Module0->Buffered->BufferedData->get_Item(index) <<endl;
}
cout <<endl << "Module Two Results:";
NumReads = Module1->Buffered->BufferedData->Length;
for (int index = 0; index < NumReads; index++)
{
        cout <<"\t" << index << "\t" +
                Module1->Buffered->BufferedData->get_Item(index) <<endl;
}
```

By default, the controller within an OrbitNetwork is deemed the master: responsible for sending sample commands. Buffered mode can be configured with an external master; usually in the form of an encoder input module. The MasterAddress is set by its OrbitModule index; setting to *-1* configures the controller as the master. The number of counts before triggering a sample command can be set with ModuleEIM.TxSample. This property works in a similar way to ModuleEIM.TxSync (used for dynamic mode).

C# (.NET)

```
// Configure and enable OrbitModules.  Should be in Sample mode.
// Shall read every 100 EIM counts once Buffered started.
((OrbitModuleEIM)MyNetwork.Modules[EimIndex]).TxSample = 100;
MyNetwork.Buffered.MasterAddress = EimIndex;
// Start Buffered mode
// Rotate EIM
// Stop Buffered mode
// Get results
```

C++ (COM)

```
// Configure and enable OrbitModules.  Should be in Sample mode.
OrbitNetworkPtr OrbNet = Orbit->Networks->get_Item( NETINDEX);
OrbitModulesPtr OrbModules = OrbNet->Modules;
OrbitModuleEIMPtr ModuleEIM = (OrbitModuleEIMPtr)OrbModules->get_Item(EimIndex);
// Shall read every 100 EIM counts once Buffered started.
ModuleEIM->TxSample = 100;
OrbNet->Buffered->MasterAddress = EimIndex;
// Start Buffered mode
// Rotate EIM
```

```
        // Stop Buffered mode
        // Get results
```

10.6.4 Difference Mode

If DifferenceModeCapable, an OrbitModule may be enabled into Difference mode by setting Orbit.Module.Difference.Enable to *true*.

Enabled Difference mode OrbitModules can then be started into reading in Difference mode with a call to OrbitNetwork.Difference.Start(). Results are read with OrbitNetwork.Buffered.ReadDifference(). ReadDifference exposes the following data:

- Minimum reading in units and counts.
- Maximum reading in units and counts.
- Difference between the max and min, in units and counts.
- The sum of all the readings taken (DP/AIM/AGM).
- The number of reads taken (DP/AIM/AGM).
- The error state.

Difference mode can then be stopped with OrbitNetwork.Difference.Stop(). *A detailed example can be found in the* Orbit Library Test *example in DiffModeUC.cs.*

10.6.5 RefMark Mode

Provided for the Linear Encoder range, RefMark mode provides functionality for setting a datum from the linear encoder's reference mark; this datum is lost when the Module powers down, thus must be resettable upon powering up.
The Orbit Library Test example project includes an example implementation of reference mark mode in RefMarkUC.cs.

11 ORBIT LIBRARY TEST

11.1 INTRODUCTION

The Orbit Library Test program has been developed as a functional example to help ease users into working with the Orbit Library. Included is clearly commented source code written to provide reference, as well as simple, clean examples of interfacing to an Orbit3 Measurement System. The Library Test also operates as a suitable tool for analysing and setting up an Orbit3 system.

The Library Test is designed in a modular manner allowing users to quickly pull code straight from the Library Test and integrate into existing user developed applications.

This document provides an overview of the Library Test's features and usage, an examination of the structure of the source code and a partial walk-through of recommended development tools.

11.2 FEATURES

The Library Test features the following functionality:

- Orbit Server Interfacing
  - Connecting
  - Disconnecting
  - Accessing Networks
  - Version Information

- Orbit Network Interfacing
  - Network Information
  - Network Capabilities Listing
  - Accessing Modules
  - Network Management
    - Add Module
    - Notify Add
    - Find Hot swapped
    - Ping
    - Remove Module
    - Clear Modules
    - Clear TCONs
  - Saving and Loading XML Network Configurations
  - Network Speed Configuration
  - Laser Beam Control

- Orbit Module Interfacing
  - Module Information
  - Module Capabilities Listing
  - Traditional Reading
    - Units of Measure
    - Counts
  - Module Type Specific Configuration
    - DP
    - LT / LTA
    - LTH
    - AIM
    - DIM
    - DIOM
    - DIOM2
    - EIM
    - LE
    - Confocal
  - Get Modules Status
  - Reset Module Status

- Reading Mode Configuration
  - Read Burst Mode
  - Dynamic Mode
    - Dynamic External Master
    - Dynamic 2
  - Difference Mode
  - Reference Mark Mode
  - Buffered Mode
    - Buffered External Master

11.3 USER GUIDE

The following guide defines the different areas of the application and explains their usage.

11.3.1 Getting Started

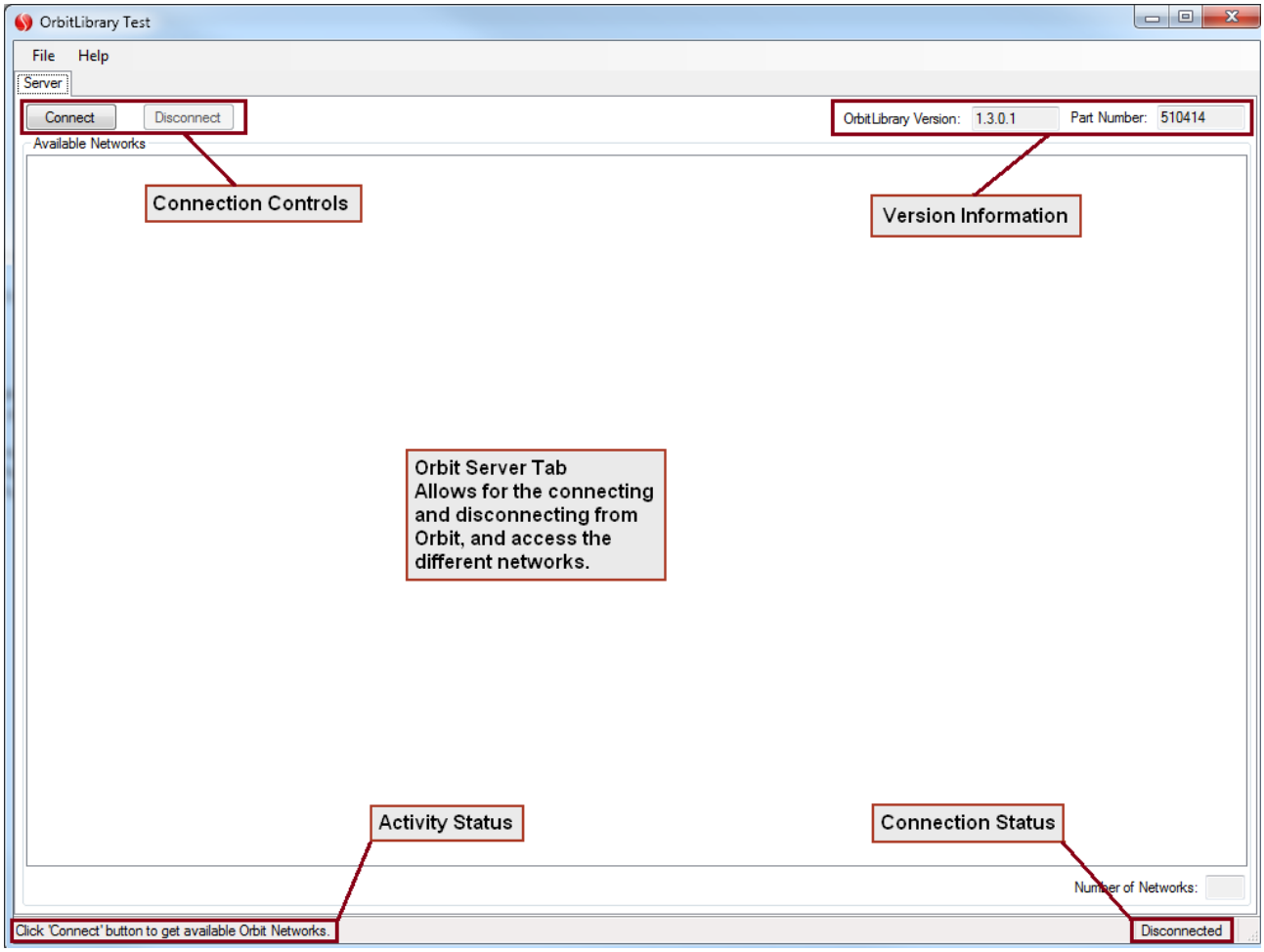
If the Orbit3 Support Pack for Windows is not yet installed, please see the relevant section in the Orbit3 System manual. Once installed, the Library Test's compiled binaries and source code can be found under then *OrbitLibraryTest*\ folder in the Orbit3 Support Pack for Windows installation directory: i.e.
C:\Program Files\Solartron Metrology\Orbit3 Support Pack for Windows

Ensure there are no other applications running which are connected to Orbit and run *OrbitLibraryTest.exe* to start the application.

11.3.2 Usage

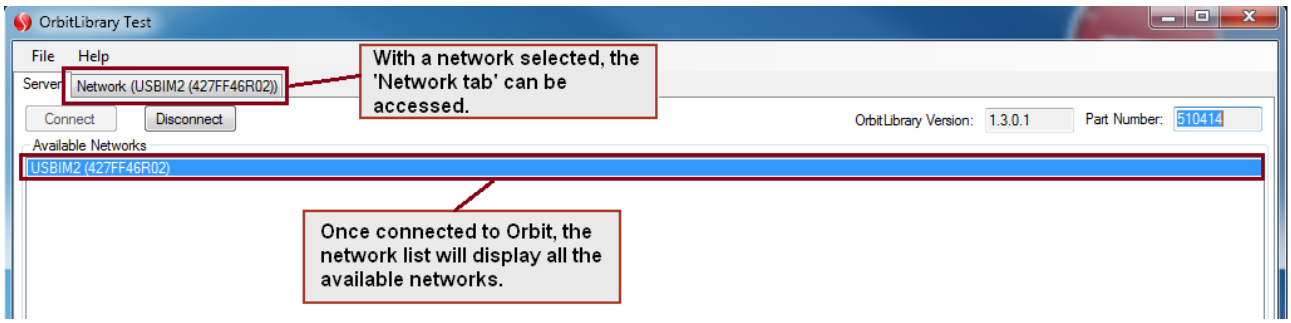
The screen shots displayed below illustrate and explain the different areas and functions of user interface elements.

11.3.2.1 Server Tab

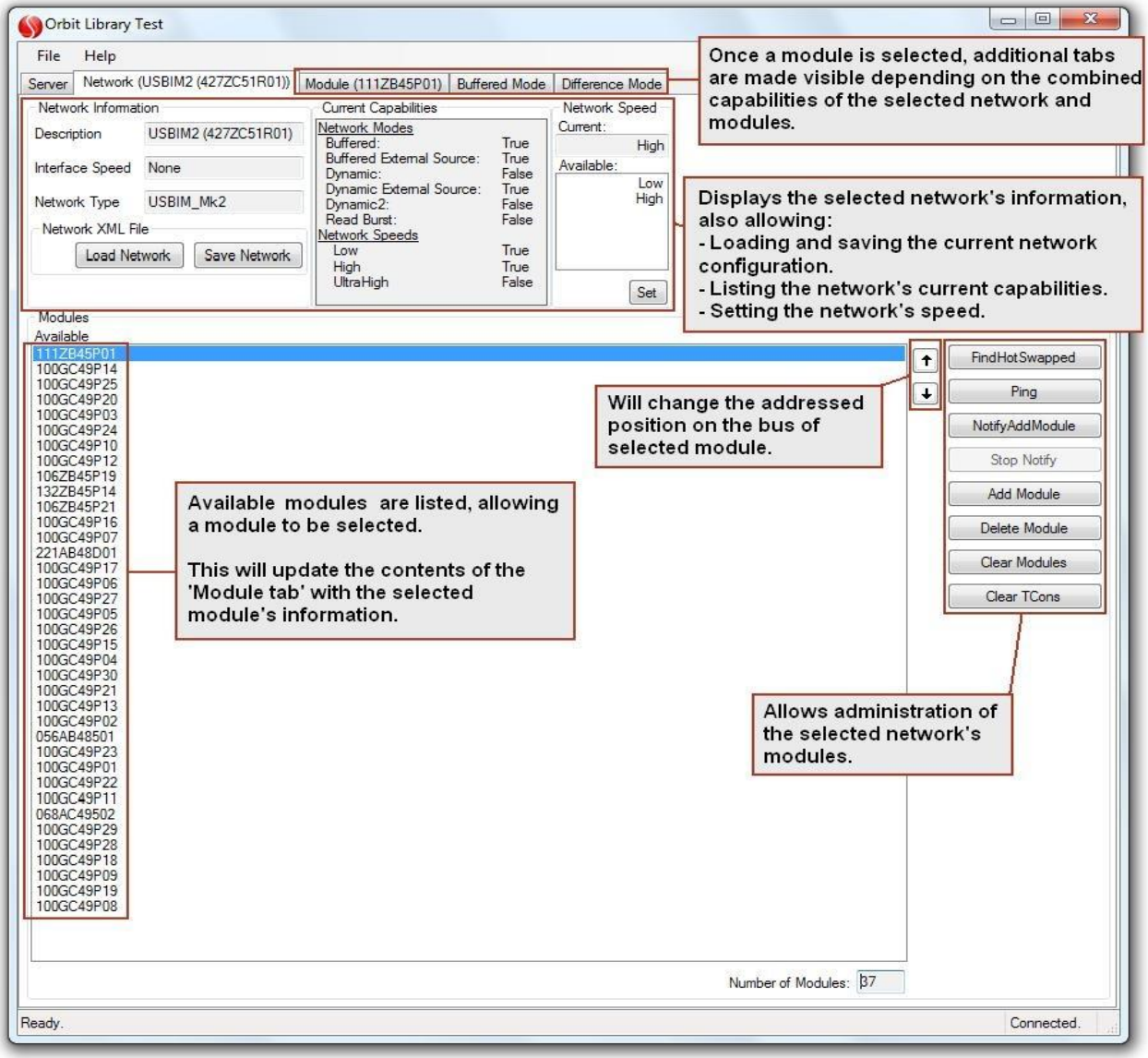


The status bar will be updated when the Library Test is performing a task.

Once connected to Orbit:

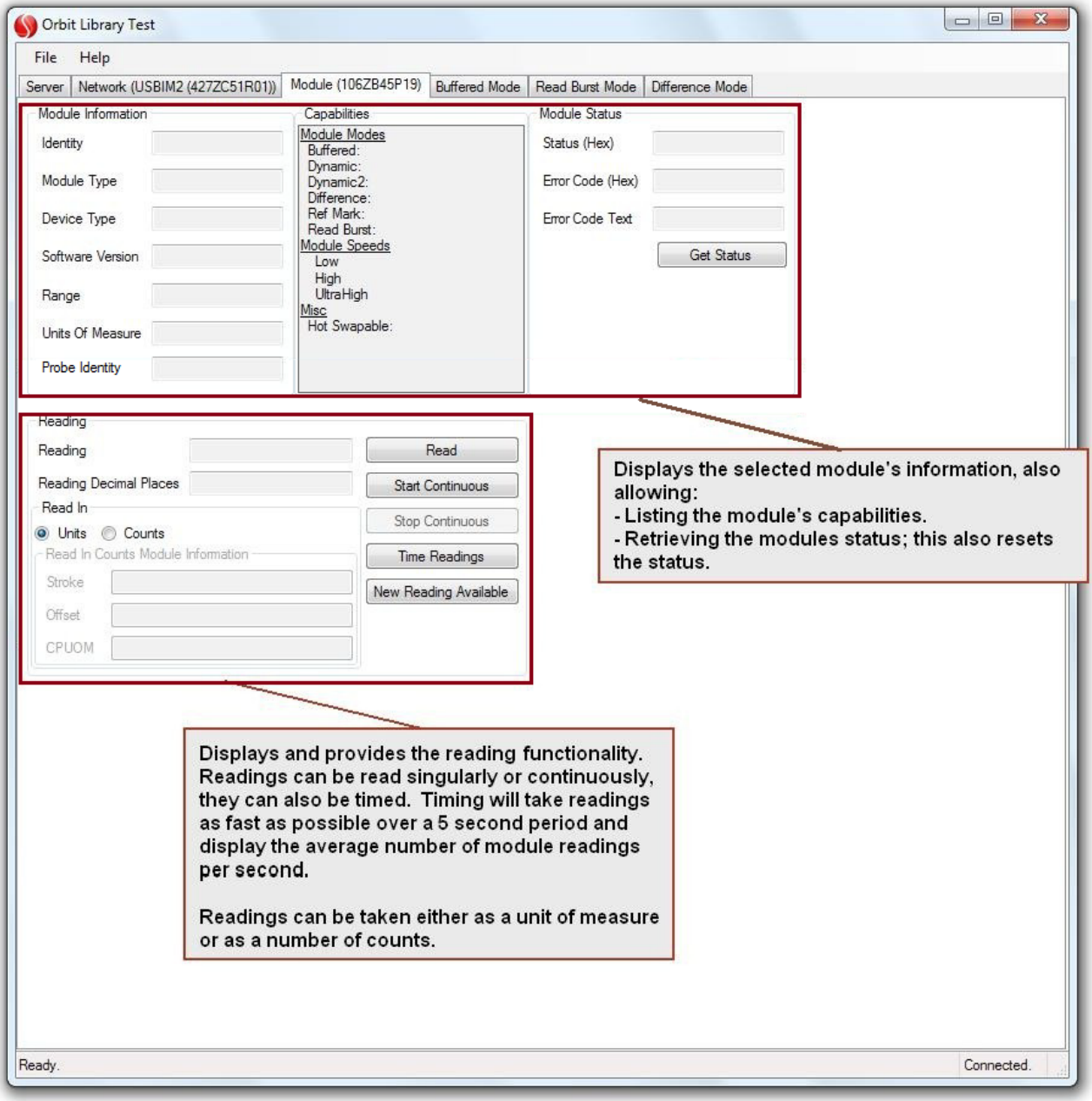


11.3.2.2 Network Tab



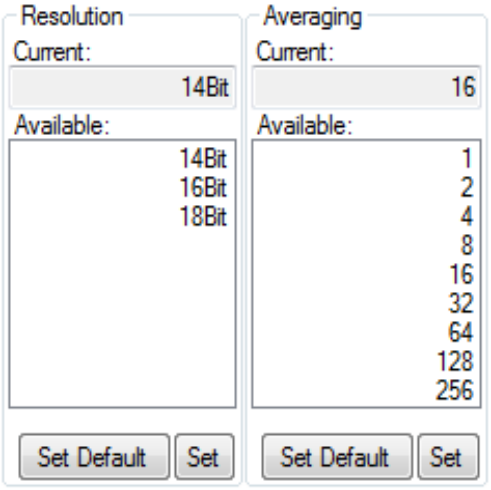
11.3.2.3 Module Tab

The following displays the module tab with only the generic controls visible.


Orbit Library Test
File
Help
Server
Network (USBIM2 (427ZC51R01))
Module (106ZB45P19)
Buffered Mode
Read Burst Mode
Difference Mode
Module Information
Identity
Module Type
Device Type
Software Version
Range
Units Of Measure
Probe Identity
Capabilities
Module Modes
Buffered:
Dynamic:
Dynamic2:
Difference:
Ref Mark:
Read Burst:
Module Speeds
Low
High
UltraHigh
Misc
Hot Swapable:
Module Status
Status (Hex)
Error Code (Hex)
Error Code Text
Get Status
Reading
Reading
Reading Decimal Places
Read In
Units
Counts
Read In Counts Module Information
Stroke
Offset
CPUOM
Read
Start Continuous
Stop Continuous
Time Readings
New Reading Available
Displays the selected module's information, also allowing:
- Listing the module's capabilities.
- Retrieving the modules status; this also resets the status.
Displays and provides the reading functionality. Readings can be read singularly or continuously, they can also be timed. Timing will take readings as fast as possible over a 5 second period and display the average number of module readings per second.
Readings can be taken either as a unit of measure or as a number of counts.
Ready.
Connected.

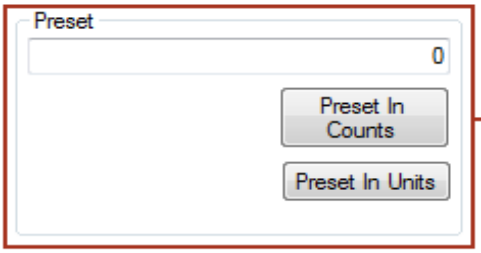
Differing module types have specialist configuration options. These options are only visible when a module of the correct type is selected.

11.3.2.3.1 Resolution and Averaging Configuration



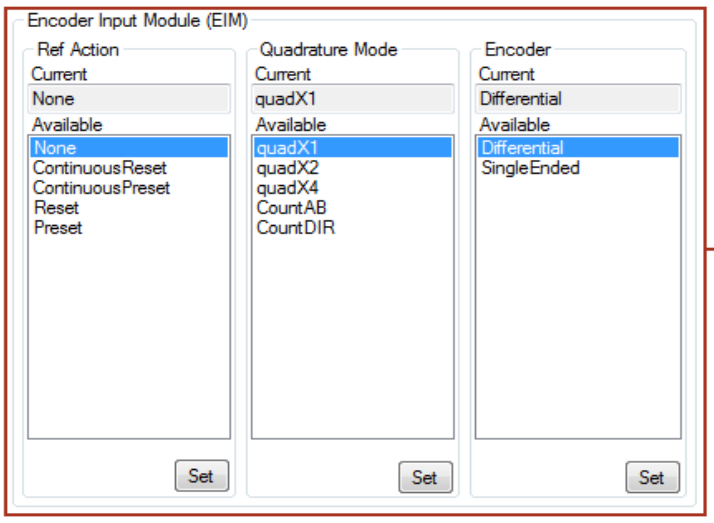
Only available for DPs, AIMs and LT/LTA/LTH laser products; allows configuration of a module's resolution and averaging.

11.3.2.3.2 Pre-set Configuration



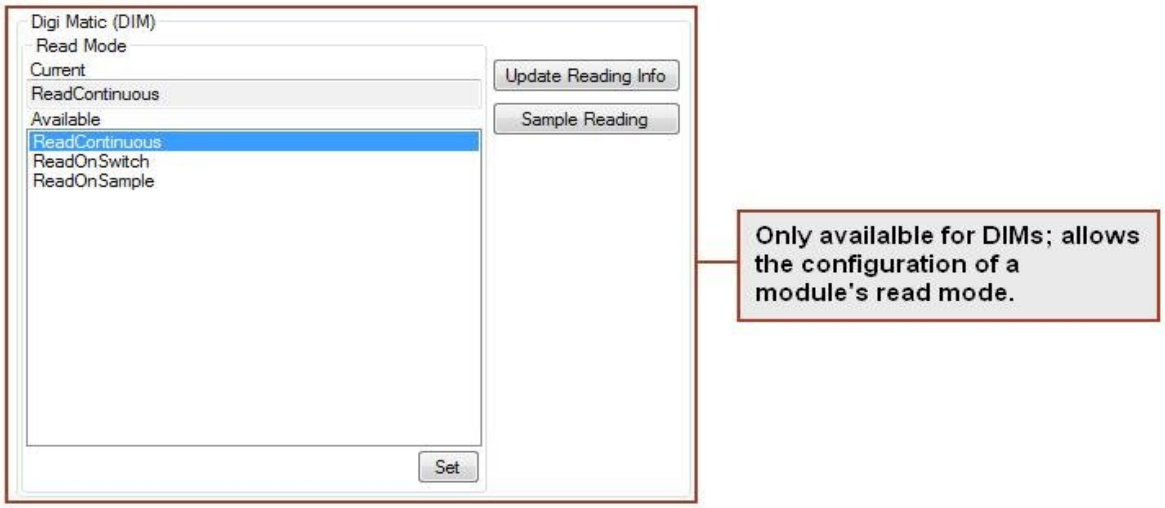
Available for the DIOM, EIM and LE; allows the configuration of the preset value. This can be set in either units or counts.

11.3.2.3.3 EIM Configuration

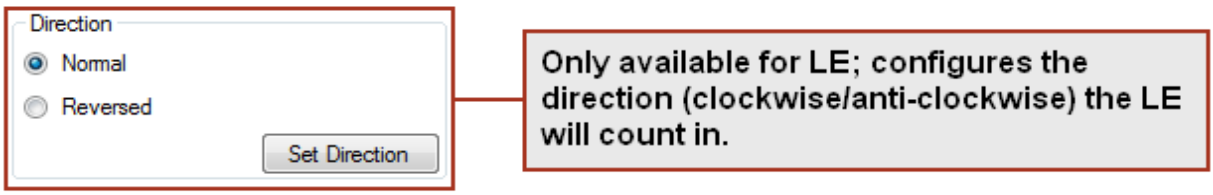


Only available for the EIM, allows the configuration of a module's Ref Action, Quadrature and Encoder modes.

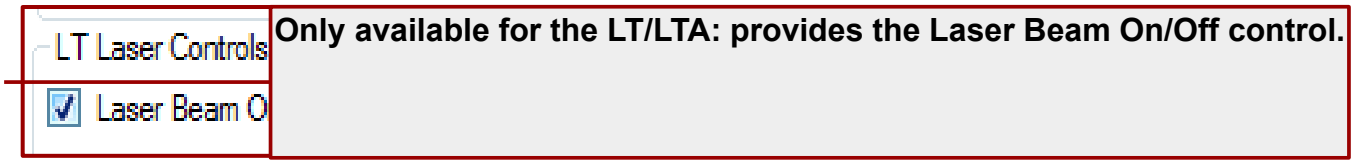
11.3.2.3.4 DIM Configuration



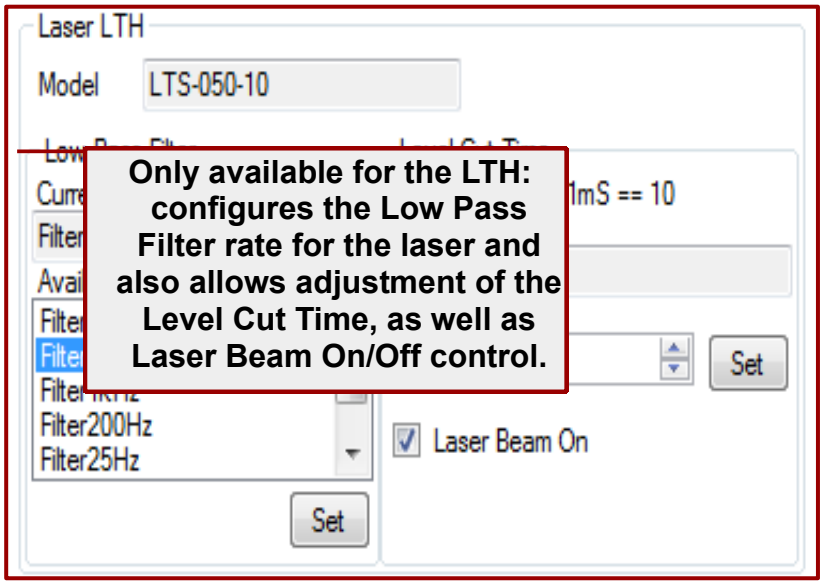
11.3.2.3.5 LE Configuration



11.3.2.3.6 LT/LTA Configuration

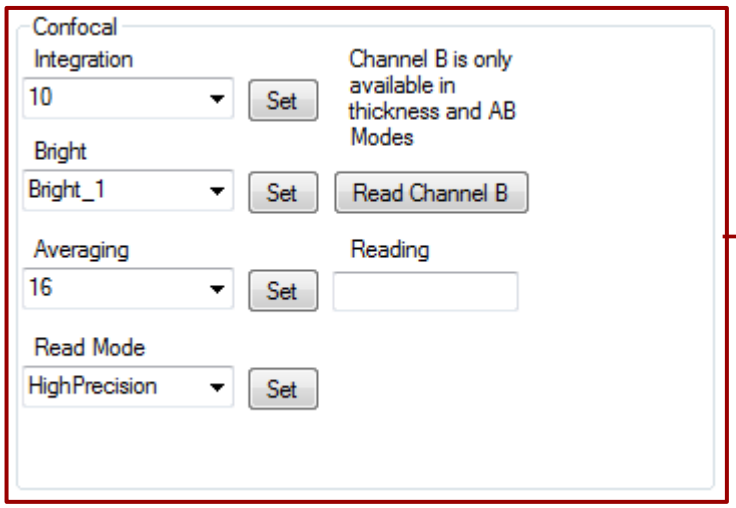


11.3.2.3.7 LTH Configuration



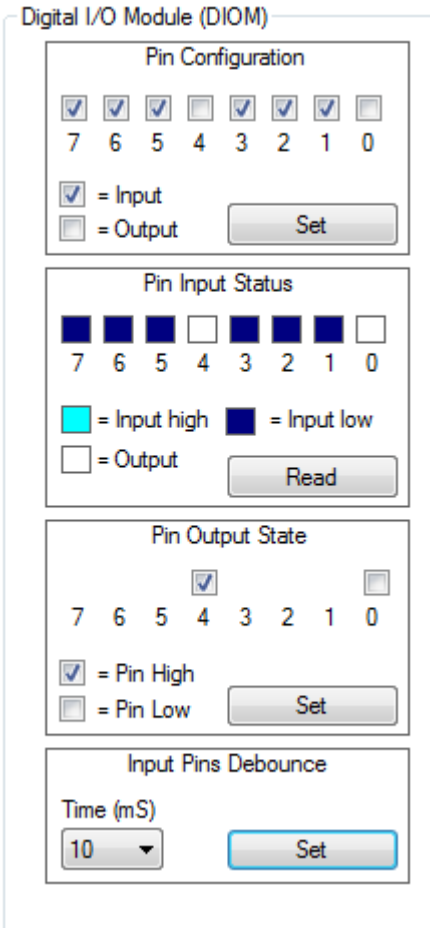
!

11.3.2.3.8 Confocal Configuration



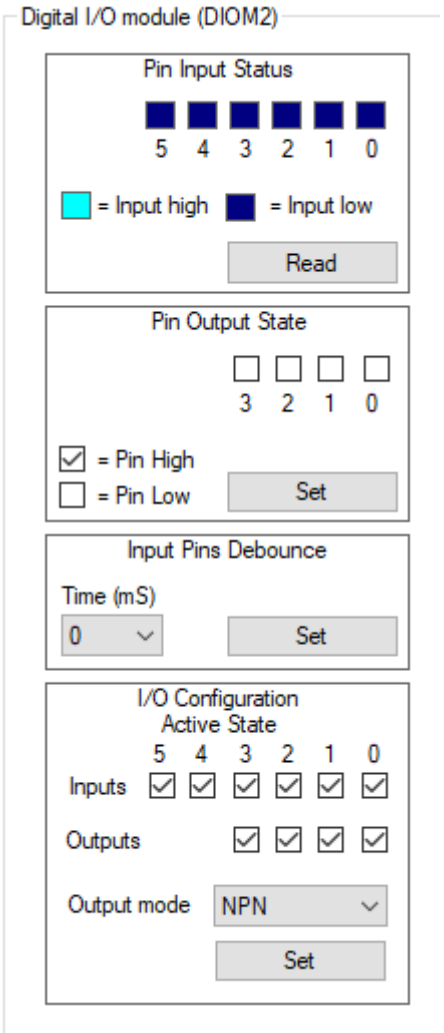
Only availa
Configures the Integration, Bright, Av
Channel B reading. Channel A readi

11.3.2.3.9 DIOM Configuration



Only available for DIOM products: allows configuration of a module's Input and Output pins and Input pin debounce time.

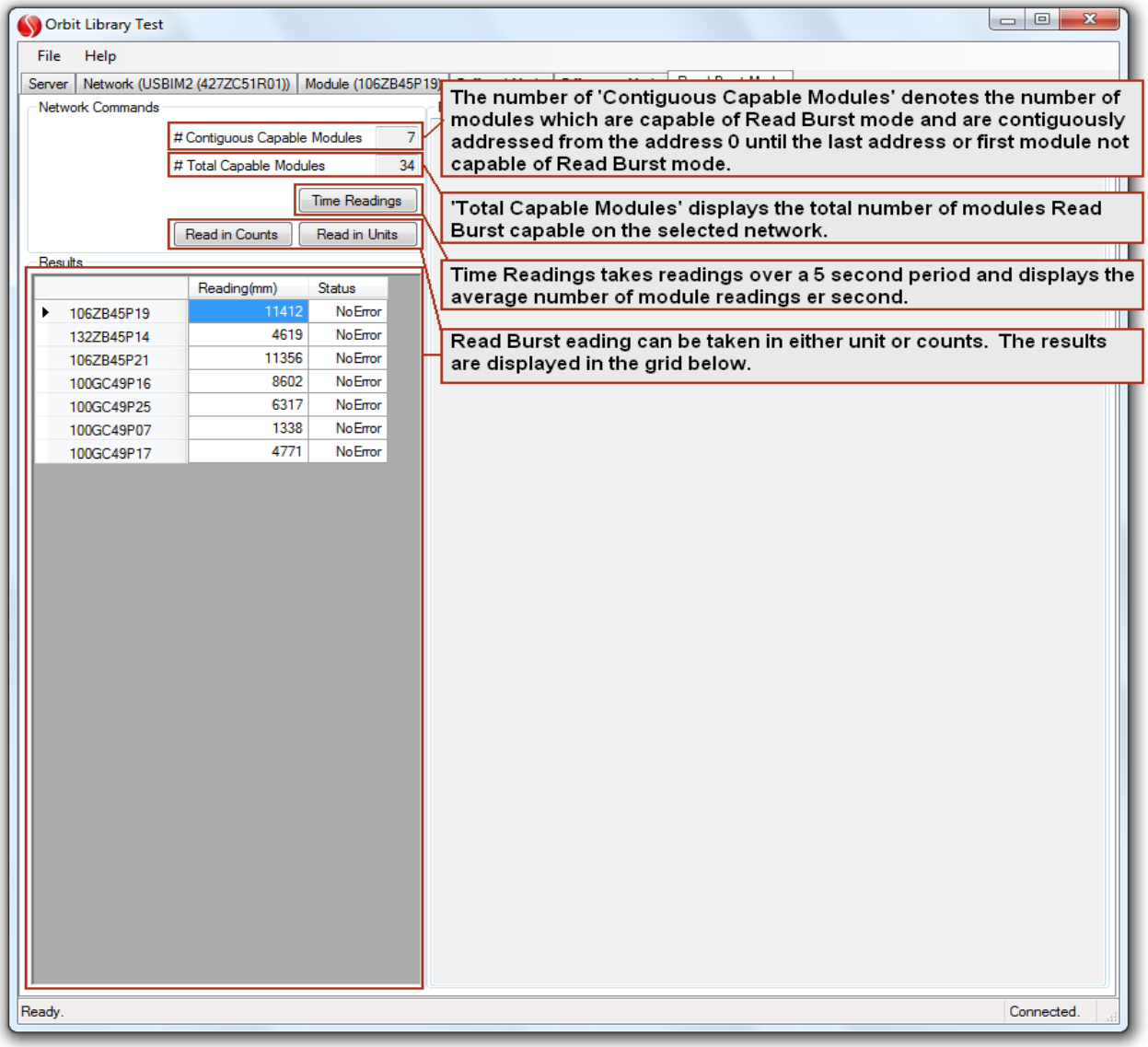
11.3.2.3.10 DIOM2 Configuration



Only available for DIOM2 products: allows configuration of a module's Input and Output pins and Input pin debounce time.

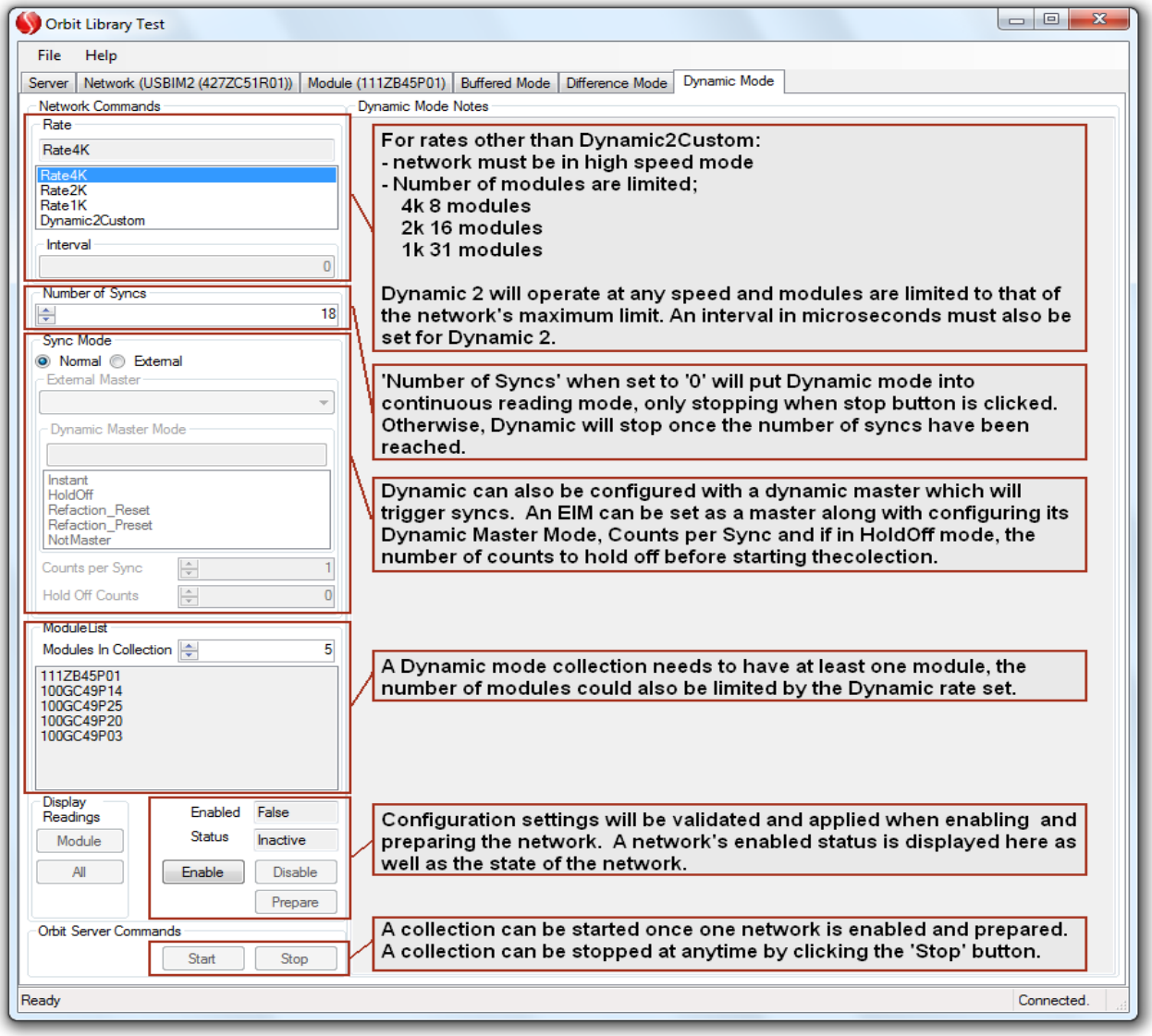
11.3.2.4 Read Burst Mode Tab

Only displayed when a Read Burst capable module and network is selected.



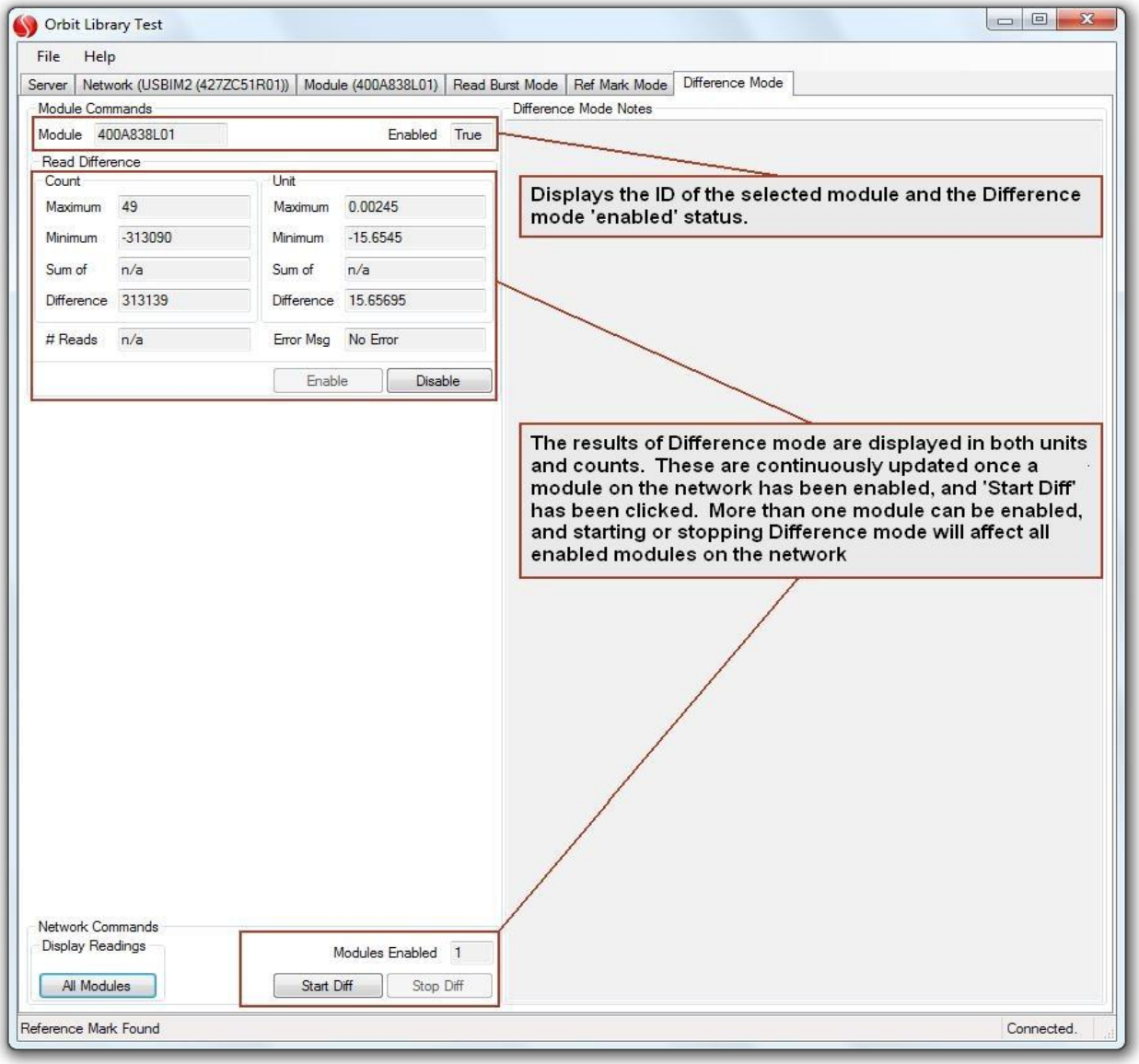
11.3.2.5 Dynamic Mode Tab

Only displayed when a Dynamic or a Dynamic 2 capable module and network configuration is selected.



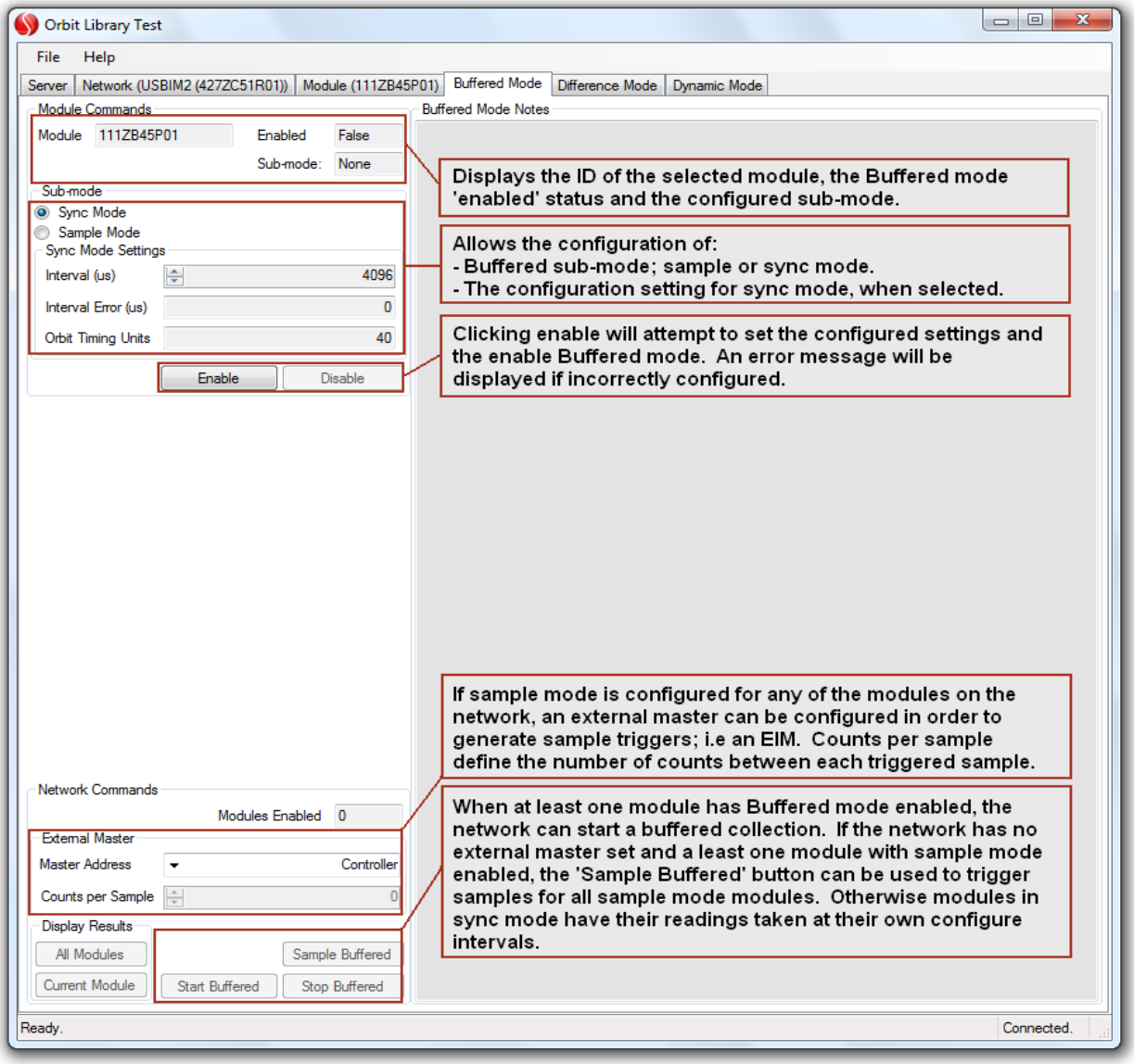
11.3.2.6 Difference Mode Tab

Only displayed when a Difference mode capable module is selected.



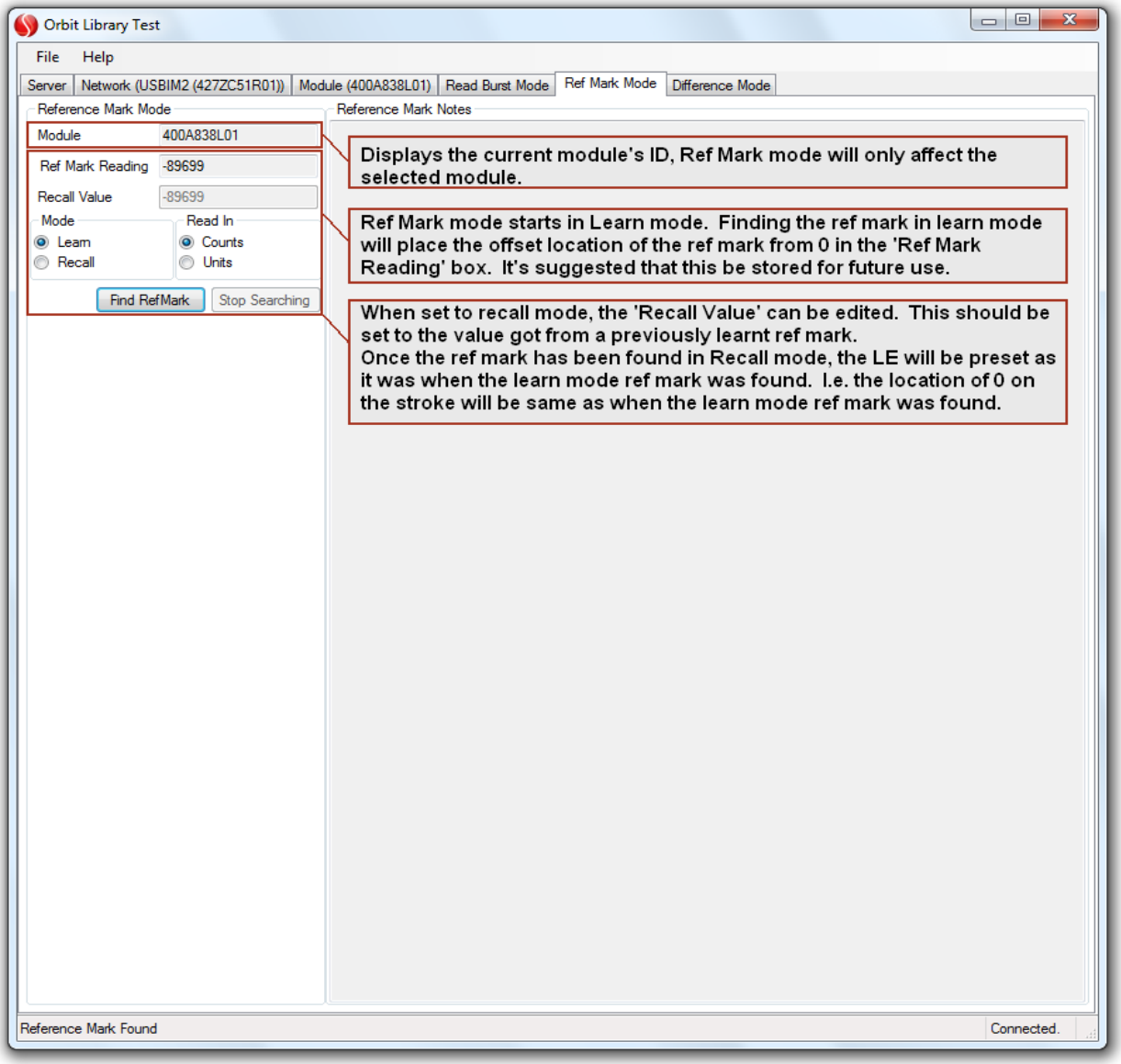
11.3.2.7 Buffered Mode Tab

Only displayed when a Buffered mode capable module is selected.

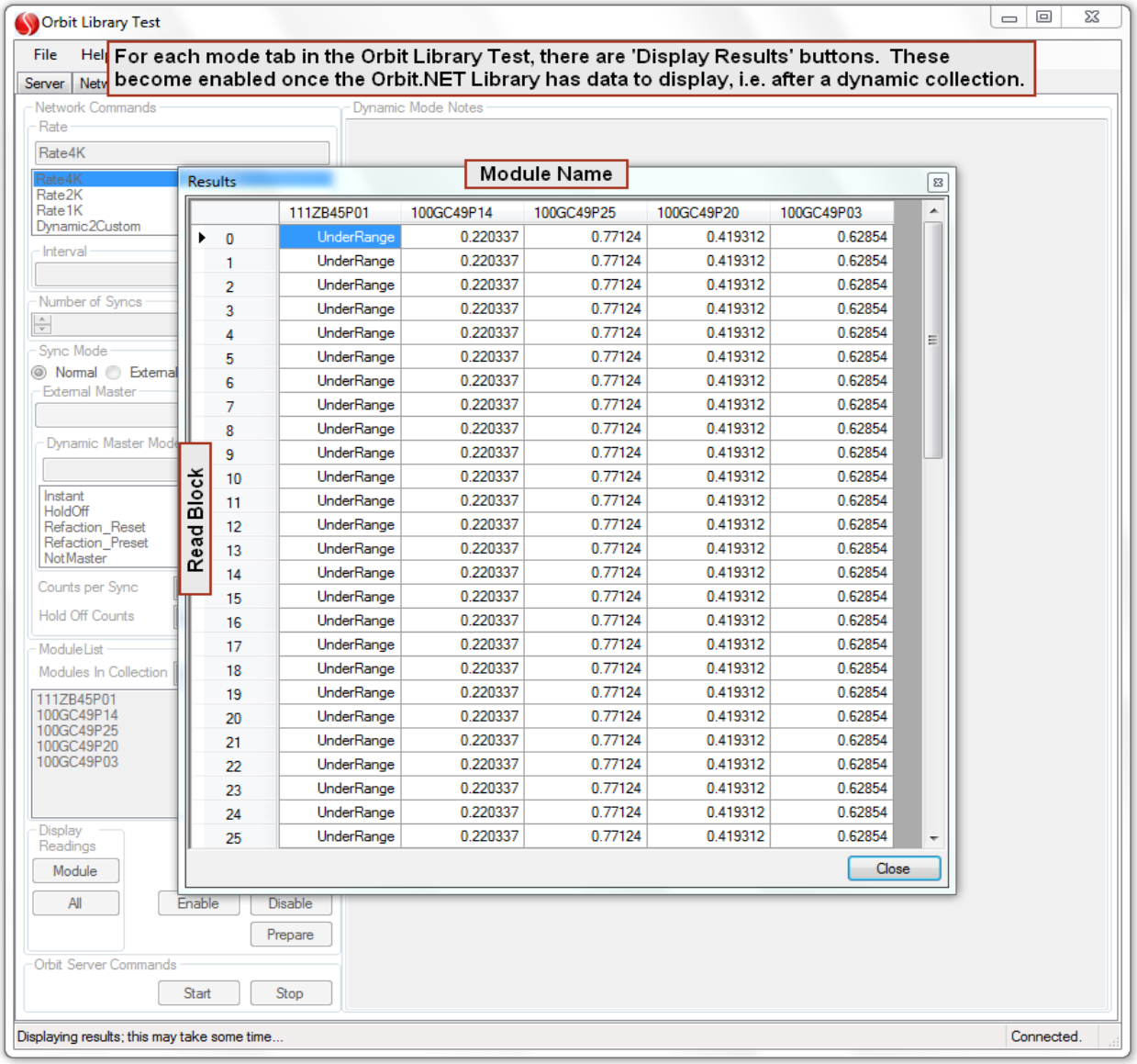


11.3.2.8 Ref Mark Mode Tab

Only displayed when a Ref Mark mode capable module is selected.



11.3.2.9 Results Window



11.4 SOURCE CODE

This section contains the technical details of the Library Test. This will include a partial walk through of the source code and steps towards familiarisation and navigation.

11.4.1 Development Tools

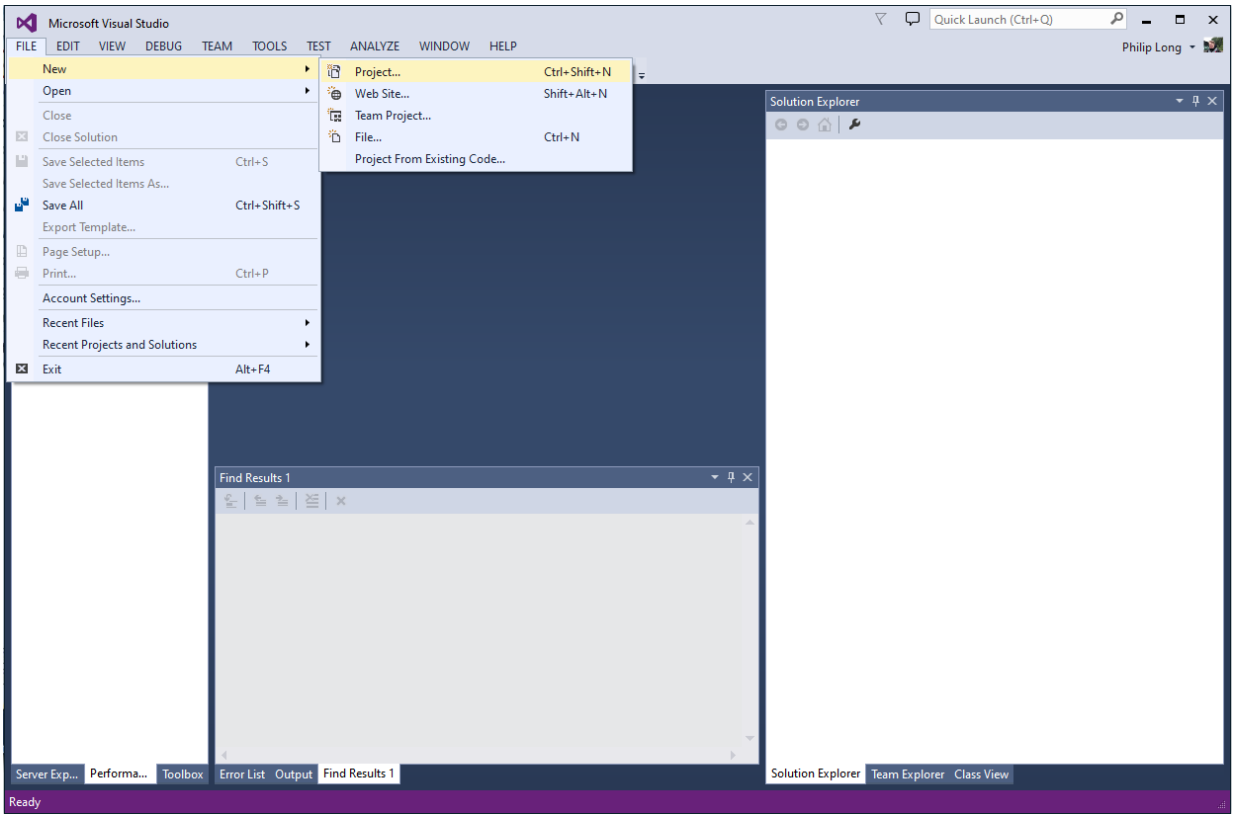
The Library Test is written in C# for the Microsoft .NET 3.5 Framework and has been tested on 32bit and 64bit variants of Windows 7, 8 and 10. The application was written in Microsoft Visual Studio Professional 2013.

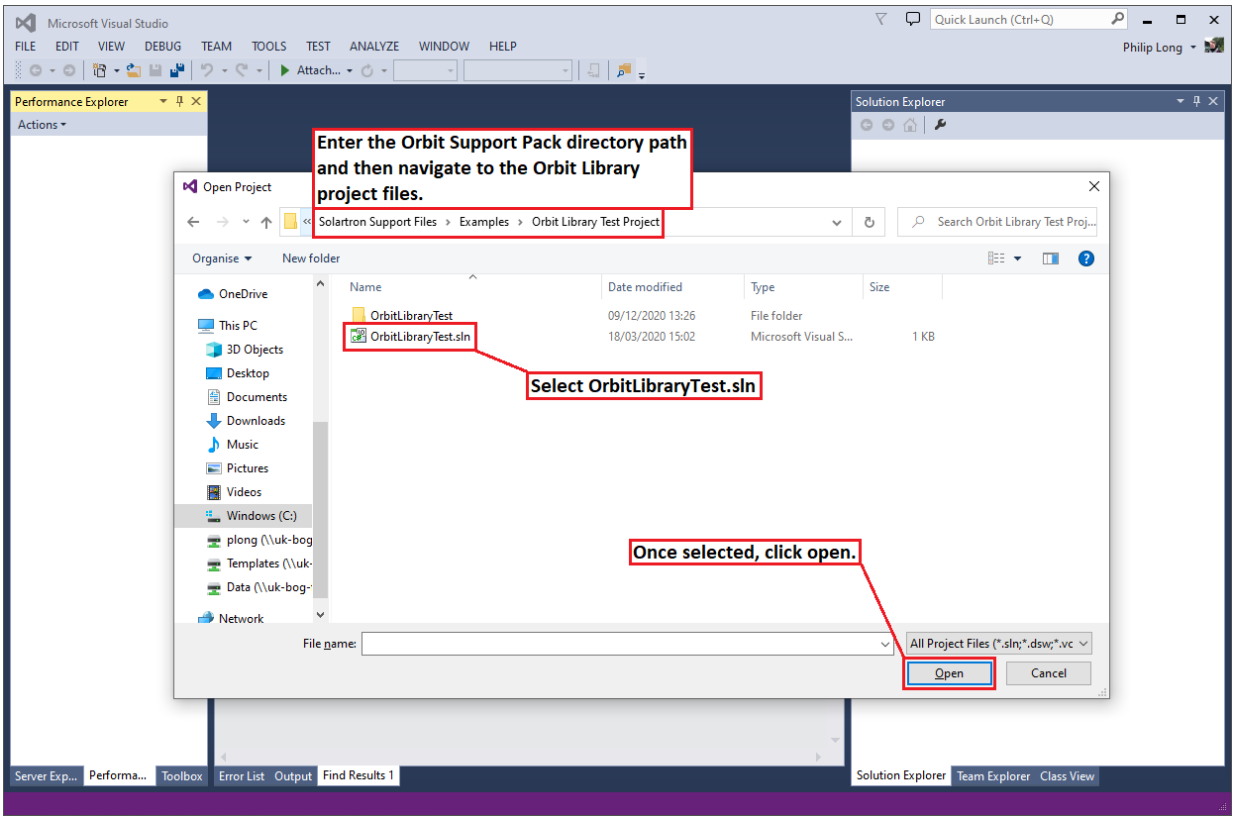
11.4.1.1 Microsoft Visual Studio Professional 2013

It is recommended that Visual Studio Professional 2013 is used for browsing and modification of the source code provided; although, there is nothing restricting the use of another compiler or IDE.
For further details and feature lists for Visual Studio Professional 2013, see the Microsoft website.

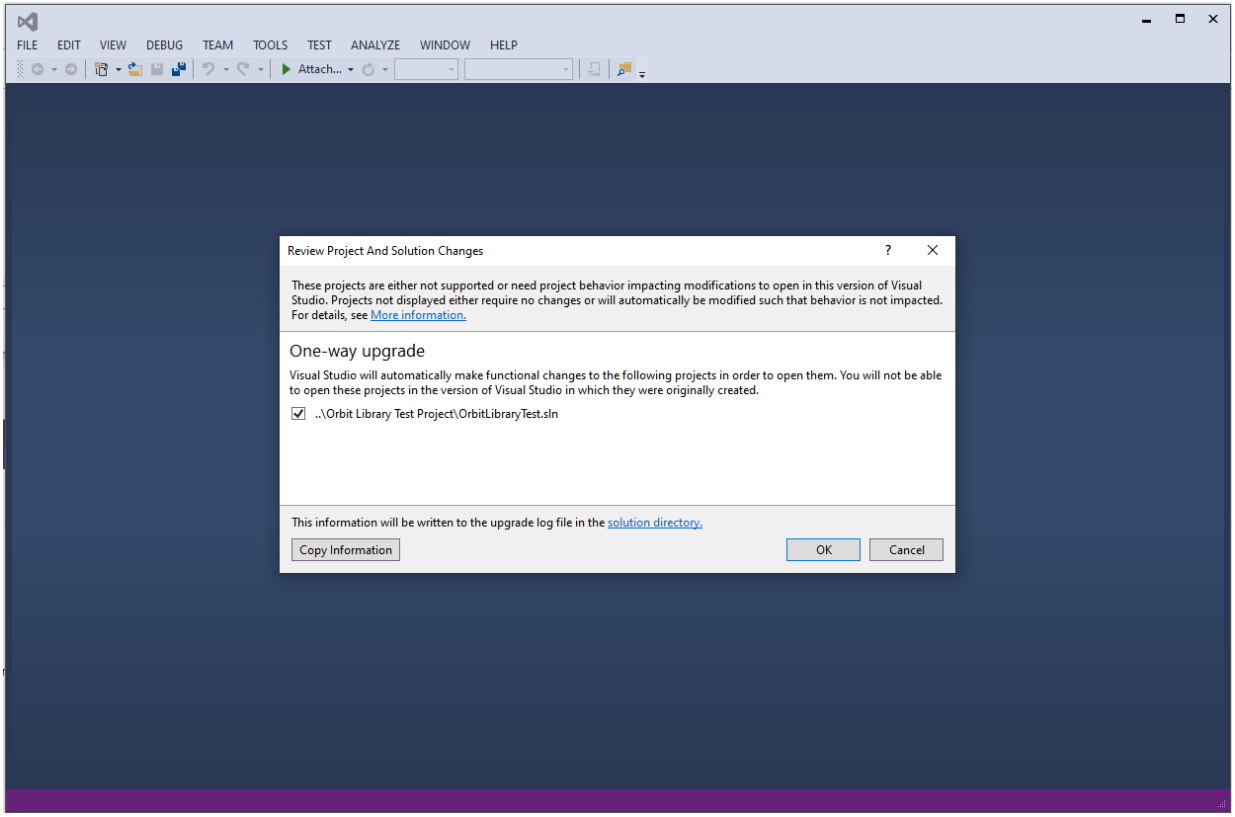
11.4.2 Opening the Project File

To open the Library Test 'solution file' in visual studio; run visual studio, go to *File* > *Open Solution* and navigate to the Library Test directory and open '*OrbitLibraryTest.sln'*.



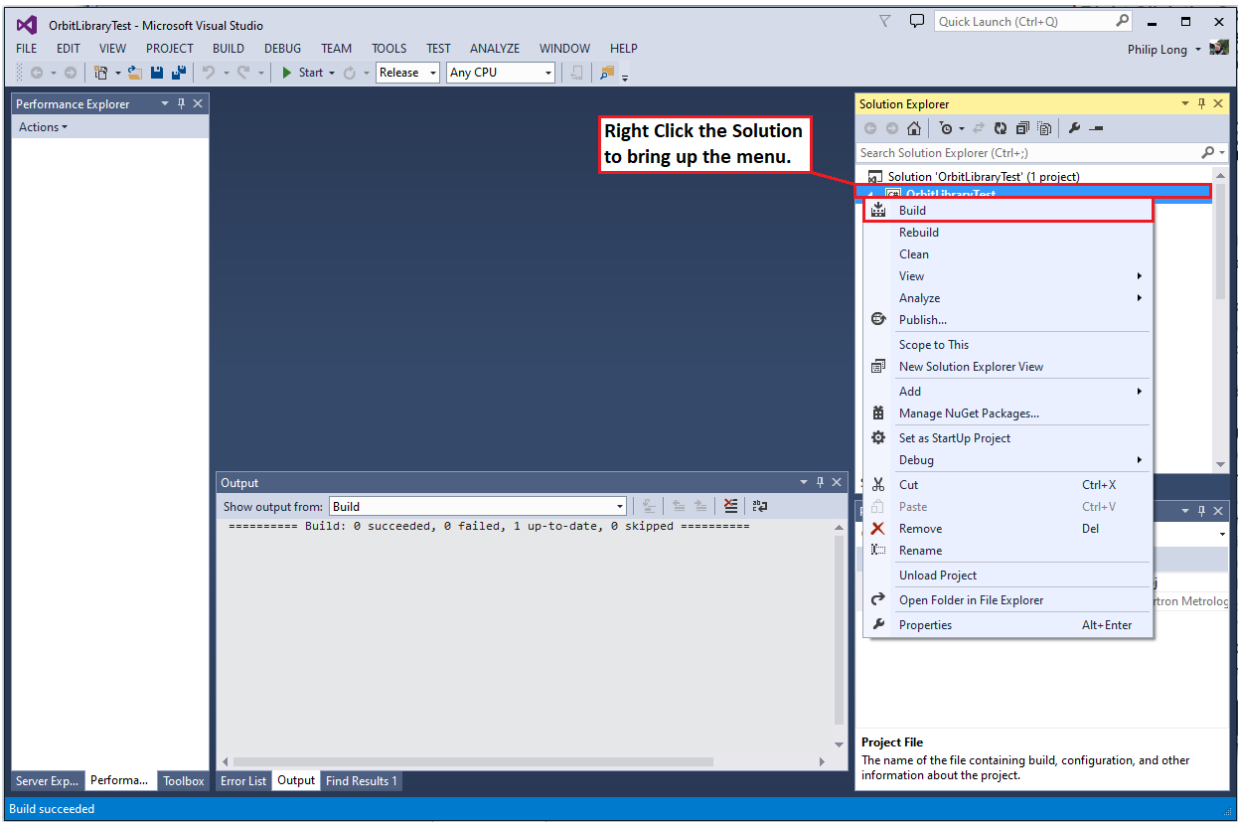


Note; Visual Studio may try and convert the project file depending on the version of Visual Studio the project was created with to the current version being used.
Follow the conversion wizard's instructions and the project should open correctly.

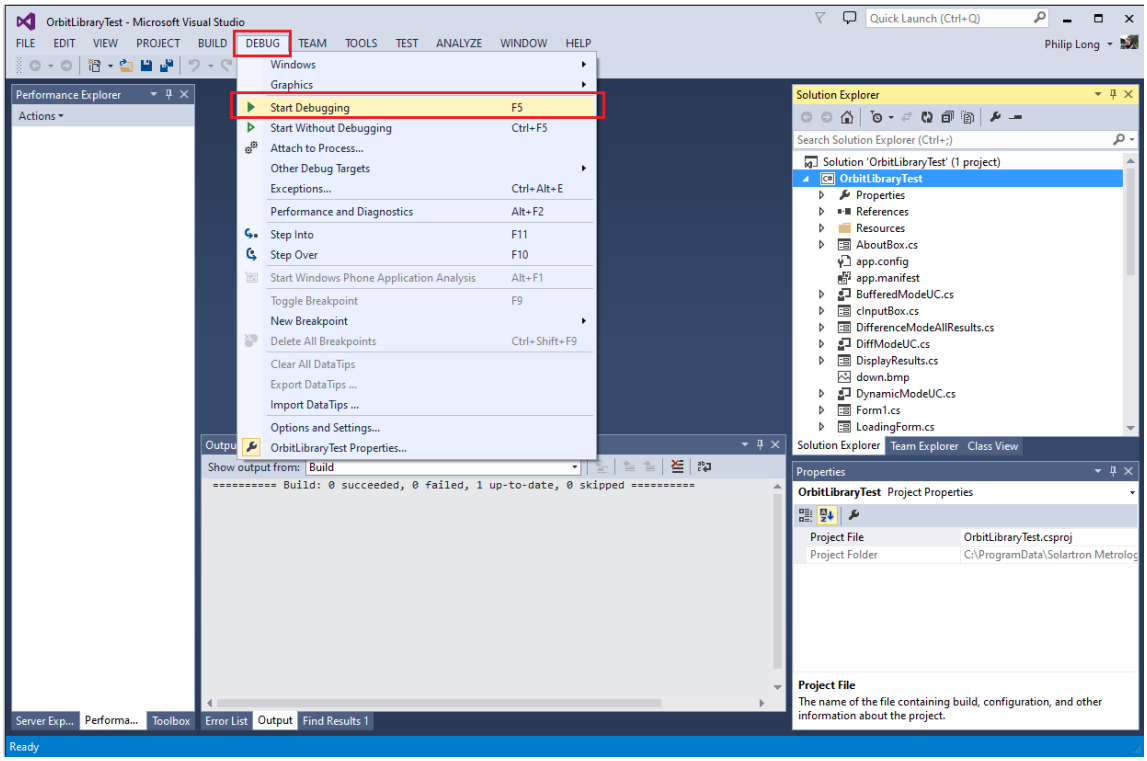


11.4.2.1 Compiling

In its initial state, the library test needs compiling before it can be run from the source code. This should be as simple as opening the 'solution' file in visual studio and clicking 'Build'.



11.4.2.2 Running

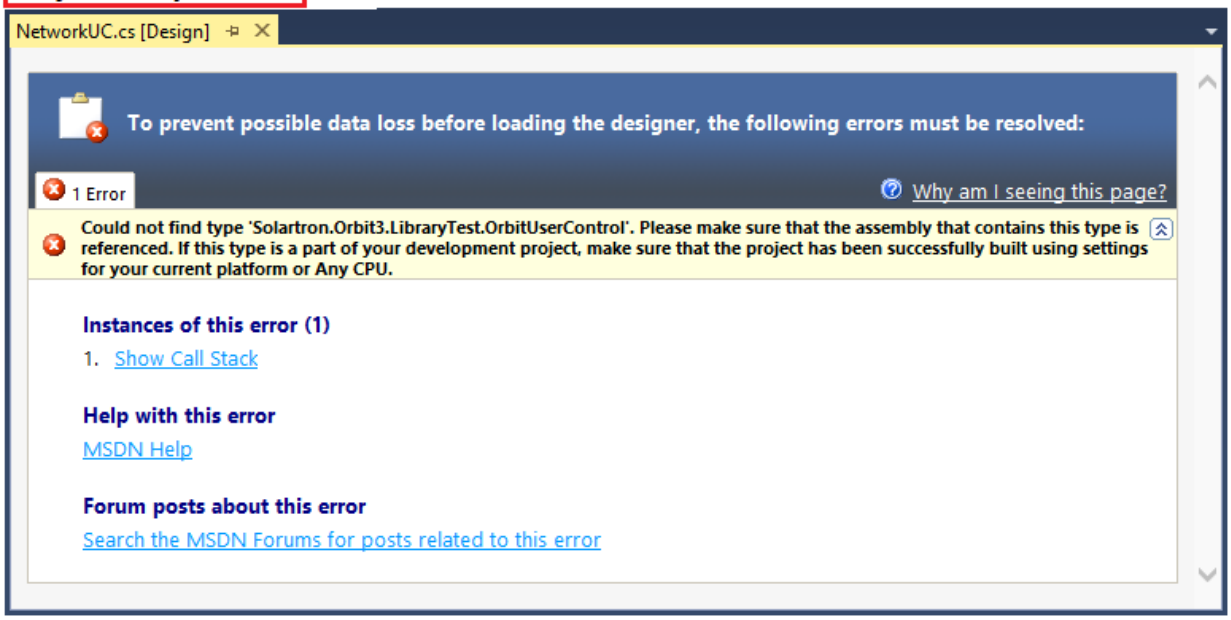


The application can be run from the same menu, or can be run from Windows Explorer from the bin/ directory in the Library Test's project directory.

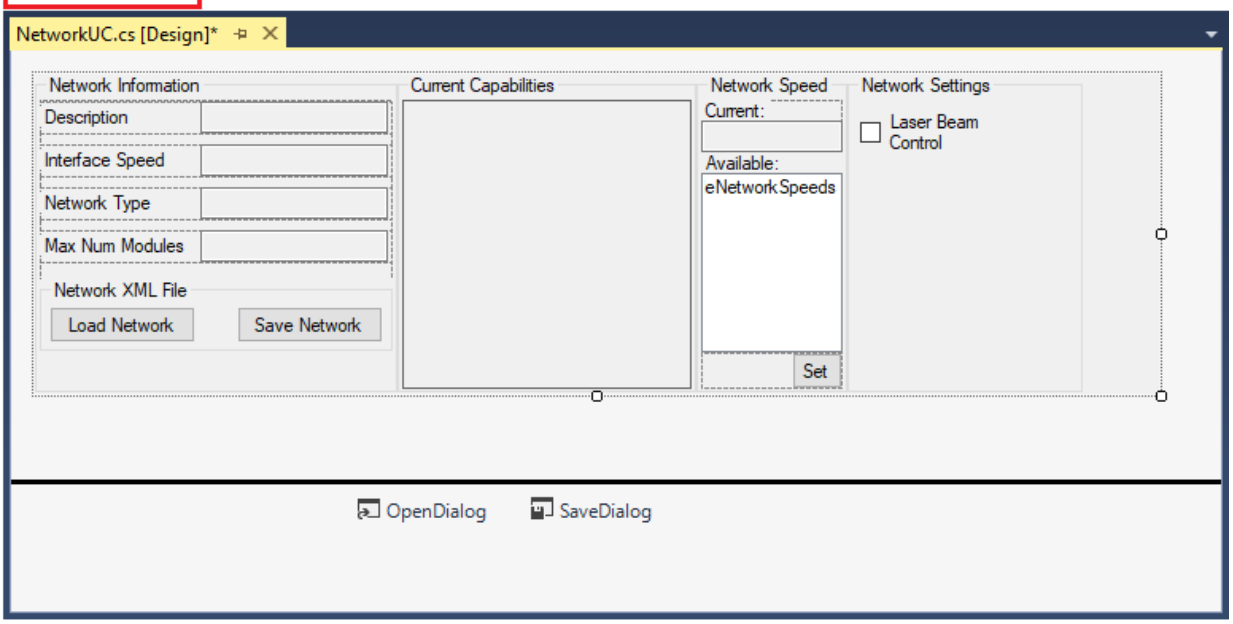
11.4.3 Navigating the Source Code

It is recommended that the source code is in a compiled state before opening any of the designer files (GUI files; forms user controls); in order to render certain parts, the designer requires them to be compiled.

Project not yet built.

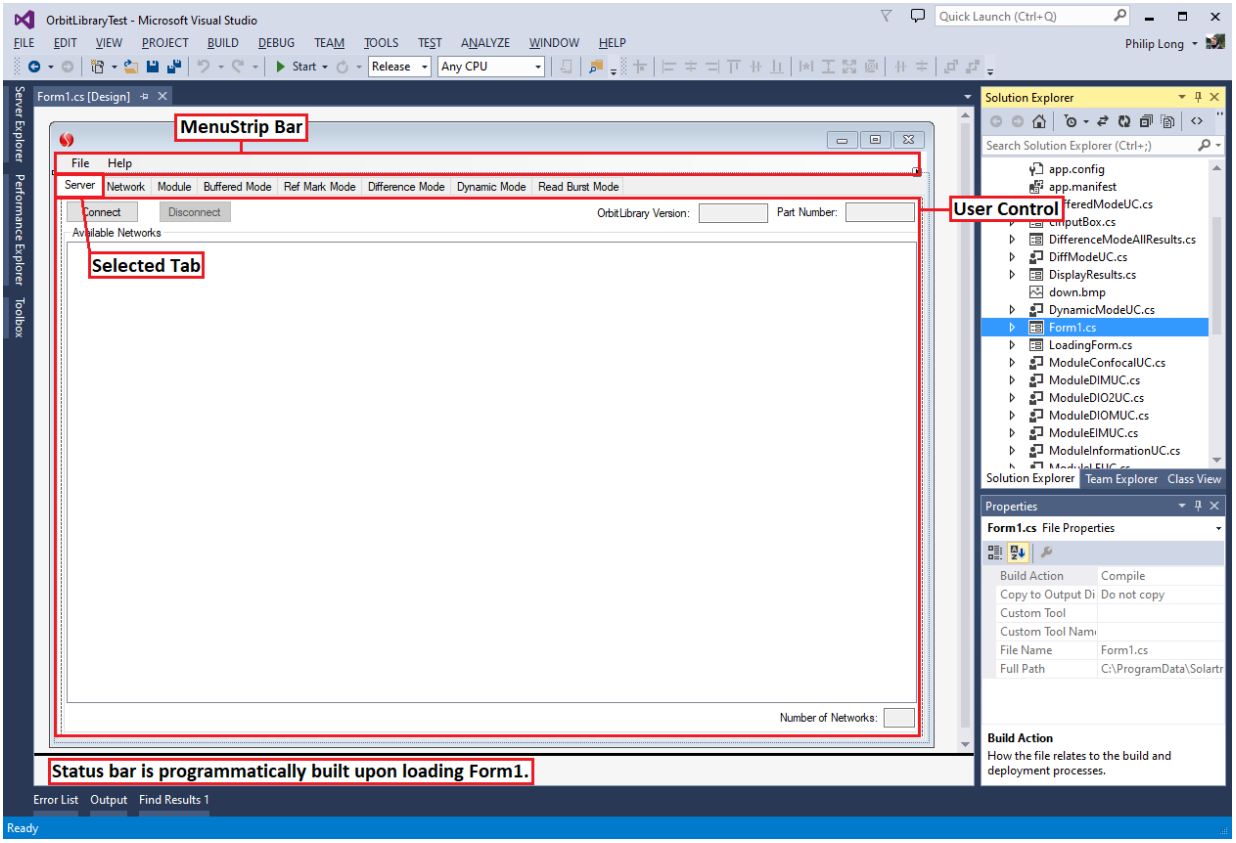


Project built.



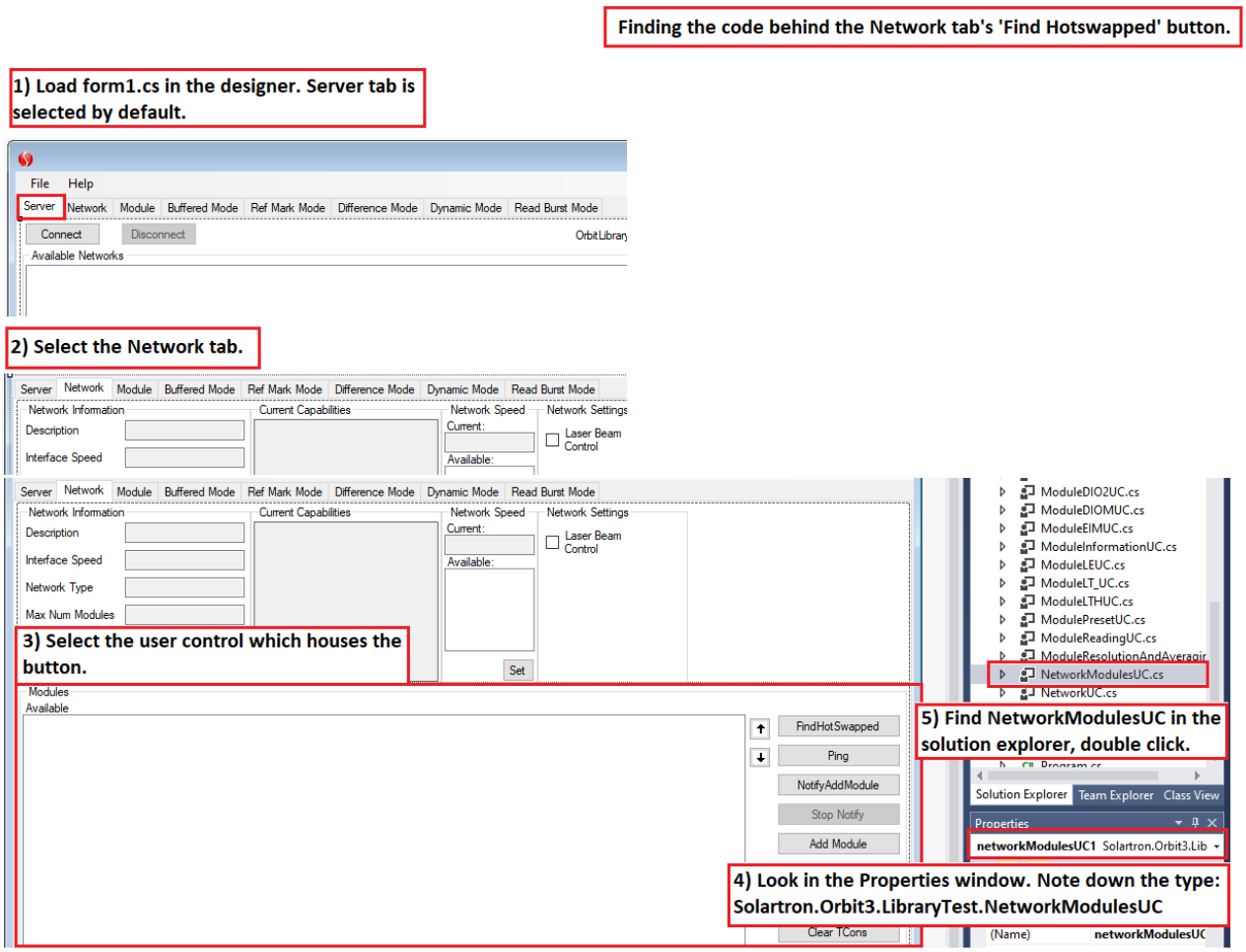
Once the Library Test is compiled and can run, the project is in a good position to be explored. The Library Test consists of a Windows Form form control containing a menu strip bar, a status bar and a tab container control. There is a tab for the Orbit server, network and module, as well a tab for each mode, Read Burst, Dynamic, Buffered, Difference and Ref Action. Each of these tabs contain user control which inherit from *OrbitUserControl*.

OrbitUserControl contains common functions to many user controls, thus, inheriting the *OrbitUserControl* class allows for much of the user interface code to be kept out of the way.

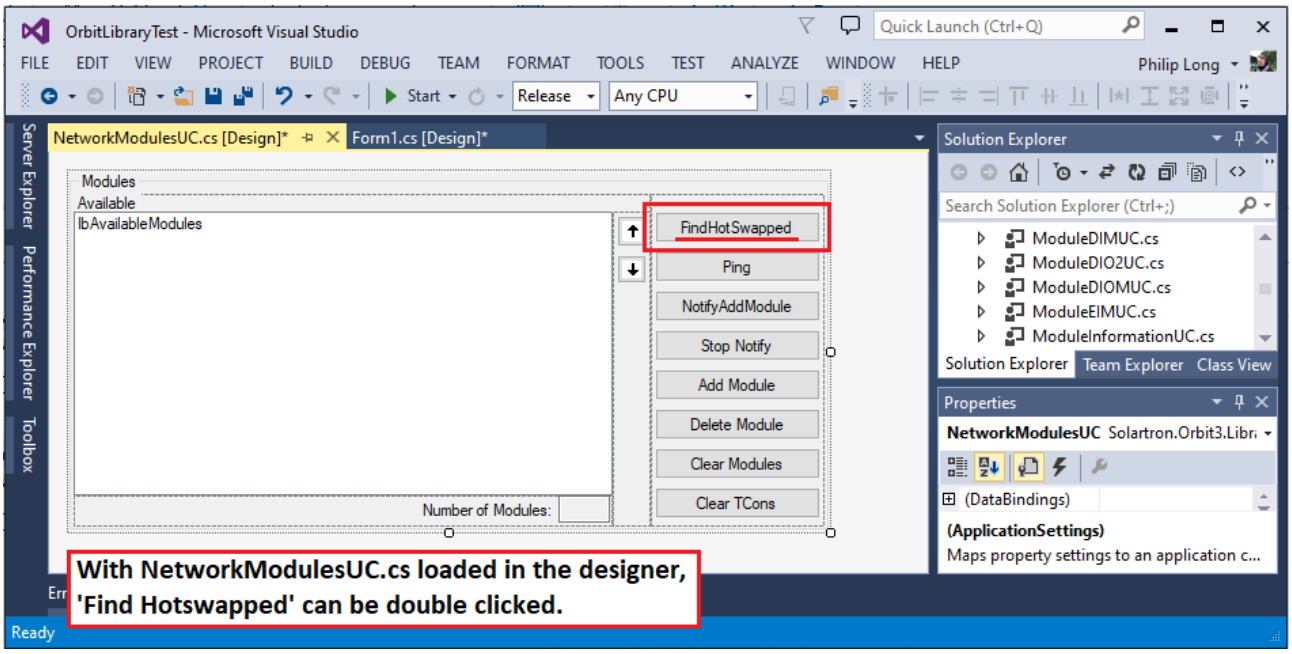


All the orbit functionality has been split up into these controls to ease of reading. In order to quickly navigate to an area of the Library Test, first double click *form1.cs* so that the windows forms designer loads.

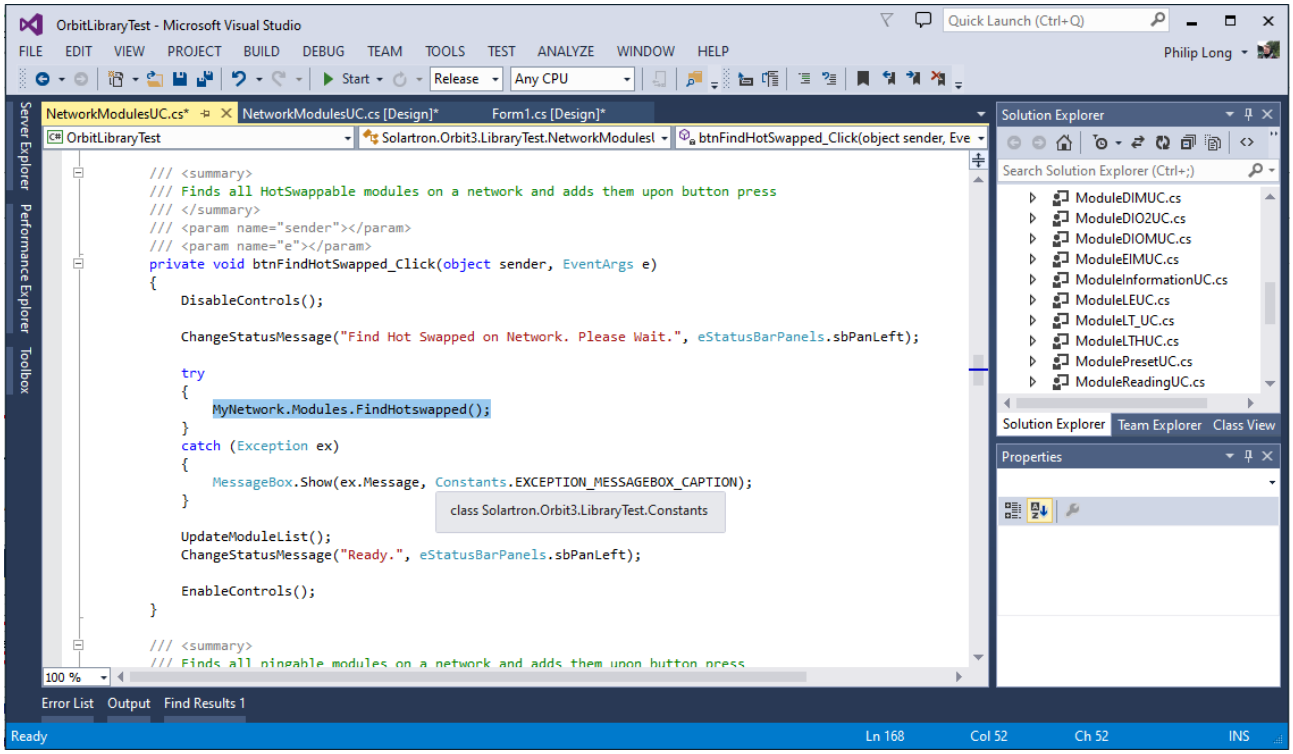
Next navigate the tab container and identify the user control which contains the area of interest. Note down the type on the control and select the *UserControl* type from the solution explorer.



The user control should now be loaded into the windows forms designer. Locate the button that interests you and double click it.



The source for the user control should open in the C# text editor, and the text cursor will be on the line of the button event handler. The code within the function provides an example on how to call that particular Orbit command.



12 MODULE SPECIFIC OPERATION

There are many Orbit Library commands that are specific to particular Orbit Modules. Also, newer Orbit features may not be available on some of the older Modules.
See Orbit Compatibility Roadmap - Modules for details.

12.1 DIGITAL PROBE (DP)

12.1.1 Introduction

All digital probe products are treated the same by the Orbit Library.
As well as digital gauging probes, 'Digital Probes' includes all digital Displacement, Mini Probes, Lever probes, Block gauges, Flexures and LT, LTA & LTH laser modules.

The Digital Probe was the first available Orbit Module, therefore it is compatible with nearly all Orbit Library members.
For probe readings, we recommend the ReadingInUnits property. This greatly simplifies taking scaled readings from the probe.

12.1.2 Programmable Resolution

The Modules are capable of being set to operate with the digital data output resolutions as defined below:

Resolution	Number of Bits
Resolution A	14 (Default)
Resolution B	16
Resolution C	18

Change using the Module's Resolution property.
Module's isResolutionAvailable method can be used to check whether a particular resolution is available before attempting to set it, (Orbit1 DPs only support 14 bit resolution).
Module's AvailableResolutions property returns an array of the module's supported resolutions.

Note: The number of bits represent the full stroke of the module.
(e.g. for a 10mm DP set to 16 bit resolution, 2^{16} is the reading for 10mm).

A default startup value is configurable on a module by module basis with the DefaultResolution property, This saves the default resolution to the module which restores this new default resolution on power up.

12.1.3 Programmable Electrical Measurement Bandwidth

This provides digital filtering of the Module readings. The update rate is reduced as the averaging is increased.

The Measurement bandwidth of the Module is programmable by setting the Number of Averages of the Measurement cycle as below (change using the Module's Averaging property).

Module's isAveragingAvailable method can be used to check whether a particular averaging is available before attempting to set it, (Orbit1 DPs do not support averaging).
Module's AvailableAveragings property returns an array of the module's supported averaging settings.

Measurement Bandwidth (Hz)	Number of Averages of Measurement Cycle
450	1
420	2
320	4
200	8
100	16 (Default)
50	32
25	64
12	128
6	256

A default startup value is configurable on a module by module basis with the DefaultAveraging property, This saves the default averaging to the module which restores this new default averaging on power up.

12.1.4 Probe disconnect detection

This provides a method to detect whether the probe has become disconnected from the PIE module. This feature was introduced in newer hardware in mid 2017. Prior to this, there was no indication that the probe had become disconnected from the PIE module.

In order to provide the user with information about whether the module has this feature or not, the "ProbeDisconnectSupported" module property (available in the Orbit Library) should be interrogated.
If the property is TRUE, this feature is available.

On detection that the probe has been disconnected (e.g. cut wire), a "NO_PROBE" error (see Orbit Error Codes and Error Handling) is returned when requesting a reading and the module's red status LED is lit.

- This error can only be cleared by fixing the probe disconnection issue and re-powering the PIE module.

12.2 ANALOGUE INPUT MODULE (AIM)

The AIM has similar functionality to the Digital Probe Module, therefore it is compatible with nearly all Orbit Library members, including Averaging and Resolution.
For module readings, we recommend the ReadingInUnits property. This greatly simplifies taking scaled readings from the probe.

12.3 AIR GAUGE MODULE (AGM)

12.3.1 Introduction

The AGM module provides air gauging functionality to the Orbit Network. It has similar functionality to a Digital Probe. After mastering it is read in the same way as a Digital Probe and therefore supports nearly all Orbit Library Members.

For applications that use the Orbit Library, readings are converted to mastered readings automatically.
Internally, readings communicated over the Orbit Network are always Zero To Scale to provide maximum resolution. Therefore if not using the Orbit Library the minimum Master value and end-band must be added to the reading.

12.3.2 Mastering

Air Gauge Modules must be mastered before use.
The simplest way to do this is to use the Air Gauge Utility (refer to the Orbit Module manual).
Alternatively, for the AGM-A its keypad/display can be used.
If it's necessary to implement your own solution, the following applies, use the properties of the `OrbitModuleAGM` class.
See Implementing Mastering.

12.3.2.1 Re-Mastering while communicating on an Orbit Network

Re-mastering the AGM device to the same size masters as previously performed, can be performed without issue while the AGM device is communicating on an Orbit network.

However, re-mastering to differing master max/min values results in the device changing its internal ‘range’, this information must then be re-read by any controlling device. To aid trapping this, the AGM will return an Orbit error code (“Module Information needs reading”) The error flag will be cleared by either a full network re-initialisation or by calling the OrbitModule ‘updateInfo’ method.

12.3.2.2 Mastering using the Air Gauge Utility

The Air Gauge Utility can be started with `/module` and `/network` command line parameters to launch directly into mastering a specific module on an Orbit Network. Your application must be disconnected from the Orbit Network during this process.

Example command line:
`Air_Gauge_Utility.exe /module=[MODULEID] /network=[NETWORKID]`

If mastered successfully the utility will exit with an exit code of 0. If not successful a non zero exit code will be returned.

Code	Description
0	No error
9000	Failed to load the Orbit Library
9001	Failed to connect to Orbit
9002	No Orbit networks found
9003	Failed to find the network specified on the command line
9004	Failed to find the module specified on the command line
9005	Mastering cancelled by the user.

12.3.2.3 Implementing Mastering

This section is only required if you need to implement your own non Orbit Library solution.

The `OrbitModuleAGM` class makes the properties and methods below available for mastering:

Properties:

MasterMinReadingInUOM	The read value at the minimum set point e.g. 35mm.
MasterMaxReadingInUOM	The read value at the maximum set point e.g. 35.05mm.
MasterMinPressure	The pressure at the minimum set point in PSI.
MasterMaxPressure	The pressure at the maximum set point in PSI.
PressureReading	The pressure reading from the pressure sensor in PSI (readonly).

Methods:

ApplyMastering	Saves the mastered values to the module non volatile memory and sets up the module for reading using the minimum and maximum set points.

Mastering Process

1. Instruct the operative to insert the probe into the minimum master.

2. Use the `PressureReading` property to continually read the pressure sensor pressure reading and instruct the operative to adjust the pressure to close to the recommended value (24psi±0.5psi) using the needle value.

3. Once the pressure is set, read the actual pressure read from the `PressureReading` property for use later.

4. Instruct the operative to insert the probe into the maximum master.

5. Once the probe is inserted, read the actual pressure read from the `PressureReading` property for use later.

6. At this stage, optionally display the mastering values to the operator.

7. Set the minimum set point value using the `MasterMinReadingInUOM` property. (`MasterMinReadingInUOM = 35).`

8. Set the maximum set point value using the `MasterMaxReadingInUOM` property (`MasterMaxReadingInUOM = 35.05).`

9. Assign to the `MasterMinPressure` property the pressure value sampled.

10. Assign to the `MasterMaxPressure` property the pressure value sampled.

11. Call the `ApplyMastering();` method to save the mastering and initialise the module with the new set points.

Once mastering is applied, the AGM adds 30uM to the range above and below Master Min and Master Max sample points.

For example, if Mastering range is 50 microns, the total ‘span’ of the device will be 110 microns to provide ‘end-bands’.

Over Orbit this can seem confusing, for instance if a master min of 35.000mm is used and a master max of 35.050mm is used, the ‘mastering range’ is 0.050mm, however the AGM will expand this to cover 0.110mm including a 30 micron ‘end-band’ at either end to keep the mastering points within the range of the device.

Values communicated over the Orbit Network are always Zero to Stroke. The Orbit Library automatically converts the values communicated over the Orbit Network into mastered readings for all measurements in units of measure. Note: As the stroke has 0.03mm end bands, if the $Master_{Min}$ value is zero, the minimum value read will be -0.03mm and the maximum value will be $Master_{Max}$ + 0.03mm.

Graphical representation – including Orbit reading values (OrbitLibrary ReadingInUnits values).

12.3.3 AGM Module Additional Properties and Methods

In addition to the mastering properties and settings above, the OrbitModuleAGM class provides the properties and methods below to configure AGM specific settings.

Properties:

DisplayUnitsOfMeasure	The units of measure of the AGM display (mm, inch or mils). This setting will not affect readings via the Orbit Library. ReadingInUnits will always return mm.
IsMastered	Returns true once the AGM has been mastered (readonly).
ShowPSI	Set true to show both the measurement reading and pressure readout on the AGM screen.
Passcode	The pass code to access the AGM menu. Integer in the range 0 to 99999999. Set to 0 to disable the pass code.
RotateDisplay	The AGM display screen rotation, None, 90, 180, 270 degrees or Auto Rotate.
RotateKeyboard	Set true to rotate they keyboard to match the display rotation.
LimitLo	Low limit measurement threshold for display purposes.
LimitHi	High limit measurement threshold for display purposes.

Methods:

ResetFactorySettings	Resets all AGM Settings to factory defaults including setting the module as unmastered

12.4 LINEAR ENCODER (LE)

12.4.1 Introduction

The Linear Encoder is an incremental, high accuracy measurement module. This module does not support dynamic or buffered modes. Readings are returned via the ReadingInUnits property.

12.4.2 Linear Encoder & Reference Mark

All Linear Encoders are incremental in reading; they lose their datum on power down. A reference mark is provided in order to provide an absolute datum.
This eliminates the need to re-master and calibrate your sensor. It is simply required to extend/retract the sensor´s tip which can be done comfortably with pneumatic / motorized linear encoders.

The Reference Mark is located approximately 5mm from fully in (retracted).
It will only be read when moving INWARDS. During outward movement the Reference Mark will not be read.

The following instructions on use of the Reference Mark are taken from Orbit Library Test :

Select Read In 'Counts' or 'Units' (Raw Counts or Units Of Measure)

Select 'Learn' Mode to store the value read at the reference mark.
Before starting, ensure that the module has already been set to the desired reading via the 'Preset' option in the 'Module' tab.

Click 'Find RefMark' to start.
Now move the tip of the LE until the reference mark has been found. Once found, the 'Reference Mark Reading' should be stored for future use in 'Recall' mode.

Select 'Recall' Mode to set the module back to the 'Learn' value. Enter the previously stored 'Reference Mark Reading' (obtained via 'Learn' Mode) in the 'Recall Value To Set'. Click 'Find RefMark' to start.
Now move the tip of the LE until the reference mark has been found. Once found, the reading will be restored to that in 'Learn' Mode.

Click ‘Stop Searching’ to abort finding the Reference mark.

12.5 ENCODER INPUT MODULE (EIM)

12.5.1 Introduction

The Encoder Input Module (EIM) is an Orbit Module which can interface to incremental and rotary encoders with square wave outputs, allowing these sensors to be interfaced into the Orbit Measurement System.
Using rotary encoders via the EIM in conjunction
with linear measurement sensors allows the Orbit Measurement System to perform part profiling.

The EIM can be read on command like any other Orbit Module including as part of a dynamic collection.
The EIM can also be used provide synchronization for a dynamic collection (Dynamic External Master Mode). Similarly, it can be used to Externally trigger/sample readings in buffered mode (External Master Mode).

12.5.2 EIM Module Properties

Inputs (EimInterfaceType Property)	Single Ended Differential
Interpolation (QuadratureMode Property)	X1 (default) X2 X4 Count AB Count DIR
Reference Pulse (RefAction Property)	Do nothing Reset counter on each reference pulse Preset counter on each reference pulse Reset counter on first reference pulse only Preset counter on first reference pulse only Reset counter on first reference pulse only and enable, Sync, Transmit and Holdoff functions (used in dynamic external master mode) Preset counter on first reference pulse only and enable, Synch, Transmit and Holdoff functions (used in dynamic external master mode)
	Note that if RefAction is set other than “Do nothing”, a manual PresetInCounts will **not** be actioned until RefAction has finished.

Refer to the *OrbitLibrary Code Reference* for details and Orbit Library Test for example code.

12.6 DIGITAL INPUT OUTPUT MODULE (DIOM)

12.6.1 Introduction

The Digital Input / Output Module (DIOM) enables the 'Orbit Network' to interface with the outside world. The module provides 8 general purpose input / output lines. Each line can be individually configured as Input or Output.

The OrbitLibrary has (since V1.2.0.6 - 2016) specific properties for the DIOM Pins to ease usage. Prior to this, the ReadingInCounts and PresetInCounts properties were used. See Legacy.

Default state on all pins at switch on is INPUTs.

12.6.2 DIOM Module Properties

The DIOM module class has two specific module properties:

- Pins (a collection of OrbitModuleDIOMPin classes – one for each pin)
  - e.g. `DIOMmoduleInstance.Pins[4]` gives the OrbitModuleDIOMPin class for pin 4
- DebounceTime (type `eDiomInputDebounceTime`)
  - An enumeration of possible debounce times for ALL input pins of the DIOM. Note that the default (power on) DebounceTime value is zero.

12.6.3 DIOM Pin Members

Each Pin class object has the following properties:

- `Config` Get / Sets the pin configuration (input or output)
- `InputPinState` Returns the pin state (high or low) in input mode. Note that this setting is irrelevant if the pin is set into output configuration. Read Only
- `OutputPinState` Gets or sets the pin state (high or low) in output mode. Note that this setting is irrelevant if the pin is set into input configuration.
- `PinNumber` This is the pin I/O number (0 to 7) of the DIOM

12.6.4 Setting Pin Configuration

Each DIOM pin can be set to input or output. Use the Pin.Config property to select e.g.

```
DIOMmoduleInstance.Pins[4].Config = eDiomIoPinConfig.output;
```

Sets pin 4 to an output

12.6.5 Read Inputs

Note that the DIOM pin needs to be configured as an input for a meaningful result to be obtained. Also, a ReadingInCounts **must** be actioned before readings are read. Use the Pin.InputPinState property to read, e.g.

```
int DummyRead = DIOMmoduleInstance.ReadingInCounts;
if (DIOMmoduleInstance.Pins[4].InputPinState ==  eDiomIoPinState.low)
```

Reads the state of Pin 4. If it is low state, then …

12.6.6 Set Outputs

The DIOM pin needs to be configured as an output for a meaningful result to be obtained. Use the Pin.OutputPinState property to read, e.g.

```
DIOMmoduleInstance.Pins[4]. OutputPinState = eDiomIoPinState.low
```

Sets the state of Pin 4 to low.

Note that a DIOM pin in output mode set high is equivalent to a DIOM pin set in Input state. The output is pulled high by a pull-up resistor, so is not able to source much current. See Module manual for hardware details of pins.

12.6.7 Improving Reading Integrity

When reading external inputs, it is wise to issue multiple reads to help reduce the effects of:

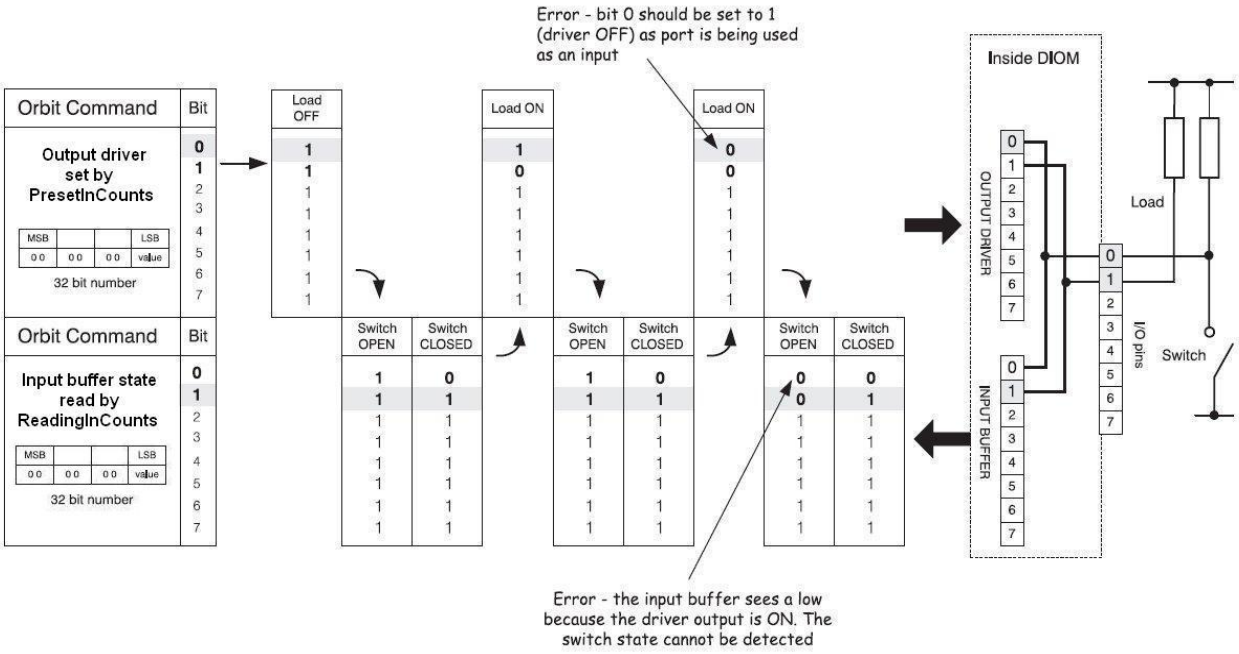
- External electrical interference
- Switch bounce
- Noisy supplies
- Mechanical vibration

Otherwise, a single read may miss the event or may even just see noise.

The DIOM has a built in debounce functionality designed to filter out spurious readings.
The debounce times available (for all inputs) are: 0 (default), 5, 10, 25, 50mS
See DebounceTime - DIOM Module Properties.

12.6.8 DIOM operation example

In the example below, Port 1 (bit 1) is driving a load and Port 0 (bit 0) is an input to a switch. No other port is used in this example.
All example input states are read back correctly except the last, where Port 1 (bit 1) output driver is switched ON, causing a false input state to be read by the Port 1 input buffer.
Note. This last condition is an example of the DIOM monitoring its own output state



Remember: *Input and output ports are internally connected.*
Therefore, for each I/O line, the output driver must be OFF (bit=1) before it can be used as an input.

12.6.9 Legacy

These options are still valid, but the DIOM Pins classes were introduced to simplify / ease implementation. Original information retained here for legacy.
The DIOM uses the following commands to access its common Input/Output bus.

12.6.9.1 Read Inputs

Uses ReadingInCounts property.
The 8 least significant bits of the returned 32-bit number show the state of the input pins.

Each pin MUST first be set high (via PresetInCounts) if it is to be used as an input.
A Low state on the input pin will be returned as a logic Low (0).
A High state on the input pin will be returned as a logic High (1).

12.6.9.2 Set Outputs

Uses PresetInCounts command.

The 8 least significant bits of the 32-bit number are used to set the output pins.

A logic Low (0) will turn the output driver ON; The output pin will be set Low (0 V).
A logic High (1) bit will turn the output driver OFF; the pin will be pulled up to Orbit +5 V via 4K7 and series diode or external load if connected.
The pin MUST be set High if it is to be used as an input.

Note. A Preset of >255 will cause a ‘Preset value out of range (0 to 255)’ exception to be thrown.

12.7 DIGITAL INPUT OUTPUT MODULE V2 (DIOM2)

12.7.1 Introduction

The Digital Input / Output Module V2 (DIOM2) is an enhanced version of the DIOM. Rather than have configurable I/O, it has fixed inputs and outputs and has improved functionality over the DIOM.
The module provides:

- 4 general purpose output lines
  - All outputs default to OFF on power on.
  - Configurable active state per pin
  - Configurable drive mode
- 6 input lines.
  - Configurable active state per pin

Thus, there are 10 pins available in total.

As with the DIOM, OrbitLibrary has specific properties for the Pins to ease usage.

The DIOM2 can be read on command like any other Orbit Module including as part of a dynamic collection.

The DIOM2 can also be used provide synchronization for a dynamic collection (Dynamic External Master Mode). Similarly, it can be used to Externally trigger/sample readings in buffered mode (External Master Mode).

12.7.2 Output Pins

- The hardware circuitry configured (see Output Mode) will also have an effect on the actual state of the pin
- On power on (or after a reset), the DIOM2 de-activates all output pin states (i.e. sets output states to the inverse of the output polarity mask (see Output - Active States).
  - For example, if the output polarity mask for a pin is set to active high and the Output mode is set to TTL (Logic 5V), then the output will be in-active (0 / LOW) on power up.

12.7.3 Active States

Each individual input and output pin has a bit in the associated polarity mask that sets the pin's active state.

12.7.3.1 Outputs

Each output pin has an associated bit in a polarity mask that sets the pin's active state. This is a non-volatile setting (i.e. saved and restored on power-up) that configures whether the pin outputs a '1' (high) or '0' (low) when activated.

The output polarity mask format is:

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Unused	Unused	Unused	Unused	Output4 bit mask	Output3 bit mask	Output2 bit mask	Output1 bit mask

An individual pin's mask operates at follows:

Bit Mask	**Active Value**	**Pin State**
0 (active low)	0	1

0 (active low)	1	0
1 (active high)	0	0
1 (active high)	1	1

Examples

- an output polarity mask set to 0 would set all outputs to be active low (i.e. if the pin is activated low, the output pin state is high)
- an output polarity mask set to 0x0f (00001111b) would set all outputs to be active high (i.e. if the pin is activated low, the output pin state is low).
- an output polarity mask set to 0x03 (00000011b) would set outputs 1 and 2 to be active high and outputs 3 and 4 to be active low.

12.7.3.2 Inputs

Each input pin has an associated bit in a polarity mask that sets the pin's active state. This is a non-volatile setting (i.e. saved and restored on power-up) that configures whether the pin reads a '1' (high) or '0' (low) according to pin value.

The input polarity mask format is:

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Unused	Unused	Input6 bit mask	Input5 bit mask	Input4 bit mask	Input3 bit mask	Input2 bit mask	Input1 bit mask

An individual pin's mask operates at follows:

Bit Mask	**Pin Value**	**Read State**
0 (active low)	0	1
0 (active low)	1	0
1 (active high)	0	0
1 (active high)	1	1

Examples

- an input polarity mask set to 0 would set all inputs to be active low (i.e. if the pin is set low, the read pin state is high)
- an input polarity mask set to 0x3f (00111111b) would set all inputs to be active high (i.e. if the pin is set low, the read pin state is low).
- an input polarity mask set to 0x03 (00000011b) would set inputs 1 and 2 to be active high and inputs 3, 4, 5 and 6 to be active low.

12.7.4 Output Mode

The DIOM output pins can be set to:

- 5V Logic Output (TTL)
- Pull up to Externally applied positive supply (PNP)
- Pull down using Load connected to a positive supply (NPN)

Refer to the Orbit Module Manual (DIOM2 section) for connection details.

Note that setting output mode applies to **all** output pins.

12.7.5 DIOM2 Module Properties

The DIOM2 module class has specific module properties including:

- Pins (a collection of OrbitModuleDIOMPin classes – one for each pin – 10 in total)
  - Pins are assigned as:

Pins[9]	Pins[8]	Pins[7]	Pins[6]	Pins[5]	Pins[4]	Pins[3]	Pins[2]	Pins[1]	Pins[0]
Input6	Input5	Input4	Input3	Input2	Input1	Output4	Output3	Output2	Output1

  - e.g. `DIOMmoduleInstance.Pins[4]` gives the OrbitModuleDIOMPin class for pin 4
- DebounceTime (type `eDiomInputDebounceTime`)
  - An enumeration of possible debounce times for ALL input pins of the DIOM2. Note that the default (power on) DebounceTime value is zero.
- Output Mode – this sets the mode for how output pins are configured see Output Mode.
- Input Polarity Mask – see Inputs
- Output Polarity Mask – see Outputs
- TxSync – used by Dynamic External Master Mode.
- TxSample – used by Buffered External Master Mode.
- Dynamic Master Mode – used by Dynamic External Master Mode.

12.7.6 DIOM2 Pin Members

As for a DIOM, each Pin class object has the following properties:

- `Config` Get / Sets the pin configuration (input or output). Note that DIOM2 has fixed input / output pins, so setting has no effect.
- `InputPinState` Returns the pin state (high or low) in input mode. Note that this setting is irrelevant if the pin is set into output configuration. Read Only.
- `OutputPinState` Gets or sets the pin state (high or low) in output mode. Note that this setting is irrelevant if the pin is set into input configuration.
- `PinNumber` This is the pin I/O number (0 to 9) of the DIOM2.

12.7.7 Read Inputs

Use the Pin.InputPinState property to read. Note that the only DIOM2 input pin states are returned – the output pin states are **not** included. Also, a ReadingInCounts **must** be actioned before readings are read, e.g.

```
int DummyRead = DIOM2moduleInstance.ReadingInCounts;
if (DIOM2moduleInstance.Pins[4].InputPinState ==  eDiomIoPinState.low)
```

This code reads the state of Pin 4. If it is low state, then …

If reading via ReadingInCounts, ReadingInUnits, ReadBurst or as part of a Dynamic collection, then the reading returned will **only** contain the input pin states:

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
0	0	Input6 bit	Input5 bit	Input4 bit	Input3 bit	Input2 bit	Input1 bit

12.7.8 Set Outputs

Use the Pin.OutputPinState property to set outputs e.g.

```
DIOMmoduleInstance.Pins[2].OutputPinState = eDiomIoPinState.low
```

This code sets the state of Pin 2 to low.

12.7.9 Improving Reading Integrity

When reading external inputs, it is wise to issue multiple reads to help reduce the effects of:

- External electrical interference
- Switch bounce
- Noisy supplies
- Mechanical vibration

Otherwise, a single read may miss the event or may even just see noise.

The DIOM2 has a built in debounce functionality designed to filter out spurious readings.
The debounce times available (for all inputs) are: 0 (default), 5, 10, 25, 50mS
See DebounceTime - Module Properties.

12.7.10 External Master Mode

The DIOM2 can also be used provide synchronization for a dynamic collection (Dynamic External Master Mode). Similarly, it can be used to Externally trigger/sample readings in buffered mode (External Master Mode).

If set as an external master, low to high changes on input pin 1 are used as the trigger source.

Notes

- The input pin's active state can be altered by the input polarity mask if desired.
- Whilst acting as an external master, other inputs and output pins cannot be read or set.

12.8 DIGIMATIC INTERFACE MODULE (DIM)

12.8.1 Introduction

The Digimatic Interface Module is designed to connect to any Digital Gauge with a Digimatic code output. The connection to the gauge is via a 10 way male connector, which will connect to any Mitutoyo Digimatic compatible gauge plug.

The reading mode can be continuous, triggered via the gauge data switch or triggered from software. The ReadingInUnits property is used to return the reading.

Before connecting to Orbit, it is important to have the Digital Gauge switched on. This will ensure that the Orbit Library is able to Notify the Module correctly.

12.8.2 Changing the Mode of Operation

When in its default state after power up the DIM will read the gauge continuously. To set other reading modes, change the ReadMode property of the DIM.
See the *OrbitLibrary Code Reference* for details.

12.8.3 Update Reading Information

If the units of measure is changed on the Digimatic gauge itself, then the Orbit Library needs to be alerted to this.
This can be actioned using the 'UpdateInfo' method of the DIM.

12.9 CONFOCAL MODULE

The confocal system is based off reflected light and therefore has properties that can be configured to enable users to get the best reading possible from the device.

12.9.1 Optimising settings

To optimally use the confocal system you want the signal bar to be approximately half to two thirds of the way up when in range on a target, its important to note that due to the nature of the technology signal strength will vary slightly with position and signal saturation will result in loss of accuracy.

To achieve a good signal the integration time and brightness should be set when in range of the target material to be measured, note changing integration will affect the responsiveness of the system so increasing bright levels is often preferable.
The High Precision reading mode provides maintains its accuracy better than the normal precision mode over surfaces with a slight texture (non polished and non glassy) surfaces.

12.9.2 Confocal Properties

Confocal specific device properties can be accessed by casting to the confocal device type eg:

```
OrbitModuleCONFOCAL MyModule = (OrbitModuleCONFOCAL)OrbModules[ModuleIdx];
```

12.9.2.1 Get/Set Integration

Note this property is non volatile – it is remembered through a power cycle

To maintain responsiveness of the system it is often favourable to increase brightness rather than change the integration level, however if the signal level is still low after increasing the bright level it can be adjusted either using the menu or the orbit interface.
The integration time is the exposure of device in milliseconds and can be set by the following method:

```
MyModule.IntegrationTime =  Time_5mS;
```

Possible values (defined in eIntegrationTime): Time_5mS, Time_10mS, Time_20mS, Time_50mS.

12.9.2.2 Get/Set Brightness

Note this property is non volatile – it is remembered through a power cycle
Brightness affects the light output of the device and should be adjusted before integration.
There are 8 predefined levels of light that can be selected (Bright_1 to Bright_8) defined in eBrightness.
The property is set using the following command:

```
MyModule. Brightness = Bright_1;
```

12.9.2.3 Get/Set Read Mode

Note this property is non volatile – it is remembered through a power cycle
There are 2 read modes in the device High Precision and Normal Precision, for highly polished mirror surfaces or glassy surfaces the normal precision is suitable and gives high bandwidth performance. For textured/non polished/non glassy surfaces normal precision mode is less suitable and high precision mode should used instead.

The possible values are: HighPrecision, and NormalPrecision defined in eConfocalReadMode
The property is set using the following command:

```
MyModule. ReadMode = HighPrecision;
```

12.9.2.4 Get/Set Averaging

Note this property is non volatile – it is remembered through a power cycle.

It should be noted that the bandwidth is affected by the read mode and the integration time and averaging , this should be considered when selecting the level of averaging.
There are 9 possible averaging values that can be selected incrementing in powers of 2 from (average1 to average256) defined in eAveraging.

```
MyModule.Averaging =average16;
```

12.9.2.5 Read Second Channel in Units

This property gets the current reading for the second channel if there is one, if there is not a reading of there is a range error, NaN (not a number) is returned.

```
double MyReading = MyModule.ReadingInUnits_B;
```

12.10 LASER TRIANGULATION SENSORS (LT, LTA & LTH)

The Laser Triangulation Sensors provide non contacting measuring to be integrated into the Orbit Network.
There are two types available:

- LT / LTA – Entry level product.
  - See the separate user leaflet (503145), supplied with the product, for operational details.
- LTH – High performance product.
  - See the separate user leaflet (503158), supplied with the product, for operational details.

They are interfaced in the same way as a standard Digital Probe.
As it has similar functionality to the Digital Probe Module, it is compatible with nearly all Orbit Library members, including Averaging and Resolution.

Laser specific configurations such as laser beam on/off (BeamOn) are configured in the OrbitModuleLT / OrbitModuleLTA / OrbitModuleLTH objects.

Overall control of all laser beam states in normal reading mode can be enabled from the Network Object (BeamControl). This sequences the laser beams and provides a stabilisation time (eg turns off the other beams and waits for a stable reading before re-enabling and returning a reading from any specific Laser).

12.10.1 LT Configuration

Although the LT product is used in the same way as a standard Digital Probe, there is a configuration option on the module for scaling the output reading to a different range. Therefore ensure that any software using the Orbit library can accommodate the scale changes. As the output from the Orbit module will be 0 to MaxCount in all cases. (Max count = 16384 for 14 bit resolution, 65536 for 16 bit and 262144 for 18 bit).

Direct enabling and disabling of the laser beam is possible using the BeamOn Property.

12.10.1.1 Reading and Writing Settings

```
OrbitModuleLT lt = (OrbitModuleLT)Orbit.Networks[0].Modules[0];
lt.BeamOn = false; // to turn the beam off
```

12.10.2 LTA Configuration

The LTA product is used in the same way as a standard Digital Probe.

Direct enabling and disabling of the laser beam is possible using the BeamOn Property.

12.10.2.1 Reading and Writing Settings

```
OrbitModuleLTA lta = (OrbitModuleLTA)Orbit.Networks[0].Modules[0];
lta.BeamOn = false; // to turn the beam off
```

12.10.3 LTH Configuration

The LTH has a number of additional readable and writeable configuration settings. The key settings being the Filter rate and the Level Cut Time; both of which can be read and set. There are also readable settings; Model Number, Serial Number and the Firmware Version of the laser unit.

12.10.3.1 Reading and Writing Settings

A setting specific to an LTH can be modified or read by first casting the LTH OrbitModule object connect to a network to an OrbitModuleLTH:

```
OrbitModuleLTH lth = (OrbitModuleLTH)Orbit.Networks[0].Modules[0];
```

The OrbitModuleLTH object will then make available LTH specific settings:

```
lth.LevelCutTime = 10000; // 100ms
```

See the OrbitLibraryTest's ModuleLTHUC class for a more detailed example.

12.11 WIRELESS CONNECTION MODULE (WCM)

The WCM enables Wireless handtool devices to be connected to the Orbit system.

- See the Orbit3 Module Manual for WCM installation, configuration and operational details.

The WCM Module class in the OrbitLibrary has various properties and methods, as summarised, next:

Refer to the OrbitLibrary Code UML Diagram for a hierarchical view of the WCM Module class.

Individual Devices (up to 6 per WCM) are accessed by the Devices[] property
This returns a WCM_Device class object.

e.g. WCM_Module.Devices[<DeviceIndex>].*****
Where DeviceIndex is the device in question (0 to 5) and ***** is the device's property to access

For an example, refer to the Orbit3 Code Examples.

12.11.1 Compatibility

To add a WCM to an Orbit network, use PING or manually enter its ID, as it does not support the Orbit Notify feature.

12.11.2 WCM_Device class

This is a separate class that encapsulates a device's capabilities. This has the following members:

12.11.2.1 Properties

- Type – the device type connect to (e.g. None, WHT, WHT-M)
- Name – 20 character Device 'Name' - user configurable (set via WCM configuration application)
- OrbitIdentity – This is the configured OrbitIdentity that the WCM will try to connect to (set via WCM configuration application).
- NumChannels – The number of channels for this device
- ConnectionStatus – Device status (not connected, connecting etc)
- BatteryStatus – % battery remaining for the device (0 to 100)
- UOM – Units Of Measure (e.g. mm, Inches, etc.)
- Settings class

12.11.2.2 Methods

- Read(...) – see Reading Devices

- ReadAllChannels(…) – see Reading Devices
- GetChannelData(int ChannelIndex)
  - Returns a class object containing the device's Channel data information (stroke, Orbit ID) for the passed channel index

12.11.2.3 Reading Devices

The WCM, itself has no valid reading. Instead, it has 6 devices, each with its own channels. Each device is read independently.

- Reading a device on the WCM will trigger a connection to that device from the WCM (unless already connected).
- To read all channels for a device:

```
OrbitChannelReadings rdgs = OrbitModuleWCM.Devices[<DeviceIndex>].ReadAllChannels(ControlCode, UOM)
```

- To read specific channels of a device, use a Channel Mask

```
OrbitChannelReadings rdgs = OrbitModuleWCM.Devices[<DeviceIndex>].Read(ChannelMask, ControlCode, UOM)
```

- Reading results are returned in an OrbitChannelReadings class.
  - Note that each channel read has its own individual reading and status.

12.11.2.3.1 OrbitChannelReadings

This class is returned from device read () calls. It contains a collection of OrbitChannelReading class objects – one for each channel requested in the device read() channel mask (eg. for a WHT-M with 8 channels there will be 8 OrbitChannelReading class objects).

12.11.2.3.2 OrbitChannelReading

This encapsulates the reading for an individual channel.
It has the following properties:

- int ChannelIndex // 1 based channel index – i.e. which channel this class pertains to
- int ReadingInt // The reading expressed as a 32 bit scaled integer = 7DPs or 10^7 * Reading
- double Reading // The reading expressed as a floating point number
- eOrbitErrors Status // Status of reading – expressed as an enumeration of possible errors.
- Tag Number – Tag number for tagged reading. Not used for normal (i.e. not tagged) readings.
- PZA Mode – this details which measurement mode the device is set to (Preset, Zero or Absolute). PZA mode is set by a Device Setting.
- OutputMode – this details which reading mode the device is set to (Normal, Max, Min etc.). Output mode is set by a Device Setting.
- LimitStatus – this details whether the reading is outside limits (if that Device Setting is enabled)
- Battery Status – this is percentage of battery remaining for the channel.

12.11.2.3.3 Details

To configure whether to obtain normal or tagged readings, use the Control Code parameter.
For normal readings:

- If not already connected, a <Reading Holdoff error> will be returned in the reading status (until a new reading is available).
- The user is advised to poll readings until the <Reading Holdoff error> has cleared - indicating the device has successfully connected and there are valid readings.

- A device's readings are deemed valid by the WCM if they are retrieved within the <MaxReadingAge> milliseconds limit. If too old:
  - A new reading is requested (this will connect to the device, if necessary)
  - A <Reading Holdoff error> will be returned in the reading status (until a new reading is available).
  - The user is advised to poll readings until the <Reading Holdoff error> has cleared - indicating valid readings
- Tag number returned will always be zero.

For tagged readings:
- Tag number is the tag assigned to this reading
  - Note that this will be the same for all channels on this device.

Note that if a standard Orbit read (e.g. ReadInCounts, ReadInUnits) is requested from the WCM module itself (which has no reading), zero is returned, rather than an error. This is to avoid errors (i.e. does nor ‘break’ the Orbit network) when reading a previously working Orbit network that now has a WCM added to it.

12.11.2.3.4 Control Code

This code determines what data is read and in what format it is returned in.

Value	**Meaning**
0	Invalid - must have a control code.
1	Normal Reading - returns a structure containing reading and status data for up to 12 channels - populated with valid channel data and 'invalid channel' status for channels not in use. The channels to be read are specified by the Channel Mask parameter. Format of reading data is floating point. Single channel devices should specify channel 1 only.
2 (Default)	Same as '1' with format as signed 32 bit integer with implied 7 decimal places.
3	Same as ‘1’ but using Tagged Readings. If no tag has been taken yet, a 'NoTag' status code is returned instead.
4	Same as '3' with format as signed 32 bit integer with implied 7 decimal places.

12.11.2.3.5 Channel Mask

In order to return a combination of any of the (up to) 12 channels (multi channel devices only), a 16 bit-wise mask is provided.
- On single channel devices (e.g. WHT), this parameter should always be set to channel 1.
- Any channels requested via the mask, that are not available will return a ‘Invalid Channel’ reading status.

Bit	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Channel	CH12	CH11	CH10	CH9	CH8	CH7	CH6	CH5	CH4	CH3	CH2	CH1	N/A

Channels 0, 13 and above are not used – reserved for future use

Examples:
- To select Channel 3 only, set Bit 3 = 8

- To select Channels 1 & 3, set Bits 1 & 3
- To select All Channels, set Bits 1 to 12 = 8191

12.11.2.3.6 Reading Format

The reading is returned either as a:

- 4 byte integer (32 bit) with implied 7 decimal places
  - This has maximum value of ±214.7483647 (231/10,000,000). Readings of more than these values should use floating point number format.
  - A reading exceeding this maximum value will return a “Reading Overflow” error. In this case, use the Single precision number (floating point) method.
  - Example: 10mm stroke = 393.7 mils. In UOM of mils, the reading would overflow above the 5.45mm reading point.
- 4 byte Single precision number (floating point) – as per IEEE 754 single-precision binary floating-point format.
  - This has no maximum value. But loses precision with higher numbers.

The reading format is determined via the Control Code parameter.
The Units Of Measure is determined via the Units Of Measure parameter.

12.11.2.3.7 Units Of Measure

This allows the reading Units Of Measure (UOM) to be altered.

Value	**Meaning**
1 (Default)	Returns the normal UOM for the module. (eg. mm)
2	Inches = mm / 25.4
3	Mil = mm * 1000 / 25.4

12.11.2.4 Settings class

In order to be able to change device settings (e.g. WHT button functionality, WHT preset / zero etc.), a “Settings” class property is provided in the WCM_Device class.
This class property gives access to the settings available for that device. For a complete list of available settings, refer to the Orbit Module manual (“WCM – Device Settings” section), *the OrbitLibrary Code Reference* and the OrbitLibrary Code UML Diagram.
In order to change a setting, the WCM Device needs to be entered into 'ConfigMode'.

12.11.2.4.1 Config Mode

This is a special mode that each WCM device is set to in order to read or write to (configuration) settings.
To enter *config* mode, simply set the boolean WCM device's *Settings.ConfigMode* to true (false to exit). If the device is not connected, setting *ConfigMode* may take some time to complete (as in initiates a connection).

```
//Enter the device's config mode
OrbitModuleWCM.Devices[<DeviceIndex>].Settings.ConfigMode = true;
//  Change setting(s)
...
//Exit the device's config mode
OrbitModuleWCM.Devices[<DeviceIndex>].Settings.ConfigMode = false;
```

If the device is not in *config* mode already, reading / writing a setting will automatically enter (and subsequently exit) *config* mode. This is effective for a single setting, but is time consuming when changing multiple settings. For this reason, it is recommended to use the manual entering and exiting of *config* mode for multiple setting changes.

The WCM Settings class is different for WHT and WHT-M devices. However, they share the same base class. Therefore, it is necessary to check the device Type field to work out the appropriate class to cast the settings to – as detailed in the code snippet, next:

```
if (OrbitModuleWCM.Devices[<DeviceIndex>].Type == eWCM_DeviceType.WHT)
{
        //CAST settings class for WHT
        WCM_DeviceSettings_WHT Settings =
(WCM_DeviceSettings_WHT)OrbitModuleWCM.Devices[<DeviceIndex>].Settings;
        //Enter Config mode, change setting(s) and Exit Config mode
        Settings.ConfigMode = true;
        [change other settings]
        Settings.ConfigMode = false;
}
else if (OrbitModuleWCM.Devices[<DeviceIndex>].Type == eWCM_DeviceType.WHT_M)
{
        //CAST settings class for WHT-M
        WCM_DeviceSettings_WHTM Settings =
(WCM_DeviceSettings_WHTM)OrbitModuleWCM.Devices[<DeviceIndex>].Settings;
        //Enter Config mode, change setting(s) and Exit Config mode
        Settings.ConfigMode = true;
        [change settings]
        Settings.ConfigMode = false;
}
```

12.11.2.4.2 Send Command

A method named SendCommand allows individual device settings that don't have corresponding properties in the DeviceSettings class to be changed. An example of this would be if the user wished to change the Limits and Direction settings on a Wireless Hand Tool.

```
if (OrbitModuleWCM.Devices[<DeviceIndex>].Type == eWCM_DeviceType.WHT)
{
        //CAST settings class for WHT
        WCM_DeviceSettings_WHT Settings =
(WCM_DeviceSettings_WHT)OrbitModuleWCM.Devices[<DeviceIndex>].Settings;

        // Send low level command to set direction
        Settings.SendCommand("SET DIRECTION 0");

        // Send low level command to set upper limit
        Settings.SendCommand("SET LIMIT UP 004.0000");

        // Send low level command to set lower limit
        Settings.SendCommand("SET LIMIT LO 001.5000");
}
```

The method takes a command string as an argument. Valid commands are detailed in the documentation included in the Wireless Support Pack.

12.11.3 Tagged readings Example

For this function, the devices must be configured to provide tagged readings when pressing a button – refer to WCM Device Configuration Settings in the Orbit Module manual.

User software should first request current tagged reading from the WCM to obtain the current 'Tag number'.
The tagged reading data should then be polled until the 'Tag number' changes – the data obtained when the tag number changes is the new tag reading.

i.e.

1. Read current tagged data, note “TagNumberAtStart”
2. (STARTLOOP)
3. Read current tagged data
4. If TagNumber within data different to TagNumberAtStart?
    a. **Save Reading as next ‘new’ Tagged Reading**.
    b. Another tagged reading required?
        i. Yes
            1. Save TagNumber within data to TagNumberAtStart
            2. Go to STARTLOOP.
        ii. No
            1. Exit / Done
5. Else
    a. Go to STARTLOOP.

Refer to Reading Devices for details.

12.12 SINGLE CHANNEL CONDITIONER MODULE (SC1-A & SCD1-A)

This is a separate module with its own USB interface that allows readings to be obtained from a standard gauging probe (LVDT or Half-bridge). Although this does not conform to a standard Orbit module or controller, it connects to Orbit software via the OrbitLibrary and is treated as an Orbit controller with one module. See SCD1-A user manual (503899) for more details.

13 ORBIT ERROR CODES AND ERROR HANDLING

13.1 GENERAL

When communicating with the Orbit Library, errors are returned via exceptions.

Errors can be broken down into the following sections:

- Orbit Library Errors

The information passed to the Library is incorrect or not in the required format – or the requested function is not allowed in the current state.

- Orbit Controller Errors

The command failed to send from the Orbit Controller.

- Module Errors
  The module has been asked to perform a function that is not allowed or the module has a fault.

The Orbit Library decodes the returned errors and throws an exception to the user application with a meaningful text (Exception.Message property)of the error. In this way, the user application avoids having to decode the actual error code.

13.2 HANDLING ERRORS

13.2.1 Error Handling When Using the Orbit Library

An error will generate an exception in the Orbit Library. In C# and C++ , a try and catch block surrounding the call will 'handle' the exception.

Refer to the Orbit3 Code Examples for examples on error handling with the Orbit Library.

13.3 COMMON ERRORS

Below are listed common error codes.

13.3.1 No Error

No exception thrown implies that the requested command has been successful.

13.3.2 Under and Over Range

Digital Probe, LT, LTA, LTH, AGM and AIM only

Under and over range are special cases of errors. These are ‘soft’ errors and thus will not cause an exception to be thrown from the Orbit Library.
Instead, the ModuleStatus object should be used.

13.3.3 Overspeed Error

Linear Encoder only
This error is returned when the LE has been moved too fast. The readings (incremental) are now suspect and thus should be reset. The Orbit Library will raise an exception for this error.

13.3.4 No Probe Error

Dp only
This error is generated if the probe has been disconnected from its PIE module.
See Probe disconnect detection.

13.4 MODULESTATUS

This object is used to read the module's status and error conditions (e.g. under and over range for a DP / LT / LTA / LTH / AIM / AGM).

Note that ModuleStatus.Error and ModuleStatus.ErrorString are updated automatically when reading a module.

Refer to the Orbit Library Test application for example code.

13.5 ORBIT ERRORS

Error	Value (Hex)	Value (Decimal)	Description
NoError		0	No Errors On the orbit Module
ORBIT_5V_CURRENT_LIMIT	0xF	209	Controller is current limiting! PSIM required or short on network
ORBIT_5V_LOW		210	Orbit Low Voltage - PSIM required
ORBIT_5V_HIGH		211	Warning Orbit Supply voltage is too high
BLOCK_CHECKSUM_ERROR		224	Block Checksum error
XRAM_CHECKSUM_ERROR		225	XRAM Checksum Error - Fault on external memory
BAD_START_ADDRESS		226	Bad Start Address
FLASH_VERIFY_ERROR		227	Flash Verify Error
DYNAMIC_BUFFER_UNDERRUN		241	Dynamic Buffer Underrun This error indicates that windows has not been able to read the buffered data for longer than the controller has space to buffer the continuous dynamic data
DynamicStopped		242	This indicates that dynamic is stopped
SYNC_TIMING_VIOLATION		250	Sync Timing Violation This occurs when a module is acting as master for the dynamic collection and its trigger source requires it to send a sync before the previous data has finished being sent.

FRAMING_ERROR		251	Framing Error On RS485 Communications
OVERRUN_ERROR		252	Overrun Error On RS485 Communications
Checksum		253	Checksum Error on buffered readings
Parity		254	Parity Error On RS485 Communications
Timeout		255	Orbit Command Timeout Error
Empty		4194304	Empty - No Dynamic Data Here
ModuleBase		8448	Base Part of module error code - error detailed in lease significant byte
COIL_RANGE		8450	Coil Frequency out of range
NO_CMD		8451	Unknown Orbit Command
BCAST_NA		8452	Broadcast Address Not Allowed (Legacy)
ADDR_NA		8453	Broadcast Address 0 Expected (Legacy)
ADDR_CH_NA		8454	Address Change not allowed (in difference/buffered or dynamic mode)
WRONG_MODE		8455	Cal Mode Pin High (Legacy)
NO_CALTABLE		8456	No Calibration Table - Probe uncalibrated
MISSED_RDG		8457	Reading Missed - New Adc cycle started before previous reading read (Legacy)
RDG_HOLDOFF		8458	Reading Holdoff - Reading not valid - waiting for measurement
FRAMING_ERR		8459	(Legacy)
MODULEINFO_NEEDS_REREAD	0x210C	8460	Module Information needs reading
GT16BIT		8465	Error - Greater than 16 bits between calibration points - bad cal table
UnderRange		8466	Under Range - Probe outside of normal reading range
OverRange		8467	Over Range - Probe outside of normal reading range
MULT_OVERFLOW		8468	Calibration - Multiply overflow error - bad cal table
BUFFER_FULL		8470	Buffer Full
RDBFR_NOT_VALID		8471	Read Buffer Not Valid
READ_NOT_VALID		8472	Read Not Valid
BFR_RD_WR_ERR		8473	Buffer Read Write Error
BFR_NOT_EMPTY		8474	Buffer Not Empty
BFR_EMPTY_STOP		8475	Buffer is empty and probe is stopped
DIFF_NOT_SET		8481	Difference Mode Not Set

DIFF_WAITING		8482	Difference waiting for triggered to be set
DIFFMODE_NA		8483	Difference mode not allowed acquire flag set (Obsolete)
NUM_OVERFLOW		8484	Difference count overflowed (greater than 3 bytes)
SUM_OVERFLOW		8485	Difference sum overflowed (greater than 5 bytes)
DIFF_RUNNING		8486	Difference mode set and running
DIFF_NA_HIRES		8487	Difference mode not allowed at this resolution
ARGUMENT_NOT_VALID		8511	Argument Not Valid
MODE_NOT_VALID		8512	Mode Not Valid
DELAY_NOT_VALID		8513	
SYNC_WAITING		8514	Sync Waiting
SYNC_RUNNING		8515	Buffered Sync Running
SAMPLE_RUNNING		8529	Buffered Sample Running
AVERAGE_NOT_VALID		8544	Average Not Valid
AVE_CHANGE_NOT_ALLOWED		8545	Average Change Not Allowed
RESOLUTION_NOT_VALID		8546	Resolution Not Valid
RESO_CHANGE_NOT_ALLOWED		8547	Resolution change not allowed
PROBE_SET_TO_HI_RES		8548	Probe set to High Resolution
PROBE_SET_TO_LO_RES		8549	Probe Set to Low Resolution
NOT_IN_NORMAL_MODE		8550	Not in Normal Mode
ADDR_RANGE_ERROR_DYN		8551	Dynamic - Address Range Error
NOT_IN_HIGH_BAUD		8552	Error Not in high baud rate
INPUT_TYPE_NOT_VALID		8560	Input Type Not Valid
QUAD_CODE_NOT_VALID		8561	Quadrature code not valid
REF_ACTION_NOT_VALID		8562	Reference Action Not Valid
PRESET_OUT_OF_RANGE		8565	Preset out of range
PRESET_IGNORED		8566	Preset Ignored
ENCODER_ERROR		8567	Encoder error
SYNC_GAP_ERROR		8568	Dynamic Sync Gap Error
CONTROL_GAP_ERROR		8569	Control Gap Error
Voltage_Error_5V_LOW_WARNING		8592	Orbit 5V Low Warning
Voltage_Error_5V_TOO_LOW		8593	Orbit 5V Too Low Error
Voltage_Error_5V_TOO_HIGH		8594	Orbit 5V Too High Error
Voltage_Error_5V_HIGH_WARNING		8596	Orbit 5V High Warning
ADC_FAULT		8595	ADC Fault
I2C Memory Fault		8598	I2C communications failed to EEPROM memory
I2C Accelerometer Fault		8599	I2C communications failed to the accelerometer chip
I2C Pressure Sensor Fault		8600	I2C communications failed to pressure sensor.
Inph_Low		8624	In Phase Low
Inph_High		8625	In Phase High

Quad_Low		8626	Quadrature Low
Quad_High		8627	Quadrature High
Counter_Error		8628	Counter Error
Ref_running		8643	Refmark mode set and running
Too_fast		8644	Too Fast - Over speed Error
Low_signal		8645	Low Signal Flag
INCOMPATIBLE_BOARD		8688	Incompatible Board
INCOMPATIBLE_STROKE		8689	Incompatible Stroke
INCOMPATIBLE_COMPANY		8690	Incompatible Company
BAD_CALIBRATION_DATA		8691	Bad Calibration Data
INCOMPATIBLE_HOTSWAP		8692	Incompatible Hotswap
ONEWIRE_HOTSWAP		8693	One Wire Hotswap Error
INCOMPATIBLE_PROBE		8694	Incompatible Probe
NO_PROBE		8695	No probe connected to PIE case
INCORRECT_COMMAND_FORMAT		8696	Incorrect command format
COMMUNICATIONS_ERROR		8697	Communications Error
OverAndUnderRange		65537	Over And Under Range (Difference Mode)
Unknown		65538	Unknown error

14 APPENDIX A - ORBIT COMPATIBILITY ROADMAP

The Orbit system has evolved over the years and it is now in its third generation. Some features will only be available with newer hardware and some features have been removed over time.
A summary of the Orbit commands and hardware with their usage is provided on the following pages.

14.1 MODULES

14.1.1 Orbit3

The Orbit Library provides a totally different software interface to the original Orbit COM and DLL libraries. Therefore a simple comparison of commands is difficult. Generally, it provides the same level of commands, but with a much simplified user interface.
For the full list of Orbit1,2, & Orbit3 Orbit COM library commands, see the original Orbit3 Software manual.

Key features with Orbit3 products are:

- Readburst
- Dynamic2
- Orbit Ultraspeed (2.25MBaud)
- Hotswap

14.1.2 Module Compatibility

The Orbit Library handles compatibilities of all versions of each Module type, from Orbit1 through to present day.
The table below details which features are available with the various Modules and legacy products.
Where Orbit Modules are denoted by a number, i.e.

- '1' denotes Orbit1 Modules
- '2' denotes Orbit2 Modules
- '3' denotes Orbit3 Modules

Note. The table only covers the features that do not work on all Module types.

Feature	DP	LT	LTA	LTH	AIM	AGM	LE	DIOM	DIOM2	EIM	WCM	DIM
Orbit Standard Speed (187.5k Baud)	1,2,3	3	3	3	2,3	3	1,2,3	1,2,3	3	2,3	3	2,3
Orbit High Speed (1.5M Baud)	2,3	3	3	3	2,3	3	3	2,3	3	2,3	3	2,3
Orbit Ultra Speed (2.25M Baud)	3	3	3	3	3	3	3	3	3	3	3	3
ReadBurst	3	3	3	3	3	3	3	3	3	3	3	3
Dynamic	2,3	3	3	3	2,3	3	3	2**,3	3	2,3	3	
Dynamic2	3	3	3	3	3	3	3	3	3	3	3	3
Buffered *	1*,2*,3	3	3	3	2*,3	3		3	3			
Difference	1,2,3	3	3	3	2,3	3	1,2,3					
HotSwap	3	3	3	3	3	3	3	3	3	3	3	3
Ping	3	3	3	3	3	3	3	3	3	3	3	3
Resolution	2,3	3	3	3	2,3	3						
Averaging	2,3	3	3	3	2,3	3						
Preset							1,2,3	1,2,3	3	2,3		
Ref Mark							1,2,3					
Ref Action										2,3		
Quadrature										2,3		
Direction							1,2,3					
External Master									3	2,3		
TCON clear	3	3	3	3	3	3	3	3	3	3	3	3
Firmware upgradeable	2,3	3	3	3	2,3	3	3	2**,3	3	2,3	3	2,3
Diagnostic / Status Leds	3	3	3	3	3	3	3	3	3	3	3	3
200 Modules per network	3	3	3	3	3	3	3	3	3	3	3	3

Notes.
* Pre Orbit3, Buffered Mode was a factory installed option only.
** Orbit2 DIOM was only Dynamic capable and Firmware upgradable from 2008 onwards.

14.1.3 Module Release History

This section details when particular Modules were introduced.

Orbit1
DP, LE, DIOM

Orbit2
DP, LE, DIOM, AIM, EIM, DIM

Orbit3
DP, LT, LTA, LTH, LE, DIOM, AIM, AGM, EIM, DIM, DIOM2, WCM

14.2 CONTROLLERS & SOFTWARE

The following Orbit Controllers are all compatible with the Orbit Library.

Hardware / Mode	RS232IM MK2	RS485IM MK2	WIM	USBIM MK2 & USBIM LITE	ETHIM 2.0	ETHIM
Standard Orbit Speed (187500)	✓	✓	✓	✓	✓	✓
Orbit High Speed (1.5M)	✓	✓	✓	✓	✓	✓
Orbit Ultra Speed (2.25M)				✓	✓	
Buffered Mode	✓	✓	✓	✓	✓	✓
Readburst	✓	✓	✓	✓	✓	✓
Dynamic Mode				✓	✓	
Dynamic2				✓	✓	
Orbit Ping Command	✓	✓	✓	✓	✓	✓
Maximum Modules per network	200	200	200	USBIM MK2 = 200 USBIM LITE = 4	200	200
Software						
Orbit Library	✓	✓	✓	✓	✓	✓
Serial Command Capable	✓	✓	✓ Virtual Com Port	✓ Virtual Com Port	✓ Via 'Sockets'	✓ Via 'Sockets'

Note.
The above Orbit Controllers are also compatible with the Orbit COM/DLL Libraries, but operate with reduced functionality.

The following Orbit Controllers are not compatible:
Orbit PCI Card, PC104 Card & USBIM MK1.

15 REVISION HISTORY

REVISION	DATE	COMMENTS
1	16/06/11	Initial Issue
2	30/09/11	References to .NET updated
3	14/11/12	Linear Encoder (LE) added
4	20/05/13	Orbit high performance Laser Triangulation (LTH) added
5	23/10/13	Added setting of default Averaging & Resolution
6	04/11/13	LT & LTH Laser beam on/off added
7	22/11/13	Wireless Interface Module (WIM) added
8	09/03/15	Added Confocal Module
9	04/08/15	Added Orbit3 Excel® add-in & connecting to WIM Controllers method
10	18/09/15	References to 'menu screen' removed as no longer provided
11	02/11/15	Added Orbit3 Confocal Updater
12	06/11/15	Added Confocal Configuration
13	20/09/16	Added C# example project reference
14	14/10/16	Added AllowOSSuspend reference 10.2.2
15	12/04/17	Added 12.1.4.Probe disconnect detection
16	17/05/17	Updated WIM Controllers - Walk through and WIM Controller reading rates
17	21/06/17	Updated WIM Controllers - Walk through
18	02/10/17	Minor updates
19	19/01/18	Added Wireless Connection Module (WCM)
20	23/03/18	Added DIOM Configuration & updated DIOM section
21	01/06/18	USBIM LITE added
22	03/10/18	WCM Device Settings class added WCM OrbitChannelReading section updated RefAction note added to EIM Module Properties
23	24/10/18	Added reference to DIOM input debounce in 12.5.7 Improving Reading Integrity
24	07/11/18	Added Battery Status to WCM OrbitChannelReading
25	27/02/19	DIOM2 added. Buffered Mode walkthrough example improved.
26	26/03/19	Air Gauge Module (AGM) added.
27	23/04/19	AGM Mastering ‘end-bands’ description added.
28	22/05/19	WCM_Device Settings SendCommand added.
29	28/06/19	Orbit GCS added & OrbMeasure Lite retired.
30	23/10/19	AGM details updated – limits added and end band region changed.
31	23/12/19	Update for the Orbit Library to provide true AGM readings.
32	15/01/20	Updated to reflect AGM ‘EndBand’ increase to 30microns.
33	20/07/20	Number of Orbit Modules on a controller increased to 200 from 150
34	01/09/20	Removed “MaxTagAge” from Control Code 3 from WCM options
35	12/01/20	ETHIM 2.0 added
36	27/07/21	Update to AGM mastering.
37	29/10/21	Orbit Laser Triangulation (LTA) added
38	28/11/22	Reference to LTA Laser Utility added
39	28/11/23	Reference to SC1-A and SCD1-A added
40	27/08/24	TOC hyperlinks fixed.