

SIMPLified 2 User Guide

SIMPL Windows Programming Assistant

Contents

Preface - The Origins of SIMPLified 23

Feature Overview4

Getting Started - Loading a SIMPL Program or Module.....5

Viewing a Program's Device Tree7

Settings8

Adding Entitlements 10

Program Analysis..... 11

Refactoring a SIMPL program.....21

Analysis Toast Notifications30

Analyser Unlocked Pop-up.....31

Design.....32

Debugging33

Console Window41

Productivity.....42

Program Generation45

Command Line Compiler.....45

Key Bindings.....46

Roadmap47

Release History48

Preface - The Origins of SIMPLified 2

SIMPL (sometimes with, sometimes without the Windows suffix) has been the cornerstone of Crestron programming for over a decade, and while it is certainly mature in a fast moving, technological world, it still represents the finest mechanism for programming control systems for a large majority of Crestron professionals.

There are newer, more sophisticated, tools for developing Crestron programs and many traditional developers have made the transition to the new environments and, at the other end of the scale, there are new iterations of the "automatic" programming tools that promise the possibility of developing complex systems without the need for a programmer at all – however – we believe that SIMPL continues to occupy the optimal position between technical complexity and expressiveness for most practical applications. The beauty of Crestron is in its flexibility and, sadly, automated/templated solutions often stifle that flexibility, while programs built entirely within the .NET environment must balance ultimate power with a significantly heavier cost of ownership/maintenance.

SIMPL's flexibility comes at a cost however. It is easy to develop programs that are difficult to maintain in many different ways, from inconsistent naming conventions and convoluted logic, to swamped warnings and unrecognised slips. At best, some programs are simply hard to follow and at worst, there can simply be functional bugs that defy detection – usually leading to yet more convoluted logic which does more to hide the issue than resolve the underlying problem.

If you have had cause to maintain code from another dealer, you may well sympathise with the comments above, and this was our motivation behind the development of SIMPLified 2.

SIMPLified was an internal project that proved the concept that would later become SIMPLified 2 – a complementary tool to SIMPL, specifically targeted towards programmers that want to develop clean, efficient, reliable, maintainable SIMPL programs.

From that proof of concept, SIMPLified 2 has become a rapidly expanding set of analytical and refactoring tools that can quickly identify potential issues in a SIMPL program or module, outside of the standard SIMPL workflow which generally requires a time-consuming compile step before even the most obvious issues are exposed.

If all of your programs are perfect, then SIMPLified 2 isn't going to give you anything useful, but for the rest of us fallible programmers, we hope you find the insights that SIMPLified 2 can provide an invaluable tool that will both save you time in diagnosing issues, and ultimately allow you to produce better Crestron systems.

We have a growing list of feature ideas for SIMPLified 2 that we'll continue to implement and we welcome ideas from fellow Crestron programmers – writing better software is in everyone's interest.

Feature Overview

SIMPLified 2 is a free application with limited functionality. To realise the full potential, a number of “entitlements” can be purchased on an annual subscription. While the free version is still useful in terms of mirroring some of the metrics that SIMPL Windows already provides, it's these additional entitlements that unlock the power of the full application.

Each entitlement covers functionality related to a specific task. As a feature is added to SIMPLified 2, it will be included under one of the entitlements and any with a valid entitlement will automatically benefit from the feature.

The following list describes each entitlement – some of which exist, and some are planned.

Base

The base entitlement covers the basic, free, functionality. This is essentially program loading, a small number of program analysers and the basic program score.

Debugging

The debugging entitlement is where functionality related to diagnosing programming issues will reside. This includes analysers that are likely to contribute to errors in functionality and includes our SIMPLscope feature for debugging signals in real time.

Refactoring

Refactoring is the process of refining existing code without changing functionality. This includes functionality such as the DIFF tool, and Module Argument Extraction and analysers that will help identify redundant logic for removal. This also includes the Search and Replace feature for replacing entire modules in a program all at once.

Commissioning (proposed)

The commissioning entitlement will include tools to aid commissioning engineers, such as reporting, device discovery and enhanced console features.

Productivity

This entitlement targets programming productivity to help minimise the time consuming processes that tie up the development environment. You will lose coffee break time at the expense of getting more done in a shorter time. The productivity entitlement enables exporting branches of logic and hardware to reuse in another project. Plus our Program Generation feature.

Design (proposed)

The design entitlement will target measuring program design such as standards compliance, and also document program structure. Some of these features may overlap with other entitlements – for example, our vision is to combine debugging +

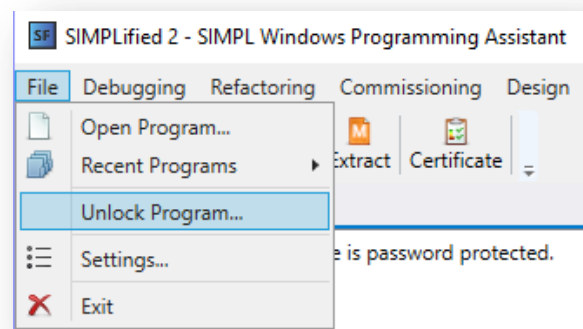
design to provide a graphical program flow and superimpose real-time debug information.

Getting Started - Loading a SIMPL Program or Module

To begin; you will want to load an existing SIMPL Windows program. From the File menu, select Open Program... and choose the file you're working on.

SIMPLified will load the program, look for any dependent modules and once complete, show the program tree in a new window. You can pick up this window from the title bar and drag it anywhere on your desktop, which can be very convenient in multi-monitor setups.

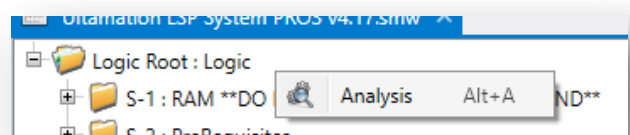
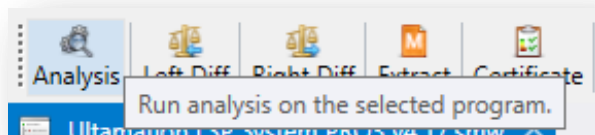
From the program window you can browse through the structure in much the same way as the Program View of SIMPL Windows. If the program you are loading is password protected, you will not be able to see anything beyond the Logic Root node of the tree. To unlock the program for browsing and analysis, you must first provide the correct password with the option shown to the right.



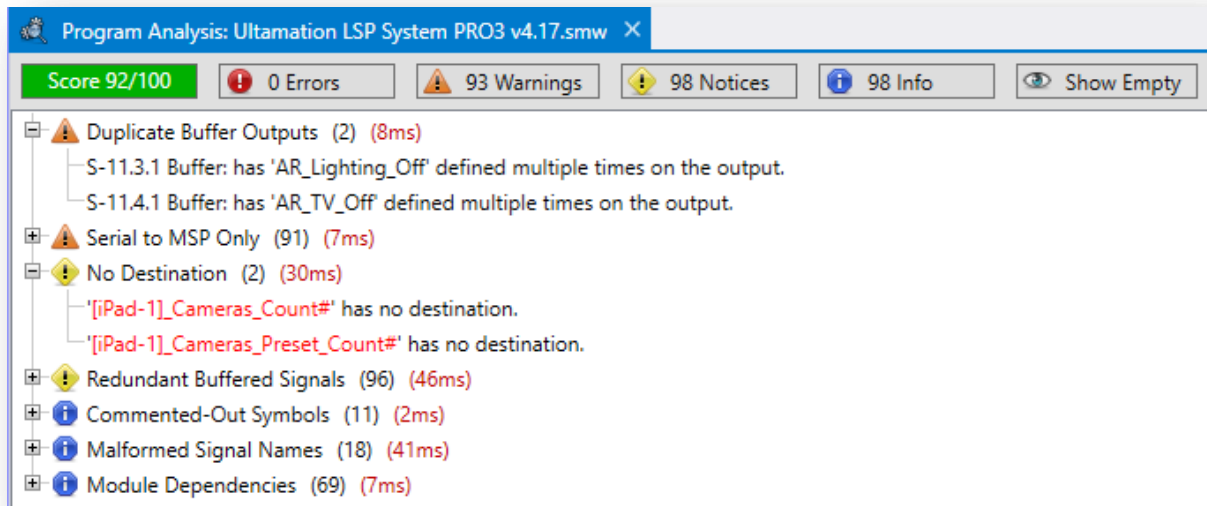
Naturally, if you cannot provide the correct password, SIMPLified's functions will not provide any information about the internal structure of the program to protect the original programmer's intellectual property.

Remember; if you have one of your own modules that is protected and you have forgotten the password, you will always have an unencrypted .BAK file available to fall back on. If you don't have this file, then it's time to reconsider your archival strategy – SIMPLified will not provide passwords for you.

Once you have an unprotected program, the analysis functions will be available. A number of analysers are included to provide insight into various elements of the program. These analysers are described in more detail in the subsequent section.



This will open a new window beneath the program tree showing the results from each analyser. Analysers with zero results will not be shown, and those with 10 results or less will be expanded by default. You can see all of the available analysers, whether they have results or not, by clicking on the "Show Empty" button.



As you work on your SIMPL program in SIMPL Windows, each time you save your work, SIMPLified will reload the program and re-run all analysers to provide an up-to-date summary of the program.

This allows you to resolve such things as signal driving source and destination issues, or duplicate cross-point ids without having to compile the program.

Closing the program tree pane will also close any related windows.

Since one of the motivations behind SIMPLified is to improve efficiency we have tried to make invocation of each function as simple as possible. Therefore, you can trigger the analysis and most other functions in any of the following ways:

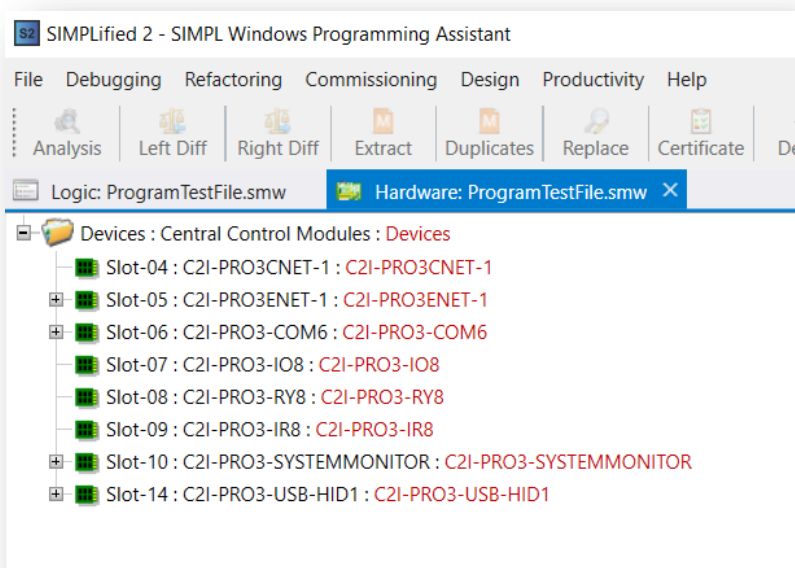
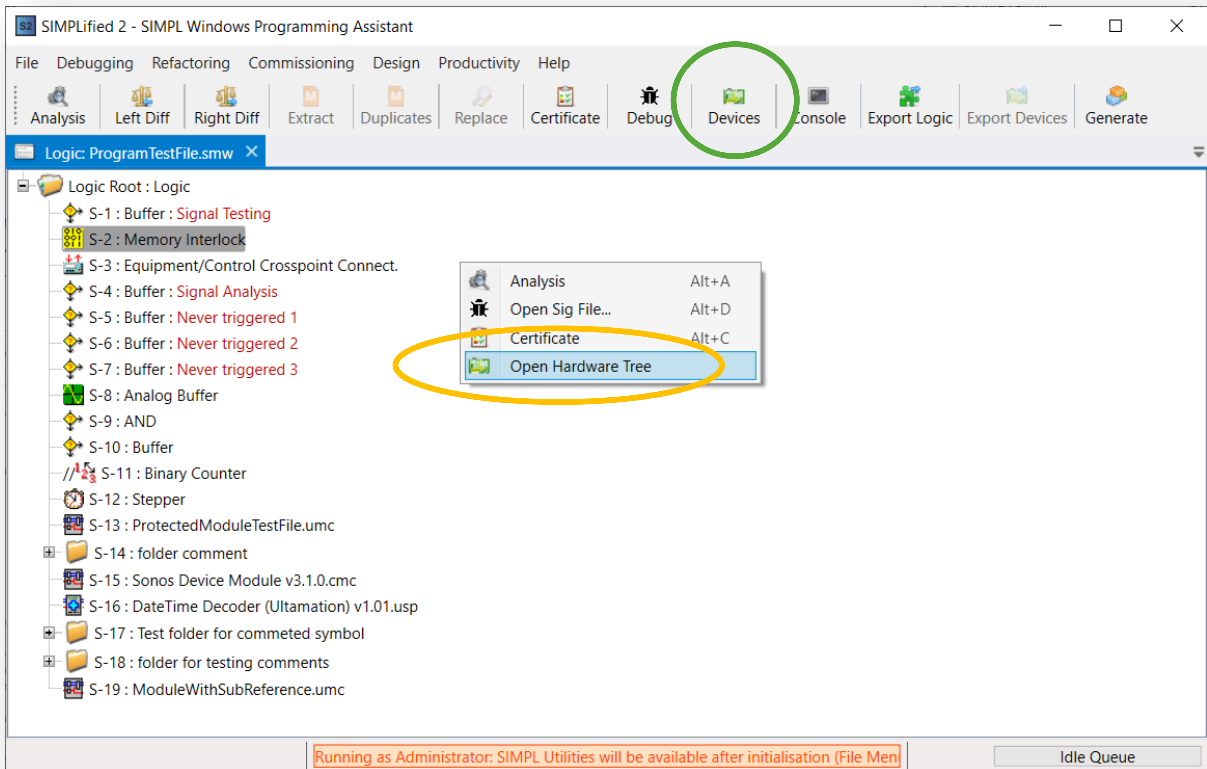
- The main menu - e.g. Debugging -> Run Analysis
- The tool bar – e.g. Analysis
- Context menus – e.g. On a program tree window, Right Click -> Analysis
- Key bindings – e.g. On a program tree window, Alt+A

Other features exposed on the menus, and toolbar are program diff, refactoring features and program score certificates.

These are explained in more depth in later sections of this document.

Viewing a Program's Device Tree

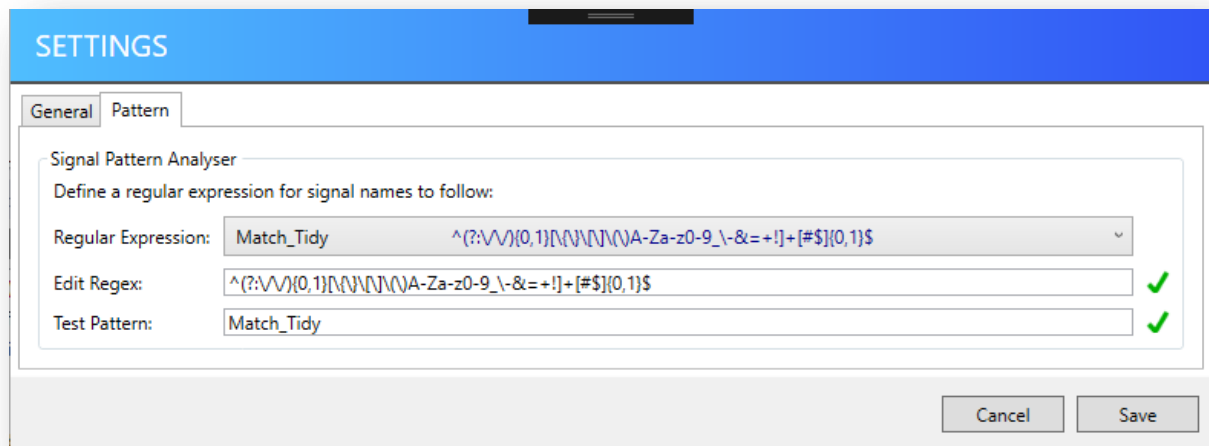
With a program open and in view, you can choose to view the device tree of the program too. To do so, make sure the selected pane is the program tree, this will enable the 'devices' button in the toolbar. You can also right click on the white area of the program pane to open the menu, and choose 'Open Hardware Tree' from there. This will open a new tab containing a tree structure of all the devices in the program, as shown below.



Settings

SIMPLified's behaviour can be modified via the settings dialog found under the File menu.

For instance, if you have the refactoring entitlement, one of the analysers will test for signal names that don't match your preferred standard. You can change regular expression used for this matching process within the settings as shown below.

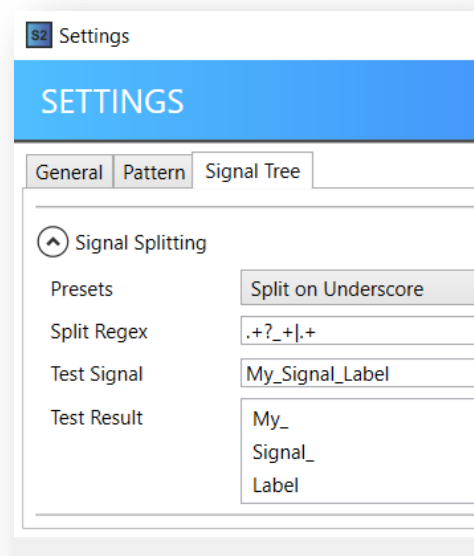


This feature does require some comfort with regular expressions but is incredibly powerful when used correctly. We recommend <https://regex101.com/> as an excellent resource to help define regular expressions, and test them against example signal names for conformity.

SIMPLified will also perform two tests in the settings dialog. The first check mark establishes if your regular expression is well-formed. The second tests this regex against a test pattern of your choosing.

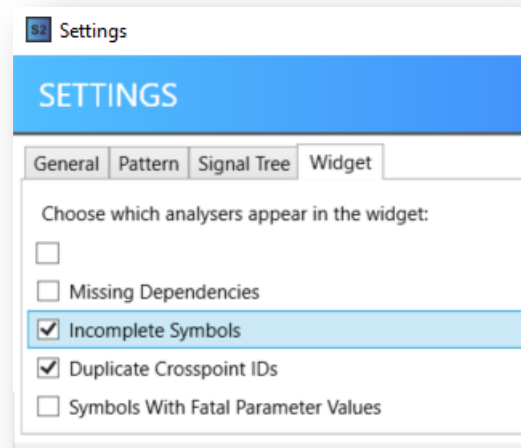
Debug Settings

In File -> Settings you can find settings for the signal tree. This lets you choose how signal names are grouped together in the signal tree. You can either use a pre-set value, or make your own regex to use instead. This example splits the name on underscores. So all signals beginning with 'My_' would be listed under the 'My_' branch, and names with 'My_Signal_' would be in the nested branch called 'Signal_', and so on.



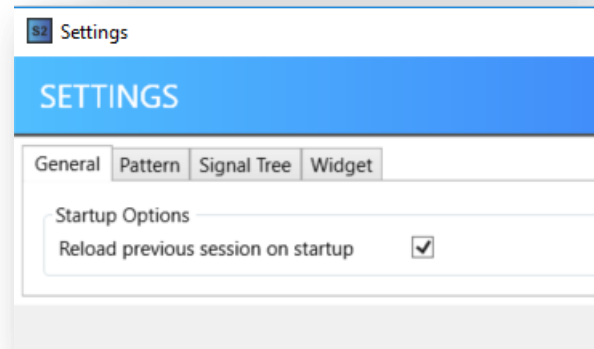
Widget Settings

The Widget settings let you choose which analysers appear in the widget. Simply untick the ones you don't want to see. The first tick-box will select/unselect all.



General Settings

Here you can choose whether the current session is saved on exit, and reloaded when SIMPLified 2 is next opened. When this is not ticked, a blank workspace will be opened.



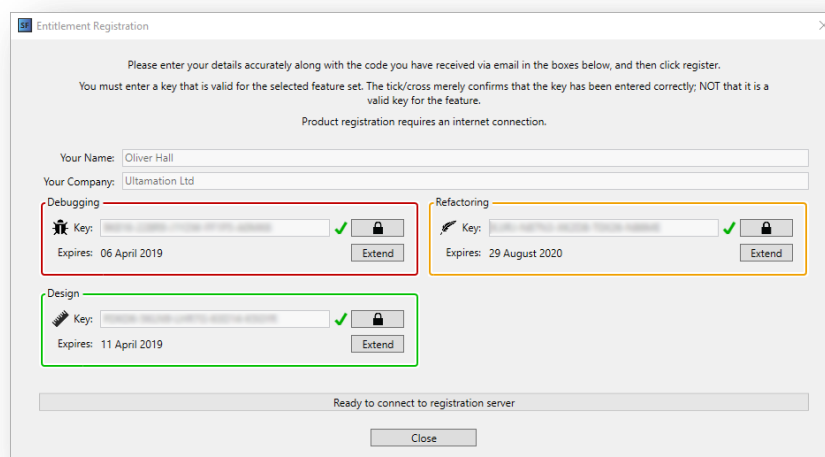
Adding Entitlements

SIMPLified's features are unlocked by applying category entitlements, with each entitlement enabling analysers and features.

You can check which entitlements you currently have active from the About... box (under the Help menu)



Or by opening the Entitlement Registration dialog shown below.



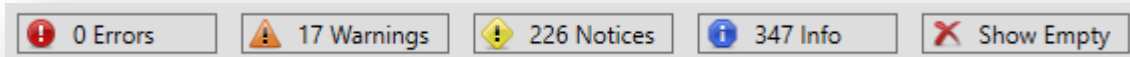
Before you enter any keys, enter your name and company name in the text boxes. These cannot be changed when an entitlement is active (though they can be changed when all entitlements are unlocked).

Each entitlement is given by a key and is locked to a single "seat". It can be locked and unlocked as many times as you wish to enable easy migration between machines. The key's format is checked upon entry and the green check mark will appear when the key format has been verified. This does NOT mean that the key is valid for the product, or has not expired – this check is performed when you click on the open padlock, which will then allocate the entitlement key to your machine.

Each entitlement also has an expiry date which is shown below the key. You can extend your entitlements as many times as you wish.

Program Analysis

Program analysis provides a collection of diagnostics analysers to run over a program and generate a set of results to help inform the programmer where potential issues might lie, or optimisations could be made. Results are given a severity level much like the errors, notices and warnings of SIMPL's own compilation step.



SIMPLified's analysis provides a number of benefits over SIMPL's post-compile report.

First of all, the analysis runs across the SIMPL program each time it is saved from SIMPL. This enables the programmer to continue working within SIMPL while producing a near real-time list of warnings and notices as they develop the Crestron solution, only invoking the expensive compile option once all warnings and notices have been resolved.

The standard analysers also provide a number of additional informational results that SIMPL doesn't. This can help resolve issues such as missing symbols, commented logic, or diagnosing module path issues.

The analysers detailed below are available, either included in the free SIMPLified 2 program, or enabled through licence keys. All analysers are visible in SIMPLified even when inactive, and clicking on an analyser will provide information about its purpose and which licence is required to active the analyser. We also encourage Crestron programmers to suggest new analysers that would be beneficial to the development process.

Analyser	Multiple Driving Sources
Licence Required	Free
Severity	Warning
Score Category	Functionality
Description	Lists all digital signals with multiple driving sources where one or more of the source symbols do not allow 'jamming'
Hint	This is either poor design or, if not, you can suppress the warning by first passing the signal through a digital buffer.
Analysis Time	400ms




In addition to the analysis results, each analyser may also contribute to an overall 'program score'. This score provides a measure of various program attributes, such as completeness and functional regularity. It must be noted that this score is purely a subjective measure using Ultamation's scoring algorithm and is in no way

endorsed by Crestron. The analyser information panel shows which score category the analyser contributes towards.

Each analyser generates a set of results which relate to a signal, logic symbol or module. Selecting the result will provide detailed information for the object under scrutiny, such as which symbols are connected by a particular signal.

Driving Sources	Destinations
S-3.4.13.4.2.7: Multiple One Shots	Slot-02.1: Ethernet Intersystem Communications (Packed)
S-3.5.41.4.2.7: Multiple One Shots	
S-3.5.38.4.2.7: Multiple One Shots	
S-3.5.33.4.2.7: Multiple One Shots	
S-3.4.6.4.2.7: Multiple One Shots	
S-3.5.45.4.2.7: Multiple One Shots	
S-3.4.1.4.2.7: Multiple One Shots	
S-3.5.28.4.2.7: Multiple One Shots	

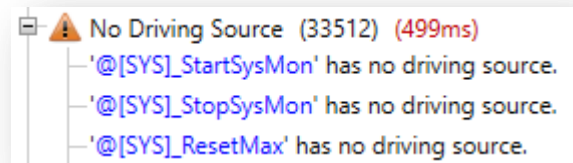
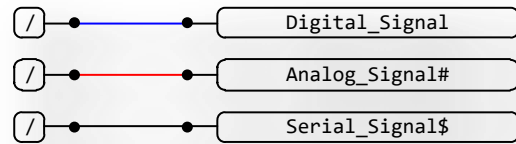
Each time you save changes to your SIMPL program, the analysers update to reflect the changes you made. If your changes cause the number of results increase or decrease, the analysers show this change next to each one. Red text indicates that the number of problems has increased, green means you have fixed some. In this example, 'commented-Out Symbols' has increased by one, and 'Malformed Signal Names' has decreased by 19.

 Commented-Out Symbols (1) (0ms)
 Malformed Signal Names (0) (-19) (0ms)
 Module Dependencies (0) (0ms)

Analyser: No Driving Source

This analyser provides a similar role to SIMPL's compilation warning report with a few important differences. As already stated, the first benefit is speed – you no longer need to compile the program to get a report of unconnected signals.

The second difference is that, whereas Crestron's current implementation of their signal highlighting will show a signal as a valid connection when it's only driving source is commented out within the program. SIMPLified will flag these signals as without a driving source just as they will be reported in the compilation report.



Thirdly, the analyser mirrors Crestron's current implementation of their signal highlighting logic, though this is actually inconsistent with the compilation report.

A signal can exit on the input of a module, be unconnected in the host program, and yet not generate a warning. This is probably not the behaviour one would want as it suggests to the casual observer that the signal is in use. The reason the warning can be suppressed is that – internally – the signal on the input definition of the module is *also* being driven by a symbol within the module. The signal is in use, just not in the way you might expect.

This is why SIMPLified's result count sometimes disagrees with the SIMPL compile, however, if you look for the symbol in SIMPL, it will appear highlighted (if you have the function enabled).

Analyser: Duplicate Crosspoint IDs

It is a fatal compilation error to have any equipment and or control crosspoint symbols using the same numeric identifier. This can easily happen when duplicating logic, such as a user interface sub-tree. This analyser not only checks literal instances of the equipment and control crosspoints, but also performs parameter substitution for dependent modules that contain crosspoint symbols.

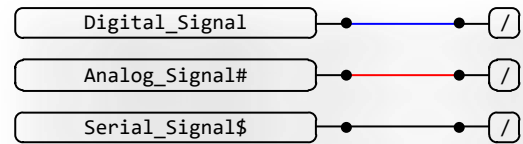
The results are a list of symbols that contribute to crosspoint conflicts allowing you to quickly resolve the duplicate identifiers for a successful compile.

Analyser: No Destination

This analyser is also very similar to SIMPL's compilation notice report.

Again, in some respects, the analyser mirrors Crestron's current implementation of their signal highlighting logic; again, this is inconsistent with the compilation report.

A signal can currently exist on the output of a module, be unconnected in the host program, and yet not generate a warning. This is similar to the symptom found with "No Driving Sources" except that in this case, the signal on the output definition of the module is *also* used to drive a symbol within the module.



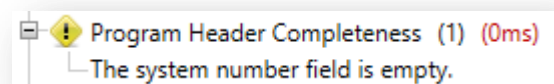
SIMPLified's driving source analyser ignores internal signal linkage which provides a more intuitive report of unconnected signals.

As for "No Driving Sources", this is why SIMPLified's result count sometimes disagrees with the SIMPL compile.

Analyser: Program Header Completeness

A completed program header helps prevent commission slips, and in the maintenance of systems when trying to identify original programmers, dealers or program versions.

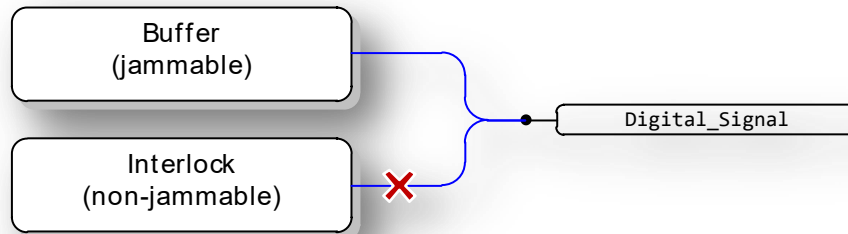
The program header analyser highlights where these fields have been left empty – it's down to you to ensure the information is correct and useful though!



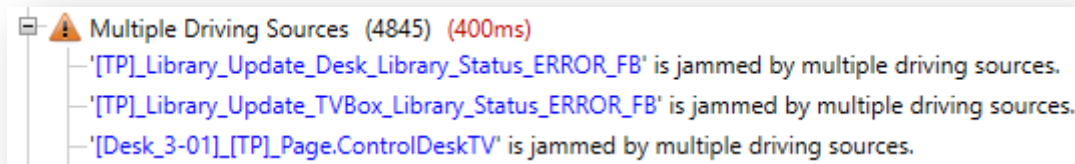
This program header info is also presented on the SIMPL Program Score certificate.

Analyser: Multiple Driving Sources

This analyser replicates the multiple driving sources warning of the SIMPL compile, again, without the overhead of an actual compile.



Selecting the signal will show the driving sources to assist in correcting the jammed signal.

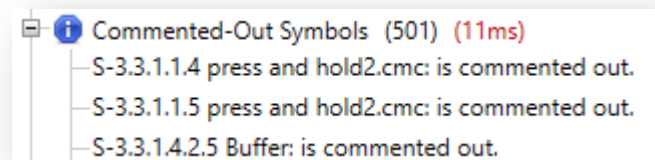


Analyser: Redundant Forced Signals

Sometimes it is necessary to “force” a signal to a particular type (digital, analogue or serial). However, this also presents an opportunity to hide signals that would otherwise report without destinations where they don't contribute to the program functionality. This analyser is for that situation.

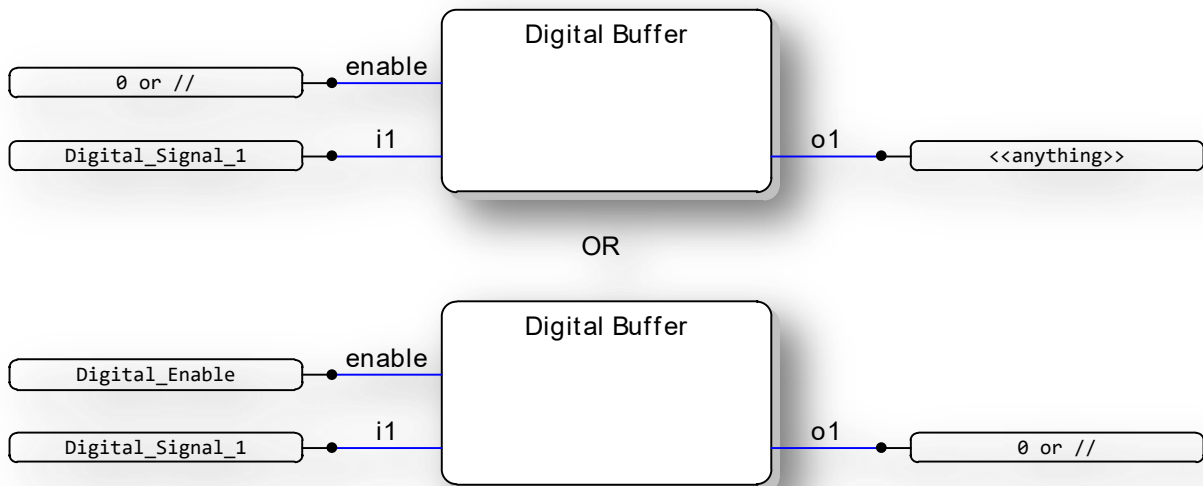
Analyser: Commented-Out Symbols

This informational analyser allows the programmer to quickly establish the whereabouts of symbols that cause the “There are commented out symbols...” dialog to appear on compilation. It can be hard work to find these symbols or logic sub-trees in large programs. The results of this analyser enable you to identify logic that isn't contributing to the program so that it can be removed.



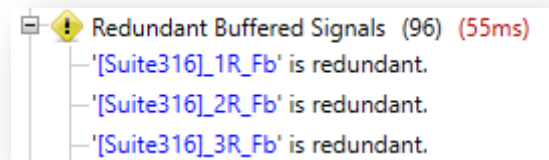
Analyser: Redundant Buffered Signals

Sometimes “hanging” signals can be put to one side by terminating them onto buffers. This is sometimes desirable during development to avoid the compile log being overrun with notices that we are – at the time – comfortable with.



However, there is absolutely no value in these signals remaining in a production system. They may well add little in terms of resource overhead, but if they have no value there is little argument for keeping them in the program.

This analyser will find signals of all three varieties, and report any that terminate ONLY on a buffer which is either guaranteed to be disabled (i.e. '0' or '//') or the output side of the buffer is empty (i.e. '0' or '//').

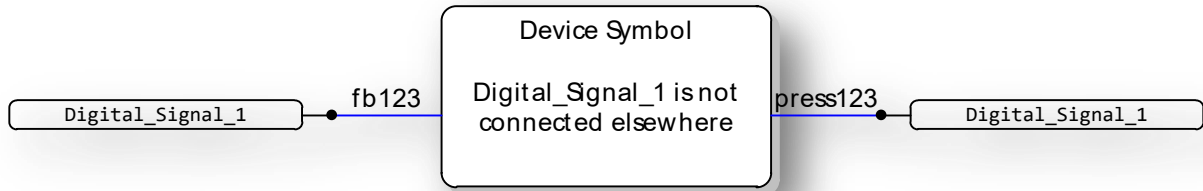


Analyser: Incomplete Symbols

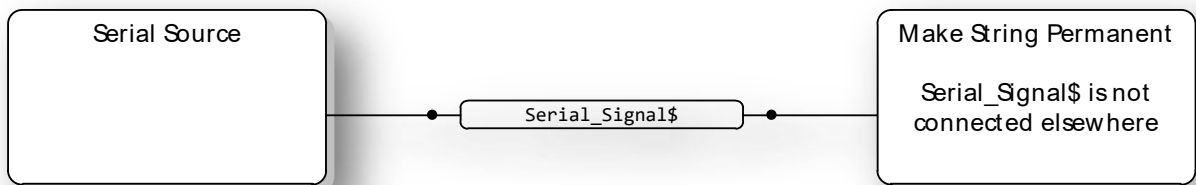
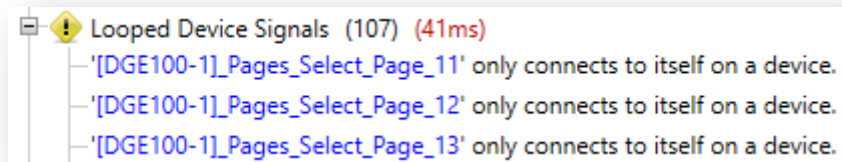
Symbols with incomplete signals or parameters will inhibit compilation. This analyser will flag any incomplete symbols so that the errors can be resolved quickly.

Analyser: Looped Device Signals

When building up a touch panel object, the programmer may create signals for, say, “presses” and duplicate them on the “feedback” side of the same symbol so that the button will (programmatically) appear to press when it is touched.



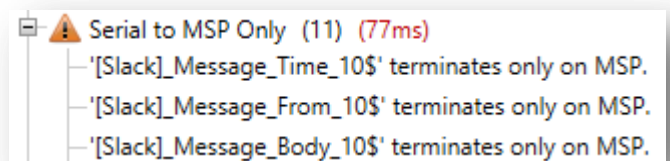
This will appear functional in the resulting program, and no warnings or notices will be generated during compile. This analyser will flag any such symbols.



Analyser: Serials with only MSPs as Destinations

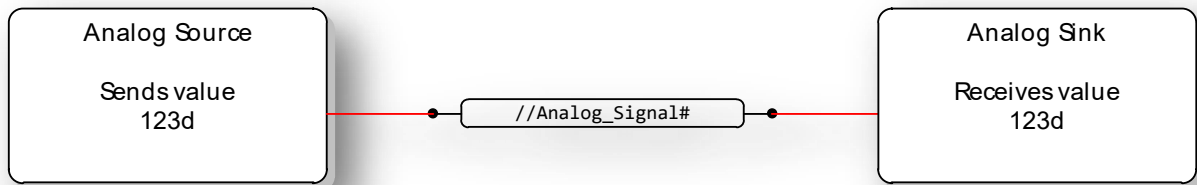
Any serial signal that is connected only to one or more Make String Permanent symbols will not appear as a compile time warning, but will also not contribute to the program functionality.

This analyser will help identify bugs where serial join behaviour is not as expected but highlighting any signal that is connected to nothing more than MSP symbols.



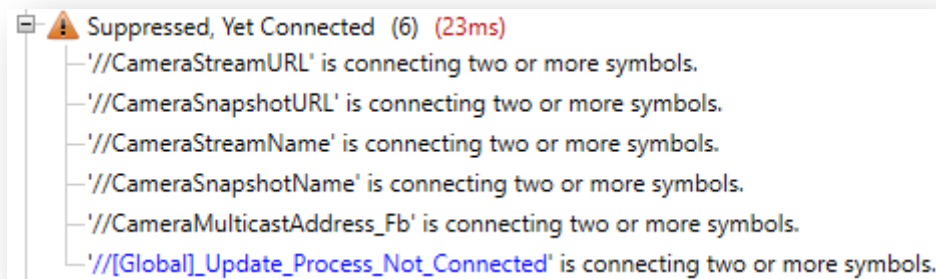
Analyser: Suppressed, Yet Connected, Signals

It is sometimes misunderstood that signals with a prefix of '//' are not, in fact, 'commented out' in the sense that they are omitted from the final compiled program. The '//' prefix provides no other function than to suppress "No Driving Source" or "No Destination" warnings. In all other respects, the signal is very much part of the program logic.



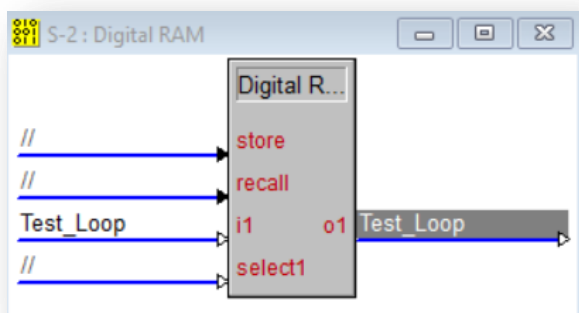
Therefore, for any such signal which connects two or more symbols, the signal path is still intact, which may not be what one expects.

This analyser will warn of any 'supressed' signals (e.g. //[Lounge]_TV_Self_Destruct) that connect two or more active symbols.



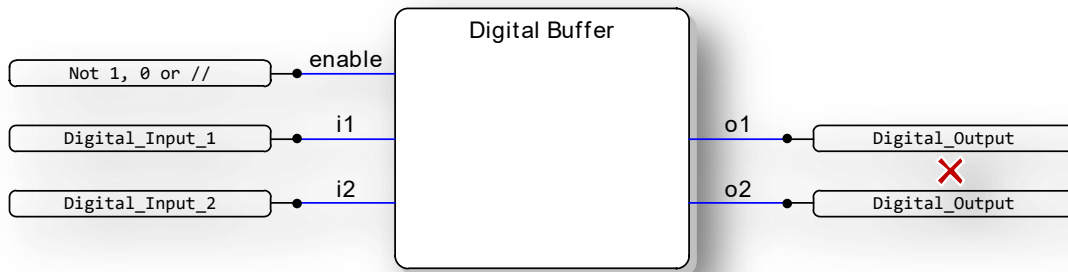
Analyser: Looped RAM Signals

Finds any signals where the only destination and driving source is the same symbol. For Analog RAM and Digital RAM symbols



Analyser: Duplicate Buffer Outputs

This scenario has been covered at Crestron Masters' training and represents a particularly easy trap to fall into, but which can be very difficult to diagnose an issue should it arise.

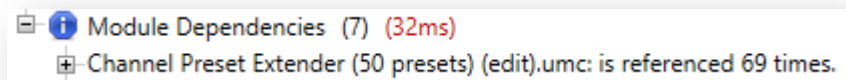


The issue is that when a digital buffer is enabled, the outputs are evaluated based in the inputs from top to bottom within a single logic wave. Any signal which is defined multiple times on the output of such a buffer will take on the final value ONLY, regardless of any preceding values on the same buffer. This is almost certainly not the desired behaviour, and is flagged by this analyser.

The analyser does NOT check buffers which are either constantly enabled with a '1' or constantly disabled with '0' or '/' as these represent a different and, more likely, legitimate use case.

Analyser: Module Dependencies

This provides a list of all included modules (whether active or commented-out) so that the programmer can check for version inconsistencies or other unexpected instances.



Analyser: Missing Dependencies

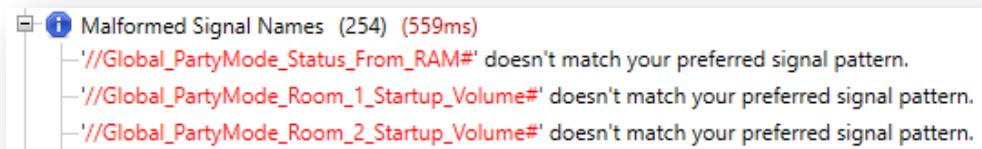
This analyser reports on the critical error where a module cannot be located on the file system.

SIMPL only reports these situations with a generic *SMWMACRO* placeholder, whereas SIMPLified will provide both the locations of the missing symbol, and its filename.

Analyser: Malformed Signal Names

This is very much a stylistic analyser for those who like consistent signal naming conventions.

Based on a regular expression defined under the refactoring settings, the analyser will check every signal name against the regular expression pattern. Any signal name that doesn't match will be itemised.



The default regular expression will match names of the following format:

```
[ // ] '[<word1>]_' x n '<word2>_' x n '<word2>' [#$]
```

Where:

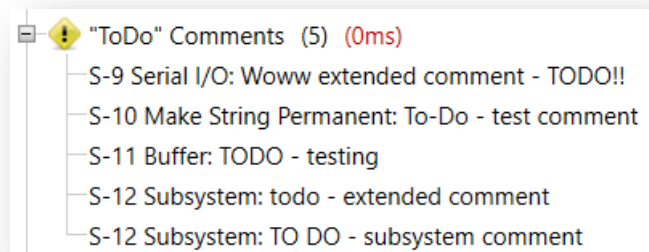
- [...] is optional
- Word1 = Capitalised or 'i' prefix
- Word2 = Capitalised or 'i' prefix or numeric

e.g.

```
[iPad-1]_[Source]_Transport_Play
```

Analyser: 'Todo' Comments

This analyser searches each symbol for comments that contain 'todo's. Symbols will be captured if there is a todo in the standard comment or the extended comment. The variations that will be detected are – 'todo', 'to do', and 'to-do'. These are **not** case sensitive.



Refactoring a SIMPL program

As mentioned earlier, refactoring is the process of refining existing code without changing functionality. Modern IDEs provide a great deal of functionality in this area which greatly aids in the maintainability of code which, in turn, leads to more reliable code.

SIMPL Windows provides little in the way of refactoring tools (global signal rename is probably the most useful), so SIMPLified offers a number of tools to fill that void.

Differences

Providing the facility to compare program logic has a number of benefits when building complex Crestron programs.

By comparing two versions of the same code – or perhaps two programs with the same filename, that you suspect might have been modified – the difference feature provides a symbol-by-symbol report of all the differences between the two programs, or sub-trees.

In the example below, we're comparing the "S-5: User Interfaces" folder for differences between the 4.15 and 4.17 versions of a program. The difference feature is telling us that 4.17 has a number of additional symbols and signals on 5.4.1.6.8.1-3 have changed, which is where we'd added some additional hardware to the Media Player Router symbol.

Left: Ultamation LSP System PRO3 v4.15.smw (S-5)		Right: Ultamation LSP System PRO3 v4.17.smw (S-5)	
Address	Comment	Address	Comment
---		S-5.3.5.11.1	Symbol: Framework UI Cameras (Ultamation) v1.00.usp
---		S-5.3.5.11.2	Symbol: Make String Permanent
---		S-5.3.5.11.3	Symbol: Make String Permanent
S-5.4.1.6.8.1	Signal Difference	S-5.4.1.6.8.1	Signal Difference
S-5.4.1.6.8.2	Signal Difference	S-5.4.1.6.8.2	Signal Difference
S-5.4.1.6.8.3	Signal Difference	S-5.4.1.6.8.3	Signal Difference
---		S-5.8	Symbol: [HR310] Template
---		S-5.8.1	Symbol: [HR310-1] (x102)
---		S-5.8.1.1	Symbol: Custom Buttons

Many programs have a high proportion of duplicated code. Often, it would be good practice to consolidate these repeated structures into modules and any changes in the module will automatically be duplicated across instances. Sometimes it is preferable to leave these logic blocks within the program so that minor differences can be made to specific instances.

This practice has one significant drawback. It can be hard to maintain functional parity across each instance of the "identical" logic. One approach might be to delete each duplicate, and then recreate with judicious use of search-and-replace, though this can be tedious and error prone.

SIMPLified's difference functionality provides an easy way to manage differences in logic trees, both across program versions and within a single program by allowing you to compare logic trees to see if edits have been made inconsistently.

This example shows that S-5.3.2 and S.5.3.4 are structurally identical (other than in the actual naming of signals and any parameter settings).

Left: Ultamation LSP System PRO3 v4.17.smw (S-5.3.2)		Right: Ultamation LSP System PRO3 v4.17.smw (S-5.3.4)	
Address	Comment	Address	Comment

This is because, when comparing two logic trees within the same program, a "lighter" difference test is used because logic trees will always have different signal names.

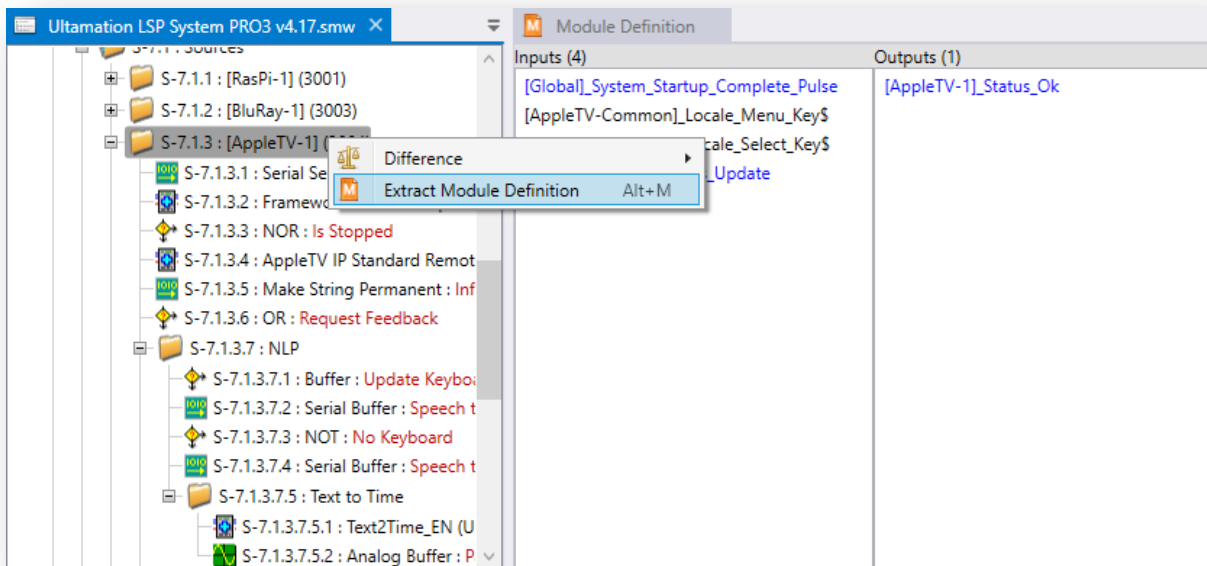
Conversely, the next test between S-5.3.2 and S-5.3.3 show a number of differences, despite S-5.3.3 originally being a copy/rename from S-5.3.2.

Left: Ultamation LSP System PRO3 v4.17.smw (S-5.3.2)		Right: Ultamation LSP System PRO3 v4.17.smw (S-5.3.3)	
Address	Comment	Address	Comment
---		S-5.3.3.1.5	Symbol: Analog Buffer
---		S-5.3.3.1.6	Symbol: Analog Force
S-5.3.2.1.5	Signal Difference	S-5.3.3.1.7	Signal Difference
S-5.3.2.7.6	Symbol: Framework UI Source Control (Ultamation) v1.00.us	---	
S-5.3.2.8.1	Symbol: Framework UI Page Manager (Ultamation) v1.00.us	---	
---		S-5.3.3.8.10.3	Symbol: Localisation
---		S-5.3.3.8.10.3	Symbol: Translator[Max 10] (Ultamation) v1.00.usp
S-5.3.2.11.1	Symbol: Framework UI Cameras (Ultamation) v1.00.usp	---	

Module Definition Extraction

When you decide a section of logic has become a good candidate for modularisation, one of the tasks you must complete is to define which signals will be required on the argument definition for the SIMPL User Module.

The module definition extraction tool will provide a list of all inputs and outputs that reach logic beyond the root node selected. So, in the example below, we have selected to extract the module definition from the AppleTV-1 source logic. This shows that, despite the internal logic containing 10s of signals) only a small number of signals (4 inputs and 1 output) actually need to be exposed if this were to be made into a module.



SIMPLified supports SIMPL's native clipboard format so that these input and output signals can be copied from the Module Definition window and pasted directly onto the Argument Definition in SIMPL Windows.

All that is left is then to copy the logic tree from the main program to the new SIMPL module, and – with a little more housekeeping – the module is complete and ready for use in the main program.

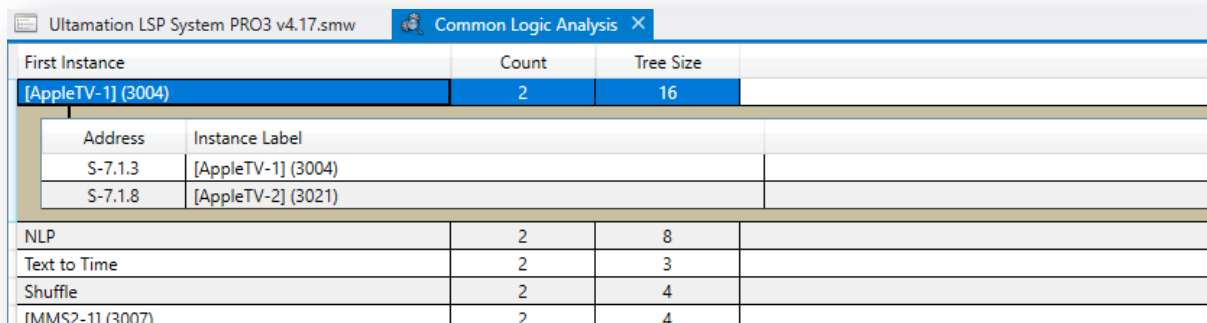
It is important to note that parameters are not listed in the extracted module definition as there is no way to establish if these are used internally, or should be exposed via the argument definition. This is up to you.

Duplicate-Logic Candidates

Being able to identify common logic is an incredibly powerful tool, and by combining various heuristic methods, along with the difference tool and the module definition extraction, SIMPLified is able offer suggestions as to which parts of a program could be converted to modules for greater maintainability.

The Duplicate-Logic Candidates feature will exhaustively work through the program categorising and comparing sub-systems (logic folders), using a number of methods to establish whether or not two logic trees are functionally equivalent.

The example below indicates that the tool has identified a number of sub-folders (under the S-7 branch, "Sources") appear to have a high degree of commonality. Here we can see that the AppleTV-1 and AppleTV-2 logic trees appear to be equivalent, and the logic tree has a size (the number of child symbols under, and including, the root node) of 16.



First Instance	Count	Tree Size						
[AppleTV-1] (3004)	2	16						
<table border="1"> <thead> <tr> <th>Address</th> <th>Instance Label</th> </tr> </thead> <tbody> <tr> <td>S-7.1.3</td> <td>[AppleTV-1] (3004)</td> </tr> <tr> <td>S-7.1.8</td> <td>[AppleTV-2] (3021)</td> </tr> </tbody> </table>			Address	Instance Label	S-7.1.3	[AppleTV-1] (3004)	S-7.1.8	[AppleTV-2] (3021)
Address	Instance Label							
S-7.1.3	[AppleTV-1] (3004)							
S-7.1.8	[AppleTV-2] (3021)							
NLP	2	8						
Text to Time	2	3						
Shuffle	2	4						
[MMS2-1] (3007)	2	4						

This may therefore be a candidate for turning the AppleTV source logic for our framework into a module that can easily be dropped in whenever an Apple TV is required. We have already seen that the module would only need a handful of signals connecting up and, with the addition of a few parameters to hook up the underlying framework and Apple TV addressing, we'd be good to replace the program logic with two module instances – with any future modifications being made to the re-useable module.

What's more, every other program that would benefit from any changes to this new source module could pick up these changes with a simple recompile. Clients receive these new benefits for very little effort, and mistakes are kept to a minimum.

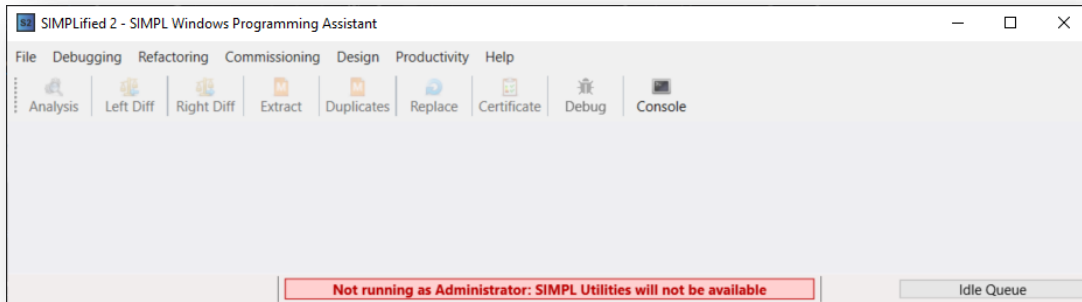
It is important to note that the results of the duplicate-logic analysis are suggestions only. The heuristic methods used to compare logic trees are not 100% accurate and care should still be taken when making use of the information this tool provides.

Always review the suggestions before replacing candidates with their modular versions and always test the new modularised code before deploying to production!

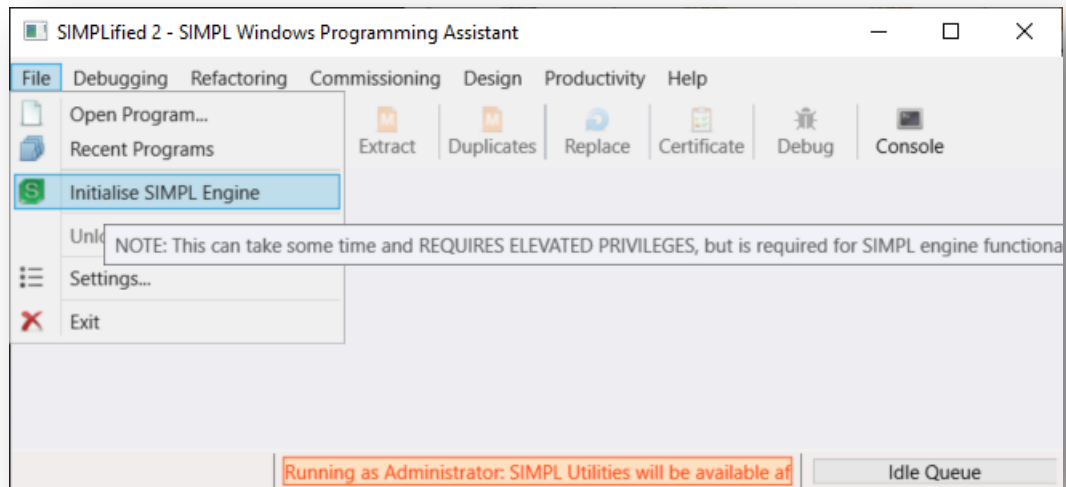
Search and Replace

This feature lets you replace a module with another module in your program.

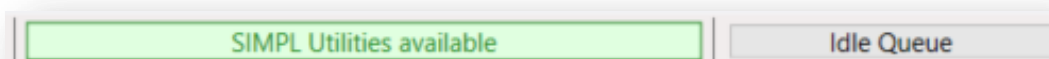
To use Search and Replace, the SIMPL Engine must be initialised, which also requires running SIMPLified as administrator. If you are not running as Administrator, then you'll see the warning pictured below on the status bar. This is not an issue if you don't plan to use the search and replace



When running in administrator, the status bar will be orange and the option to initialise the SIMPL Engine will be enabled:

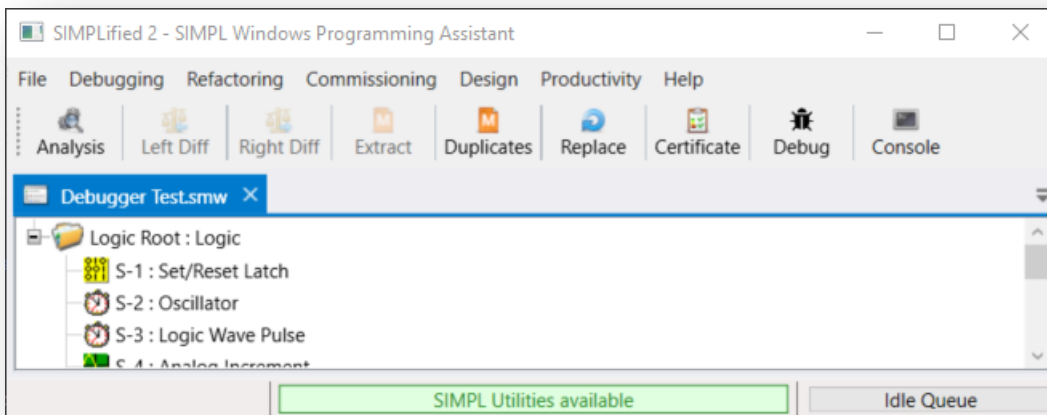


And once initialised, the status bar will turn green. Be aware the initialisation takes a little time

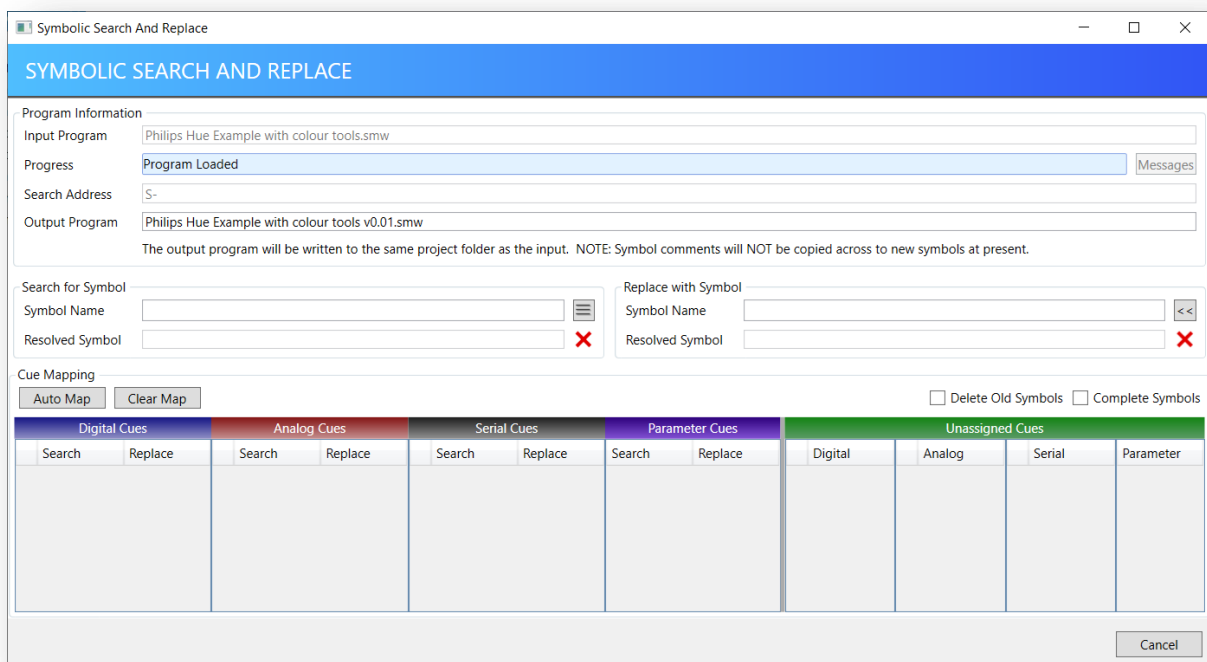


SIMPLified 2 User Guide

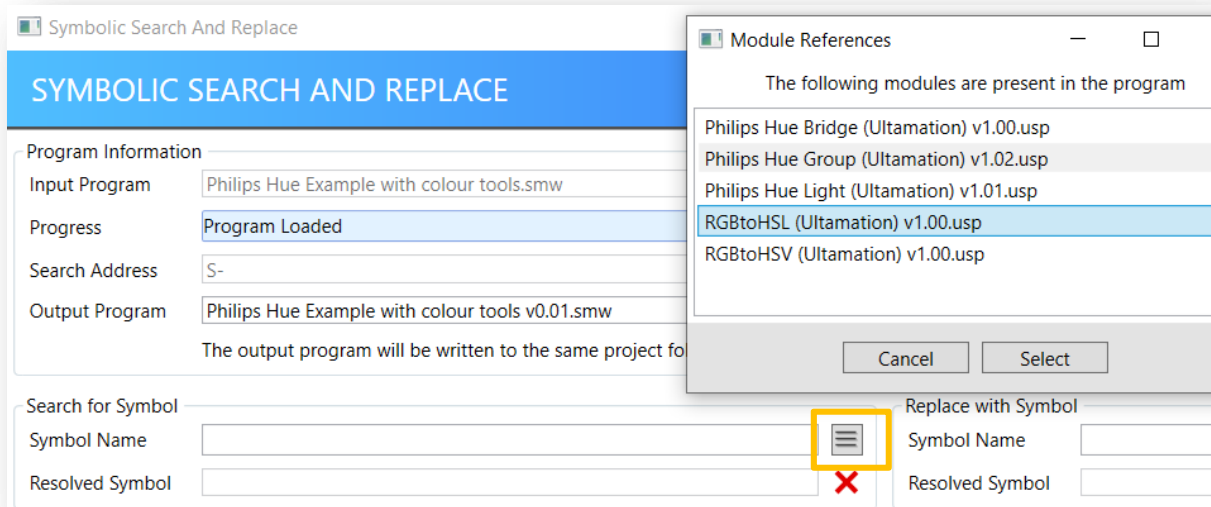
Load a program to enable the Replace button on the toolbar,



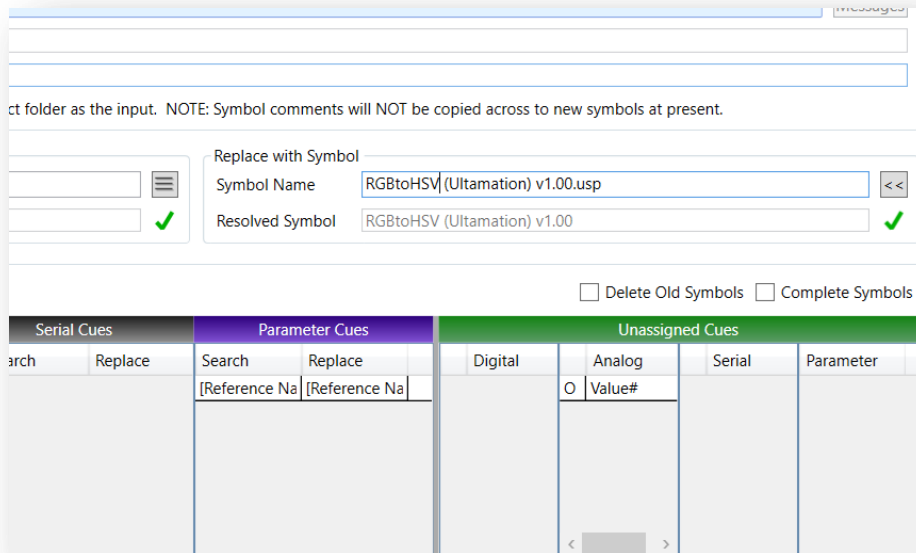
Pressing the replace button will open the window shown below. With the loaded program ready to be modified.



The next step is to select the module you want to replace. Click the **menu button** to open a list of modules in the program. Choose one and press select.

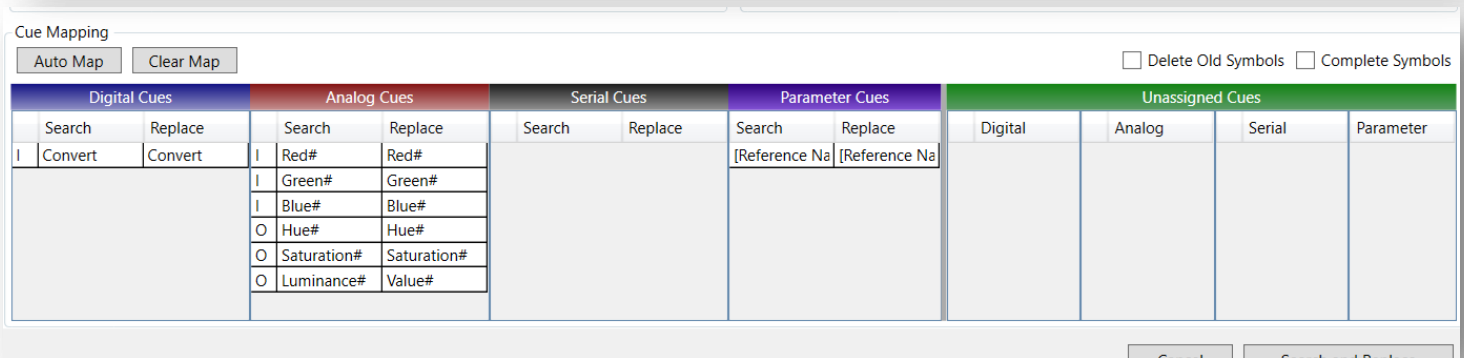
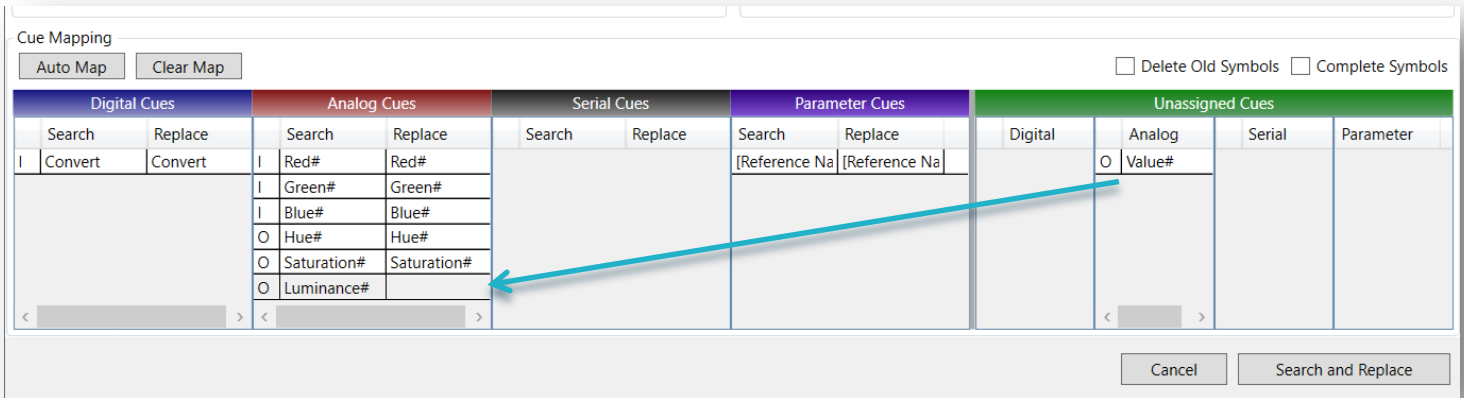


Then choose the module you want to replace it with. Type the module name, or click the '<<' to use the same module you want to replace.

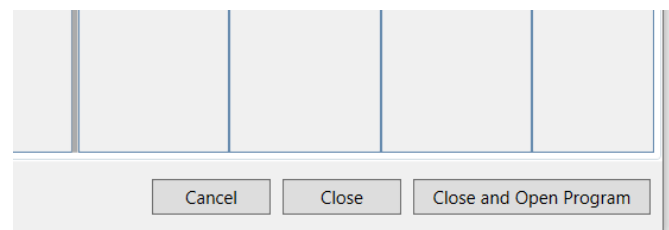


SIMPLified 2 User Guide

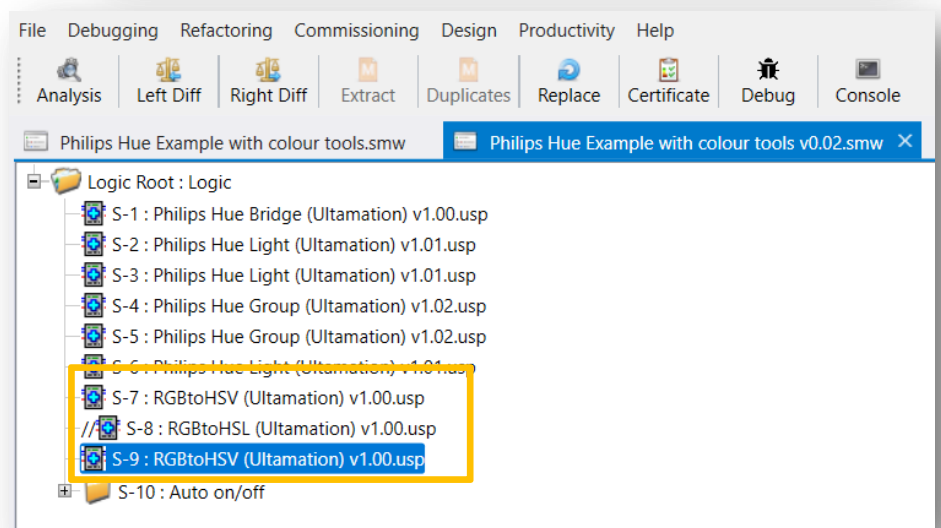
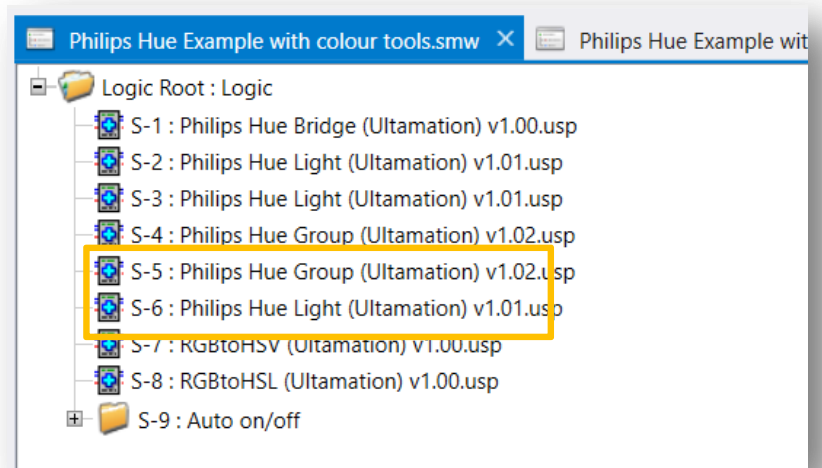
The signals from each module will be listed as below. The program will match the signals names that it can, and others will be listed under 'unassigned cues'. You can drag from the unassigned column to match them with the signals you want to replace. E.g. here, we want the Luminance signal to be replaced with the Value signal, so we drag the Value signal to the 'replace' column next to Luminance.



Once you're happy with the above arrangement, click 'Search and Replace'. Then clicking 'close and Open Program' will close the replace window, and open the newly made program.



This is the program, before replacing a module. It has one HSV and one HSL module. And in the new version below, the HSL module has been replaced with another HSV module. The old module has also been commented out. It's possible to remove the module when replacing, by ticking the 'Delete Old Symbols' box on the configuration page. There's also a tick for to complete the new symbol automatically after creating.



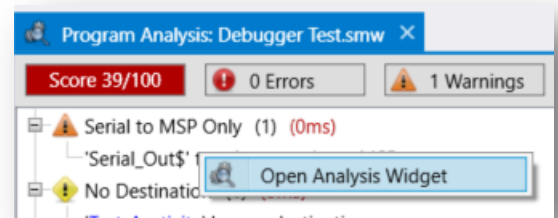
Analysis Widget

Perfect for anyone working from a single monitor, the widget will sit on top of other windows and show which analysers have problems. This lets you keep working in SIMPL Windows and each time you save, the widget will update to let you know if you have created or fixed any issues; no need to keep opening SIMPLified2 to check.

To open the widget, you must have a program open and ran an analysis, which open the analysis pane. **Right click** on the analysis pane and click 'Open Analysis Widget'.

The widget displays a list of the analysers which have at least one result. To limit this selection further, in the settings, you can select and deselect analysers to show.

The first column indicates how many problems the analyser has increased or decreased by in the most recent save. Green indicating you have decreased this number, red meaning there's been an increase. The Total column shows how many issues exist in that analyser type overall. And the final column is the analyser name. The widget can be moved by dragging from the main table or the header text.

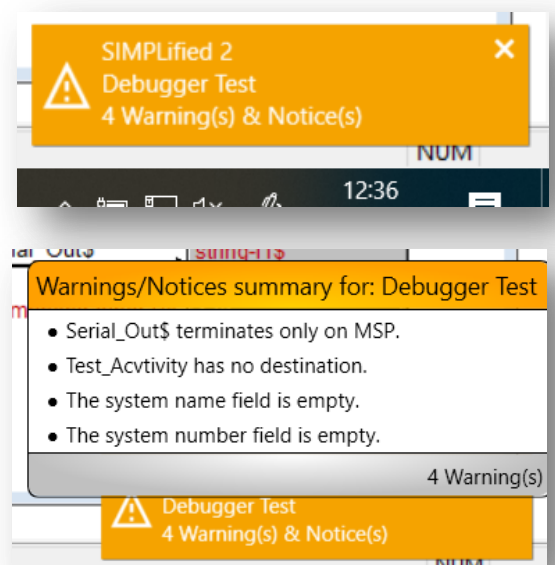


+/-	Total	Analysers	Analysers Name
0	1	Serial to MSP Only	Serial to MSP Only
-1	1	No Destination	No Destination
0	2	Program Header Completeness	Program Header Completeness
1	1	Commented-Out Symbols	Commented-Out Symbols
0	19	Malformed Signal Names	Malformed Signal Names

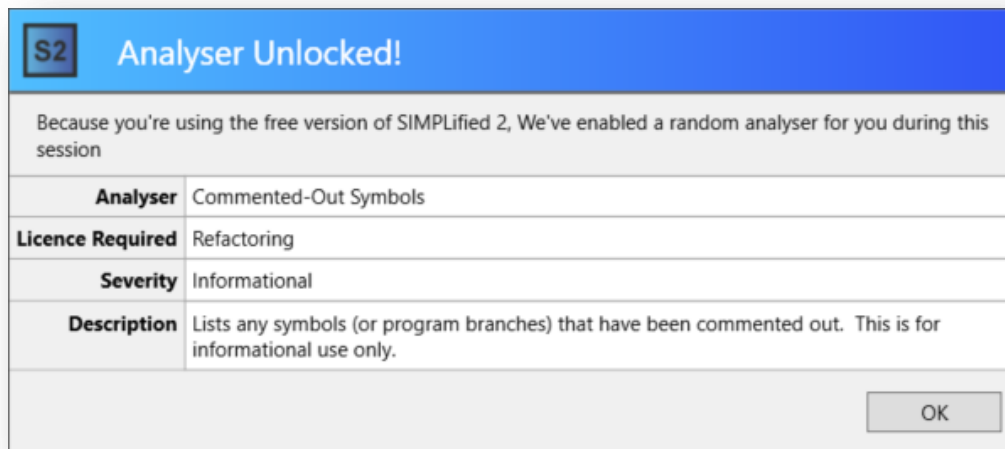
Analysis Toast Notifications

Similar to the widget, these notifications pop up when you save the program to update you on any changes to issues. They are activated when SIMPLified is minimised only. The colour of the popup indicates the severity (green / orange / red), and if you fix all errors you will be notified of that too.

Hovering over the popup will open a more detailed description of the changes, and clicking the popup will open SIMPLified for you to look at the analysers in more detail.



Analyser Unlocked Pop-up



If you are using the free version of SIMPLified 2, and have no entitlements unlocked, you will occasionally see this dialog appear upon start-up. It's to let you know that for the current session, we have unlocked a random analyser for you to try. The analyser that has been unlocked is then detailed in the table, as shown in the above image.

Design

We have big plans for the design entitlement, though, for the present, these remain plans rather than demonstrable functionality. SIMPLified does provide a scoring mechanism which takes the results from the various analysers and gives an aggregate score for the entire program. Some analysers contribute more or less strongly to this overall score, depending on whether we feel a “hit” is, say, an aesthetic issue or plain bad practice. Any analyser that reports an error, which are considered issues that would prevent the SIMPL Windows compiler from completing successfully, will yield an instant “fail” (0/100).

Program Certificate

A summary of the program analysis score can be generated using the Generate Program Certificate which is enabled as a beta feature when the design entitlement is absent.

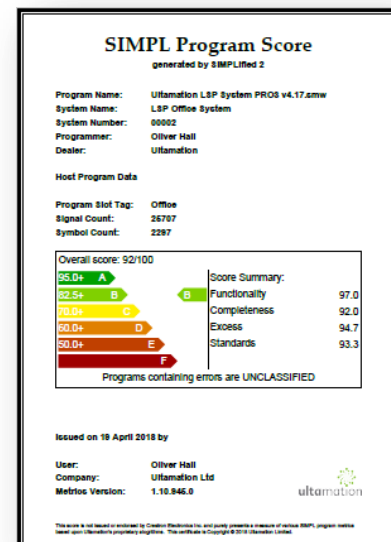
The certificate provides an overview of the program, including the program header information detailing the dealer, programmer and data relating to the program size.

Finally, a summary of the various analyser scores is given with a graphical grading for the program.

While this is purely an automated grading based on the current collection of analysers, and provides absolutely no guarantee of functional correctness, this measure could be used to provide clients with some level of comfort that the program being delivering has met certain standards in development.

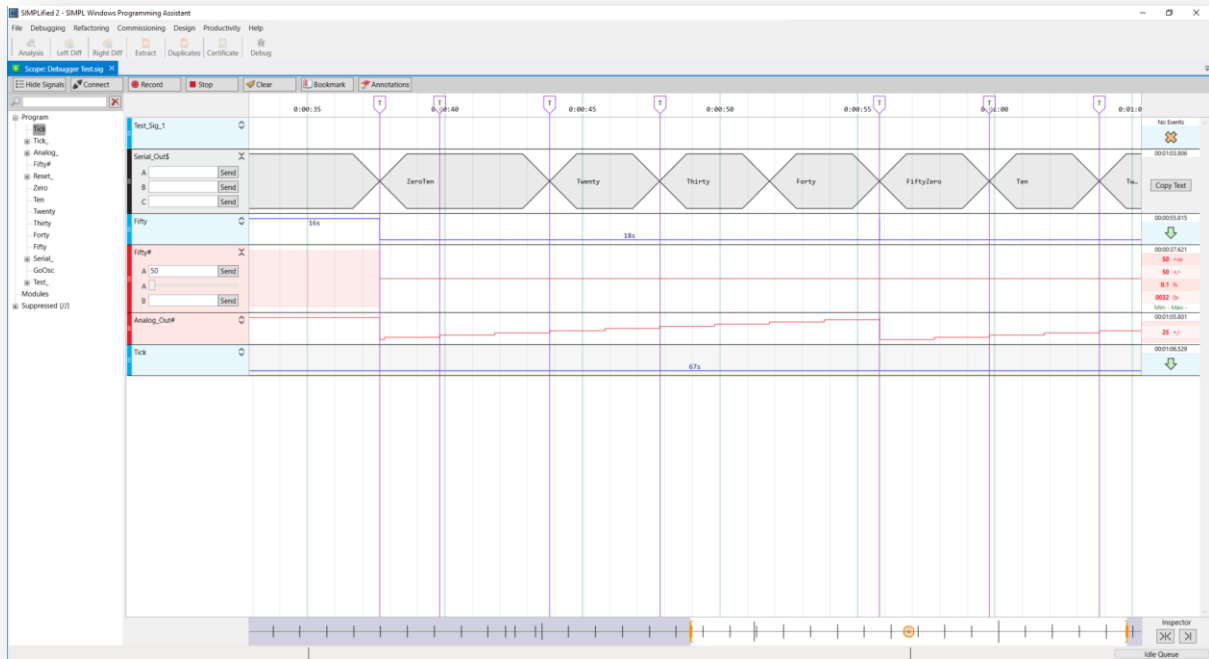
More information on the certificate can be found in the SIMPL Program Static Analysis White Paper document that can be found from the Help menu in SIMPLified 2 or on the shop page here –

<https://shop.ultamation.com/index.php/hikashop-category-information-menu-129/product/95-simplified-2>



Debugging

The debugging pane gives you a timeline view of your chosen signals, so you can see when each signal changes. This section will go into detail of how to use each element of the debugger.

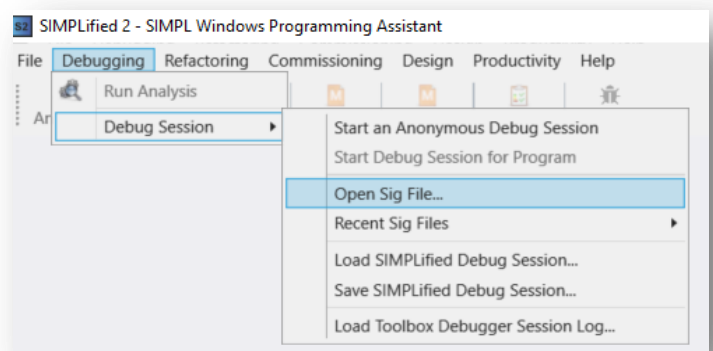


Opening a Debug Session

There are three ways to open a debug session:

Method 1 - Open the sig file directly

From the Debugging menu choose 'Debugging -> Debug Session -> Open Sig File...'. This will open the file explorer window for you to browse and choose your desired sig file. 'Recent Sig Files' will show a list of files you have recently opened for fast access.



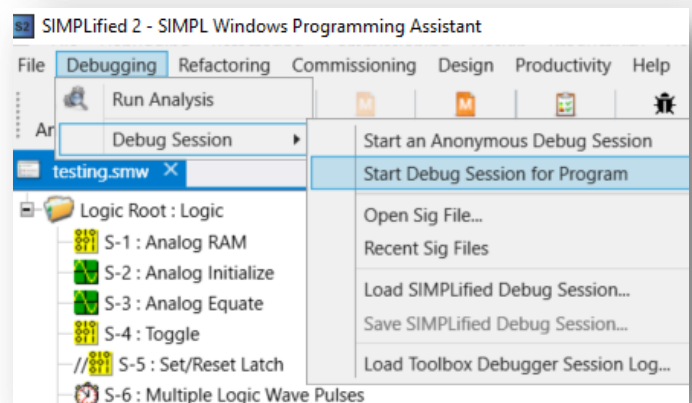
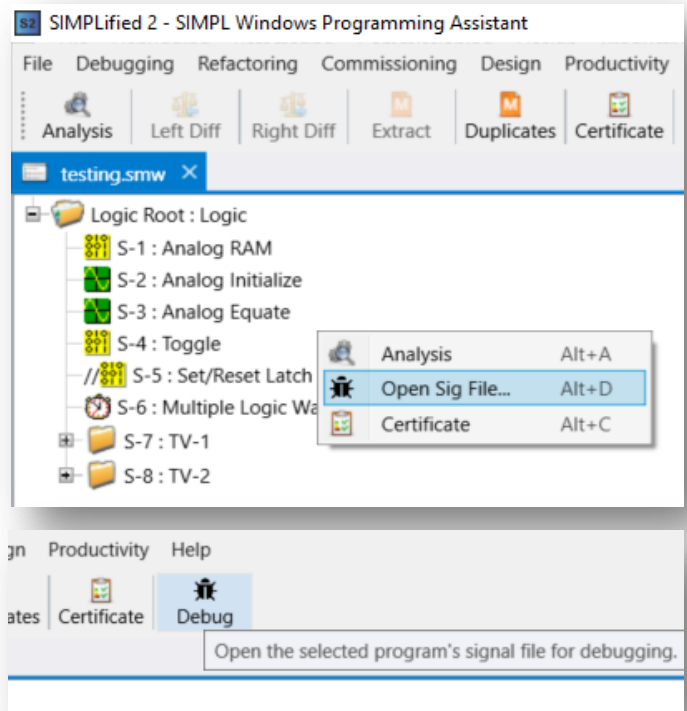
Method 2 – Open from Program Tree

Through an already open program – right click on the program tree and choose 'Open Sig File...'. Alternatively the shortcut Alt + D will also do this if the program is the active pane.

This method only works if the program's sig file is located in the same directory as the program. If no file is found, the file explorer is opened for you do find it yourself

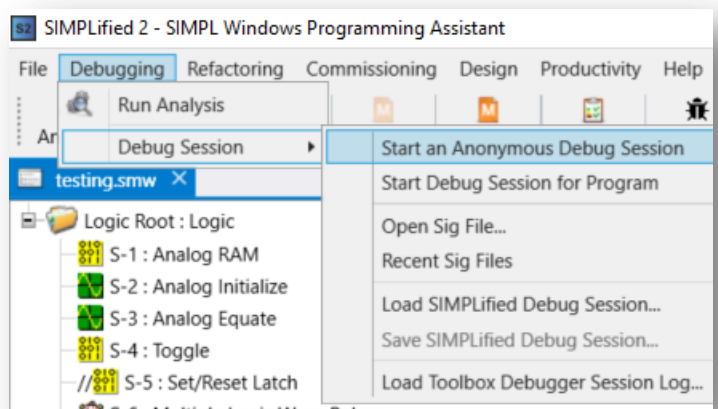
Selecting the Debug icon in the toolbar will also open the selected program tree's sig file if it can be found.

With the program tree selected, the 'Start Debug Session for Program' option will be available in the Debugging menu too.



Method 3 – Anonymous Debug Session

An anonymous debug session will open a blank debugging pane which allows you to connect to a processor and choose which program to debug. As signals come through they will be displayed on the timeline with random names which can be changed for readability.



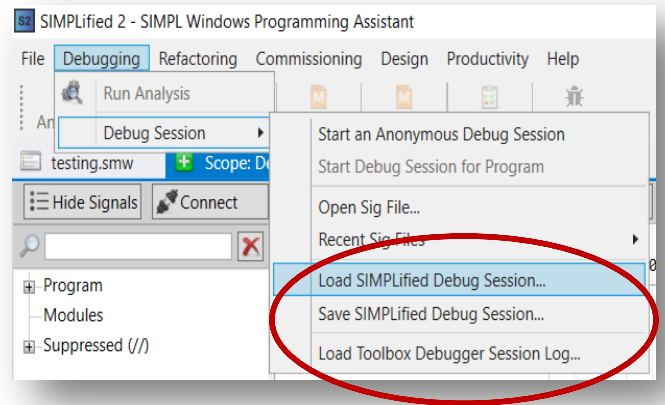
Save or Load a Debug Session

There's also the option to save a debug session, and load it again at a later time. You can Load either a SIMPLified debug session, or a Toolbox Debug Session to view.

Navigate to Debugging -> Debug Session, where the save and load options are displayed.

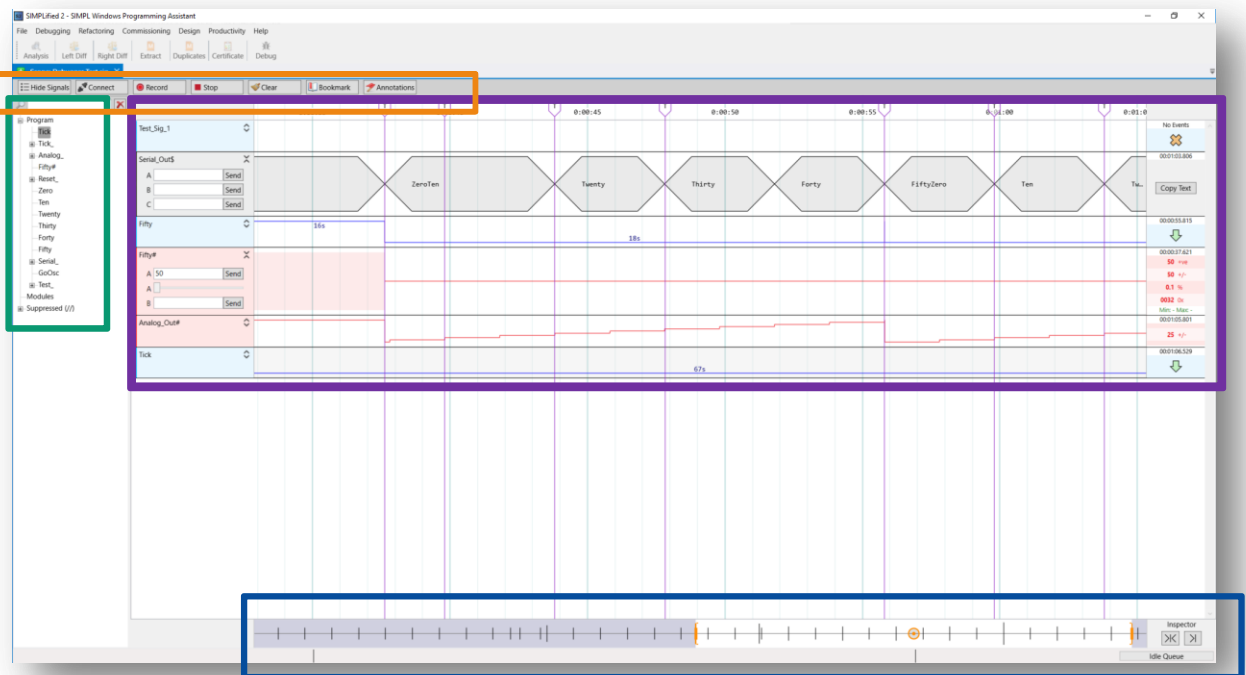
Choosing a 'load' option will open the file explorer to pick either a SIMPLified debug file, or Toolbox file. Then open The session in a debug pane as normal.

The Save option will open the explorer window to choose a save location.



Using the Debugger

This section will cover how to use each aspect of the debugger including – the **signal tree**, **scope**, **toolbar**, and **timeline**.



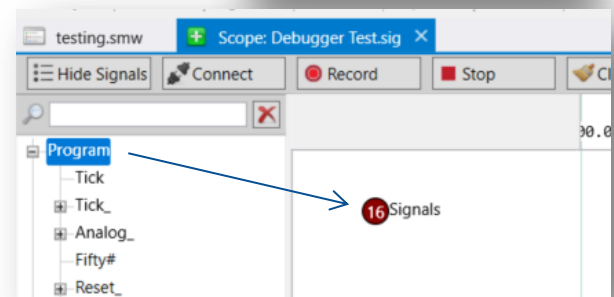
Signal Tree

Clicking the **hide signals button** will toggle the signal tree section to collapse out of view. You can open it by pressing the button again, which will read 'Show signals' when collapsed.

The **search bar** above the signal tree lets you search for signals by name. A search will begin once you have typed more than 3 characters into the box. Signals that match the term are displayed in a list and can be dragged onto the scope as normal. Press the cross to clear the search text and the whole signal tree will return. Currently the search term is case sensitive.

The **signal tree** area displays a tree structure of all signals in your program. The root is split into Program, Modules, and Suppressed. These are expandable and collapsible, signals inside are organised into branches based on their name. (How these branches are organised can be changed in settings, more information [in settings](#)).

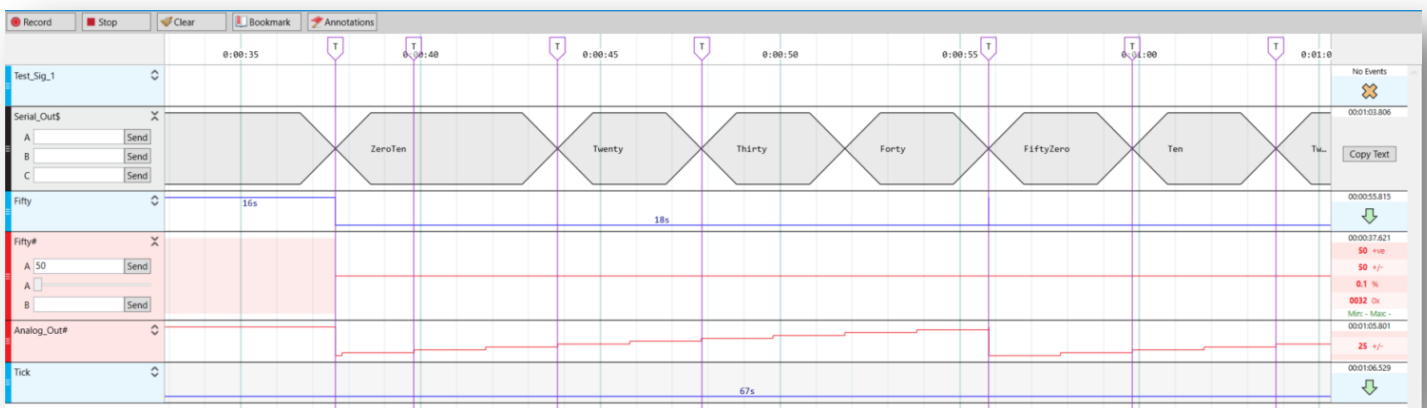
To add signals to the scope, drag a branch of the signal tree to the scope section - as shown in the image on the right. Drag individual signals or a whole branch to add all signals in that branch. Signals can also be added to the scope from the program tree, more information [in settings](#).



Debug Scope

This is the main debugging section. It contains a list of signals. Each row is a different signal; colour coded by type – blue for digital, black for serial, and red for analogue. Expanding a row brings up the stimulus options. The middle section displays the change in values of each signal over time. And the right section displays the current event for that signal.

Signals in this list can be re-ordered using drag and drop, or deleted by selecting and pressing delete on the keyboard. Multiple signals can be selected at once by holding ctrl, or all by ctrl + A.

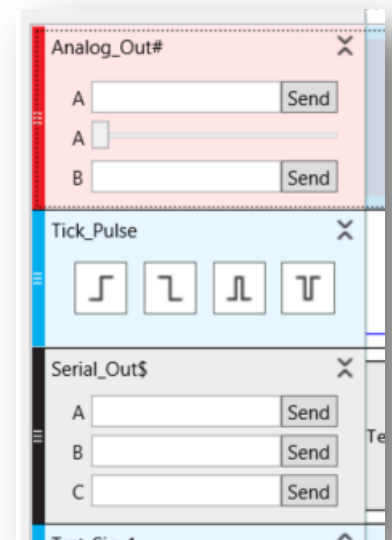


Signals can be expanded by clicking the icon in the top right of the box, or pressing the spacebar. This reveals the stimulus section shown on the right.

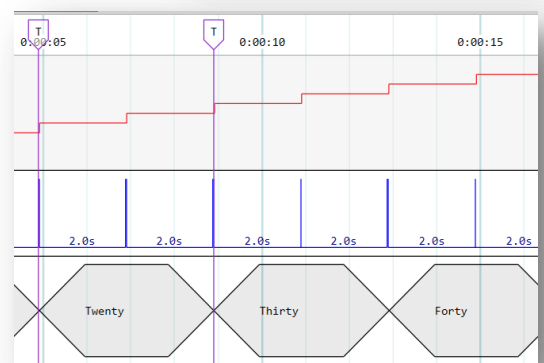
The analogue stimulus provides two text boxes to change the analogue value. The input can be numerical or a percentage. Textbox A and slider A are linked to the same value, textbox B is independent.

Digital stimulus contains four buttons to manipulate the signal.

Serial Stimulus has three textboxes – A, B, and C, they are independent of each other and allow you to send values to that signal by pressing 'send'



The scope shows the value of each signal, and at what time they change. Digital signal changes are labelled with how long they have been in that state (when there is enough space to do so).

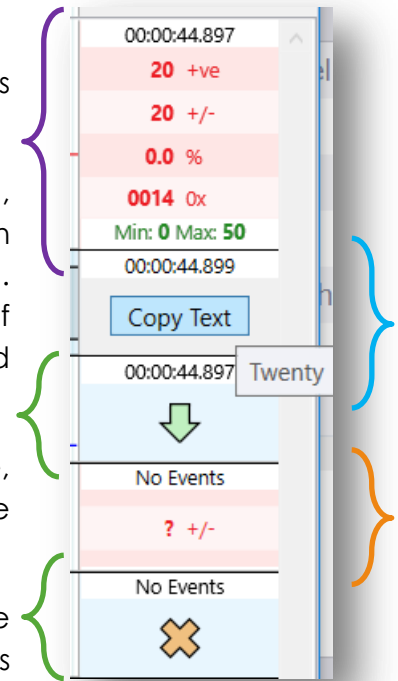


The column on the right of the scope shows the current events for each row.

An **expanded analogue signal** will display the value as signed, unsigned, percentage, and hexadecimal. It also gives an overall min and max value since the debugging session started. A **collapsed analogue** row just displays the unsigned value. If there have not been any events yet, a '?' will be displayed instead.

The **serial row** provides a button to copy the current value, hovering over it will display a tooltip of the value too, here the value is twenty.

The **digital row** shows an up or down arrow depending if the value is high or low. If there is no event yet there will be a cross instead.



Timeline

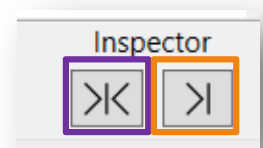
The timeline at the bottom of the pane can be adjusted to choose a specific timeframe to observe. The two orange thumbs at either end can be dragged to select a start and end point. The thumb in the middle lets you drag the whole bar left and right. The right thumb can be locked into the far right of the timeline; this keeps new data flowing into view on the scope.

The vertical lines indicate where events occurred, showing areas of more or less activity to help you navigate easier. When a signal in the scope is selected, the events of that signal are highlighted in orange.



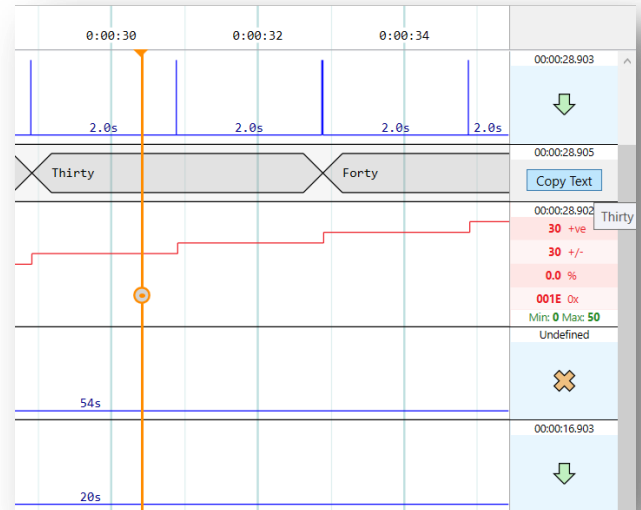
Inspector

The **inspector function** lets you inspect signal values at a point in the past, opposed to the most recent value. The **button on the right** removes the line, and takes you back to the most recent events.



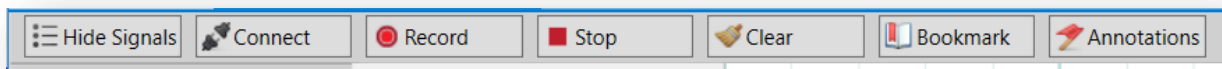
The inspector button brings up a vertical orange line on the scope, which can be dragged back and forth. The events column then displays the values at that time.

E.g. in this image, the events column shows the serial value as 'thirty', and the analogue value at 30, as that's where the inspector has been placed; whereas the most recent values are different to those.



Debugging Toolbar

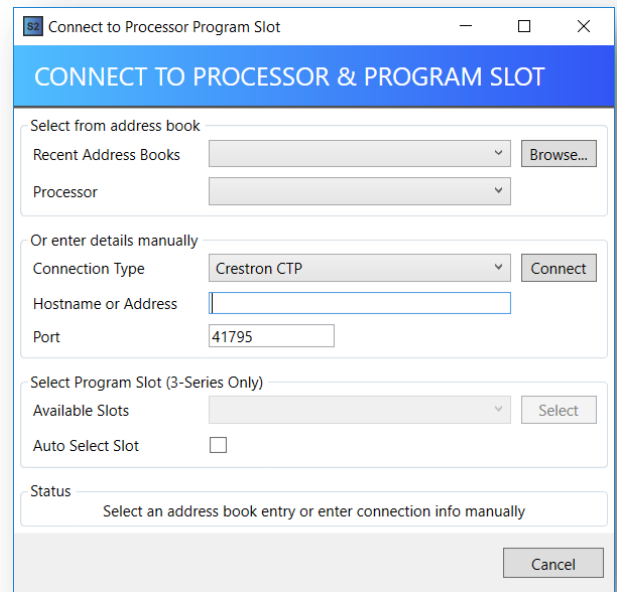
- Hide signals – opens and closes the signal tree
- Connect – opens a dialog box to open a connection with the processor
- Record – Begin recording signals from the current connection, used if you have stopped recording.
- Stop – Stop recording signals, the scope remains at its current state
- Clear – clears all signal data from the trace. Does not remove any signals from the list.
- Bookmark – inserts a green 'B' tag on the trace at the time you press the button
- Annotations – toggles the visibility of any notes and bookmarks on the trace.



Open a Connection

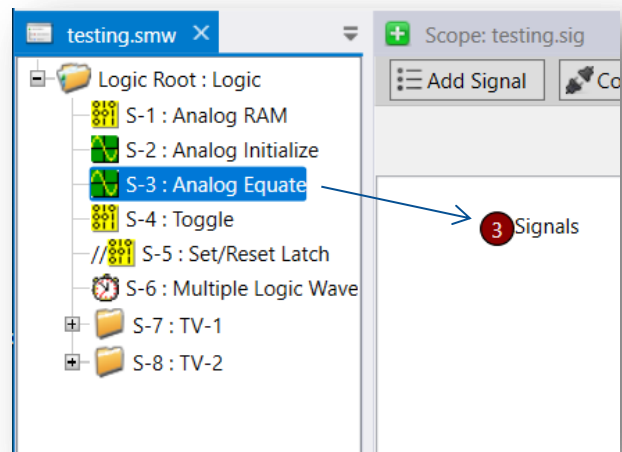
Clicking the connect button on the debugging toolbar will open this dialog. Either choose a connection from an address book or input the details manually in this window.

Pressing Connect will open a connection to the processor, then a list of available programs to debug will be given in the 'Available Slots' dropdown. If you already have a sig file open, the dropdown will automatically select the program to match. Press 'select' and the debugging session will begin.

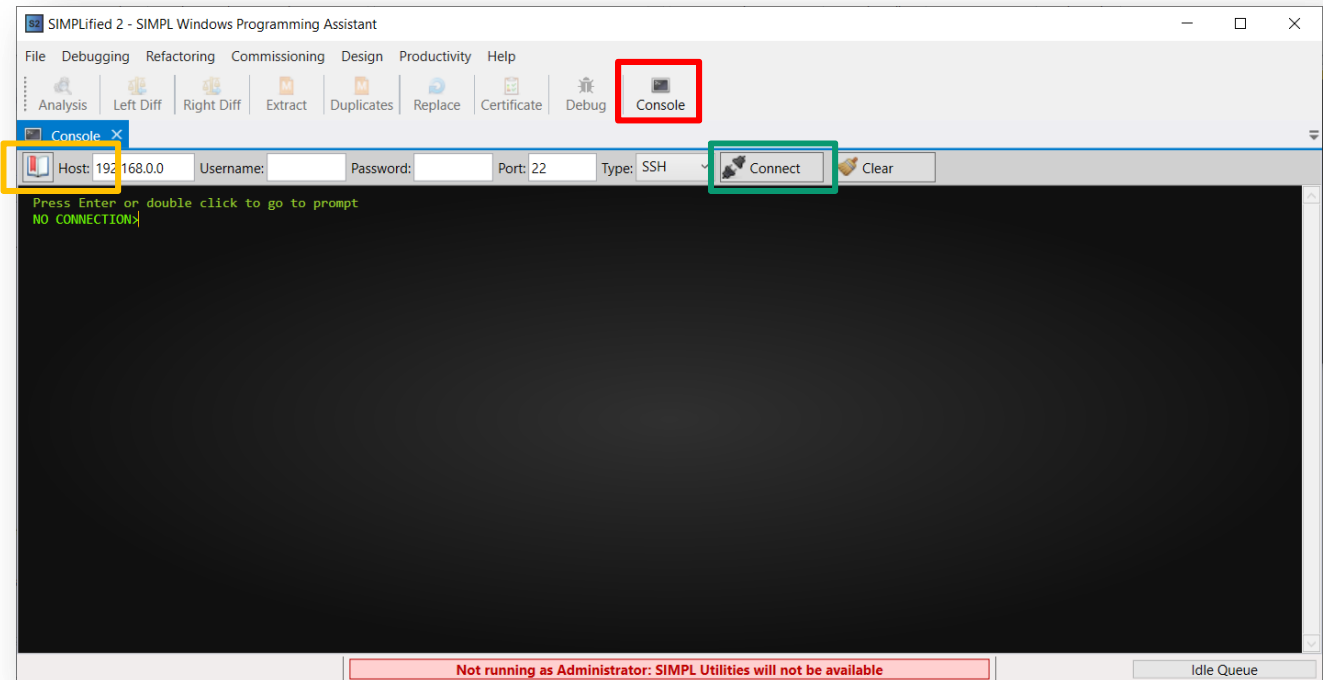


Adding Signals

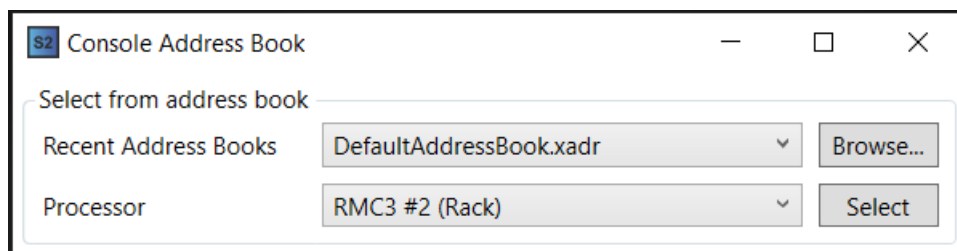
Signals can be added to the debugger either from the signal tree as explained [here](#), or from the program tree. If you have a debugging session open while the corresponding program tree is open, you can drag from the program tree to the debug pane to insert signals from that symbol, as shown on the right. It's possible to drag whole branches/subfolders at once.



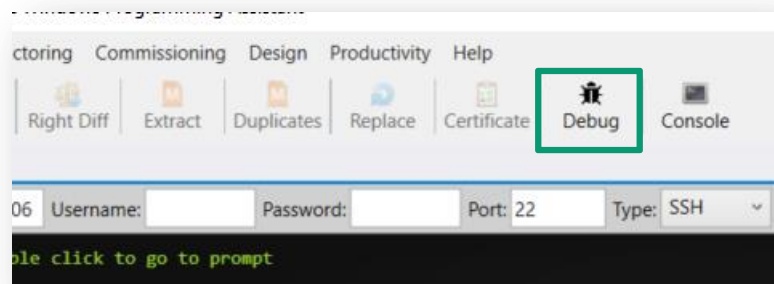
Console Window



Press the **console icon** on the toolbar to open the console window. Input your connection details and click **connect** to open a connection. The window remembers the details from the last connection to make it easy for you. You can also change address books by clicking the **address book**, and the window below will open.



And when the console has an active connection, the **debug icon** will enable so you can easily open a debug session for that connection.

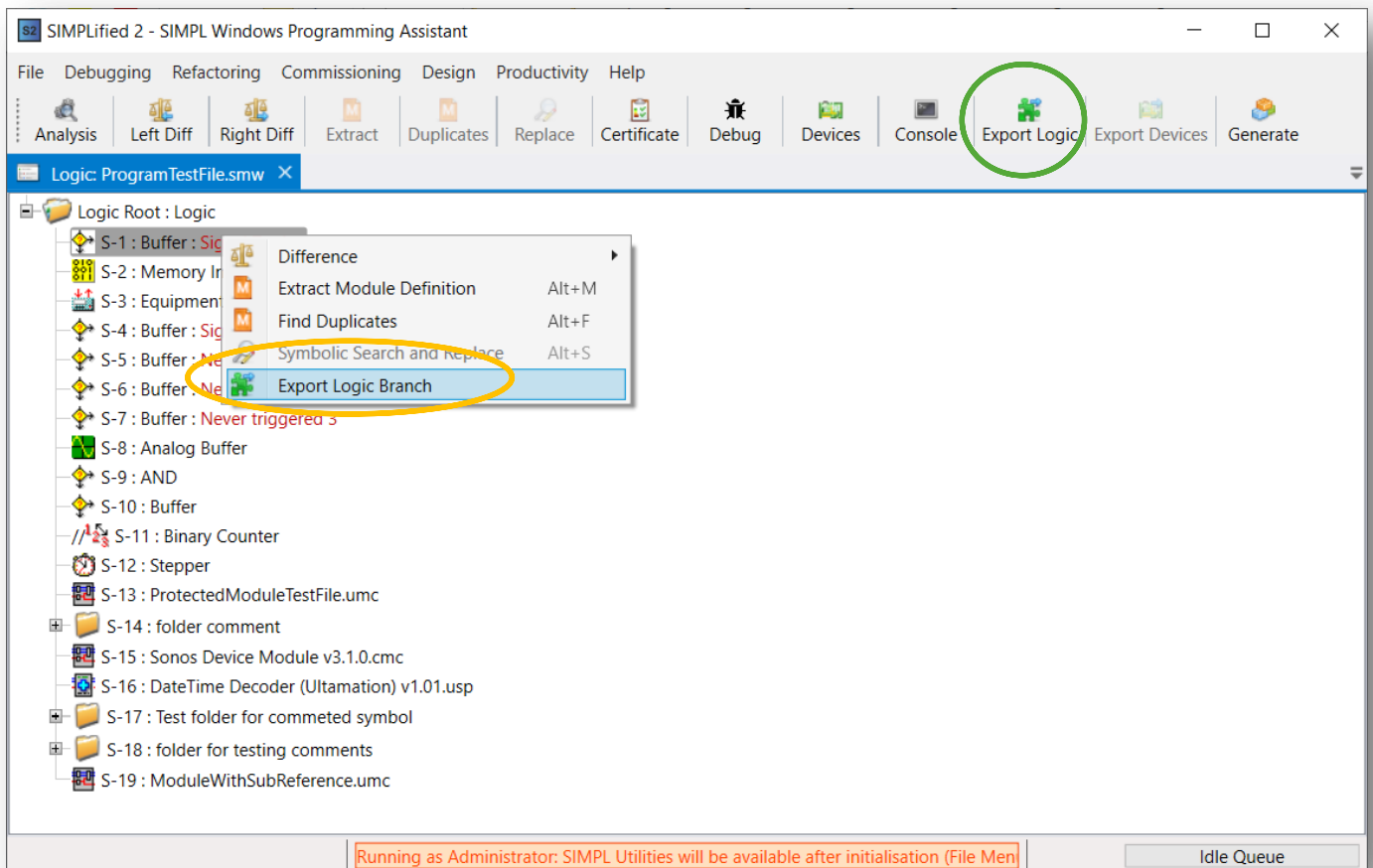


Productivity

Exporting from the Logic Tree

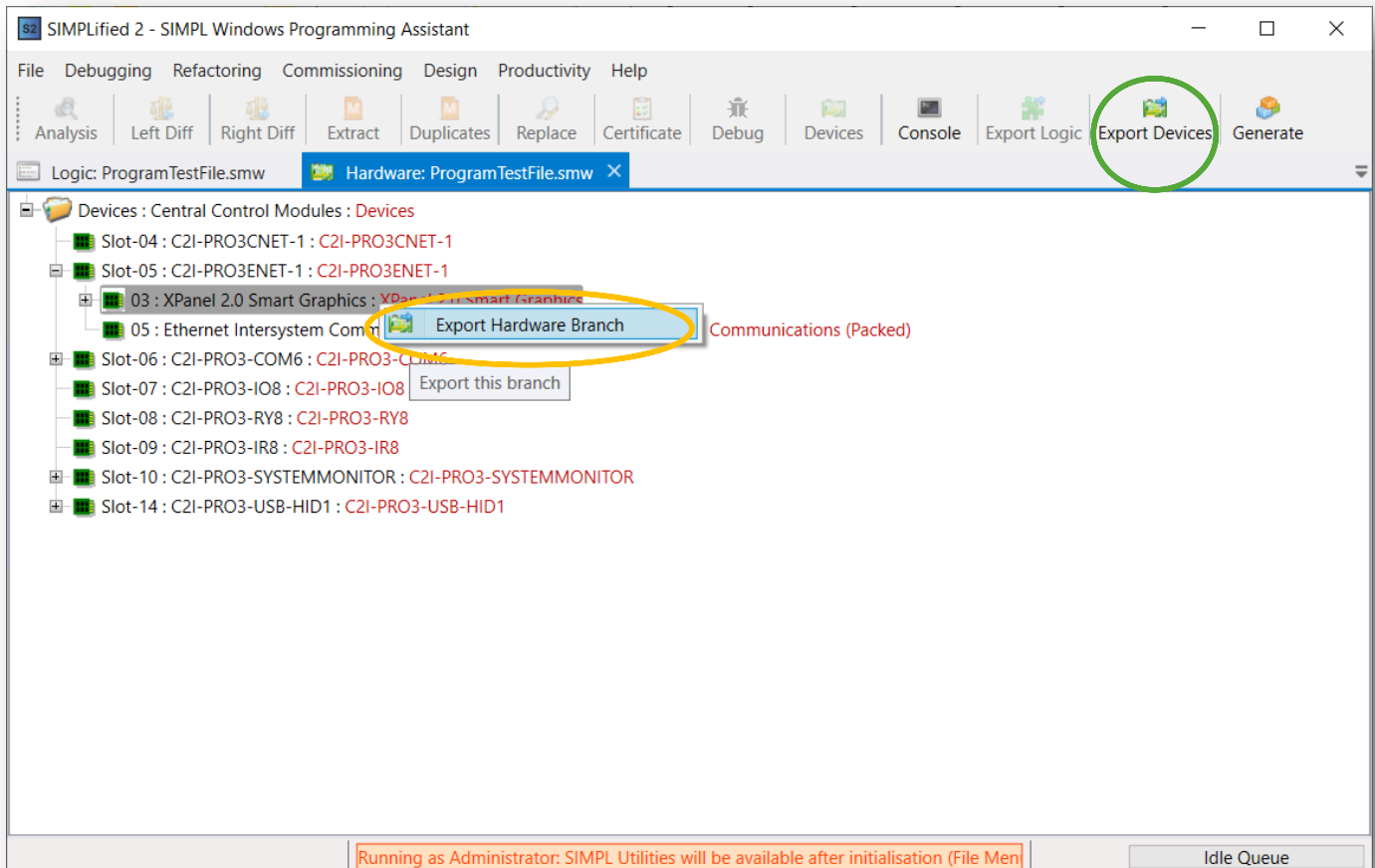
The export feature lets you export part of the logic or hardware tree to XML. This XML can then be used in our Program Generator to quickly create new programs with the same devices or logic. For more information on Program Generation check the 'SIMPL Program Generation Guide', downloadable from the shop page here - <https://shop.ultamation.com/index.php/product/100-simplified-2-productivity>

With a program pane open, you can select one of the items in the program to be exported. This can be a folder or a single item in the tree. When an item is selected, the 'Export Logic' menu option will be enabled. Either click the **button in the toolbar**, or you can right click the chosen item to open the context menu and select '**Export Logic Branch**' from that menu, as shown below. A dialogue box then opens to let you customise the export further; this is explained in the next section.



Exporting from the Hardware Tree

To export a branch of the hardware tree, the hardware tree pane must be open. Then select a branch of the Ethernet, Cresnet, or BACnet slots to enable the export option. You can either use the **toolbar button**, or right click the item, to show the **context menu option**, as highlighted in the image below. A dialogue box then opens to let you customise the export further; this is explained in the next section.



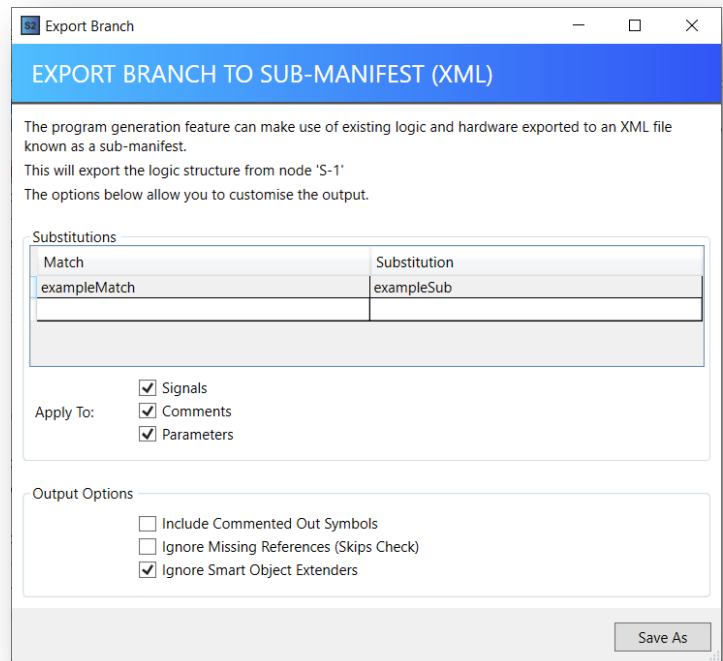
Customising the export

This dialog box will then appear to customise how you export the selected item.

Upon exporting, you'll have the option to replace strings. In the Substitutions section, input the string(s) you want to replace under 'Match' and the string(s) to insert instead under 'Substitution'. Then use the check boxes to specify which areas the substitution should be applied to – Signals, comments, and parameters. In our example, any instances of 'exampleMatch' will be replaced with 'exampleSub' in signals, comments, and parameters.

In the 'Output Options' section you have the option to include or exclude commented out symbols, they are excluded by default. 'Ignore Missing References' will skip the dependency warning if any missing ones are found. And Smart Object extenders are also excluded by default, as they are not supported fully right now.

Click 'Save As' and you'll be asked to choose the location to save your export.



Program Generation

The program generation feature allows you to create SIMPL windows programs from either an XML or csv file. You can also develop your own plugin to generate programs from your own framework.

We have made a separate document for explaining the program generation feature, this is called 'SIMPL Program Generation Guide' and can be found in the Help menu of SIMPLified 2, or on the downloads section of the productivity page on the shop here - <https://shop.ultamation.com/index.php/product/100-simplified-2-productivity>

Command Line Compiler

The Command Line Compiler can compile your program with a simple command in the console window.

We have made a separate document for explaining the Command Line Compiler program, called 'SIMPLified 2 Command Line Compiler User Guide'. This can be found in the Help menu of SIMPLified 2, or on the downloads section of the productivity page on the shop here - <https://shop.ultamation.com/index.php/product/100-simplified-2-productivity>

Key Bindings

Alt + A	Analyses the program that's active
Alt + C	Generates certificate for the open program
Alt + D	Opens a debug session of the active program, if the sig file is found
Alt + E	Extract Module
Alt + M	Module Diff
Alt + L	Sets the current selection as the left side of a diff
Alt + R	Sets the current selection as the right side of a diff

In a debugging window-

Ctrl	Select Multiple signals
Del	Delete the selected signals
Alt + Del	Delete the unselected signals
Space	Expand/collapse the signal row

Roadmap

The following 'significant' features are ideas that we think would make a good addition to SIMPLified 2. That doesn't mean any of them are in active development, or will ever be (they may turn out to be impractical, too big, or not a good fit on reflection) but we still wanted to give them some visibility so you see what direction SIMPLified 2 may take in the future.

- Debugging
 - Additional analysers for logic issues
- Refactoring
 - Additional analysers for logic simplification
 - Extract logic tree from a running program for DIFF
 - Improve the identification of common logic blocks for modularisation
- Commissioning
 - Commissioning reports to aid on-site installation
 - Device discovery with customisable command menus
 - Auto-generate toolbox address books
 - Text console with debug info suppressed and formatted/coloured output
- Productivity
 - Automated SIMPL compile
 - Automated upload
- Design
 - SIMPL program score
 - Load testing
 - Auto-generated SIMPL program layout (localised)
 - Overlay real-time debug info

Release History

1.0 The initial release of SIMPLified 2.

1.2 First release of timeline debugger.

1.4 **New Features –**

Added support for username/passwords with SSH

Added analysis widget

Added analyser Toast notifications

Added Device Discovery Beta Module

New analyser – for symbol with fatal parameters: Lists symbols where unchecked parameter values can cause the processor to crash.

Bug Fixes –

Jammable analyser was incorrectly identifying internal SIMPLPlus modules as exclusive.

Cross-point IDs of 0 are now excluded from the duplicate Cross-point ID analyser.

Fixed address book loader for entries that require passwords.

1.5 **New Features –**

'Todo' analyser

Bug Fixes –

Signal tree now ordered alphabetically

Toast notifications appear correctly

1.6 **New Features –**

Looped RAM Signals Analyser

Analyser results now display in order of signal type then alphabetically

Randomly unlocked analysers

Bug Fixes –

SSH connection now more reliable

1.7 **New Features –**

Updated the encryption method used

Bug Fixes –

The timescale on the debug timeline is now correct and consistent

Handles if invalid directories are trying to be opened from recent list

1.8 - **Bug Fixes –**

Issue registering keys in recent build fixed

1.9 - **New Features –**

Console window added

Search and Replace feature added

Recalculated certificate scores

Updated the UI of the debugger signals

1.11 – **New Features –**

Comments added to Search and Replace

Hardware Tree added

Logic Exporting added

Program Generation Added

Bug Fixes –

Some fixes to the toolbar buttons being enabled and disabled

Hardware path is more accurate (eg slot numbers and IP-ID numbers)

1.12 – **New Features –**

All help documents added to the help menu

Bug Fixes –

Program Generator wouldn't generate

1.13 – **New Features –**

More help documents added to the help menu

Added a transformer to generate programs with BACnet devices

Ability to export BACnet devices from the device tree

Command Line Compiler compatibility

Bug Fixes –

Issue copying large groups of signals on the Module Extraction

Search and Replace mismatching parameters due to `_SKIP_` cues

1.14 - New Features –

Added command Line Compiler project

Added remote and remote host devices to manifest

Bug Fixes –

Fixed issue with Search and Replace

Fixed issue copying long lists of signals

1.15 - New Features –

BACnet added to Program generation

1.16 - New Features –

Added Comissioning menu back in

Added Device Discovery feature back in

1.17 - New Features –

Bug Fixes –

Improved connection overall, especially for 4-series processors

Changes to the program-diff algorithm