CockpitHardwareHUB_v2 – v2.1.1

Tool for Cockpit Builders to connect hardware to Microsoft Flight Simulator 2020

There are many tools available to connect flight simulator hardware to MSFS 2020. But a lot of these tools (most of them?) require some configuration before the connected hardware can work. Other tools only work with one specific brand of microcontrollers (example: Arduino).

This is where CockpitHardwareHUB_v2 comes in with 2 main advantages:

- It works with any microcontroller, as long as it provides a serial interface through USB
- Each device "pushes" its own configuration requirements, without the need of extra maintenance

Although, CockpitHardwareHUB_v2 requires that you write your own software for your hardware. But that of course provides endless flexibility.

The first version of CockpitHardwareHUB made use of its own WASM module. CockpitHardwareHUB_v2 uses an adapted version of the WASM module based on WASimCommander written by mpaperno. WASimCommander handles all the communication with MSFS 2020 via SimConnect and a WASM Module.

The current version of CockpitHardwareHUB_v2 also uses the C# class SerialPortManager.cs written by Paul van Dinther. This class embeds the detection of Serial Port USB devices in a hot swappable way. The implementation has been adapted allowing it to find more devices (example: devices using the CH340 chip could not be found with the original version of the class).

Properties and Variables

Once CockpitHardwareHUB_v2 is started and connected with MSFS 2020 (MSFS 2020 must be running), it uses WMI (Windows Management Instrumentation) to listen to 'Win32_PnPEntity' devices where PnPClass = 'Ports' and Name contains 'COM'. When hardware is connected, it is identified by the PNPDevice string with the following format:

```
USB\VID_vvvv&PID_pppp\nnnnnnnn
```

This string contains the following elements:

USB	This indicates that it is a USB device
VID_vvvv	This is the VendorID of the device in a 16 bit hexadecimal format
PID_pppp	This is the ProductID of the device in a 16 bit hexadecimal format
nnnnnnnn	This is the serial number of the device

For CockpitHardwareHUB_v2 to work correctly, each device should have a unique VendorID, ProductID and SerialNumber combination. It is possible to filter on these values, which allows even isolating one single device without having to disconnect other devices from the USB bus.

When a device is detected, CockpitHardwareHUB_v2 opens its SerialPort and enables DTR and RTS. After that, CockpitHardwareHUB_v2 waits 2 seconds for an ‘Acknowledge’, which simply is the string “A\n” (character ‘A’ with newline). This is specifically designed for Arduino type of devices. Enabling DTR/RTS is forcing the Arduino to reset, which means that ‘setup()’ is called. At the end of the ‘setup()’, a ‘Serial.print(“A\n”);’ should be added, which tells the outside world that the reset is finished, and serial commands can be sent. By capturing that “A\n” string, CockpitHardwareHUB_v2 knows that the Arduino is ready. Not waiting, and starting to send commands too early, might keep the Arduino in a ‘special state’ that is used by the Arduino IDE to load new sketches.

CockpitHardwareHUB_v2 doesn’t do anything with it this “A\n” string. Earlier developed hardware that doesn’t send this “A\n” string will still work. In this case, 2 seconds are being waited, and that’s it. Because CockpitHardwareHUB_v2 uses asynchronous methods, even if multiple devices start connecting at the same moment, the total time lost will only be 2 seconds.

Once a device is detected, a ‘Registration Process’ takes place. For this, CockpitHardwareHUB_v2 will send the below commands to the device and expect some answers.

Command	Reaction by device
RESET\n	Brings the device in 'Registration Mode'
	Here is a delay of 200 msec
IDENT\n	Device answers with its DeviceName followed by '\n'
	Device answers with its ProcessorType followed by '\n'
REGISTER\n	Device starts sending all its Property strings one by one followed by '\n'
	If no more properties, device sends a final '\n' (empty string)

If this fails, CockpitHardwareHUB_v2 waits 12 seconds, and then starts over again. This 12 second is also specifically for Arduino devices. If an Arduino is in that ‘special state’, but keeps receiving serial data that is not complying with some kind of protocol, it keeps waiting 10 seconds after each character received (it keeps resetting that timer). If CockpitHardwareHUB_v2 would resend the above commands within less than 10 seconds, this would go on forever. That’s where the 12 seconds waiting time comes from, as it guarantees that the Arduino is definitely out of that ‘special state’.

Once all properties are successfully received by CockpitHardwareHUB_v2, they are added in the Property Pool where they become ‘Variables’. These variables are then registered with MSFS using SimConnect and/or the WASM module. If different devices register the same property, the same variable is used. This prevents unnecessary registrations which would be a waste of resources and might slow down the system.

From then onwards, properties are identified by ‘Property Id’, which is the order in which the Properties have been registered, starting at 1. CockpitHardwareHUB_v2 translates these in its own internal numbering system for the variables. Working with numbers is a lot more efficient in microcontroller hardware, in which comparing with a number (example: in a switch statement) is faster than comparing with strings.

Once the registration process is finished, the device can start doing it's normal operation, which is listening to data updates coming from MSFS 2020 and sending data when the user interacts with buttons, switches, encoders, etc…

The power of this concept

The power of CockpitHardwareHUB_v2 lays in the fact that each 'Device Property' contains all the information needed to instruct MSFS what is required. This replaces the configuration files that you need in other tools. It means that each device is 'self-containing', which gives the advantage that when you develop your device, you can make changes in properties on the fly without having to adapt configuration files somewhere else. CockpitHardwareHUB_v2 doesn't need any configuration depending on the devices connected to it. Each time you disconnect and reconnect the device, the previous registrations are removed, and the new registrations are loaded.

This can improve your development workflow a lot. You only have to concentrate on developing your hardware, and not looking to any other application or tool.

The fact that devices can are hot swappable is also an advantage during developing your firmware of your devices. It means that you can iterate between different compilations and load the new versions which will repeatedly reboot your device. CockpitHardwareHUB_v2 will react by disconnecting and unregistering all properties while you're your device is rebooting (connection is lost) and will reconnect and register the new properties when the device is restarted.

This concept is also an advantage when you unplug and plug devices while you are building your cockpit or are doing troubleshooting. You don't need to press any button on CockpitHardwareHUB_v2, as all will be done automatically. Once CockpitHardwareHUB_v2 is connected, you don't have to look at it any longer.

Construction of a ‘Device Property’

The property strings returned by the device always have the below format:

ValueType	_	RW	_	VarType	:	VarName	:Index	,Unit
VOID		R		A		Variable name with optional index and unit		
INT8		W		L				
INT16		RW		K				
INT32				X				
INT64								
FLOAT32								
FLOAT64								
STRING16								
STRING32								
STRING64								
STRING128								
STRING256								

- **ValueType**: This is the type of the value of the property when data is being sent or received. CockpitHardwareHUB_v2 will check if the value fits in the type.
- **RW**: This indicates if the variable is a ‘R’ead (means that MSFS can send data to your device) or ‘W’rite (means that the device can send data to MSFS). A variable can be both Read and Write.
  In case the ValueType is ‘VOID’, then the ‘_[R][W]’ should not be included.
- **VarType**: These are the variable types that are currently supported:
  - A: These are ‘Simulation Variables’
  - L: These are ‘Local Variables’ (specific for the add-on)
  - K: These are events. If an event has a ‘.’ in the VarName, then it is a ‘Custom Event’ (example: A32NX.FCU_SPD_INC). Otherwise it is a normal ‘key events’.
  - X: These are variables that are using the Gauge API function ‘execute_calculator_code’.
- **VarName**: This is the name of the variable
- **Index:** (optional): Some variables have an index, which is added by a colon and the extension number (example: LIGHT POTENTIOMETER:84).
- **Unit:** (only for VarType ‘A’): Only A-Vars need a unit.

Some examples of Property strings:

ID	Property
001	INT32_R_A:LIGHT STROBE,bool
002	INT32_RW_L:LIGHTING_LANDING_2
003	VOID_K:A32NX.FCU_SPD_INC
004	INT32_R_A:LIGHT POTENTIOMETER:84,percent
005	VOID_K:LIGHT_POTENTIOMETER_SET
006	VOID_K:ANTI_ICE_TOGGLE_ENG1
007	STRING16_R_X:(A:ATC FLIGHT NUMBER,string)
008	STRING64_R_X:(A:TITLE,string)

CockpitHardwareHUB_v2 applies some restrictions per VarType:

- ‘A’-Var
  - Can only be INT8, INT16, INT32, INT64, FLOAT32 and FLOAT64
  - ‘Unit’ is mandatory and needs to be recognized by MSFS/WASimCommander
- ‘L’-Var
  - Can only be INT8, INT16, INT32, INT64, FLOAT32 and FLOAT64
- ‘K’-Var
  - Can only be INT8, INT16, INT32 or VOID
  - Can only be ‘W’rite
- ‘X’-Var
  - Can not be both ‘R’ead and ‘W’rite
  - Can be parameterized with maximum 5 parameters using {0}, {1}, {2}, {3} and {4}

Also note that ‘VOID’ can’t have a ‘R’ead or ‘W’rite, as it always implies a ‘W’rite operation (is mainly used for ‘K’-Vars).

Variable strings are case insensitive. In CockpitHardwareHUB_v2, they are converted to Upcase. This means that ‘INT32_R_A:LIGHT STROBE,bool’ and ‘INT32_R_A:LIGHT STROBE,BOOL’ are the same variables.

Be carefull with spaces. If you intend to use the same variable between 2 different devices, but one device uses ‘INT32_R_A:LIGHT STROBE,bool’, and the other uses ‘INT32_R_A:LIGHT STROBE, bool’ (extra space before unit), this is stored as a different variable.

// Future improvement: String data should preserve case.

Sending and receiving data

Commands have the following format:

```
NNN[=[data1[;data2;[data3[;data4[;data5]]]]]\n
```

- NNN is the property number according to the sequence sent by the device during registration, starting with 001.
- ‘=’ -sign normally means that some data will follow. A command can be sent without data, which implies the value 0.
- ‘dataN’ which is the data fitting in the type ValType. A command can have up to 5 values, which are separated by ‘;’-character. For FLOAT type of data, only 3 fractional digits are used.
- Every command is terminated by ‘\n’

A device can only send data to MSFS 2020 for ‘W’rite variables. This means that in the example above, the following variables can be used to send data: 002, 003, 005 and 006.

A will only receive data for ‘R’ead variables. This means that in the example above, the following variables can be used to receive data: 001, 002, 004, 007 and 008.

After all properties have been registered, CockpitHardwareHUB_v2 will send the values of all ‘R’read commands to the device. After that, only the changes are being sent.

Acknowledge sequence

If CockpitHardwareHUB_v2 sends data to a device, it waits maximum 150msec for an acknowledge sequence, which is a short sequence of '**A\n**'. This means that when the device receives data, and it is able to parse it correctly, it should immediately send an 'A\n' sequence, and after that proceed with other work. If the sequence is not received in time, CockpitHardwareHUB_v2 will send the command a second time. If that doesn't succeed, an error is logged.

There is no acknowledge sequence if CockpitHardwareHUB_v2 receives data. The reasoning behind it is that PC's these days are fast enough to process simple serial data in time (each 'RxPump' runs in its own thread). It also makes development of device firmware a lot simpler and more efficient, because some simple microcontroller architectures don't have multi-threading capabilities. This concept avoids that devices must wait for an acknowledge, and can already continue processing the next step. Especially when using rotating encoders, which can trigger more than 10 commands per second, it is crucial to allow devices to process these quickly, and not loose time while waiting for an acknowledge sequence.

Some examples

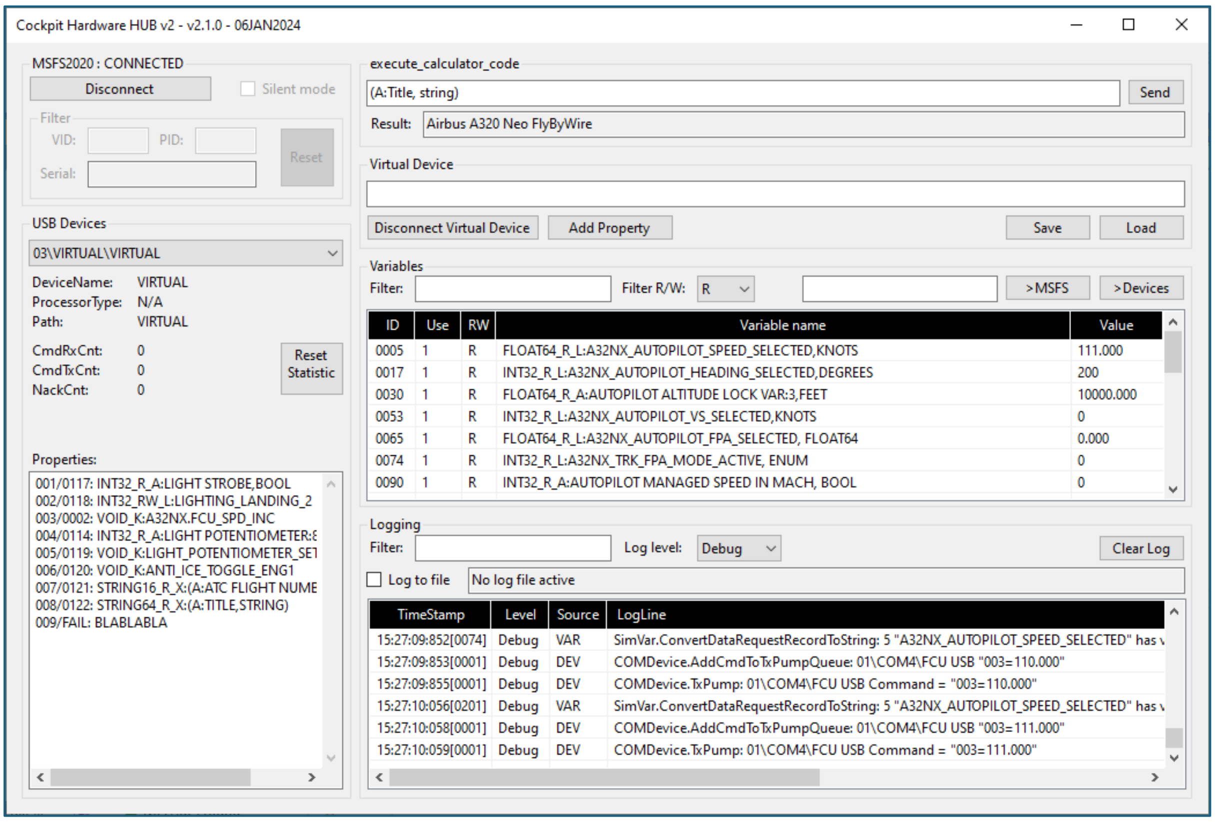
- MSFS 2020 can send the string '001=0\n' or '001=1\n', returning the status of the 'LIGHT STROBE'. The device answers with 'A\n'.
- MSFS 2020 can send the string '002=0\n', '002=1\n' or '002=2\n', depending on the position of the 'LIGHTING LANDING 2' switch. The device answers with 'A\n'.
  Your device can also send the same commands back to MSFS 2020 to change the 'LIGHTING LANDING 2' switch.
- Your device can send the string '003\n', '003=\n' or '003=1\n' to MSFS 2020 to increase the SPD by one. Be aware that some 'K'-Vars completely ignore the value (in the above example, '003=0\n' would also increase the SPD). Also note that the '=' is optional if no value is required (which is typical for a VOID).
- MSFS 2020 can send the string '004=50\n', which indicates that the 'LIGHT POTENTIOMETER:84' is at 50 percent. The device answers with 'A\n'. Using the correct 'Unit' is important. The variable could have been registered with a FLOAT32 using the unit 'Enum', which would return the value 0.5 instead. (It seems that 'enum' can be used for a lot of value types).
- Your device can use the string '005=84;25\n' to set the value of 'LIGHT POTENTIOMETER:84' to 25%. (check the SDK Documentation to find out the command syntax).
- MSFS can send the string '007=1123\n' to return the 'ATC FLIGHT NUMBER'. The device answers with 'A\n'.
- MSFS can send the string '008=Airbus A320 Neo FlyByWire' to return the 'TITLE'. The device answers with 'A\n'.

User Interface tailored to help developing your hardware

Some research will be required to know what variables are needed and how to use them. This is the typical struggle of every Cockpit Hardware builder. It is well known that the SDK Documentation is not always very clear and might be missing some details. Variables might also have different meanings for different planes. For add-ons, you must refer to the information provided by the owner of the package, which is also not always obvious.

CockpitHardwareHUB_v2 is a tool to help facilitate this development cycle. It is not only a tool to interface hardware with MSFS 2020, but it also can assist in experimenting with variables before implementing them in your hardware.

The User Interface is divided in 6 groups:


Cockpit Hardware HUB v2 - v2.1.0 - 06JAN2024

MSFS2020 : CONNECTED

Filter
VID:
PID:
Serial:

USB Devices
03\VIRTUAL\VIRTUAL

DeviceName:	VIRTUAL
ProcessorType:	N/A
Path:	VIRTUAL
CmdRxCnt:	0
CmdTxCnt:	0
NackCnt:	0

Properties:
```
001/0117: INT32_R_A:LIGHT STROBE,BOOL
002/0118: INT32_RW_L:LIGHTING_LANDING_2
003/0002: VOID_K:A32NX.FCU_SPD_INC
004/0114: INT32_R_A:LIGHT POTENTIOMETER:8
005/0119: VOID_K:LIGHT_POTENTIOMETER_SET
006/0120: VOID_K:ANTI_ICE_TOGGLE_ENG1
007/0121: STRING16_R_X:(A:ATC FLIGHT NUME
008/0122: STRING64_R_X:(A:TITLE,STRING)
009/FAIL: BLABLABLA
```

execute_calculator_code
(A:Title, string)
Result: Airbus A320 Neo FlyByWire

Virtual Device

Variables
Filter:
Filter R/W: R

ID	Use	RW	Variable name	Value
0005	1	R	FLOAT64_R_L:A32NX_AUTOPILOT_SPEED_SELECTED,KNOTS	111.000
0017	1	R	INT32_R_L:A32NX_AUTOPILOT_HEADING_SELECTED,DEGREES	200
0030	1	R	FLOAT64_R_A:AUTOPILOT ALTITUDE LOCK VAR:3,FEET	10000.000
0053	1	R	INT32_R_L:A32NX_AUTOPILOT_VS_SELECTED,KNOTS	0
0065	1	R	FLOAT64_R_L:A32NX_AUTOPILOT_FPA_SELECTED, FLOAT64	0.000
0074	1	R	INT32_R_L:A32NX_TRK_FPA_MODE_ACTIVE, ENUM	0
0090	1	R	INT32_R_A:AUTOPILOT MANAGED SPEED IN MACH, BOOL	0

Logging
Filter:
Log level: Debug
Log to file: No log file active

```
TimeStamp           Level  Source  LogLine
15:27:09:852[0074]  Debug  VAR     SimVar.ConvertDataRequestRecordToString: 5 "A32NX_AUTOPILOT_SPEED_SELECTED" has v
15:27:09:853[0001]  Debug  DEV     COMDevice.AddCmdToTxPumpQueue: 01\COM4\FCU USB "003=110.000"
15:27:09:855[0001]  Debug  DEV     COMDevice.TxPump: 01\COM4\FCU USB Command = "003=110.000"
15:27:10:056[0201]  Debug  VAR     SimVar.ConvertDataRequestRecordToString: 5 "A32NX_AUTOPILOT_SPEED_SELECTED" has v
15:27:10:058[0001]  Debug  DEV     COMDevice.AddCmdToTxPumpQueue: 01\COM4\FCU USB "003=111.000"
15:27:10:059[0001]  Debug  DEV     COMDevice.TxPump: 01\COM4\FCU USB Command = "003=111.000"
```

Connection group

This shows the Connect/Disconnect button. Connecting means that CockpitHardwareHUB_v2 will connect with MSFS 2020 through SimConnect and will then connect with the WASM module. Only if both connections are successful, CockpitHardwareHUB_v2 considers this as 'connected'.

If 'Silent mode' is enabled, all user interface activity, including logging, is disabled. This improves efficiency and speed and is used for just 'flying with the sim'.

It is possible to filter on specific 'VID' (Vendor ID), 'PID' (Product ID) and/or Serial Number. VID and PID can be entered in hexadecimal format (0xHHHH) or decimal format (NNNNN). Values should be between 1 and 65535. The Serial Number can be maximum 8 characters long. CockpitHardwareHUB_v2 stores the values in the registry. They can easily be cleared by pressing the Reset button.

Toggling 'Silent mode' or changing 'Filter' values can only be done when CockpitHardwareHUB_v2 is disconnected.

USB Devices

This shows the properties of each detected device. Using the dropdown box, you can switch between connected devices that are shown as 'DeviceID\Comport\DeviceName'.

For each device you can see the DeviceName and Processortype (see 'IDENT\n' command above), the full path, and communication statistics CmdRxCnt, CmdTxCnt and NackCnt (which are pretty much self-explanatory). The statistics can be reset by pressing the button 'Reset Statistics'.

Finally, a list with all registered properties for the selected device. Each entry shows the 'PropertyID' which is the command ID used to send data to/from the device, the corresponding 'Variable ID', and the full 'Property String'. If a Property could not correctly be registered (Example: due to some parsing error), the 'Variable ID' will show 'FAIL'.

execute_calculator_code

This is a very helpful tool to experiment with commands. The Gauge API function 'execute_calculator_code' can be used to send all kinds of commands to MSFS. If return values are available, they will be shown in the result field. This function uses 'Reverse Polish Notation'. RPN-strings that are can correctly be executed, can also be registered using an 'X'-Var in CockpitHardwareHUB_v2, and then be used by your hardware by simply sending the 'PropertyID'. If registered as a 'R'ead 'X'-Var, the returned data is sent back to your device.

Example using FBW A32NX (free) add-on airplane:

The below RPN-sequence will set both Left and Right EFIS to 'CSTR'.

1 (>L:A32NX_EFIS_L_OPTION,enum) 1 (>L:A32NX_EFIS_R_OPTION,enum)

This RPN string doesn't return a value, in which case 'Result' will show '0'.

You can experiment with this sequence and use the other possible values such as 2 (VOR.D), 3 (WPT), 4 (NDB) and 5 (ARPT).

Now you know to use this command, you can use it in your device with the below property string:

VOID_X: 1 (>L:A32NX_EFIS_L_OPTION,enum) 1 (>L:A32NX_EFIS_R_OPTION,enum)

Assuming that this is command 10, your device can send the string '010' to simply put both EFIS left and right to 'CSTR'.

You can even go one step further by using parameters which are allowed for 'X'-Vars. The below property string uses 2 parameters. Be aware that when using parameters, you need to provide a ValType (CockpitHardwareHUB_v2 needs to know what type the parameters can be – only restriction is that they all need to be the same type).

INT8_W_X: {0} (>L:A32NX_EFIS_L_OPTION,enum) {1} (>L:A32NX_EFIS_R_OPTION,enum)

Now you can independently control the EFIS left and right. Sending the string '010=2;3' will set the EFIS left to 'VOR.D' and the EFIS right to 'WPT'.

Virtual Device

This is a second tool to help finding the correct properties for your device. Once you have identified the correct commands and values (maybe with the help of 'execute_calculator_code'), you can now go one step further and build property strings. A Virtual Device just acts as a real USB device connected to your computer.

- **Connect/Disconnect Virtual Device**: Once you are connected with MSFS 2020, you can use 'Connect Virtual Device', which will add it to the 'USB Devices' group. If connected, the other buttons become active. To unregister all properties registered by the VIRTUAL device (see below), you must Disconnect. There is no way to unregister individual variables, because they need to stay in sequence (no gaps allowed in the Property ID numbering).
- **Add Property**: This adds the property string from the input field to the virtual device and registers it. You will get feedback if something is wrong. If all is ok, the property will be added and registered, and will be found in the Property list on the left. This is completely simulating as if your device would receive the "REGISTER" command and is responding with this Property string.
- **Save**: This allows you to save all already registered properties of the VIRTUAL device to a *.txt file.
- **Load**: This appends Property Strings from a *.txt file to the already registered properties in your VIRTUAL device. Be aware that the same property can always be registered twice (there is no added value of course). If you want to remove properties, you will have to Disconnect the VIRTUAL device.

You can easily create your own property files, which are simple text-files with lines of property strings. This is an ideal way to prepare your firmware developments.

Variables

This ListBox shows all the successfully registered variables. You can filter on 'Variable Name' by entering some text and pressing 'ENTER'. You can also filter on the RW flag (very handy to only show the 'R' variables and see their values changing in real time).

You can send a command to MSFS or devices by entering it in the entry field and pressing the '>MSFS' or '>Devices' button. Be aware to use the 'Variable ID'.

- **>MSFS**: This is only possible for '[R]W'-variables and allows to send a variable and its data to MSFS 2020. It is as if you would send a command from your device to MSFS 2020, but here you directly use the 'Variable ID' instead of the 'Property ID'.
- **>Devices**: This is only possible for 'R[W]'-variables and allows to send a variable and its data to all devices that have registered for it (excluding the VIRTUAL device).

The Variable list shows the following information:

- **ID**: The Variable ID
- **Use**: The usage count. If more than one device registers for the same variable, the usage count will increase. If a device is disconnected, the usage count of its registered variables will decrease. Once the usage count becomes 0, the variable is unregistered with MSFS 2020.
- **Variable name**: The full Property String of the variable.

- **Value**: For Read variables, this shows the value in real time. If you want to see specific Read variables, you can Filter 'R'-variables, combined with filtering on the Variable name.

Logging

This shows all kinds of logging information. Although a lot can be cryptic, if the level shows 'Error', there is definitely something wrong. You can Filter on the LogLine text and change the Log Level. Be careful with Log Level 'Trace' because that will make the UI less responsive. You can Clear the Log by pressing the 'Clear Log' button.

You can also log into a file. When enabling this option, you are asked for a filename. Be aware that logging to file can create some performance drop. But it can be handy, when you have an issue, and want to send some logging data.

The Logging list shows the following information:

- **TimeStamp**: This is the time when the LogLine was registered up to msec precision. The difference in msec between this LogLine and the previous one is shown between square brackets. If the difference is 10 seconds or more, '[9999]' is shown. Be aware that sometimes negative values can be shown because logging is in some cases asynchronous.
- **Level**: Shows the log level. Only logging information of a level higher than the chosen level will be shown. By default, the level is 'none', which means that logging is disabled. It is not a bad idea to use the level 'warning', which only logs if something goes wrong, or if the attention of the user is required.
- **Source**: This is a code of the source of the Logging information and is only useful for the developer of this application.
    - CLT: WASimCommander Client Module
    - SRV: WASimCommander WASM Module
    - APP: Common application
    - DEV: Related to the COMDevice
    - VAR: Related to the SimVar
    - PPL: Related to the PropertyPool
- **LogLine**: The logging information

Credits

CockpitHardwareHUB_v2 is the successor of CockpitHardwareHUB. This was the result of several posts that I wrote on the Flight Simulator Forum such as:

- SimConnect + WASM combined project using VS2019
- Tool to connect Serial USB devices to MSFS2020 via SimConnect/WASM using VS2022
- My A320 FCU and MCDU finished

Thanks to all the people that have contributed in these posts by giving advice, good ideas and several insights, I could build this new application.

Special thanks also to Maxim (Max) Paperno, who is the creator of WASimCommander. Max has contributed a lot by answering my numerous emails to understand the internals of his module. Without his patience and efforts (and extremely long and detailed emails), I couldn't have done this.

CockpitHardwareHUB_v2 is fully built in Visual Studio 2022.

Copyright, License and Disclaimer

___