
Sil

Release 6.14/1.0

Embention Sistemas Inteligentes, S.A.

2026-04-29

Contents

Veronte Console	4
Veronte S-Function	4
Download	6
Requirements	7
dll_config.vcfg file.....	9
Step by step - Veronte S-Function	11
Step by step - Veronte Console	16
Inputs and outputs	21
Inputs.....	21
Outputs	23
Sensors.....	25
Environment.....	26
Pressure sensors	28
Static Pressure	28
Dynamic Pressure	30
IMU	31
Magnetometer.....	38
GNSS	39
ADC	43
Serial Communications	43
Telemetry.....	44
Simulation	46
dll_config.vcfg file not working	50
Connection	51
S-Function connection failed	51
Simulink IP adress	51
Logs.....	52
License ID warning in Veronte Ops	52
Loaded with errors in Veronte Link	52
Running Veronte Console.....	53

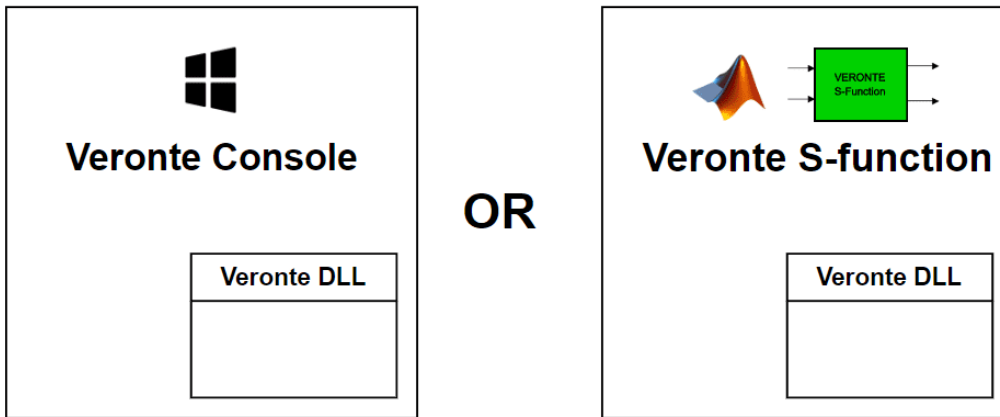
Introduction

SIL Simulator software acts as a '**virtual Autopilot 1x**', providing a realistic simulation experience for testing and development purposes.



The essence of SIL Simulator lies in its capacity to replicate **Autopilot 1x** behavior through the utilization of a dynamic-link library (DLL) embedded with **Veronte Autopilot 1x** code, **Veronte DLL**.

This code can simulate the physical autopilot firmware through two primary interfaces, **Veronte Console** or **Simulink workspace** (by means of the **Veronte S-Function** block). To suit customer preferences, Veronte DLL could also be run with other languages, such as Python. However, this integration must be undertaken independently by clients.



SIL Simulator interfaces

! Error

Running Veronte DLL with Veronte Console and Simulink workspace **simultaneously** will interfere with the functioning of the system, causing **the simulation not to work**.

Veronte Console

Veronte Console is a **Windows executable** that allows to simulate the 'virtual Autopilot 1x' (VeronteDLL).

Through this simulation option, users can employ Veronte applications with the simulated autopilot. However, at present, the main difference with **Veronte S-Function** is that inputs cannot be emulated in this simulation option.

Veronte S-Function

Veronte S-Function consists of a Simulink block which can be integrated in a **Simulink model** to virtually simulate the **behavior of Veronte Autopilot 1x in a customized environment**.

In the workspace of Simulink, users can design the dynamic model of their own aircraft, the desired input signals of the system, and therefore, analyse the response of the provided virtual autopilot in a emulated environment.

Note

Signal conditioning and calculation depend on the aircraft to simulate and the purpose of the simulation. Users must program their own Simulink workspace accordingly to these considerations.

SIL has several advantages when compared to a [HIL Simulator](#) setup:

- Complete simulations without any hardware.
- Possibility to use the user's vehicle model: users can define the dynamics of their vehicle (with the desired complexity) without the need to use external programs, such as Plane Maker.
- Possibility to simulate different types of sensors even if they are not installed in **Veronte Autopilot 1x**. All that is needed is the raw sensor reading.
- All results can be exported/visualized to MATLAB workspace simultaneously.
- **Veronte Autopilot 1x** blocks run faster than real-time, allowing the user to execute a series of simulations in a short time. This feature depends on the complexity of the model and the capability of the computer where the simulation is running.

Quick Start

This section sums up the **basic requirements** to start using **SIL Simulator**, both with Veronte Console and with Simulink blocks.

Note

For further details about SIL Simulator configuration, please proceed to the subsequent section [Configuration](#).

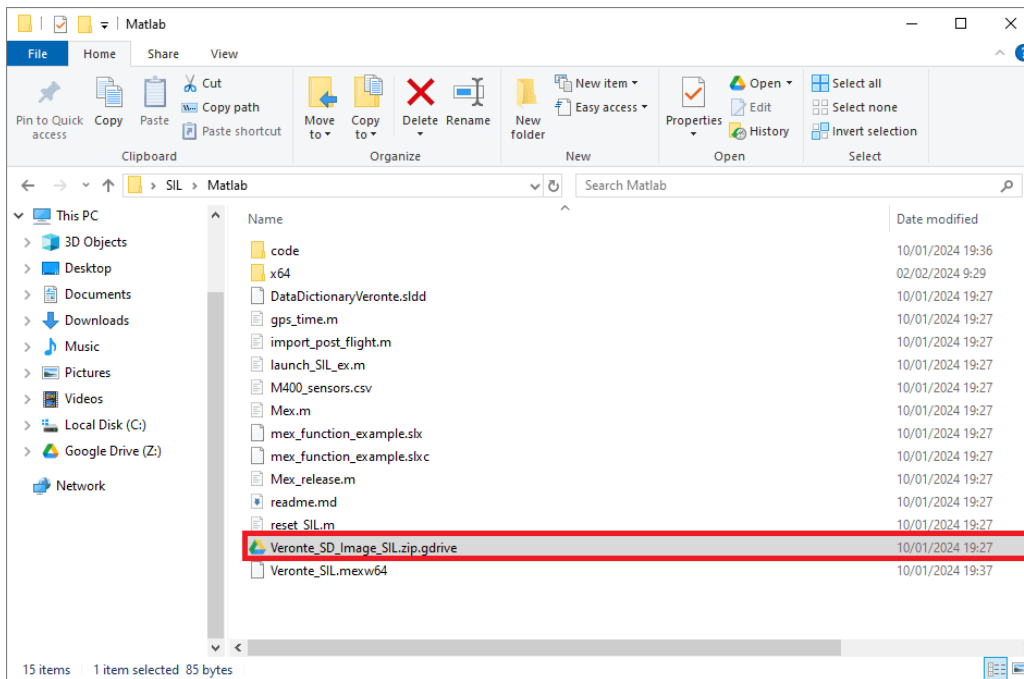
Download

Once **SIL package** has been purchased, a GitHub release should be created for the customer with the application.

- Download the **SIL zip** file from its corresponding release and decompress it in the desired location.
- Download **Veronte Autopilot 1x SD image downloadable** from the Drive folder linked in the `Veronte_SD_Image_SIL.zip.gdrive` file and decompress it in the desired location.

Important

- This image can also be physically sent to the user on an SD card.
- **Veronte Autopilot 1x SD image** occupies more than **100 GB** of disk space.



SIL folder - Drive file

Note

For further information about how to access to the release and download the software, read the [Releases](#) section of the **Joint Collaboration Framework** manual.

Requirements

- **System Requirements** Before executing this software, users should check the following sections with the minimum and recommended PC hardware requirements.
 - **Minimum requirements**
 - CPU: Intel Core i5-8365UE
 - RAM: 8 GB DDR4
 - STO: 256 GB SSD
 - **Recommended requirements**
 - CPU: 12th Gen Intel(R) Core(TM) i7-12700H 14 cores up to 4,70 GHz
 - RAM: 32,0 GB
 - STO: 1TB SSD M.2 NVMe PCIe

-
- **Veronte Software Package:**
 - **Veronte Link** (latest version available): Used to connect Autopilot 1x to the other tools.
 - **1x PDI Builder** (v6.12.X): To build and load PDIs.
 - **Veronte Ops** (v6.12): Operations interface.
 - **SIL with Simulink:** To perform a SIL simulation using Simulink with the Veronte Autopilot 1x, the following programs and toolboxes are required in addition to the requirements described above:
 - **MATLAB + Simulink** (basic package).
 - The user can be helped by other simulink toolboxes when implementing their model:
 - **Simulink Real-Time:** This blockset contains useful blocks to be used with buses: UDP/RS232/CAN.
 - **Aerospace toolbox:** Contains sensor blocks, flight instruments and environment blocks.
 - **Microsoft Visual Studio 2015** (or later) as your MEX compiler. Despite .mex file is already compiled and it works as a black box, some libraries are necessary.
 1. First, get Microsoft Visual Studio from [here](#).
 2. Follow the onscreen steps, please make sure that C++ tools are selected (they may appear as an optional item).
 3. When finished, select it as your default MEX compiler by typing in MATLAB console `mex -setup c++`.

Configuration

This section details a step by step explanation of how to configure **SIL Simulator**.

dll_config.vcfg file

This file sets **the simulation configuration**, and it is common for both simulation options: **Veronte Console** and **Veronte S-Function for Simulink**.

`dll_config.vcfg` file indicates the location of the **Veronte DLL** and **Veronte image** and establishes some parameters of the simulation, as the **hardware version** to simulate or the **ID of the virtual autopilot**.

This is an example of `dll_config`:

```
# Path to DLL
\dll .\x64\VeronteDLL.dll

# Path to image file (absolute or relative to dll)
\image .\..\Veronte_SD_Image.img

# Hardware version
\hwversion 4.8

# Autopilot ID
\idversion 2
```

Before configuring **SIL Simulator**, these are aspects of the `dll_config.vcfg` file to consider for a proper functioning of the simulation:

- **Location of `dll_config.vcfg` file:** It must be placed in the same path as **the executable code** to run. Depending on the simulation option to use, it must be placed in:
 - `VeronteConsole.exe` path if using **Veronte Console**
 - `.slx` path if using **Veronte S-Function for Simulink**

Note

.slx files store Simulink model information in a reduced size. As previously explained, users must program their own Simulink workspace according to their aircraft and simulation goal, but an example is provided in the **SIL folder** (`mex_function_example.slx`).

- **DLL and image files paths:** It is crucial to precisely indicate the paths of DLL and image files.

Hardware version parameter

```
\hwversion version
```

Users can decide which hardware version to simulate:

4.0	hwversion 4.0
4.5	hwversion 4.5
4.8	hwversion 4.8

ID version parameter

```
\idversion index
```

Available **Autopilot IDs** vary depending on the selected hardware version. The index to be entered will be indicated by the following table:

idversion	0	1	2	3	4	5
hw v4.0	1008	1025	1128	1373	1559	1654
hw v4.5	1805	1862	1871	2375	2680	2821

idversion	0	1	2	3	4	5
hw v4.8	4041	4064	4065	4144	4146	4213

Step by step - Veronte S-Function

1. Once decompressed, open the **SIL folder**.



SIL folder

1. Ensure that the `dll_config.vcfg` file is in the same path as the `.slx` file.

Note

`.slx` files store Simulink model information in a reduced size. As previously explained, users must program their own Simulink workspace according to their aircraft and simulation goal, but an example is provided in the **SIL folder** (`mex_function_example.slx`).

2. Configure the `dll_config.vcfg` file in the following cases:
 - If the hardware version to simulate differs from v4.0.
 - If the DLL and image files are not located in the paths specified in `dll_config.vcfg`.

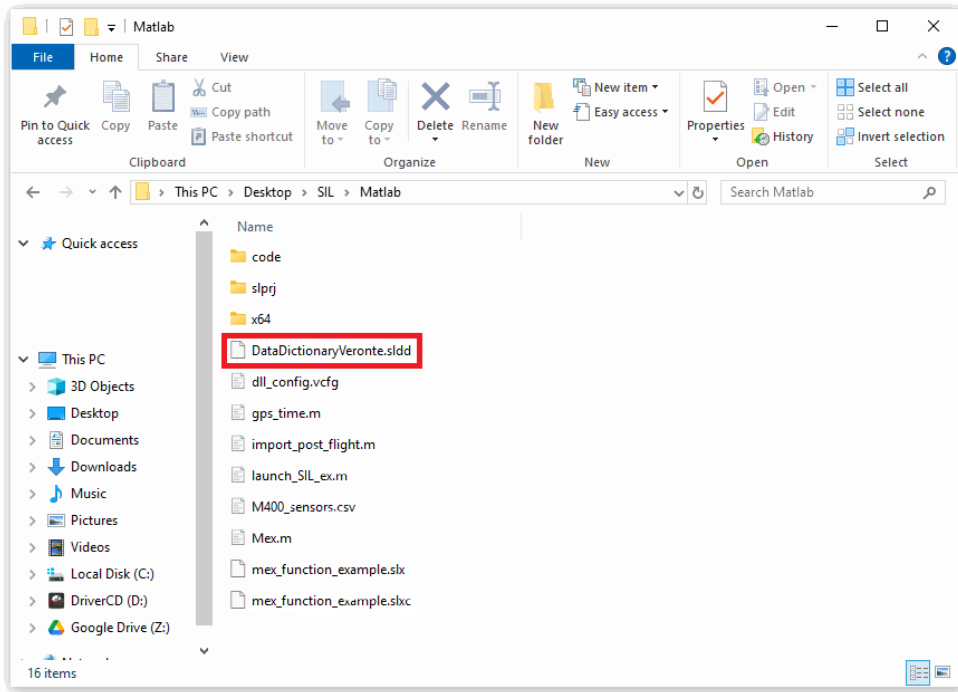
Parameters to configure:

- Path to DLL file `VeronteDLL.dll` (**absolute** or **relative** to the `.slx` file)
- Path to image file `Veronte_SD_Image.img` (**absolute** or **relative** to DLL file)
- (Optional) Hardware version to simulate
- (Optional and dependent on hardware version) Autopilot ID to simulate

Warning

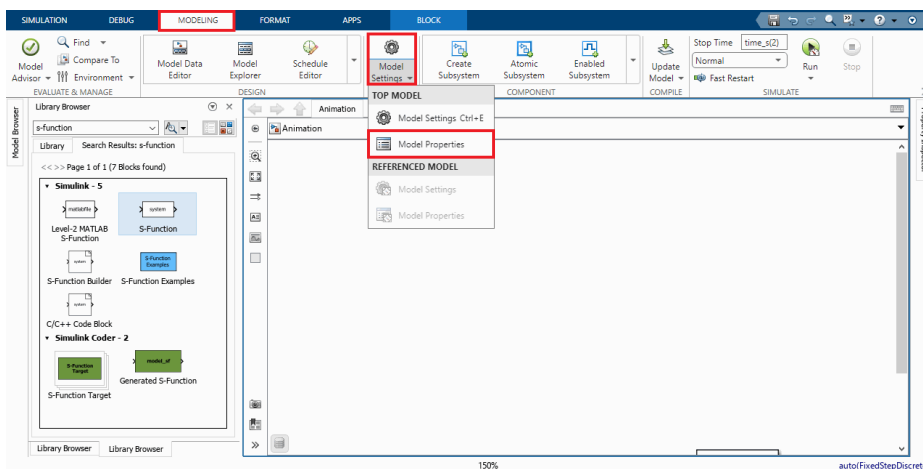
Autopilot ID parameter only can be chosen if hardware version is also specified. For more information about setting the ID parameter, consult [dll_config.vcfg](#) file section of the present manual.

3. Load the file `DataDictionaryVeronte.sldd`, located in the SIL folder, into the Simulink configuration.



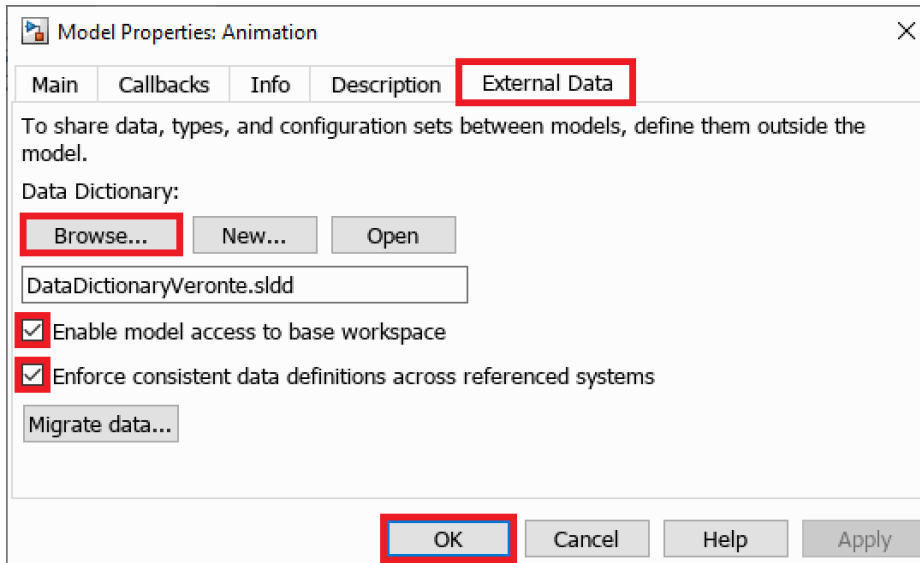
SIL folder

- Open **Simulink**, go to the **Modeling** tab in the top menu and select **Model Properties**



Model properties option

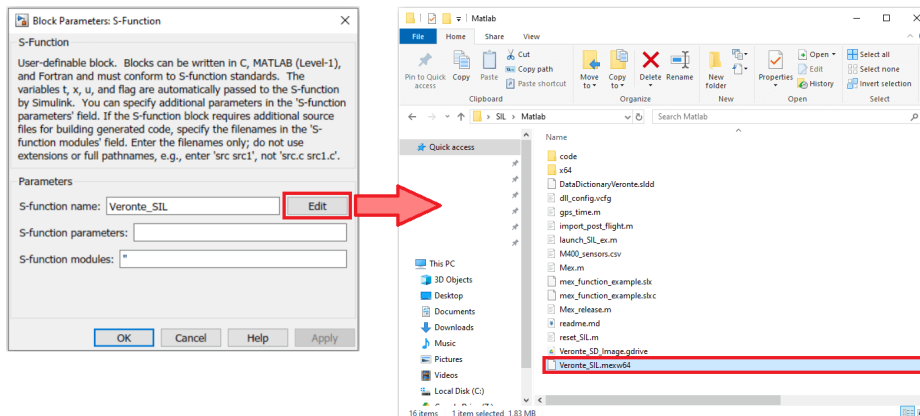
- In the pop-up window, go to the **External Data** tab. Click **Browse...** and select `DataDictionaryVeronte.sldd` file located in the SIL folder. Check the two available boxes.



External Data option

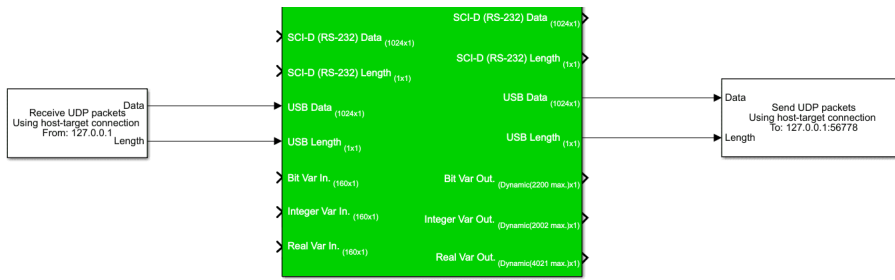
4. Open **Simulink** and configure the blocks explained below:

- Add a S-Function block, and point to `Veronte_SIL.mexw64` code by editing the S-Function name:



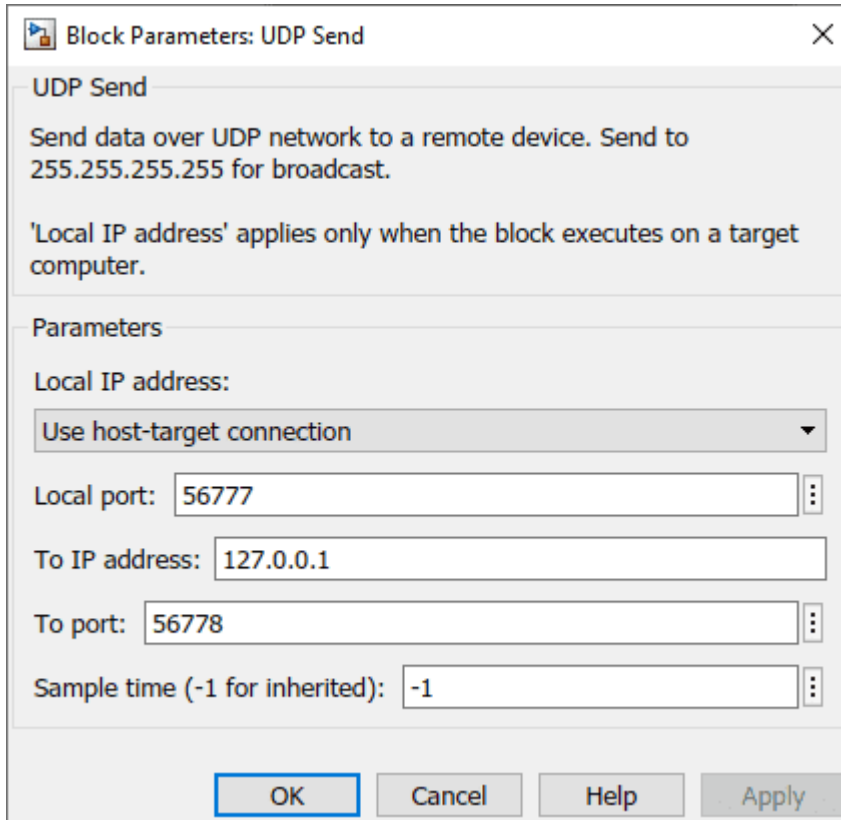
S-Function block parameters

- Add a UDP serial communication block and connect it to USB data and length inputs of Veronte S-Function.
- Add a second UDP serial communication block and connect it to the USB outputs of Veronte S-Function.



UDP Blocks

- Configure the desired destination port.



Destination UDP Port

- For extra information about **Simulink configuration**, consult the [Simulink workspace](#) section of this manual.

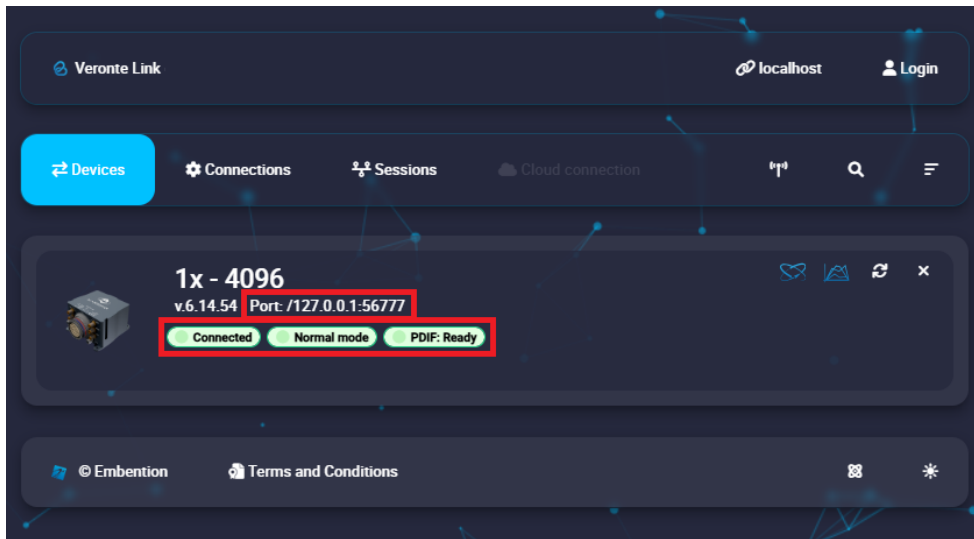
5. In **Veronte Link** application, configure a UDP connection with the following parameters:

Note

For more information about configuring connections, please consult [Connection](#) section of the **Veronte Link** user manual.

- **Type of connection:** UDP
- **IP:** IP previously configured in Simulink workspace

- **Port:** Port previously configured in Simulink workspace
6. Run the simulation by running the **Simulink model**.
 7. Check that the Autopilot 1x appears in Veronte Link as **Connected** and **Ready**:



Veronte Link

Note

The autopilot may appear as **Loaded with errors** in Veronte Link for reasons not related to the simulation configuration. Refer to [Loaded with errors in Veronte Link - Troubleshooting](#) section of this manual.

8. Once the simulation starts, user can proceed as with a physical autopilot:
 - [1x PDI Builder](#) for configuration.
 - [Veronte Ops](#) for operation and mission.
 - [HIL Simulator](#) for simulating the virtual autopilot with external simulators.
9. Important events or messages occurring within the simulation are registered on `SIL.log`.

Note

For more information or problems, please consult [Connection - Troubleshooting](#) section of this manual.

Step by step - Veronte Console

1. Once decompressed, open the **SIL folder**.



SIL folder

2. Ensure that the `dll_config.vcfg` file is in the same path as `VeronteConsole.exe`.
3. Configure the `dll_config.vcfg` file in the following cases:
 - If the hardware version to simulate differs from v4.0.
 - If the DLL and image files are not located in the paths specified in `dll_config.vcfg`.

Parameters to configure:

- Path to DLL file `VeronteDLL.dll` (absolute or relative to `VeronteConsole.exe`)
- Path to image file `Veronte_SD_Image.img` (absolute or relative to DLL file)
- (Optional) Hardware version to simulate
- (Optional and dependent on hardware version) Autopilot ID to simulate

Warning

Autopilot ID parameter only can be chosen if hardware version is also specified. For more information about setting the ID parameter, consult [dll_config.vcfg file](#) section of the present manual.

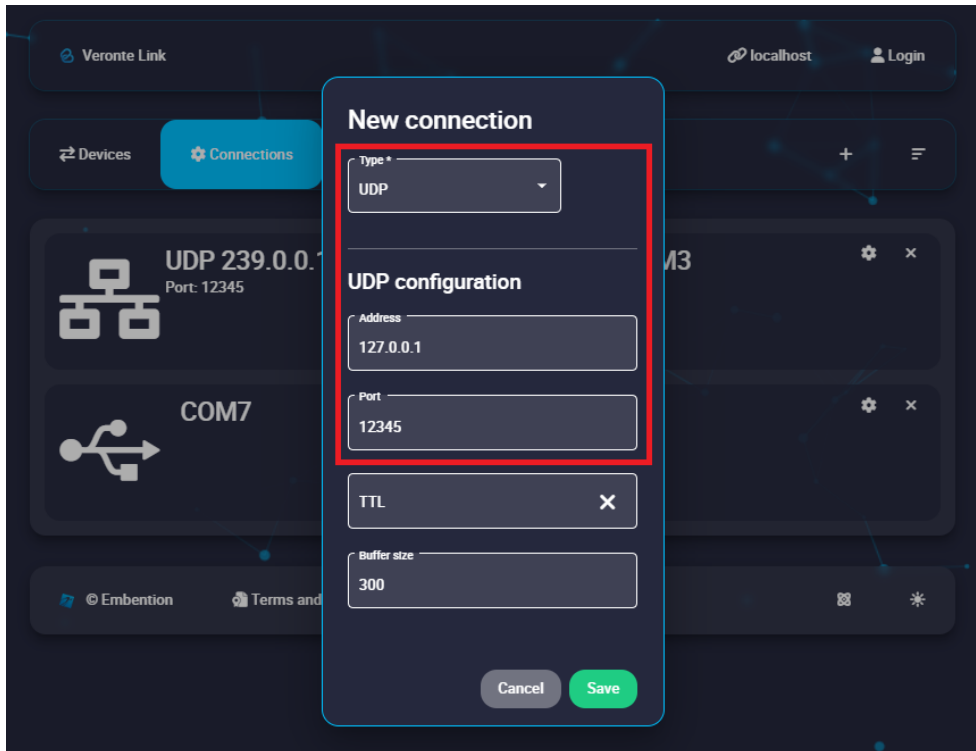
4. In **Veronte Link** application, configure a UDP connection with the following parameters:

Note

For more information about configuring connections, please consult [Connection](#) section of the **Veronte Link** user manual.

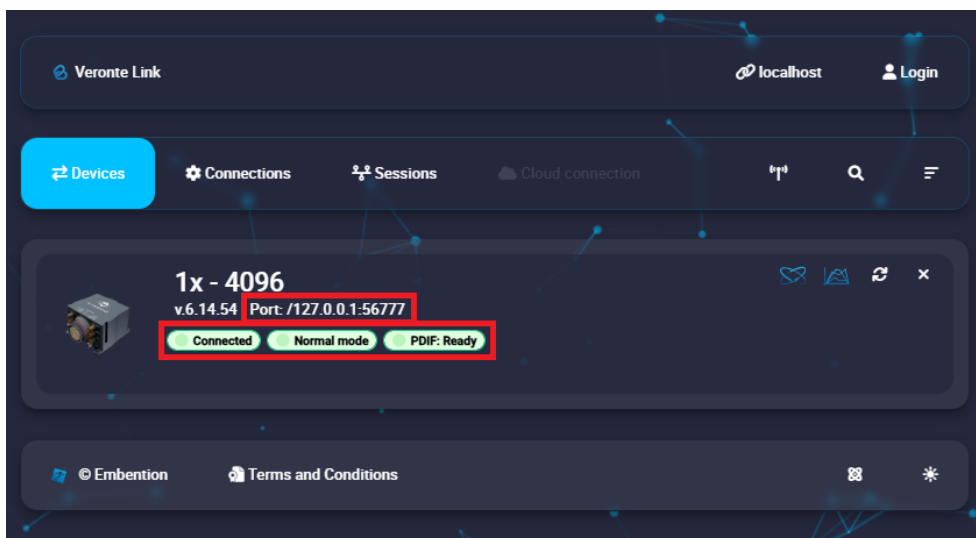
- **Type of connection:** UDP

- **IP:** 127.0.0.1
- **Port:** 12345



Veronte Link - UDP Connection for Veronte Console

5. Run the simulation by executing **Veronte Console.exe**.
6. Check that the Autopilot 1x appears in Veronte Link as **Connected** and **Ready**:



Veronte Link

Note

The autopilot may appear as **Loaded with errors** in Veronte Link for reasons not related to the simulation configuration. Refer to [Loaded with errors in Veronte Link - Troubleshooting](#) section of this manual.

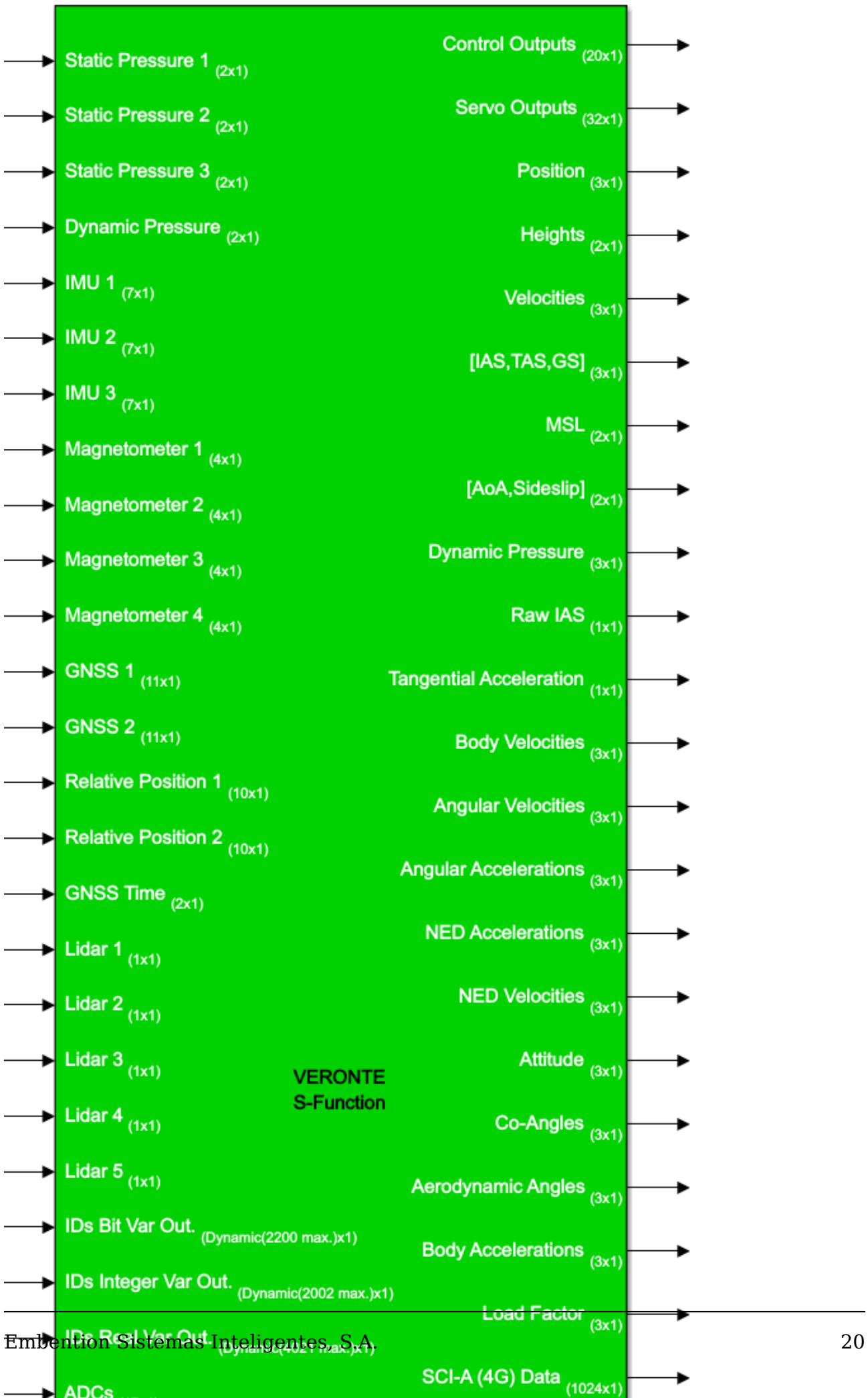
7. Once the simulation starts, user can proceed as with a physical autopilot:
 - [1x PDI Builder](#) for configuration.
 - [Veronte Ops](#) for operation and mission.
 - [HIL Simulator](#) for simulating the virtual autopilot with external simulators.

Simulink workspace

SIL Simulator can be run with Simulink software, by means of the **S-Function** block.

This kind of block takes a C, C++, Fortran or even Matlab code, and implements it in a block containing a certain number of inputs and outputs. As explained in [Step by step steps](#) section, in order to simulate Autopilot 1x the **S-Function** block must be configured to point to Veronte code (`Veronte_SIL.mexw64`).

In the following subsections, **Veronte S-Function** parameters are listed and general aspects of **Simulink** workspace are explained.



S-Function containing the Veronte Autopilot 1x embedded code

Inputs and outputs

Inputs

Inputs are described in the next table:

PIN	Description	Form
1	Static Pressure 1	[pressure_measurement;sensor temperature]
2	Static Pressure 2	[pressure_measurement;sensor temperature]
3	Static Pressure 3	[pressure_measurement;sensor temperature]
4	Dynamic Pressure	[pressure_measurement;sensor temperature]
5	IMU 1	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]
6	IMU 2	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]
7	IMU 3	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]
8	Magnetometer 1	[mag_x;mag_y;mag_z;sensor temperature]
9	Magnetometer 2	[mag_x;mag_y;mag_z;sensor temperature]
10	Magnetometer 3	[mag_x;mag_y;mag_z;sensor temperature]
11	Magnetometer 4	[mag_x;mag_y;mag_z;sensor temperature]
12	GNSS 1	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]
13	GNSS 2	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]
14	Relative Position 1	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]
15	Relative Position 2	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]
16	GNSS Time	[week_number;milliseconds_of_week]
17	Lidar 1	[lidar_measurement]

18	Lidar 2	[lidar_measurement]
19	Lidar 3	[lidar_measurement]
20	Lidar 4	[lidar_measurement]
21	Lidar 5	[lidar_measurement]
22	IDs Bit Var Out.	[Var_IDs]
23	IDs Unsigned Var Out.	[Var_IDs]
24	IDs Real Var Out.	[Var_IDs]
25	ADCs	[adc(1-17)]
26	SCI-A (4G) Data	[serial_data]
27	SCI-A (4G) Length	[serial_length]
28	SCI-B (Radio) Data	[serial_data]
29	SCI-B (Radio) Length	[serial_length]
30	SCI-C (RS-485) Data	[serial_data]
31	SCI-C (RS-485) Length	[serial_length]
32	SCI-D (RS-232) Data	[serial_data]
33	SCI-D (RS-232) Length	[serial_length]
34	USB Data	[serial_data]
35	USB Length	[serial_length]
36	Bit Var In.	[Var0_ID;Var0_value;...;Var80_ID;Var80_value]
37	Unsigned Var In.	[Var0_ID;Var0_value;...;Var80_ID;Var80_value]
38	Real Var In.	[Var0_ID;Var0_value;...;Var80_ID;Var80_value]

Note

In the table above, the size of the inputs "IDs Bit/Unsigned/Real Var Out." (pins 22, 23 and 24 respectively) have been described as **Dynamic** because they don't need to have a fixed size. However, the size has to be continuous throughout the simulation.

Outputs

Outputs are the following:

PIN	Description	Form	S
1	Control Outputs	[control_outputs(1-20)]	2
2	Servo Outputs	[servos(1-32)]	3
3	Position	[lon;lat;alt]	
4	Heights	[msl,agl]	
5	Velocities	[longitudinal_v;lateral_v;velocity(module)]	
6	IAS, TAS, GS	[ias,tas,gs]	
7	MSL	[msl_from_qnh;msl_from_ISA]	
8	Angle of Attack, Sideslip	[angle_of_attack;sideslip]	
9	Dynamic Pressure	[dynamic_pressure]	
10	Raw IAS	[ias_raw]	
11	Tangential Acceleration	[tangential_acceleration]	
12	Body Velocities	[longitudinal_v;lateral_v;vertical_v]	
13	Angular Velocities	[roll_rate;pitch_rate;yaw_rate]	
14	Angular Acceleration	[acc_z_axis;acc_y_axis;acc_x_axis]	

PIN	Description	Form	S
15	NED Acceleration	[acc_north;acc_east;acc_down]	
16	NED Velocities	[v_north;v_east;v_down]	
17	Attitude	[Yaw;Pitch;Roll]	
18	Co-Angles	[co-Yaw;co-Pitch;co-Roll]	
19	Aerodynamic Angles	[heading,flight_path;bank_angle]	
20	Body Accelerations	[acc_x,acc_y;acc_z]	
21	Load factor	[nx;ny;nz]	
22	SCI-A (4G) Data	[serial_data]	10
23	SCI-A (4G) Length	[serial_length]	
24	SCI-B (Radio) Data	[serial_data]	10
25	SCI-B (Radio) Length	[serial_length]	
26	SCI-C (RS-485) Data	[serial_data]	10
27	SCI-C (RS-485) Length	[serial_length]	
28	SCI-D (RS-232) Data	[serial_data]	10
29	SCI-D (RS-232) Length	[serial_length]	
30	USB Data	[serial_data]	10
31	USB Length	[serial_length]	
32	Bit Var Out.	[Var_values]	Dynamic(2)
33	Unsigned Var Out.	[Var_values]	Dynamic(2)
34	Real Var Out.	[Var_values]	Dynamic(4)

Note

The outputs "Bit/Unsigned/Real Var Out." (pins 32, 33 and 34 respectively) correspond to the inputs "IDs Bit/Unsigned/Real Var Out." (pins 22, 23 and 24 respectively), so they will have the same size as defined in the inputs.

In the following sections, the user can have a look at how to implement the [sensor](#) and [telemetry](#) blocks, as well as general visualisation of a [complete simulation](#).

Sensors

Sensors measurements are the inputs of the mex block (embedded code).

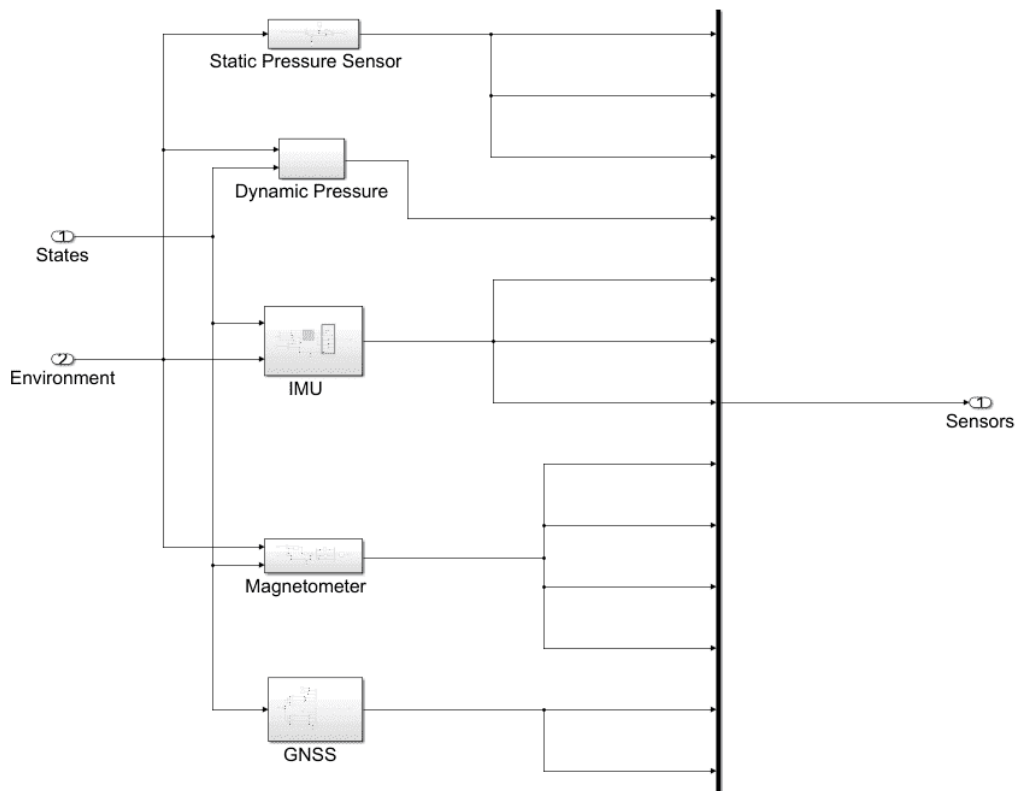
To perform a correct simulation, the user has to configure the inputs with the same scheme as Veronte Autopilot 1x reads them. Each sensor has a certain vector/array which usually includes raw data in one or more coordinates, sensor temperatures, variances or squared errors.

Warning

Users cannot set constant values for these variables as this may be interpreted by Veronte Autopilot 1x as sensor failure.

For this reason, if the simulated signal is constant, it is recommended to add some **white noise** to it.

This section aims to illustrate how to implement the inputs described in the previous section. **The structures shown here are indicative** and can of course be adapted by the user:



Sensors inputs

Next, the user will find some examples of how to implement the following sensors:

- [Environment](#)
- [Pressure sensors](#)
- [IMU](#)
- [Magnetometer](#)
- [GNSS](#)
- [ADC](#)
- [Serial Communications](#)

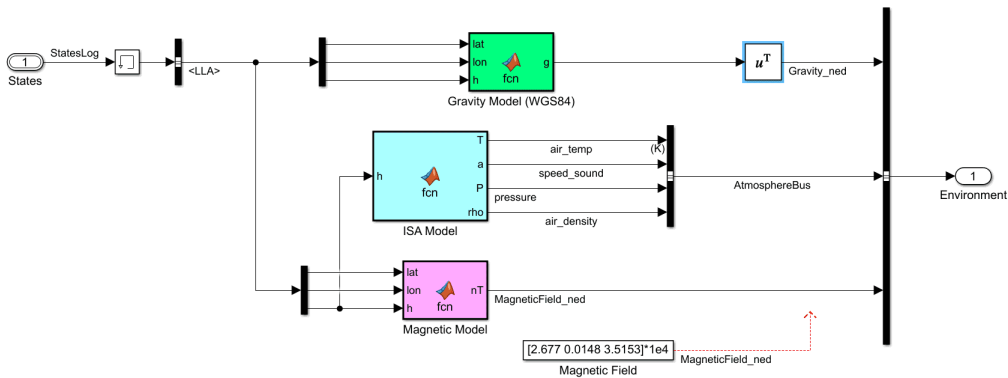
Environment

To simulate a model correctly, it is necessary to take into account that the environmental variables change depending on the position of the UAV. The user can choose between using a simple and constant model or modifying at each step the environmental variables according to a complex model.

This model should group the atmospheric properties (temperature, pressure, etc.) which change with altitude (an offset can also be added), the gravity vector, as well as the magnetic field which changes according the UAV

coordinates. All this information can be used for a better characterisation of the sensor measurements.

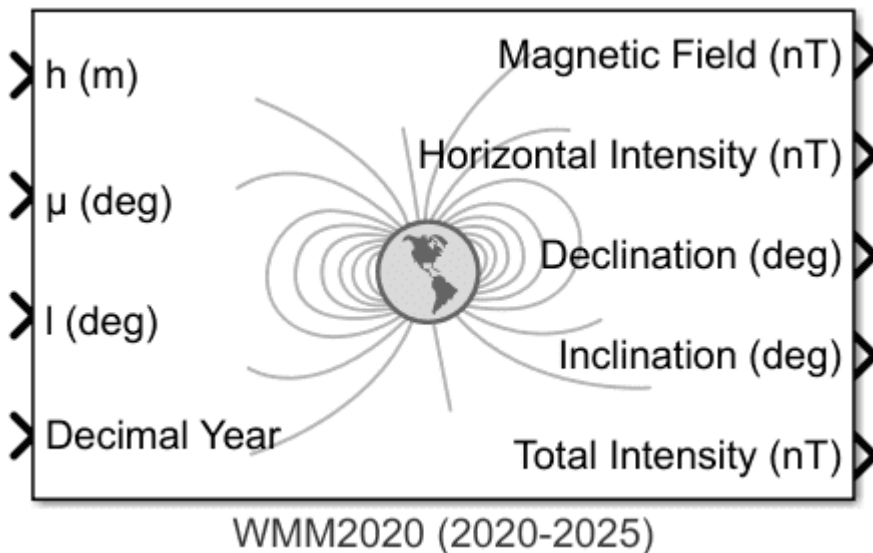
A basic example is shown below. It is divided into 3 different models (ISA atmosphere model, WGS84 model for gravity vector, and the World Magnetic Model). Each model is included in a user Matlab function whose arguments are the inputs of the block.



Environment block

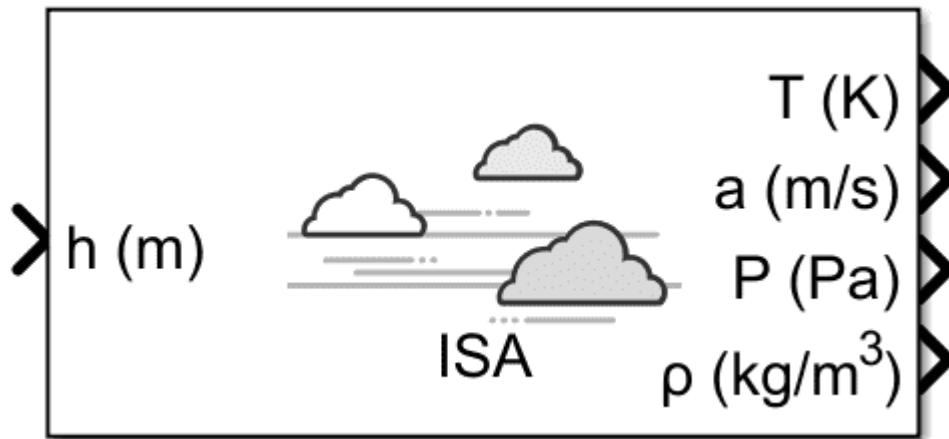
Instead of creating their own functions, users can use those included in the **Aerospace Toolbox**:

1. World Magnetic Model 2020:



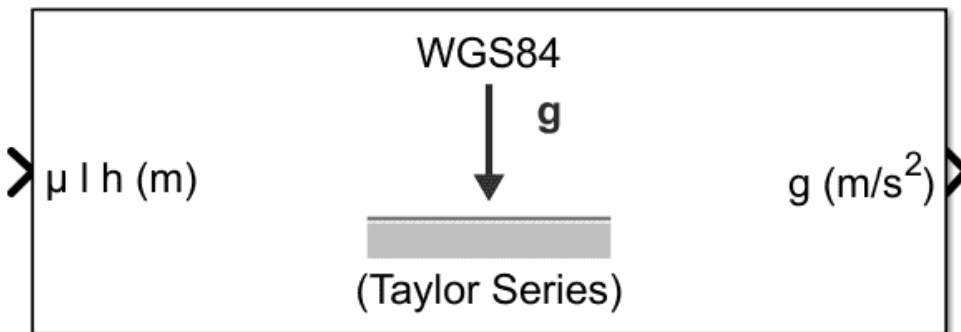
Aerospace blockset function - WMM2020

2. ISA Atmosphere Model:



Aerospace blockset function - ISA Atmosphere Model

3. WGS84 Gravity Model:



Aerospace blockset function - WGS84 Gravity Model

Pressure sensors

Static Pressure

Static Pressure inputs in the S-Function simulate the internal ones in Veronte Autopilot 1x. The required information consists of **raw measurements and the sensor device temperature**.

The S-Function has **3 ports** which correspond to the sensors that would be installed in **Autopilot 1x**. The port (or ports) receiving the raw static pressure should be that which matches the static pressure sensor selected in the simulated autopilot's PDI configuration (modified through the **1x PDI Builder** software).

In the following table, the user can consult the static pressure sensors available for each **hardware version**:

Hardware version	S-Function Input		1x PDI Builder equivalence
	PIN	Description	
4.0	1	Static Pressure 1	0 - Internal 0
	2	Static Pressure 2	1 - Internal 1
4.5	1	Static Pressure 1	0 - Internal 0
	2	Static Pressure 2	1 - Internal 1
	3	Static Pressure 3	2 - Internal 2
4.8	2	Static Pressure 2	1 - Internal 1
	3	Static Pressure 3	2 - Internal 2

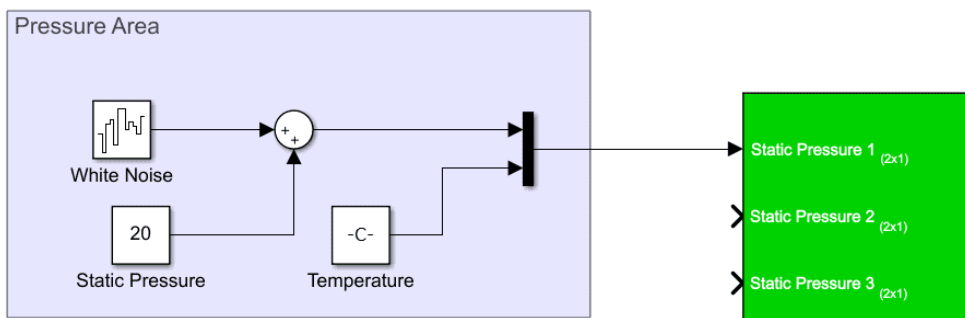
Note

There may be inputs (ports) not connected since the S-Function contains the maximum number of possible inputs for the different hardware versions, as can be seen in the aforementioned table.

Below are some examples of how to implement the static pressure:

• Constant value

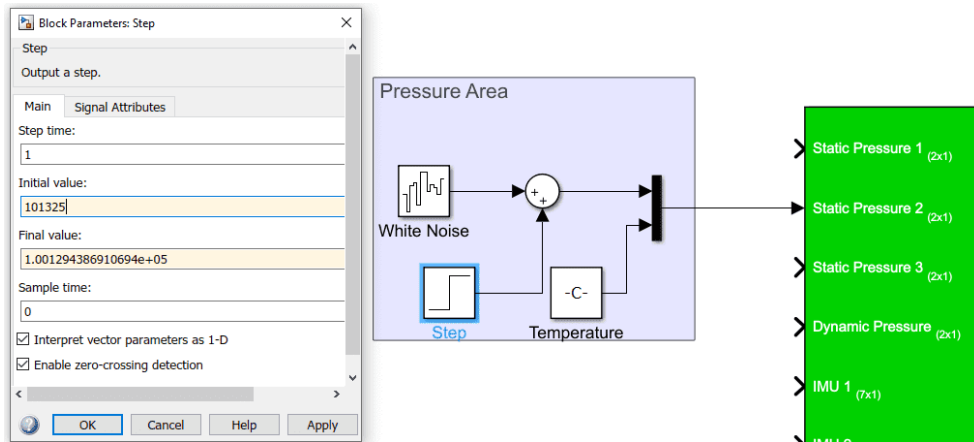
Only one block constant for raw pressure and one for temperature.



Static pressure - Constant

• Step

If users wish to simulate a leap in pressure measurements, it is possible to add a step to the previous configuration. In the example below a difference in 100 meters is represented.



Static pressure - Step input

Important

Note that in both examples a **White Noise** block has been added.

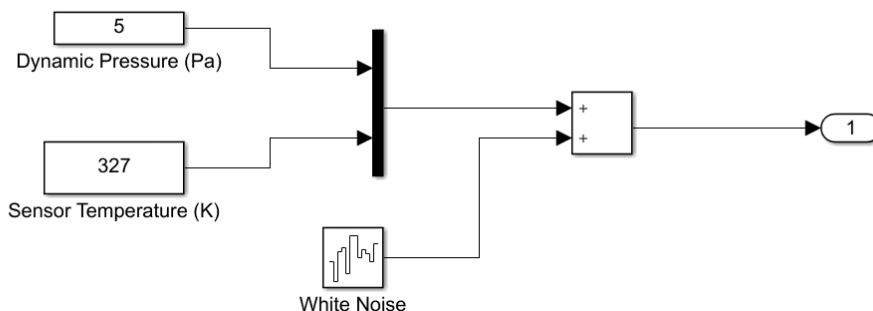
Dynamic Pressure

The dynamic or velocity pressure input **requires the raw measurement of dynamic pressure and sensor device temperature.**

Some examples of how to implement dynamic pressure are shown below:

- **Constant value**

As the raw measurement of dynamic pressure has been added as a constant, a **white noise block** is also added:

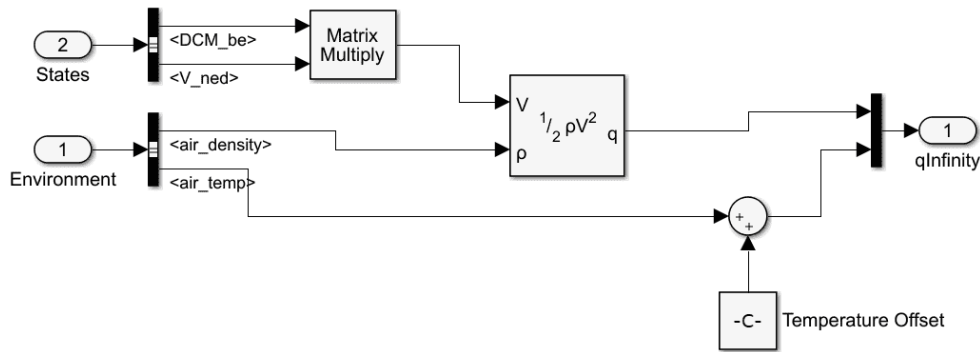


Dynamic pressure - Constant

- **Complex model**

In this example, Autopilot 1x is assumed to be mounted on the X-axis in Body rame. Therefore, from the velocity in NED frames, a rotation is applied to obtain the velocity in Body frames, and then the first component of the vector is taken.

In addition, the density value is taken from the environment model.



Dynamic Pressure - Subsystem

IMU

IMU measures and informs about velocity, attitude and forces by combining the accelerometer and gyroscope readings.

Veronte Autopilot 1x needs to receive **7 measurements: 3-axis accelerometer, 3-axis gyroscope** and **sensor device temperature**.

In the S-Function there are **3 inputs for IMUs**. These IMUs are mounted differently on the **Autopilot 1x** (they may not be aligned with the autopilot), so the user has to keep in mind the rotation matrix that the **Autopilot 1x** is using.

⚠ Important

- The rotation matrices listed in the following table are the required rotations between each sensor and the **Autopilot 1x** board coordinates (for more information on **Veronte Autopilot 1x** coordinates, please check the [Orientation - Hardware Installation](#) section of the **1x Hardware Manual**).
- There may be inputs (ports) not connected since the S-Function contains the maximum number of possible inputs for the different hardware versions, as can be seen in the following table.
- As detailed in the table below, if users wish to enter **measurements for IMU 3** of hardware version **4.8**, they must enter them into the **IMU 1 input** of **SIL S-Function**. This is because the block input values belonging to IMU 1 also go internally to this IMU. Therefore, the same S-Function can be used for both version **4.0** and **4.8**.

Hardware version	S-Function Input		Rotation Mat
	PIN	Description	
4.0	5	IMU 1	$\left(R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$
	6	IMU 2	$\left(R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \right)$
4.5	5	IMU 1	$\left(R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$
	6	IMU 2	$\left(R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \right)$
	7	IMU 3	$\left(R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$
4.8	5	IMU 1	$\left(R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$
	6	IMU 2	$\left(R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$
	7	IMU 3	$\left(R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right)$

Within the S-Function, the introduced data is transformed to match the coordinates of **Veronte Autopilot 1x**. So, in order to be correctly transformed inside the S-Function, the data must have been previously "prepared" to be introduced into it in the following way:

$$data_{(input)} = (Sensor\ rotation\ matrix)^T \cdot data_{(board)}$$

In the example provided by Embention with the SIL package, the **simin_IMU0** block already has this transformation implemented, so it can be entered directly into the S-Function without prior "preparation".

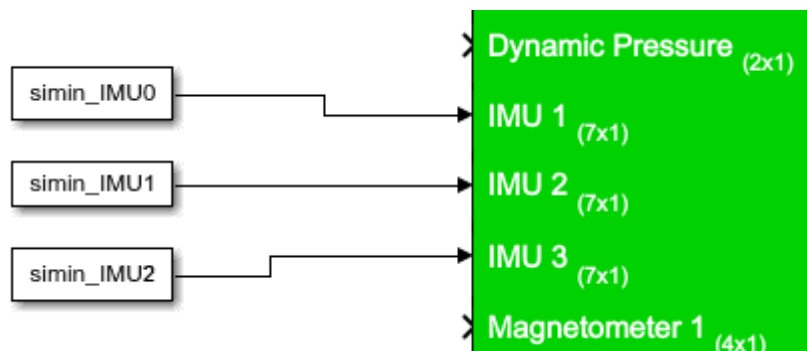
There are several ways to implement a suitable read group for an IMU:

- **Constant value**

The user can create a vector with constant values.

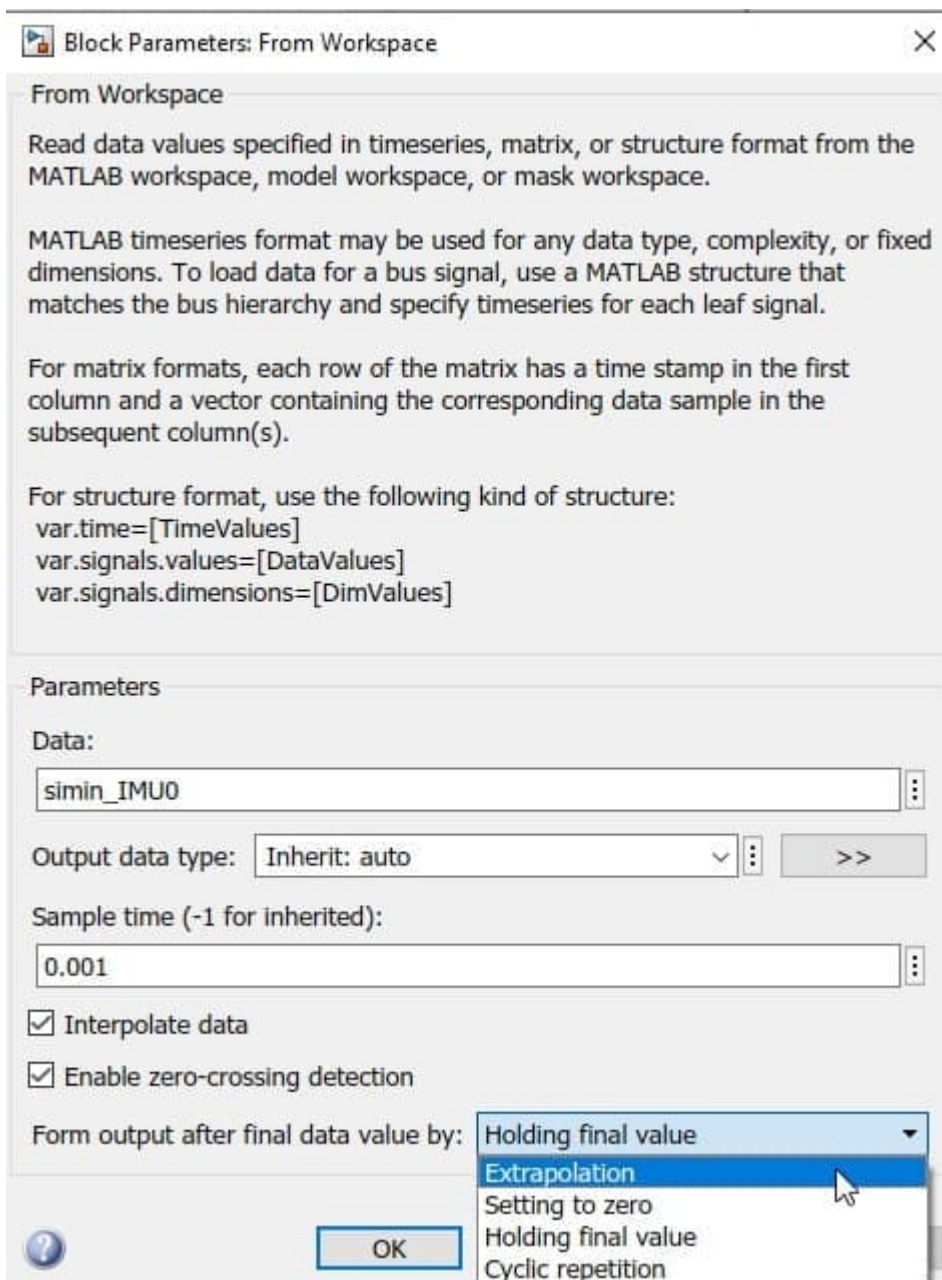
- **From Workspace block**

Another option could be to store some data (i.e. from a previous flight), load it into the matlab workspace, and then send these values to Simulink using the block name as **From Workspace**.



IMU - From workspace blocks

This block allows the user to read from an array of values (and interpolate when there is no information in this step). Moreover, users can choose between several options in case data vector is over. For example, it is possible to extrapolate the information or reset the list of values.



From workspace block configuration

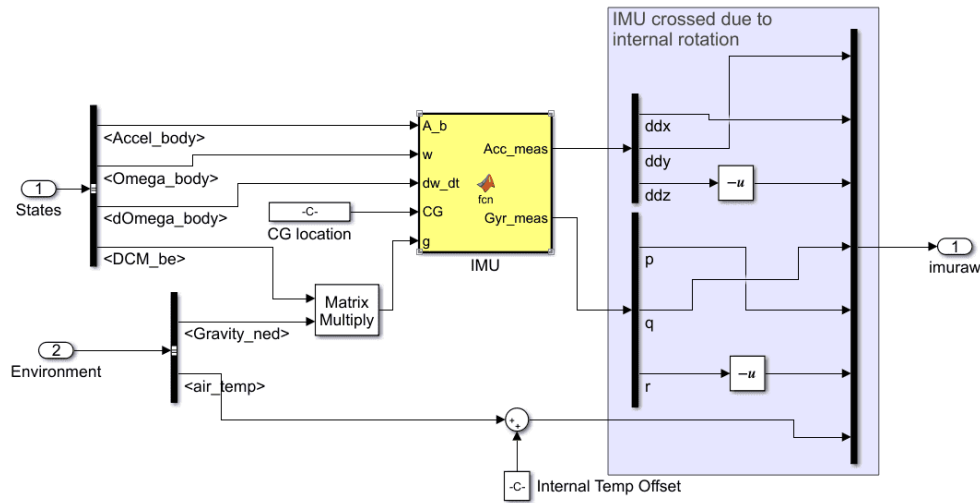
- **Complex model**

Another method is to read these values from **Environment** (gravity vector in NED frame and air temperature) and from **States** (acceleration in body axes, angular velocity, angular acceleration and the rotation matrix from NED to Body).

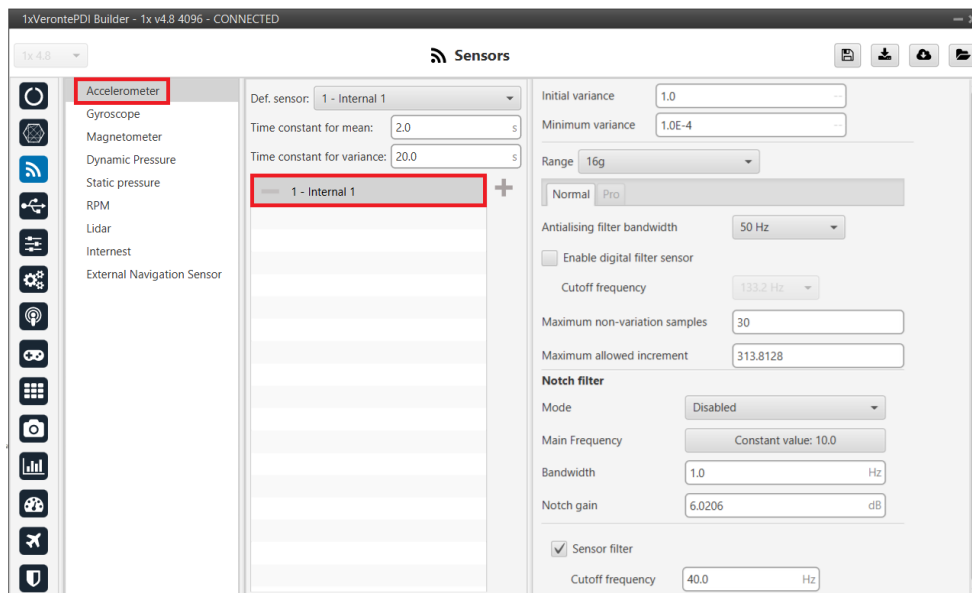
These values are fed into a Matlab function in which the IMU behavior is simulated and the measurements are computed. In addition, users have to cross-reference the measurements or apply a rotation matrix depending on the orientation of the IMU sensor.

In the example below, this data feeds the first port (this IMU is selected in the **1x PDI Builder** configuration). Therefore, the user has to cross the signals to adjust the rotation matrix.

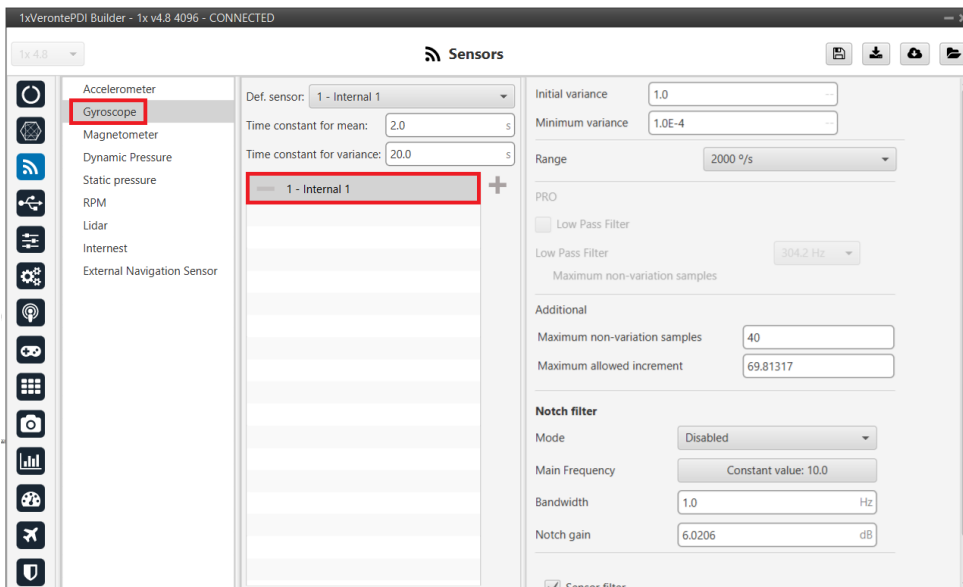
The complete subsystem results as follows:



IMU - Subsystem

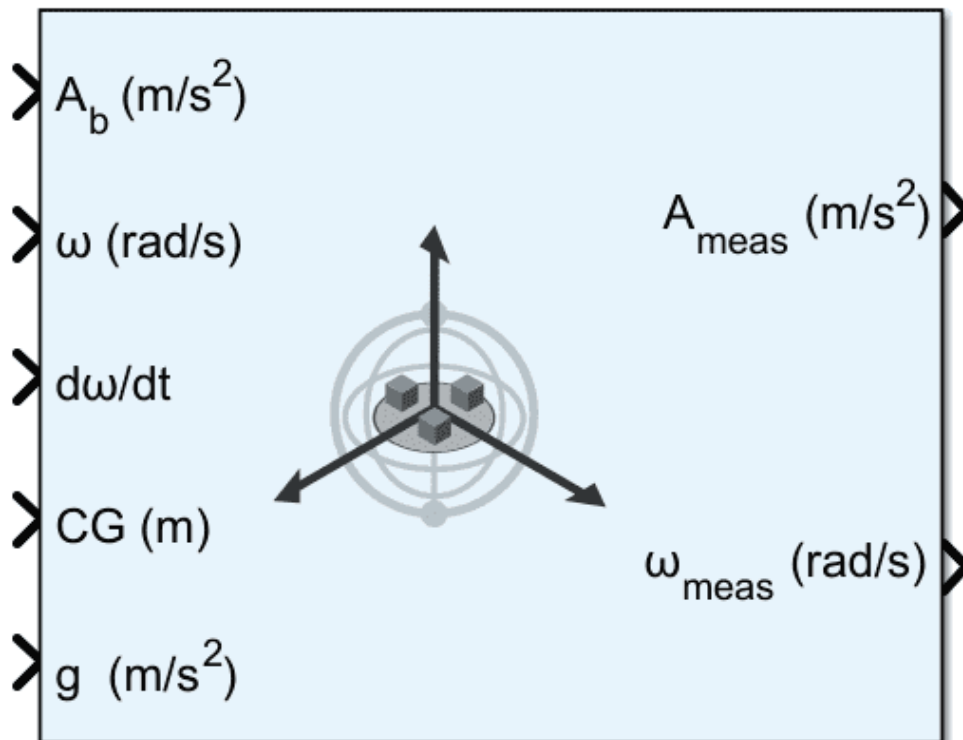


IMU - Accelerometer selected in 1x PDI Builder



IMU - Gyroscope selected in 1x PDI Builder

However, instead of using a user function, **Aerospace blockset** includes some functions for IMU simulation that can be employed:



IMU - Aerospace Toolbox block

- **Specific example from the example provided by Embention** The data of the provided example are already prepared to be introduced in the IMU 1 input of the S-Function, this is because it corresponds to the IMU 0 sensor in hardware version **4.0**.

Therefore, if users want to use the data provided by Embention (simin_IMU0) and in their PDI configuration they have selected the IMU 3 with hardware version 4.8, the following transformations are necessary for correct operation:

1. Transform from IMU 0 to board coordinates

Undo the "preparation for input" (this "preparation" has been described at the beginning of the section) to obtain the data provided by the sensor in body coordinates:

$$data_{(board)} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix} \cdot data_{(example)}$$

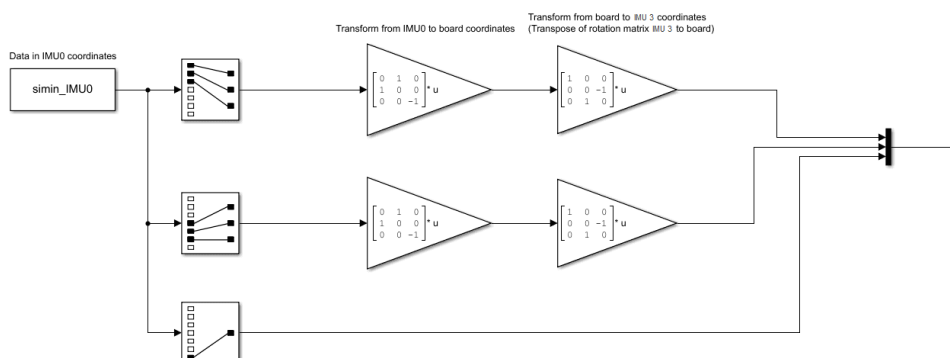
- $data_{(example)}$ is the data inside the **simin_IMU0** block

2. Transform from board to IMU 3 coordinates

Prepare the data for conversion to IMU 3 coordinates by multiplying the sensor body data by the IMU 3 rotation matrix transpose:

$$data_{(input)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \cdot data_{(board)}$$

Finally, the whole transformation should look like this:



IMU - Example

Magnetometer

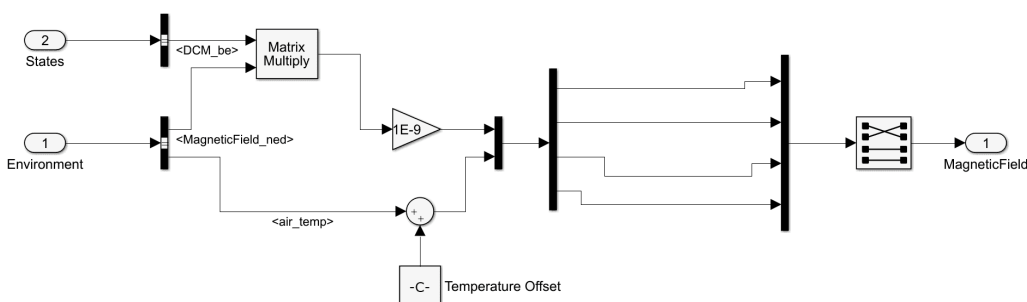
The magnetometer inputs expect to receive **magnetic field measurements in 3 axes**, as well as the **sensor device temperature**.

The S-Function has **4 ports** for magnetometer reading. In addition, as with IMUs, the user must take into account how the magnetometer is mounted (rotation matrix).

Hardware version	S-Function Input		Rotation Matrix
	PIN	Description	
4.0	8	Magnetometer 1	$\left(R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$
4.5	8	Magnetometer 1	$\left(R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$
	9	Magnetometer 2	$\left(R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$
4.8	8	Magnetometer 1	$\left(R = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$
	9	Magnetometer 2	$\left(R = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$
	10	Magnetometer 3	$\left(R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$

Note

There may be inputs (ports) not connected since the S-Function contains the maximum number of possible inputs for the different hardware and SIL Simulator versions, as can be seen in the aforementioned table.



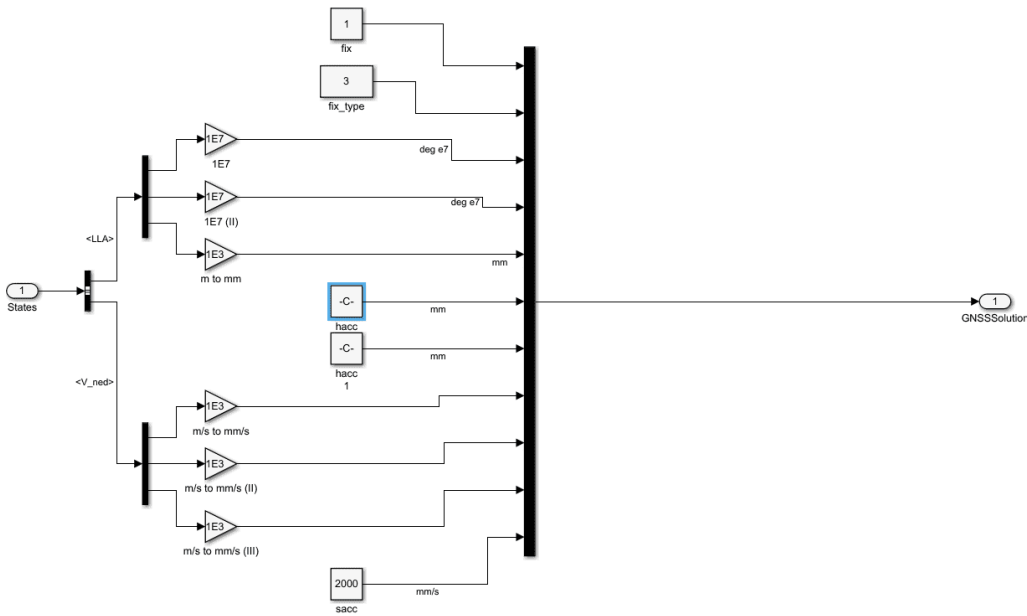
Magnetometer - Subsystem

The user can also simulate another magnetometer (external magnetometer) and send the information through a serial port. For more information on serial communication, refer to [Serial communications](#) section of this manual.

GNSS

GNSS receiver ports (there are 2 ports, GNSS1 and GNSS2) expect to receive an **array with the following information:**

1. Fix status
2. Fix type
 - 0: no fix
 - 1: dead reckoning only
 - 2: 2D-fix
 - 3: 3D-fix
 - 4: GNSS + dead reckoning fix
3. Longitude
4. Latitude
5. Altitude
6. Horizontal accuracy
7. Vertical accuracy
8. North Velocity
9. East velocity
10. Down Velocity
11. Speed Accuracy

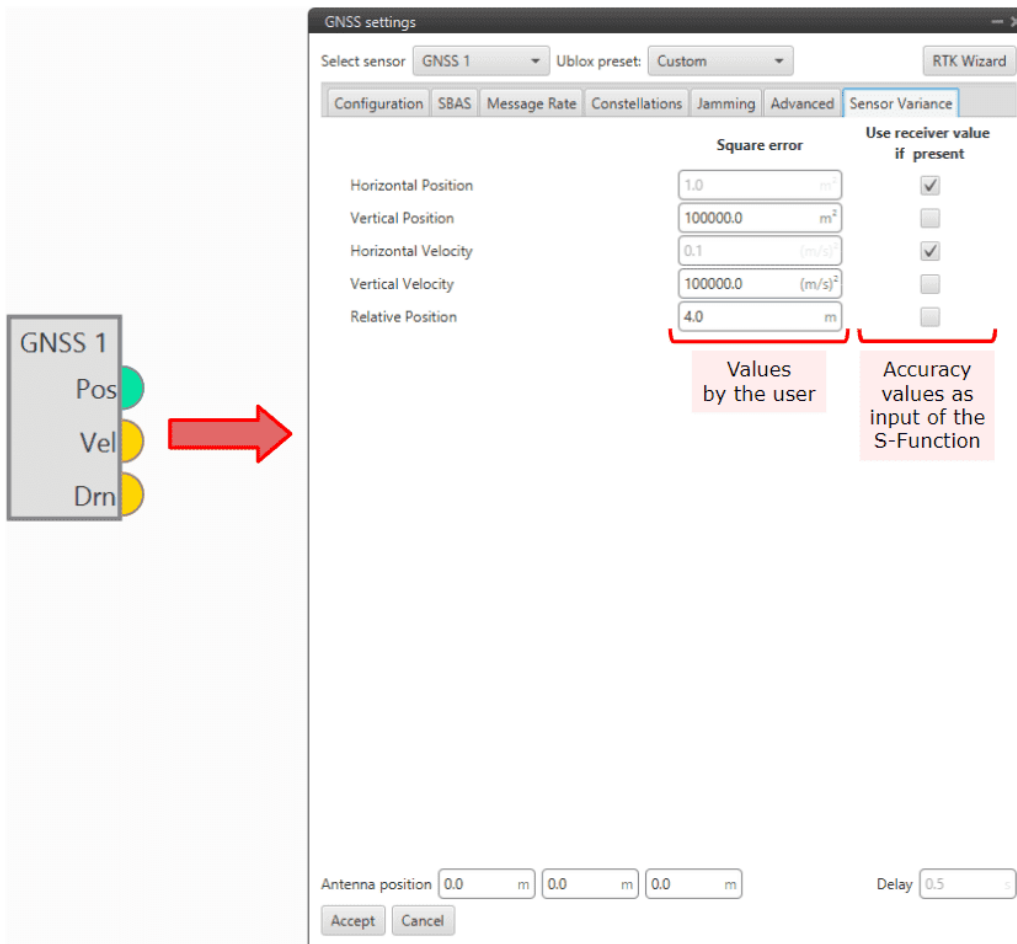


GNSS array

Note

- The **angle inputs** are in **degrees · 10⁷** units.
- The **distance inputs** are in **millimetres** units.
- The **speed inputs** are in **millimeters per second** units.

The **accuracy values are equal to the square root of the "Square error" parameter**. These values are supposed to be computed by the GPS device and are used in the EKF for GNSS solution. However, in the configuration files user can choose between these ones or values set by user.



GNSS variances - 1x PDI Builder

RTK Example Block

To enable the RTK feature, the user has to modify the configuration (for more information on this, see [GNSS sensor block - Block Programs](#) section of the **1x PDI Builder** user manual), and include more inputs via the S-Function.

This input is called Relative Position, and requires an **array of 10 elements**:

1. **Status**: 0 is Data invalid and 1 is Data valid.
2. **RelPosN**: North component of relative position vector (cm)
3. **RelPosE**: East component of relative position vector (cm)
4. **relPosD**: Down component of relative position vector (cm)

Note

RelPos represents the relative position between two GNSS antennas, the vector between the primary and secondary antenna. This position is composed of RelPosN, RelPosE, RelPosD.

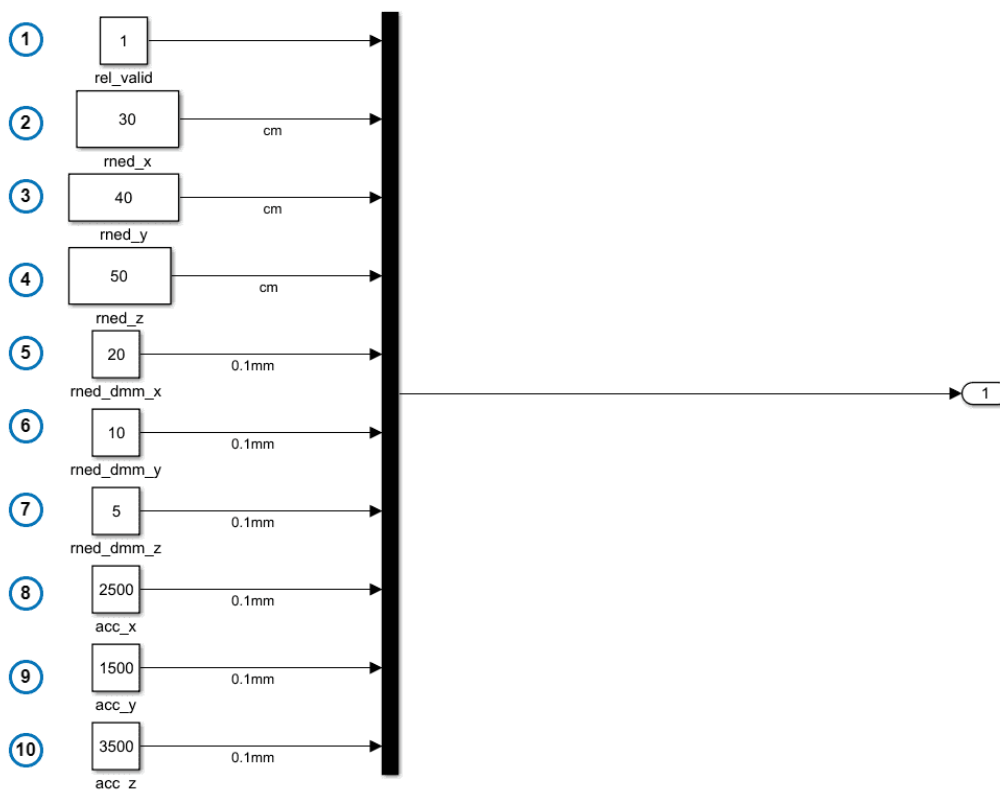
5. **relPosHPN**: High precision North component (mm)
6. **relPosHPE**: High precision East component (mm)
7. **relPosHPD**: High precision Down component (mm)
8. **accN**: Accuracy of relative position North component (mm)
9. **accE**: Accuracy of relative position East component (mm)
10. **accD**: Accuracy of relative position DOWn component (mm)

High precision components must be **in range -99 to 99 millimetres**. The full component of the relative position vector (in cm) is given by the addition of the 2 components.

Note

The relative position vector is only required when the GNSS Compass is implemented.

An example of this subgroup is shown below:



RTK inputs

ADC

Veronte Autopilot 1x is equipped with **5 external ADC channels** (linked to 5 pins) and **12 internal channels**. Therefore, in total, user has to create an **array of 17 elements**. These values are stored as internal variables in Veronte Autopilot 1x, and it is possible to use them in certain user programs.

The order of this array is: Internal ADC Channel 0, External ADC Channel 0-4, Internal ADC 1-11.

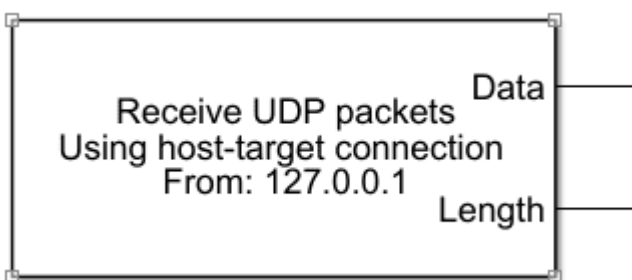
The following image shows an example (with the first external ADC pin):

ADC reading

Serial Communications

Veronte Autopilot 1x can manage input and output serial ports (for more information on this, see the [I/O Setup - Input/Output](#) section of the **1x PDI Builder** user manual).

A **simply way to create serial frames** (data in length wires) is by using the **simulink UDP block**. Therefore, the **data** entering **Veronte Autopilot 1x** should be **sent via UDP** (if this approach is adopted):



UDP Block

The ports included in **Autopilot 1x** and represented in the S-Function are as follows:

- **USB**: USB port
- **SCI-A**: 4G connection
- **SCI-B**: Radio
- **SCI-C**: Serial Port 485
- **SCI-D**: Serial Port 232

Example: Sending a RS-232 message

In the following example, a constant value is sent as a RS-232 message.

First, the message is created as a bit array with **Byte Pack block**.

Next, it is necessary to **receive this information as UDP packets** on the corresponding port (in this case 16003). **Width block** is used to compute data length. This UDP packet is then sent to the S-Function:

Sending a RS-232 message in Simulink

Finally, the **Autopilot 1x** configuration should be able to parse this information using **Serial Custom Messages consumers**. For more information on this, refer to [Serial Custom Messages - Input/Output](#) section of the **1x PDI Builder** user manual.

Custom message - 1x PDI Builder

Warning

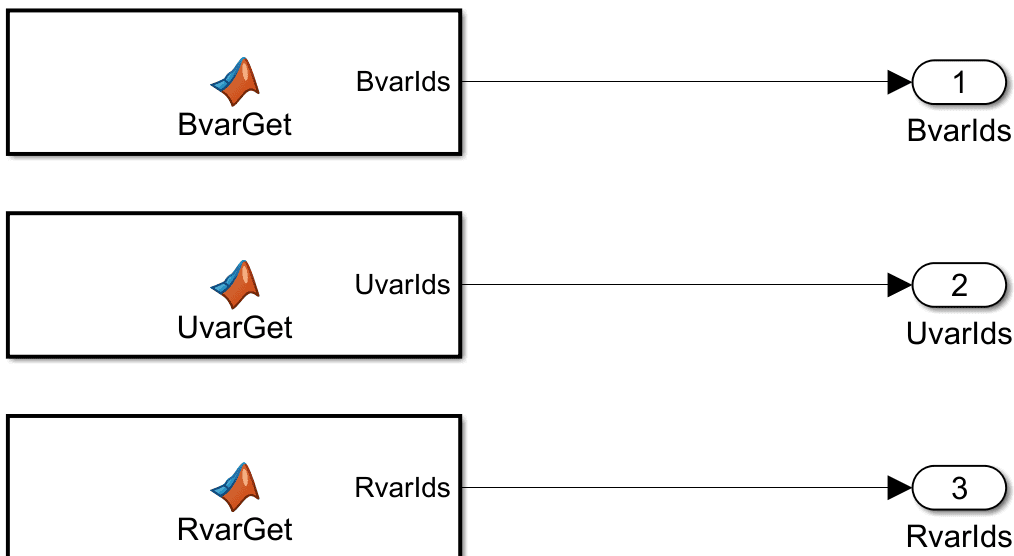
The variable type parsed by **Veronte Autopilot 1x** has to match the variable type generated in the **Byte Pack** block.

Telemetry

In the S-Function there are 3 inputs specially dedicated to select custom telemetry (pin 22 for Bit variables, pin 23 for Unsigned variables and pin 24 for Real variables). The size of each of these inputs is not fixed, although it has to be continuous throughout the simulation.

Users must enter the corresponding Ids of the variables that is aiming to monitor. The ID of each variable can be easily found in the [Lists of Variables](#) section of the **1x Software Manual**.

In the following example, a block function has been configured for each type of variables:



Telemetry blocks

The example function for real variables is given below. In it, the desired real variables are configured to be displayed.

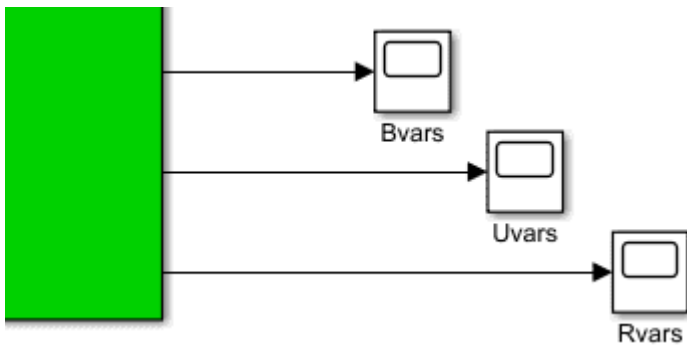
```

Requested Variables/MATLAB Function2* x +
1  function RvarIds = RvarGet()
2  % Requested Ids (row vector)
3  requestedIds = [...
4      330:336, ... % IMU 0 Raw Measurements
5      337:343, ... % IMU 1 Raw Measurements
6      361:367, ... % IMU 2 Raw Measurements
7      386:392, ... % IMU 3 Raw Measurements
8      348:349, ... % Static Pressure 0 (HSC) Raw Measurements
9      344:345, ... % Static Pressure 1 (MS56) Raw Measurements
10     368:369, ... % Static Pressure 2 (DPS310) Raw Measurements
11     322:325, ... % Internal Magnetometer LIS3MDL Raw Measurements
12     370:373, ... % Internal Magnetometer MMC5883MA Raw Measurements
13     393:396, ... % Internal Magnetometer RM3100 Raw Measurements
14     346:347, ... % Dynamic Pressure Raw Measurement
15     1504:1506, ... % GNSS 1 LLA
16     1604:1606, ... % GNSS 2 LLA
17     ];
18  RvarIds = [requestedIds];

```

Telemetry block - Real variables function

Finally, the last 3 outputs of the S-Function are vectors containing the Bit, Unsigned and Real variables information respectively. The user can postprocess them as desired (Scope block, To Workspace block, etc.).



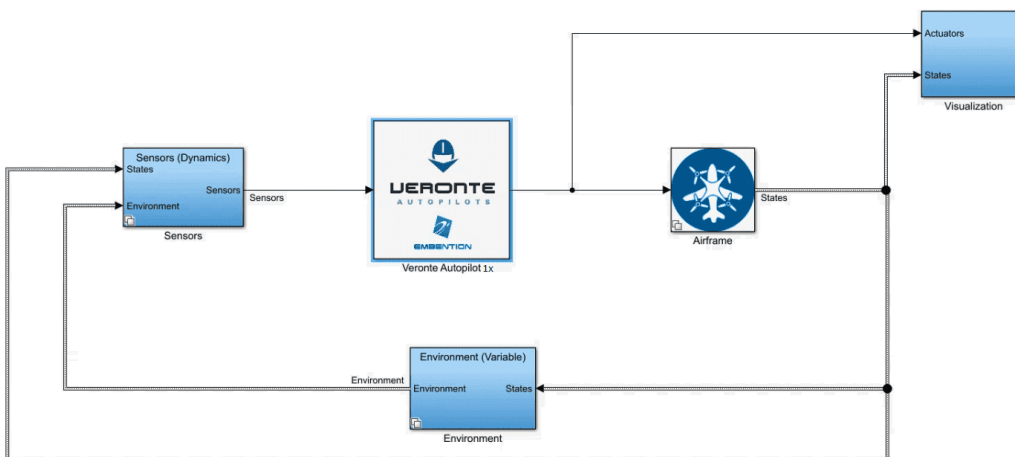
Display variables - Scope block example

Simulation

A complete simulation is composed of many systems or blocks.

In this manual the **sensors**, the **environment** and the **Veronte Autopilot 1x** subsystem have been already **introduced**. All these blocks must be combined with others, such as Airframe block to simulate the vehicle behavior.

Once the main blocks are configured, the complete simulation result should look like this:



Complete Setup Example

Important

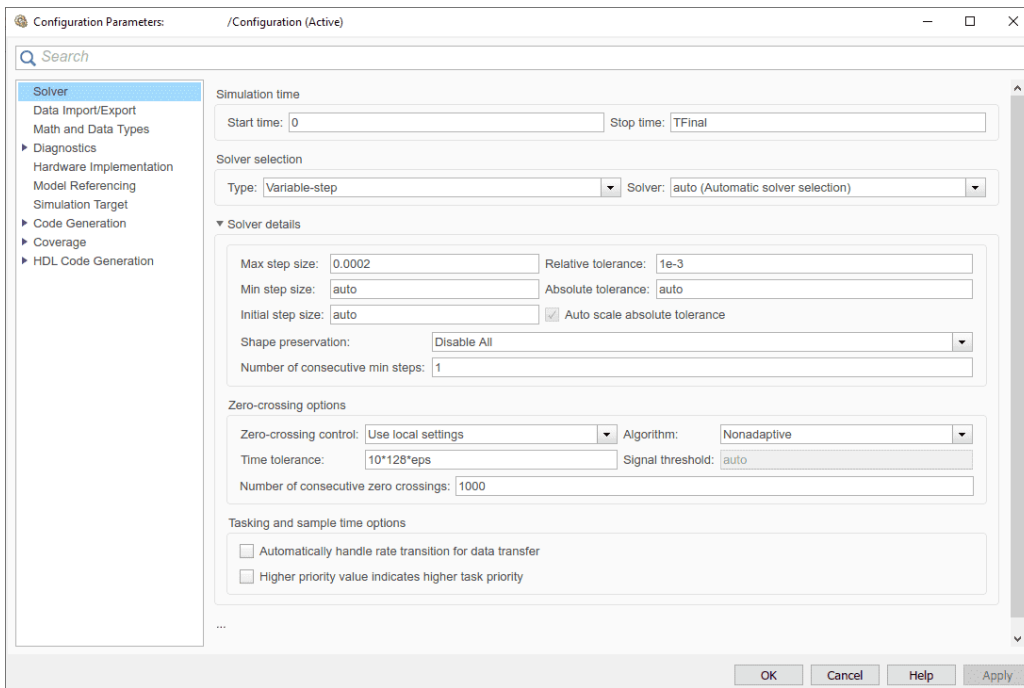
- Please note that this is only an example.
- Users will not receive this example with SIL Simulator package.

The main systems are:

- **Veronte Autopilot 1x:** Consists basically of the S-Function and its link with the rest of the blocks (sensors, outputs, etc.)
- **Airframe:** A model of the flight dynamics. The inputs of this system are the outputs of the Veronte Autopilot 1x system (nominal value for servos). For example, for a quadcopter, the input to this block consists of the PWM signal values (one for each motor). Then, with this value, the airframe system updates the status of the platform. The state vector is used to predict new environmental conditions and sensor readings.
- **Environment:** A model of the atmosphere, magnetic field, WGS84, etc.
- **Sensors:** It contains individual blocks or subgroups of all sensors that Veronte Autopilot 1x needs as input.
- **Visualization:** Contains display blocks, scopes, flight instruments, etc.

To ensure that everything works correctly, users must choose the simulation time step according to GNC frequency and core 1 frequency. Core 1 frequency is 1000 Hz fixed while GNC frequency is configurable. **The simulation time step must be a common divisor of both time steps.**

For example: if GNC has a time step of 0.0025 s (frequency = 400 Hz) and core 1 has a time step of 0.001 s (1000 Hz), the simulation can be set to 0.0005 s (2000 Hz), so the S-Function will execute core 1, GNC or both everytime they are required.



Time step settings

CAN support

The 6.14 version of the SIL (Software-in-the-Loop) simulator introduces the implementation of logic for CAN protocol management and simulation through SIL.

- The CAN inputs and outputs of the SIL block are structured according to the CAN Message format established by Simulink. This uniformity ensures compatibility, allowing for the use of either Simulink's native CAN blocks or third-party developed blocks within the simulation environment.
- It is possible to implement a CAN connection without using the Simulink CAN blocks, for example by directly connecting the CAN-A output to the CAN-B input within the SIL block. However, to ensure correct operation in this configuration, the insertion of a delay in the signal path using the Delay block is mandatory.
- The Simulink CAN Transmit block continuously sends the message present at the input, a behavior that diverges from reality, where CAN messages are typically transmitted on event.

Note

To align the simulation with real-world behavior, two solutions exist: the block can be configured so that transmission occurs only when a change in data occurs, or the block itself can be disabled in Simulink, managing the transmission via the timestamp

- The CAN messages exiting the SIL block are completed with the timestamp. This data is crucial: it allows the SIL block to unequivocally distinguish a new message and also serves as a mechanism to controllably enable or disable the CAN Transmit block, thereby solving the problem of continuous transmission.
- To ensure the correct operation of the simulated CAN system, it is essential to adequately configure the receiving mailboxes. The simulation rigorously takes into account both the filters set and the number of mailboxes specifically assigned to each ID.

Troubleshooting

dll_config.vcfg file not working

A usual mistake while configuring `dll_config.vcfg` is defining paths using quoting marks. Note that paths must be defined as explained in [dll_config.vcfg file](#) section of this manual.



Not valid configuration:

```
# Path to DLL  
\dll ".\x64\VeronteDLL.dll"
```



Valid configuration:

```
# Path to DLL  
\dll .\x64\VeronteDLL.dll
```

Connection

S-Function connection failed

To connect to the S-function in Simulink, it is necessary to have both inbound and outbound firewall permissions for Matlab. If these are not available, a firewall exception for Matlab must be configured to allow only the desired IP address to pass through.

Simulink IP adress

When utilizing a Virtual Box, with MATLAB running on the guest system (not the host computer), the IP adress in the Simulink input and output blocks must be modified.

Logs

SIL.log provides a record of any issues that have arisen during the execution process, please take a look at it if anything fails and do not hesitate to contact the support team using the **Joint Collaboration Framework**; for more information, please consult the [JCF user manual](#).

In case the log shows a PDI error, please refer to the [List of PDI Errors](#) section in the **1x Software Manual**.

License ID warning in Veronte Ops

Veronte_SD_Image has a license associated to a specific ID. Therefore, customizing the ID of the virtual autopilot may result in a limited performance of the system.

For solving this issue, open **Veronte Ops** application and update the limited license as explained in the [Platform license - Platform](#) section of **Veronte Ops** user manual.



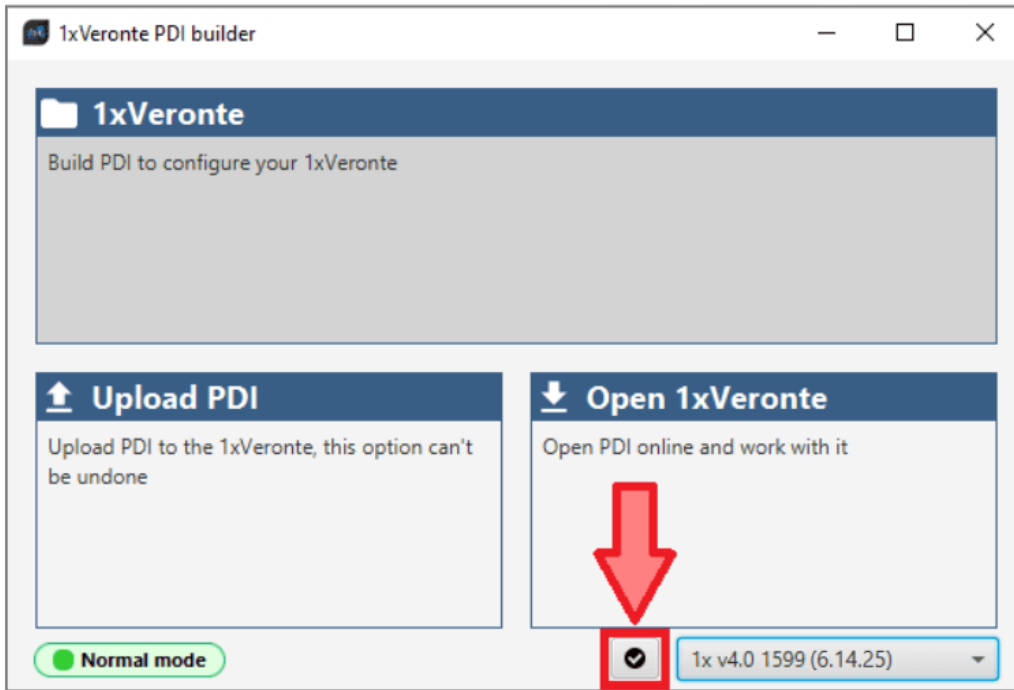
TIVeronte Ops - Platform licenseTLE

Loaded with errors in Veronte Link

Once the simulation is running, the system of Veronte works as if working with a physical autopilot. This means that the simulated autopilot can appear in

Veronte Link as **Loaded with errors** for several reasons unrelated to SIL Simulator.

To handle these, users should consult the error type in the **1x PDI Builder**, referring to the [1x PDI Builder](#) user manual for more information.



1x PDI Builder - PDI Errors button

Running Veronte Console

In the case of using **VeronteConsole.exe**, it is also possible to pass the paths as **arguments to the windows command**, for this:

1. Open windows command => `cmd`
2. Pass to it:

```
VeronteConsole.exe \dll C:\Users\user\Folder\VeronteDLL.dll \image C:\Users\user\Folder\Veronte_SD_Image.img
```

Acronyms and Definitions

ADC	Analog to Digital Converter
CAN	Controller Area Network
DLL	Dynamic Link Library
EKF	Extended Kalman Filter
FTP	File Transfer Protocol
GNC	Guidance Navigation Control
GNSS	Global Navigation Satellite Systems
GPS	Global Positioning System
HIL	Hardware In the Loop
IMU	Inertial Measurement Unit
ISA	International Standard Atmosphere
JCF	Joint Collaboration Framework
NED	Noth East Down (coordinates)
PC	Personal Computer
PDI	Parameter Data Item
PWM	Pulse Width Modulation
RS232	Recommended Standard 232
RTK	Real Time Kinematic
SCI	Serial Communication Interface
SIL	Software In the Loop

SRTM	Shuttle Radar Topography Mission
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
USB	Universal Serial Bus
WGS84	World Geodetic System 84
WMM	World Magnetic Model