SIL Simulator

Release 6.12.92

Embention

2024-04-09

CONTENTS

1 Introduction **3**
- 1.1 Veronte Console . . . 4
- 1.2 Veronte S-function . . . 4

2 Quick Start **5**
- 2.1 Download . . . 5
- 2.2 Requirements . . . 6

3 Configuration **9**
- 3.1 dll_config.vcfg file . . . 9
- 3.2 Step by step - Veronte S-Function . . . 10
- 3.3 Step by step - Veronte Console . . . 14

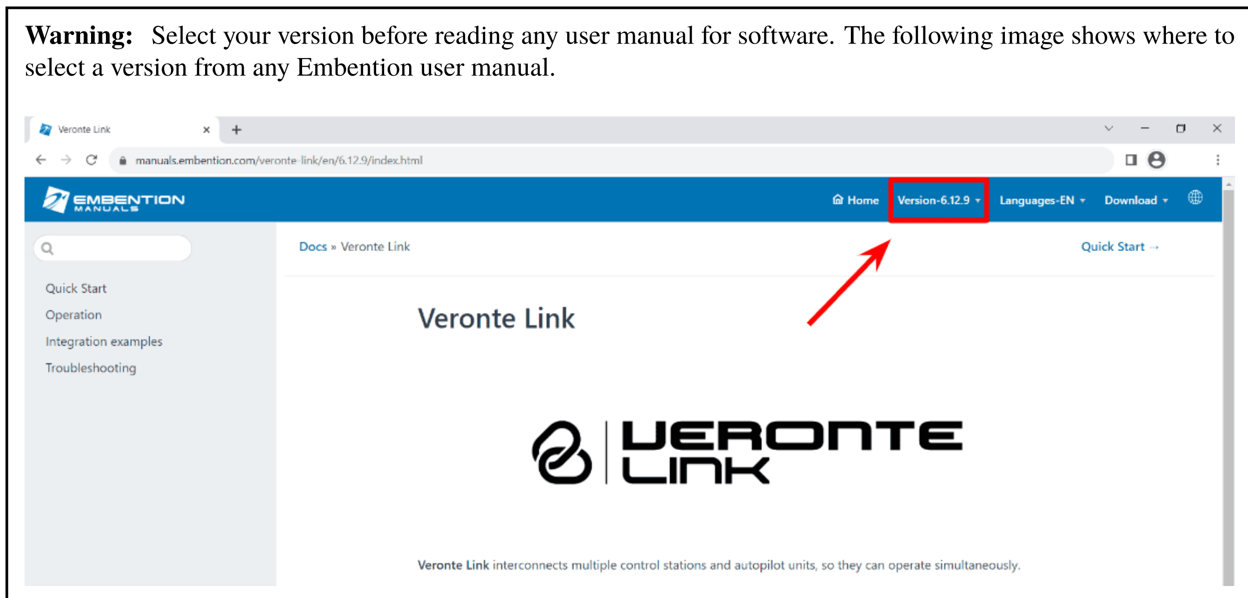
4 Simulink workspace **17**
- 4.1 Inputs . . . 19
- 4.2 Outputs . . . 20
- 4.3 Sensors . . . 21
  - 4.3.1 Environment . . . 22
  - 4.3.2 Pressure sensors . . . 24
    - 4.3.2.1 Static Pressure . . . 24
    - 4.3.2.2 Dynamic Pressure . . . 25
  - 4.3.3 IMU . . . 27
  - 4.3.4 Magnetometer . . . 34
  - 4.3.5 GNSS . . . 35
  - 4.3.6 ADC . . . 39
  - 4.3.7 Serial Communications . . . 40
- 4.4 Telemetry . . . 41
- 4.5 Simulation . . . 43

5 Troubleshooting **45**
- 5.1 dll_config.vcfg file not working . . . 45
- 5.2 Logs . . . 45
- 5.3 License ID warning in Veronte Ops . . . 45
- 5.4 Loaded with errors in Veronte Link . . . 46
- 5.5 Running Veronte Console . . . 47

6 Acronyms and Definitions **49**

SIL Simulator, or *Software-in-the-Loop Simulator*, is an **advanced simulation system** designed to replicate the functionality of an autopilot in a virtual environment.

Warning: Select your version before reading any user manual for software. The following image shows where to select a version from any Embention user manual.



CHAPTER

ONE

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.

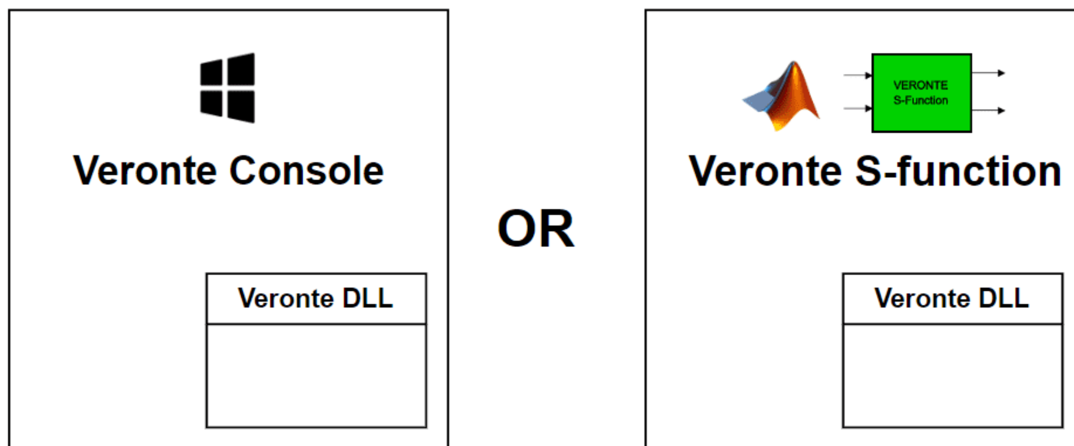


Fig. 1: **SIL Simulator interfaces**

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

1.1 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.

1.2 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.

CHAPTER

TWO

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*.

2.1 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.

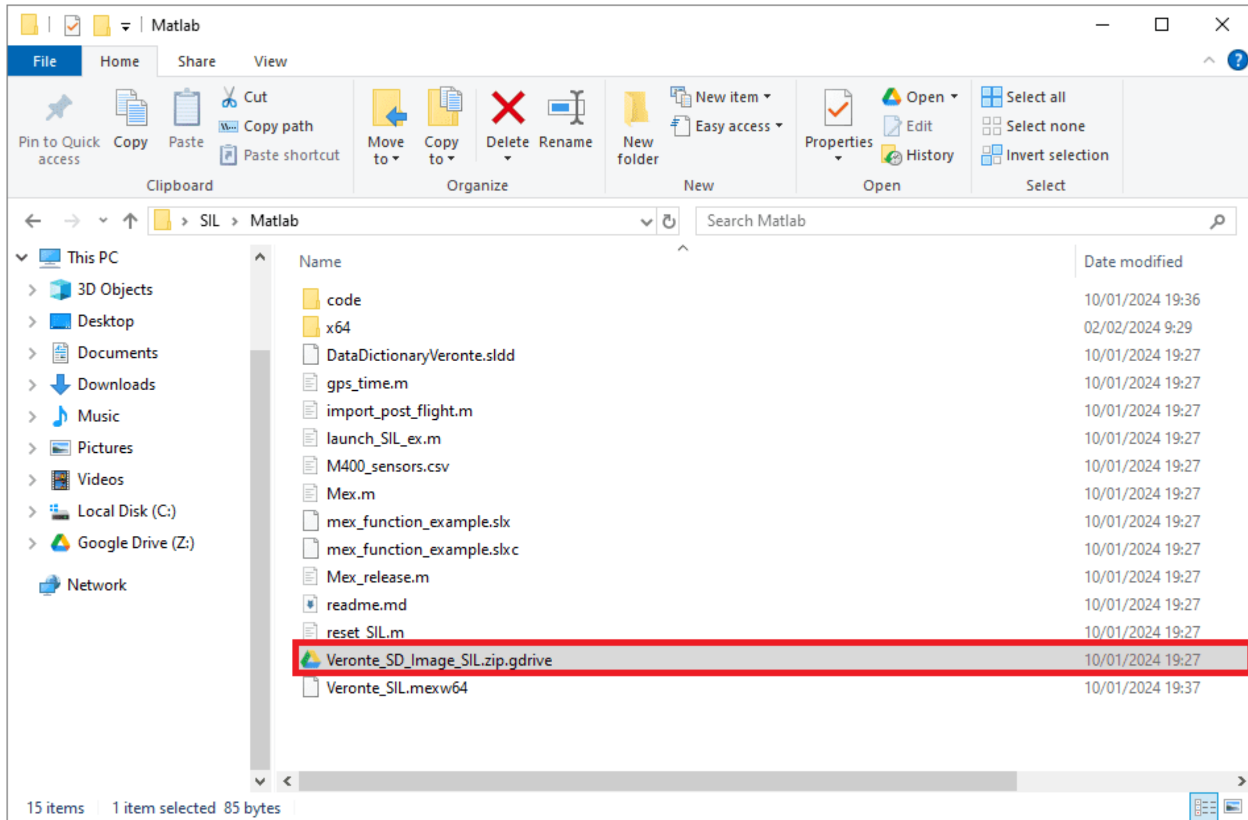


Fig. 1: **SIL folder - Drive file**

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.

2.2 Requirements

- **Veronte Software Package**:
  - **Veronte Link** (v6.12.X): 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++.

CHAPTER THREE

CONFIGURATION

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

3.1 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
hw v4.8	4041	4064	4065	4144	4146	4213

3.2 Step by step - Veronte S-Function

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

Fig. 1: **SIL folder**

2. 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`).

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 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.

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*:

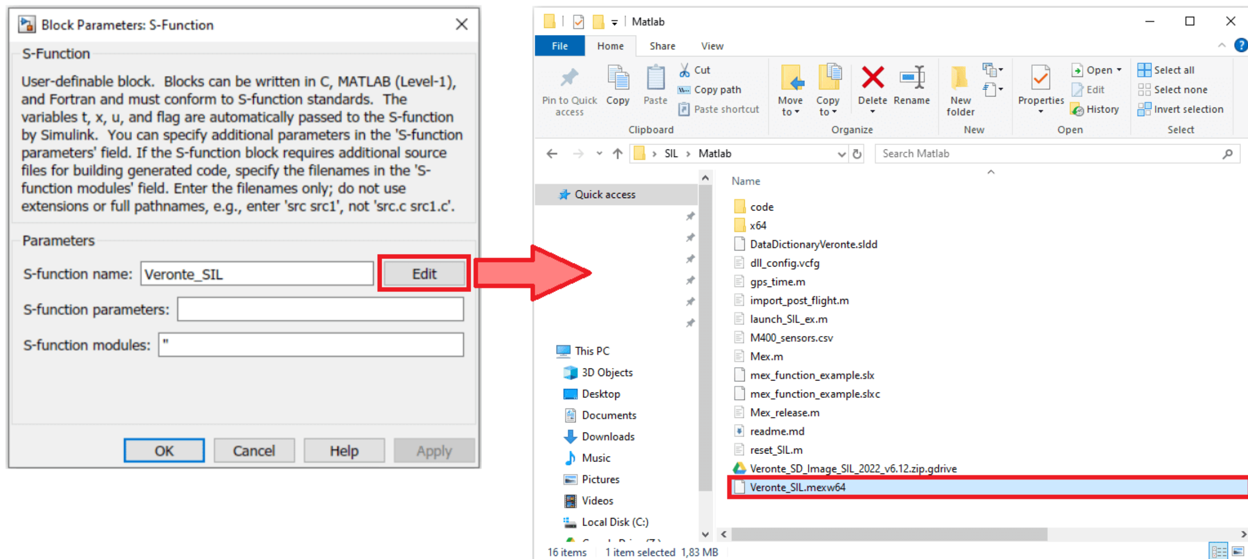


Fig. 2: **S-Function block parameters**

- 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*.

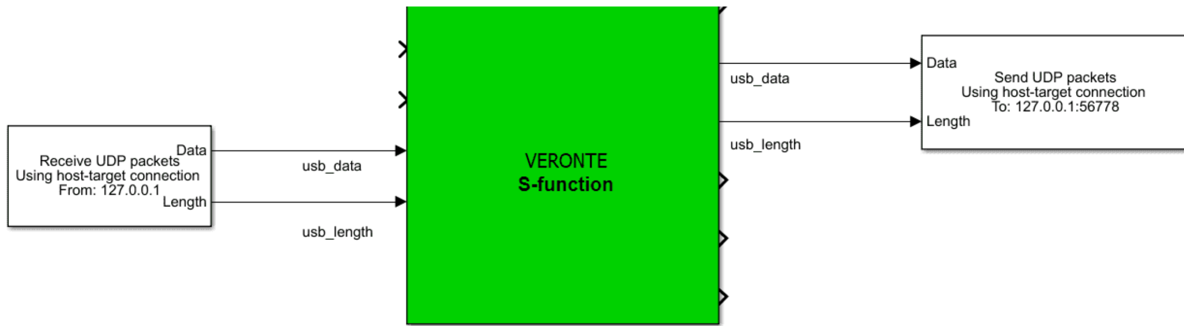


Fig. 3: **UDP Blocks**

- Configure the desired destination port.

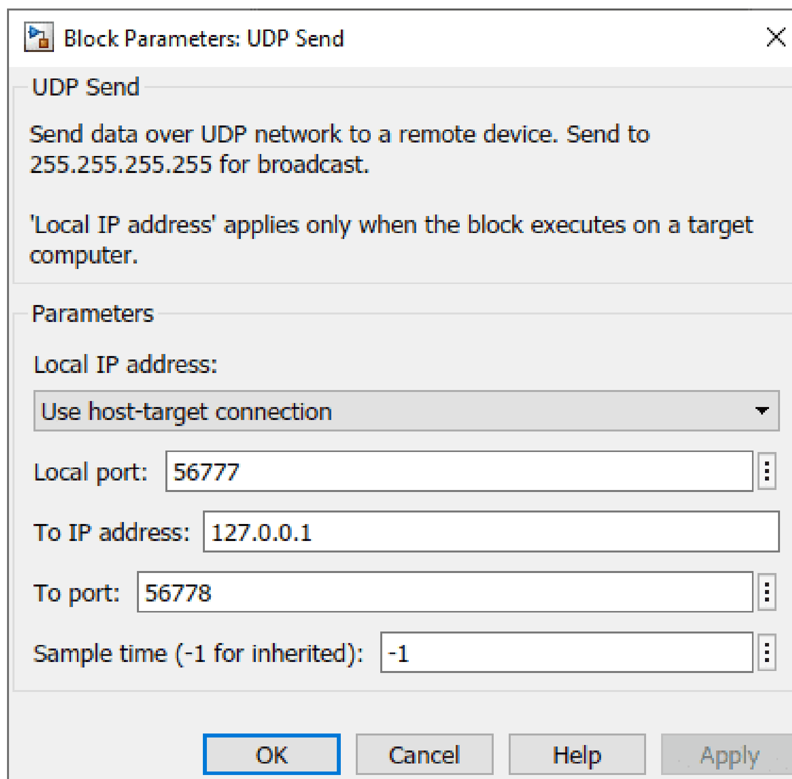


Fig. 4: **Destination UDP Port**

- 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 configurating 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**:

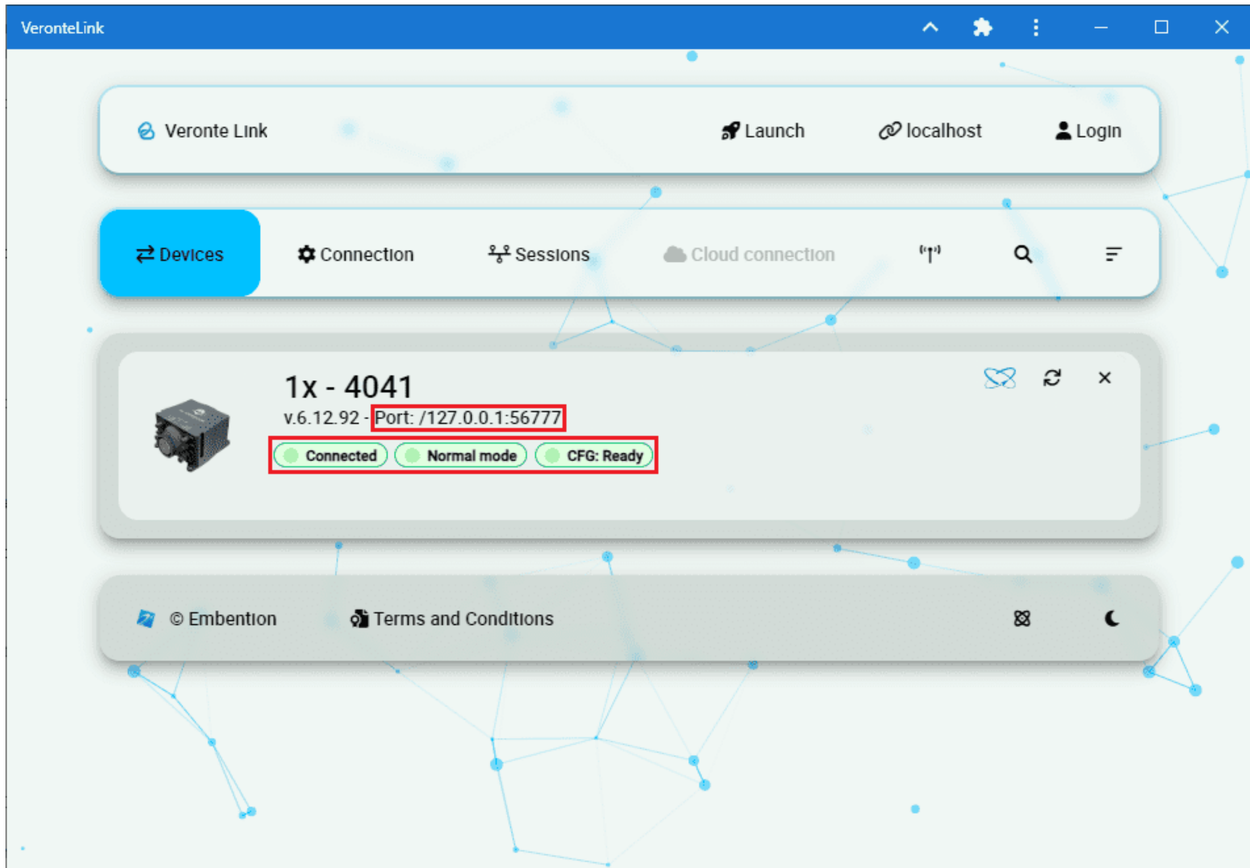


Fig. 5: **Veronte Link**

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 occuring within the simulation are registered on `SIL.log`.

3.3 Step by step - Veronte Console

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

Fig. 6: **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 configurating connections, please consult Connection section of the **Veronte Link** user manual.

   - **Type of connection**: UDP
   - **IP**: 127.0.0.1
   - **Port**: 12345

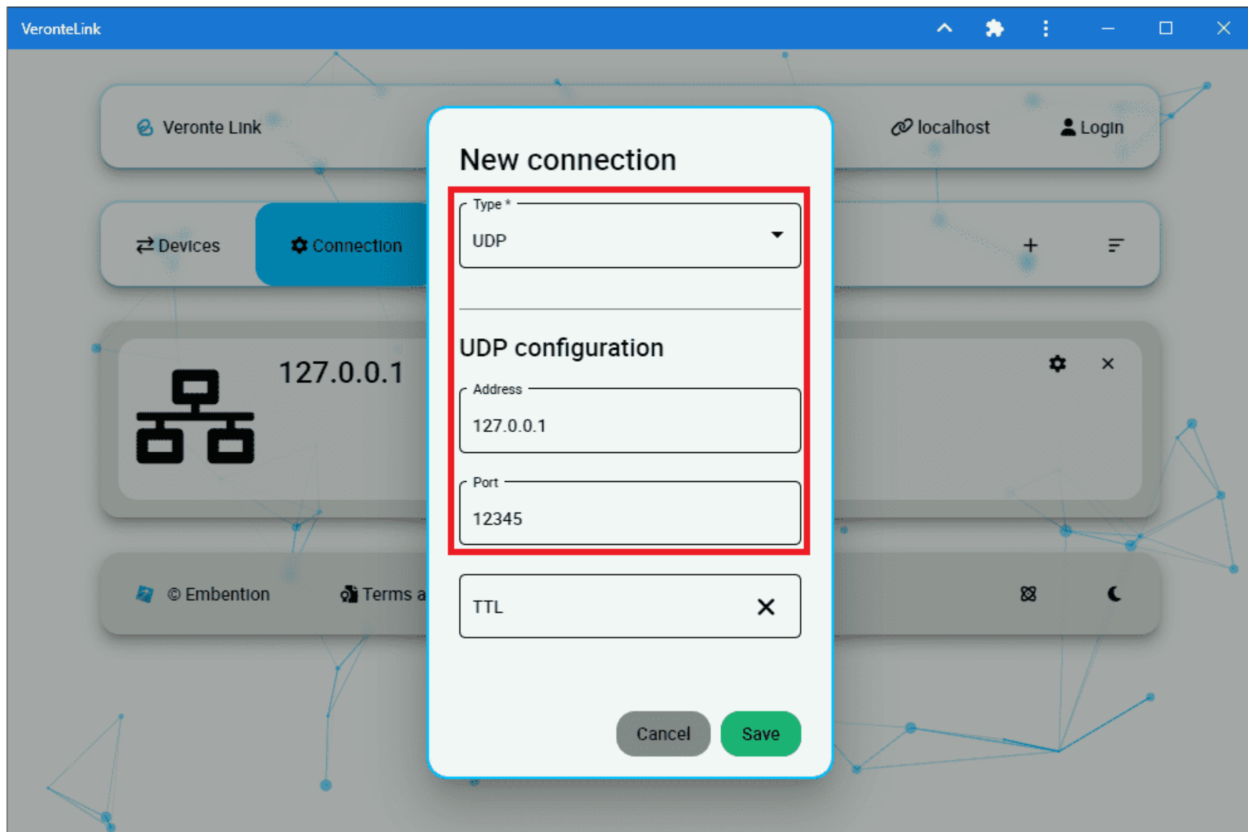


Fig. 7: **Veronte Link - UDP Connection for Veronte Console**

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

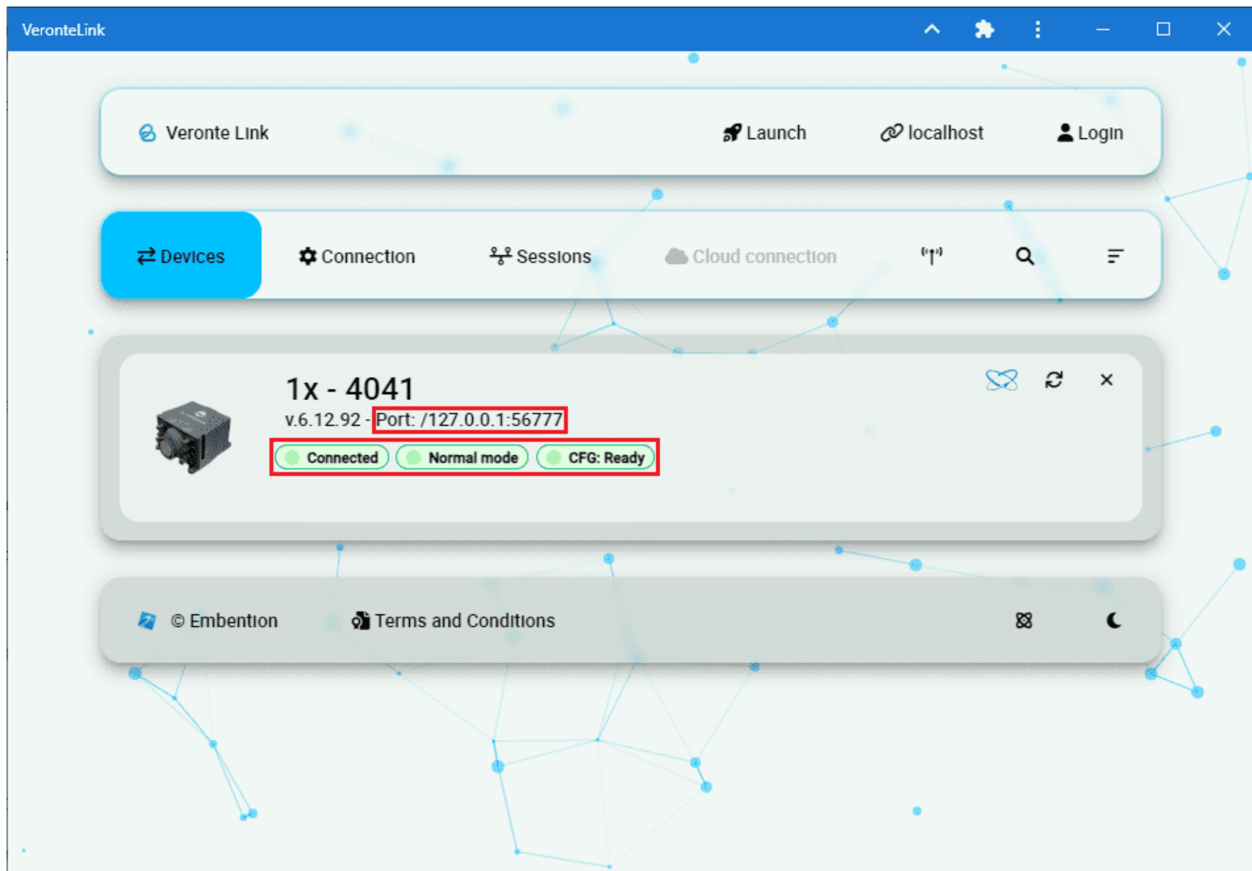


Fig. 8: **Veronte Link**

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.

CHAPTER

FOUR

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* 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.

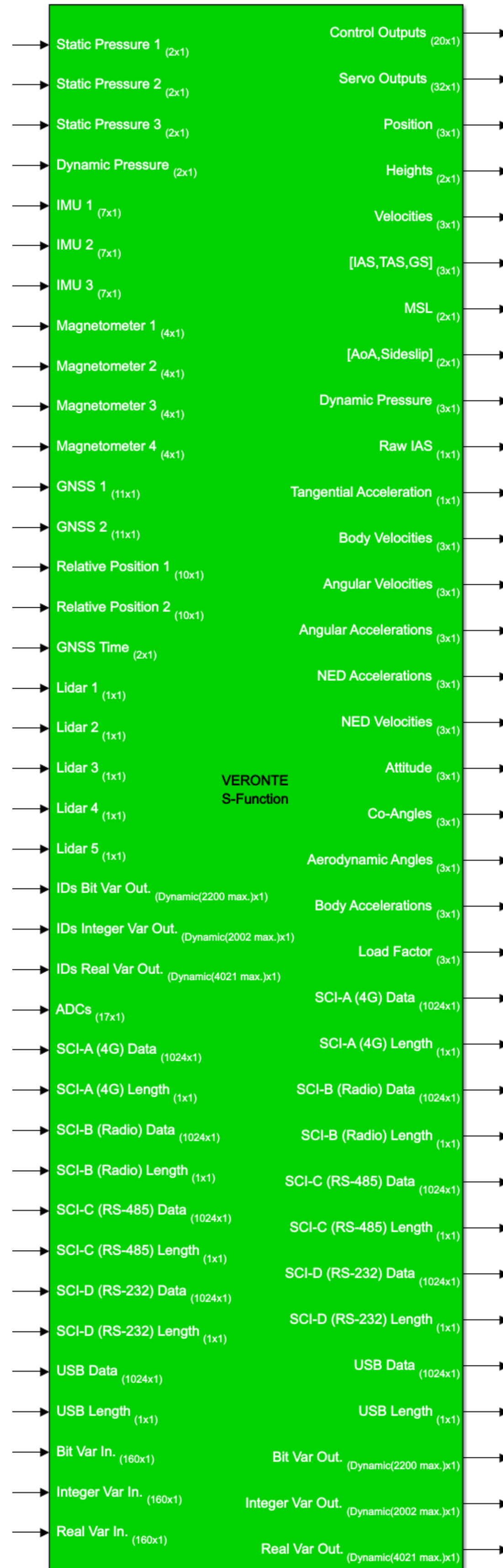


Fig. 1: **S-Function containing the Veronte Autopilot 1x embedded code**

4.1 Inputs

Inputs are described in the next table:

PIN	Description	Form	Size	Units
1	Static Pressure 1	[pressure_measurement;sensor temperature]	2x1	Pa ; K
2	Static Pressure 2	[pressure_measurement;sensor temperature]	2x1	Pa ; K
3	Static Pressure 3	[pressure_measurement;sensor temperature]	2x1	Pa ; K
4	Dynamic Pressure	[pressure_measurement;sensor temperature]	2x1	Pa ; K
5	IMU 1	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$;
6	IMU 2	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$;
7	IMU 3	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$;
8	Magnetometer 1	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
9	Magnetometer 2	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
10	Magnetometer 3	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
11	Magnetometer 4	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
12	GNSS 1	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]	11x1	$deg \cdot 10^7$
13	GNSS 2	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]	11x1	$deg \cdot 10^7$
14	Relative Position 1	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]	10x1	cm ; mm
15	Relative Position 2	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]	10x1	cm ; mm
16	GNSS Time	[week_number;milliseconds_of_week]	2x1	-; ms
17	Lidar 1	[lidar_measurement]	1x1	cm
18	Lidar 2	[lidar_measurement]	1x1	cm
19	Lidar 3	[lidar_measurement]	1x1	cm
20	Lidar 4	[lidar_measurement]	1x1	cm
21	Lidar 5	[lidar_measurement]	1x1	cm
22	IDs Bit Var Out.	[Var_IDs]	Dynamic(2200 max.)x1	-
23	IDs Unsigned Var Out.	[Var_IDs]	Dynamic(2002 max.)x1	-
24	IDs Real Var Out.	[Var_IDs]	Dynamic(4021 max.)x1	-
25	ADCs	[adc(1-17)]	17x1	-
26	SCI-A (4G) Data	[serial_data]	1024x1	-
27	SCI-A (4G) Length	[serial_length]	1x1	-
28	SCI-B (Radio) Data	[serial_data]	1024x1	-
29	SCI-B (Radio) Length	[serial_length]	1x1	-
30	SCI-C (RS-485) Data	[serial_data]	1024x1	-
31	SCI-C (RS-485) Length	[serial_length]	1x1	-
32	SCI-D (RS-232) Data	[serial_data]	1024x1	-
33	SCI-D (RS-232) Length	[serial_length]	1x1	-
34	USB Data	[serial_data]	1024x1	-
35	USB Length	[serial_length]	1x1	-
36	Bit Var In.	[Var0_ID;Var0_value;…;Var80_ID;Var80_value]	160x1	-
37	Unsigned Var In.	[Var0_ID;Var0_value;…;Var80_ID;Var80_value]	160x1	-
38	Real Var In.	[Var0_ID;Var0_value;…;Var80_ID;Var80_value]	160x1	-

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.

4.2 Outputs

Outputs are the following:

PIN	Description	Form	Size	Units
1	Control Outputs	[control_outputs(1-20)]	20x1	-
2	Servo Outputs	[servos(1-32)]	32x1	-
3	Position	[lon;lat;alt]	3x1	rad ; m
4	Heights	[msl,agl]	2x1	m
5	Velocities	[longitudinal_v;lateral_v;velocity(module)]	3x1	$\frac{m}{s}$
6	IAS, TAS, GS	[ias,tas,gs]	3x1	$\frac{m}{s}$
7	MSL	[msl_from_qnh;msl_from_ISA]	2x1	m
8	Angle of Attack, Sideslip	[angle_of_attack;sideslip]	2x1	rad
9	Dynamic Pressure	[dynamic_pressure]	3x1	Pa
10	Raw IAS	[ias_raw]	1x1	$\frac{m}{s}$
11	Tangential Acceleration	[tangential_acceleration]	1x1	$\frac{m}{s^2}$
12	Body Velocities	[longitudinal_v;lateral_v;vertical_v]	3x1	$\frac{m}{s}$
13	Angular Velocities	[roll_rate;pitch_rate;yaw_rate]	3x1	$\frac{rad}{s}$
14	Angular Acceleration	[acc_z_axis;acc_y_axis;acc_x_axis]	3x1	$\frac{rad}{s}$
15	NED Acceleration	[acc_north;acc_east;acc_down]	3x1	$\frac{m}{s^2}$
16	NED Velocities	[v_north;v_east;v_down]	3x1	$\frac{m}{s}$
17	Attitude	[Yaw;Pitch;Roll]	3x1	rad
18	Co-Angles	[co-Yaw;co-Pitch;co-Roll]	3x1	rad
19	Aerodynamic Angles	[heading,flight_path;bank_angle]	3x1	rad
20	Body Accelerations	[acc_x,acc_y;acc_z]	3x1	$\frac{m}{s^2}$
21	Load factor	[nx;ny;nz]	3x1	-
22	SCI-A (4G) Data	[serial_data]	1024x1	-
23	SCI-A (4G) Length	[serial_length]	1x1	-
24	SCI-B (Radio) Data	[serial_data]	1024x1	-
25	SCI-B (Radio) Length	[serial_length]	1x1	-
26	SCI-C (RS-485) Data	[serial_data]	1024x1	-
27	SCI-C (RS-485) Length	[serial_length]	1x1	-
28	SCI-D (RS-232) Data	[serial_data]	1024x1	-
29	SCI-D (RS-232) Length	[serial_length]	1x1	-
30	USB Data	[serial_data]	1024x1	-
31	USB Length	[serial_length]	1x1	-
32	Bit Var Out.	[Var_values]	Dynamic(2200 max.)x1	-
33	Unsigned Var Out.	[Var_values]	Dynamic(2002 max.)x1	-
34	Real Var Out.	[Var_values]	Dynamic(4021 max.)x1	-

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 *sensors* and *telemetry* blocks, as well as general visualisation of a *complete simulation*.

4.3 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 ilustrate 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:

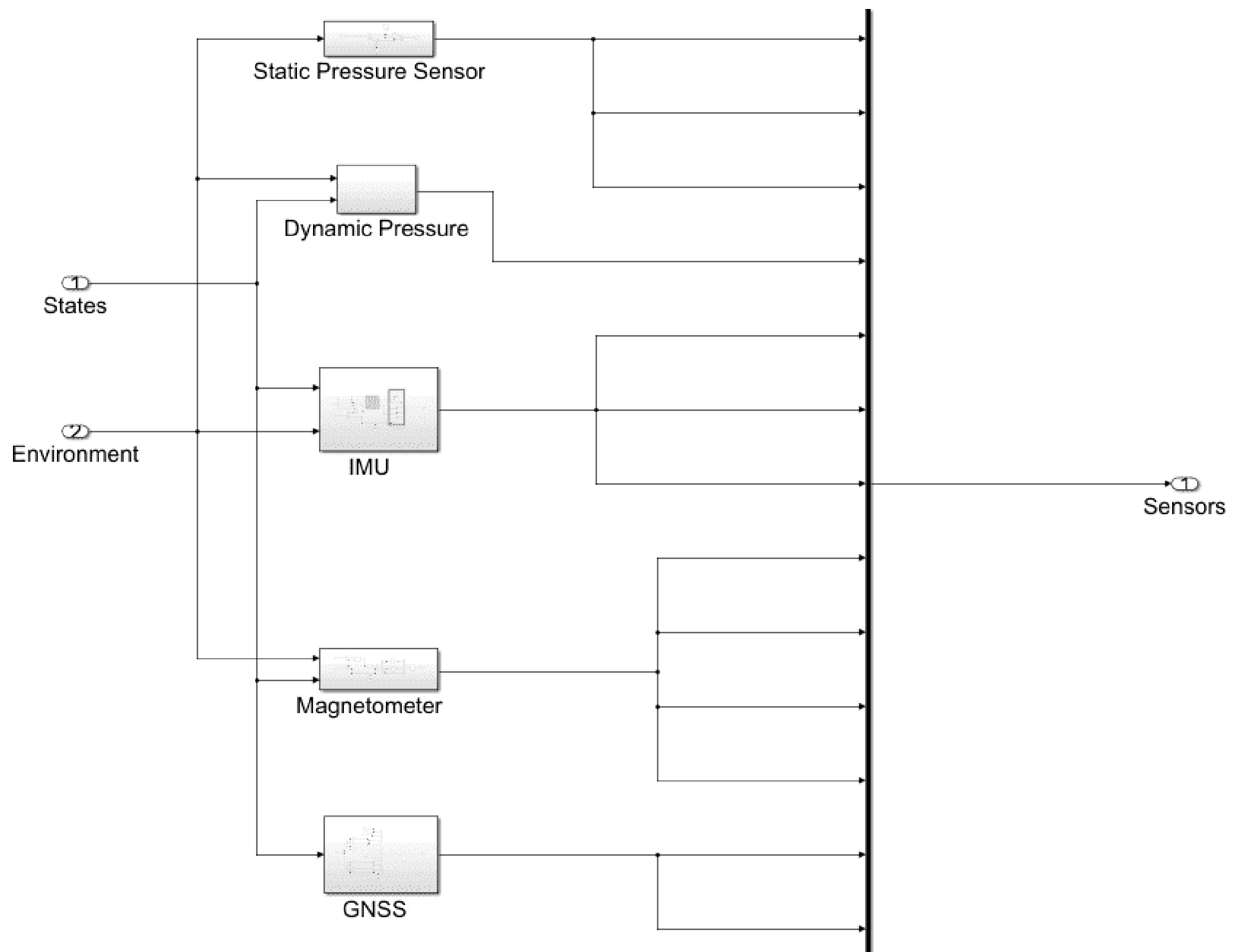


Fig. 2: **Sensors inputs**

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

- *Environment*

- *Pressure sensors*
- *IMU*
- *Magnetometer*
- *GNSS*
- *ADC*
- *Serial Communications*

4.3.1 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.

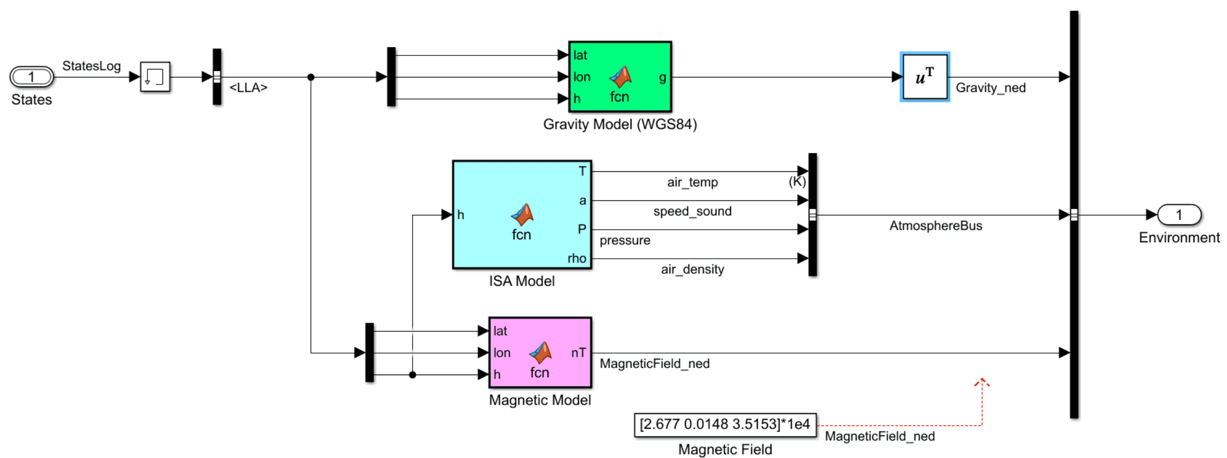


Fig. 3: **Environment block**

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

1. World Magnetic Model 2020:

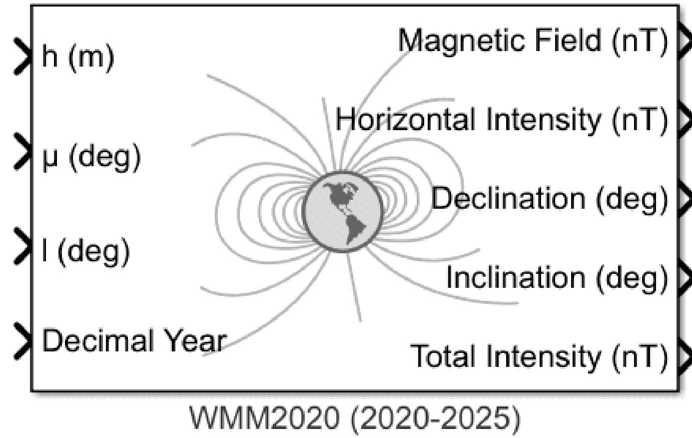


Fig. 4: **Aerospace blockset function - WMM2020**

2. ISA Atmosphere Model:

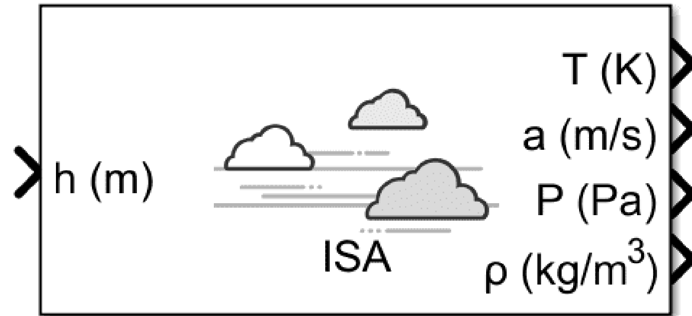


Fig. 5: **Aerospace blockset function - ISA Atmosphere Model**

3. WGS84 Gravity Model:

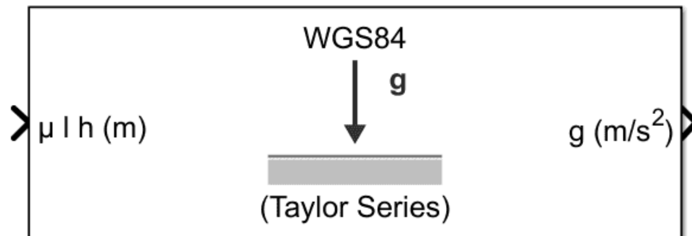


Fig. 6: **Aerospace blockset function - WGS84 Gravity Model**

4.3.2 Pressure sensors

4.3.2.1 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 contains **3 ports representing the 3 static pressure sensors that are included in Autopilot 1x**. Then this information should be used according to the static pressure sensor selected in the configuration (in the **1x PDI Builder** software).

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

Hardware version	Static Pressure
4.0	Static Pressure sensor 0
	Static Pressure sensor 1
4.5	Static Pressure sensor 0
	Static Pressure sensor 1
	Static Pressure sensor 2
4.8	Static Pressure sensor 1
	Static Pressure sensor 2

Important: Please note that, the number of inputs (ports) correspond to the maximum number of inputs available on all 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.

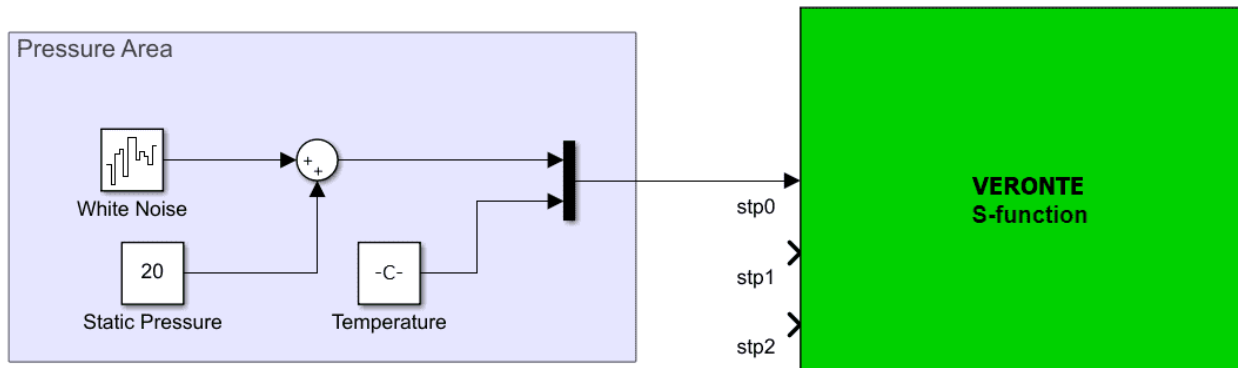


Fig. 7: **Static pressure - Constant**

- **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.

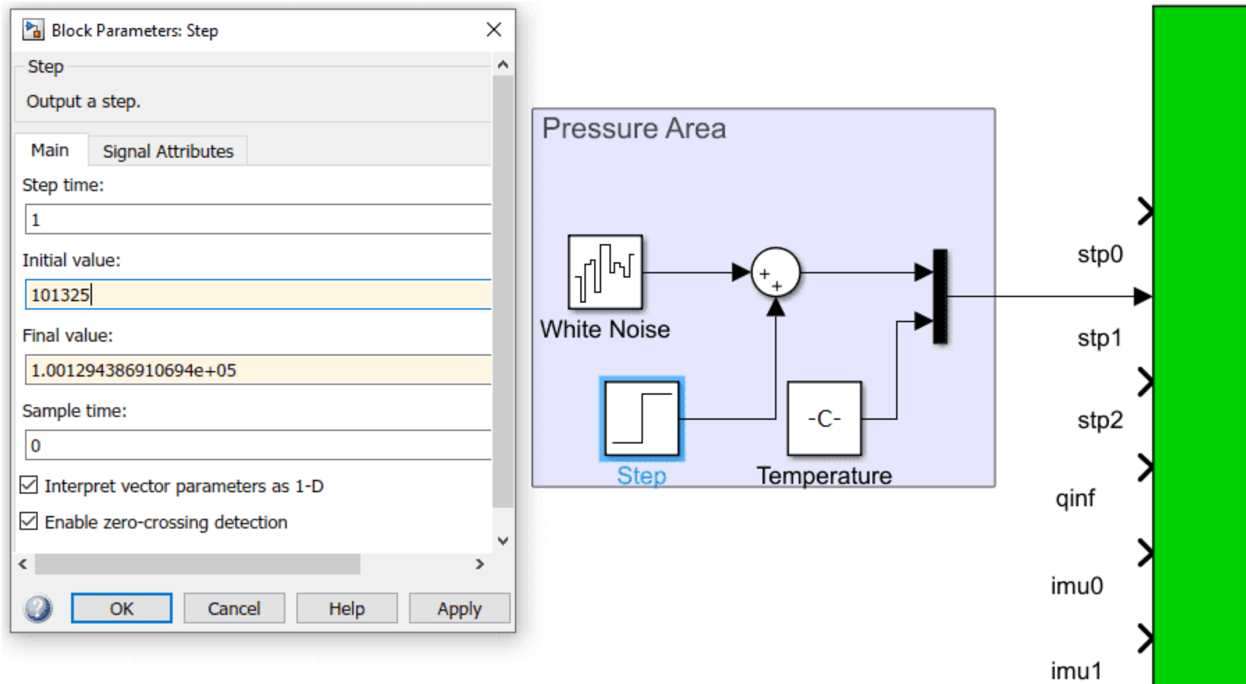


Fig. 8: **Static pressure - Step input**

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

4.3.2.2 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:

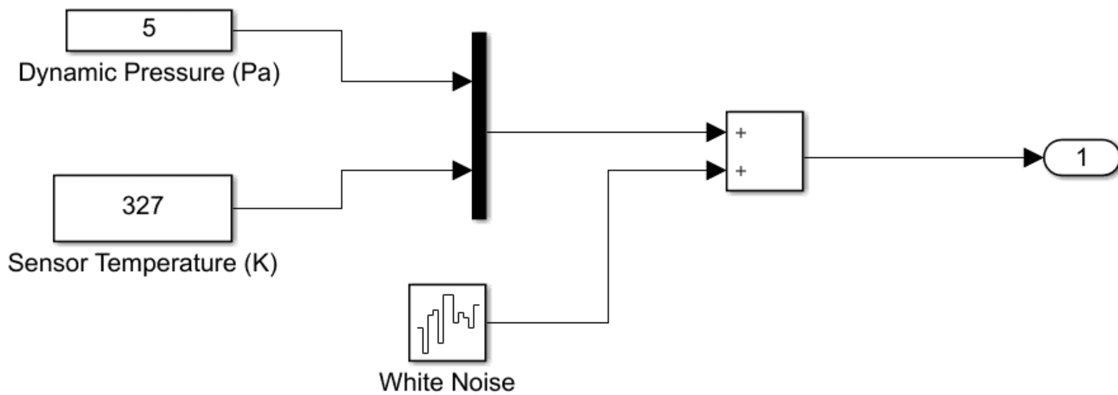


Fig. 9: **Dynamic pressure - Constant**

- **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.

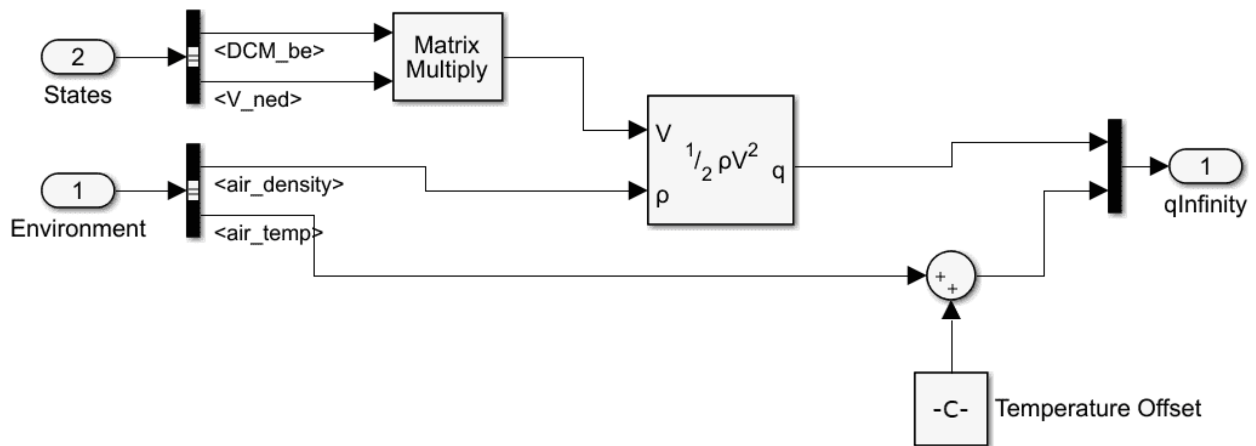


Fig. 10: **Dynamic Pressure - Subsystem**

4.3.3 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 the Veronte Autopilot 1x coordinates, please check the Orientation - Hardware Installation section of the **1x Hardware Manual**).
- Please note that, the number of inputs (ports) correspond to the maximum number of inputs available on all hardware versions, as can be seen in the following table.
- As detailed in the table below, if users wish to enter **measurements into the IMU 3** with hardware version `4.8`, they must enter them into the **IMU0 input**.

  This is because the block input values belonging to IMU0 also go internally to this IMU. Therefore, the same S-Function can be used for both hardware version `4.0` and `4.8`.

Hardware version	IMU	Rotation Matrix
4.0	**IMU0**	$R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	IMU1	$R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$
4.5	**IMU0**	$R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	IMU1	$R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$
	IMU2	$R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
4.8	**IMU3** **Note:** IMU3 corresponds to IMU0 input as explained in the *previous note*	$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$
	IMU1	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$
	IMU2	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Within the S-Function, the introduced data is transformed to match the coordinates of the 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**.

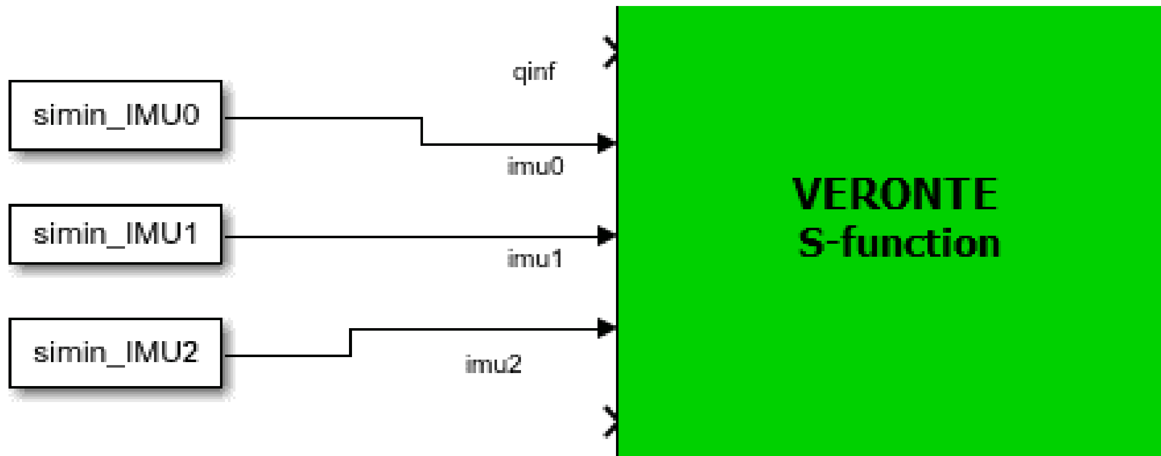


Fig. 11: **IMU - From workspace blocks**

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.

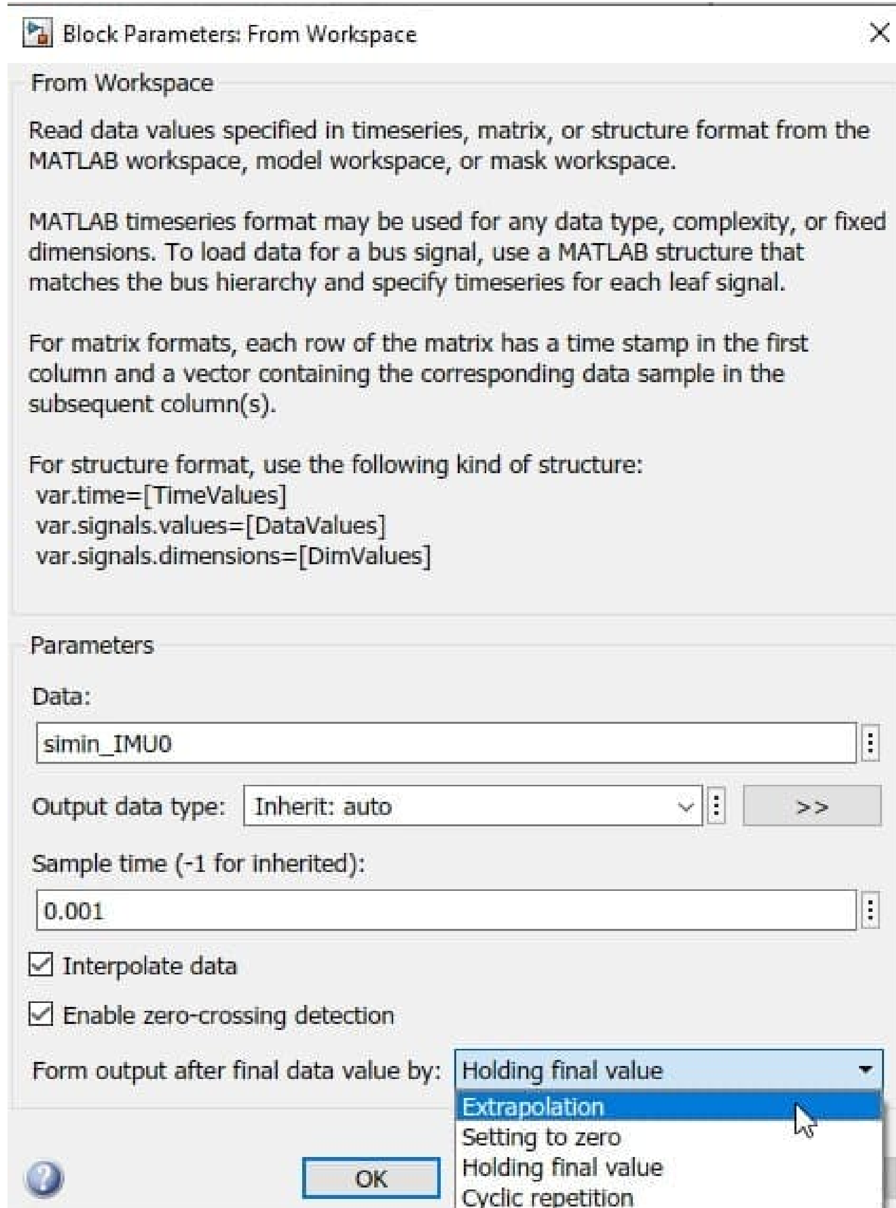


Fig. 12: **From workspace block configuration**

- **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 behaviour 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:

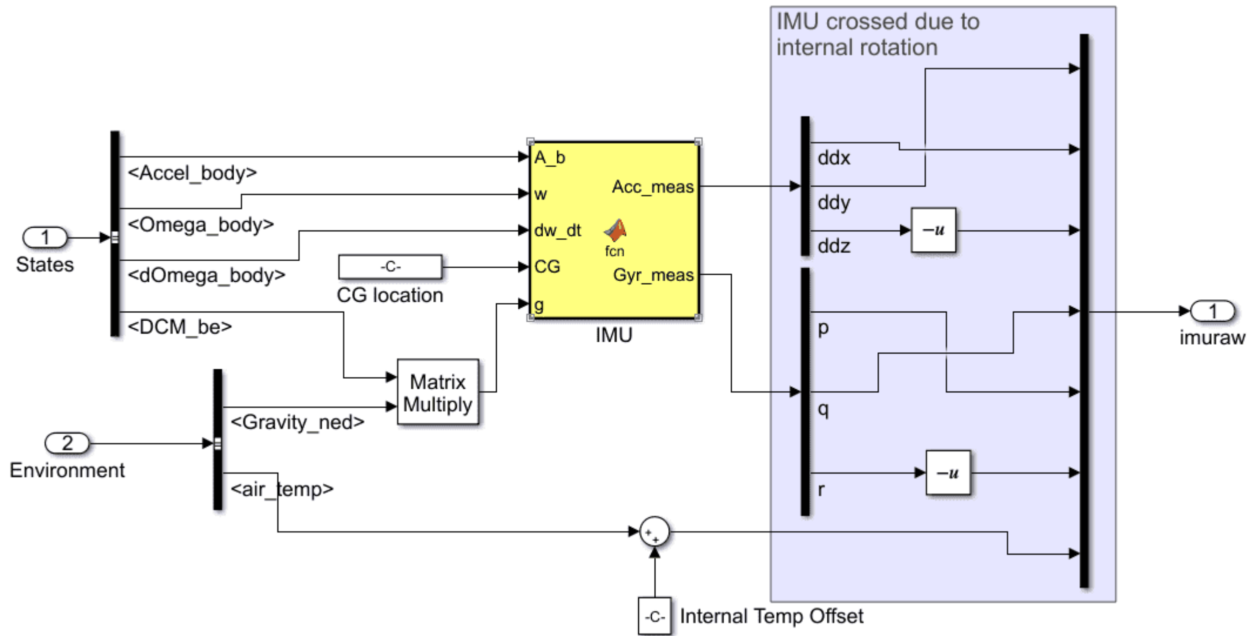


Fig. 13: **IMU - Subsystem**

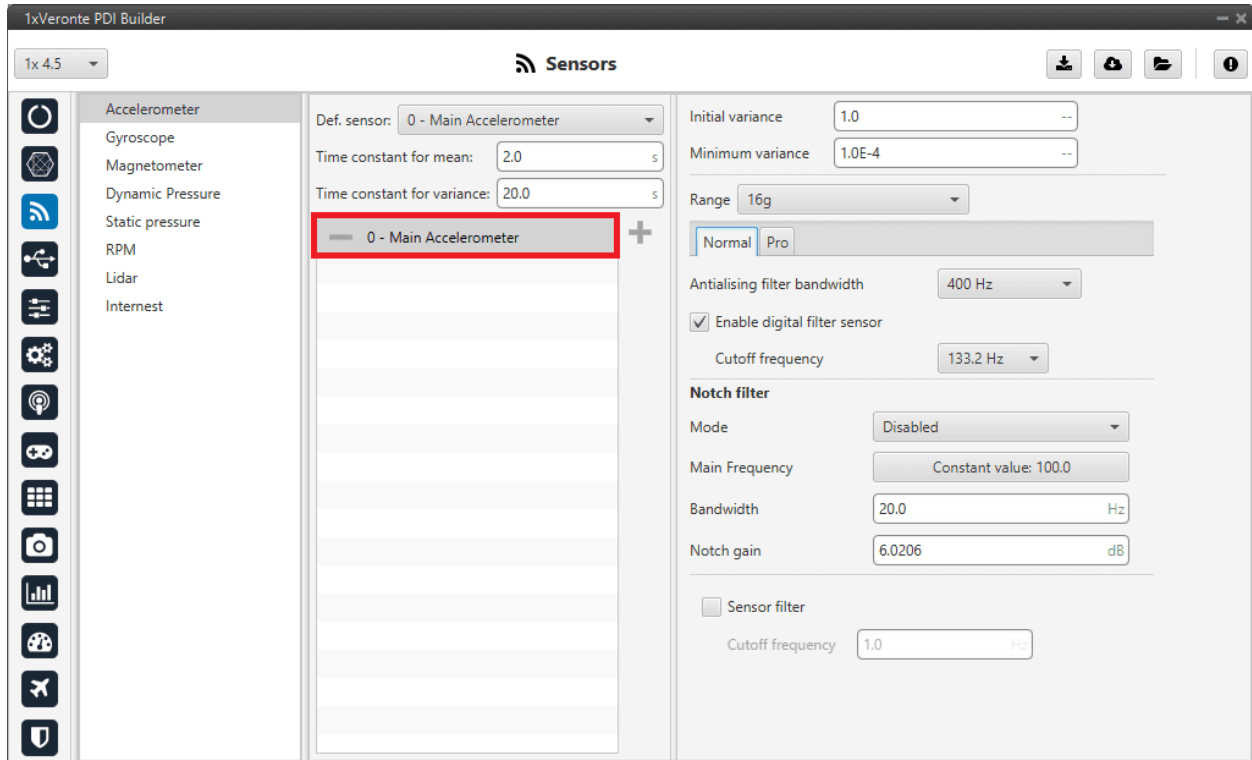


Fig. 14: **IMU - Accelerometer selected in 1x PDI Builder**

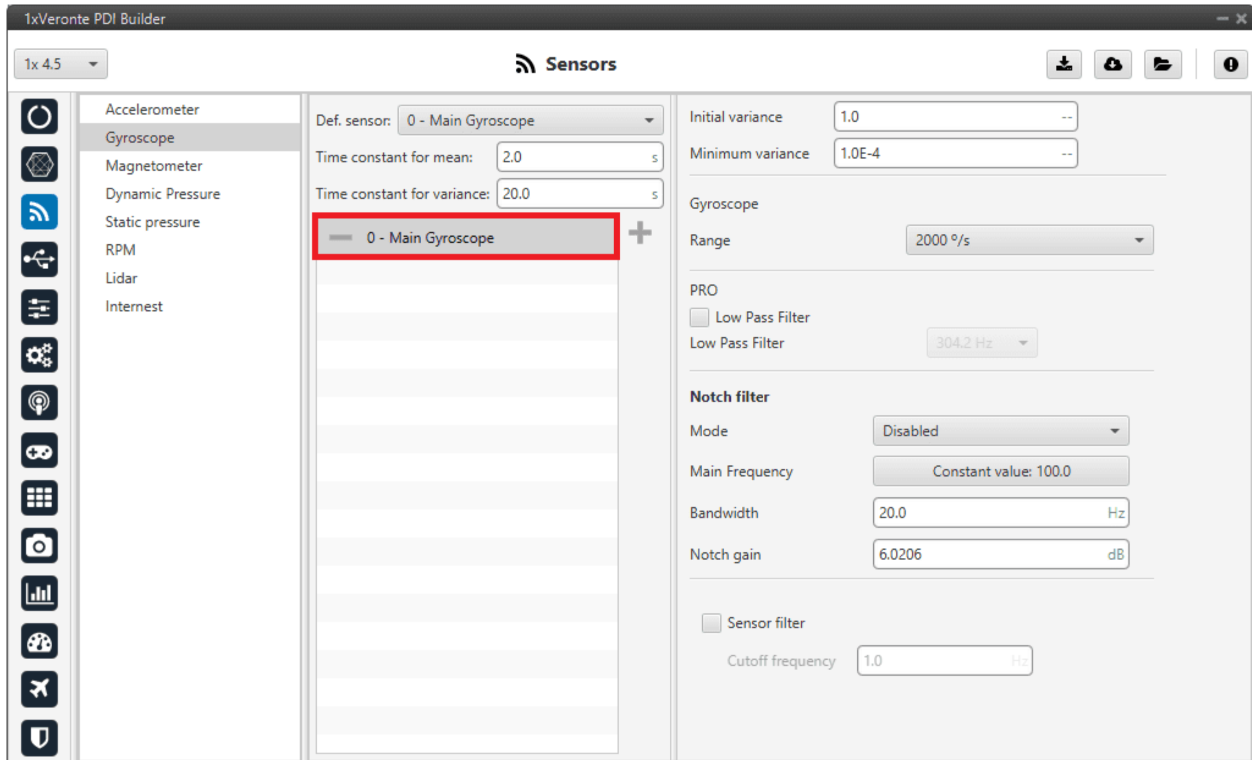


Fig. 15: **IMU - Gyroscope selected in 1x PDI Builder**

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

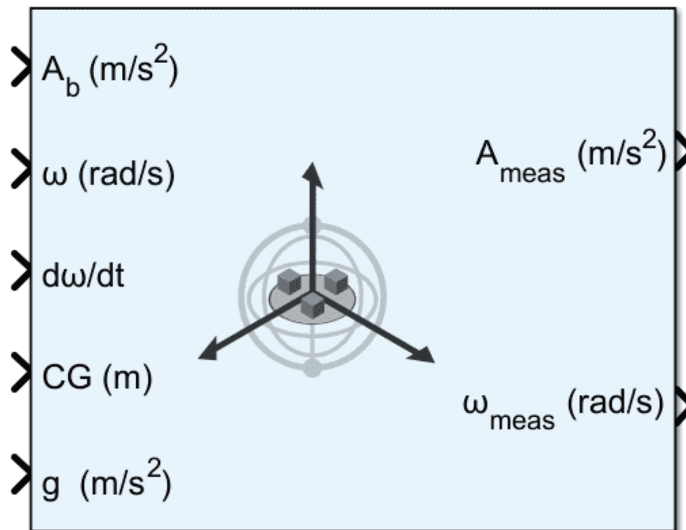


Fig. 16: **IMU - Aerospace Toolbox block**

- **Specific example from the example provided by Embention**

  The data in the example provided is already prepared for being introduced into the S-Function as the IMU 0 data,

that is the IMU0 sensor in hardware version `4.0`.

Therefore, if users want to use the data provided by Embention (simin_IMU0) and in their configuration with a `4.8` hwv they have the IMU 3 selected, 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:

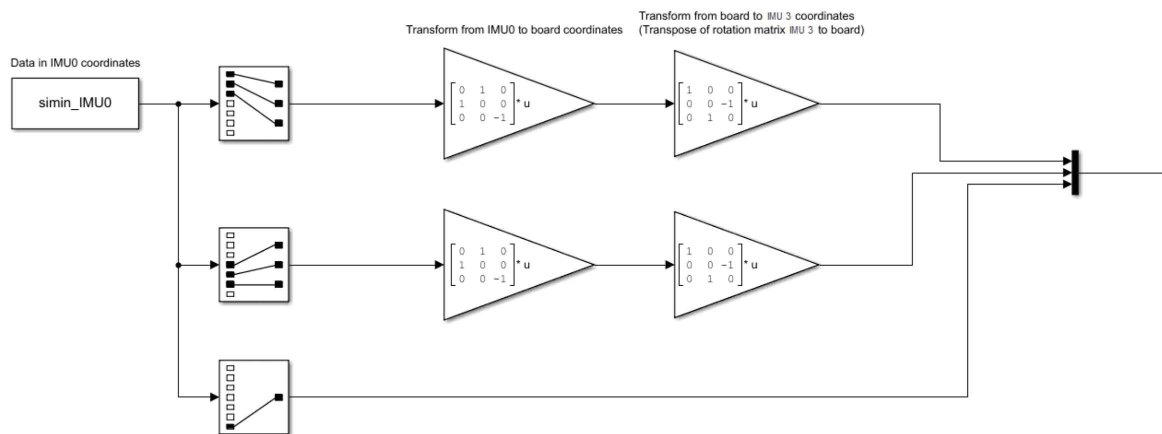


Fig. 17: **IMU - Example**

4.3.4 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	Magnetometer	Rotation Matrix
4.0	**MAG0**	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
4.5	**MAG0**	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	MAG1	$R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
4.8	**MAG0**	$R = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$
	MAG1	$R = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$
	MAG2	$R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Important: Please note that the number of inputs (ports) corresponds to the maximum number of inputs available on all hardware and SIL Simulator versions, as can be seen in the aforementioned table.

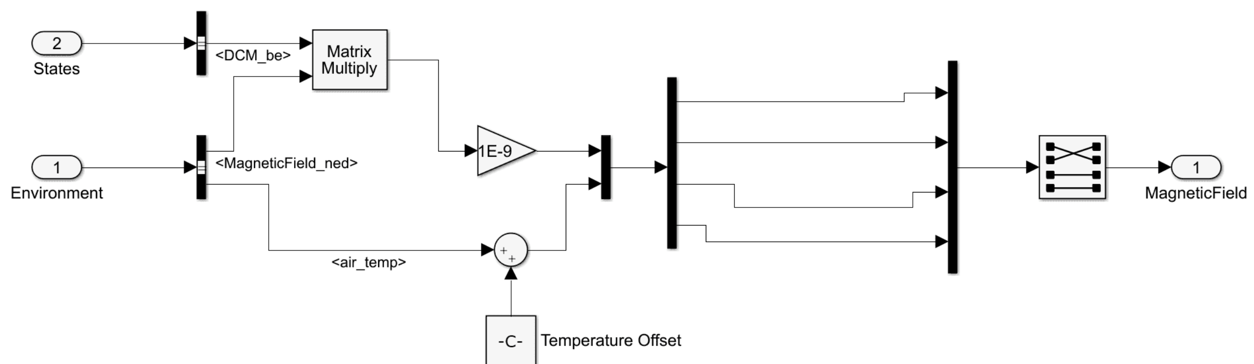


Fig. 18: **Magnetometer - Subsystem**

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.

4.3.5 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

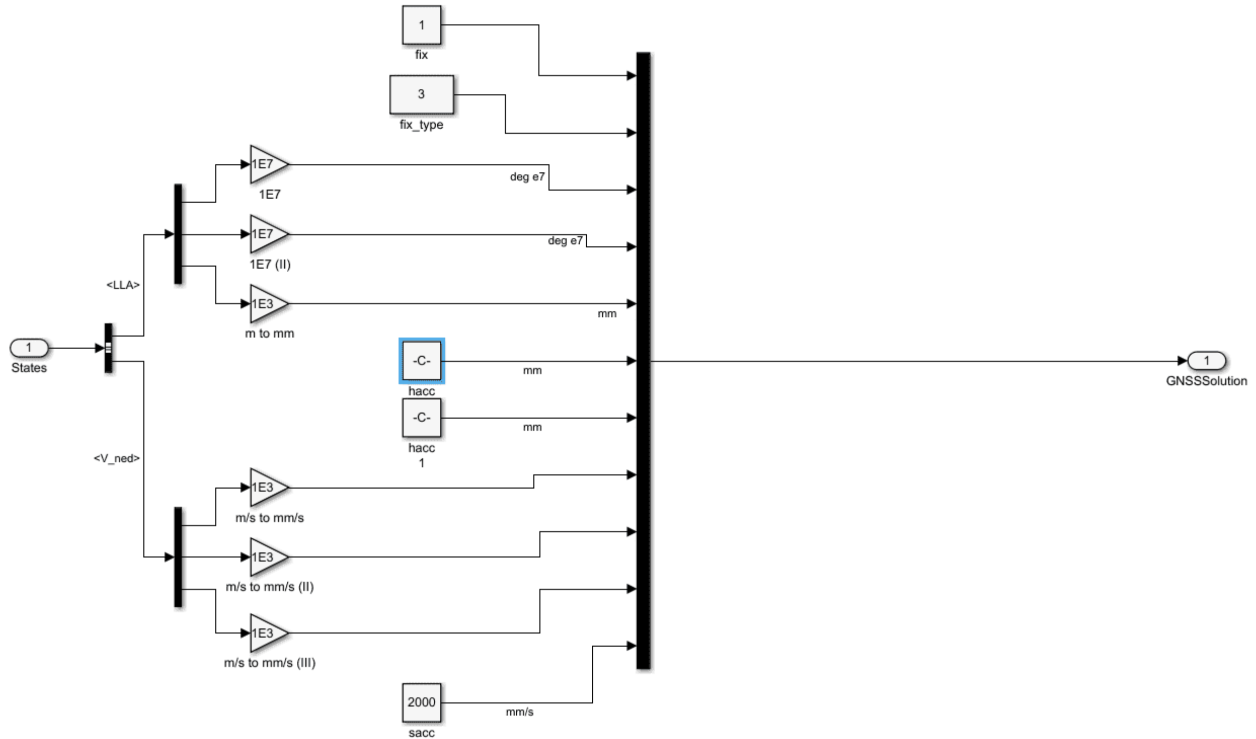


Fig. 19: **GNSS array**

Note:

- The **angle inputs** are in $degrees \cdot 10^7$ units.
- The **distance inputs** are in $millimetres$ units.
- The **speed inputs** are in $millimetres\ per\ second$ 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.

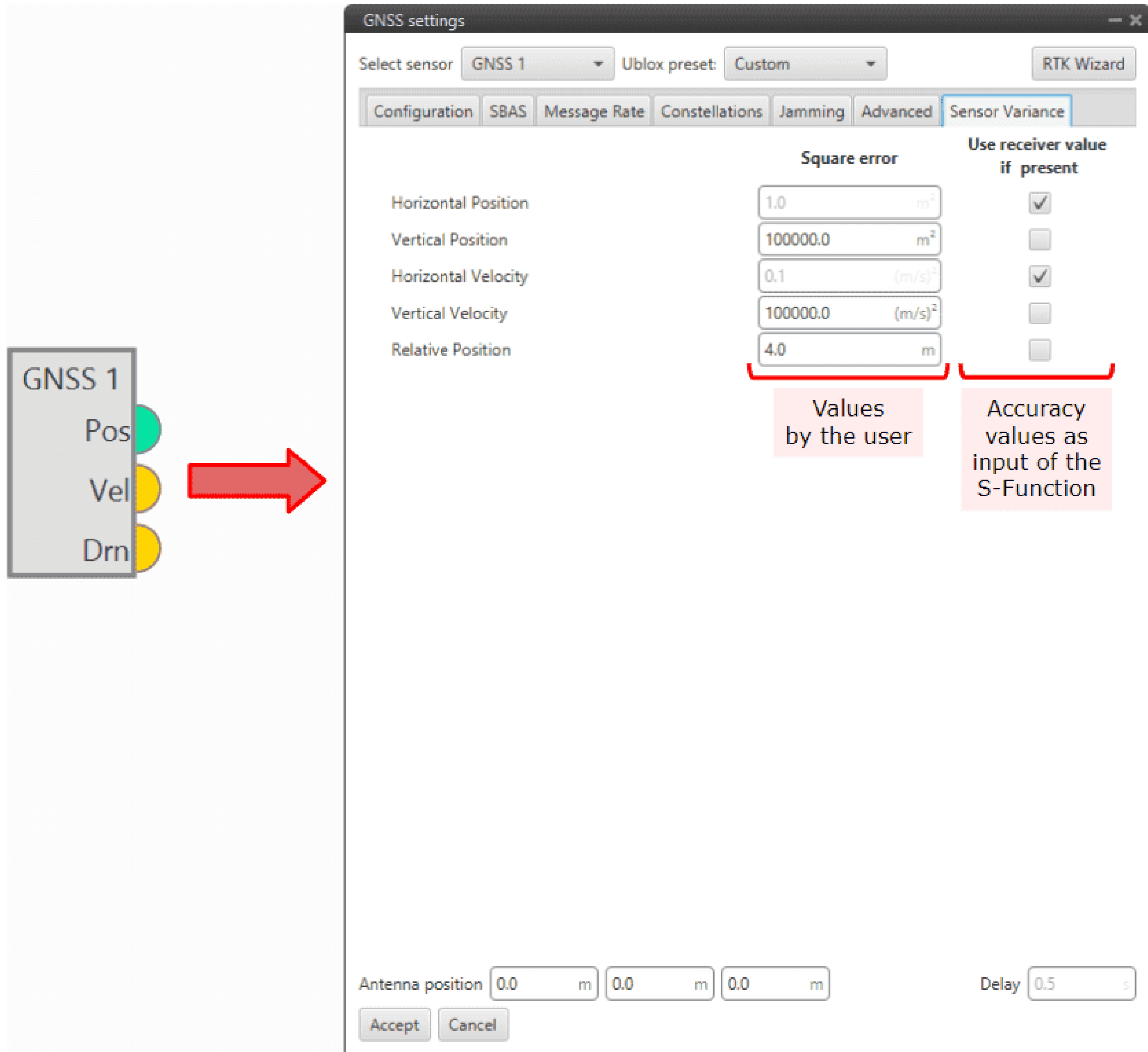


Fig. 20: **GNSS variances - 1x PDI Builder**

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)

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.

An example of this subgroup is shown below:

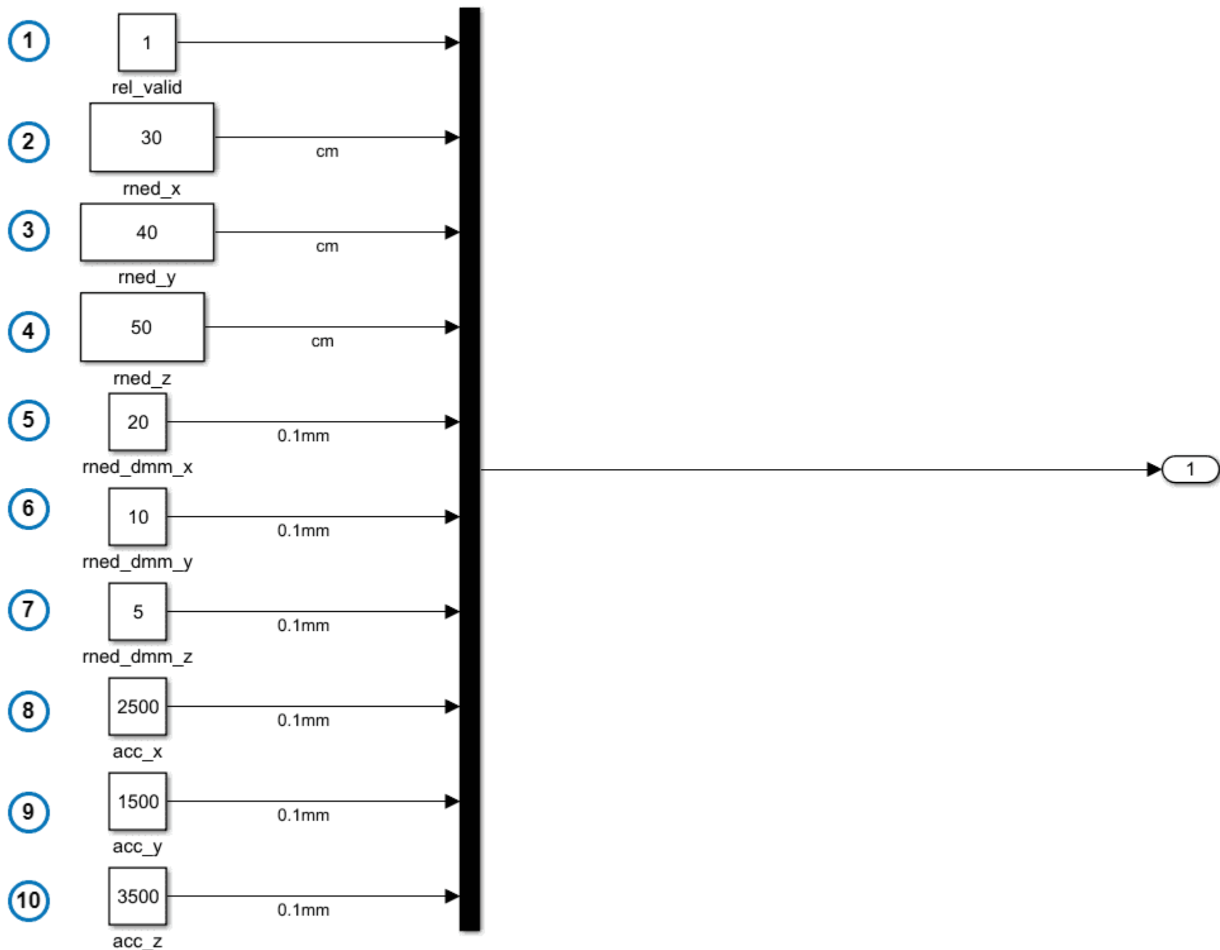


Fig. 21: **RTK inputs**

4.3.6 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):

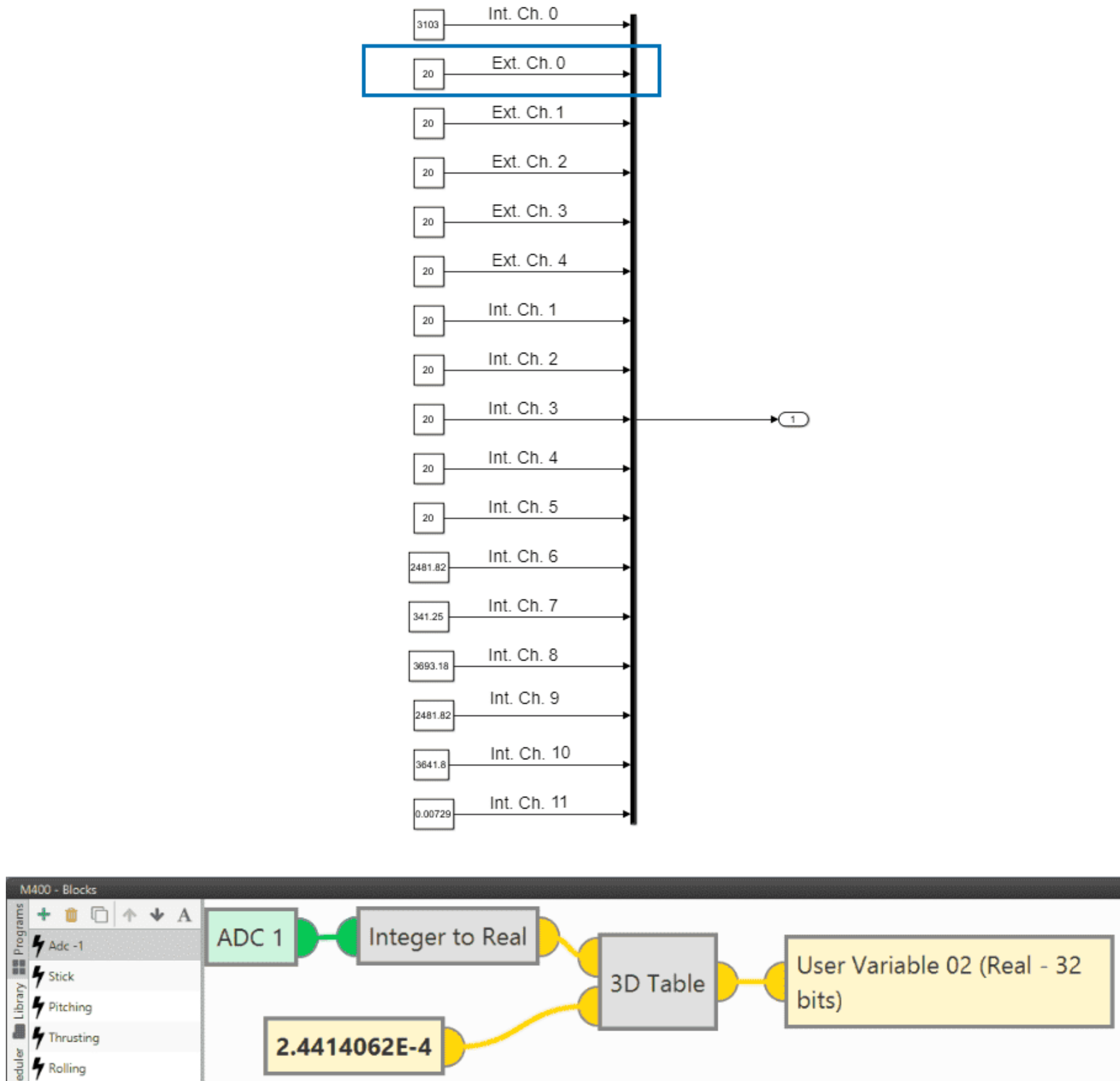


Fig. 22: **ADC readings**

4.3.7 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):

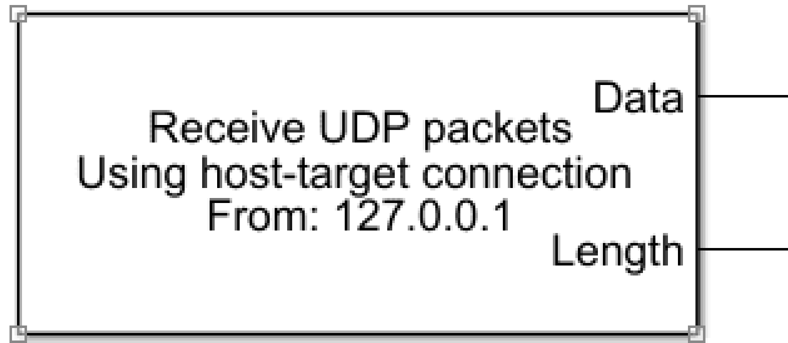


Fig. 23: **UDP Block**

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 neccesary 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:

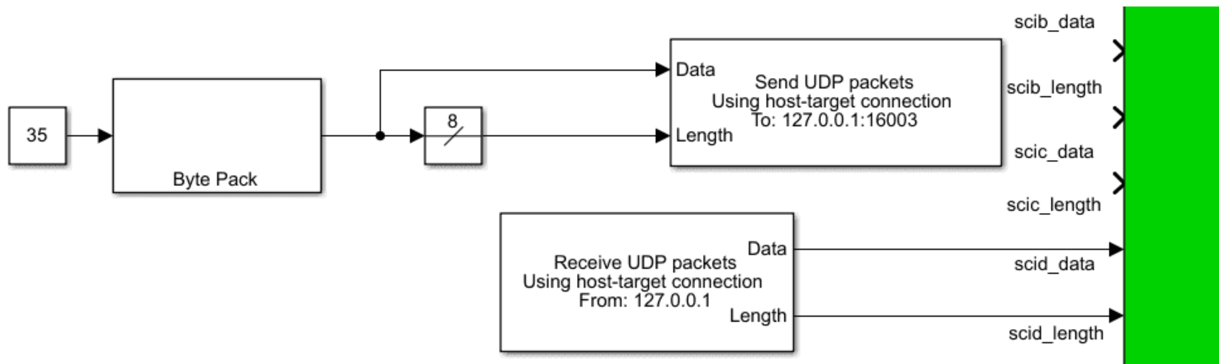


Fig. 24: **Sending a RS-232 message in Simulink**

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.

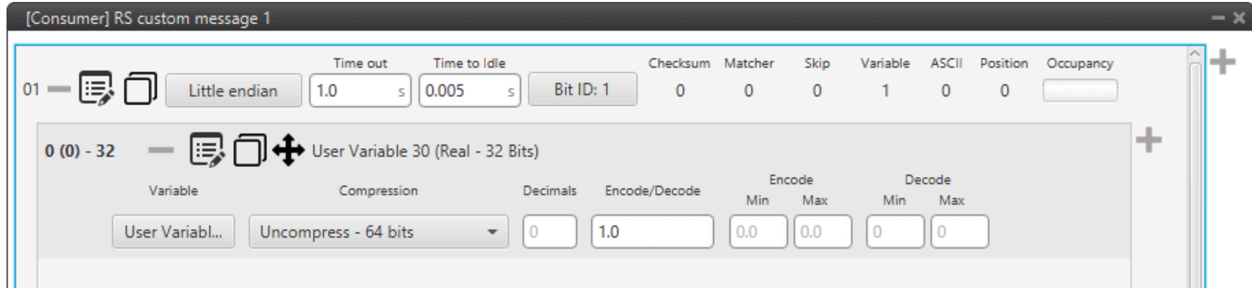


Fig. 25: **Custom message - 1x PDI Builder**

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

4.4 Telemetry

In the S-function there are 3 inputs specially dedicated to select custom temeletry (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 List of variables section of the **1x Software Manual**.

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

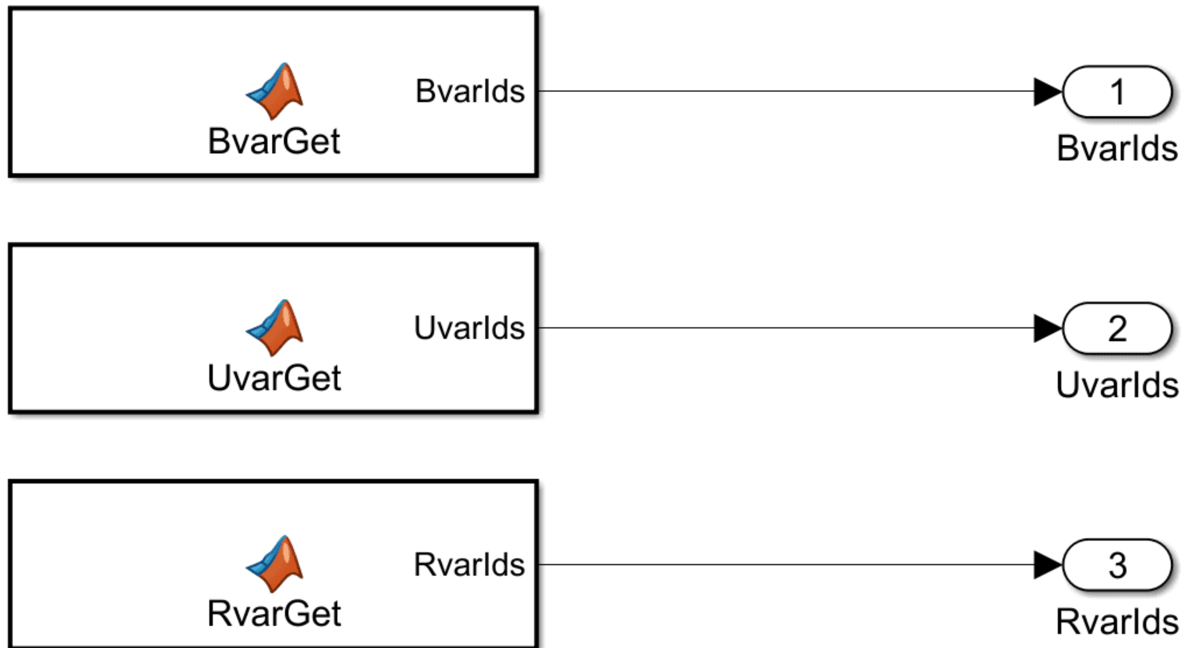


Fig. 26: **Telemetry blocks**

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

Requested Variables/MATLAB Function2*

```
function RvarIds = RvarGet()
% Requested Ids (row vector)
requestedIds = [...
    330:336,    ...    % IMU 0 Raw Measurements
    337:343,    ...    % IMU 1 Raw Measurements
    361:367,    ...    % IMU 2 Raw Measurements
    386:392,    ...    % IMU 3 Raw Measurements
    348:349,    ...    % Static Pressure 0 (HSC) Raw Measurements
    344:345,    ...    % Static Pressure 1 (MS56) Raw Measurements
    368:369,    ...    % Static Pressure 2 (DPS310) Raw Measurements
    322:325,    ...    % Internal Magnetometer LIS3MDL Raw Measurements
    370:373,    ...    % Internal Magnetometer MMC5883MA Raw Measurements
    393:396,    ...    % Internal Magnetometer RM3100 Raw Measurements
    346:347,    ...    % Dynamic Pressure Raw Measurement
    1504:1506,  ...    % GNSS 1 LLA
    1604:1606,  ...    % GNSS 2 LLA
    ];
RvarIds = [requestedIds];
```

Fig. 27: **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.).

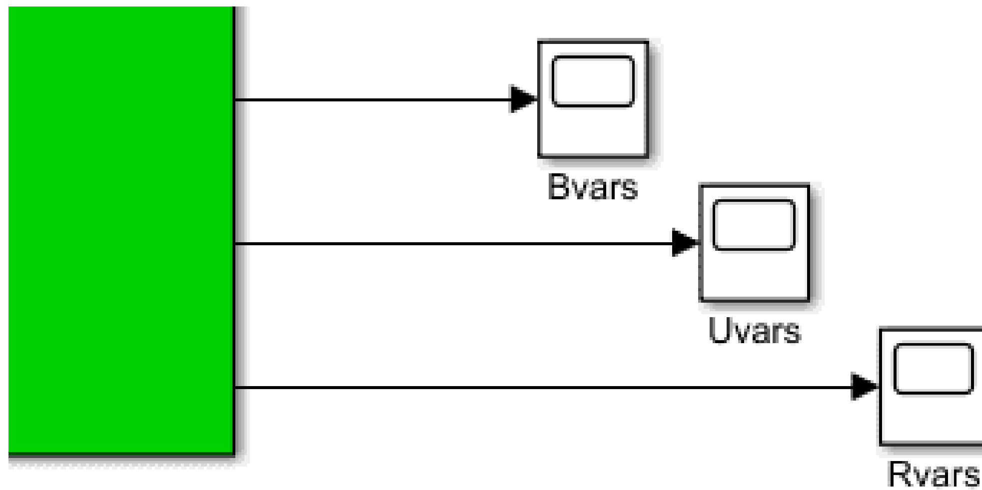


Fig. 28: **Display variables - Scope block example**

4.5 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 theses 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:

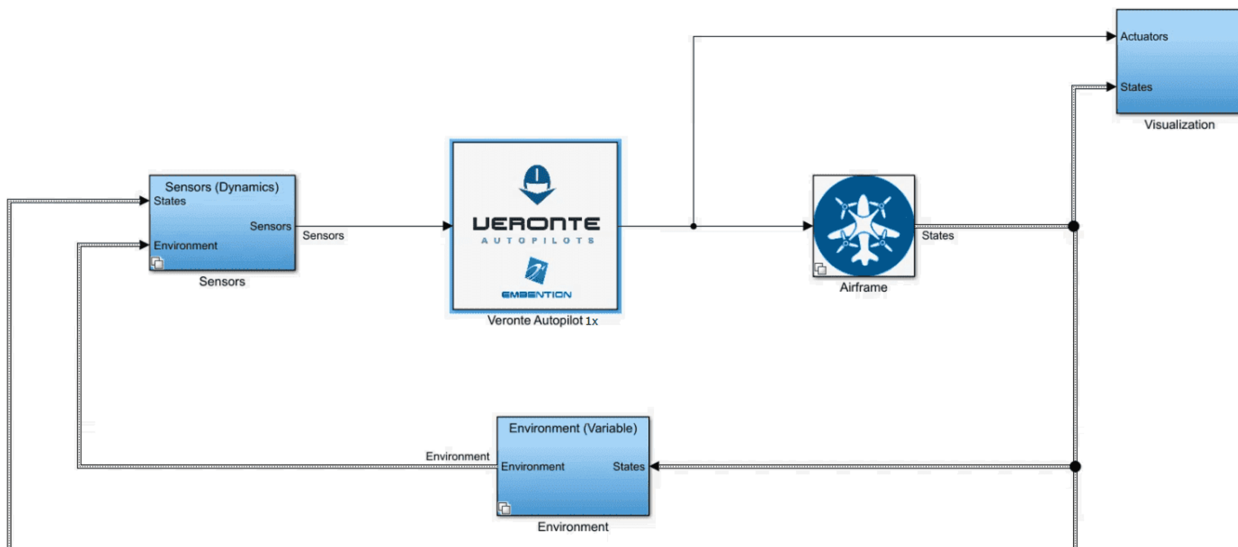


Fig. 29: **Complete Setup Example**

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.

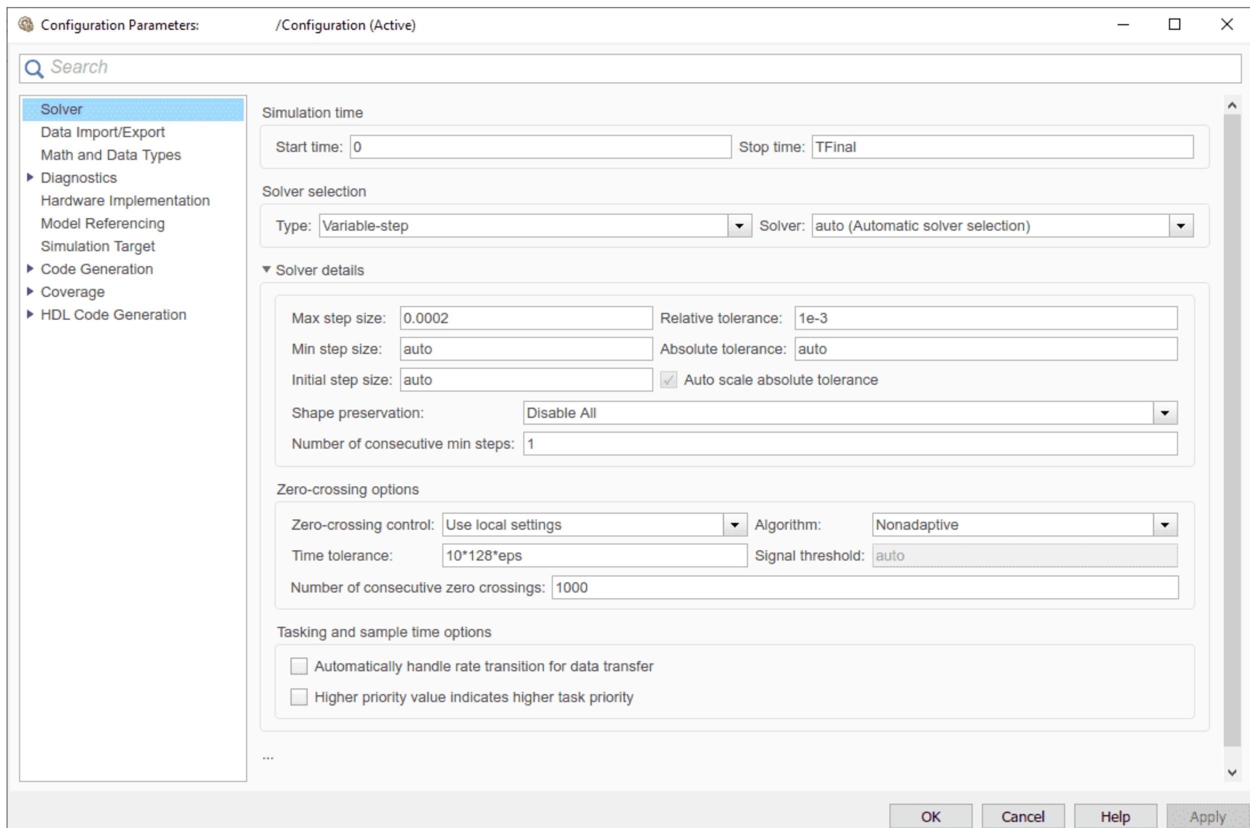


Fig. 30: **Time step settings**

CHAPTER

FIVE

TROUBLESHOOTING

5.1 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
```

5.2 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**.

5.3 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.

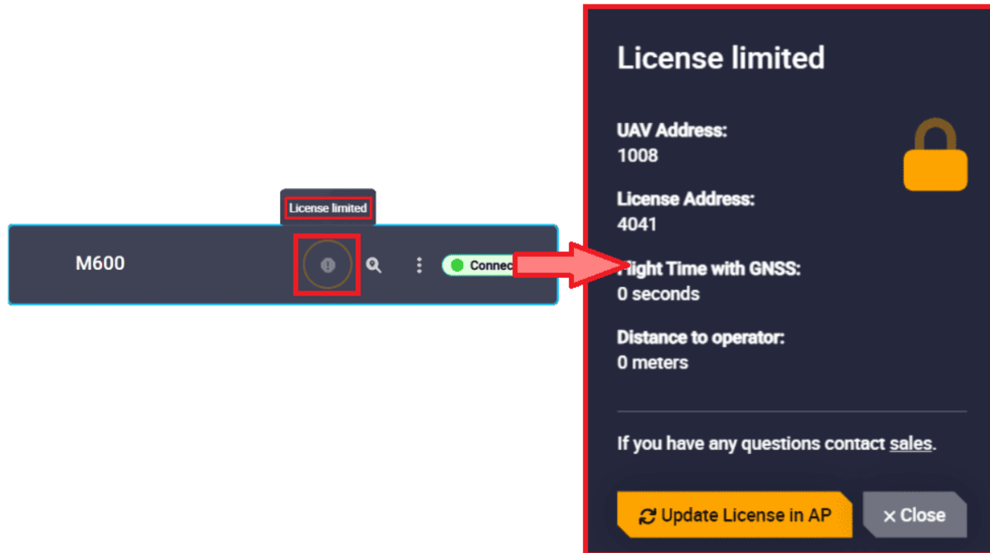


Fig. 1: **Veronte Ops - Platform license**

5.4 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.

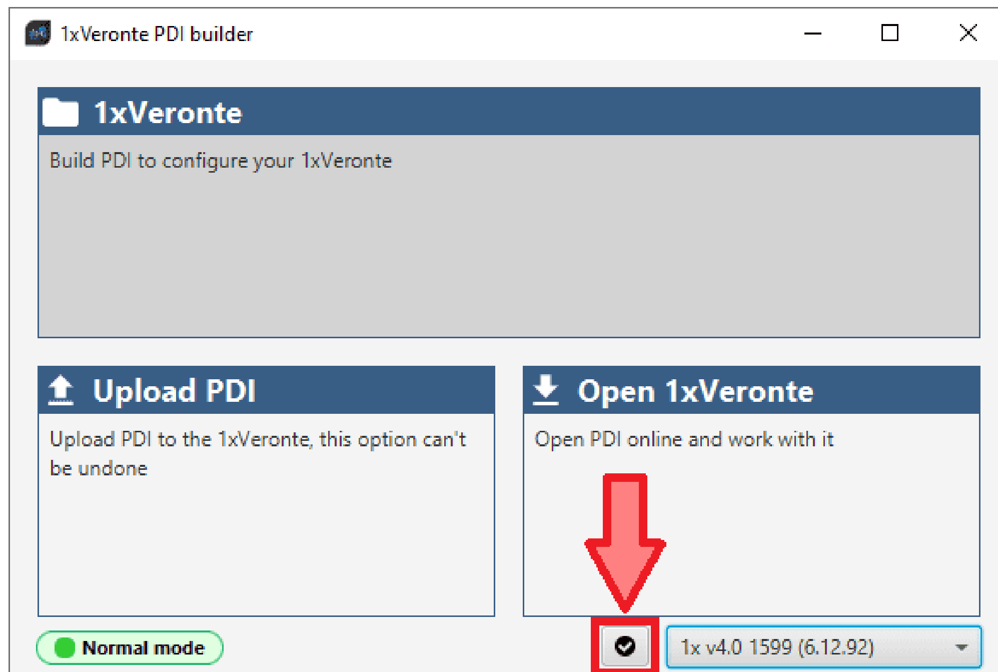


Fig. 2: **1x PDI Builder - PDI Errors button**

5.5 Running Veronte Console

In the case of using **VeronteConsole.exe**, it is also possible to pass the paths as 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
```

CHAPTER

SIX

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