SIL Simulator

Release 6.8.126

Embention

2023-08-25

CONTENTS

SIL Simulator allows running Software in the Loop simulations using Matlab and Simulink.

CHAPTER

ONE

INTRODUCTION

Veronte DLL

Veronte Autopilot 1x code. This is the code that simulates the physical 1x Autopilot firmware, i.e. it is like a 'virtual Veronte Autopilot 1x'.

This code can be **executed with Veronte Console or with Simulink blocks** (hereafter referred to as SIL Simulink). In addition, Veronte DLL can also be run with other languages, such as python, for the customer's desired use. But this would be done by the user, something similar to the Veronte Console.

Error: Do not run simultaneously with Veronte Console and SIL Simulink, because it will not work.

1.1 Veronte Console

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

However, at present, it has the disadvantage that inputs cannot be simulated through it.

1.2 SIL Simulink

A Software in the Loop simulation consists of a **Simulink model** that simulates the **behaviour of the system formed by 1x Autopilot and a vehicle**, **without having the physical devices connected to the computer**; unlike HIL Simulator which has both the autopilot and (optionally) the vehicle connected to the PC.

SIL has several advantages when compared to a HIL 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

SIL Simulator user manual includes a basic description of how Veronte Autopilot 1x works with Veronte Console and with Simulink blocks.

2.1 SIL zip files

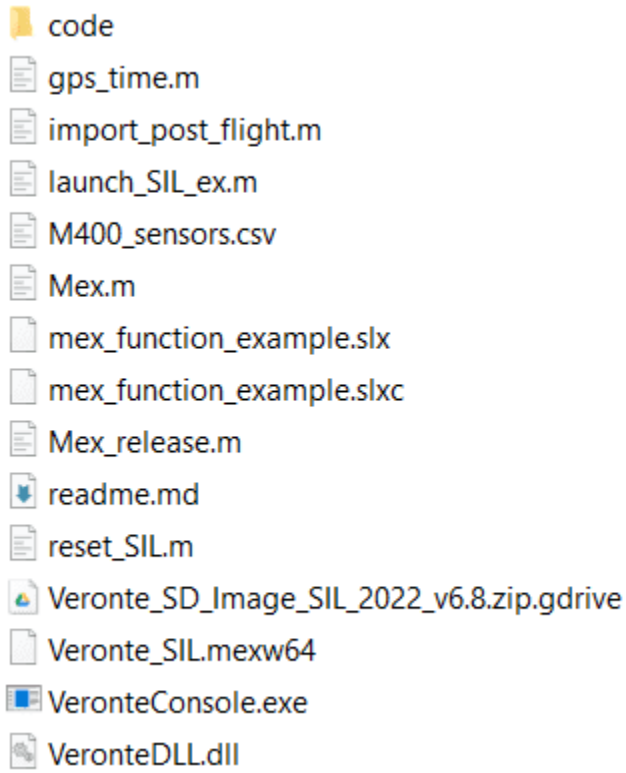

Fig. 1: **SIL zip files**

The basic SIL package consists of the followings files:

- **code** folder: Folder containing the code to access (compile) the Veronte DLL.
- **gps_time.m**: A matlab function that calculates the GPS/GNSS number of weeks.

- **import_post_flight.m**: A matlab script to **load an external source of inertial data (IMU)**. It reads this information from a **csv file**.
- **launch_SIL_ex.m**: Launches the `import_post_flight.m` and `mex_function_example.slx` files.
- **M400_sensors.csv**: Example of **csv file** to import with **import_post_flight.m**.
- **Mex.m**: Consists of all compiled embedded Veronte Autopilot 1x code.

  Compiles the code in `.cpp` contained in the **code** folder, and thereby creates the **simulink block**.
- **mex_function_example.slx**: Example of the simulink block, the **S-Function**. This is what the customer has to modify to create his own model.
- **mex_function_example.slxc**: File that simulink auto-generates from the `.slx` file.
- **readme.md**: Readme file to know how to work with SIL Simulator.
- **reset_SIL**: Script to clear mex related code in matlab to ensure a good reboot. If everything works as expected, it is executed at the end of every simulation, but it might be neccesary to execute it by hand if something goes wrong.
- **Veronte_SD_Image_SIL_2022_v6.8.zip.gdrive**: Link to Google Drive to **download the SIL Image**.

  In case of having any problem with permission, please contact the support team (create a ticket in the customer's **Joint Collaboration Framework**; for more information, see Tickets section of the JCF manual).
- **VeronteConsole.exe**: Veronte Console executable file.
- **VeronteDLL.dll**: File to be executed by Veronte Console or SIL Simulink.

  If this file is located in a folder other than this one, the user must create a file called "**dll_config.vcfg**" to specify the path. For more information on this, see *Connection section* of this manual.

2.2 Requirements

- **SIL image**: Veronte Autopilot 1x SD image downloadable from Guest FTP.
- **VeronteDLL**: Dynamic Link Library containing the Veronte AP code.
- Veronte Software Package:
  - **Veronte Link** (v6.8.X): Used to connect 1x Autopilot to the other tools.
  - **1x PDI Builder** (v6.8.X): To build and load PDIs.
  - **Veronte Ops** (v6.8): Operations interface.

SIL 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

CONNECTION

To stablish the connection between SIL Simulator and 1x PDI Builder to build/open a configuration, the following steps should be followed:

1. **Open Veronte Link** and configure it for a UDP connection. For more information, see the UDP connection -> Integration examples section of **Veronte Link** user manual.
    - If using **Veronte Console**, the UDP address needed to be configured in Veronte Link must be: **127.0.0.1**.
    - If **SIL Simulink** is used: The required address to be configured in **Veronte Link** must **match** the UDP **address specified in the Simulink Blocks**.

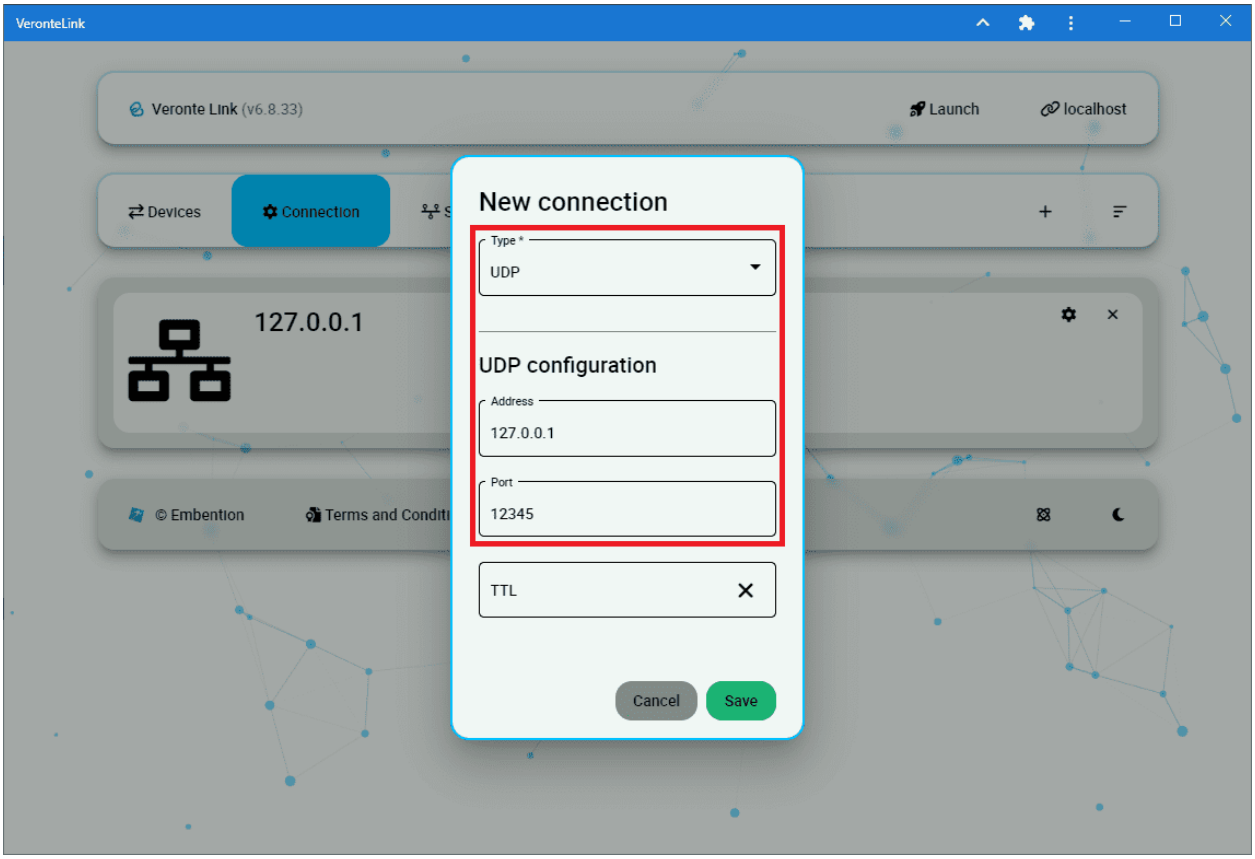

Fig. 1: **Veronte Link - UDP configuration for Veronte Console**

2. **Execution**: As explained previously, SIL Simulator can be executed in 2 different ways, through a standalone **Veronte Console Executable** or through **SIL Simulink**.

   Either way, both methods require:

   - **VeronteDLL.dll**: Dynamic Link Library containing the Veronte AP code.
   - **Veronte_SD_Image.img**: Veronte Autopilot 1x SD image.

   These **files** must be **placed in the root directory of SIL** (the unzipped SIL 6.8 folder).

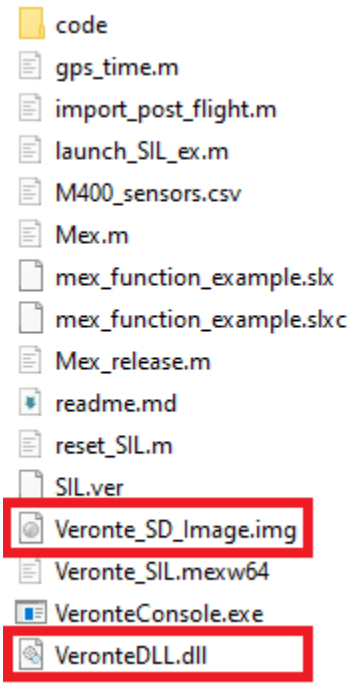

Fig. 2: **Image and Veronte DLL files in the unzipped SIL folder**

   If any of the above files are located in a directory other than the standard unzipped SIL 6.8 folder, it is **possible to specify a different path** by **creating** a file named "**dll_config.vcfg**" which **must be placed in the unzipped SIL 6.8 folder**.

   Within this file, the absolute path to the DLL and Image file must be provided as follows:

   `\dll C:\Users\user\Folder\VeronteDLL.dll`

   `\image C:\Users\user\Folder\Veronte_SD_Image.img`

- **VERONTE CONSOLE**:

   If the files are not located in the unzipped SIL folder, the path described above, can be passed 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`

   After this, when the user runs **VeroteConsole.exe**, if everything has been correctly configured, it should look like this:

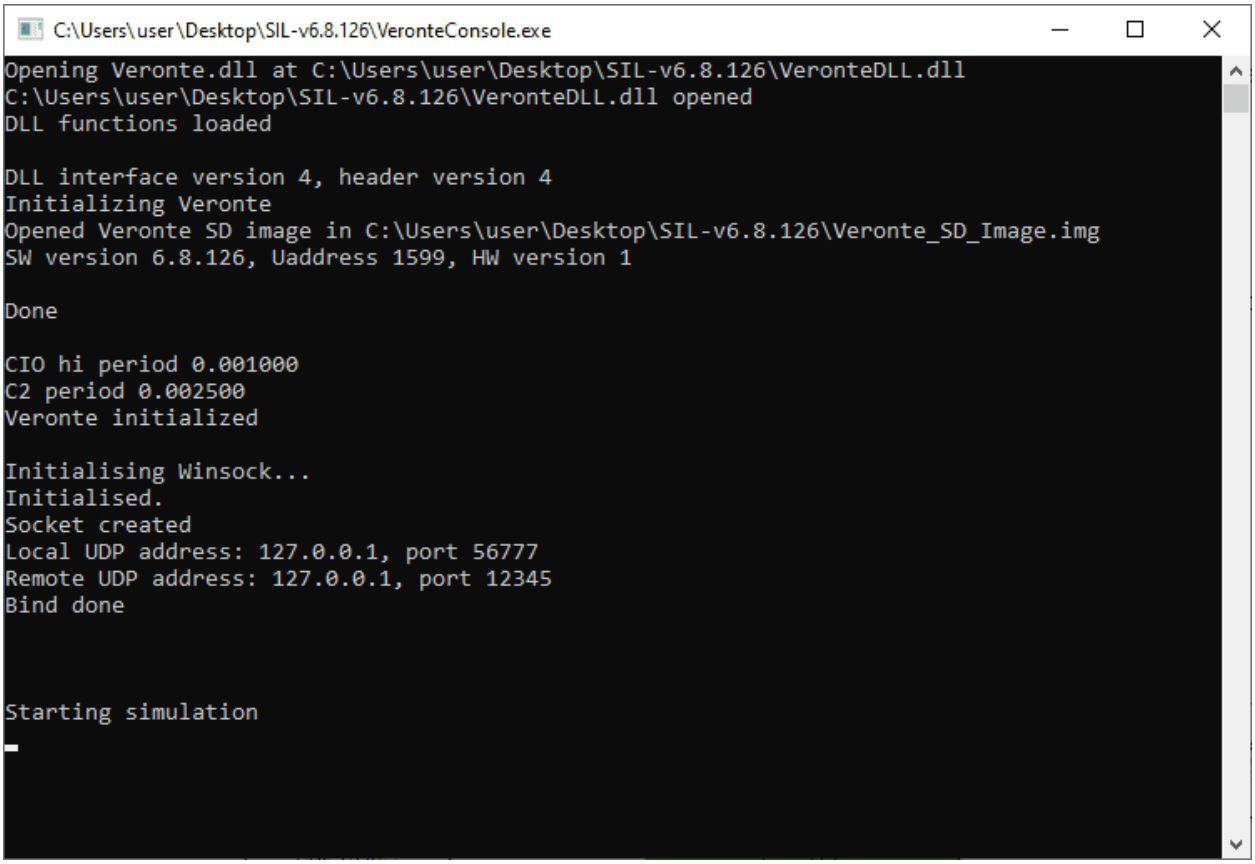

Fig. 3: **Veronte Console**

- **SIMULINK**:

  Create an **Ethernet connection** in **Simulink Blocks**:

  1. Add a UDP serial communication block and connect it to USB data and length.
  2. Add a second UDP serial communication block and connect it to the USB output of veronte.

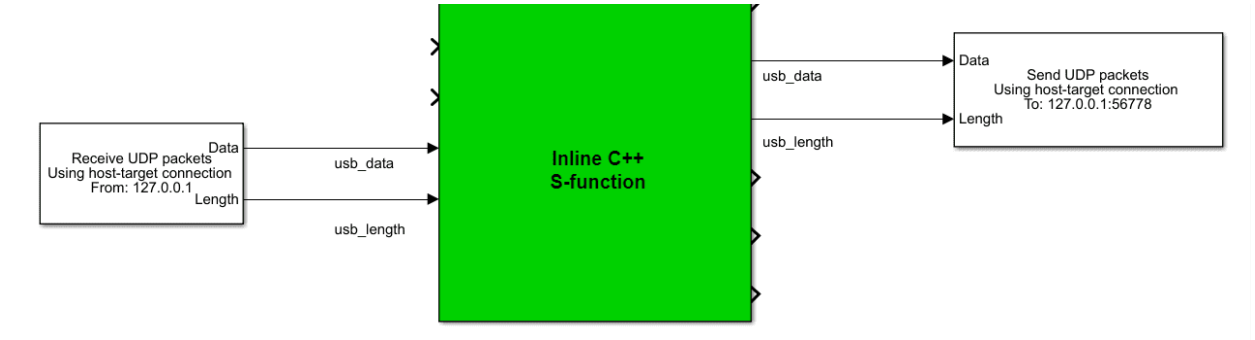

Fig. 4: **UDP Blocks**

  3. Configure the desired destination port.

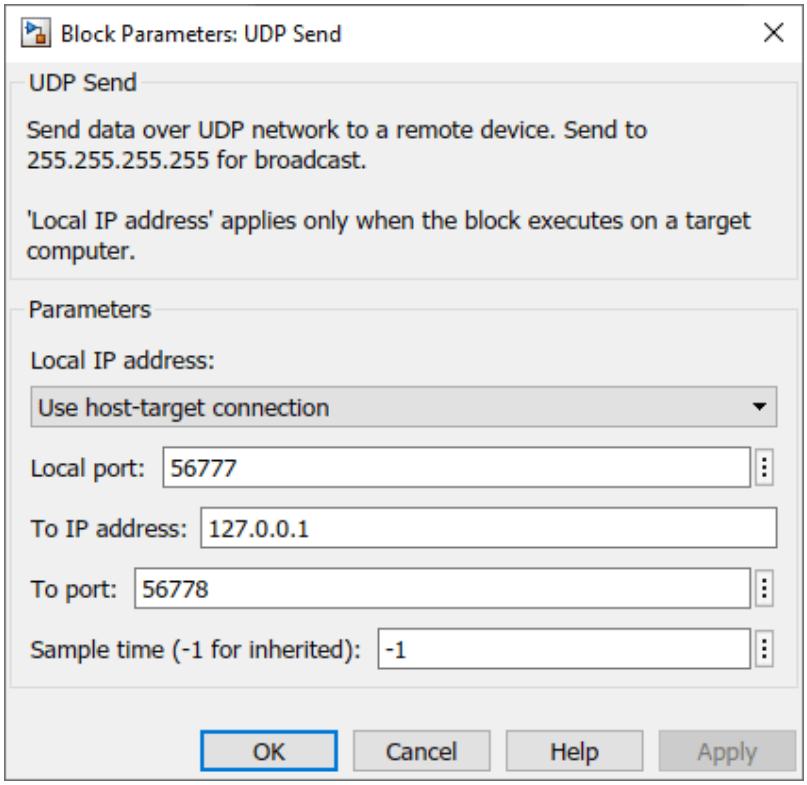

Fig. 5: **Destination UDP Port**

4. **Run the Simulink model** to simulate the Autopilot 1x information.

3. Check that the autopilot appears in Veronte Link as **Connected** and **Ready**:

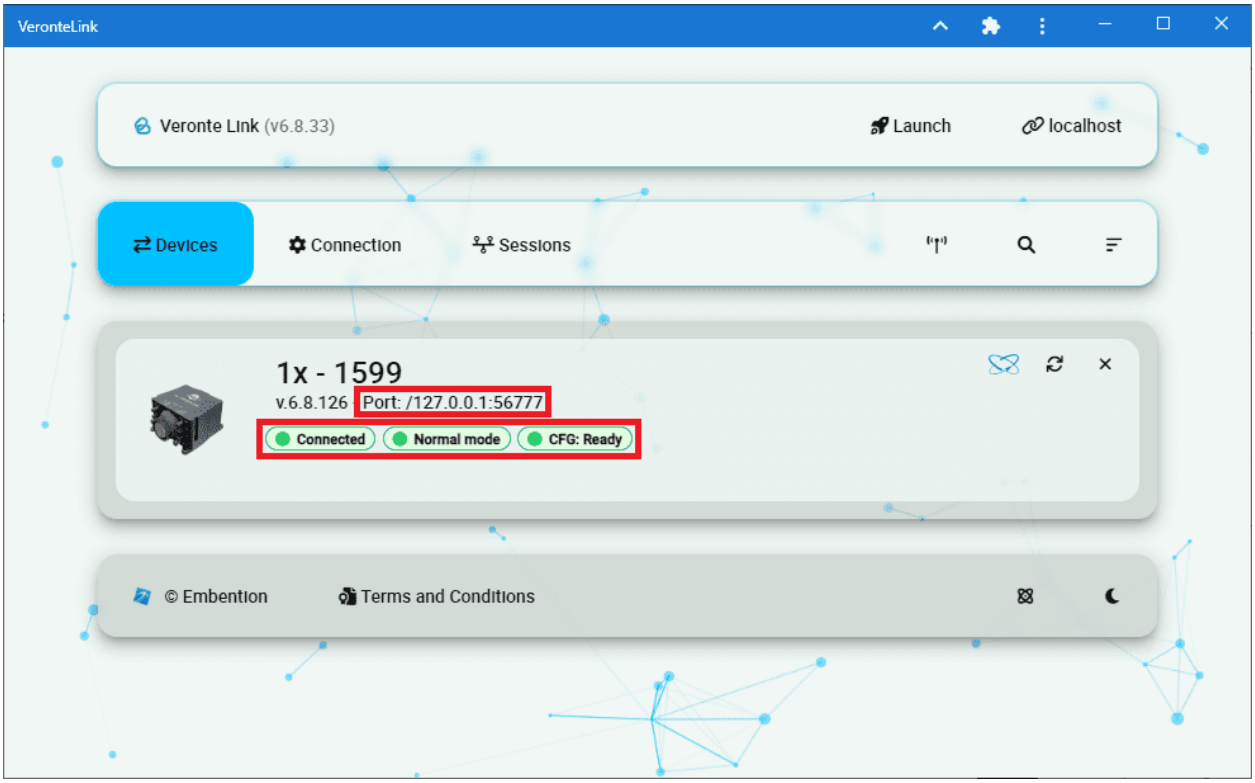

Fig. 6: **Veronte Link**

Important: In the **first generation of the Veronte Autopilot 1x**, it is possible that its status is: '**Maintenance Mode**' and '**CFG: Failed to load**'.

This is because the **Autopilot PDI has no productionLine calibration**. This is fixed by **uploading the SIL.ver file** through 1x PDI builder.

The calibration uploaded through the SIL.ver file is **saved in the Veronte_SD_Image.img**, so there is **no need to upload this file again**.

---

4. **Open 1x PDI Builder** and select the 1x Autopilot. For more information on this software, see 1x PDI Builder user manual.

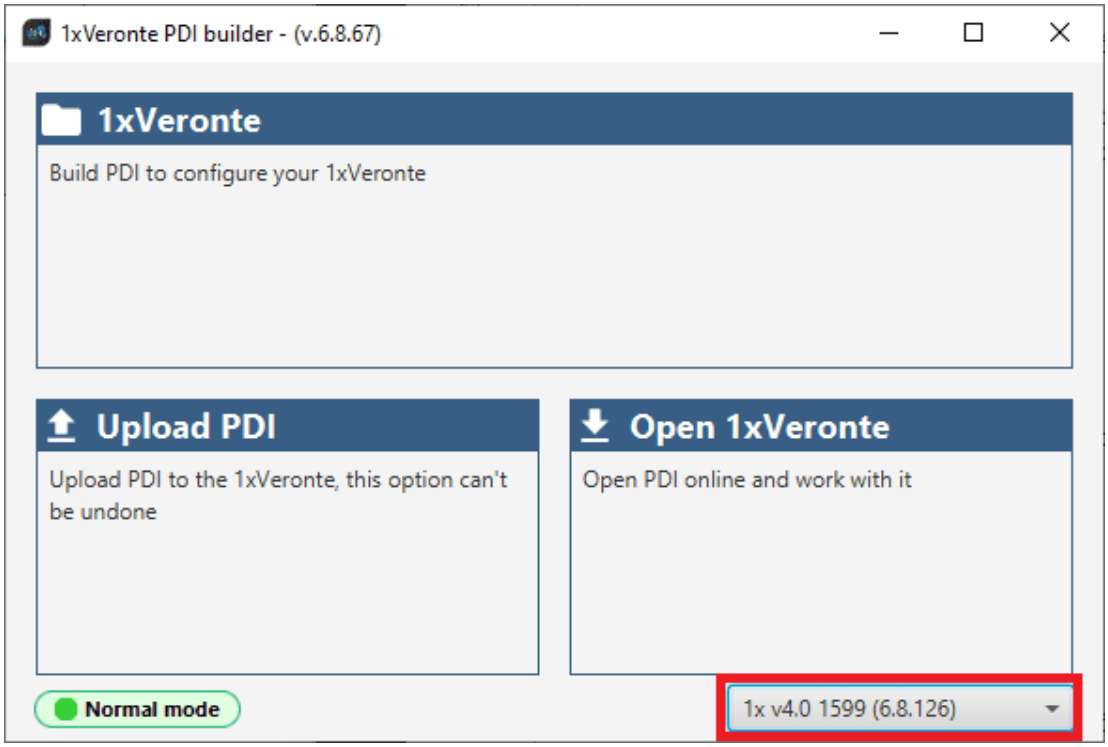

Fig. 7: **1x PDI builder**

Warning: In this version it is **not possible to upload the configuration (PDI files) while running the Veronte DLL with SIL Simulink**.

Therefore, if the user is using SIL Simulink for the simulation, he/she must switch to Veronte Console execution to upload the configuration, and then switch back to running SIL Simulink.

CHAPTER

FOUR

SIL SIMULINK

The Veronte Autopilot 1x is implemented in Simulink blocks with an **S-Function**.

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. A typical 1x Autopilot S-Function is shown below.

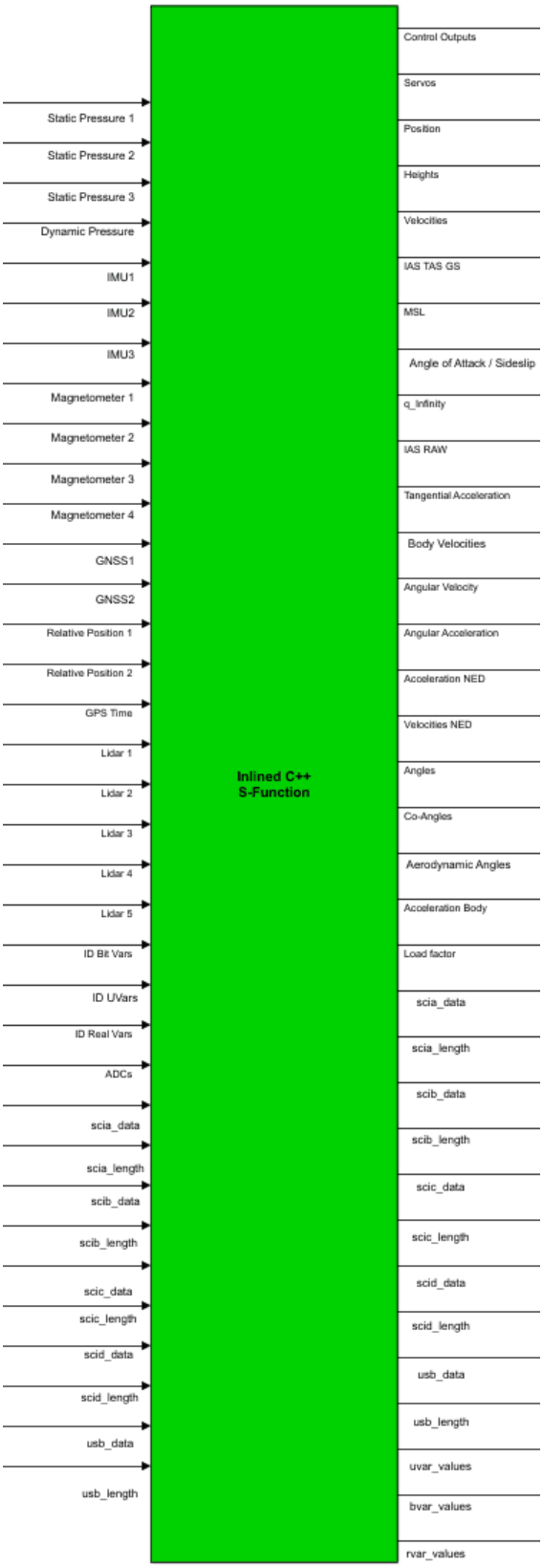

Fig. 1: **S-Function containing the autopilot embedded code**

4.1 Inputs

Inputs are described in the next table:

PIN	Signal Type	Description	Form	Size	Units
1	Input	Static Pressure 1	[pressure_measurement;sensor temperature]	2x1	Pa ; K
2	Input	Static Pressure 2	[pressure_measurement;sensor temperature]	2x1	Pa ; K
3	Input	Static Pressure 3	[pressure_measurement;sensor temperature]	2x1	Pa ; K
4	Input	Dynamic Pressure	[pressure_measurement;sensor temperature]	2x1	Pa ; K
5	Input	IMU 1	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$; K
6	Input	IMU 2	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$; K
7	Input	IMU 3	[acc_x;acc_y;acc_z;gyr_x;gyr_y;gyr_z;sensor temperature]	7x1	$\frac{m}{s^2}$; $\frac{rad}{s}$; K
8	Input	Magnetometer 1	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
9	Input	Magnetometer 2	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
10	Input	Magnetometer 3	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
11	Input	Magnetometer 4	[mag_x;mag_y;mag_z;sensor temperature]	4x1	T ; K
12	Input	GNSS 1	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]	11x1	$deg \cdot 10^7$; mm
13	Input	GNSS 2	[1;3;lon;lat;alt;hr_accu;vt_accu;v_n;v_e;v_d;v_accu]	11x1	$deg \cdot 10^7$; mm
14	Input	Relative Position 1	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]	10x1	cm ; $mm \cdot 10^-$
15	Input	Relative Position 2	[1;x_rel;y_rel;z_rel;d_x;d_y;d_z;x_accu;y_accu;z_accu]	10x1	cm ; $mm \cdot 10^-$
16	Input	GPS Time	[week_number;milliseconds_of_week]	2x1	-; ms
17	Input	Lidar 1	[lidar_measurement]	1x1	cm
18	Input	Lidar 2	[lidar_measurement]	1x1	cm
19	Input	Lidar 3	[lidar_measurement]	1x1	cm
20	Input	Lidar 4	[lidar_measurement]	1x1	cm
21	Input	Lidar 5	[lidar_measurement]	1x1	cm
22	Input	ID Bit Var	[Var_IDs]	50x1	-
23	Input	ID Unsigned Var	[Var_IDs]	50x1	-
24	Input	ID Real Var	[Var_IDs]	50x1	-
25	Input	ADCs	[adc(1-17)]	17x1	-
26	Input	SCIA Data	[serial_data]	1024x1	-
27	Input	SCIA Length	[serial_length]	1x1	-
28	Input	SCIB Data	[serial_data]	1024x1	-
29	Input	SCIB Length	[serial_length]	1x1	-
30	Input	SCIC Data	[serial_data]	1024x1	-
31	Input	SCIC Length	[serial_length]	1x1	-
32	Input	SCID Data	[serial_data]	1024x1	-
33	Input	SCID Length	[serial_length]	1x1	-
34	Input	USB Data	[serial_data]	1024x1	-
35	Input	USB Length	[serial_length]	1x1	-

4.2 Outputs

Outputs are the following:

PIN	Signal Type	Description	Form	Size	Units
1	Output	Control Outputs	[control_outputs(1-20)]	20x1	-
2	Output	Servo Values	[servos(1-32)]	32x1	-
3	Output	Position	[lon;lat;alt]	3x1	rad ; m
4	Output	Heights	[msl,agl]	2x1	m
5	Output	Velocities	[longitudinal_v;lateral_v;velocity(module)]	3x1	$\frac{m}{s}$
6	Output	IAS TAS GS	[ias,tas,gs]	3x1	$\frac{m}{s}$
7	Output	MSL	[msl_from_qnh;msl_from_ISA]	2x1	m
8	Output	Angle of Attack / Sideslip	[angle_of_attack;sideslip]	2x1	rad
9	Output	Q_Infinty	[dynamic_pressure]	1x1	Pa
10	Output	IAS RAW	[ias_raw]	1x1	$\frac{m}{s}$
11	Output	Tangential Acceleration	[tangential_acceleration]	1x1	$\frac{m}{s^2}$
12	Input	Body Velocities	[longitudinal_v;lateral_v;vertical_v]	3x1	$\frac{m}{s}$
13	Output	Angular Velocities	[roll_rate;pitch_rate;yaw_rate]	3x1	$\frac{rad}{s}$
14	Output	Angular Acceleration	[acc_z_axis;acc_y_axis;acc_x_axis]	3x1	$\frac{rad}{s}$
15	Output	Acceleration NED	[acc_north;acc_east;acc_down]	3x1	$\frac{m}{s^2}$
16	Output	Velocity NED	[v_north;v_east;v_down]	3x1	$\frac{m}{s}$
17	Output	Angles	[Yaw;Pitch;Roll]	3x1	rad
18	Output	Co-Angles	[co-Yaw;co-Pitch;co-Roll]	3x1	rad
19	Output	Aerodynamic Angles	[heading,flight_path;bank_angle]	3x1	rad
20	Output	Acceleration Body	[acc_x,acc_y;acc_z]	3x1	$\frac{m}{s^2}$
21	Output	Load factor	[nx;ny;nz]	3x1	-
22	Output	SCIA Data	[serial_data]	1024x1	-
23	Output	SCIA Length	[serial_length]	1x1	-
24	Output	SCIB Data	[serial_data]	1024x1	-
25	Output	SCIB Length	[serial_length]	1x1	-
26	Output	SCIC Data	[serial_data]	1024x1	-
27	Output	SCIC Length	[serial_length]	1x1	-
28	Output	SCID Data	[serial_data]	1024x1	-
29	Output	SCID Length	[serial_length]	1x1	-
30	Output	USB Data	[serial_data]	1024x1	-
31	Output	USB Length	[serial_length])	1x1	-
32	Output	Unsigned Variables	[selected variables(1-50)]	50x1	-
33	Output	Bit Variables	[selected variables(1-50)]	50x1	-
34	Output	Real Variables	[selected variables(1-50)]	50x1	-

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*.

4.2.1 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:

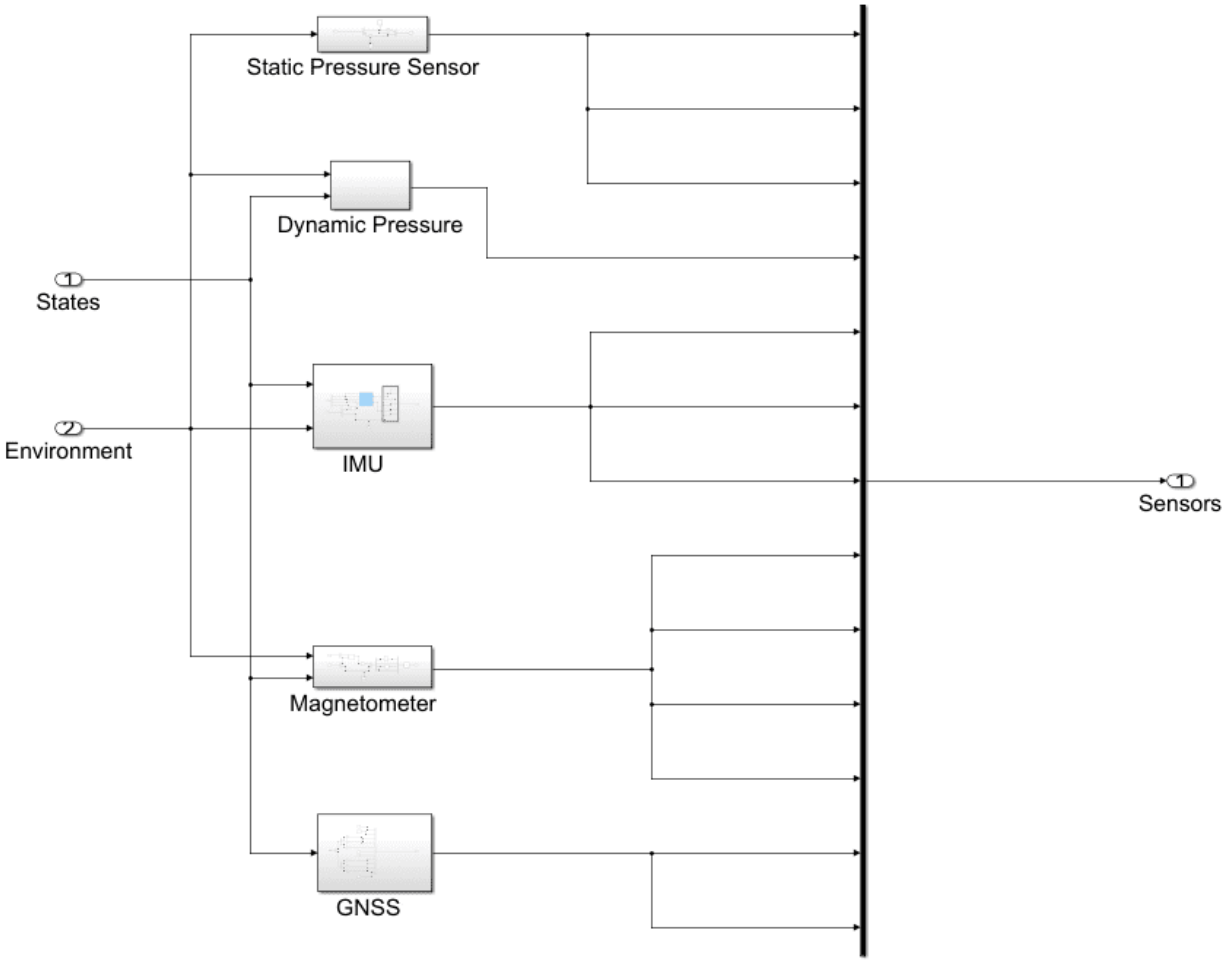

Fig. 2: **Sensors inputs**

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

- *Environment*
- *Pressure sensors*

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

4.2.1.1 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 a simple and constant model or modify 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.

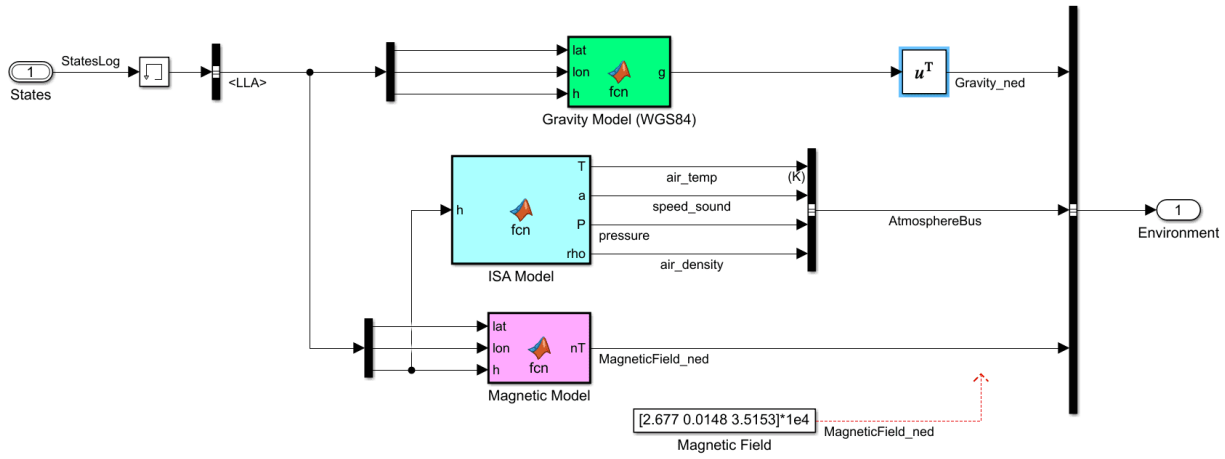

Fig. 3: **Environment block**

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

1. World Magnetic Model 2020:

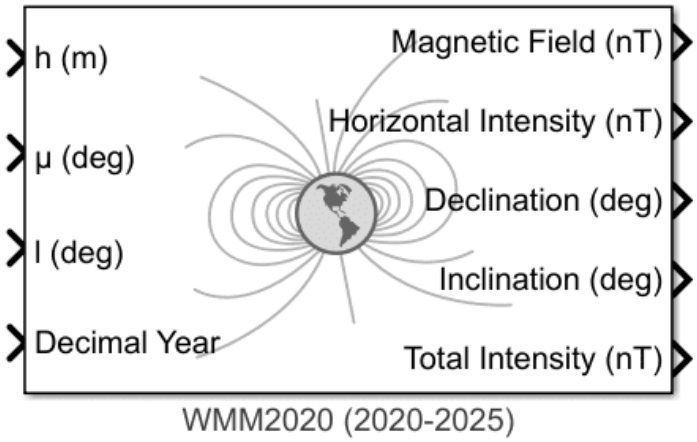

Fig. 4: **Aerospace blockset function - WMM2020**

2. ISA Atmosphere Model:

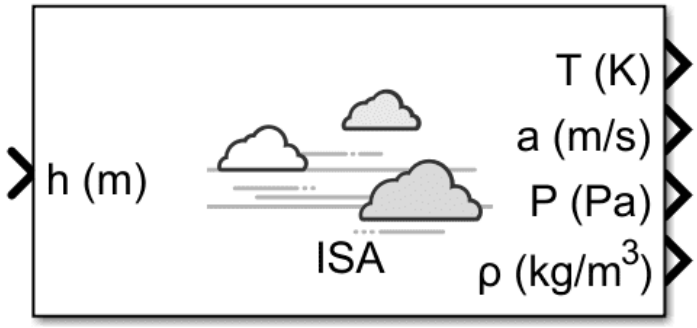

Fig. 5: **Aerospace blockset function - ISA Atmosphere Model**

3. WGS84 Gravity Model:

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

4.2.1.2 Pressure sensors

4.2.1.2.1 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	Internal Sensor (HSC)
	Internal Sensor (MS56)
4.5	Internal Sensor (HSC)
	Internal Sensor (MS56)
	Internal Sensor (DPS310)
4.8	Internal Sensor (HSC)
	Internal Sensor (MS56)
	Internal Sensor (DPS310)

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.

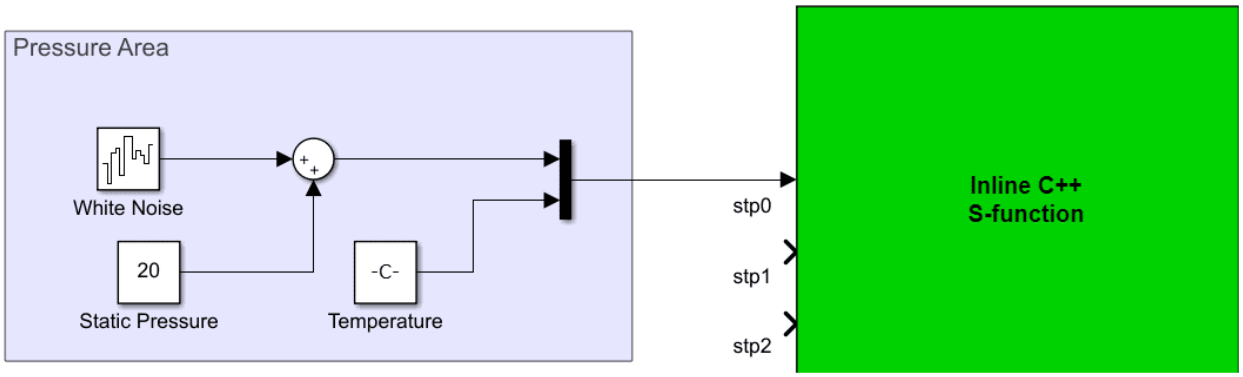

Fig. 7: **Static pressure - Constant**

- **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.

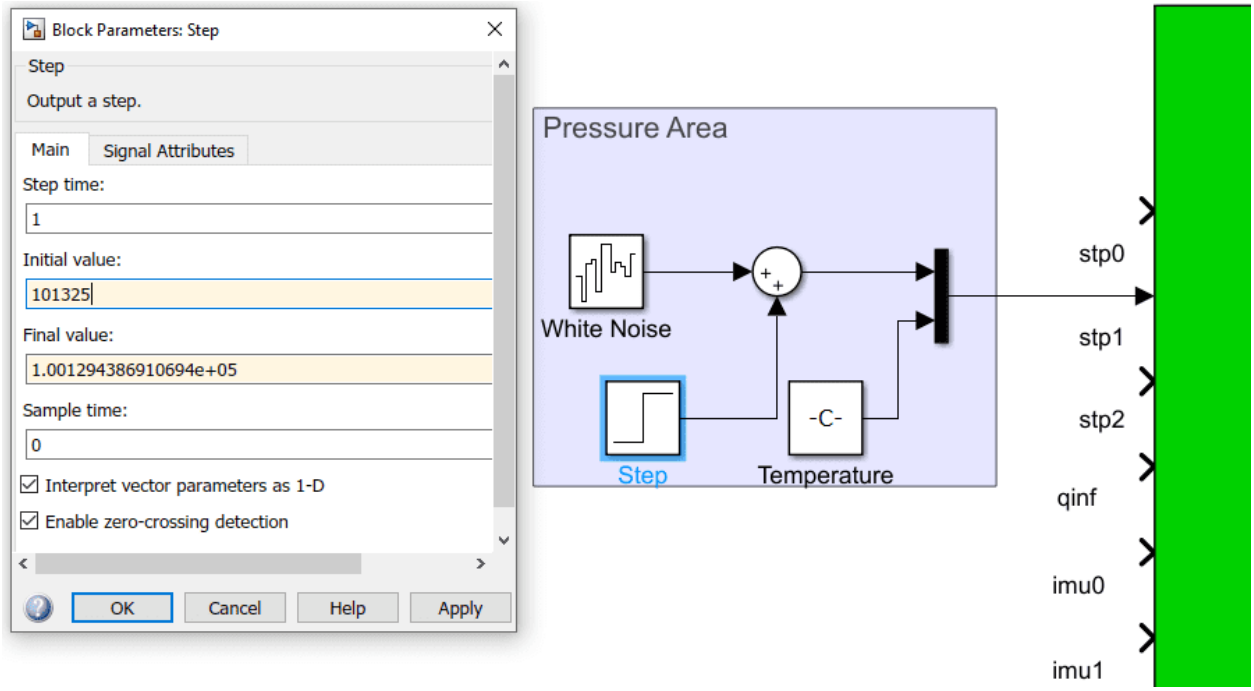

Fig. 8: **Static pressure - Step input**

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

4.2.1.2.2 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:

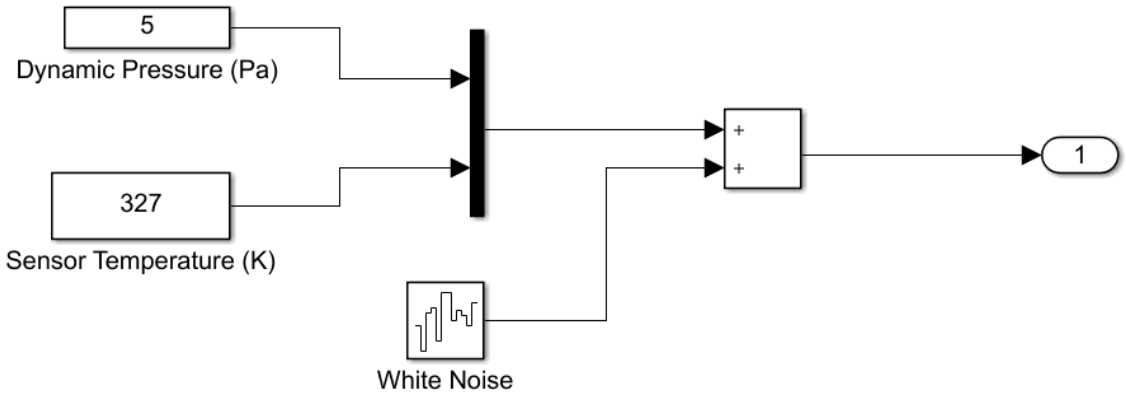

Fig. 9: **Dynamic pressure - Constant**

- **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.

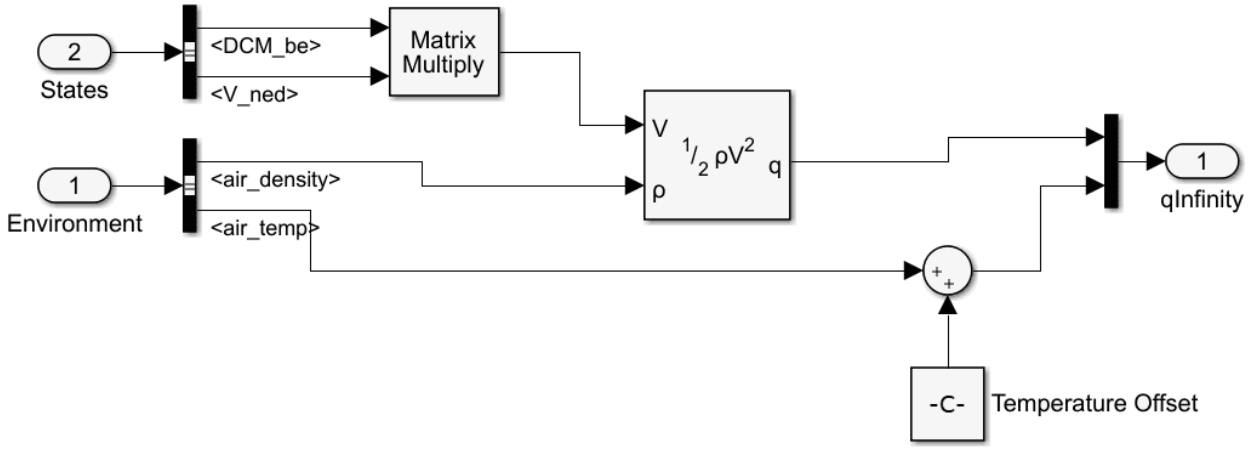

Fig. 10: **Dynamic Pressure - Subsystem**

4.2.1.3 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 1x Autopilot 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, if users wish to enter **measurements into the ADIS16505-3 IMU** 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**	Main IMU	$R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	IMU1	Secondary IMU	$R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$
4.5	**IMU0**	Main IMU	$R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	IMU1	Secondary IMU	$R = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$
	IMU2	BMI088 IMU	$R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
4.8	**IMU0**	ADIS16505-3 IMU	$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$
	IMU1	Secondary IMU	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$
	IMU2	BMI088 IMU	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Within the S-Function, the introduced data is transformed to match the coordinates of the 1x Autopilot. 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**.

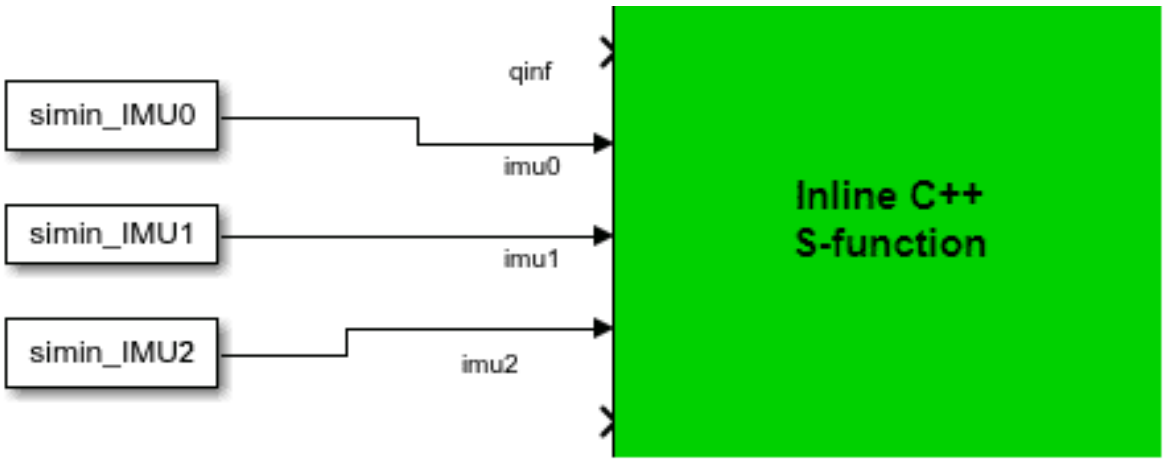

Fig. 11: **IMU - From workspace blocks**

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.

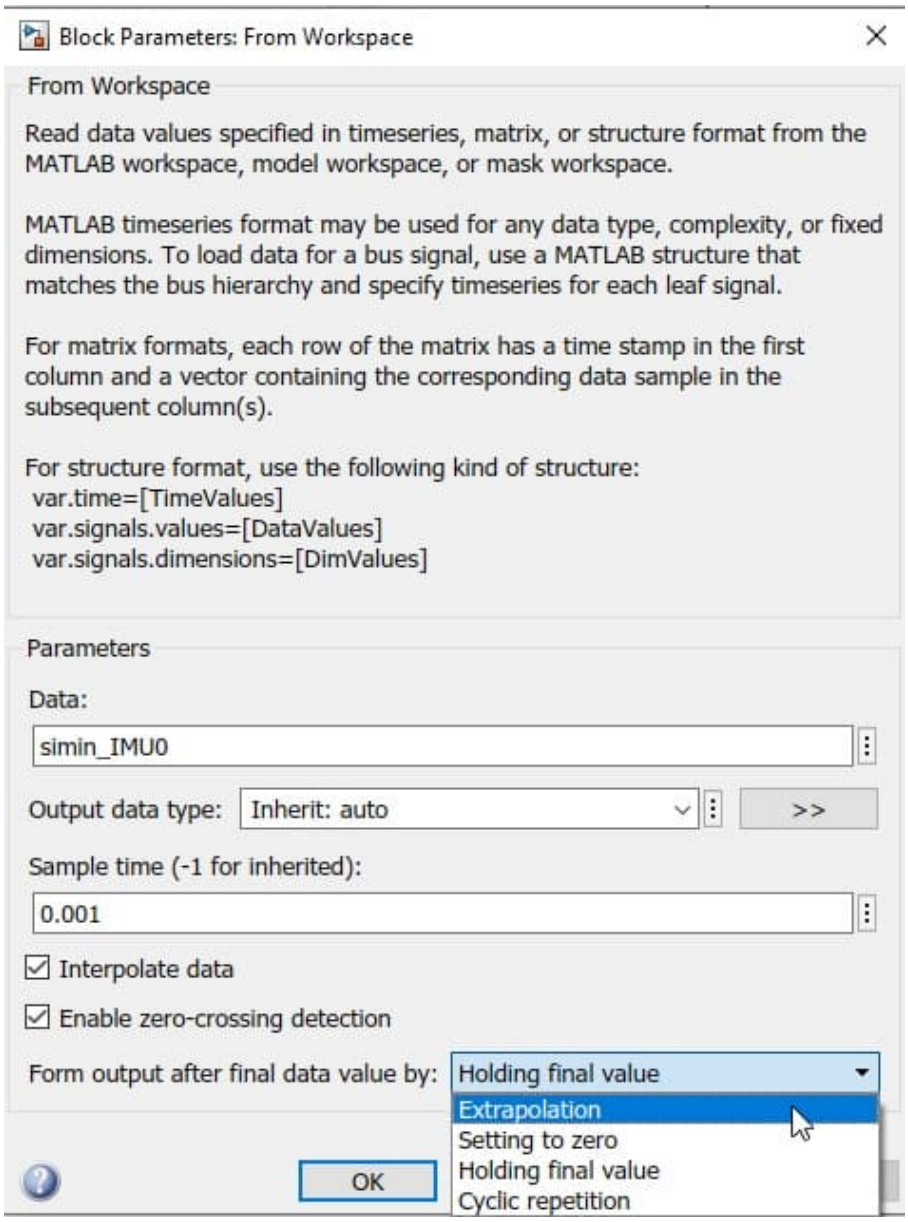

Fig. 12: **From workspace block configuration**

- **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:

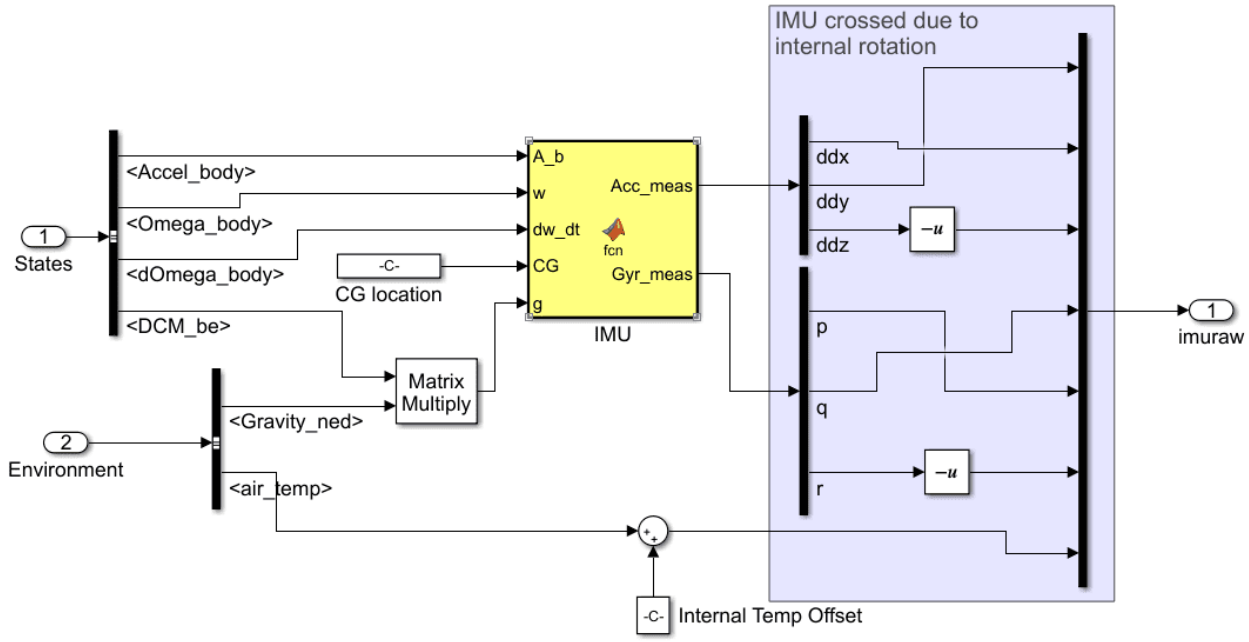

Fig. 13: **IMU - Subsystem**

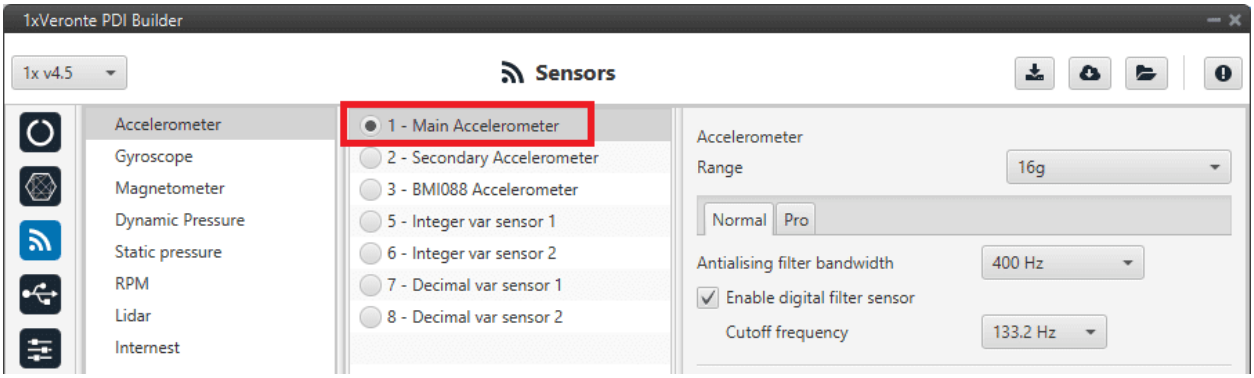

Fig. 14: **IMU - Accelerometer selected in 1x PDI Builder**

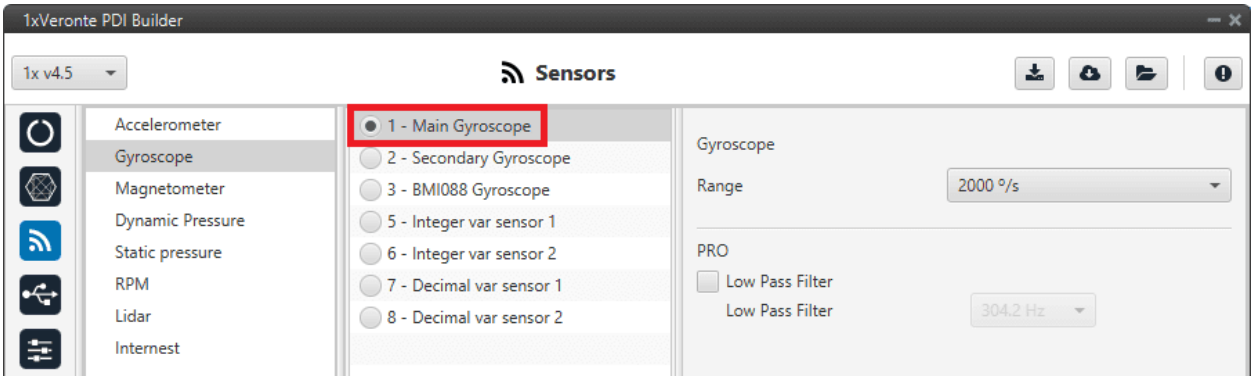

Fig. 15: **IMU - Gyroscope selected in 1x PDI Builder**

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

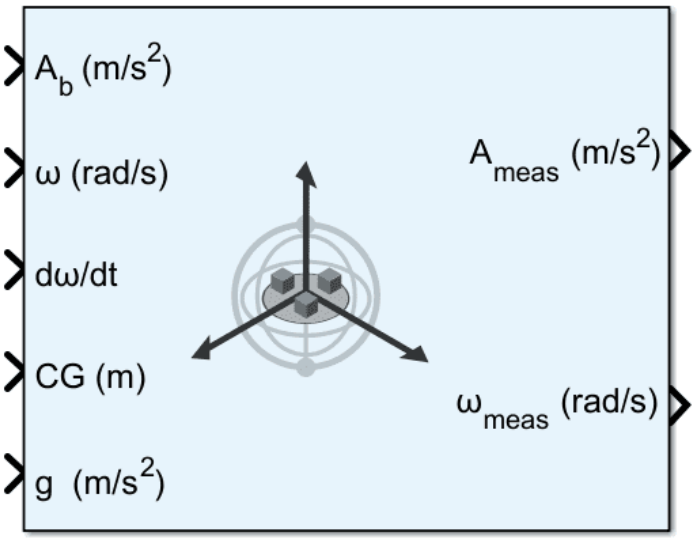

Fig. 16: **IMU - Aerospace Toolbox block**

- **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 Main IMU 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 **ADIS16505-3** IMU 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 ADIS coordinates**

   Prepare the data for conversion to ADIS coordinates by multiplying the sensor body data by the ADIS 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:

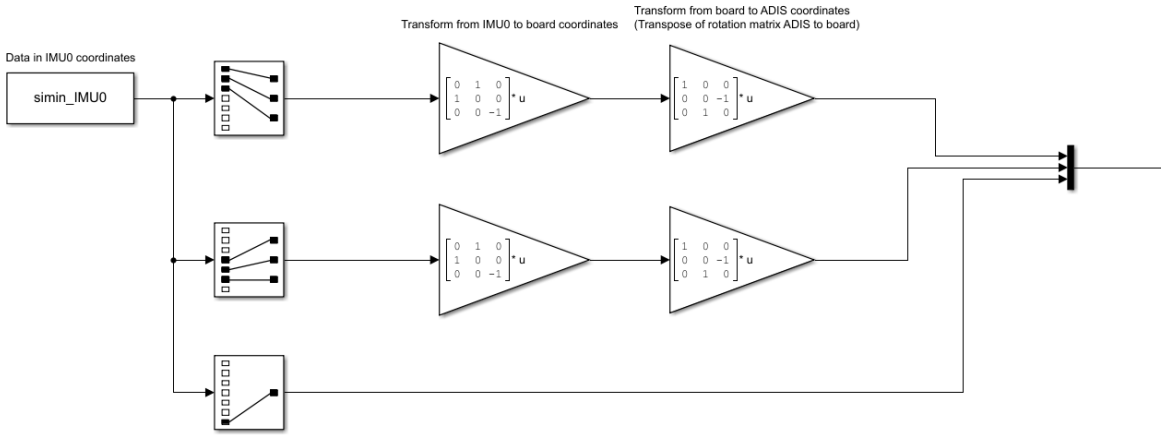

Fig. 17: **IMU - Example**

4.2.1.4 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**	Internal LIS3MDL	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
4.5	**MAG0**	Internal LIS3MDL	$R = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
	MAG1	Internal MMC5883MA	$R = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
4.8	**MAG0**	Internal LIS3MDL	$R = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$
	MAG1	Internal MMC5883MA	$R = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$
	MAG2	Internal RM3100	$R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

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.

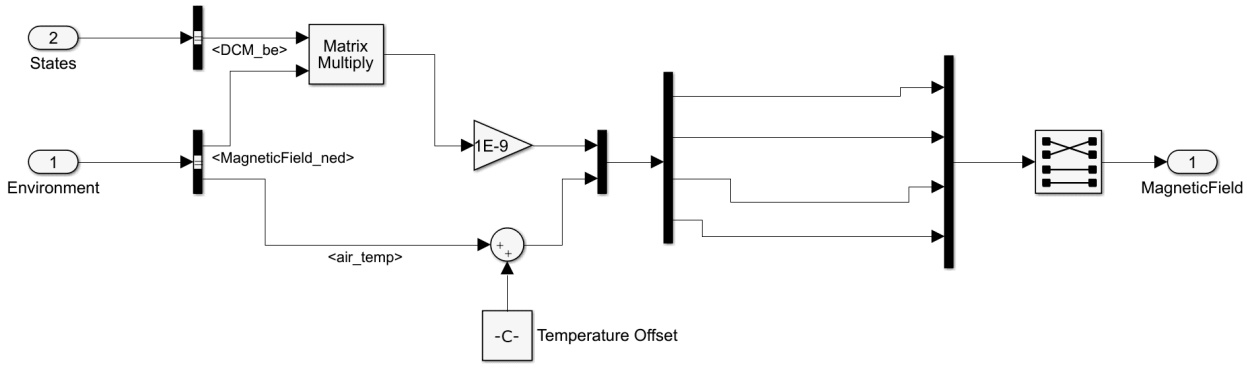

Fig. 18: **Magnetometer - Subsystem**

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.

4.2.1.5 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

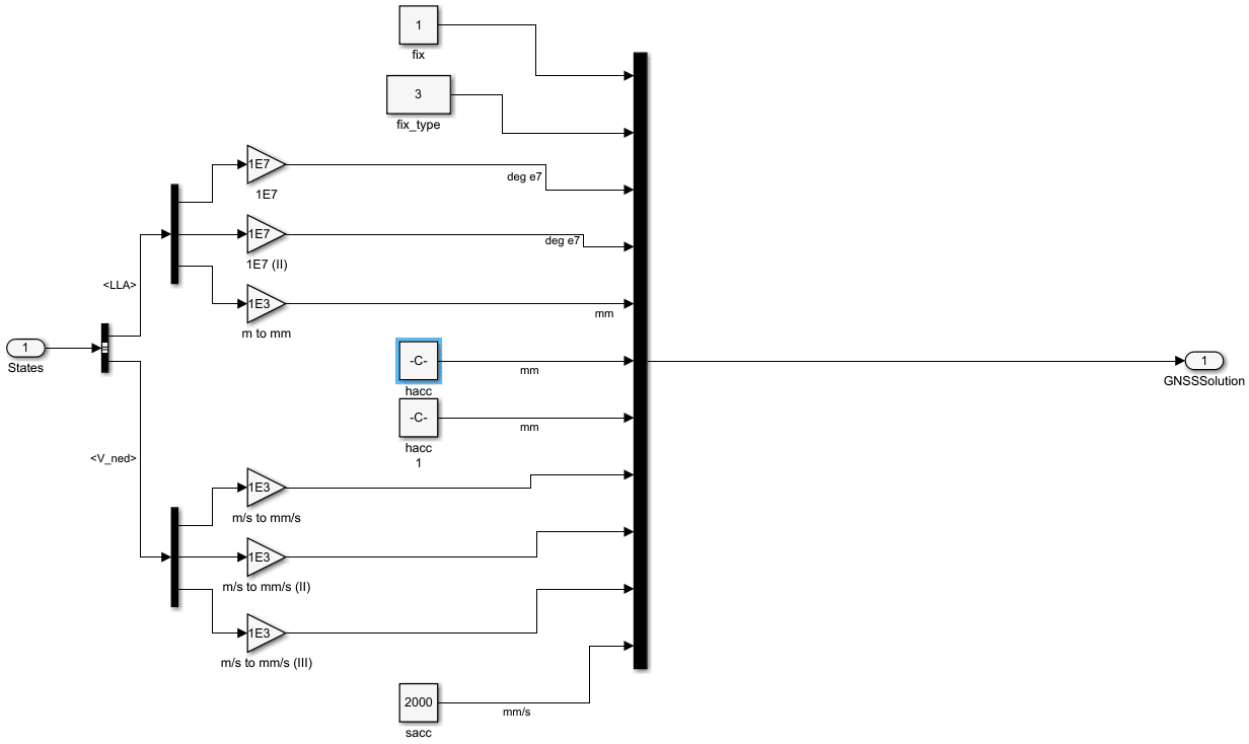

Fig. 19: **GNSS array**

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 squared error** ($accuracy = \sqrt{error^2}$). 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.

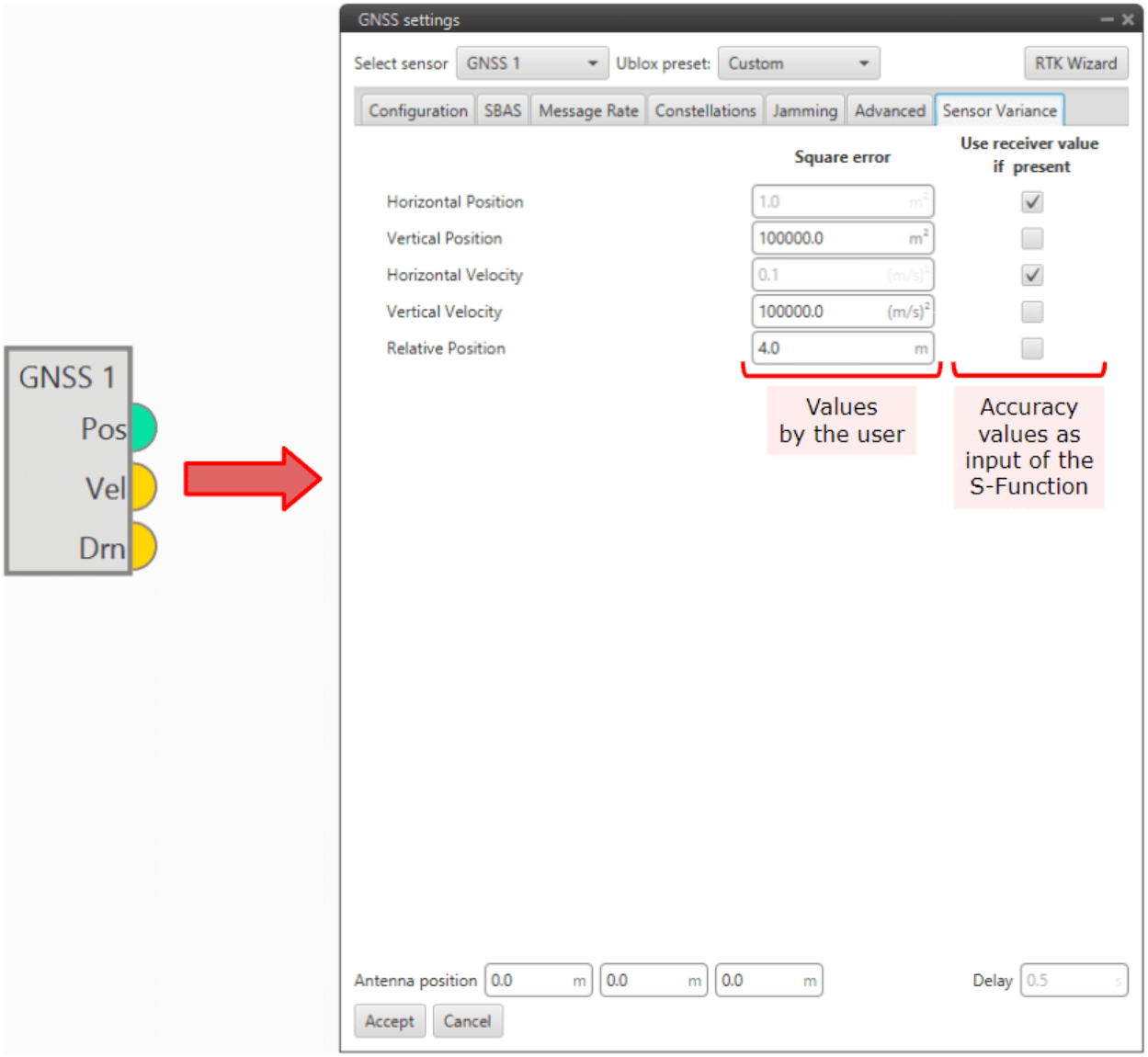

Fig. 20: **GNSS variances - 1x PDI Builder**

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:

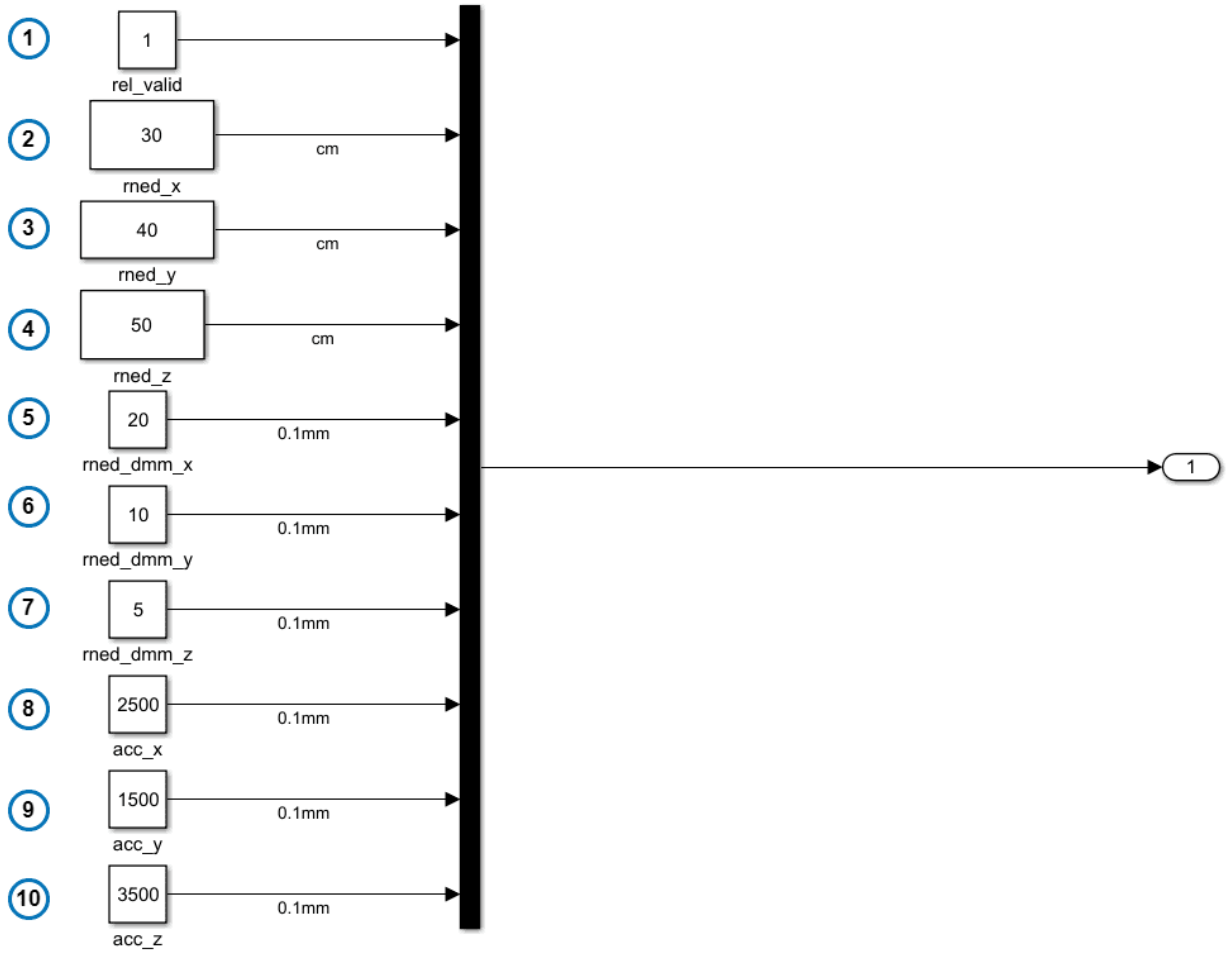

Fig. 21: **RTK inputs**

4.2.1.6 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 1x Autopilot, and it is possible to use them in certain user programs.

The order of this array is: Internal ADC Channel 1, External ADC Channel 1-5, Internal ADC 2-12.

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

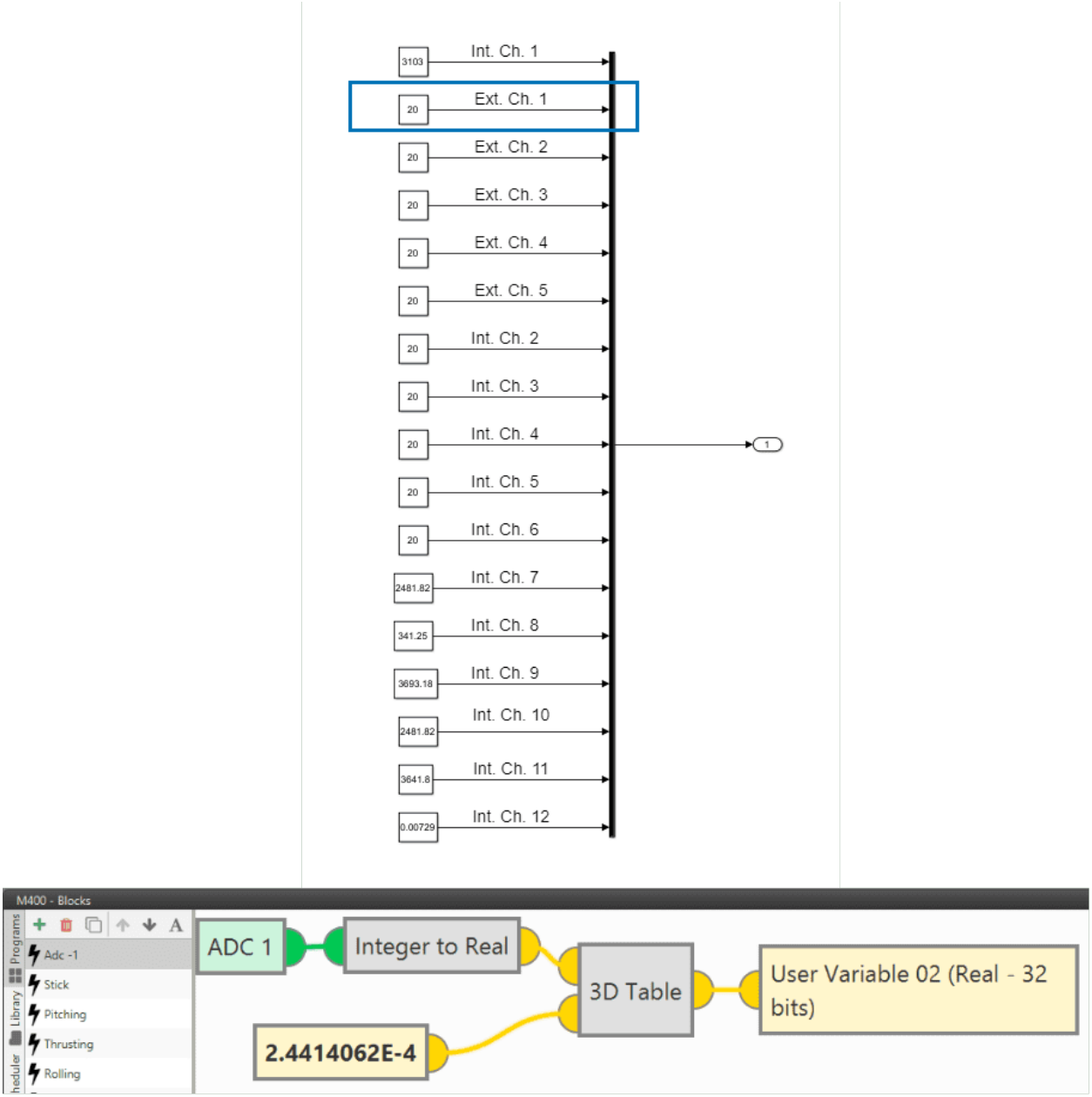

Fig. 22: **ADC readings**

4.2.1.7 Serial Communications

Veronte Autopilot 1x can manage input and output serial ports (for more information on this, see the 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 1x Autopilot should be **sent via UDP** (if this approach is adopted):

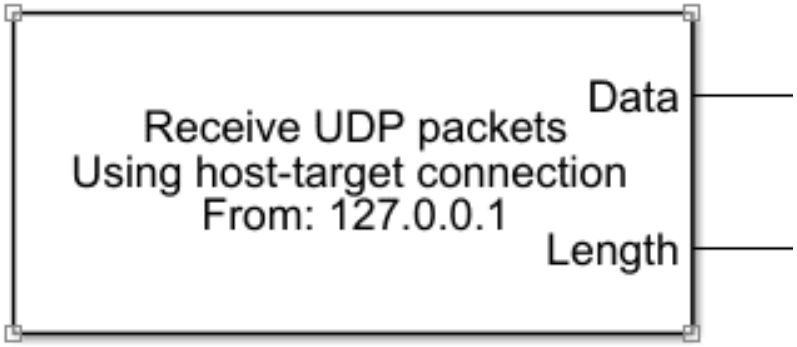

Fig. 23: **UDP Block**

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

- **USB**: USB port
- **SCIA**: 4G connection
- **SCIB**: Radio
- **SCIC**: Serial Port 485
- **SCID**: 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:

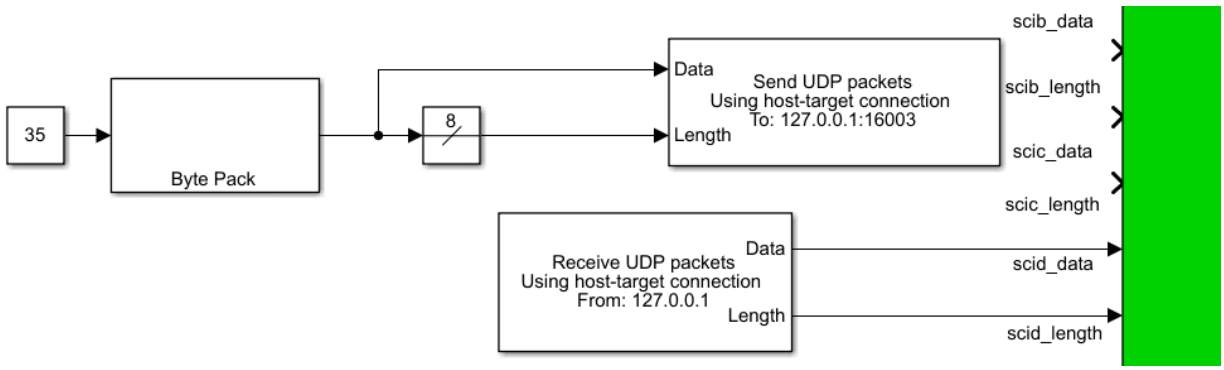

Fig. 24: **Sending a RS-232 message in Simulink**

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 -> I/O Setup section of the **1x PDI Builder** user manual.

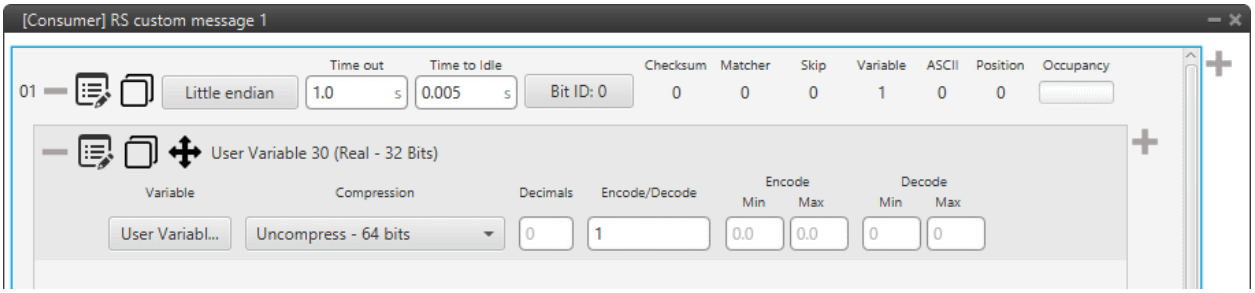

Fig. 25: **Custom message - 1x PDI Builder**

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

4.2.2 Telemetry

In the S-function there are 3 inputs specially dedicated to select custom temeletry (pin 22 for Bit variables, pin 23 for Integer variables and pin 24 for Real variables).

Each of these variables has an ID. Their input structure is fixed and must be of **size 50**.

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:

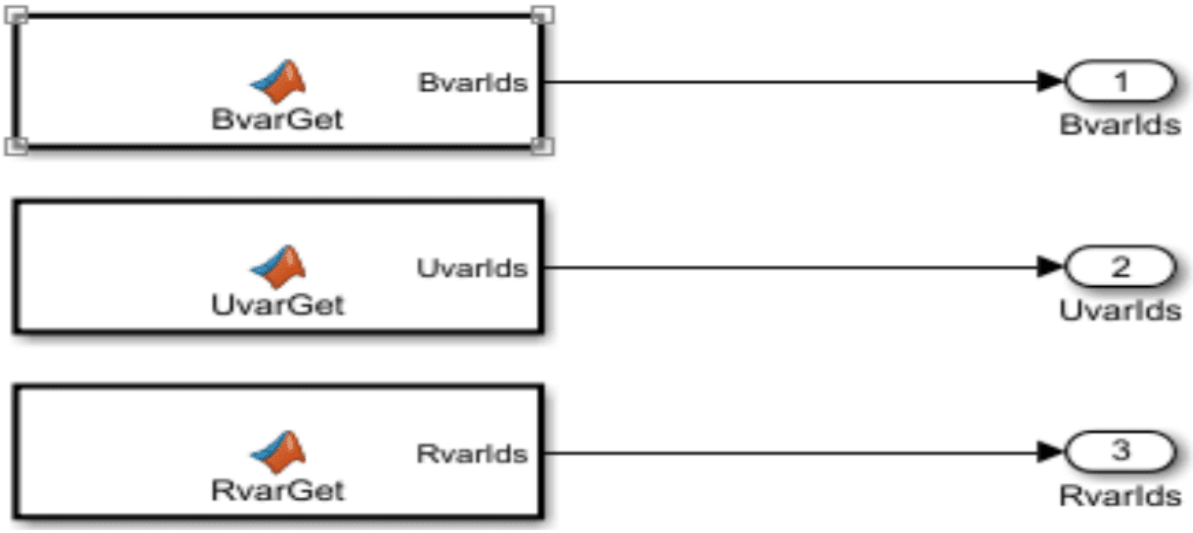

Fig. 26: **Telemetry blocks**

The example function for real variables is given below. In it, the desired real variables are configured to be displayed. In addition, a line of code has been added so that **Matlab fills the variable vector with zeros until it reaches size 50** if the vector itself does not do so, in order to achieve the expected structure size.

Requested Variables/MATLAB Function2*

```
1   function RvarIds = RvarGet()
2   % Requested Ids (row vector)
3   requestedIds = [...
4       330:336,    ...    % IMU 1 Raw Measurements
5       337:343,    ...    % IMU 2 Raw Measurements
6       361:367,    ...    % IMU 3 Raw Measurements
7       386:392,    ...    % IMU 4 Raw Measurements
8       348:349,    ...    % Static Pressure 1 (HSC) Raw Measurements
9       344:345,    ...    % Static Pressure 2 (MS56) Raw Measurements
10      368:369,    ...    % Static Pressure 3 (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, zeros(1,50-size(requestedIds,2))];
```

Fig. 27: **Telemetry block - Real variables function**

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

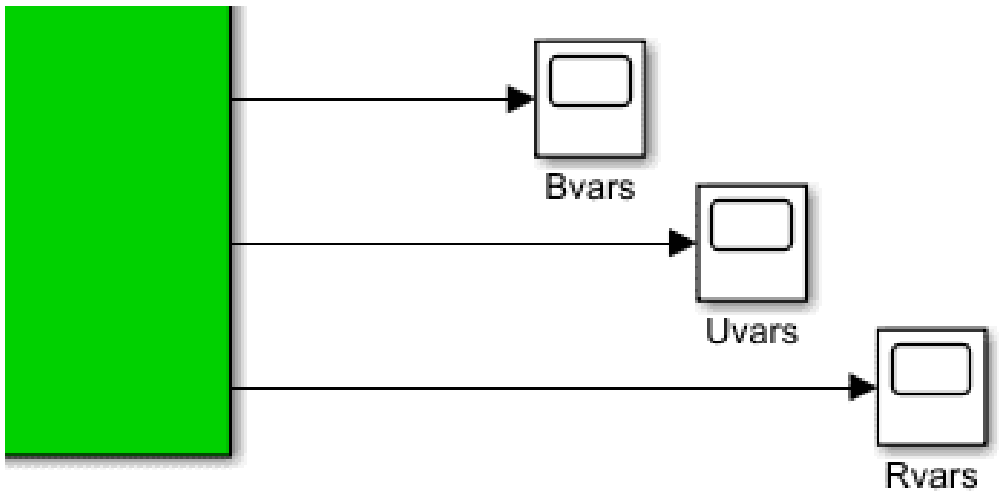

Fig. 28: **Display variables - Scope block example**

4.2.3 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 behaviour.

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

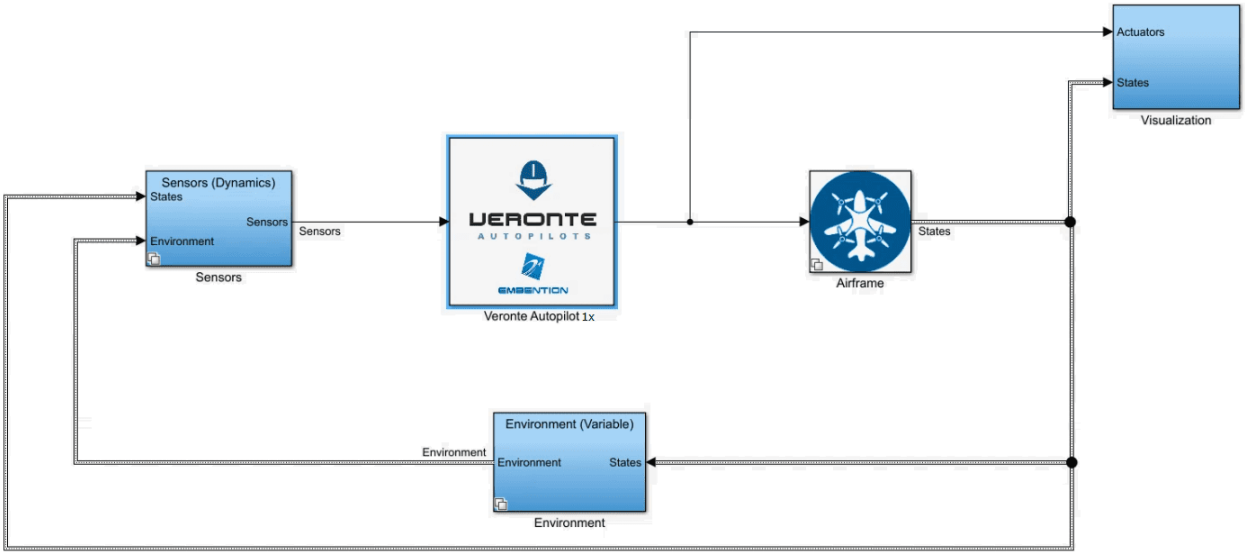

Fig. 29: **Complete Setup Example**

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.

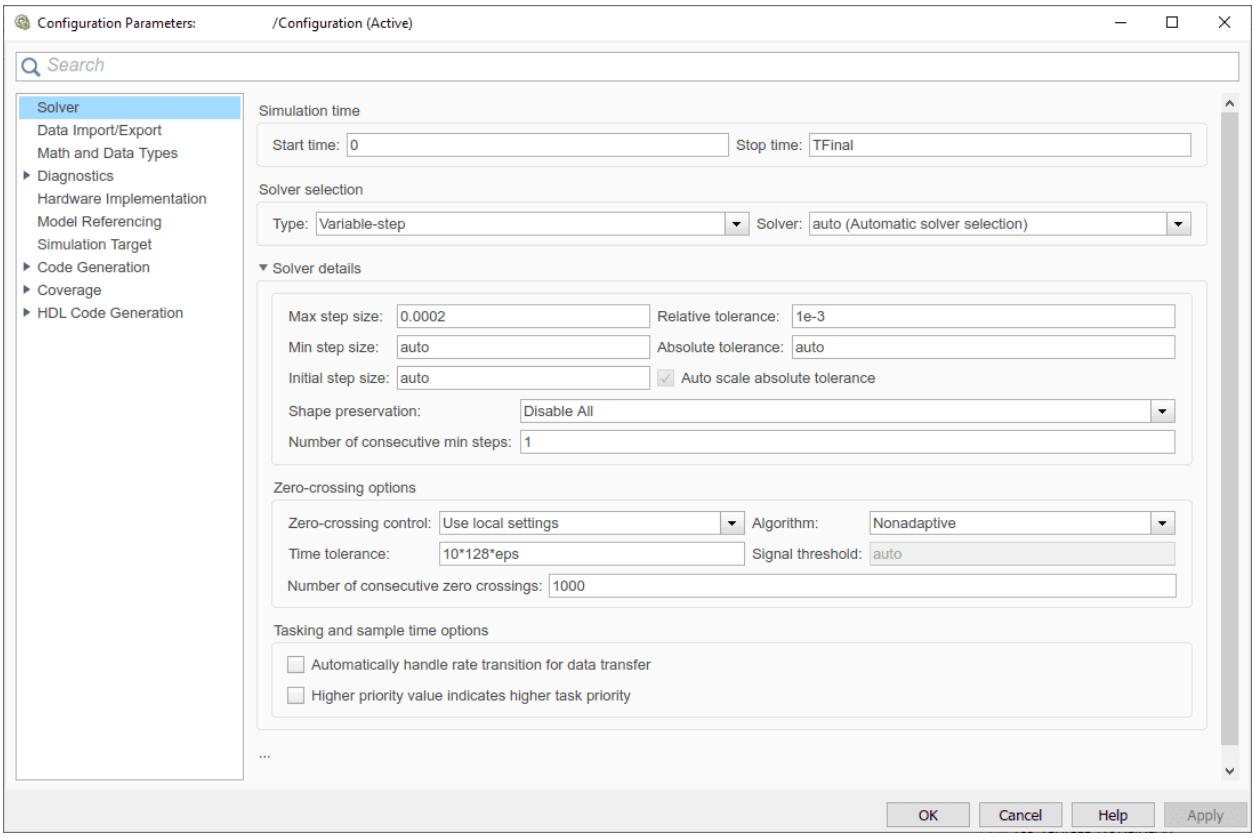

Fig. 30: **Time step settings**

CHAPTER

FIVE

TROUBLESHOOTING

5.1 DLL and Image paths

A common error from the log has to do with the **path to the dll or image**.

If this is the case, please make sure that these files are **placed in the root directory of SIL** (the unzipped SIL 6.8 folder).

If any of them are located in a directory other than the standard unzipped SIL 6.8 folder, it is **possible to specify a different path** by **creating** a file named "**dll_config.vcfg**". This file must be in the same folder as the slx files if using Simulink or VeronteConsole.exe if using the standalone executable.

Within this file, the absolute path to the DLL and Image file must be provided as follows:

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

Make sure that the path is as detailed above, it should end with the image file or the dll file, not the folder where they reside.

5.2 Hardware version change

As **the hardware version will be that of the Veronte Image**, if the user wishes to change the hardware version it should specify it.

For this, in the "**dll_config.vcfg**" file explained above, specify the hardware version by using the following command: `\hwversion version`.

So for example, to specify 4.8 hardware version, the command would be `\hwversion 4.8`.

5.3 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 in the **1x Software Manual**.

CHAPTER

SIX

ACRONYMS AND DEFINITIONS

ADC	Analog to Digital Converter
CAN	Controller Area Network
DLL	Dynamic Link Library
EKF	Extended Kalman Filter
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
NED	Noth East Down (coordinates)
PC	Personal Computer
PDI	Parameter Data instruments
PWM	Pulse Width Modulation
RS232	Recommended Standard 232
RTK	Real Time Kinematic
SCI	Serial Communication Interface
SIL	Software In the Loop
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
USB	Universal Serial Bus
WGS84	World Geodetic System 84
WMM	World Magnetic Model